一、 入手方向

1. 通过阅读 CSAPP 第五章的内容，可以看到，要是想提高程序的性能对其优化，可以从三个方面入手：1.消除循环的低效率 2.减少过程调用 3.消除不必要的存储器引用。
2. 通过查看 kernels.c 的 rotate 函数，可以看到是由两层循环构成，对于过程的引用只有 RIDX（）。那么，要是想优化该程序，则需要消除循环的低效率或者是减少过程的调用。
3. 消除循环的低效率的方法：交换内外循环的次序，循环展开和循环分块处理
4. 减少过程的调用即不使用 RIDX（）

二、 优化程序

版本 1：减少过程调用

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.7     2.2     5.1     8.2     7.8
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.9     18.0    9.0     8.0     12.2    10.7

Rotate: Version = rotate: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.7     2.3     5.1     8.3     7.8
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.9     17.8    9.1     7.9     12.1    10.7
```

```
48 void rotate(int dim, pixel *src, pixel *dst)
49 {
50     int i,j;
51     for (i = 0; i < dim; i++)
52     for (j = 0; j < dim; j++)
53         dst[(dim-1-j)*dim+i] = src[i*dim+j];
54 }
```

在 defs.h 文件中可以得到 RIDX 的定义：

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

减少其调用，即用数学表达式来替换，发现效率根本没有提升，说明该切入点不对。

因此不打算将这个优化版本作为提交版本。

版本 2：循环分块（代码一）

```
48 void rotate(int dim, pixel *src, pixel *dst)
49 {
50     int i,j;
51     int i1,j1,block;
52     block=4;
53     for (i1=0;i1<dim;i1+=block)
54         for (j1=0;j1<dim;j1+=block)
55             for (i=i1;i<i1+block;i++)
56                 for (j=j1;j<j1+block;j++)
57                     dst[RIDX(dim-1-j, i, dim)] = src[RIDX(i, j, dim)];
58 }
```

分块必然是将其分为 2 的 x 次方个块，先从 4 开始，对其分块处理，分成 4*4 的划分，以此来充分利用 cache

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.6     2.3     4.7     8.2     7.5
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.9     17.2    9.9     8.1     12.6    10.9

Rotate: Version = rotate: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.9     2.2     2.5     2.8     3.7
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         7.6     18.5    18.4    23.3    25.8    17.3
```

有较为明显的提升。

查看 fcyc.c 中的代码可以看到 cache 的大小

```
#define CACHE_BYTES (1<<19)
#define CACHE_BLOCK 32
```

所以其效率与分块的大小有关。通过从 2 依次增加，可以发现得到的 speedup 有一个峰值，位于 block=8 处，也就是说，分为 8*8 可以得到最佳性能，而当分块大于 32 时，则无法得到正确的程序。

为了进一步优化，可以保证 cache 写命中。但是进行这样的改进后，性能有提升，而且最佳 speedup 则位于 block=16 处。

```
48 void rotate(int dim, pixel *src, pixel *dst)
49 {
50     int i,j;
51     int i1,j1,block;
52     block=16;
53     for (i1=0;i1<dim;i1+=block)
54         for (j1=0;j1<dim;j1+=block)
55             for (i=i1;i<i1+block;i++)
56                 for (j=j1;j<j1+block;j++)
57                     dst[RIDX(i, j, dim)] = src[RIDX(j, dim-i-1, dim)];
58 }
```

将 block 设置为最优的 16，改变 dst 和 src 每次进行操作的位置，使对 dst 操作的顺序连续，保证 cache 写命中。将原先 src 的顺序读取更改为 dst 的顺序写入。

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.7     2.3     4.9     8.2     7.6
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.8     17.2    9.4     8.1     12.5    10.7

Rotate: Version = rotate: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.5     1.6     1.7     2.9
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         10.0    26.7    28.7    38.3    32.4    24.9
```

可以看出性能有极大的提升，speedup 比为 2.33。

该版本为提交版本 1。

版本 3：循环展开（代码二）

```
48 void rotate(int dim, pixel *src, pixel *dst)
49 {
50     int i, j;
51     int i1,j1;
52     for (i1 = 0; i1 < dim; i1+=2)
53     for (j1 = 0; j1 < dim; j1+=2)
54     {
55         i=i1;
56         j=j1;
57         dst[RIDX(dim-1-j, i, dim)] = src[RIDX(i, j, dim)];
58         j++;
59         dst[RIDX(dim-1-j, i, dim)] = src[RIDX(i, j, dim)];
```

先对 i 和 j 各做两次展开，在内部补足减少的循环的次数，即对原循环的展开的模拟。为节省版面，剩余 4 行为 56-59 的重复。

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.7     2.3     4.4     8.4     7.5
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.8     17.5    10.6    7.8     12.5    11.0

Rotate: Version = rotate: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.6     2.2     3.1     4.6     7.9
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         9.0     18.6    15.1    14.3    11.9    13.4
```

可以看到有较小的提升

通过增大展开的幅度，可以找到当展开次数为 8 时，能够得到最优。

不过，考虑到对 j 的展开对程序的影响不大，而对 i 展开则有较大改善，所以做出如下优化

```
48 void rotate(int dim, pixel *src, pixel *dst)
49 {
50     int i, j;
51     int i1;
52     for (i = 0; i < dim; i+=16)
53     for (j = 0; j < dim; j++)
54     {
55         i1=i;
56         dst[RIDX(dim-1-j, i1, dim)] = src[RIDX(i1, j, dim)];
57         i1++;
58         dst[RIDX(dim-1-j, i1, dim)] = src[RIDX(i1, j, dim)];
```

只对 i 进行展开，每次展开 16 次（根据多次测验得到的最优），内部模拟每次 i 的循环的展开对其进行写入。

通过这种方式，可以增加 cache 写命中，同时更好的减少循环的低效性。由于篇幅原因，其余部分为 57 58 行代码的重复，共有 16 次。

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.7     2.3     4.3     8.2     7.5
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         8.9     17.5    10.7    8.1     12.6    11.1

Rotate: Version = rotate: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       1.5     1.5     1.4     1.5     2.3
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         10.1    27.5    32.2    44.2    41.6    27.8
```

根据数据可以看到，该方法的优化使得 speedup 之比变为 2.53，为最为优化的一个版本。该版本作为提交版本 2。