Lab 8

Computer Architecture I ShanghaiTech University
Lab 7 Lab 8 Lab 9

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

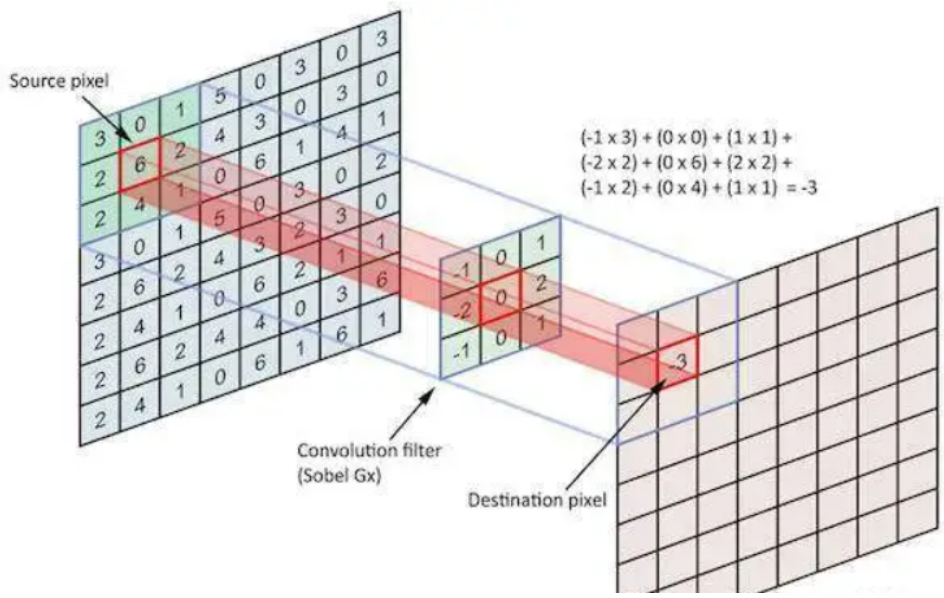
range of its hit rate? Why is this the case? ("The cache eviction is random" is not a sufficient answer)
2. Which Cache parameter can you modify in order to get a constant hit rate? Record the parameter and its value (and be prepared to show your TA a few runs of the simulation). How does this parameter 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: Matrix multiplication and Execution order

Gaussian Blur On Image

In image processing, a Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function. Mathematically, applying a Gaussian blur to an image is the same as convolving the image with a Gaussian function. In this lab, we adopt a 1-dimensional Gaussian distribution kernel, and the blurring process is done in two steps: Given image A as our input, we first convolve the kernel over the rows of image A to produce a horizontally blurred image B. We then convolve the kernel over the columns of image B to produce a horizontally and vertically blurred image C. The image C is our final blurred image

The process of convolving a image works like below. It consists a simple multiplication and add.



We provide an implement of Gaussian Blur here and your job is to optimize the program without changing the algorithm. To make things easy, you only need to focus on `apply_gb_fast.c`.

In `apply_gb_fast.c`, there is a function called `apply_gb()`. This function will receive two parameters, where `Image a` indicates the input image and `FVec gv` indicates the kernel. It will call `gb_h` and `gb_v` to do convolvtion horizontally and vertically. `gb_h` and `gb_v` will return a new image.

At first, you can use `make base_test` to run the origin version of gaussian blur, which will show you the time of `gb_h` and `gh_v`. Then, you will find there is a gap between the two time.

Then, to optimize the program, we can take another look on the execution order of Gaussian Blur. The vertial onvolvtion equals to apply horizontal convolvtion to a transposed matrix. Thus, we can first transpose the image , apply horizontal concolvtion to it and finally transpose it angin to get a correct result. In this way, we can optimize the memory access performance of the program.

In `apply_gb_fast.c`, there is a completed function `transpose()`, which will return a transposed image of the input image. You can use it to optimize your program following the method mentioned above.

You can run `make all` to test your modified program. The program `test_accuracy` will test the result of your program and output the average error between your result and the correct result.

(Optional)

To make the program even faster, we can apply **cache blocking** to the function `transpose()`, which can be learned from here

Checkoff: Show your program to your TA and answer the following questions:

1. Why there is a gap between `gb_v` and `gb_h` ?

2. Why the changed execution order will achieve a better performance even if we do more things(transpose)?

Exercise 3: Effort of Cache Miss

Cache Friendly Data Structure

Some data structures are cache friendly while others will cause a lot of cache miss. For those programs whose workloads are mainly in data access instead of calculation, cache miss will influence to performance significantly.

Demo Web Log Engine

Every time there is someone who visits our website, the website log engine will record some information such as ip and state. The website log engine will do some operations on the recorded logs, where the main work is accessing data. To simplify the situation, we provide a demo web log engine which will tranverse all logs and do map function to some information

Your task is to modify the given `struct log_entry` in `log_fast.c` to make the data structure more cache friendly.You can use the following command to test your program's

performance compared with the origin one.(If you use a virtual machine, you may need to increase the memory if there comes a sigment fault)

```
$ make all
```

Hint

In the function `tranverse()`, we only use three members in the log_entry. However, the three members are seperated be the large arries, which make them placed into three differnet cache lines. Thus, each access of one element in the array `logs` will cause 3 cache misses.

> **Checkoff:** Show your result to your TA and explain why you do this modification.

Linjie Ma <malj *AT* shanghaitech.edu.cn>
Modeled after UC Berkeley's CS61C.
Last modified: 2022-04-5