暨南大學
JINAN UNIVERSITY

Undergraduate Lab Report

Course Title: Experiment of Computer Organization

Course No: 60080014

Student Name: ____________

Student No: ____________

School: International School

Department: ____________

Major: Computer Science & Technology

Instructor：SUN Heng

Academic Year:202~202,Semester : 1st [√]   2nd[   ]

Academic Affairs Office of Jinan University

Date (dd/mm/yyyy)____________

Computer Organization Lab List

Student Name:________________ Student No:___________________

ID	Lab Name	Type
1	Number Storage Lab	Individual
2	Manipulating Bits	Individual
3	Simulating Y86-64 Program	Individual
4	Performance Lab	Team
5	A Simple Real-life Control System	Team
6	System I/O	Individual

Undergraduate Lab Report of Jinan University

Course Title__Experiment of Computer Organization__Evaluation______
Lab Name_______Performance Lab________Instructor___SUN Heng_
Lab Address____
Student Name________________________Student No____________
College_________________International School_________________
Department________________________Major_______CST_________
Date________/________/________ Afternoon

1. Introduction

The assembly language in Lab 3 is one of the most effective means for gaining an understanding how the code will run in high performance. In this lab, students should optimize the performance of an application kernel function such as convolution or matrix transposition. It provides a clear demonstration of the properties of cache memories and gives them experience with low-level program optimization.

2. Lab Instructions or Steps

2.1 Image processing problem

This lab 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.

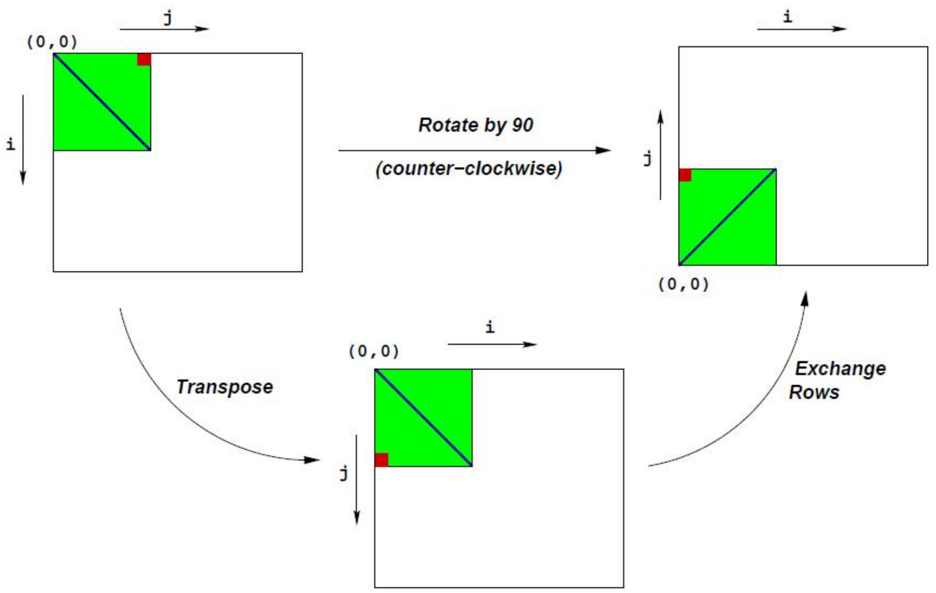

Figure 1 Rotation of an image by 90° counterclockwise

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:

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

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

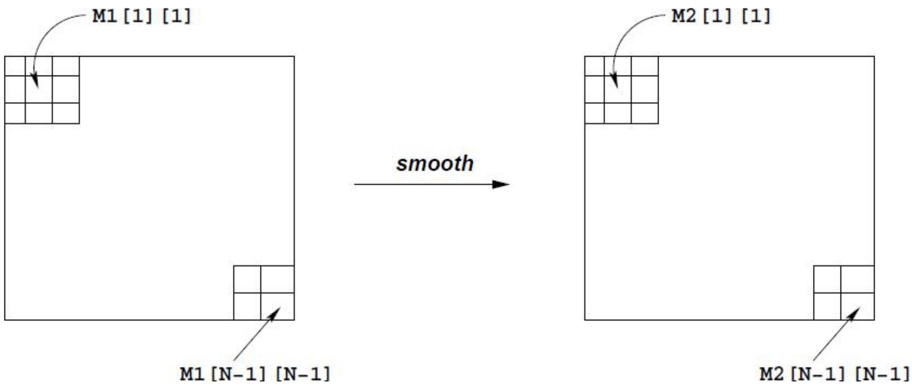

Figure 2 Smoothing an image

2.2 Source code

The only file you will be modifying is *kernels.c*. 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.

2.3 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.

2.4 Rotate operation

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 discuss the reasons of performance improvement from the functions *rotate_c*、*rotate_b4*、*rotate_b8*、*rotate_b8_u4*、*rotate_b8_u4_c*、*rotate_b16_u2*、*rotate_b16_u4*、*rotate_b16_u4_c*、*rotate_b32_u2*、*rotate_b32_u4*、*rotate_b32_u4_c.*

See the file *kernels.c* for this code.

2.5 Smooth operation

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.

This code (and an implementation of *avg*) is in the file *kernels.c*.

Your task is to discuss the reasons of performance improvement.

2.6 Performance measures

Our main performance measure is *CPE* (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. Performance is shown for 5 different values of N.

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

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

2.7 Running

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

3. Lab Device or Environment

Ubuntu 16.04 (64-bit) with AMD Ryzen 9 5900HS CPU @ 3.30GHz and 4GB memory on virtual machine (Oracle VM VirtualBox)

4. Results and Analysis

Rotate operation

Results and analysis:

rotate_c: This version is an improvement over naive. We know that the rotate function involves src reads and dst writes, and rotate_c makes the writes sequential in memory, so optimizations for write hits are better than optimizations for read hits.

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.7     3.0     6.3     9.5
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.7     23.5    15.3    10.4    9.9     12.9

Rotate: Version = rotate_c: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.6     1.7     3.1     3.6     7.9
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.4     23.3    15.0    18.2    12.0    14.8

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       8.9     8.6     8.3     8.4     8.4
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         78.4    81.3    84.1    85.4    86.2    83.0

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       29.8    30.4    30.7    30.1    30.9
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         23.3    23.0    22.9    23.8    23.4    23.3

Summary of Your Best Scores:
  Rotate: 14.8 (rotate_c: Current working version)
  Smooth: 83.0 (smooth: Current working version)
```

rotate_b4: In this version, the matrix to be processed is divided into 4 x 4 blocks, improving the spatial locality of the program and achieving some performance improvement.

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.7     3.0     6.1     9.5
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.8     24.2    15.3    10.8    9.9     13.1

Rotate: Version = rotate_b4: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       2.4     1.9     2.0     2.1     3.2
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         6.2     21.6    22.7    30.8    29.6    19.4

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       9.1     8.7     8.5     8.6     8.4
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         76.7    80.0    82.4    83.4    86.4    81.7

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       29.8    30.2    30.3    31.6    31.6
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         23.3    23.1    23.1    22.7    22.8    23.0

Summary of Your Best Scores:
  Rotate: 19.4 (rotate_b4: Current working version)
  Smooth: 81.7 (smooth: Current working version)
```

rotate_b8: This version divides the matrix into larger chunks (8 x 8) than the previous b4 version, and you can see that performance can continue to improve.

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.7     3.0     6.4     9.8
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.9     24.0    15.3    10.3    9.6     12.9

Rotate: Version = rotate_b8: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.7     1.7     1.9     1.9     2.6
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.8     23.8    24.1    34.8    35.9    22.9

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       8.9     8.7     8.4     8.4     8.4
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         78.1    80.4    83.4    85.3    86.0    82.6

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       30.1    30.6    30.1    30.4    30.9
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         23.1    22.8    23.4    23.6    23.4    23.2

Summary of Your Best Scores:
  Rotate: 22.9 (rotate_b8: Current working version)
  Smooth: 82.6 (smooth: Current working version)
```

rotate_b8_u4: This version uses four-way loop unrolling based on the b8 chunking, but this may have hit a throughput bottleneck for the b8 version, with only a slight improvement.

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.7     3.1     6.3     9.5
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.9     24.0    15.1    10.5    10.0    13.0

Rotate: Version = rotate_b8_u4: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.7     1.7     1.8     1.9     2.7
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.9     23.3    25.1    34.7    35.5    23.0

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       9.0     8.6     8.6     8.4     8.4
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         77.6    81.6    82.0    85.0    85.9    82.4

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       30.4    31.1    30.9    30.8    31.4
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         22.9    22.4    22.7    23.2    23.0    22.8

Summary of Your Best Scores:
  Rotate: 23.0 (rotate_b8_u4: Current working version)
  Smooth: 82.4 (smooth: Current working version)
```

rotate_b8_u4_c: As with the previous version, the b8 loop unrolling reaches a certain performance bottleneck, even causing the performance of this version to be lower than the previous version after swapping row and column order.

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.7     3.1     6.2     9.7
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.8     23.9    14.9    10.6    9.8     12.9

Rotate: Version = rotate_b8_u4_c: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.7     1.8     1.9     2.0     2.6
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.4     22.6    23.9    32.3    36.3    22.2

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       9.0     8.6     8.4     8.5     8.5
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         77.6    81.5    83.3    84.7    84.9    82.4

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       31.7    31.9    31.6    31.5    31.0
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         21.9    21.9    22.2    22.8    23.3    22.4

Summary of Your Best Scores:
  Rotate: 22.2 (rotate_b8_u4_c: Current working version)
  Smooth: 82.4 (smooth: Current working version)
```

rotate_b16_u2: As with the previous version, the b8 loop unrolling reaches a certain performance bottleneck, even causing the performance of this version to be lower than the previous version after swapping row and column order.

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.7     3.1     6.3     9.6
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.7     24.1    15.0    10.4    9.8     12.9

Rotate: Version = rotate_b16_u2: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.6     1.6     1.7     1.7     3.3
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.3     25.3    26.7    38.5    28.9    23.3

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       9.0     8.6     8.5     8.4     8.5
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         77.3    80.9    82.6    84.9    84.7    82.1

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       32.3    32.3    32.1    31.6    32.6
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         21.5    21.6    21.9    22.7    22.1    22.0

Summary of Your Best Scores:
  Rotate: 23.3 (rotate_b16_u2: Current working version)
  Smooth: 82.1 (smooth: Current working version)
```

rotate_b16_u4: With four-way loop unrolling, it is possible to squeeze a little more performance out of b16_u2, but not much.

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.7     3.1     6.4     9.6
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.7     23.8    14.9    10.3    9.8     12.8

Rotate: Version = rotate_b16_u4: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.6     1.6     1.7     1.7     2.8
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.0     24.4    26.9    37.9    33.6    23.8

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       9.0     8.6     8.5     8.4     8.5
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         77.0    80.8    82.1    85.0    84.9    81.9

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       30.8    31.3    31.6    31.7    31.4
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         22.6    22.3    22.2    22.7    23.0    22.5

Summary of Your Best Scores:
  Rotate: 23.8 (rotate_b16_u4: Current working version)
  Smooth: 81.9 (smooth: Current working version)
```

rotate_b16_u4_c: As with the previous version, the b16 loop unrolling reaches a certain performance bottleneck, even exchanging the row and column sequence can' t make the program reach a higher performance.

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.7     3.1     6.4     9.7
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.8     24.0    14.9    10.2    9.7     12.8

Rotate: Version = rotate_b16_u4_c: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.6     1.6     1.8     1.8     2.6
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.2     24.9    26.1    35.7    36.6    23.9

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       9.2     8.7     8.5     8.5     8.6
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         75.1    80.3    82.9    84.7    83.8    81.3

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       31.1    31.4    31.7    31.2    30.4
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         22.3    22.2    22.1    23.0    23.8    22.7

Summary of Your Best Scores:
  Rotate: 23.9 (rotate_b16_u4_c: Current working version)
  Smooth: 81.3 (smooth: Current working version)
```

rotate_b32_u2: This version continues to divide the matrix into larger chunks (32 x 32) with a double-loop expansion, but does not get much of a boost, and is roughly on par with the performance of b16_u2.

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.7     3.1     6.3     9.7
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.8     23.9    15.2    10.5    9.7     12.9

Rotate: Version = rotate_b32_u2: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.6     1.6     1.6     1.7     4.2
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.5     25.9    28.5    37.9    22.7    22.7

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       9.0     8.6     8.5     8.5     8.7
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         77.3    81.0    82.9    84.4    83.1    81.7

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       30.9    31.8    32.1    31.4    32.0
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         22.5    22.0    21.8    22.8    22.5    22.3

Summary of Your Best Scores:
  Rotate: 22.7 (rotate_b32_u2: Current working version)
  Smooth: 81.7 (smooth: Current working version)
```

rotate_b32_u4: Four-way loop unrolling version compared with b32_u2, we can find that the b32 version can get a significant performance improvement when more loops are unrolled.

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.8     3.4     6.3     9.7
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.7     21.8    13.7    10.4    9.7     12.4

Rotate: Version = rotate_b32_u4: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.6     1.6     1.7     1.7     2.9
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.4     25.4    27.3    38.5    33.0    24.2

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       9.2     8.7     8.5     8.5     8.6
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         75.1    80.7    82.8    84.6    83.7    81.3

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       32.0    32.4    32.4    31.5    31.9
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         21.7    21.6    21.7    22.7    22.7    22.1

Summary of Your Best Scores:
  Rotate: 24.2 (rotate_b32_u4: Current working version)
  Smooth: 81.3 (smooth: Current working version)
```

rotate_b32_u4_c: By swapping the rows and rows of version b32_u4, you can see that the CPE can be improved by about 1.

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.7     3.1     6.4     9.7
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.8     23.8    15.2    10.4    9.7     12.9

Rotate: Version = rotate_b32_u4_c: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.6     1.6     1.7     1.7     2.3
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.4     25.5    27.8    37.9    41.7    25.4

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       9.2     8.7     8.5     8.7     8.5
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         75.7    79.8    82.2    82.0    84.5    80.8

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       33.1    32.8    32.7    31.8    31.0
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         21.0    21.3    21.5    22.6    23.3    21.9

Summary of Your Best Scores:
  Rotate: 25.4 (rotate_b32_u4_c: Current working version)
  Smooth: 80.8 (smooth: Current working version)
```

Smooth operation

Result:

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.7     3.0     6.3     9.5
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.7     23.5    15.3    10.4    9.9     12.9

Rotate: Version = rotate_c: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.6     1.7     3.1     3.6     7.9
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.4     23.3    15.0    18.2    12.0    14.8

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       8.9     8.6     8.3     8.4     8.4
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         78.4    81.3    84.1    85.4    86.2    83.0

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       29.8    30.4    30.7    30.1    30.9
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         23.3    23.0    22.9    23.8    23.4    23.3

Summary of Your Best Scores:
  Rotate: 14.8 (rotate_c: Current working version)
  Smooth: 83.0 (smooth: Current working version)
```

Analysis:

(1) In the original version, **max** and **min** functions need to be called every time the value of a certain pixel is calculated to determine the boundary. In fact, this step is unnecessary for other points except the boundary part, resulting in huge performance overhead. Therefore, it is considered to calculate the edge, corner and internal pixel points respectively.

(2) In the original version, calculating the value of a pixel needs to call multiple functions for calculation, and during the calculation process needs to assign values in multiple temporary variables, **data transfer** and **function call** cost a lot of performance, so consider using the entire pixel calculation process is encapsulated into a single function.

(3) In the process of encapsulating the pixel point calculation function, the loop that calculates the corner (4 points), edge (6 points), and interior (9 points) is directly **unrolled**, and a good performance

improvement can be obtained.

(4) Originally, the function return value was used to assign dst pixels. However, it was found that passing the pointer of dst into the function and **directly assigning the value in the function** can improve certain performance. Therefore, this method was used to complete the encapsulation of the pixel point calculation function.

(5) Since most points in an image are not edges and corners, the operation of **calculating multiple points** in a single loop can greatly improve program performance. At the same time, because an internal point requires the RGB values of the surrounding 8 points, the calculation of adjacent points can **reuse the values that have been obtained around** at the same time, so I abandoned the original internal pixel calculation function and re-written an internal pixel calculation function.

(6) The function finally written uses a lot of loop unrolling, and calculates the values of two adjacent rows in the same column at the same time, and calculates 4 columns at a time, and finally uses a supplementary loop to process the unprocessed values of step 4 loop, finally a **2 x 4 loop unrolling** is implemented, but the code is very poorly readable.

5. Appendix (Program Code)

```c
#include <stdio.h>
#include <stdlib.h>

#include "defs.h"

/*
 * Please fill in the following team struct
 */
team_t team = {
    "", /* Team name */

    "H3Art",            /* First member full name */
    "", /* First member email address */

    "", /* Second or third member full name (leave blank if none) */
    ""              /* Second or third member email addr (leave blank if none) */
};

/***************
 * ROTATE KERNEL
 ***************/

/******************************************************
 * Your different versions of the rotate kernel go here
 ******************************************************/

/*
 * naive_rotate - The naive baseline version of rotate
 */
char naive_rotate_descr[] = "naive_rotate: Naive baseline implementation";
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)];
        }
    }
}

```

```
/*
 * rotate - Your current working version of rotate
 * IMPORTANT: This is the version you will be graded on
 */

/*
 * Exchange the loop sequence of row and column
 */
void rotate_c(int dim, pixel *src, pixel *dst) {
    int i, j;

    for (j = 0; j < dim; j++) {
        for (i = 0; i < dim; i++) {
            dst[RIDX(dim - 1 - j, i, dim)] = src[RIDX(i, j, dim)];
        }
    }
}

/*
 * Partition the matrix into 4 x 4 blocks
 */
void rotate_b4(int dim, pixel *src, pixel *dst) {
    int i, j, k, s;

    for (i = 0; i < dim; i += 4) {
        for (j = 0; j < dim; j += 4) {
            for (k = i; k < i + 4; k++) {
                for (s = j; s < j + 4; s++) {
                    dst[RIDX(dim - 1 - s, k, dim)] = src[RIDX(k, s, dim)];
                }
            }
        }
    }
}

/*
 * Partition the matrix into 8 x 8 blocks
 */
void rotate_b8(int dim, pixel *src, pixel *dst) {
    int i, j, k, s;

    for (i = 0; i < dim; i += 8) {
        for (j = 0; j < dim; j += 8) {
```

```
            for (k = i; k < i + 8; k++) {
                for (s = j; s < j + 8; s++) {
                    dst[RIDX(dim - 1 - s, k, dim)] = src[RIDX(k, s, dim)];
                }
            }
        }
    }
}

/*
 * Partition the matrix into 8 x 8 blocks
 * Unroll the loop with a step size of 4
 */
void rotate_b8_u4(int dim, pixel *src, pixel *dst) {
    int i, j, k, s;

    for (i = 0; i < dim; i += 8) {
        for (j = 0; j < dim; j += 8) {
            for (k = i; k < i + 8; k += 4) {
                for (s = j; s < j + 8; s++) {
                    dst[RIDX(dim - 1 - s, k, dim)] = src[RIDX(k, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 1, dim)] =
                        src[RIDX(k + 1, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 2, dim)] =
                        src[RIDX(k + 2, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 3, dim)] =
                        src[RIDX(k + 3, s, dim)];
                }
            }
        }
    }
}

/*
 * Exchange the loop sequence of row and column
 * Partition the matrix into 8 x 8 blocks
 * Unroll the loop with a step size of 4
 */
void rotate_b8_u4_c(int dim, pixel *src, pixel *dst) {
    int i, j, k, s;

    for (j = 0; j < dim; j += 8) {
```

```
            for (i = 0; i < dim; i += 8) {
                for (k = i; k < i + 8; k += 4) {
                    for (s = j; s < j + 8; s++) {
                        dst[RIDX(dim - 1 - s, k, dim)] = src[RIDX(k,
    s, dim)];
                        dst[RIDX(dim - 1 - s, k + 1, dim)] =
                            src[RIDX(k + 1, s, dim)];
                        dst[RIDX(dim - 1 - s, k + 2, dim)] =
                            src[RIDX(k + 2, s, dim)];
                        dst[RIDX(dim - 1 - s, k + 3, dim)] =
                            src[RIDX(k + 3, s, dim)];
                    }
                }
            }
        }
    }

    /*
     * Partition the matrix into 16 x 16 blocks
     * Unroll the loop with a step size of 2
     */
    void rotate_b16_u2(int dim, pixel *src, pixel *dst) {
        int i, j, k, s;

        for (i = 0; i < dim; i += 16) {
            for (j = 0; j < dim; j += 16) {
                for (k = i; k < i + 16; k += 2) {
                    for (s = j; s < j + 16; s++) {
                        dst[RIDX(dim - 1 - s, k, dim)] = src[RIDX(k,
    s, dim)];
                        dst[RIDX(dim - 1 - s, k + 1, dim)] =
                            src[RIDX(k + 1, s, dim)];
                    }
                }
            }
        }
    }

    /*
     * Partition the matrix into 16 x 16 blocks
     * Unroll the loop with a step size of 4
     */
    void rotate_b16_u4(int dim, pixel *src, pixel *dst) {
        int i, j, k, s;
```

```

    for (i = 0; i < dim; i += 16) {
        for (j = 0; j < dim; j += 16) {
            for (k = i; k < i + 16; k += 4) {
                for (s = j; s < j + 16; s++) {
                    dst[RIDX(dim - 1 - s, k, dim)] = src[RIDX(k, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 1, dim)] =
                        src[RIDX(k + 1, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 2, dim)] =
                        src[RIDX(k + 2, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 3, dim)] =
                        src[RIDX(k + 3, s, dim)];
                }
            }
        }
    }
}

/*
 * Exchange the loop sequence of row and column
 * Partition the matrix into 16 x 16 blocks
 * Unroll the loop with a step size of 4
 */
void rotate_b16_u4_c(int dim, pixel *src, pixel *dst) {
    int i, j, k, s;

    for (j = 0; j < dim; j += 16) {
        for (i = 0; i < dim; i += 16) {
            for (k = i; k < i + 16; k += 4) {
                for (s = j; s < j + 16; s++) {
                    dst[RIDX(dim - 1 - s, k, dim)] = src[RIDX(k, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 1, dim)] =
                        src[RIDX(k + 1, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 2, dim)] =
                        src[RIDX(k + 2, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 3, dim)] =
                        src[RIDX(k + 3, s, dim)];
                }
            }
        }
    }
}
```

```
/*
 * Partition the matrix into 32 x 32 blocks
 * Unroll the loop with a step size of 2
 */
void rotate_b32_u2(int dim, pixel *src, pixel *dst) {
    int i, j, k, s;

    for (i = 0; i < dim; i += 32) {
        for (j = 0; j < dim; j += 32) {
            for (k = i; k < i + 32; k += 2) {
                for (s = j; s < j + 32; s++) {
                    dst[RIDX(dim - 1 - s, k, dim)] = src[RIDX(k, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 1, dim)] =
                        src[RIDX(k + 1, s, dim)];
                }
            }
        }
    }
}

/*
 * Partition the matrix into 32 x 32 blocks
 * Unroll the loop with a step size of 4
 */
void rotate_b32_u4(int dim, pixel *src, pixel *dst) {
    int i, j, k, s;

    for (i = 0; i < dim; i += 32) {
        for (j = 0; j < dim; j += 32) {
            for (k = i; k < i + 32; k += 4) {
                for (s = j; s < j + 32; s++) {
                    dst[RIDX(dim - 1 - s, k, dim)] = src[RIDX(k, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 1, dim)] =
                        src[RIDX(k + 1, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 2, dim)] =
                        src[RIDX(k + 2, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 3, dim)] =
                        src[RIDX(k + 3, s, dim)];
                }
            }
        }
```

```
    }
}

/*
 * Exchange the loop sequence of row and column
 * Partition the matrix into 32 x 32 blocks
 * Unroll the loop with a step size of 4
 */
void rotate_b32_u4_c(int dim, pixel *src, pixel *dst) {
    int i, j, k, s;

    for (j = 0; j < dim; j += 32) {
        for (i = 0; i < dim; i += 32) {
            for (k = i; k < i + 32; k += 4) {
                for (s = j; s < j + 32; s++) {
                    dst[RIDX(dim - 1 - s, k, dim)] = src[RIDX(k, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 1, dim)] =
                        src[RIDX(k + 1, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 2, dim)] =
                        src[RIDX(k + 2, s, dim)];
                    dst[RIDX(dim - 1 - s, k + 3, dim)] =
                        src[RIDX(k + 3, s, dim)];
                }
            }
        }
    }
}

char rotate_descr[] = "rotate_b32_u4_c: Current working version";
void rotate(int dim, pixel *src, pixel *dst) { rotate_b32_u4_c(dim, src, dst); }

/*******************************************************************
 * register_rotate_functions - Register all of your different versions
 *     of the rotate kernel with the driver by calling the
 *     add_rotate_function() for each test function. When you run the
 *     driver program, it will test and report the performance of each
 *     registered test function.
```

```
 ************************************************************************/

void register_rotate_functions() {
    add_rotate_function(&naive_rotate, naive_rotate_descr);
    add_rotate_function(&rotate, rotate_descr);
    /* ... Register additional test functions here */
}

/***************
 * SMOOTH KERNEL
 **************/

/******************************************************************
 * Various typedefs and helper functions for the smooth function
 * You may modify these any way you like.
 *****************************************************************/

/* A struct used to compute averaged pixel value */
typedef struct {
    int red;
    int green;
    int blue;
    int num;
} pixel_sum;

/* Compute min and max of two integers, respectively */
static int min(int a, int b) { return (a < b ? a : b); }
static int max(int a, int b) { return (a > b ? a : b); }

/*
 * initialize_pixel_sum - Initializes all fields of sum to 0
 */
static void initialize_pixel_sum(pixel_sum *sum) {
    sum->red = sum->green = sum->blue = 0;
    sum->num = 0;
    return;
}

/*
 * accumulate_sum - Accumulates field values of p in corresponding
 * fields of sum
 */
```

```c
static void accumulate_sum(pixel_sum *sum, pixel p) {
    sum->red += (int)p.red;
    sum->green += (int)p.green;
    sum->blue += (int)p.blue;
    sum->num++;
    return;
}

/*
 * assign_sum_to_pixel - Computes averaged pixel value in current_pixel
 */
static void assign_sum_to_pixel(pixel *current_pixel, pixel_sum sum) {
    current_pixel->red = (unsigned short)(sum.red / sum.num);
    current_pixel->green = (unsigned short)(sum.green / sum.num);
    current_pixel->blue = (unsigned short)(sum.blue / sum.num);
    return;
}

/*
 * avg - Returns averaged pixel value at (i,j)
 */
static pixel avg(int dim, int i, int j, pixel *src) {
    int ii, jj;
    pixel_sum sum;
    pixel current_pixel;

    initialize_pixel_sum(&sum);
    for (ii = max(i - 1, 0); ii <= min(i + 1, dim - 1); ii++)
        for (jj = max(j - 1, 0); jj <= min(j + 1, dim - 1); jj++)
            accumulate_sum(&sum, src[RIDX(ii, jj, dim)]);

    assign_sum_to_pixel(&current_pixel, sum);
    return current_pixel;
}

/******************************************************
 * Your different versions of the smooth kernel go here
 ******************************************************/

/*
```

```
 * naive_smooth - The naive baseline version of smooth
 */
char naive_smooth_descr[] = "naive_smooth: Naive baseline 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 - Your current working version of smooth.
 * IMPORTANT: This is the version you will be graded on
 */

/* Calculate angle block (0, 0) */
static void avg_left_up(int dim, int row, int col, pixel *dst, pixel *src) {
    dst[RIDX(row, col, dim)].red =
        (src[RIDX(row, col, dim)].red + src[RIDX(row, col + 1, dim)].red +
         src[RIDX(row + 1, col, dim)].red +
         src[RIDX(row + 1, col + 1, dim)].red) >>
        2;
    dst[RIDX(row, col, dim)].blue =
        (src[RIDX(row, col, dim)].blue + src[RIDX(row, col + 1, dim)].blue +
         src[RIDX(row + 1, col, dim)].blue +
         src[RIDX(row + 1, col + 1, dim)].blue) >>
        2;
    dst[RIDX(row, col, dim)].green =
        (src[RIDX(row, col, dim)].green + src[RIDX(row, col + 1, dim)].green +
         src[RIDX(row + 1, col, dim)].green +
         src[RIDX(row + 1, col + 1, dim)].green) >>
        2;
}

/* Calculate angle block(0, dim - 1) */
static void avg_right_up(int dim, int row, int col, pixel *dst, pixel *src) {
    dst[RIDX(row, col, dim)].red =
```

```
            (src[RIDX(row, col - 1, dim)].red + src[RIDX(row, col, dim)].red +
             src[RIDX(row + 1, col - 1, dim)].red +
             src[RIDX(row + 1, col, dim)].red) >>
            2;
        dst[RIDX(row, col, dim)].blue =
            (src[RIDX(row, col - 1, dim)].blue + src[RIDX(row, col, dim)].blue +
             src[RIDX(row + 1, col - 1, dim)].blue +
             src[RIDX(row + 1, col, dim)].blue) >>
            2;
        dst[RIDX(row, col, dim)].green =
            (src[RIDX(row, col - 1, dim)].green + src[RIDX(row, col, dim)].green +
             src[RIDX(row + 1, col - 1, dim)].green +
             src[RIDX(row + 1, col, dim)].green) >>
            2;
    }

    /* Calculate angle block(dim - 1, 0) */
    static void avg_left_down(int dim, int row, int col, pixel *dst, pixel *src) {
        dst[RIDX(row, col, dim)].red =
            (src[RIDX(row - 1, col, dim)].red +
             src[RIDX(row - 1, col + 1, dim)].red + src[RIDX(row, col, dim)].red +
             src[RIDX(row, col + 1, dim)].red) >>
            2;
        dst[RIDX(row, col, dim)].blue =
            (src[RIDX(row - 1, col, dim)].blue +
             src[RIDX(row - 1, col + 1, dim)].blue + src[RIDX(row, col, dim)].blue +
             src[RIDX(row, col + 1, dim)].blue) >>
            2;
        dst[RIDX(row, col, dim)].green =
            (src[RIDX(row - 1, col, dim)].green +
             src[RIDX(row - 1, col + 1, dim)].green +
             src[RIDX(row, col, dim)].green + src[RIDX(row, col + 1, dim)].green) >>
            2;
    }

    /* Calculate angle block(dim - 1, dim - 1) */
```

```
static void avg_right_down(int dim, int row, int col, pixel *dst, pixel *src) {
    dst[RIDX(row, col, dim)].red =
        (src[RIDX(row - 1, col - 1, dim)].red +
         src[RIDX(row - 1, col, dim)].red + src[RIDX(row, col - 1, dim)].red +
         src[RIDX(row, col, dim)].red) >>
        2;
    dst[RIDX(row, col, dim)].blue =
        (src[RIDX(row - 1, col - 1, dim)].blue +
         src[RIDX(row - 1, col, dim)].blue + src[RIDX(row, col - 1, dim)].blue +
         src[RIDX(row, col, dim)].blue) >>
        2;
    dst[RIDX(row, col, dim)].green =
        (src[RIDX(row - 1, col - 1, dim)].green +
         src[RIDX(row - 1, col, dim)].green +
         src[RIDX(row, col - 1, dim)].green + src[RIDX(row, col, dim)].green) >>
        2;
}

/* Calculate edge blocks(0, x) */
static void avg_up(int dim, int row, int col, pixel *dst, pixel *src) {
    dst[RIDX(row, col, dim)].red =
        (src[RIDX(row, col - 1, dim)].red + src[RIDX(row, col, dim)].red +
         src[RIDX(row, col + 1, dim)].red +
         src[RIDX(row + 1, col - 1, dim)].red +
         src[RIDX(row + 1, col, dim)].red +
         src[RIDX(row + 1, col + 1, dim)].red) /
        6;
    dst[RIDX(row, col, dim)].blue =
        (src[RIDX(row, col - 1, dim)].blue + src[RIDX(row, col, dim)].blue +
         src[RIDX(row, col + 1, dim)].blue +
         src[RIDX(row + 1, col - 1, dim)].blue +
         src[RIDX(row + 1, col, dim)].blue +
         src[RIDX(row + 1, col + 1, dim)].blue) /
        6;
    dst[RIDX(row, col, dim)].green =
        (src[RIDX(row, col - 1, dim)].green + src[RIDX(row, col, dim)].green +
```

```
         src[RIDX(row, col + 1, dim)].green +
         src[RIDX(row + 1, col - 1, dim)].green +
         src[RIDX(row + 1, col, dim)].green +
         src[RIDX(row + 1, col + 1, dim)].green) /
        6;
}

/* Calculate edge blocks(dim - 1, x) */
static void avg_down(int dim, int row, int col, pixel *dst, pixel *src) {
    dst[RIDX(row, col, dim)].red =
        (src[RIDX(row - 1, col - 1, dim)].red +
         src[RIDX(row - 1, col, dim)].red +
         src[RIDX(row - 1, col + 1, dim)].red +
         src[RIDX(row, col - 1, dim)].red + src[RIDX(row, col, dim)].red +
         src[RIDX(row, col + 1, dim)].red) /
        6;
    dst[RIDX(row, col, dim)].blue =
        (src[RIDX(row - 1, col - 1, dim)].blue +
         src[RIDX(row - 1, col, dim)].blue +
         src[RIDX(row - 1, col + 1, dim)].blue +
         src[RIDX(row, col - 1, dim)].blue + src[RIDX(row, col, dim)].blue +
         src[RIDX(row, col + 1, dim)].blue) /
        6;
    dst[RIDX(row, col, dim)].green =
        (src[RIDX(row - 1, col - 1, dim)].green +
         src[RIDX(row - 1, col, dim)].green +
         src[RIDX(row - 1, col + 1, dim)].green +
         src[RIDX(row, col - 1, dim)].green + src[RIDX(row, col, dim)].green +
         src[RIDX(row, col + 1, dim)].green) /
        6;
}

/* Calculate edge blocks(x, 0) */
static void avg_left(int dim, int row, int col, pixel *dst, pixel *src) {
    dst[RIDX(row, col, dim)].red =
        (src[RIDX(row - 1, col, dim)].red +
         src[RIDX(row - 1, col + 1, dim)].red + src[RIDX(row, col, dim)].red +
```

```
                src[RIDX(row, col + 1, dim)].red + src[RIDX(row + 1, col, dim)].red +
                src[RIDX(row + 1, col + 1, dim)].red) /
               6;
        dst[RIDX(row, col, dim)].blue =
               (src[RIDX(row - 1, col, dim)].blue +
                src[RIDX(row - 1, col + 1, dim)].blue + src[RIDX(row, col, dim)].blue +
                src[RIDX(row, col + 1, dim)].blue + src[RIDX(row + 1, col, dim)].blue +
                src[RIDX(row + 1, col + 1, dim)].blue) /
               6;
        dst[RIDX(row, col, dim)].green =
               (src[RIDX(row - 1, col, dim)].green +
                src[RIDX(row - 1, col + 1, dim)].green +
                src[RIDX(row, col, dim)].green + src[RIDX(row, col + 1, dim)].green +
                src[RIDX(row + 1, col, dim)].green +
                src[RIDX(row + 1, col + 1, dim)].green) /
               6;
    }

    /* Calculate edge blocks(x, dim - 1) */
    static void avg_right(int dim, int row, int col, pixel *dst, pixel *src) {
        dst[RIDX(row, col, dim)].red =
               (src[RIDX(row - 1, col - 1, dim)].red +
                src[RIDX(row - 1, col, dim)].red + src[RIDX(row, col - 1, dim)].red +
                src[RIDX(row, col, dim)].red + src[RIDX(row + 1, col - 1, dim)].red +
                src[RIDX(row + 1, col, dim)].red) /
               6;
        dst[RIDX(row, col, dim)].blue =
               (src[RIDX(row - 1, col - 1, dim)].blue +
                src[RIDX(row - 1, col, dim)].blue + src[RIDX(row, col - 1, dim)].blue +
                src[RIDX(row, col, dim)].blue + src[RIDX(row + 1, col - 1, dim)].blue +
                src[RIDX(row + 1, col, dim)].blue) /
               6;
        dst[RIDX(row, col, dim)].green =
               (src[RIDX(row - 1, col - 1, dim)].green +
                src[RIDX(row - 1, col, dim)].green +
```

```
         src[RIDX(row, col - 1, dim)].green + src[RIDX(row, col, dim)].green +
         src[RIDX(row + 1, col - 1, dim)].green +
         src[RIDX(row + 1, col, dim)].green) /
        6;
}

char smooth_descr[] = "smooth: Current working version";
void smooth(int dim, pixel *src, pixel *dst) {
    int i, j;

    /* Calculate 4 angles' value */
    avg_left_up(dim, 0, 0, dst, src);
    avg_right_up(dim, 0, dim - 1, dst, src);
    avg_left_down(dim, dim - 1, 0, dst, src);
    avg_right_down(dim, dim - 1, dim - 1, dst, src);

    /* Calculate 4 edges' value */
    for (i = 1; i < dim - 1; i++) {
        avg_up(dim, 0, i, dst, src);
        avg_down(dim, dim - 1, i, dst, src);
        avg_left(dim, i, 0, dst, src);
        avg_right(dim, i, dim - 1, dst, src);
    }

    /*
     * Calculate the center parts' value
     * Using loop unrolling way
     * Set 4 rows to record the value of RGB
     * Calculate the results of 2 adjacent rows per loop
     */
    pixel *pix1 = &src[0];
    pixel *pix2 = &src[dim];
    pixel *pix3 = &src[dim + dim];
    pixel *pix4 = &src[dim + dim + dim];

    /*
     * schematic diagram
     * Every variable sumup/down stores the sum
     * of the color values of three adjacent color blocks vertically
     *       _______ _______ _______ _______
     * pix1 |       |       |       |       |
     *  →   |       |       |       |       |
```

```
     *      |_______|_______|_______|_______|
     * pix2 |   ↑   |   ↑   |   ↑   |       |
     *  →   |sumup1 |sumup2 |sumup3 |       |
     *      |___↓___|___↓___|___↓___|_______|
     * pix3 |   ↑   |   ↑   |   ↑   |       |
     *  →   |sumdow1|sumdow2|sumdow3|       |
     *      |___↓___|___↓___|___↓___|_______|
     * pix4 |       |       |       |       |
     *  →   |       |       |       |       |
     *      |_______|_______|_______|_______|
     */

    int sumup1_red, sumup1_green, sumup1_blue;
    int sumup2_red, sumup2_green, sumup2_blue;
    int sumup3_red, sumup3_green, sumup3_blue;
    int sumdown1_red, sumdown1_green, sumdown1_blue;
    int sumdown2_red, sumdown2_green, sumdown2_blue;
    int sumdown3_red, sumdown3_green, sumdown3_blue;

    int index_uprow = dim + 1;
    int index_downrow = index_uprow + dim;

    /* loop unrolling by the step length of 2*/
    for (i = 1; i < dim - 2; i += 2) {
        sumup1_red = pix2->red;
        sumup1_blue = pix2->blue;
        sumup1_green = pix2->green;
        sumup1_red += pix3->red;
        sumup1_blue += pix3->blue;
        sumup1_green += pix3->green;
        sumdown1_red = sumup1_red + pix4->red;
        sumdown1_green = sumup1_green + pix4->green;
        sumdown1_blue = sumup1_blue + pix4->blue;
        sumup1_red += pix1->red;
        sumup1_blue += pix1->blue;
        sumup1_green += pix1->green;
        /* Right shift and assign the value for sumup/down1 */
        pix1++;
        pix2++;
        pix3++;
        pix4++;

        sumup2_red = pix2->red;
        sumup2_blue = pix2->blue;
```

```
            sumup2_green = pix2->green;
            sumup2_red += pix3->red;
            sumup2_blue += pix3->blue;
            sumup2_green += pix3->green;
            sumdown2_red = sumup2_red + pix4->red;
            sumdown2_green = sumup2_green + pix4->green;
            sumdown2_blue = sumup2_blue + pix4->blue;
            sumup2_red += pix1->red;
            sumup2_blue += pix1->blue;
            sumup2_green += pix1->green;
            /* Right shift and assign the value for sumup/down2 */
            pix1++;
            pix2++;
            pix3++;
            pix4++;

            sumup3_red = pix2->red;
            sumup3_blue = pix2->blue;
            sumup3_green = pix2->green;
            sumup3_red += pix3->red;
            sumup3_blue += pix3->blue;
            sumup3_green += pix3->green;
            sumdown3_red = sumup3_red + pix4->red;
            sumdown3_green = sumup3_green + pix4->green;
            sumdown3_blue = sumup3_blue + pix4->blue;
            sumup3_red += pix1->red;
            sumup3_blue += pix1->blue;
            sumup3_green += pix1->green;
            /* Right shift and assign the value for sumup/down3 */
            pix1++;
            pix2++;
            pix3++;
            pix4++;

            /* Calculate the dst pixels' value */
            dst[index_uprow].red = ((sumup1_red + sumup2_red + sumup3_red) / 9);
            dst[index_uprow].blue = ((sumup1_blue + sumup2_blue + sumup3_blue) / 9);
            dst[index_uprow].green =
                ((sumup1_green + sumup2_green + sumup3_green) / 9);
            index_uprow++;
            dst[index_downrow].red =
                ((sumdown1_red + sumdown2_red + sumdown3_red) / 9);
```

```
        dst[index_downrow].blue =
            ((sumdown1_blue + sumdown2_blue + sumdown3_blue) / 9);
        dst[index_downrow].green =
            ((sumdown1_green + sumdown2_green + sumdown3_green) / 9);
        index_downrow++;

        /*
         * Column Loop Unrolling: 4 operations per loop
         * update the value
         * right shift the pointer
         * assign the value
         * drive the temporary variable
         */
        for (j = 2; j < dim - 4; j += 4) {
            /* First operation */
            /* Move sumup/down1 and sumup/down2 to the right with the existing
             * values */
            sumup1_red = sumup2_red;
            sumup2_red = sumup3_red;
            sumup1_green = sumup2_green;
            sumup2_green = sumup3_green;
            sumup1_blue = sumup2_blue;
            sumup2_blue = sumup3_blue;
            sumdown1_red = sumdown2_red;
            sumdown2_red = sumdown3_red;
            sumdown1_green = sumdown2_green;
            sumdown2_green = sumdown3_green;
            sumdown1_blue = sumdown2_blue;
            sumdown2_blue = sumdown3_blue;

            /* Assign the new value to sumup/down3 */
            sumup3_red = pix2->red;
            sumup3_blue = pix2->blue;
            sumup3_green = pix2->green;
            sumup3_red += pix3->red;
            sumup3_blue += pix3->blue;
            sumup3_green += pix3->green;
            sumdown3_red = sumup3_red + pix4->red;
            sumdown3_green = sumup3_green + pix4->green;
            sumdown3_blue = sumup3_blue + pix4->blue;
            sumup3_red += pix1->red;
```

```
                sumup3_blue += pix1->blue;
                sumup3_green += pix1->green;
                pix1++;
                pix2++;
                pix3++;
                pix4++;

                /* Calculate the dst pixels' value */
                dst[index_uprow].red = ((sumup1_red + sumup2_red + sumup3_red) / 9);
                dst[index_uprow].blue =
                    ((sumup1_blue + sumup2_blue + sumup3_blue) / 9);
                dst[index_uprow].green =
                    ((sumup1_green + sumup2_green + sumup3_green) / 9);
                index_uprow++;

                dst[index_downrow].red =
                    ((sumdown1_red + sumdown2_red + sumdown3_red) / 9);
                dst[index_downrow].blue =
                    ((sumdown1_blue + sumdown2_blue + sumdown3_blue) / 9);
                dst[index_downrow].green =
                    ((sumdown1_green + sumdown2_green + sumdown3_green) / 9);
                index_downrow++;

                /* Second operation */
                sumup1_red = sumup2_red;
                sumup2_red = sumup3_red;
                sumup1_green = sumup2_green;
                sumup2_green = sumup3_green;
                sumup1_blue = sumup2_blue;
                sumup2_blue = sumup3_blue;
                sumdown1_red = sumdown2_red;
                sumdown2_red = sumdown3_red;
                sumdown1_green = sumdown2_green;
                sumdown2_green = sumdown3_green;
                sumdown1_blue = sumdown2_blue;
                sumdown2_blue = sumdown3_blue;

                sumup3_red = pix2->red;
                sumup3_blue = pix2->blue;
```

```
                sumup3_green = pix2->green;
                sumup3_red += pix3->red;
                sumup3_blue += pix3->blue;
                sumup3_green += pix3->green;
                sumdown3_red = sumup3_red + pix4->red;
                sumdown3_green = sumup3_green + pix4->green;
                sumdown3_blue = sumup3_blue + pix4->blue;
                sumup3_red += pix1->red;
                sumup3_blue += pix1->blue;
                sumup3_green += pix1->green;
                pix1++;
                pix2++;
                pix3++;
                pix4++;

                /* Calculate the dst pixels' value */
                dst[index_uprow].red = ((sumup1_red + sumup2_red + sumup3_red) / 9);
                dst[index_uprow].blue =
                    ((sumup1_blue + sumup2_blue + sumup3_blue) / 9);
                dst[index_uprow].green =
                    ((sumup1_green + sumup2_green + sumup3_green) / 9);
                index_uprow++;

                dst[index_downrow].red =
                    ((sumdown1_red + sumdown2_red + sumdown3_red) / 9);
                dst[index_downrow].blue =
                    ((sumdown1_blue + sumdown2_blue + sumdown3_blue) / 9);
                dst[index_downrow].green =
                    ((sumdown1_green + sumdown2_green + sumdown3_green) / 9);
                index_downrow++;

                /* Third operation */
                sumup1_red = sumup2_red;
                sumup2_red = sumup3_red;
                sumup1_green = sumup2_green;
                sumup2_green = sumup3_green;
                sumup1_blue = sumup2_blue;
                sumup2_blue = sumup3_blue;
                sumdown1_red = sumdown2_red;
```

```
                sumdown2_red = sumdown3_red;
                sumdown1_green = sumdown2_green;
                sumdown2_green = sumdown3_green;
                sumdown1_blue = sumdown2_blue;
                sumdown2_blue = sumdown3_blue;

                sumup3_red = pix2->red;
                sumup3_blue = pix2->blue;
                sumup3_green = pix2->green;
                sumup3_red += pix3->red;
                sumup3_blue += pix3->blue;
                sumup3_green += pix3->green;
                sumdown3_red = sumup3_red + pix4->red;
                sumdown3_green = sumup3_green + pix4->green;
                sumdown3_blue = sumup3_blue + pix4->blue;
                sumup3_red += pix1->red;
                sumup3_blue += pix1->blue;
                sumup3_green += pix1->green;
                pix1++;
                pix2++;
                pix3++;
                pix4++;

                /* Calculate the dst pixels' value */
                dst[index_uprow].red = ((sumup1_red + sumup2_red + sumup3_red) / 9);
                dst[index_uprow].blue =
                    ((sumup1_blue + sumup2_blue + sumup3_blue) / 9);
                dst[index_uprow].green =
                    ((sumup1_green + sumup2_green + sumup3_green) / 9);
                index_uprow++;

                dst[index_downrow].red =
                    ((sumdown1_red + sumdown2_red + sumdown3_red) / 9);
                dst[index_downrow].blue =
                    ((sumdown1_blue + sumdown2_blue + sumdown3_blue) / 9);
                dst[index_downrow].green =
                    ((sumdown1_green + sumdown2_green + sumdown3_green) / 9);
                index_downrow++;

```

```
                /* Fourth operation */
                sumup1_red = sumup2_red;
                sumup2_red = sumup3_red;
                sumup1_green = sumup2_green;
                sumup2_green = sumup3_green;
                sumup1_blue = sumup2_blue;
                sumup2_blue = sumup3_blue;
                sumdown1_red = sumdown2_red;
                sumdown2_red = sumdown3_red;
                sumdown1_green = sumdown2_green;
                sumdown2_green = sumdown3_green;
                sumdown1_blue = sumdown2_blue;
                sumdown2_blue = sumdown3_blue;

                sumup3_red = pix2->red;
                sumup3_blue = pix2->blue;
                sumup3_green = pix2->green;
                sumup3_red += pix3->red;
                sumup3_blue += pix3->blue;
                sumup3_green += pix3->green;
                sumdown3_red = sumup3_red + pix4->red;
                sumdown3_green = sumup3_green + pix4->green;
                sumdown3_blue = sumup3_blue + pix4->blue;
                sumup3_red += pix1->red;
                sumup3_blue += pix1->blue;
                sumup3_green += pix1->green;
                pix1++;
                pix2++;
                pix3++;
                pix4++;

                /* Calculate the dst pixels' value */
                dst[index_uprow].red = ((sumup1_red + sumup2_red + s
umup3_red) / 9);
                dst[index_uprow].blue =
                    ((sumup1_blue + sumup2_blue + sumup3_blue) / 9);
                dst[index_uprow].green =
                    ((sumup1_green + sumup2_green + sumup3_green) /
9);
                index_uprow++;

                dst[index_downrow].red =
                    ((sumdown1_red + sumdown2_red + sumdown3_red) /
9);
```

```
            dst[index_downrow].blue =
                ((sumdown1_blue + sumdown2_blue + sumdown3_blue) / 9);
            dst[index_downrow].green =
                ((sumdown1_green + sumdown2_green + sumdown3_green) / 9);
            index_downrow++;
        }

        /*
         * The pixels unassigned in above calculation should be considered
         * Only use the step length 1 to deal the rest of pixels
         */
        for (; j < dim - 1; j++) {
            sumup1_red = sumup2_red;
            sumup2_red = sumup3_red;
            sumup1_green = sumup2_green;
            sumup2_green = sumup3_green;
            sumup1_blue = sumup2_blue;
            sumup2_blue = sumup3_blue;
            sumdown1_red = sumdown2_red;
            sumdown2_red = sumdown3_red;
            sumdown1_green = sumdown2_green;
            sumdown2_green = sumdown3_green;
            sumdown1_blue = sumdown2_blue;
            sumdown2_blue = sumdown3_blue;

            sumup3_red = pix2->red;
            sumup3_blue = pix2->blue;
            sumup3_green = pix2->green;
            sumup3_red += pix3->red;
            sumup3_blue += pix3->blue;
            sumup3_green += pix3->green;
            sumdown3_red = sumup3_red + pix4->red;
            sumdown3_green = sumup3_green + pix4->green;
            sumdown3_blue = sumup3_blue + pix4->blue;
            sumup3_red += pix1->red;
            sumup3_blue += pix1->blue;
            sumup3_green += pix1->green;
            pix1++;
            pix2++;
            pix3++;
            pix4++;
```

```

                dst[index_uprow].red = ((sumup1_red + sumup2_red + sumup3_red) / 9);
                dst[index_uprow].blue =
                    ((sumup1_blue + sumup2_blue + sumup3_blue) / 9);
                dst[index_uprow].green =
                    ((sumup1_green + sumup2_green + sumup3_green) / 9);
                index_uprow++;

                dst[index_downrow].red =
                    ((sumdown1_red + sumdown2_red + sumdown3_red) / 9);
                dst[index_downrow].blue =
                    ((sumdown1_blue + sumdown2_blue + sumdown3_blue) / 9);
                dst[index_downrow].green =
                    ((sumdown1_green + sumdown2_green + sumdown3_green) / 9);
                index_downrow++;
            }

            /* Change the pointer value, make it point to next row */
            pix1 += dim;
            pix2 += dim;
            pix3 += dim;
            pix4 += dim;
            index_uprow += dim + 2;
            index_downrow += dim + 2;
        }
    }

    /***********************************************************************
     * register_smooth_functions - Register all of your different versions
     *      of the smooth kernel with the driver by calling the
     *      add_smooth_function() for each test function.  When you run the
     *      driver program, it will test and report the performance of each
     *      registered test function.
```

```
 *****************************************************************
******/

void register_smooth_functions() {
    add_smooth_function(&smooth, smooth_descr);
    add_smooth_function(&naive_smooth, naive_smooth_descr);
    /* ... Register additional test functions here */
}
```