性能优化分析实验报告

劳马东 16337113
数据科学与计算机学院
计算机科学与技术（超算方向）

2018 年 9 月 17 日

1 图像旋转

1.1 简单旋转

当发生 cache 缺失时，数据会从高级存储层次拷贝到低级存储层次（如从 L2 cache 到 L1 cache），拷贝的单位是一条 cache line（或者说一个块）而不是所访问的一个数据单位。cache line 的典型大小时 64B，在本例中就是 32 个 pixel 的大小，L1 cache 的大小为 32K。图 1 中的代码是一个简单的二位数组循环，从 L1 数据 cache 利用率的角度来讲，

a. 对 src 数组的访问，空间局部性较好，每读 32 个 pixel 会导致一次 L1 级 cache 缺失 (cache line 的长度为 64 个字节)，因此 cache 缺失次数为：

$$dim \times \frac{dim}{32} = \frac{dim^2}{32} \tag{1}$$

b. 对 dst 数组的访问，空间局部性极差，假设 dim 大于 L1 级数据 cache 的行数，那么对 dst 的每一次写操作都会导致一次缺失，因此 cache 缺失次数为：

$$dim \times dim = dim^2 \tag{2}$$

于是总的 cache 缺失次数为：

$$\frac{dim^2}{32} + dim^2 = \frac{33dim^2}{32} \tag{3}$$

```
typedef short pixel;

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

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)];
}
```

图 1: 简单旋转

1.1.1 测试结果

测试数组 $dim = 2048$，用 $perfstat$ 指令记录 L1 数据 cache 的读取和缺失次数。如图 2，总共读取 59148201 次，发生 5173774 次 cache 缺失，缺失率为 8.75%。

```
laomd@lenovo:rotate$ perf stat  -e L1-dcache-loads,L1-dcache-load-misses ./naive_rotate 0

 Performance counter stats for './naive_rotate 0':

       59,148,201      L1-dcache-loads
        5,173,774      L1-dcache-load-misses     #    8.75% of all L1-dcache hits

      0.098690775 seconds time elapsed
```

图 2: 简单旋转结果

1.2 第一次尝试：4×4 分块

在图 1 中，对 dst 的写操作一次仅仅只是利用了一条 cache line 的一个数据单位。为了增加写操作的空间局部性，提高 cache line 的利用率，可采用分块的方法。图 3 采用了 4×4 分块，读操作的 cache 缺失图一相同，而写操作的 cache 缺失数目减少，一次连续利用了一条 cache line 的 4 个数据单位，即每写 4 次发生一次缺失，因此，写操作的 cache 缺失数目为：

$$dim \times \frac{dim}{4} = \frac{dim^2}{4} \tag{4}$$

总的 cache 缺失数目为：

$$\frac{dim^2}{32} + \frac{dim^2}{4} = \frac{9dim^2}{32} \tag{5}$$

```
void naive_rotate_1(int dim, pixel *src, pixel *dst)
{
    int i, j, ii, jj;
    for (ii = 0; ii < dim; ii+=4)
        for (jj = 0; jj < dim; jj+=4)
            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)];
}
```

图 3: 4×4 分块

1.2.1 测试结果

如图 4，4×4 分块的 cache 缺失的次数明显比不分块的小很多，是其 $\frac{1547961}{5173774} \approx \frac{1}{3.34} \approx \frac{\frac{9dim^2}{32}}{\frac{33dim^2}{32}}$

```
laomd@lenovo:rotate$ perf stat  -e L1-dcache-loads,L1-dcache-load-misses ./naive_rotate 1

 Performance counter stats for './naive_rotate 1':

       66,967,553      L1-dcache-loads
        1,547,961      L1-dcache-load-misses     #    2.31% of all L1-dcache hits

      0.040922393 seconds time elapsed
```

图 4: 4×4 分块结果

1.3 第二次尝试：循环展开

一条 cache line 最多可以存储 32 个 pixel，因此采用 32×32 分块对于写操做的 cache 利用率是最好的，此时写操作的 cache 缺失数目为：

$$\frac{dim^2}{32} \tag{6}$$

故总的 cache 缺失数目为：

$$\frac{dim^2}{32} + \frac{dim^2}{32} = \frac{dim^2}{16} \tag{7}$$

此外，循环展开可以降低循环开销，为具有多个功能单元的处理器提供指令级并行。如图 6 是该方案的代码。

1.3.1 测试结果

如图 5：

```
laomd@lenovo:rotate$ perf stat  -e L1-dcache-loads,L1-dcache-load-misses ./naive_rotate 2

 Performance counter stats for './naive_rotate 2':

       47,474,164      L1-dcache-loads
        1,432,214      L1-dcache-load-misses     #    3.02% of all L1-dcache hits

      0.028030431 seconds time elapsed
```

图 5: 32×32 分块，4×4 路循环展开结果

```
void naive_rotate_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-2,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)];
                }
}
```

图 6: 32×32 分块，4×4 路循环展开

1.4 第三次尝试：采用不同的巡回路线

第一次访问出现 cache 缺失的数据时，其对应的 cache line 被拷贝到 cache，之后对该 cache line 其余数据的访问都能命中。因此，对 cache line 元素的访问顺序不会影响 cache 缺失数，即采用不同巡回路线的 cache 缺失数与第二次尝试相同。

1.4.1 测试结果

```
void naive_rotate_3(int dim, pixel *src, pixel *dst)
{
    int i, j, ii, jj;
    for (ii = 0; ii < dim; ii+=4)
        for (jj = 0; jj < dim; jj+=4)
            for (i = ii; i < ii+4; i++) {
                for (j = jj + 1; j < jj+4; j++)
                    dst[RIDX(dim-1-j, i, dim)] = src[RIDX(i, j, dim)];
                dst[RIDX(dim-1-jj, i, dim)] = src[RIDX(i, jj, dim)];
            }
}
```

图 7: 4×4 分块，不同巡回路线

```
laomd@lenovo:rotate$ perf stat  -e L1-dcache-loads,L1-dcache-load-misses ./naive_rotate 3

Performance counter stats for './naive_rotate 3':

       63,887,455      L1-dcache-loads
        1,568,055      L1-dcache-load-misses     #    2.45% of all L1-dcache hits

      0.035584059 seconds time elapsed
```

图 8: 4×4 分块，不同巡回路线结果

1.5 最后的尝试

分块的两个维度的大小分别影响这 src 和 dst 数组。第一维的大小决定连续访问 src 数组的多少个元素以及访问 dst 数组连续元素的频率，第二维同理，因此需要一个折中。显然，连续访问 dst 数组的元素是有好处的，这样可以减少 cache line 写回的次数，也就节省了时间。

对于图 9 中的方案，采用 32×1 分块，读操作在访问了一列的 32 个元素后重新回到第一行，即访问同一条 cache line 的频率是 $\frac{1}{32}$，能及时地利用被拷贝的 cache line，而写操作在 dst 数组中访问了一行连续的 32 个元素，这是最优的；再将对 32 个元素的访问展开，就能并行地利用多个功能单元，增加指令级并行。

```
#define COPY(d,s) *(d) = *(s)
void naive_rotate_4(int dim, pixel *src, pixel *dst)
{
    int i, j;
    for (i = 0; i < dim; i+=32)
        for (j = dim-1; j >= 0; j--) {
            pixel *dptr = dst+RIDX(dim-1-j,i,dim);
            pixel *sptr = src+RIDX(i,j,dim);
            COPY(dptr, sptr); sptr += dim;      COPY(dptr+1, sptr); sptr += dim;
            COPY(dptr+2, sptr); sptr += dim;    COPY(dptr+3, sptr); sptr += dim;
            COPY(dptr+4, sptr); sptr += dim;    COPY(dptr+5, sptr); sptr += dim;
            COPY(dptr+6, sptr); sptr += dim;    COPY(dptr+7, sptr); sptr += dim;
            COPY(dptr+8, sptr); sptr += dim;    COPY(dptr+9, sptr); sptr += dim;
            COPY(dptr+10, sptr); sptr += dim;   COPY(dptr+11, sptr); sptr += dim;
            COPY(dptr+12, sptr); sptr += dim;   COPY(dptr+13, sptr); sptr += dim;
            COPY(dptr+14, sptr); sptr += dim;   COPY(dptr+15, sptr); sptr += dim;
            COPY(dptr+16, sptr); sptr += dim;   COPY(dptr+17, sptr); sptr += dim;
            COPY(dptr+18, sptr); sptr += dim;   COPY(dptr+19, sptr); sptr += dim;
            COPY(dptr+20, sptr); sptr += dim;   COPY(dptr+21, sptr); sptr += dim;
            COPY(dptr+22, sptr); sptr += dim;   COPY(dptr+23, sptr); sptr += dim;
            COPY(dptr+24, sptr); sptr += dim;   COPY(dptr+25, sptr); sptr += dim;
            COPY(dptr+26, sptr); sptr += dim;   COPY(dptr+27, sptr); sptr += dim;
            COPY(dptr+28, sptr); sptr += dim;   COPY(dptr+29, sptr); sptr += dim;
            COPY(dptr+30, sptr); sptr += dim;   COPY(dptr+31, sptr);
        }

}
```

图 9: 32×1 分块，32 路循环展开

1.5.1 测试结果

```
laomd@lenovo:rotate$ perf stat  -e L1-dcache-loads,L1-dcache-load-misses ./naive_rotate 4

 Performance counter stats for './naive_rotate 4':

        26,683,235      L1-dcache-loads
           645,947      L1-dcache-load-misses     #    2.42% of all L1-dcache hits

       0.020277338 seconds time elapsed
```

图 10: 32×1 分块，32 路循环展开结果