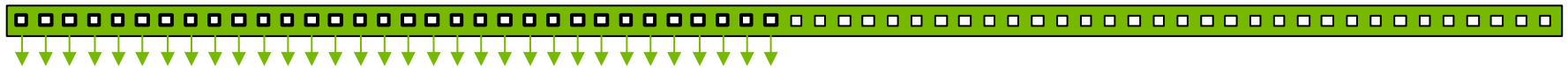
内存合并

回想一下，线程块被划分为 32 个线程的 Warp。

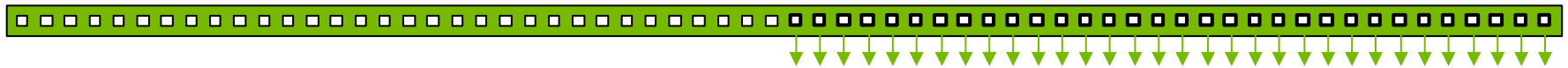
回想一下，线程块被划分为 32 个线程的 Warp。

回想一下，线程块被划分为 32 个线程的 Warp。

指令在 32 个线程的 warp 级别并行发出

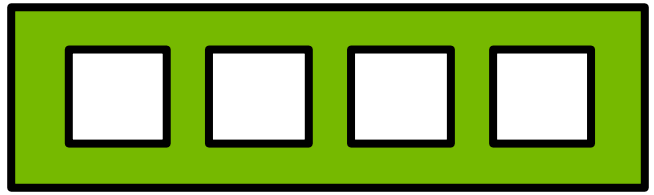


指令在 32 个线程的 warp 级别并行发出

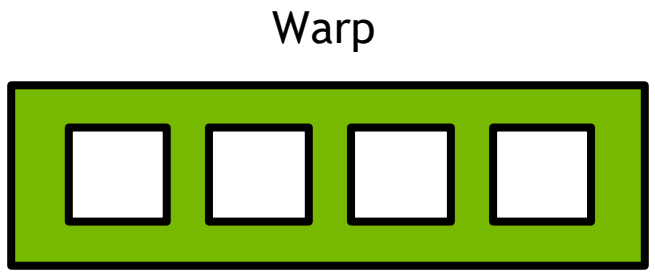


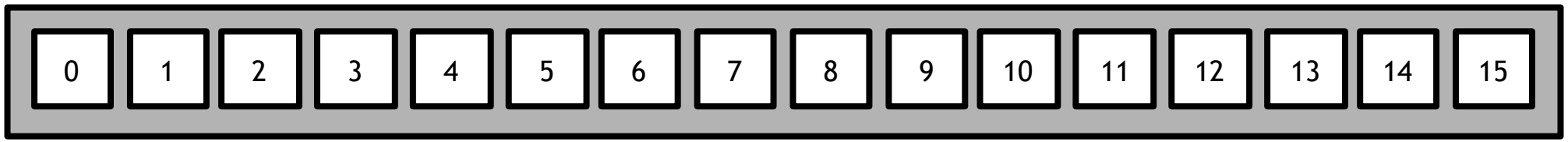
为节省每一页上的空间，我们仅将 4 个线程视为一个 Warp

Warp



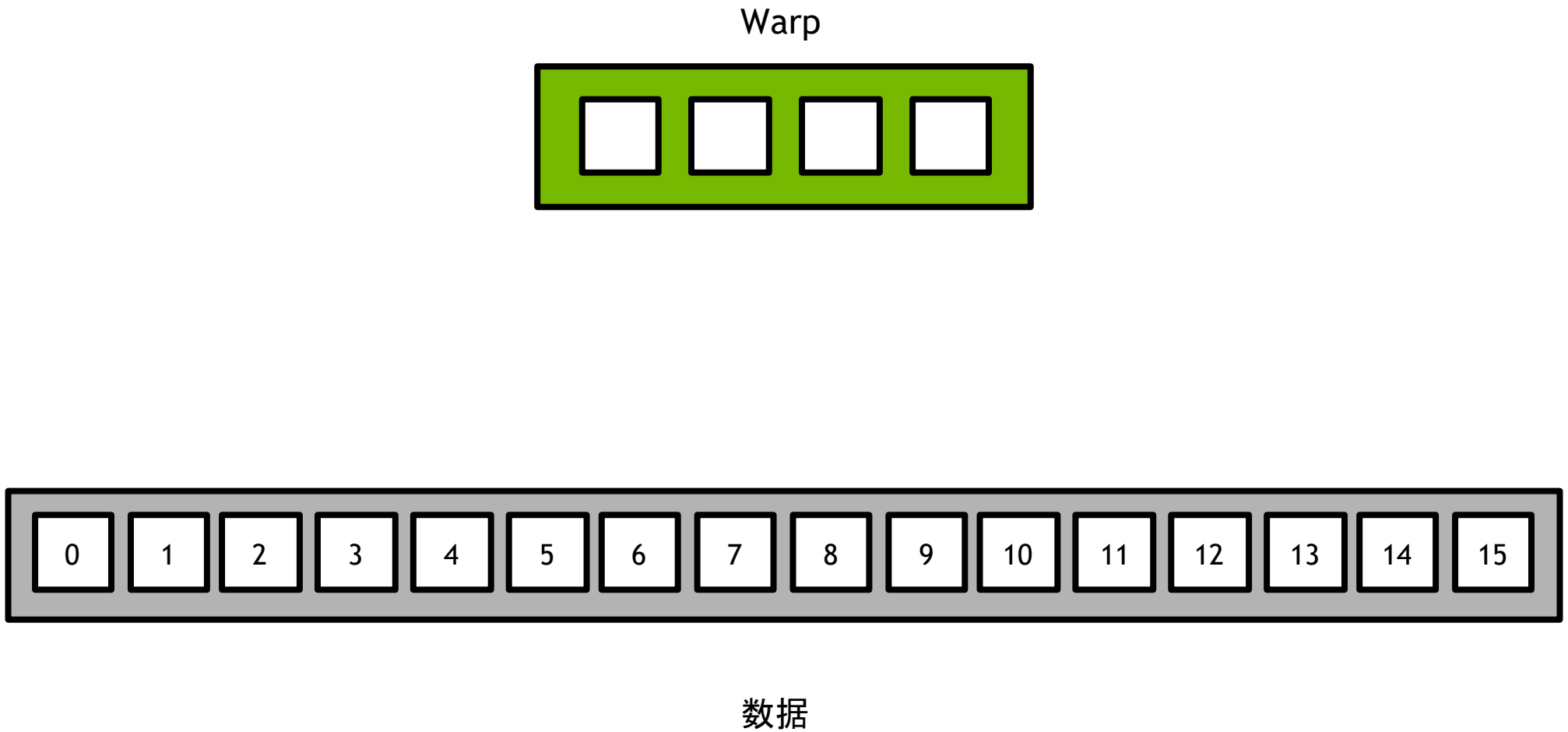
数据以 32 字节段的形式传入和传出全局设备内存 *



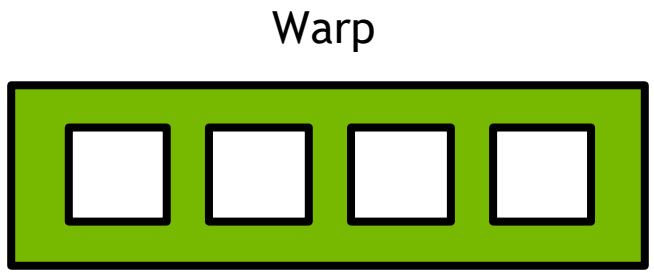


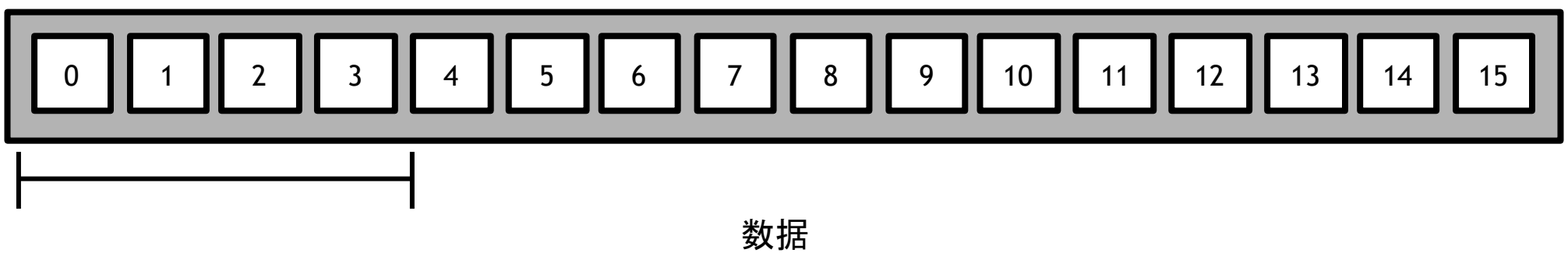
数据

(* 如果数据在 L1 缓存中，它将在 128 字节缓存行中传输 - 有关详细信息，请参阅实验的 notebook)

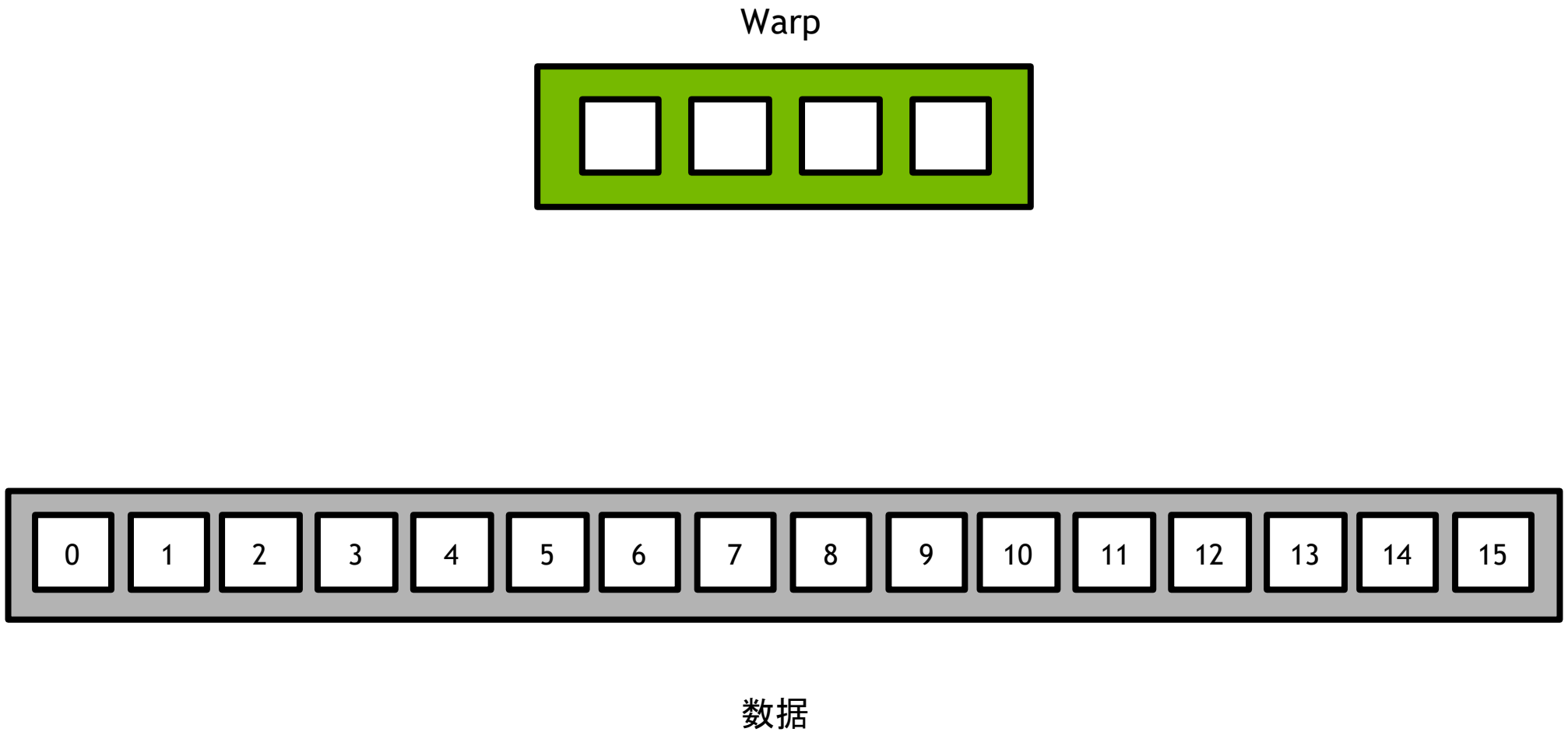


在这里的演示中，我们将 4 个数据元素视为连续内存中的固定长度的一行。

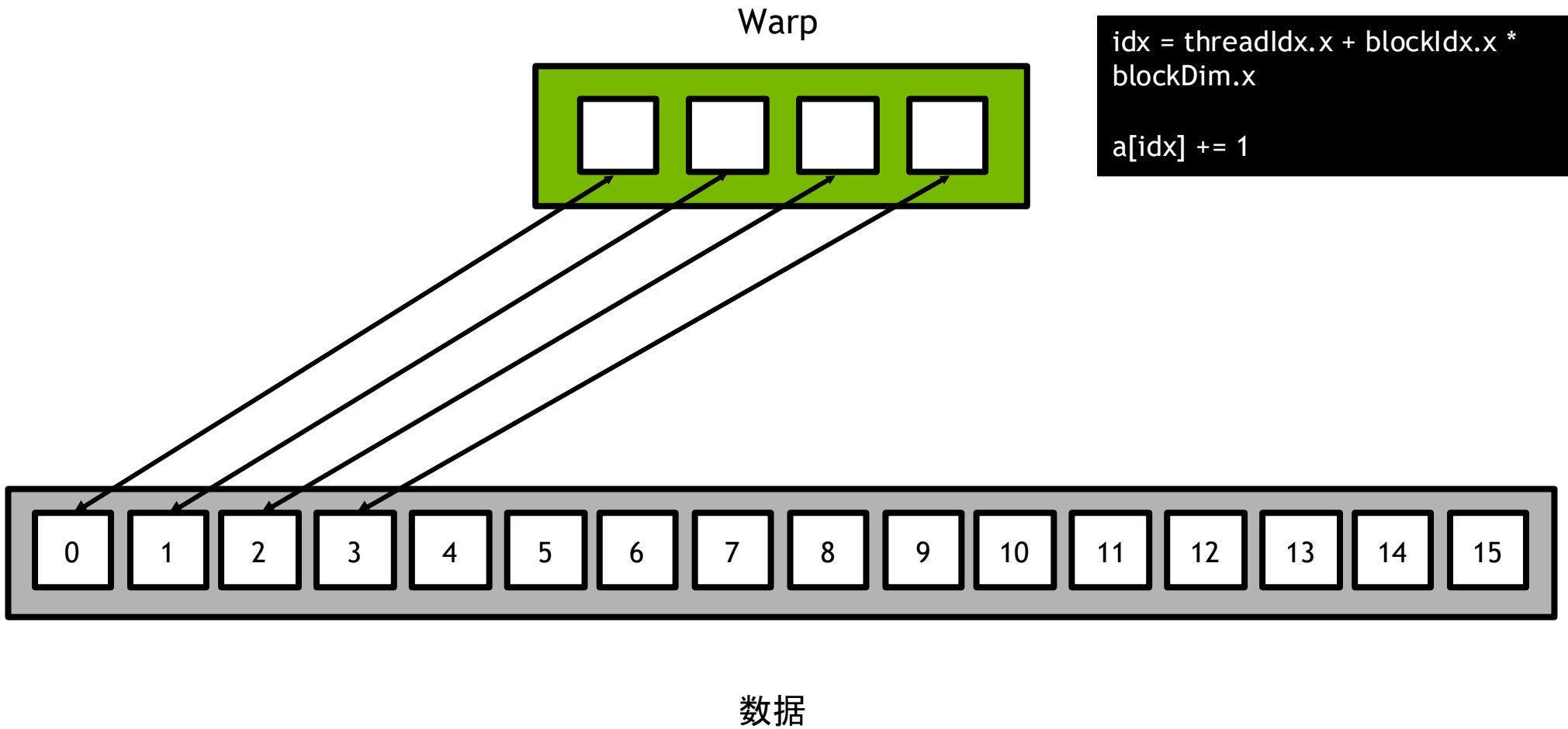




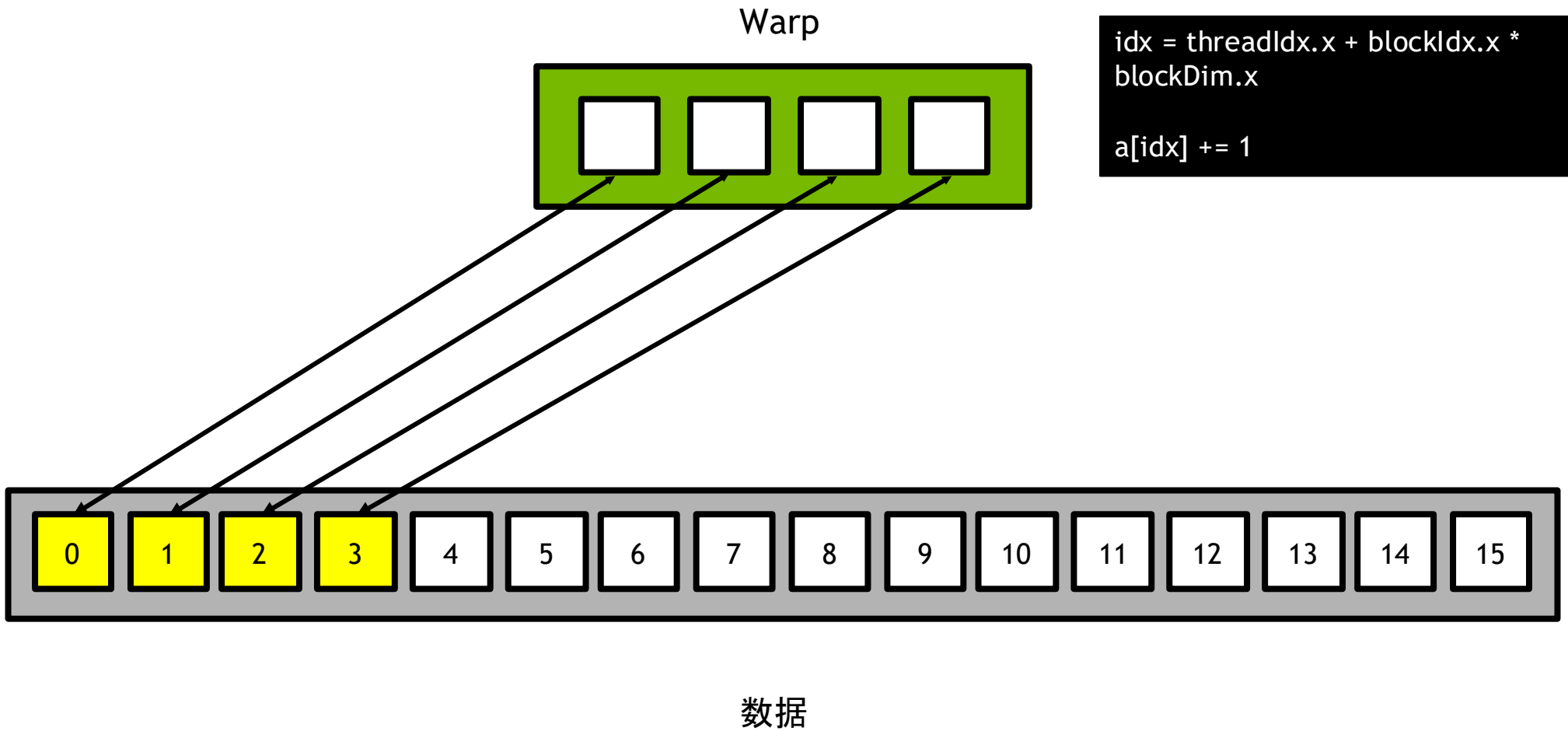
内存子系统将试图使满足 warp 的读/写要求所需的行数最小



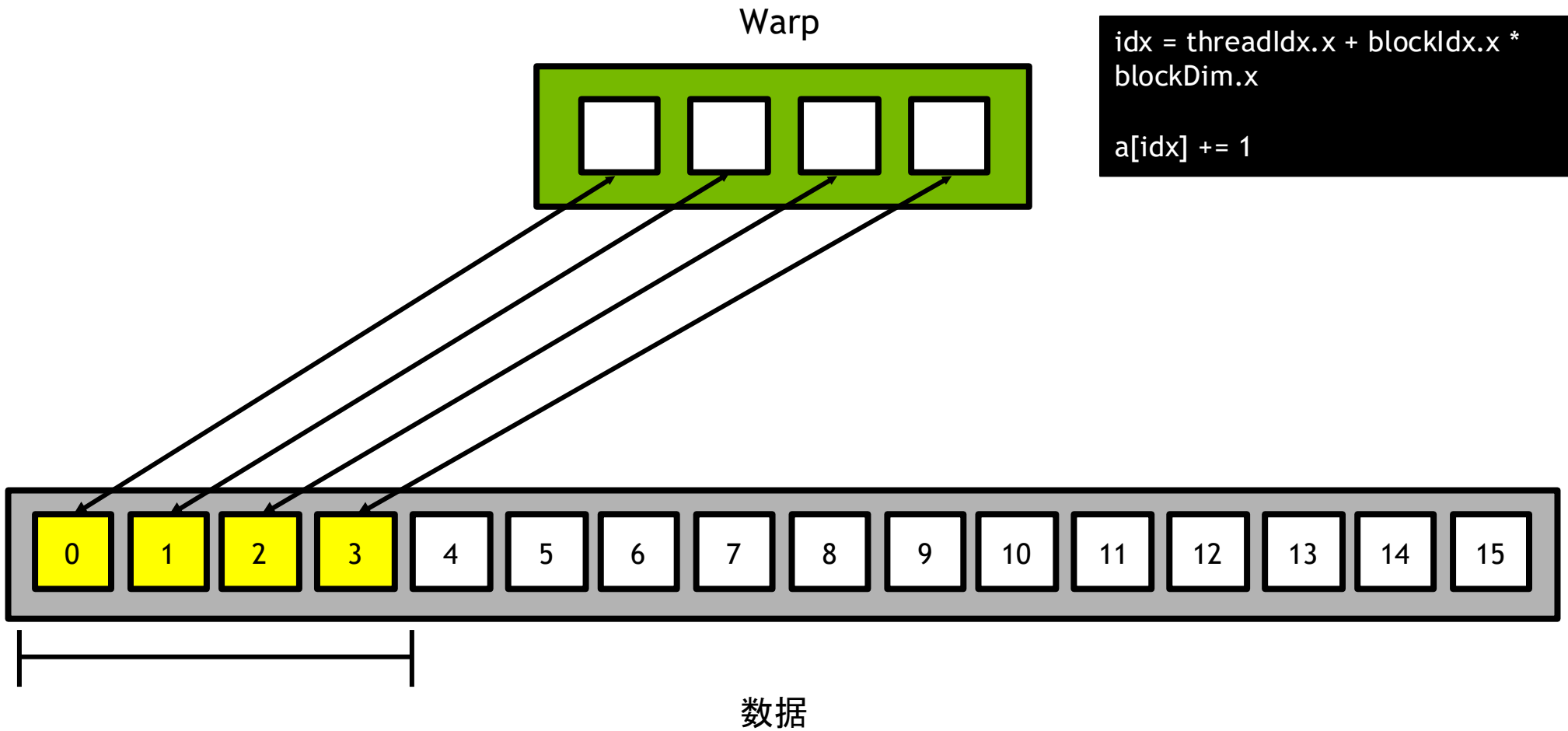
如果请求的地址是连续的



该行中的所有数据都将被使用

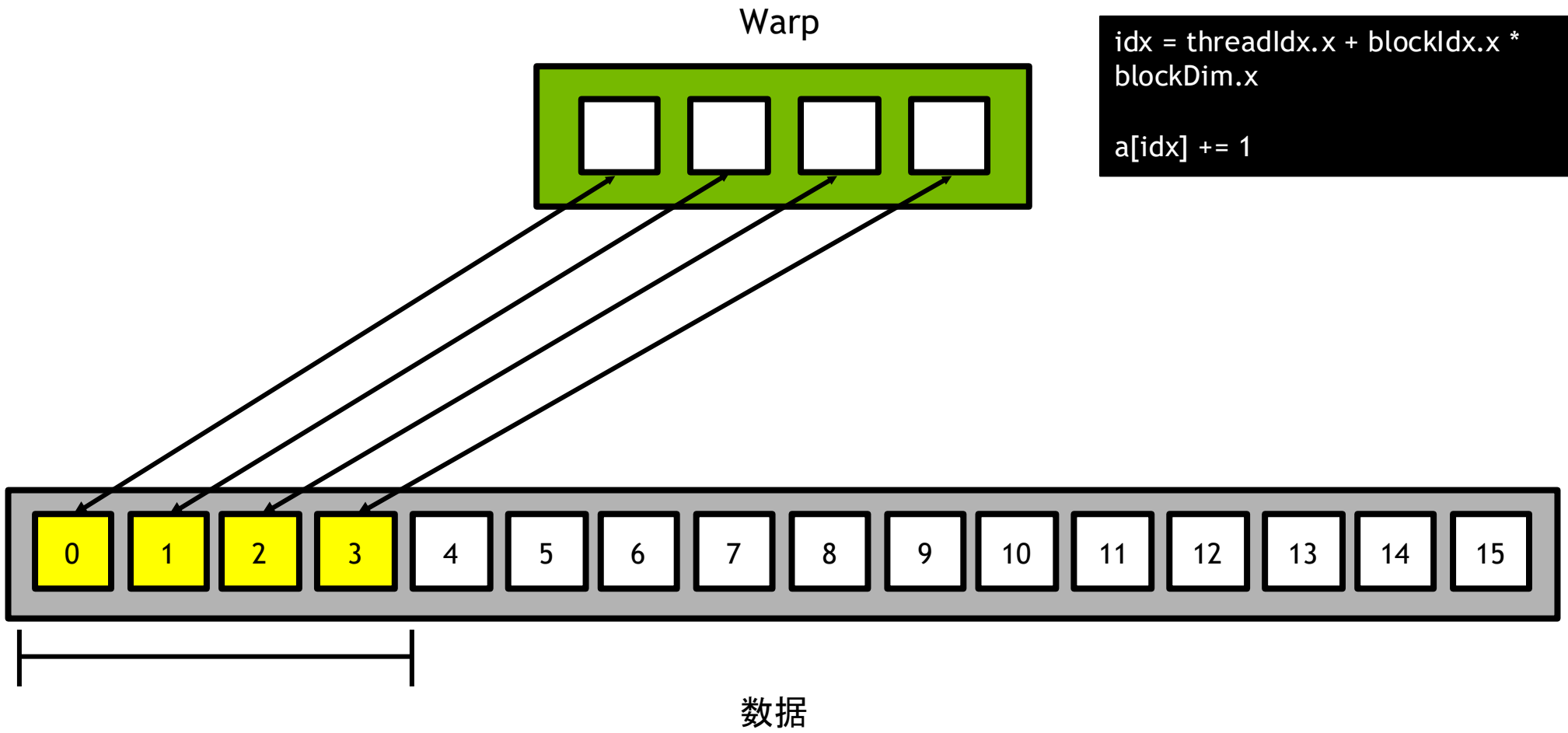


并且数据迁移将在尽可能少的行中发生



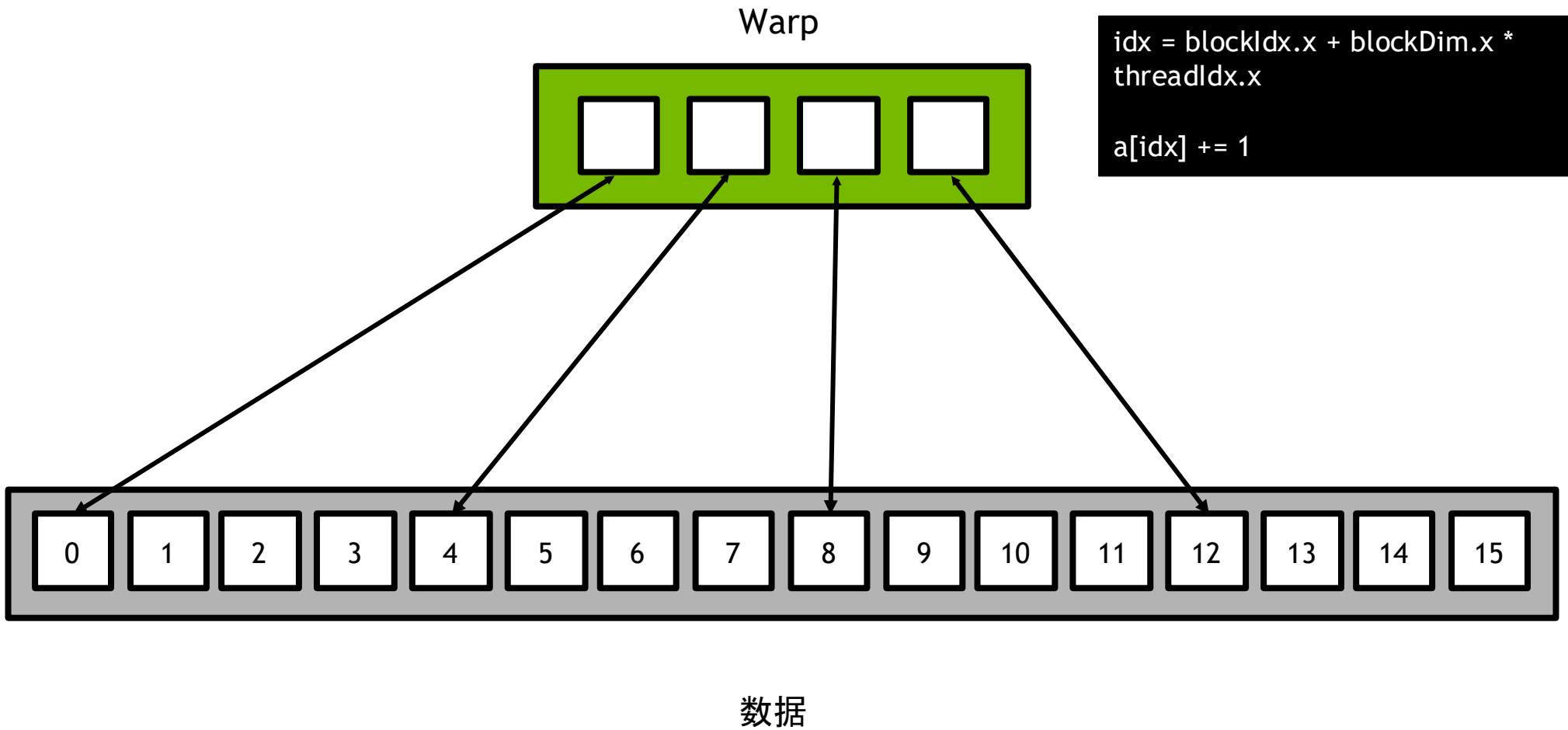
出现这种情况时，内存访问是完全合并的

```
idx = threadIdx.x + blockIdx.x *
blockDim.x

a[idx] += 1
```

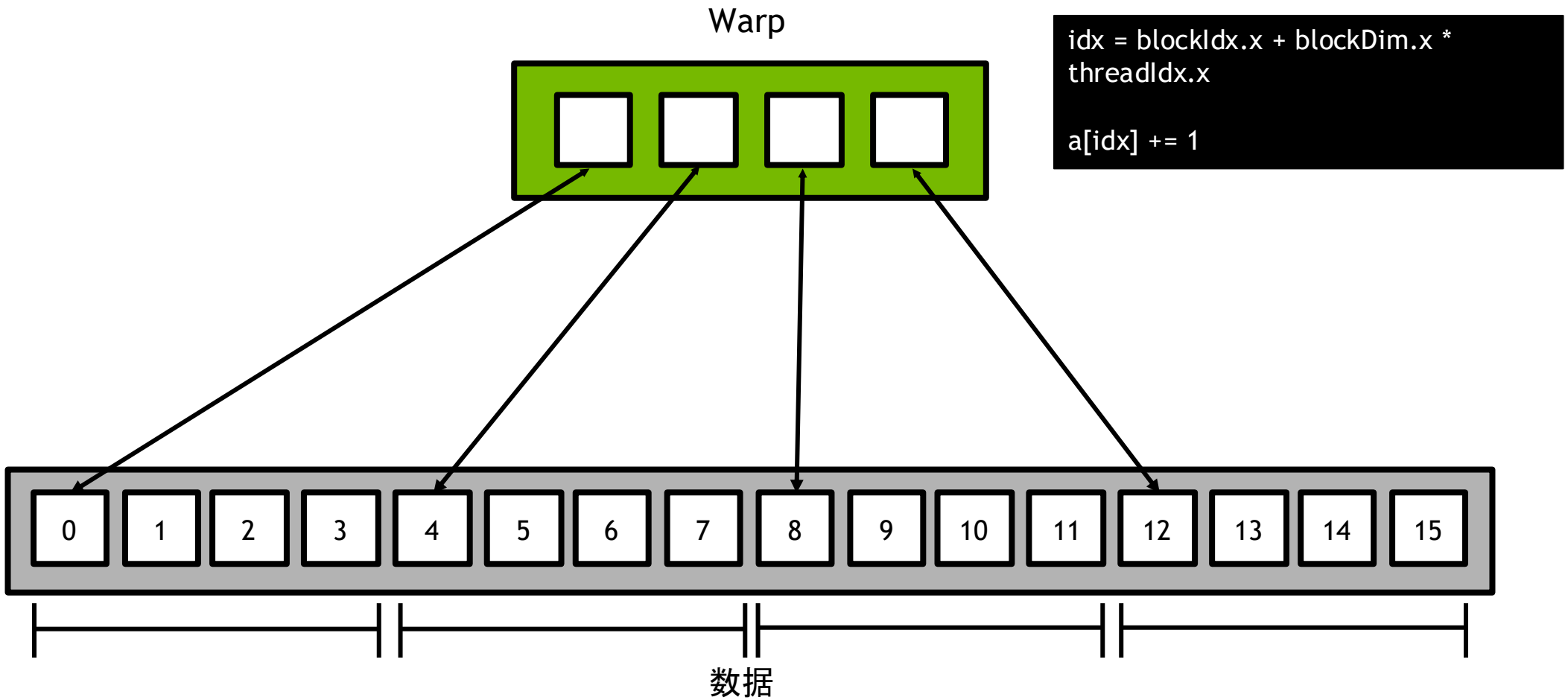


随着请求的内存变得不那么连续

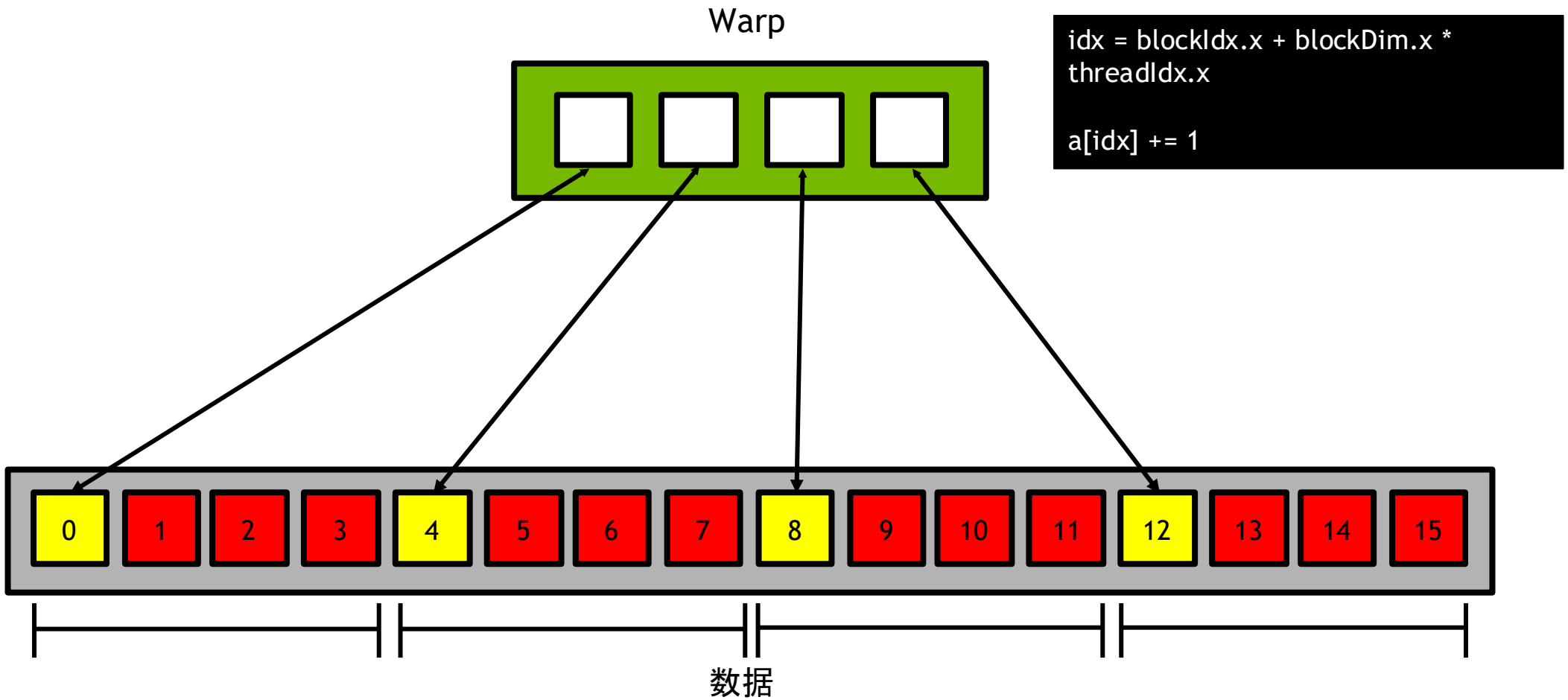
```
idx = blockIdx.x + blockDim.x *
threadIdx.x

a[idx] += 1
```



必须迁移更多的行以满足Warp的需要

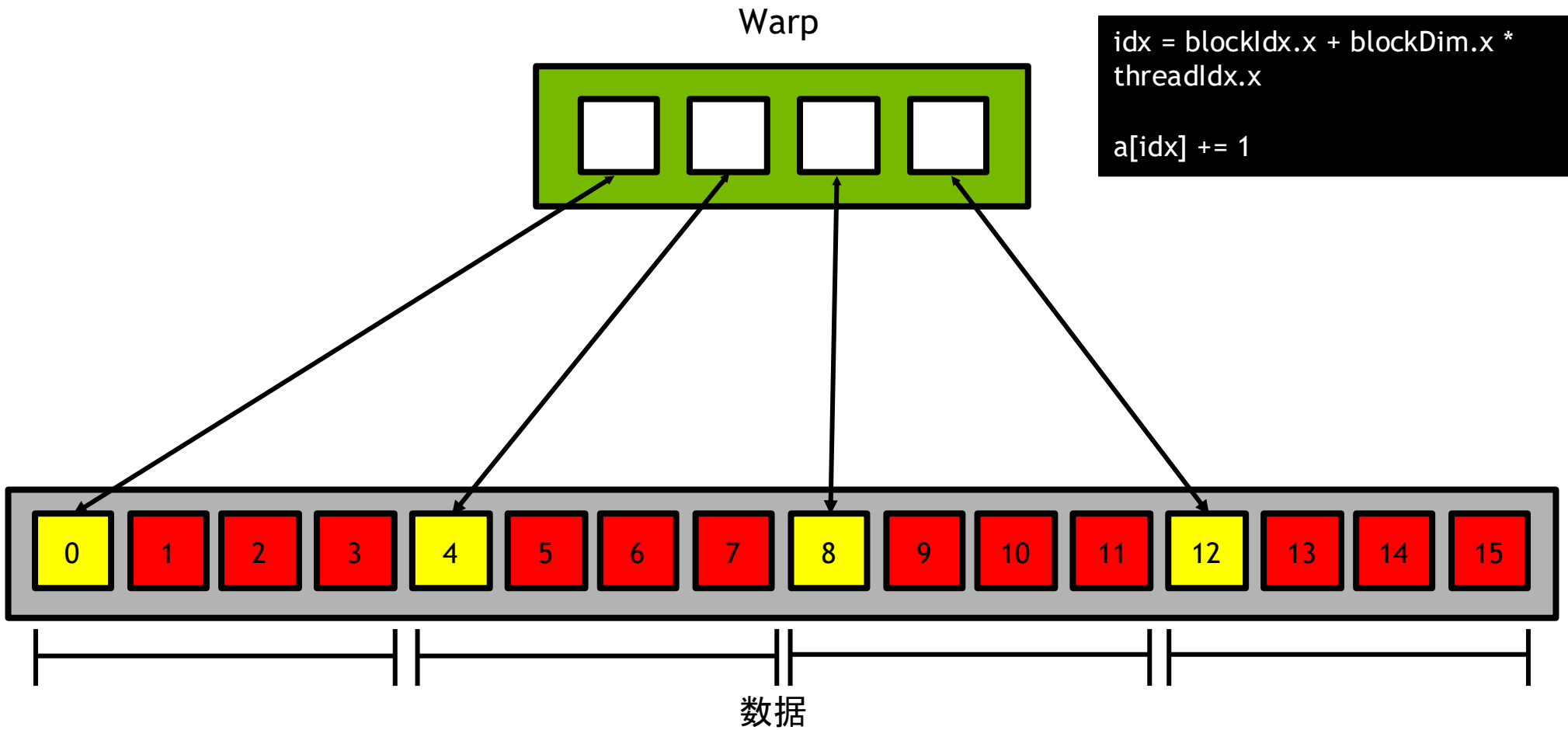


而且更多被传输的数据将被闲置



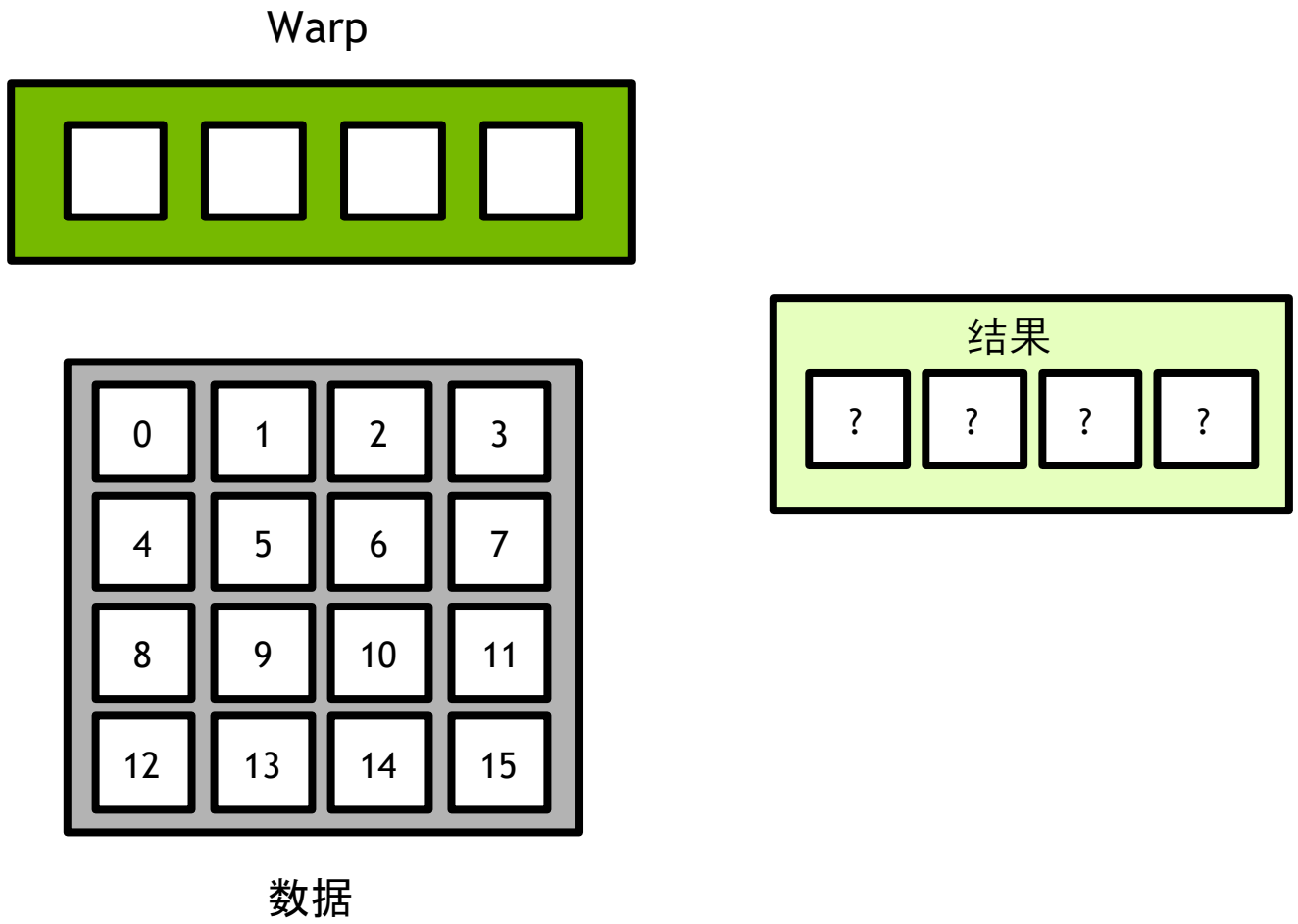
内存吞吐量下降，传输需要额外的时间：导致性能损失

```
idx = blockIdx.x + blockDim.x * threadIdx.x

a[idx] += 1
```

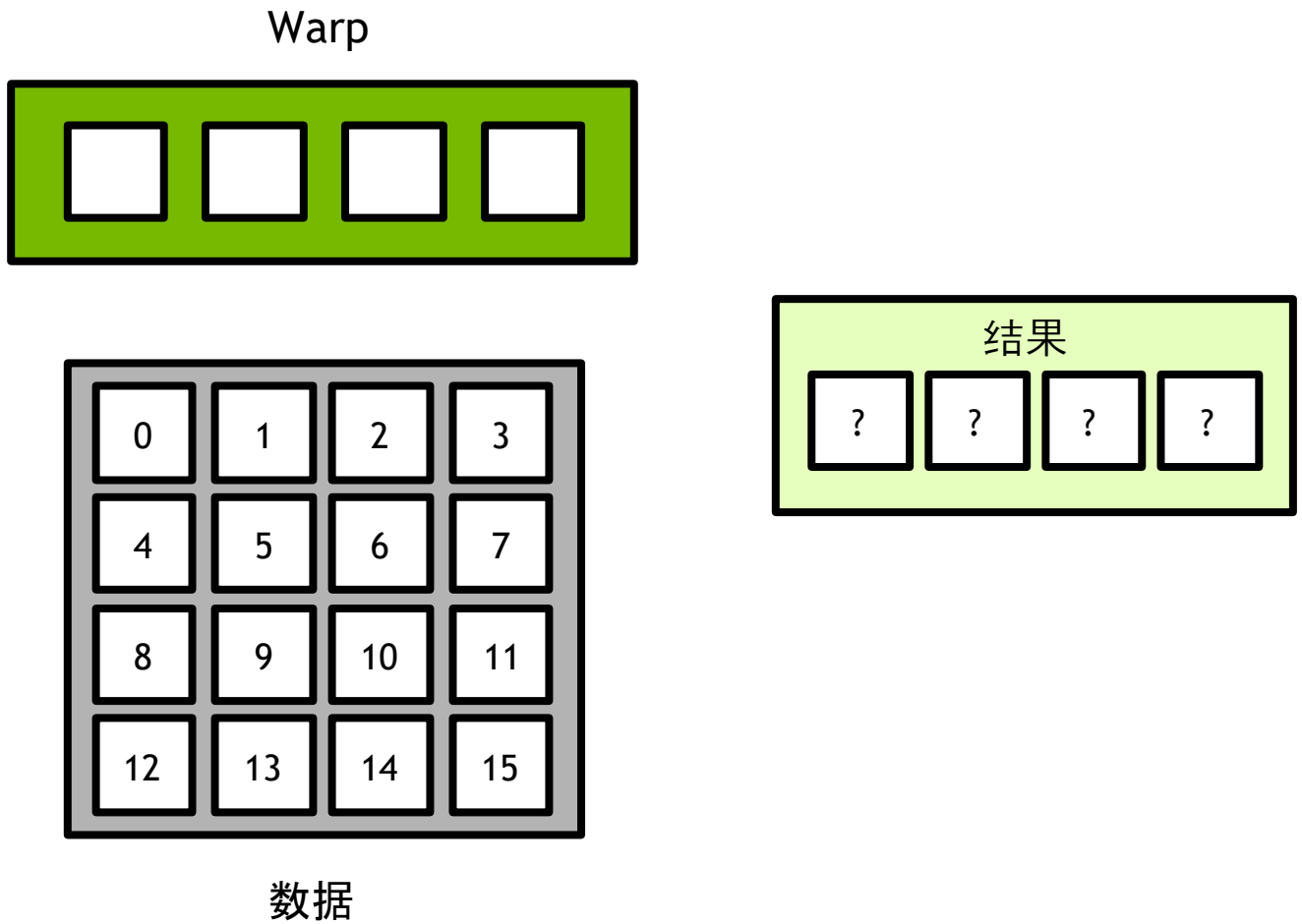


行之和与列之和的比较

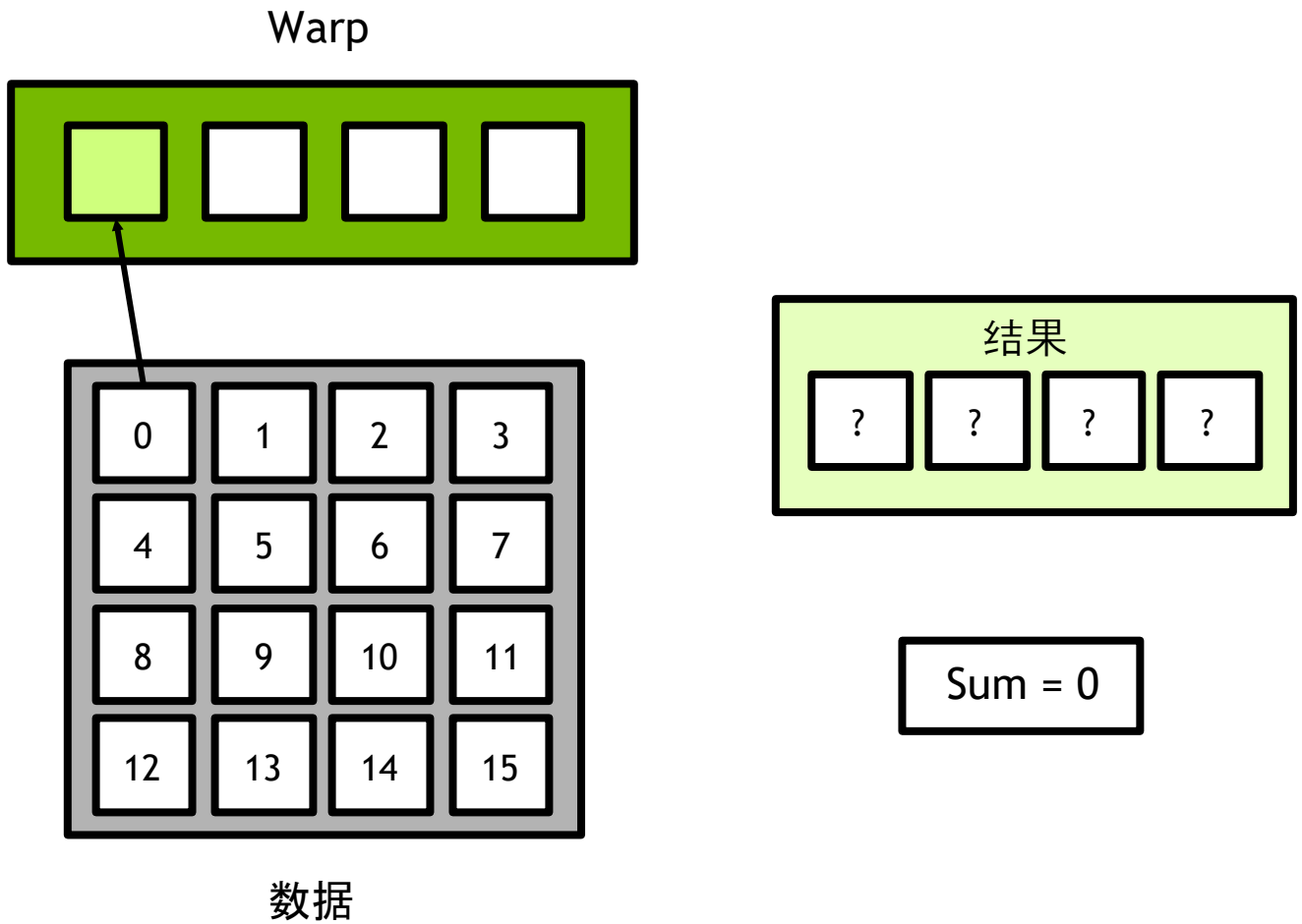
考虑一个核函数，它将矩阵的每一行（在这里是 4 个连续的数据元素）的和存储在一个结果向量中



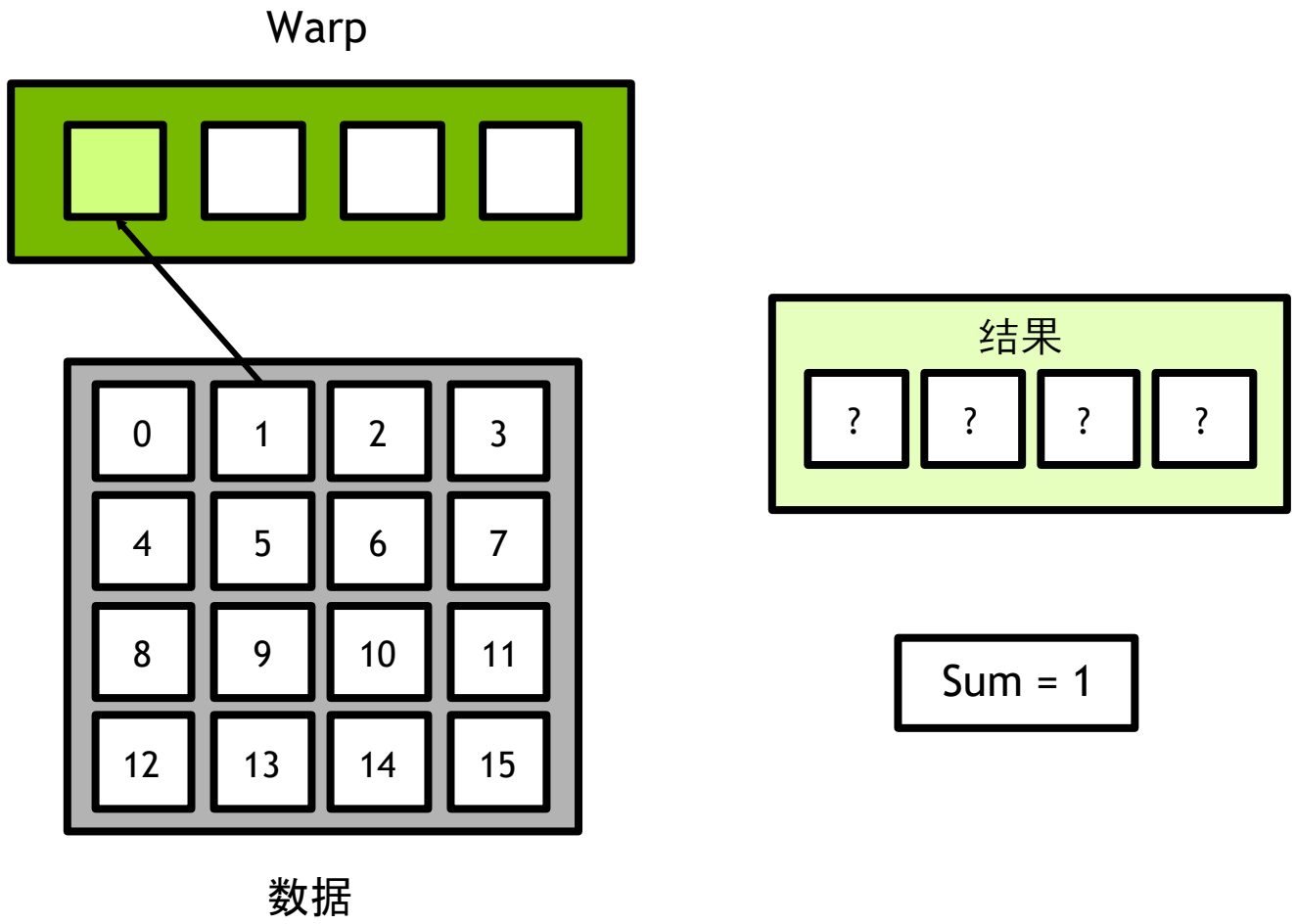
单个线程可以迭代一行，将其求和，然后将结果写入结果向量中。



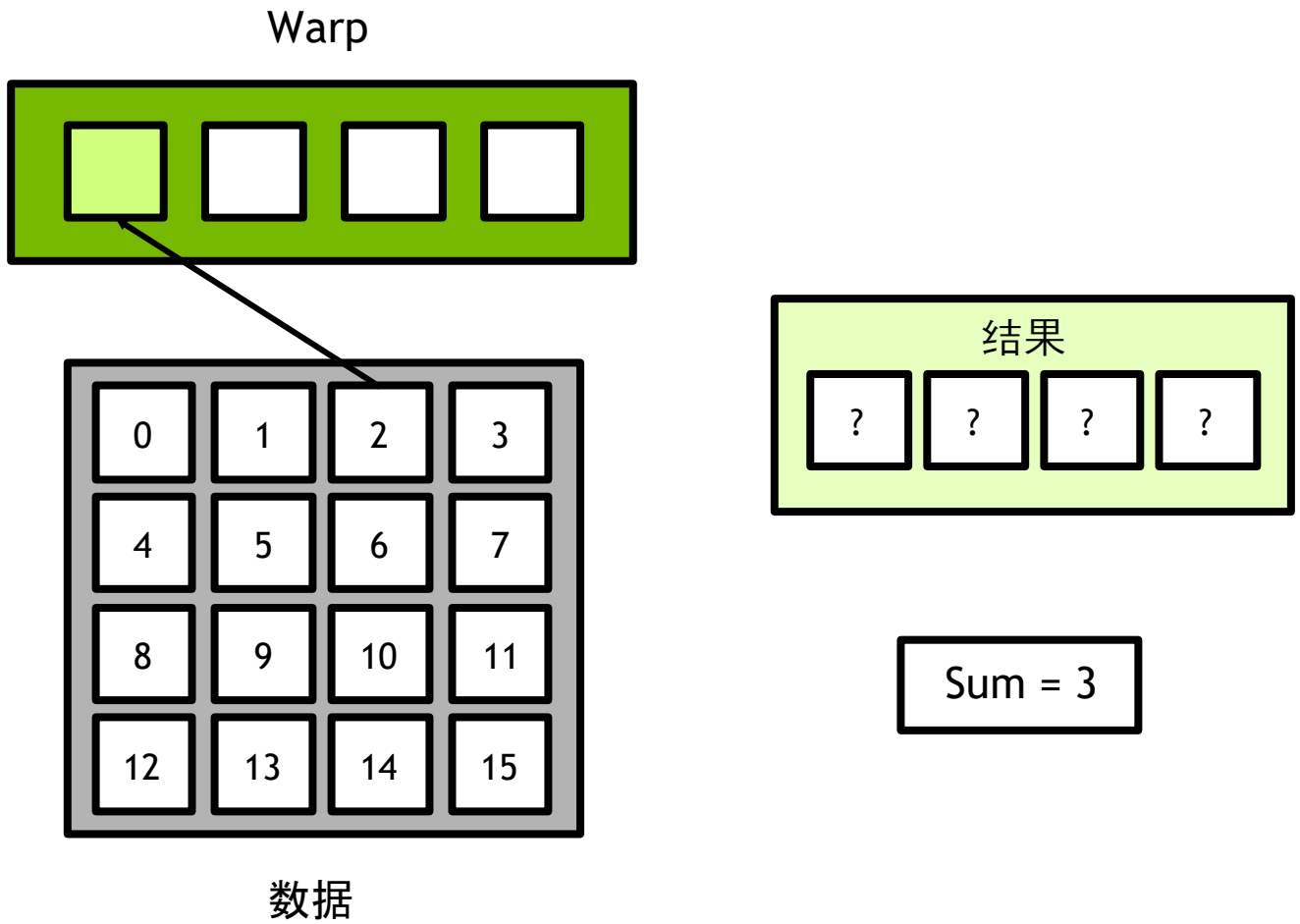
单个线程可以迭代一行，将其求和，然后将结果写入结果向量中。



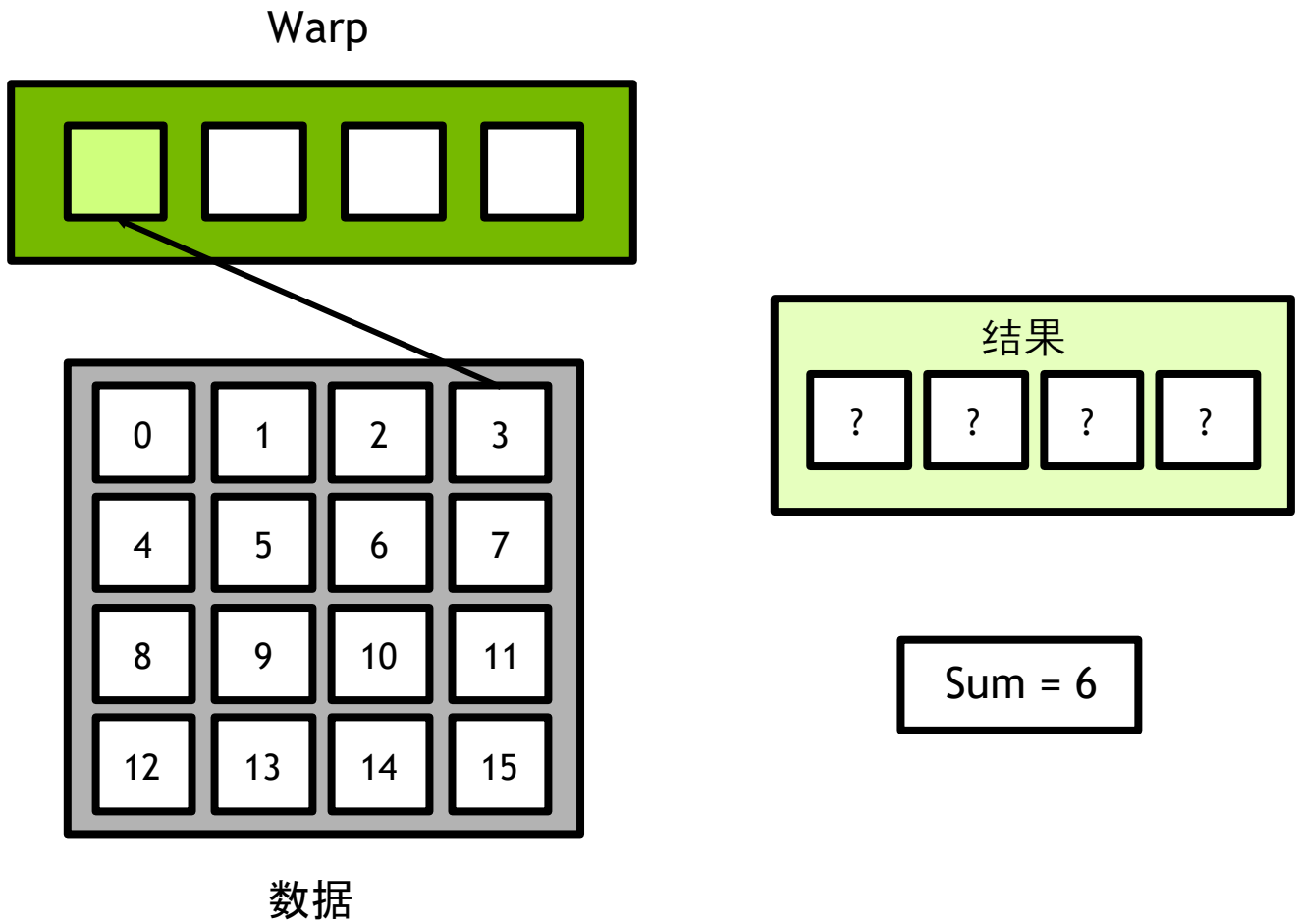
单个线程可以迭代一行，将其求和，然后将结果写入结果向量中。



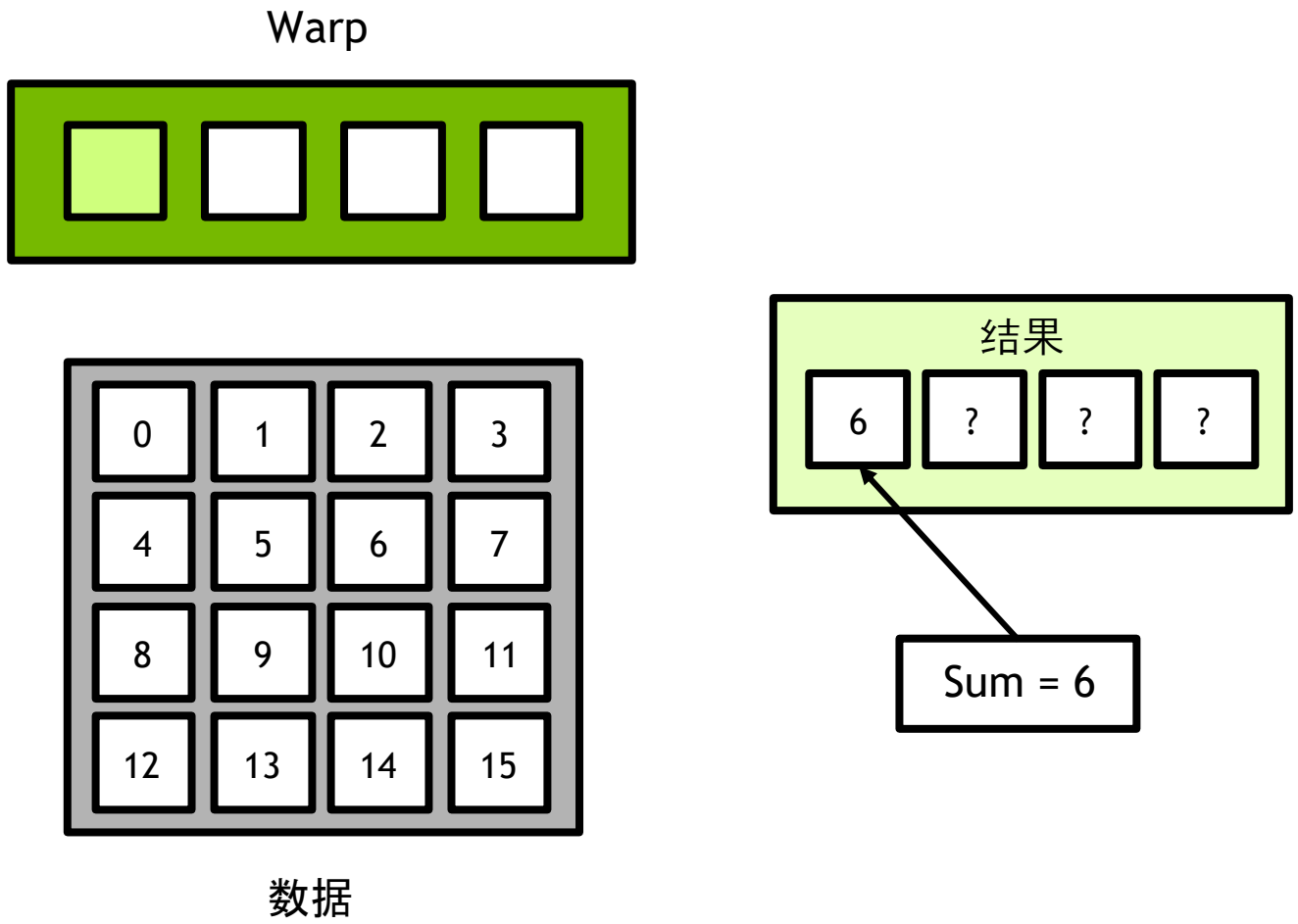
单个线程可以迭代一行，将其求和，然后将结果写入结果向量中。



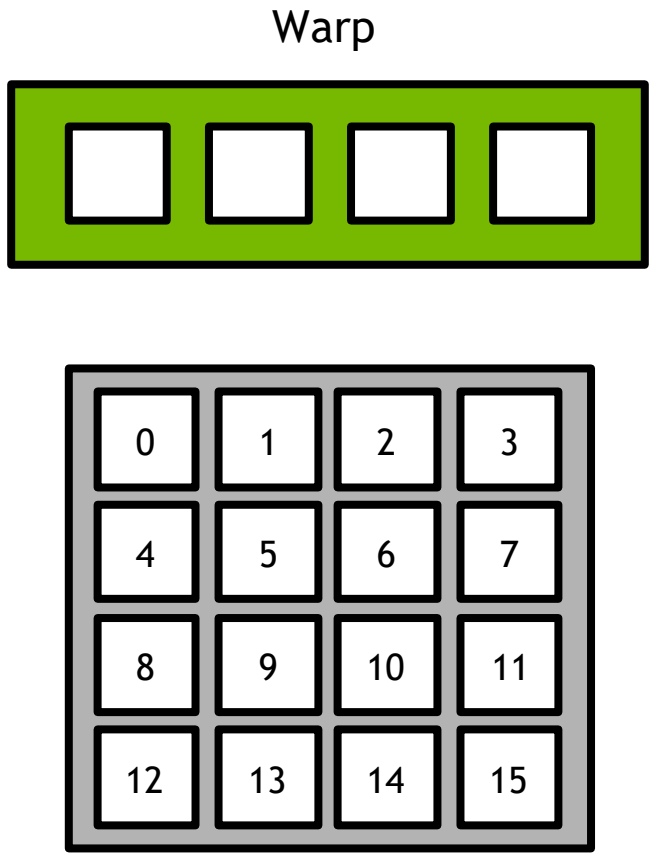
单个线程可以迭代一行，将其求和，然后将结果写入结果向量中。



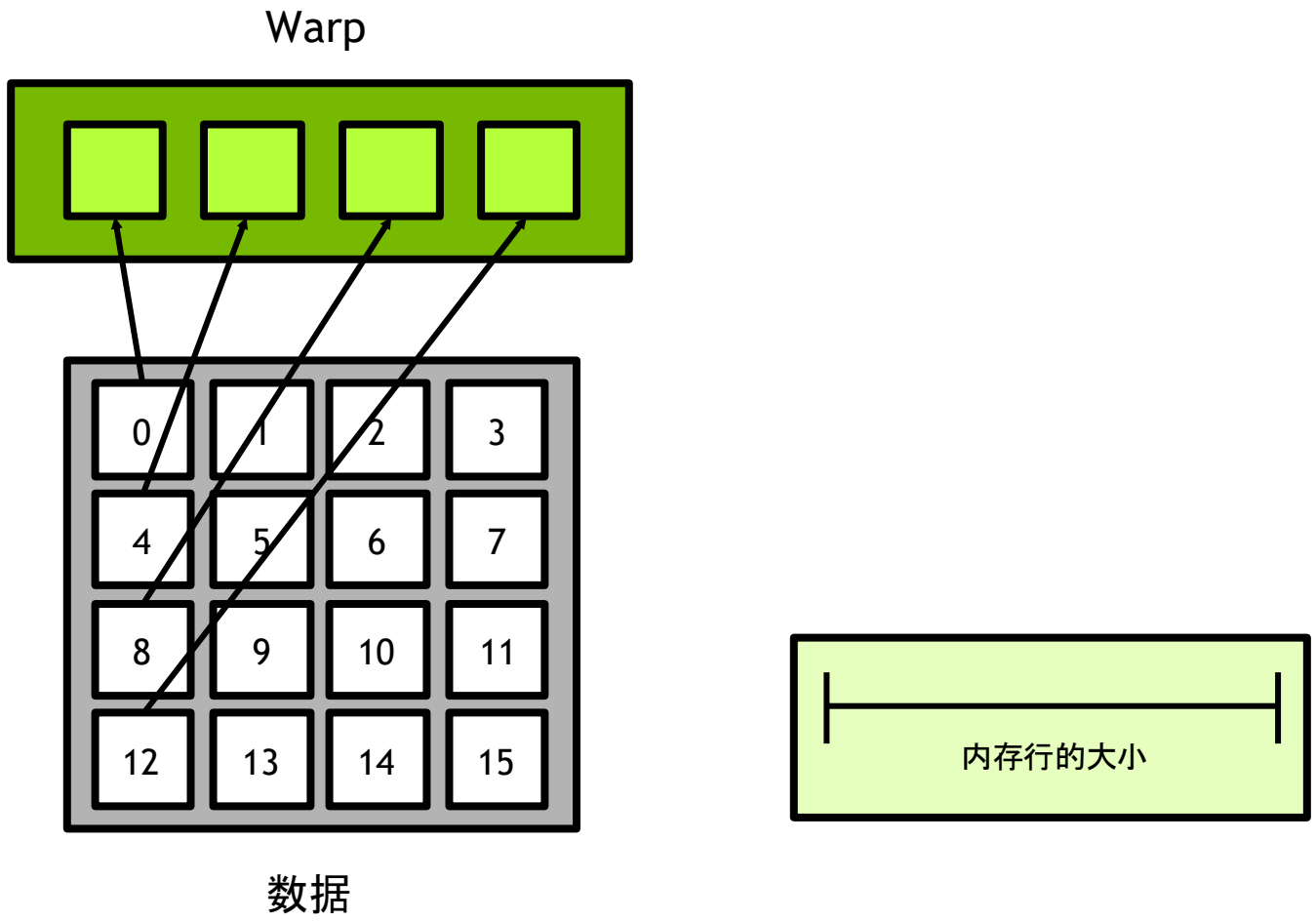
单个线程可以迭代一行，将其求和，然后将结果写入结果向量中。



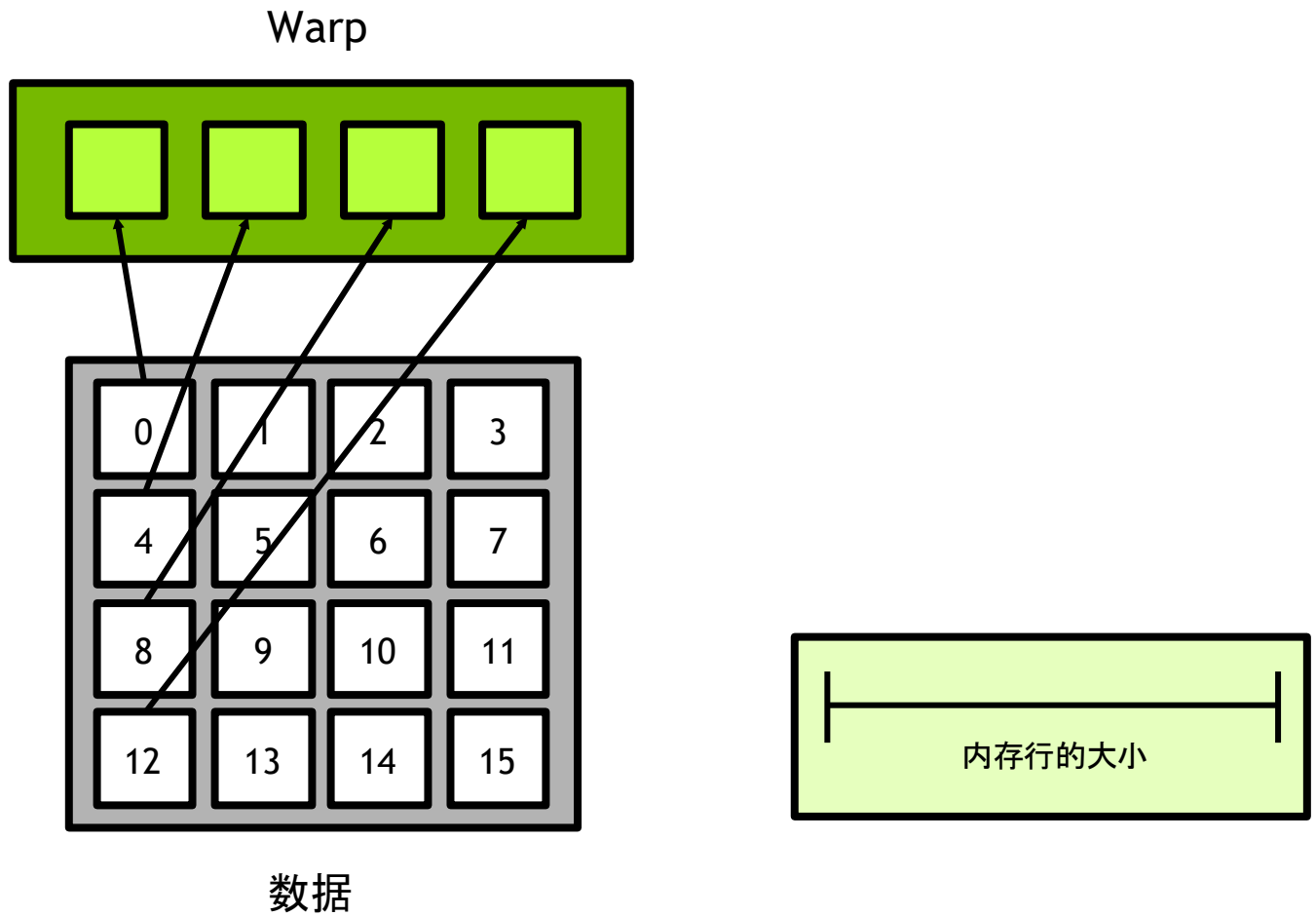
这看起来很自然，但是当我们考虑
warp 中的线程并行执行时会发生什么



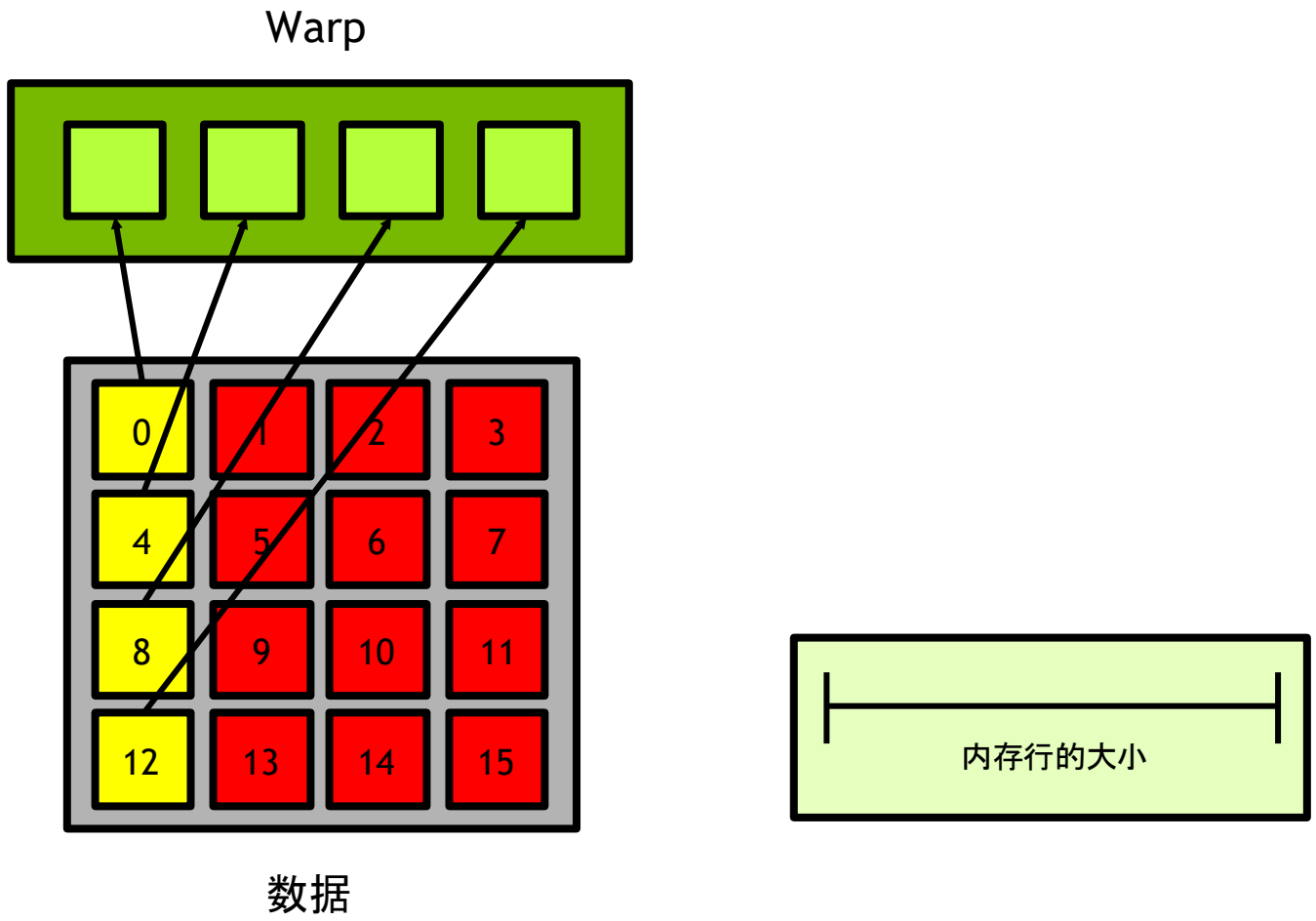
Warp中的每个线程都在不同的内存行中请求数据



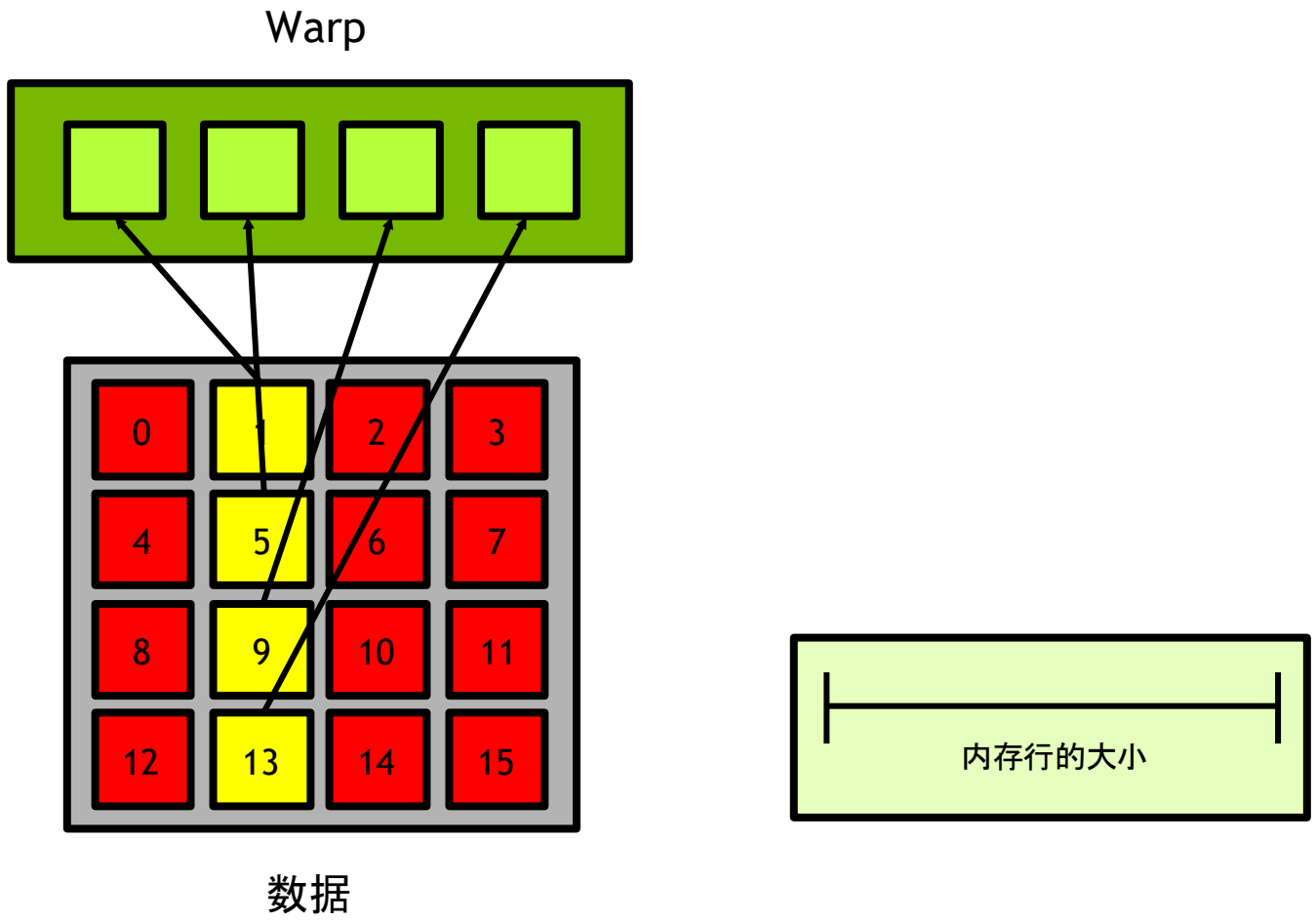
请注意， threadIdx.x 的增量映射到沿 y 轴的数据增量



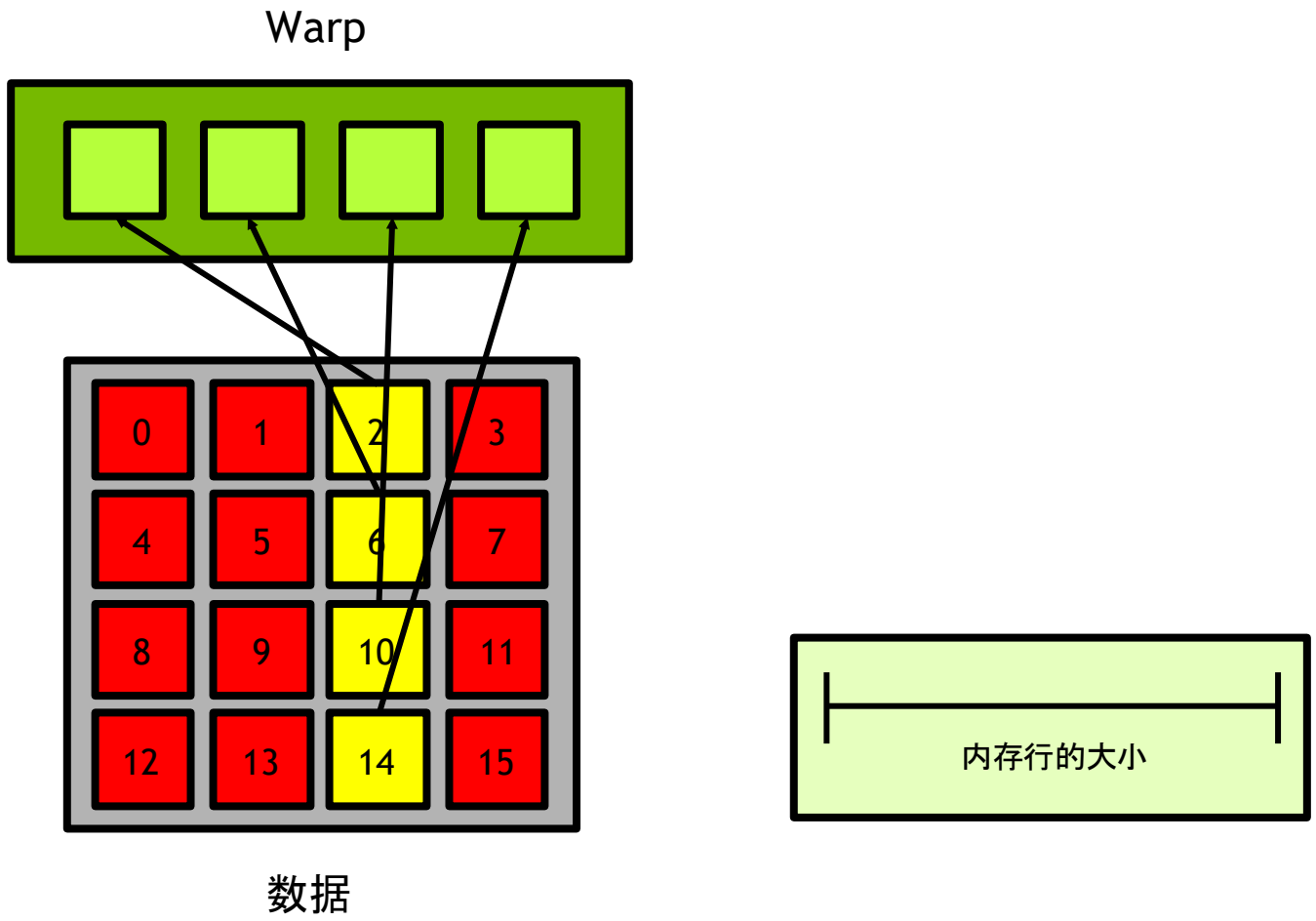
这意味着（在我们的示例中）需要加载 4 行数据，并且加载的数据中有 75% 未使用



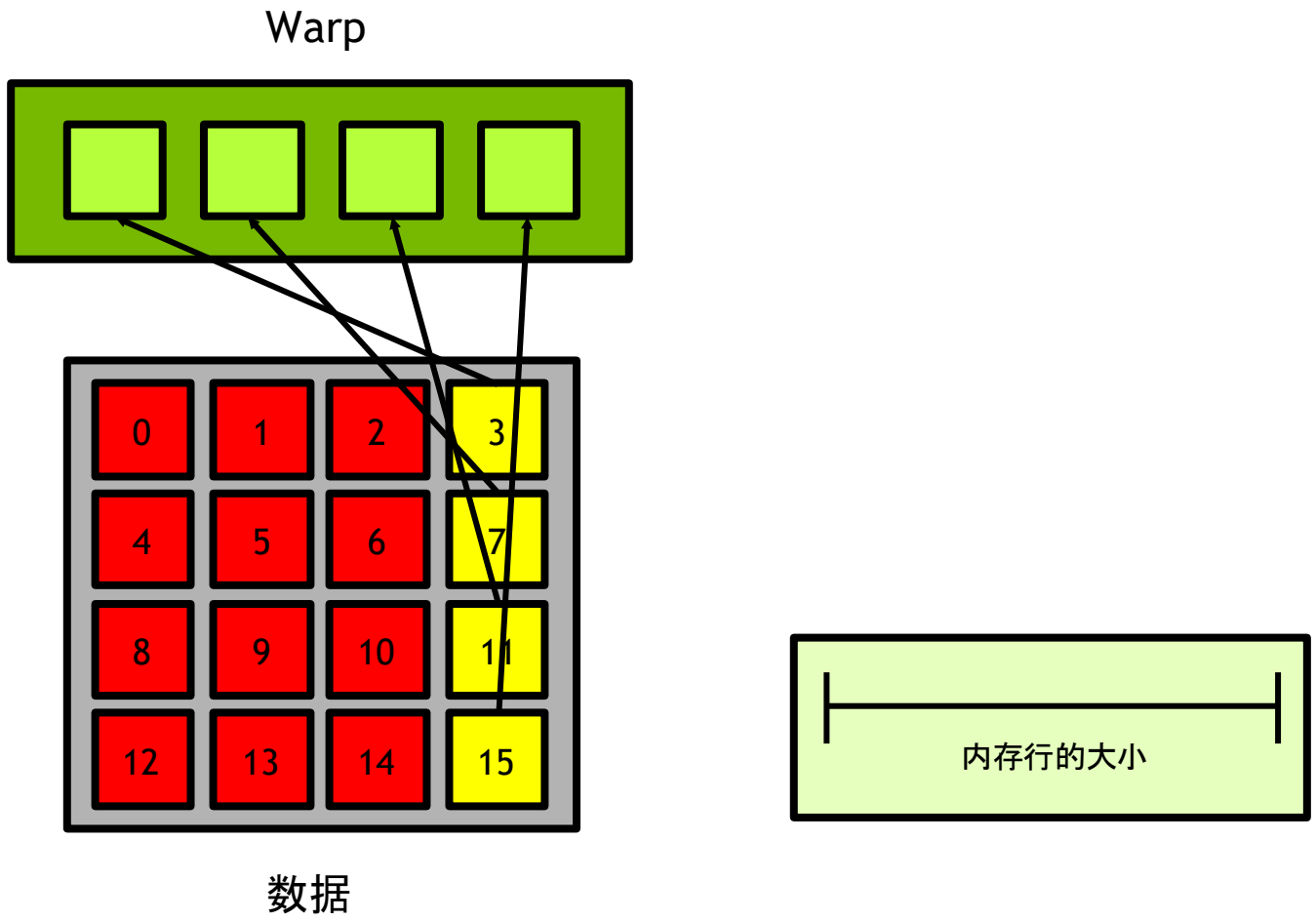
不幸的是，当每个线程在它所在的数据行上迭代时，相同的未合并模式仍在继续。



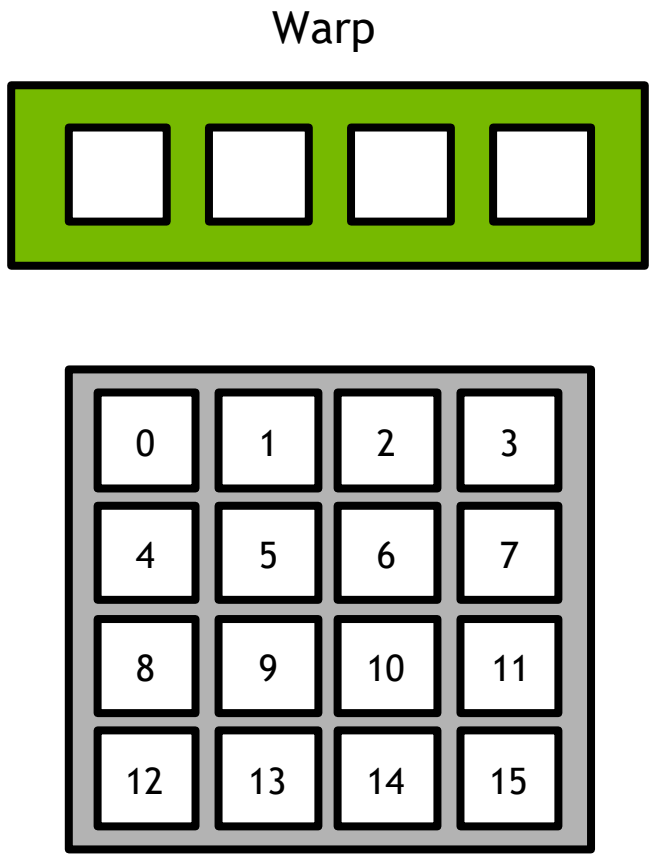
不幸的是，当每个线程在它所在的数据行上迭代时，相同的未合并模式仍在继续。



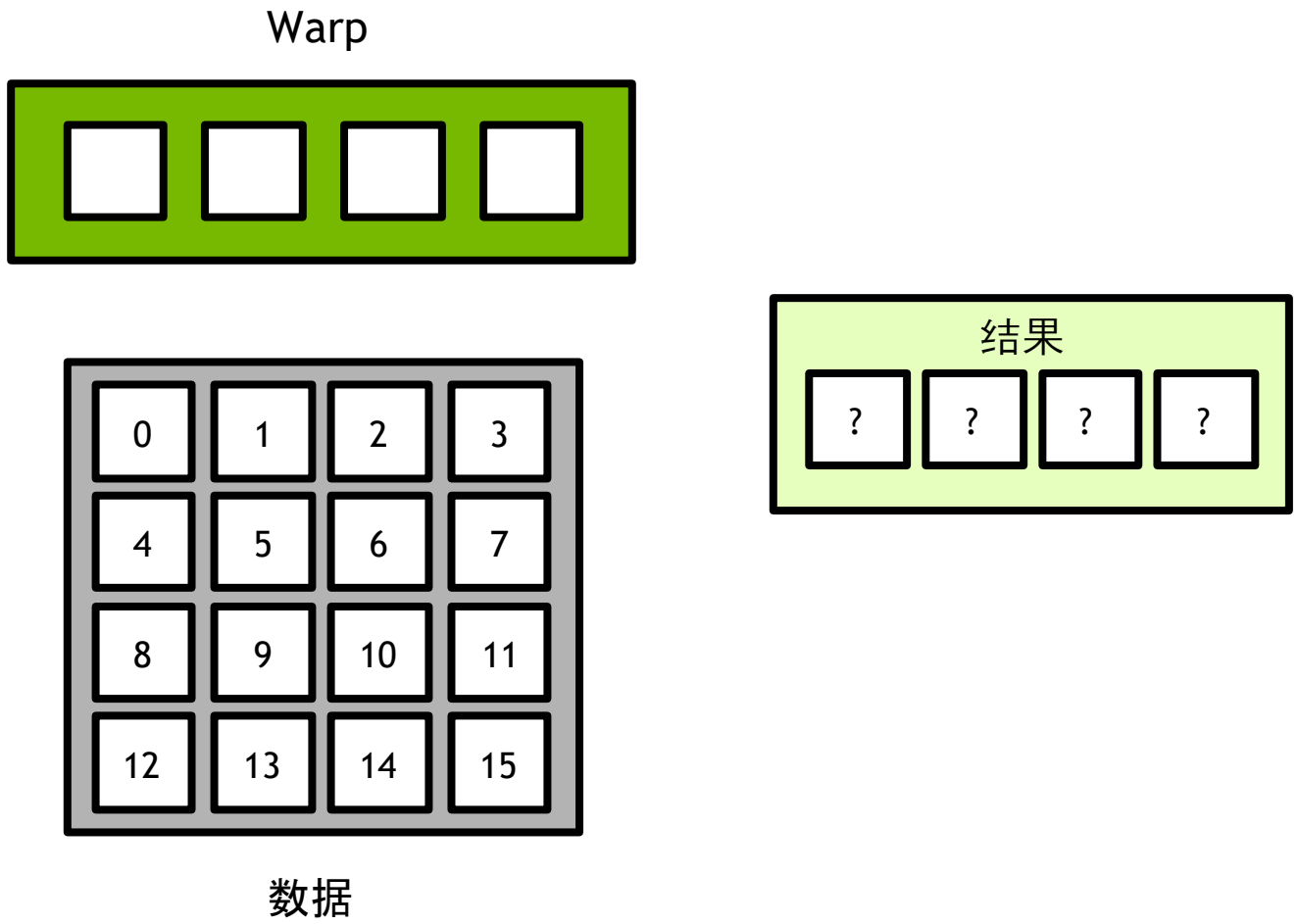
不幸的是，当每个线程在它所在的数据行上迭代时，相同的未合并模式仍在继续。



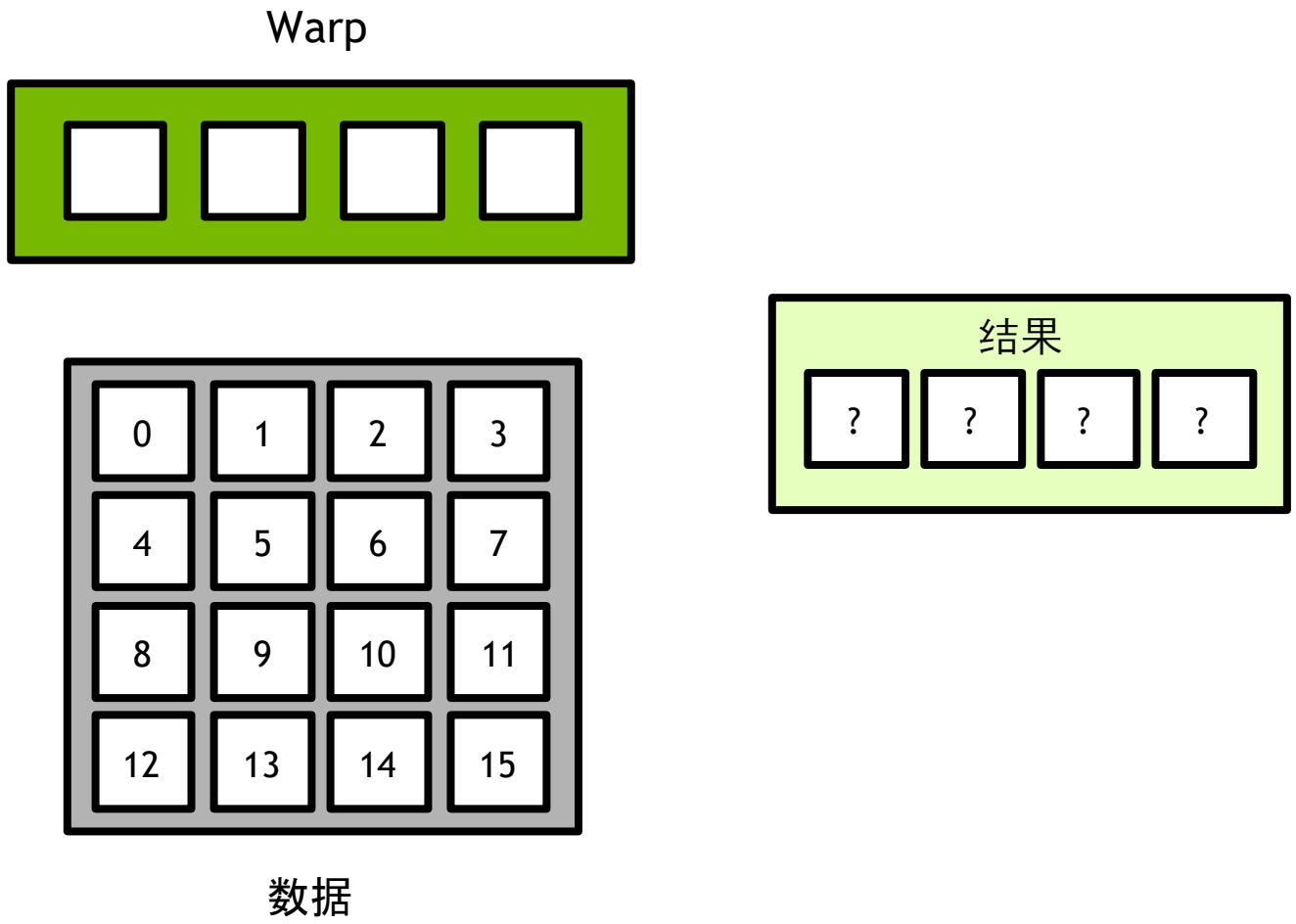
在这个例子中，我们传输了 16 个内存行，每个传输行使用了 25% 的数据



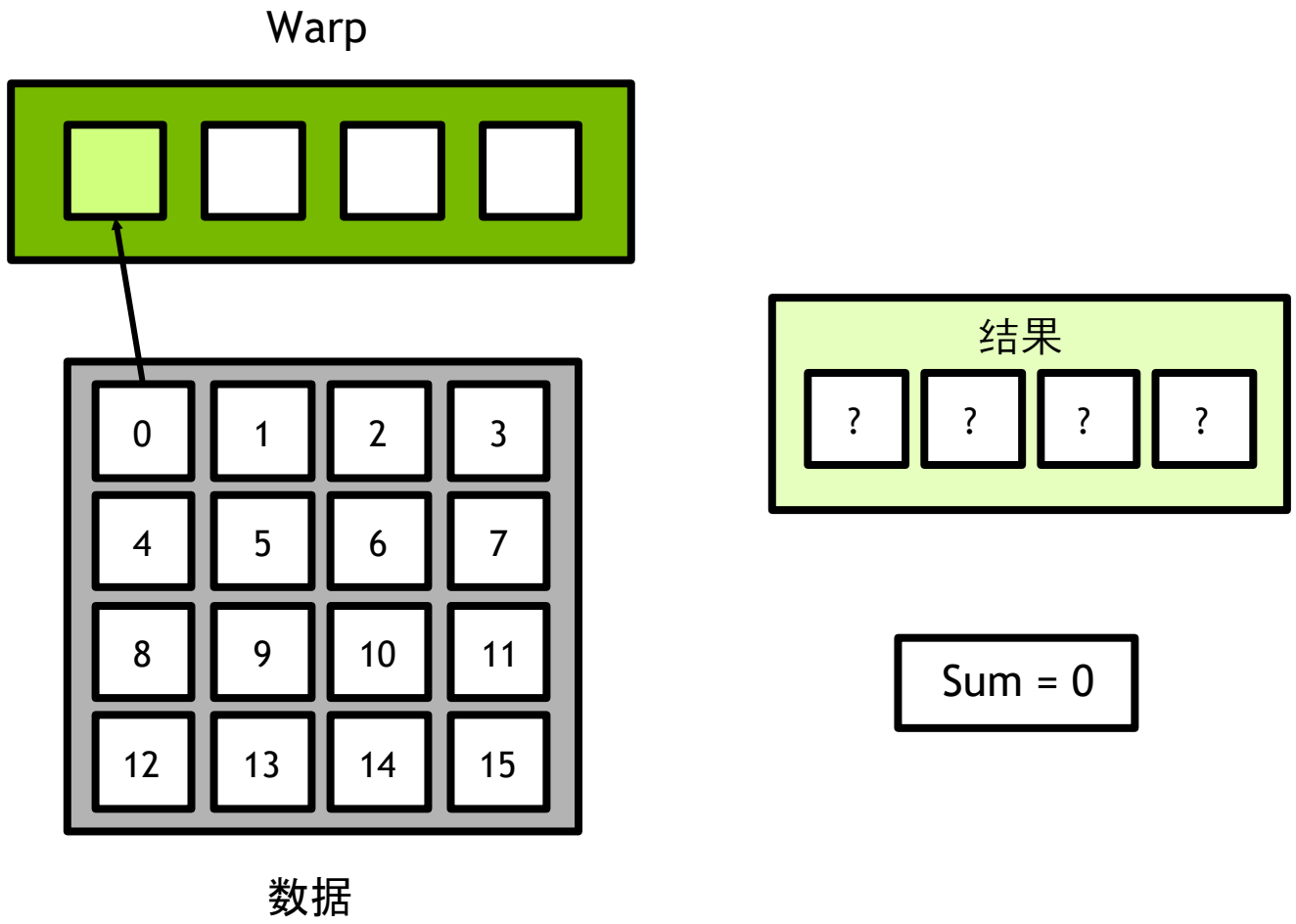
让我们比较一个将矩阵的每列之和存储在结果向量中的核函数



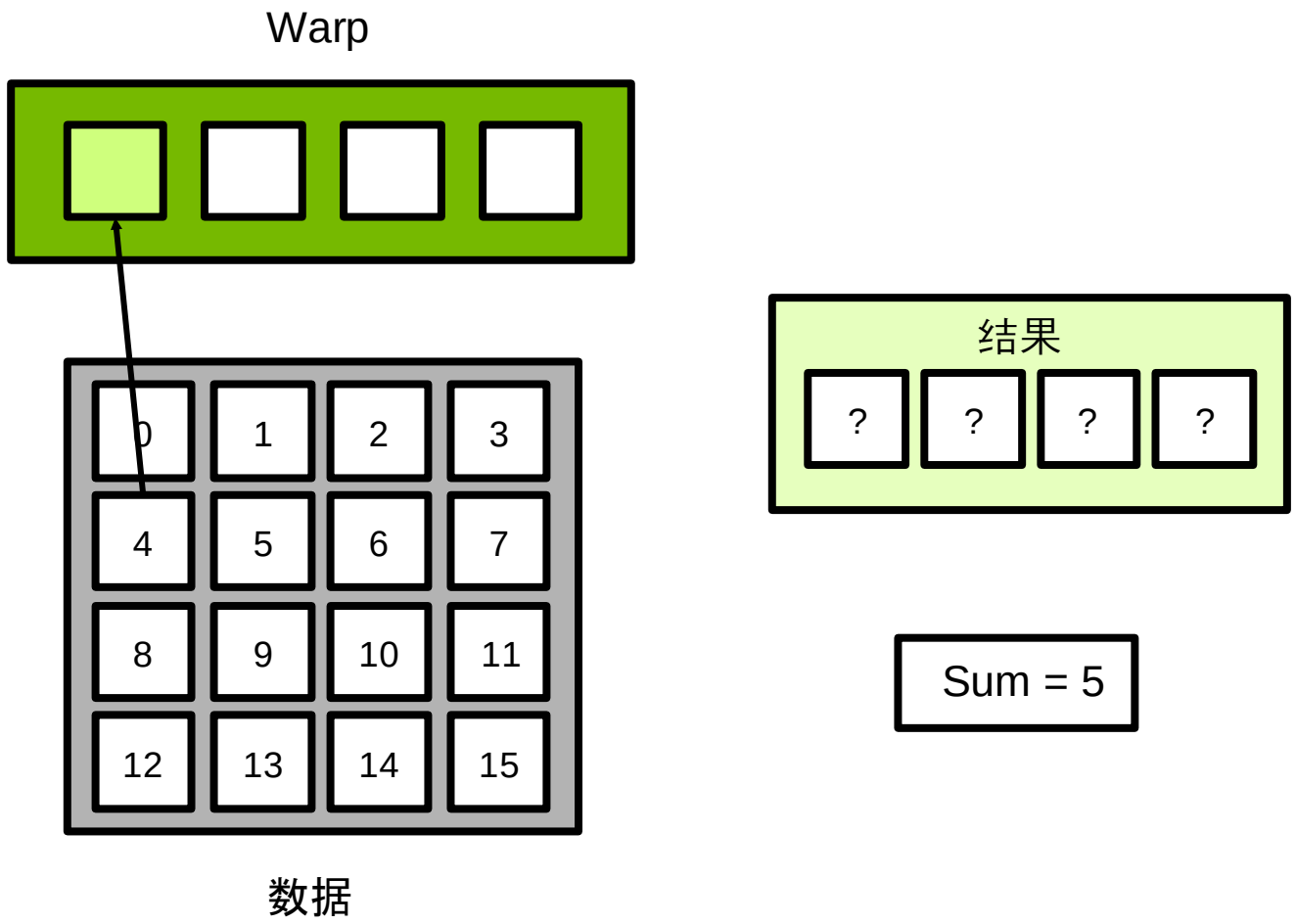
单个线程可以遍历一列，求和，然后将结果写入结果向量



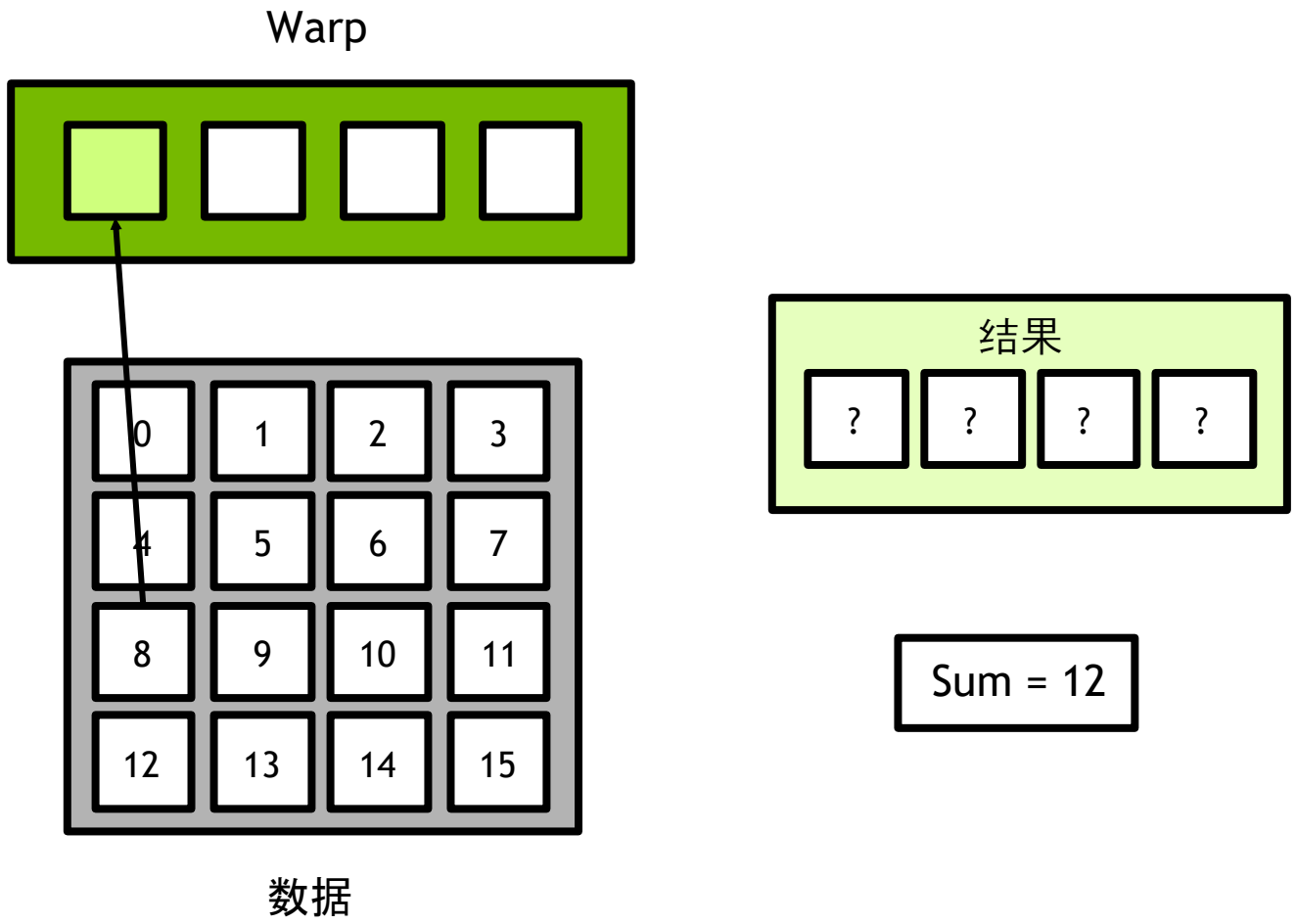
单个线程可以迭代一列，将其求和，然后将结果写入解向量中。



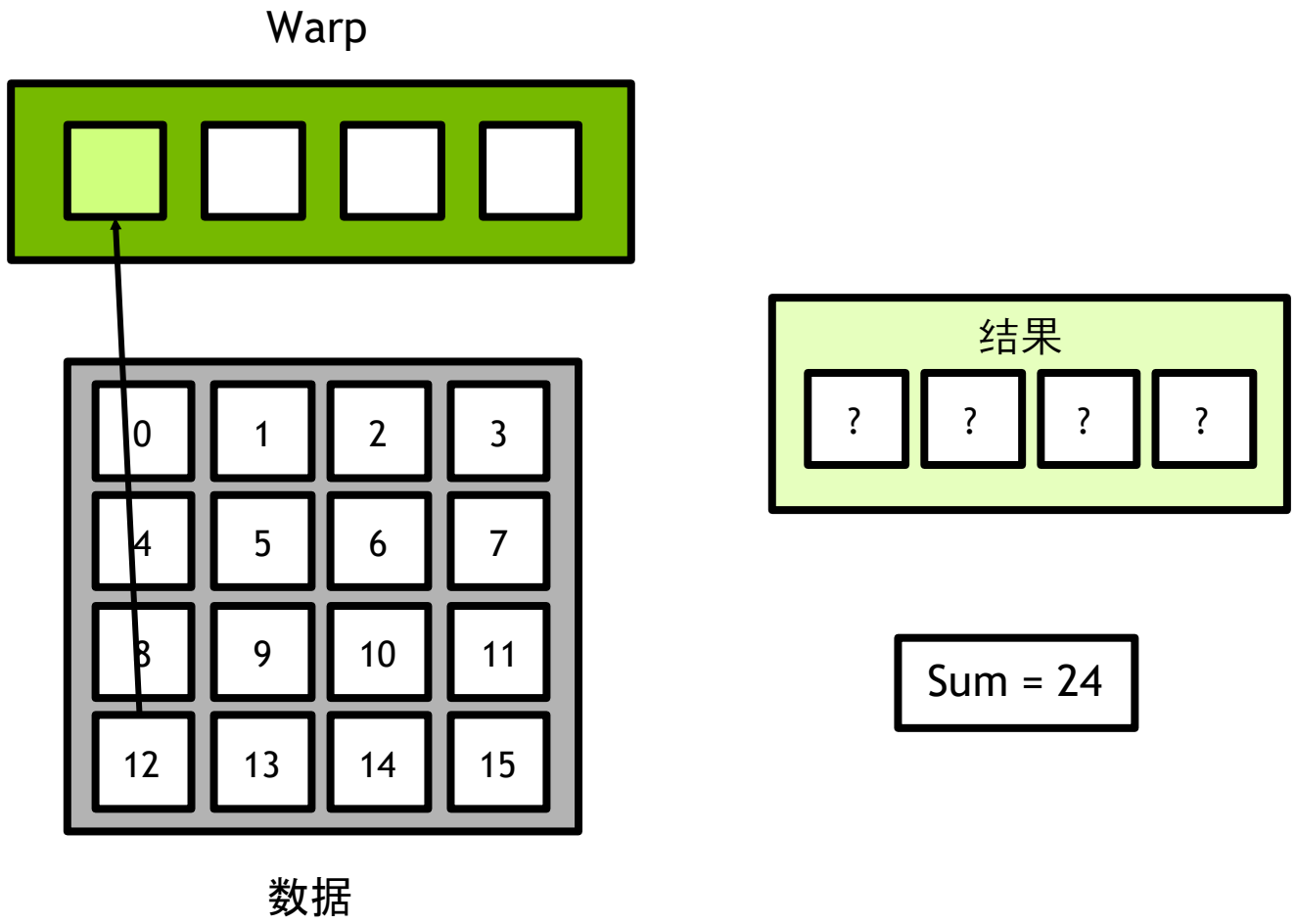
单个线程可以迭代一列，将其求和，然后将结果写入解向量中。



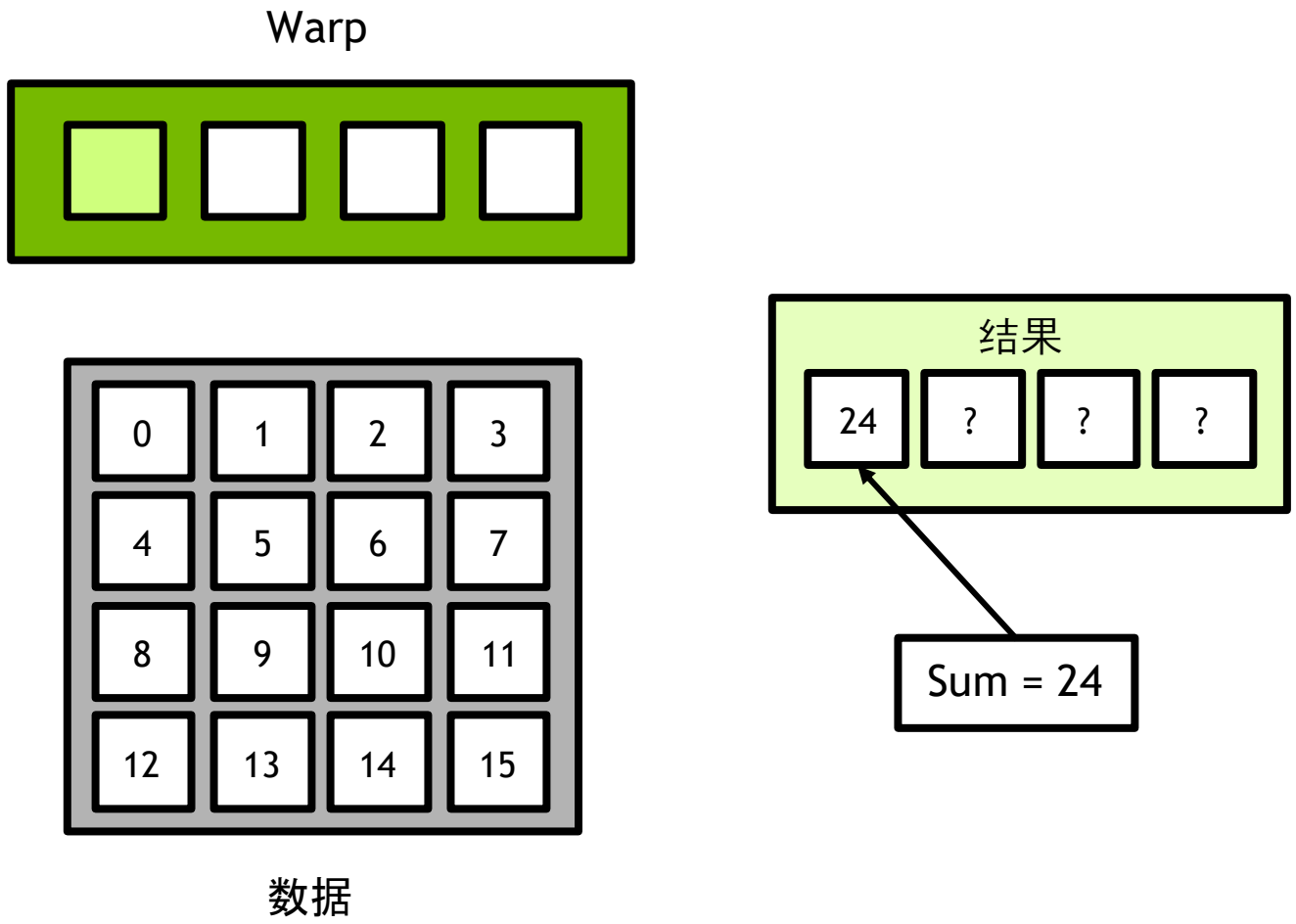
单个线程可以迭代一列，将其求和，然后将结果写入解向量中。



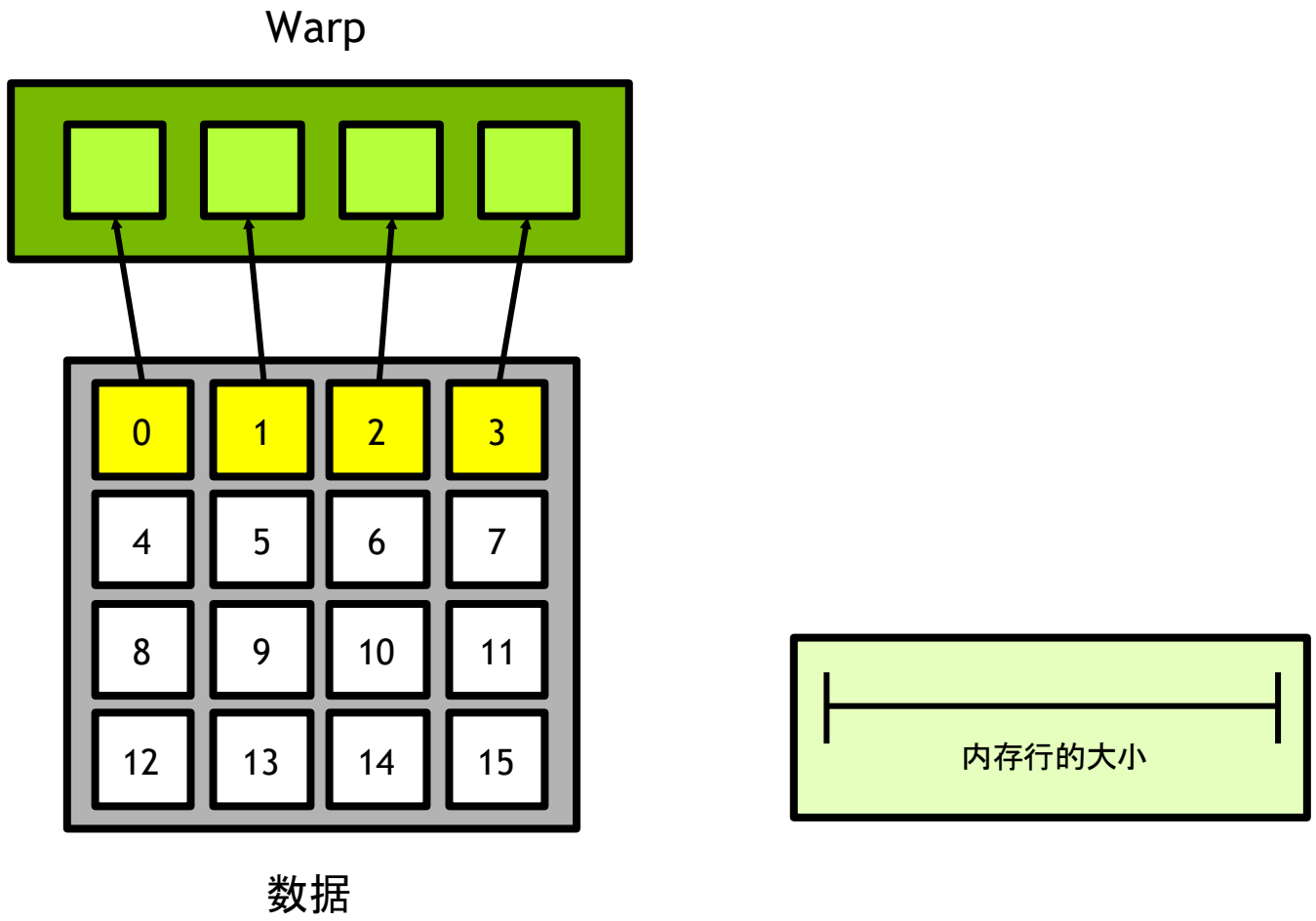
单个线程可以迭代一列，将其求和，然后将结果写入解向量中。



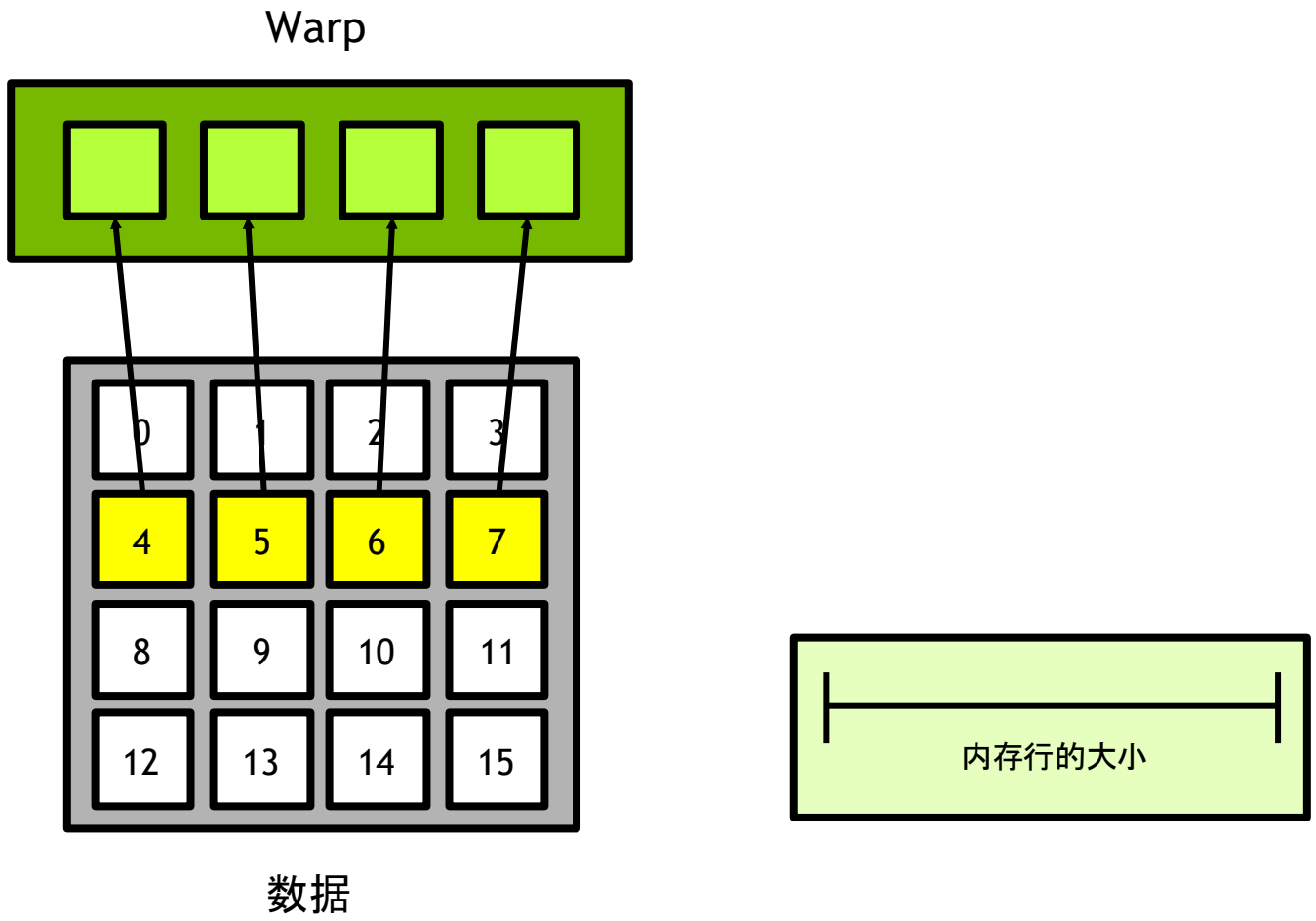
单个线程可以迭代一列，将其求和，然后将结果写入解向量中。



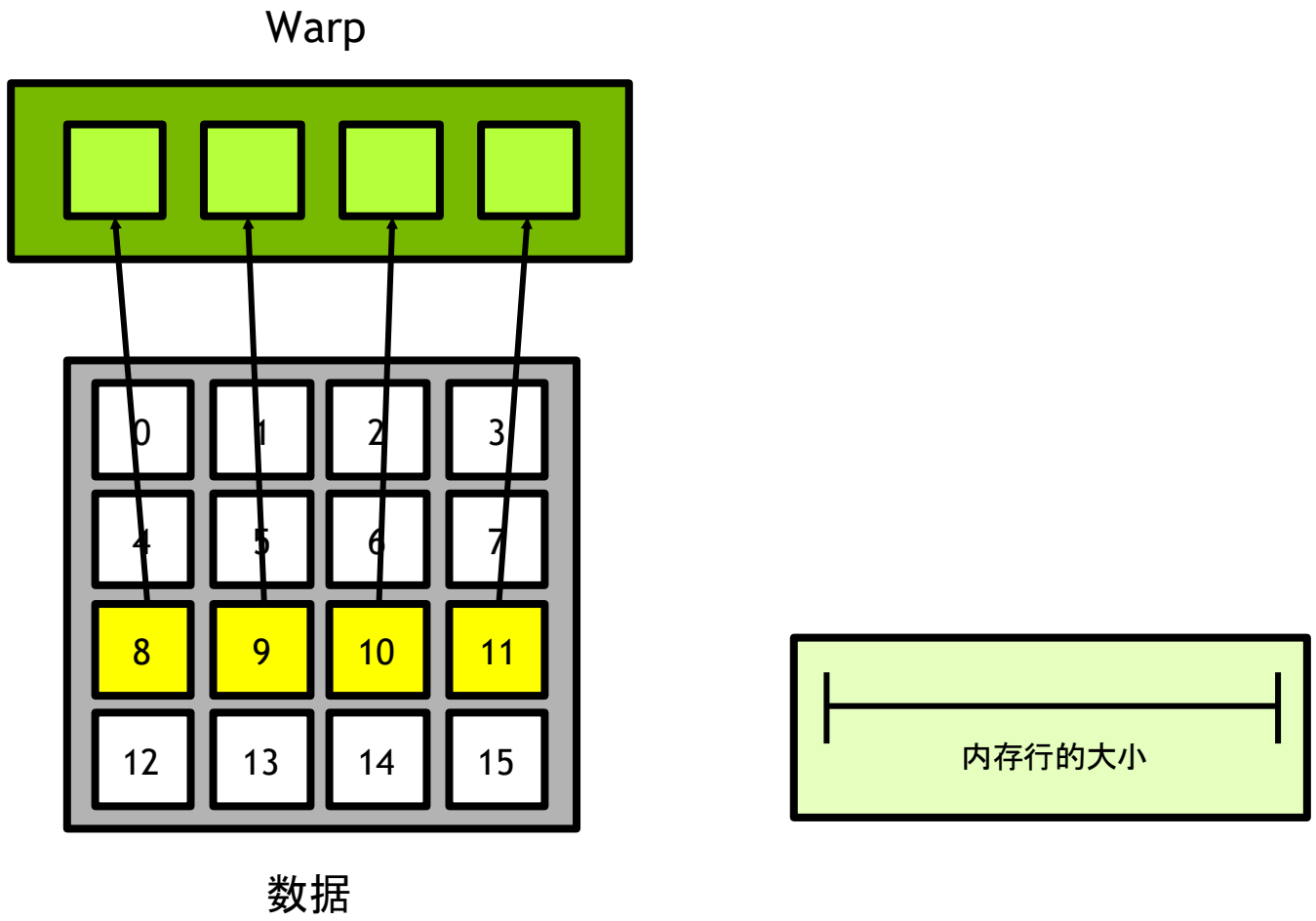
在这里，当我们考虑并行执行时，我们看到 warp 的内存访问是合并的。



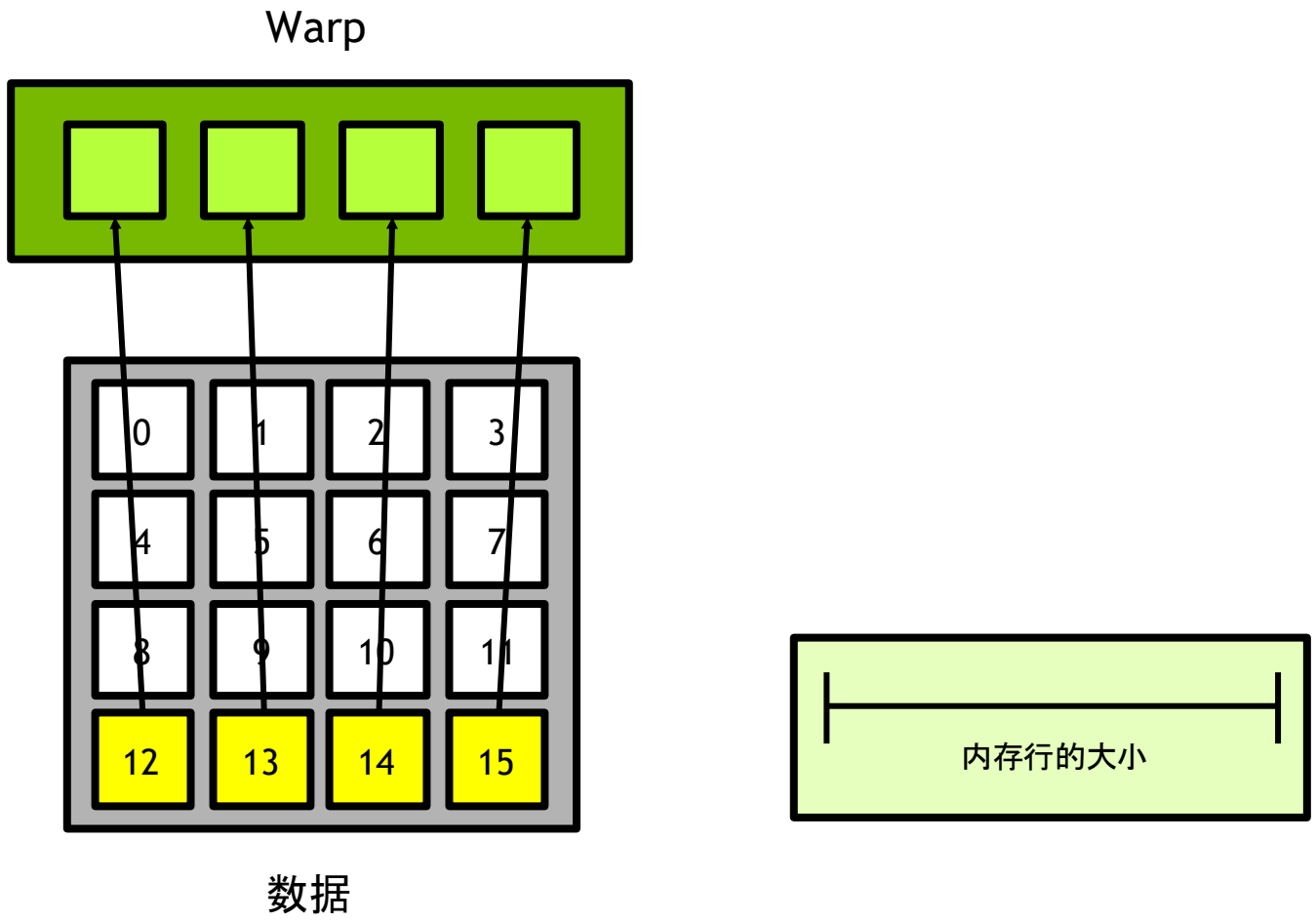
在这里，当我们考虑并行执行时，我们看到 warp 的内存访问是合并的。



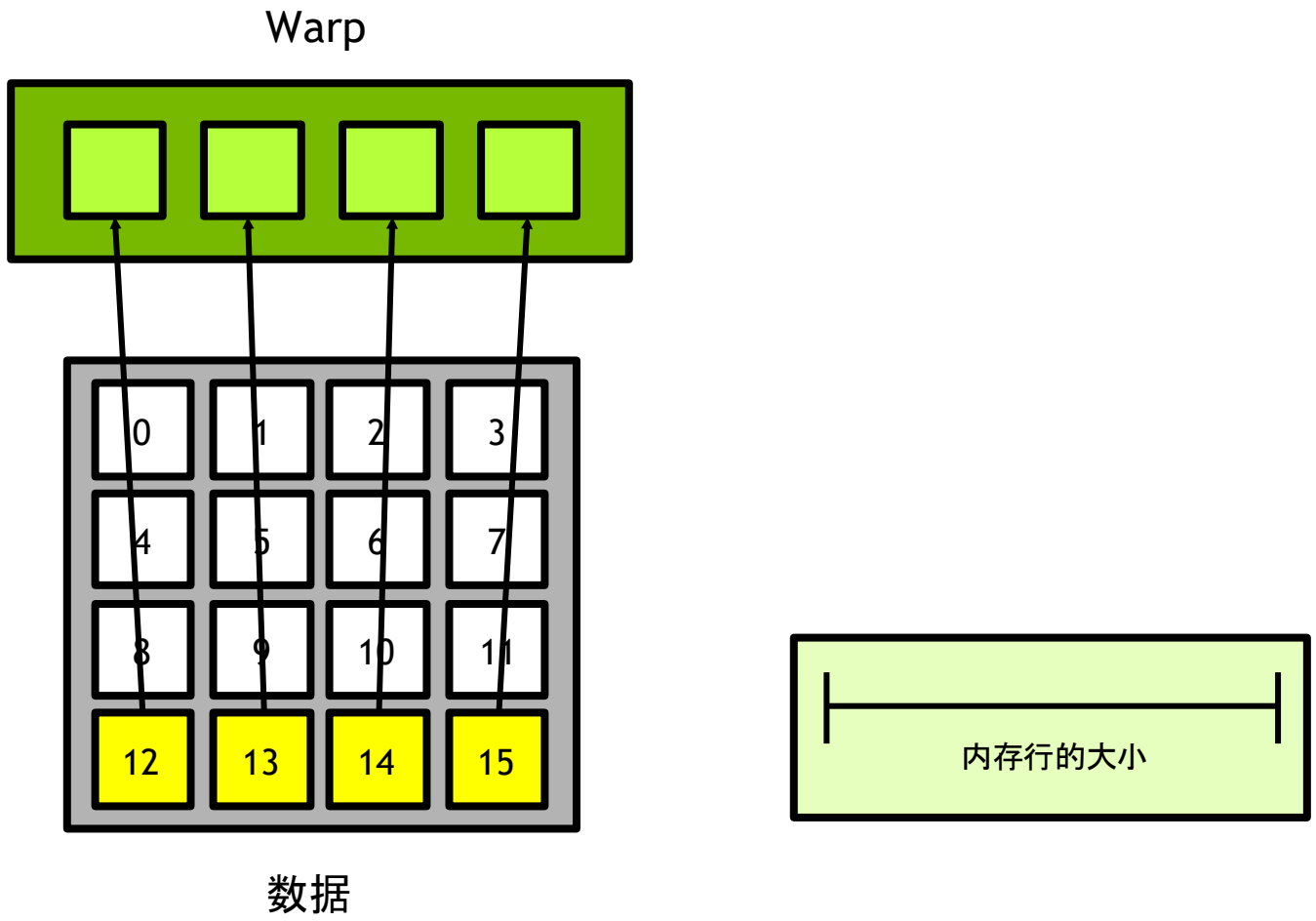
在这里，当我们考虑并行执行时，我们看到 warp 的内存访问是合并的。



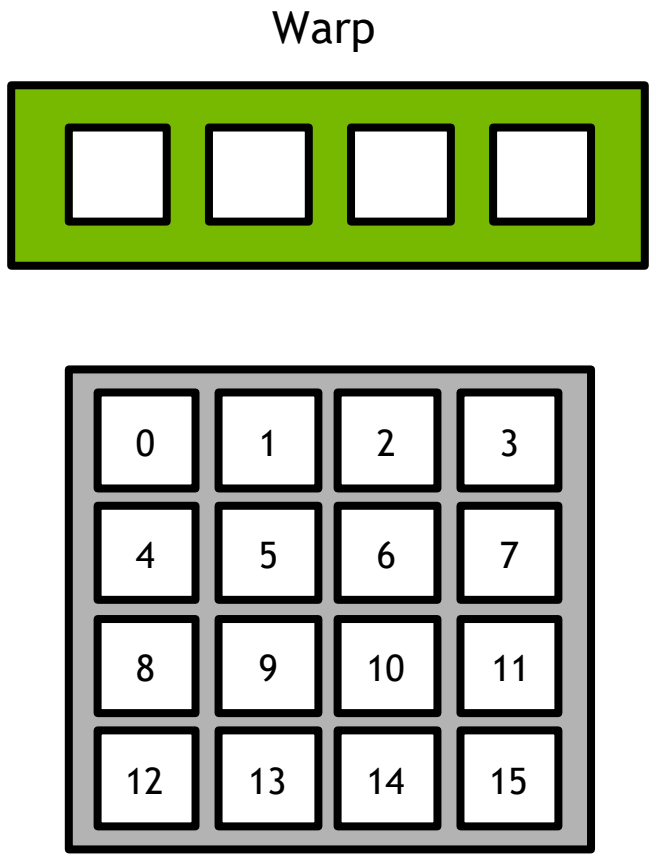
在这里，当我们考虑并行执行时，我们看到 warp 的内存访问是合并的。



要记住的一个有用提示是，threadIdx.x 的增量应该映射到数据增量变化最快的索引方向上 – 在这个例子中是 x 轴。



在本例中，我们传输了 4 行内存（与 16 行相比），并且每个传输行使用了 100% 的数据（与 25% 相比）。



NVIDIA | DEEP LEARNING INSTITUTE

学习更多课程，请访问 www.nvidia.cn/DLI