Lab 8

Computer Architecture I ShanghaiTech University
Lab 7 Lab 8

Exercise 1: Cache Visualization

Caches is typically one of the hardest topics for students in Computer Architecture to grasp at first. This exercise will use some cool cache visualization tools in Venus to get you more familiar with cache behavior and performance terminology with the help of the file `cache.s`. At this point, read through `cache.s` to get a rough idea of what the program does.

To get started with each of the scenarios below:

1. Copy the code in `cache.s` to Venus.
2. In the code for `cache.s`, set the appropriate Program Parameters as indicated at the beginning of each scenario (by changing the immediates of the commented `li` instructions in `main`).
3. Click Simulator, in right partition, there is a tag called `Cache`.
4. Set the appropriate Cache Parameters as indicated at the beginning of each scenario.
5. Click Simulator-->Assemble & Simulate from Editor.
6. Click Simulator-->Step and see the cache states.

Get familiar with the parameters in cache windows:

1. Cache Levels: The number of layers your cache simulator will have. We will only use L1 cache in this lab, but you can play with it to learn more.
2. Block Size: Every block's size, should be a power of 2, take a quick review of the lecture content, the number of offset should be decided by block size.
3. Number of Blocks: How many blocks your cache have totally. Attention, the number is the total number, no matter how many ways you choose, therefore, if you want to satisfy the requirement, please take care of it (divide associativity).
4. Associativity: The number of ways, only if you select the N-way Set Associative can you change this value.
5. Cache size: The result of block size multiply number of blocks. You cannot change it.

The Data Cache Simulator will show the state of your data cache. Please remember that these are running with your code, so if you reset your code, it will also reset your cache and memory status.

If you run the code all at once, you will get the final state of the cache and hit rate. You will probably benefit the most from setting breakpoints at each memory access to see exactly where the hits and misses are coming from. Themethod to set a breakpoint in Venus is just click the corresponding line in the simulator, if the line become red, that means your program will stop when the execution meets that line.

Simulate the following scenarios and record the final cache hit rates. Try to reason out what the hit rate will be BEFORE running the code. After running each simulation, make sure you understand WHY you see what you see (the TAs will be asking)!

Do not hesitate to ask questions if you feel confused! This is perfectly normal and the TA is there to help you out!

Good questions to ask yourself as you do these exercises:

- How big is your cache block? How many consecutive accesses fit within a single block?
- How big is your cache? How many jumps do you need to make before you "wrap around?"
- What is your cache's associativity? Where can a particular block fit?
- Have you accessed this piece of data before? If so, is it still in the cache or not?

Scenario 1:

Cache Parameters:

- **Cache Levels:** 1
- **Block Size (Bytes):** 8
- **Number of blocks:** 4
- **Associativity:** 1 (Venus won't let you change this, why?)
- **Cache Size (Bytes):** 32 (Why?)
- **Placement Policy:** Direct Mapping
- **Block Replacement Policy:** LRU
- **Enable current selected level of the cache.**

Program Parameters:

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

Checkoff

1. What combination of parameters is producing the hit rate you observe? (Hint: Your answer should be the process of your calculation.)
2. What is our hit rate if we increase Rep Count arbitrarily? Why?
3. How could we modify our program parameters to maximize our hit rate?

Scenario 2:

Cache Parameters:

- **Cache Levels:** 1
- **Block Size (Bytes):** 16
- **Number of blocks:** 16
- **Associativity:** 4
- **Cache Size (Bytes):** 256
- **Placement Policy:** N-Way Set Associative

- **Block Replacement Policy:** LRU
- **Enable current selected level of the cache.**

Program Parameters:

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

Checkoff

1. What combination of parameters is producing the hit rate you observe? (Hint: Your answer should be the process of your calculation.)
2. What happens to our hit rate as Rep Count goes to infinity? Why?
3. Suppose we have a program that uses a very large array and during each Rep, we apply a different operator to the elements of our array (e.g. if Rep Count = 1024, we apply 1024 different operations to each of the array elements). How can we restructure our program to achieve a hit rate like that achieved in this scenario? (Assume that the number of operations we apply to each element is very large and that the result for each element can be computed independently of the other elements.) What is this technique called? (Hint)

Scenario 3:

Cache Parameters:

- **Cache Levels:** 1
- **Block Size (Bytes):** 16
- **Number of blocks:** 16
- **Associativity:** 4
- **Cache Size (Bytes):** 256
- **Placement Policy:** N-Way Set Associative
- **Block Replacement Policy:** Random
- **Enable current selected level of the cache.**

Program Parameters:

- **Array Size:** 256
- **Step Size:** 8
- **Rep Count:** 2
- **Option:** 0

Checkoff

1. Run the simulation a few times. Every time, set a different seed value (bottom of the cache window). Note that the hit rate is non-deterministic. What is the

range of its hit rate? Why is this the case? ("The cache eviction is random" is not a sufficient answer)
2. Which Cache parameter can you modify in order to get a constant hit rate? Record the parameter and its value (and be prepared to show your TA a few runs of the simulation). How does this parameter allow us to get a constant hit rate? And explain why the constant hit rate value is that value.
3. Ensure that you thoroughly understand each answer. Your TA may ask for additional explanations.

Exercise 2: Loop Ordering and Matrix Multiplication

If you recall, matrices are 2-dimensional data structures wherein each data element is accessed via two indices. To multiply two matrices, we can simply use 3 nested loops, assuming that matrices A, B, and C are all n-by-n and stored in one-dimensional column-major arrays:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i+j*n] += A[i+k*n] * B[k+j*n];
```

Matrix multiplication operations are at the heart of many linear algebra algorithms, and efficient matrix multiplication is critical for many applications within the applied sciences.

In the above code, note that the loops are ordered i, j, k. If we examine the innermost loop (k), we see that it moves through B with stride 1, A with stride n and C with stride 0. To compute the matrix multiplication correctly, the loop order doesn't matter. However, the order in which we choose to access the elements of the matrices can have a large impact on performance. Caches perform better (more cache hits, fewer cache misses) when memory accesses exhibit spatial and temporal locality. Optimizing a program's memory access patterns is essential to obtaining good performance from the memory hierarchy.

Take a glance at matrixMultiply.c. You'll notice that the file contains multiple implementations of matrix multiply with 3 nested loops.

Compile this code using the '-O3' flag. *It is important here that we use the '-O3' flag to turn on compiler optimizations*. Compile and run this code and then answer the questions:

a. Which ordering(s) perform best for 1000-by-1000 matrices?
b. Which ordering(s) perform the worst?
c. How does the way we stride through the matrices with respect to the innermost loop affect performance?

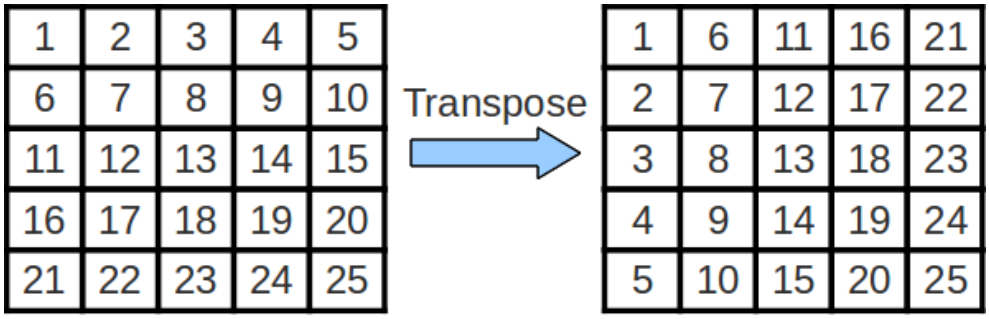
Checkoff: Be prepared to explain your responses to the above questions to your TA.

Exercise 3: Cache Blocking and Matrix Transposition

Note: The term cache blocking is not referring to the same thing as the cache blocks we talked about when discussing the structure of caches.

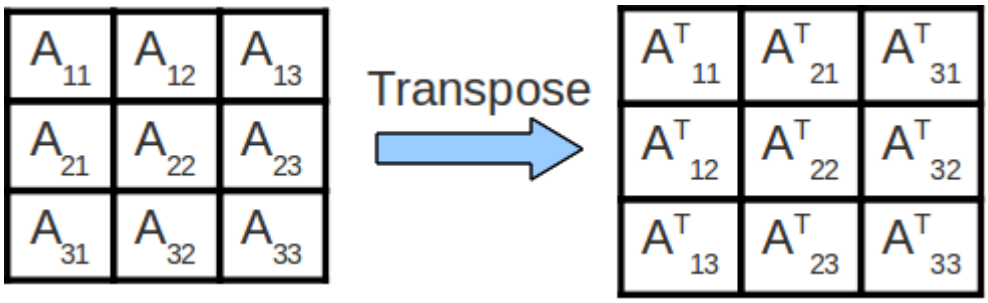
Matrix Transposition

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^T.



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 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 block A_{ij} of matrix A into its final location in the output matrix, one block at a time. With this scheme, we significantly reduce the magnitude of the working set in cache during the processing of any one block. 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 transpose.c. By default, the function does nothing, so the benchmark function will report an error. **You may NOT assume that the matrix width (`n`) is a multiple of the blocksize.** After you have implemented cache blocking, compile it with '-O3'and then execute it:

```
$ ./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`=1000 and `blocksize`=33 Once your code is working, complete the following exercises and record your answers (we will ask about it during checkoff).

Part 1: Changing Array Sizes

Fix the `blocksize` to be 20, and run your code with `n` equal to 100, 1000, 2000, 5000, and 10000. At what point does cache blocked version of transpose become faster than the non-cache blocked version? Why does cache blocking require the matrix to be a certain size before it outperforms the non-cache blocked code?

Part 2: Changing Blocksize

Fix `n` to be 10000, and run your code with `blocksize` equal to 50, 100, 500, 1000, 5000. How does performance change as the blocksize increases? Why is this the case?

> **Checkoff:** Be prepared to explain your responses to the above questions to your TA.

Schwertfeger, Sören <`soerensch` *AT* `shanghaitech.edu.cn`>
Chundong Wang <`wangchd` *AT* `shanghaitech.edu.cn`>
Modeled after UC Berkeley's CS61C.
Last modified: 2020-04-18