ASSIGNMENT-2 : CACHES

INSTRUCTIONS:

1. **SUBMIT BY PUSHING TO YOUR OWN REPOSITORY IN GITHUB CLASSROOM.**
2. **SUBMISSION SHOULD HAVE A cache.s FILE, A PDF OF YOUR EXPLANATIONS, AND UPDATED CODE OF TRANSPOSE PROBLEMS**

Problem-1

Write a RISC-V code for the following C code:

```
#define size 1024
int main() {
     int reps, int step, option, i;
     reps=10;
     step=2;
     option=1;
     char a[size];
     for(i=0;reps;i++) {
          for(j=0;j<size; j+=step){
               if(option==0)
                    a[0]=0;
               else
                    a[j]+=1;
          }
     }
     return 1;
}
```

Now simulate the code in cache simulator of Venus.
https://venus.cs61c.org/

Answer the following common questions first:

- How big is your cache block?
- How many consecutive accesses (taking into account the step size) fit within a single block?
- How much data fits in the WHOLE cache?
- How far apart in memory are blocks that map to the same set (and could create conflicts)?
- What is your cache's associativity?

- Where in the cache does a particular block map to?

Setting-1:

Program Parameters: (set these by initializing the a registers in the code)

- **Array Size :** 128 (bytes)
- **Step Size** : 8
- **Rep Count** : 4
- **Option** : 0

Cache Parameters: (set these in the Cache tab)

- **Cache Levels**: 1
- **Block Size**: 8
- **Number of Blocks**: 4
- **Enable?**: Should be green
- **Placement Policy**: Direct Mapped
- **Associativity**: 1 (Venus won't let you change this with your placement policy, why?)
- **Block Replacement Policy**: LRU

Answer the following pertaining to this setting:

1. When considering why a specific access is a miss or hit: Have you accessed this piece of data before? If so, is it still in the cache or not?
2. What combination of parameters is producing the hit rate you observe? Write your answer in the form "[parameter A], [parameter B]" where the two parameters complete the following response: "Because [parameter A] in bytes is exactly equal to [parameter B] in bytes." **Note:** Don't forget that 'cache size' is a valid parameter that you implicitly set by choosing the block size and the # of blocks.
3. What is our hit rate if we increase Rep Count arbitrarily? Write your answer as a decimal (e.g. "1.0" if the HR is 100%).
4. How could we modify one program parameter get the highest possible hit rate? Write your answer in the form "[parameter], [value]" where [parameter] is the program parameter you want to change and [value] is the value you want to change it to. **Note**: We don't care if we access the same array elements. Just give us a program parameter modification that would increase the hit rate. However, do make sure that your proposed value is valid.

Setting-2

Program Parameters: (set these by initializing the a registers in the code)

- **Array Size :** 256 (bytes)
- **Step Size**: 2
- **Rep Count** : 1
- **Option** : 1

Cache Parameters: (set these in the Cache tab)

- **Cache Levels**: 1
- **Block Size**: 16
- **Number of Blocks**: 16
- **Enable?**: Should be green
- **Placement Policy**: N-Way Set Associative
- **Associativity**: 4
- **Block Replacement Policy**: LRU

Answer the following now:

1. How many **memory accesses** are there per iteration of the **inner loop** (not the one involving Rep Count)?
2. What is the **repeating hit/miss pattern**? Write your answer in the form "MMHHMH" and so on, where your response is the **shortest** pattern that gets repeated.
3. Keeping everything else the same, what does our hit rate approach as Rep Count goes to infinity? Try it out by changing the appropriate program parameter and letting the code run! Write your answer as a decimal.

Setting-3

Program Parameters: (set these by initializing the a registers in the code)

- **Array Size** : 128 (bytes)
- **Step Size** : 1
- **Rep Count** : 1
- **Option** : 0

Cache Parameters: (set these in the Cache tab)

- **Cache Levels**: 2

NOTE: Make sure the following parameters are for the L1 cache! (Select L1 in the dropdown right next to the replacement policy)

- **Block Size**: 8
- **Number of Blocks**: 8
- **Enable?**: Should be green
- **Placement Policy**: Direct Mapped
- **Associativity**: 1
- **Block Replacement Policy**: LRU

NOTE: Make sure the following parameters are for the L2 cache! (Select L2 in the dropdown right next to the replacement policy)

- **Block Size**: 8
- **Number of Blocks**: 16
- **Enable?**: Should be green
- **Placement Policy**: Direct Mapped
- **Associativity**: 1
- **Block Replacement Policy**: LRU

Answer the following:

1. What is the hit rate of our L1 cache? Our L2 cache? Overall? Write your answer in the form "[L1 HR], [L2 HR], [Overall HR]" where each hit rate is a decimal rounded to two places.
2. How many accesses do we have to the L1 cache total? How many of them are misses? Write your answer in the form "[# of L1 accesses], [# of L1 misses]".
3. How many accesses do we have to the L2 cache total? **HINT:** Think about how this relates to the L1 cache (think about what the L1 cache has to do in order to make us access the L2 cache)?
4. What program parameter would allow us to increase our L2 hit rate, but keep our L1 hit rate the same?
5. Do our L1 and L2 hit rates decrease (-), stay the same (=), or increase (+) as we (1) increase the number of blocks in L1, or (2) increase the L1 block size? Write your answer in the form "[1_L1], [1_L2], [2_L1], [2_L2]" (e.g. if I thought L1 will stay the same for both modifications while L2 will decrease for the first and increase for the second, I would answer "=, -, =, +").

Problem-2

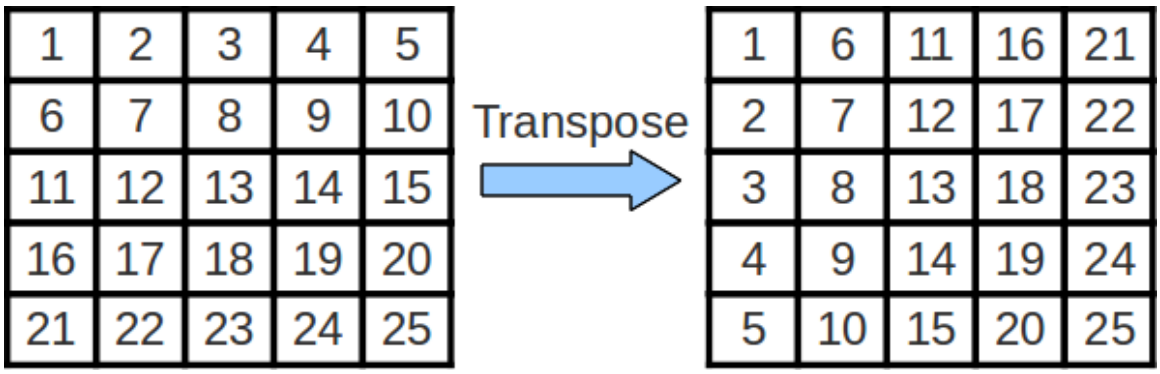
Matrix operations and loop ordering.

Refer to the code on matrix multiplication discussed in class and answer the following:

1. Which 2 orderings perform best for these 1000-by-1000 matrices? Write your answer in the form "[Ordering1], [Ordering2]" (e.g. "ijk, ikj").
2. Explain why this ordering works best.
3. Which 2 orderings perform the worst?

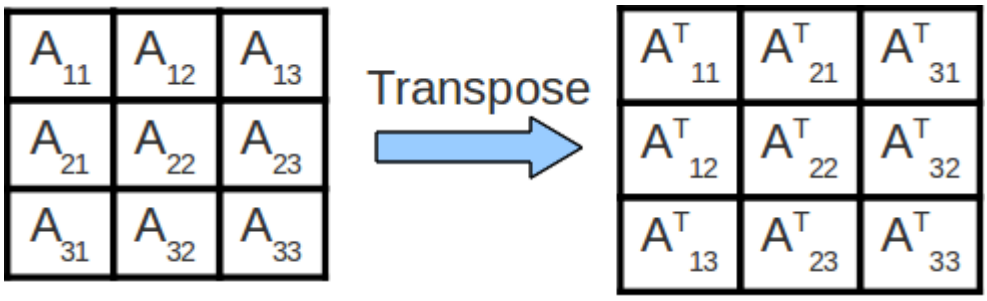
Matrix Transpose:

Sometimes, we wish to swap the rows and columns of a matrix. This operation is called a "transposition" and an efficient implementation can be quite helpful while performing more complicated linear algebra operations. The transpose of matrix A is often denoted as A^{TT}.



Cache Blocking

In the above code for matrix multiplication, note that we are striding across the entire A and B matrices to compute a single value of C. As such, we are constantly accessing new values from memory and obtain very little reuse of cached data! We can improve the amount of data reuse in the caches by implementing a technique called cache blocking. More formally, cache blocking is a technique that attempts to reduce the cache miss rate by further improving the temporal and/or spatial locality of memory accesses. In the case of matrix transposition, we consider performing the transposition one block at a time.



In the above image, we transpose each submatrix A_{ij} of matrix A into its final location in the output matrix, one submatrix at a time. It is important to note that transposing each individual subsection is equivalent to tranposing the entire matrix.

Since we operate on and finish transposing each submatrix one at a time, we consolidate our memory accesses to that smaller chunk of memory when transposing that particular submatrix, which increases the degree of temporal (and spatial) locality that we exhibit, which makes our cache performance better, which makes our program run faster.

This (if implemented correctly) will result in a substantial improvement in performance. For this lab, you will implement a cache blocking scheme for matrix transposition and analyze its performance.

Your task is to implement cache blocking in the transpose_blocking() function inside the code provided in transpose.c :

You may NOT assume that the matrix width (n) is a multiple of the blocksize. By default, the function does nothing, so the benchmark function will report an error. After you have implemented cache blocking, you can compile and run your code by typing.

```
$ make

$./transpose <n> <blocksize>
```

where n, the width of the matrix, and blocksize are parameters that you will specify. You can verify that your code is working by setting n=10000 and blocksize=33. The blocked version should finish significantly faster.

The following section is meant to serve as a guideline for if you have no idea how to start. If you think you know how to use the parameter blocksize, then just jump right in and get started.

Start by looking at the transpose_naive function included in the file. Notice that the index y strides vertically across the WHOLE src matrix in one iteration of the outer loop before resetting to 0. Another way to say this is that the index x only updates

after y is done going from 0 all the way to n. This is the behavior which we want to change. We want to step not stride across the array indices.

fill out dst square chunks at a time, where each square chunk is of dimension blocksize by blocksize.

Instead of updating x only when y goes through ALL of 0 through n, we want to jump down to the next row of dst after we stride across the width (horizontal axis) of just a single block. How big is a block? Exactly the number of integers specified by the parameter blocksize. In addition, we only want to stride vertically through the height (vertical axis) of a block before we move on to the next block. We don't want to make x stride all the way down n rows of dst before we move on to the next block.

Hint: A standard solution needs 4 (four) for loops.

Finally, since we can't assume that n is a multiple of blocksize, the final block column for each block row will be a little bit cut-off, i.e. it won't be a full blocksize by blocksize square. In addition, the final block row will all be truncated. To fix this problem, you can do the exercise assuming that n is a multiple of the blocksize and then add in a special case somewhere to do nothing when your indices reach out of bounds of the array.

Once your code is working, complete the following exercises and record your answers.

Setting-1 : Changing array sizes

Fix the blocksize to be 20, and run your code with n equal to 100, 1000, 2000, 5000, and 10000. Record your output in exercise3.txt.

Question 1: At what point does cache blocked version of transpose become faster than the non-cache blocked version?

Question 2: Why does cache blocking require the matrix to be a certain size before it outperforms the non-cache blocked code?

(Sanity check: the blocked version isn't faster than the naive version until the matrix size is sufficiently big.)

Setting-2: Changing blocksize

Fix n to be 10000, and run your code with blocksize equal to 50, 100, 500, 1000, 5000. Record your output in exercise3.txt.

Question 3: How does performance change as blocksize increases? Why is this the case?

(Sanity check: as you increase blocksize, the amount of speedup should change in one direction, then change in the other direction.)

Notice that in neither of the last two exercises did we actually know the cache parameters of our machine. We just made the code exhibit a higher degree of locality, and this magically made things go faster! This tells us that caches, regardless of their specific parameters, will always benefit from operating on code which exhibits a high degree of locality.