哈尔滨工业大学（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验设计报告

开课学期：2025年春季
课程名称：计算机系统
实验名称：Lab2 Perflab
实验性质：课内实验
实验时间：地点：T2507
学生班级：20233119
学生学号：2023313409
学生姓名：房效民
评阅教师：
报告成绩：

实验与创新实践教育中心印制
2025年4月

1. 实验内容

（描述关键优化技术及实现细节）

1. 图像旋转方法与优化策略

原始代码分析：

```c
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(j, dim - 1 - i, dim)] = src[RIDX(i, j, dim)];
}
```

问题：

1. i和j循环顺序不对，写访存不是按行读取而是按列读取，效率低。如果改成写访存按列读取，而读访存按行读取会更好，因为写cache还涉及写回与写访存等，明显效率不如读cache
2. 有一些临时变量多次计算没有必要

一些其他的性能提升方法：

1. 使用指针访存，防止频繁地址访问
2. 充分利用缓存，每次按块处理，一个BLOCK可以定义为32
3. 循环展开

我的修改：（已添加详细注释）

```c
void rotate(int dim, pixel *src, pixel *dst)
{
    //naive_rotate(dim, src, dst);
    // TODO 实现优化后的rotate方法
    int i,j;    //定义，两个循环变量
    for (i = 0; i < dim; i += BLOCK) {   //每次取BLOCK*BLOCK个大小的矩阵进行计算
        for (j = 0; j < dim; j += BLOCK) {
            // 处理每个块
            int jj;   //先索引列，因为写入时可以做到按行访问，避免较大程度的跳跃降低性能
            for (jj = j; jj < j + BLOCK && jj < dim; jj++) {
                int a= jj*dim;   //代码移动，减少计算次数
                pixel *src_row = src + jj;   //采用指针地址索引，避免复杂的地址计算
                pixel *dst_col = dst + a;
                int ii;   //4x1循环展开，较大程度提高程序性能
                for (ii = i; ii < i + BLOCK-3 && ii < dim; ii+=4) {
                    dst_col[dim-1-ii] = src_row[ii*dim];
                    dst_col[dim-2-ii] = src_row[(ii+1)*dim];
                    dst_col[dim-3-ii] = src_row[(ii+2)*dim];
                    dst_col[dim-4-ii] = src_row[(ii+3)*dim];
                }
                for(;ii<i+BLOCK;ii++)   //末尾处理
                {
                    dst_col[dim-1-ii] = src_row[ii*dim];
                }
            }
        }
    }
}
```

2. 图像平滑方法与优化策略

原始代码分析：

```
void naive_smooth(int dim, pixel *src, pixel *dst)
{
    int i, j;

    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            // 调用 avg 函数计算当前像素 (i, j) 的平滑值
            // 并将结果存储到目标图像中对应位置
            dst[RIDX(i, j, dim)] = avg(dim, i, j, src);

}
```

没有在函数体内写代码，而是直接函数调用，开销很大

```
static pixel avg(int dim, int i, int j, pixel *src)
{
    int ii, jj;
    pixel_sum sum;
    pixel current_pixel;

    initialize_pixel_sum(&sum);
    for (ii = max(i - 1, 0); ii <= min(i + 1, dim - 1); ii++)
    {
        for (jj = max(j - 1, 0); jj <= min(j + 1, dim - 1); jj++)
        {
            if (kernel_array[ii - i + 1][jj - j + 1] == 1)
            {
                accumulate_sum(&sum, src[RIDX(ii, jj, dim)]);
            }
        }
    }

    assign_sum_to_pixel(&current_pixel, sum);
    return current_pixel;
}
```

1. 频繁调用max, min,accumulate等函数，开销很大
2. 总共进行n^2次的分支预测，分支预测惩罚很高

优化思路：

1. 几乎不使用函数调用，全部写在一个函数内
2. 把一些临时变量移到循环外

3. 不使用任何条件判断语句，手动进行边界检查与卷积核计算
我生成的卷积核如下：

111
010
010

因此每对一个点进行运算，我只需要取其左上，上，右上，下，以及本身的颜色值进行加和。
我的代码：（注释已经写得很详细了，可以放大观看）

```
void smooth(int dim, pixel *src, pixel *dst)
{
    //naive_smooth(dim, src, dst);
    // TODO 实现优化后的smooth方法
    int i, j;
    pixel current_pixel;
    for (i = 1; i < dim-1; i++){    //先处理除边框可四角之外的中间部分
        int t=dim*i;    //代码移动，避免重复计算
        int a=(i-1)*dim;
        int b=(i+1)*dim;
        for (j = 1; j < dim-1; j++){
            pixel_sum sum;
            initialize_pixel_sum(&sum);    //去除条件判断，直接计算求和获取结果
            current_pixel.red = (unsigned short)((((int)src[a+j-1].red+(int)src[a+j].red+(int)src[a+j+1].red+(int)src[t+j].red+(int)src[b+j].red)/5);
            current_pixel.green = (unsigned short)((((int)src[a+j-1].green+(int)src[a+j].green+(int)src[a+j+1].green+(int)src[t+j].green+(int)src[b+j].green) / 5);
            current_pixel.blue = (unsigned short)((((int)src[a+j-1].blue+(int)src[a+j].blue+(int)src[a+j+1].blue+(int)src[t+j].blue+(int)src[b+j].blue) /5);
            dst[t+j] = current_pixel;
        }
    }
```

```
for(j = 1; j < dim-1; j++)      //处理上下两边，除去四角
    {
        int a=dim+j;
        int b=(dim-2)*dim;
        int c=(dim-1)*dim;
        int d=j-1;
        int e=j+1;
        current_pixel.red = (unsigned short)((((int)src[j].red+(int)src[a].red)/2);
        current_pixel.green = (unsigned short)((((int)src[j].green+(int)src[a].green)/2);
        current_pixel.blue = (unsigned short)((((int)src[j].blue+(int)src[a].blue)/2);
        dst[j] = current_pixel;
        current_pixel.red = (unsigned short)((((int)src[b+d].red+(int)src[b+j].red+(int)src[b+e].red+(int)src[c+j].red)/4);
        current_pixel.green = (unsigned short)((((int)src[b+d].green+(int)src[b+j].green+(int)src[b+e].green+(int)src[c+j].green)/4);
        current_pixel.blue = (unsigned short)((((int)src[b+d].blue+(int)src[b+j].blue+(int)src[b+e].blue+(int)src[c+j].blue)/4);
        dst[c+j] = current_pixel;
    }
    for(i = 1; i < dim-1; i++)    //处理左右两边，除去四角
    {
        int a=(i-1)*dim;
        int b=i*dim;
        int c=(i+1)*dim;
        current_pixel.red = (unsigned short)((((int)src[a].red+(int)src[a+1].red+(int)src[b].red+(int)src[c].red)/4);
        current_pixel.green = (unsigned short)((((int)src[a].green+(int)src[a+1].green+(int)src[b].green+(int)src[c].green)/4);
        current_pixel.blue = (unsigned short)((((int)src[a].blue+(int)src[a+1].blue+(int)src[b].blue+(int)src[c].blue)/4);
        dst[b] = current_pixel;
        current_pixel.red = (unsigned short)((((int)src[a+dim-2].red+(int)src[a+dim-1].red+(int)src[b+dim-1].red+(int)src[c+dim-1].red)/4);
        current_pixel.green = (unsigned short)((((int)src[a+dim-2].green+(int)src[a+dim-1].green+(int)src[b+dim-1].green+(int)src[c+dim-1].green)/4);
        current_pixel.blue = (unsigned short)((((int)src[a+dim-2].blue+(int)src[a+dim-1].blue+(int)src[b+dim-1].blue+(int)src[c+dim-1].blue)/4);
        dst[b+dim-1] = current_pixel;
```

```
//int a=dim*dim;
    int b=(dim-2)*dim;
    int c=(dim-1)*dim;    //单独处理四角
    current_pixel.red = (unsigned short)(((int)src[0].red+(int)src[dim].red)/2);
    current_pixel.green = (unsigned short)(((int)src[0].green+(int)src[dim].green)/2);
    current_pixel.blue = (unsigned short)(((int)src[0].blue+(int)src[dim].blue)/2);
    dst[0] = current_pixel;
    current_pixel.red = (unsigned short)(((int)src[dim-1].red+(int)src[dim+dim-1].red)/2);
    current_pixel.green = (unsigned short)(((int)src[dim-1].green+(int)src[dim+dim-1].green)/2);
    current_pixel.blue = (unsigned short)(((int)src[dim-1].blue+(int)src[dim+dim-1].blue)/2);
    dst[dim-1] = current_pixel;
    current_pixel.red = (unsigned short)(((int)src[b].red+(int)src[b+1].red+(int)src[c].red)/3);
    current_pixel.green = (unsigned short)(((int)src[b].green+(int)src[b+1].green+(int)src[c].green)/3);
    current_pixel.blue = (unsigned short)(((int)src[b].blue+(int)src[b+1].blue+(int)src[c].blue)/3);
    dst[c] = current_pixel;
    current_pixel.red = (unsigned short)(((int)src[b+dim-2].red+(int)src[b+dim-1].red+(int)src[c+dim-1].red)/3);
    current_pixel.green = (unsigned short)(((int)src[b+dim-2].green+(int)src[b+dim-1].green+(int)src[c+dim-1].green)/3);
    current_pixel.blue = (unsigned short)(((int)src[b+dim-2].blue+(int)src[b+dim-1].blue+(int)src[c+dim-1].blue)/3);
    dst[c+dim-1] = current_pixel;
}
```

2. 实验结果与分析

（性能对比和结果分析）

```
2023313409@comp4:~/csapp_fang/perflab/perflab_student$ ./driver
姓名：房效民
学号：2023313409
由学号生成的卷积核如下：
1 1 1
0 1 0
0 1 0

Rotate:
Dim             64      128     256     512     Mean
naive CPEs      3.1     4.5     8.0     12.6
your CPEs       3.5     3.7     5.8     6.1
Speedup         0.9     1.2     1.4     2.1     1.3

Smooth:
Dim             32      64      128     256     Mean
naive CPEs      70.8    70.8    71.0    71.0
your CPEs       10.4    10.9    11.0    11.0
Speedup         6.8     6.5     6.5     6.4     6.6

Summary of Your Best Scores:
  Rotate: 1.3
  Smooth: 6.6
```

3. 实验中遇到的问题及解决方法

（详细描述在实验过程中遇到的问题，包括错误描述、排查过程以及最终的解决方案。）

关于循环展开，一开始忘记进行边界处理，直接自增2一直增上去，结果segmentation fault,及时发现并且在最后添加边界条件的一点处理最终

成功解决问题

4. 请总结本次实验的收获，并给出对本次实验内容的建议

收获：理解了cache的功能，对程序性能有更深刻的认识
建议：提供markdown格式报告