一、入手方向

1. 从上周的学习中可以知道，要是想提高程序的性能对其优化，可以从三个方面入手：1.消除循环的低效率 2.减少过程调用 3.消除不必要的存储器引用。而 5.8 节着重于循环展开。
2. 通过查看 kernels.c 的 rotate 函数，可以看到是由两层循环构成，对于过程的引用只有 RIDX（）。那么，要是想优化该程序，则需要消除循环的低效率或者是减少过程的调用，或是消除不必要的存储器。
3. 通过查看 kernels.c 的 smooth 函数，可以看到是由两层循环构成，对于过程的引用除了影响不大的 RIDX（）外，还有一个 avg（）函数。
4. 消除循环的低效率的方法：交换内外循环的次序，循环展开和循环分块处理，本次着重于通过循环展开来进行优化。

二、优化程序

版本一：对 rotate 进行循环展开

记得在上周的 rotate 的优化过程中尝试过循环展开，并且保证写的命中，可以得到一个很好的效率，而且在不断地尝试的过程中，还发现了对内层循环（j）的展开对程序的影响不大。为了保证正确性，于是我便单独测试了一下对 j 展开时对程序性能有无提升，事实是确实没有影响。那么如何对程序继续从循环展开方面进行优化呢，这是一个问题。

由于我上周尝试了很多方法，选择了当前已知的尝试过的性能最佳的两个方法放在了报告中，于是我本周的实验或许再次提升性能。不过，在上周的实验中，有一个优化角度我是没有尝试的：消除不必要的存储器引用。因为当时想尝试查看汇编代码来对从该方面进行优化，但是未能成功。不过再一想，如果直接向内存写入，不就可以进行优化了吗？

```
char rotate_descr[] = "rotate: Current working version";
void rotate(int dim, pixel *src, pixel *dst)
{
    int i,j;
    dst+=RIDX(dim-1,0,dim);//将dst定位到src[0]要写入的位置
    for(i=0; i< dim; i+=2){//展开的跨度为2
        for(j=0;j<dim;j++){
            *dst=*src;//src写入dst
            dst++;src+=dim;//dst顺序进入到下一位置，src进入到对应位置
            *dst=*src;
            src++;//src跳转到下一列 记跳转前位置为A
            src-=dim;//src返回到该列的起点
            dst-=dim+1;//dst跳转到当前写入的上一行 记跳转前位置为B
        }
        src+=dim;//src回到第一次内层循环断开的位置即位置A继续
        dst+=dim*dim+2;//dst回到第一次内层循环断开的位置即位置B继续
    }
}
```

这是在消除了不必要的存储器，直接对其地址操作并且进行了简单的循环展开后的代码，虽然较为晦涩难懂，但还是有规律可循的，这样得到的效率有所提升。

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.7     2.4     5.0     8.6     7.8
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.8     17.0    9.3     7.6     12.1    10.5

Rotate: Version = rotate: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.6     2.0     2.8     4.4     6.2
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.5     20.1    16.3    15.1    15.4    14.8
```

经过上周的实验，很显然的可以知道，2 的跨度不可能是最优的效率，所以进行多次尝试，每次的跨度均为 2 的倍数，一直达到 32，当超过 32 时，就会产生错误。

经过多次测试，果然如我所料，当跨度为 16 时，性能可以达到最佳。

```
47 char rotate_descr[] = "rotate: Current working version";
48 void rotate(int dim, pixel *src, pixel *dst)
49 {
50     int i,j;
51     dst+=RIDX(dim-1,0,dim);//将dst定位到src[0]要写入的位置
52     for(i=0; i< dim; i+=16){//展开的跨度为2
53         for(j=0;j<dim;j++){
54             *dst=*src;//src写入dst
55             dst++;src+=dim;//dst顺序进入到下一位置，src进入到对应位置
56             *dst=*src;
57             dst++;src+=dim;
58             *dst=*src;
59             dst++;src+=dim;
82             *dst=*src;
83             dst++;src+=dim;
84             *dst=*src;
85             src++;//src跳转到下一列 记跳转前位置为A(指第一次循环发生时,j==0)
86             src-=dim*15;//src返回到该列的起点
87             dst-=dim+15;//dst跳转到当前写入的上一行 记跳转前位置为B (指第一次循环发生时,j==0)
88         }
89         src+=dim*15;//src回到第一次内层循环断开的位置即位置A继续
90         dst+=dim*dim+16;//dst回到第一次内层循环断开的位置即位置B继续
91     }
92 }
```

中间部分为 56-57 行的重复，加上已有的共有 16 组。具体解释如前相同。

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim              64      128     256     512     1024    Mean
Your CPEs        1.7     2.4     5.0     8.4     8.3
Baseline CPEs    14.7    40.1    46.4    65.9    94.5
Speedup          8.9     17.0    9.3     7.8     11.4    10.5

Rotate: Version = rotate: Current working version:
Dim              64      128     256     512     1024    Mean
Your CPEs        1.4     1.4     1.4     1.8     2.3
Baseline CPEs    14.7    40.1    46.4    65.9    94.5
Speedup          10.2    27.8    32.5    36.8    41.5    26.9
```

可以看到当前的 speedup 与原相比，比值为 2.56，效率也是非常好。

版本二：对 smooth 进行循环展开

在做 rotate 时，已经见识到了循环展开的巨大魅力，屡试不爽，对程序的性能的提升可谓是巨大的。所以也就如法炮制，对 smooth 也进行如此的展开，主要是对 i 进行展开，先尝试在跨度为 16 时的效果（因为之前的尝试已经多次验证了 16 时是最优化的性能）

```
char smooth_descr[] = "smooth: Current working version";
void smooth(int dim, pixel *src, pixel *dst) {
    int i, j;
    for (i = 0; i < dim; i+=16)
    for (j = 0; j < dim; j++){
        dst[RIDX(i, j, dim)] = avg(dim, i, j, src);
        dst[RIDX(i+1, j, dim)] = avg(dim, i+1, j, src);
        dst[RIDX(i+2, j, dim)] = avg(dim, i+2, j, src);
```

省略部分为最后一行的重复，共 16 次，并且+数每行+1

然而对 smooth 采取最简单的循环展开，却难以很好的体现其性能的“飞跃”，仅仅是 speedup 增加了 0.x 而已，而且显而易见的是，在 dim 较小的时候，CPE 有所下降，但是幅度不大，也是微乎其微的，但是在 dim 较大的时候，CPE 却会被反向优化，导致了 CPE 的增加，这是不愿看到的。

```
Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       41.3    41.9    38.8    42.6    44.1
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         16.8    16.7    18.1    16.8    16.4    17.0

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       41.4    42.2    42.4    42.1    42.9
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         16.8    16.5    16.6    17.0    16.8    16.8
```

不过 smooth 与 rotate 的不同，在于其内部有一个 avg（）的函数调用，而且 avg（）内部还有一个两层循环，所以要是想要提升该程序的性能，应当从 avg 函数入手，改善其内部的性能，降低时间复杂度，以达到目的。

直接将 avg（）实现的功能 copy 到 smooth 函数内，并且将 avg（）内引用的函数的功能一并直接调用。于此同时还发现，在 avg（）内还有两个循环，并且每次循环都调用了 max 和 min 函数，通过阅读教材可知，在循环时的调用会影响程序运行速度，于是将其在循环前，便赋值给一个临时变量，进一步减少过程的调用。

```
238 void smooth(int dim, pixel *src, pixel *dst)
239 {
240     int i, j;
241     for (i = 0; i < dim; i++)
242     for (j = 0; j < dim; j++){
243         int ii, jj;
244         pixel_sum sum;
245         pixel current_pixel;
246         //initialize_pixel_sum(&sum);
247         sum.red=sum.green=sum.blue=0;
248         sum.num=0;
249         //临时变量
250         int ti1=max(i-1, 0);
251         int ti2=min(i+1, dim-1);
252         int tj1=max(j-1, 0);
253         int tj2=min(j+1, dim-1);
254         for(ii = ti1; ii <= ti2; ii++)
255         for(jj = tj1; jj <= tj2; jj++) {
256             //accumulate_sum(&sum, src[RIDX(ii, jj, dim)]);
257             sum.red+=(int)src[RIDX(ii, jj, dim)].red;
258             sum.green+=(int)src[RIDX(ii, jj, dim)].green;
259             sum.blue+=(int)src[RIDX(ii, jj, dim)].blue;
260             sum.num++;
261         }
262         //assign_sum_to_pixel(&current_pixel, sum);
263         current_pixel.red=(unsigned short)(sum.red/sum.num);
264         current_pixel.green=(unsigned short)(sum.green/sum.num);
265         current_pixel.blue=(unsigned short)(sum.blue/sum.num);
266         dst[RIDX(i, j, dim)] = current_pixel;
267     }
268 }
```

各个部分调用的函数的展开部分的原调用函数以注释形式给出。

```
Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       35.1    35.4    35.6    36.1    36.4
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         19.8    19.7    19.7    19.9    19.8    19.8

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       41.5    42.2    42.4    43.5    43.8
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         16.8    16.5    16.6    16.5    16.5    16.6
```

可以看到，通过这种减少过程调用来得到的函数，确实可以起到对代码进行优化的作用，但是效果却没有令人惊叹。