CSCI 2400, Summer 2018
Performance Lab
Due: Monday, July 16th, 11:55 PM MDT

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 kinds of general image processing operations: `flip`, which contains several image operations e.g. flip with vertical, or horizontal, or both axes; and `convolve`, which is a general concept that contains several image operations such as smoothing, blurring, sharpening, embossing, edge detection for images.

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, we have seven available `flip` operations(you only need to do one flip operation for this lab which is determined by our program) for this lab as follows:

- *Transpose the image*: for each (i, j) pair, $M_{i,j}$ and $M_{j,i}$ are interchanged.
- *Flip with vertical axis*: for each (i, j) pair, $M_{i,j}$ and $M_{i,N-1-j}$ are interchanged.
- *Flip with horizontal axis*: for each (i, j) pair, $M_{i,j}$ and $M_{N-1-i,j}$ are interchanged.
- *Flip with both axes*: for each (i, j) pair, $M_{i,j}$ and $M_{N-1-i,N-1-j}$ are interchanged.
- *Reflect with both axes*: for each (i, j) pair, $M_{i,j}$ and $M_{N-1-j,N-1-i}$ are interchanged.
- *Rotate image by* 90° *clockwise*: for each (i, j) pair, $M_{i,j}$ is moved to $M_{j,N-1-i}$
- *Rotate image by* 90° *counterclockwise*: for each (i, j) pair, $M_{i,j}$ is moved to $M_{N-1-j,\,i}$.

Rotation image by 90° can be implemented quite simply as the combination of "transpose" and "flip with horizontal axis". For example, the rotation by 90° counterclockwise is illustrated in Figure 1.

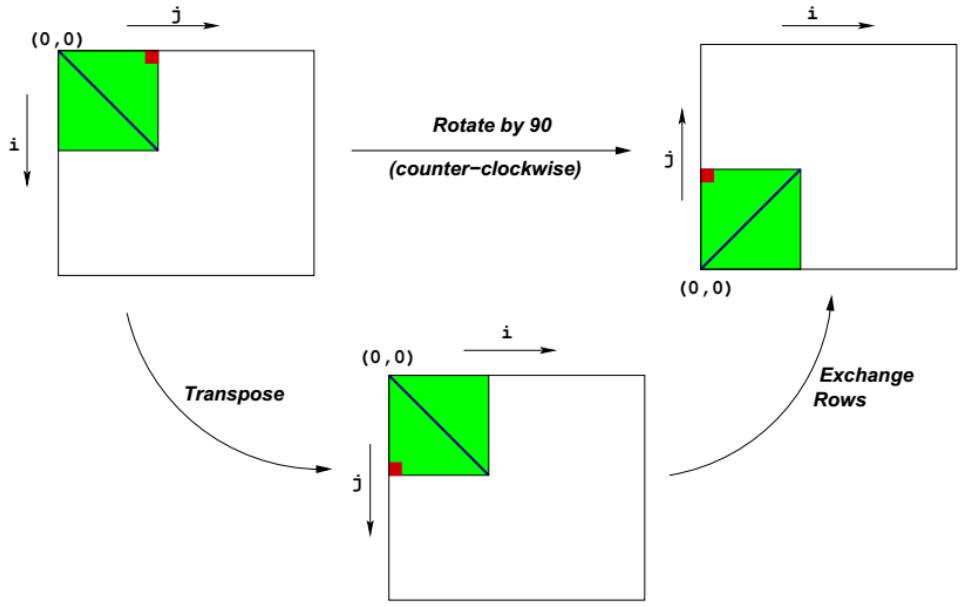

Figure 1: Rotation of an image by 90° counterclockwise

As for `convolve` operation, we have five candidate operations in this category as follows (the reference link just gives an intuitive description of the operator.)

- ***Sharpen kernel:*** https://photography.tutsplus.com/tutorials/what-is-image-sharpening--cms-26627
- ***Guassian blur kernel***: https://en.wikipedia.org/wiki/Gaussian_blur
- ***Sobel edge detection kernel***: https://en.wikipedia.org/wiki/Sobel_operator
- ***Emboss kernel***: https://en.wikipedia.org/wiki/Image_embossing
- ***Smooth kernel:*** see Figure 2 and equation (1)

Take the last one as an example, the `smoothing` operation is implemented by replacing every pixel value with the weighted average of all the pixels around it (in a maximum of 3×3 window centered at that pixel). Consider Figure 2. The values of pixels `M2[1][1]` and `M2[N-1][N-1]` are given below.

:

$$\texttt{M2}[1][1] = \frac{\sum_{i=0}^{2}\sum_{j=0}^{2} w_{ij}\texttt{M1}[i][j]}{\sum w_{ij}} = \frac{1}{35}\left(\underbrace{\begin{bmatrix}1 & 3 & 7\\ 4 & 1 & 3\\ 3 & 5 & 8\end{bmatrix}}_{Convolution\ Matrix} * \underbrace{\begin{bmatrix}\texttt{M1}[1][1] & \texttt{M1}[1][2] & \texttt{M1}[1][3]\\ \texttt{M1}[2][1] & \texttt{M1}[2][2] & \texttt{M1}[2][3]\\ \texttt{M1}[3][1] & \texttt{M1}[3][2] & \texttt{M1}[3][3]\end{bmatrix}}_{Image\ fraction}\right)[2,2]$$

$$\texttt{M2}[\texttt{N}-1][\texttt{N}-1] = \frac{\sum_{i=N-2}^{N-1}\sum_{j=N-2}^{N-1}\texttt{M1}[i][j]}{4}, \tag{1}$$

where $*$ means convolution operator: $(A * B)[i,j] = \sum_{m=0}^{N-1}\sum_{n=0}^{N-1} A[m,n]B[i-m, j-n]$.

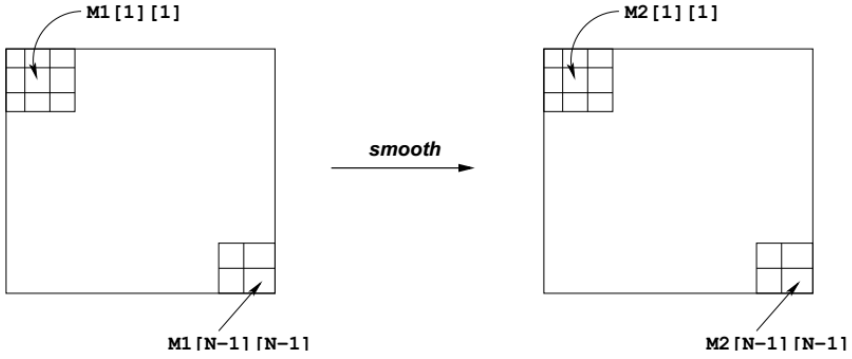

Figure 2: Smoothing an image

Although the different operations do different things, all the convolve operation can be achieved by doing

$$(ConvolutionMatrix) * (ImageFraction).$$

So that means different convolution matrix will give us different operator. And you will be given one convolution matrix by the program when you run your driver.

2 Hand Out Instructions

Start by copying `perflab-handout.tar` to the directory in which you plan to do your work. Then give the command: `tar xvf perflab-handout.tar`. 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` **which will show you your flip and convolution matrix.**

Looking at the file `kernels.c` you'll notice a C structure `team` into which you should insert the requested identifying information. **Do this right away so you don't forget, this is required !!!**

3 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 stored 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.

You should think of `I[RIDX(i,j,n)]` as equivalent to `I[i][j]` for most purposes – the reason `RIDX` is used at all is because it allows run-time changes of the array size, which is needed for the testing/grading code.

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. Note that your RIDX_F function is given by the program and is different from that of other students. As you optimize further, you may find yourself wanting to replace calls to RIDX_F with something more specific, customized, or optimized. This is fine, as long as the output is still correct.

```
void naive_flip(int dim, pixel *src, pixel *dst)
{
    int i, j;
    for (i = 0; i < dim; i++){
        for (j = 0; j < dim; j++){
            dst[RIDX_F(i, j, dim)].red   = src[RIDX(i, j, dim)].red;
            dst[RIDX_F(i, j, dim)].green = src[RIDX(i, j, dim)].green;
            dst[RIDX_F(i, j, dim)].blue  = src[RIDX(i, j, dim)].blue;
        }
    }
}
```

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.

Convolve

The smoothing function takes as input a source image `src` and returns the smoothed result in the destination image `dst`. Here is the native implementation. Note that your kernel matrix(convolution matrix) is given by the program and is different from that of other students. Again, as you optimize, you're welcome to store and/or reference the values in your kernel matrix however you like, as long as the output remains correct.

```
void naive_convolve(int dim, pixel *src, pixel *dst)
{
    int i, j, ii, jj, curI, curJ;
    pixel_sum ps;

    for (j = 0; j < dim; j++){
        for (i = 0; i < dim; i++){
            ps.red    = 0.0;
            ps.green  = 0.0;
            ps.blue   = 0.0;
            ps.weight = 0.0;
            for (jj = -2; jj <= 2; j++){
            for (ii = -2; ii <= 2; i++){
                    curJ = j+jj;
                    if(curJ<0 || curJ>=dim){
                        continue;
                    }
                    curI = i+ii;
                    if(curI<0 || curI>=dim){
                        continue;
                    }
                    ps.red  += src[RIDX(curI, curJ, dim)].red *  kernel[ii+2][jj+2];
                    ps.green += src[RIDX(curI, curJ, dim)].green * kernel[ii+2][jj+2];
                    ps.blue  += src[RIDX(curI, curJ, dim)].blue *  kernel[ii+2][jj+2];
                    ps.weight += kernel[ii+2][jj+2];
                }
            }
            dst[RIDX(i,j,dim)].red = (unsigned short)(ps.red/ps.weight);
            dst[RIDX(i,j,dim)].green= (unsigned short)(ps.green/ps.weight);
            dst[RIDX(i,j,dim)].blue   = (unsigned short)(ps.blue/ps.weight);
        }
    }
}
```

This code (and the helper functions) are all in the file `kernels.`

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 5 different values of N. All measurements were made on a perf server.

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 5 values. That is, if the measured speedups for $N = \{256, 512, 1024, 2048\}$ are R_{256}, R_{512}, R_{1024}, and R_{2048}, then we compute the overall performance as

$$R = \sqrt[4]{R_{256} \times R_{512} \times R_{1024} \times R_{2048}}$$

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.

Test case	1	2	3	4	
Method N	256	512	1024	2048	Geom. Mean
Naive `flip` (CPE)	21.00	22.00	23.00	24.00	
Optimized `flip` (CPE)	17.95	18.92	18.18	20.38	
Speedup (naive/opt)	1.17	1.16	1.27	1.18	1.19
Method N	256	512	1024	2048	Geom. Mean
Naive `convolve` (CPE)	375.00	385.00	455.8	975.00	
Optimized `convolve` (CPE)	725.33	785.85	899.29	1040.51	
Speedup (naive/opt)	0.52	0.49	0.51	0.94	0.59

Table 1: CPEs and Ratios for Optimized vs. Naive Implementations

4 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 find yourself writing many versions of the `flip` and `convolve` 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:

```
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 convolve 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:

```
unix> ./driver
```

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

- *Default mode*, in which all versions of your implementation are run.
- *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:

`-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 `team` struct in `kernels.c` with your email address and that of your teammate (if applicable).

5 Assignment Details

Optimizing Flip (25 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`) might generate the output shown below:

```
unix> ./driver
Email: bovik@nowhere.edu

team_hash: 3553431680
Your flip description:
    You must reflect about both axes (img[i][j] -> img[dim-1-j][dim-1-i])
Rotate: Version = naive_flip: Naive baseline implementation:
Dim             256     512     1024     2048   Mean
Your CPEs       17.95   18.92   18.18   20.38
Baseline CPEs   21.00   22.00   23.00   24.00
Speedup         1.17    1.16    1.27    1.18    1.19
```

Optimizing Convolve (25 points)

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

For example, running driver with the supplied naive version (for `convolve`) might generate the output shown below:

```
unix> ./driver
Email: bovik@nowhere.edu

team_hash: 3553431680
Your convolution kernel:
{{1.000000, 1.000000, 1.000000, 1.000000, 0.000000},
{1.000000, 16.000000, 4.000000, 0.000000, -1.000000},
{1.000000, 4.000000, 0.000000, -4.000000, -1.000000},
{1.000000, 0.000000, -4.000000, -16.000000, -1.000000},
{0.000000, -1.000000, -1.000000, -1.000000, -1.000000}
};
flip: Version = flip: Current working version:
Dim      256 512 1024    2048    Mean
Your CPEs   18.07   30.03   38.38   49.21
Baseline CPEs   21.90   21.90   43.81   307.62
Speedup     1.21    0.73    1.14    6.25    1.58

convolve: Version = convolve: Current working version:
Dim      256 512 1024    2048    Mean
Your CPEs   232.47  247.83  399.76  541.74
Baseline CPEs   375.00  385.00  455.00  950.00
Speedup     1.61    1.55    1.14    1.75    1.50
```

Some advice. Focus on optimizing the inner-most loop (the code that gets repeatedly executed in a loop) using the optimization tricks covered in class.

Coding Rules

You may write any code in `kernels.c` you want, as long as it satisfies the following:

- It must not interfere with the time measurement mechanism. You may also be penalized if your code has any extraneous prints.

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

6 Evaluation

Your solutions for `flip` and `convolve` will each count for 50% of your grade, or up to 25 code-execution points. The other 50 points come from your interview grading. In your interview, you will be asked to explain what you changed about your code, and why. You might be asked how some small additional change would effect performance.

Your code-execution points will be graded based on your speedup ratio (Your CPEs divided by Baseline CPEs). For the "flip" question, you will get full credit if your speedup ratio is no less than 6. For the "convolve" question, you will get full credit if your speedup ratio is no less than 2. You may also get extra credit (up to 10% of your grade) if

you achieve particularly fast solutions. In this case, the extra credit you'll get is based on:

- your speedup ratios compared with the whole class
- optimization methods you tried

Note, you will get zero code-execution points 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

Run on server

Not applied here for your lab! The server breaks down. We will grade on your VM.

Performance can vary widely from computer to computer, even if all of the machines are using the say virtual-machine image. Thus, we have a set of identical machines (perf-01 - perf-08), and a grading script which will automatically run your code on an available machine (waiting until a machine is available, if there aren't any). Your final score will be determined using this script. You can use this same script yourselves, but to minimize server impact, please only test on the server occassionally - you can check for compile errors and incremental improvements on your own machine. This script/command is `./checkScore.py`. It will upload your `kernels.c` file to the server, run your program on one of the perf machines and send back the results.

7 Hand In Instructions

When you have completed the lab, you will upload one file, `kernels.c`, to Moodle.

- Make sure you have included your identifying information in the team struct in `kernels.c`.
- Make sure that the `flip()` and `convolve()` functions correspond to your fastest implementations, as these are the only functions that will be tested when we use the driver to grade your assignment.
- Remove any extraneous print statements.

Good luck!