DOCUMENT MENTIONS:

It appears that a lot of this is to go under “Data Collection” section

“Data Collection – measure the performance with the tool provided with the lab, examine the code to analyze its quality. Describe the reasoning you followed to generate the human code, describe several attempts you might have tried. With the AI-generated code: save all the prompts, save the whole conversation with the chatbot that led to code that has good performance. Show all the attempts you made at generating code and then analyze the most successful ones in the next section about the discussion. You need to collect enough information to provide substantial discussions and give foundation to your conclusions.”

The 40 points for the quality of the analysis of results and conclusions are distributed among the following, roughly 10 points each.
• Discussion of the results: what did you get from the human-generated code; how did you produce that code; which aspects and specific approaches were critical to getting a good performance, also what did not work; if you tried something and didn’t work, explain that too.

ideas pursued, reasoning about the code written

conclusions about the gains in performance on human-generated code;

Comparison: compare all the human-generated code with the corresponding AI-generated code, notice the differences in style, in readability (for a human reader), in other aspects that you think will be relevant for the discussion about pros and cons of each <= gonna start thinking about that after Nathan is done?

Methods – the problems, model used, prompting method. => maybe this is a good separation out of what the specific challenges were with each matrix?

in each writeup, probably beneficial to mention the baseline number of misses with the trans function given.

==========

32 x 32 WRITEUP

==========

EXPLANATION OF THE ATTEMPTS/THOUGHT PROCESS

As with all of the attempts made, the initial 32x32 baseline miss rate was found through the simple *trans* function given. At worst, the number of misses was 1184. A far-cry from where the miss bound needed to be at < 300, there was some exploring to be done on how the sets of the cache laid.

There are some pieces of key mathematics needed to understand how the sets of this direct-access cache lay and how we can best use it. With the numbers given, as discussed, we have 32 sets, and each set can hold 32 bytes, which will end up holding 8 ints in a set.

For the 32x32, we can see a couple key patterns. First, the sets exactly fit within the dimensions, the first row being sets 0 - 3, next row being sets 4 - 7, and on. Further, the sets repeat their row overlay every 8 rows.

Therefore, when we are transposing, if we store a value in B and access in A, it is in the best interest in that cache to stay within those sets as much as possible. So, the first attempt past a baseline transposition attempts to make use of this fact and conduct a straightforward transpose procedure but within 8x8 blocks at a time. By staying within these 8x8 blocks, we can make use of the 8 int values loaded into the cache lines as much as possible before moving on to the next 8x8 blocks.

This attempt drops the miss rate significantly, only having 344 misses. From there, we need to analyze where the worst of the misses occur. We can see, by assuming and then verifying empirically by the cache simulation addresses, that A and B can be treated as contiguous in memory. The significance of this is that the sets of the cache overlap in the current access pattern specifically at the diagonal blocks (rows/columns 0 - 7, 8 - 15, …, 24 - 31). When we transpose these specific 8x8 blocks, accessing A and storing in B, we are thrashing the sets heavily due to this overlap.

So, one way to address this is to load things preemptively, and allow one matrix full control, so to speak, over a set until it is ready to relinquish control to the other. To accomplish this, we will load a diagonal value of A into a temporary variable (for example, temp = A[0][0]). This allows A to claim control over the set overlaying this small block row (A[0] allows A control over set 0, following our example). Then, we allow B to have control over all the other sets, setting all values of B except for the diagonal (avoid B[0][0], set B[1..7][0] = A[0][1..7]). This allows A to hit on all accesses in this iteration past the initial access. Then, when we are done allowing A to have control of a set, we hand over control to B, storing the temporary value grabbed at the beginning into the appropriate diagonal slot (B[0][0] = temp). With this attempt at careful control hand-off between sets on the diagonal blocks, we manage to plummet the miss rate below our boundary: 288 misses.

==========

64 x 64 WRITEUP

==========

As with the 32x32, we will start with a baseline of running the *trans* function to identify the worst-case miss rate, which comes in far higher than the prior square at 4724 misses.

As an initial measure, since the 64x64 is also equally divisible into 8x8 blocks, we will attempt the same blocking mechanic as the 32x32, negating the diagonal handling, just to see the behavior untouched. This unfortunately has no tangible effect, having the exact same amount of misses as the baseline: 4724.

However, with blocking being a remarkably successful method in mitigating the miss rate, we can try changing the blocksize to see if there is any change. After some exploring, a blocksize of 4x4 instead ended up having the best decrease in misses, dropping to 1892.

From here, some closer analysis was required to get that number lower (along with a fair amount of failures). There is a definitive difference in how the 64x64 sets overlay A and B matrices compared to the 32x32. For the 32x32, in each grouping of 8 columns (or 8 ints that will be loaded into a set line), every 8 rows, a group of sets repeat. The 64x64 has the same pattern, but with far more difficult consequences: in the same 8 column groupings, the set groups repeat instead every 4 rows. This is the reason that the 8x8 blocking alone does nothing of use–the filling of B column-wise thrashes itself within the 8x8 block. Nevertheless, that is exactly why the 4x4 block improves: B thrashing decreases considerably by at least hitting instead of missing on those smaller 4 int accesses per row before thrashing itself in the next iteration/block.

We still run into an unfortunate issue: in the general case (we will cover diagonal blocks in a moment), we are not making the best use of the ints loaded into the cache line. When we 4x4 block, we never use the second half of the set line when filling B column-wise. So, there must be an access pattern that minimizes misses between A and B both with filling blocks. Through many drawings, a mixture of the first two attempts was developed. First, we will overall still deal with 8x8 blocks since the cache lines still load in 8 ints at a time and we want to use that fact to our advantage. However, we will change how we fill the 8x8 blocks by using smaller 4x4's within the larger block. We will focus on the access pattern of A: why not access the smaller 4x4 sections of the 8x8 block as a sideways-U-shape. Meaning, we access the block in A as first to late: upper-left 4x4, upper-right 4x4, bottom-right 4x4, and bottom-left 4x4. This has 2 benefits: (1) we can use all the 8 ints loaded into the set-lines of the top 4 rows on the A-block before allowing A to thrash itself on the bottom 4 rows, using then all of those bottom 8 ints loaded up; (2) we can lessen the self-thrashing of storing in B-blocks since we are filling the 8x8 as an upside-down-U-shape, which admittedly will self-thrash twice, but overall will decrease evictions in the general case. Changing over all blocking to fill using this access pattern decreases the misses further to 1644.

Now, on to the massive problem of the diagonal blocks. The thrashing that occurs at these diagonals is far worse than that of the 32x32, as evidenced by the 8x8 first attempt blocking having no benefit to speak of. Not only is there the typical A and B thrashing each other behavior, but now A and B are additionally thrashing themselves. Not even the U access pattern is enough to fix this problem. We need a unique solution to decrease the miss rate in these blocks specifically to even approach our goal of < 1300 misses.

So, the next attempt utilizes a convoluted loophole. We cannot modify the contents of A, but we can modify B in any way we wish. So, there is no reason we cannot use B as a conversion to get the diagonals to force them into being a general case block fill. The idea is that we can handle the diagonals first as a special case. First, we place the contents of the A-diagonal-block into the 8x8 block next to the B-diagonal block (for the first 7 diagonal blocks, place to the next right 8x8, for the last, place in the next left 8x8). This effectively copies the problematic overlapping-set values into sets that will be guaranteed to not overlap with the B-diagonal-block sets. Next, use the U-shape access pattern to fill the B-diagonal blocks as we did in the prior attempt, but instead fill from the temporary storage block next door. This idea is successful, dropping our tally to 1412 misses. So, this idea was in the correct direction, but needed improvement.

Instead of using the blocks next to each diagonal, we will instead use one consistent storage unit in B for the left half of diagonals, and then a different consistent unit for the right half of diagonals. The choice is somewhat arbitrary, but the following choice has some specific benefits: use rows 0 - 4, columns 48 - 63 for the left half, and use rows 60 - 63, columns 0 - 15 for the right half. The benefit to choosing these spots and filling them with top and bottom rows from A-diagonal-blocks is that, past the initial miss on accessing these 8 sets in B, we will hit every time after as we fill in each iteration from A. These spots in B, further, will not self-thrash when filling B due to them being to the left/right of each other and not above/below. Even better, we can access A row-wise when filling these storage areas, avoiding all except 4 initial misses and 4 self-thrashes. The same holds for B: we can now fill the B-diagonal-blocks row-wise with the needed corresponding values from the B storage units, minimizing to 4 initial misses and 4 self-thrashes. Despite being arbitrarily chosen, we can depend upon these spots for the left/right halves of the matrix to procedurally conduct these diagonal transpositions in a convoluted, but cache-optimized fashion. Then, we use the general case U-shaped access pattern for all other 8x8 blocks in the matrix as we did before. Although the implementation of this is far more difficult than the 5 lines of *trans*, we finally get below the bound using this technique, arriving at 1268 misses.

==========

61 x 67 WRITEUP

==========

Yet again, we will consider the baseline miss rate by running the transposition functionality with the given typical *trans* function. This yields a base miss rate that is about as high as the 64x64 baseline: 4424.

The uniqueness of this size lies in its rectangular shape, no longer being the nicely consistent set overlay that the 32x32 and 64x64 exhibit. By drawing out some of the sets briefly assuming a start at address 0x0 and A (67x61) and B (61x67) being contiguous, we can see that the sets repeat in patterns of 4 as did the 64x64. However, due to the rectangular shape, we do not see the exact 8x8 block issues as the 64x64, the sets repeating in a diagonal pattern instead of straight vertical. This, as intimidating as it initially appears, ends up being a benefit over the squares–the sets do not overlap so horribly at the 8x8 diagonal blocks.

So, the first attempt made was to simply try blocking tactics that worked well with the other 32x32 and 64x64 sizes. First, we will try to block 8x8 (as the size allows within the rectangle), and fill in the same sideways U-shape used in the 64x64 since the same 4-set repetition pattern occurs here as that size, albeit at an angle. We will leave diagonals alone since they do not align as consistently. That simple access pattern already works very well, yielding 2426 misses.

Since the usage of normal tactics appears to work well, it seemed worthwhile to ensure that the straightforward approach of an 8x8 blocking (without the smaller 4x4 access patterns) does not provide benefit. Upon running, attempt 2 with a simple 8x8 block actually does end up providing an even lower miss count at 2119.

With such a close number to the goal boundary of < 2000, this was likely the approach to toy with further. By simply flipping the outer 2 loops, which effectively move the B storage blocks left-right row-wise and A access blocks up-down column-wise, we succeed in getting the misses below the bound, coming in at a final 1914 misses. Surprisingly, the square matrices ended up being the most complicated to analyze, whereas the rectangular size ended up being simpler to optimize toward the goal–especially since it did not have the same horrendous diagonal overlap thrashing. However, it still has a far higher miss rate than the other matrices, making it still far less efficient in terms of cache usage.