CUDA高性能科学计算

2023-2024-1 秋季

公选-06110

Lecture 6

理学院 赖欣

授课平台/讨论QQ群

CUDA高性能科学计算(GX)
课程编号:107016

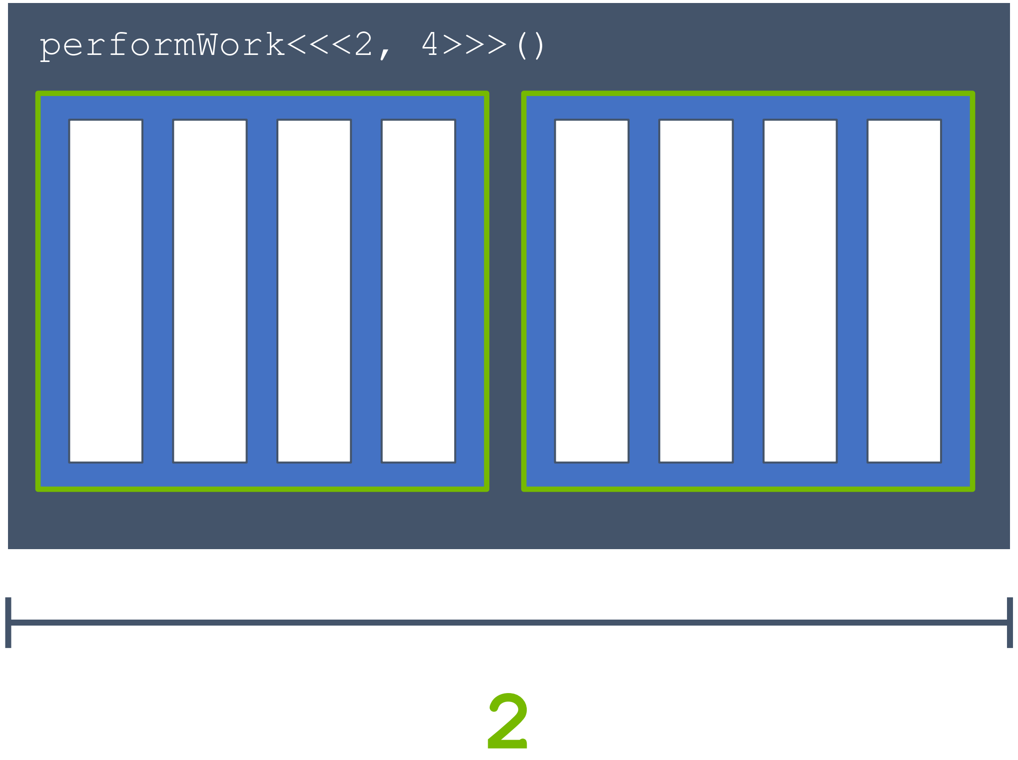
加速计算基础——CUDA C/C++

8 学时 | 中文 | 90 美元 | C/C++，CUDA ®

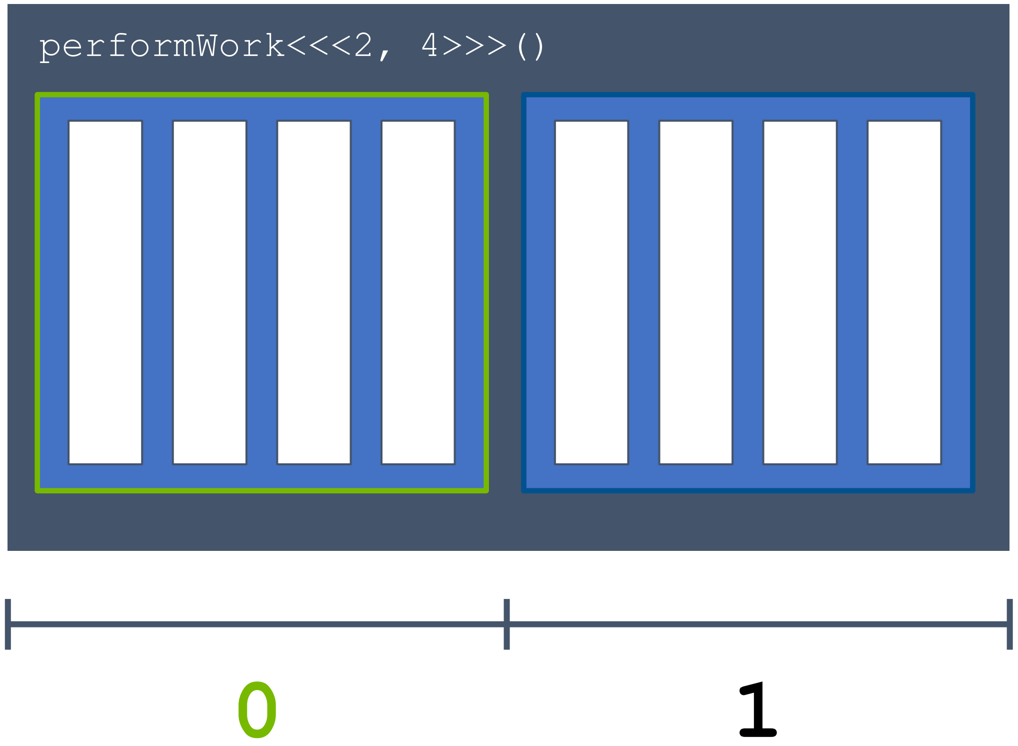
有培训证书

CUDA 提供的线程层次结构变量

CUDA 提供的线程层次结构变量

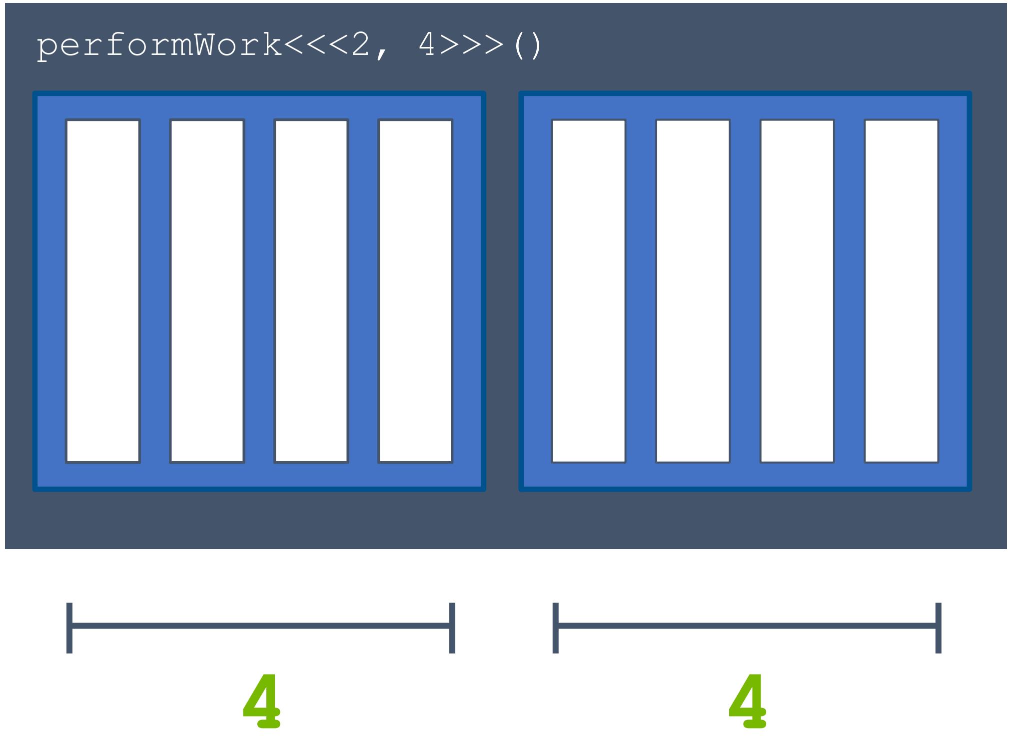


gridDim.x

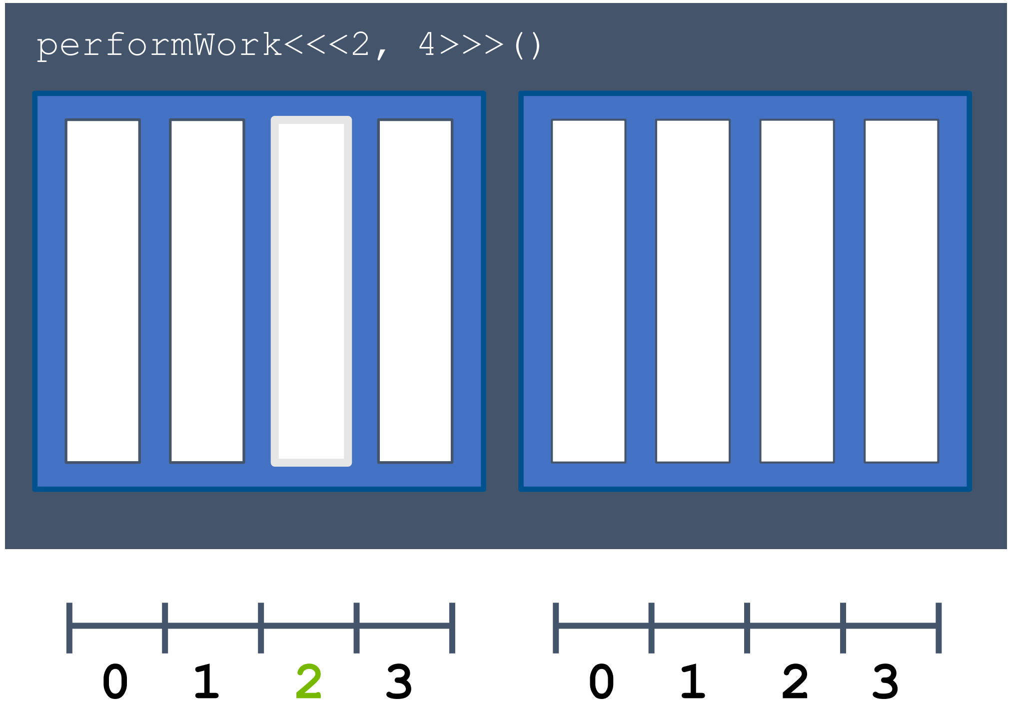


blockIdx.x

CUDA 提供的线程层次结构变量



blockDim.x

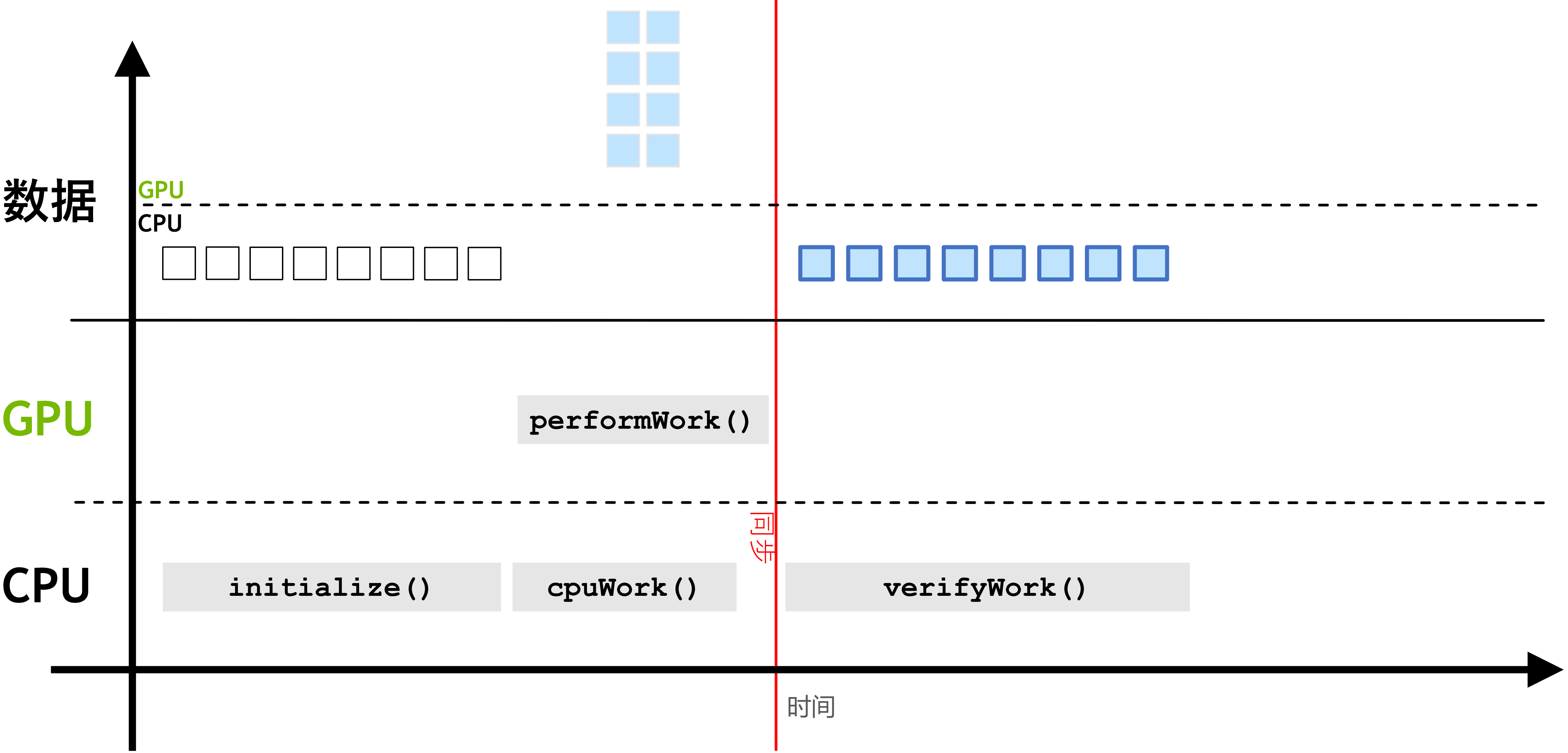


threadIdx.x

blockIdx.x*blockDim.x + threadIdx.x

协调并行线程

协调并行线程

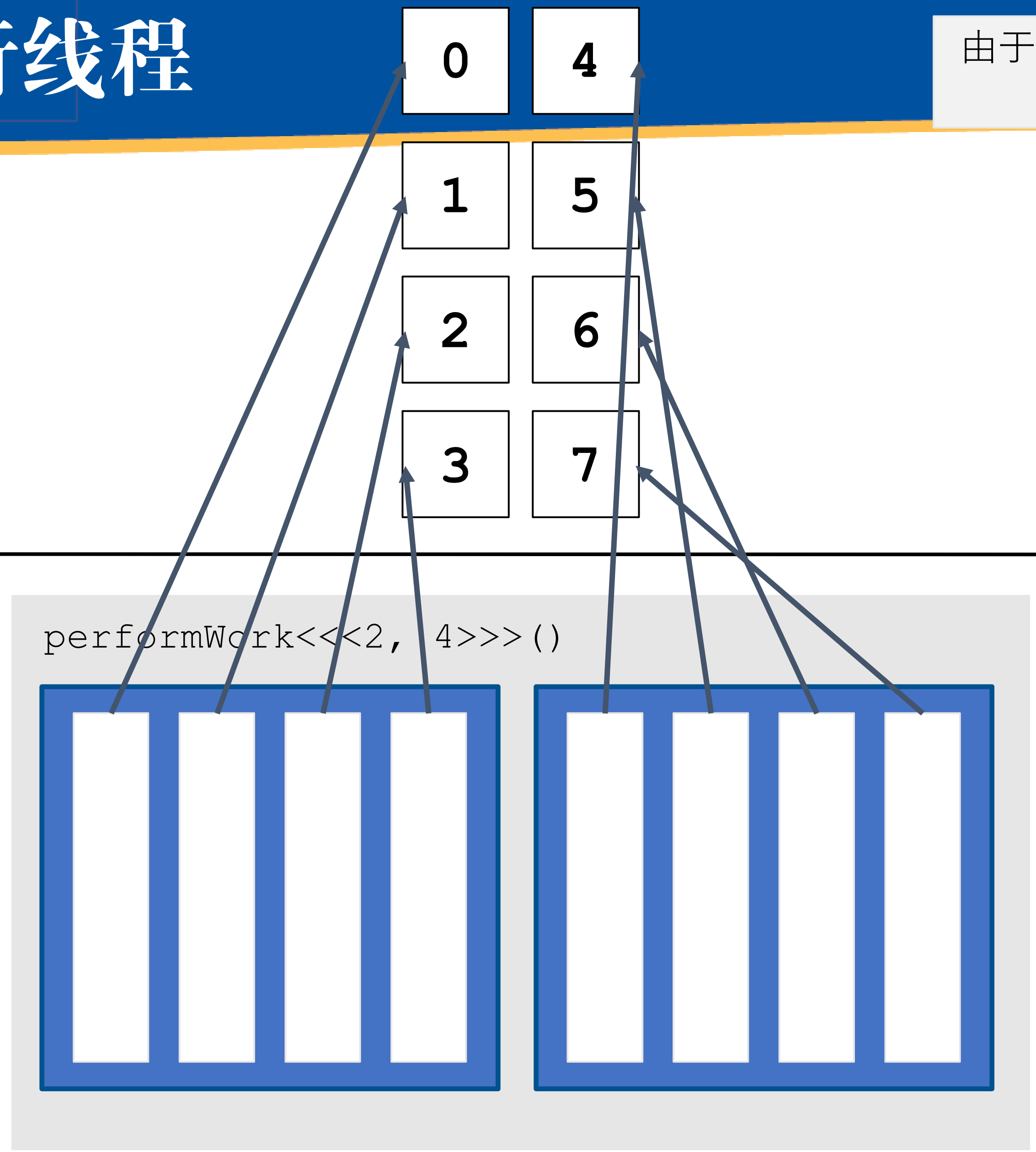


数据
GPU
CPU
GPU
performWork()
CPU
initialize()
cpuWork()
同步
verifyWork()
时间

协调并行线程

由于某种未知原因，必须映射每个线程以处理向量中的元素

GPU
数据

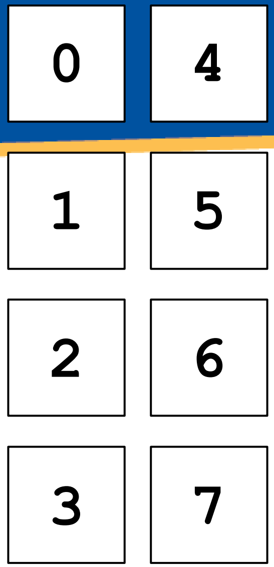


GPU

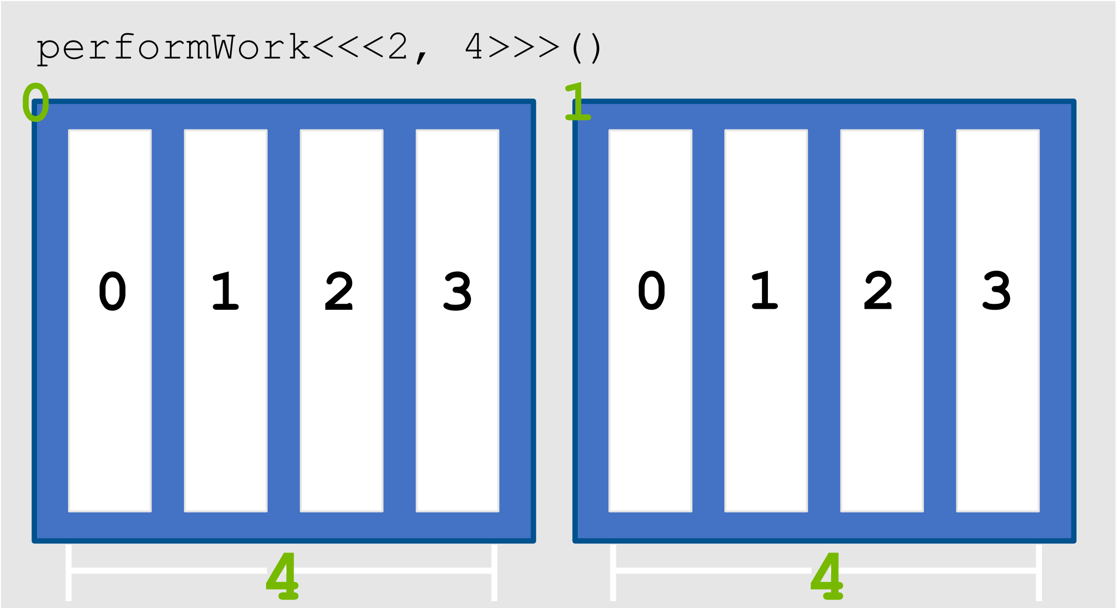
协调并行线程

通过这些变量，公式 **threadIdx.x + blockIdx.x * blockDim.x** 可将每个线程映射到向量的元素中

GPU
数据



GPU



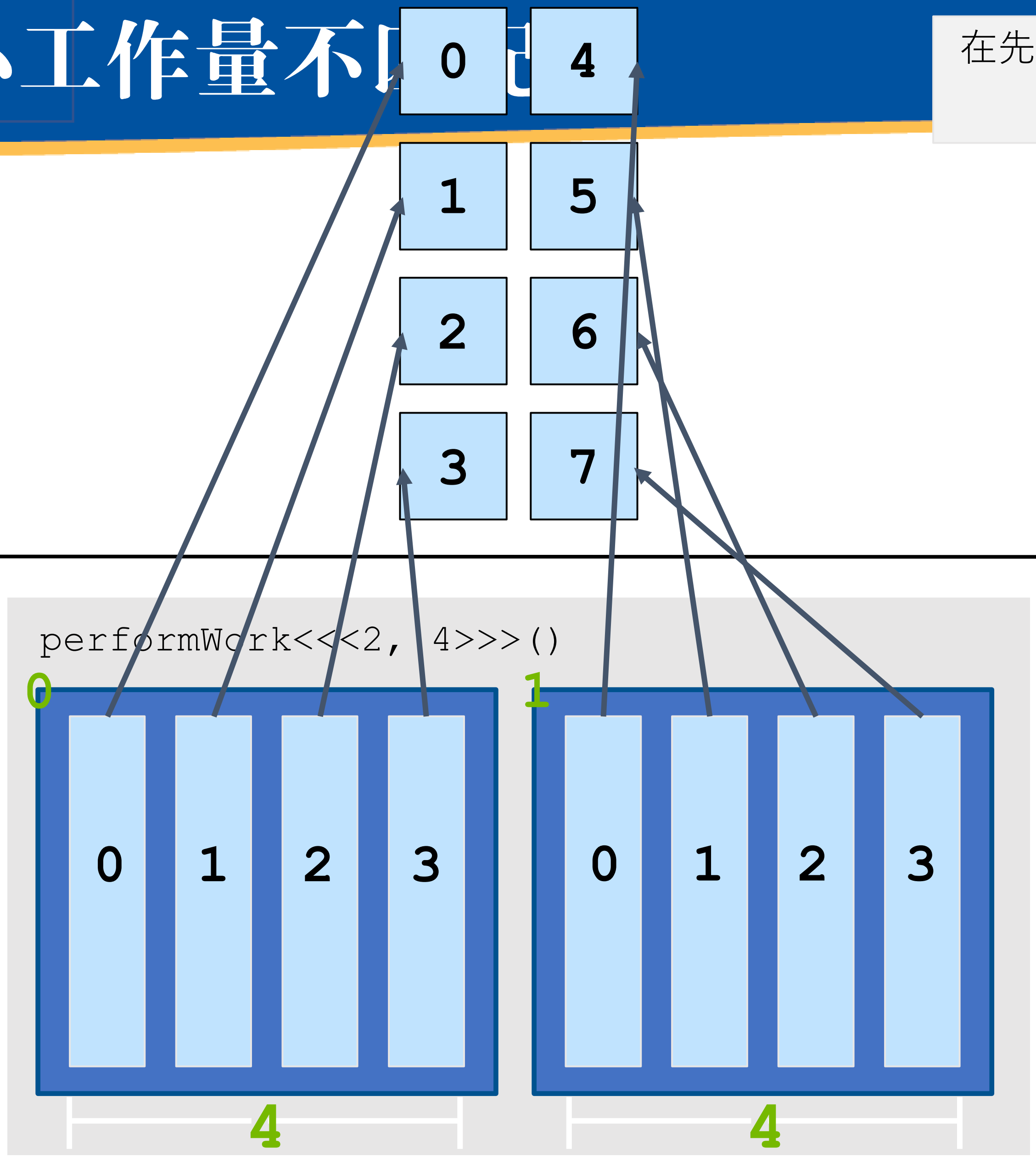
网格大小工作量不匹配

网格大小工作量不匹配

在先前场景中，网络中的线程数与元素数量完全匹配

GPU
数据

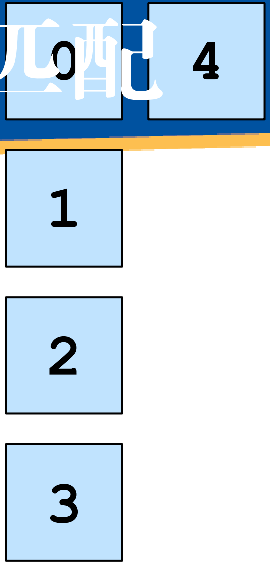
GPU



网格大小工作量不匹配

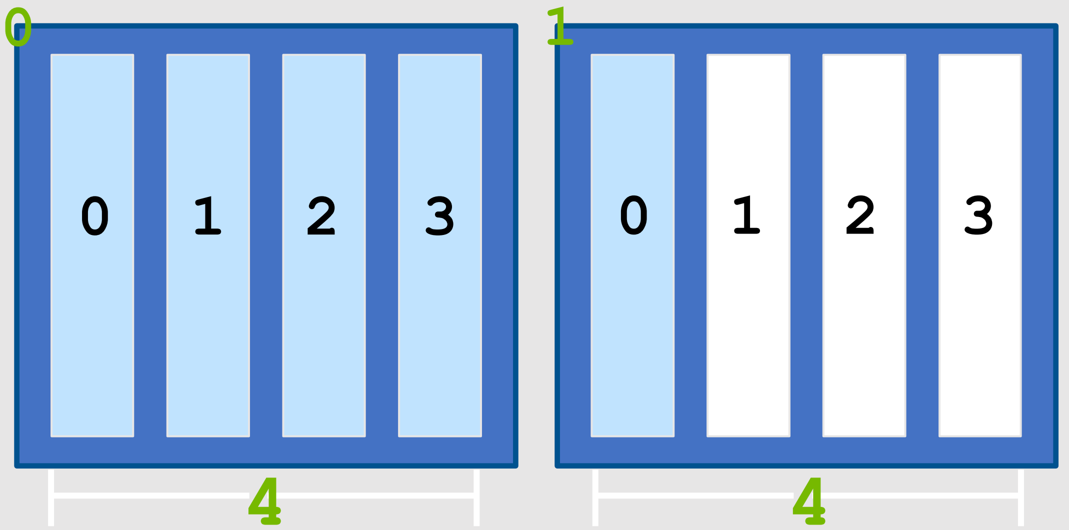
如果线程数超过要完成的工作量，该怎么办？

GPU
数据



GPU

```
performWork<<<2, 4>>>()
```

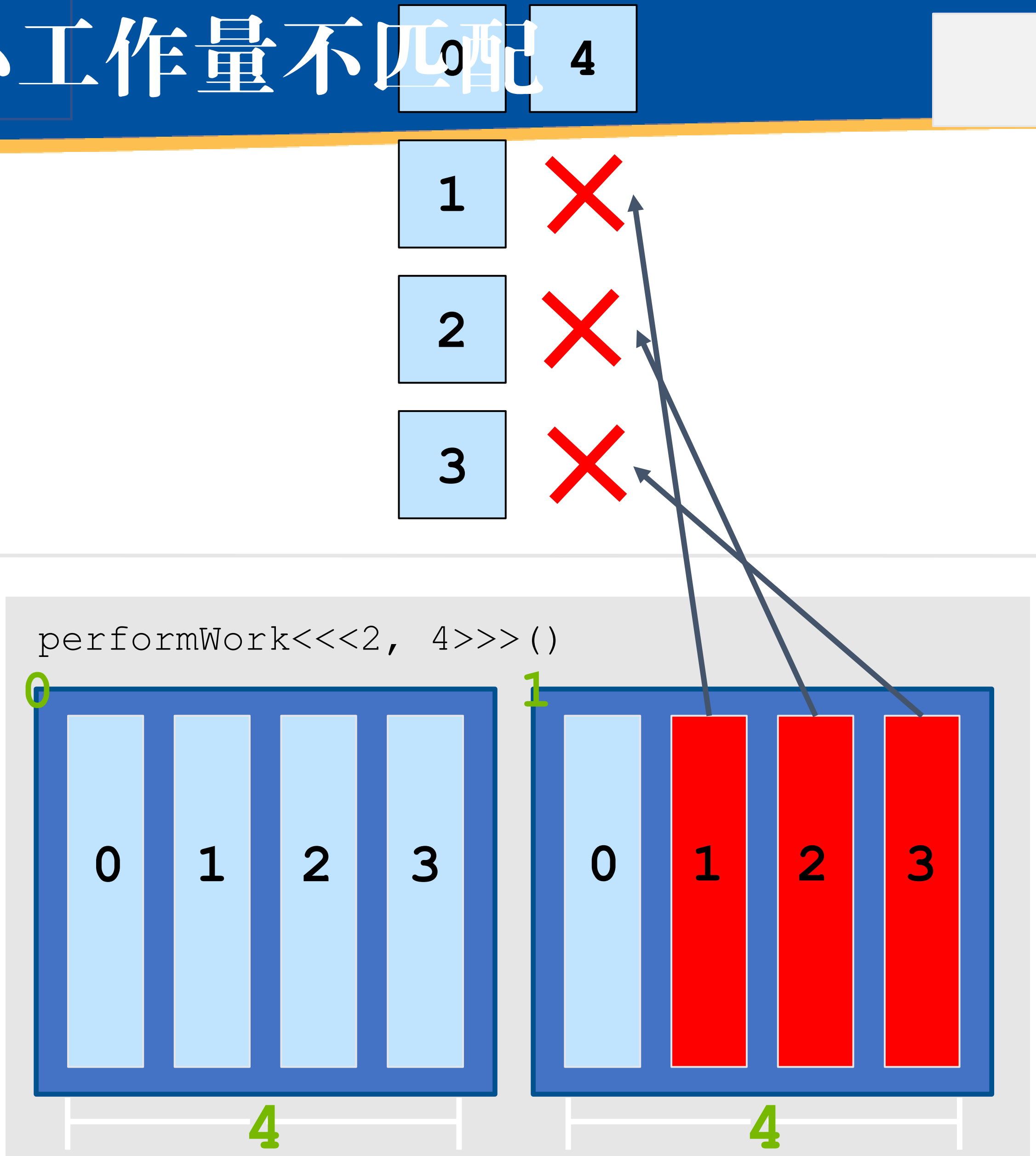


网格大小工作量不匹配

尝试访问不存在的元素会导致运行时错误

GPU
数据

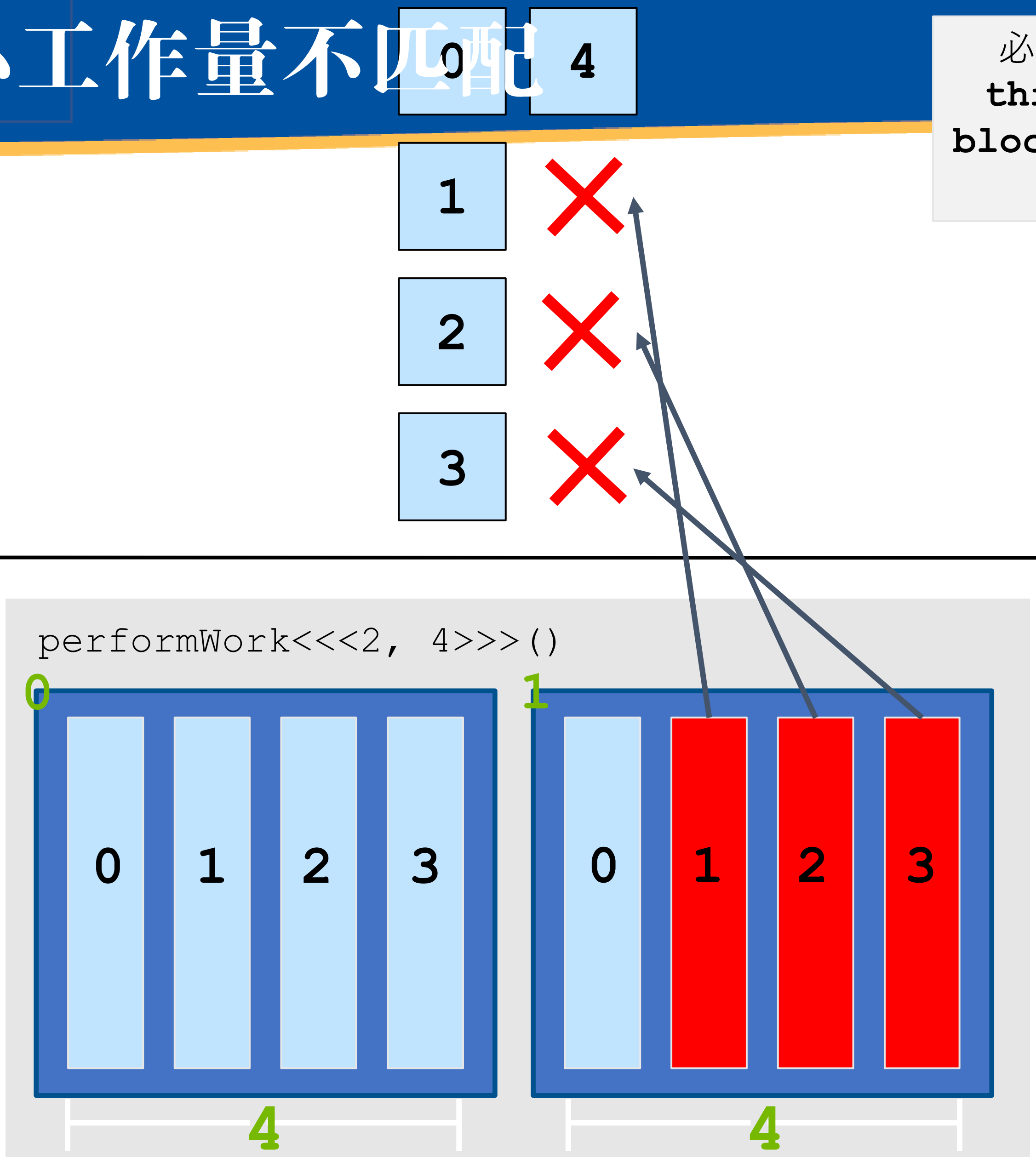
GPU



网格大小工作量不匹配

必须使用代码检查并确保经由公式 **threadIdx.x + blockIdx.x * blockDim.x** 计算出的 **dataIndex** 小于 **N**（数据元素数量）。

GPU
数据



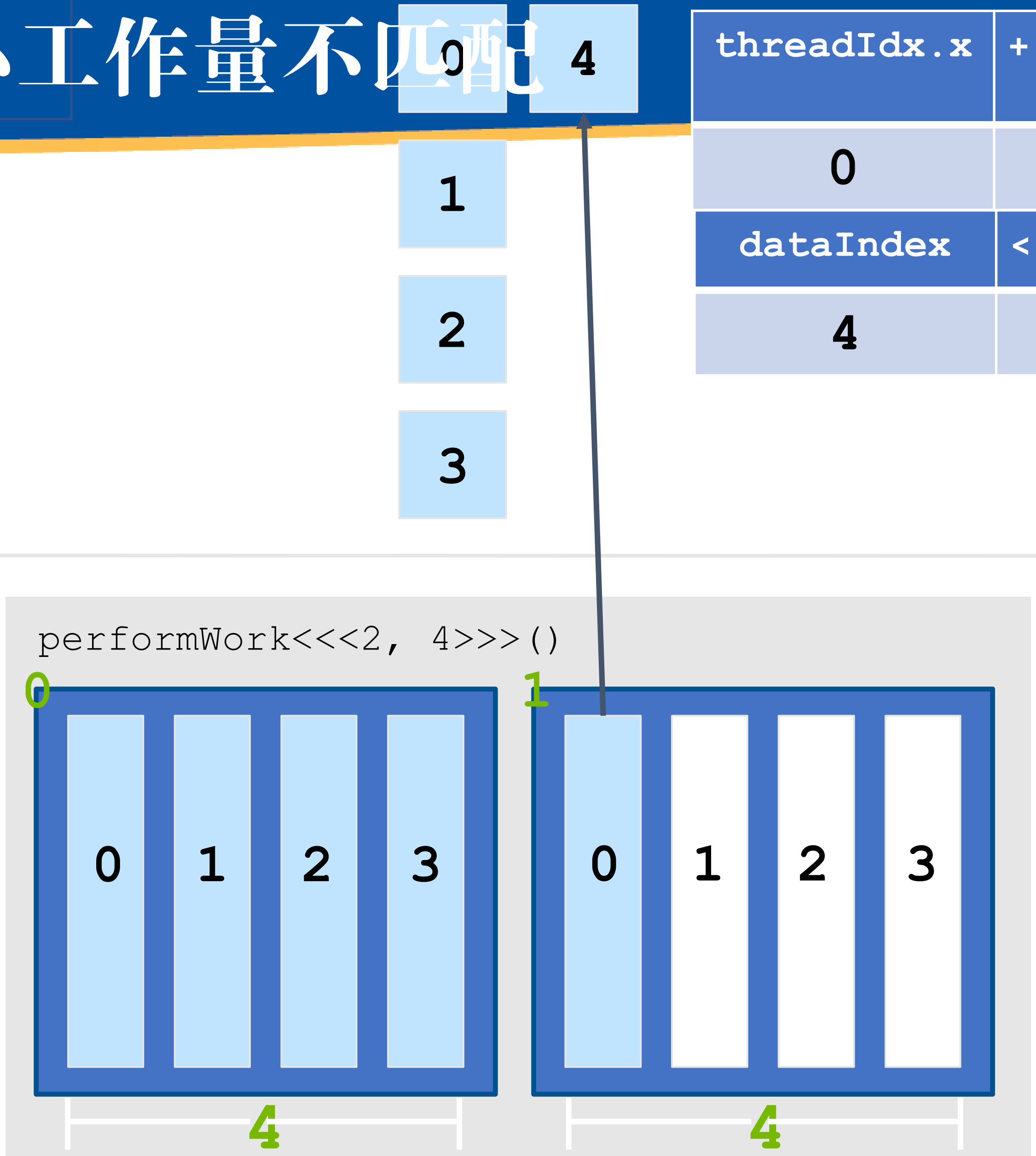
GPU

网格大小工作量不匹配

threadIdx.x	+	blockIdx.x	*	blockDim.x
0		1		4
dataIndex	<	N	=	可以运行
4		5		true

GPU
数据

GPU



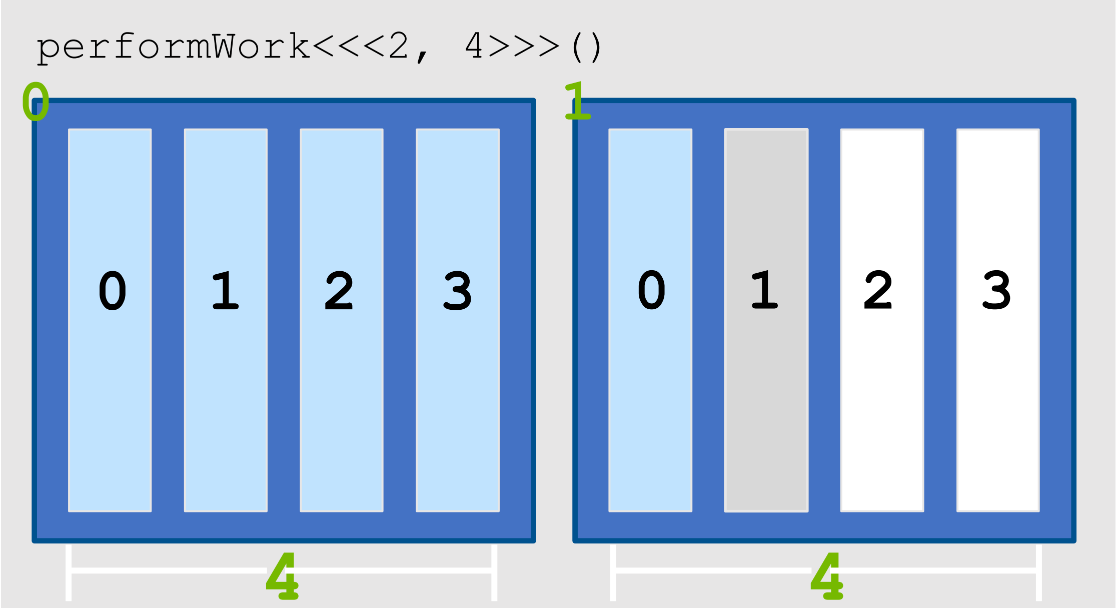
网格大小工作量不匹配

GPU
数据

0 4
1
2
3

threadIdx.x	+	blockIdx.x	*	blockDim.x
1		1		4
dataIndex	<	N	=	可以运行
5		5		false

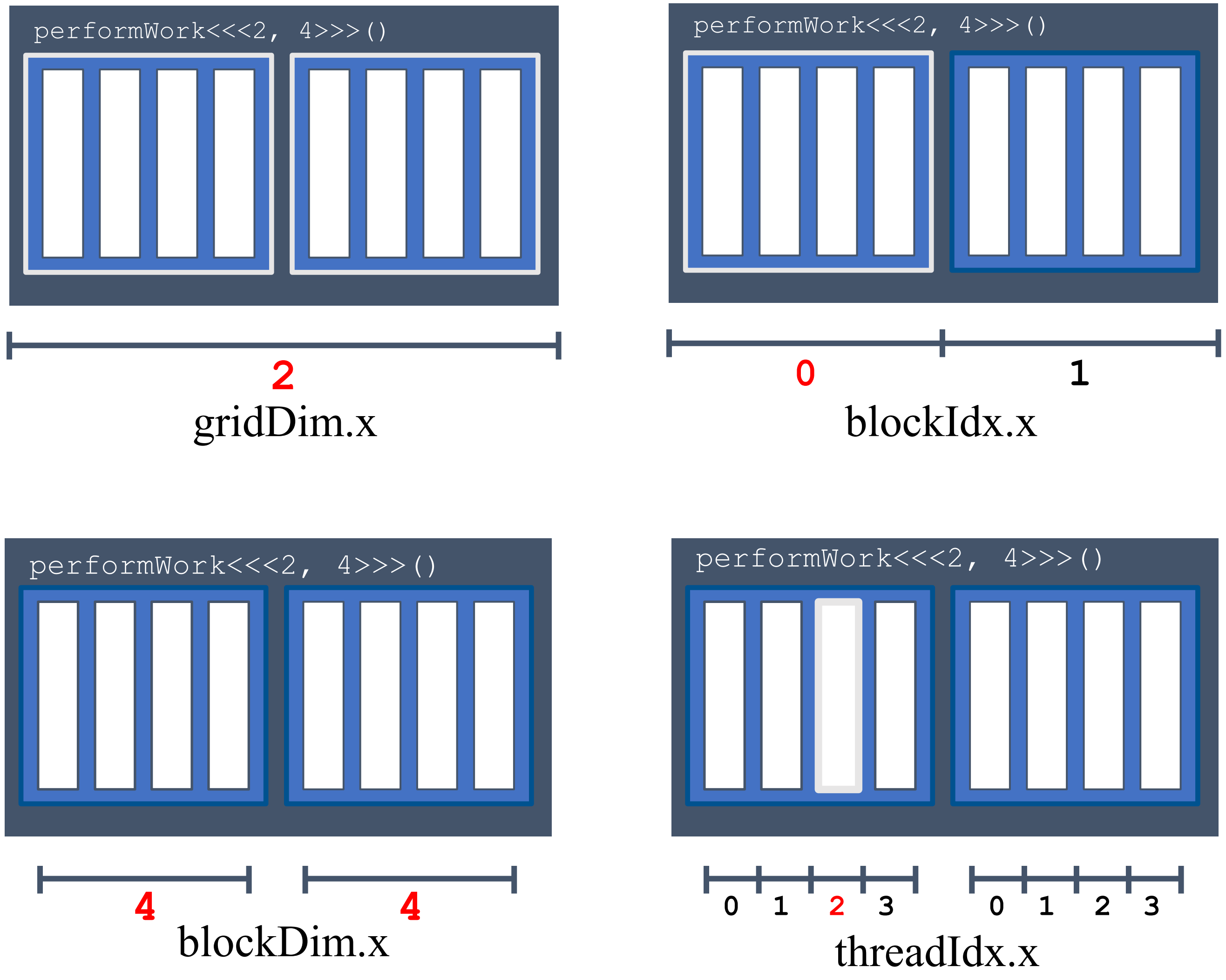
GPU



如何处理块配置与所需线程数不匹配

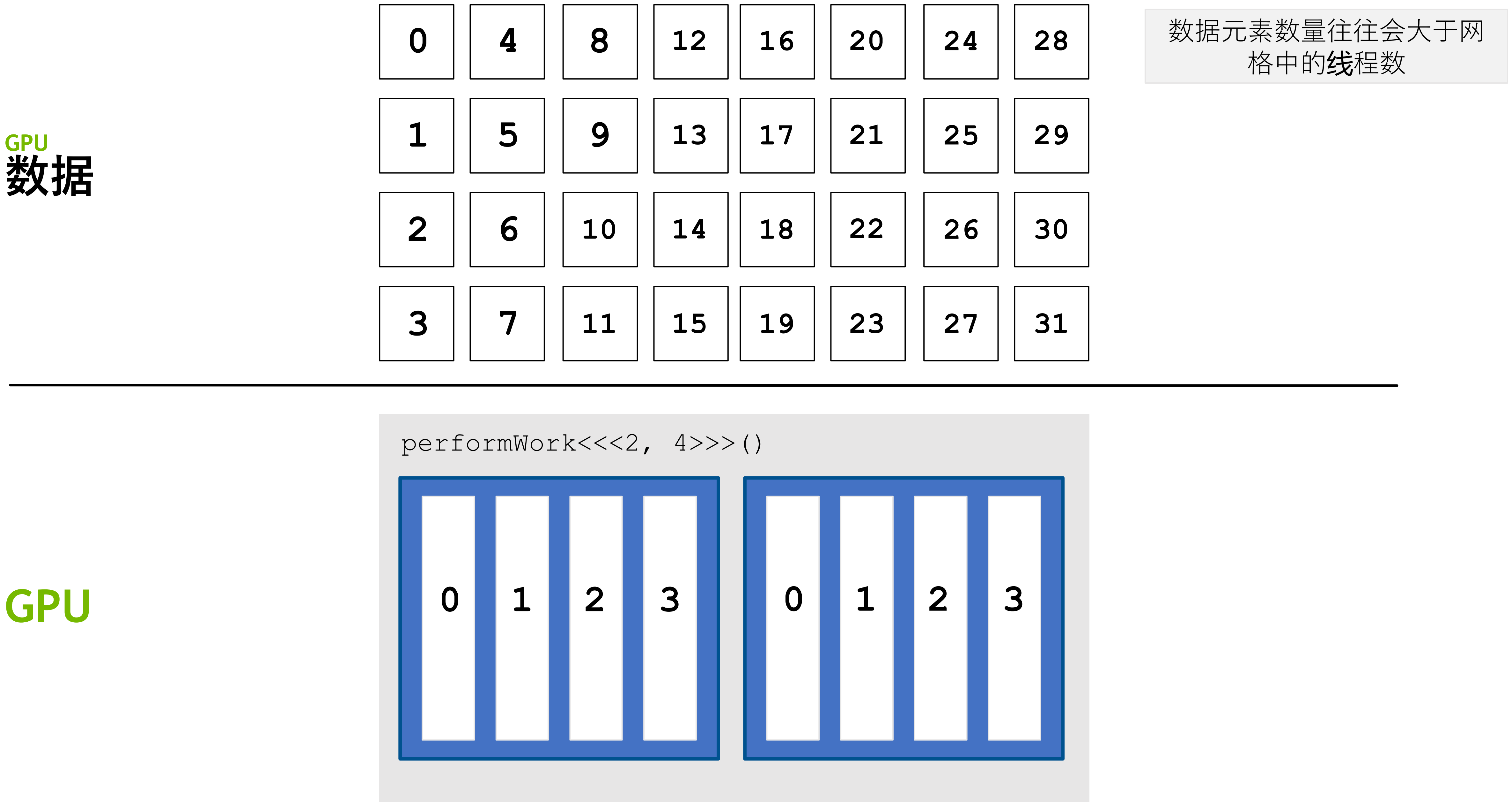
- 1000数据　1000/256 = 3.9　3+1=4　4*256=1024
- 1024数据　1024/256 = 4
- 1025数据　1025/256 = 4.01　4+1=5　5*256=1280
- size_t number_of_blocks = (N + threads_per_block - 1) / threads_per_block;

网格与线程（一维）



performWork<<<2, 4>>>()
2
gridDim.x
performWork<<<2, 4>>>()
0
1
blockIdx.x
performWork<<<2, 4>>>()
4
4
blockDim.x
performWork<<<2, 4>>>()
0 1 2 3 0 1 2 3
threadIdx.x

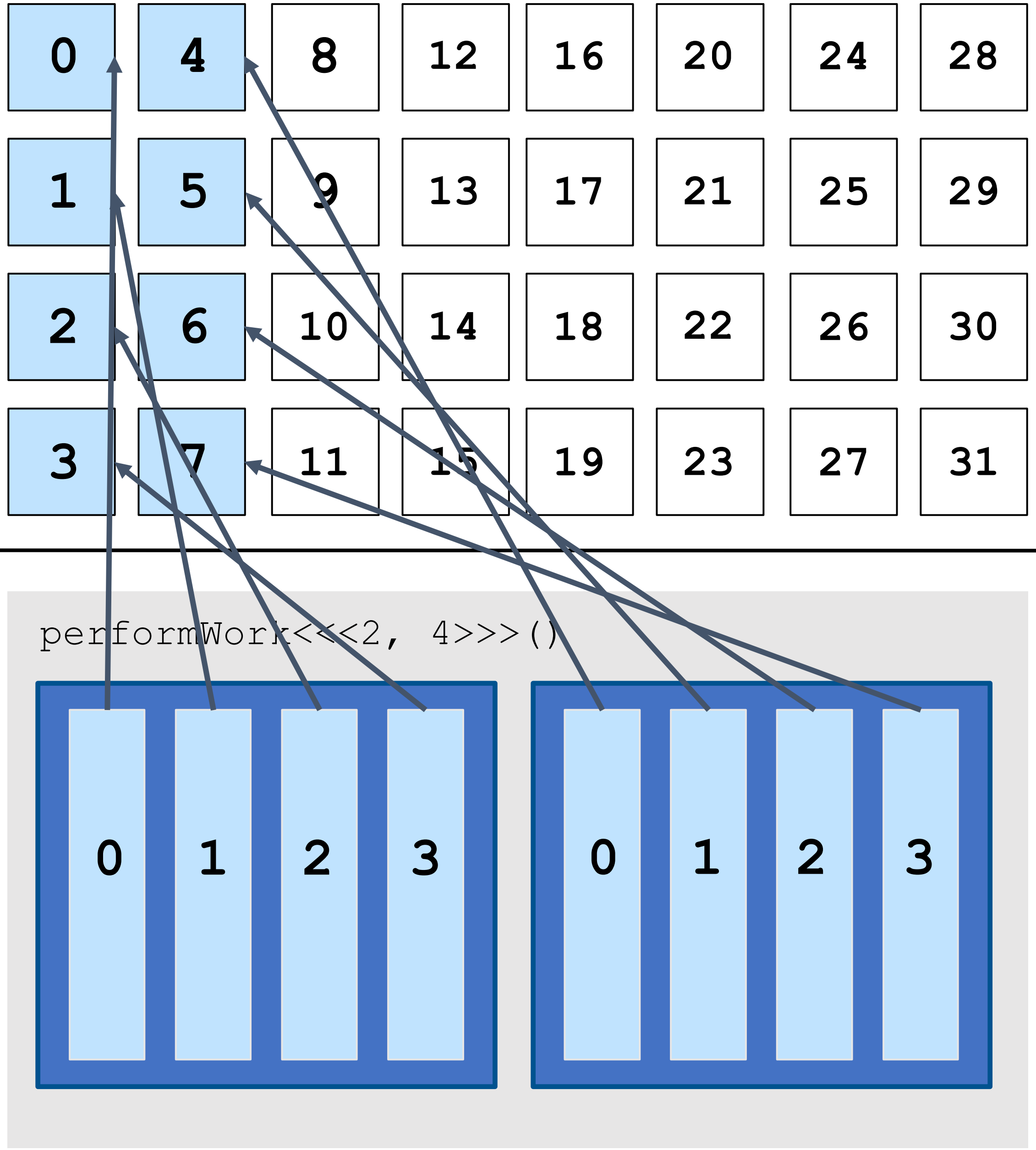
网格跨度循环



GPU
数据
0 4 8 12 16 20 24 28
1 5 9 13 17 21 25 29
2 6 10 14 18 22 26 30
3 7 11 15 19 23 27 31
数据元素数量往往会大于网格中的线程数
GPU
performWork<<<2, 4>>>()
0 1 2 3
0 1 2 3

GPU
数据

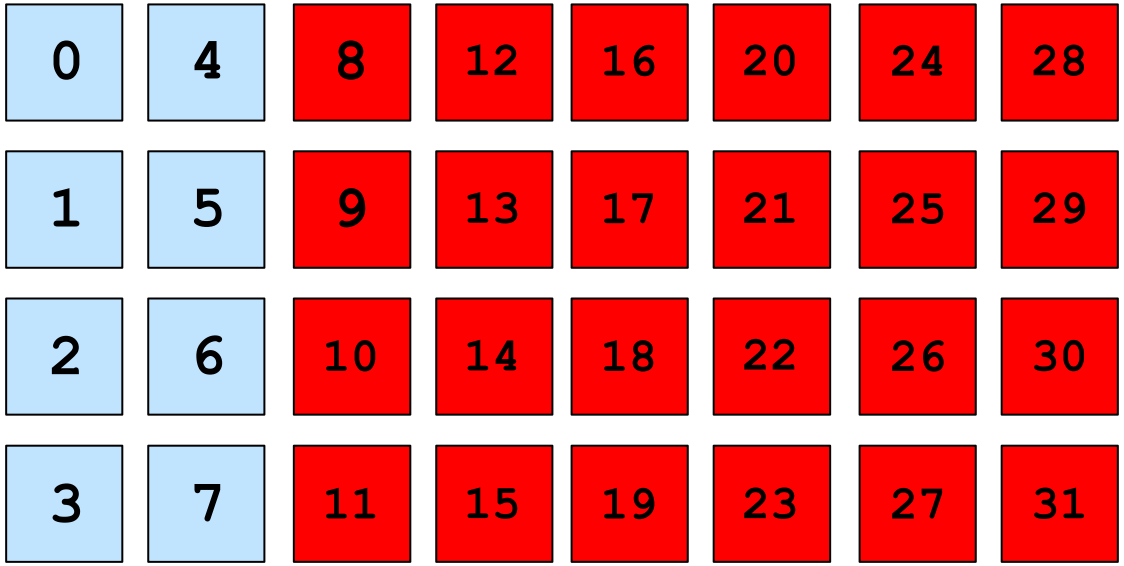
在此类情况下，线程无法只处理一个元素



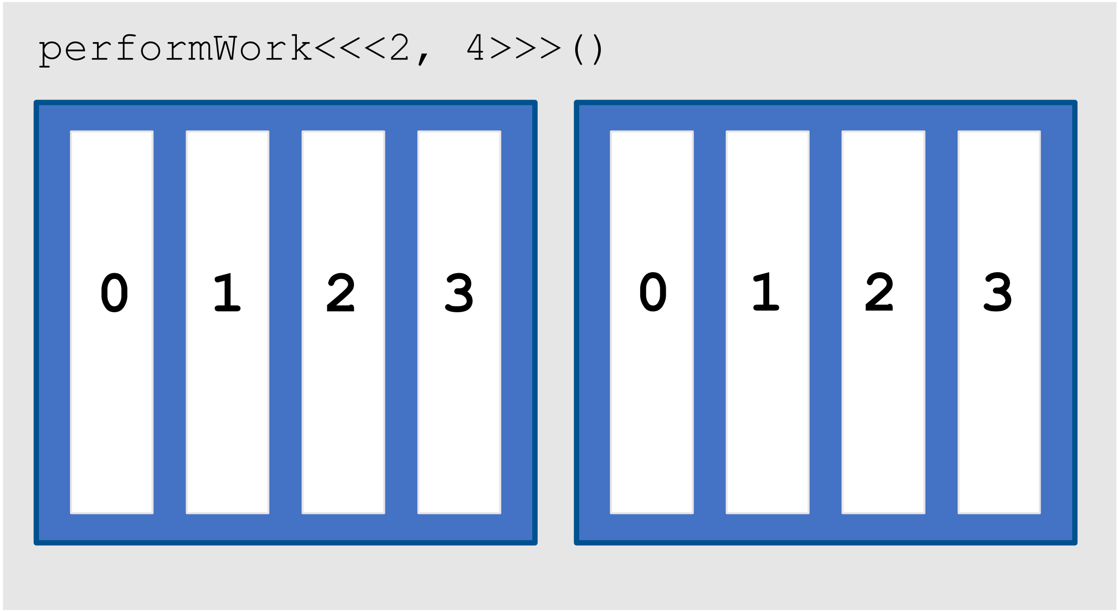
GPU

…否则工作便无法完成

GPU
数据



GPU



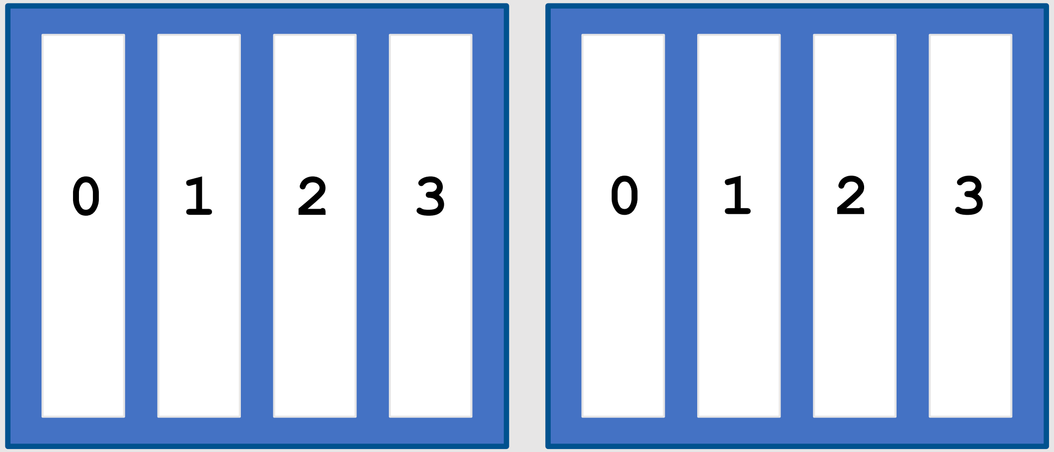
以编程方式解决此问题的其中一种方法是使用**网格跨度循环**

GPU
数据

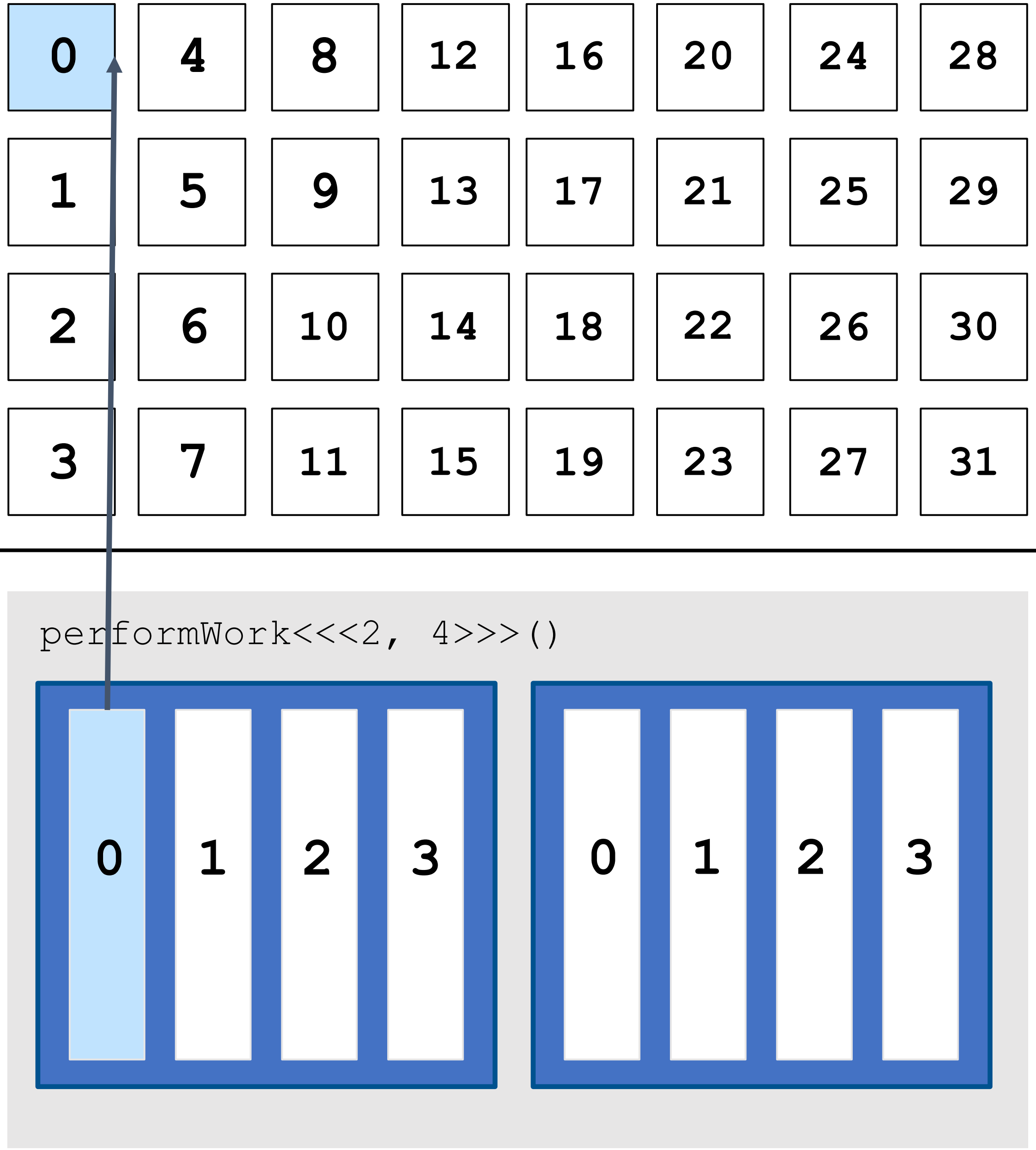
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

GPU

```
performWork<<<2, 4>>>()
```

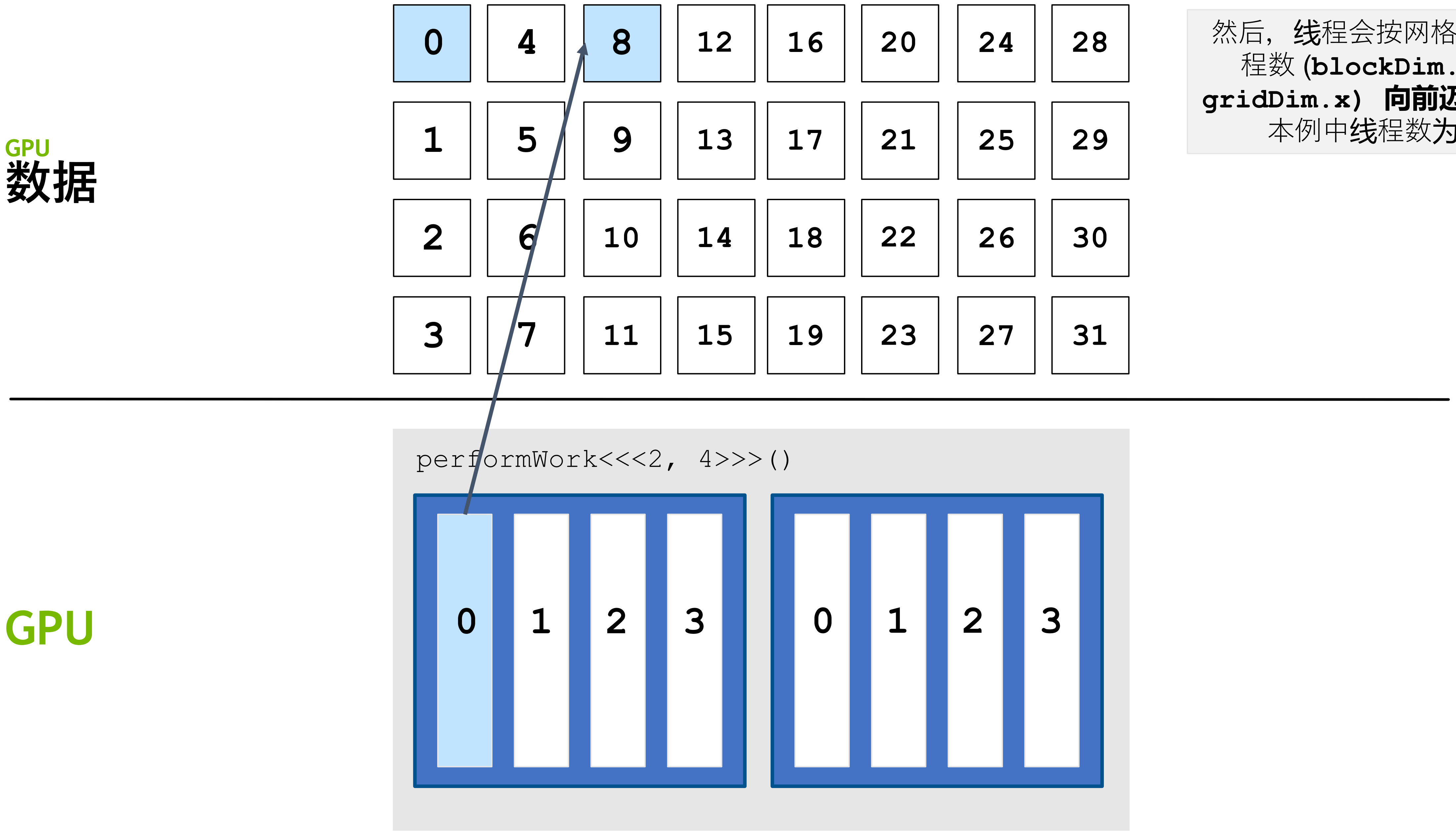


GPU
数据



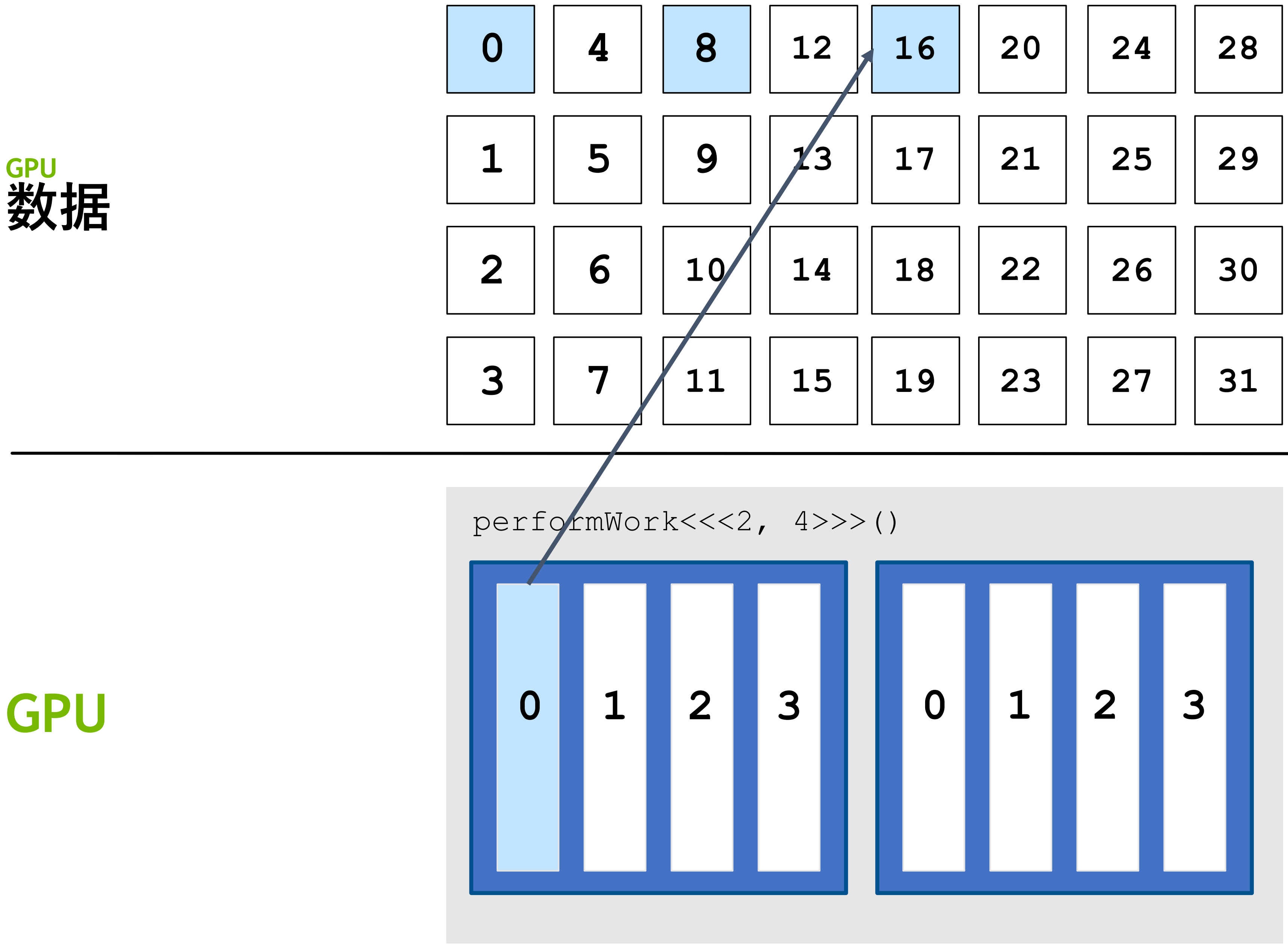
在网格跨度循环中，线程的第一个元素依旧使用 **threadIdx.x + blockIdx.x * blockDim.x** 计算得出

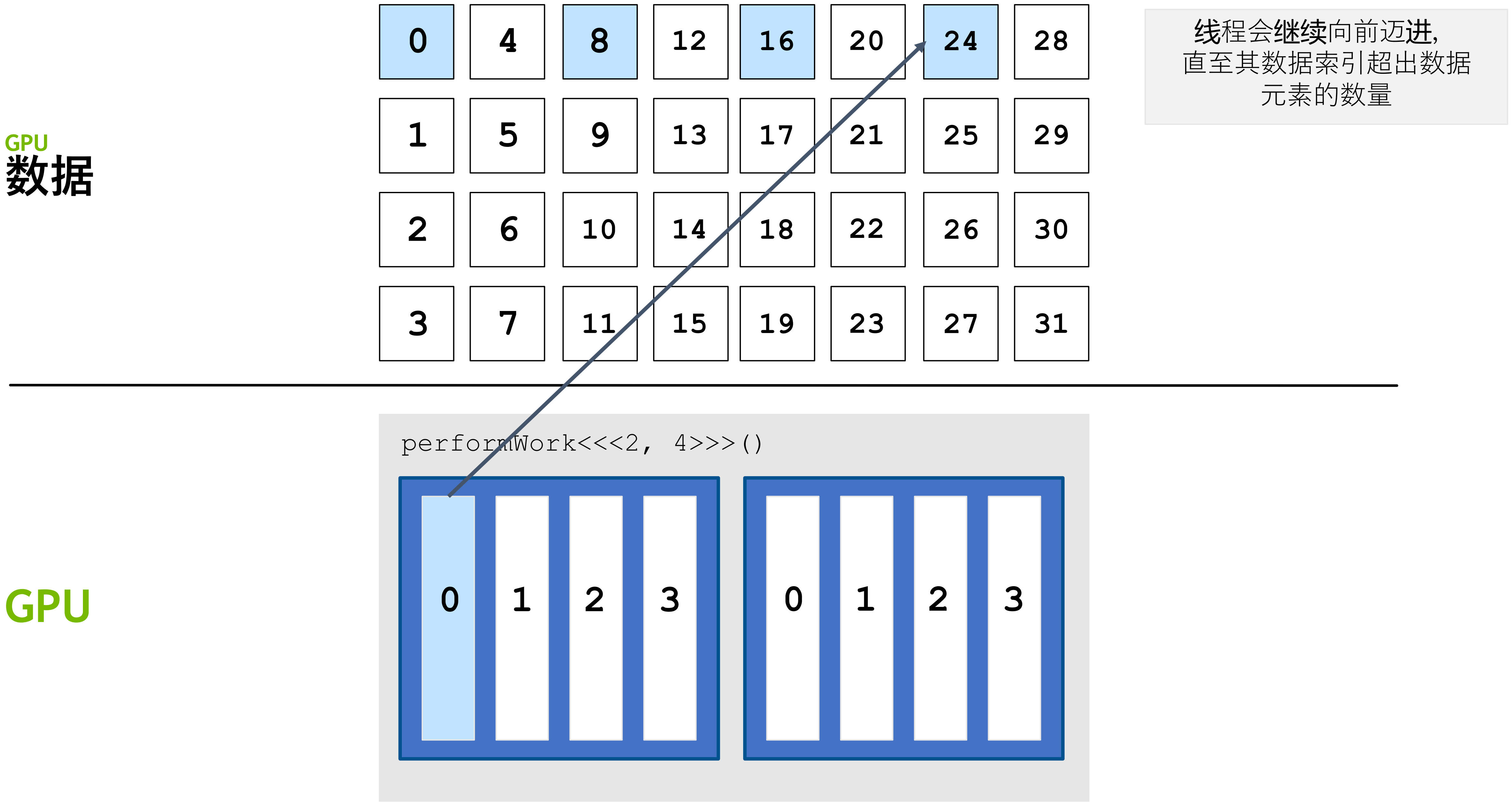
GPU



然后，**线程会按网格中的线程数 (`blockDim.x * gridDim.x`) 向前迈进**，在本例中线程数为 8

线程会继续向前迈进，
直至其数据索引超出数据
元素的数量



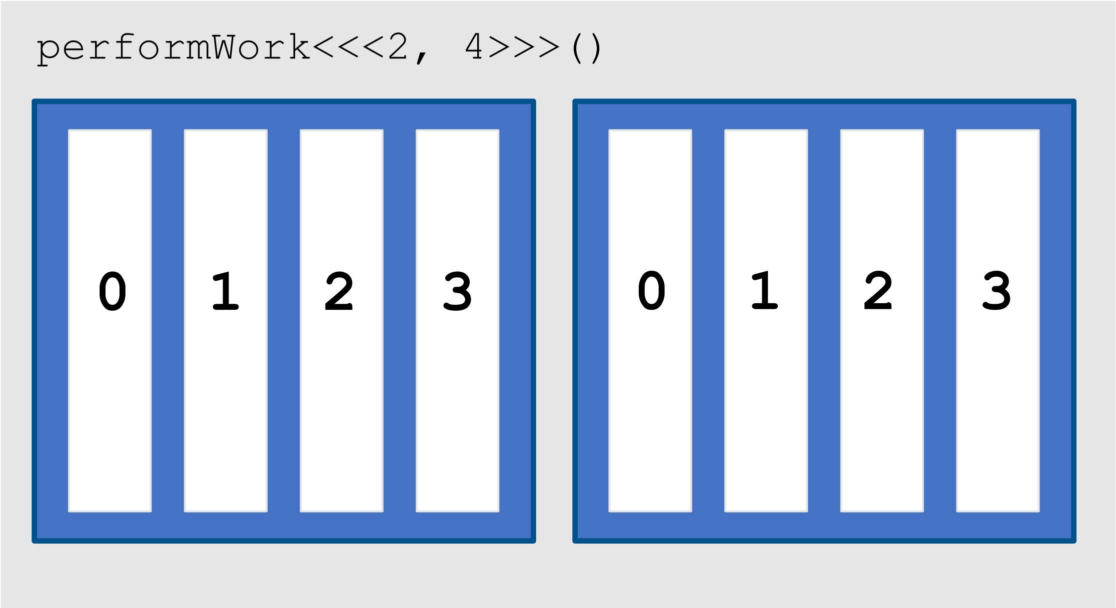

线程会继续向前迈进，
直至其数据索引超出数据
元素的数量
GPU
数据
0 4 8 12 16 20 24 28
1 5 9 13 17 21 25 29
2 6 10 14 18 22 26 30
3 7 11 15 19 23 27 31
GPU
performWork<<<2, 4>>>()
0 1 2 3
0 1 2 3

所有线程均按此种方式运作，
如此便会涵盖所有元素

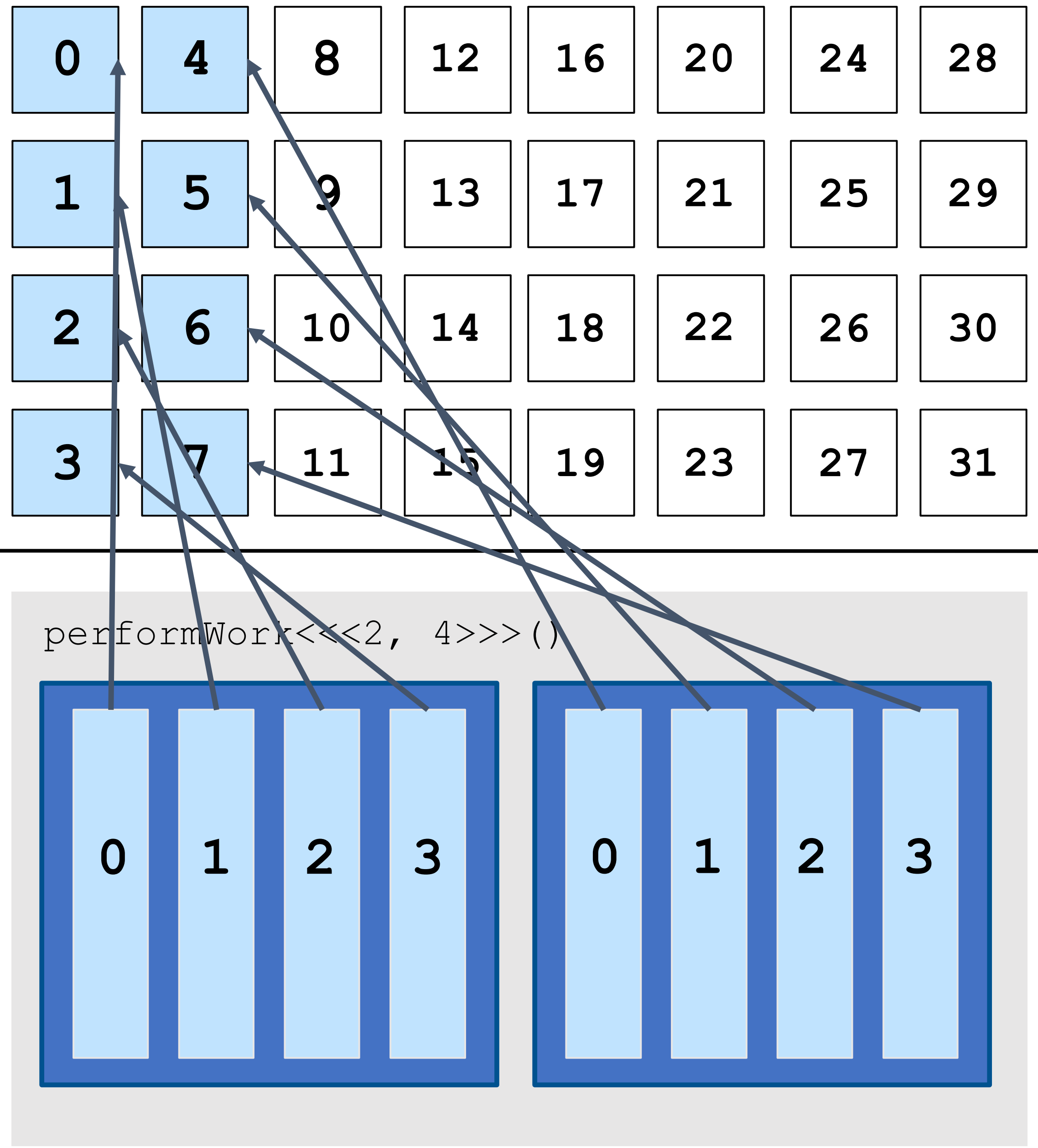
GPU
数据

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

GPU



GPU
数据

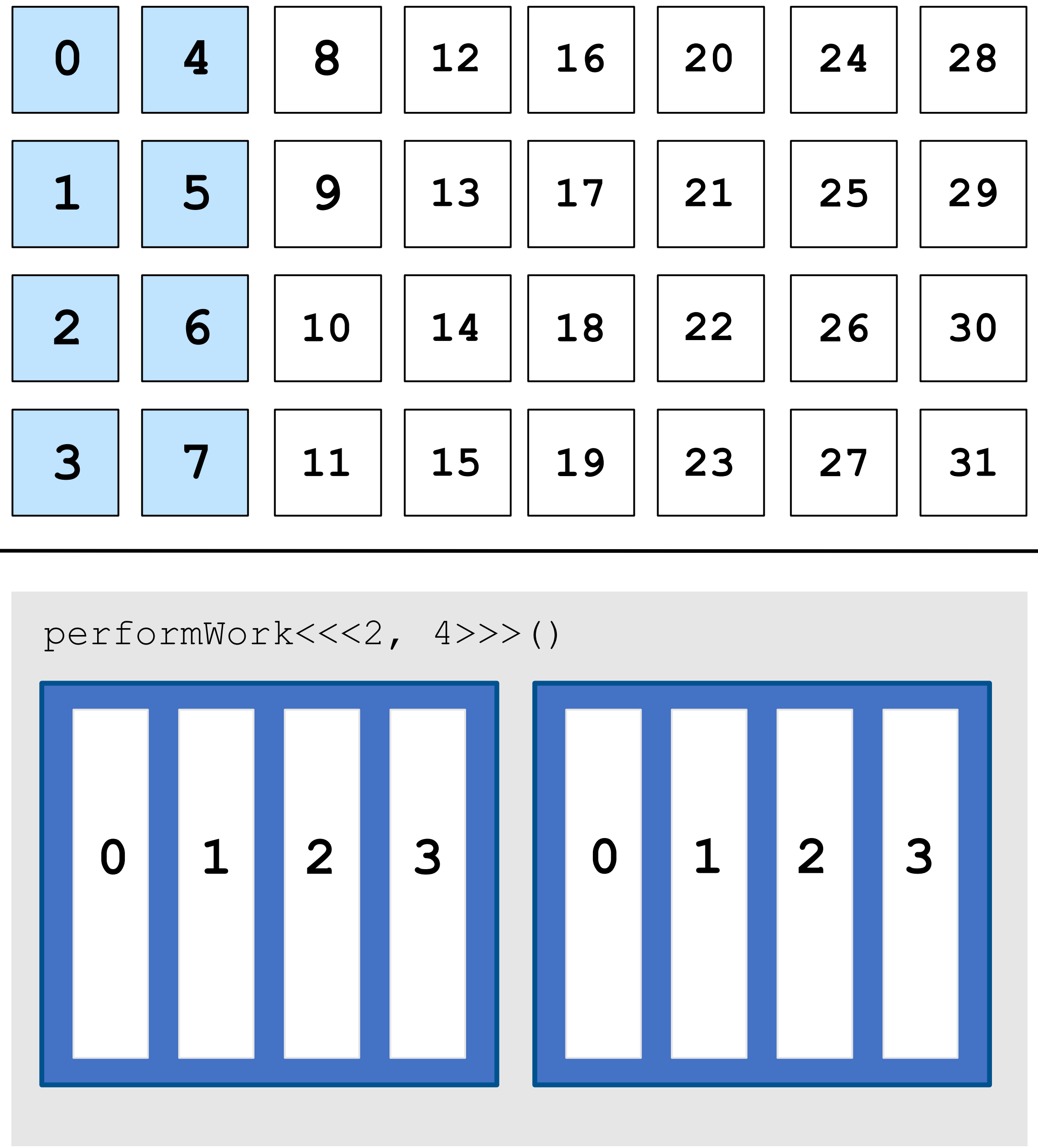


所有线程均按此种方式运作，
如此便会涵盖所有元素

GPU

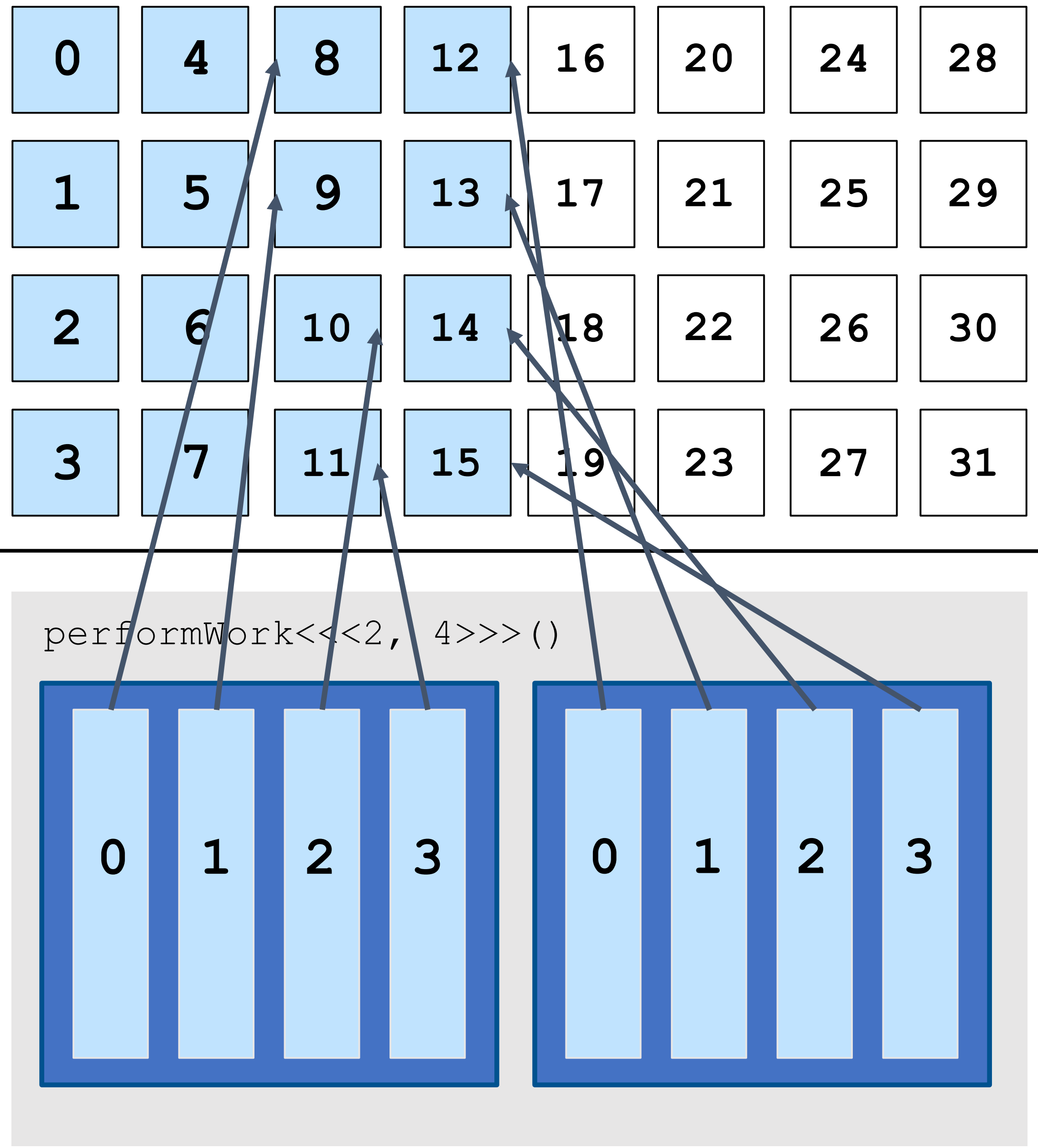
所有线程均按此种方式运作，
如此便会涵盖所有元素

GPU
数据



GPU

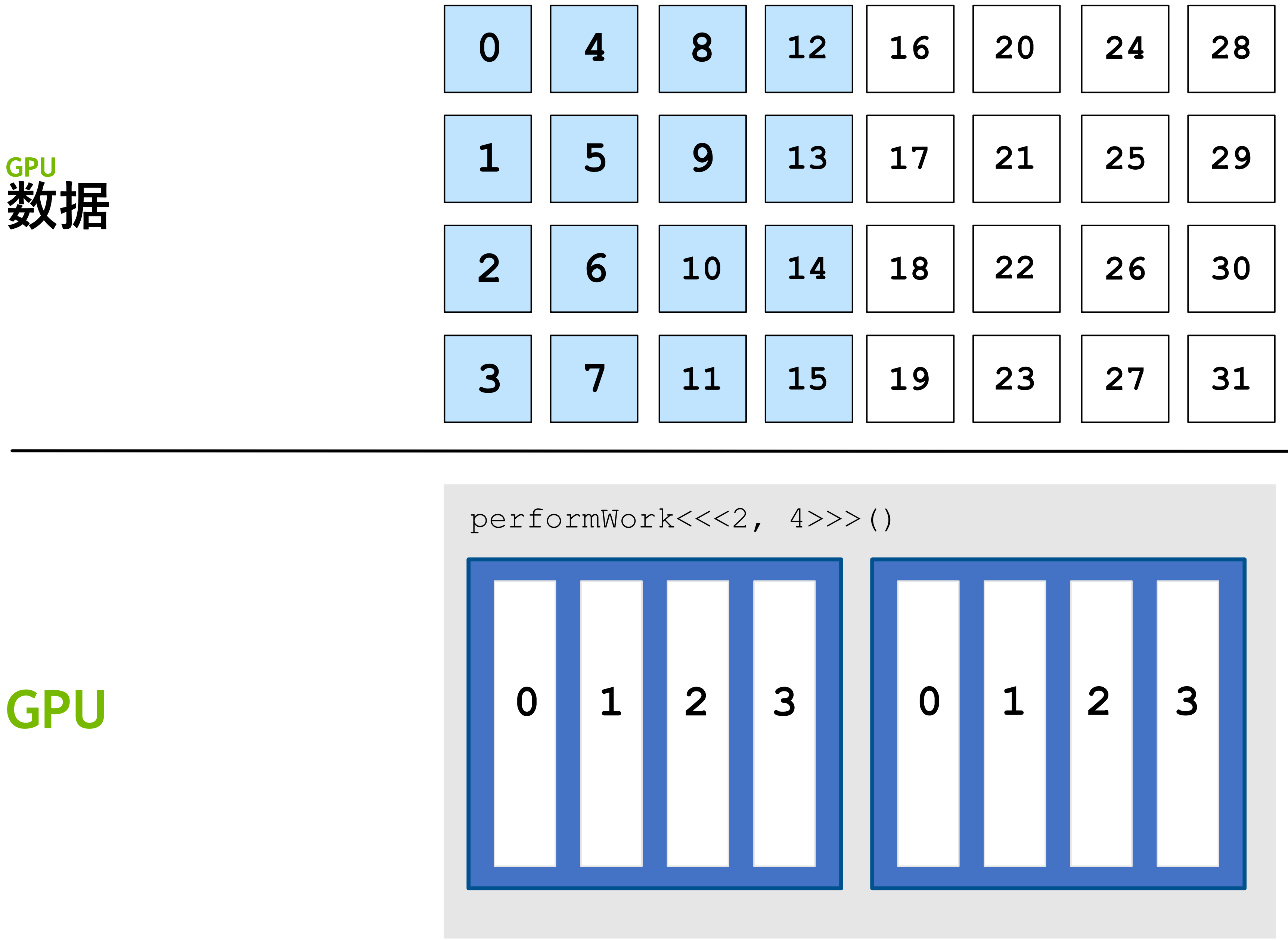
GPU
数据

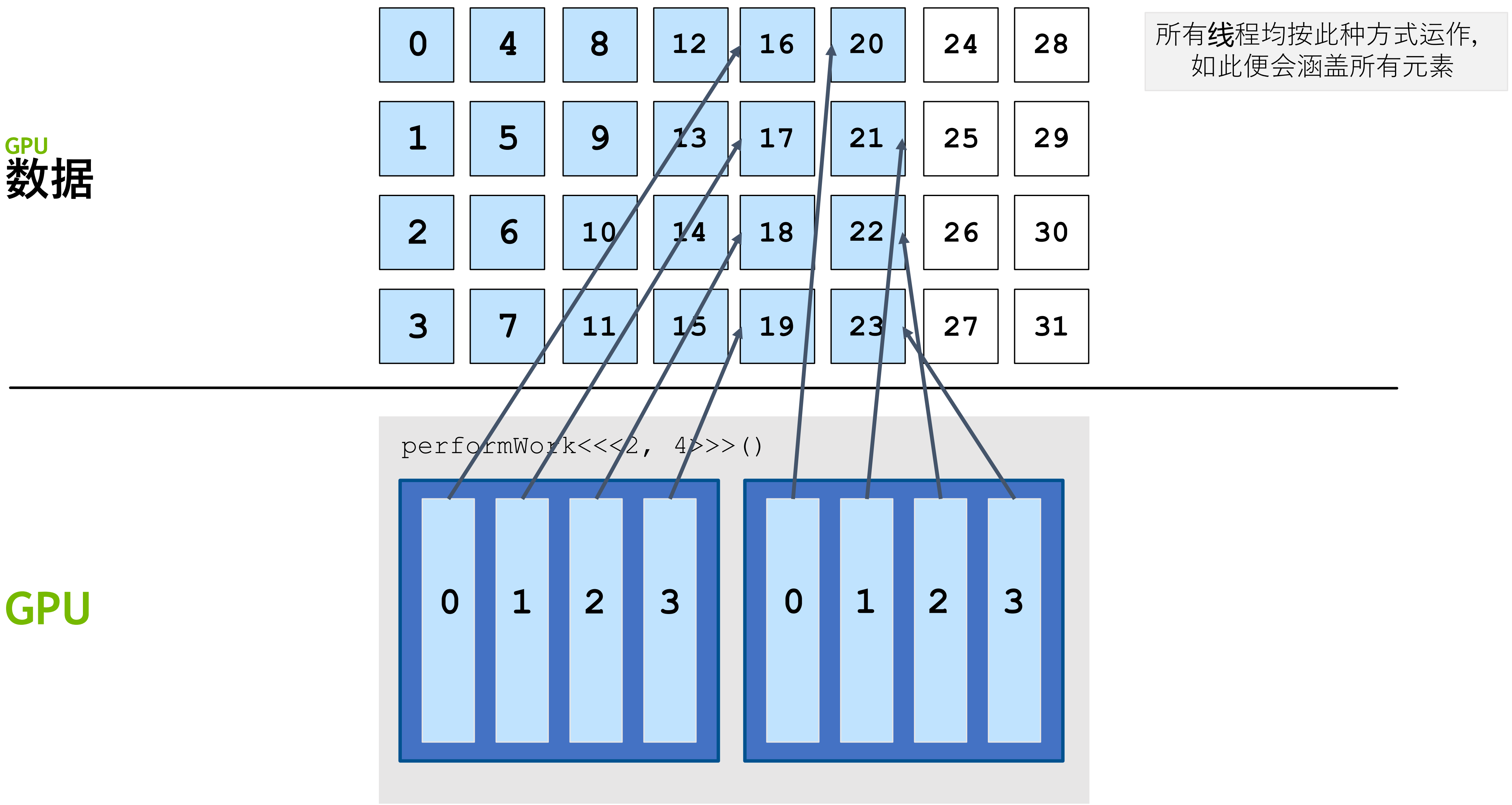


所有线程均按此种方式运作，
如此便会涵盖所有元素

GPU

所有线程均按此种方式运作，
如此便会涵盖所有元素



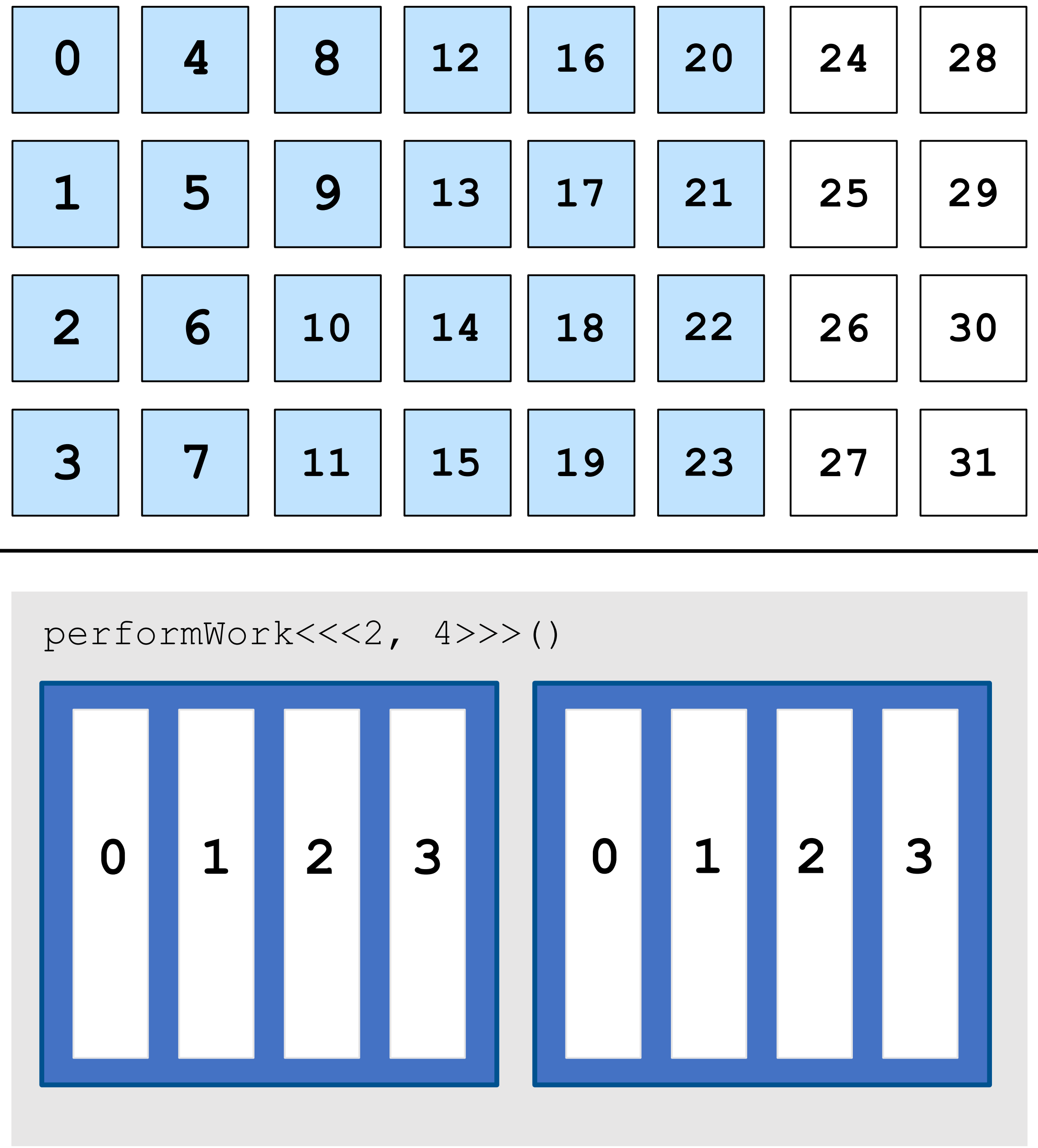


GPU
数据
0
4
8
12
16
20
24
28
1
5
9
13
17
21
25
29
2
6
10
14
18
22
26
30
3
7
11
15
19
23
27
31
所有线程均按此种方式运作，
如此便会涵盖所有元素
GPU
performWork<<<2, 4>>>()
0
1
2
3
0
1
2
3

GPU
数据

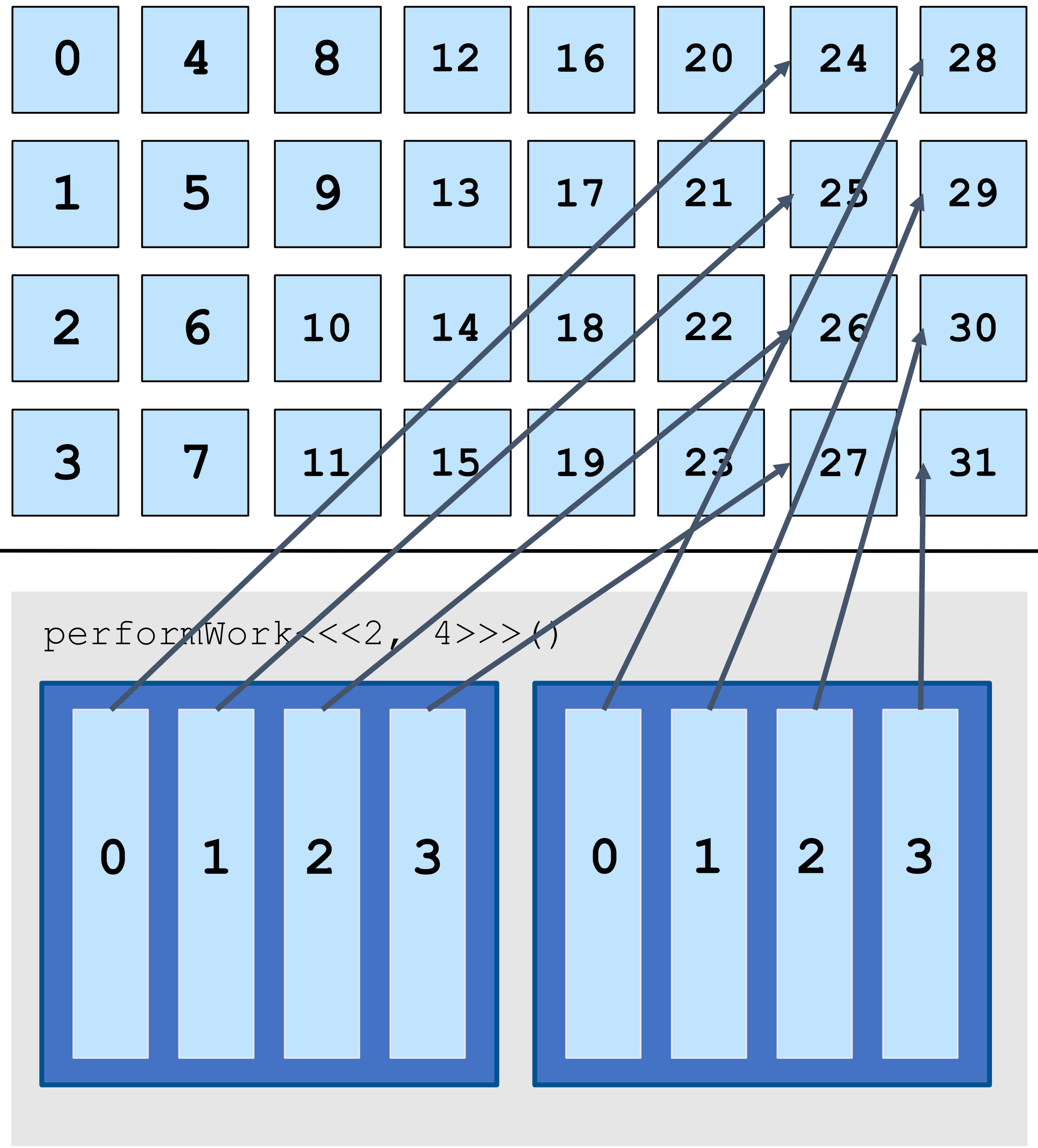
所有线程均按此种方式运作,
如此便会涵盖所有元素

GPU



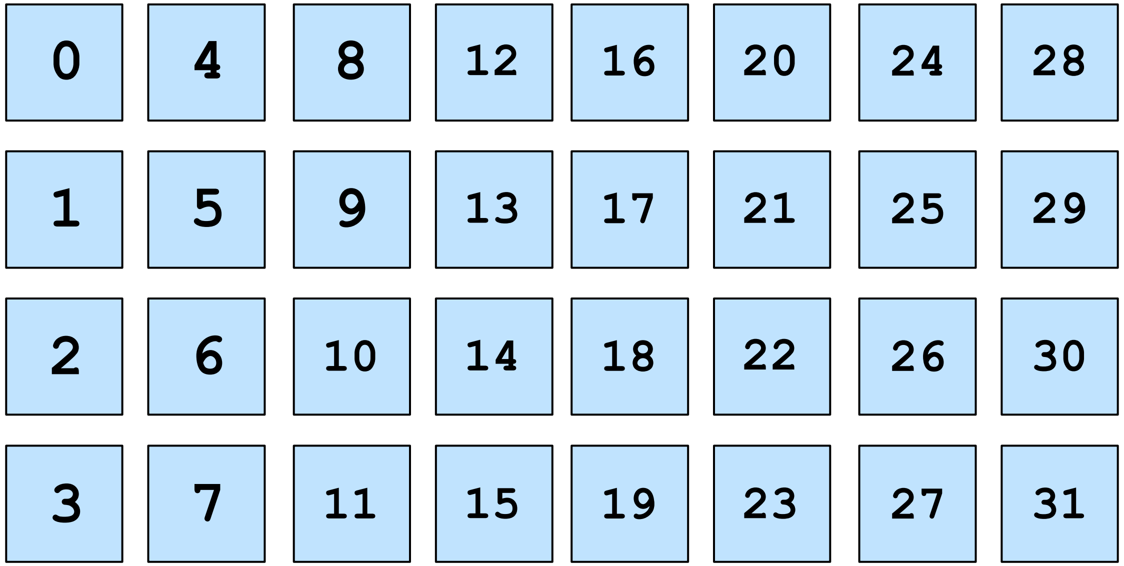
所有线程均按此种方式运作，
如此便会涵盖所有元素

GPU
数据



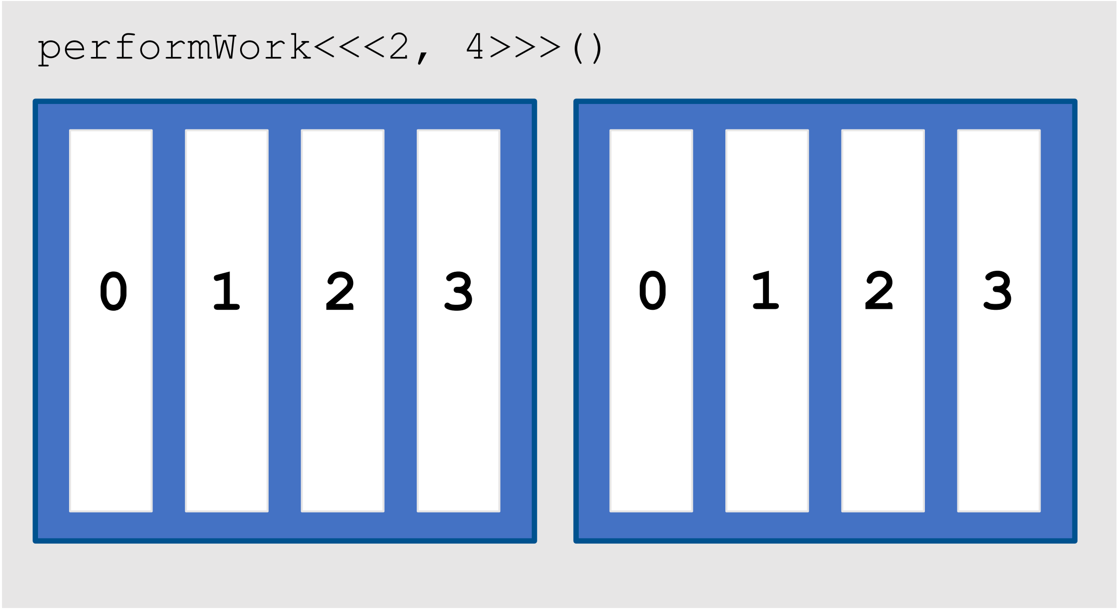
GPU

GPU
数据



所有线程均按此种方式运作，
如此便会涵盖所有元素

GPU



错误处理

```
cudaError_t err;
err = cudaMallocManaged(&a, N)                    // Assume the existence of `a` and `N`.

if (err != cudaSuccess)                           // `cudaSuccess` is provided by CUDA.
{
  printf("Error: %s\n", cudaGetErrorString(err)); // `cudaGetErrorString` is provided by CUDA.
}
```

错误处理

```
someKernel<<<1, -1>>>();  // -1 is not a valid number of threads.

cudaError_t err;
err = cudaGetLastError(); // `cudaGetLastError` will return the error from above.
if (err != cudaSuccess)
{
  printf("Error: %s\n", cudaGetErrorString(err));
}
```

已经完成以下列出的所有实验学习目标

- 编写、编译及运行既可调用 CPU 函数也可启动GPU核函数的 C/C++ 程序。
- 使用执行配置控制并行线程层次结构。
- 重构串行循环以在 GPU 上并行执行其迭代。
- 分配和释放可用于 CPU 和 GPU 的内存。
- 处理 CUDA 代码生成的错误。

目标

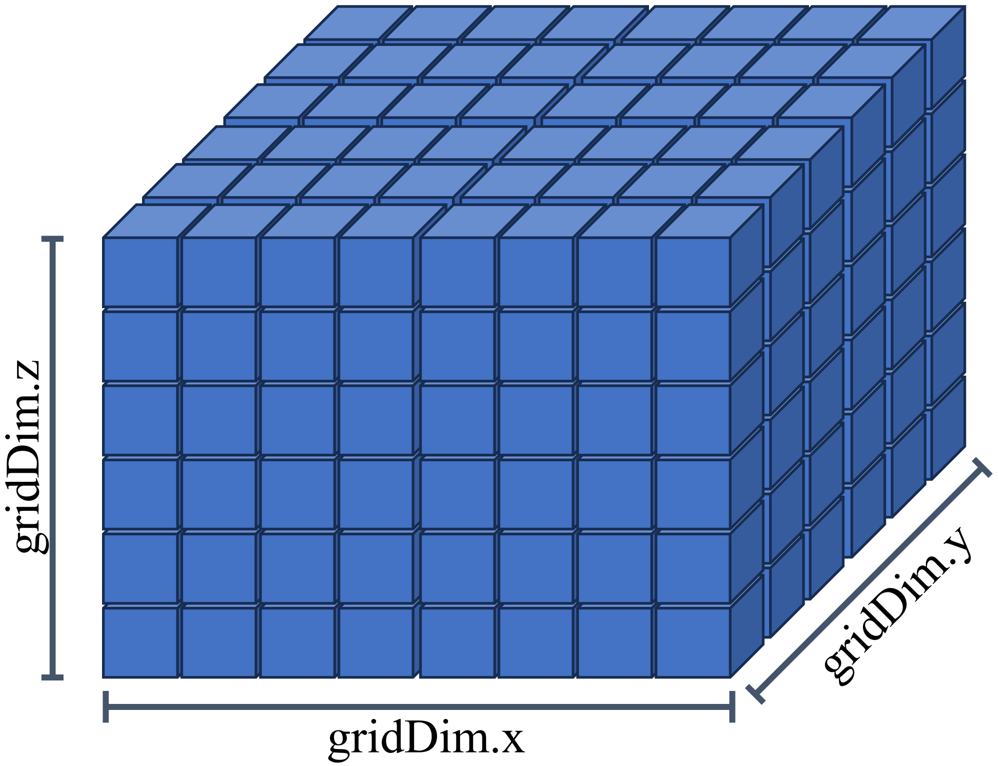
- 加速 CPU 应用程序
  - 练习：加速向量加法
- 进阶内容
  - 2维和3维的网格和块
  - 练习：加速2D矩阵乘法应用
  - 练习：给热传导应用程序加速

练习：加速向量加法

- 核函数 __global__
- 内存分配 cudaMallocManaged
- 执行启动配置
- 跨网格循环
- 改写CPU循环为GPU循环
- 错误处理

网格与线程（二维和三维）

- gridDim.x
- gridDim.y
- gridDim.z



- blockDim.x
- blockDim.y
- blockDim.z

- blockIdx.x
- blockIdx.y
- blockIdx.z

- threadIdx.x
- threadIdx.y
- threadIdx.z

练习：加速2D矩阵乘法应用

- **您将需创建执行配置，其参数均为 dim3 值，且 x 和 y 维度均设为大于 1。**
- **在核函数主体内部，您将需要按照惯例在网格内建立所运行线程的唯一索引，但应为线程建立两个索引：一个用于网格的 x 轴，另一个用于网格的 y 轴。**

```
void matrixMulCPU( int * a, int * b, int * c )
{
  int val = 0;

  for( int row = 0; row < N; ++row )
    for( int col = 0; col < N; ++col )
    {
      val = 0;
      for ( int k = 0; k < N; ++k )
        val += a[row * N + k] * b[k * N + col];
      c[row * N + col] = val;
    }
}
```


NVIDIA