

课程实验报告

课 程 名 称： 计算机系统试验

实验项目名称： PerfLab

专 业 班 级： 计算机科学

姓 名： 周思宇

学 号： 201608030201

指 导 教 师：

完 成 时 间： 2018 年 5 月 11 日

信息科学与工程学院

|  |  |
| --- | --- |
| **实验题目：**perflab | |
| **实验目的：**了解程序的时间局部性和空间局部性，并能根据这种性质对实验代码进行优化，获得更好的加速比。 | |
| **实验资源：Ubuntu环境，PC** | |
| **实验任务：**  (1)修改 kernel.c 的源代码，对 rotate 和 smooth 函数进行优化。  (2)查看目录下的 Makefile 文件，清楚不同的 make 指令完成的功能。用命令 make clean 清除原来的内容，用命令 make 编译生成新的可执行文件 driver。键 入命令 ./driver，测试修改后的 rotate 和 smooth 两个函数的加速比和 naive\_rotate 和 naive\_smooth 两个函数的加速比，并分析原因。  (3)针对 rotate 和 smooth，每个写出至少三种不同的优化版本。   1. 优化 rotate 函数，rotate 函数是将图像实现逆时针旋转 90°，如下图所示:     源代码：    第一种优化方式：改进循环测试的效率：  把dim-1-j不放在两个循环里，而是把它从内层循环取出，于是就不需要每次迭代时都执行这个函数。    第二种优化方式：循环展开  增加每次迭代计算的元素的数量，减少迭代的循环次数。  减少了不直接有助于程序结果的操作的数量，并且可以进一步变化代码：    第三种优化方式：提高并行性  利用功能单元的流水线能力，做4路并行变换。    三种优化对比：    2.smooth的优化  原函数：  /\*  \* naive\_smooth - The naive baseline version of smooth  \*/  char naive\_smooth\_descr[] = "naive\_smooth: Naive baseline 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);  }  这次有 尝试过循环展开，并行运算等，但是最后的结果都是和原函数的cpe极其相似，并且全是负优化。  所以考虑到可能是思路或者方法错误。  在头文件def.h中，写着#defineRIDX(i,j,n) ((i)\*(n)+(j))  这是我们平时不常接触的函数型宏定义，传入i和j还有n，就计算出(i)\*(n)+(j)，这是dst的地址。  一个二维的图片，一般用二维数组存储，但是二维数组在内存中依然是连续的物理地址。**可以把dst[RIDX(i,j, dim)] 直接理解为二维数组a[i][j]。**  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++)               accumulate\_sum(&sum,src[RIDX(ii, jj, dim)]);     assign\_sum\_to\_pixel(&current\_pixel, sum);     return current\_pixel; }  这个函数循环的效率太低了，所以重点不是改原函数，而是对这个被调用的函数做出一定的修改（或者干脆放弃他，自己编写循环效率高的算法）。  **需要改写代码，不调用avg函数**  像素点平均颜色是如何计算的？  static void accumulate\_sum(pixel\_sum \*sum, pixel p) //统计颜色数据 {     sum->red += (int) p.red;     sum->green += (int) p.green;     sum->blue += (int) p.blue;     sum->num++;     return; }  static void assign\_sum\_to\_pixel(pixel \*current\_pixel, pixel\_sumsum) //计算平均值 {     current\_pixel->red = (unsigned short) (sum.red/sum.num);     current\_pixel->green = (unsigned short) (sum.green/sum.num);     current\_pixel->blue = (unsigned short) (sum.blue/sum.num);     return; }  **把相邻的像素点的RGB颜色各取平均**  首先，学习之前rotate函数的优化方法，为了减少调用max和min的时间，我们可以自定义宏定义。  /\*  \* smooth - Your current working version of smooth.  \* IMPORTANT: This is the version you will be graded on  \*/  char smooth\_descr[] = "smooth: Current working version";  void smooth(int dim, pixel \*src, pixel \*dst)  {  int i, j;  for (i = 0; i < dim; i++)  {  int k1=max(i-1,0),k2=min(i+1,dim-1);  for (j = 0; j < dim; j++)  {  int ii, jj;  int k3=max(j-1,0),k4=min(j+1,dim-1);  pixel\_sum sum;  pixel current\_pixel;  //initialize\_pixel\_sum(&sum);  sum.red= sum.green = sum.blue = 0;  sum.num= 0;  for(ii= k1; ii <= k2; ii++)  for(jj = k3; jj <=k4; jj++)  //accumulate\_sum(&sum, src[RIDX(ii,jj, dim)]);  {  pixel p=src[RIDX(ii, jj, dim)];  sum.red += (int) p.red;  sum.green+= (int) p.green;  sum.blue+= (int) p.blue;  sum.num++;  }  //assign\_sum\_to\_pixel(&current\_pixel,sum);  {  current\_pixel.red = (unsigned short)(sum.red/sum.num);  current\_pixel.green = (unsigned short)(sum.green/sum.num);  current\_pixel.blue= (unsigned short) (sum.blue/sum.num);  dst[RIDX(i, j, dim)] = current\_pixel;  }  }  }  }  第二种方式：  建立一个数组rowsum，着个数组专门保存并且计算算到的像素值，因为计算像素的平均值并保存之后，这个值附近的值值再去计算平均值的时候会被重复计算，所以用一个二维数组保存比较方便，也好调用查找。  /\*  \* smooth - Your current working version of smooth.  \* IMPORTANT: This is the version you will be graded on  \*/  char smooth\_descr2[] = "smooth: Current working version";  void smooth2(int dim, pixel \*src, pixel \*dst)  {  pixel\_sum rowsum[530][530];  int i, j, snum;  for(i=0;i<dim; i++)  {  rowsum[i][0].red = (src[RIDX(i, 0, dim)].red+src[RIDX(i, 1, dim)].red);  rowsum[i][0].blue = (src[RIDX(i, 0, dim)].blue+src[RIDX(i, 1,dim)].blue);  rowsum[i][0].green = (src[RIDX(i, 0, dim)].green+src[RIDX(i, 1,dim)].green);  rowsum[i][0].num = 2;  for(j=1;j<dim-1; j++)  {  rowsum[i][j].red = (src[RIDX(i, j-1, dim)].red+src[RIDX(i, j,dim)].red+src[RIDX(i, j+1, dim)].red);  rowsum[i][j].blue = (src[RIDX(i, j-1, dim)].blue+src[RIDX(i, j,dim)].blue+src[RIDX(i, j+1, dim)].blue);  rowsum[i][j].green = (src[RIDX(i, j-1, dim)].green+src[RIDX(i, j,dim)].green+src[RIDX(i, j+1, dim)].green);  rowsum[i][j].num = 3;  }  rowsum[i][dim-1].red = (src[RIDX(i, dim-2, dim)].red+src[RIDX(i, dim-1,dim)].red);  rowsum[i][dim-1].blue = (src[RIDX(i, dim-2, dim)].blue+src[RIDX(i,dim-1, dim)].blue);  rowsum[i][dim-1].green = (src[RIDX(i, dim-2, dim)].green+src[RIDX(i,dim-1, dim)].green);  rowsum[i][dim-1].num = 2;  }  for(j=0;j<dim; j++)  {  snum=rowsum[0][j].num+rowsum[1][j].num; dst[RIDX(0,j,dim)].red=(unsignedshort)((rowsum[0][j].red+rowsum[1][j].red)/snum);  dst[RIDX(0,j, dim)].blue = (unsigned short)((rowsum[0][j].blue+rowsum[1][j].blue)/snum);  dst[RIDX(0,j, dim)].green = (unsigned short)((rowsum[0][j].green+rowsum[1][j].green)/snum);  for(i=1;i<dim-1; i++)  {  snum =rowsum[i-1][j].num+rowsum[i][j].num+rowsum[i+1][j].num;  dst[RIDX(i, j, dim)].red = (unsigned short)((rowsum[i-1][j].red+rowsum[i][j].red+rowsum[i+1][j].red)/snum);  dst[RIDX(i, j, dim)].blue = (unsigned short)((rowsum[i-1][j].blue+rowsum[i][j].blue+rowsum[i+1][j].blue)/snum);  dst[RIDX(i, j, dim)].green = (unsigned short)((rowsum[i-1][j].green+rowsum[i][j].green+rowsum[i+1][j].green)/snum);  }  snum =rowsum[dim-1][j].num+rowsum[dim-2][j].num;  dst[RIDX(dim-1, j, dim)].red = (unsigned short)((rowsum[dim-2][j].red+rowsum[dim-1][j].red)/snum);  dst[RIDX(dim-1, j, dim)].blue = (unsigned short)((rowsum[dim-2][j].blue+rowsum[dim-1][j].blue)/snum);  dst[RIDX(dim-1, j, dim)].green = (unsigned short)((rowsum[dim-2][j].green+rowsum[dim-1][j].green)/snum);  }  }  第三种优化：  把求平均值分成了几种情况来考虑，第一组：四个顶点；第二组：四个边位置的数块；第三组：最中间的快；  这样会减少函数调用，因为每次都是直接计算。  char smooth\_descr3[] = "smooth: Current working version";  void smooth3(int dim, pixel \*src, pixel \*dst)  {  int i,j;  int dim0=dim;  int dim1=dim-1;  int dim2=dim-2;  pixel \*P1, \*P2, \*P3;  pixel \*dst1;  P1=src;  P2=P1+dim0; //左上角像素处理  dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red)>>2;  dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green)>>2;  dst->blue=(P1->blue+(P1+1)->blue+P2->blue+(P2+1)->blue)>>2;  dst++; //上边界处理  for(i=1;i<dim1;i++) {  dst->red=(P1->red+(P1+1)->red+(P1+2)->red+P2->red+(P2+1)->red+(P2+2)->red)/6;  dst->green=(P1->green+(P1+1)->green+(P1+2)->green+P2->green+(P2+1)->green+(P2+2)->green)/6;  dst->blue=(P1->blue+(P1+1)->blue+(P1+2)->blue+P2->blue+(P2+1)->blue+(P2+2)->blue)/6;  dst++;  P1++;  P2++;  } //右上角像素处理  dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red)>>2;  dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green)>>2;  dst->blue=(P1->blue+(P1+1)->blue+P2->blue+(P2+1)->blue)>>2;  dst++;  P1=src;  P2=P1+dim0;  P3=P2+dim0; //左边界处理  for(i=1;i<dim1;i++) {  dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red+P3->red+(P3+1)->red)/6;  dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green+P3->green+(P3+1)->green)/6;  dst->blue=(P1->blue+(P1+1)->blue+P2->blue+(P2+1)->blue+P3->blue+(P3+1)->blue)/6;  dst++;  dst1=dst+1; //主体中间部分处理  for(j=1;j<dim2;j+=2) { //同时处理2个像素  dst->red=(P1->red+(P1+1)->red+(P1+2)->red+P2->red+(P2+1)->red+(P2+2)->red+P3->red+(P3+1)->red+(P3+2)->red)/9;  dst->green=(P1->green+(P1+1)->green+(P1+2)->green+P2->green+(P2+1)->green+(P2+2)->green+P3->green+(P3+1)->green+(P3+2)->green)/9;  dst->blue=(P1->blue+(P1+1)->blue+(P1+2)->blue+P2->blue+(P2+1)->blue+(P2+2)->blue+P3->blue+(P3+1)->blue+(P3+2)->blue)/9;  dst1->red=((P1+3)->red+(P1+1)->red+(P1+2)->red+(P2+3)->red+(P2+1)->red+(P2+2)->red+(P3+3)->red+(P3+1)->red+(P3+2)->red)/9;  dst1->green=((P1+3)->green+(P1+1)->green+(P1+2)->green+(P2+3)->green+(P2+1)->green+(P2+2)->green+(P3+3)->green+(P3+1)->green+(P3+2)->green)/9;  dst1->blue=((P1+3)->blue+(P1+1)->blue+(P1+2)->blue+(P2+3)->blue+(P2+1)->blue+(P2+2)->blue+(P3+3)->blue+(P3+1)->blue+(P3+2)->blue)/9;  dst+=2;dst1+=2;P1+=2;P2+=2;P3+=2;  }  for(;j<dim1;j++) {  dst->red=(P1->red+(P1+1)->red+(P1+2)->red+P2->red+(P2+1)->red+(P2+2)->red+P3->red+(P3+1)->red+(P3+2)->red)/9;  dst->green=(P1->green+(P1+1)->green+(P1+2)->green+P2->green+(P2+1)->green+(P2+2)->green+P3->green+(P3+1)->green+(P3+2)->green)/9;  dst->blue=(P1->blue+(P1+1)->blue+(P1+2)->blue+P2->blue+(P2+1)->blue+(P2+2)->blue+P3->blue+(P3+1)->blue+(P3+2)->blue)/9;  dst++; P1++;P2++;P3++;  } //右侧边界处理  dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red+P3->red+(P3+1)->red)/6;  dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green+P3->green+(P3+1)->green)/6; dst->blue=(P1->blue+(P1+1)->blue+P2->blue+(P2+1)->blue+P3->blue+(P3+1)->blue)/6;  dst++; P1+=2; P2+=2; P3+=2;  } //左下角处理  dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red)>>2; dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green)>>2;  dst->blue=(P1->blue+(P1+1)->blue+P2->blue+(P2+1)->blue)>>2;  dst++; //下边界处理  for(i=1;i<dim1;i++) {  dst->red=(P1->red+(P1+1)->red+(P1+2)->red+P2->red+(P2+1)->red+(P2+2)->red)/6;  dst->green=(P1->green+(P1+1)->green+(P1+2)->green+P2->green+(P2+1)->green+(P2+2)->green)/6;  dst->blue=(P1->blue+(P1+1)->blue+(P1+2)->blue+P2->blue+(P2+1)->blue+(P2+2)->blue)/6;  dst++; P1++; P2++;  } //右下角像素处理  dst->red=(P1->red+(P1+1)->red+P2->red+(P2+1)->red)>>2;  dst->green=(P1->green+(P1+1)->green+P2->green+(P2+1)->green)>>2;  dst->blue=(P1->blue+(P1+1)->blue+P2->blue+(P2+1)->blue)>>2;  }    彩蛋：      收获与体会：  一开始做这个实验的时候，没有什么想法，只是想通过老师给的范例去进行相似的优化。  后来找老师亲自询问过之后才发现思维不能太狭窄。比如说rotate，就可以从循环展开和并行运算这两个角度考虑优化。并且，在循环展开的时候，可以在参考他人代码的同时，有一定的自己的想法。不一定要32位展开，可以尝试不同的数字展开，有可能会得到更优解。比如rotate就适合12路展开。  在处理优化的时候，首先要有目的有目标有想法。  总结了一下优化的方法：  1减少函数调用  2提前计算  3循环展开  4并行运算  5提高cache利用率 | |
| 实  验成绩 |  |