

性能优化实验

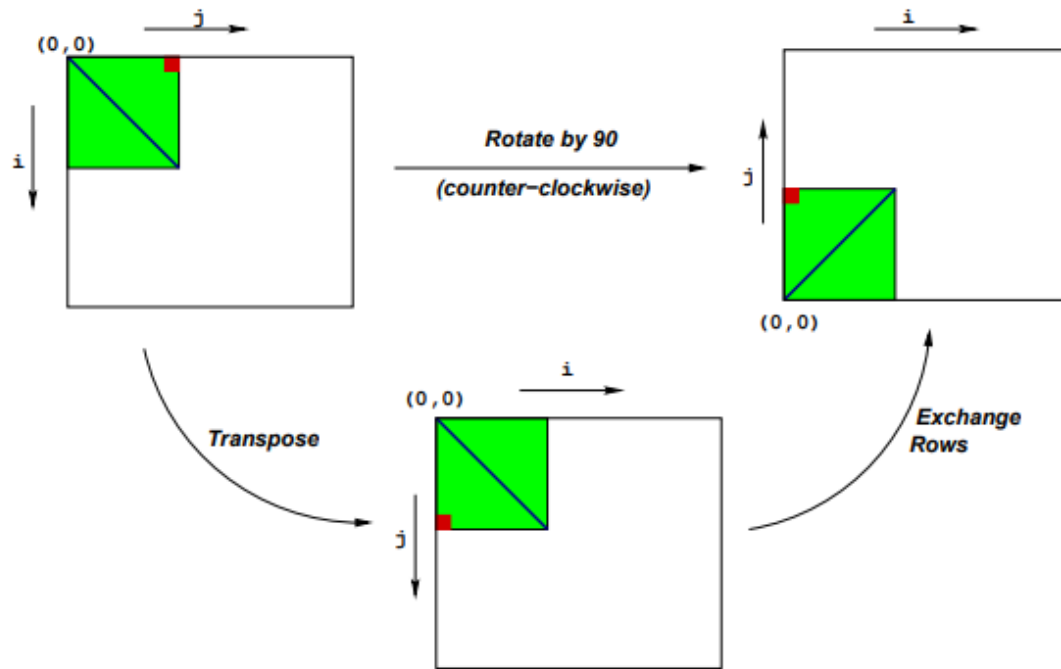
小组成员：孙旻聿、周欣姣、赵薇、胡昊天

任务介绍

本实验来自国外某性能优化实验“perflab”，该任务旨在优化某段内存密集型代码，即完成涉及两个图像处理操作：旋转——使图像逆时针旋转90。平滑——使图像“平滑”或“模糊”。为了简化实验，图像表示维度为N的二维矩阵 $M_{N \times N}$ ， $M_{i,j}$ 表示M矩阵中坐标为 (i,j) 的值，行和列以C语言的方式编号为从0到N-1。鉴于此表示，旋转操作可以非常简单地实现为以下两种矩阵操作的组合：

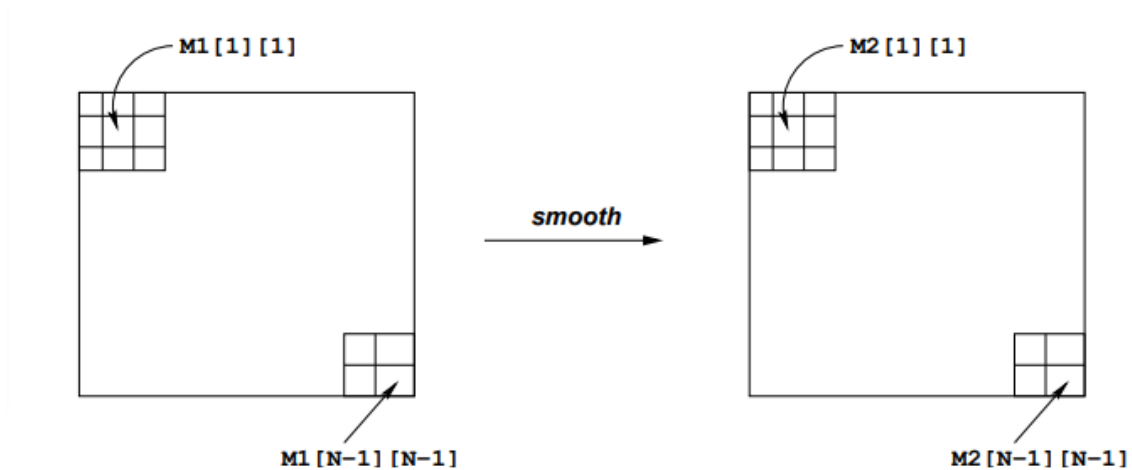
- 转置：对于图像中每个 (i,j) 对, 将 $M_{i,j}$ 和 $M_{j,i}$ 进行互换。
- 换行:将行 i 与行 $N-1-i$ 进行互换。

置换结果如下图所示：



平滑操作通过将每个像素值替换为周围所有像素的平均值来实现（在以该像素为中心的最大为 3×3 窗口中）。像素点 $M2(1,1)$ 和 $M2(N-1,N-1)$ 的值与图例如下所示：

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



实验要求

本实验的驱动代码与基本代码已经给出并存放在压缩包 `perflab-handout.tar`，解压后在文件夹中运行 `make driver` 生成 `driver` 代码，再输入 `./driver` 运行 `driver` 程序。代码加速核心部分在文件 `kernels.c` 中，函数原型如下：

```
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;
}
```

```
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;
}
```

我们需要修改以上两个函数以达到最高的加速比。

任务1: rotate

原始函数 `naive_rotate` 中调用了 `RIDX` 函数，其意思是读取一个 $N \times N$ 的二维数组的第 i 行，第 j 列。所以这段代码将所有的像素进行了行列调位、导致整幅图片逆时针旋转 90° 。由于原代码使用了两重 `for` 循环，而且每一重 `for` 循环的循环次数为 `dim` 次，循环次数过多导致程序的性能非常差，优化方法一是减少循环次数，采用循环展开可以减少循环的次数达到优化效果。

try 1: 循环分块

本程序段的效率主要在于外层循环的次数和内层循环进行的效率，两者都和步长有关。如果步长太小，那么外层循环的进行会变得缓慢，但内层循环由于规模较小所以运行得很快。如果步长太大，那么外层循环进行的很快，但内层循环由于规模相对较大而需要花费更多时间，所以我们需要找到一个相对合适的步长，使内外循环的速率之和达到最佳。优点在于简单易行，容易分解，只需要改变循环的步长。而缺点就是要综合考虑二维矩阵的大小，首先要比 `cache` 小，还要做到能被 `dim` 整除。所以对于未知大小的矩阵或者是数据，步长的选择就变得很困难了。

下面分别列举出展开步长为4和6的代码块，每个代码块下方对应的是加速情况，可以看到当步长为4时的效果是原代码的1.675倍，而步长为16时效果是改进前的1.96875倍，比步长为4时更佳。

```
void rotate(int dim, pixel *src, pixel *dst) {
    int i,j,i1,j1;
    for(i=0;i<dim;i+=4)//展开步长为4
    for(j=0;j<dim;j+=4)
        for(i1=i;i1<i+4;++i1)
            for(j1=j;j1<j+4;++j1)
                dst[RIDX(dim-1-j1, i1, dim)] =src[RIDX(i1,j1,dim)];
}
```

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim          64      128      256      512      1024      Mean
Your CPEs     3.4      3.8      2.6      10.6      14.5
Baseline CPEs 14.7     40.1     46.4     65.9     94.5
Speedup       4.4      10.6     17.6      6.2      6.5      8.0

Rotate: Version = rotate: Current working version:
Dim          64      128      256      512      1024      Mean
Your CPEs     3.8      3.6      2.5      1.5      7.7
Baseline CPEs 14.7     40.1     46.4     65.9     94.5
Speedup       3.9      11.2     18.2     44.9     12.3     13.4
```

```
void rotate(int dim, pixel *src, pixel *dst) {
    int i,j,i1,j1;
    for(i=0;i<dim;i+=16)//展开步长为16
    for(j=0;j<dim;j+=16)
        for(i1=i;i1<i+16;++i1)
            for(j1=j;j1<j+16;++j1)
                dst[RIDX(dim-1-j1, i1, dim)] =src[RIDX(i1,j1,dim)];
}
```

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim          64      128      256      512      1024      Mean
Your CPEs     3.4      3.8      6.7      12.5     14.6
Baseline CPEs 14.7     40.1     46.4     65.9     94.5
Speedup       4.3      10.5     6.9      5.3      6.5      6.4

Rotate: Version = rotate: Current working version:
Dim          64      128      256      512      1024      Mean
Your CPEs     3.3      1.9      3.0      4.4      6.4
Baseline CPEs 14.7     40.1     46.4     65.9     94.5
Speedup       4.5      21.2     15.4     14.8     14.8     12.6
```

try 2: 局部变量与循环交换

本函数 `rotate` 中定义了一个局部变量 `t` 用来储存 `dim-1` 的值，然后再在循环内部调用这个值，这样可以减少循环过程对 `dim` 的调用。在循环过程中将 `i` 和 `j` 的位置交换了，即外层循环 `j`，内层循环 `i`，从矩阵的角度上来看就是按照先列后行的顺序执行将 `src` 中的值提取出来赋给 `dst`。修改代码与得出结果如下。

```

void rotate(int dim, pixel *src, pixel *dst) {
    int i, j;
    int t=dim-1
    for(j=0; j < dim; j++)
    for(i=0; i < dim; i++)
        dst[RIDX(t-j,i,dim)] = src[RIDX(i,j,dim)];
}

```

Rotate: Version = naive_rotate: Naive baseline implementation						
Dim	64	128	256	512	1024	Mean
Your CPEs	3.4	3.8	6.6	10.6	14.6	
Baseline CPEs	14.7	40.1	46.4	65.9	94.5	
Speedup	4.4	10.6	7.0	6.2	6.5	6.7

Rotate: Version = rotate: Current working version:						
Dim	64	128	256	512	1024	Mean
Your CPEs	2.9	2.9	0.2	1.1	9.2	
Baseline CPEs	14.7	40.1	46.4	65.9	94.5	
Speedup	5.0	14.0	287.4	59.8	10.3	26.2

try 3: 循环分块+循环展开

本改动不仅利用了循环分块，而且运用了循环展开，将循环分为32x32的块，而且还在循环中将要执行的操作展开成一条一条的直观的句子。本改动的优点在于思路容易理解而且操作起来比较机械，简单。缺点在于这样做大大的降低了程序的可读性，使代码显得非常冗长不容易理解。

```

void rotate(int dim, pixel *src, pixel *dst) {
    int i, j;
    for(i=0; i < dim; i+=32){
        for(j=dim; j>=0; j-=1){
            pixel *dptr=dst+RIDX(dim-1-j,i,dim);
            pixel *sptr=src+RIDX(i,j,dim);
            *dptr=sptr;sptr+=dim;
            (*dptr+1)=sptr;sptr+=dim;
            (*dptr+2)=sptr;sptr+=dim;
            (*dptr+3)=sptr;sptr+=dim;
            (*dptr+4)=sptr;sptr+=dim;
            (*dptr+5)=sptr;sptr+=dim;
            (*dptr+6)=sptr;sptr+=dim;
            (*dptr+7)=sptr;sptr+=dim;
            (*dptr+8)=sptr;sptr+=dim;
            (*dptr+9)=sptr;sptr+=dim;
            (*dptr+10)=sptr;sptr+=dim;
            (*dptr+11)=sptr;sptr+=dim;
            (*dptr+12)=sptr;sptr+=dim;
            (*dptr+13)=sptr;sptr+=dim;
            (*dptr+14)=sptr;sptr+=dim;
            (*dptr+15)=sptr;sptr+=dim;
            (*dptr+16)=sptr;sptr+=dim;
            (*dptr+17)=sptr;sptr+=dim;
            (*dptr+18)=sptr;sptr+=dim;
            (*dptr+19)=sptr;sptr+=dim;
            (*dptr+20)=sptr;sptr+=dim;
            (*dptr+21)=sptr;sptr+=dim;
        }
    }
}

```

```

        (*dptr+22)=sptr;sptr+=dim;
        (*dptr+23)=sptr;sptr+=dim;
        (*dptr+24)=sptr;sptr+=dim;
        (*dptr+25)=sptr;sptr+=dim;
        (*dptr+26)=sptr;sptr+=dim;
        (*dptr+27)=sptr;sptr+=dim;
        (*dptr+28)=sptr;sptr+=dim;
        (*dptr+29)=sptr;sptr+=dim;
        (*dptr+30)=sptr;sptr+=dim;
        (*dptr+31)=sptr;
    }
}

```

Rotate: Version = naive_rotate: Naive baseline implementation:						
Dim	64	128	256	512	1024	Mean
Your CPEs	4.2	4.8	5.2	15.8	21.5	
Baseline CPEs	14.7	40.1	46.4	65.9	94.5	
Speedup	3.5	8.4	8.8	4.2	4.4	5.4

Rotate: Version = rotate1: Current working version:						
Dim	64	128	256	512	1024	Mean
Your CPEs	7.1	6.8	5.0	4.1	8.7	
Baseline CPEs	14.7	40.1	46.4	65.9	94.5	
Speedup	2.1	5.9	9.2	16.2	10.9	7.3

综合分析

改进思路主要是通过改变循环方式，比如分块、展开、循环次序改变等来达到提升代码速度的效果。每种方法都能使代码得到一定的速度提升。但这仅限于rotate函数这种计算次数并不多的函数。通过上述比较可以发现，普通的循环分块或者循环展开只能给程序带来非常有限的改进，而根据函数特点改变函数中循环的结构则能够带来相对较大的改进。综上可知，循环分块展开是一种非常具有普遍性的改进方法，但是改进的效果非常有限，而且会降低程序的可读性。而根据函数特点进行改进的话能够更大的提升函数的性能，缺点是局限性很大，一种改进方法仅适用于一种函数。由Amdahl law对上述改进进行分析，通过比较各个改进函数的加速比来判断函数的改进程度。

方法	改进措施	原加速比	现加速比	提升比例
循环分块	将原矩阵执行16x16分块循环操作	6.4	12.6	96.875%
循环顺序改变	在循环过程中将i和j的位置进行交换	6.7	26.2	291%
循环分块+ 循环展开	将循环分为32x32的块，在循环中将操作展开	5.4	7.3	35.18%

任务2：smooth

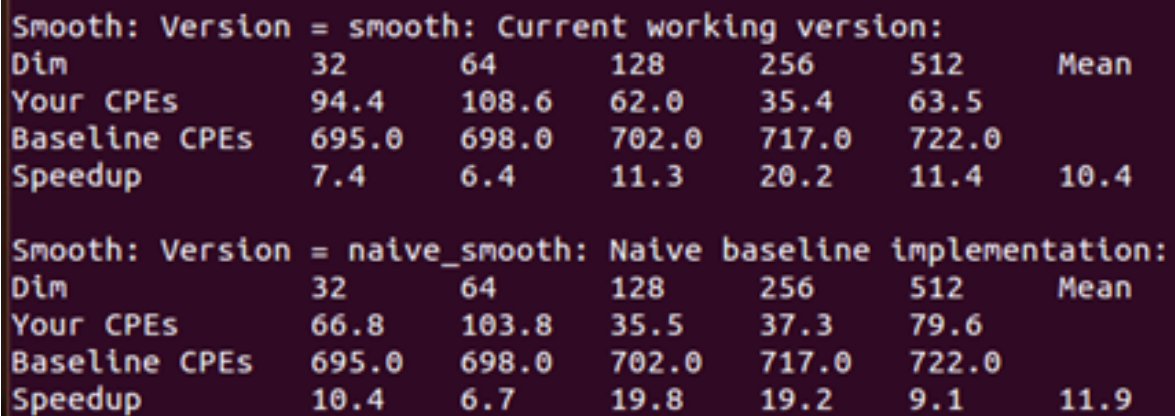
原 naive_smooth 函数执行的操作是建立一个副本，将原图像的像素点计算之后存入该副本中，副本的某点的像素为对应原画的周围像素的平均值（如果该像素点在四个顶点，则副本中该点像素值等于原画包括该点的周围四个点的像素值平均，如果该像素点为四条边界上的像素点，则腰品的该点像素点等于原画包括该点的周围六个点的像素平均值，如果该像素点为中间点，则该点像素为原画的周围九个点的平均值）。这段代码频繁地调用 avg 函数，并且 avg 函数中也频繁调用 initialize_pixel_sum 、

accumulate_sum、assign_sum_to_pixel 这几个函数，且又含有2层for循环，而我们应该减少函数调用的时间开销。所以，需要改写代码，不调用 avg 函数。

try 1: 循环分块+展开

减少循环次数，进行步长为2的循环展开，尽管跟原代码差别不是很大，但是还是有了一些优化，之后进行步长为4的展开，效果也并不是很理想。本方法按照基本优化方法循环展开，思想简单，但是优化效果比较弱，而且降低了代码的可读性。之后尝试根据分析，发现是因为调用的avg函数包含太多计算，所以一般的循环优化方式并不能起到预期的优化作用。

```
void smooth(int dim, pixel *src, pixel *dst) {
    int i, j;
    for (i = 0; i < dim; i++)//2x2的分块展开
        for (j = 0; j < dim-1; j+=2){
            dst[RIDX(i, j, dim)] = avg(dim, i, j, src);
            dst[RIDX(i, j+1, dim)] = avg(dim, i, j+1, src);
        }
    for (; j < dim; j++)
        dst[RIDX(i, j, dim)] = avg(dim, i, j, src);
}
```



Smooth: Version = smooth: Current working version:

Dim	32	64	128	256	512	Mean
Your CPEs	94.4	108.6	62.0	35.4	63.5	
Baseline CPEs	695.0	698.0	702.0	717.0	722.0	
Speedup	7.4	6.4	11.3	20.2	11.4	10.4

Smooth: Version = naive_smooth: Naive baseline implementation:

Dim	32	64	128	256	512	Mean
Your CPEs	66.8	103.8	35.5	37.3	79.6	
Baseline CPEs	695.0	698.0	702.0	717.0	722.0	
Speedup	10.4	6.7	19.8	19.2	9.1	11.9

```
void smooth(int dim, pixel *src, pixel *dst) {
    int i, j;
    for (i = 0; i < dim; i++)//2x2的分块展开
        for (j = 0; j < dim-1; j+=2){
            dst[RIDX(i, j, dim)] = avg(dim, i, j, src);
            dst[RIDX(i, j+1, dim)] = avg(dim, i, j+1, src);
            dst[RIDX(i, j+2, dim)] = avg(dim, i, j+2, src);
            dst[RIDX(i, j+3, dim)] = avg(dim, i, j+3, src);
            dst[RIDX(i+1, j, dim)] = avg(dim, i+1, j, src);
            dst[RIDX(i+1, j+1, dim)] = avg(dim, i+1, j+1, src);
            dst[RIDX(i+1, j+2, dim)] = avg(dim, i+1, j+2, src);
            dst[RIDX(i+1, j+3, dim)] = avg(dim, i+1, j+3, src);
            dst[RIDX(i+2, j, dim)] = avg(dim, i+1, j, src);
            dst[RIDX(i+2, j+1, dim)] = avg(dim, i+1, j+1, src);
            dst[RIDX(i+2, j+2, dim)] = avg(dim, i+1, j+2, src);
            dst[RIDX(i+2, j+3, dim)] = avg(dim, i+1, j+3, src);
            dst[RIDX(i+3, j, dim)] = avg(dim, i+1, j, src);
            dst[RIDX(i+3, j+1, dim)] = avg(dim, i+1, j+1, src);
            dst[RIDX(i+3, j+2, dim)] = avg(dim, i+1, j+2, src);
            dst[RIDX(i+3, j+3, dim)] = avg(dim, i+1, j+3, src);
        }
}
```



```
}
```

Smooth: Version = smooth: Current working version:						
Dim	32	64	128	256	512	Mean
Your CPEs	66.9	65.4	9.6	48.0	62.2	
Baseline CPEs	695.0	698.0	702.0	717.0	722.0	
Speedup	10.4	10.7	73.1	14.9	11.6	17.0

Smooth: Version = naive_smooth: Naive baseline implementation:						
Dim	32	64	128	256	512	Mean
Your CPEs	66.4	65.6	65.3	58.5	63.4	
Baseline CPEs	695.0	698.0	702.0	717.0	722.0	
Speedup	10.5	10.6	10.8	12.3	11.4	11.1

try 2: 增加并行性

并行改进这一种改进方式尝试过多次，仅有二路并行是有效果的，当拆分到四路并行时，优化之后的代码加速比更低了，所以本次仅采用二路并行方式优化。优点在于，本次改动相较于其他改动很简单，只是将双层循环中的内层循环分解提高函数的并行性，操作并不复杂。缺点在于这种改动对函数的时间效率提升效果并不明显，仅仅只提升了26%。

```
void smooth(int dim, pixel *src, pixel *dst) {
    int i, j;
    for (i = 0; i < dim; i++){
        for (j = 0; j <= dim/2; j++)
            dst[RIDX(i, j, dim)] = avg(dim, i, j, src);
        for (j=dim/2+1; j < dim; j++)
            dst[RIDX(i, j, dim)] = avg(dim, i, j, src);
    }
}
```

Smooth: Version = smooth: Current working version:						
Dim	32	64	128	256	512	Mean
Your CPEs	72.9	70.0	13.0	57.1	65.7	
Baseline CPEs	695.0	698.0	702.0	717.0	722.0	
Speedup	9.5	10.0	54.0	12.6	11.0	14.8

Smooth: Version = naive_smooth: Naive baseline implementation:						
Dim	32	64	128	256	512	Mean
Your CPEs	66.6	65.7	65.3	43.2	65.3	
Baseline CPEs	695.0	698.0	702.0	717.0	722.0	
Speedup	10.4	10.6	10.7	16.6	11.1	11.7

try 3: 改写分块

本次改动的幅度非常大，直接将代码重写了一遍。首先对图片进行分块处理，像素点分成图片四个角、图片四条边、图片内部三块分别进行处理。对角而言只需要4个像素点的均值，对于边而言为6个像素点均值，图片内部则需要9个像素点均值。这样可以减少循环判断的次数，而且原代码的avg函数是在一次循环中只进行一个像素点的累加，avg函数里用了两重循环，而函数主体有两重循环，相当于原代码用了四重循环，所以性能很差，所以改进后不使用avg函数，并且通过分块，将每个分块周围的像素点一次性加起来而不是一个循环累加一个像素点，达到大大减少循环调用的功能。。优点在于通过函数重写大

大提高了加速比，省去了效率极低的avg函数，使程序运行效率大大提高。缺点在于这样的改动降低了程序的可读性，用大量的空间去换取时间。

```
void smooth(int dim, pixel *src, pixel *dst)
{
    int i,j,lastr,lastb,lastg;
    int row = dim;
    int curr;//四个角处理
    dst[0].red=(src[0].red+src[1].red+src[dim].red+src[dim+1].red)>>2;
    dst[0].blue=(src[0].blue+src[1].blue+src[dim].blue+src[dim+1].blue)>>2;
    dst[0].green=(src[0].green+src[1].green+src[dim].green+src[dim+1].green)>>2;
    dst[dim-1].red=(src[dim-1].red+src[dim-2].red+src[dim*2-2].red+src[dim*2-1].red)>>2;
    dst[dim-1].blue=(src[dim-1].blue+src[dim-2].blue+src[dim*2-2].blue+src[dim*2-1].blue)>>2;
    dst[dim-1].green=(src[dim-1].green+src[dim-2].green+src[dim*2-2].green+src[dim*2-1].green)>>2;

    dst[(dim-1)*dim].red=(src[(dim-1)*dim].red+src[(dim-1)*dim+1].red+src[(dim-2)*dim].red+src[(dim-2)*dim+1].red)>>2;
    dst[(dim-1)*dim].blue=(src[(dim-1)*dim].blue+src[(dim-1)*dim+1].blue+src[(dim-2)*dim].blue+src[(dim-2)*dim+1].blue)>>2;
    dst[(dim-1)*dim].green=(src[(dim-1)*dim].green+src[(dim-1)*dim+1].green+src[(dim-2)*dim].green+src[(dim-2)*dim+1].green)>>2;

    dst[dim*dim-1].red=(src[dim*dim-1].red+src[dim*dim-2].red+src[(dim-1)*dim-1].red+src[(dim-1)*dim-2].red)>>2;
    dst[dim*dim-1].blue=(src[dim*dim-1].blue+src[dim*dim-2].blue+src[(dim-1)*dim-1].blue+src[(dim-1)*dim-2].blue)>>2;
    dst[dim*dim-1].green=(src[dim*dim-1].green+src[dim*dim-2].green+src[(dim-1)*dim-1].green+src[(dim-1)*dim-2].green)>>2;
    //四条边
    for (j=1; j < dim-1; j++) {
        dst[j].red=(src[j].red+src[j-1].red+src[j+1].red+src[j+dim].red+src[j+1+dim].red+src[j-1+dim].red)/6;
        dst[j].green=(src[j].green+src[j-1].green+src[j+1].green+src[j+dim].green+src[j+1+dim].green+src[j-1+dim].green)/6;
        dst[j].blue=(src[j].blue+src[j-1].blue+src[j+1].blue+src[j+dim].blue+src[j+1+dim].blue+src[j-1+dim].blue)/6;
    }
    for (j=dim*(dim-1)+1; j < dim*dim-1; j++) {
        dst[j].red=(src[j].red+src[j-1].red+src[j+1].red+src[j-dim].red+src[j+1-dim].red+src[j-1-dim].red)/6;
        dst[j].green=(src[j].green+src[j-1].green+src[j+1].green+src[j-dim].green+src[j+1-dim].green+src[j-1-dim].green)/6;
        dst[j].blue=(src[j].blue+src[j-1].blue+src[j+1].blue+src[j-dim].blue+src[j+1-dim].blue+src[j-1-dim].blue)/6;
    }

    for (i=dim; i < dim*(dim-1); i+=dim) {
        dst[i].red=(src[i].red+src[i-dim].red+src[i+1].red+src[i+dim].red+src[i+1+dim].red+src[i-dim+1].red)/6;
        dst[i].green=(src[i].green+src[i-dim].green+src[i+1].green+src[i+dim].green+src[i+1+dim].green+src[i-dim+1].green)/6;
    }
}
```



```

        dst[i].blue=(src[i].blue+src[i-
dim].blue+src[i+1].blue+src[i+dim].blue+src[i+1+dim].blue+src[i-dim+1].blue)/6;
    }

    for (i=dim+dim-1; i < dim*dim-1; i+=dim) {
        dst[i].red=(src[i].red+src[i-1].red+src[i-dim].red+src[i+dim].red+src[i-
dim-1].red+src[i-1+dim].red)/6;
        dst[i].green=(src[i].green+src[i-1].green+src[i-
dim].green+src[i+dim].green+src[i-dim-1].green+src[i-1+dim].green)/6;
        dst[i].blue=(src[i].blue+src[i-1].blue+src[i-
dim].blue+src[i+dim].blue+src[i-dim-1].blue+src[i-1+dim].blue)/6;
    }

    for(i=1;i<dim-1; i++){
        lastr=src[row-dim].red+src[row-dim+1].red+src[row-dim+2].red+src[row].red+
src[row+1].red+src[row+2].red+src[row+dim].red+src[row+dim+1].red+src[row+dim+2
].red;
        lastb=src[row-dim].blue+src[row-dim+1].blue+src[row-
dim+2].blue+src[row].blue+
src[row+1].blue+src[row+2].blue+src[row+dim].blue+src[row+dim+1].blue+src[row+d
im+2].blue;
        lastg=src[row-dim].green+src[row-dim+1].green+src[row-
dim+2].green+src[row].green+src[row+1].green+src[row+2].green+src[row+dim].green
+src[row+dim+1].green+src[row+dim+2].green;
        dst[row+1].red=lastr/9;
        dst[row+1].blue=lastb/9;
        dst[row+1].green=lastg/9;
        for(j=2;j<dim-1;j++){
            curr=row+j;
            lastr=lastr-src[curr-dim-2].red+src[curr-dim+1].red-src[curr-
2].red+src[curr+1].red-src[curr+dim-2].red+src[curr+dim+1].red;
            lastb=lastb-src[curr-dim-2].blue+src[curr-dim+1].blue-src[curr-
2].blue+src[curr+1].blue-src[curr+dim-2].blue+src[curr+dim+1].blue;
            lastg=lastg-src[curr-dim-2].green+src[curr-dim+1].green-src[curr-
2].green+src[curr+1].green-src[curr+dim-2].green+src[curr+dim+1].green;
            dst[curr].red=lastr/9;
            dst[curr].blue=lastb/9;
            dst[curr].green=lastg/9;//内部其他点参考该行前一个点的像素值得到结果
        }
        row+=dim;
    }
}

```

```

Smooth: Version = smooth: Current working version:
Dim          32      64      128      256      512      Mean
Your CPEs    18.9    17.3    17.0    10.7     14.2
Baseline CPEs 695.0   698.0   702.0   717.0   722.0
Speedup      36.7    40.4    41.3    67.0    51.0    46.2

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim          32      64      128      256      512      Mean
Your CPEs    139.2   43.7    56.2    34.4    52.1
Baseline CPEs 695.0   698.0   702.0   717.0   722.0
Speedup       5.0    16.0    12.5    20.8    13.9    12.4

```

综合分析

对于本函数而言，不管是循环分块或者是提高循环的并行性都有一定的提升作用，但这种提升作用不仅降低了程序的可读性，而且提升效果也并不明显。相比较而言对于函数的重新改写起到的作用更大。经过对比之后，发现将函数改写的加速比提升比例最大，效果最好。所以总结出，在调用的函数比较复杂的情况下最好的改写方式就是改变算法重写函数以实现优化效果。

方法	改进措施	原加速比	现加速比	提升比例
循环分块+展开	将原矩阵执行4x4分块循环操作	11.1	17	53.15%
提高并行性	在循环过程中将j的计算改成二路并行	11.7	14.8	26.50%
分块改写	从矩阵边、角、中间分别进行计算	12.4	46.2	272.58%