# 6.25

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 高速缓存 | m | C | B | E | S | t | s | b |
| 1 | 32 | 1024 | 4 | 4 | 64 | 24 | 6 | 2 |
| 2 | 32 | 1024 | 4 | 256 | 1 | 30 | 0 | 2 |
| 3 | 32 | 1024 | 8 | 1 | 128 | 22 | 7 | 3 |
| 4 | 32 | 1024 | 8 | 128 | 1 | 29 | 0 | 3 |
| 5 | 32 | 1024 | 32 | 1 | 32 | 22 | 5 | 5 |
| 6 | 32 | 1024 | 32 | 4 | 8 | 24 | 3 | 5 |

# 6.29

## A

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CT | CT | CT | CT | CT | CT | CT | CT | CI | CI | CO | CO |

## B

|  |  |  |  |
| --- | --- | --- | --- |
| 操作 | 地址 | 命中？ | 读出的值 |
| 读 | 0x834 | 否 |  |
| 写 | 0x836 | 是 | 未知 |
| 读 | 0xFFD | 是 | C0 |

第一次访问0x834时不命中，cache向下请求数据，第二次访问该行时该行应该已经有数据了，所以命中但是不知道是什么

# 6.33



第2组只有标记位为BC和B6是有效的。

BC:1011 1100

B6:1011 0110

组标记位：010

块偏移量：00~11

组合以后1011 1100 010 00~1011 1100 010 11，也就是0x1788~0x178B

1011 0110 010 00~1011 0110 010 11，也就是0x16C8~0x16CB

# 6.37

|  |  |  |
| --- | --- | --- |
| 函数 | N=64 | N=60 |
| sumA | 25% | 25% |
| sumB | 100% | 25% |
| sumC | 50% | 25% |

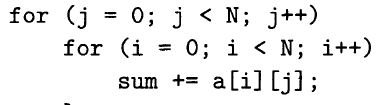
# N=64时

每一个block能放4个元素，16个缓存行能保存一个数组行，整个cache一共256个缓存行，只能放下1/4的整个二维数组，

#### 对于A

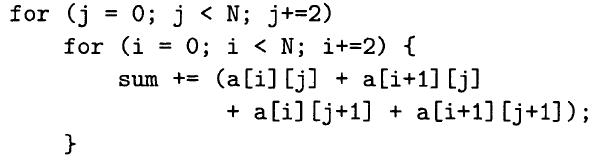
访问顺序很友好，按顺序一个一个访问，每四个数字，都会有第一个不命中然后后三个命中，所以总体命中率大约25%

#### 对于B



访问顺序是跳跃的，每次跳一行，所以刚刚开始全是冷不命中，当i来到1/4的N时，cache已经被全部放满，之后1/4N次循环也是每次都不命中而且会替换之前的缓存行，之后的1/2N次也是同理。当j终于来到1时，cache中还是没有需要的信息（全是后16行数组的数据，而需要的是前16行）所以，总体不命中率100%

#### 对于C



每次访问2\*2的小方块，但是每次访问跨行还是和B同理，都是会造成不命中，但是因为是2\*2访问的，每次不命中之后会有一次命中，也就是a[i][j]不命中a[i][j+1]命中，a[i+1][j]不命中，a[i+1][j+1]命中，所以，总体命中率50%

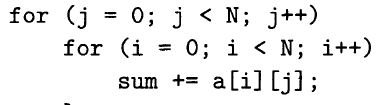
# N=60时

每一个block能放4个元素，15个缓存行能保存一个数组行，整个cache一共256个缓存行。

#### 对于A

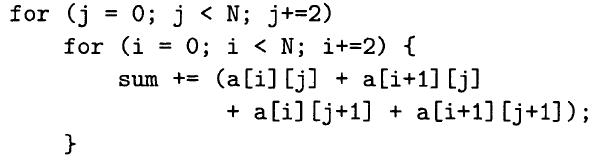
没有什么变化，还是25%，分析与n=64时一样

#### 对于B

****

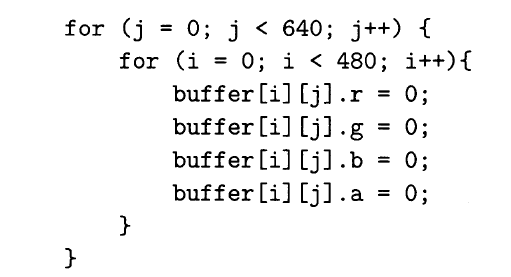
**每15个缓存行能保存一个数组行，因为256不是15的倍数，所以数字映射在cache中的位置将会错开，比如当j=0时，一次访问a[0][0]到a[59][0] ，此时a[0][0]会放在第1个缓存行，a[1][0]会放在第16个缓存行，以此类推1 16 31 46 61 76 91 106 121 136 151 166 181 196 211 226 241 256 15 30 45 60 75 90 105 120 135 150 165 180 195 210 225 240 255 14 29 44 59 74 89 104 119 134 149 164 179 194 209 224 239 254 13 28 43 58 73 88 103 118，不难发现，这些缓存行都没有重复的，所以没有数据会被覆盖，故访问a[0][1]到a[59][1]，a[0][2]到a[59][2]，a[0][3]到a[59][3]，的时候都会命中，之后同理，所以，总体不命中率25%**

#### 对于C



**与B几乎只存在访问顺序上的区别，开始时a[i][j]不命中a[i][j+1]命中，a[i+1][j]不命中，a[i+1][j+1]命中，j+=2之后，每次的四个元素都会命中，原理和B一样（没有数据会被驱逐），也就是每8次访问，不命中2次。**

# 6.41



**一个块保存一个像素点，所以r不命中之后g，b，a都命中，不命中率25%**

# 6.45

由矩阵转置实验得出的经验，将数据分成4\*4，8\*8等大小的矩阵依次处理能显著提高命中率（分块大小取决于cache的block大小，比如在转置实验，块大小为32字节时我们就发现8\*8块操作很高效，因为一个block就能存8个int）

for(int i=0; i<N; i+=B)

{

for(int j=0; j< N; j+= B)

{

for(int x=i; x<N&&x<i+ B; x++)

{

for(int y=j; y< N &&y<j+ B; y++)

{

B[y][x]=A[x][y];

}

}

}

}

这里我们假设cache的块大小为B，我们一次操作B\*B的矩阵以提高命中率。