LAB Writeup

CS-4473-001
Team 4
Nicholas Babb, Andy Le, Matthew Carroll
Lab 01 and Lab 02 Writeup

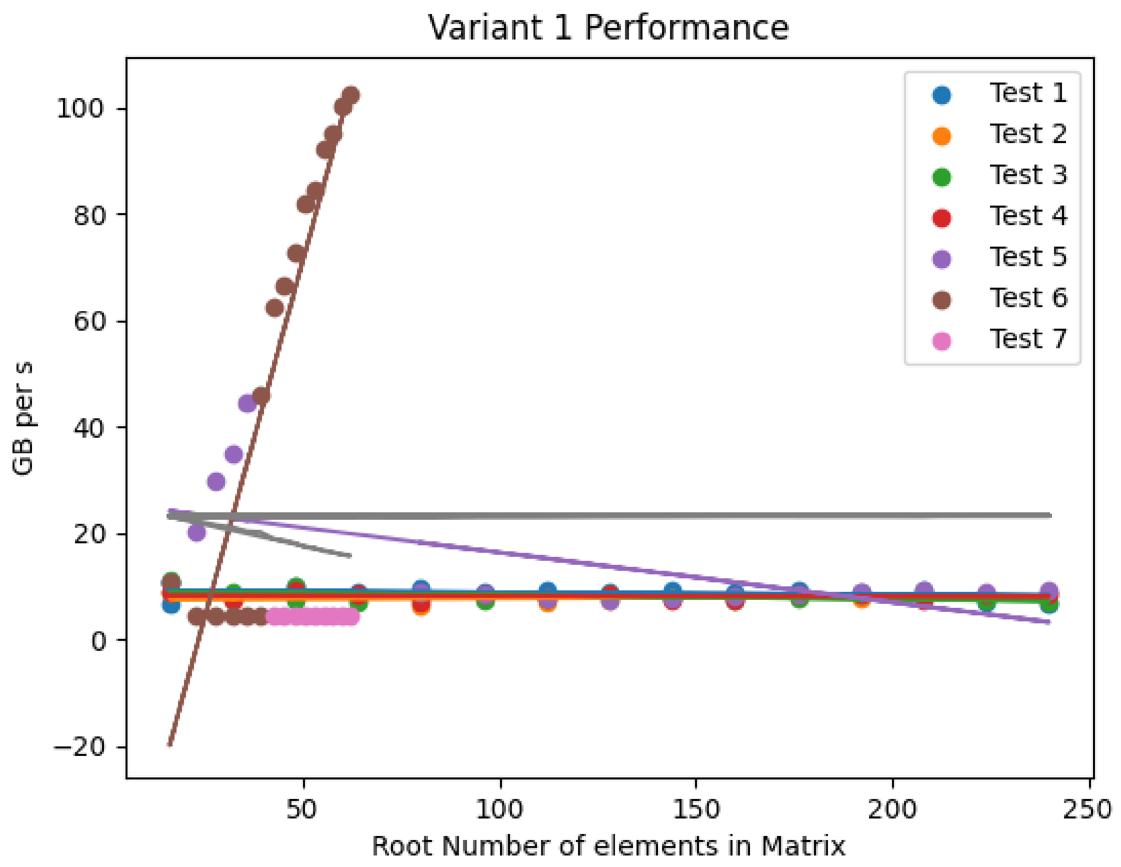
TRANSPOSE:

The Transpose operation is a concept that deals with matrix operations. The transpose of a matrix is obtained by switching the row and column indices and vice versa. The problem this operation solves is that it can efficiently manipulate matrices in a parallel computing environment. It also can relate to the topic of matrix multiplication. Let's say we want to multiply two matrices A & B, we would have to traverse one of them in a way that's very slow, either row-major or column-major. If we used the transpose operation on one matrix, then traversing both matrices would lead to the fastest way that best utilizes caches. Some pain points that we are experiencing in this problem is trying to find multiple variants that would give us an optimized form of the transpose operation. A couple plans we've thought of when trying to solve this were multi-threading the existing algorithm, modifying the base algorithm, and utilization or blocking/striding of the caches. This operation relates back to the material covered in class as we have talked extensively about matrix operations and their relationship to parallelization.

Variants:

optimized_op_02_var_01.c

For our first variant, our approach centered around traversing the matrix in such a way that cache traversal would be at a maximum, meaning that we would have a low amount of misses and the cache would be utilized frequently to reduce the amount of time spent searching for values. As matrices are stored in row-major order inside of the operating system, regardless of the specified ordering schema in the program, it is beneficial to read from column-major order matrices in a row-major order manner, which can cause some issues with the logic involved, as we then have to leap across blocks of memory rather than simply iterating through the array the matrix is placed into. Below, we can see a graph of the optimizations provided by this operation.



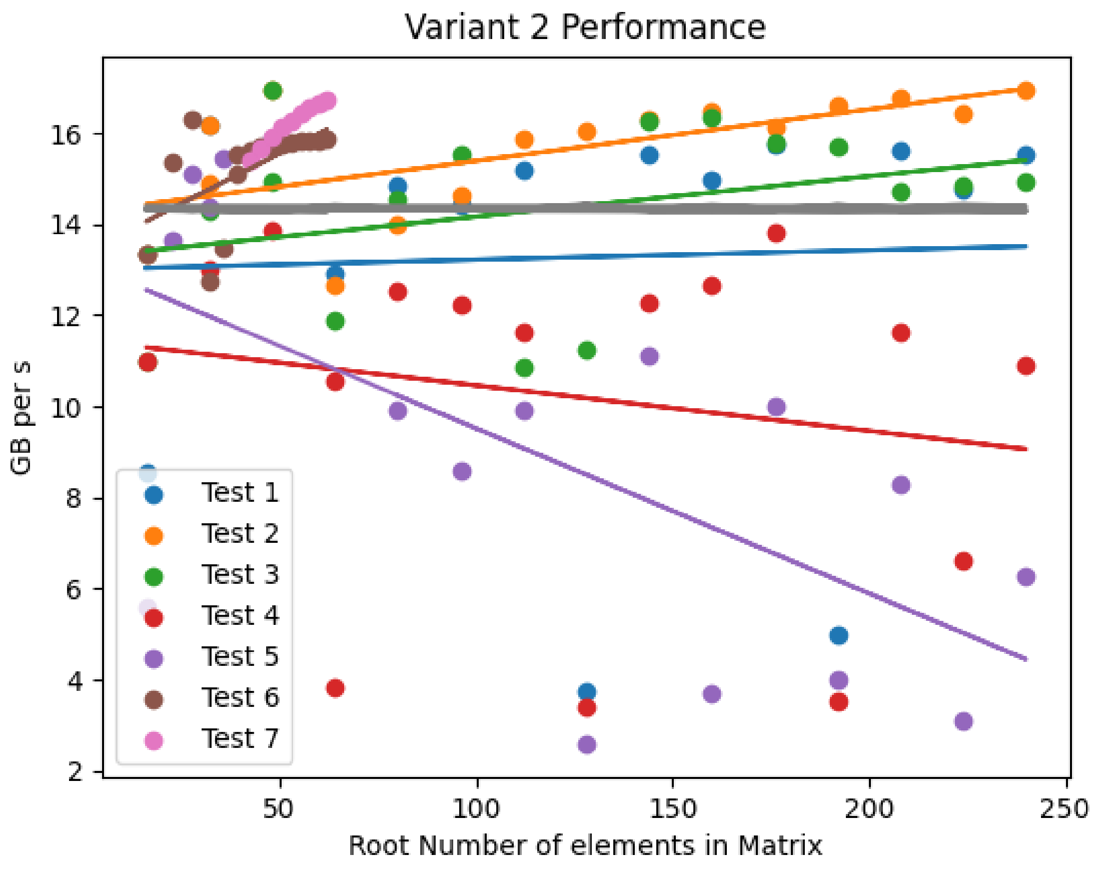
Due to some of the issues with this operation, only a selection of the tests are passing, and we can see that the optimizations thanks to this change are rather minor. While we are traversing the matrices in a more optimal manner than before, these matrices are small enough that the differences are not very worthy of note.

optimized_op_02_var_02.c

The goal with this variant was to parallelize the task using four threads. The main idea behind the algorithm was to split the matrix into four sub-matrices and find the transpose of each of those, and then bring them back together and rearrange them to produce the correct transpose.

This would complete a portion of the transposition time in approximately ¼ of the total time. However, the remainder of the transposition is completed in parallel (switching the two corners to finish the transposition).

In terms of performance, this variant did not end up performing as intended. The overhead required to carry out the algorithm in parallel was not made up for by the gains in parallelization; the algorithm's data numbers were too small for the overhead to be worth it. Inspecting the plot below, you may notice that some of the variants' average performances improve through increasing the number of elements in the matrix, while others actually get worse. This is likely due to column-major vs row-major differences in each test case. Other variances include case 6 and 7's small data sizes but large performance values. We believe this to be an average

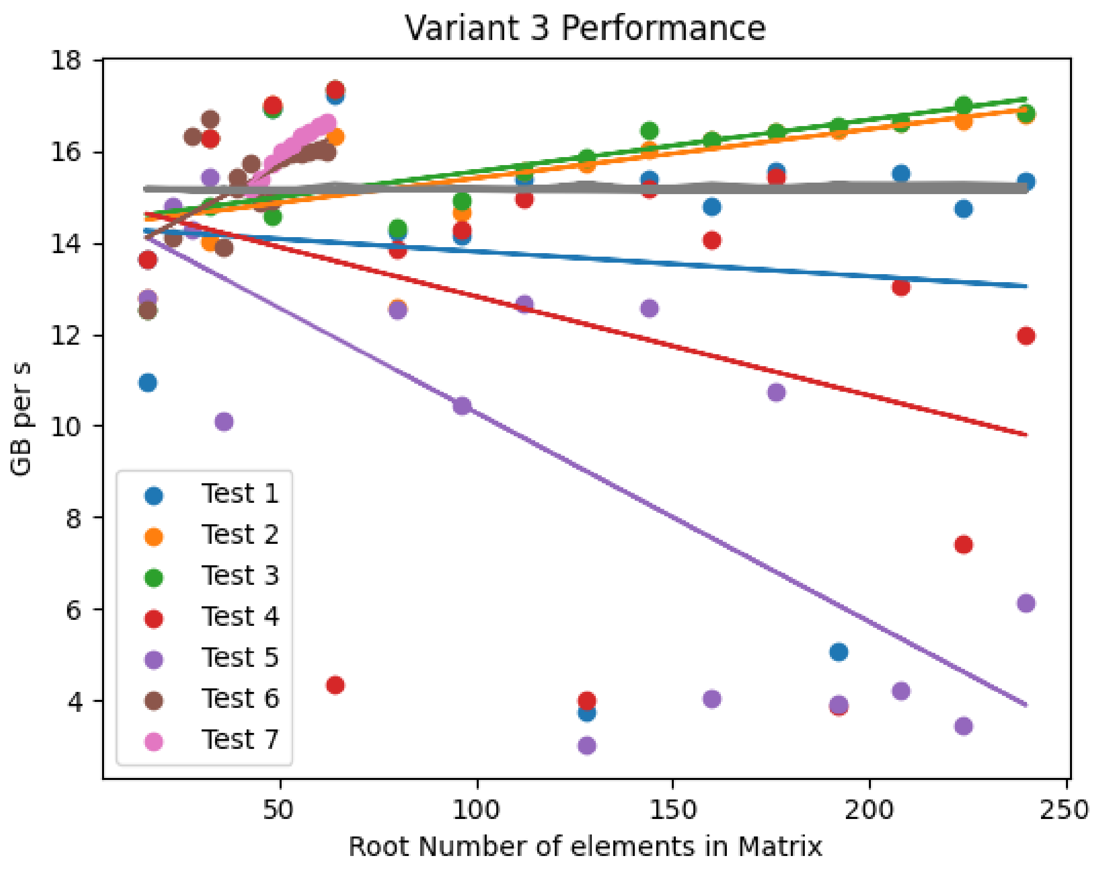


optimized_op_02_var_03.c

For the third variant, we kept a similar pattern from the second transpose variant and continued utilizing threads. In the third variant we can see that the variant first checks the source and destination matrices. Depending on the specific value of these parameters, the function will iterate through the multiple loops to find different methods that will perform the transposition on the matrix.

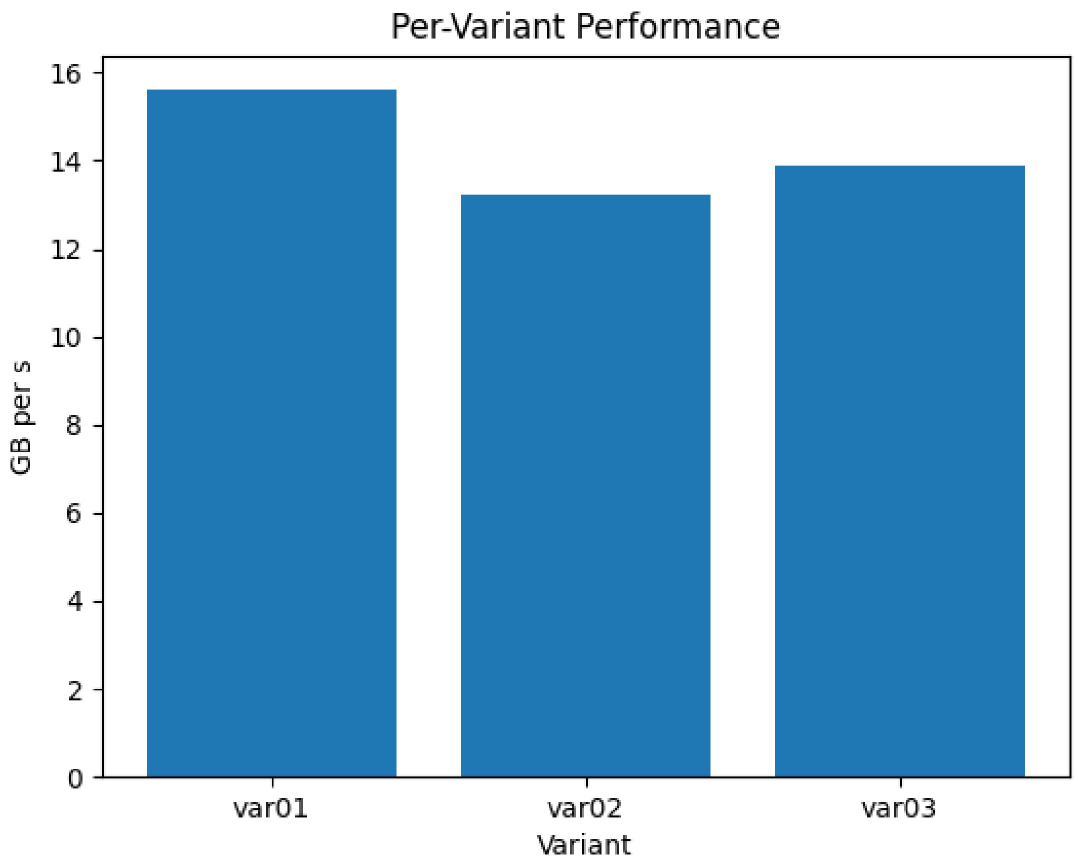
An example can be observed on one of the many if statements in variant 3, if the column stride of the source matrix and the column side of the destination matrix are both 1, then the third variant will utilize parallelization to transpose the matrix in a more efficient manner. It will also test other cases such as if column stride of source equals row stride of destination matrices and vice versa. Overall the third variant attempts to optimize the operation of transpose by the input parameters from the source and destination matrices.

From the graph below, we can see that the performance of variant 3 only differed slightly from variant 2. Even with the different thread parameters, we observed that the relationship between (GB per S) and the (Root number of elements in the matrix) in relation to the test cases didn't give us a more optimized variant of the transpose operation.



<u>Overall Optimizations:</u>

As seen in the image below, the optimizations for this operation were ultimately minor. As we are dealing with small arrays rather than large arrays, the optimizations can be fairly miniscule, especially compared to one another rather than compared to the base.



Stencil:

The Stencil Operation is a concept that deals with distilling states of an array, matrix, or other data structure into a single value that can be used to evaluate the status of the board. For a number of states, there can be one or multiple places in the data structure that only change a few times. Rather than determining the entire state of the data structure, we can simply check the state of this singular place and use it to quickly separate the state of the data structure. This may need to be done multiple times to ensure the state is accurate, but it is a far more efficient method than individually iterating through the entire data structure to determine its state. This ties into the concept of MISD as we are converting multiple sources of data, that being the multiple states a data structure could be, into a single set of readable values that still allow us to do multiple types of instructions based on that single known value. Some of the pain points we are experiencing with the stencil operations include determining ways to optimize 3 separate variants of the function.

Variants:

Due to time constraints and issues setting up the existing code given to us, we were not able to develop satisfactory variants to the Stencil problem.