CS 202: Spring 2018
Assignment #4: Code Optimization
Assigned: Tuesday, March 20th, Due: Mon, April 2nd, 2018 at 11:59PM

1 Introduction

This assignment deals with optimizing memory intensive code. Image processing offers many examples of functions that can benefit from optimization. In this lab, we will consider two image processing operations: `flip`, which flip an image both horizontally and vertically , and `sharpen`, which "sharpens" 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 $N - 1$.

Given this representation, the `flip` operation can be implemented quite simply as the following matrix operations:

- *Exchange rows*: Row i is exchanged with row $N - 1 - i$.
- *Exchange columns*: Column i is exchanged with column $N - 1 - i$.

The `sharpen` operation is implemented by replacing every pixel value with a transformation based on all the pixels around it (in a maximum of 3×3 window centered at that pixel). The values of pixels `M2[1][1]` and `M2[N-1][N-1]` are given below:

$$\mathtt{M2[1][1]} = (-1 * \sum_{\mathtt{i}=0}^{2}\sum_{\mathtt{j}=0}^{2} \mathtt{M1[i][j]}) + (10 * \mathtt{M2[1][1]})$$

$$\mathtt{M2[N-1][N-1]} = (-1 * \sum_{\mathtt{i}=N-2}^{N-1}\sum_{\mathtt{j}=N-2}^{N-1} \mathtt{M1[i][j]}) + (5 * \mathtt{M2[N-1][N-1]})$$

Specifically, where X is the number of neighboring pixels: the resulting pixel is

$$result = ((X + 1) * pixel) - \sum_{\mathtt{i}=0}^{X} neighbor_i$$

2 Logistics

Unlike prior assignments, there is no scoreboard for this assignment. The only “hand-in” will be electronic. Any clarifications and revisions to the assignment will be communicated in class, via email or via Piazza/CourseSite.

Make sure you are using gcc-7.1.0. You can enter in the command `module load gcc-7.1.0` into Sunlab to use gcc-7.1.0 for the duration of that session.

3 Hand Out Instructions

The handout file `perflab-handout.tgz` can be accessed via the sunlab machines in:

```
~jloew/CSE202/perflab-handout.tgz
```

As a reminder, to copy the file to the current directory, you can use:

```
cp ~jloew/CSE202/perflab-handout.tgz ./
```

Start by copying `perflab-handout.tgz` to a protected directory in which you plan to do your work. Then give the command: `tar -xzvf perflab-handout.tgz`. This will cause a number of files to be unpacked into the directory. The only file you will be modifying and handing in is `kernels.c`. The `driver.c` program 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 `kernels.c` you’ll notice a C structure `student_t` into which you should insert the requested identifying information about yourself. **Do this right away so you don’t forget.**

4 Implementation Overview

Data Structures

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

```c
struct pixel_t
{
  unsigned short red : 8;
  unsigned short green : 8;
  unsigned short blue : 8;
  unsigned short unused : 8;
}
```

As can be seen, RGB values have 32-bit representations (“32-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:

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

Flip

The following C function computes the result of flipping the source image, `src`, and stores the result in destination image, `dst`. `dim` is the dimension of the image.

```c
void naive_flip(int dim, struct pixel_t *src, struct pixel_t *dst)
{
  for(int i = 0; i < dim; i++)
  {
    for(int j = 0; j < dim; j++)
    {
      dst[RIDX(dim-1-i, dim-1-j, dim)].red = src[RIDX(i, j, dim)].red;
      dst[RIDX(dim-1-i, dim-1-j, dim)].green = src[RIDX(i, j, dim)].green;
      dst[RIDX(dim-1-i, dim-1-j, dim)].blue = src[RIDX(i, j, dim)].blue;
      dst[RIDX(dim-1-i, dim-1-j, dim)].unused = src[RIDX(i, j, dim)].unused;
    }
  }
}
```

The above code scans the pixels of the source image matrix, copying to the *flipped* position of the destination image matrix. 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 `kernels.c` for this code.

Sharpen

The sharpen function takes as input a source image `src` and returns the sharpened result in the destination image `dst`. Here is part of an implementation:

```c
void naive_sharpen(int dim, struct pixel_t *src, struct pixel_t *dst)
{
  for(int i = 0; i < dim; i++)
  {
    for(int j = 0; j < dim; j++)
    {
      double red = 0.0, green = 0.0, blue = 0.0;

      int neighbors = 0;
      for(int fX = max(i-1, 0); fX <= min(i+1, dim-1); fX++)
      {
        for (int fY = max(j-1, 0); fY <= min(j+1, dim-1); fY++)
        {
          red -= src[RIDX(fX, fY, dim)].red;
          green -= src[RIDX(fX, fY, dim)].green;
          blue -= src[RIDX(fX, fY, dim)].blue;
          neighbors++;
        }
      }
      if(neighbors == 4)
      {
```

```
      red += 5 * src[RIDX(i,j,dim)].red;
      green += 5 * src[RIDX(i,j,dim)].green;
      blue += 5 * src[RIDX(i,j,dim)].blue;
    }
    else if(neighbors == 6)
    {
      red += 7 * src[RIDX(i,j,dim)].red;
      green += 7 * src[RIDX(i,j,dim)].green;
      blue += 7 * src[RIDX(i,j,dim)].blue;
    }
    else if(neighbors == 9)
    {
      red += 10 * src[RIDX(i,j,dim)].red;
      green += 10 * src[RIDX(i,j,dim)].green;
      blue += 10 * src[RIDX(i,j,dim)].blue;
    }
    else
    {
      //Invalid neighbor count
      fprintf(stderr, "Invalid neighbor count of %d\n", neighbors);
    }

    int r = min(max(0, (int)red), 255);
    int g = min(max(0, (int)green), 255);
    int b = min(max(0, (int)blue), 255);

    dst[RIDX(i, j, dim)].red = r;
    dst[RIDX(i, j, dim)].green = g;
    dst[RIDX(i, j, dim)].blue = b;
    dst[RIDX(i, j, dim)].unused = 0;
  }
 }
}
```

Your task is to optimize `sharpen` to run as fast as possible.

This code is in the file `kernels.c`.

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 8 different values of N.

The ratios (speedups) of the optimized implementation over the naive 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 8 values. That is, if the measured speedups for $N = \{64, 128, 256, 320, 512, 1024, 2048, 8192\}$ are R_{64}, R_{128}, R_{256}, R_{320}, R_{512}, R_{1024}, R_{2048} and R_{8192} then we compute the overall performance as

$$R = \sqrt[8]{R_{64} \times R_{128} \times R_{256} \times R_{320} \times R_{512} \times R_{1024} \times R_{2048} \times R_{8192}}$$

Test case	1	2	3	4	5	6	7	8	
Method	64	128	256	320	512	1024	2048	8192	Geom. Mean
Naive `flip` (CPE)	1.9	1.7	1.6	1.5	1.4	1.4	1.6	1.7	
Optimized `flip` (CPE)	0.4	0.4	0.4	0.4	0.4	0.4	1.3	1.3	
Speedup (naive/opt)	5.0	4.6	4.3	3.5	3.2	3.3	1.3	1.3	2.7
Method	64	128	256	320	512	1024	2048	8192	Geom. Mean
Naive `sharpen` (CPE)	64.1	64.6	64.4	64.6	64.6	64.6	64.6	67.9	
Optimized `sharpen` (CPE)	34.3	36.9	34.7	37.0	37.2	37.2	37.8	66.6	
Speedup (naive/opt)	1.9	1.7	1.9	1.7	1.7	1.7	1.7	1.0	1.6

Table 1: CPEs and Ratios for Optimized vs. Naive Implementations (Numbers may not match)

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 8 values shown in Table 1.

Important Note

All performance measurements were made on the SunLAB machines (meaning only these should be used for your testing). Specifically, these were done on the machine *vesta*.

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 file you will be modifying is `kernels.c`.

Versioning

You will be writing many versions of the `flip` and `sharpen` routines. 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 `kernels.c` that we have provided you contains the following function:

```c
void register_flip_functions()
{
    add_flip_function(&flip, flip_descr);
}
```

This function contains one or more calls to `add_flip_function`. In the above example, `add_flip_function` registers the function `flip` along with a string `flip_descr` which is an ASCII description of what the function does. See the file `kernels.c` to see how to create the string descriptions. This string can be at most 256 characters long.

A similar function for your sharpen kernels is provided in the file `kernels.c`.

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 in `kernels.c`. 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 `flip()` and `sharpen()` functions 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*). Other modes and options can be specified by command-line arguments to `driver`, as listed below:

`-g` : Run only `flip()` and `sharpen()` functions (*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.

Student Information

Important: Before you start, you should fill in the struct in `kernels.c` with information about yourself. This information is just like the one for the Data Lab.

6 Assignment Details

Optimizing Flip (50 points)

In this part, you will optimize `flip` to achieve as low a CPE as possible. You should compile `driver` and then run it with the appropriate arguments to test your implementations.

For example, running driver with the supplied naive version (for `flip`) generates the output shown below (numbers may not match your outputs):

```
> ./driver
Student: Harry Q. Bovik
Email: bovik@nowhere.edu

Flip Version = naive_flip: Naive baseline implementation:
Dim             64  96  128 256 512 1024 2048 4096 8192 Mean
Your CPEs       1.9 1.7 1.6 1.5 1.4 1.4  1.6  1.7  1.7
Baseline CPEs   1.9 1.7 1.6 1.5 1.4 1.4  1.7  1.7  1.7
Speedup         1.0 1.0 1.0 1.0 1.0 1.0  1.1  1.0  1.0  1.0
```

Optimizing Sharpen (50 points)

In this part, you will optimize `sharpen` to achieve as low a CPE as possible.

For example, running driver with the supplied naive version (for `sharpen`) generates the output shown below (numbers may not match your outputs):

```
> ./driver

Sharpen: Version = naive_sharpen: Naive baseline implementation:
Dim             32   64   96   128  256  512  1024 2048 4096 Mean
Your CPEs       83.3 83.8 83.9 84.1 84.0 84.0 84.1 87.6 87.7
Baseline CPEs   83.2 83.7 83.9 84.0 84.0 84.1 84.1 87.6 87.7
Speedup         1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

Some advice. Look at the assembly code generated for the `flip` and `sharpen`. Focus on optimizing the inner loop (the code that gets repeatedly executed in a loop) using the optimization tricks covered in class. The `sharpen` is more compute-intensive and less memory-sensitive than the `flip` function, so the optimizations are of somewhat different flavors.

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 may not include any additional libraries.

- You may not add any global variables.

You can only modify code in `kernels.c`. You are allowed to define macros and other procedures in these files.

Evaluation

Your solutions for `flip` and `sharpen` will each count for 50% of your grade. 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 full credit for your implementations of `flip` and `sharpen` if they are **correct** and achieve mean speedups above thresholds 3.2 and 2.1 respectively. You will get partial credit for a correct implementation that does better than the supplied naive one.

7 Hand In Instructions

When you have completed the lab, you will hand in one file, `kernels.c`, that contains your solution. Here is how to hand in your solution:

- Make sure you have included your identifying information in the student_t struct in `kernels.c`.
- Make sure that the `flip` and `sharpen` functions correspond to your fastest implemnentations, as these are the only functions that will be tested when we use the driver to grade your assignement.
- Remove any extraneous print statements. This includes output to standard error as it will slow down your results.
- To handin your `kernels.c` file, type:

  ```
  make submit
  ```

- After the handin, if you discover a mistake and want to submit a revised copy, type

  ```
  make submit
  ```

8 Academic Integrity

I know you can find solutions to these problems online. Don't. Learning to optimize your code can be incredibly rewarding and fun, and doing so will help you to understand how caches work. If I have any reason to suspect that you copied a solution from the internet, you will receive zero points.