CS 105
Lab 5: Code Optimization
See Calendar for Dates

1 Introduction

This assignment deals with optimizing C 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:

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

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

2 Logistics

You are to work in a group of two people in solving the problems for this assignment. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on the lab Web page or course email.

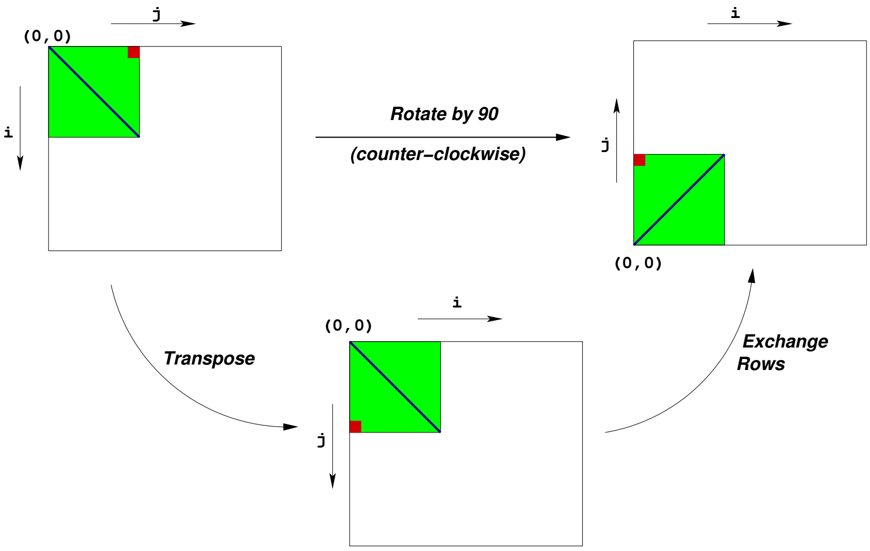

Figure 1: Rotation of an image by 90° counterclockwise

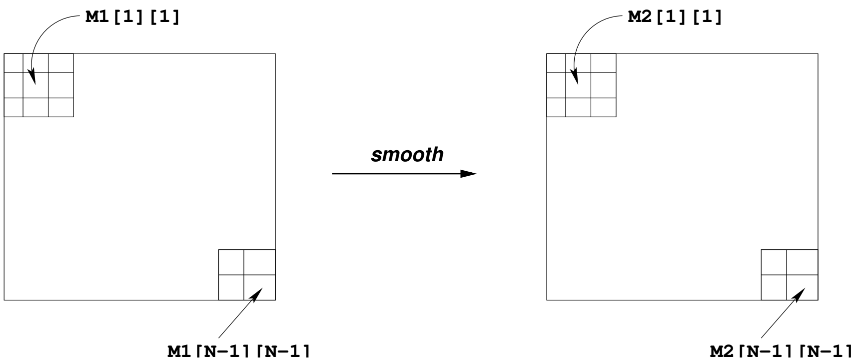

Figure 2: Smoothing an image

3 Handout Instructions

The materials for this lab are on the course 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. Note that you are *not* allowed to change the Makefile, which also means you are not allowed to fiddle with compiler switches.

Looking at the file kernels.c you'll notice a C structure, team, into which you should insert the requested identifying information about the 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 pixels, 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.)

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 5 different values of N. All measurements were made on Wilkes, which is a Pentium III Xeon machine.

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

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

Test case	1	2	3	4	5	
Method N	64	128	256	512	1024	Geom. Mean
Naive `rotate` (CPE)	22.1	21.6	27.6	79.8	220.9	
Optimized `rotate` (CPE)	8.0	8.6	14.8	22.1	25.3	
Speedup (naive/opt)	2.8	2.5	1.9	3.6	8.7	3.1
Method N	32	64	128	256	512	Geom. Mean
Naive `smooth` (CPE)	524	525	527	522	523	
Optimized `smooth` (CPE)	41.5	41.6	41.2	53.5	56.4	
Speedup (naive/opt) /	12.6	12.6	12.8	9.8	9.3	11.3

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

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 (note that the CPEs and speedups in this table will not match those you'll actually see).

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 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, you should fill in the struct in `kernels.c` with information about your team (group name, team member names and email addresses). This information is just like the one for the Data Lab.

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 the output like that 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        14.6    40.9    46.8    63.5    90.9
Baseline CPEs    14.7    40.1    46.4    65.9    94.5
Speedup          1.0     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 driver with the supplied naive version (for `smooth`) generates the output like that shown below:

```
unix> ./driver

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
```

Grading

There are two things to consider in grading. First, the `Basline CPEs` are calculated by running the rotate and smooth code with no performance modifications. `Speedup` is the critical factor for the grading program. The grading formula is linear in the speedup, with a diminishing rate of return once you have gotten past a certain threshold, and a maximum beyond which you only get bragging rights. The rules are:

Rotate Grading	
Speedup Range	Points
$1.0 < \text{Rotate Speedup} \leq 2.1$	$40 \times (\text{Speedup} - 1.0)/(2.1 - 1.0)$
$2.1 < \text{Rotate Speedup} \leq 2.5$	$40 + 10 \times (\text{Speedup} - 2.1)/(2.5 - 2.1)$
$2.5 < \text{Rotate Speedup}$	50

Smooth Grading	
Speedup Range	Points
$1.0 < \text{Smooth Speedup} \leq 2.0$	$40 \times (\text{Speedup} - 1.0)/(2.0 - 1.0)$
$2.0 < \text{Smooth Speedup} \leq 4.0$	$40 + 10 \times (\text{Speedup} - 2.0)/(4.0 - 2.0)$
$4.0 < \text{Smooth Speedup}$	50

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 optimizations that will work best depend on the particular function; tricks that work well on `rotate` won't help much with `smooth`, and vice versa. It may be worth experimenting with the order of loops, or even running loops backwards. It can also help to make the most common case run faster even if the less usual case runs slower.

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.

You can only modify code in `kernels.c`. You are allowed to define macros, additional global variables, and other procedures in these files. **You may not modify the `Makefile`.**

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 2.5 and 4.0 respectively. You will get partial credit for a correct implementation that does better than the supplied naive one.

7 Handin Instructions

Use `cs105submit` on Wilkes or Knuth to hand your code in.

8 Notes on Measurement

The measurement techniques used by this lab are somewhat statistical. You can expect to get slightly different results on different runs. Occasionally, larger variations may occur; if so, just try again before you conclude that your changes were made a huge difference. To protect against this problem, we will run your solutions several times and choose the best result.

9 Notes on GCC's Optimization Techniques

The Gnu C Compiler has some "helpful" optimization behaviors that can make things confusing for you. Our strongest advice for dealing with these problems is to look at the assembly code before you assume what's going on in the machine. You can generate relatively readable assembly in `kernels.s` by running "`gcc -O2 -S kernels.c`".

For early optimization, it can be good to concentrate on the inner loop in the generated code; this can be found by looking for backwards jumps.

9.1 Code Motion

Note that under some circumstances, the compiler will in fact move function calls out of loops. In particular, it may do this for "`max`" and "`min`" calls.

9.1.1 Inlining

The most surprising thing the compiler may do for you is called inlining. Inlining is (basically) substituting the source code of one function into another function that calls it. For example:

```
static int mean(int a, int b)
{
    return (a + b) / 2;
}

static int loopy(int x[10][10], int y[10][10])
{
    int i, j;
    for (i = 0;  i < 10;  i++)
        for (j = 0;  j < 10;  j++)
            x[i][j] = mean(x[i][j], y[i][j]);
}
```

Under high optimization levels (including `-O2`), the compiler may decide to replace the "`mean` call with its actual code, so that "`loopy`" would read (in effect):

```
static int loopy(int x[10][10], int y[10][10])
{
    for (i = 0;  i < 10;  i++)
        for (j = 0;  j < 10;  j++)
            x[i][j] = (x[i][j] + y[i][j]) / 2;
}
```

The exact conditions for inlining are complex, but seemingly small changes to your code can cause the compiler to change its mind in either direction. For example, if "`mean`" is called from only one place, it is more likely to be inlined; adding another call in a different function can cause the inlining to go away.

Also, inlining can sometimes hurt performance. This happens most commonly when the compiler runs out of registers. The current compiler behaves very badly when it is out of registers; sometimes it even generates code like this:

```
movl %eax, -88(%ebp)
movl -88(%ebp),%eax
```

and then never uses the value in `-88(%ebp)` again!

To control inlining, you can play around with the compiler's view of the world. If you add the "`inline`" keyword to a function declaration, you are telling the compiler you think it should be inlined:

```
static inline int mean(int a, int b)
{
    ...
}
```

This only works if “`mean`” is defined *before* it is used.

Contrariwise, there are several ways to (try to) *prevent* inlining. In order of probable effectiveness (we haven't experimented with all of these), they are:

- Compile with `-fno-inline` (this prevents inlining of *all* functions). (This works in general, but won't work for this lab because we use our own `Makefile` for grading.)
- Declare the function using the very ugly gcc attribute extension:

  ```
  static int mean(int a, int b)
      __attribute__ ((noinline))
  {
      ...
  }
  ```

  (Note that the doubled parentheses are necessary.)
- Call the function from two or more different places. (Note: if one of the places is a dummy function that isn't actually used, the dummy must be global or the compiler will optimize it out of existence.)
- Declare the function as a global rather than a static:

  ```
  int mean(int a, int b)
  {
      ...
  }
  ```
- Declare the function with a prototype before it is used, but define it only after all uses:

  ```
  static int mean(int a, int b);

  static int loopy(...)
  {
      ...
  }

  static int mean(int a, int b)
  {
      ...
  }
  ```