Computer Systems for AI-programmers
baiCSAI3, Spring 2011
Lab Assignment: Code Optimization
Assigned: Week 8, Due: Week 10
Tuesday March 8, 10:00

Arnoud Visser (`A.Visser@uva.nl`) is the lead person for this assignment.
Eva Greiner will assist during class hours.

1 Introduction

This assignment deals with optimizing a naive piece of code. Image processing and graphics offers many examples of functions that can benefit from optimization. In this lab, we will consider a single graphics operation: `line`, which draws a line with a certain angle over an image.

For this lab, we will consider an image to be represented as a two-dimensional matrix M, where $M_{i,j}$ denotes the value of (i, j)th pixel of M. Pixel values are triples of red, green, and blue (RGB) values. We will only consider square images. Let N denote the number of rows (or columns) of an image. Rows and columns are numbered, in C-style, from 0 to $N - 1$.

2 Logistics

You may work in a group of up to two people in solving the problems for this assignment. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be mailed to your student.uva.nl adress.

3 Hand Out Instructions

Start by copying `perflab-handout.tgz` to a protected directory in which you plan to do your work. Then give the command: `tar zxvf perflab-handout.tgz`. This will cause a number of files to be unpacked into a directory. The primary C-file you will be modifying is `line-versions.c`. In this file you can define multiple versions of the function `line()`. To test the performance of a version, you register

this version to a benchmark-list in `kernel.c`, the second file that you have to modify. To be able to register your versions of the `line()` function, the name and description of each version has to be known, which you can define in `line-versions.h` The main routine can be found in `driver.c`, which is a driver program that allows you to evaluate the performance of your solutions. Use the command `make driver` to generate the driver code and run it with the command `./driver`.

Looking at the file `kernel.c` you'll notice a C structure `team` into which you should insert the requested identifying information about the one or two individuals comprising your programming team. **Do this right away so you don't forget.**

4 Implementation Overview

Data Structures

The core data structure deals with image representation. A `pixel` is a struct as shown below:

```
typedef struct {
   unsigned short red;   /* R value */
   unsigned short green; /* G value */
   unsigned short blue;  /* B value */
} pixel;
```

As can be seen, RGB values have 16-bit representations ("16-bit color"). An image `I` is represented as a one-dimensional array of `pixel`s, where the (i, j)th pixel is `I[RIDX(i,j,n)]`. Here `n` is the dimension of the image matrix, and `RIDX` is a macro defined as follows:

```
#define RIDX(i,j,n) ((i)*(n)+(j))
```

See the file `defs.h` for this code.

Line

The following C function draws a line under an angle of $30°$ in destination image `dst`. The value used for this line is the pixel with the maximum intensity in the `src` image. `dim` is the dimension of the image.

```
void naive_line(int dim, pixel *src, pixel *dst) {
    int x0 = 0;
    int y0 = floor (dim / 3); /* left endpoint */
    int x1 = dim - 1;
    int y1 = ceil (dim - 1 - dim / 3); /* right endpoint */

    double dy = y1 - y0;
    double dx = x1 - x0;
    double slope = dy / dx;

    double y = y0;
    int x = x0;
```

```
    for (; x <= x1; x++) {
        dst[RIDX(x,(int )rint(y), dim)] = *maximum(dim, src);
        y += slope;
    }
}
```

Your task is to rewrite this code to make it run as fast as possible using techniques like code motion, loop unrolling and blocking.

See the file `line-versions.c` for this code.

Performance measures

Our main performance measure is *CPE* or *Cycles per Element*. If a function takes C cycles to run for an image of size $N \times N$, the CPE value is C/N^2. Table 1 summarizes the performance of the naive implementations shown above and compares it against an optimized implementation. Performance is shown for for 5 different values of N. All measurements were made on a Xeon server.

The ratio (speedup) of the optimized implementation over a Baseline one will constitute a *score* of your implementation. To summarize the overall effect over different values of N, we will compute the *geometric mean* of the results for these 5 values. That is, if the measured speedups for $N = \{32, 64, 128, 256, 512\}$ are L_{32}, L_{64}, L_{128}, L_{256}, and L_{512} then we compute the overall performance as

$$L \quad = \quad \sqrt[5]{L_{32} \times L_{64} \times L_{128} \times L_{256} \times L_{512}}$$

Assumptions

To make life easier, you can assume that N is a multiple of 32. Your code must run correctly for all such values of N, but we will measure its performance only for the 5 values shown in Table 1.

Test case	1	2	3	4	5	
Method N	128	256	512	1024	1536	Geom. Mean
Optimized `line` (CPE)	5.1	8.1	10.9	13.2	17.5	
Baseline `line` (CPE)	1415.0	3540.0	8276.9	17895.0	33234	
Speedup (naive/opt)	279.4	435.6	761.6	1356.7	1894.2	750.5

Table 1: CPE and Ratio for Optimized vs Baseline Implementations

5 Infrastructure

We have provided support code to help you test the correctness of your implementations and measure their performance. This section describes how to use this infrastructure. The exact details of each part of the assignment is described in the following section.

Note: The only source files you will be modifying are `kernel.c, line-versions.c, line-versions.h`.

Versioning

You will be writing many versions of the `line` routine. To help you compare the performance of all the different versions you've written, we provide a way of "registering" functions.

For example, the file `kernel.c` that we have provided you contains the following function:

```
void register_line_functions() {
   add_line_function(&line, LINE_DESCR);
}
```

This function contains one or more calls to `add_line_function`. In the above example, `add_line_function` registers the function `line` along with a string `LINE_DESCR` which is an ASCII description of what the function does. See the file `line-versions.h` to see how to create the string descriptions. This string can be at most 256 characters long.

Driver

The source code you will write will be linked with object code that we supply into a `driver` binary. To create this binary, you will need to execute the command

```
make driver
```

You will need to re-make driver each time you change the code. To test your implementations, you can then run the command:

```
./driver
```

The `driver` can be run in four different modes:

- *Default mode*, in which all versions of your implementation are run.
- *Autograder mode*, in which only the main `line()` function are run. This is the mode we will run in when we use the driver to grade your handin.
- *File mode*, in which only versions that are mentioned in an input file are run.
- *Dump mode*, in which a one-line description of each version is dumped to a text file. You can then edit this text file to keep only those versions that you'd like to test using the *file mode*. You can specify whether to quit after dumping the file or if your implementations are to be run.

If run without any arguments, `driver` will run all of your versions (*default mode*). The naive version provided to you is very slow, and it will take several minutes to test. You will like to test only versions of your own very soon. When you made progress, copy and rename the function, to be sure that overwrite good working versions. Document your progress in a `labbook.txt`. Other modes and options can be specified by command-line arguments to `driver`, as listed below:

`-g` : Run only the final `line()` function (*autograder mode*).

`-f <funcfile>` : Execute only those versions specified in `<funcfile>` (*file mode*).

`-d <dumpfile>` : Dump the names of all versions to a dump file called `<dumpfile>`, *one line* to a version (*dump mode*).

`-q` : Quit after dumping version names to a dump file. To be used in tandem with `-d`. For example, to quit immediately after printing the dump file, type `./driver -qd dumpfile`.

`-h` : Print the command line usage.

Team Information

Important: Before you start, you should fill in the struct in `kernels.c` with information about your team (group name, team member names and email addresses). This information is just like the one for the Data Lab.

6 Assignment Details

Some advice. Focus on optimizing the inner loop (the code that gets repeatedly executed in a loop) using the optimization tricks covered in class. Look first at the algorithm, then at the details. Look at the assembly code for the `line` details.

Coding Rules

You may write any code you want, as long as it satisfies the following:

- It must be in ANSI C. You may not use any embedded assembly language statements.
- It must not interfere with the time measurement mechanism. You will also be penalized if your code prints any extraneous information.

You can only modify code in `kernel.c, line-versions.c, line-versions.h`. You are allowed to define macros, additional global variables, and other procedures in these files.

Evaluation

The score for each will be based on the following:

- Correctness: You will get NO CREDIT for buggy code that causes the driver to complain! This includes code that correctly operates on the test sizes, but incorrectly on image matrices of other sizes. As mentioned earlier, you may assume that the image dimension is a multiple of 32.
- CPE: You will get partial credit for your implementations of `line` if they are correct and achieve mean speedups above 500x the baseline.
- Innovation: You will get extra credit for your achievements as documented in the `labbook.txt`.

7 Hand In Instructions

When you have completed the lab, you will hand in one file, `teamname-version-perflab.tgz`, that contains your solution. Here is how to hand in your solution:

- Make sure you have included your identifying information in the team struct in `kernel.c`.

- Make sure that the `line()` function correspond to your fastest implementation, as this is the only function that will be tested when we use the driver to grade your assignement.
- Remove any extraneous print statements.
- Decribe your achievements in the file `labbook.txt`
- To handin your `tar` file, type:

  ```
  make handin TEAM=teamname
  ```

  where `teamname` is the team name defined in your team struct.
- After the handin, if you discover a mistake and want to submit a revised copy, type

  ```
  make handin TEAM=teamname VERSION=2
  ```

  Keep incrementing the version number with each submission.

Good luck!