Homework 3

CS Osvita

January 14, 2026

Due to Moore's law, it is now much faster to execute an arithmetic instruction than it is to access the main memory. Consequently, one of the best ways to speed up many applications is to consider how the CPU will cache your data.

This homework will require you to apply your understanding of CPU caches to a series of short programs. In each case, you must consider how the program's data currently resides in memory, form hypotheses about cache performance, actually **measure cache performance with profiling tools**, and then refactor the program to improve performance.

We strongly recommend using Linux; students who have previously used other OSes (especially Mac) have struggled with limited documentation or poorly maintained tools. If you don't have a Linux machine, here are a few options to consider:

- Recommended approach: use the instructions provided alongside the exercise files to get set up with Docker (this will work for *valgrind*/*cachegrind*, but not $perf$)
- Provision a cloud VM, such as Digital Ocean (this will also work for $perf$ as long as you get an instance with a dedicated CPU)

We hope you find this a fun and particularly practical exercise!

1 Loop order

In this problem, you will start playing with the effect of CPU caches by making a small change to dramatically improve the performance of a program that converts a given image from RGB to grayscale. It does so by looping over each pixel value and computing a single "luminance" value that replaces each of the red, green, and blue values. The purpose of this exercise is to develop your understanding by using profiling tools to probe the program. To do so, make an improvement, add comments to the program to isolate one option or the other, compile each version, and run them through cachegrind.

1. Which function takes longer to run, if any?
2. Do they execute the same number of instructions?

3. What do you notice about the cache utilization of each program?
4. How much better is one than the other? Does this match your expectations?

2 Matrix multiplication

Matrix multiplication is a fundamental operation in mathematics and machine learning. Given its frequent use, especially with large matrices, optimizing its performance is crucial to reduce computation time and improve efficiency. There are many ways to speed up matrix multiplication, including parallelism, clever algorithms, and, of course, cache utilization, which will be the focus of this exercise.

In matrix-multiply.c, you'll find a naive implementation of matrix multiplication and a function optimistically called fast_matrix_multiply for you to implement. To compile and test your code, run:

```
cc -Wall matrix-multiply.c benchmark.c && ./a.out 512
```

For simplicity, we use square matrices. The benchmark program will create two random matrices of the given size, multiply them against one another using both the naive baseline program and your implementation, verify that the results are the same, and measure the time taken to execute each.

If you run the code without changing anything, you should be able to verify that both functions run in approximately the same amount of time:

```
$ cc -Wall matrix-multiply.c benchmark.c && ./a.out 512
Naive: 1.187s
Fast: 1.174s
1.01x speedup
```

The 512 command line argument represents the size of the matrices to be multiplied, but you will probably want to change this when testing. Especially if you have a faster computer, you might need 1024 or 2048 to see meaningful speedups from your optimizations.

As you work towards a solution, you may wish to utilize both first principles of thinking as well as benchmarking and profiling. Doing this exercise in *C* should make it easier to use tools like cachegrind as you did previously.

As a stretch goal, you may also wish to run the program at various compiler optimization levels, and with different matrix sizes, to see how these factors affect performance. You may even enjoy plotting the results for various sizes to see what patterns emerge. Do you expect linear growth or something else?

3 Metrics

As a final optimization exercise, you will improve the performance of a typical Go/Python program written without considering CPU caches. Using what you know about CPU caches, you should be able to substantially improve its performance without using any fancy algorithms or low-level optimization techniques.

From the metrics directory of the included files, you can run:

```
go test -bench=.
goos: darwin
goarch: amd64
BenchmarkMetrics/Average_age-12          606        1879392 ns/op
BenchmarkMetrics/Average_payment-12      56        28236512 ns/op
BenchmarkMetrics/Payment_stddev-12       22        54153588 ns/op
PASS
```

... for a basic benchmark. Your task is to reduce the time taken per execution of the functions under test (ns/op).

You are welcome to change any aspects of the organization of the data in memory, including types, so long as the tests still pass. You are also welcome to change the code used to load the data from disk, and in fact, you may need to do so to reflect changes you make elsewhere. However, please do not precompute the answers directly in the *LoadData* function :)

You are also welcome to make minor changes to the benchmark code, so long as it doesn't affect the substance of the test. You should not need to alter the code used to generate the test data, but this is provided for your information and may be useful to know the ranges of the provided data.

4 Mysterious binary search

You're given two sorted arrays. Array A contains 2^{25} elements. Array B contains $2^{25} + 2^{10}$ elements. We want to run a binary search over both arrays multiple times. Intuitively, you might think: "Binary search on the smaller array should be faster — fewer steps, right?". But benchmarking reveals something surprising. Even though Array B is larger, binary search on it can be faster than on Array A. Why? Your task is to analyze what's happening under the hood. Write a brief explanation (2–3 paragraphs) of your findings after benchmarking on your machine.

Further Resources

For an alternative explanation of the memory hierarchy, see the blog post Why do CPUs have multiple cache levels? by Fabian Giesen. This should help you rationalize why we've settled (for now) on the configuration of CPU caches that you'll typically see.

For motivation and context on why this matters for programmers, watch this talk by Mike Acton.

For more on Cachegrind, see the cachegrind manual.

What Every Programmer Should Know About Memory may be ambitiously named given its depth, but certainly, you should aspire to know much of what's contained. If you are excited about making the most, as a programmer, of your CPU caches, then this will be a great starting point. P:H “covers” the memory hierarchy in sections 5.1-5.4, but it is designed to be a kind of appetizer for the chapters on memory hierarchy design in Computer Architecture: A Quantitative Approach (the more advanced “H:P” version of P:H). Berkeley CS 61C covers caches over three classes: 1 2 and 3.

Useful set of commands

```
cc -g loop-order.c
```

This command compiles the loop-order.c file using the GNU C compiler (cc) with the -g flag. The -g flag includes debugging information in the generated executable, which is useful when profiling or debugging with tools like gdb or valgrind.

```
valgrind --tool=cachegrind --cache-sim=yes ./a.out
```

This command runs the executable *a.out* (compiled from the previous step) under valgrind using the cachegrind tool. The –cache-sim=yes option enables a more detailed simulation of the cache. It provides information on cache misses and hits during program execution.

```
cg_annotate cachegrind.out.<pid>
```

This command is used to interpret and display the cachegrind output. The file cachegrind.out.<pid> contains the profiling data, where <pid> represents the process ID of the executed program. The cg_annotate tool formats and displays the output in a human-readable format, showing cache usage statistics per line of code.

```
cc -g loop-order.c -o option_one
perf stat -e L1-cache-load-misses,L1-dcache-loads ./option_one
```

This command uses the $perf$ tool to run the option_one executable and gather performance statistics, specifically counting events like L1-cache-load-misses (the number of L1 cache misses during loads) and L1-dcache-loads (the total number of L1 data cache loads). The stat subcommand provides detailed performance event statistics related to cache usage, which helps you measure how efficiently the CPU’s cache is used.

```
cc -O2 -g matrix-multiply.c benchmark.c
./a.out 1000
```

The -O2 flag enables optimizations during compilation to improve performance without increasing compilation time significantly.