

Lab4 Performance Lab

notes: 这个 Lab 源自 CMU 的 CSAPP 中的 perflab

姓名: 李宇哲

学号: SA25011049

这个 Lab 主要要完成对 rotate 和 smooth 的优化, 然后与 baseline 比较, 最终看得到的分数。

rotate

naive 版本的 rotate 实现如下:

```
1 void naive_rotate(int dim, pixel *src, pixel *dst)
2 {
3     int i, j;
4
5     for (i = 0; i < dim; i++)
6         for (j = 0; j < dim; j++)
7             dst[RIDX(dim-1-j, i, dim)] = src[RIDX(i, j, dim)];
8 }
```

Optimization 1: blocking 16x16

原始代码按行读取 src, 按列写入 dst, 按列写这里 step 太大了, 容易出现 cache miss, 可以使用 blocking 将一个原始图像分成 16x16 或者 32x32 的 block, 然后每次加载一个 block 的数据到 L1 Cache, 在 Cache 内完成 rotate。

实现如下: 16x16 block 小块

```
1 char rotate_descr[] = "rotate: Blocked 16x16 implementation";
2 void rotate(int dim, pixel *src, pixel *dst)
3 {
4     int i, j, ii, jj;
5     int bsize = 16; // Block size
6
7     for (i = 0; i < dim; i += bsize) {
8         for (j = 0; j < dim; j += bsize) {
9             // Processing a small block
10            // Adding boundary check (ii < dim) just in case dim is not multiple of 16
11            for (ii = i; ii < i + bsize && ii < dim; ii++) {
12                for (jj = j; jj < j + bsize && jj < dim; jj++) {
13                    dst[RIDX(dim-1-jj, ii, dim)] = src[RIDX(ii, jj, dim)];
14                }
15            }
16        }
17    }
18 }
```

Rotate: Version = naive_rotate: Naive baseline implementation:						
Dim	256	512	1024	2048	4096	Mean
Your CPEs	5.1	6.4	11.1	20.0	52.5	
Baseline CPEs	14.7	40.1	46.4	65.9	94.5	
Speedup	2.9	6.3	4.2	3.3	1.8	3.4

Rotate: Version = rotate: Blocked 16x16 implementation:						
Dim	256	512	1024	2048	4096	Mean
Your CPEs	2.1	2.3	3.4	10.8	10.0	
Baseline CPEs	14.7	40.1	46.4	65.9	94.5	
Speedup	7.0	17.5	13.8	6.1	9.4	9.9

在这种优化下，取得了 9.9 的性能分数

Optimization 2: blocking 32x32

将 block 的粒度改为 32

```

1  char rotate_descr[] = "rotate: Blocked 16x16 implementation";
2  void rotate(int dim, pixel *src, pixel *dst)
3  {
4      int i, j, ii, jj;
5      int bsize = 32; // Block size
6
7      for (i = 0; i < dim; i += bsize) {
8          for (j = 0; j < dim; j += bsize) {
9              // Processing a small block
10             // Adding boundary check (ii < dim) just in case dim is not multiple of 16
11             for (ii = i; ii < i + bsize && ii < dim; ii++) {
12                 for (jj = j; jj < j + bsize && jj < dim; jj++) {
13                     dst[RIDX(dim-1-jj, ii, dim)] = src[RIDX(ii, jj, dim)];
14                 }
15             }
16         }
17     }
18 }
```

得到了相对更差的性能分

Rotate: Version = naive_rotate: Naive baseline implementation:						
Dim	256	512	1024	2048	4096	Mean
Your CPEs	7.3	9.9	26.1	23.9	75.7	
Baseline CPEs	14.7	40.1	46.4	65.9	94.5	
Speedup	2.0	4.0	1.8	2.8	1.2	2.2

Rotate: Version = rotate: Blocked 16x16 implementation:						
Dim	256	512	1024	2048	4096	Mean
Your CPEs	3.7	4.3	13.9	16.6	19.1	
Baseline CPEs	14.7	40.1	46.4	65.9	94.5	
Speedup	4.0	9.3	3.3	4.0	5.0	4.8

因此，这里我的最佳实现就是 16x16 blocking

smooth

naive 版本的 smooth 实现如下：

```

1 void naive_smooth(int dim, pixel *src, pixel *dst)
2 {
3     int i, j;
4
5     for (i = 0; i < dim; i++)
6         for (j = 0; j < dim; j++)
7             dst[RIDX(i, j, dim)] = avg(dim, i, j, src);
8 }
```

naive 的实现对每个 pixel 都调用 avg，avg 内部又有循环和 min/max 判断，函数调用开销比较大，同时条件判断会影响 CPU 的分支预测

可以使用 inline，一个常见的编译优化方式，直接在循环内计算 avg，去掉函数调用开销

同时，图像中的大部分像素都有 9 个邻居，不需要做边界检查，从而减小边界检查的开销，对于 4 个角和 4 个像素，可以单独写代码处理，去掉 if 判断，避免分支预测。

```

1 void smooth(int dim, pixel *src, pixel *dst)
2 {
3     int i, j;
4     int dim0 = dim;
5     int dim1 = dim - 1;
6     int dim2 = dim - 2;
7     // No boundary checks needed here, always 9 neighbors.
8     for (i = 1; i < dim1; i++) {
9         for (j = 1; j < dim1; j++) {
10             int r = 0, g = 0, b = 0;
11             int idx;
12
13             // Row i-1
14             idx = RIDX(i-1, j-1, dim);
```

```

15         r += src[idx].red; g += src[idx].green; b += src[idx].blue;
16         r += src[idx+1].red; g += src[idx+1].green; b += src[idx+1].blue;
17         r += src[idx+2].red; g += src[idx+2].green; b += src[idx+2].blue;
18
19         // Row i
20         idx = RIDX(i, j-1, dim);
21         r += src[idx].red; g += src[idx].green; b += src[idx].blue;
22         r += src[idx+1].red; g += src[idx+1].green; b += src[idx+1].blue;
23         r += src[idx+2].red; g += src[idx+2].green; b += src[idx+2].blue;
24
25         // Row i+1
26         idx = RIDX(i+1, j-1, dim);
27         r += src[idx].red; g += src[idx].green; b += src[idx].blue;
28         r += src[idx+1].red; g += src[idx+1].green; b += src[idx+1].blue;
29         r += src[idx+2].red; g += src[idx+2].green; b += src[idx+2].blue;
30
31         dst[RIDX(i, j, dim)].red = (unsigned short)(r / 9);
32         dst[RIDX(i, j, dim)].green = (unsigned short)(g / 9);
33         dst[RIDX(i, j, dim)].blue = (unsigned short)(b / 9);
34     }
35 }
36
37 // Top-Left (0,0) -> 4 neighbors
38 dst[0].red = (src[0].red + src[1].red + src[dim].red + src[dim+1].red) >> 2;
39 dst[0].green = (src[0].green + src[1].green + src[dim].green + src[dim+1].green)
>> 2;
40 dst[0].blue = (src[0].blue + src[1].blue + src[dim].blue + src[dim+1].blue) >> 2;
41
42 // Top-Right (0, dim-1) -> 4 neighbors
43 i = 0; j = dim1;
44 dst[RIDX(i, j, dim)].red = (src[RIDX(0, dim2, dim)].red + src[RIDX(0, dim1,
dim)].red +
45                             src[RIDX(1, dim2, dim)].red + src[RIDX(1, dim1,
dim)].red) >> 2;
46 dst[RIDX(i, j, dim)].green = (src[RIDX(0, dim2, dim)].green + src[RIDX(0, dim1,
dim)].green +
47                             src[RIDX(1, dim2, dim)].green + src[RIDX(1, dim1,
dim)].green) >> 2;
48 dst[RIDX(i, j, dim)].blue = (src[RIDX(0, dim2, dim)].blue + src[RIDX(0, dim1,
dim)].blue +
49                             src[RIDX(1, dim2, dim)].blue + src[RIDX(1, dim1,
dim)].blue) >> 2;
50
51 // Bottom-Left (dim-1, 0) -> 4 neighbors
52 i = dim1; j = 0;
53 dst[RIDX(i, j, dim)].red = (src[RIDX(dim2, 0, dim)].red + src[RIDX(dim2, 1,
dim)].red +
54                             src[RIDX(dim1, 0, dim)].red + src[RIDX(dim1, 1,
dim)].red) >> 2;
55 dst[RIDX(i, j, dim)].green = (src[RIDX(dim2, 0, dim)].green + src[RIDX(dim2, 1,
dim)].green +
56                             src[RIDX(dim1, 0, dim)].green + src[RIDX(dim1, 1,
dim)].green) >> 2;

```

```

57     dst[RIDX(i, j, dim)].blue = (src[RIDX(dim2, 0, dim)].blue + src[RIDX(dim2, 1,
58     dim)].blue +
59
60         src[RIDX(dim1, 0, dim)].blue + src[RIDX(dim1, 1,
61     dim)].blue) >> 2;
62
63     // Bottom-Right (dim-1, dim-1) -> 4 neighbors
64     i = dim1; j = dim1;
65     dst[RIDX(i, j, dim)].red = (src[RIDX(dim2, dim2, dim)].red + src[RIDX(dim2, dim1,
66     dim)].red +
67
68         src[RIDX(dim1, dim2, dim)].red + src[RIDX(dim1, dim1,
69     dim)].red) >> 2;
70     dst[RIDX(i, j, dim)].green = (src[RIDX(dim2, dim2, dim)].green + src[RIDX(dim2,
71     dim1, dim)].green +
72
73         src[RIDX(dim1, dim2, dim)].green + src[RIDX(dim1,
74     dim1, dim)].green) >> 2;
75     dst[RIDX(i, j, dim)].blue = (src[RIDX(dim2, dim2, dim)].blue + src[RIDX(dim2,
76     dim1, dim)].blue +
77
78         src[RIDX(dim1, dim2, dim)].blue + src[RIDX(dim1,
79     dim1, dim)].blue) >> 2;
80
81     // Top Edge (Row 0, cols 1 to dim-2)
82     for (j = 1; j < dim1; j++) {
83         dst[j].red = (src[j-1].red + src[j].red + src[j+1].red +
84
85             src[dim+j-1].red + src[dim+j].red + src[dim+j+1].red) / 6;
86         dst[j].green = (src[j-1].green + src[j].green + src[j+1].green +
87
88             src[dim+j-1].green + src[dim+j].green + src[dim+j+1].green) /
89         6;
90         dst[j].blue = (src[j-1].blue + src[j].blue + src[j+1].blue +
91
92             src[dim+j-1].blue + src[dim+j].blue + src[dim+j+1].blue) / 6;
93     }
94
95     // Bottom Edge (Row dim-1, cols 1 to dim-2)
96     for (j = 1; j < dim1; j++) {
97         int idx = RIDX(dim1, j, dim);
98         int idx_up = RIDX(dim2, j, dim);
99         dst[idx].red = (src[idx_up-1].red + src[idx_up].red + src[idx_up+1].red +
100
101             src[idx-1].red + src[idx].red + src[idx+1].red) / 6;
102         dst[idx].green = (src[idx_up-1].green + src[idx_up].green +
103
104             src[idx_up+1].green +
105
106             src[idx-1].green + src[idx].green + src[idx+1].green) / 6;
107         dst[idx].blue = (src[idx_up-1].blue + src[idx_up].blue + src[idx_up+1].blue +
108
109             src[idx-1].blue + src[idx].blue + src[idx+1].blue) / 6;
110     }
111
112     // Left Edge (Col 0, rows 1 to dim-2)
113     for (i = 1; i < dim1; i++) {
114         int idx = RIDX(i, 0, dim);
115         dst[idx].red = (src[idx-dim].red + src[idx-dim+1].red +
116
117             src[idx].red + src[idx+1].red +
118
119             src[idx+dim].red + src[idx+dim+1].red) / 6;
120         dst[idx].green = (src[idx-dim].green + src[idx-dim+1].green +
121
122             src[idx].green + src[idx+1].green +
123
124             src[idx+dim].green + src[idx+dim+1].green) / 6;

```

```

100         dst[idx].blue = (src[idx-dim].blue + src[idx-dim+1].blue +
101                           src[idx].blue + src[idx+1].blue +
102                           src[idx+dim].blue + src[idx+dim+1].blue) / 6;
103     }
104
105     // Right Edge (Col dim-1, rows 1 to dim-2)
106     for (i = 1; i < dim1; i++) {
107         int idx = RIDX(i, dim1, dim);
108         dst[idx].red = (src[idx-dim-1].red + src[idx-dim].red +
109                       src[idx-1].red + src[idx].red +
110                       src[idx+dim-1].red + src[idx+dim].red) / 6;
111         dst[idx].green = (src[idx-dim-1].green + src[idx-dim].green +
112                          src[idx-1].green + src[idx].green +
113                          src[idx+dim-1].green + src[idx+dim].green) / 6;
114         dst[idx].blue = (src[idx-dim-1].blue + src[idx-dim].blue +
115                         src[idx-1].blue + src[idx].blue +
116                         src[idx+dim-1].blue + src[idx+dim].blue) / 6;
117     }
118 }

```

由此得到的性能提升如下：

Smooth: Version = smooth: Optimized with boundary separation:

Dim	256	512	1024	2048	4096	Mean
Your CPEs	11.5	11.7	13.6	11.6	11.5	
Baseline CPEs	695.0	698.0	702.0	717.0	722.0	
Speedup	60.4	59.4	51.6	61.7	62.6	59.0

Smooth: Version = naive_smooth: Naive baseline implementation:

Dim	256	512	1024	2048	4096	Mean
Your CPEs	26.8	33.0	31.3	29.5	28.2	
Baseline CPEs	695.0	698.0	702.0	717.0	722.0	
Speedup	25.9	21.1	22.5	24.3	25.6	23.8

由此得到了 59.0 的加速比，性能分为 59

因此我的最佳性能表现为：

Summary of Your Best Scores:

Rotate: 9.9 (rotate: Blocked 16x16 implementation)

Smooth: 59.0 (smooth: Optimized with boundary separation)