CSE 378: 08 AU - Anderson

Homework 3: Cache Simulator and Optimization

Due: Monday, November 24, 2008 at 5:00 PM.

Because it can take much more time to build hardware than to write software, microprocessor designers always build software simulators to test and optimize their designs. You will be writing a cache simulator and using it to observe how cache parameters can affect the performance of a simple image-processing algorithm.

This assignment should be done on one of the linux machines in the lab. The skeleton code for this assignment can be found in `cache-sim.tgz` on the class homepage. This file contains the c code that you will need to augment and modify as well as image handling code and a few test files. All of your changes should be confined to `cache.c` and `sobel.c`.

Cache Simulator

Your implementation will simulate a general write-back cache using the LRU replacement policy.

You will notice that there is a structure defined in `cache-sim.h` to represent a cache line. It contains fields for all of the cache line attributes and data. All of the data types are either `BYTE` or `WORD`. These are data types that are also defined in the header file to represent the untyped data found in cache hardware.

To complete the simulator, add code to all the spots marked with `TODO` comments in `cache.c`. All of the functions you need have been defined. Some of the bookkeeping functions are already implemented for you.

The LRU replacement policy means that the cache line that was accessed least recently will be evicted when there is contention. The skeleton code provides a simple quasi-clock to use for keeping track of when cache lines are accessed. Use the `get_current_ts` function to get the current time.

When you are ready, compile your simulator using `make`. To help you test and for the second part of the assignment, the skeleton code includes an edge-detection routine that uses your cache. The compiled executable, `cache-sim`, takes an input jpeg and a filename for the output image. Try it out using one of the test images included in the tar file. Running `cache-sim` without any parameters will print usage information. There is also a global variable, `_use_cache`, defined in sobel.c that you can use to turn on and off the cache to help with testing.

Cache Optimization

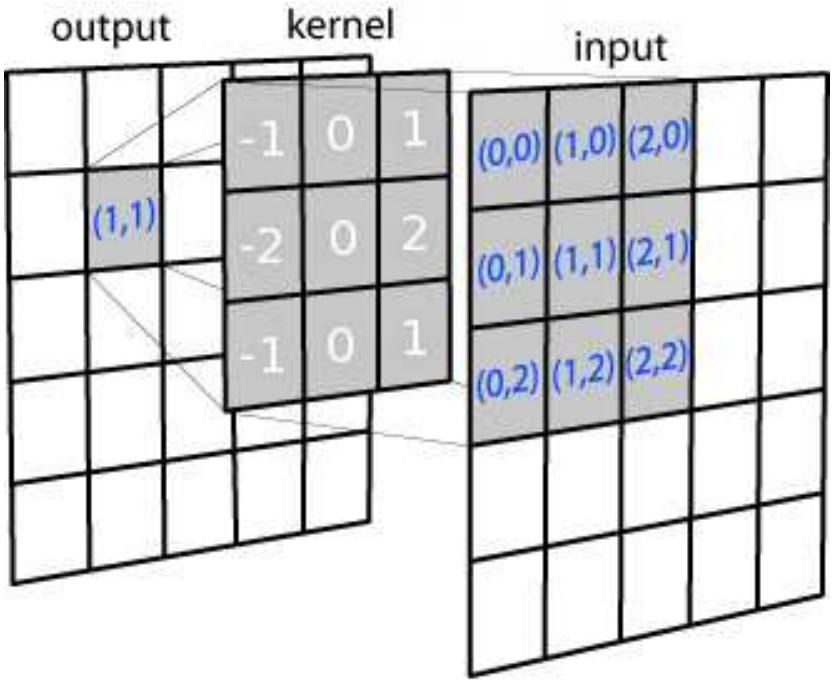
After you have a working cache implementation, experiment with how changing the cache parameters affects the performance of the edge-detector. `cache-sim` takes optional arguments that specify the cache parameters to make this easier. The most interesting statistics are the cache hit rate (how many cache reads or writes didn't have to go all the way to memory) and the number of memory cycles used (which corresponds directly to performance). You should be able to see up to an order of magnitude speedup from slowest cache settings to the fastest. Each time you run the cache simulator, statistics are printed detailing the performance of the cache that will help you understand how it's working.

Next, look at the implementation of the Sobel filter in `sobel.c`. Sobel filters are a simple edge-detection mechanism for images. First, the input image is converted to a grayscale matrix representation like:

```
BYTE image[height][width]
```

Each pixel's brightness is represented by one byte: 0 is completely black and 255 is completely white.

The filter iterates over each pixel in the image and produces a corresponding output pixel value. The output pixel is the sum of multiplying each of the neighboring pixels in the input image with the corresponding value in the filter kernel. For example, in this figure, the value of pixel (1, 1) in the output image will be –(0,0) – 2(0,1) –(0,2) + (2,0) + 2(2,1) + (2,2)



The final value of the output pixel is the geometric mean of the value produced by two slightly different kernels. The first kernel finds horizontal edges and the second kernel finds vertical edges.

Using the default cache parameters, this algorithm, as implemented, has a low cache hit rate. The way an algorithm is implemented can have a huge effect on performance because of cache effects. As part of the assignment, you need to optimize this implementation to better take advantage of the spatial and temporal locality of the algorithm.

A few test images have been included for you to play with, but you can use any jpeg you'd like for testing. Here's an example of the output produced by running `statue.jpg` through the filter. The skeleton code does compile, but will produce black images until your cache code is implemented (or you turn off the cache as described above).

Questions

Put the answers to these questions in comments at the top of `sobel.c`

1. Why does the cache behave much differently for dew.jpg (hint: look at the image dimensions)?

2. What happens to the hit rate when you change the block size? Why?

3. What changes did you make to the Sobel filter implementation?

What you should do:

1. Download the code skeleton from the website.
2. Complete the cache simulator implementation in `cache.c`
3. Run the Sobel filter program on the test images and observe the output.
4. Experiment with different images and cache parameters.
5. Optimize the `filter_sobel` function for a 1KB, 4-byte block, 2-way set associative cache.
6. Answer the three questions.
7. Put `cache.c` and `sobel.c` in a zip file for submission.
8. Submit your assignment via the Catalyst WebTools at:
   https://catalysttools.washington.edu/collectit/dropbox/rea2000/3707
   Please include your name at the top of `sobel.c` file in a comment.