Perflab3实验日志

PartB 优化

原始版本代码：

```
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);
}
```

优化二：逐行写将循环展开从而提高并行性、减少过程调用、使用对应行的三个指针实现。

```
void smooth3(int dim, pixel *src, pixel *dst)
{
    int i,j;
    pixel *pixelA,*pixelB,*pixelC;
    int size=dim-1;
    pixelB=src;
    pixelC=pixelB+dim;
    dst->red=(pixelB->red+(pixelB+1)->red+pixelC->red+(pixelC+1)->red)/4;
    dst->green=(pixelB->green+(pixelB+1)->green+pixelC->green+(pixelC+1)->green)/4;
    dst->blue=(pixelB->blue+(pixelB+1)->blue+pixelC->blue+(pixelC+1)->blue)/4;
    pixelB++;
    pixelC++;
    dst++;
    for(i=1;i<size;i++)
    {
        dst->red=(pixelB->red+(pixelB-1)->red+(pixelB+1)->red+pixelC->red+(pixelC-1)->red+(pixelC+1)->red)/6;
        dst->green=(pixelB->green+(pixelB-1)->green+(pixelB+1)->green+pixelC->green+(pixelC-1)->green+(pixelC+1)->green)/6;
        dst->blue=(pixelB->blue+(pixelB-1)->blue+(pixelB+1)->blue+pixelC->blue+(pixelC-1)->blue+(pixelC+1)->blue)/6;
        pixelB++;
        pixelC++;
        dst++;
    }
    dst->red=(pixelC->red+(pixelC-1)->red+pixelB->red+(pixelB-1)->red)>>2;
    dst->green=(pixelC->green+(pixelC-1)->green+pixelB->green+(pixelB-1)->green)>>2;
    dst->blue=(pixelC->blue+(pixelC-1)->blue+pixelB->blue+(pixelB-1)->blue)>>2;
    dst++;
    pixelA=src;
    pixelB=pixelA+dim;
    pixelC=pixelB+dim;
    for(i=1;i<size;i++)
    {
        dst->red=(pixelA->red+(pixelA+1)->red+pixelB->red+(pixelB+1)->red+pixelC->red+(pixelC+1)->red)/6;
        dst->green=(pixelB->green+(pixelB-1)->green+(pixelB+1)->green+pixelC->green+(pixelC-1)->green+(pixelC+1)->green)/6;
        dst->blue=(pixelB->blue+(pixelB-1)->blue+(pixelB+1)->blue+pixelC->blue+(pixelC-1)->blue+(pixelC+1)->blue)/6;
        dst++;
        pixelA++;
        pixelB++;
        pixelC++;
    }
}
```

优化二文字描述：

基于 PartA 优化部分的行优先读写顺序能带来显著的优化提升，并结合循环展开思想，通过逐行写的方法将循环展开，并将像素中的点进行分类（三类点，每类所要平滑处理的元素个数不同），同时考虑到对于任何像素点，待求平均的像素点所构成的块大小都不会超过三行，每行都不会超过三个。所以可以通过三个指针指向对应的三行，每个指针控制行相邻的两个或三个像素点的读运算，从而减少 cache 维护开销、减少对存储器与寄存器的读写操作、提高 cache 利用率，且平滑处理过程减少过程调用。循环内操作间无数据相关性，通过并行执行循环内操作提高并行性。

性能测试：

```
Smooth: Version = smooth1: 优化1:
Dim             32      64      128     256     512     Mean
Your CPEs       16.3    17.8    17.4    18.2    18.2
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         42.7    39.3    40.3    39.4    39.6    40.2

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       37.2    37.4    37.5    37.7    38.6
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         18.7    18.7    18.7    19.0    18.7    18.8
```

该思路的优化带来了显著的性能提升，平均性能提升在 2 倍左右。然而仍存在较大或者说可以延伸的优化空间——逐行写的循环展开中对于不同类像素的处理类似于分块处理，启发我可以将块的大小进一步增大从而不再局限于一行行地写，从而进一步提高程序的并行性与空间局部性(CSAPP6.6.2《重新排列循环以提高空间局部性》)。

优化三：行优先写，分块计算不同位置的像素点并减少不必要的函数调用（多种优化思路的结合）

```
void smooth2(int dim, pixel *src, pixel *dst) {//消除过多的函数调用，独立计算不同位置的像素点，由于确定了不同位置的需要计算的元素个数从而将所有除法运算中除数变量改为除常量
    int i=1,j=0;
    //优先处理特殊顶点，处理左上角顶点，该顶点只需处理 4 个元素
    dst[0].red=(src[0].red+src[1].red+src[dim].red+src[dim+1].red)/4; //处理4个元素，减少函数调用直接计算
    dst[0].green=(src[0].green+src[1].green+src[dim].green+src[dim+1].green)/4;
    dst[0].blue=(src[0].blue+src[1].blue+src[dim].blue+src[dim+1].blue)/4;
    //处理边上的点（单顶点），处理第一行其他非左上角顶点
    for(j=1; j<dim-1; j++) {
        dst[j].red=(src[j-1].red+src[j].red+src[j+1].red+src[dim+j-1].red+src[dim+j].red+src[dim+j+1].red)/6;//处理6个元素，减少函数调用直接计算
        dst[j].green=(src[j-1].green+src[j].green+src[j+1].green+src[dim+j-1].green+src[dim+j].green+src[dim+j+1].green)/6;
        dst[j].blue=(src[j-1].blue+src[j].blue+src[j+1].blue+src[dim+j-1].blue+src[dim+j].blue+src[dim+j+1].blue)/6;
    }
    //处理右上角顶点
    dst[j].red=(src[j].red+src[j-1].red+src[dim+j].red+src[dim+j-1].red)/4;//处理4个元素，减少函数调用直接计算
    dst[j].green=(src[j].green+src[j-1].green+src[dim+j].green+src[dim+j-1].green)/4;
    dst[j].blue=(src[j].blue+src[j-1].blue+src[dim+j].blue+src[dim+j-1].blue)/4;
    //处理1（0开头）至dim-2行
    for(; i<dim-1; i++) {
        //处理每一行的边界点，处理每一行的第一个像素点
        dst[i*dim].red=(src[(i-1)*dim].red+src[(i-1)*dim+1].red+src[i*dim].red+src[i*dim+1].red+src[(i+1)*dim].red+src[(i+1)*dim+1].red)/6;//处理6个元素，减少函数调用直接计算
        dst[i*dim].green=(src[(i-1)*dim].green+src[(i-1)*dim+1].green+src[i*dim].green+src[i*dim+1].green+src[(i+1)*dim].green+src[(i+1)*dim+1].green)/6;
        dst[i*dim].blue=(src[(i-1)*dim].blue+src[(i-1)*dim+1].blue+src[i*dim].blue+src[i*dim+1].blue+src[(i+1)*dim].blue+src[(i+1)*dim+1].blue)/6;
        //处理每一行第二个至第dim-1个像素点
        for(j=1; j<dim-1; j++) {//处理9个元素，减少函数调用直接计算
            dst[i*dim+j].red=(src[(i-1)*dim+j-1].red+src[(i-1)*dim+j].red+src[(i-1)*dim+j+1].red+src[i*dim+j-1].red+src[i*dim+j].red+src[i*dim+j+1].red+src[(i+1)*dim+j-1].red+src[(i+1)*dim+j].red+src[(i+1)*dim+j+1].red)/9;//
            dst[i*dim+j].green=(src[(i-1)*dim+j-1].green+src[(i-1)*dim+j].green+src[(i-1)*dim+j+1].green+src[i*dim+j-1].green+src[i*dim+j].green+src[i*dim+j+1].green+src[(i+1)*dim+j-1].green+src[(i+1)*dim+j].green+src[(i+1)*
            dst[i*dim+j].blue=(src[(i-1)*dim+j-1].blue+src[(i-1)*dim+j].blue+src[(i-1)*dim+j+1].blue+src[i*dim+j-1].blue+src[i*dim+j].blue+src[i*dim+j+1].blue+src[(i+1)*dim+j-1].blue+src[(i+1)*dim+j].blue+src[(i+1)*dim+j+1].
        }
        //处理每一行的边界点，处理每一行最后一个像素点
        dst[i*dim+j].red=(src[(i-1)*dim+j-1].red+src[(i-1)*dim+j].red+src[i*dim+j-1].red+src[i*dim+j].red+src[(i+1)*dim+j-1].red+src[(i+1)*dim+j].red)/6;//处理6个元素，减少函数调用直接计算
        dst[i*dim+j].green=(src[(i-1)*dim+j-1].green+src[(i-1)*dim+j].green+src[i*dim+j-1].green+src[i*dim+j].green+src[(i+1)*dim+j-1].green+src[(i+1)*dim+j].green)/6;
        dst[i*dim+j].blue=(src[(i-1)*dim+j-1].blue+src[(i-1)*dim+j].blue+src[i*dim+j-1].blue+src[i*dim+j].blue+src[(i+1)*dim+j-1].blue+src[(i+1)*dim+j].blue)/6;
    }
    //处理特殊顶点，处理左下角像素点
    dst[i*dim].red=(src[(i-1)*dim].red+src[(i-1)*dim+1].red+src[i*dim].red+src[i*dim+1].red)/4;//处理4个元素，减少函数调用直接计算
    dst[i*dim].green=(src[(i-1)*dim].green+src[(i-1)*dim+1].green+src[i*dim].green+src[i*dim+1].green)/4;
    dst[i*dim].blue=(src[(i-1)*dim].blue+src[(i-1)*dim+1].blue+src[i*dim].blue+src[i*dim+1].blue)/4;
    //处理最后一行非左下角、非右下角的像素点
    for(j=1; j<dim-1; j++) {
        dst[i*dim+j].red=(src[(i-1)*dim+j-1].red+src[(i-1)*dim+j].red+src[(i-1)*dim+j+1].red+src[i*dim+j-1].red+src[i*dim+j].red+src[i*dim+j+1].red)/6;//处理6个元素，减少函数调用直接计算
        dst[i*dim+j].green=(src[(i-1)*dim+j-1].green+src[(i-1)*dim+j].green+src[(i-1)*dim+j+1].green+src[i*dim+j-1].green+src[i*dim+j].green+src[i*dim+j+1].green)/6;
        dst[i*dim+j].blue=(src[(i-1)*dim+j-1].blue+src[(i-1)*dim+j].blue+src[(i-1)*dim+j+1].blue+src[i*dim+j-1].blue+src[i*dim+j].blue+src[i*dim+j+1].blue)/6;
    }
    //处理特殊顶点，处理右下角像素点
    dst[i*dim+j].red=(src[(i-1)*dim+j-1].red+src[(i-1)*dim+j].red+src[i*dim+j-1].red+src[i*dim+j].red)/4;//处理4个元素，减少函数调用直接计算
    dst[i*dim+j].green=(src[(i-1)*dim+j-1].green+src[(i-1)*dim+j].green+src[i*dim+j-1].green+src[i*dim+j].green)/4;
    dst[i*dim+j].blue=(src[(i-1)*dim+j-1].blue+src[(i-1)*dim+j].blue+src[i*dim+j-1].blue+src[i*dim+j].blue)/4;
}//分块独立平滑处理图像中的各像素
```

优化三文字描述：

将程序分块处理提高了程序的空间局部性，提高了 cache 命中率（此部分详见CSAPP6.6.2《重新排列循环以提高空间局部性》，按行写的方式提高了 cache 命中率并降低了其写开销从而达到优化效果。此外分块可以看作是另一种形式的将循环展开，在减少循环迭代次数的同时，循环展开使得处理器得以并行执行各操作，提高程序的并行性。此次优化结合了多种优化思路，对第一、二周的优化思路均有涉猎，这种多种优化思路结合的思想给我留下了深刻的印象。

性能测试：

```
Smooth: Version = smooth2: 优化2:
Dim             32      64      128     256     512     Mean
Your CPEs       8.8     9.2     9.3     9.0     9.4
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         78.7    76.2    75.5    79.4    77.1    77.4

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       36.2    36.4    36.5    38.7    39.7
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         19.2    19.2    19.2    18.5    18.2    18.9
```

从性能测试结果可以看到，程序性能（加速比）达到了很高的水平，甚至到达了原函数的 4.14 倍。

Perflab3实验报告

一、 PartA优化比较

优化 1：降低循环的低效率

通过代码移动，将循环中结果不变的部分移动到循环外部并用中间变量保存，从而避免了重复计算同时减少计算次数同时以逐行写的方式提高程序空间局部性源代码。

```
void rotate1(int dim, pixel *src, pixel *dst)
{
    int i, j, temp; //设置一个中间变量tmp，用来存储中间值
    for (j = 0; j < dim; j++)
    {
        temp = dim - 1 - j; //由于dim-1-j会经常使用，所以这里进行提前计算，省去了每次计算的时间
        for (i = 0; i < dim; i++)
            dst[RIDX(temp, i, dim)] = src[RIDX(i, j, dim)]; //这里不再对dim-1-j再进行运算
    }
}
```

```
Rotate: Version = rotate1: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.4     1.4     1.9     2.7     3.9
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         10.7    29.0    25.1    24.1    24.3    21.5
```

优点：随着数据规模的不断增大，降低循环的低效率对于程序往往有着显著的提升。改变读写方式提高了空间局部性。减少了重复计算，该优化是程序员需要首先考虑的最基本优化。

缺点：根据Amadhl定律，该优化部分占比过小无法阻碍了更好的优化提升。

优化 2：减少过程调用

头文件中宏定义了该函数。但在我们的程序中每次计算函数位置都需要调用到该函数，造成了过程调用这一部分的额外开销，我们根据函数体可以直接在程序中计算i*n+j 来实现该函数功能从而减少过程调用，达到减少开销、优化程序的目的。

```
void rotate2(int dim, pixel *src, pixel *dst)
{
    int i, j;
    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            dst[(dim - 1 - j) * dim + i] = src[i * dim + j]; //省去调用函数的时间
}
```

```
Rotate: Version = rotate2: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.4     1.9     3.4     7.0     6.8
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         10.6    20.9    13.6    9.5     13.9    13.2
```

优点：同优化1，该优化直观明了、易于上手，同样为最基础的优化，实现简单。

缺点：消除过程调用极大地损害了程序的模块性使程序可读性降低，且根据Amadhl定律，该优化部分占比较小且优化效果差，带来了甚微的优化提升。

优化 3：将优化 1、优化 2、旧版本优化 3 结合起来

```
void new_rotate3(int dim,pixel *src,pixel *dst)
{
    int i,j;
    for(j=0;j<dim;j=j+2)
    {
        int temp1=dim-1-j;
        int temp2=dim-1-j-1;
        for(i=0;i<dim;i=i+2)
        {
            dst[temp1*dim+i]=src[i*dim+j];
            dst[temp1*dim+i+1]=src[(i+1)*dim+j];
            dst[temp2*dim+i]=src[i*dim+j+1];
            dst[temp2*dim+i+1]=src[(i+1)*dim+j+1];
        }
    }
}
```

```
Rotate: Version = new_rotate3: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.4     1.5     2.0     2.4     3.3
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         10.8    27.0    23.7    27.4    28.3    22.2
```

优点：循环展开为两步，在循环展开的基础上结合降低循环低效率减少重复计算，增加并行性同时提高程序空间局部性。

缺点：循环展开的步长过小，优化部分占整个程序比例过低，可以将循环展开为更多的步长。

二、 PartA Amdahl 定律分析

若 60%的系统能够加速到不耗费时间的程度， 我们所获得的加速比也仅仅只有 2.5。

$$\text{加速比：}\ S=\frac{Told}{Tnew}=\frac{1}{(1-\alpha)+\alpha/k}$$

要想大幅提高整个系统的速度，我们必须提高整个系统很大一部分的速度!

综合对比分析 PartA 所有优化版本可以看到对于 Part A 影响最大的部分是**循环展开提高并行性。**因为 part A 进行的是一个类似于“复制”的操作，上一次的操作结果不会影响到这次的操作步骤，所以应该尽可能利用处理器流水线工作的特性，尽量在循环中执行多次操作，并行处理，提高并行性，使得优化部分占比尽可能提高，根据定律分析从而获得巨大的优化效果。

同时需要思考的是优化的效率。同样有着巨大影响的部分是**存储器读写（程序的空间局部性）**。由于 PartA 实现过程中需要频繁的对存储器进行读写操作，存储器缓存机制对数组以行为单位缓存在高层次的存储器中，通过改变读写方式使得程序以行优先的方式进行读写，最大程序上利用了缓存机制，从而提高了程序的空间局部性，获得良好的优化效率。这是保证循环展开提高并行性的同时能获得最佳优化效率的保证。

获取最优的加速比，往往不是某一种优化策略所能决定的，就如我们需要同时提高公式中的α与 k 般，需要我们不断地发现、结合所有优化策略，在提高优化程序占比的同时提高优化效率！

三、 PartB 优化比较

优化 1：纵向处理矩阵块减少纵向的动态规划（减少重复计算）

优点：改进算法，纵向 DP 算法平滑处理像素从而减少重复计算，极大提高算法部分优化效率。

缺点：纵向的 DP 导致空间局部性过差从而极大地降低了优化效率。

优化 2：行优先逐行写将循环展开从而提高并行性、减少过程（函数）调用、使用对应行的三个指针实现

优点：通过行优先写的方法将循环展开使得优化占比加大，并将像素中的点进行分类通过三个指针指向对应的三行，减少cache 维护（写）开销、减少对存储器（内存）与寄存器的读写操作、提高 cache 命中率，减少过程调用、提高并行性，从而提高优化效率。

缺点：存在着巨大的优化空间、分行处理并未使并行处理达到最佳效果，且存在着许多重复计算。

优化 3：行优先写，分块计算不同位置的像素点并减少不必要的函数调用（多种优化思路的结合）

优点：分块计算不同类像素点，极大的提高了程序空间局部性，提高了对存储器缓存的命中率，同时分块处理使得并行性进一步提升。

缺点：处理过程中存在着许多重复计算，降低优化效率。

四、 PartB Amdahl 定律分析

可以看到对于 part B 影响最大的部分是**分类处理（循环展开）+avg 函数实现（空间局部性的改进）**。前者保证优化占比的最大化的同时，后者保证了最大化的优化效率。同 PartA 上一次的操作结果不会影响到这一次的运行结果，各操作之间的关系应该为并行而不是串行从而最大化处理器流水线工作效率。**对像素的分类处理能将循环展开，提高程序的并行性。**

而其原本的 avg 函数本身的效率并不高，若只是将 avg 函数简单的拿到主函数里面只会起到甚微的优化效果。所以优化思路应该从修改 avg 函数本身的方向入手。可以看到 avg 函数的思路是取九宫格，然后取整个九宫格的平均值给中间的格。而对于角、边的边界情况分别对应的是 22 和 23 的矩形，所以我们可以直接将整个图像的像素进行分类，直接分成“四个角+四条边+中心内部”三个区域进行计算，并在计算过程中以行优先的方式进行读写，从而提高程序的空间局部性，最终可以达到极高的优化效率。

五、 实验总结

至此，Perflab实验圆满结束，在优化程序的过程中，我一直从计算机的角度来看待程序，仿佛我真的触及计算机底层了。同时，从高级语言的角度进行优化大大提高了我的思维过程，对编译系统的认识有了极大的飞跃，对计算机组成原理的认识有了更深的层次。