

# CS 211 Homework 6, Fall 2008

## Code Optimization

### Due: December 2nd

Homework 6 is optional. You may choose to complete this homework and earn 20 points towards your homework totals – this could help you offset poor scores that you may have received in your earlier homeworks. Note that even if you are not planning on submitting HW6, you must still read through this question in order to answer Question 2 in Homework 5.

## 1 Introduction

This assignment deals with optimizing memory intensive code. Image processing offers many examples of functions that can benefit from optimization. We consider two image processing operations: `rotate`, which rotates an image counter-clockwise by  $90^\circ$ , and `smooth`, which “smooths” or “blurs” an image. You have probably used these operations when dealing with images (in Photoshop, Paint, etc.).

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  $3 \times 3$  window centered at that pixel). Consider Figure 2. The values of pixels  $M2[1][1]$  and  $M2[N-1][N-1]$  are given below:

$$M2[1][1] = \frac{\sum_{i=0}^2 \sum_{j=0}^2 M1[i][j]}{9}$$
$$M2[N-1][N-1] = \frac{\sum_{i=N-2}^{N-1} \sum_{j=N-2}^{N-1} M1[i][j]}{4}$$

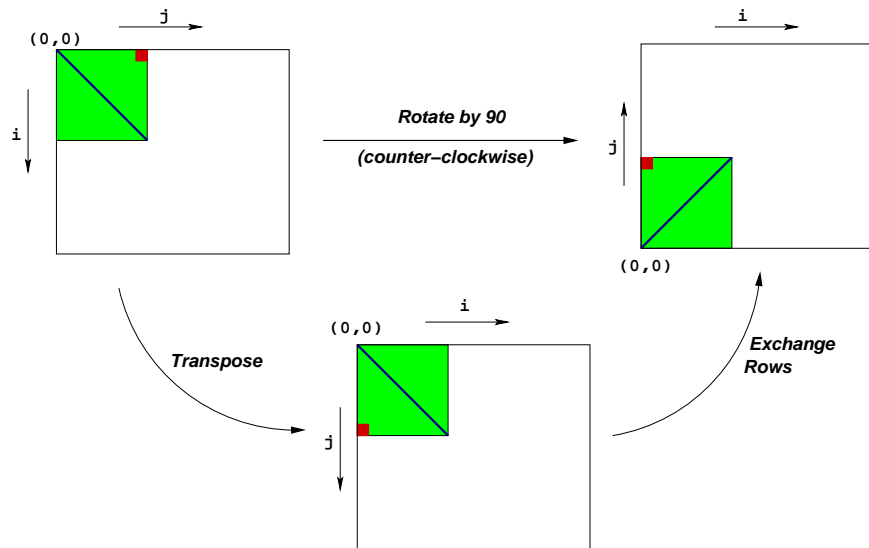


Figure 1: Rotation of an image by 90° counterclockwise

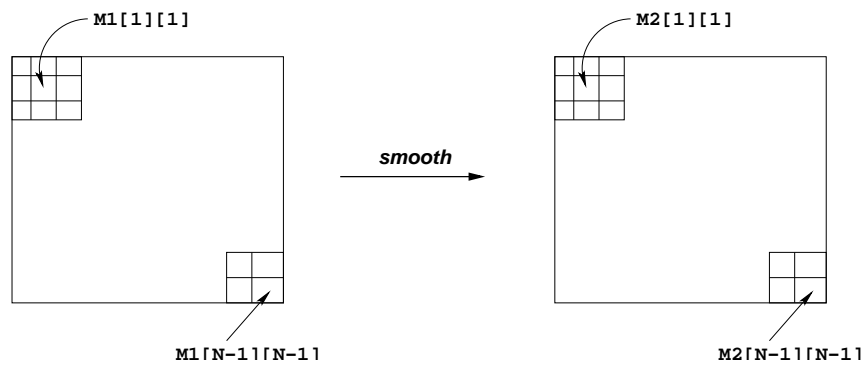


Figure 2: Smoothing an image

## 2 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))
```

### Rotate

The following C function computes the result of rotating the source image `src` by  $90^\circ$  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 some of the techniques discussed in the course.

### 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.)

## 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  $C/N^2$ .
- Since you are familiar with the `simpleScalar` infrastructure, you should use this to measure your performance for different values of  $N$ , for  $N=32,64,128,512$  (if  $N=512$  takes a long time to run then you can stop with the smaller values).
- you are required to “hand optimize” – i.e., rewrite your code by applying the various optimization techniques you have learnt. You cannot use gcc optimization levels to improve your code (this was covered in of homework 5).
- You should turn in a table summarizing the performance of your improved code, and your C code.
- Please submit the (1) table and results summary and (2) the C code as a separate attachments – as specified in blackboard.

## 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  $N$ .