rotate 和 smooth 优化

小组成员：李宗鸿-S211000810，文清华-S211000813，黄爽-S211000791，赵思蓉-B2110Z0956，鲁时宇-S211000847

介绍：

此任务涉及优化内存密集型代码。图像处理提供了许多可以从优化中受益的函数示例。在这个实验中，我们将考虑两种图像处理操作：rotate，将图像逆时针旋转 90◦, smooth，用于“平滑”或“模糊”图像。

我们将考虑一个图像来表示为二维矩阵 M，Mi,j 表示 M 的第（i，j）个像素的值。像素值是红色、绿色和蓝色（RGB）值的三元组。我们只考虑方形图像。行数（或列数）以 C 语言风格编号，从 0 到 N−1.考虑到这种表示。

Part A 旋转操作可以非常简单地作为以下两个矩阵运算：

转置：对于每个（i，j）对，Mi,j 和 Mj,i 互换。

交换行：第 i 行与第 N−1 行交换

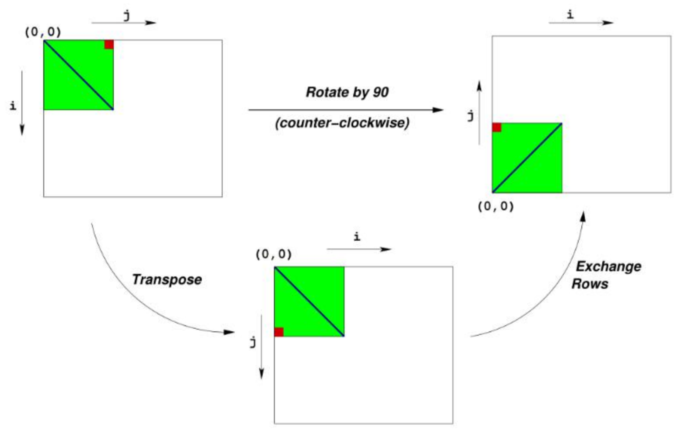


Figure 1: Rotation of an image by 90° counterclockwise

Part B 平滑操作就是通过讲每个像素点地值替换成周围像素地平均值来实现的，如下所示

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

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

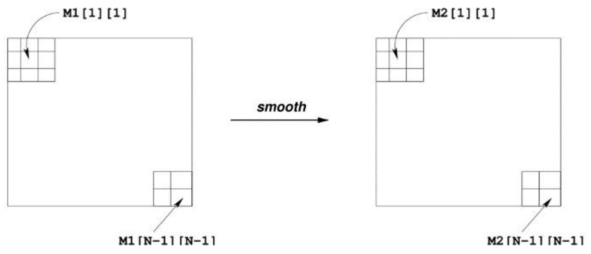
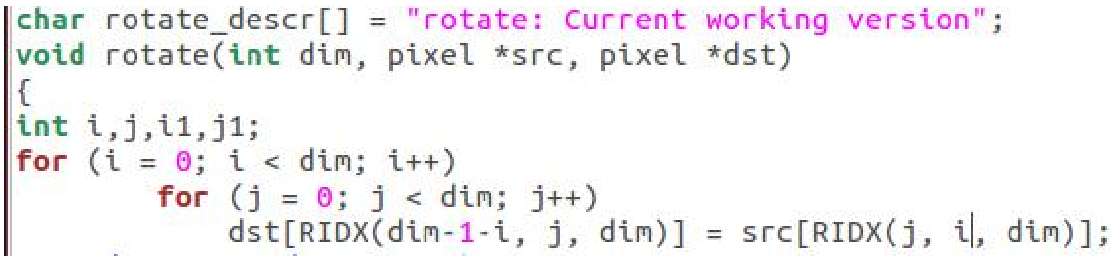


Figure 2: Smoothing an image

1. 比较所完成的 PartA 三段优化程序的优缺点并详细说明

（1）交换内外循环次序，降低循环的低效率

在 rotate 函数中，每一次操作都是从 src 中读，写到 dst 中。对于读操作，是按行读取步长为 1，cache 命中率较高；而对于写操作，是按列写入，步长为 dim，cache 命中率较低。因此，可以交换内外循环的次序进行优化，优先考虑写的操作。



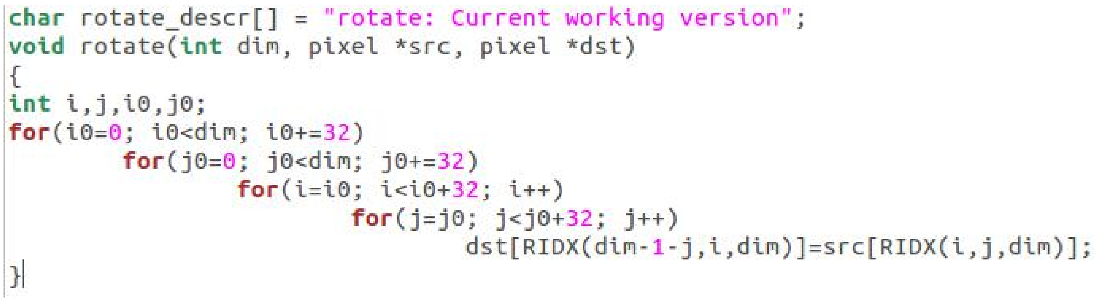
```
char rotate_descr[] = "rotate: Current working version";
void rotate(int dim, pixel *src, pixel *dst)
{
int i,j,i1,j1;
for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            dst[RIDX(dim-1-i, j, dim)] = src[RIDX(j, i, dim)];
```

优点：可以提高 cache 命中率，从而提高循环效率

缺点：没有考虑程序的并行性

（2）循环分块，消除不必要的内存引用

构造程序，将 1 个片加载到 L1 高速缓存，并在这个块进行所有的读和写，然后丢掉这个块，加载下一个，以此类推。由于测试 dim 规格分别为 64、128、256、512、1024，块的大小要能被 dim 整除，所以可以选择 4、8、16、32 等等。



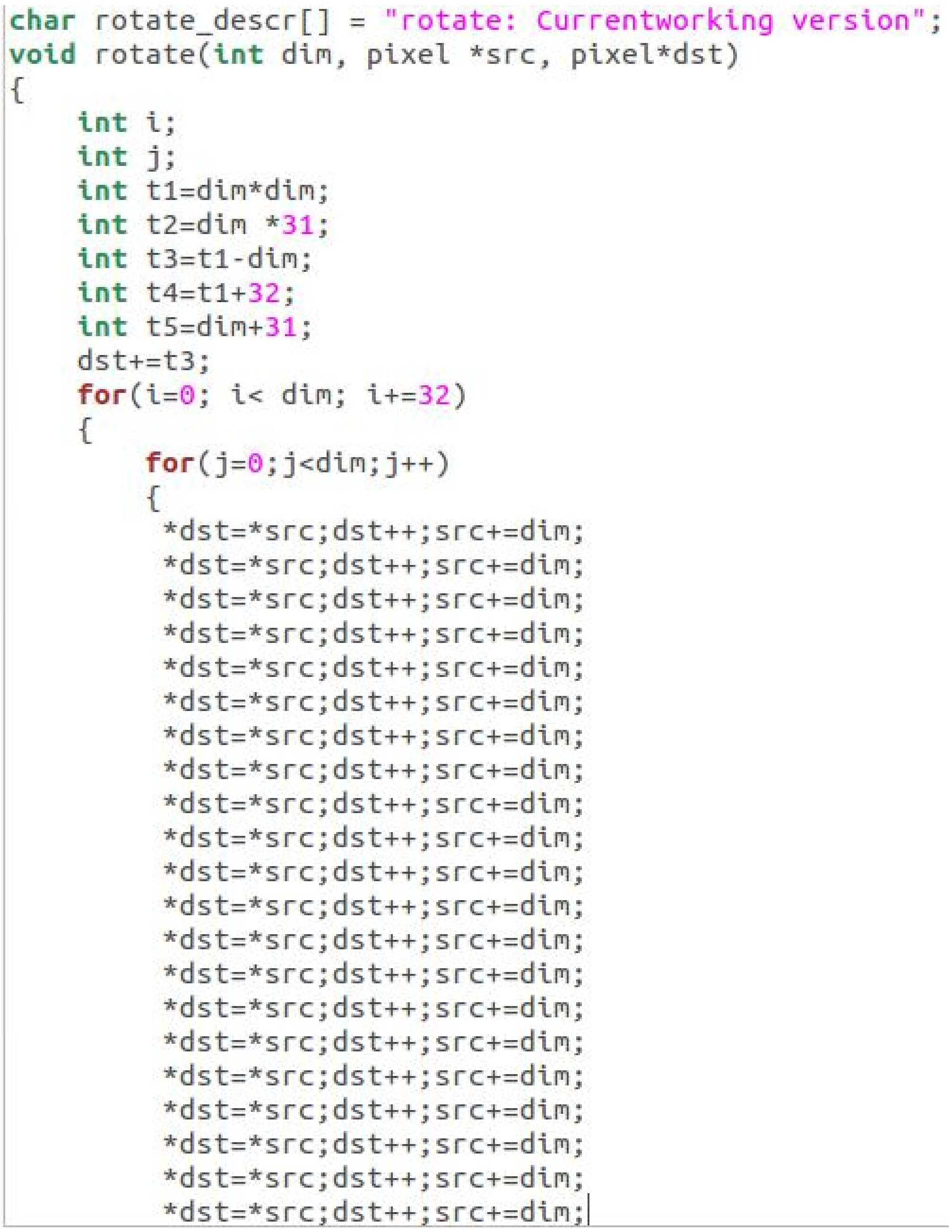
```
char rotate_descr[] = "rotate: Current working version";
void rotate(int dim, pixel *src, pixel *dst)
{
int i,j,i0,j0;
for(i0=0; i0<dim; i0+=32)
        for(j0=0; j0<dim; j0+=32)
                for(i=i0; i<i0+32; i++)
                        for(j=j0; j<j0+32; j++)
                                dst[RIDX(dim-1-j,i,dim)]=src[RIDX(i,j,dim)];
}
```

优点：充分利用 L1 高速缓存，每次将 32*32 矩阵的数据放到 L1 高速缓存中，这样大大减少了每次程序需要去内存中取数据的时间开销

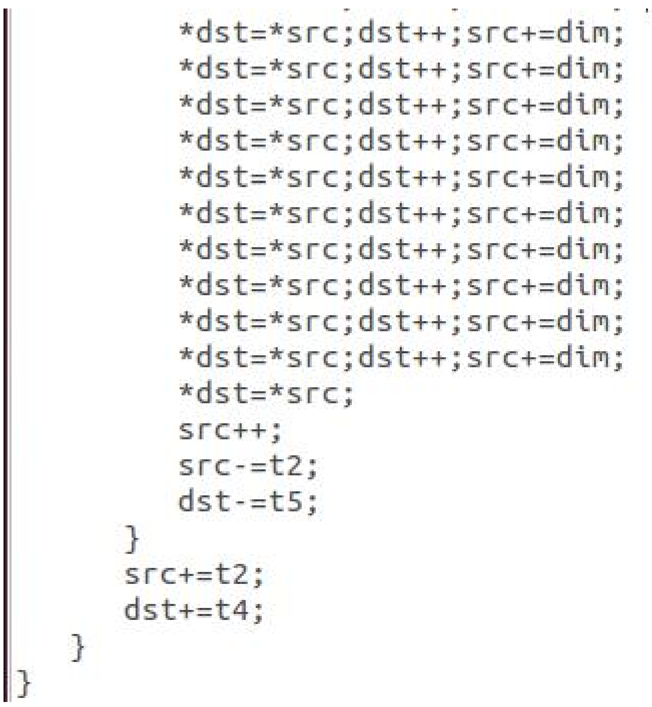
缺点：没有考虑程序的并行性

（3）循环展开

采用循环展开，先读取矩阵第一列的前 32 个元素写入到最后一行的前 32 个元素，再读取第二列前 32 个元素写入到倒数第二行的前 32 个位置。读取完前 32 行的所有元素后，再读取接下来的 32 行，直至完成操作。



```c
char rotate_descr[] = "rotate: Currentworking version";
void rotate(int dim, pixel *src, pixel*dst)
{
    int i;
    int j;
    int t1=dim*dim;
    int t2=dim *31;
    int t3=t1-dim;
    int t4=t1+32;
    int t5=dim+31;
    dst+=t3;
    for(i=0; i< dim; i+=32)
    {
        for(j=0;j<dim;j++)
        {
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
         *dst=*src;dst++;src+=dim;
```

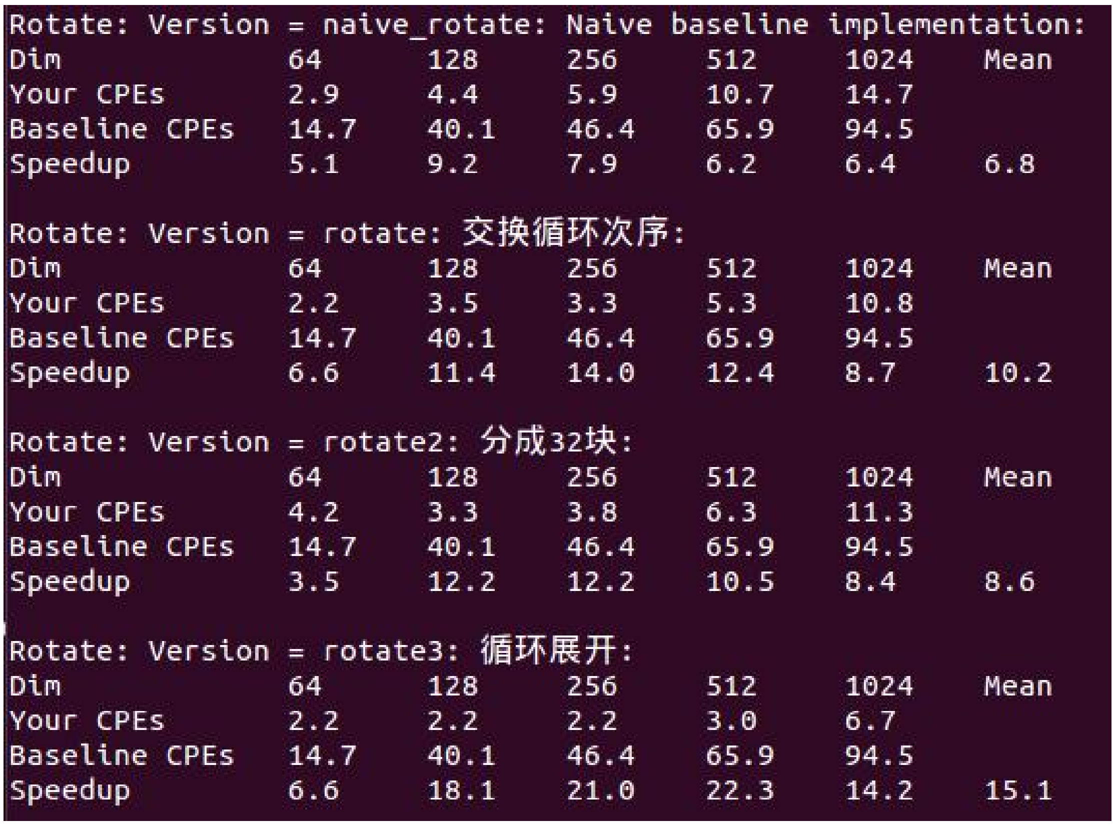


```
            *dst=*src;dst++;src+=dim;
            *dst=*src;dst++;src+=dim;
            *dst=*src;dst++;src+=dim;
            *dst=*src;dst++;src+=dim;
            *dst=*src;dst++;src+=dim;
            *dst=*src;dst++;src+=dim;
            *dst=*src;dst++;src+=dim;
            *dst=*src;dst++;src+=dim;
            *dst=*src;dst++;src+=dim;
            *dst=*src;dst++;src+=dim;
            *dst=*src;
            src++;
            src-=t2;
            dst-=t5;
        }
        src+=t2;
        dst+=t4;
    }
}
```

优点：提高了程序的并行性

缺点：代码的可读性大大降低，空间开销增大

三种优化方法与优化前对比：



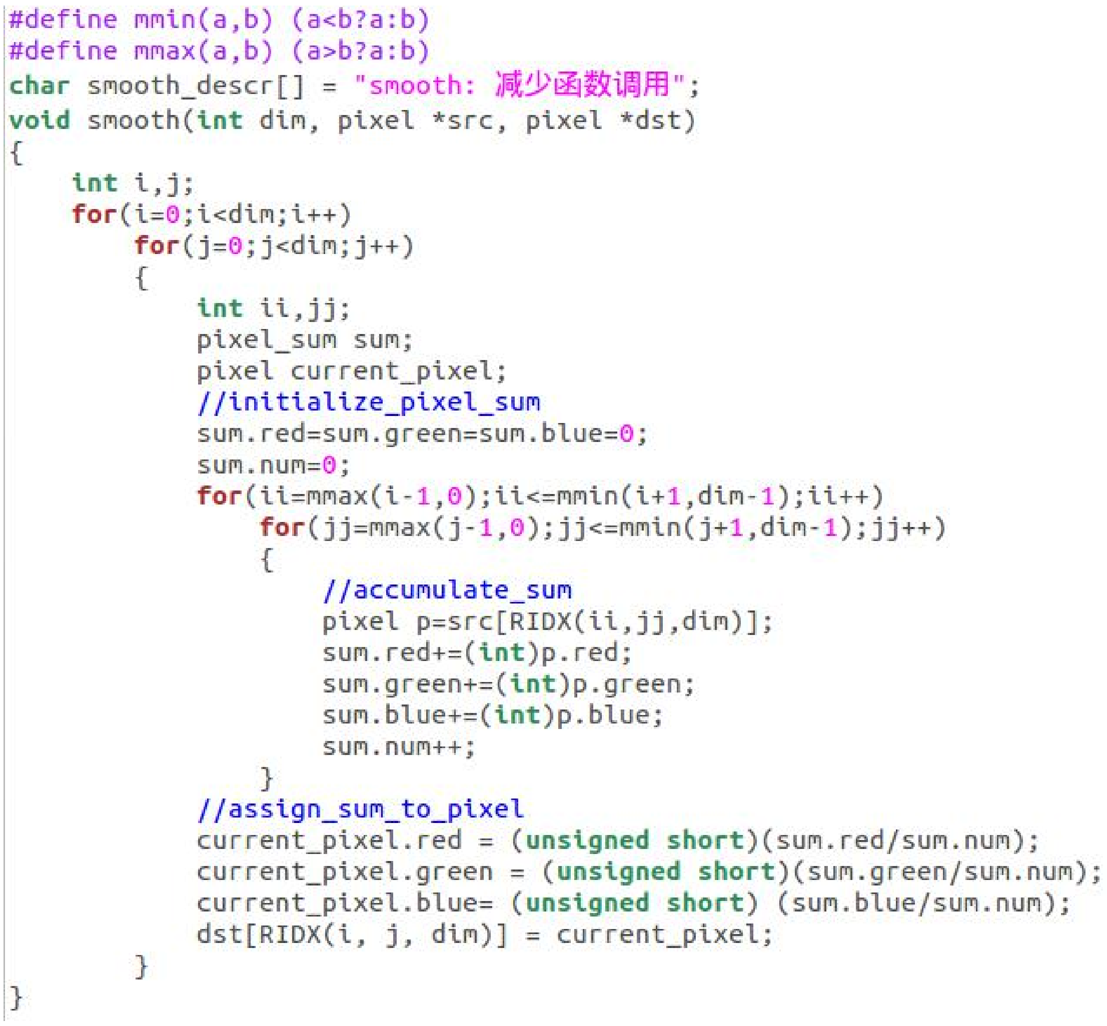
```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       2.9     4.4     5.9     10.7    14.7
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         5.1     9.2     7.9     6.2     6.4     6.8

Rotate: Version = rotate: 交换循环次序:
Dim             64      128     256     512     1024    Mean
Your CPEs       2.2     3.5     3.3     5.3     10.8
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         6.6     11.4    14.0    12.4    8.7     10.2

Rotate: Version = rotate2: 分成32块:
Dim             64      128     256     512     1024    Mean
Your CPEs       4.2     3.3     3.8     6.3     11.3
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         3.5     12.2    12.2    10.5    8.4     8.6

Rotate: Version = rotate3: 循环展开:
Dim             64      128     256     512     1024    Mean
Your CPEs       2.2     2.2     2.2     3.0     6.7
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         6.6     18.1    21.0    22.3    14.2    15.1
```

可以看到，循环展开的优化效果最好。

2. 比较所完成的 PartB 三段优化程序的优缺点并详细说明

（1）减少过程调用

原代码定义了很多函数，有多次函数调用，可以考虑将函数整合到一起，将 min,max 函数改为宏定义。



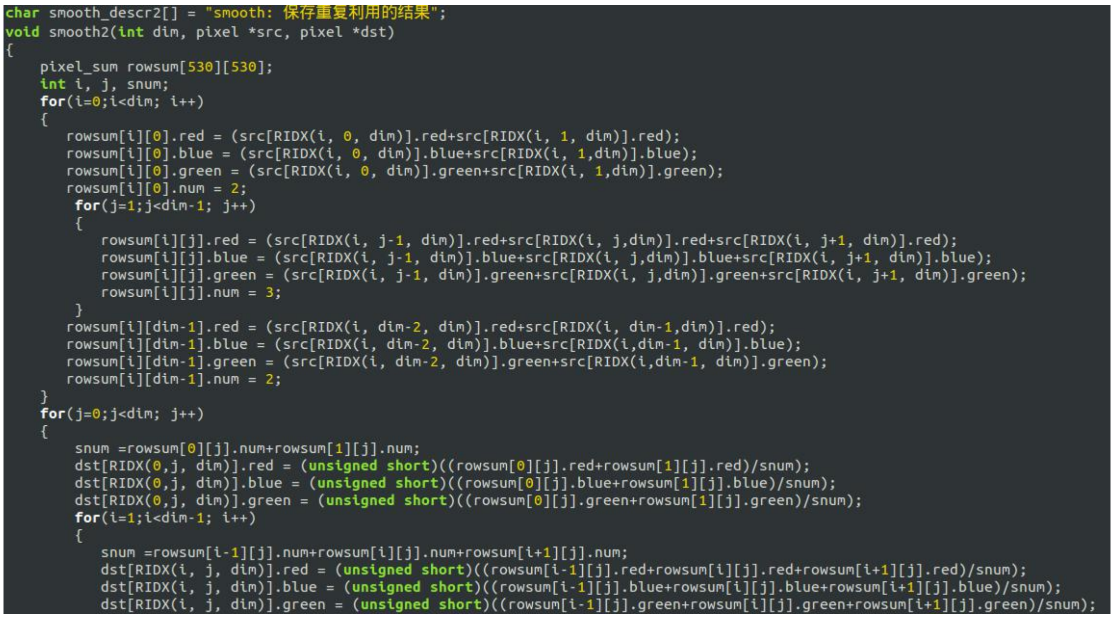
```
#define mmin(a,b) (a<b?a:b)
#define mmax(a,b) (a>b?a:b)
char smooth_descr[] = "smooth: 减少函数调用";
void smooth(int dim, pixel *src, pixel *dst)
{
    int i,j;
    for(i=0;i<dim;i++)
        for(j=0;j<dim;j++)
        {
            int ii,jj;
            pixel_sum sum;
            pixel current_pixel;
            //initialize_pixel_sum
            sum.red=sum.green=sum.blue=0;
            sum.num=0;
            for(ii=mmax(i-1,0);ii<=mmin(i+1,dim-1);ii++)
                for(jj=mmax(j-1,0);jj<=mmin(j+1,dim-1);jj++)
                {
                    //accumulate_sum
                    pixel p=src[RIDX(ii,jj,dim)];
                    sum.red+=(int)p.red;
                    sum.green+=(int)p.green;
                    sum.blue+=(int)p.blue;
                    sum.num++;
                }
            //assign_sum_to_pixel
            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);
            dst[RIDX(i, j, dim)] = current_pixel;
        }
}
```

优点：减少了过程的调用，从而提高了程序的性能

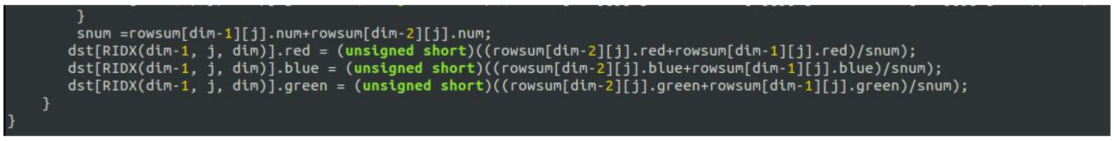
缺点：优化效果较小

（2）减少过程调用。

利用动态规划的算法思想，保存需要重复利用的中间结果，可以提高程序的性能。



```c
char smooth_descr2[] = "smooth: 保存重复利用的结果";
void smooth2(int dim, pixel *src, pixel *dst)
{
    pixel_sum rowsum[530][530];
    int i, j, snum;
    for(i=0;i<dim; i++)
    {
       rowsum[i][0].red = (src[RIDX(i, 0, dim)].red+src[RIDX(i, 1, dim)].red);
       rowsum[i][0].blue = (src[RIDX(i, 0, dim)].blue+src[RIDX(i, 1,dim)].blue);
       rowsum[i][0].green = (src[RIDX(i, 0, dim)].green+src[RIDX(i, 1,dim)].green);
       rowsum[i][0].num = 2;
        for(j=1;j<dim-1; j++)
        {
           rowsum[i][j].red = (src[RIDX(i, j-1, dim)].red+src[RIDX(i, j,dim)].red+src[RIDX(i, j+1, dim)].red);
           rowsum[i][j].blue = (src[RIDX(i, j-1, dim)].blue+src[RIDX(i, j,dim)].blue+src[RIDX(i, j+1, dim)].blue);
           rowsum[i][j].green = (src[RIDX(i, j-1, dim)].green+src[RIDX(i, j,dim)].green+src[RIDX(i, j+1, dim)].green);
           rowsum[i][j].num = 3;
        }
       rowsum[i][dim-1].red = (src[RIDX(i, dim-2, dim)].red+src[RIDX(i, dim-1,dim)].red);
       rowsum[i][dim-1].blue = (src[RIDX(i, dim-2, dim)].blue+src[RIDX(i,dim-1, dim)].blue);
       rowsum[i][dim-1].green = (src[RIDX(i, dim-2, dim)].green+src[RIDX(i,dim-1, dim)].green);
       rowsum[i][dim-1].num = 2;
    }
    for(j=0;j<dim; j++)
    {
        snum =rowsum[0][j].num+rowsum[1][j].num;
        dst[RIDX(0,j, dim)].red = (unsigned short)((rowsum[0][j].red+rowsum[1][j].red)/snum);
        dst[RIDX(0,j, dim)].blue = (unsigned short)((rowsum[0][j].blue+rowsum[1][j].blue)/snum);
        dst[RIDX(0,j, dim)].green = (unsigned short)((rowsum[0][j].green+rowsum[1][j].green)/snum);
        for(i=1;i<dim-1; i++)
        {
           snum =rowsum[i-1][j].num+rowsum[i][j].num+rowsum[i+1][j].num;
           dst[RIDX(i, j, dim)].red = (unsigned short)((rowsum[i-1][j].red+rowsum[i][j].red+rowsum[i+1][j].red)/snum);
           dst[RIDX(i, j, dim)].blue = (unsigned short)((rowsum[i-1][j].blue+rowsum[i][j].blue+rowsum[i+1][j].blue)/snum);
           dst[RIDX(i, j, dim)].green = (unsigned short)((rowsum[i-1][j].green+rowsum[i][j].green+rowsum[i+1][j].green)/snum);
```



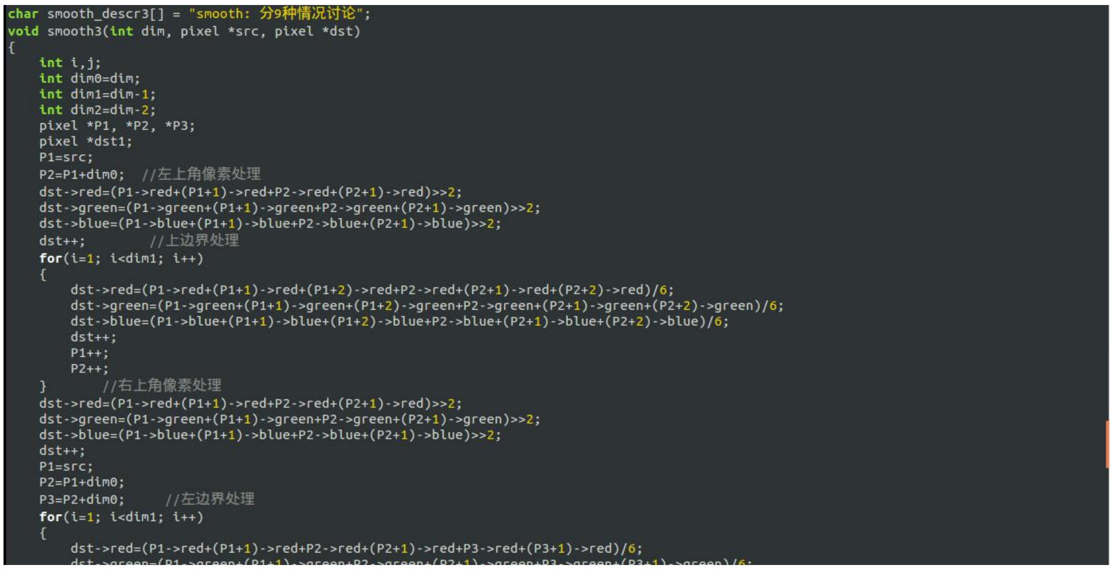
```c
        }
        snum =rowsum[dim-1][j].num+rowsum[dim-2][j].num;
       dst[RIDX(dim-1, j, dim)].red = (unsigned short)((rowsum[dim-2][j].red+rowsum[dim-1][j].red)/snum);
       dst[RIDX(dim-1, j, dim)].blue = (unsigned short)((rowsum[dim-2][j].blue+rowsum[dim-1][j].blue)/snum);
       dst[RIDX(dim-1, j, dim)].green = (unsigned short)((rowsum[dim-2][j].green+rowsum[dim-1][j].green)/snum);
    }
}
```

优点：很大程度上减少了重复计算需要的时间

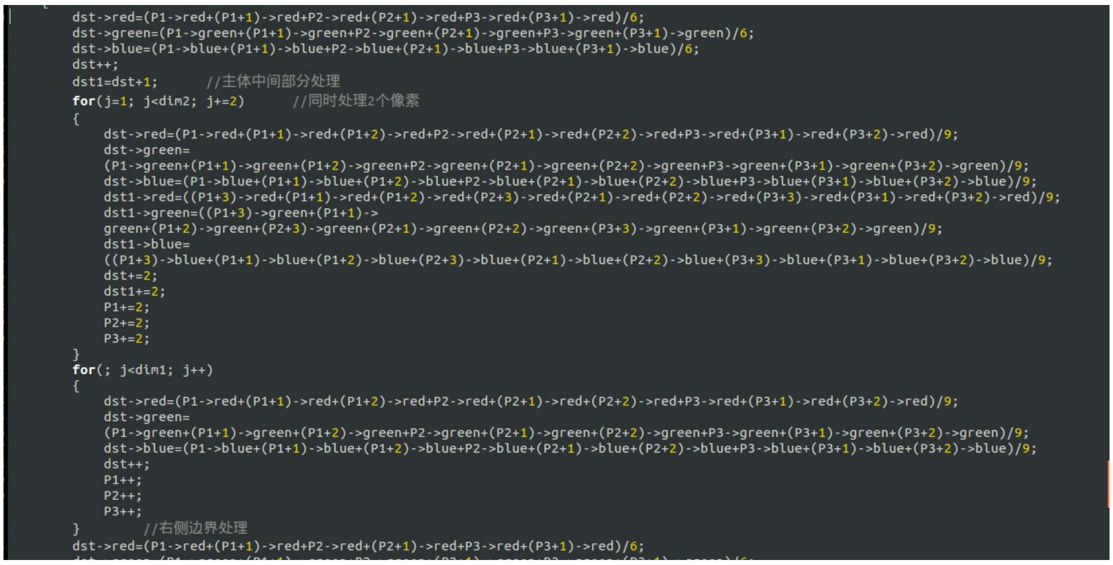
缺点：增大了空间开销，代码可读性降低

（3）循环展开

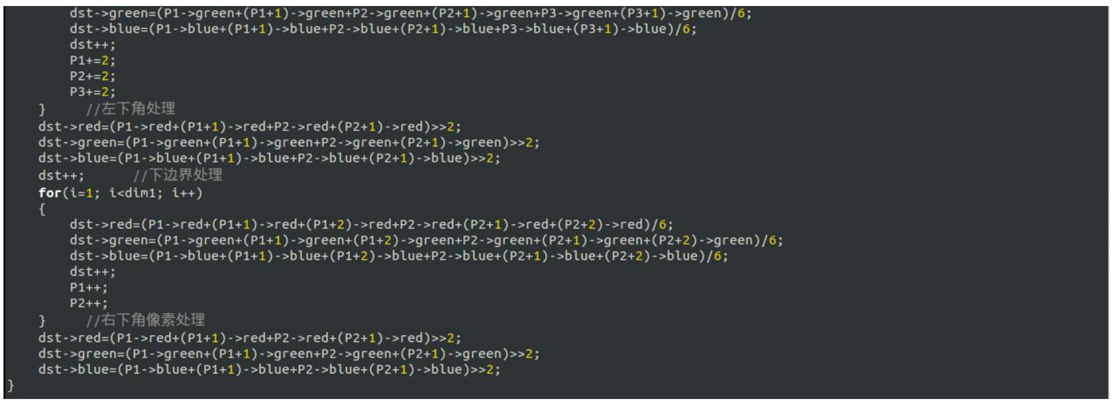
分 9 中情况讨论，通过增加每次迭代计算的元素的数量，减少循环的迭代次数。



```c
char smooth_descr3[] = "smooth: 分9种情况讨论";
void smooth3(int dim, pixel *src, pixel *dst)
{
    int i,j;
    int dim0=dim;
    int dim1=dim-1;
    int dim2=dim-2;
    pixel *P1, *P2, *P3;
    pixel *dst1;
    P1=src;
    P2=P1+dim0;  //左上角像素处理
    dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red)>>2;
    dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green)>>2;
    dst->blue=(P1->blue+(P1+1)->blue+P2->blue+(P2+1)->blue)>>2;
    dst++;          //上边界处理
    for(i=1; i<dim1; i++)
    {
        dst->red=(P1->red+(P1+1)->red+(P1+2)->red+P2->red+(P2+1)->red+(P2+2)->red)/6;
        dst->green=(P1->green+(P1+1)->green+(P1+2)->green+P2->green+(P2+1)->green+(P2+2)->green)/6;
        dst->blue=(P1->blue+(P1+1)->blue+(P1+2)->blue+P2->blue+(P2+1)->blue+(P2+2)->blue)/6;
        dst++;
        P1++;
        P2++;
    }       //右上角像素处理
    dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red)>>2;
    dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green)>>2;
    dst->blue=(P1->blue+(P1+1)->blue+P2->blue+(P2+1)->blue)>>2;
    dst++;
    P1=src;
    P2=P1+dim0;
    P3=P2+dim0;     //左边界处理
    for(i=1; i<dim1; i++)
    {
        dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red+P3->red+(P3+1)->red)/6;
        dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green+P3->green+(P3+1)->green)/6;
```



```c
        dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red+P3->red+(P3+1)->red)/6;
        dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green+P3->green+(P3+1)->green)/6;
        dst->blue=(P1->blue+(P1+1)->blue+P2->blue+(P2+1)->blue+P3->blue+(P3+1)->blue)/6;
        dst++;
        dst1=dst+1;       //主体中间部分处理
        for(j=1; j<dim2; j+=2)      //同时处理2个像素
        {
            dst->red=(P1->red+(P1+1)->red+(P1+2)->red+P2->red+(P2+1)->red+(P2+2)->red+P3->red+(P3+1)->red+(P3+2)->red)/9;
            dst->green=
            (P1->green+(P1+1)->green+(P1+2)->green+P2->green+(P2+1)->green+(P2+2)->green+P3->green+(P3+1)->green+(P3+2)->green)/9;
            dst->blue=(P1->blue+(P1+1)->blue+(P1+2)->blue+P2->blue+(P2+1)->blue+(P2+2)->blue+P3->blue+(P3+1)->blue+(P3+2)->blue)/9;
            dst1->red=((P1+3)->red+(P1+1)->red+(P1+2)->red+(P2+3)->red+(P2+1)->red+(P2+2)->red+(P3+3)->red+(P3+1)->red+(P3+2)->red)/9;
            dst1->green=((P1+3)->green+(P1+1)->
            green+(P1+2)->green+(P2+3)->green+(P2+1)->green+(P2+2)->green+(P3+3)->green+(P3+1)->green+(P3+2)->green)/9;
            dst1->blue=
            ((P1+3)->blue+(P1+1)->blue+(P1+2)->blue+(P2+3)->blue+(P2+1)->blue+(P2+2)->blue+(P3+3)->blue+(P3+1)->blue+(P3+2)->blue)/9;
            dst+=2;
            dst1+=2;
            P1+=2;
            P2+=2;
            P3+=2;
        }
        for(; j<dim1; j++)
        {
            dst->red=(P1->red+(P1+1)->red+(P1+2)->red+P2->red+(P2+1)->red+(P2+2)->red+P3->red+(P3+1)->red+(P3+2)->red)/9;
            dst->green=
            (P1->green+(P1+1)->green+(P1+2)->green+P2->green+(P2+1)->green+(P2+2)->green+P3->green+(P3+1)->green+(P3+2)->green)/9;
            dst->blue=(P1->blue+(P1+1)->blue+(P1+2)->blue+P2->blue+(P2+1)->blue+(P2+2)->blue+P3->blue+(P3+1)->blue+(P3+2)->blue)/9;
            dst++;
            P1++;
            P2++;
            P3++;
        }         //右侧边界处理
        dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red+P3->red+(P3+1)->red)/6;
```

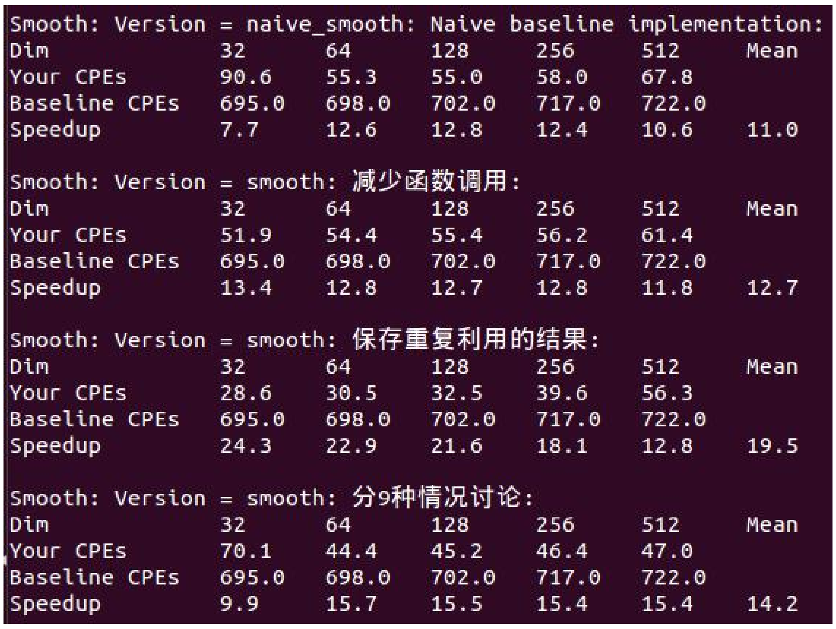


```c
        dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green+P3->green+(P3+1)->green)/6;
        dst->blue=(P1->blue+(P1+1)->blue+P2->blue+(P2+1)->blue+P3->blue+(P3+1)->blue)/6;
        dst++;
        P1+=2;
        P2+=2;
        P3+=2;
    }     //左下角处理
    dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red)>>2;
    dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green)>>2;
    dst->blue=(P1->blue+(P1+1)->blue+P2->blue+(P2+1)->blue)>>2;
    dst++;      //下边界处理
    for(i=1; i<dim1; i++)
    {
        dst->red=(P1->red+(P1+1)->red+(P1+2)->red+P2->red+(P2+1)->red+(P2+2)->red)/6;
        dst->green=(P1->green+(P1+1)->green+(P1+2)->green+P2->green+(P2+1)->green+(P2+2)->green)/6;
        dst->blue=(P1->blue+(P1+1)->blue+(P1+2)->blue+P2->blue+(P2+1)->blue+(P2+2)->blue)/6;
        dst++;
        P1++;
        P2++;
    }     //右下角像素处理
    dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red)>>2;
    dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green)>>2;
    dst->blue=(P1->blue+(P1+1)->blue+P2->blue+(P2+1)->blue)>>2;
}
```

优点：使用指针，能够动态分配空间，减少内存浪费

缺点：没有提高程序的并行性

三种优化方式与优化前对比:



```
Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       90.6    55.3    55.0    58.0    67.8
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         7.7     12.6    12.8    12.4    10.6    11.0

Smooth: Version = smooth: 减少函数调用:
Dim             32      64      128     256     512     Mean
Your CPEs       51.9    54.4    55.4    56.2    61.4
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         13.4    12.8    12.7    12.8    11.8    12.7

Smooth: Version = smooth: 保存重复利用的结果:
Dim             32      64      128     256     512     Mean
Your CPEs       28.6    30.5    32.5    39.6    56.3
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         24.3    22.9    21.6    18.1    12.8    19.5

Smooth: Version = smooth: 分9种情况讨论:
Dim             32      64      128     256     512     Mean
Your CPEs       70.1    44.4    45.2    46.4    47.0
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         9.9     15.7    15.5    15.4    15.4    14.2
```

可以看到，保存重复利用的结果优化程度最高，循环展开，分 9 种情况讨论次之，减少过程调用优化程度最低。