性能优化分析实验

数据科学与计算机学院

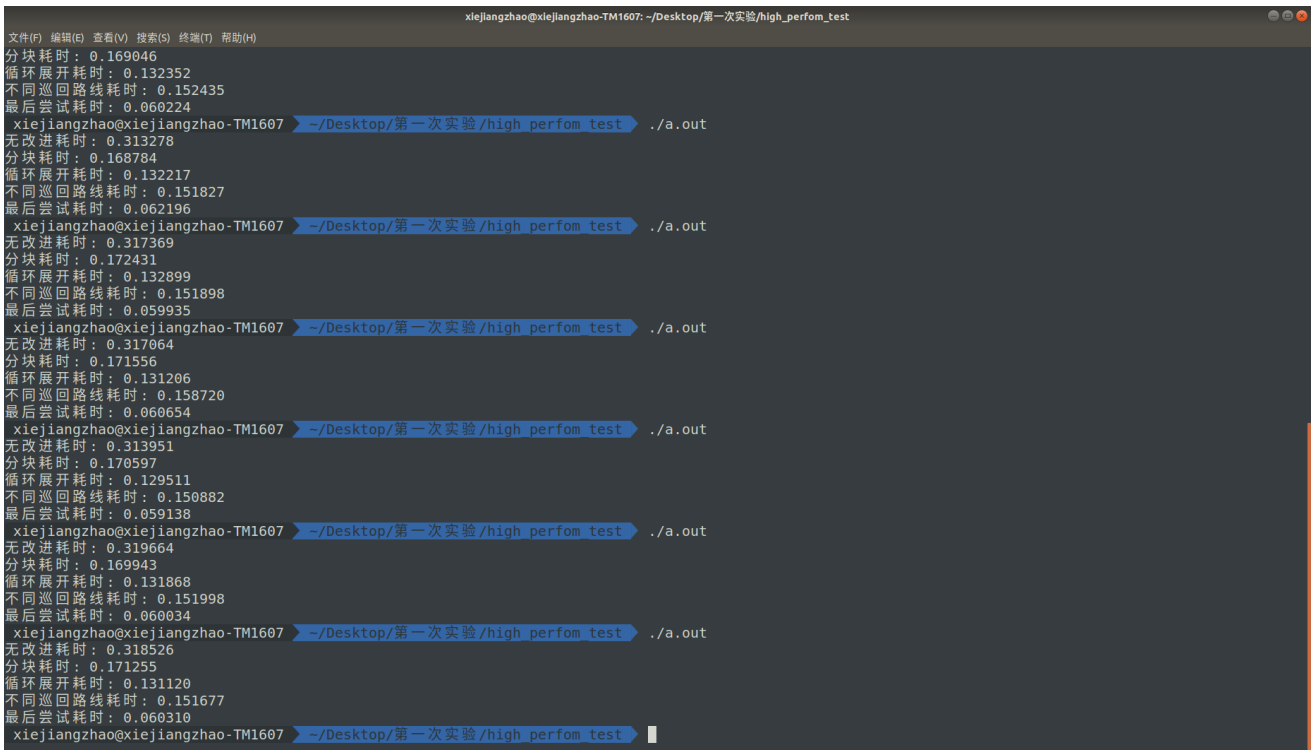
计算机科学与技术(超算方向)2016级

16337259 谢江钊

实验结果

矩阵大小设定为4096*4096

耗时对比



取前五次平均数据进行对比:

	耗时（s）	耗时百分比
无改进耗时	0.31718	1
分块耗时	0.170602	0.537871
循环展开耗时	0.131523	0.414663
不同巡回路线耗时	0.151656	0.47814
最后尝试耗时	0.060323	0.190184

可见都得到了不同程度的提高。

缓存命中情况

无改进：

```
 xiejiangzhao@xiejiangzhao-TM1607 ❱ ~/Desktop/第一次实验/high_perfom_test ❱ sudo perf stat -e cache-references,cache-misses,L1-dcache-loads,L1-dcache-mis
ses,branches,branch-misses ./a.out
无改进耗时：0.324734

 Performance counter stats for './a.out':

       64,937,472      cache-references                                       (66.08%)
       55,771,183      cache-misses              #   85.884 % of all cache refs      (66.81%)
      831,370,399      L1-dcache-loads                                       (67.25%)
       28,642,180      L1-dcache-misses          #    3.45% of all L1-dcache hits    (67.27%)
      405,697,122      branches                                              (66.67%)
          683,942      branch-misses             #    0.17% of all branches          (65.93%)

      0.537857496 seconds time elapsed
```

分块：

```
 xiejiangzhao@xiejiangzhao-TM1607 ❱ ~/Desktop/第一次实验/high_perfom_test ❱ sudo perf stat -e cache-references,cache-misses,L1-dcache-loads,L1-dcache-mis
ses,branches,branch-misses ./a.out
分块耗时：0.192633

 Performance counter stats for './a.out':

       22,976,419      cache-references                                       (66.43%)
       20,043,078      cache-misses              #   87.233 % of all cache refs      (66.41%)
      902,522,026      L1-dcache-loads                                       (66.41%)
        9,725,063      L1-dcache-misses          #    1.08% of all L1-dcache hits    (66.92%)
      415,675,289      branches                                              (67.16%)
          728,228      branch-misses             #    0.18% of all branches          (66.66%)

      0.405379097 seconds time elapsed
```

循环展开：

```
 xiejiangzhao@xiejiangzhao-TM1607 ❱ ~/Desktop/第一次实验/high_perfom_test ❱ sudo perf stat -e cache-references,cache-misses,L1-dcache-loads,L1-dcache-mis
ses,branches,branch-misses ./a.out
循环展开耗时：0.149307

 Performance counter stats for './a.out':

       15,855,107      cache-references                                       (66.82%)
       11,490,593      cache-misses              #   72.473 % of all cache refs      (66.82%)
      779,257,594      L1-dcache-loads                                       (66.82%)
        9,350,253      L1-dcache-misses          #    1.20% of all L1-dcache hits    (66.82%)
      386,706,576      branches                                              (66.36%)
          843,215      branch-misses             #    0.22% of all branches          (66.36%)

      0.361990863 seconds time elapsed
```

不同巡回：

```
xiejiangzhao@xiejiangzhao-TM1607 ~/Desktop/第一次实验/high perfom test sudo perf stat -e cache-references,cache-misses,L1-dcache-loads,L1-dcache-misses,branches,branch-misses ./a.out
不同巡回路线耗时: 0.189036

 Performance counter stats for './a.out':

       22,354,813      cache-references                                       (65.78%)
       19,475,530      cache-misses              #   87.120 % of all cache refs      (66.13%)
      905,091,081      L1-dcache-loads                                       (67.14%)
        9,976,263      L1-dcache-misses          #    1.10% of all L1-dcache hits    (67.79%)
      416,584,853      branches                                              (67.08%)
          701,722      branch-misses             #    0.17% of all branches          (66.08%)

      0.397659854 seconds time elapsed
```

最后尝试:

```
xiejiangzhao@xiejiangzhao-TM1607 ~/Desktop/第一次实验/high perfom test sudo perf stat -e cache-references,cache-misses,L1-dcache-loads,L1-dcache-misses,branches,branch-misses ./a.out
最后尝试耗时: 0.083859

 Performance counter stats for './a.out':

       36,239,484      cache-references                                       (66.06%)
       11,061,684      cache-misses              #   30.524 % of all cache refs      (67.41%)
      543,703,734      L1-dcache-loads                                       (67.52%)
       18,575,717      L1-dcache-misses          #    3.42% of all L1-dcache hits    (67.52%)
      327,115,529      branches                                              (66.41%)
          665,731      branch-misses             #    0.20% of all branches          (65.06%)

      0.295899040 seconds time elapsed
```

测试	Cache miss次数	miss百分比
无改进	55771183	1
分块	20043078	0.3593
循环展开	11490593	0.2060
不同巡回	19475530	0.3492
最后测试	11061684	0.1983

原因分析

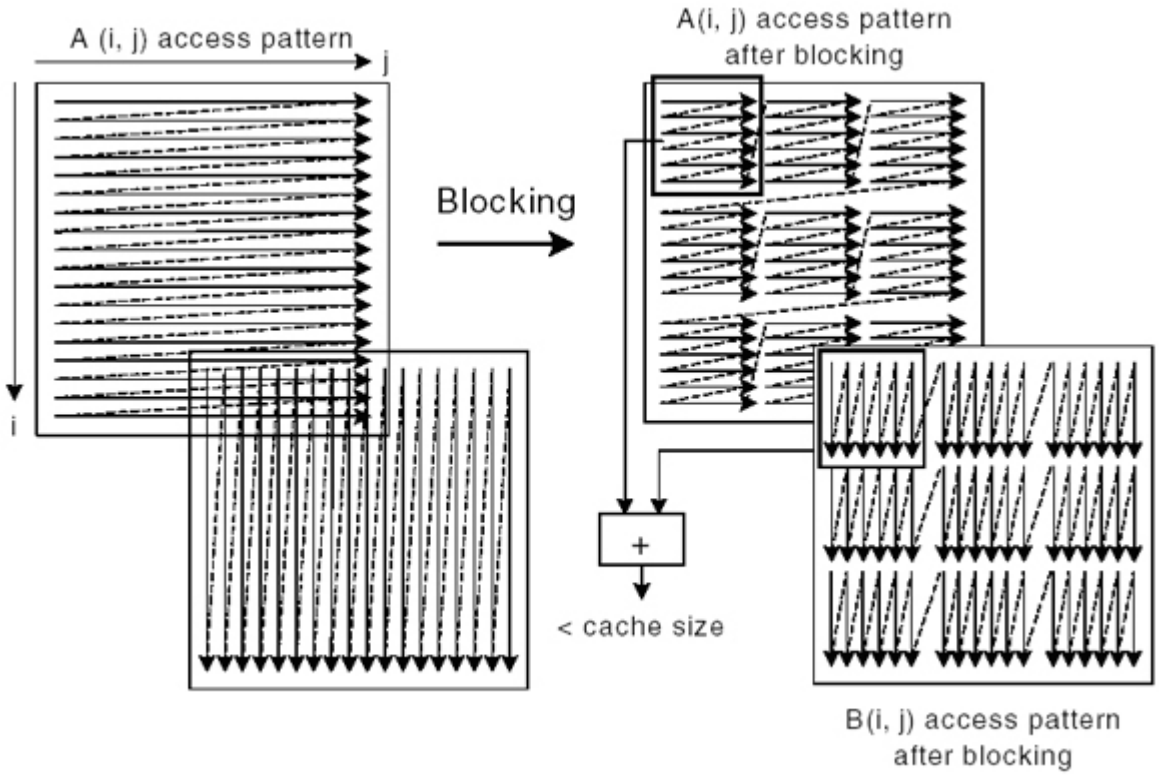
未改进方案

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


    for (int i = 0; i < dim; i++)
    {


        for (int j = 0; j < dim; j++)
        {
            dst[RIDX(dim - 1 - j, i, dim)] = src[RIDX(i, j, dim)];
        }
    }
}
```

分块技术

分块是一个很好的做法，因为注意到我们的矩阵有两个，其中一个在读写的时候会一行一行地取，而另一个读写时会一列一列地取。我们知道，在C语言中，二维矩阵其实是以一维的连续内存存储起来的。在进行缓存的时候，也是取的内存的相邻区域。因此，其中一个矩阵按行逐个读取会准确命中，但是对于另一个，尤其在矩阵比较大时，那么每次按列逐个读取的元素在内存中相距离较远，就容易发生缓存不足而miss掉的情况。尤其是当一列读取到最后一个元素，要回到第二列第一个元素读取的时候，这时两个元素在内存的距离是最远的，往往就会miss，为了解决这个问题，分块将这个范围缩小了，使得每个分块内部的元素距离比较小，因此提高了效率。

查阅了相关的资料（来源Intel:https://software.intel.com/en-us/articles/how-to-use-loop-blocking-to-optimize-memory-use-on-32-bit-intel-architecture），可以看到:



很形象地表现出了这个优化地过程。

循环展开

```
void rotate_1_2(int dim, pixel *src, pixel *dst)
{
    int i, j, ii, jj;
    for (ii = 0; ii < dim; ii += 32)
        for (jj = 0; jj < dim; jj += 32)
            for (i = ii; i < ii + 32; i += 4)
                for (j = jj; j < jj + 32; j += 4)
                {
                    dst[RIDX(dim - 1 - j, i, dim)] = src[RIDX(i, j, dim)];
                    dst[RIDX(dim - 1 - j, i + 1, dim)] = src[RIDX(i + 1, j, dim)];
                    dst[RIDX(dim - 1 - j, i + 2, dim)] = src[RIDX(i + 2, j, dim)];
                    dst[RIDX(dim - 1 - j, i + 3, dim)] = src[RIDX(i + 3, j, dim)];
                    dst[RIDX(dim - 1 - j - 1, i, dim)] = src[RIDX(i, j + 1, dim)];
                    dst[RIDX(dim - 1 - j - 1, i + 1, dim)] = src[RIDX(i + 1, j + 1, dim)];
                    dst[RIDX(dim - 1 - j - 1, i + 2, dim)] = src[RIDX(i + 2, j + 1, dim)];
                    dst[RIDX(dim - 1 - j - 1, i + 3, dim)] = src[RIDX(i + 3, j + 1, dim)];
                    dst[RIDX(dim - 1 - j - 2, i, dim)] = src[RIDX(i, j + 2, dim)];
                    dst[RIDX(dim - 1 - j - 2, i + 1, dim)] = src[RIDX(i + 1, j + 2, dim)];
                    dst[RIDX(dim - 1 - j - 2, i + 2, dim)] = src[RIDX(i + 2, j + 2, dim)];
                    dst[RIDX(dim - 1 - j - 2, i + 3, dim)] = src[RIDX(i + 3, j + 2, dim)];
                    dst[RIDX(dim - 1 - j - 3, i, dim)] = src[RIDX(i, j + 3, dim)];
                    dst[RIDX(dim - 1 - j - 3, i + 1, dim)] = src[RIDX(i + 1, j + 3, dim)];
                    dst[RIDX(dim - 1 - j - 3, i + 2, dim)] = src[RIDX(i + 2, j + 3, dim)];
                    dst[RIDX(dim - 1 - j - 3, i + 3, dim)] = src[RIDX(i + 3, j + 3, dim)];
                }
}
```

循环展开主要是减少了分支预测失败带来的代价，否则根据for循环每次则都要进行判断，而分支预测在开始和结束的时候就可能要因为预测错误而回滚。

采用不同巡回路线

```
void rotate_1_3(int dim, pixel *src, pixel *dst)
{
    int i, j, ii, jj;
    for (ii = 0; ii < dim; ii += 4)
        for (jj = 4; jj < dim + 4; jj = (jj + 4) % dim)
        {
            for (i = ii; i < ii + 4; i++)
                for (j = jj; j < jj + 4; j++)
                    dst[RIDX(dim - 1 - j, i, dim)] = src[RIDX(i, j, dim)];
            if (jj == 0)
                break;
        }
}
```

采用不同巡回路线，其实还是基于分块技术，性能差别不大，另外受到缓存本身大小的影响，相邻分块之间的物理位置可能会会稍微影响性能。前一个分块结束如果下一个分块部分已经在缓存中则会有相应的改善。

最后尝试

最后一个尝试将分块和循环展开结合了起来，而且，写入的地址连续了，这样可以减少存储器的写次数，因此能得到较大的加速。