ECEn 424
The Performance Lab: Code Optimization

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:

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

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

2 Logistics

You may work in a group of up to two people on this lab. Any clarifications and revisions to the assignment will be posted on the web page for the lab.

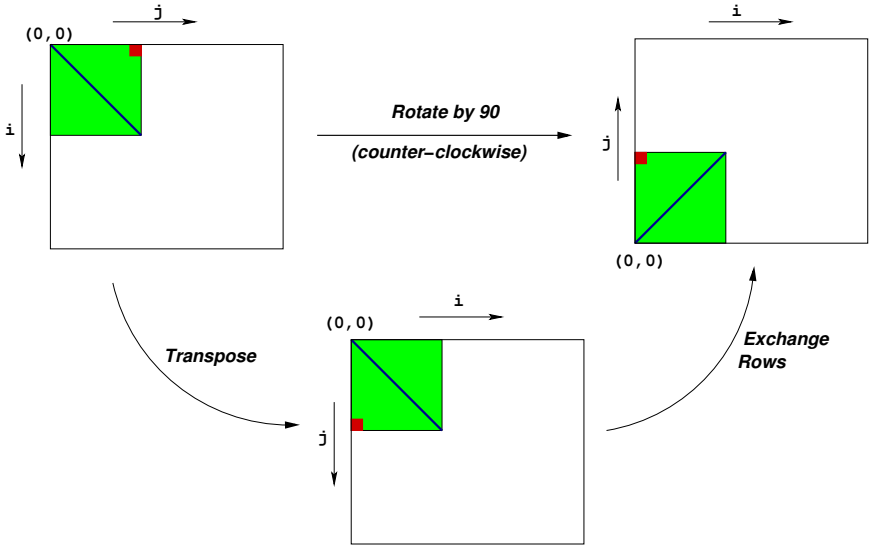

Figure 1: Rotation of an image by 90° counterclockwise

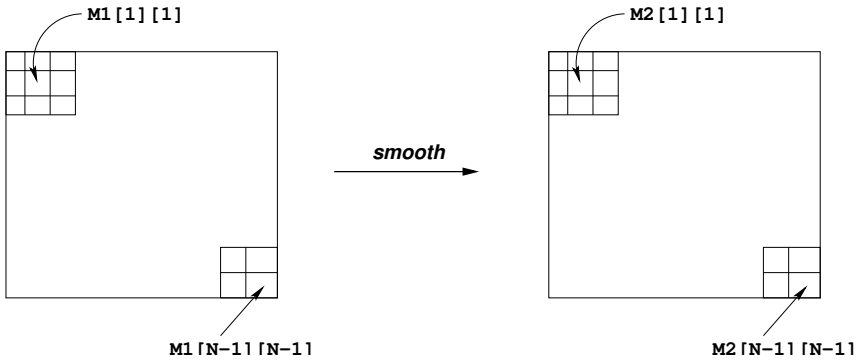

Figure 2: Smoothing an image

3 Handout Instructions

The handout for this lab is the `perflab-handout.tar` file that can be downloaded from the lab web page. Start by copying `perflab-handout.tar` to a protected 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 `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.

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

  return;
}
```

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;

  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 if you choose.)

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.

Your implementation will be scored based on the speedup of your optimized implementation relative to the naive version. To summarize the overall effect over different values of N, we will compute the *geometric mean* of the measured speedup for 5 different vector lengths. That is, if the measured speedups for $N = \{32, 64, 128, 256, 512\}$ are R_{32}, R_{64}, R_{128}, R_{256}, and R_{512} respectively, then we compute the overall performance as

$$R \quad = \quad \sqrt[5]{R_{32} \times R_{64} \times R_{128} \times R_{256} \times R_{512}}.$$

Table 1 shows an example of the output produced by the lab software for the naive implementation and an optimized version. As you can see, it reports the performance for each of 5 different values of N, and it reports the bottom-line number used to grade this lab: the geometric mean of the 5 measured speedups. All of these measurements were made on the ECEn Spice machines.

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	64	128	256	512	1024	Geom. Mean
Naive `rotate` (CPE)	3.1	3.9	6.7	10.6	14.5	
Optimized `rotate` (CPE)	2.3	2.3	2.4	2.5	8.2	
Speedup (naive/opt)	1.4	1.7	2.8	4.2	1.8	2.2
Method N	32	64	128	256	512	Geom. Mean
Naive `smooth` (CPE)	57.7	57.8	57.9	58.0	57.5	
Optimized `smooth` (CPE)	10.8	10.9	10.9	10.9	10.9	
Speedup (naive/opt)	5.4	5.3	5.3	5.3	5.3	5.3

Table 1: CPEs and Ratios for Optimized vs. Naive 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 file you will be modifying is `kernels.c`.

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 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

```
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 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.

Team Information

Important: Before you start, make sure you fill in the struct in `kernels.c` with all the requested information.

6 Assignment Details

Optimizing Rotate (50 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 something very similar to the output shown below:

```
unix> ./driver
[...]

Rotate: Version = naive_rotate: Naive baseline implementation:
Dim              64      128     256     512     1024    Mean
Your CPEs        2.9     4.0     6.5     10.7    14.5
Baseline CPEs    3.1     3.9     6.7     10.6    14.5
Speedup          1.1     1.0     1.0     1.0     1.0     1.0
```

Optimizing Smooth (50 points)

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

For example, running the driver with the supplied naive version (for `smooth`) generates something very similar to the output shown below:

```
unix> ./driver
[...]

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim              32      64      128     256     512     Mean
Your CPEs        57.6    58.3    58.2    58.1    58.1
Baseline CPEs    57.7    57.8    57.9    58.0    57.5
Speedup          1.0     1.0     1.0     1.0     1.0     1.0
```

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.

The only file you are allowed to modify is `kernels.c`. You are allowed to define macros, additional global variables, and other procedures in these files.

Recommendations

There is an art to code optimization: there is not a "correct" or single best answer that all of us will arrive at, given enough time. It is entirely possible that you will come up with a quite novel approach that meets the requirements of the lab. It is also quite likely that you will try out some ideas that do not improve performance; some changes that seem perfectly reasonable on the surface may actually make things worse. *Simply put, there is no shortcut to trying out a number of your ideas and seeing what works.* Welcome to the exciting world of software optimization.

In general, you are likely to see the best options to try once you fully understand what the code is doing. Think it through carefully. It can be quite helpful to look at the assembly code that GCC generates, particularly when the result of a code change is counterintuitive. Whether or not you decide to study the assembly code, make sure you understand what the baseline C code is doing in the inner loops. You might find it useful to ask yourself questions like the following. What are the access patterns to the array elements? How well do those access patterns map to cache blocks? How much work is performed in each macro? Is all that work necessary at every point? How much work is performed within each function call? Can that work be reduced? Can the function itself be eliminated?

For `rotate`, think about making good use of each cache block once it is loaded into the cache. Loop unrolling and blocking (see page 647) are worth thinking about.

For `smooth`, think about the principal sources of inefficiency in the baseline code. Making small changes can be quite disappointing: the compiler inlines all the functions in the naive code, and small code changes can cause the compiler to make regular function calls instead, increasing the runtime overhead. Focus on the big picture. Consider, for example, how many times each of the relevant pixels is read from the source array in one row pass (writing one row of pixels to the destination array). Can you devise a way of storing relevant values in local variables (hopefully

a manageable number) so that your code only reads each relevant pixel value once per row pass? Consider also the overhead of computing the conditionals in min, max, and avg. Can you think of a way to dramatically reduce or eliminate this overhead? Might it help to separate the outer loop (processing row by row) into separate pieces, allowing special handling of the first row, then interior rows, then the last row? If you go down this path, your code will get a lot longer, but carefully crafted macros can keep it much more readable.

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 CPEs above thresholds 2.0 and 4.0 respectively. You will get some partial credit for any submission that is both correct and better than the naive version.

7 Hand In Instructions

When you are satisified with the performance of your best `rotate` and `smooth` functions, submit your `kernels.c` file via Learning Suite. Before you submit, please double check the following:

- 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 assignment.
- Remove any extraneous print statements.