Take Assessment: Exercise 5: Cache Lab

Cache Lab: Improving Program Locality

INTRODUCTION

This exercise deals with optimizing memory-intensive code. Image processing is one area that benefits greatly from such optimizations.

In this exercise we'll be optimizing two functions: rotate, a function designed to rotate an image 90 degrees clockwise, and smooth, a function designed to smooth (or more precisely to blur) an image. Your goal is to maximize the cache hit rate of these functions on a simulated L1 cache. We provide you with a cache simulator that simulates the performance of a computer's cache.

For our purposes, we will consider an image to be represented by a two-dimensional matrix M, where $M_{i,j}$ denotes the (i,j)th pixel of M. To simplify things, this assignment will deal only with square images. Rows and columns are numbered in zero-indexed fashion (like arrays), so rows and columns number from 0 to N-1, where N is the width/height of the image matrix. Pixel values consist of four 1-byte fields representing red, green, blue, and alpha.

LOGISTICS

The files needed for this assignment can be downloaded here. Once you've extracted the zip file to its own directory, you'll see a number of C source and header files:

File:	Function:
cache.c	Contains the code used for the cache simulation
cache.h	Header file for cache simulator
defs.h	Contains commonly used definitions and structures
exercise5.dsp	Exercise 5 project file
exercise5.dsw	Exercise 5 workspace file
driver.c	The main program for testing various functions
rotate.c	Contains the rotate functions. You will modify this file.
smooth.c	Contains the smooth functions. You will modify this file.

The only files you need to change, and the only files you will submit, are rotate.c and smooth.c. You're free to change the driver program as you see fit, but such changes won't be submissible.

IMPLEMENTATION DETAILS

Data Representation:

The fundamental data structure of our images is the pixel structure, shown below:

```
typedef struct {
        unsigned short red      : 8;
        unsigned short green : 8;
        unsigned short blue     : 8;
        unsigned short alpha : 8;
} pixel;
```

The structure definition above defines a 32-bit pixel, with 8 bits for each of the red, green, blue and alpha (opacity) components.

A two-dimensional square image of width n is stored in a one-dimensional array of pixels; the (i, j)th pixel of the image is at Img[PIXEL(i,j,n)], and PIXEL is defined as follows:

```
#define    PIXEL(i,j,n)    ((i)*(n)+(j))
```

In order to use the cache simulator, we call it indirectly through use of the COPY and SMOOTH macros defined in defs.h. You **must** use these macros for doing your COPY and SMOOTH operations.

These are all defined in defs.h.

Cache Structure:

The cache we will be simulating is a 16 KB direct-mapped cache, with 32 byte cache lines. You may wish to refer back to the notes to determine how best to optimize for such a configuration.

Rotate:

The following C function takes a source image, src, of size dim x dim and puts a rotated copy into the destination image dst.

```
void rotate_naive(int dim, pixel* src, pixel* dst) {
            int i, j;
            for(i=0; i<dim; i++) {
                        for(j=0; j<dim; j++) {
                                    COPY(&dst[PIXEL(dim-1-j,i,dim)],&src[PIXEL(i,j,dim)]);
                        }
            }
            return;
}
```

This code traverses the rows of the source image, copying each into a column of the destination image. Your task is to try to maximize the number of cache hits by adjusting the algorithm to take advantage of the cache.

Smooth:

The following C function takes a source image of size dim x dim that is specified by src, and puts a 'smoothed' copy into the destination image dst. The actual smoothing is done by the SMOOTH macro, which takes in first the address of the destination pixel, and then the addresses of the source pixel and the 8 pixels surrounding it. For cases on the border of the image, COPY the pixel straight from the source to the destination, so as to not have to deal with the special case of not having 8 surrounding pixels.

```
void smooth_naive(int dim, pixel *src, pixel *dst) {
        int i, j;
        for(i=0; i<dim;i++) {
                COPY(&dst[PIXEL(i,0,dim)], &src[PIXEL(i,0,dim)]);
                COPY(&dst[PIXEL(i,dim-1,dim)], &src[PIXEL(i,dim-1,dim)]);
        }
        for(j=1; j<dim-1;j++) {
                COPY(&dst[PIXEL(0,j,dim)], &src[PIXEL(0,j,dim)]);
                COPY(&dst[PIXEL(dim-1,j,dim)], &src[PIXEL(dim-1,j,dim)]);
        }
        for(i=1; i<dim-1; i++) {
                for(j=1; j<dim-1; j++) {
                        SMOOTH(&dst[PIXEL(j,i,dim)],
                                        &src[PIXEL(j,i,dim)],
                                        &src[PIXEL(j-1,i,dim)],
                                        &src[PIXEL(j+1,i,dim)],
                                        &src[PIXEL(j,i+1,dim)],
                                        &src[PIXEL(j,i-1,dim)],
                                        &src[PIXEL(j-1,i-1,dim)],
                                        &src[PIXEL(j+1,i+1,dim)],
                                        &src[PIXEL(j-1,i+1,dim)],
                                        &src[PIXEL(j+1,i-1,dim)]);
                }
        }
        return;
}
```

The code first takes care of the edge cases and does a straight copy for the border. It then traverses the image in standard fashion, smoothing each pixel as it comes. As with rotate, your goal is to maximize cache hitrate by improving locality.

Evaluation:

The improved algorithms you submit will be graded based on the cache simulator included in the zip file you downloaded earlier. Functions will be run on images of a number of different sizes (listed below), and for each size will be given a hitrate equal to the total number of cache hits divided by the number of cache attempts in the image. (Higher numbers are better.) A function's 'hit score' will be determined by taking the geometric mean (explained below) of the ratios produced by dividing your function's hitrate by the naive implementation's hitrate.

For both rotate and smooth, a geometric mean of 5 numbers is computed by taking the 5th root of the product of those numbers, so for the five dimensions listed below the formula would be:

$$\textbf{hit score} = (\text{ratio}_{64} * \text{ratio}_{128} * \text{ratio}_{256} * \text{ratio}_{512} * \text{ratio}_{1024})^{1/5}$$

Assumptions:

To make optimization easier, you may assume that the image dimensions will always be a multiple of 32. Your code must be able to correctly rotate for all dimensions that are multiples of 32, but your performance scores will be determined based solely upon the values listed below:

64	128	256	512	1024

SETUP

Versioning

The rotate.c and smooth.c that you unzip contain only two functions each: the original naive implementation of their respective function, and a "register" function. This function provides an easy way to compare multiple functions at the same time, and is called by the driver program before testing.

```
void register_rotate_functions() {
        add_rotate_function(&rotate, rotate_descr);
}
```

The 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 rotate.c to see how to create the string descriptions? The string can be at most 256 characters. The functions for smooth work analogously.

Testing

To test your functions, simply open the project file in Visual C++. Choosing Build, Execute will show the output of the driver program. (This is the same input that will be used for grading your submissions.).

GRADING

This is how grading for the exercise will work:

- Correctness: Your solutions must be 100% correct for any square image matrix with edge dimensions that are a multiple of 32. (The driver program will check correctness and will tell you if a particular implementation is incorrect.
- Speed improvement: Your solutions will earn credit based on reaching a certain threshold, according to the tables below:

Rotate:

Hit Score:	Credit:
1.60	70/70
1.45	60/70

Smooth:

Hit Score:	Credit:
1.30	30/30
1.25	25/30

1.30	55/70
1.25	40/70
1.10	25/70

1.20	20/30
1.15	15/30
1.10	10/30

HINTS

- The rotate function focuses on spatial locality: because each pixel is used only once, you should focus on using any pixels put into the cache by a previous pixel operation.
- The smooth function benefits from spatial locality, but also reuses pixels it has read previously. Consequently, you should consider trying to improve temporal locality as well.
- Try a large number of different functions. There is a FIND_BEST_OF_MANY #define flag in driver.c that can be used to find out which function provides the highest hit rate for each problem.
- Just because your image is square doesn't mean you have to deal with the image in square pieces.
- Remember the way things will be laid out in memory and how this affects what is put into the cache.