Project 3

Matthew Rackley

May 2023

Introduction

Below is a description of the cache simluator and breaking down cache efficient transposing of matrices.

1 Cache Simulator

My cache simulator takes advantage of the struct data type and creates a 2d array of structs. This seemed the easiest way to describe a cache line and since a cache can just be abstracted to a 2d array of lines where the row is the set number and the column is the line number this felt like an adequate solution to the problem.

After my 2d array of cache is created I ensure that all values are set to 0 then run fscanf on each line of the trace file. If it is "I" I ignore the line and if it's any other I generate the needed information and run my search function which performs the necessary operations for the cache. After all of that is done and I've reached the end of file, I free the memory and run the printSummary command.

The search function checks if the valid bit is 1 and if the tag matches, if it does it registers a hit in the global variable, otherwise if it's not valid it registers a miss and loads the data. If all lines are valid but the tag doesn't match it registers a miss and eviction and increments those global values accordingly. When an eviction needs to take place, it evicts the oldest one (Highest count value).

1.1 Cache Simulator Data Structures

The only data structure being used is the structure for the cache line:

```
struct line {
    unsigned int valid; //Valid bit 0 = invalid, !0 = valid
    unsigned int tag; //Tag for the line.
    unsigned int block;
    unsigned int count; //Incrementing last time accessed.
};
    Elements:
        valid: Integer representing the valid bit.
        tag:   Integer representing the tag.
        block: Integer to represent the block of data.
        count: Integer for last time it was accessed.
    Description: Represents a cache line in memory.
```

1.2 Cache Simulator Functions

```
void args(int argc, char *argv[], unsigned int *s, unsigned int *E,
   unsigned int *b, unsigned int *verbose);
    Parameters:
        argc: The number of arguments being passed.
        argv[]: The array of arguments.
        *s: Grabs the value set for -s
        *E: Grabs the value set for -E
        *b: Grabs the value set for -b
        *v: Whether or not verbose was used in command line.
    Return: Void
    Implementation:
        Takes the globals inside main and assigns them values based on the
            command line input of the function.
```

```
FILE* open(int argc, char* argv[]);
    Parameters:
        argc: Command line argc value for parsing.
        argv: Array of args for parsing.
    Return: File pointer
    Implementation:
        Takes argc and argv and returns a file pointer of an opened file.
```

```
void printBits(unsigned int num, int size);
    Parameters:
        num: Integer value.
        size: How many bits to print.
    Return: Void
    Implementation:
        Prints the bit vector representation of an int. Used for bug
            testing the bit shifting.
```

```
void search(struct line **cache, int E, int tag, int setIndex, int block,
    int *hit, int *miss, int *eviction,unsigned int verbose);
    Parameters:
        **cache: Pointer to the 2d array.
        E: The value of E from command line.
        tag: The isolated tag value.
        setIndex: The isolated set index value.
        block: The data for block that needs to be stored.
        *hit: Pointer to the hit counter.
        *miss: Pointer to the miss counter.
        *eviction: Pointer to the eviction counter.
        verbose: verbose value 0 = false !0 = true.
    Return: Void
    Implementation: Runs the standard search algorithm on cache lines. If
       valid bit is 0 then it's a miss. Otherwise if valid bit is true and
        tag is a match it's a hit. Otherwise if all valid bits are 1 but
       tags don't match it is a miss eviction. Increments count
       accordingly.
```

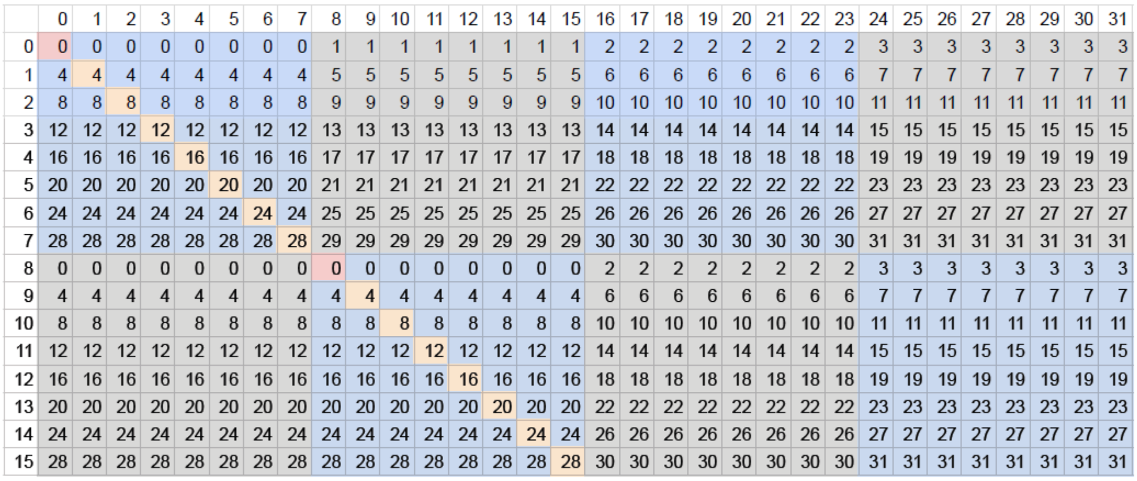
There are other operations performed inside main that weren't yet converted into a standalone function. However they still remain an important part of the program. These include:

1. Assigning values of S and m.
2. Creating the 2d array of struct lines.
3. Bit shifting to get the appropriate values for each of the t_bits, s_bits and b_bits.
4. Handling verbose and operation commands.
5. Freeing the malloc'd memory after use.

2 Transposing Matrices Cache Efficiently

2.1 32x32 Matrix

Transposing a 32x32 matrix using cache efficient methods utilizes the theory of blocking. By looking at the set index of each element in the matrix, we can get a visual representation of how the elements are using the sets in the cache. Given that it's a 1KB cache with 32 Bytes of block size, this means each each cache line will hold 8 integers. Let's look at this in figure 1.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3
1	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7
2	8	8	8	8	8	8	8	8	9	9	9	9	9	9	9	9	10	10	10	10	10	10	10	10	11	11	11	11	11	11	11	11
3	12	12	12	12	12	12	12	12	13	13	13	13	13	13	13	13	14	14	14	14	14	14	14	14	15	15	15	15	15	15	15	15
4	16	16	16	16	16	16	16	16	17	17	17	17	17	17	17	17	18	18	18	18	18	18	18	18	19	19	19	19	19	19	19	19
5	20	20	20	20	20	20	20	20	21	21	21	21	21	21	21	21	22	22	22	22	22	22	22	22	23	23	23	23	23	23	23	23
6	24	24	24	24	24	24	24	24	25	25	25	25	25	25	25	25	26	26	26	26	26	26	26	26	27	27	27	27	27	27	27	27
7	28	28	28	28	28	28	28	28	29	29	29	29	29	29	29	29	30	30	30	30	30	30	30	30	31	31	31	31	31	31	31	31
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3
9	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7
10	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	10	10	10	10	10	10	10	10	11	11	11	11	11	11	11	11
11	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	14	14	14	14	14	14	14	14	15	15	15	15	15	15	15	15
12	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	18	18	18	18	18	18	18	18	19	19	19	19	19	19	19	19
13	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	22	22	22	22	22	22	22	22	23	23	23	23	23	23	23	23
14	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	26	26	26	26	26	26	26	26	27	27	27	27	27	27	27	27
15	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	30	30	30	30	30	30	30	30	31	31	31	31	31	31	31	31

Figure 1: Portion of a 32x32 Matrix.

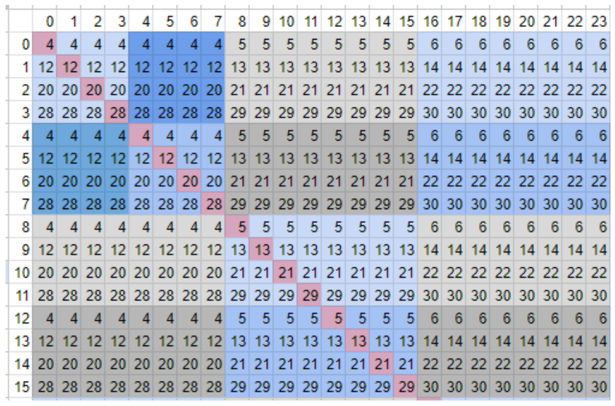
As you can see, each cache line will load 8 elements and those are aligned perfectly with a 32x32 matrix. If we run transposing on each individual blocks of 8x8 integers then run naive transposing inside those blocks will get us most of the way there. The issue lies on the diagonal.

According to multiple readings and sources, saving the diagonal transposition for last is always of benefit because if not it will overwrite other values that will later need to be fetched again. Thus increasing the amount of cache misses. To do this I wrote an if statement that checked if it was a diagonal element. If it was, I dumped it to a temporary value before running the rest of the transposition of that block. This brought me over to full points in this assignment.

Final Total of Misses: 287

2.2 64x64 Matrix

This matrix was by far the hardest of the three. Let's look at a small section of the matrix in figure 2.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6
1	12	12	12	12	12	12	12	12	13	13	13	13	13	13	13	13	14	14	14	14	14	14	14	14
2	20	20	20	20	20	20	20	20	21	21	21	21	21	21	21	21	22	22	22	22	22	22	22	22
3	28	28	28	28	28	28	28	28	29	29	29	29	29	29	29	29	30	30	30	30	30	30	30	30
4	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6
5	12	12	12	12	12	12	12	12	13	13	13	13	13	13	13	13	14	14	14	14	14	14	14	14
6	20	20	20	20	20	20	20	20	21	21	21	21	21	21	21	21	22	22	22	22	22	22	22	22
7	28	28	28	28	28	28	28	28	29	29	29	29	29	29	29	29	30	30	30	30	30	30	30	30
8	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6
9	12	12	12	12	12	12	12	12	13	13	13	13	13	13	13	13	14	14	14	14	14	14	14	14
10	20	20	20	20	20	20	20	20	21	21	21	21	21	21	21	21	22	22	22	22	22	22	22	22
11	28	28	28	28	28	28	28	28	29	29	29	29	29	29	29	29	30	30	30	30	30	30	30	30
12	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6
13	12	12	12	12	12	12	12	12	13	13	13	13	13	13	13	13	14	14	14	14	14	14	14	14
14	20	20	20	20	20	20	20	20	21	21	21	21	21	21	21	21	22	22	22	22	22	22	22	22
15	28	28	28	28	28	28	28	28	29	29	29	29	29	29	29	29	30	30	30	30	30	30	30	30

Figure 2: Portion of a 64x64 Matrix.

The integers filling in this small representation of the matrix represent the set value that each element evaluates to. With the cache size being 1KB and 32 bytes of block available to store information. This means that each line can hold 8 integers. It also looks like the 64x64 matrix can take advantage by setting block size to 8 similarly to the 32x32 problem. However an issue is presented in the darker shaded regions of each color. There is a repeating pattern of set indexes inside each block. If we were to naively utilize the solution for 32x32 and 61x67 matrix, we would be constantly overwriting lines in the cache that were saved.

Labeling the vertical distance between repeating the same line as a "stride", we can see that the stride is 4. Which is half of the vertical block. So how do we deal with a block size that also has repeating sets within itself along a vertical stride? Looking at the structure of the entire array, we can make a few educated guesses about the situation:

1. Set numbers repeat in sub blocks.
2. Cannot directly transpose due to over-writing values prematurely across multiple cached lines.
3. Reading from A will ALWAYS result in a miss on every new/different row and column block.
4. We should take advantage of the unavoidable cache misses to do an operation into matrix B since the information will already be loaded into cache.
5. We need to handle each stride section separately.
6. Matrix A and B share the same cache space.
7. The limitation is only half of the block can be stored in cache.

The answer took a lot of thinking and discussing to recognize that the limitation is that only half of the block can be stored in cache at any one time. Hence the stride. So the answer is reading each stride/block combination and storing them into unused areas inside matrix B that do not conflict with the same set values. This would be the same as shifting all the values to the right along the column index by the size of the block which was eight. Here is a step by step process of how the function works. Let's start with figure 3.

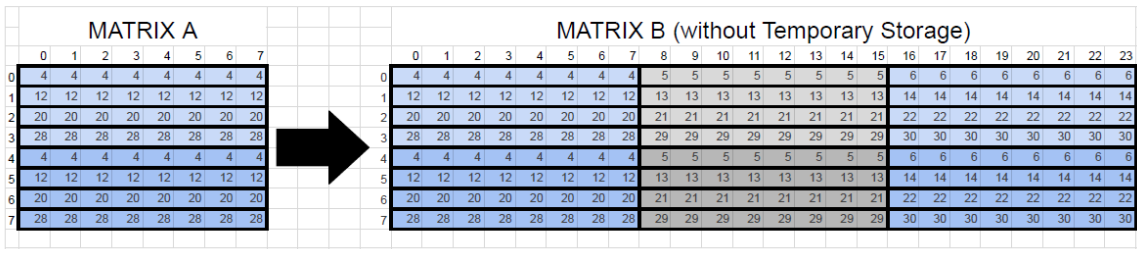


Figure 3: Beginning of the cache friendly transpose.

Let the leftmost grid represent a singular block that we need to transpose in matrix A into matrix B. Let the leftmost block in matrix B be the final location that the elements from A need to be transposed to. The elements inside the matrix are labeled with their set value as to show the repeating stride as well as showing how the set index changes by 1 for each row once you travel a block to the right. Now lets highlight each stride/block segment so we can follow it's transposition. See figure 4.

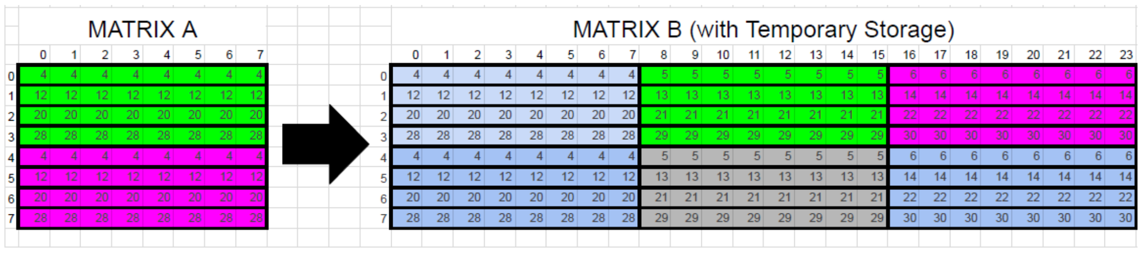


Figure 4: Dumping to temporary storage into Matrix B.

The first step is to use blocks ahead of it's final location and load each stride separately. This is because if we were to access row 0 and row 4 this would result in more misses than needed. So we only access each stride to it's fullest before loading the second subsection into cache.

Note where the green and pink values are stored in matrix B and that their set indexes now differ rather than repeat. Now that we have values loaded into matrix B with different set indexes, we can perform a more optimal transformation into the final leftmost block of B. See figure 5.

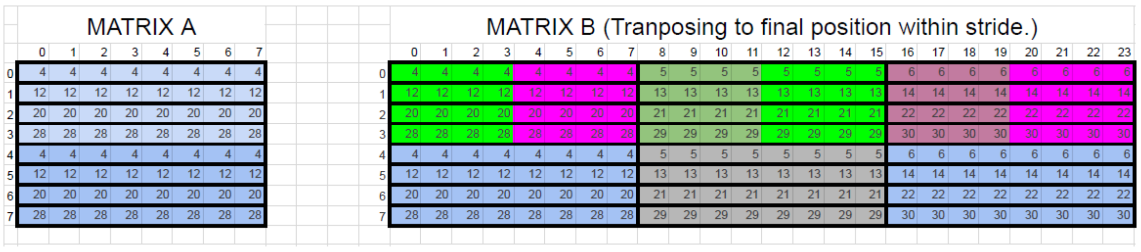
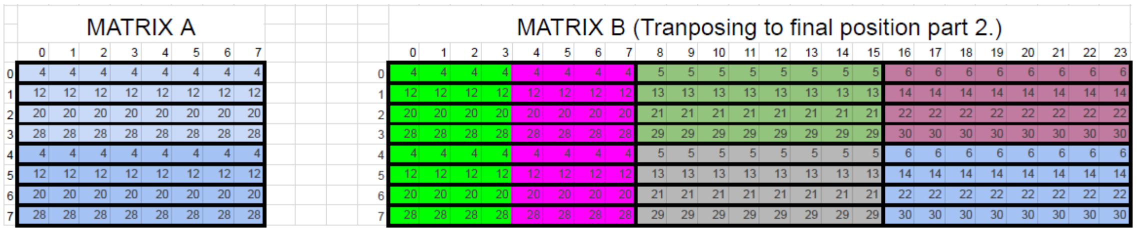


Figure 5: Beginning final transpose.

In figure 5 we see that Matrix A is no longer needed in the transpose to the final location. We take the leftmost half of green and the leftmost half of pink and transpose those values within the stride/block segment. Note that the rule about only performing operations within a single stride still holds even for writing. Thus we need to divide the stride up in additional sub-problems to help with the transposition.



MATRIX A

	0	1	2	3	4	5	6	7
0	4	4	4	4	4	4	4	4
1	12	12	12	12	12	12	12	12
2	20	20	20	20	20	20	20	20
3	28	28	28	28	28	28	28	28
4	4	4	4	4	4	4	4	4
5	12	12	12	12	12	12	12	12
6	20	20	20	20	20	20	20	20
7	28	28	28	28	28	28	28	28

MATRIX B (Tranposing to final position part 2.)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6
1	12	12	12	12	12	12	12	12	13	13	13	13	13	13	13	13	14	14	14	14	14	14	14	14
2	20	20	20	20	20	20	20	20	21	21	21	21	21	21	21	21	22	22	22	22	22	22	22	22
3	28	28	28	28	28	28	28	28	29	29	29	29	29	29	29	29	30	30	30	30	30	30	30	30
4	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6
5	12	12	12	12	12	12	12	12	13	13	13	13	13	13	13	13	14	14	14	14	14	14	14	14
6	20	20	20	20	20	20	20	20	21	21	21	21	21	21	21	21	22	22	22	22	22	22	22	22
7	28	28	28	28	28	28	28	28	29	29	29	29	29	29	29	29	30	30	30	30	30	30	30	30

Figure 6: Beginning final transpose.

Finally figure 6 shows that all the stored data has been used to transpose the information into the final location in the leftmost block in this representation of matrix B. We would then run the same operation by moving along to another block of matrix A and perform the same function. It is simple enough to write edge case checkers to make sure that the temporary storage wraps around to the next line when it reaches the end. Additionally this strategy gives us enough wiggle room to perform a naive transpose on the last two blocks of matrix A into B due to the limitation of temporary storage space.

I also had some additional thoughts on how to improve upon this but did not have time to test:

1. It might be possible instead of running a naive transpose on the last few blocks of matrix A to utilize the now used sections of A as temporary storage before shifting all elements into their final section of B.

2. It might also be possible to use only one storage section of matrix A by having some sort of in-place operation on cache lines preserving misses and loads. I doubt this is possible since the transposed location will occupy the same cache space in memory, would require more testing and comprehension.

Overall the concept of this problem seemed very difficult and hard to comprehend. I doubt without the many hints to print out the sets and create an excel sheet and talking to others I would have figured this out in time.

Final Number of misses: 1148

2.3 61x67 Matrix

This was the second easiest to solve. Running the transpose function for the 32x32 matrix on this non-square matrix resulted in a decent score. Narrowing down the block size to see if this was affected anyways I tried 1,2,4,8,16, and 32. 16 seemed to be the sweet spot for this that got me full points. After talking with a tutor I attempted 17 and got a few cache misses lower than 16. Additionally I'm unsure why but alternating the variables for the two outer for loops that iterate the blocking from row-major to col-major improved the misses by a significant margin. I am assuming it's because each row of the matrix isn't easily divisible by the block size so prioritizing a specific direction of travel helped with misses.
Pre-Swap Misses: 1949 Swap Misses: 1812

Final Total of Misses: 1812