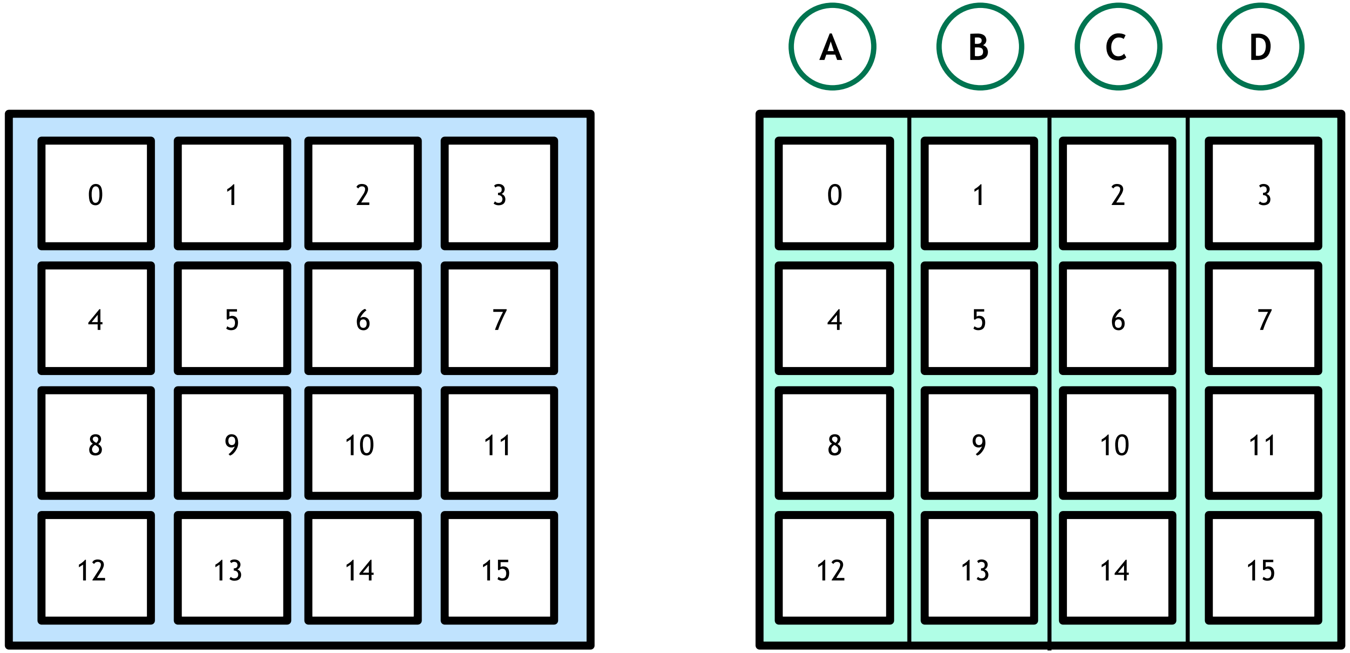
共享内存区的冲突

共享内存在物理上是以区的形式存储的。

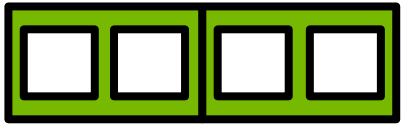


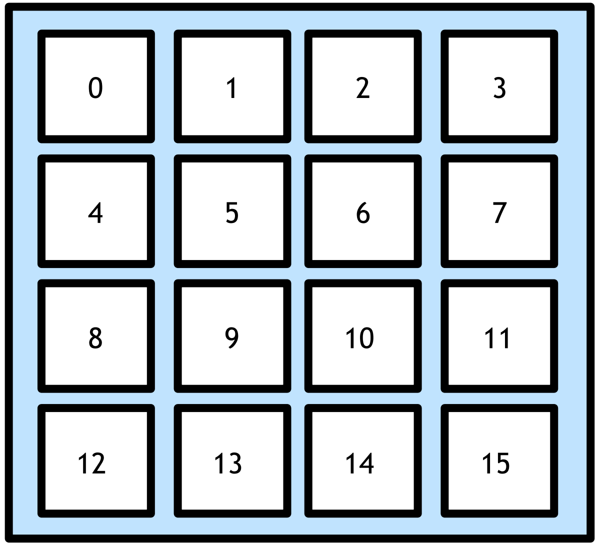
逻辑共享内存
cuda.shared.array(4,4)

物理共享内存
分为 4 个区

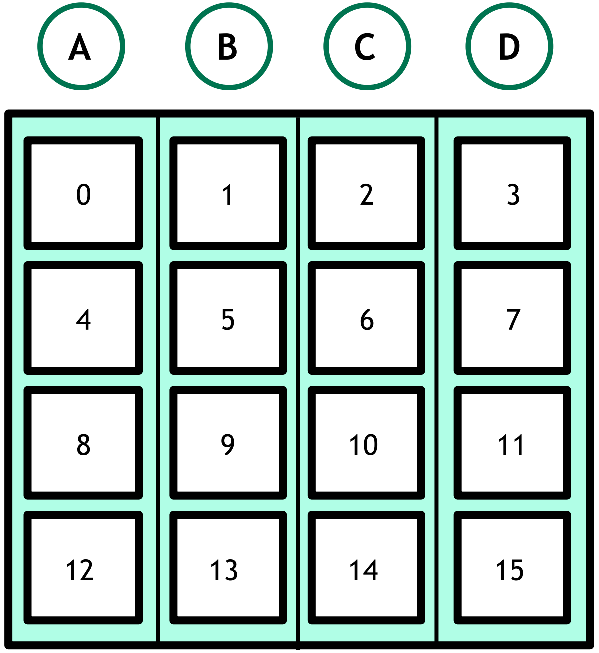
实际共享内存是 32 个 4 字节宽的存储区。 为了利用演示中的页面空间， 我们将共享内存描述为具有 4 个存储区 （A、B、C、D），而一个Warp描述为具有 4 个线程的单位。

Warp



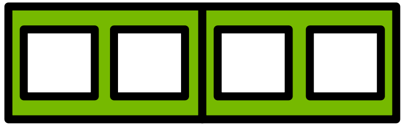


逻辑共享内存
cuda.shared.array(4,4)

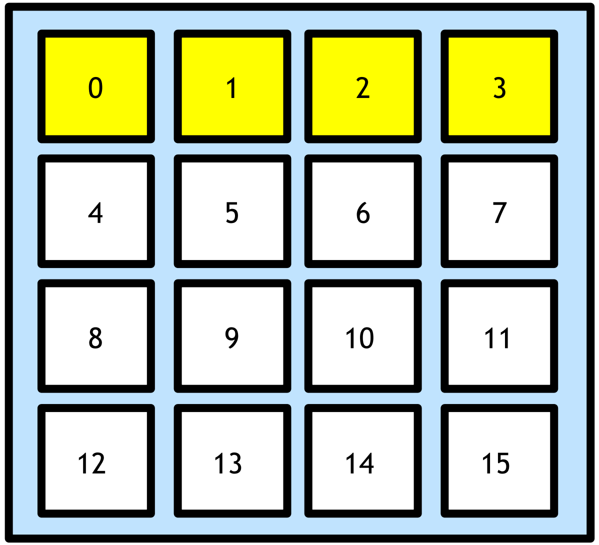


物理共享内存
分为 4 个区

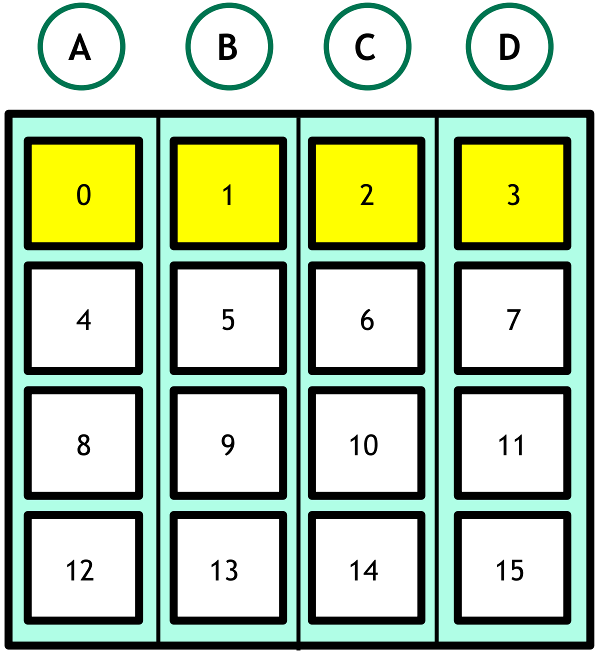
Warp



连续的 4 字节的字（图中的 1 个方块）属于不同的区。

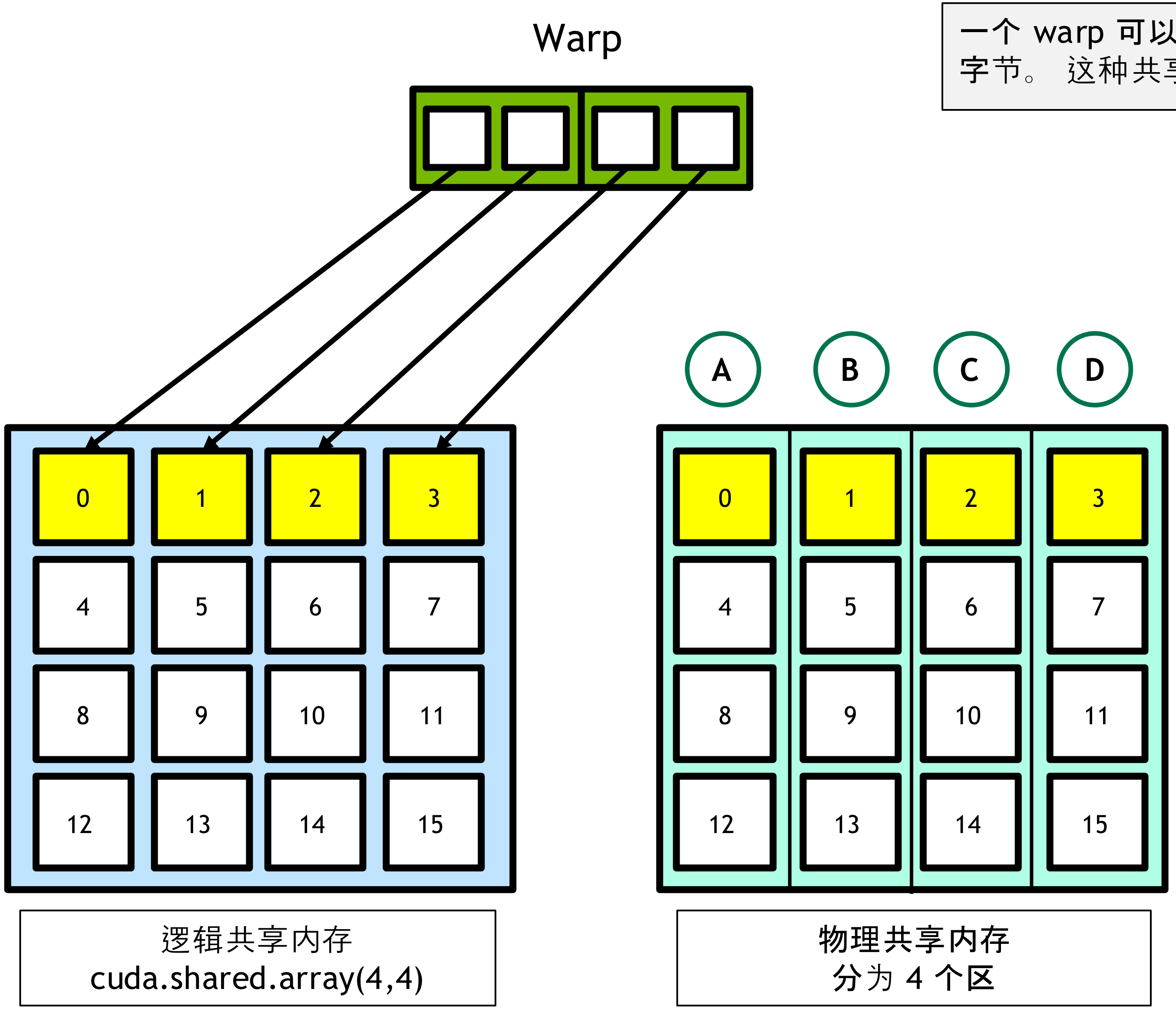


逻辑共享内存
cuda.shared.array(4,4)

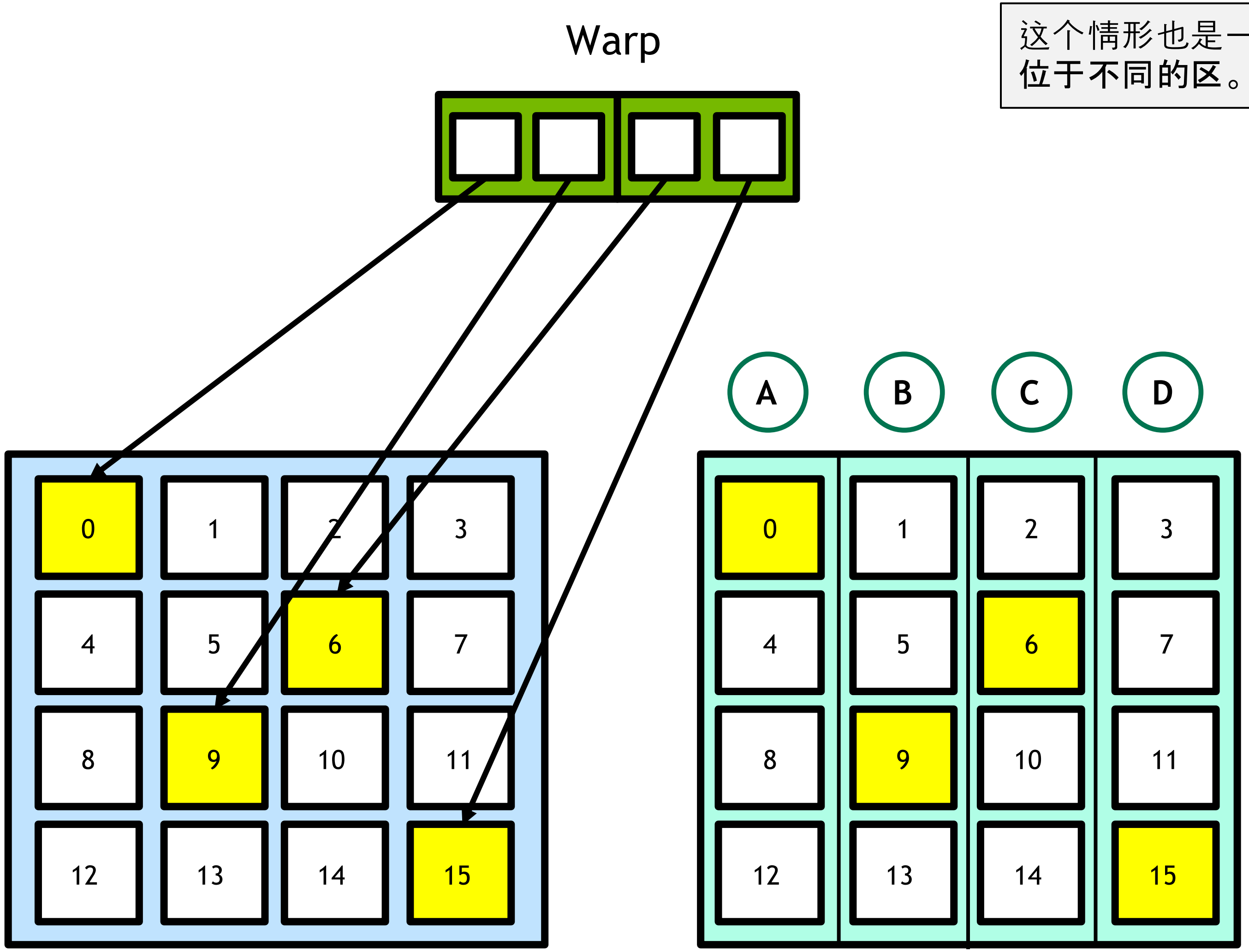


物理共享内存
分为 4 个区

一个 warp 可以并行访问每个区的 4 个字节。 这种共享内存访问将同时发生。



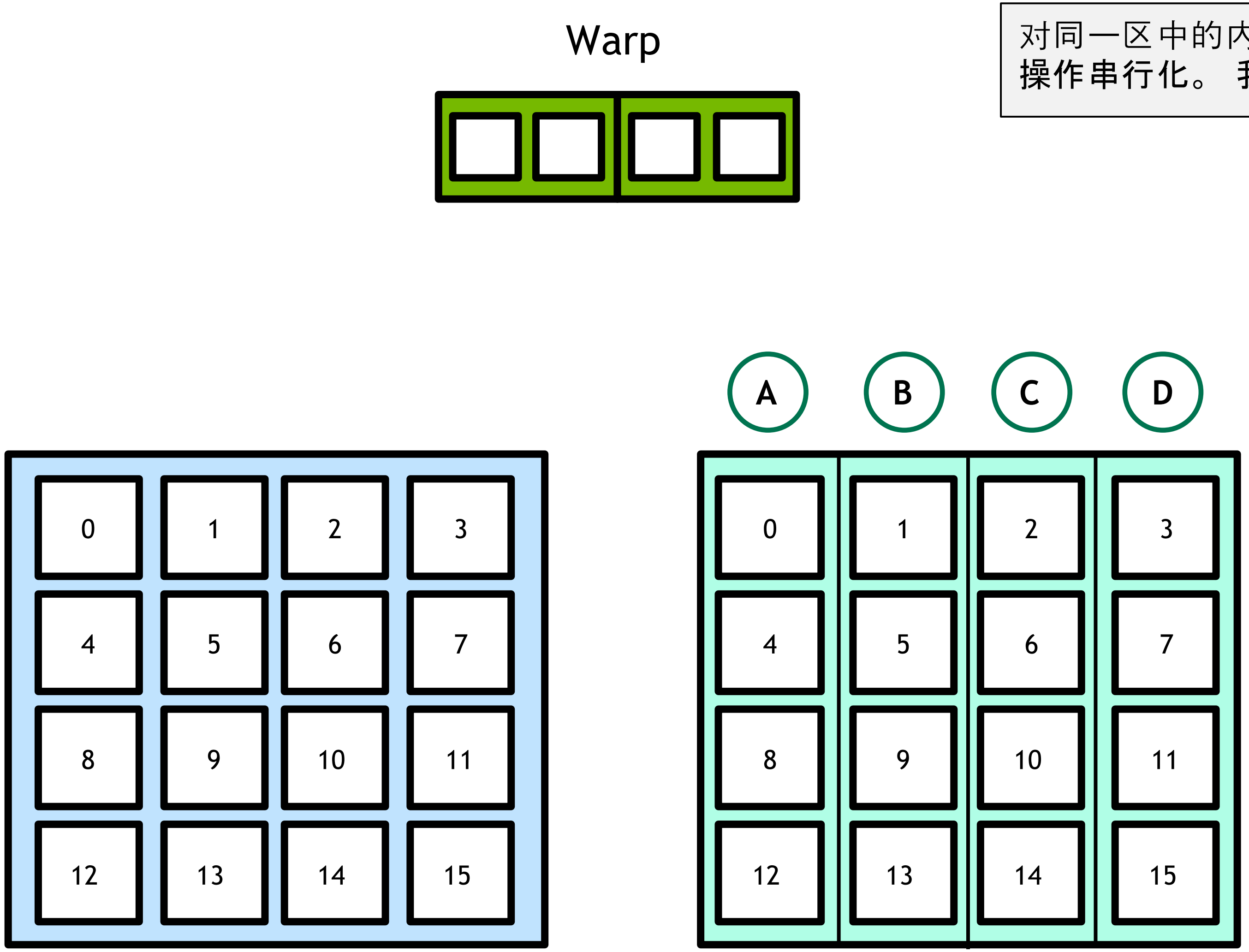
这个情形也是一样，因为每个数据元素位于不同的区。



逻辑共享内存
cuda.shared.array(4,4)

物理共享内存
分为 4 个区

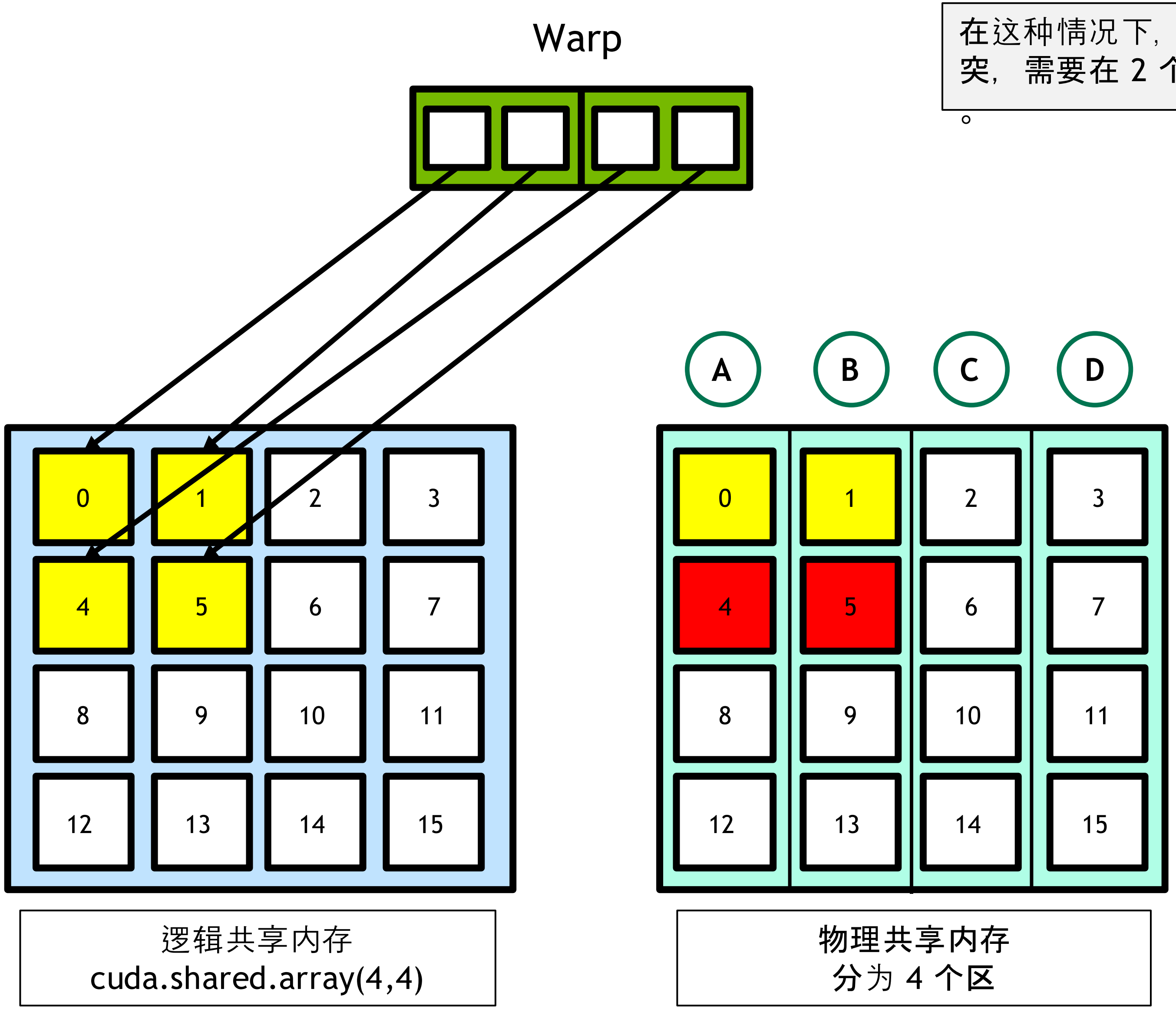
对同一区中的内存进行访问会导致访问操作串行化。 我们称之为**区冲突**。



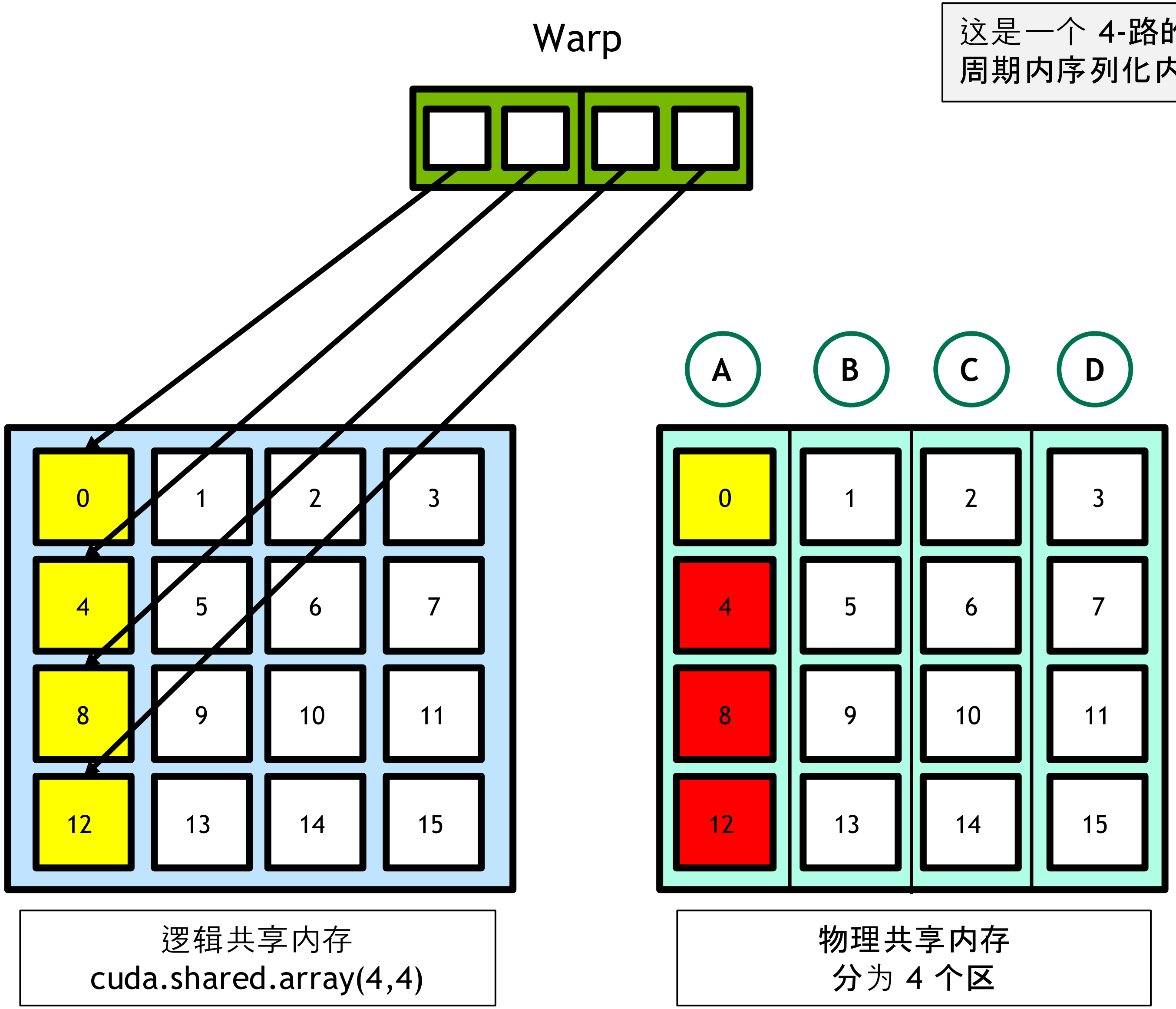
逻辑共享内存
cuda.shared.array(4,4)

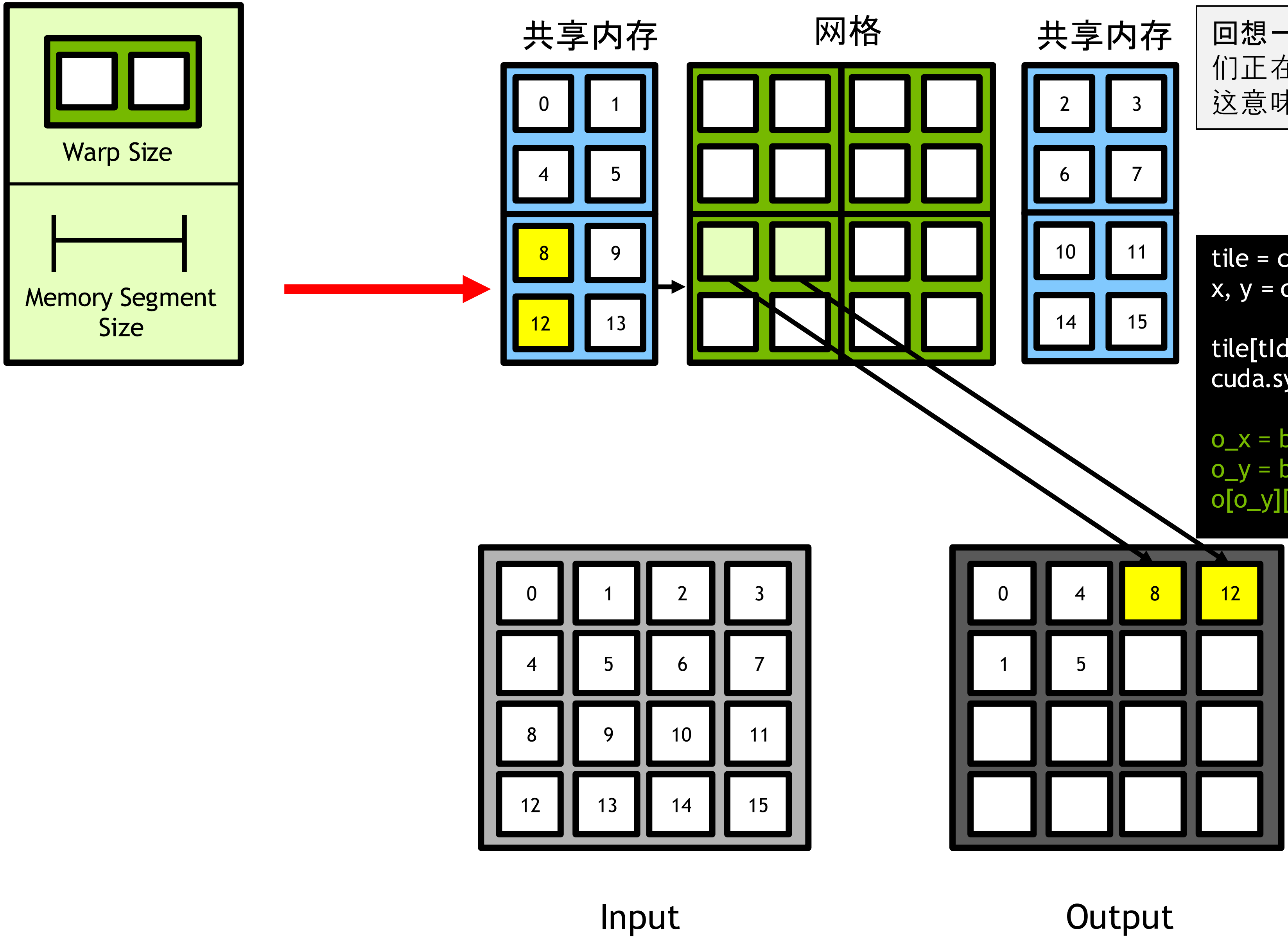
物理共享内存
分为 4 个区

在这种情况下，我们有一个 2-路的区冲突，需要在 2 个周期内序列化内存访问。



这是一个 4-路的区冲突，需要在 4 个周期内序列化内存访问。





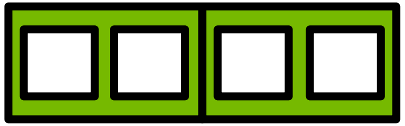
回想一下我们之前的矩阵转置示例，我们正在从共享内存进行这种**列式读取**，这意味着我们有严重的**区冲突**。

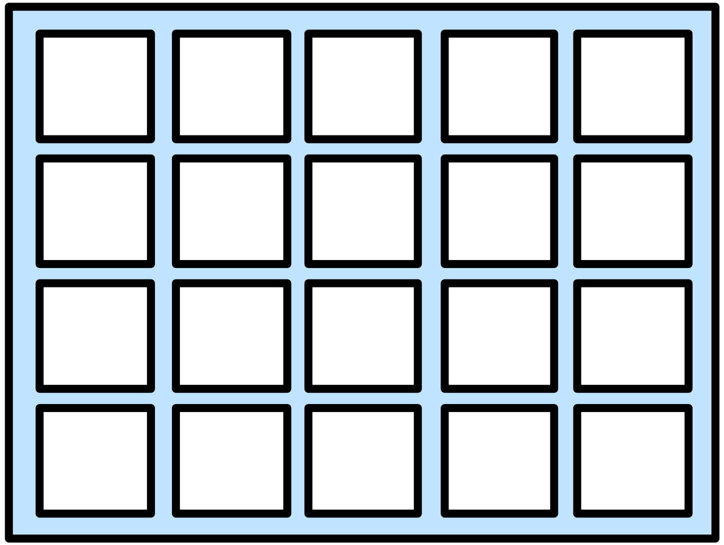
```
tile = cuda.shared.array(2,2)
x, y = cuda.grid(2)

tile[tIdx.y][tIdx.x] = in[y][x]
cuda.syncthreads()

o_x = bId.y*bDim.y + tId.x
o_y = bId.x*bDim.x + tId.y
o[o_y][o_x] = tile[tIdx.x][tIdx.y]
```

当我们知道需要对共享内存进行列访问时，我们可以使用以下技术来避免区冲**突**。

首先，当我们分配共享内存块时，**我们将用**额外的列填充它。

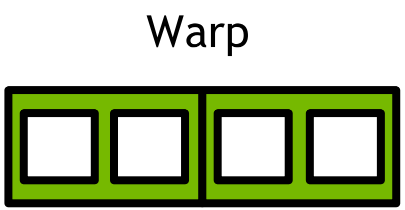
Warp

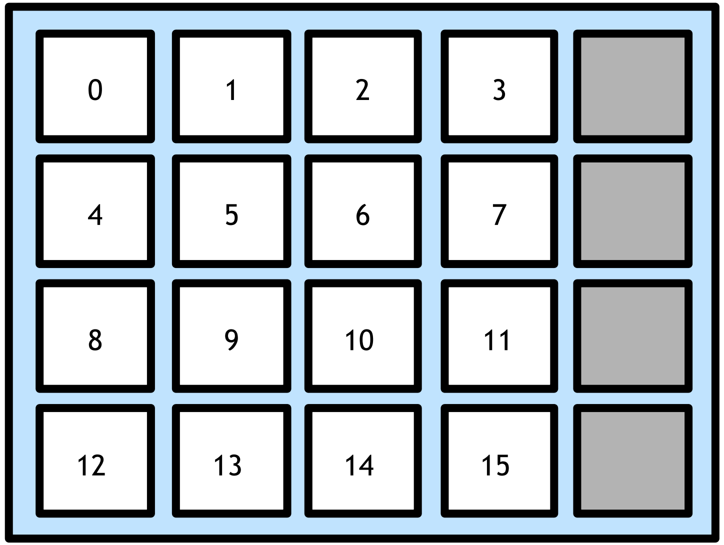




逻辑共享内存
cuda.shared.array(4,5)

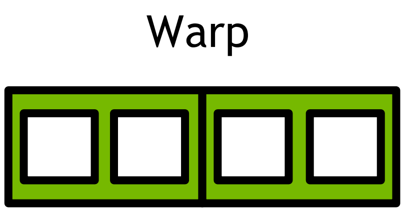
接下来，当我们向共享内存块写入数据时，我们就当它是 (4,4) 一样，只写入范围 [0:4][0:4] 中的地址。

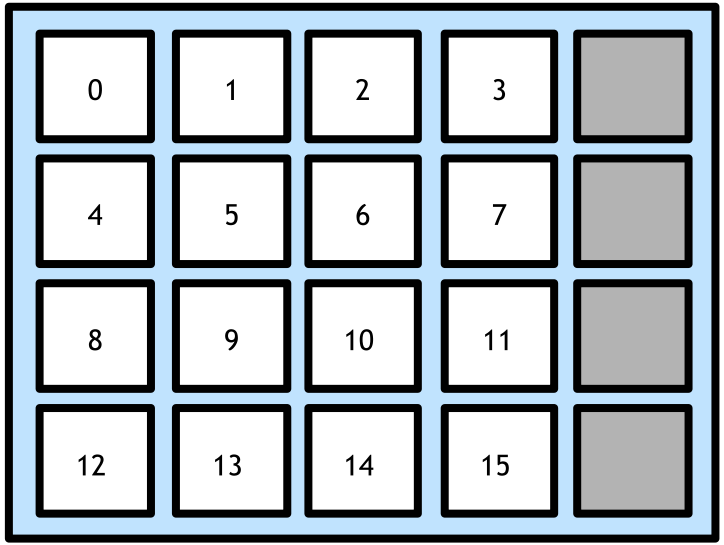




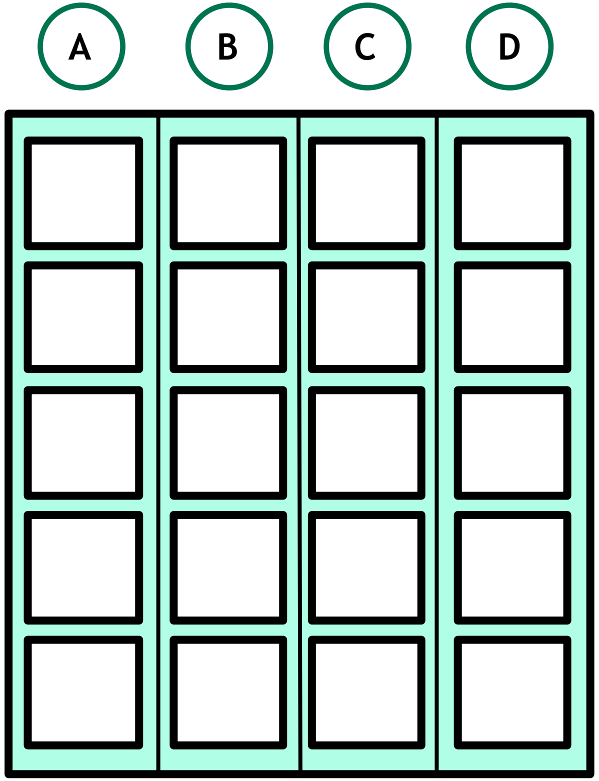
逻辑共享内存
cuda.shared.array(4,5)

物理共享内存的固定大小为 32 个存储区（我们的示例中使用 4 个存储区，以节省页面空间），因此我们对共享存储阵列的填充不会影响存储区的数量。



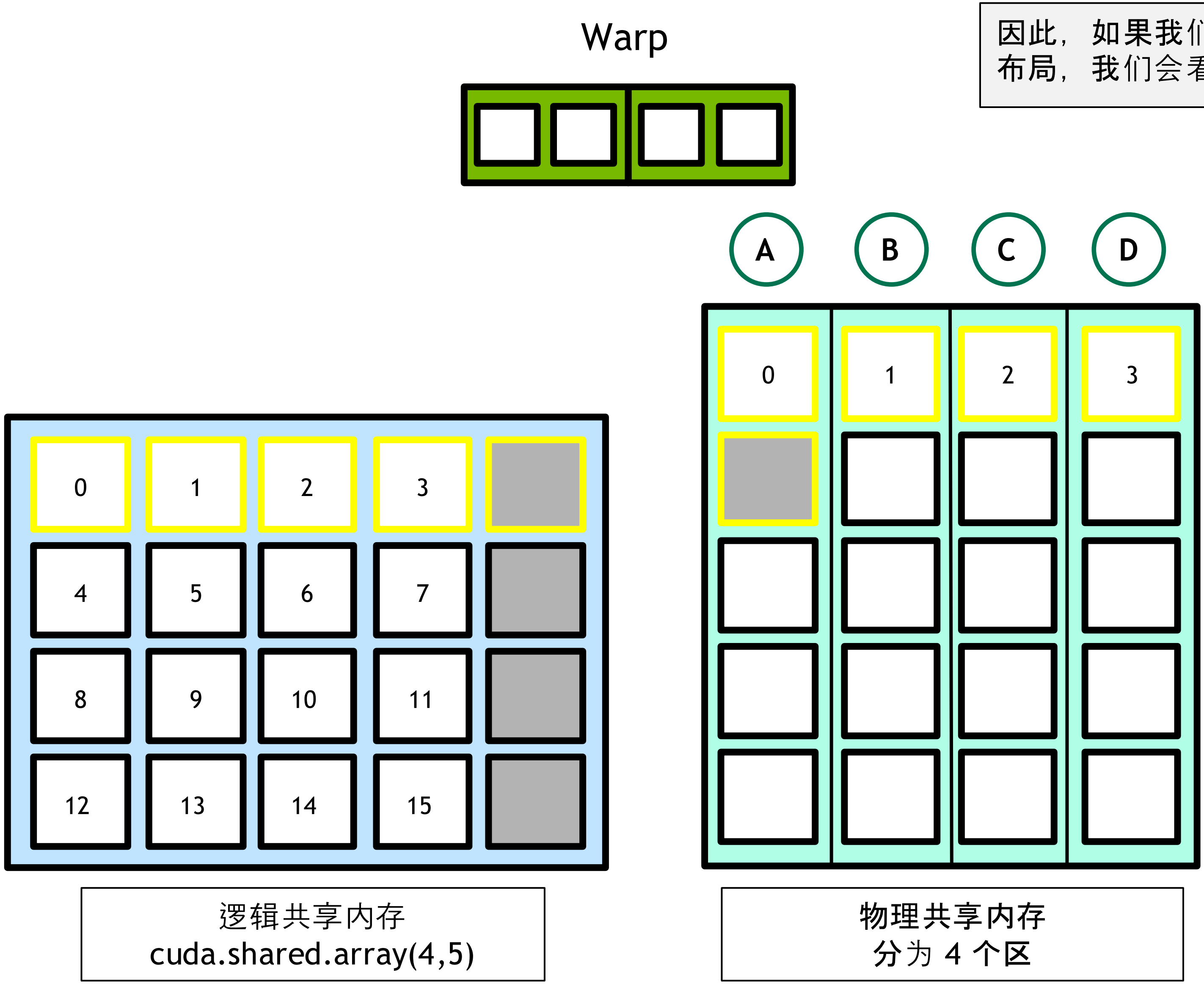


逻辑共享内存
cuda.shared.array(4,5)

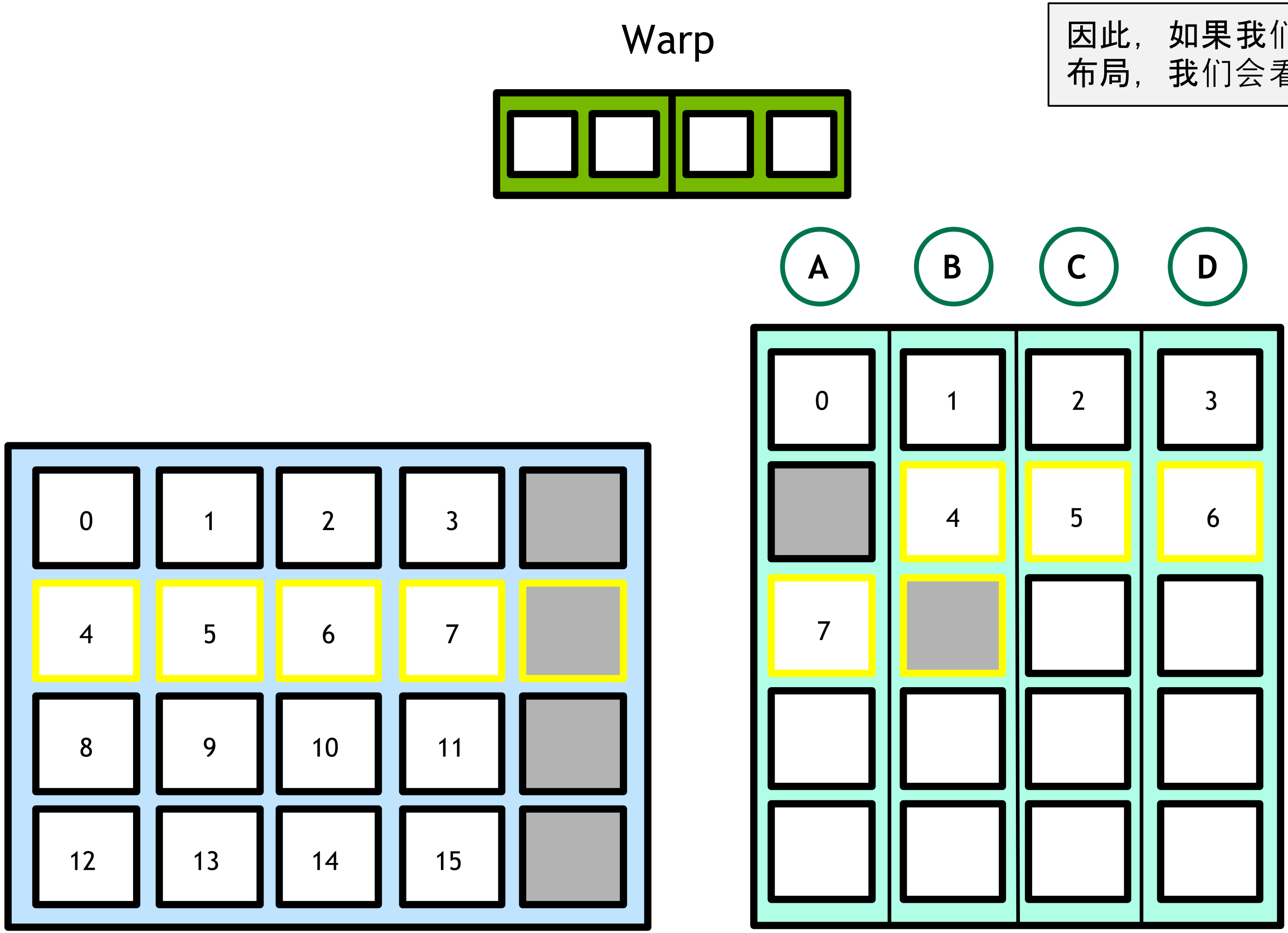


物理共享内存
分为 4 个区

因此，如果我们考虑数组在内存区中的布局，我们会看到以下情景：



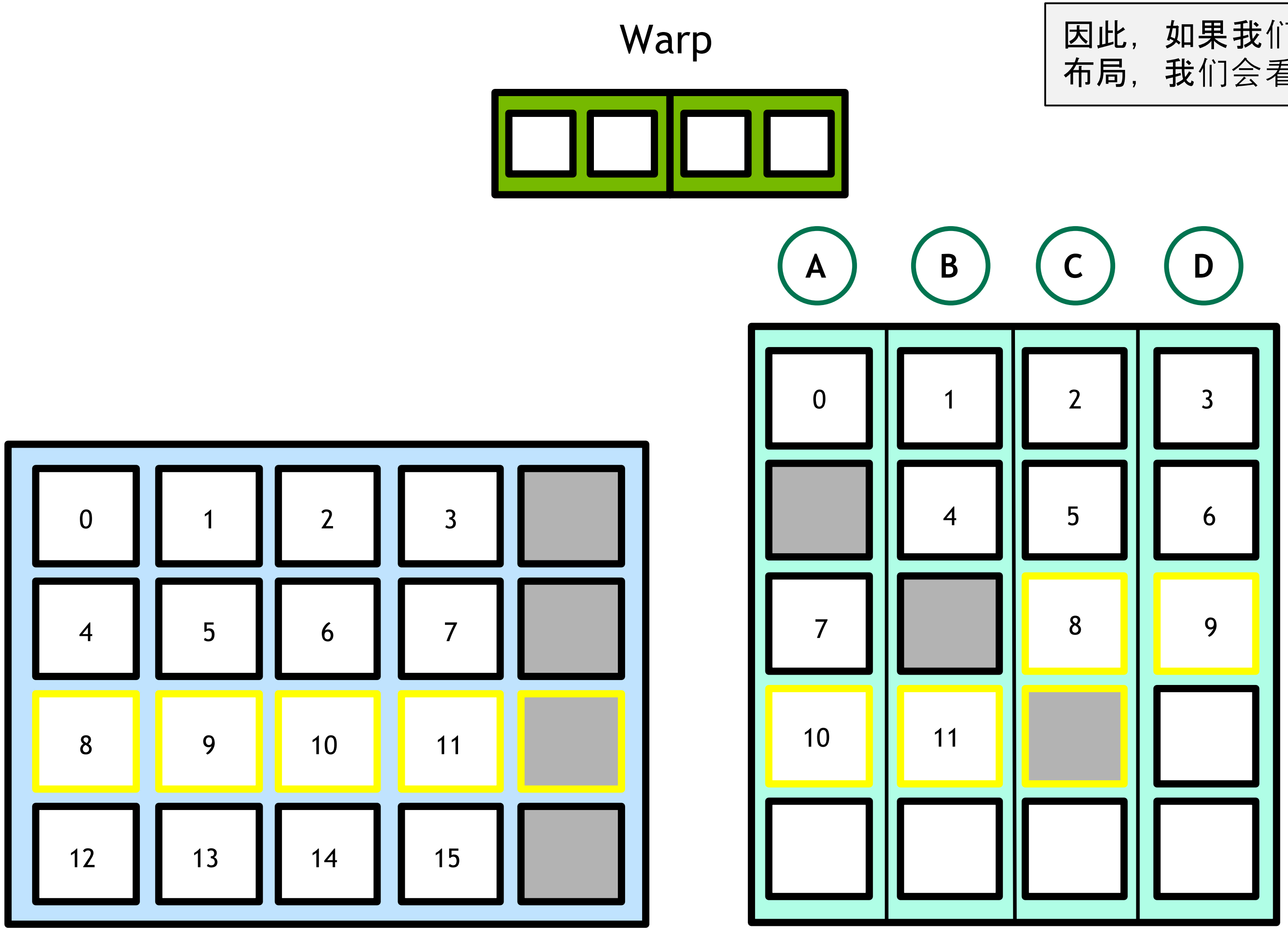
因此，如果我们考虑数组在内存区中的布局，我们会看到以下情景：



逻辑共享内存
cuda.shared.array(4,5)

物理共享内存
分为 4 个区

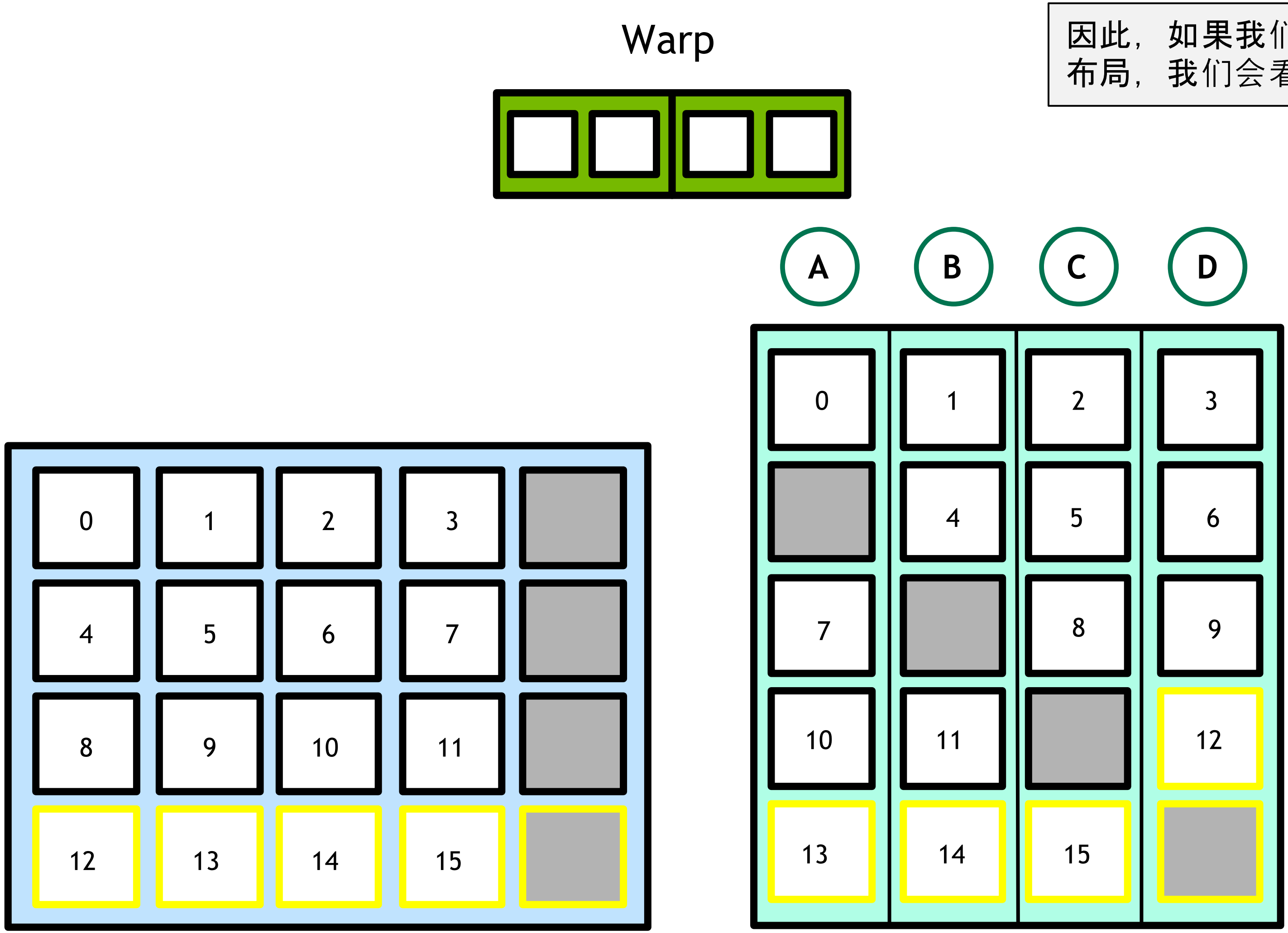
因此，如果我们考虑数组在内存区中的布局，我们会看到以下情景：



逻辑共享内存
cuda.shared.array(4,5)

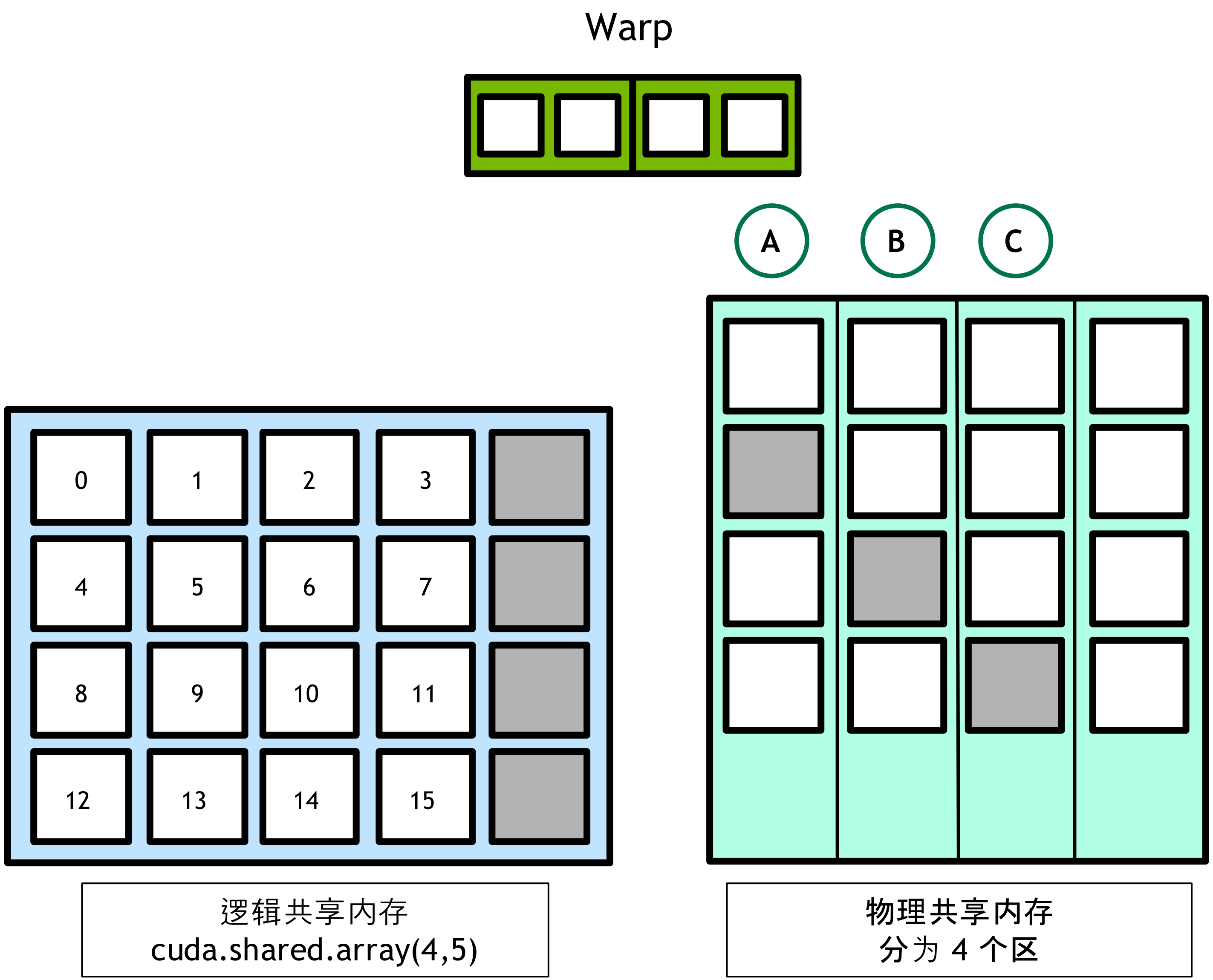
物理共享内存
分为 4 个区

因此，如果我们考虑数组在内存区中的布局，我们会看到以下情景：

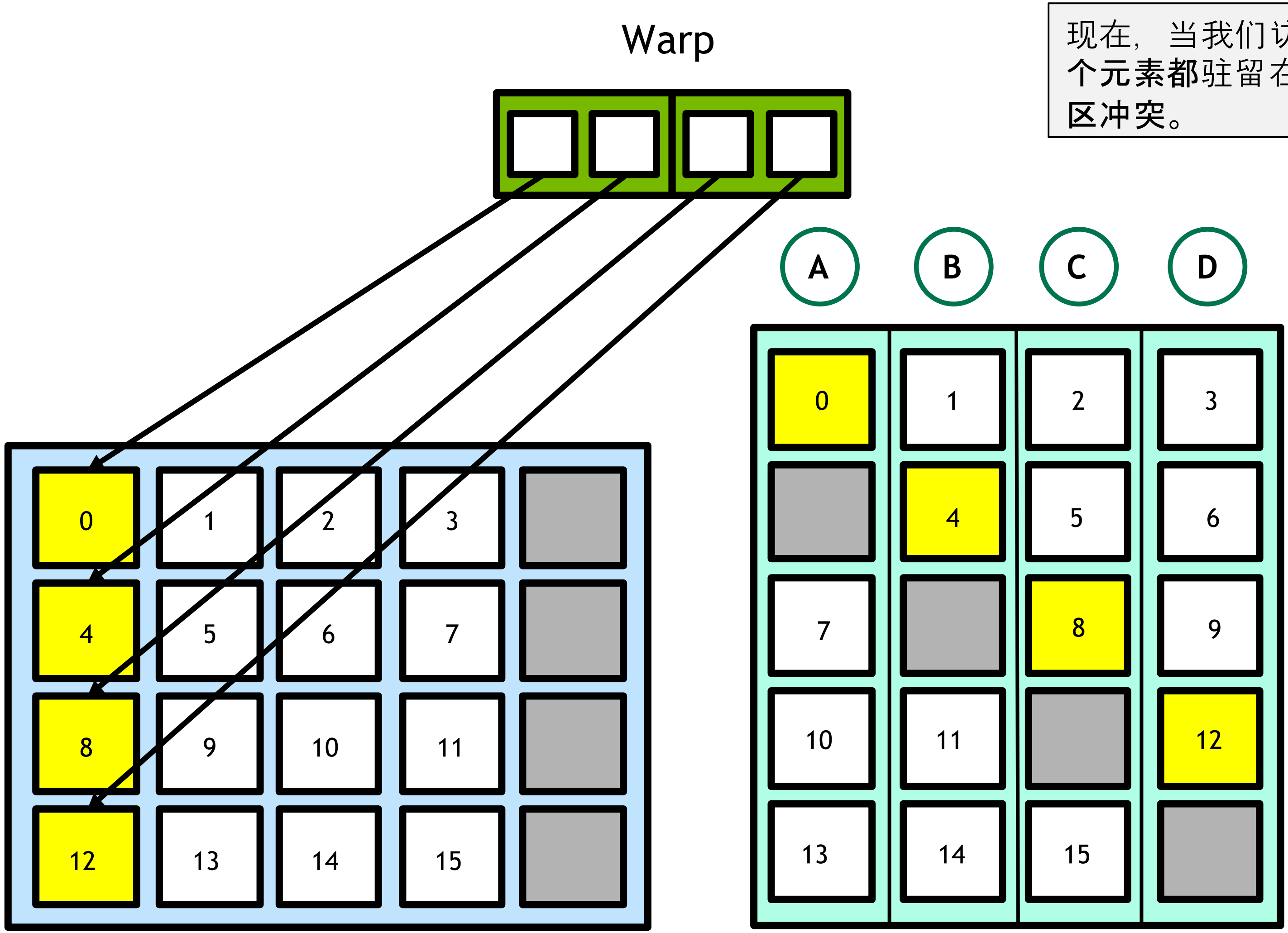


逻辑共享内存
cuda.shared.array(4,5)

物理共享内存
分为 4 个区


Warp
A
B
C
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
逻辑共享内存
cuda.shared.array(4,5)
物理共享内存
分为 4 个区

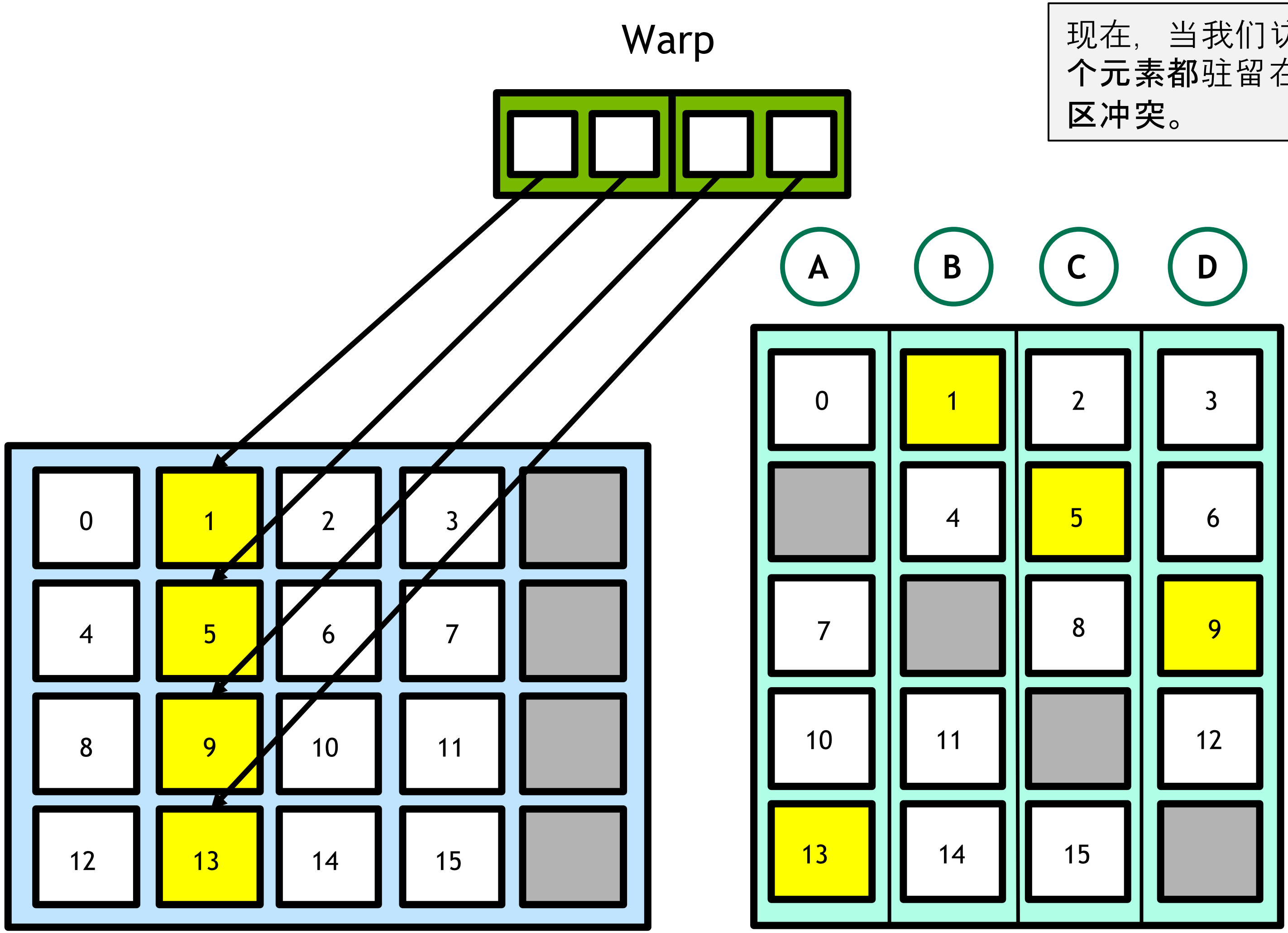
现在，当我们访问一列共享内存时，每个元素都驻留在不同的区中，并且没有区冲突。



逻辑共享内存
cuda.shared.array(4,5)

物理共享内存
分为 4 个区

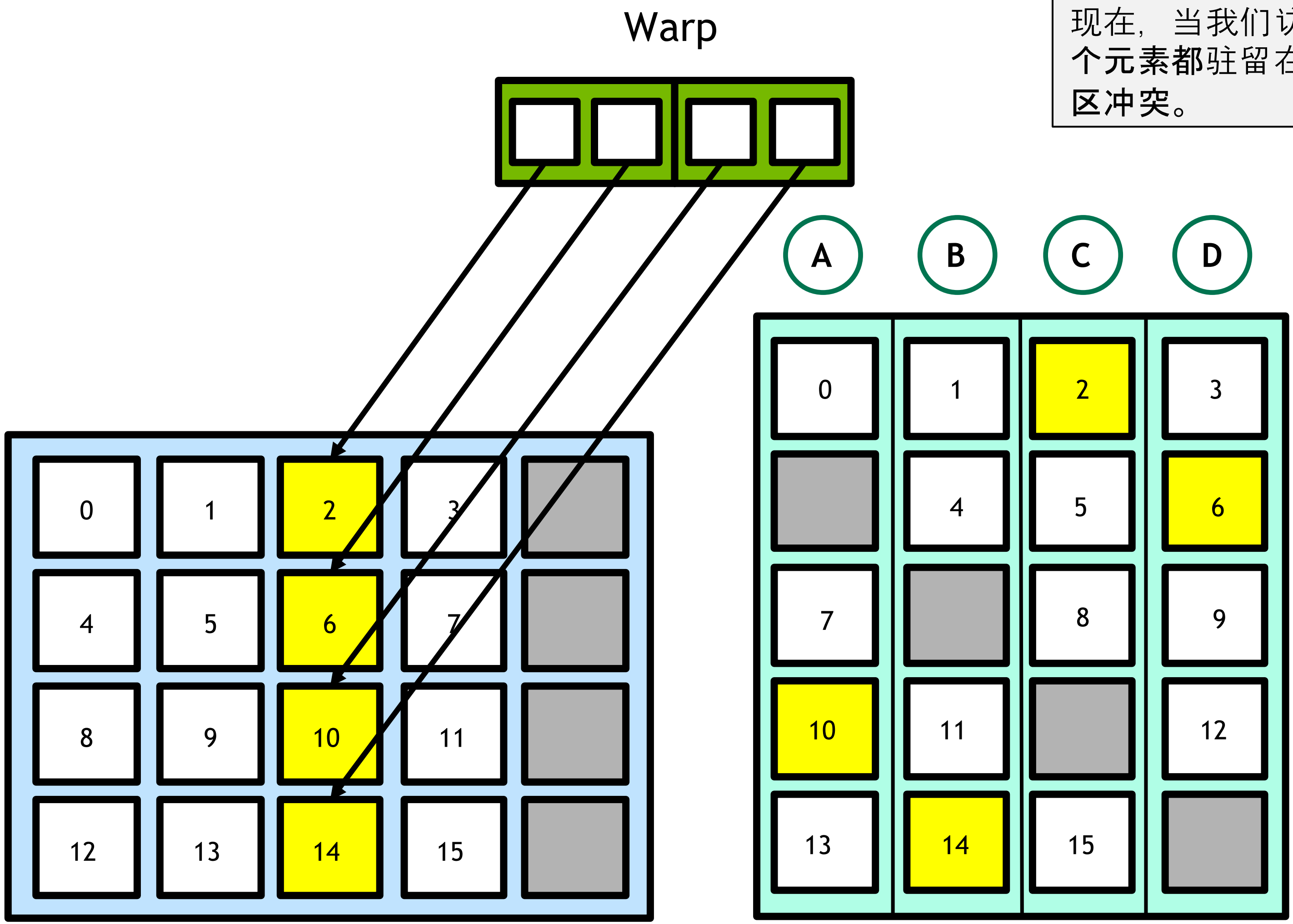
现在，当我们访问一列共享内存时，每个元素都驻留在不同的区中，并且没有区冲突。



逻辑共享内存
cuda.shared.array(4,5)

物理共享内存
分为 4 个区

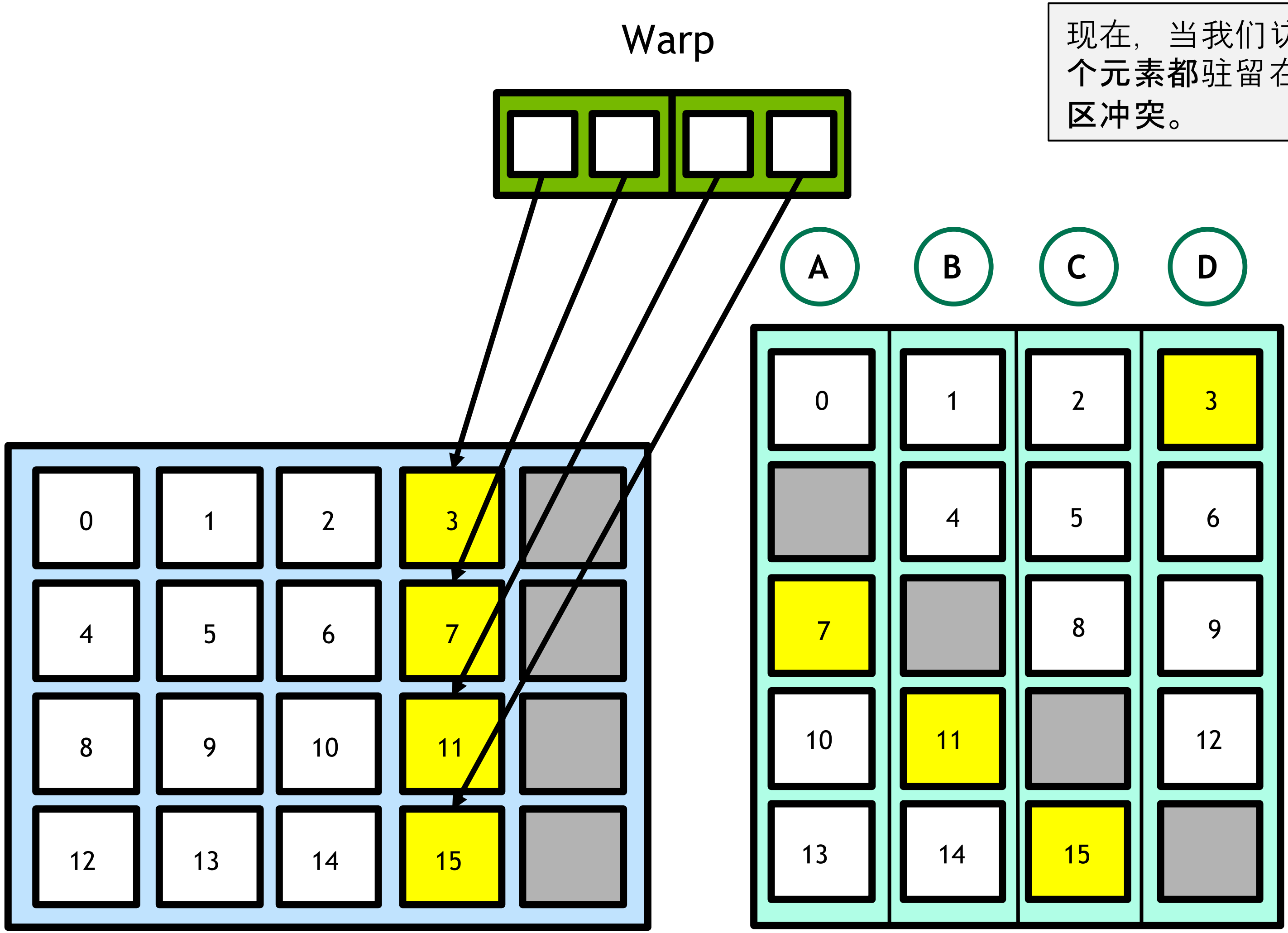
现在，当我们访问一列共享内存时，每个元素都驻留在不同的区中，并且没有区冲突。



逻辑共享内存
cuda.shared.array(4,5)

物理共享内存
分为 4 个区

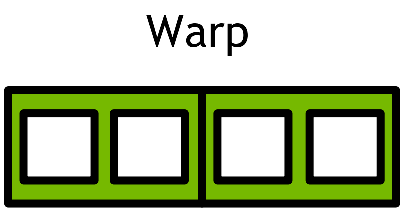
现在，当我们访问一列共享内存时，每个元素都驻留在不同的区中，并且没有区冲突。

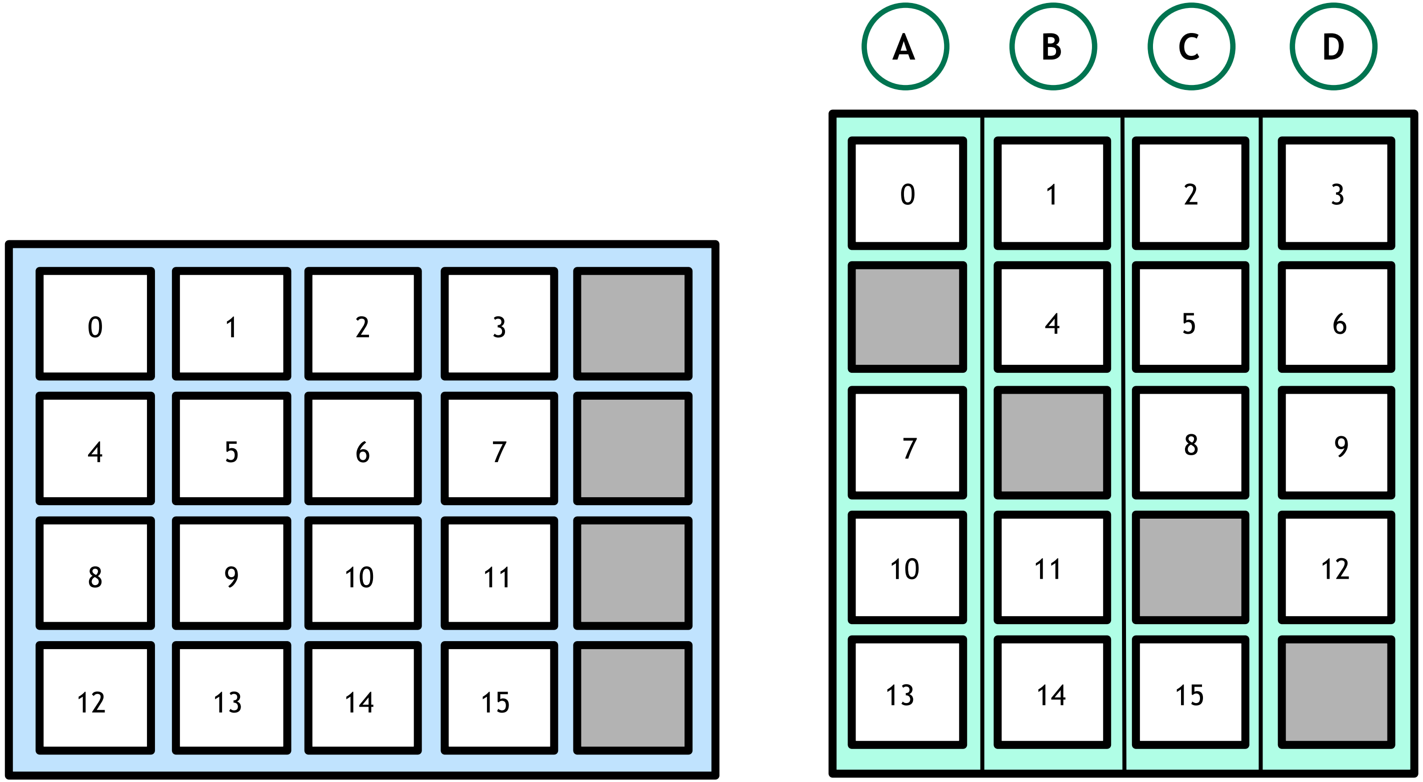


逻辑共享内存
cuda.shared.array(4,5)

物理共享内存
分为 4 个区

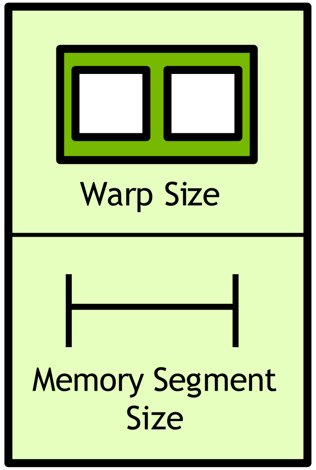
值得一提的是，要在此示例中使用此技术，我们必须对代码进行的唯一更改是**在共享内存分配中添加一个**额外的列。

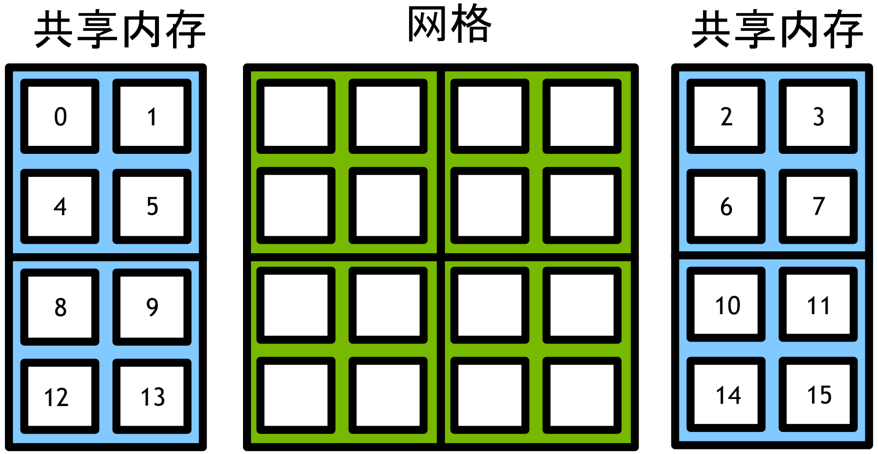




逻辑共享内存
cuda.shared.array(4,5)

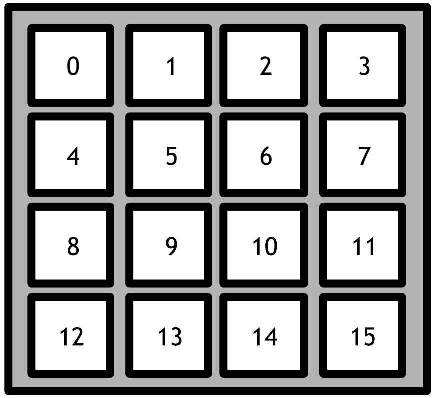
物理共享内存
分为 4 个区



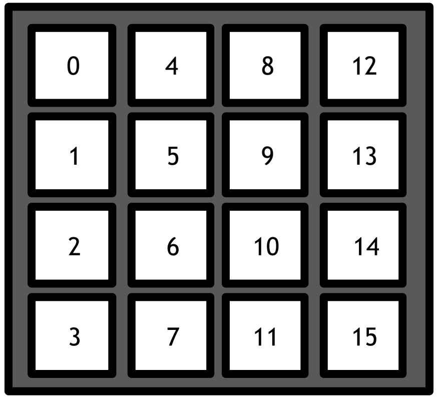


从我们之前的矩阵转置示例中，下面绿色的单个更改足以在保持正确性的同时避免区冲突。

```
tile = cuda.shared.array(2,3)
x, y = cuda.grid(2)

tile[tIdx.y][tIdx.x] = in[y][x]
cuda.syncthreads()

o_x = bId.y*bDim.y + tId.x
o_y = bId.x*bDim.x + tId.y
o[o_y][o_x] = tile[tIdx.x][tIdx.y]
```



输入



输出

NVIDIA

DEEP LEARNING INSTITUTE

学习更多课程，请访问 www.nvidia.cn/DLI