性能优化 实验报告

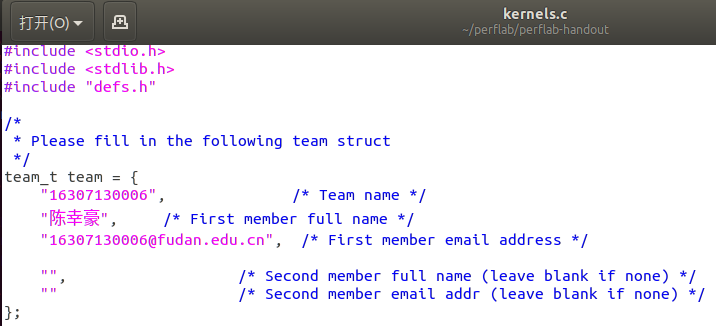
16307130006 陈幸豪

1. 准备工作

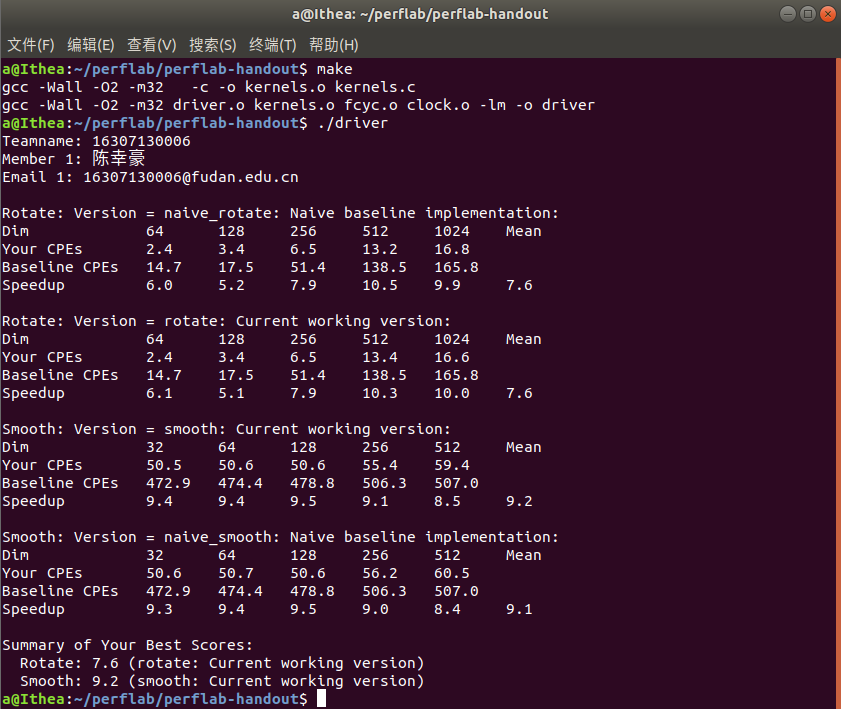
连接校园网内网，在Ubuntu内打开terminal，运行：

scp -r stu16307130006@10.48.76.8:~/ ~/perflab

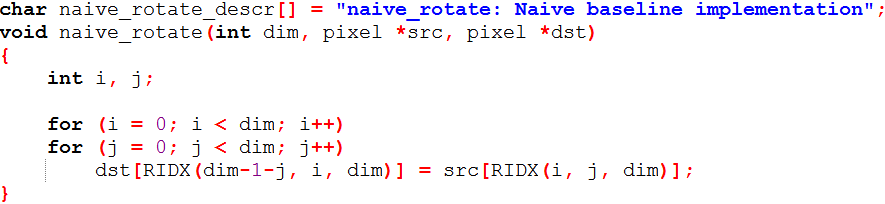
输入密码，回车，即从服务器取回文件。解压perflab-handout.tar得到一些.c和.h文件，还有README和Makefile。首先根据说明，修改kernels.c内的struct team，写入自己的学号姓名并保存：



在Makefile所在目录打开terminal，执行make得到一个可执行文件driver。直接运行它：./driver。稍等一会即可看到kernels.c中各种版本的rotate和smooth函数运行所花的时间。



1. Rotate的优化
   1. 原始代码分析



其中RIDX的定义在defs.h中。具体为：

#define RIDX(i,j,n) ((i)\*(n)+(j))

也就是说src[RIDX(i,j,dim)]应当是src[i][j]的意思，而dim即为每行有多少列。此外，图片的行列数相等。

* 1. 优化思路

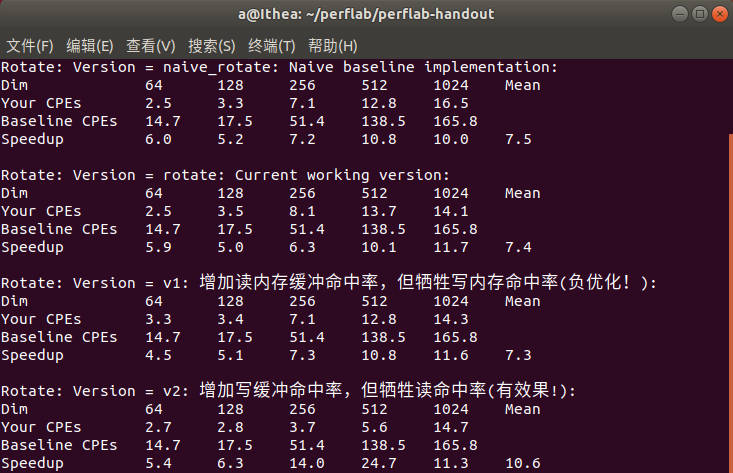
首先意识到这是内存读写密集型的任务，因此考虑更多地命中缓存。一个缓存miss会损失几百条指令的时间！那么是命中写缓存更快，还是命中读缓存更快呢？于是测试一下：

char rotate\_v1\_descr[] = "v1: 增加读内存缓冲命中率，但牺牲写内存命中率(负优化！)";

char rotate\_v2\_descr[] = "v2: 增加写缓冲命中率，但牺牲读命中率(有效果!)";

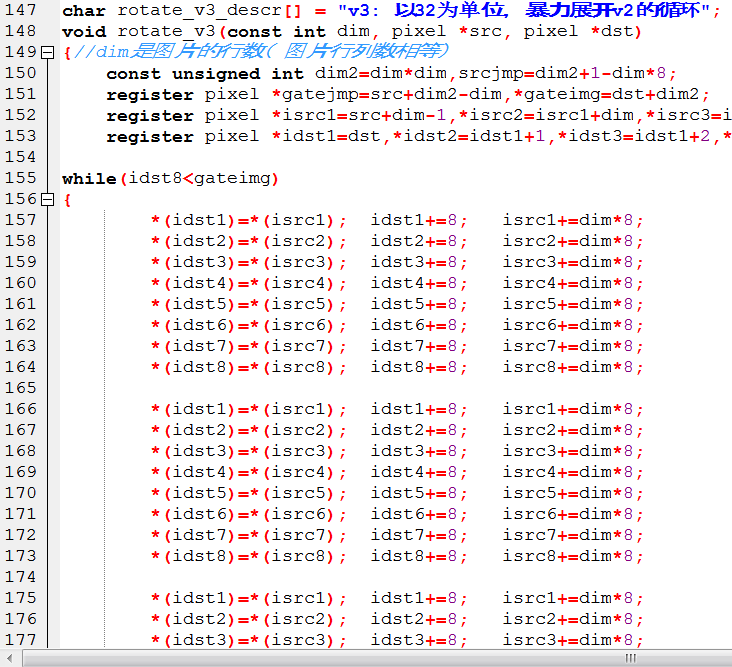
试验表明，按行写入dst（按列读取src），更多地命中写缓存可以略微提高速度，但按列写入dst（按行读取src）反而略微减慢速度。所以应当多命中写缓存。这与写缓存的复杂性是一致的。

在这个尝试中，顺便把src[某某下标]改成了直接用指针访问图片。这样就不再需要从src和dst反复做加法。

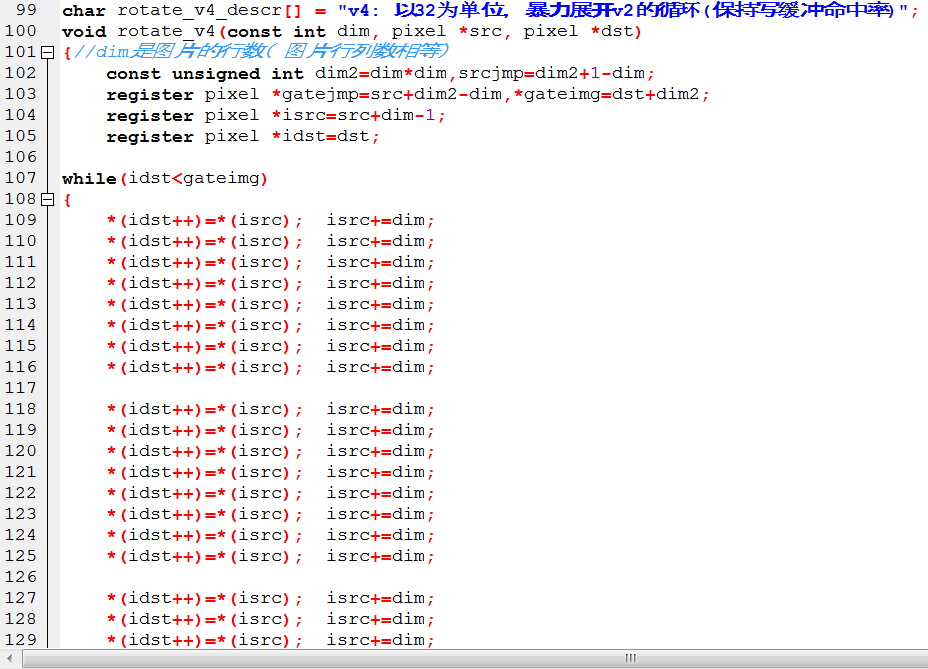


接下来原本考虑试试多线程并行旋转，但drivers.c不允许修改，于是无法#include <pthreads.h>（其他神奇的操作也不允许了）。所以似乎只能考虑暴力展开循环。由于说明中指出，程序只需要对边长为32的倍数的图片可用。于是我们以32为单位来展开……

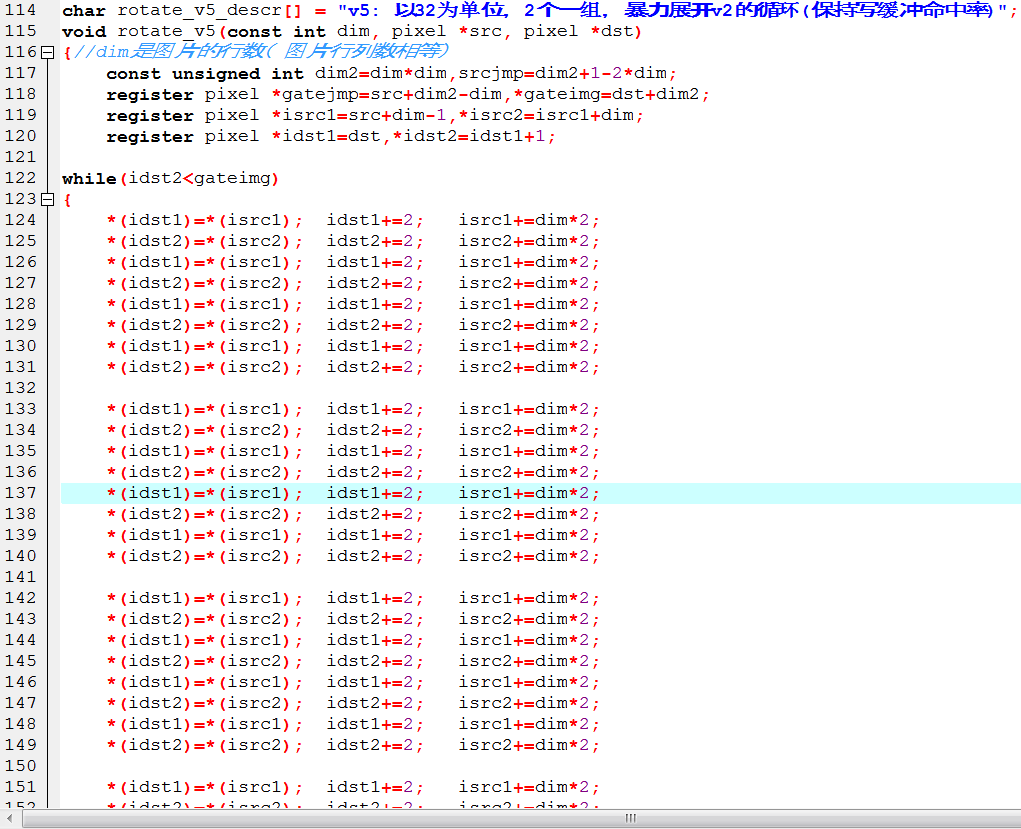
先尝试一下8个pixel一起写，即解除数据之间的依赖性，结果反而变慢：



于是意识到这样没有命中写缓存。再尝试一个一个地写。结果速度变快了：

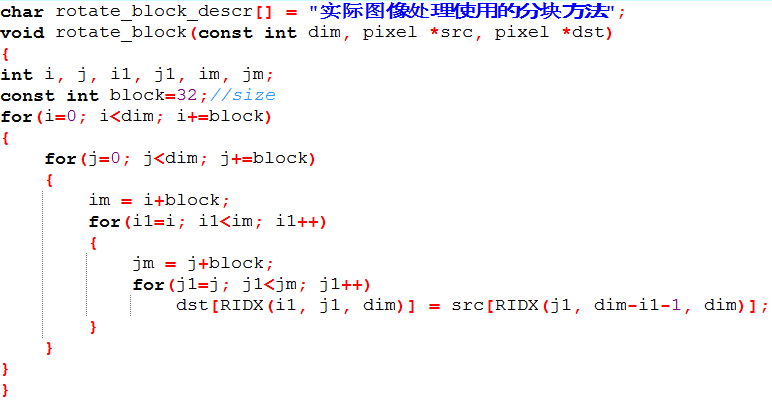


不死心。试一下两个pixel一起写：



然而这样反而又变慢了一点。看来存储的速度总是跟不上CPU。当CPU的流水线能够用足，但缓存miss导致存储跟不上时，总体速度依然被拖累。

到此已经束手无策了。看看网上的方法，发现有人采用分块，跑出了16以上的分数。考虑到实际图像处理中也经常分块，于是试试：



分数达到了惊人的15.5以上。无法理解的是，分块的速度比列向读取，行向写入更快。没有找到充分的理论依据！推测分块在一定程度上可以提升写缓存命中率。此外块的大小应与缓存匹配，以获得最多的命中。

此外，密集地读写内存导致src[RIDX(j1,dim-i1-1,dim)]这样的少量加法乘法计算并不会拖慢速度。

1. Smooth的优化

从rotate里已经了解到，优化的本质就是大力出奇迹暴力展开代码（包括减少函数调用、减少分支判断、展开循环、减少数据依赖等）以及命中缓存（在一个局部范围内进行读写。“局部”的大小与CPU缓存大小有关。），减少内存读写次数。Smooth是个内存读取与计算密集型任务。于是只将它暴力展开……



代码基本保证了每次只在一个局部范围读写，因此是缓存友好的。没有反复的if语句，能减轻分支预测错误的惩罚。没有反复调用函数，因此不需要开堆栈浪费时间。src[RIDX]之类的表达式计算比内存读写快得多，且多个src[RIDX]在流水线中同时执行，所以可以放心大胆抄下来，而不用担心速度问题。

1. 最后结果

