CS 341L Cachelab Part B Report

Roxanne Lutz, Nathan Nail | Fall 2025
8 December 2025
University of New Mexico

Summary

In this new era of technology, computer science cannot be discussed without a same-breath mention of AI. The technology is officially out of the box and can never be put back in. We would be remiss to ignore it. So, how exactly does it fit into the field and learning processes in general?

In this report, we will dive into a straightforward operation with a caveat. Solutions to the given problem must be generated through the old-fashioned way, so to speak. However, we open the door as well to generating solutions with the exciting new technology developed in recent years.

With this experiment, we hope to explore the idea that genAI can be used in a successful direction to cut through a lot of the development time needed for a purely human-generated solution. It does, importantly, come at the cost of losing understanding in the process itself which can be beneficial in learning how to build solutions generally–ones which AI may not always be able to predictably build for us.

Introduction and Motivation

The assigned experiment focused on in this report is as follows.

Optimize with and without genAI tools the trasposition of a matrix for 3 specific matrix sizes (*32x32*, *64x64*, and *61x67*) for a direct-access cache that has 32 sets and 32 bytes held per line.

Primarily, the goal, in a tangible sense is to get both methods to reach under a specified boundary:

- *32x32*: misses < 300
- *64x64*: misses < 1300
- *61x67*: misses < 2000

The overarching motivating factor of the work done in this experiment is to highlight the strengths and weaknesses of both approaches to the problem. On one side, we will explore the lengthy process to develop code to meet this goal entirely from scratch without the benefit of genAI assistance. Likewise, we will dive into the idealistic code generation with genAI, discussing its benefits and drawbacks alike. With all pieces together, we will end by discussing the main differences in the approaches to the problem, focusing on the key takeaways from our results.

This report is organized into 5 sections beyond this Introduction.

1. Methods: General summary of methods used to generate both human and genAI code for transposition.
2. Data Collection: In depth discussion of the process in which human code was created, including failues. Likewise, in depth discussion of prompting attempts with genAI to generate code with the same goal.
3. Exploratory Analysis and Discussion: Discussion of results and primary differences in code approaches.
4. Conclusion: Concluding points of the report.
5. Appendix: Brief inclusion of documentation details about documents included with submission

Methods

This project was primarily split into two natural halves between the two partners:

1. One person's focus was upon, entirely from scratch, approaching the optimization by hand.
2. The other person's focus was on the generation of solutions with generative AI.

The following parts are the methodologies used in generating potential solutions. To analyze the solutions, we will be using the provided cache simulation code to gather miss rate data to better compare and build up approches.

Human

The human generated code was developed in the following order:
1. *32x32* optimization
2. *64x64* optimization
3. *61x67* optimization

The most useful approach to each of these problems was developed by the approach to the first. Picture/color representations of the set overlays on the matrics were generated to a degree in which the problematic miss rate areas were visibly simple to identify. Then, as will be discussed in the data collection portion of this report, solutions to mitigate these miss rates were developed based upon successful changes.

GenAI

The genAI code was developed with ChatGPT 5 mini (with reasoning), which is the version of ChatGPT offered on a trial basis to those who have an OpenAI account but do not pay for any other tier. The model opted to approach the code in a rather monolothic manner and generated code for all three cases to put into one function.

When provided with code from the model, the code that was already in the `transpose-submit` function was directly and fully replaced, even if 1 or 2 out of the three ended up correct during the previous iteration. This was likely a suboptimal approach, but it demonstrates the deficiencies of ChatGPT's "memory". It likely would have been beneficial to split up the code generation into three different chunks to allow the model to analyze each case separately. The results of this method will be discussed later in the report.

Data Collection

The following section goes into depth on the processes of generating solutions, both human and genAI alike.

Human

The following discussion of code generation will be presented from worst attempt to best attempt (or, the final submission that accomplishes a miss count under the minimum bound for each matrix). It should be noted that the attempts were not necessarily in a purely successful order, but will be presented as such for clarity.

The subsections as follows are the order, as mentioned, the human code was generated in overall. They will include, in context, a discussion of attempts made, what was learned from the failure of the attempt, leading to how the final optimized function was created.

32x32

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 could best use it. With the numbers given, as discussed, we have $S = 2^5 = 32$ sets, and each set can hold *32 bytes* (again, $B = 2^5 = 32$). So, we will end up holding *8 ints* in a set, since

*(8 ints) * (4 bytes per int) = 32 bytes to a line/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 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 conducts a straightforward transpose procedure but within *8x8* blocks. 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 verification through trace files, that A and B can be treated as contiguous in memory. The use 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`).

We can add the following code as a special-case to our *8x8* blocking technique.

```
if (i == j) { // if we're in a "diagonal block"
  for (ii = i; ii < i + blocksize; ii++) {
      temp = A[ii][ii]; // give A control over the row
      for (jj = j; jj < j + blocksize; jj++) {
          if (jj != ii) { // skip over all cols of B that will thrash A's row control
              B[jj][ii] = A[ii][jj];
          } // end if
      } // end loop
      B[ii][ii] = temp; // give B control over this set by ending with a diagonal set
  } // end loop
}
```

With this careful management of thrashing, we manage to plummet the miss rate below our boundary: ***288*** misses.

64x64

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 for now, 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 any others improve. 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. There is a definitive difference in how the *64x64* sets overlay the 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 general-case filling of B column-wise thrashes itself wildly 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 efficiency issue: in the general-case (we will cover diagonal blocks in a moment), we are not making the best use of the numbers loaded into the cache line. When we utilize a *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: access the smaller *4x4* sections of the *8x8* block as a sideways-U-shape. Meaning, we access the block in A in the following order: 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 likewise 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 a rightside-up-U-shape, which admittedly will self-thrash twice, but overall will decrease evictions generally. 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 following choice of storage unit is somewhat arbitrary, but it 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 *4* and bottom *4* 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 in the diagonals. 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 continue 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 simplicity of `trans`, we finally get below the bound using this technique, arriving at ***1268*** misses.

Note that all code for these attempts can be found in `trans-human.c`. It will be kept there due to length of the functions, since even snippets of that code are many lines in length.

61x67

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 rows* as did the *64x64*. However, due to the rectangular shape, we do not see the exact *8x8* block issues as the *64x64*. The sets instead repeat in a diagonal pattern instead of straight vertical. This, as intimidating as it initially appears, ends up being a benefit over the squares in terms of meeting our bound; 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 sizes. First, we will try to block *4x4* (as the size allows within the rectangle), and fill with the straightforward blocking technique–as in, no U-shaped access patterns. 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 testing, 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 changing the outer two block iterators from

```
for (i = 0; i < N; i += blocksize) { // rows
    for (j = 0; j < N; j += blocksize) { // columns
```

to

```
for (j = 0; j < M; j += blocksize) { // swapped outer loops to move B blocks rowwise
    for (i = 0; i < N; i += blocksize) { // A blocks col-wise
```

this effectively move the B storage blocks left-right row-wise and A access blocks up-down column-wise. This succeeds in getting the misses below the bound, coming in at a final ***1914*** misses.

GenAI

For the genAI section of the code, ChatGPT 5 mini (with intelligence) was handed the entire PDF, in addition to some instructions provided to it:

> Hello, I am working on the CS:APP cache lab and was wondering how familiar you are with it. Given the assignment PDF, I was hoping that you could review it, confirm to me that you understand the goals of the assignment, and generate output that meets the criteria to the best of your ability.

From this, ChatGPT reiterated what it understood about the assignment and opted to handle all matrix sizes at the same time. To handle this, it uses if statements and calls to `return` in the case that the *32x32* or *64x64* case is being processed. The *61x67* case is processed until the end of the function call.

Below, we discuss:
- The results of each evolution of the code.
- The strategy that the model employed at each setup.
- What ended up being the winning strategy for each case, despite the three cases not able to exist in harmony with each other.
- Future experiments to conduct with the code to see if improvements are made (i.e. what if, instead of one monolothic function call, there were three smaller function calls for each case?).

32x32

The 32x32 case, despite being the "simplest", was a case that was difficult for the model to score well on immediately. It took about four attempts to generate code that hit under *300* misses.
- First run: ran correctly, but was unable to reach the desired miss threshold at 344.

- Second run: all three cases including *32x32* were 0. This does not seem to be well-defined behavior and indicates that the model potentially generated code that broke the tester.
- Third run: After some troubleshooting to deal with the previous run, this was the first iteration of the code to generate output that yielded ***288*** runs. This unfortunately came at the cost of breaking one of the other cases in code.

In addition to succeeding with the *32x32* case, the secondary goal was to remedy issues that were occurring with the final test case of *61x67*. Despite instruction to maintain the previously working case, some issues still managed to arise. This suggests that there were some side effects as the result of the "properly working" *64x64* code that went under the miss rate.

The winning strategy for this case ended up being pretty simple, employing *8x8* blocking and solving each block individually with the naive method. With less data to handle at each step, less cache misses were recorded.

64x64

This code followed a bit of a different story. The development that this code took further suggests that there may be some side effects with some of the variables being used, since we cannot currently provide a plausable explanation for what else could be happening; the model maintained the same structure and order in the code as it did previously, so some faults were unclear in source.

- First run: invalid result. This code iteration produced suboptimal yet valid *32x32* code and right-on-the-spot *61x67* code, which was strange. Taking a quick glance at the code, I was never fully able to determine the cause of the invalidity.
- Second run: Like with other versions, the code that was generated produced 0 misses, which is effectively impossible and indicates that some sort of issue occurred with this iteration. ChatGPT was then prompted to revisit this code and ensure that the setup was being conducted properly.
- Third run: Like with the *32x32* case, the *64x64* code hit well under the miss target of *1300* and netted ***1180*** misses. This came at the cost of breaking the *61x67* test case.

Runs beyond Run 2 were to try and get a round of code that would integrate the successes of these two cases with the first-run success of *61x67*; this proved unsuccessful.

The model's winning strategy proved to be somewhat successful, but after looking at what it took to hit the target miss rate in the human-generated code, this makes sense. From what we can understand of the code, it followed three major phases:

1. Handled the first four rows of each *8x8* block in a standard fashion, starting from the top left.
2. Switched to a different strategy that involved breaking it down into further *4x4* blocks and transposing them in a specific order. This seemed to handle two or three of the four *4x4* blocks that the the code partitioned off. This was the only section of code out of the three cases that used the eight extra variables alotted to it.
3. Finished off the final *4x4* block with a more standard transpose techniques.

61x67

This test case ended up being one of the simplest, with the best results happening with very few lines of code and with the fewest number of attempts.

- First run - instantly met the miss rate criteria. This code managed to just squeeze under the miss rate of *2000* at ***1993*** misses with about 2 dozen lines of code.
- Second run - like with all of the others, this was invalid due to issues with the code.
- Third run - This was the first run to nail the other two cases, but this, somehow, came at breaking the *61x67* case despite explicitly prompting the model to keep the original working code. We have yet to see whether the reason that the code broke was due to actual changes in the code or due to side effects from previous runs.

The winning strategy for the *61x67* case was fairly simple and adopted most of the code from the standard *8x8* blocking strategy. The main change involved extra checks on the outer loops (given the irregular size) to handle not writing too much or too little on either of the dimensions. The inner loops employed the naive implementation.

Second Attempt: Three Separate functions
As mentioned, there was an intimation that there were some side effects occurring under the hood as a result of having all three cases in the same function. As a result, the model was prompted to make another attempt. This time, it was given explicit directions to split the code into three separate cases. This ended up being the solution.

Upon closer inspection, it appears that ChatGPT added in some corrections to the *32x32* code, which makes begs more questions about where exactly the problem was happening before. Regardless, it appears that the model was able to reflect on its own mistakes from a previous conversation and find what went wrong. This could point to the previously mentioned “short-term memory” issues that that were mentioned earlier, or it could be something different.

Code Quality
Some important aspects in measuring code quality involve legibility, conciseness, and correctness. In all, it looks like the model was able to produce code that that was mostly followable.

For the *64x64* case, the code was not terribly concise, which made it difficult to parse. However, it was able to generate a satisfactory result. The other cases could be distilled down to simple code that did not require any lines. This made it concise and easy to follow.

Lastly, correctness seem to be hit or miss. We were not able to get all three of the individual cases to cooperate with each other in the same function, begging the question about if registering three different functions would have been better. However, over three iterations, we were eventually able to get individual cases for all three.

In addition to these main three cases, other basic conventions were followed. The code itself was readable and structurally legible, and the motivations behind implementing them were comprehendible.

nathan note: elaborate more on the comparison between human and genAI code, strengths and weaknesses,
- create csv with the results.
- ask soraya about how to organize when chatgpt decided to make one large unit of a function.
- maybe prompt more? see if there is more exploration to be had that could expand the story.

Exploratory Analysis and Discussion

Results

Overall, we were able to generate successful under-bound solutions both on the human and genAI side of code generation. Following is the comparsion of the best results from both sides.

Sizes	Human Best	GenAI Best
32x32	288	288
64x64	1268	1180
61x67	1914	1993

As we can see from the above, the *32x32* best records met exactly. From the code generated on both ends, both human and genAI came to the same technique of using a temporary variable to store the diagonal value. Further, there were approximately the same number of attempts made to come to that conclusion between the two–human took 2 attempts beyond the `trans` function baseline, and genAI took until the 3rd prompting.

The *64x64* is likely the most interesting to look at the differences of. From the human side, it was by far the most difficult to analyze, requiring much slow analysis of the set patterns and creativity in generating a solution. It also took the most failures of all the matrices, which meant days of attempts being concocted until the final success was met.

Even after all that, genAI was able to produce a better solutions by about *90* misses. Further, it only took 3 attempts of prompting to get a valid, under the target bound solution, which is impressive compared to the pains it took to generate something without being able to use it.

Finally, the *61x67* was however beat by the human by about *80* misses. The two methods used are slightly different in nature, as will be discussed below, but one strategy was clearly still slightly better than the other. However, given that both solutions are under the goal boundary, each are valid in terms of minimizing the misses.

Code Generation Comparisons

The main differences in human and genAI code appear to lie primarily in

1. Time taken to generate solutions.
2. Understandibility of the solution.
3. Ability to build up the solution methodically.

Time

Overall the human generated solutions took around a week to develop to their current best miss rates. A bulk of this time was purely in initially beginning by toying around with code without any real intelligent goals in mind. This provided enough failure to start actually attempting the optimization in a way that was based on the set layout knowledge (as mentioned, hand drawing the patterns or using Excel sheets to create set color fills was very helpful). Then, the building of each attempt, which was built upon the prior failure, involved at times complex implementation details that had to be tested for correctness locally, only to be then tested for miss rate data.

All of this together, culminating to a week's worth of frustration, did however come together into a deeper undestanding of the cache mechanisms and a layer of creative problem solving in this context that is a worthwhile skill for real-world applications.

However, that is not to say that genAI did not have it's uses here. The solutions that genAI, in spite of strange behavior in the initial prompts, took only a day to piece together. In terms of a real-life consequence, a workplace would likely perfer the inclusion of this tool to see if it can cut down the time taken on coming up with a solution to save the most important asset: money. Using this tool can be tremendously useful, then, freeing up time spent on the grunt work and providing a usable solution. This approach could be used postively to generate one or more optimized solutions on which to step in an improve as needed, cutting through the time to develop those initial failures to better focus on a final fine-tuned solution.

Understandibility

We nevertheless must discuss understandibility in two contexts: (1) Understanding the code itself in terms of legibility and conciseness and (2) understanding why the code works.

On first understanding the code from a technical perspective, the human code submitted has the benefit of the comments being as verbose as needed–as much for the developer as for the reader. Considering at least the *64x64* is an admittedly long and somewhat convoluted solution for the human code, these comments are necessary to even begin to piece together what we are doing mechanically.

On the topic of convlusion, the genAI solutions have the opposing perk of having the same or less number lines overall. This is a huge improvement over the human code as most notable in the *64x64* solution, which comes into about *133* lines, as opposed to the *70* line solution from ChatGPT that even performed better than the human solution. Generally, the shorter solution is easier to digest, along with straightforward commenting to interpret the intention of the code.

This topic naturally leads into discussing the second point: understanding why the code works. We are able to see the general mechanics of the genAI code functionality as specified in comments. A drawback, nevertheless, is in missing why this code works.

When discussing the human code development process, and walking through all failures, we can see the exact train of thought that led to the final solution, starting with the most basic algorithm to the best. This is exactly what we miss in ChatGPT code. There has to be further prompting in order to get the explanation of why exactly this works. Even then, there has to be an element of wariness when reading the explanation due to the potential hallucinations of the black-box. The human instead has an explanation that can be built up from failures, learning, and data to support the next step justification, making the process more understandable at a deeper level to both developer and reader. Additionally, the process can be read by others to improve upon, since others may see some sort of successful alternate approach from the development process that the original developer did not recognize. In all, the end answer is not always the only useful piece of the solution process.

Methodical Development

As evidenced a discussed prior, ChatGPT's greatest downfall in this process was the ability for it to ignore prompting, break its own code, and output unpredictably. This was, as seen by the secondary prompting technique of generating methods versus generating solutions all in one fixing many of the issues, a fault of the prompting. Just as people do, ChatGPT performed far better in terms of generating a functional solution by forcing it to separate the problems into separate but related pieces.

Naturally, the human code was generated starting with the perceived simplest matrix first (*32x32*) and working towards the harder cases (*64x64*, *61x67*). This methodical development made it simpler to use known successful techniques in the harder cases and then tailor the solutions according to the unique problem needs. We will therefore hypothesize that this would have been a similiarly better tactic in prompting genAI to produce a faster success. Further trials should be conducted on this strategy to verify, but so far this experiment suggests that this is the case.

Finally, as easy as it is to cast stones at genAI for having unpredictable side-effects and breaking its own code, there were admittedly many times that the human code did the same thing while testing functionality. Typically nothing was being broken outside of a specific method, but the potential for mistakes in complicated solution implementation was very high during human development. So, with more focused prompting, genAI shaves much of that headache successfully.

Conclusion

To put all pieces together, human and genAI code came in quite close in performance, either being exactly the same in performance or one ever slightly beating the other. However, both approaches to the problem resulted in a solution under the desired boundary. So finally, we must ask: which approach is better?

The term "better" here can have different meanings, and it depends on the goal of the exercise. As we have seen, genAI can produce a solution in a few moments that is better than one that took a human days to develop from scratch. From a perspective on time saving, genAI is absolutely a tool that should be utilized to this end. As long as the tool is used appropriately and consciously, such as prompting in small pieces to not overwhelm the model into clashing with itself and being aware of unpredictability, this can be an incredible starting point for solving a problem, or improving upon a decent solution.

But the term "better" can also mean the process of learning how to think in terms of the problem: to learn the intricacies and nuances of caches, how to use knowledge of the cache to improve upon failures, and to truly understand how to work within these confines. Although genAI can a quick, perfectly acceptable answer, the learning process is often far stronger when compounded by failure–something that genAI takes out of the equation entirely.

In conclusion, both approaches had their benefits and drawbacks. Depending on the goal in mind, it is up to personal reponsibility to identify the appropriate usage of this tool given the context.

Appendix

Documentation Details

Some notes on formatting that are likely useful for the reader.

- `trans_human.c`: All methods to yield results discussed in report, as formatted for enabling running of `./test-trans` and `driver.py`. The attempt numbers correspond to the order the attempt is discussed in the report. When running `./test-trans`, all of these attempts will run for the given size. Feel free to comment out the registration of any of these in `registerFunctions()`. `transpose_submit()` will run the best for each size case.

- `ai_trans.c`:
- `trans_human_results.csv`: Results as gathered from `./test-trans` functionality for each size, corresponding to function names in `trans_human.c`. Formatted in the order: size, function name, hits, misses, evictions.
- `ai_trans_results.csv`: