一、 优化程序

在对 avg（）展开的过程中，看到其内部是两个循环，那么问题就来了，可不可以对其内部的循环做循环展开呢?

然而，性能并没有提升。很容易可以看出，avg（）内部的循环是影响 speedup 和 CPE 的关键，而且既然循环展开和分块对其影响不大，那么剩下可以做的，也就只有提高并行能力和换种方式替代循环了。

版本 1：替换 avg 函数（代码一）

如果只是简单的提高并行能力，会发现性能并不会有多大的改善，因为原 avg（）函数内部的双重循环还是没有得到优化。

所以需要通过换一种方式，来替代这个循环。

已经知道 smooth 是要对一个方格，求出其周围所有方格（及其自身）的颜色的平均值赋给该方格。

那么这样，来分析一下 avg（）函数，很容易可以看出，它的功能是求出周围所有方格的平均值。那么既然如此，对于一个方格来说，它周围的方格的数量和颜色都是一个确定的数，知道了这个，就可以通过简单的加法和除法来得到所求的平均值，从而避免了使用 avg（）函数。

接下来再对这个正方形进行分析，可以知道，正方形最特殊的地方就是四个顶点和四条边，这是需要单独进行处理的，对于其他的部分，可以通过循环来解决，而这一个双重循环，循环的次数相比于原先的要少，并且只有一个双重循环，这样就可以达到对代码的优化的效果。

在已经知道周围各点的数量的情况下，也就不再需要对 sum 变量的定义。

```
int i, j;
//对左上角的顶点进行处理
i = 0;
j = 0;
dst[RIDX(0, 0, dim)].red = (unsigned short)(((int)
                    (src[RIDX(0, 0, dim)].red + src[RIDX(0, 1, dim)].red +
                    src[RIDX(1, 0, dim)].red + src[RIDX(1, 1, dim)].red)) / 4);
dst[RIDX(0, 0, dim)].green = (unsigned short)(((int)
                    (src[RIDX(0, 0, dim)].green + src[RIDX(0, 1, dim)].green +
                    src[RIDX(1, 0, dim)].green + src[RIDX(1, 1, dim)].green)) / 4);
dst[RIDX(0, 0, dim)].blue = (unsigned short)(((int)
                    (src[RIDX(0, 0, dim)].blue + src[RIDX(0, 1, dim)].blue +
                    src[RIDX(1, 0, dim)].blue + src[RIDX(1, 1, dim)].blue)) / 4);
```

由于该图像的存储方式为一个二维数组，那么根据当前位置点的坐标，可以很快的找到其周围的各个点，由于在角落里的格子与其相邻的格子只有三个，加上自身也就是 4，所以将他们的 RGB 三个颜色相加并/4，即可得到平均值，并将得到的值赋给 dst 中对应的格子即可。对其与三个点的处理也是如此。

```
//对左边进行处理（不包含左上角和左上角）
j = 0;
for (i = 1; i < dim - 1; i++)
{
    dst[RIDX(i, j, dim)].red = (unsigned short)(((int)(src[RIDX(i - 1, j, dim)].red + src[RIDX(i - 1, j + 1, dim)].red +
                            src[RIDX(i, j, dim)].red + src[RIDX(i, j + 1, dim)].red +
                            src[RIDX(i + 1, j, dim)].red + src[RIDX(i + 1, j + 1, dim)].red)) / 6);
    dst[RIDX(i, j, dim)].green = (unsigned short)(((int)(src[RIDX(i - 1, j, dim)].green + src[RIDX(i - 1, j + 1, dim)].green +
                            src[RIDX(i, j, dim)].green + src[RIDX(i, j + 1, dim)].green +
                            src[RIDX(i + 1, j, dim)].green + src[RIDX(i + 1, j + 1, dim)].green)) / 6);
    dst[RIDX(i, j, dim)].blue = (unsigned short)(((int)(src[RIDX(i - 1, j, dim)].blue + src[RIDX(i - 1, j + 1, dim)].blue +
                            src[RIDX(i, j, dim)].blue + src[RIDX(i, j + 1, dim)].blue +
                            src[RIDX(i + 1, j, dim)].blue + src[RIDX(i + 1, j + 1, dim)].blue)) / 6);
}
```

这是对最左侧一条边的处理，在处理时要注意去掉包含的两个顶点，与对点的处理相同，根据当前的格子的坐标，得到相邻格子的坐标，不过相邻的格子的数量变为了 5，

所以求得 RGB 和之后要除 6 以得到平均值。对剩余三边处理相同。

```
//对内部的各个点进行处理
for (i = 1; i < dim - 1; i++)
{
    for (j = 1; j < dim - 1; j++)
    {
        dst[RIDX(i, j, dim)].red = (unsigned short)(((int)(src[RIDX(i + 1, j, dim)].red + src[RIDX(i + 1, j - 1, dim)].red +
                                   src[RIDX(i, j, dim)].red + src[RIDX(i - 1, j, dim)].red +
                                   src[RIDX(i, j - 1, dim)].red + src[RIDX(i - 1, j - 1, dim)].red +
                                   src[RIDX(i, j + 1, dim)].red + src[RIDX(i - 1, j + 1, dim)].red +
                                   src[RIDX(i + 1, j + 1, dim)].red)) / 9);
        dst[RIDX(i, j, dim)].green = (unsigned short)(((int)(src[RIDX(i + 1, j, dim)].green + src[RIDX(i + 1, j - 1, dim)].green +
                                   src[RIDX(i, j, dim)].green + src[RIDX(i - 1, j, dim)].green +
                                   src[RIDX(i, j - 1, dim)].green + src[RIDX(i - 1, j - 1, dim)].green +
                                   src[RIDX(i, j + 1, dim)].green + src[RIDX(i - 1, j + 1, dim)].green +
                                   src[RIDX(i + 1, j + 1, dim)].green)) / 9);
        dst[RIDX(i, j, dim)].blue = (unsigned short)(((int)(src[RIDX(i + 1, j, dim)].blue + src[RIDX(i + 1, j - 1, dim)].blue +
                                   src[RIDX(i, j, dim)].blue + src[RIDX(i - 1, j, dim)].blue +
                                   src[RIDX(i, j - 1, dim)].blue + src[RIDX(i - 1, j - 1, dim)].blue +
                                   src[RIDX(i, j + 1, dim)].blue + src[RIDX(i - 1, j + 1, dim)].blue +
                                   src[RIDX(i + 1, j + 1, dim)].blue)) / 9);
    }
}
```

最后便是对内部的各个点进行处理，二重循环得到每一个点的平均值，周围的格子数为 8，最后将平均值赋给 dst 对应的格子。

```
Smooth: Version = naive_smooth: Naive baseline implementation:
Dim              32      64      128     256     512     Mean
Your CPEs        35.9    36.2    36.3    36.5    35.9
Baseline CPEs    695.0   698.0   702.0   717.0   722.0
Speedup          19.3    19.3    19.3    19.7    20.1    19.6

Smooth: Version = smooth: Current working version:
Dim              32      64      128     256     512     Mean
Your CPEs        10.6    10.6    10.8    10.8    10.9
Baseline CPEs    695.0   698.0   702.0   717.0   722.0
Speedup          65.9    65.8    65.3    66.4    66.4    65.9
```

得到的 speedup 以及 CPE 的提升都是非常的显著的，speedup 之比达到了 3.36

版本 2：替换 avg 函数基础上循环展开（代码二）

在做版本一的时候，写到对内部的点进行处理的时候，就在想，可不可以对他进行循环展开来优化，使得效率更加提升，因为在做 rotate 是使用循环展开得到的优化效果还是很不错的，但是在刚开始做 smooth 时，使用的循环展开由于 avg 函数的时间复杂度过高，所以效果微乎其微。

但是在做完版本一以后，就发现似乎可以进行循环展开了，于是就开始着手做这一部分。

变量的定义及含义：

```
//对内部各个点进行处理
//用于找出临近的色块，共指示四行，可以同时处理两个内部的点
pixel *pixelA = &src[0];
pixel *pixelB = &src[dim];
pixel *pixelC = &src[dim+dim];
pixel *pixelD = &src[dim+dim+dim];
//每一个sum存储竖着相邻的三个色块的各个颜色值的和
//其中sum0的靠下的两块和sum3靠上的两块相同，1和4，2和5同理
int sum0_red, sum0_green, sum0_blue;
int sum1_red, sum1_green, sum1_blue;
int sum2_red, sum2_green, sum2_blue;
int sum3_red, sum3_green, sum3_blue;
int sum4_red, sum4_green, sum4_blue;
int sum5_red, sum5_green, sum5_blue;
//指示要改变颜色（写入）的色块的标号
//index_firstline为第一行，index_secondline为第二行
int index_firstline = dim+1;
int index_secondline = index_firstline+dim;
```

初始化，给 sum 进行赋值，剩余还有两个相同的部分，代码段与循环内部相同，只不过第二段把 0 和 3 变为 1 和 4，第三段变为 2 和 5

```
//循环展开:2，每次两行
for (i = 1; i < dim - 2; i += 2) {
    sum0_red = pixelB->red;
    sum0_blue = pixelB->blue;
    sum0_green = pixelB->green;
    sum0_red += pixelC->red;
    sum0_blue += pixelC->blue;
    sum0_green += pixelC->green;
    sum3_red = sum0_red+pixelD->red;
    sum3_green = sum0_green+pixelD->green;
    sum3_blue = sum0_blue+pixelD->blue;
    sum0_red += pixelA->red;
    sum0_blue += pixelA->blue;
    sum0_green += pixelA->green;
    //右移，继续给1和4赋值
    pixelA++;
    pixelB++;
    pixelC++;
    pixelD++;
```

将赋完值得 sum 写入 index 指向的第一行和第二行，并继续右移。

```
dst[index_firstline].red = ((sum0_red+sum1_red+sum2_red)/9);
dst[index_firstline].blue = ((sum0_blue+sum1_blue+sum2_blue)/9);
dst[index_firstline].green = ((sum0_green+sum1_green+sum2_green)/9);
index_firstline++;
dst[index_secondline].red = ((sum3_red+sum4_red+sum5_red)/9);
dst[index_secondline].blue = ((sum3_blue+sum4_blue+sum5_blue)/9);
dst[index_secondline].green = ((sum3_green+sum4_green+sum5_green)/9);
index_secondline++;
//右推
sum0_red = sum1_red;
sum1_red = sum2_red;
sum0_green = sum1_green;
sum1_green = sum2_green;
sum0_blue = sum1_blue;
sum1_blue = sum2_blue;
sum3_red = sum4_red;
sum4_red = sum5_red;
sum3_green = sum4_green;
sum4_green = sum5_green;
sum3_blue = sum4_blue;
sum4_blue = sum5_blue;
```

将剩余的部分重复用循环进行赋值移动操作，并循环展开四次。下面的代码段（图 1）仅展示第一次操作，剩余还有三次，与当前代码段内容相同。

由于跨度为 4，可能还有未能处理的色块，因此将步长设置为 1，也就是最普通的循环，继续进行处理赋值，代码与该段相同，只不过循环条件不同，如下

```
//处理这一行没处理完的像素点
for (; j < dim-1; j++) {
```

最后改变指针的指向，对之后得行进行赋值

```
        //改变指针，指向下一行
        pixelA += dim;
        pixelB += dim;
        pixelC += dim;
        pixelD += dim;
        index_firstline += dim+2;
        index_secondline += dim+2;
    }
}
```

可以看到（见下页图 2），在版本一的基础上进行循环展开还是有效果的，性能提升到 76.3，比版本一的 speedup 还要高。如果将展开的步长扩大，性能或许还会有提高，不过代码可能会更加的冗长，所以就没有去尝试实现。

在性能提升的同时，也大大的降低了代码的可读性。

```
//循环展开:4，每一次：更新值+指针右移+赋值+临时变量右推
for (j = 1; j < dim-4; j += 4) {
    //第一次
    sum2_red = pixelB->red;
    sum2_blue = pixelB->blue;
    sum2_green = pixelB->green;
    sum2_red += pixelC->red;
    sum2_blue += pixelC->blue;
    sum2_green += pixelC->green;
    sum5_red = sum2_red+pixelD->red;
    sum5_green = sum2_green+pixelD->green;
    sum5_blue = sum2_blue+pixelD->blue;
    sum2_red += pixelA->red;
    sum2_blue += pixelA->blue;
    sum2_green += pixelA->green;
    pixelA++;
    pixelB++;
    pixelC++;
    pixelD++;

    dst[index_firstline].red = ((sum0_red+sum1_red+sum2_red)/9);
    dst[index_firstline].blue = ((sum0_blue+sum1_blue+sum2_blue)/9);
    dst[index_firstline].green = ((sum0_green+sum1_green+sum2_green)/9);
    index_firstline++;

    dst[index_secondline].red = ((sum3_red+sum4_red+sum5_red)/9);
    dst[index_secondline].blue = ((sum3_blue+sum4_blue+sum5_blue)/9);
    dst[index_secondline].green = ((sum3_green+sum4_green+sum5_green)/9);
    index_secondline++;

    sum0_red = sum1_red;
    sum1_red = sum2_red;
    sum0_green = sum1_green;
    sum1_green = sum2_green;
    sum0_blue = sum1_blue;
    sum1_blue = sum2_blue;
    sum3_red = sum4_red;
    sum4_red = sum5_red;
    sum3_green = sum4_green;
    sum4_green = sum5_green;
    sum3_blue = sum4_blue;
    sum4_blue = sum5_blue;
```

图 1

```
Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       34.7    35.2    35.2    35.8    35.7
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         20.0    19.8    19.9    20.1    20.2    20.0

Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       9.5     9.2     9.1     9.2     9.3
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         73.0    75.5    76.7    78.3    78.0    76.3
```

图 2

二、 分析总结

Part A:

在对 rotate 进行优化时，使用了三种策略：循环分块，循环展开配合读写顺序的调整，循环展开根据指针进行读写，三者的共同优点就是都提高了程序的性能，其中，采用循环分块的代码与其他两个相比，更好理解，可读性更强，但是性能却不如其余两种，只是做到了对 cache 友好，却未能大幅的消除循环的低效性；对于第二种，循环展开配合读写顺序的调整，则使性能有了一定的飞跃的提升，通过牺牲代码的可读性，来消除了循环的低效性，并且同时，改善了 cache 的写命中，使得惩罚降低，性能有了进一步的提升，不过这种方法，对于之后如果要进行代码的维护，难免会有一定的阻碍。而最后一种，循环展开根据指针进行读写，则在第二种的基础上，又消除了对函数的引用，直接对单元进行读取和写入，可以

再次提升性能，但是提升相对较小，缺点也是和第二种一样，给代码的维护和修改增加了阻碍，降低了可读性。

PartB：

对 smooth 进行优化时，采用了三种策略：减少过程调用，将 avg 函数循环分块，消除 avg 函数的基础上循环展开。共同优点都是同样的，提高了程序的性能。对于第一种减少过程调用，虽然使得原来的函数趋于冗长，但是却保留了源代码的可读性，便于理解，但是对性能的提升却不明显；第二种，将 avg 函数循环分块，直接不再使用原有的 avg 函数，而是将四个角，四个边以及内部的各点进行单独的操作，通过这样分块的方法，提高了程序的性能，有了一个非常大的提升，虽然使得修改后的程序于源程序相比，篇幅过于长，但是整体看来，还是比较条理清晰，易于理解的，可读性虽然有下降，但是依旧不难理解；对于第三种，就是在第二种的基础上进行循环展开，很显然的，第二种方法中对于图形内部的点的处理，还是有一个二重循环的，如果要对他进行展开，那么对程序的性能也会有一定的提升，这样做使得程序性能在第二种的情况下再次有了一定的提升，但是带来的代价，确实代码的可读性大幅度降低，然而这仅仅是在进行两次循环展开的情况下，倘若将展开的次数变多，那么性能也应该会有提升，但是代码将变得非常的冗长繁杂，难于阅读。

Amadal 定律：

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.7     2.3     4.3     8.6     8.1
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.8     17.5    10.8    7.6     11.7    10.8

Rotate: Version = rotate1: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.5     1.7     1.8     3.0
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         10.0    26.5    28.0    35.7    31.3    24.2

Rotate: Version = rotate2: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.5     1.5     1.5     2.4
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         10.1    27.3    31.7    43.9    38.9    27.2

Rotate: Version = rotate3: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.5     1.4     1.5     2.4
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         10.1    27.5    32.2    44.7    39.4    27.5
```

```
Smooth: Version = naive_smooth: Naive baseline implementation
Dim             32      64      128     256     512     Mean
Your CPEs       53.5    54.6    35.3    35.7    36.3
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         13.0    12.8    19.9    20.1    19.9    16.8

Smooth: Version = smooth1: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       35.3    35.5    35.6    35.6    36.5
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         19.7    19.7    19.7    20.1    19.8    19.8

Smooth: Version = smooth2: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       10.5    10.6    10.7    10.8    10.9
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         66.2    65.8    65.4    66.7    66.2    66.0

Smooth: Version = smooth3: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       9.3     9.3     9.1     9.1     9.5
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         75.0    75.3    76.7    78.4    75.8    76.2
```

左侧是 rotate 函数的各个版本的加速比，rotate 函数内是一个二重循环，做的是旋转的操作，由于每次操作之间没有关联性，所以可以进行循环展开，在一次循环内进行多次操作，可以显著的降低循环的低效性；而保证写命中的惩罚要比读命中的惩罚高，所以使写命中可以充分的利用 cache，通过各个版本的 Speedup 看出，影响最大的部分就是对于 cache 的利用，在循环展开时保证写命中能够使性能达到最佳。

右侧是 smooth 函数的各个版本的加速比，smooth 函数是一个二重循环内调用 avg 函数，而 avg 函数则又调用了其他的函数且内部有一个二重循环，且之前的操作会对当前操作有影响，并且经过多次尝试，通过各个版本的 Speedup 可以看出，在修改了 avg 函数后，性能有了非常大的提升，所以影响性能的关键就是 avg 函数的性能，当对 avg 函数进行修改，处理四个角和四个边后，再对内部进行处理，并且循环展开，使性能达到最佳。