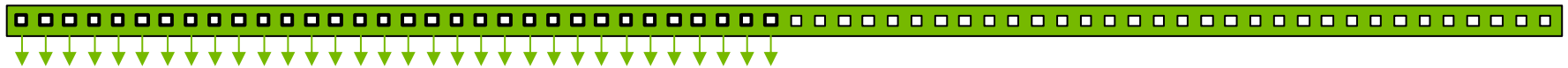
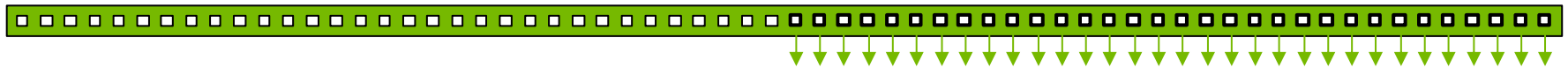
线程块被划分为 32 线程 Warp

线程块被划分为 32 线程 Warp

线程块被划分为 32 线程 Warp

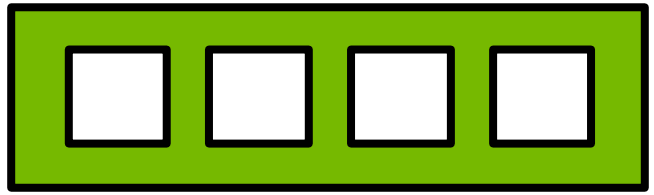


32　线程的 warp 级别并行发

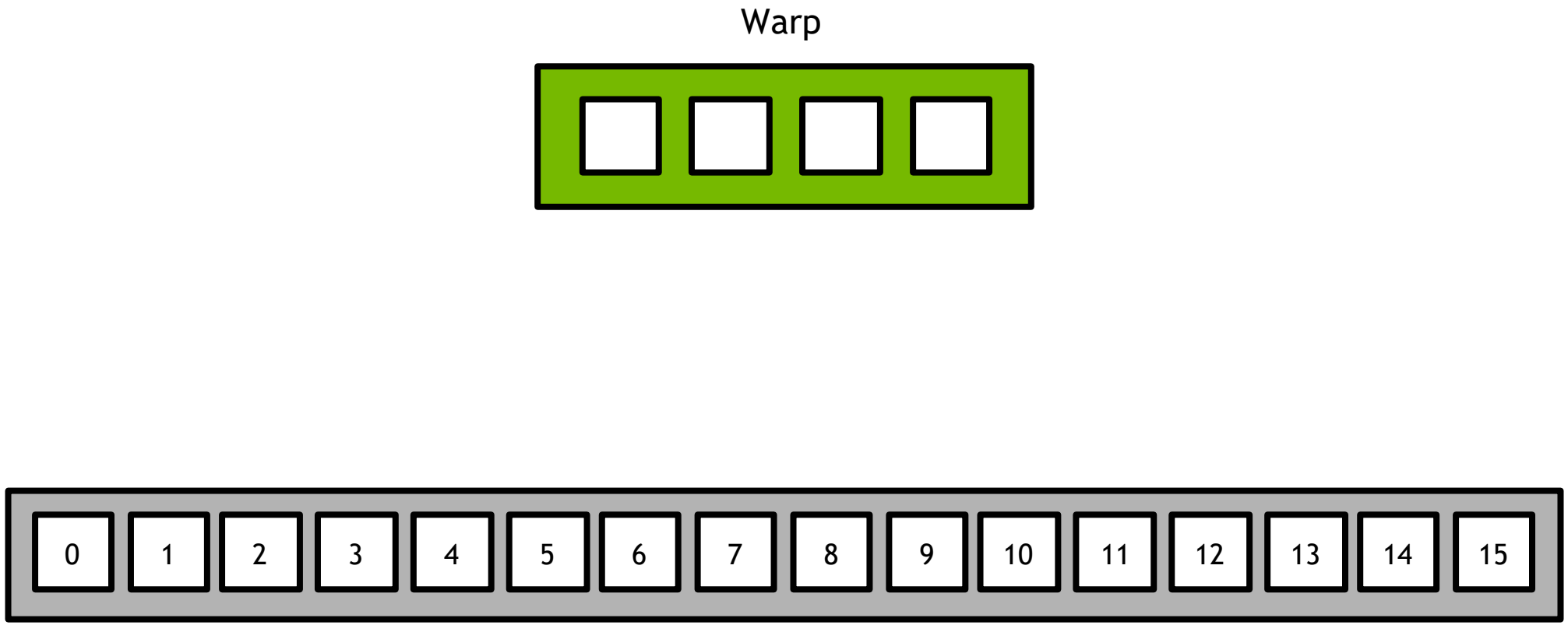


为节省每一页上的空间，我们仅将 4 线程视为一个 Warp

Warp

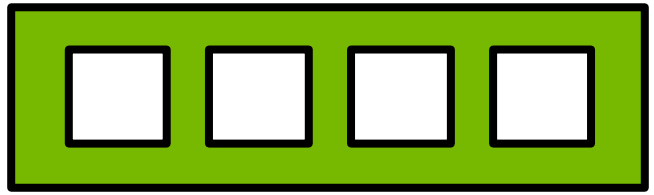


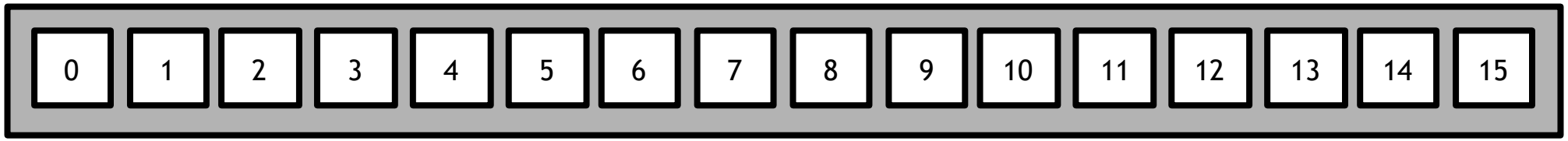
32 节段的形式传入和传出全设备内存 *



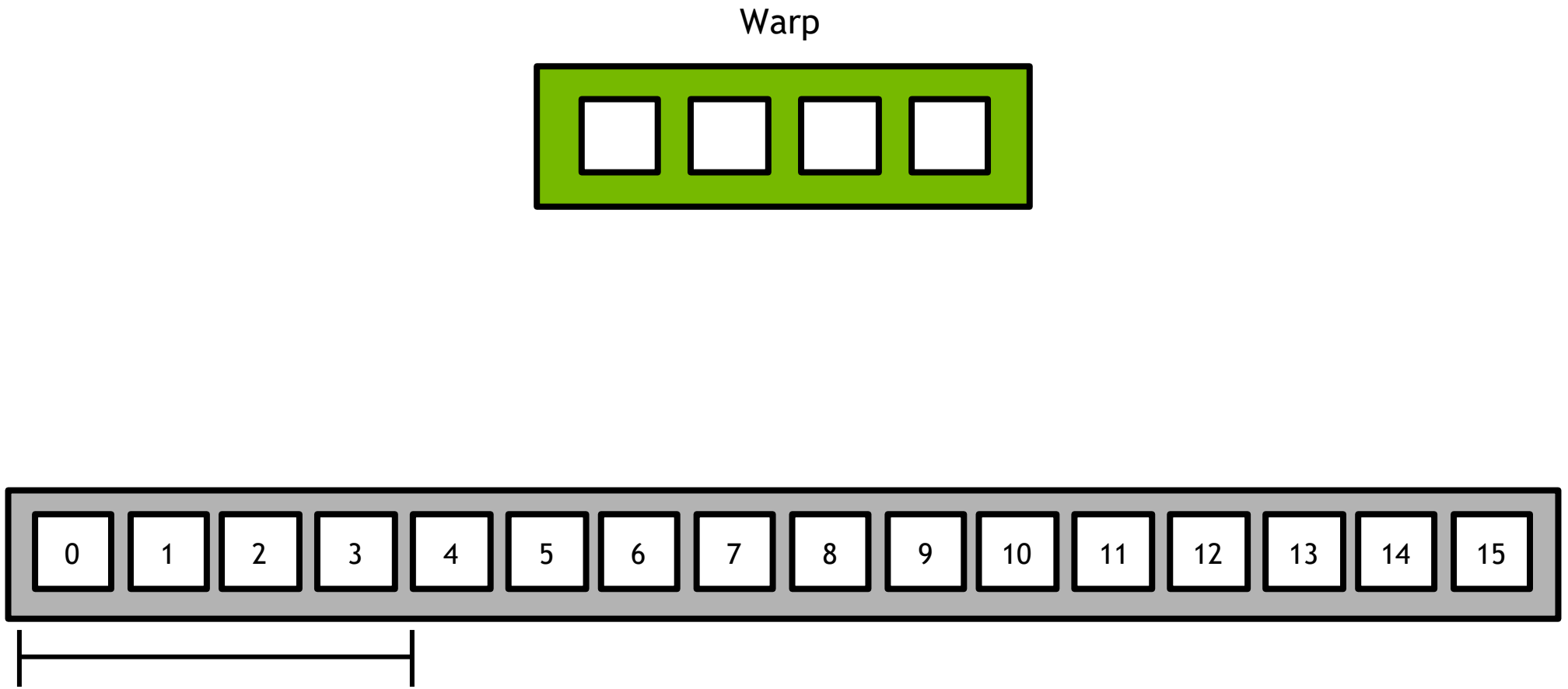
(* L1 缓存中，它将在 128 节缓存行中传输 - 详细信息，请阅实验的 notebook)

Warp



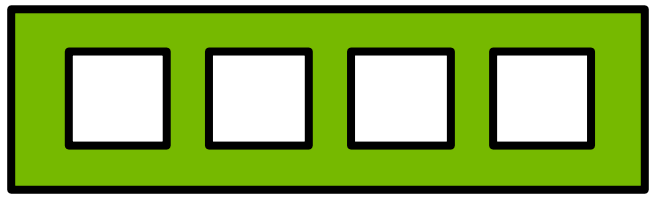


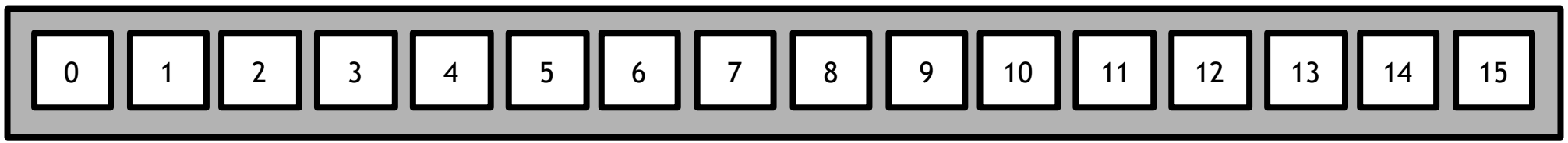
这里的演示中，我们将 4
视为连续内存中的固定长度的一行。



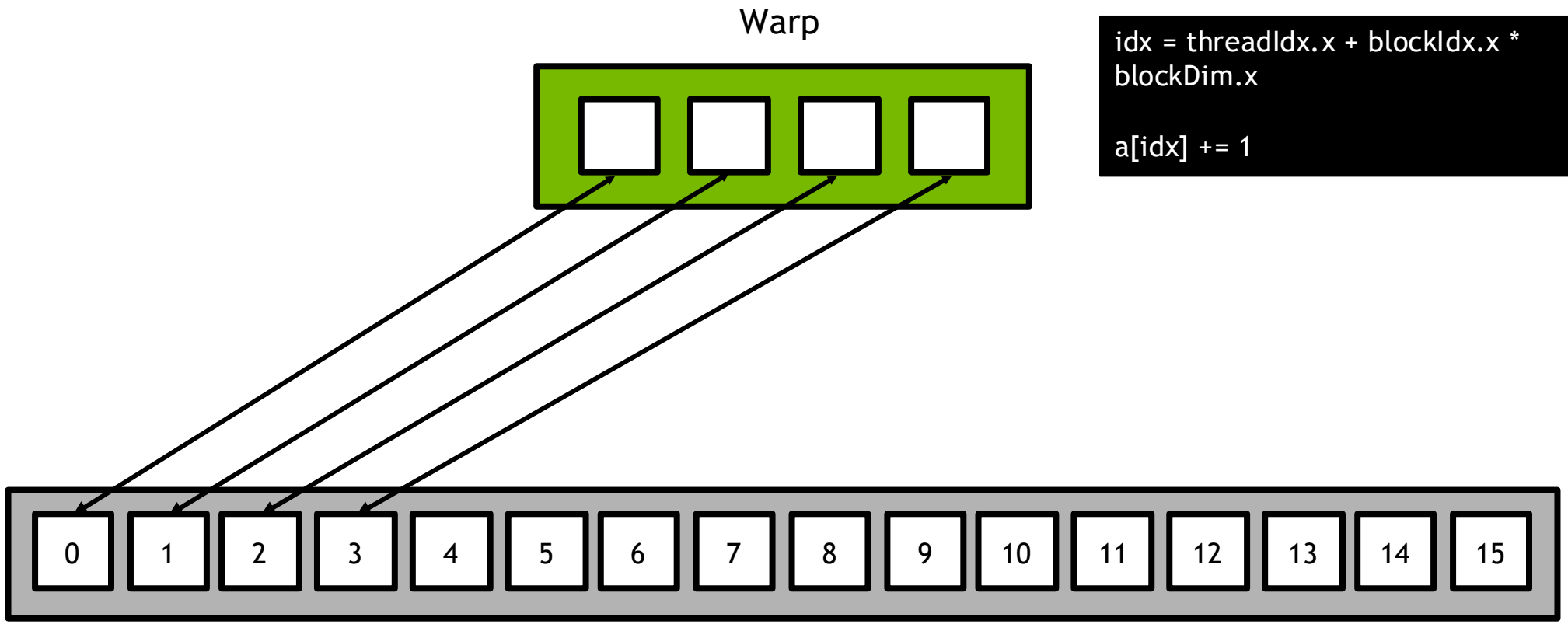
统将试图使满足 warp 读/

Warp



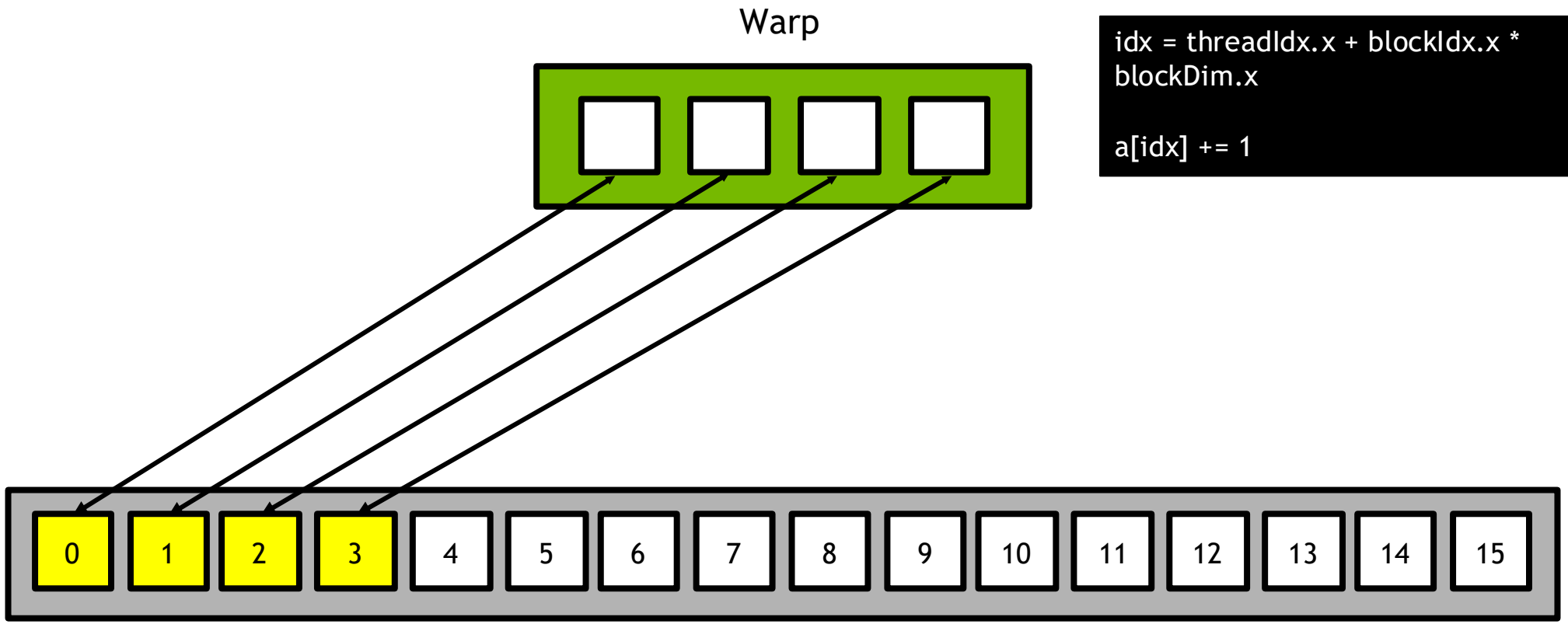


请求的地址是连续的

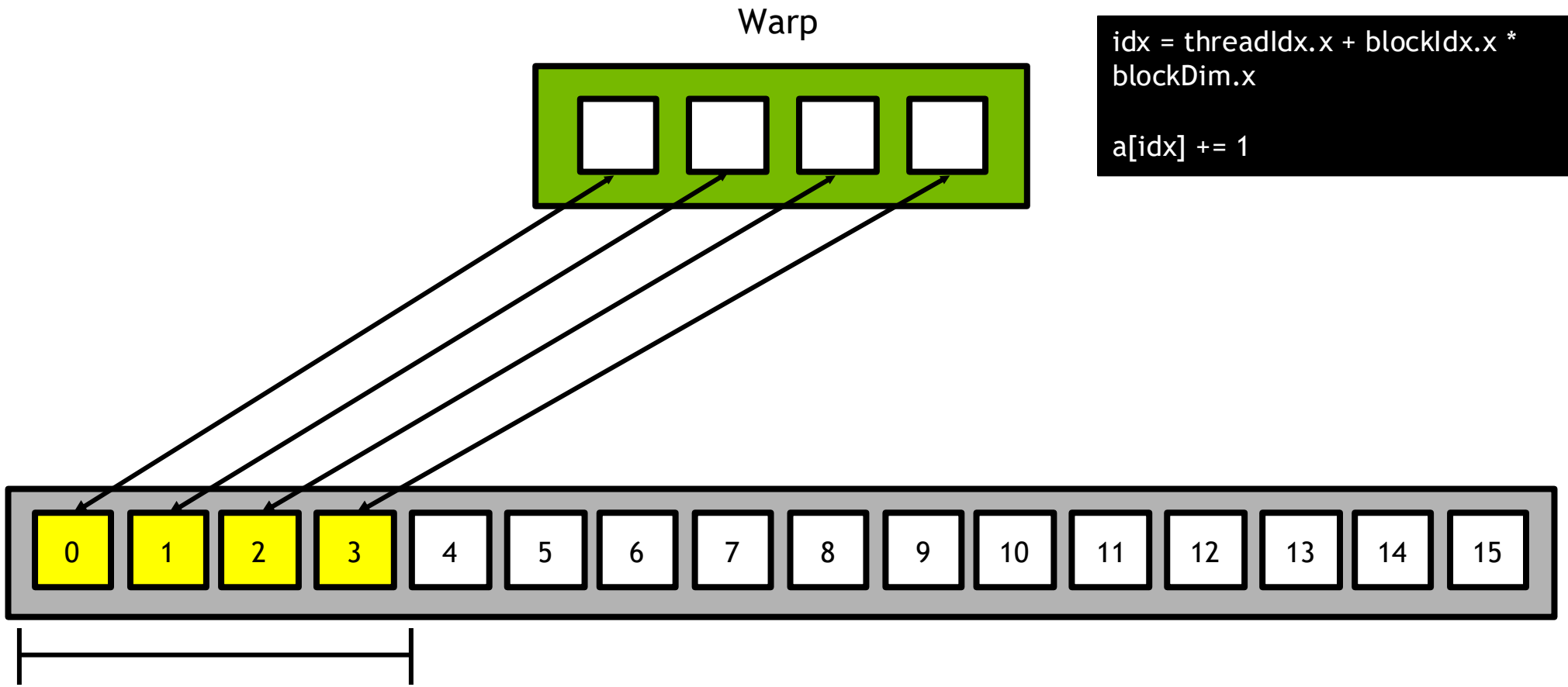



```
idx = threadIdx.x + blockIdx.x *
blockDim.x

a[idx] += 1
```

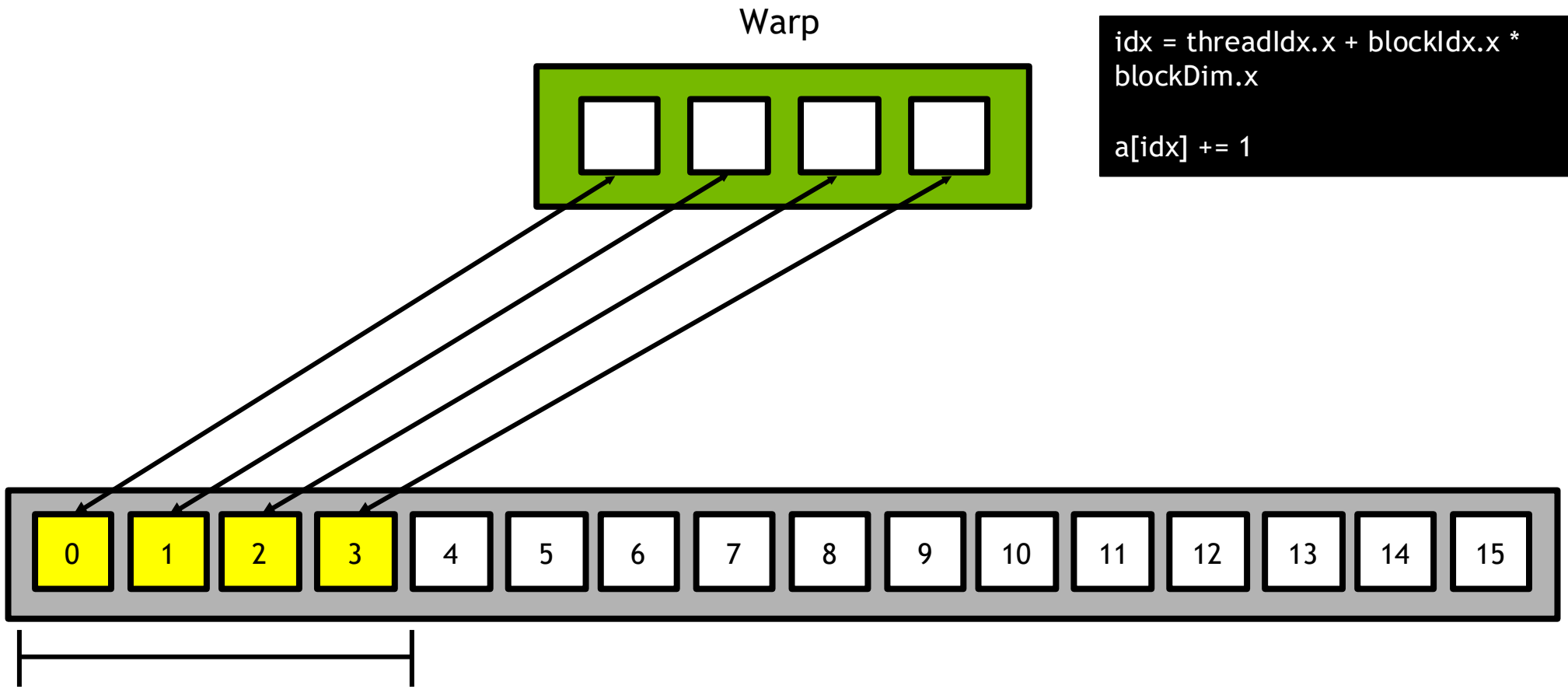
该行中的所有数据都将被使用



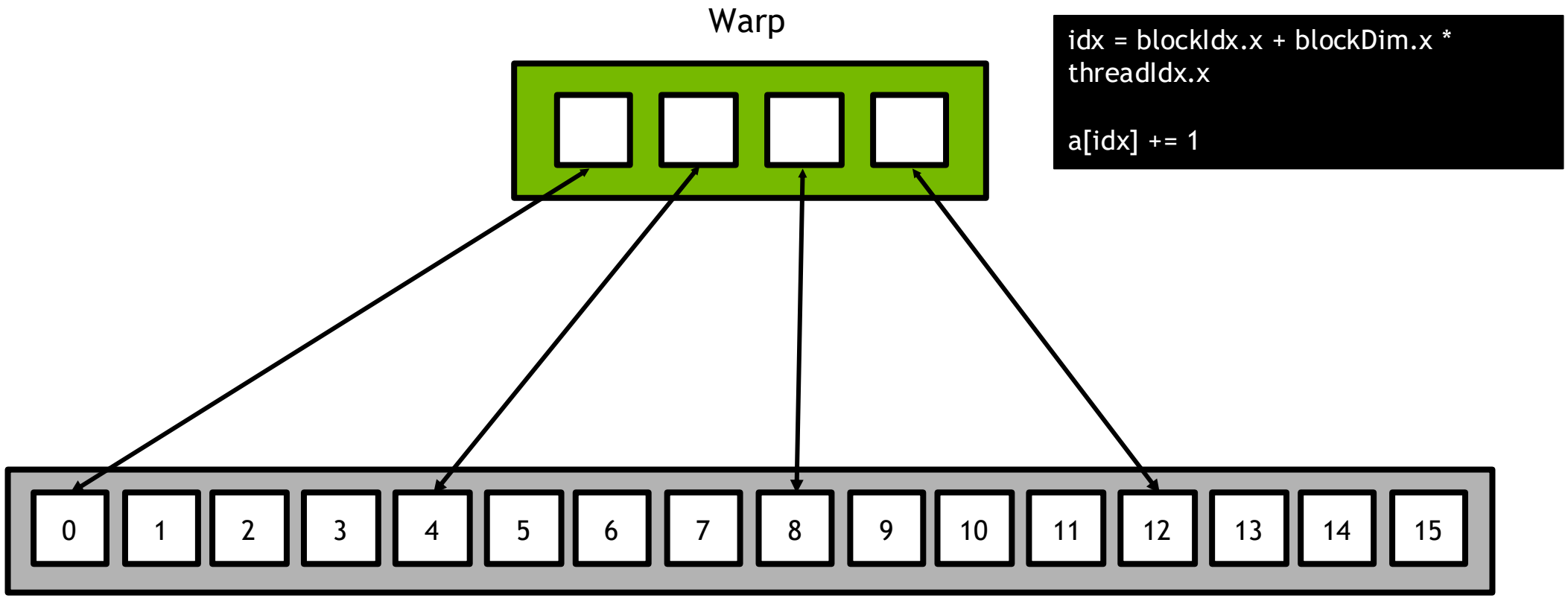
发生



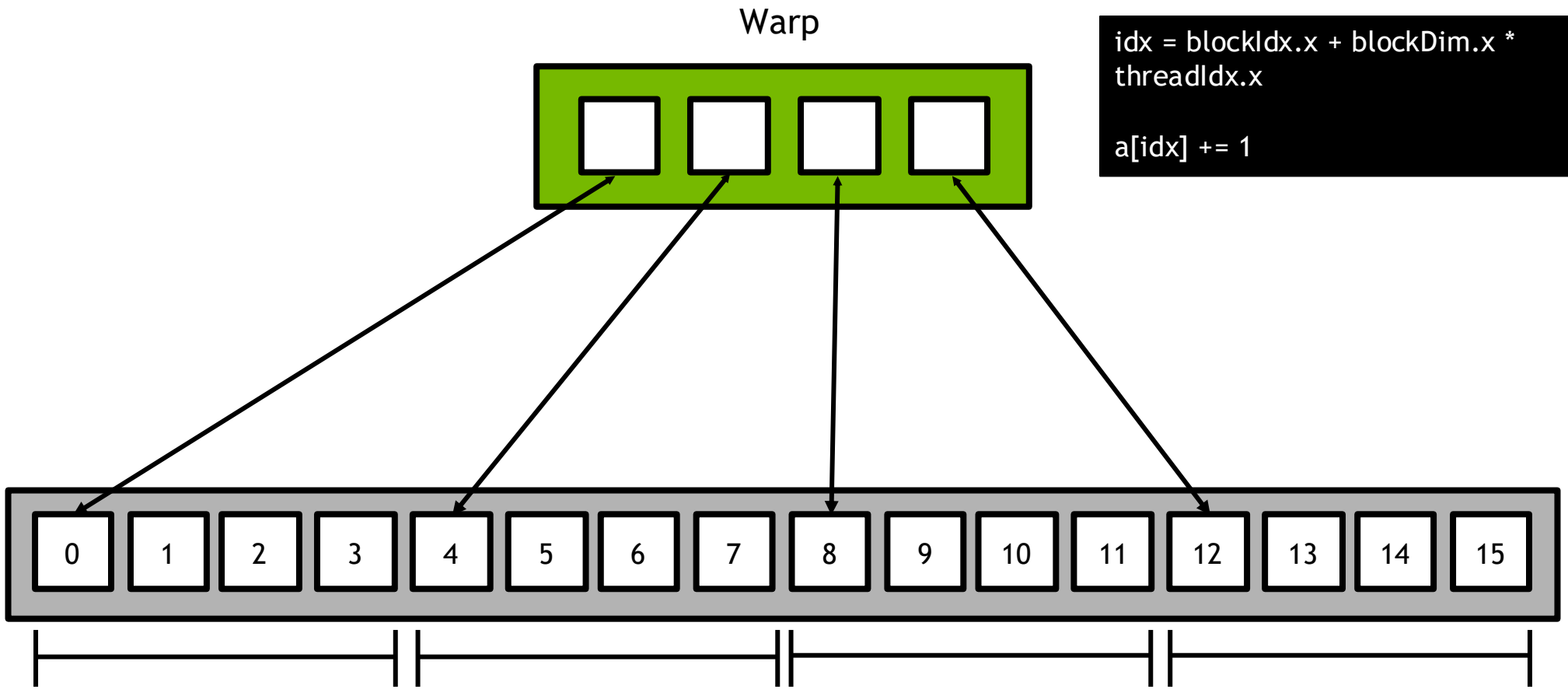
现这种情况时，内存访问是完全合并



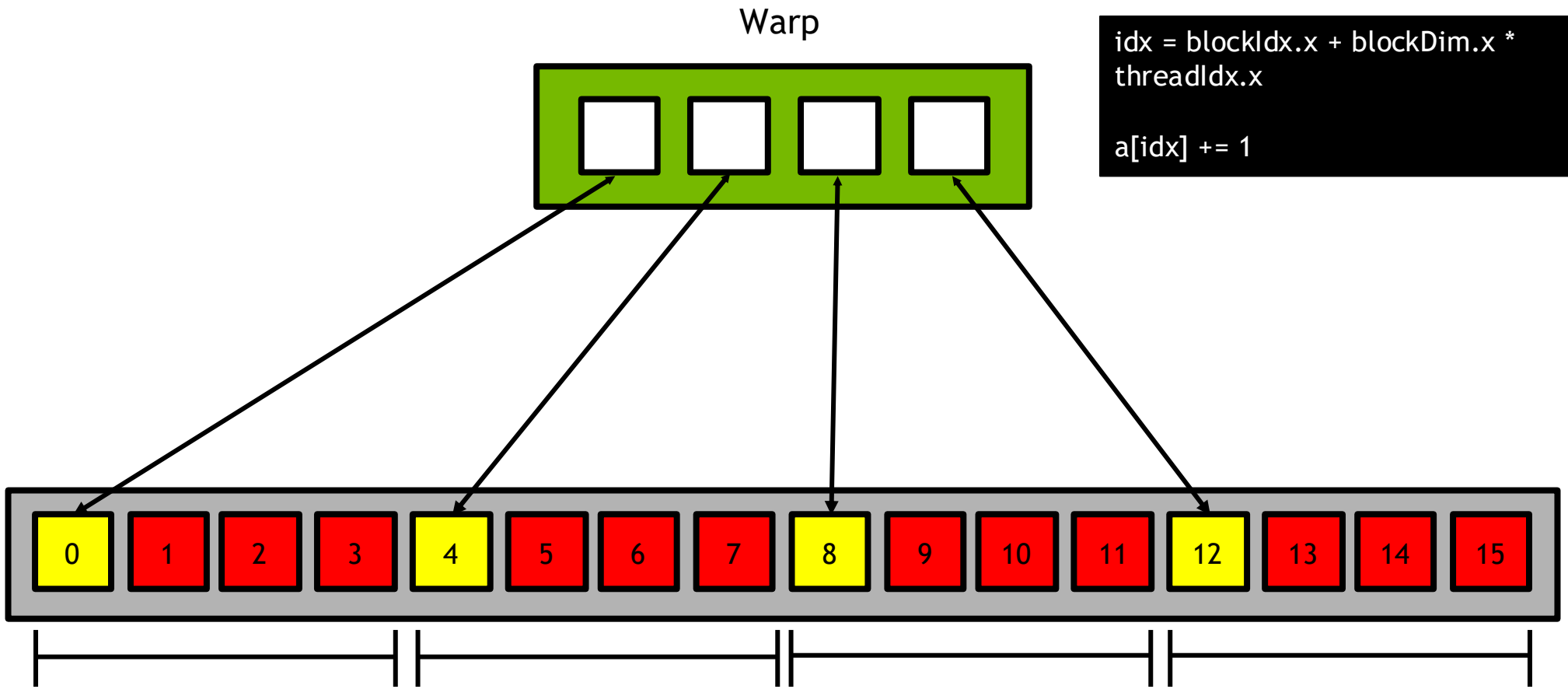
请求的内存变得不那么连续



须迁移更多的行以满足Warp

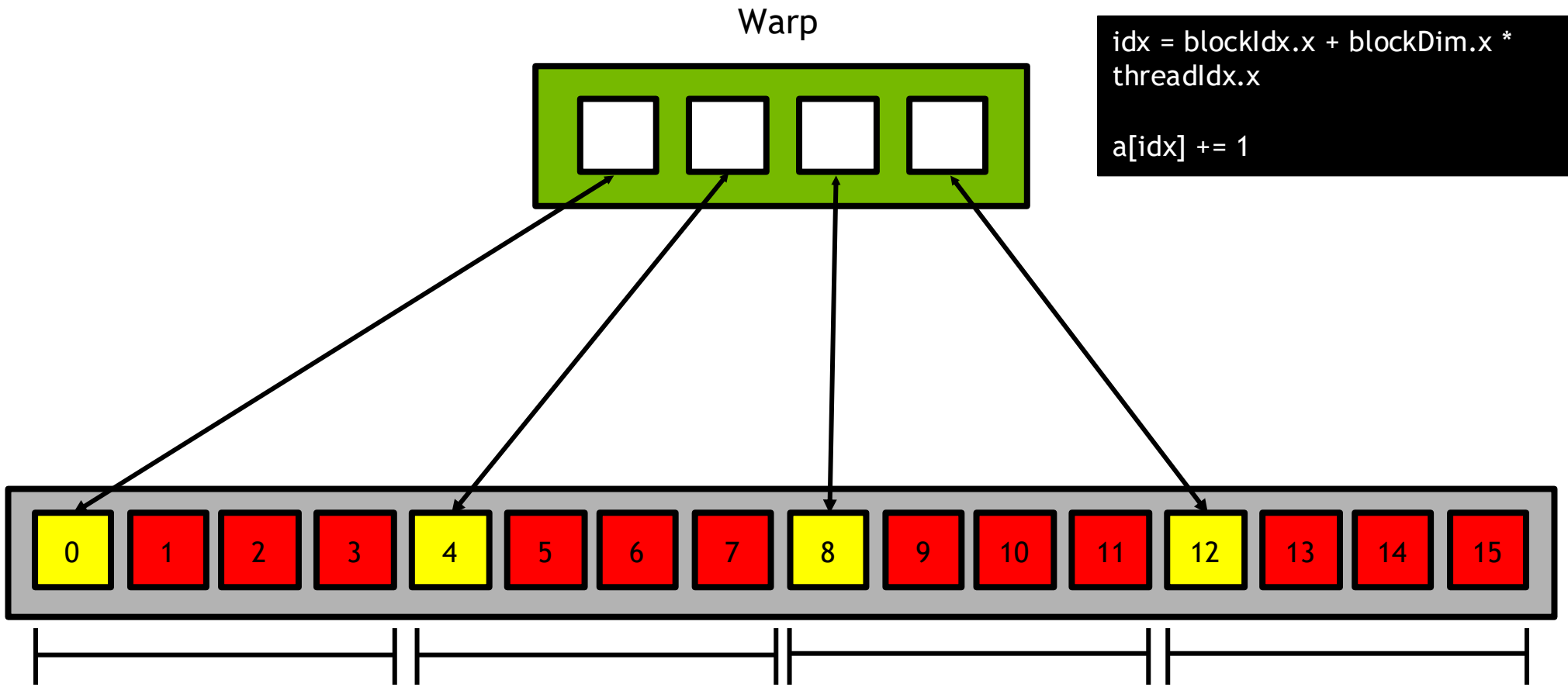


传输的数据将被闲置

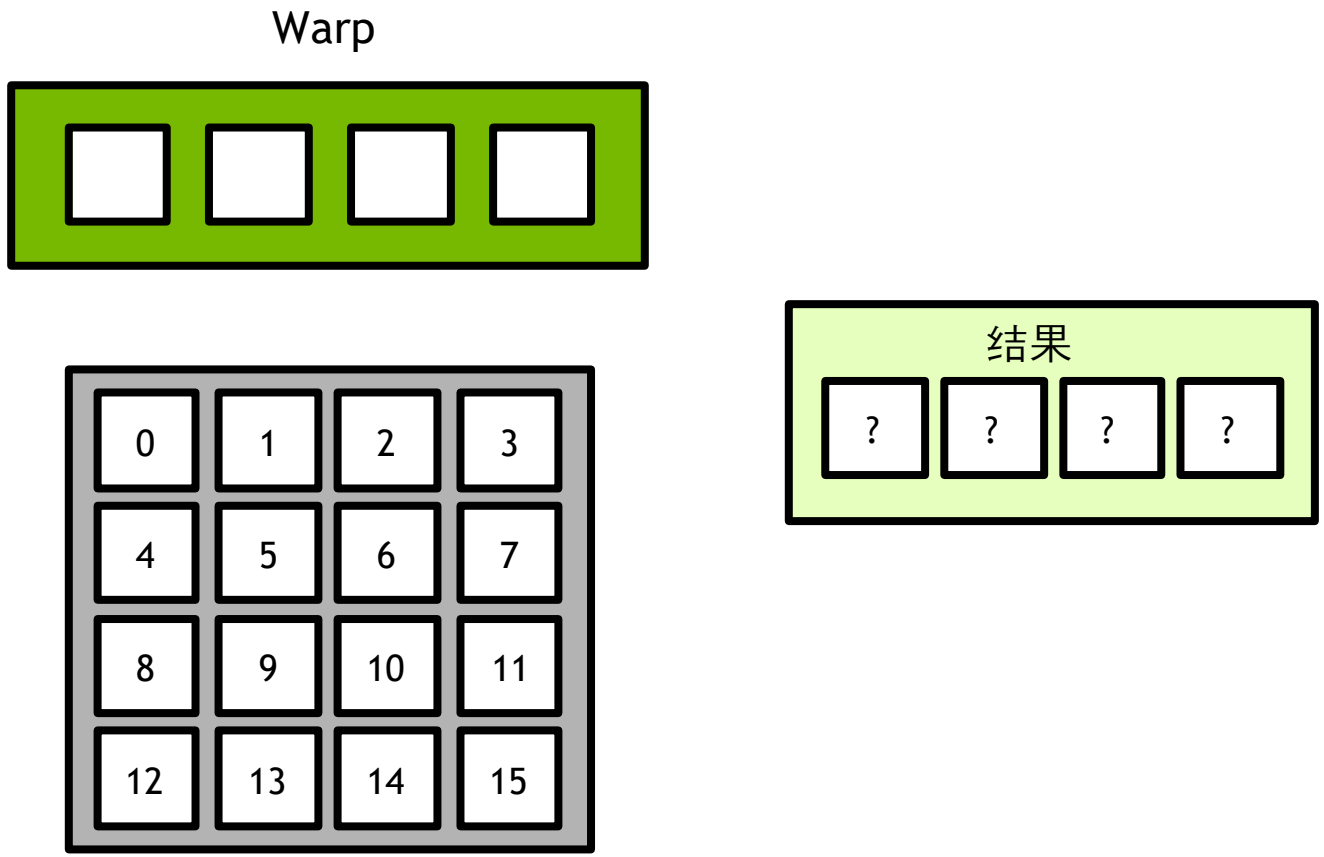


传输需要额外的时间
导致性能损失

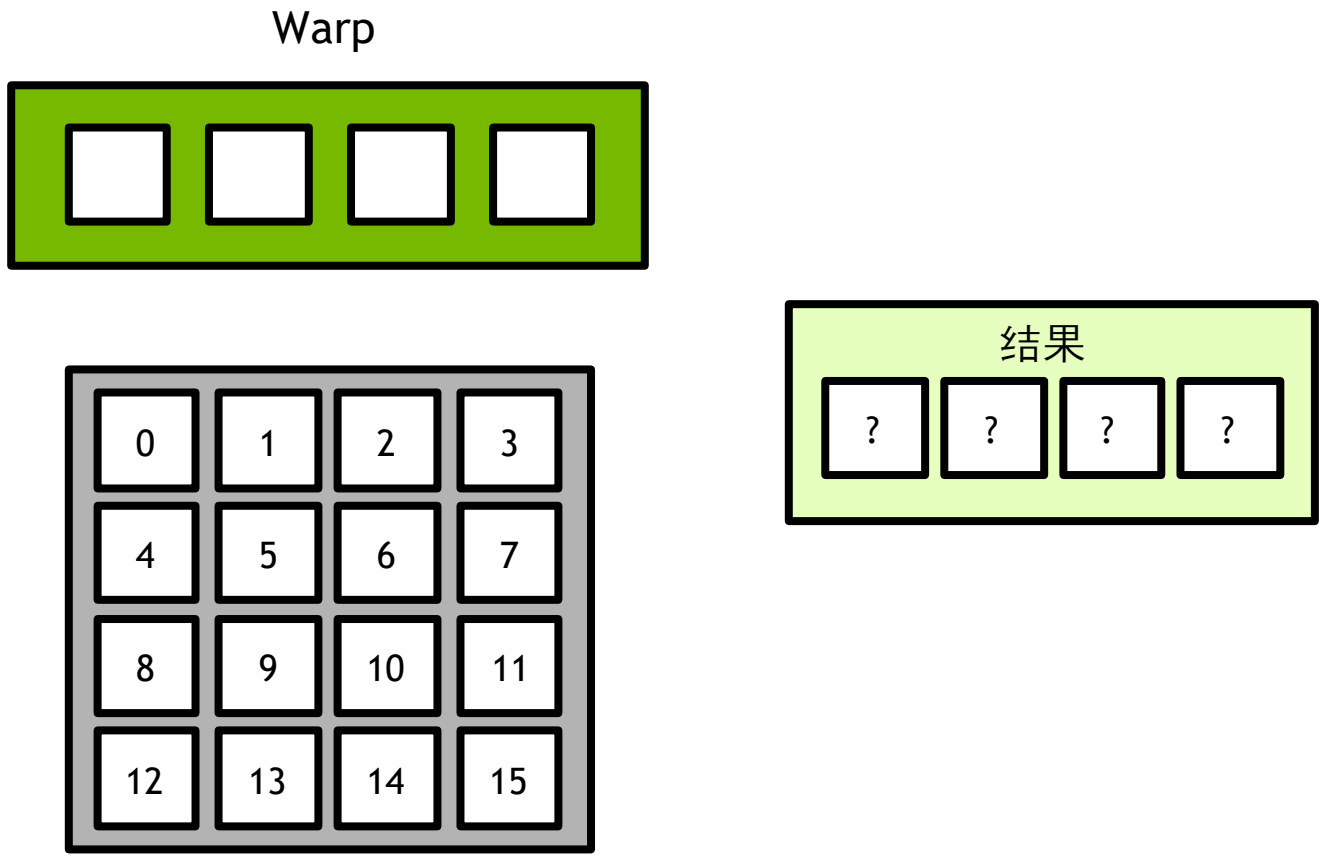
```
idx = blockIdx.x + blockDim.x *
threadIdx.x

a[idx] += 1
```



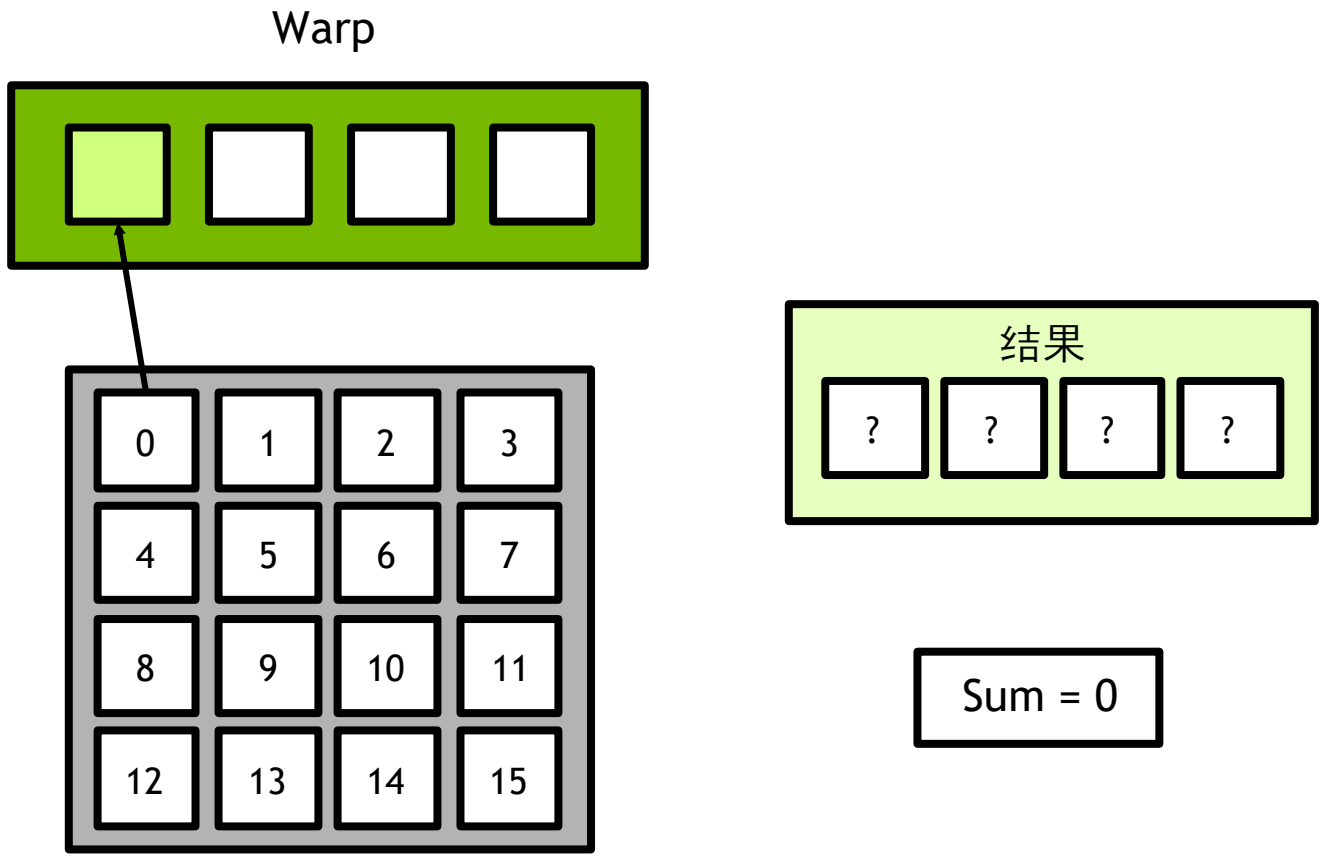
虑一个核函数，它将矩阵的每一行（这里是 4 连续的数据元素）的和存储在一个结果向量中



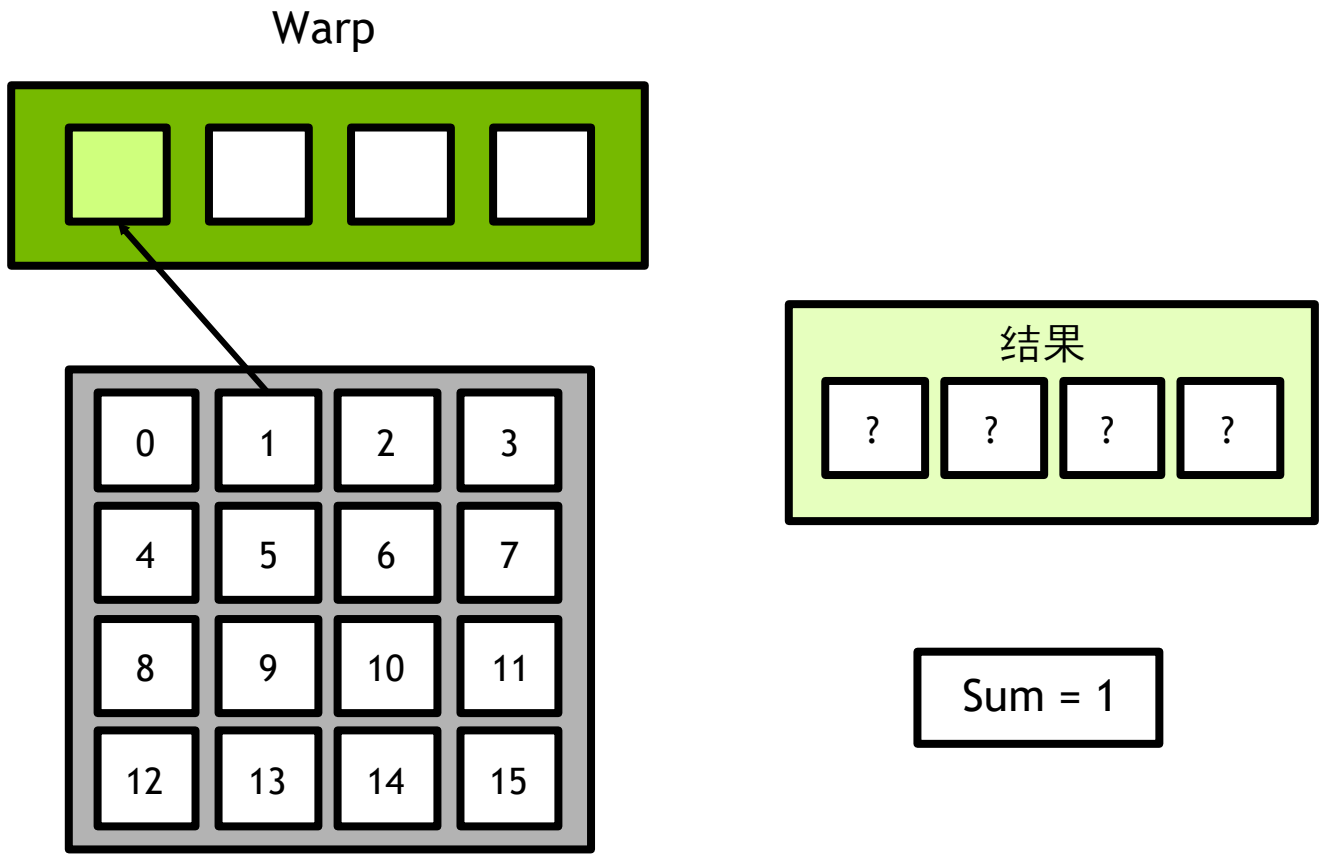
单个线程可以迭代一行，将其求和，然结果写入结果向量中。



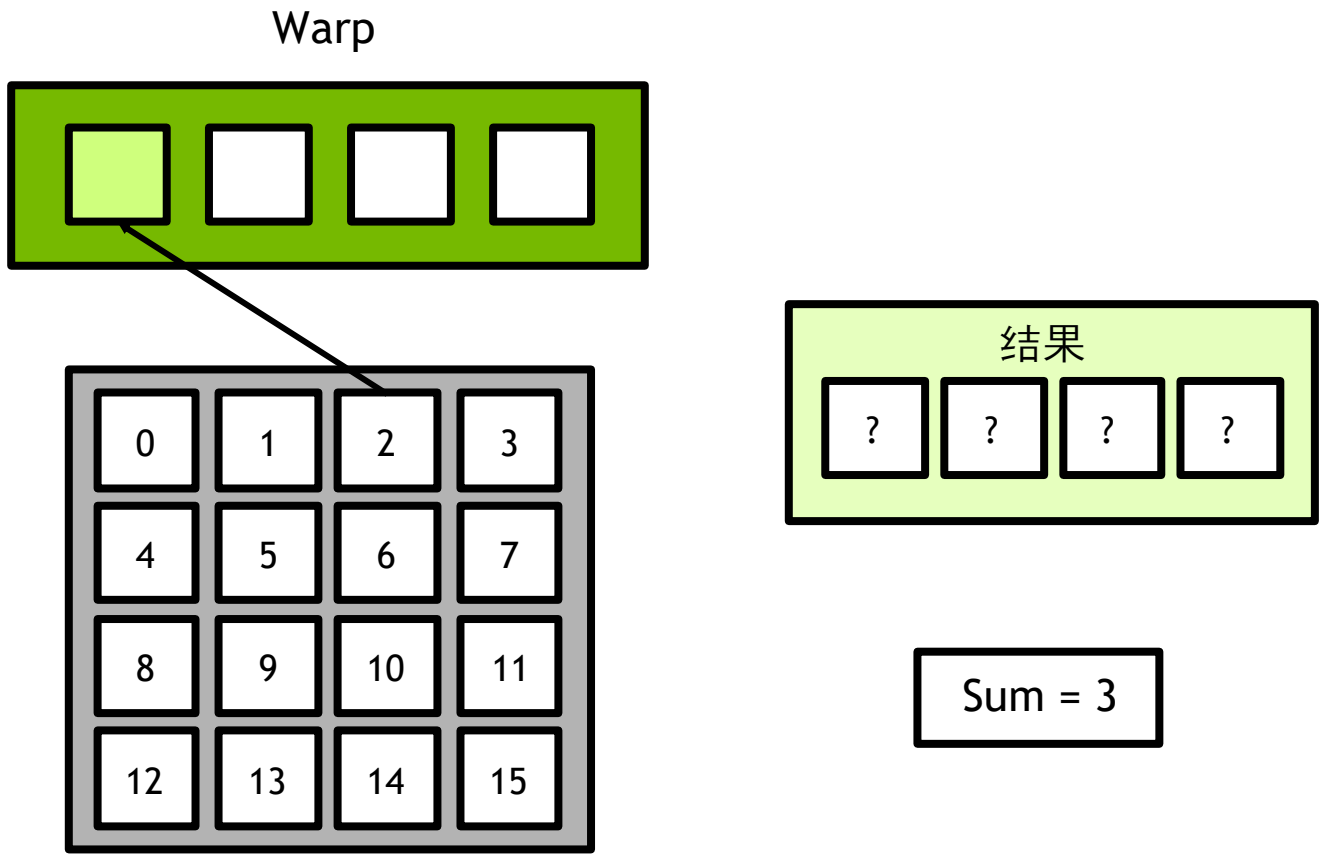
单个线程可以迭代一行，将其求和，然结果写入结果向量中。



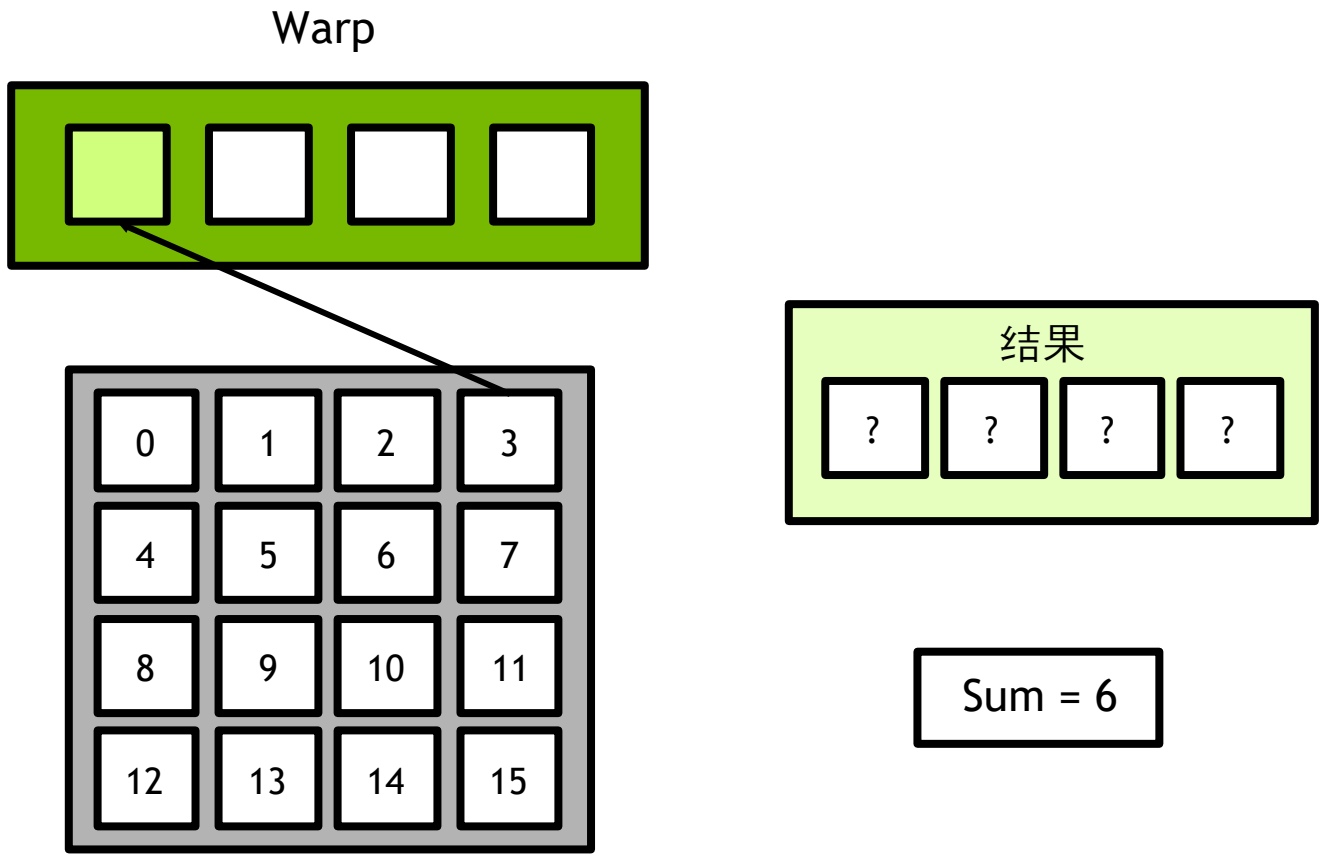
单个线程可以迭代一行，将其求和，然结果写入结果向量中。



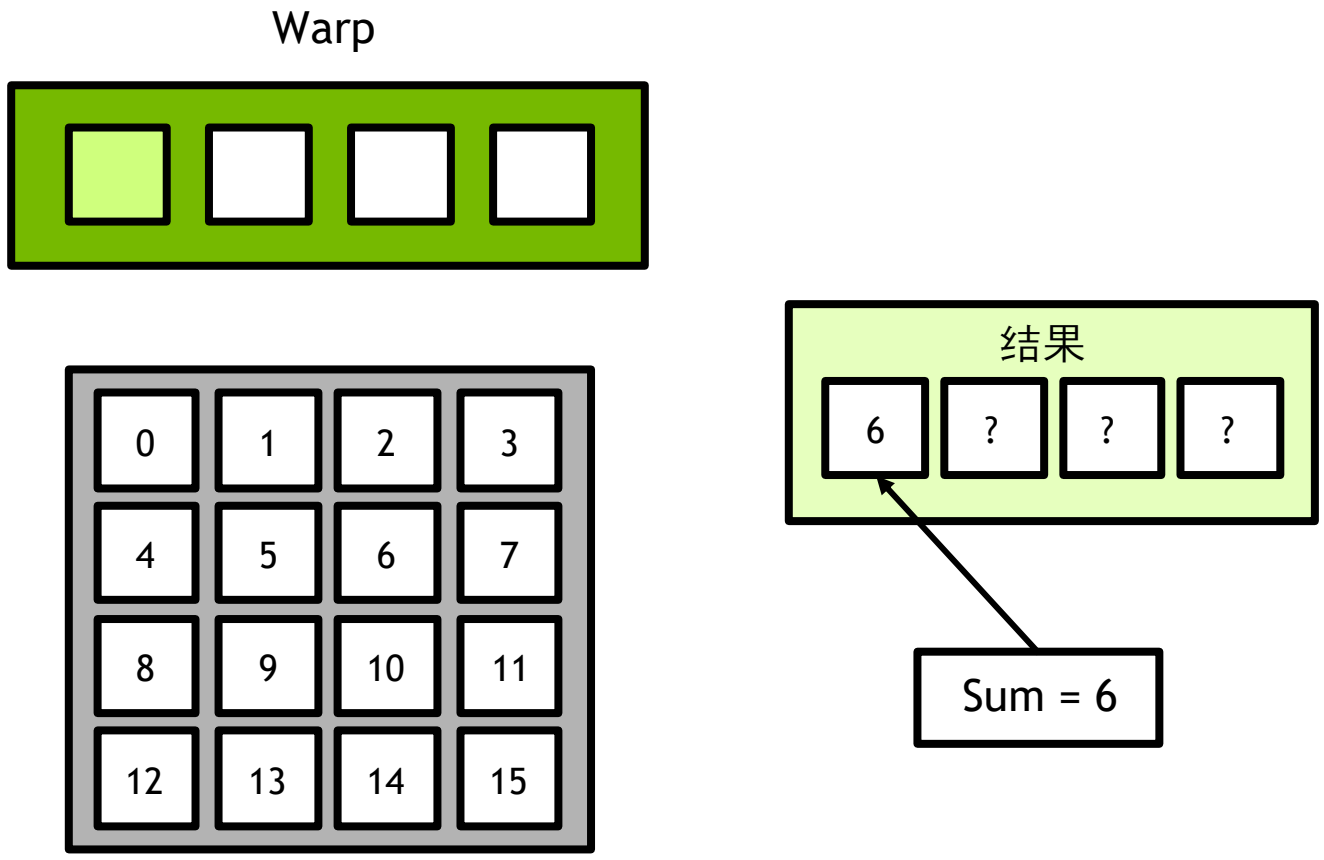
单个线程可以迭代一行，将其求和，然结果写入结果向量中。



单个线程可以迭代一行，将其求和，然结果写入结果向量中。

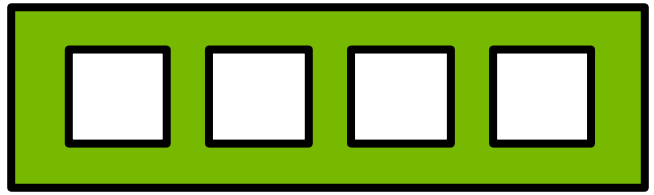


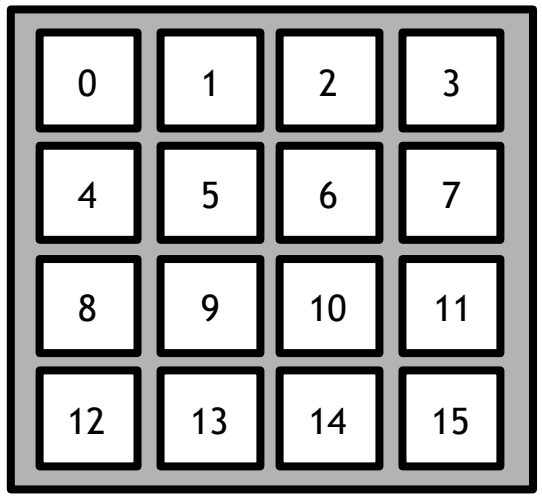
单个线程可以迭代一行，将其求和，然结果写入结果向量中。



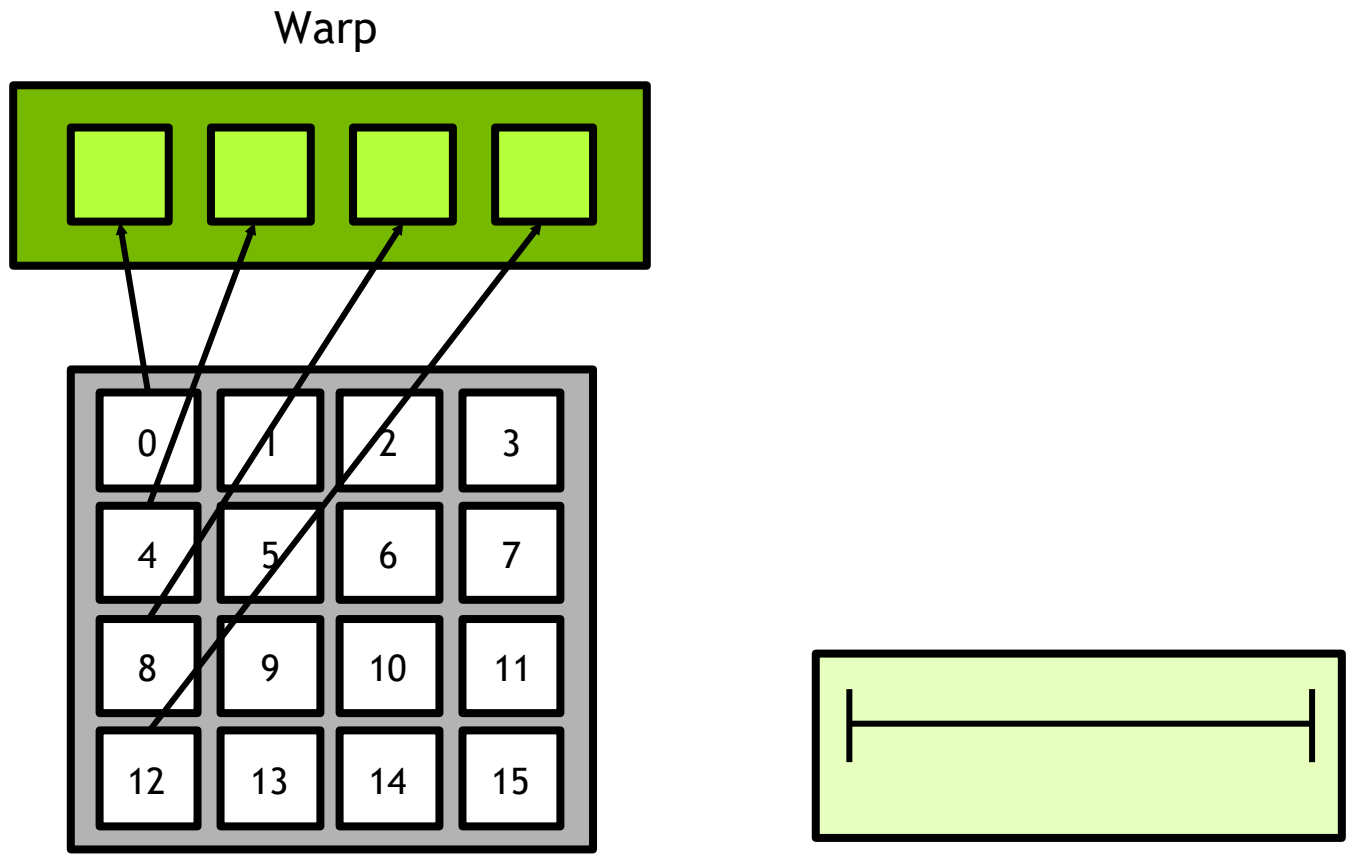
这看起来很自然，但是当我们考虑warp线程并行执行时会发生什么

Warp

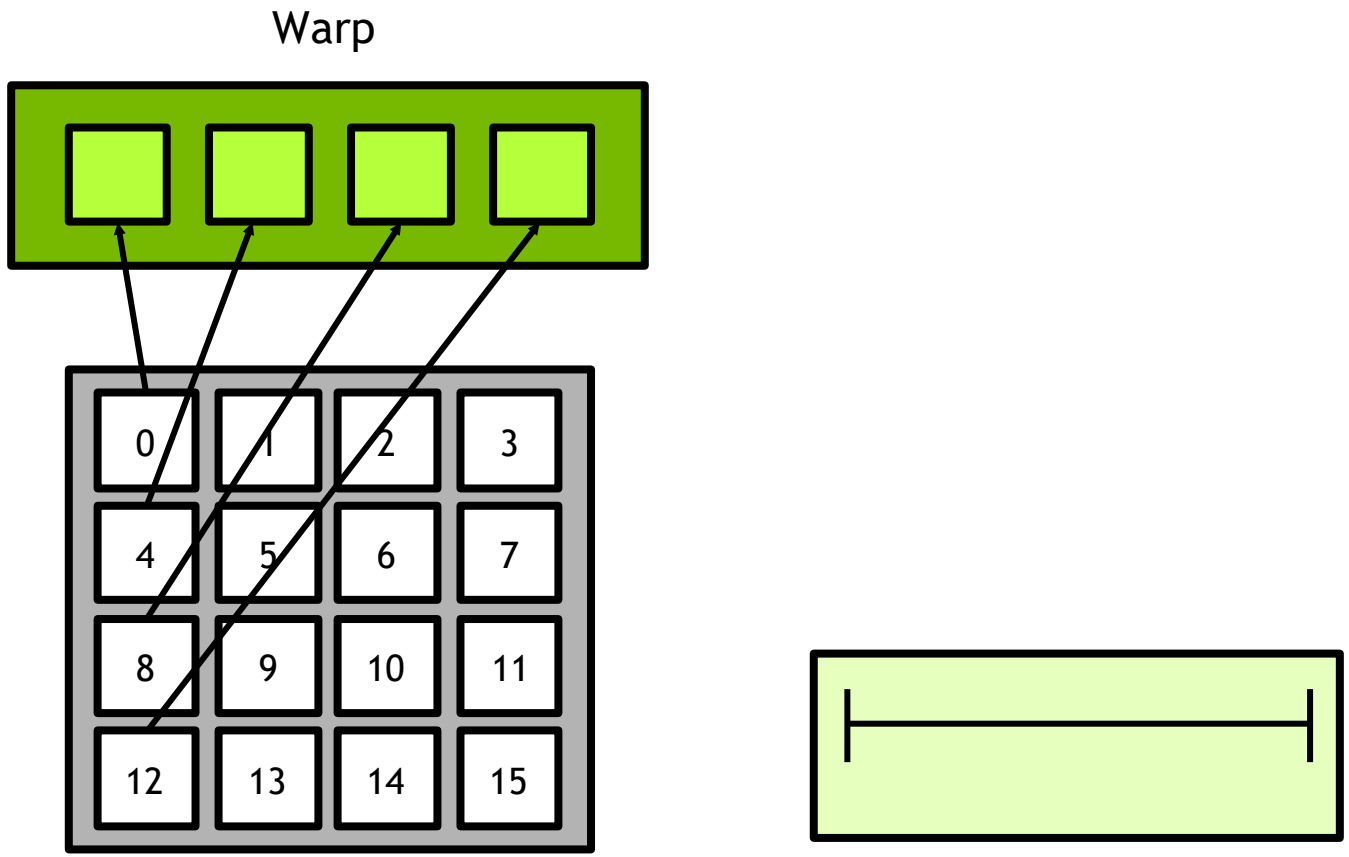




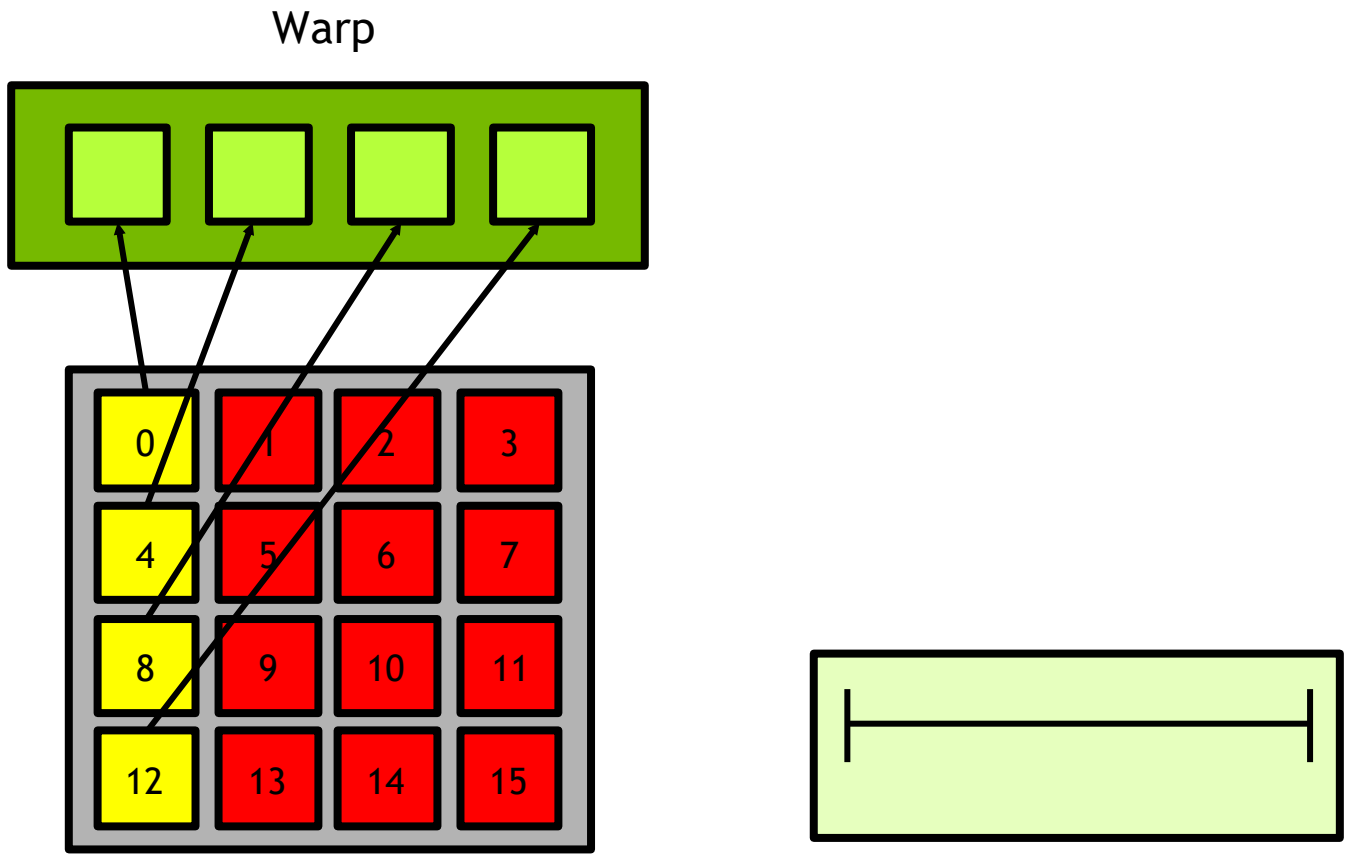
Warp 线程都在不同的内存行中请求数据



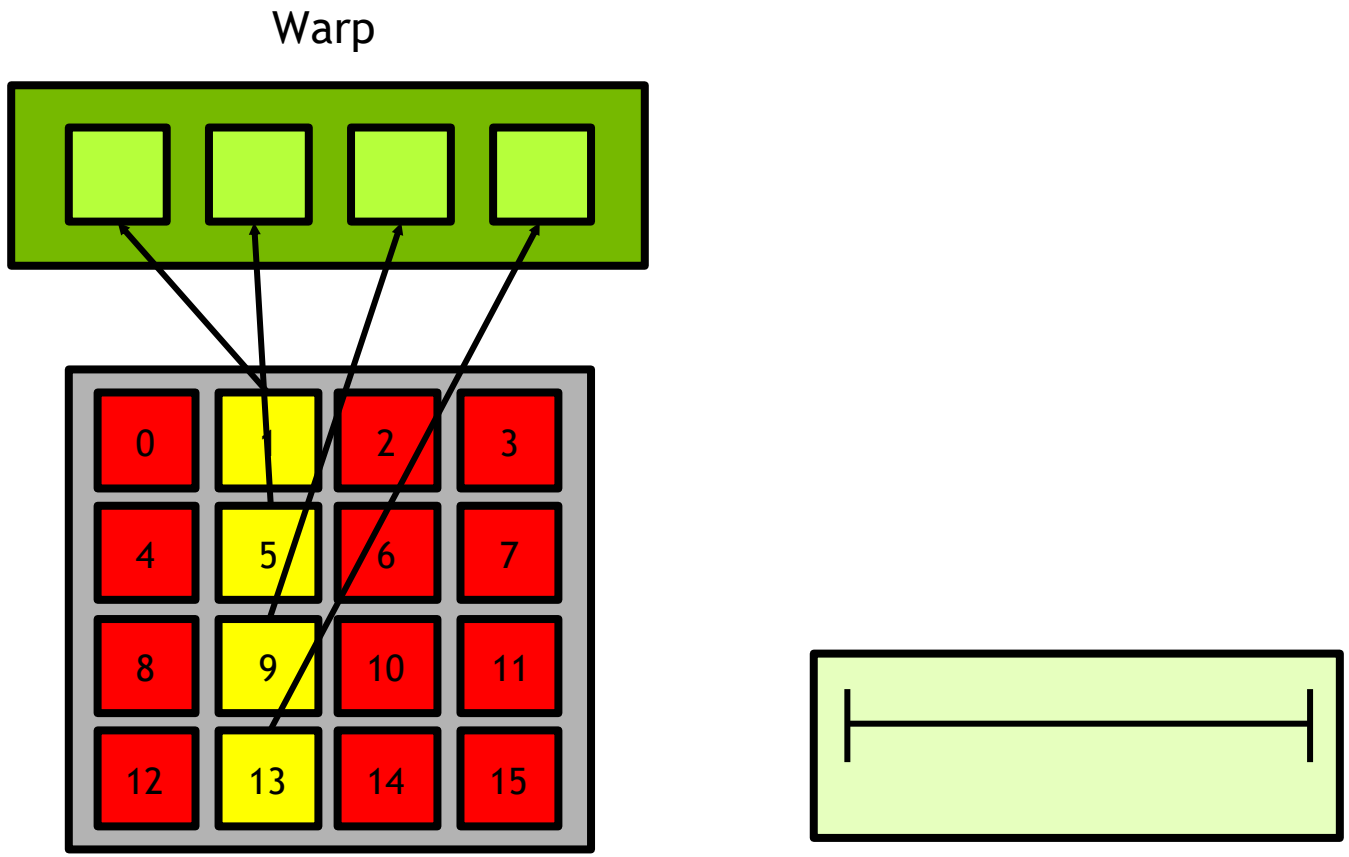
请注意，threadIdx.x 是 y 轴的数据增量



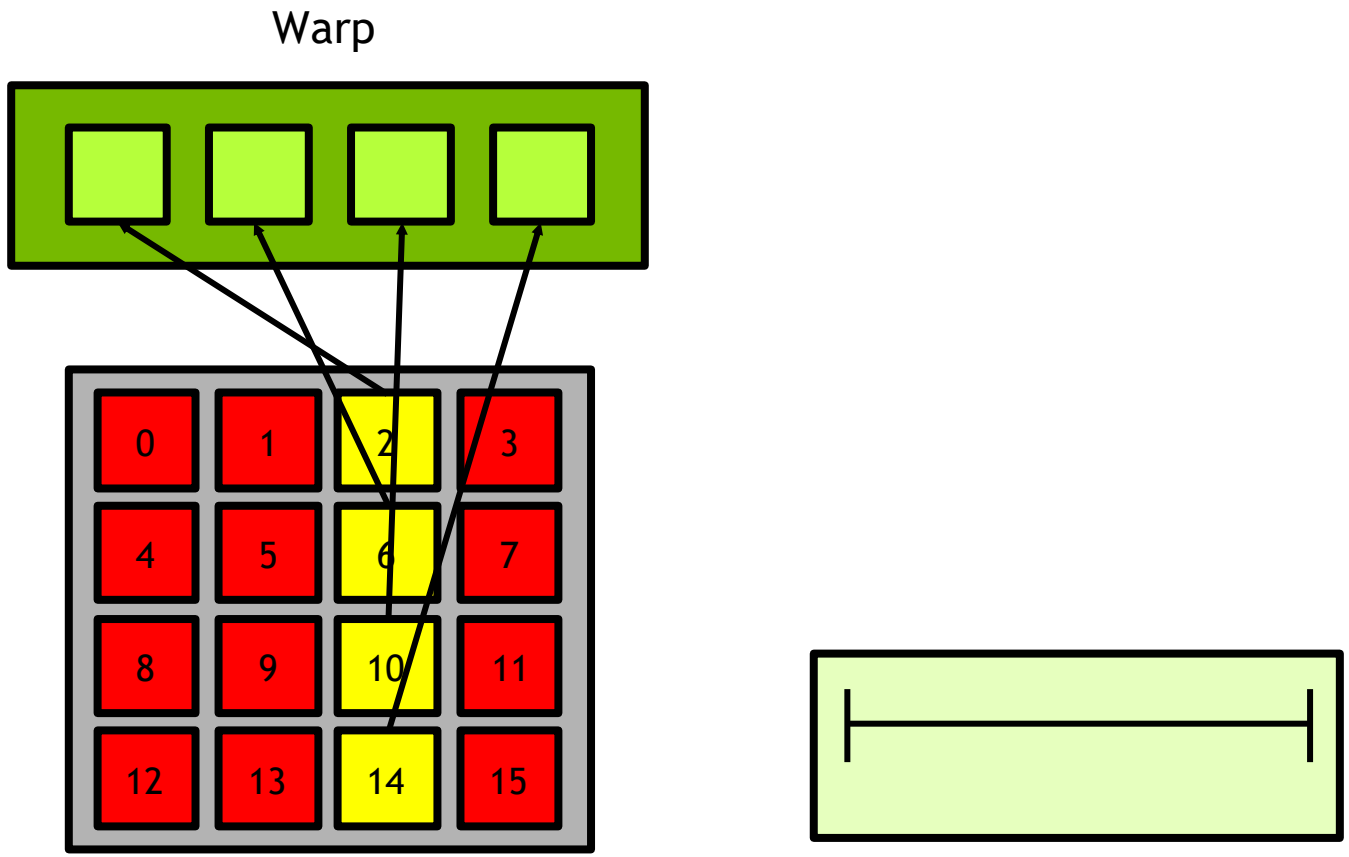
这意味着（在我们的示例中）需要加载
4 载的数据中有 75%



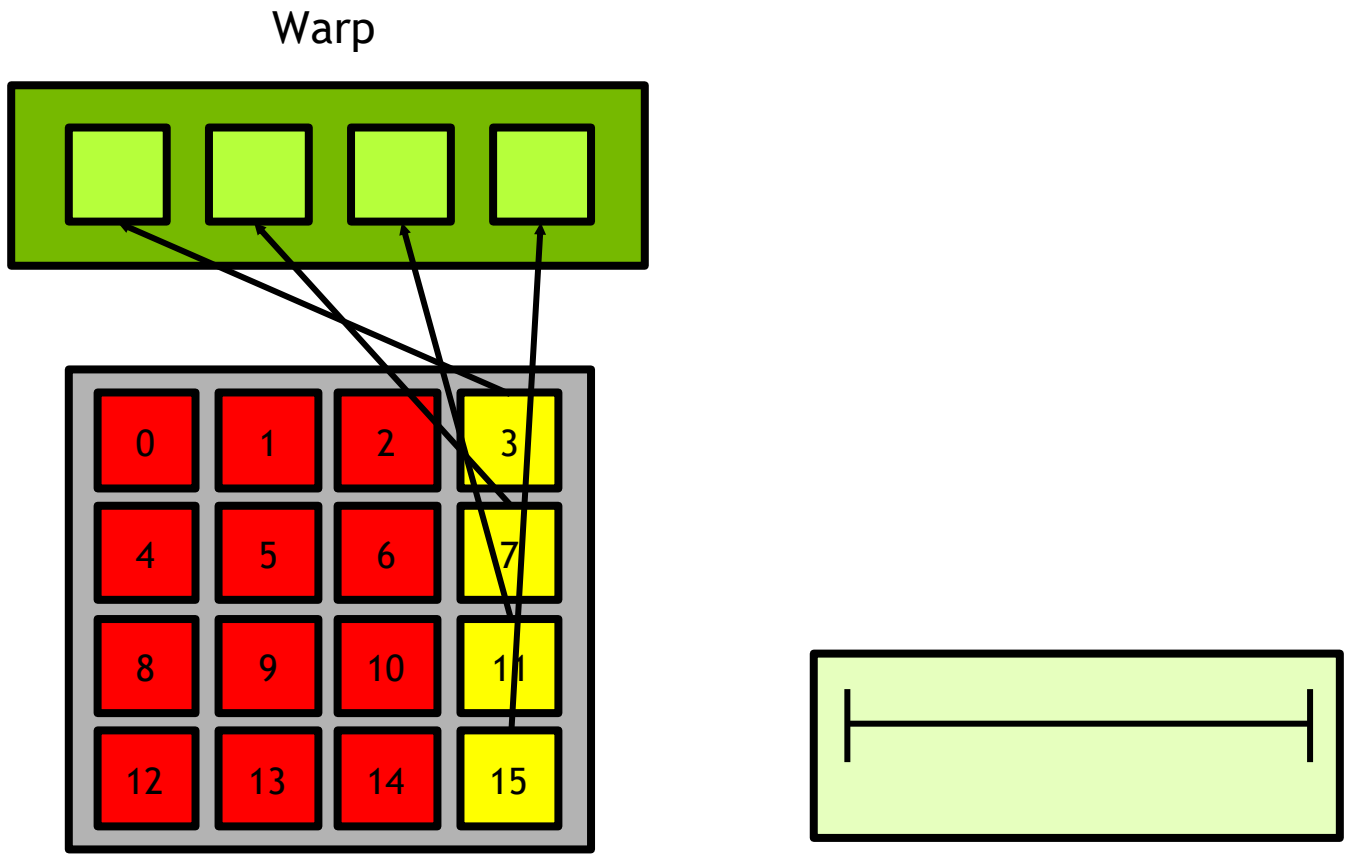
线程在它所在的数据时，相同的未合并模式仍在继续。



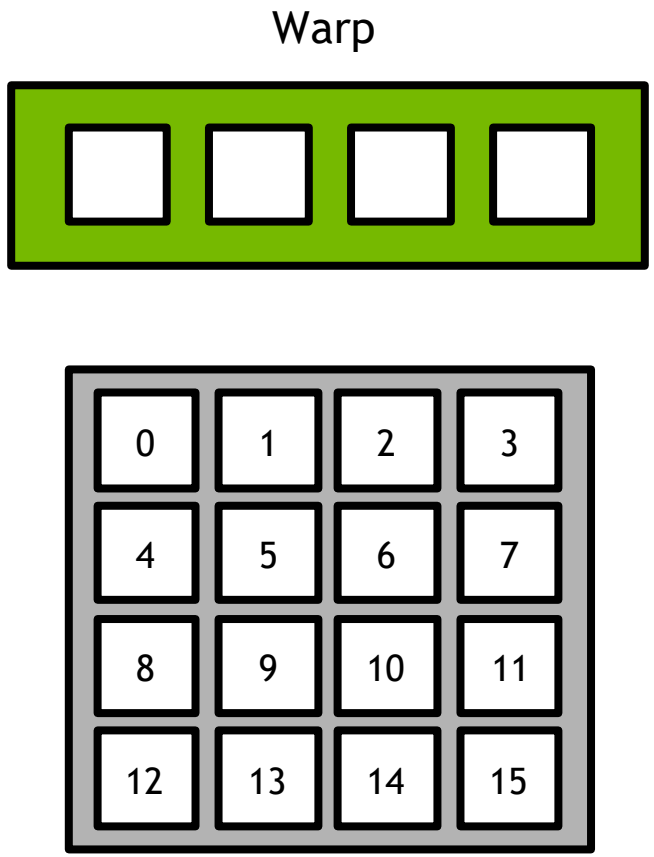
线程在它所在的数据时，相同的未合并模式仍在继续。



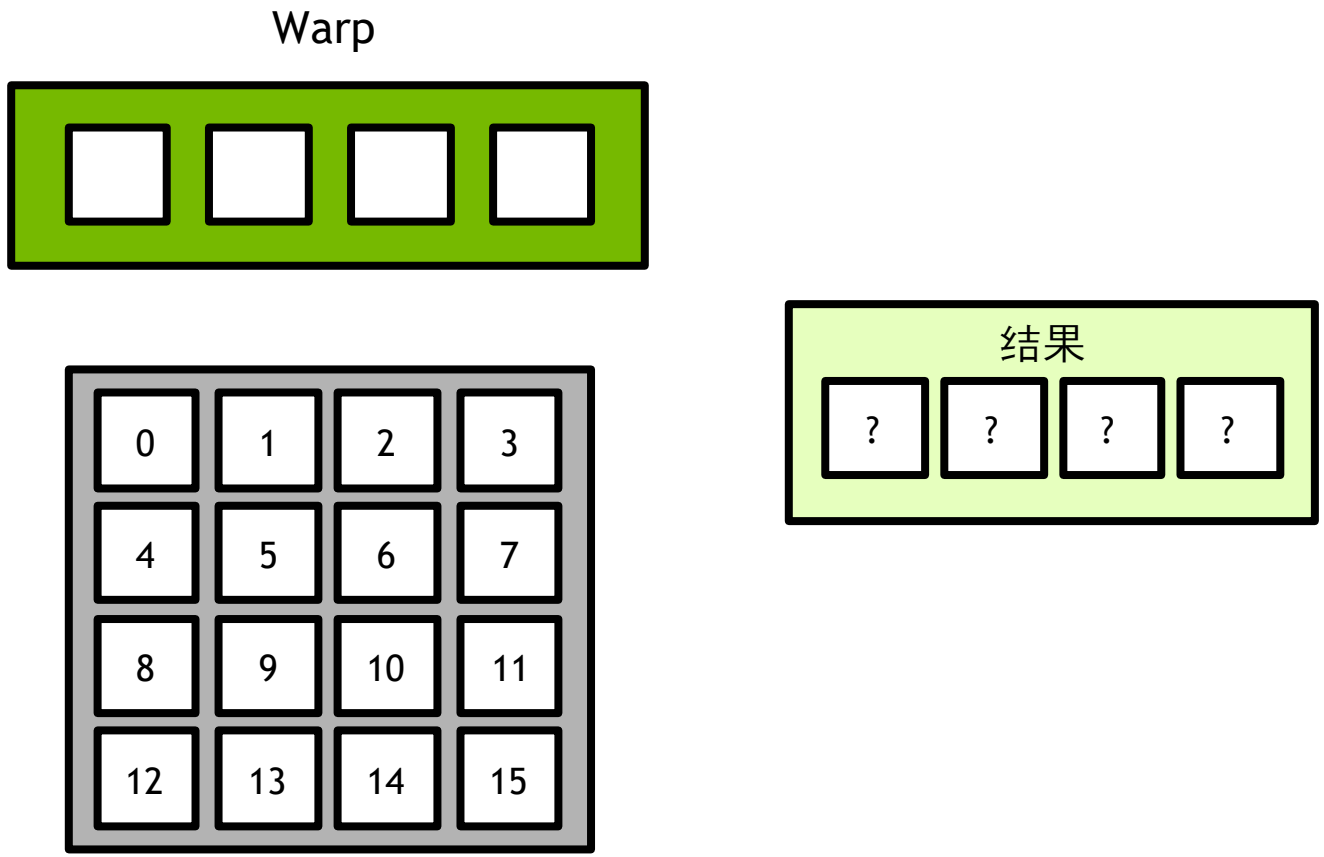
线程在它所在的数据时，相同的未合并模式仍在继续。



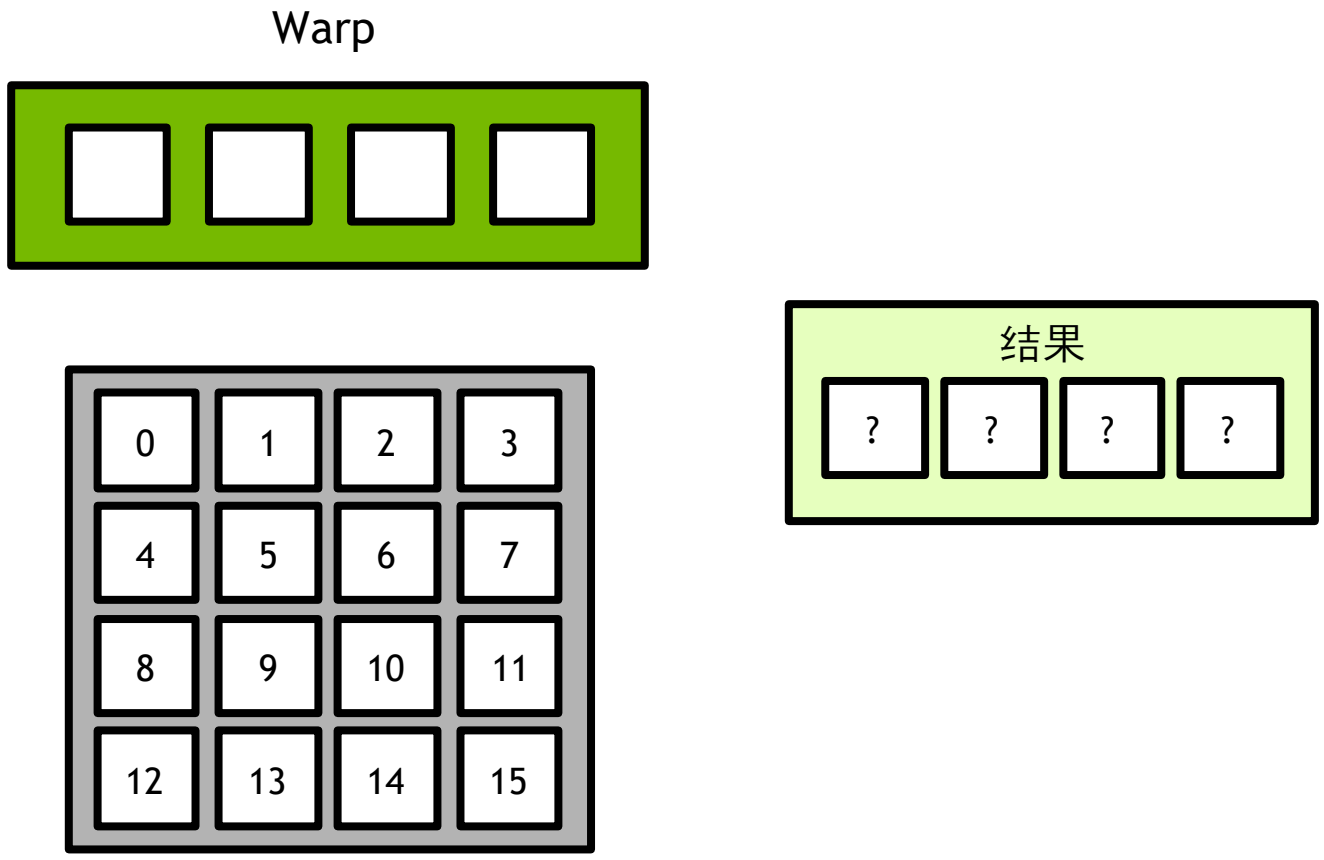
这个例子中，我们传输了 16
传输行使用了 25%



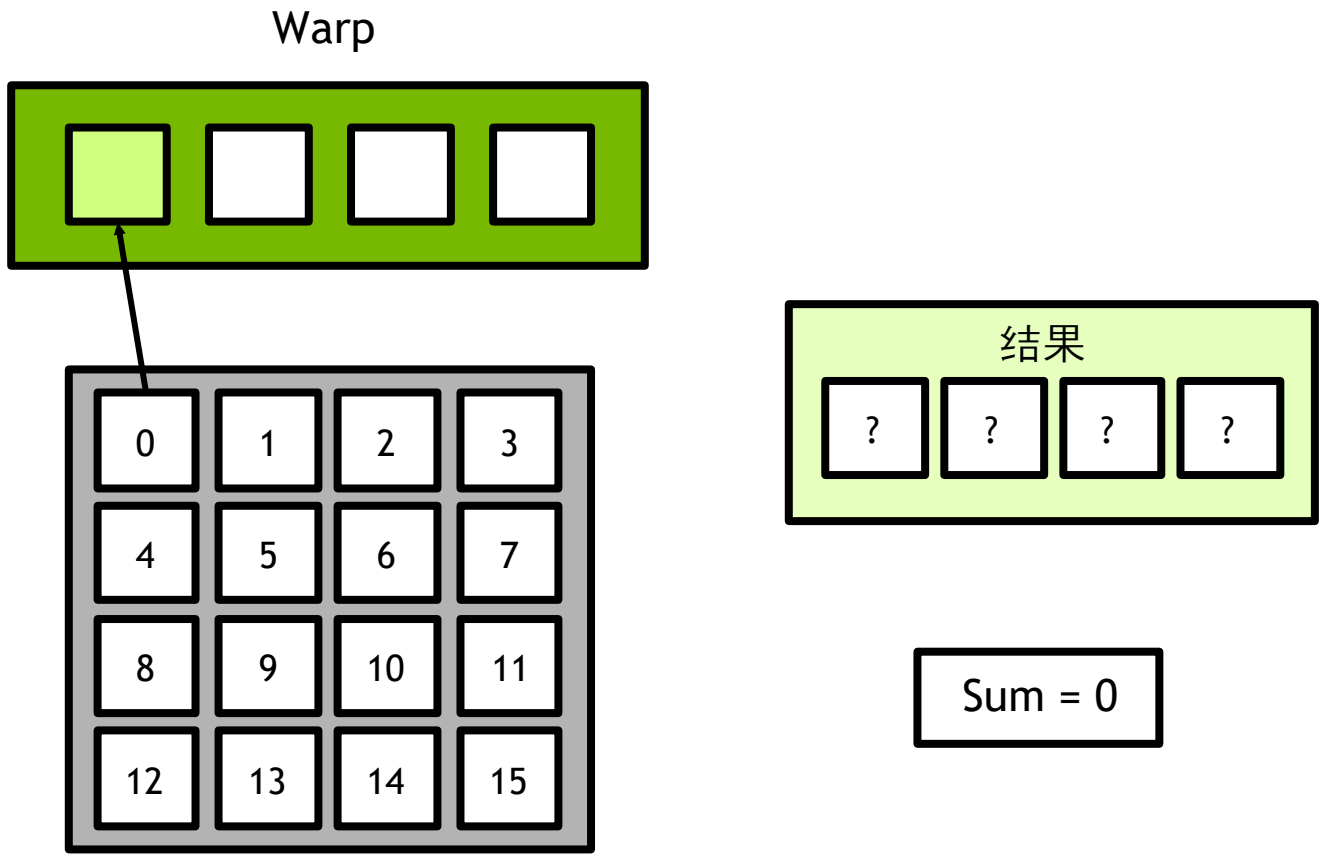
让我们比较一个将矩阵的每列之和存储结果向量中的核函数



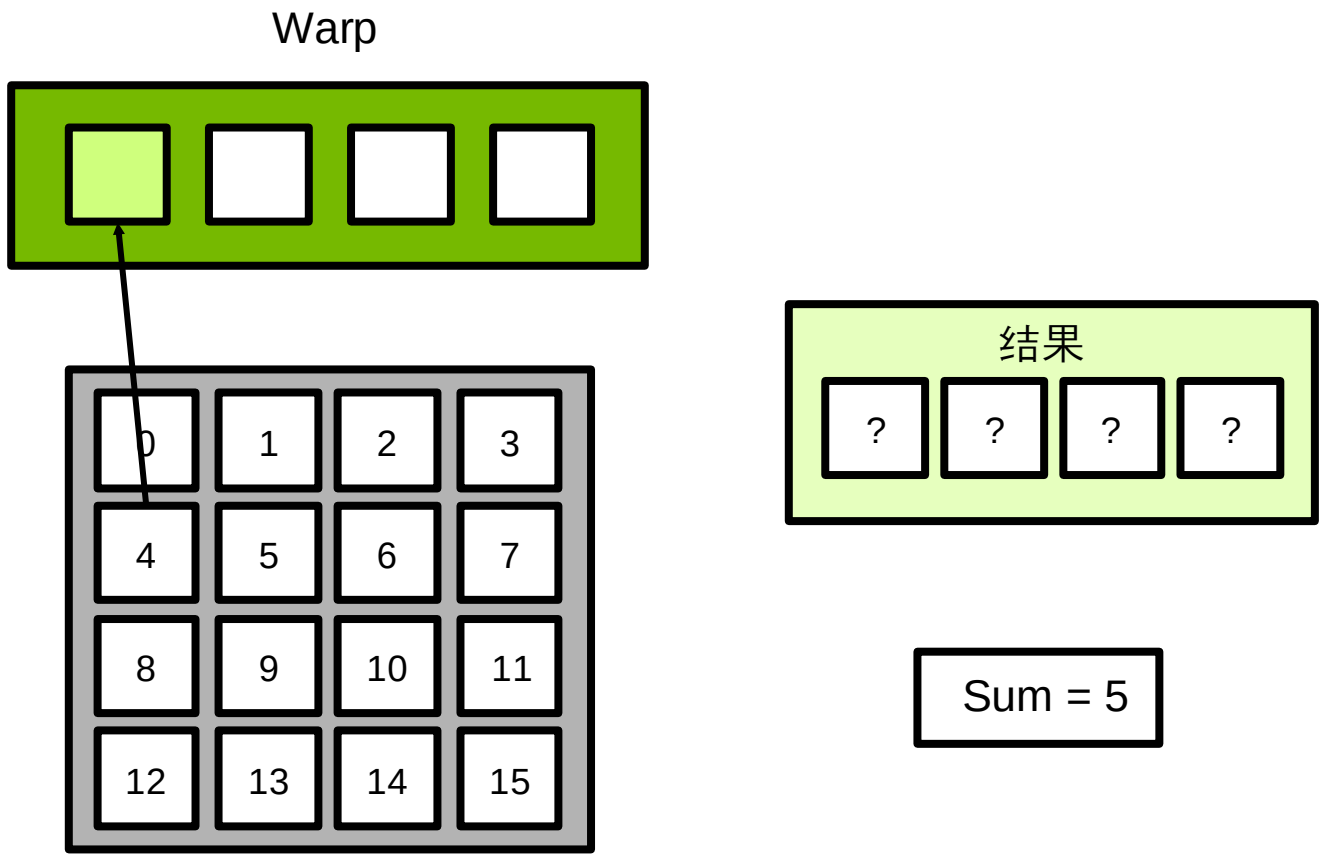
单个线程可以遍历一列，求和，然后将结果写入结果向量



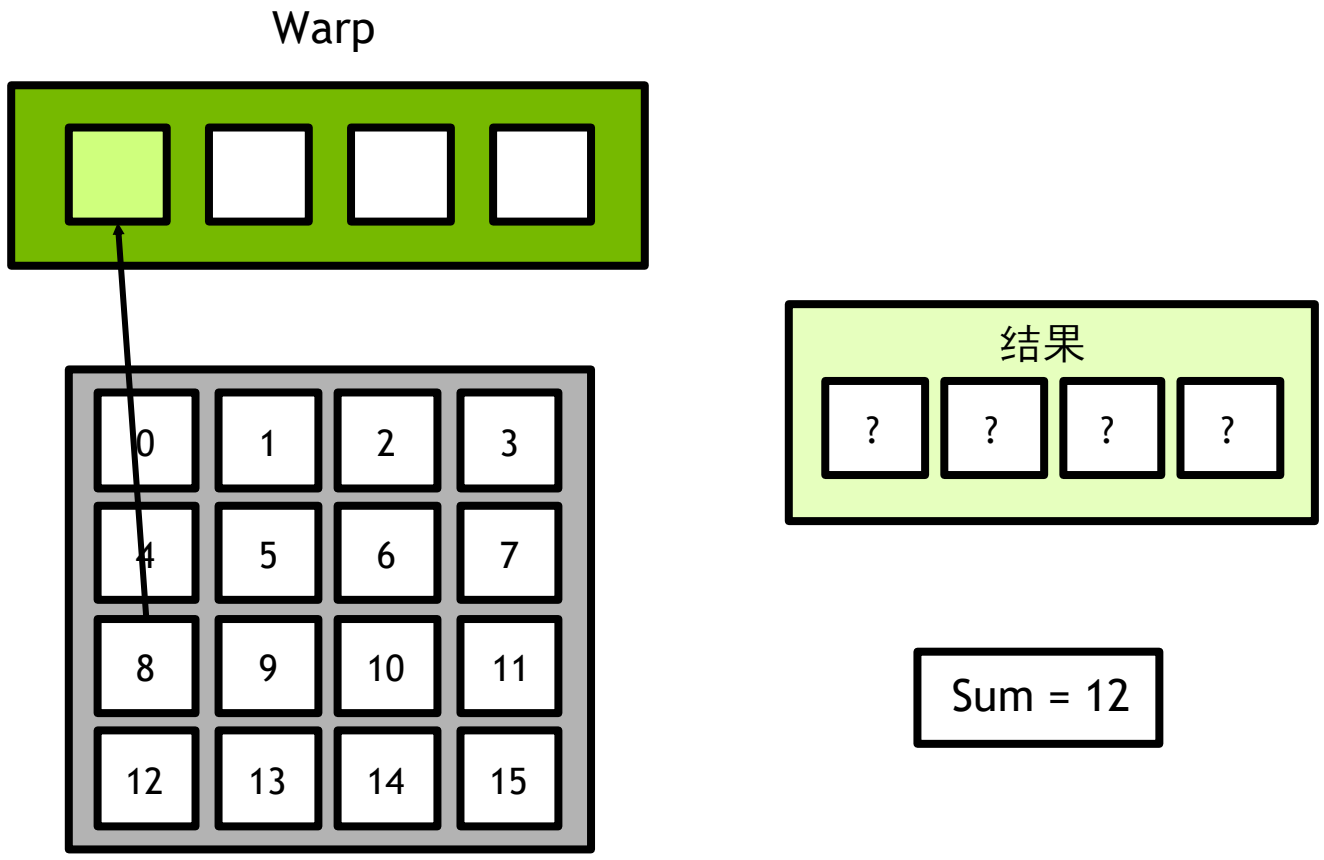
单个线程可以迭代一列，将其求和，然结果写入解向量中。



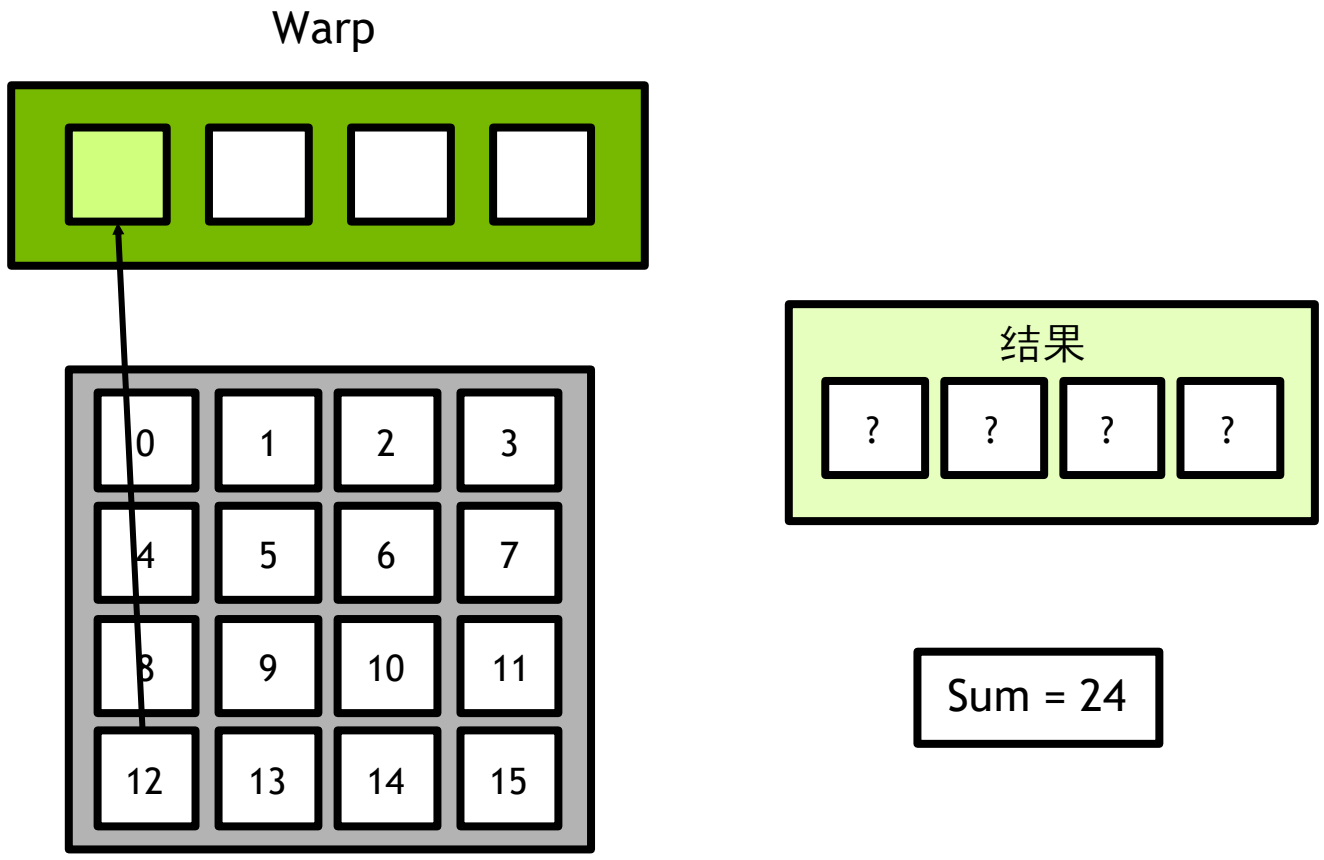
单个线程可以迭代一列，将其求和，然结果写入解向量中。



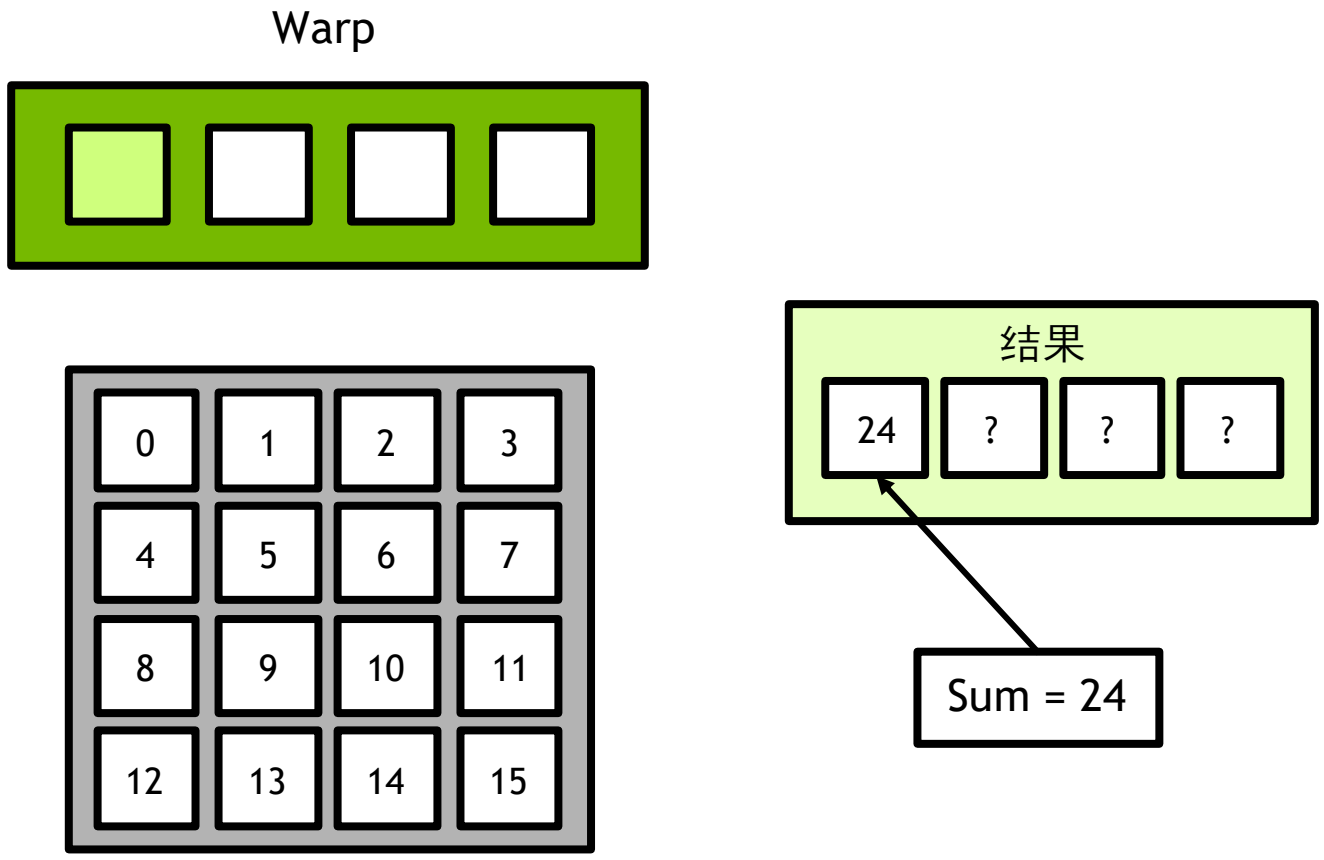
单个线程可以迭代一列，将其求和，然结果写入解向量中。



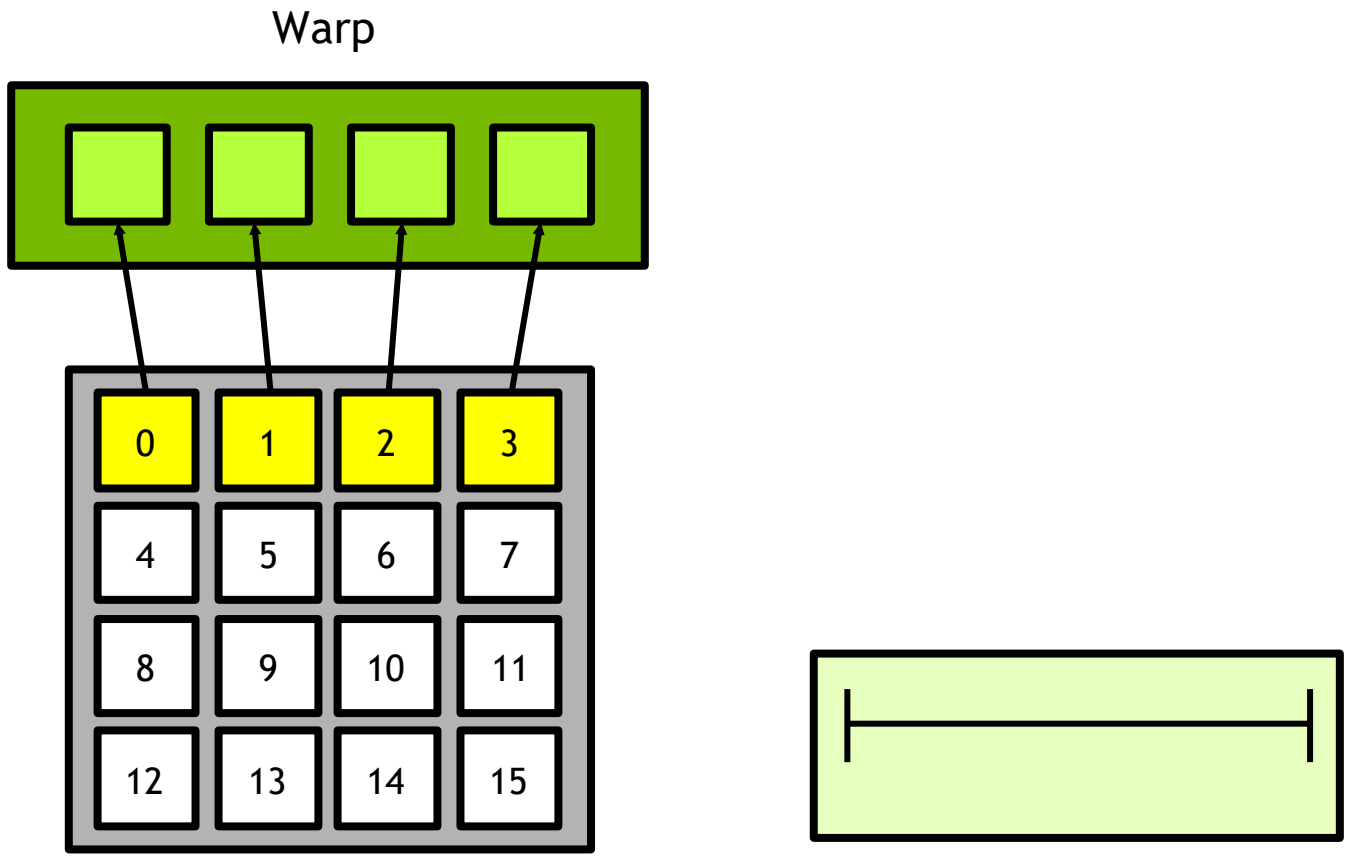
单个线程可以迭代一列，将其求和，然结果写入解向量中。



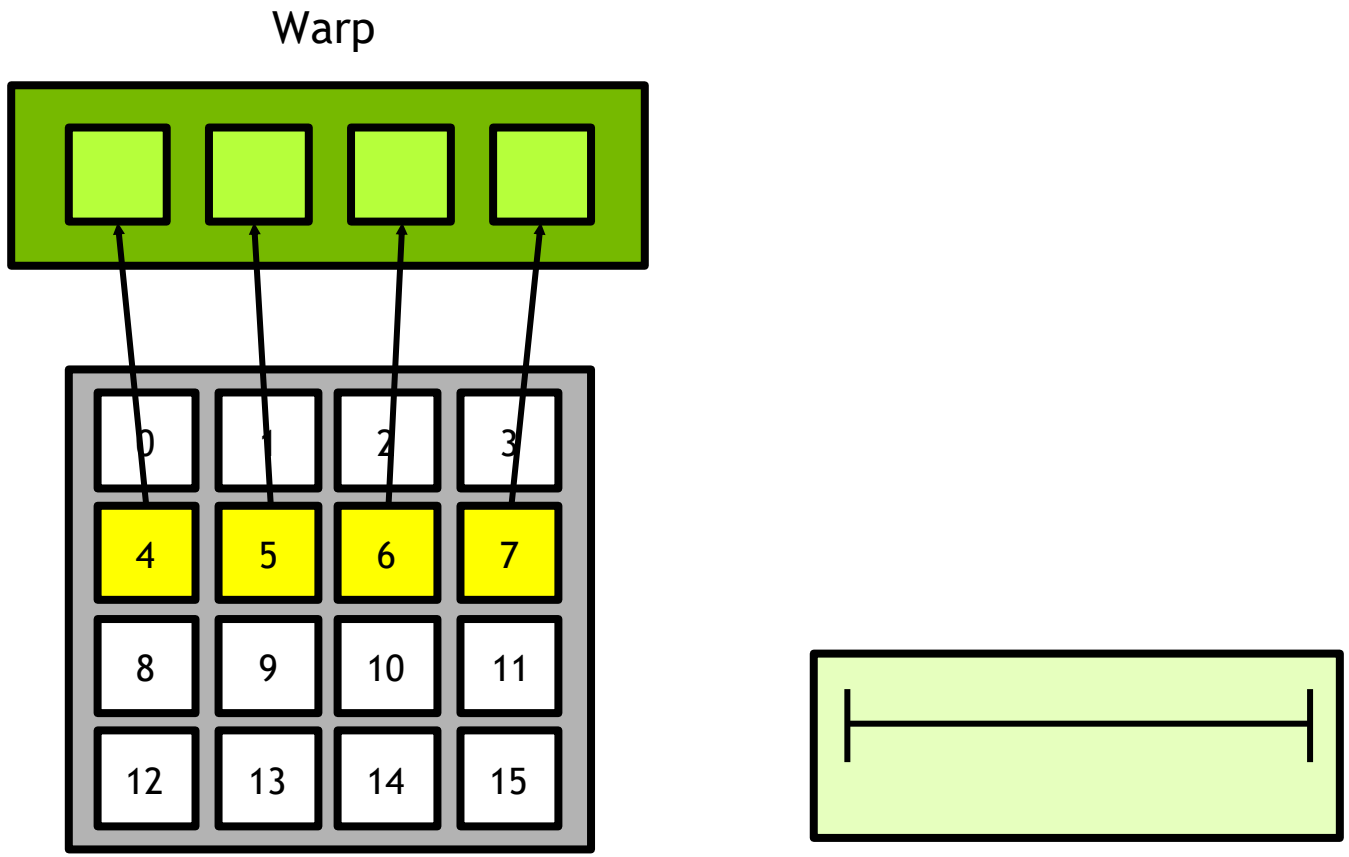
单个线程可以迭代一列，将其求和，然结果写入解向量中。



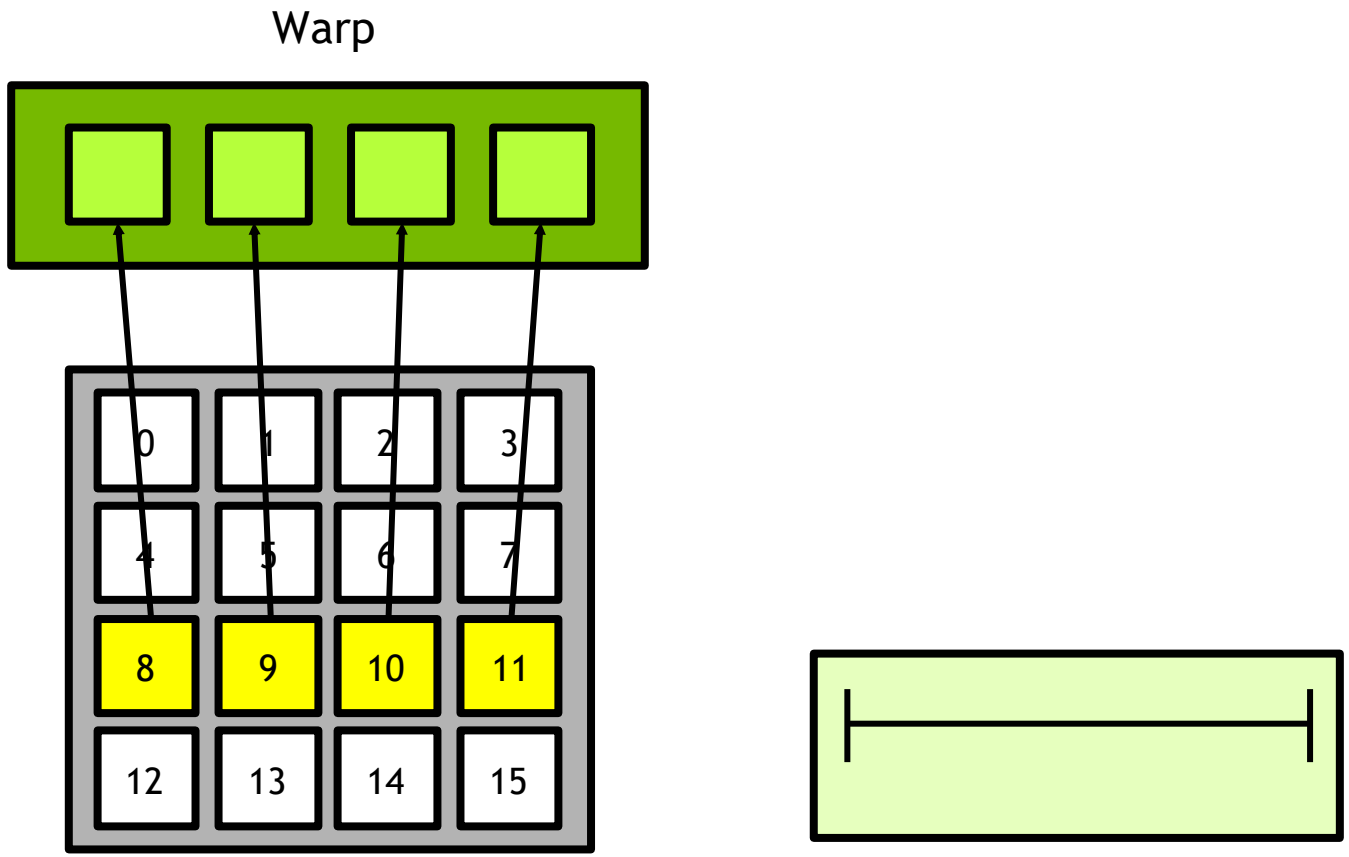
这里，当我们考虑并行执行时，我们
warp 访问是合并的。



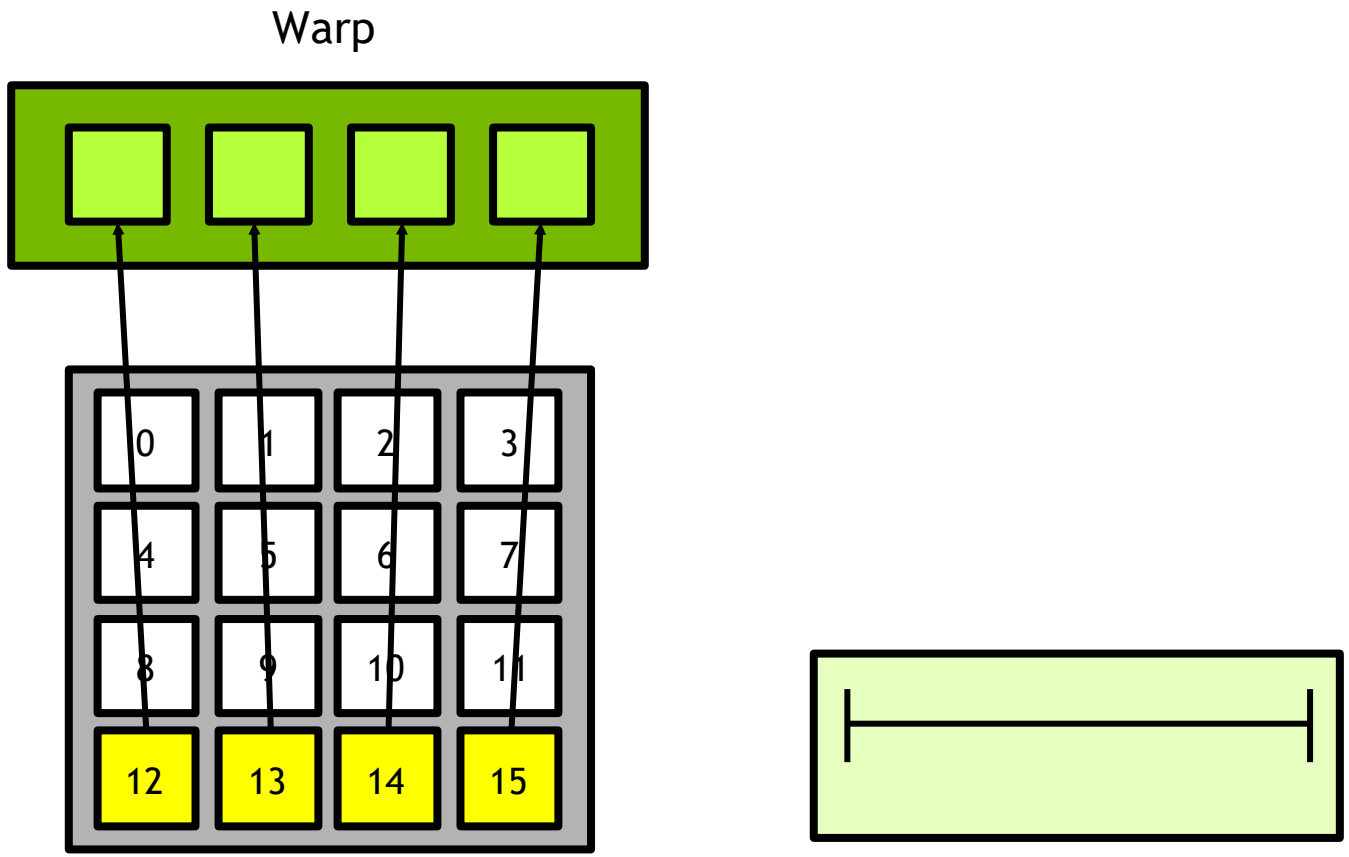
这里，当我们考虑并行执行时，我们
warp 访问是合并的。



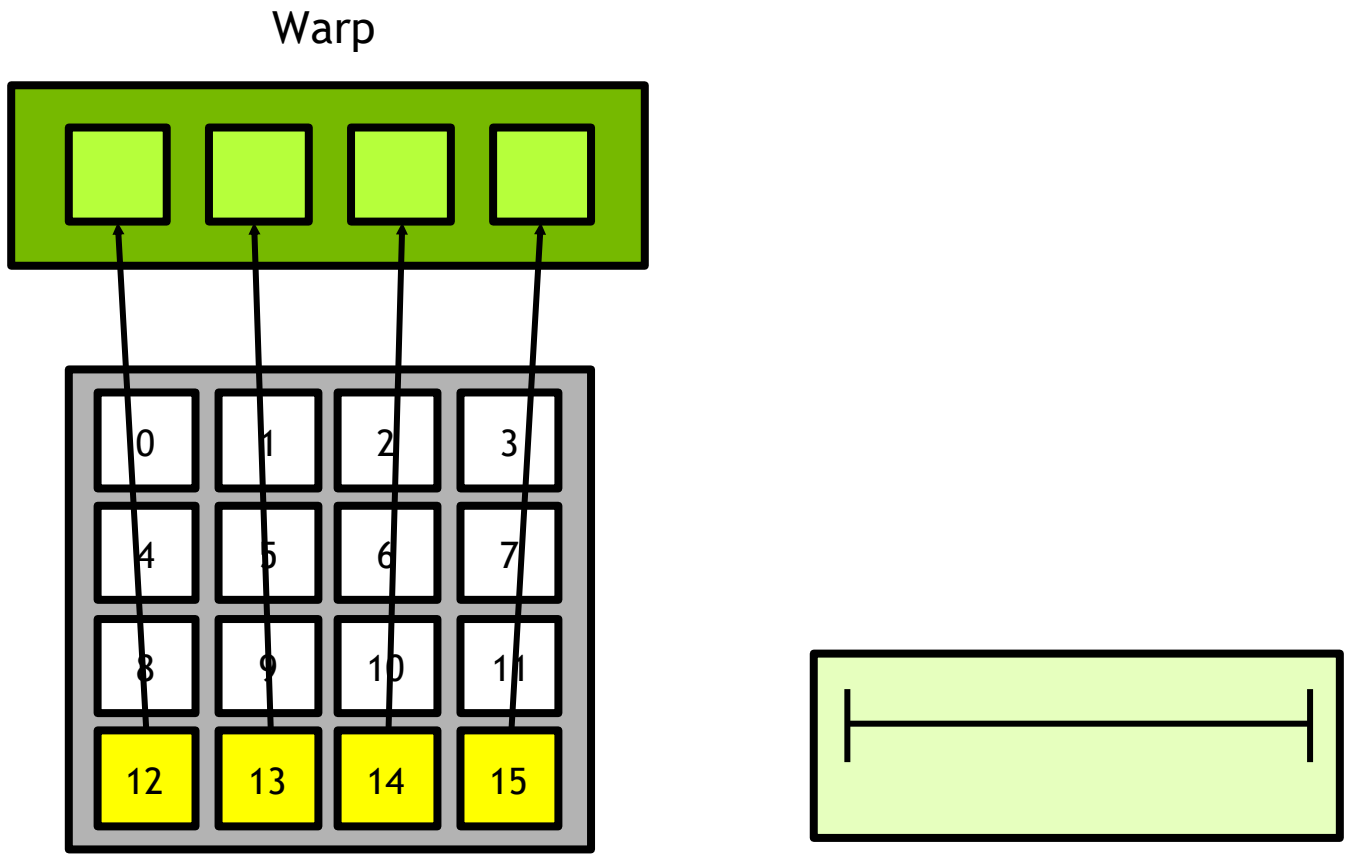
这里，当我们考虑并行执行时，我们
warp 访问是合并的。



这里，当我们考虑并行执行时，我们
warp 访问是合并的。

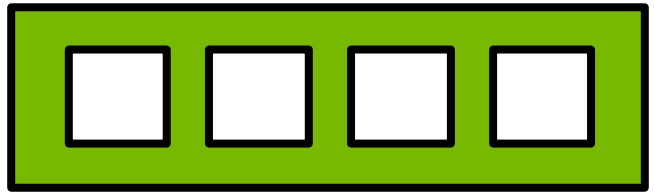


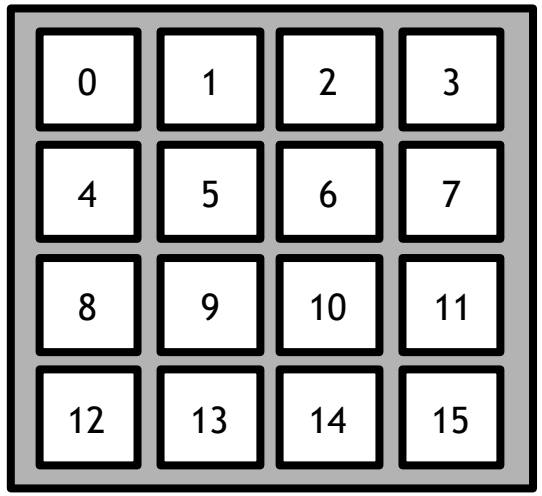
记住的一个有用提示是，threadIdx.x 应该映射到数据增量变化最快的 – 这个例子中是 x 轴。



们传输了 4
16 传输行使用了
100% 25%

Warp





NVIDIA | DEEP LEARNING INSTITUTE

习更多课程，请访问 www.nvidia.cn/DLI