CS 4400: Computer Systems
Fall 2014
Lab Assignment 4: Code Optimization
Due Date: Monday, November 04 at 11:59pm

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 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 columns*: Column i is exchanged with column $N - 1 - i$.

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). The average is "weighted" such that the (i, j)th pixel is added to the sum twice, while all of the pixels around it are added to the sum just once each. Consider Figure 2. The values of pixels `M2[1][1]` and `M2[N-1][N-1]` are given below as examples. (Note: M1 is the original image matrix and M2 is the smoothed image matrix.)

$$\texttt{M2[1][1]} = \frac{\sum_{\texttt{i}=0}^{2} \sum_{\texttt{j}=0}^{2} \texttt{M1[i][j]} + \texttt{M1[1][1]}}{10}$$

$$\texttt{M2[N-1][N-1]} = \frac{\sum_{\texttt{i}=N-2}^{N-1} \sum_{\texttt{j}=N-2}^{N-1} \texttt{M1[i][j]} + \texttt{M2[N-1][N-1]}}{5}$$

Logistics

You can discuss problems while solving the lab but the submitted solutions should be your OWN. Any clarifications and revisions to the assignment will be posted on the course Web page.

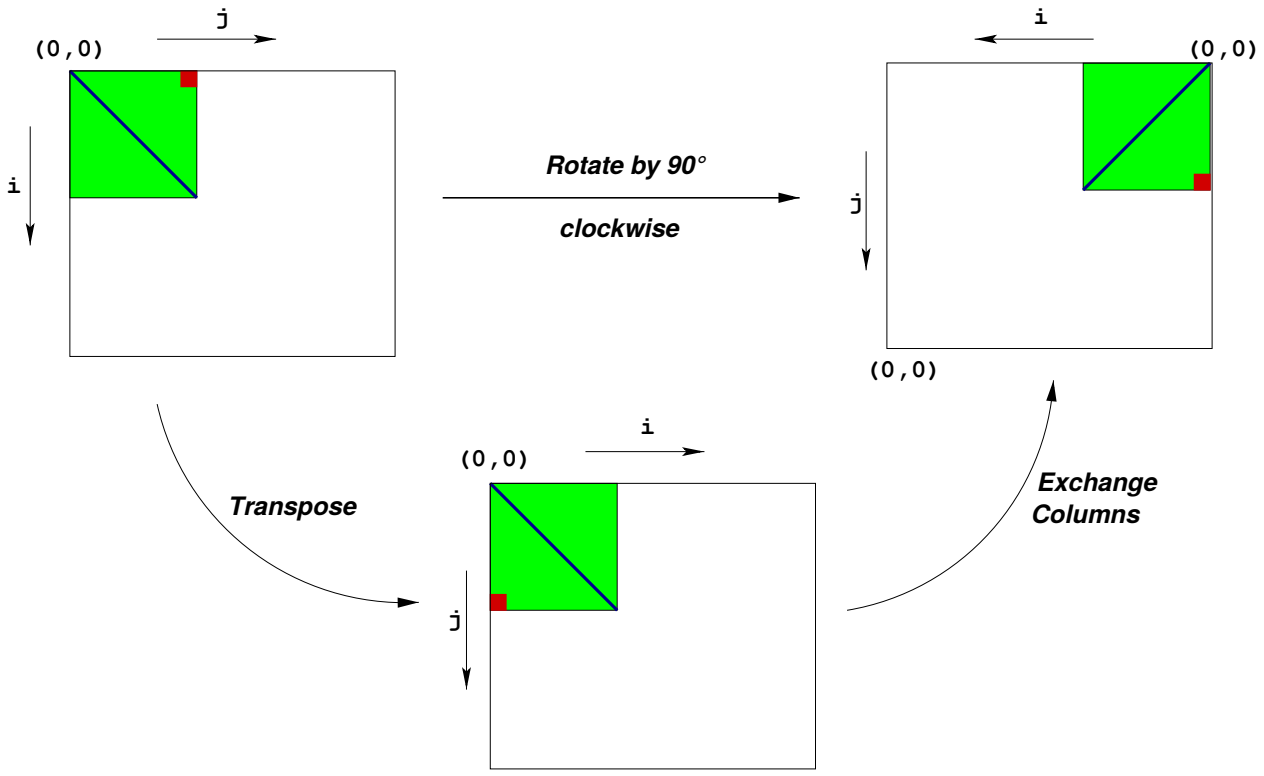

Figure 1: Rotation of an image by 90° clockwise

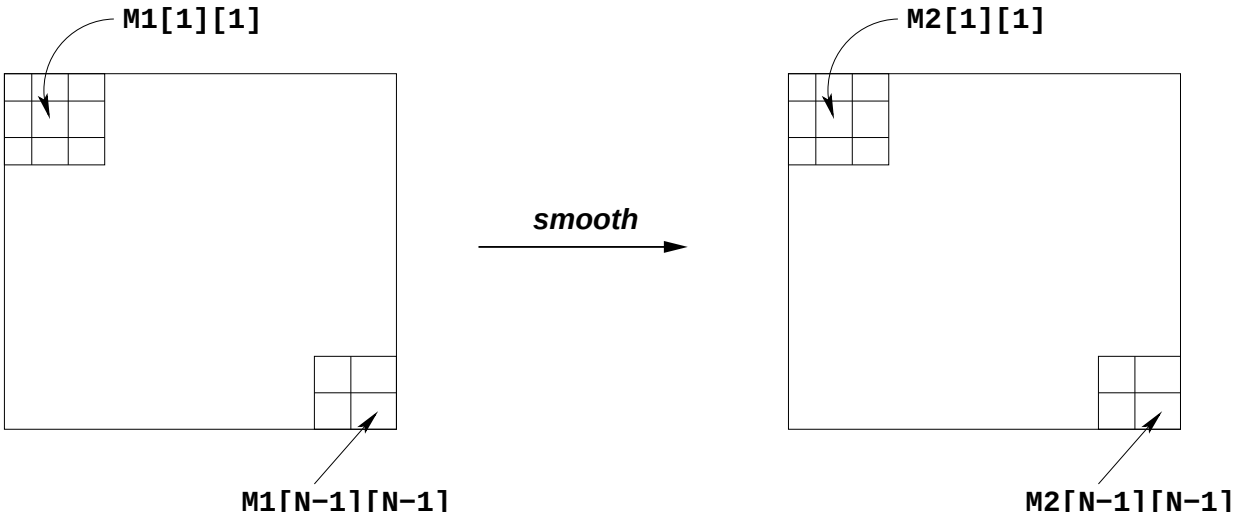

Figure 2: Smoothing an image

Hand Out Instructions

Start by copying `lab4.tar.gz` to a protected directory in which you plan to do your work. Then give the command: `tar -xzvf lab4.tar.gz`. 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 about yourself. **Do this right away so you do not forget.**

Implementation Overview

Data Structures

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

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

As can be seen, RGB values have 8-bit representations ("24-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. NOTE: The `alpha` value is added to force the pixel be 4 bytes long, but it is not used (or checked). The `alpha` does not need to be copied, but it might be faster to do so—try it and see.

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(j,dim-1-i,dim)] = src[RIDX(i,j,dim)];

  return;
}
```

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`.

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

  for(i=0; i < dim; i++)
    for(j=0; j < dim; j++)
      dst[RIDX(i,j,dim)] = avg(dim, i, j, src); /* Smooth the (i,j)th pixel */

  return;
}
```

The function `avg` returns the average of all the pixels around the `(i,j)` th pixel. Your task is to optimize `smooth` (and `avg`) to run as fast as possible. (*Note:* The function `avg` is a local function and you can get rid of it altogether to implement `smooth` in some other way.)

This code (and an implementation of `avg`) 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 six different values of N.

Important: For this lab you may do your development on any machine you like, but the machines that count for grading purposes are `lab1-1` through `lab1-40.eng.utah.edu`, so you should periodically run your code there to see how you are doing. To get information about a particular machine (such as its clock speed), run the command: `cat /proc/cpuinfo`

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 six values. That is, if the measured speedups for $N = \{64, 128, 256, 512, 1024, 2048\}$ are R_{64}, R_{128}, R_{256}, R_{512}, R_{1024} and R_{2048} then we compute the overall performance as

$$R \quad = \quad \sqrt[6]{R_{64} \times R_{128} \times R_{256} \times R_{512} \times R_{1024} \times R_{2048}}$$

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, but we will measure its performance only for the six values shown in Table 1.

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 are to modify is `kernels.c`.

Test case	1	2	3	4	5	6	
Method N	128	256	512	1024	2048	4096	Geom. Mean
Naive `rotate` (CPE)	6.0	9.0	12.0	22.0	32.0	37.0	
Optimized `rotate` (CPE)	1.3	1.7	2.5	6.2	7.14	7.5	
Speedup (naive/opt)	4.5	5.2	4.8	3.5	4.5	4.9	4.5
Method N	32	64	128	256	512	1024	Geom. Mean
Naive `smooth` (CPE)	109.0	109.0	110.0	111.0	114.0	119.0	
Optimized `smooth` (CPE)	24.2	24.5	24.8	24.8	24.1	25.2	
Speedup (naive/opt)	4.5	4.4	4.4	4.5	4.7	4.7	4.5

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

Versioning

You will likely be 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 example given above, `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 following command

```
unix> 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 following 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.

Important Note: Due to the number of times each version of your `rotate` and `smooth` functions are run for different problem sizes, and the collection of information from hardware cycle counters, `driver` is a long-running program. Therefore, blind trial and error will be slow way to develop your solution. Furthermore, if you find that a run is taking longer than 30 minutes, consider switching to a different `lab1` machine and/or reducing the number of versions you are running.

Assignment Details

Optimizing Rotate (40 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`) generates the output shown below:

```
lab1-10> ./driver
Student: Peter Parker
Login: pparker
Email: pparker@cs.utah.edu

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim                 32     64     128     256     512     1024    Mean
Your CPEs           108.7  109.3  110.1   112.1   114.5   118.4
Baseline CPEs       109.0  109.0  110.0   111.0   114.0   119.0
Speedup             1.0    1.0    1.0     1.0     1.0     1.0     1.0
...
```

Optimizing Smooth (40 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`) generates the output shown below:

```
lab1-10> ./driver
Student: Peter Parker
Login: pparker
Email: pparker@cs.utah.edu
```

```
Smooth: Version = naive_smooth: Naive baseline implementation:
Dim              32      64      128     256     512     Mean
Your CPEs        695.8   698.5   703.8   720.3   722.7
Baseline CPEs    695.0   698.0   702.0   717.0   722.0
Speedup          1.0     1.0     1.0     1.0     1.0     1.0
```

Note: Small variations between our reported numbers and what you see on the `lab1` machines are normal and to be expected. They are caused by inherent imprecision in the timing routines. If you run on some other machine (outside the CADE lab or on the other lab machines) you may see much greater variations. You may also see large variations if the machine is being used by other users (for example, your classmates just before the deadline). Remember that you will be graded on the `lab1` machines and should test your solutions there before submitting.

Some advice. Look at the assembly code generated for the `rotate` and `smooth`. Focus on optimizing the inner 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.

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 are *strongly encouraged* to comment your code with descriptions of the optimizations implemented and why they should improve the performance.
- *You will be penalized 15% 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.

Evaluation

Your solutions for `rotate` and `smooth` 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 `rotate` and `smooth` if they are correct and achieve mean speedups of 3.8 (for rotate) and 3.8 (for smooth) relative to the naive implementations. You will get partial credit for a correct implementation that does better than the supplied naive one.
- This lab is worth 80 points. You will receive 40 of these points if you tried to solve the performance problems but didnt do very well at optimizing the functions. The remaining 40 points will be on a linear scale that depends on where your best speedup falls in the spectrum between 1.0 and 3.8, for each problem.
- Five points of extra credit will be awarded to the person whose `rotate` is the fastest overall, and five points will go to the person whose `smooth` is fastest overall. In the case of a tie (where there is no statistically significant difference between the speed of the two or more best solutions) these points will be divided amongst the winners. This means that if you think you have one of the fastest solutions, it is not to your advantage to share information about your strategy.

Hand In Instructions

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

- Make sure you have included your identifying information in the struct `Student` 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.
- Submit your `kernels.c` as the solution file through Canvas.

Good luck!