Homework 5

Program Optimization & x86-64 Assembly

CS Osvita

One of the most enjoyable aspects of understanding computer architecture is using a machine's characteristics to your advantage to write faster programs. While we generally expect our compilers to do this for us, there is a limit to what a compiler can understand, so we still have a responsibility when writing high-level code to understand the performance implications of the machine code that will be generated.

This homework will help you explore some techniques for program optimization, both directly in assembly and as a "nudge" to the compiler. You will also increase the resolution of our computer model, incorporating ideas like **micro-operations**, **branch prediction**, and **out-of-order execution**.

> **Recommended reading:** CS:APP Chapter 5 as your first reference point to clarify confusing terms or concepts throughout this assignment.

1 Pagecount

The purpose of this exercise is to demonstrate how a programmer's basic understanding of their own program can give them a significant advantage over an optimizing compiler. It also provides an opportunity to use a disassembler and basic measurement tools.

We've provided a simple program in **pagecount.c**. Modern operating systems divide memory notionally into units called **pages**. If each page is 4 KB and we have 4 GB of memory, we have approximately 1 million pages. The `pagecount` function performs this calculation by dividing available memory (in bytes) by the page size (also in bytes).

Part A — Analysis

Read the code for this function, then consider:

- Which instructions would you expect your compiler to generate for this function?
- What does it *actually* generate? (Use a disassembler to check.)
- If you change the optimization level, is the function substantially different?
- Use godbolt.org to explore a few different compilers. Do any of them generate substantially different instructions?
- Using Agner Fog's instruction tables or CS:APP §5.7, can you determine which of the generated instructions may be slow?

Part B — Optimization

Now let's improve performance:

- Noting that a page size is always a power of 2, and that the size of memory will always be cleanly divisible by the page size, what optimization could we employ? You are welcome to change the function signature and test runner code.
- How much of an improvement would you expect to see?
- Go ahead and make the improvement, and measure the speedup. Did it match your expectations?
- Consider what is stopping the compiler from making the same optimization you did.

2 Vector Dot Product

In the **vec/** directory, we've provided a vector implementation and a `dotproduct` function that computes the inner product of two vectors. If you're unfamiliar with this operation, it is computed as the sum of the products of each corresponding element pair:

```
dotproduct([1, 2], [3, 4]) = (1×3) + (2×4) = 3 + 8 = 11
```

Run the tests by typing `make` in the **vec/** directory and optimize the performance of the `dotproduct` function using whichever techniques you like.

1. Many techniques discussed in CS:APP Chapter 5 can be applied directly (loop unrolling, reducing data dependencies, etc.).
2. If possible, form an expectation of the **best possible performance** of the function (assuming a single thread of execution and no SIMD operations) on your machine, then see how close you can get.
3. *(Stretch goal)* Research **SIMD instructions** and see if you can improve your function even further. Search for "AVX intrinsics" as a starting point.

To run inside Docker:

```
# Note: you may need to run the Docker container in privileged mode.
docker run --rm -it --privileged -v '.\vec:/root' csosvita bash
```

3 Colors

In this problem, you will refactor code that converts a standard **24-bit per pixel bitmap** file into one that uses **8 bits per pixel** — a kind of image retro-izer that works by grouping color ranges into buckets.

The mapping scheme used is 3-3-2 RGB, used on several computers in the mid-1980s. Red and green values each have 8 possibilities (3 bits), while blue has 4 possibilities (2 bits).

The provided implementation performs bucketing using a series of conditional statements. This branch-heavy code is not ideal and may lead to poor performance on specific images.

Your task: Rewrite the `quantize` function to use **fewer (or no) branches**, which should improve both performance and readability.

If you have access to *perf* or a similar tool, measure the **branch misprediction rate** before and after your refactor. A benchmark is provided that calls the `quantize` function directly (but requires you to install Google Benchmark), or you can measure overall execution time with the provided bitmap files.

4 Perf: Big-O vs. Hardware

Time complexity indicates how an algorithm's work scales with input size, but real performance also depends on how effectively the code utilizes the hardware. A better time complexity does *not* guarantee better performance — hardware-friendly implementation matters too.

Use Linux *perf* to compare two Euclidean GCD implementations — **subtraction-based** and **modulo-based** — and show cases where subtraction can be faster despite worse asymptotic complexity.

GCD: Subtraction-Based

```
long gcd_sub(long a, long b) {
    while (a != b) {
        if (a > b) { a -= b; }
        else       { b -= a; }
    }
    return a;
}
```

GCD: Modulo-Based

```
long gcd_mod(long a, long b) {
    while (b != 0) {
        long t = b;
        b = a % b;
        a = t;
    }
    return a;
}
```

Compile and Measure

Compile with optimizations enabled, then use `perf stat` to measure runtime behavior and hardware-level metrics:

```
gcc -O3 -march=native -o gcd_sub gcd_subtraction.c
gcc -O3 -march=native -o gcd_mod gcd_modulo.c

# Close inputs - subtraction may win here
perf stat ./gcd_sub 130000 13
perf stat ./gcd_mod 130000 13

# Far-apart inputs - modulo wins decisively
perf stat ./gcd_sub 1000000000 9223372036854775503
perf stat ./gcd_mod 1000000000 9223372036854775503
```

Compute the GCD of all combinations of integers in the range [1, 100,000) at increments of 5. What key takeaways can you make from the `perf` output?

> Focus on hardware-level metrics such as instruction count, cache misses, branch mispredictions, and IPC (instructions per cycle) — not just wall-clock time.

5 Pangram in Assembly

The following exercises will guide you through writing, assembling, and running basic **x86-64 assembly**. We suggest CS:APP §§3.1–3.6 for background. You may also find this x86 tutorial helpful.

Setup

Solve this problem on Exercism, or install the NASM assembler locally.

macOS:

```
$ xcode-select --install
$ brew install nasm
```

Linux (Ubuntu/Debian):

```
$ sudo apt-get install gcc make nasm
```

Confirm It Works

Assemble and run one of the provided "Hello World" programs:

macOS:

```
$ nasm -fmacho64 hello_mac.asm && ld -lSystem hello_mac.o && ./a.out
```

> On macOS 11 and above, you may also need the following `ld` flag (see here): `-L/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/lib`

Linux:

```
$ nasm -felf64 hello_linux.asm && ld hello_linux.o && ./a.out
```

Once it works, try modifying the program slightly to see how it works.

Task: Pangram Check

A phrase is a **pangram** if it contains every letter of the alphabet at least once — for example, *"the quick brown fox jumped over the lazy dog"*. Write a program to determine if a given phrase is a pangram. Your program should be **case-insensitive** and should **ignore punctuation**.

This exercise encourages you to think about simple data structures at a low level. How will you keep track of which letters have been encountered? In a high-level language you might reach for a hashmap; in assembly you may discover a more elegant approach.

The equivalent C logic looks like this:

```c
#define MASK 0x07fffffe
bool ispangram(char *s) {
    uint32_t map = 0;
    char c;
    while ((c = *s++) != '\0') {
        if (c < '@') continue;  // ignore first 64 chars in ASCII table
        map |= 1 << (c & 0x1f);
    }
    return (map & MASK) == MASK;
}
```

Stretch Goals

- **Minimize instructions** — use as few instructions as possible with minimal (or zero) extra memory.
- **Minimize branching** — try to handle case insensitivity and punctuation without branch instructions (you will still need a branch for the loop).
- Consider the `bts` instruction. Also consider that we care about exactly 26 characters — and 26 fits in 64 bits.

Further Resources

- Agner Fog's Optimization Manuals — especially *"Optimizing subroutines in assembly language"*
- x86-64 exercises on Exercism — excellent for further practice
- SHENZHEN I/O and Human Resource Machine — games that build assembly-style intuition
- Zachtronics: Ten Years of Terrible Games — a fun companion talk