CSCI 2400, Spring 2017
Performance Lab
Due: Monday, April 10th, 9:00AM

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: `rotate`, which rotates an image counter-clockwise by 90°, and `smooth`, which "smooths" or "blurs" 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 `rotate` operation can be implemented quite simply as the combination of the following two matrix operations:

- *Transpose*: For each (i, j) pair, $M_{i,j}$ and $M_{j,i}$ are interchanged.
- *Exchange rows*: Row i is exchanged with row $N - 1 - i$.

This combination is illustrated in Figure 1.

The `smooth` operation is implemented by replacing every pixel value with the 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:

$$\texttt{M2[1][1]} = \frac{\sum_{\texttt{i}=0}^{2} \sum_{\texttt{j}=0}^{2} \texttt{M1[i][j]}}{9}$$

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

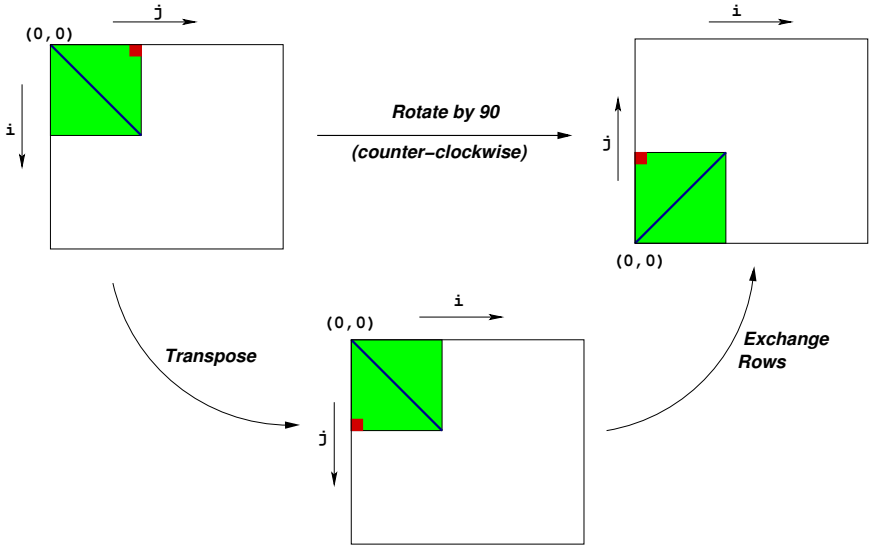

Figure 1: Rotation of an image by 90° counterclockwise

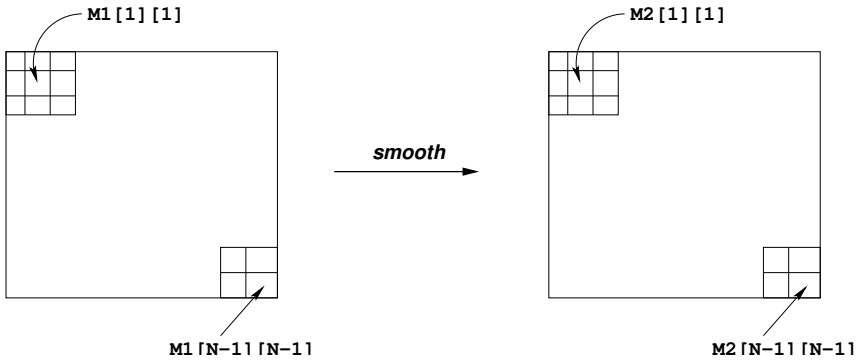

Figure 2: Smoothing an image

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`.

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

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.

Rotate

The following C function computes the result of rotating the source image `src` by $90°$ and stores the result in destination image `dst`. `dim` is the dimension of the image.

```
void naive_rotate(int dim, pixel *src, pixel *dst)
{
    int i, j;
    for (i = 0; i < dim; i++){
        for (j = 0; j < dim; j++){
            dst[RIDX(dim-1-j, i, dim)].red   = src[RIDX(i, j, dim)].red;
            dst[RIDX(dim-1-j, i, dim)].green = src[RIDX(i, j, dim)].green;
            dst[RIDX(dim-1-j, i, dim)].blue  = src[RIDX(i, j, dim)].blue;
```

```
        }
    }
}
```

The above code scans the rows of the source image matrix, copying to the columns 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.

Smooth

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

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

    for (j = 0; j < dim; j++){
        for (i = 0; i < dim; i++){
            initialize_pixel_sum(&ps);
            for(ii = max(i-1, 0); ii <= min(i+1, dim-1); ii++){
                for(jj = max(j-1, 0); jj <= min(j+1, dim-1); jj++){
                    accumulate_sum(&ps, src[RIDX(ii,jj,dim)]);
                }
            }
            dst[RIDX(i,j,dim)].red   = ps.red/ps.num;
            dst[RIDX(i,j,dim)].green = ps.green/ps.num;
            dst[RIDX(i,j,dim)].blue  = ps.blue/ps.num;
        }
    }
}
```

The functions `max`, `min`, `initialize_pixel_sum`, and `accumulate_sum` are all functions you can and probably will want to modify.

This code (and the helper functions) are all 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 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

Test case	1	2	3	4	5	
Method N	64	128	256	512	1024	Geom. Mean
Naive `rotate` (CPE)	5.8	6.6	9.8	10.5	29.5	
Optimized `rotate` (CPE)	4.2	4.1	4.1	5.1	19.6	
Speedup (naive/opt)	1.4	1.6	2.4	2.1	1.5	1.8
Method N	64	128	256	512	1024	Geom. Mean
Naive `smooth` (CPE)	224.8	237.9	240.7	250.8	364.1	
Optimized `smooth` (CPE)	37.8	38.2	38.2	38.9	41.4	
Speedup (naive/opt)	5.9	6.2	6.3	6.5	8.8	6.7

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

for these 5 values. That is, if the measured speedups for $N = \{64, 128, 256, 512, 1024\}$ are R_{64}, R_{128}, R_{256}, R_{512}, and R_{1024}, then we compute the overall performance as

$$R \quad = \quad \sqrt[5]{R_{64} \times R_{128} \times R_{256} \times R_{512} \times R_{1024}}$$

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.

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 `rotate` and `smooth` 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_rotate_functions() {
   add_rotate_function(&rotate, rotate_descr);
}
```

This function contains one or more calls to `add_rotate_function`. In the above example, `add_rotate_function` registers the function `rotate` along with a string `rotate_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 smooth 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.
- *Autograder mode*, in which only the `rotate()` and `smooth()` 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 `rotate()` and `smooth()` 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 your information (name and email address).

5 Assignment Details

Optimizing Rotate (20 points)

In this part, you will optimize `rotate` 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 `rotate`) might generate the output shown below:

```
unix> ./driver
Teamname: bovik
Member 1: Harry Q. Bovik
Email 1: bovik@nowhere.edu

Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       5.8     6.5     10.0    10.4    29.7
Baseline CPEs   5.8     6.6     9.8     10.5    29.5
Speedup         1.0     1.0     1.0     1.0     1.0     1.0
```

Optimizing Smooth (20 points)

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

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

```
unix> ./driver

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       226.0   238.1   240.3   252.4   364.1
Baseline CPEs   224.8   237.9   240.7   250.8   364.1
Speedup         1.0     1.0     1.0     1.0     1.0     1.0
```

Comments on speedup numbers

- If you run your code on your own machine, you may have different numbers, it could be higher or lower speedup numbers, that is because of different hardware and workloads on your machine, thats fine you should start from the numbers you get on your machine, and enhance the speedup.
- The numbers above are obtained from testing and running the code on one of the perf machines (listed below), and even running the code on one of the perf machines multiple times will give slightly different results.
- Your final grade will be determined on how much speedup you get eventually on the perf machines, where you can connect and test your enhanced code.

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. The `smooth` is more compute-intensive and less memory-sensitive than the `rotate` function, so the optimizations are of somewhat different flavors. Consider looking at the assembly code generated for the `rotate` and `smooth`, and/or running a profiler.

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 will also be penalized if your code prints any extraneous information.

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 `rotate` and `smooth` will each count for 50% of your grade, or up to 20 code-execution points. 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. score for each will be based on the following:

- Correctness: 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.
- CPE: You will get full credit for your implementations of `rotate` and `smooth` if they are correct and achieve mean CPEs at or above thresholds $S_r = 1.8$ and $S_s = 6.7$ respectively. You will get partial credit for a correct implementation that does better than the supplied naive one.

More specifically, with S_r your averaged speedup for `rotate` and S_s your averaged speedup for `smooth`, the following equations are used to calculate your code-execution points (assuming you made at least some degree of improvement).

- `rotate` score: $5 + ((S_r - 1.0) * 18.75)$ points
- `smooth` score: $5 + ((S_r - 1.0) * 2.64)$ points

Other Note: Extra credit will require doing even better than $S_r = 1.8$ and $S_s = 6.7$, and may require esoteric or extreme modifications. For now, getting the full 10 extra credit points will require doing 10% better (ie, $S_r >= 2.0$ and $S_s >= 7.4$).

Self-Assessment

Performance can vary widely from computer to computer, even if all of the machines are using the same virtual-machine image. Your final grade will be determined using one of the "perf" machines listed below; the same one used to produce the baseline and optimize values listed above. We have a number of servers set up, which all perform *almost* the same:

- `perf-01.cs.colorado.edu`
- `perf-02.cs.colorado.edu`
- `perf-03.cs.colorado.edu`
- `perf-04.cs.colorado.edu`

To get your files on to the server:

```
scp -r ./perflab-handout <identikey-name>@perf-XX.cs.colorado.edu:~
```

You may be prompted about a security key: type 'yes'. You will be asked for your password: enter it.(Password should be the same as your password for the CSEL machines and myCUinfo)

To ssh in to a server:

```
ssh <identikey-name>@perf-XX.cs.colorado.edu
```

You may be prompted about a security key: type 'yes'. You will be asked for your password: enter it. (Password should be the same as your password for the CSEL machines and myCUinfo)

Once your files are on the server and you are ssh'd in, you can use make to compile your code and the `driver` to test it as normal. Be sure to always recompile your code for the server (use the command `make clean` before `make` if necessary).

Note: You should work all the time on your local machine without connecting to the "perf" servers, when you make sure that you have a performance improvement on your local machine you may connect to one of the servers to verify and test your results, also make sure to logout of the perf machines when you are done with the command logout.

Note: If multiple students are connected to the same server at the same time running tests, it will effect performance. Thus, please log out of the servers when done. You can use the `who` command once ssh'd in to the server to see if anyone else is also using it. If so, you can check one of the other four machines.

Generally, we recommend you do most of your work just testing on your machine – as long as you don't modify `smooth_naive`, the speedup between that and your fastest version will be a decent approximation of your final score.

At least once before your final submission, we recommend you ssh in to one of ther `perf` machines , and run `make clean`, then `make`, then `./driver -g`. The resulting reported score should be very close to the score computed at the start of your grading interview.

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 `rotate()` and `smooth()` 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.

Good luck!