CUDA ON ARM PLATFORM—原子操作

NVIDIA企业级开发者社区 何琨

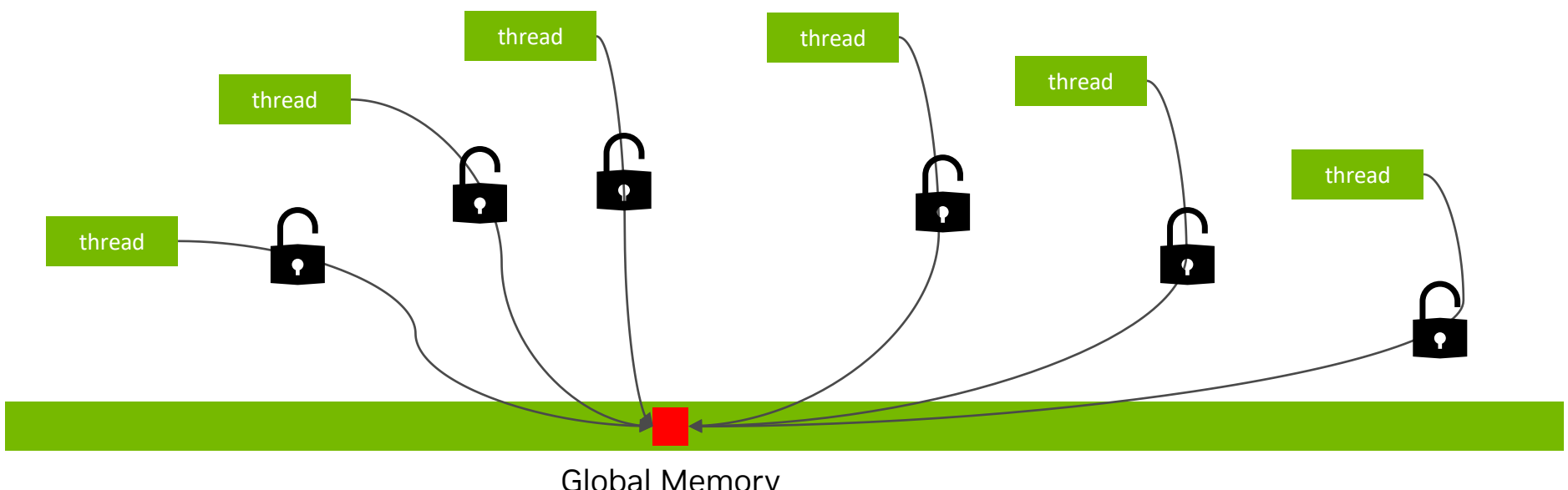
AGENDA

原子操作

- 原子操作的基本概念
- 原子操作常用函数
- CUDA中的规约问题
- CUDA中的warp级数据交换方法
- 原子操作实例

CUDA 原子操作的概念

- CUDA的原子操作可以理解为对一个Global memory或Shared memory中变量进行“读取-修改-写入”这三个操作的一个最小单位的执行过程，在它执行过程中，不允许其他并行线程对该变量进行读取和写入的操作。
- 基于这个机制，原子操作实现了对在多个线程间共享的变量的互斥保护，确保任何一次对变量的操作的结果的正确性。

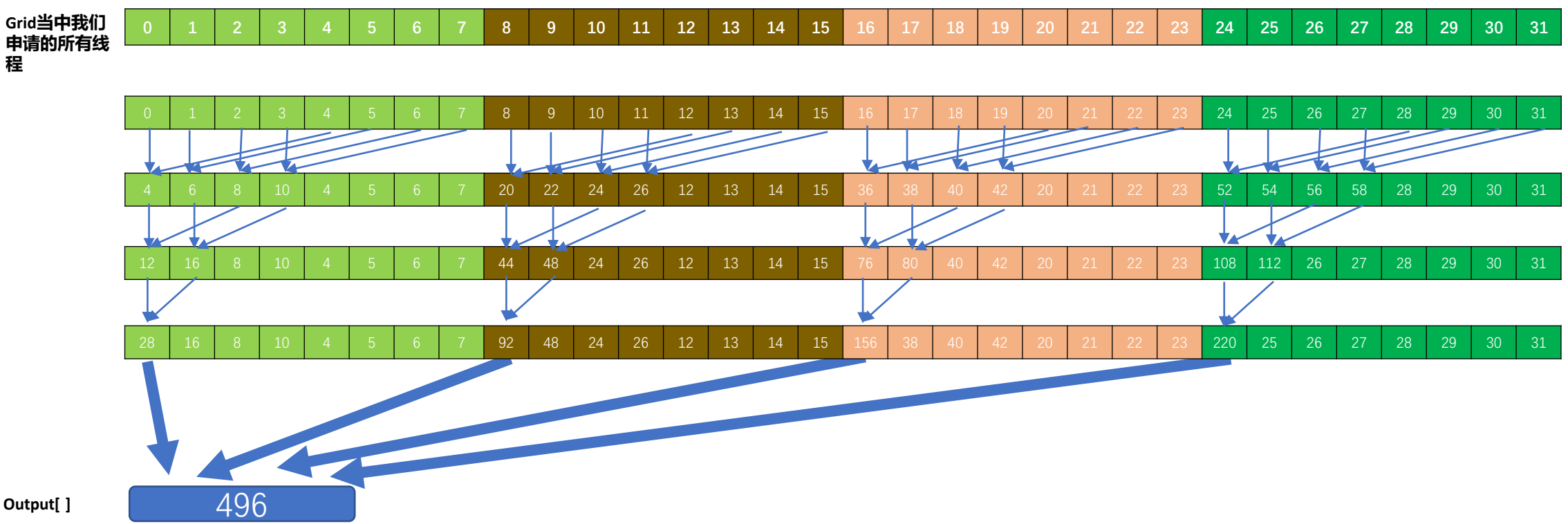


原子操作常用函数

atomicAdd(&value, num)	加法：value = value + num
atomicSub(&value, num)	减法：value = value - num
atomicExch(&value, num)	赋值：value = num
atomicMax(&value, num)	求最大：value = max(value, num)
atomicMin(&value, num)	求最小：value = min(value, num)
atomicInc(&value, num)	向上计数：如果(value <= num)则value++,否则value = 0
atomicDec(&value, num)	向下计数：如果(value > num或value == 0), 则value--,否则value = 0
atomicCAS(&value, num, val)	比较并交换：如果(value == num),则value = val
atomicAnd(&value, num)	与运算：value = value and num
atomicOr(&value, num)	或运算 value = value or num
atomicXor(&value, num)	异或运算 value = value xor num

CUDA中的规约问题

1 + 2 + 3 + 4+ ..... +N



向量元素求和

向量所有元素求和

1. 主要是想让大家理解利用cuda做reduce的操作，也是我们实际工作中会常遇到的问题。
2. 主要难点是如何利用shared memory和安排线程的过程
3. 最容易出错的地方在并不是所有线程在所有步骤都会有动作

```
__global__ void _sum_gpu(int *input, int count, int *output)
{
    __shared__ int bowman[BLOCK_SIZE];

    int komorebi = 0;
    for (int idx = threadIdx.x + blockDim.x * blockIdx.x;
         idx < count;
         idx += gridDim.x * blockDim.x
         )
    {
        komorebi += input[idx];
    }

    bowman[threadIdx.x] = komorebi;  //the per-thread partial sum is komorebi!
    __syncthreads();

    for (int length = BLOCK_SIZE / 2; length >= 1; length /= 2)
    {
        int double_kill = -1;
        if (threadIdx.x < length)
        {
            double_kill = bowman[threadIdx.x] + bowman[threadIdx.x + length];
        }
     __syncthreads();  //why we need two __syncthreads() here, and,

        if (threadIdx.x < length){
         bowman[threadIdx.x] = double_kill;
        }
        __syncthreads();
    }
    if (blockDim.x * blockIdx.x < count){
        if (threadIdx.x == 0) output[blockIdx.x] = bowman[0];
    }
}
```

Grid当中我们申请的所有线程

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

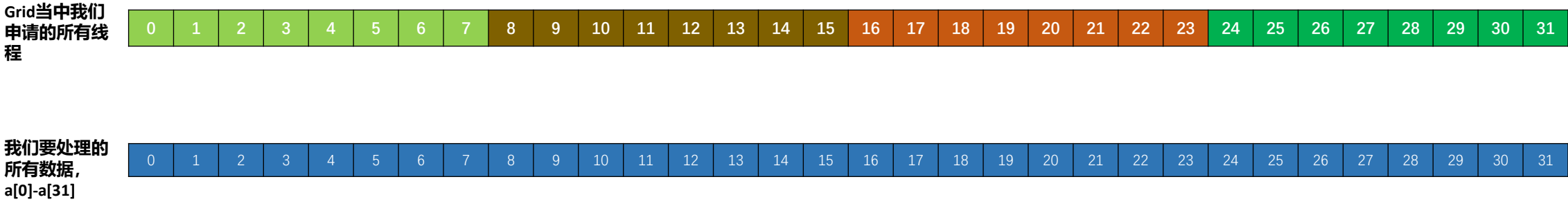
我们要处理的所有数据，a[0]-a[31]

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

向量所有元素求和

1. 申请N个线程
2. 每个线程先通过threadIdx.x + blockDim.x * blockIdx.x得到当前线程再所有线程中的index
3. 每个线程读取一个数据，并放到所在block中shared memory中，也就是bowman里面。
4. 利用__syncthreads()同步，等待所有线程执行毕。

```
int komorebi = 0;
for (int idx = threadIdx.x + blockDim.x * blockIdx.x;
     idx < count;
 idx += gridDim.x * blockDim.x
)
{
    komorebi += input[idx];
}

bowman[threadIdx.x] = komorebi;  //the per-thread partia
__syncthreads();
```



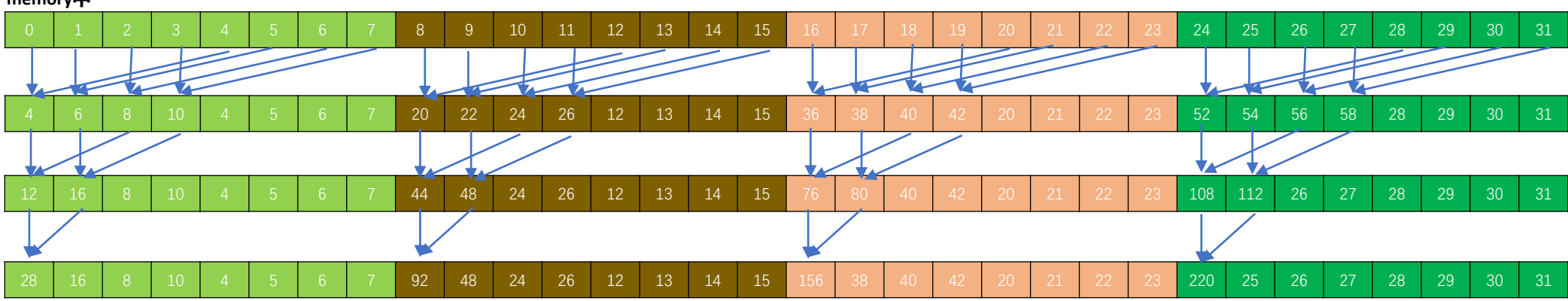
1.如下图所示，每个线程读取他所在block中shared memory中的数据(bowman），每次读取两个做加法。同步直到所有线程都做完，并将结果写到它所对应的shared memory位置中
2.直到将它所在的所有的shared memory当中的数值累加完毕。
3.这里需要注意，并不是所有线程每个迭代步骤都要工作。如下图，每个迭代步骤工作的线程数是上一个迭代步骤的一半
4.完成这个阶段，每个线程块的shared memory中第0号位置，就保存了该线程块中所有数据的总和。

```
for (int length = BLOCK_SIZE / 2; length >= 1; length /= 2)
{
    int double_kill = -1;
   if (threadIdx.x < length)
   {
       double_kill = bowman[threadIdx.x] + bowman[threadIdx.x + length];
   }
   __syncthreads();  //why we need two __syncthreads() here, and,

   if (threadIdx.x < length)
   {
       bowman[threadIdx.x] = double_kill;
   }
   __syncthreads();  //....here ?

} //the per-block partial sum is bowman[0]
```

Grid当中我们申请的所有线程

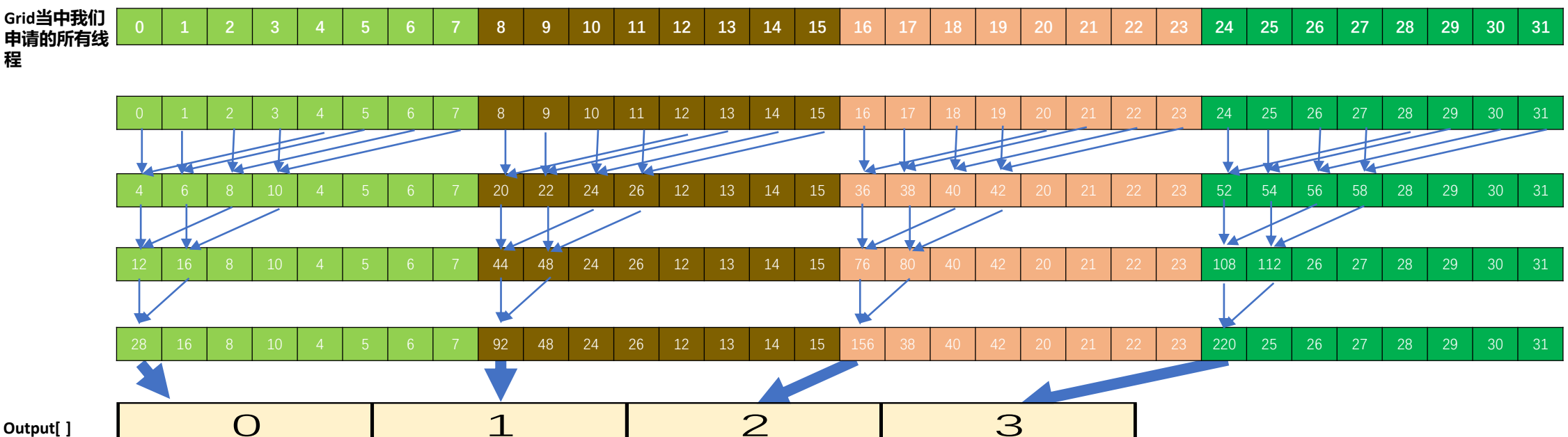
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

我们每一个线程所对应的数据就都在它所在的block中的shared memory中



1.将每个block的shared memory中第一个值bowman[0]
放在输出的向量里面。

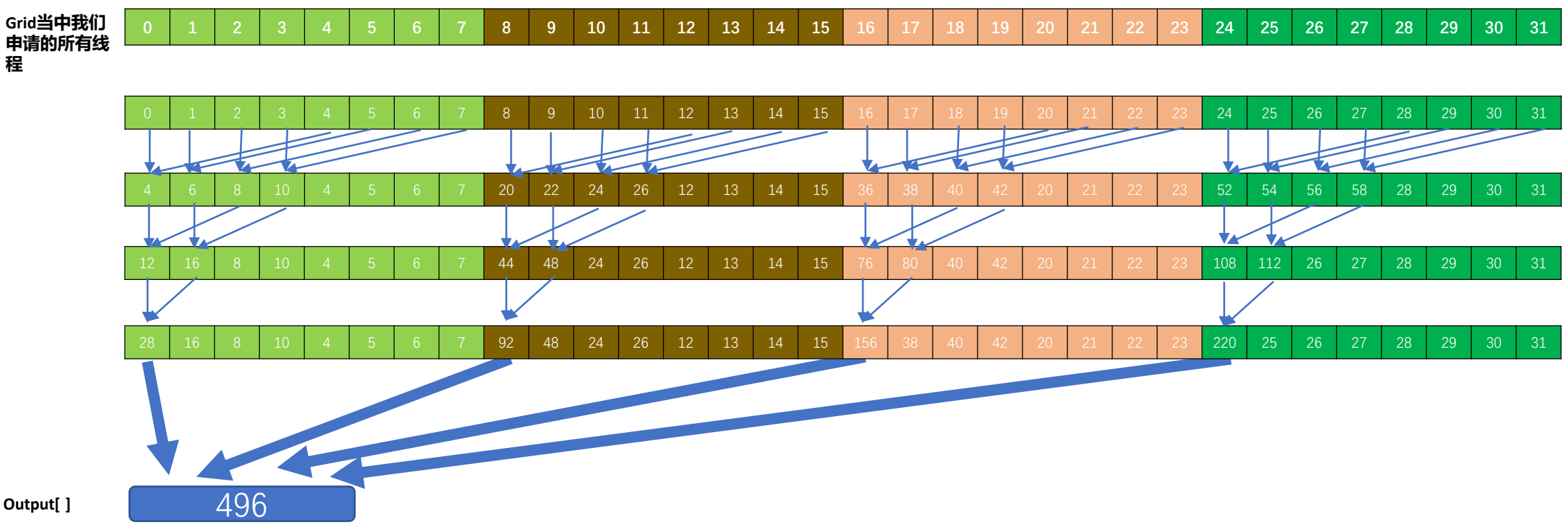
```
if (blockDim.x * blockIdx.x < count) //in case that our users are
{
    //per-block result written back, by thread 0, on behalf of a
    if (threadIdx.x == 0) output[blockIdx.x] = bowman[0];
}
```



1.如果我们能够使用一个操作，彼此之间不影响，将结果累加到output，我们就不再需要第二轮的执行了

2.这是我们就可以采用atomicAdd()

```
if (blockDim.x * blockIdx.x < count)
{
    //the final reduction performed by atomicAdd()
    if (threadIdx.x == 0) atomicAdd(output, sum_per_block[0]);
}
```



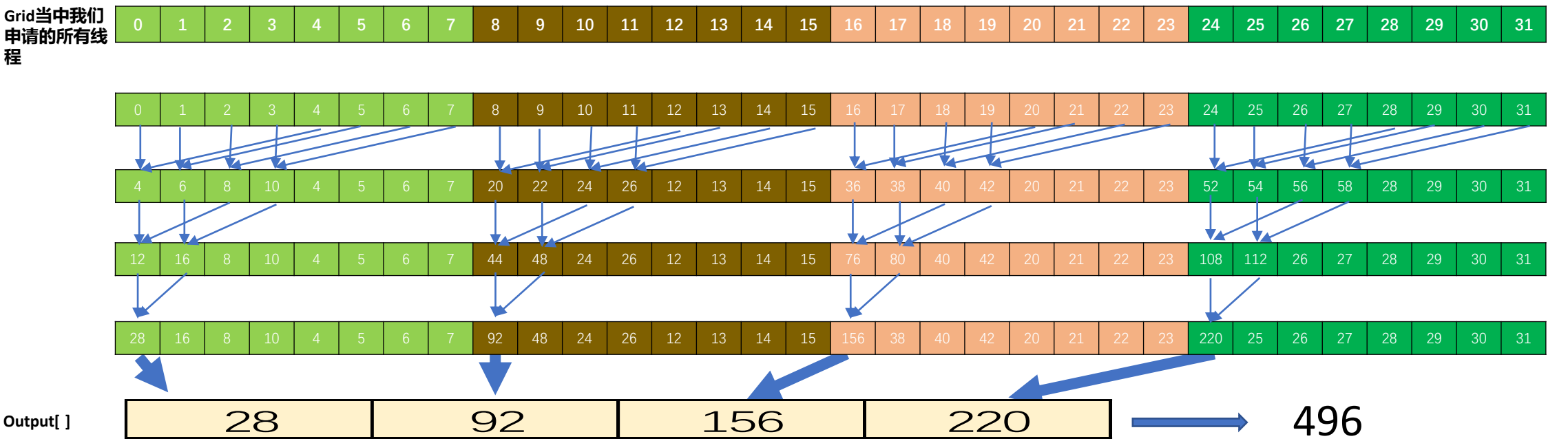
CUDA中的规约问题

如果我们不能使用原子操作

1.这里做了两步是因为，将第一次核函数执行输出的结果，当作第二次核函数执行的输入，这是只需要一个block，并且能将所有的数据放进shared memory中，重复之前的步骤，最后输出结果

2.这里如果一个线程块中的线程不够，采用的是课上讲的那个grid-loop的方法，注意第一步中那个
idx += gridDim.x * blockDim.x

```
_hawk_sum_gpu<<<BLOCKS, BLOCK_SIZE>>>(source, N, _partial_results);
    KEN_CHECK(cudaGetLastError());  //checking for launch failures

_hawk_sum_gpu<<<1, BLOCK_SIZE>>>(_partial_results, BLOCKS, final_result);
    KEN_CHECK(cudaGetLastError());  //the same
```



CUDA中的warp级方法

```
const int warpIndex = threadIdx.x / warpSize;

const int laneIndex = threadIdx.x % warpSize;
```

		laneIndex																													
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
warpIndex	1	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	2	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95

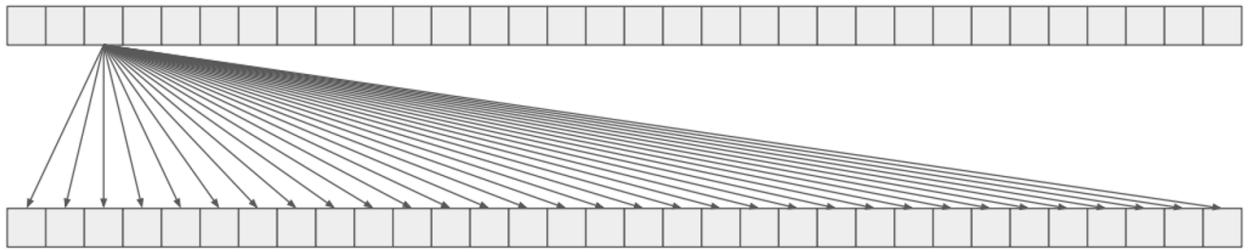
CUDA中的warp级方法

Warp shuffle是一种更快的机制，用于在相同Warp中的线程之间移动数据。

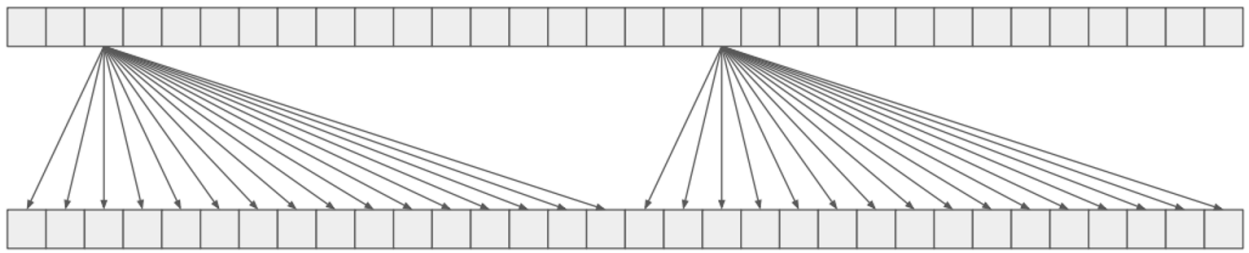
Warp Broadcast

```
T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize);
```

```
int y = __shfl_sync(0xffffffff, x, 2);
```



```
int y = __shfl_sync(0xffffffff, x, 2, 16);
```



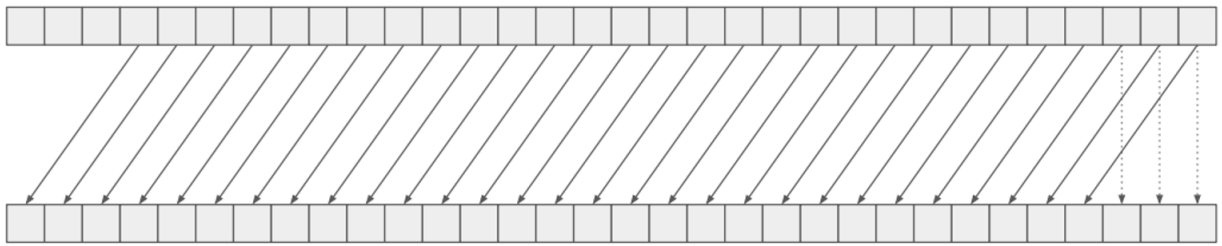
CUDA中的warp级方法

Warp shuffle是一种更快的机制，用于在相同Warp中的线程之间移动数据。

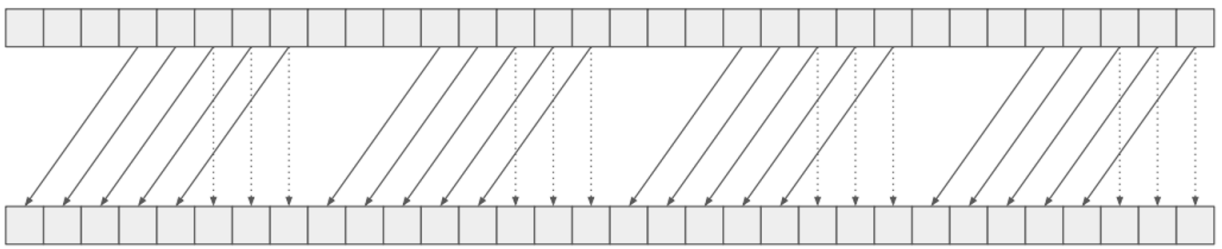
Warp Broadcast

```
T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
```

```
int y = __shfl_down_sync(0xffffffff, x, 3);
```



```
int y = __shfl_down_sync(0xffffffff, x, 3, 8);
```



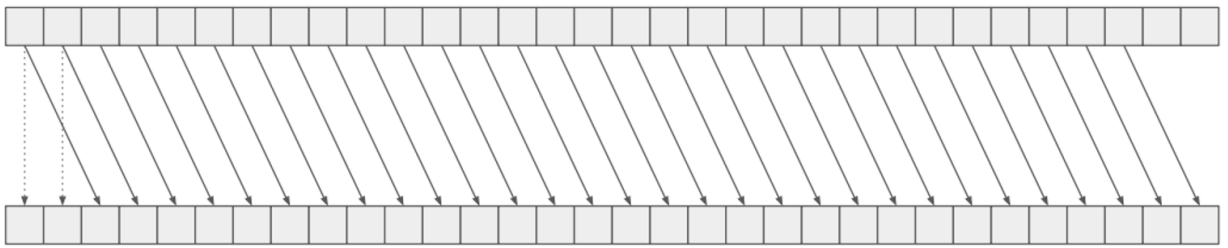
CUDA中的warp级方法

Warp shuffle是一种更快的机制，用于在相同Warp中的线程之间移动数据。

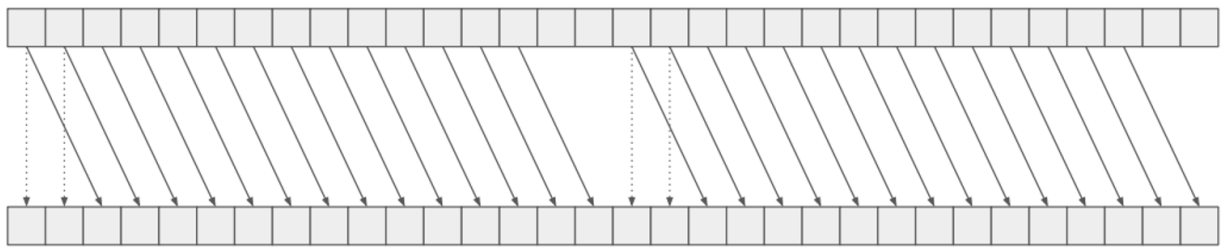
Warp Broadcast

```
T __shfl_up_sync(unsigned mask, T var, int srcLane, int width=warpSize);
```

```
int y = __shfl_up_sync(0xffffffff, x, 2);
```



```
int y = __shfl_up_sync(0xffffffff, x, 2, 16);
```



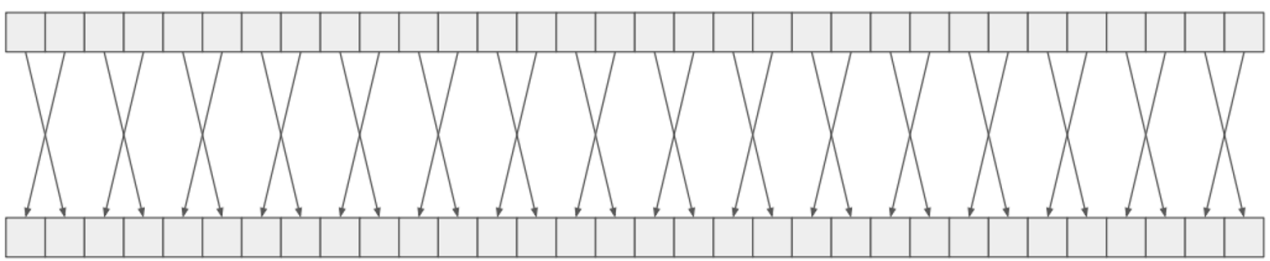
CUDA中的warp级方法

Warp shuffle是一种更快的机制，用于在相同Warp中的线程之间移动数据。

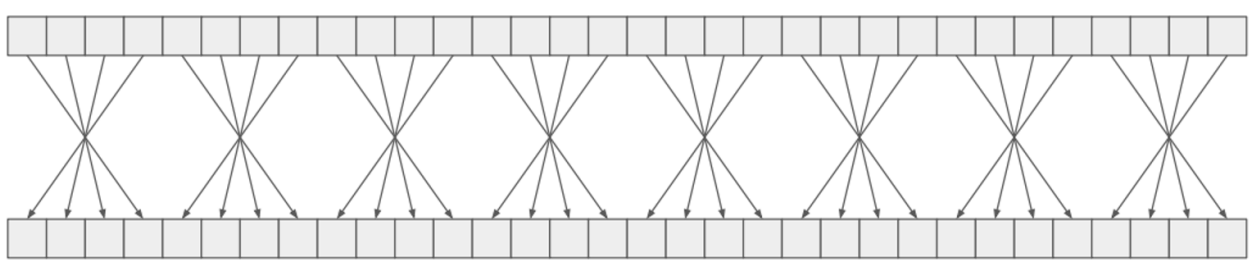
Warp Broadcast

```
T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize);
```

```
int y = __shfl_xor_sync(0xffffffff,x, 1);
```

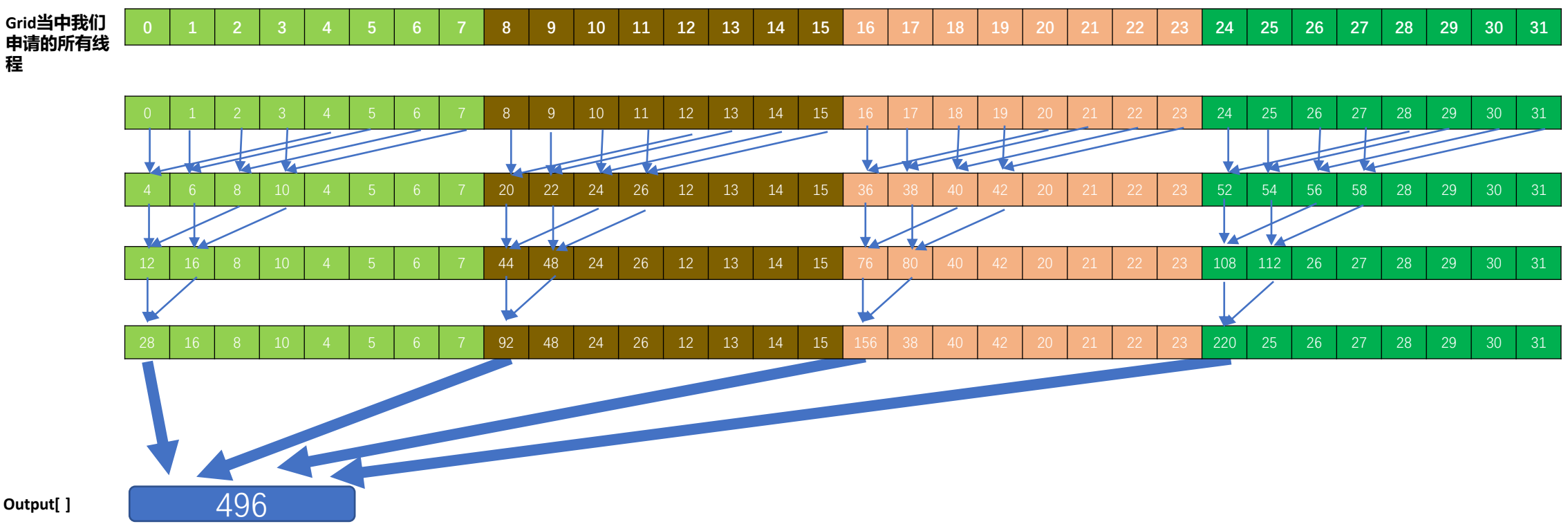


```
int y = __shfl_ xor_sync(0xffffffff,x, 3);
```



CUDA中的规约问题

1 + 2 + 3 + 4+ ..... +N



CUDA中的规约问题

1 + 2 + 3 + 4+ ……. +N

```
int val = ken[threadIdx.x];
unsigned int mask = 0xffffffff;
mask = __ballot_sync (0xffffffff, threadIdx.x < 32);
for (int offset = 16; offset > 0; offset /= 2)
{
        val += __shfl_down_sync(mask, val, offset);
}
```

更多资源:

https://developer.nvidia-china.com

何琨-Ken

北京 密云

扫一扫上面的二维码图案，加我微信

https://www.nvidia.cn/developer/community-training/

总结:

1. CUDA编程模型依赖于GPU硬件环境，不同的硬件设备，需要不同的加速手段
2. 在真正的开发过程中，其实有大量的现成的工具，希望大家能够处理一些通用问题的时候，使用现成的工具库
3. 进阶之路既有高处，也有细节。关注最新的动态，能让我们更快的掌握更好的解决问题的手段


NVIDIA