Lab4.perflab

1_rotate:

```
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;
}
```

The above code scans the rows of the source image matrix, copying to the columns of the destination image matrix. Your task is to rewrite this code to make it run as fast as possible using techniques like code motion, loop unrolling and blocking.

See the file `kernels.c` for this code.

注意到原始的实现是按行遍历 src 然后将每一行写入到 dst 每一列中. 由于写入列的时候每个元素在内存中距离较远, 并且写的速度要慢于读的速度. 所以我们牺牲读取的局部性, 将行列对调, 并且提前算好一些常量值得到初步优化的 rotate1 方案:

```
char rotate1_descr[] = "rotate1: low performance";
void rotate1(int dim, pixel *src, pixel *dst){
    int i, j, a;
    for (j = 0; j < dim; ++ j)
    {
        a = (dim - 1 - j) * dim;
        for (i = 0; i < dim; ++ i)
        {
            dst[a + i] = src[i * dim + j];
        }
    }
}
```

再仔细观察初步优化的版本, 虽然写入 dst 的局部性好了, 但是读取 src 是按照列读取的. 所以读取数据的延迟可能形成一条关键路径. 我们可以增加每次读取和写入次数, 即每次从 src 读取 k 列并在 dst 写入 k 行. 即使写入的速度有所牺牲, 但是其延迟只要不超过读取的延迟, 整体的性能就会有所提升, 经过不断地测试, k 取 4 时性能提升最好, 以下是最终的优化版本 rotate:

```
/*
 * rotate - Your current working version of rotate
 * IMPORTANT: This is the version you will be graded on
 */

char rotate_descr[] = "rotate: higher performance";
```

```
void rotate(int dim, pixel *src, pixel *dst)
{
    int i, j, a;
    for (j = 0; j + 3 < dim; j += 4)
    {
        a = (dim - 1 - j) * dim;
        for (i = 0; i < dim; ++i){
            dst[a + i]           = src[i * dim + j];
            dst[a + i - dim]     = src[i * dim + j + 1];
            dst[a + i - 2 * dim] = src[i * dim + j + 2];
            dst[a + i - 3 * dim] = src[i * dim + j + 3];
        }
    }
    for (; j < dim; ++j){
        a = (dim - 1 - j) * dim;
        for (int i = 0; i < dim; ++i){
            dst[a + i] = src[i * dim + j];
        }
    }
}
```

最终测试结果:

```
sh-4.4# ./driver
Teamname: ljh_team
Member 1: ljh
Email 1: ljh@abc.edu

Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.7     2.4     4.8     8.6     7.9
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.8     16.7    9.6     7.7     12.0    10.6

Rotate: Version = rotate1: low performance:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.6     1.7     2.3     3.5     4.5
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.0     23.1    19.9    19.0    21.2    17.5

Rotate: Version = rotate: higher performance:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.7     1.9     2.2     2.6     3.2
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.5     20.9    21.0    25.1    29.1    19.4
```

2_smooth

The smoothing function takes as input a source image `src` and returns the smoothed result in the destination image `dst`. Here is part of an implementation:

```
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;
}
```

The function `avg` returns the average of all the pixels around the `(i,j)` th pixel. Your task is to optimize `smooth` (and `avg`) to run as fast as possible. (*Note:* The function `avg` is a local function and you can get rid of it altogether to implement `smooth` in some other way.)

这题主要是用一个像素点四周的值取均值来取代原先的值. Naïve_smooth 的实现调用了大量的函数, 会导致性能下降严重. 同时在取均值的时候对于方阵边界的特判在非边界的情况也会去判断. 所以我们可以通过将所有函数都拆解出来, 并且对于边缘的特判也去掉, 具体来说就是将它从循环中取出来单独处理. 实现如下:

```
char smooth_descr[] = "smooth: Current working version";
void smooth(int dim, pixel *src, pixel *dst)
{
    int i, j, ii, jj, curr;
    for (i = 1 ; i < dim - 1 ; i++) {
        for (j = 1 ; j < dim - 1 ; j++) {
        curr = i * dim + j;
        dst[curr].red = (src[curr].red + src[curr - 1].red + src[curr + 1].red + src[curr - dim].red + src[curr - dim - 1].red + src[curr - dim + 1].red + src[curr + dim].red + src[curr + dim - 1].red + src[curr + dim + 1].red) / 9;
        dst[curr].green = (src[curr].green + src[curr - 1].green + src[curr + 1].green + src[curr - dim].green + src[curr - dim - 1].green + src[curr - dim + 1].green + src[curr + dim].green + src[curr + dim - 1].green + src[curr + dim + 1].green) / 9;
        dst[curr].blue = (src[curr].blue + src[curr - 1].blue + src[curr + 1].blue + src[curr - dim].blue + src[curr - dim - 1].blue + src[curr - dim + 1].blue + src[curr + dim].blue + src[curr + dim - 1].blue + src[curr + dim + 1].blue) / 9;
        }
    }
    int limit;

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

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

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

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

    dst[0].red   = (src[0].red + src[1].red + src[dim].red + src[dim + 1].red) >> 2;
    dst[0].green = (src[0].green + src[1].green + src[dim].green + src[dim + 1].green) >> 2;
    dst[0].blue  = (src[0].blue + src[1].blue + src[dim].blue + src[dim + 1].blue) >> 2;

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

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

```
    dst[curr].blue  = (src[curr].blue + src[curr + 1].blue + src[curr - dim].blue + src[curr
- dim + 1].blue) >> 2;

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

得到运行结果如下:

```
Smooth: Version = smooth: Current working version:
Dim             32      64      128     256     512     Mean
Your CPEs       11.1    11.7    11.8    11.9    11.9
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         62.8    59.8    59.6    60.3    60.8    60.6

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       47.4    49.5    49.6    50.1    50.3
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         14.7    14.1    14.2    14.3    14.4    14.3
```

总的结果如下:

```
Summary of Your Best Scores:
  Rotate: 19.6 (rotate: higher performance)
  Smooth: 60.6 (smooth: Current working version)
```