CS 213, Fall 2001
Lab Assignment L4: Code Optimization
Assigned: October 11
Due: October 25, 11:59PM

Sanjit Seshia (`sanjit+213@cs.cmu.edu`) is the lead person for this assignment.

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:

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

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

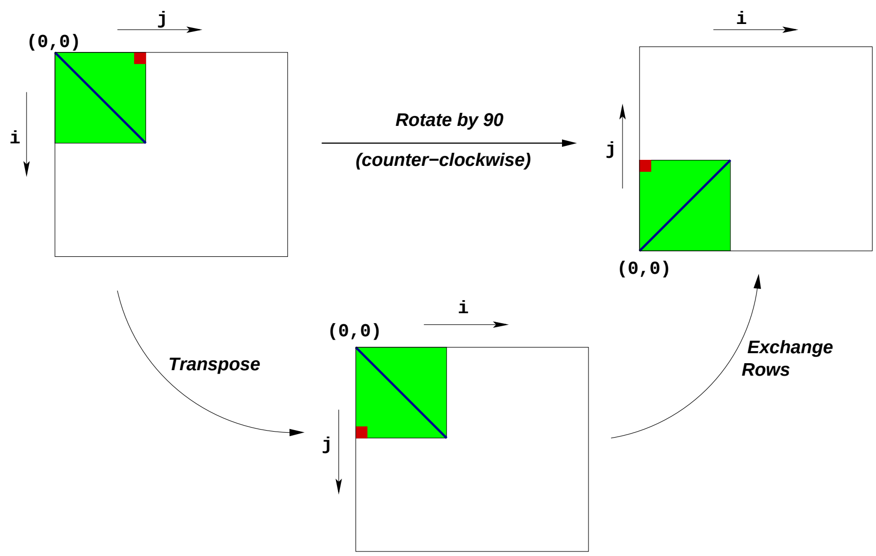

Figure 1: Rotation of an image by $90°$ counterclockwise

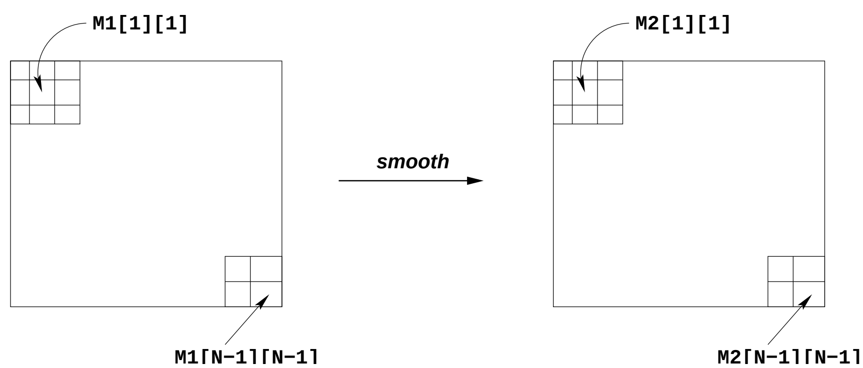

Figure 2: Smoothing an image

2 Logistics

The files for this assignment can be retrieved from

`/afs/cs.cmu.edu/academic/class/15213-f01/L4/L4.tar`

Once you've copied this file into a (private) directory, run the command `tar -xvf L4.tar` and fill in your team information in the structure at the beginning of the file `rotate.c`. You may work in a group of up to two people in solving the problems of this assignment.

When you have completed the lab, you will hand in three files: `rotate_cache.c`, `rotate.c` and `smooth.c` that contain your solution. Each file corresponds to a part of this lab. For the first part you will be graded on the cache performance of a routine `rotate` in `rotate_cache.c`. For the second part, you will be graded on the performance of your code for the routine `rotate` in `rotate.c`, and for the last part you will be graded on the performance of your code for the routine `smooth` in `smooth.c`. Your grade will be determined by how well your routines perform compared to an optimized reference solution.

In addition to running your code locally on a Fish machine, you will be able to submit your source files to a timing server. Both this and the final hand-in will be performed via a web interface. The instructions for web hand-in will be posted on the lab webpage.

Any clarifications and revisions to the assignment will be posted on the course web page.

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 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 `rotate.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.)

This code (and an implementation of `avg`) is in the file `smooth.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 5 different values of N. All measurements were made on the Pentium III Xeon Fish machines.

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 $N = \{32, 64, 128, 256, 512\}$ are R_{32}, R_{64}, R_{128}, R_{256}, and R_{512} then we compute the overall performance as

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

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)	14.74	41.30	46.14	69.64	99.67	
Optimized `rotate` (CPE)	9.89	12.33	20.89	25.01	26.56	
Speedup (naive/opt)	1.49	3.35	2.21	2.78	3.75	2.58
Method N	32	64	128	256	512	Geom. Mean
Naive `smooth` (CPE)	695.85	698.55	704.82	719.44	723.18	
Optimized `smooth` (CPE)	107.30	109.17	109.24	121.17	122.86	
Speedup (naive/opt)	6.49	6.40	6.45	5.94	5.89	6.23

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 files you will be modifying are `rotate.c`, `rotate_cache.c`, and `smooth.c`.

Versioning

You will 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 `rotate.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 `rotate.c` to see how to create the string descriptions. This string can be atmost 256 characters long.

A similar function is provided in the file `smooth.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 either `rotate.c` or `smooth.c`.

To test your implementations, you can then run the command:

```
unix> ./driver
```

`driver` can be run in three different modes.

1. *Default mode*, in which all versions of your implementation are run.
2. *File mode*, in which only versions that are mentioned in an input file are run.
3. *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. Other modes and options can be specified by command-line arguments to `driver`, as listed below:

`-f` FUNC_FILE: Execute only those versions specified in FUNC_FILE (*file mode*).

`-d` DUMP_FILE: Dump the names of all versions to DUMP_FILE, *one line* to a version (*dump mode*).

`-q`: Quit after dumping version names to a dump file. To be used in tandem with `-d`.

`-s` SEED: For creating input arrays, use seed SEED for the random number generator.

`-h`: Print the command-line usage.

cdriver

For Part I below, you will need to use a different version of `driver` called `cdriver`. Details on how to use this are given in the description for Part I.

Team Information

Important: Before you start, you should fill in the struct in `rotate.c` with information about your team (group name, team member names and email addresses). This information is just like the one for Lab 1. The group name will be used to display statistics on the webpage.

5 Assignment Details

Part I: Optimizing Simulated Cache Performance (15 points)

As can be observed from the section 3, both the operations, especially `rotate`, are fairly memory-intensive, operating on images that can be of large size. Thus, a good way to optimize performance of the code is to first focus on its cache behavior, and reduce slowdown due to memory operations.

Cache performance of a routine can be evaluated by looking at the total number of cache misses (normalized by the size of the image matrix). We call this quantity the *miss score*. Formally, the miss score is defined as $\#\texttt{misses}/N^2$. Since the miss score is directly proportional to the total number of misses, the *lower* the miss score, better the cache performance of the implementation. In doing cache optimizations, we will focus our attention on the L1 cache. The Pentium III Xeon (Fish) machines that you will be running your code on have a 16 KB 4-way set associative L1 cache with 32 byte lines.

In this part, you will only focus on optimizing cache performance of `rotate`. To help you get a feel for how good your cache performance is, you will first use a *cache simulator* to compute the miss scores for your code. In Part II, you can see how a low miss score (most often) translates into better CPE.

The tool we use to simulate cache performance, called `cacheprof`, is a public-domain cache simulator (`http://www.cacheprof.org/`). `cacheprof` instruments assembly code to capture the source (destination) addresses of read (write) instructions, and uses them to count hits and misses in a simulated cache.

Here is your task for this part of the assignment:

1. Copy `rotate.c` to a new file named `rotate_cache.c`.
2. Optimize the function `rotate` in `rotate_cache.c` to achieve as low a miss score as possible. To do this, you will use the programs `cdriver` and `miss_score`.

Using cdriver and miss_score

We have provided the object code for cdriver in the file `cdriver.o`. To compile `cdriver` execute the command

```
unix> make cdriver
```

`cdriver` takes the same command-line arguments as `driver` and runs in the same three different modes. You can handle different versions in the same way. However, most of this is hidden from you – you will only explicitly run `cdriver` in dump mode. All other runs are performed within the `miss_score` script.

Suppose you copy the provided naive implementation in `rotate.c` to `rotate_cache.c` (enter a suitable team name), compile `cdriver` using it, and run the following command:

```
unix> ./cdriver -q -d dump_file
```

You will observe the following output:

```
==cacheprof== level-2 instrumented program: startup
Teamname: Harry Q. Bovik
Member 1: Harry Bovik
Email 1: bovik@nowhere.cmu.edu

==cacheprof== level-2 instrumented program: run complete
```

Test case		1	2	3	4	5	
Method	N	64	128	256	512	1024	Geom. Mean
Naive `rotate` (Miss score)		0.3765	1.3129	1.3126	1.3125	1.3125	
Optimized `rotate` (Miss score)		0.3765	0.3754	0.3751	0.3979	0.4490	
Ratio (naive/opt)		1.00	3.50	3.50	3.30	2.92	2.60

Table 2: Miss scores for naive and optimized versions of `rotate`.

```
==cacheprof==            10 insns
==cacheprof==             6 refs   (           2 rd +             4 wr)
==cacheprof==             2 misses (           0 rd +             2 wr)
==cacheprof== 0.00 seconds, inf MIPS
```

Ignore all the lines that start with `==cacheprof==`. To get a summary of the miss score, execute the following command:

```
unix> ./miss_score --file dump_file
```

This prints a summary of the miss scores for each size, as shown below

```
Version: R:Naive Row-wise Traversal of src
Dim       64      128     256     512     1024
Score    0.3765  1.3129  1.3126  1.3125  1.3125
```

Note that the script `miss_score` *needs* the `--file` argument.

You'll be graded on this part based on how low a miss score you are able to achieve. Miss scores achieved after some optimization are shown in Table 2.

Note: Since this part deals with *simulated* cache performance, you can work on this locally, without waiting to submit it to the timing server.

Some Advice: Don't spend overly too much time tuning Part I; it is intended more as a warmup to Part II. As you might find out in the course of this lab, a lower cache miss rate reported by the simulator does not always mean better CPE.

Part II: Optimizing Rotate (35 points)

In this part, you will optimize `rotate` to achieve as low a CPE as possible. You can use your answer to Part I as a starting point for this part (copy `rotate_cache.c` to `rotate.c`). 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:

```
unix> ./driver -f rotate_func_file
```

```
Teamname: Harry Q. Bovik
Member 1: Harry Bovik
Email 1: bovik@nowhere.cmu.edu

Rotate: Version = Currently set to: Naive Row-wise Traversal of src:
Dim     64      128     256     512     1024    Score
CPE     14.75   40.11   48.11   71.51   97.79
Speedup 1.00    1.00    1.00    1.00    1.00    1.00
```

Part III: Optimizing Smooth (50 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:

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

Smooth: Version = Currently set to: Naive Implementation of Smooth:
Dim      32      64      128     256     512     Score
CPE    695.85  698.55  704.82  719.44  723.18
Speedup  1.00    1.00    1.00   1.00     1.00    1.00
```

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

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 cache simulation or time measurement mechanism. You will also be penalized if your code prints any extraneous information.

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

Evaluation

Your grade will be based on the following:

- Correctness: You will get NO CREDIT for buggy code! 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.
- Cache performance: You will get full credit for Part I if your implementation is correct and achieves a mean ratio of miss scores above a certain threshold S_1. You will get partial credit for a correct implementation that does better than the supplied naive one.
- CPE: You will get full credit for Parts II and III if your implementation is correct and achieves mean CPEs above certain thresholds S_2 and S_3 respectively. You will get partial credit for a correct implementation that does better than the supplied naive one.
- The thresholds S_1, S_2 and S_3 will be posted on the lab web page by Saturday, October 13th. Meanwhile, the scores/speedups presented in Table 1 and Table 2 can be used as guidelines (the posted thresholds will not be worse than these).

6 Epilogue

This is a pretty long handout, but don't be discouraged by the length! The length of this handout is for clarity, and the assignment is not difficult once you get warmed up. Start early and feel free to discuss the assignment with the course staff. We look forward to your feedback.

You can work on the parts in any order, but we strongly recommend that you do Part I before Part II. The concepts involved in Part III are somewhat independent of Parts I and II.

Good luck!