perflab2实验日志

一、优化方法

5.7 理解现代处理器

本节内容从目标机器硬件——处理器微体系结构优化角度着手，通过分析处理器执行指令的底层系统设计优化程序从而提高处理器运行效率。

5.8 循环展开

循环展开是一种程序变换，通过增加每次迭代计算的元素的数量，减少循环的迭代次数。循环展开能够从两个方面改善程序性能。首先，它减少了不直接有助于程序结果的操作的数量，例如循环索引计算和条件分支。其次，它提供了一些方法，可以进一步变化代码，减少整个计算中关键路径上的操作数量。

5.9 提高并行性

在此前对处理器微体系结构的分析中可以清楚的了解到处理器工作过程是并行执行的，其中执行加法和乘法的功能单元是完全流水线化，这意味着它们可以每个时钟周期开始一个新操作，结合处理器指令级并行大幅提高运算效率。主要通过以下两个方面实现：多个累计变量、重新结合变换。

二、优化实现

partA

原始版本代码

```
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)];
}
```

优化四：循环展开

```
void rotate4_1(int dim,pixel*src,pixel*dst)
{
    int i,j;
    int limit=dim-1;
    for(i=0;i<limit;i+=32)
    {
        int temp=dim-1-i;
        for(j=0;j<dim;j++)
        {
            dst[RIDX(i,j,dim)]=src[RIDX(j,temp,dim)];
            dst[RIDX(i+1,j,dim)]=src[RIDX(j,temp-1,dim)];
            dst[RIDX(i+2,j,dim)]=src[RIDX(j,temp-2,dim)];
            dst[RIDX(i+3,j,dim)]=src[RIDX(j,temp-3,dim)];
            dst[RIDX(i+4,j,dim)]=src[RIDX(j,temp-4,dim)];
            dst[RIDX(i+5,j,dim)]=src[RIDX(j,temp-5,dim)];
            dst[RIDX(i+6,j,dim)]=src[RIDX(j,temp-6,dim)];
            dst[RIDX(i+7,j,dim)]=src[RIDX(j,temp-7,dim)];
            dst[RIDX(i+8,j,dim)]=src[RIDX(j,temp-8,dim)];
            dst[RIDX(i+9,j,dim)]=src[RIDX(j,temp-9,dim)];
            dst[RIDX(i+10,j,dim)]=src[RIDX(j,temp-10,dim)];
            dst[RIDX(i+11,j,dim)]=src[RIDX(j,temp-11,dim)];
            dst[RIDX(i+12,j,dim)]=src[RIDX(j,temp-12,dim)];
            dst[RIDX(i+13,j,dim)]=src[RIDX(j,temp-13,dim)];
            dst[RIDX(i+14,j,dim)]=src[RIDX(j,temp-14,dim)];
            dst[RIDX(i+15,j,dim)]=src[RIDX(j,temp-15,dim)];
            dst[RIDX(i+16,j,dim)]=src[RIDX(j,temp-16,dim)];
            dst[RIDX(i+17,j,dim)]=src[RIDX(j,temp-17,dim)];
            dst[RIDX(i+18,j,dim)]=src[RIDX(j,temp-18,dim)];
            dst[RIDX(i+19,j,dim)]=src[RIDX(j,temp-19,dim)];
            dst[RIDX(i+20,j,dim)]=src[RIDX(j,temp-20,dim)];
            dst[RIDX(i+21,j,dim)]=src[RIDX(j,temp-21,dim)];
            dst[RIDX(i+22,j,dim)]=src[RIDX(j,temp-22,dim)];
            dst[RIDX(i+23,j,dim)]=src[RIDX(j,temp-23,dim)];
            dst[RIDX(i+24,j,dim)]=src[RIDX(j,temp-24,dim)];
            dst[RIDX(i+25,j,dim)]=src[RIDX(j,temp-25,dim)];
            dst[RIDX(i+26,j,dim)]=src[RIDX(j,temp-26,dim)];
            dst[RIDX(i+27,j,dim)]=src[RIDX(j,temp-27,dim)];
            dst[RIDX(i+28,j,dim)]=src[RIDX(j,temp-28,dim)];
            dst[RIDX(i+29,j,dim)]=src[RIDX(j,temp-29,dim)];
            dst[RIDX(i+30,j,dim)]=src[RIDX(j,temp-30,dim)];
            dst[RIDX(i+31,j,dim)]=src[RIDX(j,temp-31,dim)];
        }
    }
    for(;i<dim;i++)
    {
        int temp=dim-1-i;
        for(j=0;j<dim;j++)
        {
            dst[RIDX(i,j,dim)]=src[RIDX(j,temp,dim)];
        }
    }
}
```

优化四文字描述：

以写优先的方式将循环展开为 32 路循环，减少循环的迭代次数。减少了不直接有助于程序结果的操作的数量，如循环索引计算和条件分支。其次减少整个计算中关键路径上的操作数量。通过移动代码降低循环的低效率，将循环中重复计算部分移动到循环外部避免重复计算；循环内操作间无数据相关性，通过并行计算各步写的位置提高并行性。可以看到，该优化带来了较为显著的性能提升，但提升幅度相较于预期还是相差较大。这是因为写优先的方式以列为单位访问 dst 数组内存，而 cache 以行为单位缓存，导致了 cache 利用率低下以及多次维护 cache 的开销。

性能测试：

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.4     2.0     4.0     7.0     7.0
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         10.6    20.3    11.6    9.4     13.5    12.6

Rotate: Version = rotate4_1: 4.1优化:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.7     1.9     2.4     6.5
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.8     24.0    24.7    27.2    14.5    18.7
```

优化五：

```c
void rotate4_2(int dim,pixel*src,pixel*dst)
{
    int i,j;
    int limit=dim-1;
    for(i=0;i<limit;i+=32)
    {
        for(j=0;j<dim;j++)
        {
            int temp=dim-1-j;
            dst[RIDX(temp,i,dim)]=src[RIDX(i,j,dim)];
            dst[RIDX(temp,i+1,dim)]=src[RIDX(i+1,j,dim)];
            dst[RIDX(temp,i+2,dim)]=src[RIDX(i+2,j,dim)];
            dst[RIDX(temp,i+3,dim)]=src[RIDX(i+3,j,dim)];
            dst[RIDX(temp,i+4,dim)]=src[RIDX(i+4,j,dim)];
            dst[RIDX(temp,i+5,dim)]=src[RIDX(i+5,j,dim)];
            dst[RIDX(temp,i+6,dim)]=src[RIDX(i+6,j,dim)];
            dst[RIDX(temp,i+7,dim)]=src[RIDX(i+7,j,dim)];
            dst[RIDX(temp,i+8,dim)]=src[RIDX(i+8,j,dim)];
            dst[RIDX(temp,i+9,dim)]=src[RIDX(i+9,j,dim)];
            dst[RIDX(temp,i+10,dim)]=src[RIDX(i+10,j,dim)];
            dst[RIDX(temp,i+11,dim)]=src[RIDX(i+11,j,dim)];
            dst[RIDX(temp,i+12,dim)]=src[RIDX(i+12,j,dim)];
            dst[RIDX(temp,i+13,dim)]=src[RIDX(i+13,j,dim)];
            dst[RIDX(temp,i+14,dim)]=src[RIDX(i+14,j,dim)];
            dst[RIDX(temp,i+15,dim)]=src[RIDX(i+15,j,dim)];
            dst[RIDX(temp,i+16,dim)]=src[RIDX(i+16,j,dim)];
            dst[RIDX(temp,i+17,dim)]=src[RIDX(i+17,j,dim)];
            dst[RIDX(temp,i+18,dim)]=src[RIDX(i+18,j,dim)];
            dst[RIDX(temp,i+19,dim)]=src[RIDX(i+19,j,dim)];
            dst[RIDX(temp,i+20,dim)]=src[RIDX(i+20,j,dim)];
            dst[RIDX(temp,i+21,dim)]=src[RIDX(i+21,j,dim)];
            dst[RIDX(temp,i+22,dim)]=src[RIDX(i+22,j,dim)];
            dst[RIDX(temp,i+23,dim)]=src[RIDX(i+23,j,dim)];
            dst[RIDX(temp,i+24,dim)]=src[RIDX(i+24,j,dim)];
            dst[RIDX(temp,i+25,dim)]=src[RIDX(i+25,j,dim)];
            dst[RIDX(temp,i+26,dim)]=src[RIDX(i+26,j,dim)];
            dst[RIDX(temp,i+27,dim)]=src[RIDX(i+27,j,dim)];
            dst[RIDX(temp,i+28,dim)]=src[RIDX(i+28,j,dim)];
            dst[RIDX(temp,i+29,dim)]=src[RIDX(i+29,j,dim)];
            dst[RIDX(temp,i+30,dim)]=src[RIDX(i+30,j,dim)];
            dst[RIDX(temp,i+31,dim)]=src[RIDX(i+31,j,dim)];
        }
    }
    for(;i<dim;i++)
    {
        for(j=0;j<dim;j++)
        {
            dst[RIDX(dim-1-j,i,dim)]=src[RIDX(i,j,dim)];
        }
    }
}
```

优化五文字描述：

以读优先的方式将循环展开为 32 路循环，减少了不直接有助于程序结果的操作的数量，如循环索引计算和条件分支。其次减少整个计算中关键路径上的操作数量。使 dst 地址连续从而减少对存储器的读写操作降低循环的低效率（提高 cache 利用率），通过代码移动将循环中重复计算部分移动到循环外部避免重复计算，降低循环低效率；循环内操作间无数据相关性，利用处理器结构流水线处理特性通过并行计算各步写的位置提高并行性。

性能测试：

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.4     2.1     3.6     7.0     7.2
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         10.4    19.4    12.9    9.5     13.2    12.6

Rotate: Version = rotate4_2: 4.2优化:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.6     1.7     1.8     3.0
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         10.0    24.8    27.0    35.9    31.9    23.8
```

从性能测试结果来看，此优化版本的性能提升相较于 4.1 优化有了较大幅度的提高，性能以及接近达到原程序的 2 倍， 与预期情况相符，验证了分析中提及的优化思路正确性。

优化六：

```c
void rotate4_3(int dim, pixel *src, pixel *dst)
{
    int i, j, tmp1 = dim * dim, tmp2 = dim * 31, tmp3 = tmp1 - dim, tmp4 = tmp1 + 32, tmp5 = dim + 31; //定义中间变量
    dst += tmp3;
    for (i = 0; i < dim; i += 32)
    {
        for (j = 0; j < dim; j++) //将for循环展开，并通过指针的形式实现对图形的翻转，以32为一个步长
        {
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            dst++;
            src += dim;
            *dst = *src;
            src++;
            src -= tmp2;
            dst -= tmp5;
        }
        src += tmp2;
        dst += tmp4;
```

优化六文字描述：

使用循环展开，展开为 32 路，减少了不直接有助于程序结果的操作的数量，如循环索引计算和条件分支。其次减少整个计算中关键路径上的操作数量。 以读优先的思路使 dst 指针连续减少对存储器读写次数（提高 cache 利用率），同时通过移动代码降低循环的效率；循环内操作间无数据相关性，流水线并行，通过并行执行循环内操作，提高并行性，不再调用函数，减少过程调用。

性能测试：

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.4     2.0     4.1     7.6     7.2
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         10.4    20.1    11.5    8.6     13.1    12.2

Rotate: Version = rotate4_3: 4.3优化:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.2     1.2     1.3     1.1     3.2
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         12.0    32.4    36.3    58.7    29.4    30.0
```

此优化为优化的最终版本，结合了目前为止所有的优化思想。从程序性能测试结果来看，最终的优化带来了巨幅的性能提升，性能优化效果最为明显，为 partA rotate 函数所有优化中优化效果最好的版本。

PartB

```
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);
}
```

```
void smooth1(int dim, pixel *src, pixel *dst)
{
    int i, j;
    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j = j + 2)
        { //改变步长，尽可能多的利用循环
            dst[RIDX(i, j, dim)] = avg(dim, i, j, src);
            dst[RIDX(i, j + 1, dim)] = avg(dim, i, j + 1, src);
        }
}
```

左原始版本代码　右优化一

优化一文字描述：

这是我在第一个实验中发现的一种优化方法，可以通过增加循环的步长，来达到充分利用循环的作用，并通过这种方法来减少循环的次数，这里我仅仅采用了步长+2，还可以将步长拓展到32。

```
void smooth2(int dim, pixel *src, pixel *dst)
{
    int i, j;
    for (i = 0; i < dim; i++)
    {
        for (j = 0; j < dim; j = j + 32)
        { //for循环展开
            dst[RIDX(i, j, dim)] = avg(dim, i, j, src);
            dst[RIDX(i, j + 1, dim)] = avg(dim, i, j + 1, src);
            dst[RIDX(i, j + 2, dim)] = avg(dim, i, j + 2, src);
            dst[RIDX(i, j + 3, dim)] = avg(dim, i, j + 3, src);
            dst[RIDX(i, j + 4, dim)] = avg(dim, i, j + 4, src);
            dst[RIDX(i, j + 5, dim)] = avg(dim, i, j + 5, src);
            dst[RIDX(i, j + 6, dim)] = avg(dim, i, j + 6, src);
            dst[RIDX(i, j + 7, dim)] = avg(dim, i, j + 7, src);
            dst[RIDX(i, j + 8, dim)] = avg(dim, i, j + 8, src);
            dst[RIDX(i, j + 9, dim)] = avg(dim, i, j + 9, src);
            dst[RIDX(i, j + 10, dim)] = avg(dim, i, j + 10, src);
            dst[RIDX(i, j + 11, dim)] = avg(dim, i, j + 11, src);
            dst[RIDX(i, j + 12, dim)] = avg(dim, i, j + 12, src);
            dst[RIDX(i, j + 13, dim)] = avg(dim, i, j + 13, src);
            dst[RIDX(i, j + 14, dim)] = avg(dim, i, j + 14, src);
            dst[RIDX(i, j + 15, dim)] = avg(dim, i, j + 15, src);
            dst[RIDX(i, j + 16, dim)] = avg(dim, i, j + 16, src);
            dst[RIDX(i, j + 17, dim)] = avg(dim, i, j + 17, src);
            dst[RIDX(i, j + 18, dim)] = avg(dim, i, j + 18, src);
            dst[RIDX(i, j + 19, dim)] = avg(dim, i, j + 19, src);
            dst[RIDX(i, j + 20, dim)] = avg(dim, i, j + 20, src);
            dst[RIDX(i, j + 21, dim)] = avg(dim, i, j + 21, src);
            dst[RIDX(i, j + 22, dim)] = avg(dim, i, j + 22, src);
            dst[RIDX(i, j + 23, dim)] = avg(dim, i, j + 23, src);
            dst[RIDX(i, j + 24, dim)] = avg(dim, i, j + 24, src);
            dst[RIDX(i, j + 25, dim)] = avg(dim, i, j + 25, src);
            dst[RIDX(i, j + 26, dim)] = avg(dim, i, j + 26, src);
            dst[RIDX(i, j + 27, dim)] = avg(dim, i, j + 27, src);
            dst[RIDX(i, j + 28, dim)] = avg(dim, i, j + 28, src);
            dst[RIDX(i, j + 29, dim)] = avg(dim, i, j + 29, src);
            dst[RIDX(i, j + 30, dim)] = avg(dim, i, j + 30, src);
            dst[RIDX(i, j + 31, dim)] = avg(dim, i, j + 31, 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》)。