



中山大學  
SUN YAT-SEN UNIVERSITY



# 多核程序设计与实践 并行编程模式

陶钧

[taoj23@mail.sysu.edu.cn](mailto:taoj23@mail.sysu.edu.cn)

中山大学 数据科学与计算机学院  
国家超级计算广州中心

- 归约算法
- 扫描算法



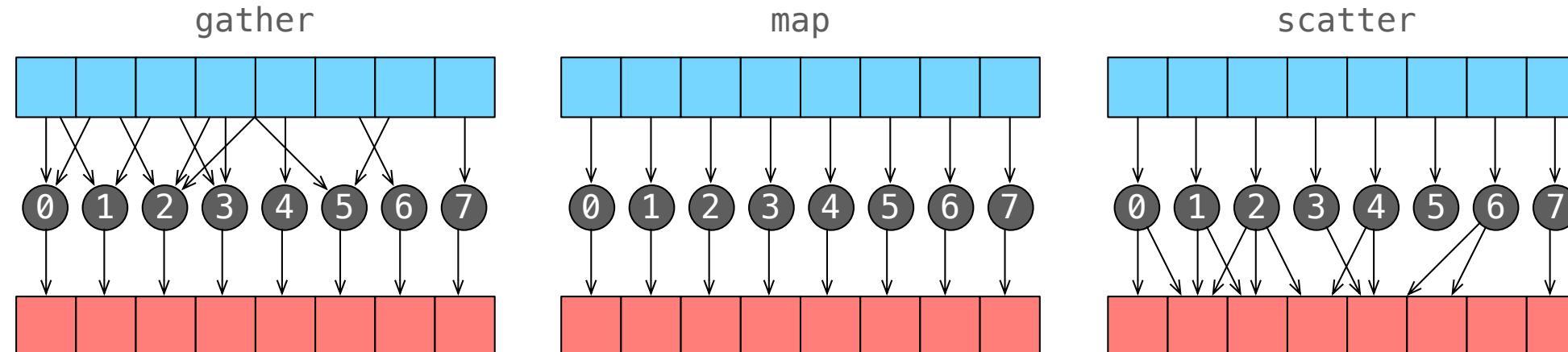
## ● 并行模式

### – 并行程序的高级抽象

- 不关心具体的数据类型或操作
- 描述并行编程中共有的模式 (pattern)
- 对比抽象代数

## ● 常见并行模式

### – gather, scatter, map, stencial, **reduce**, **scan**, sort, segmented-scan, map-reduce



## ● 归约 (reduction)

- 对传入的 $O(n)$ 个输入数据，使用二元操作符 $\oplus$ 生成 $O(1)$ 个结果
  - 二元操作符 $\oplus$ 需满足结合律
  - 例如，求最小值，最大值，求和，平方和，逻辑与，逻辑或，等
  - OpenMP中提供并行 reduction（讲义3）

```
#pragma omp for reduction(+: total)
for(int i=0; i<20; ++i){
    total += histogram[i];
}
```

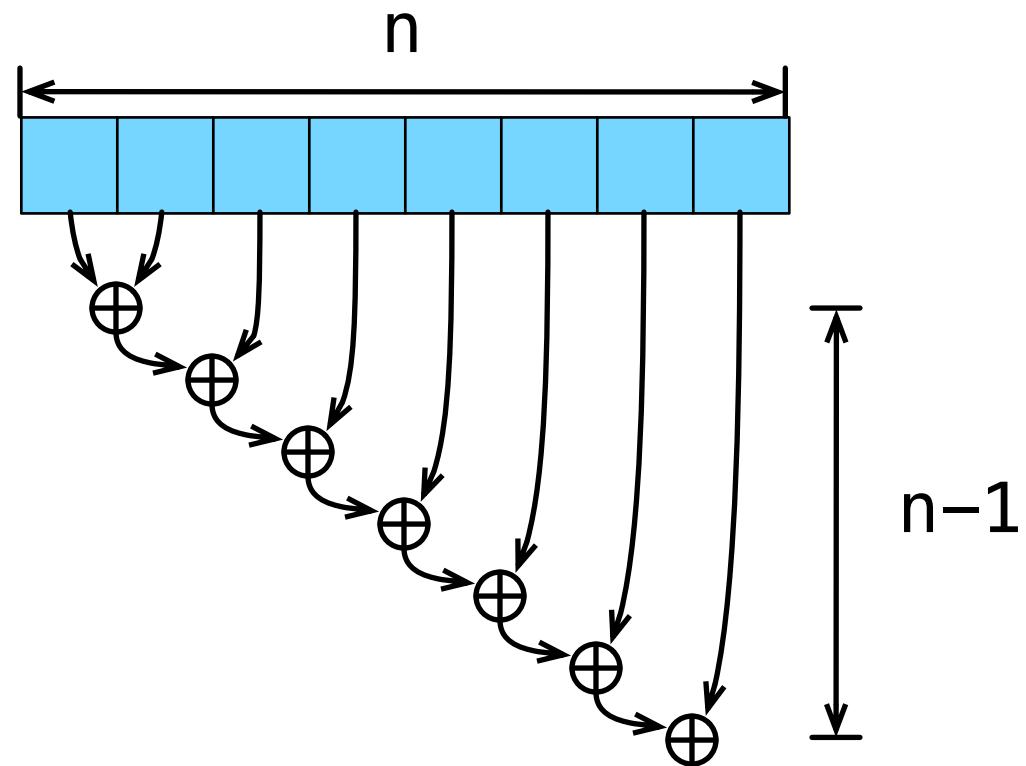


## • CPU串行归约

– 执行时间 $O(n)$

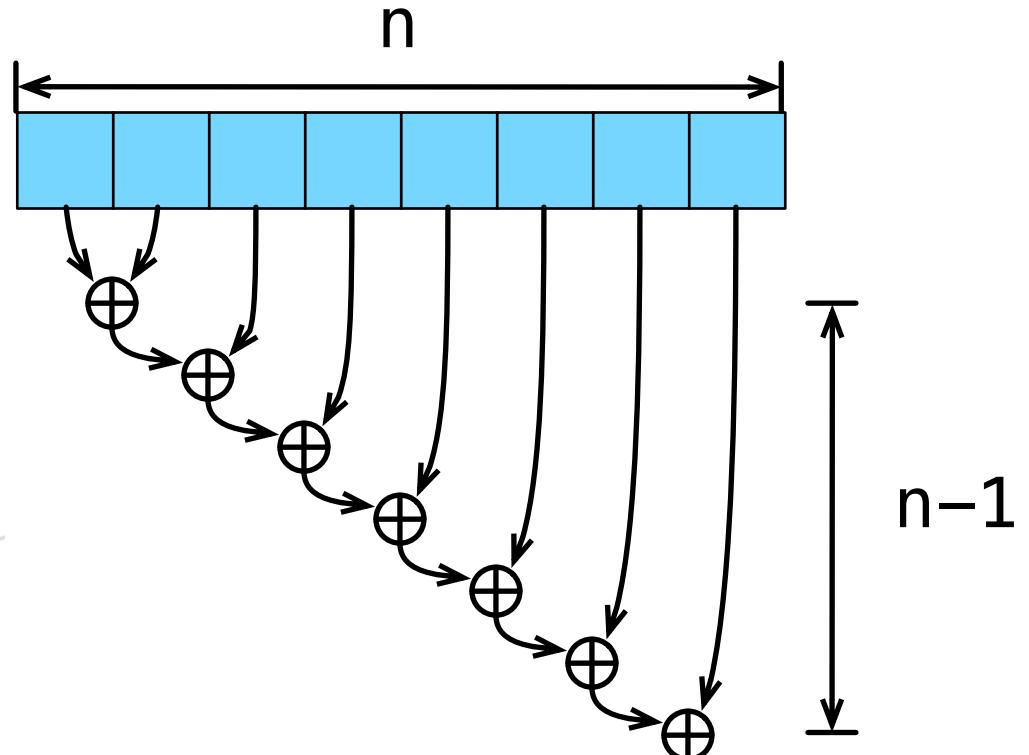
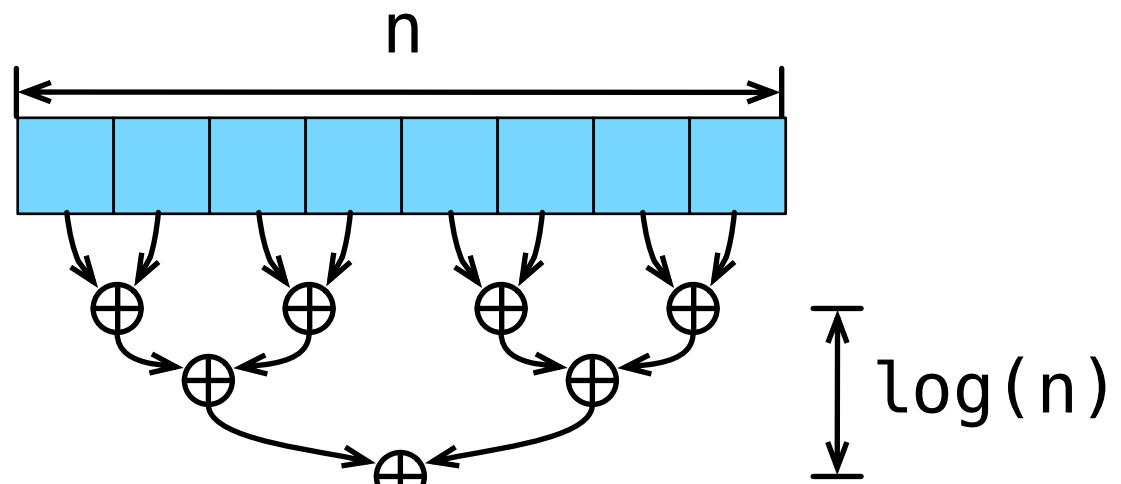
```
int data[N];
int r;

for (int i=0; i<N; ++i){
    r = reduce(r, data[i]);
}
```



## ● GPU归约：二叉树

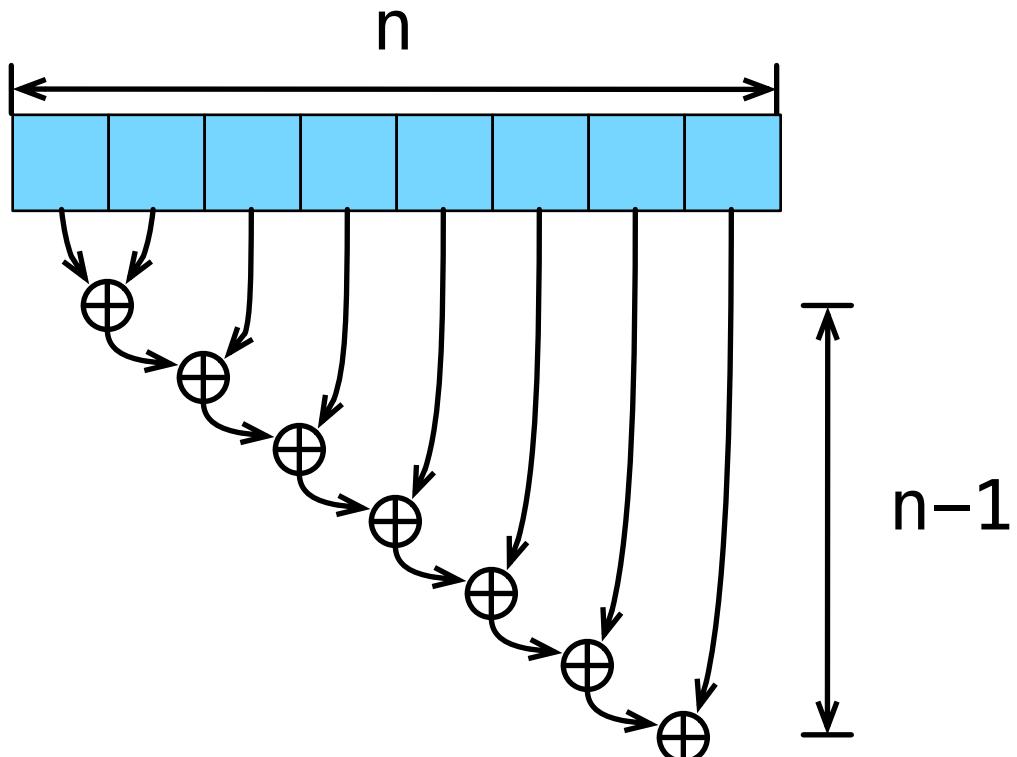
- 二元操作符  $\oplus$  需满足结合律
  - 结果与执行顺序无关
- 执行时间  $O(\lg n)$



- 例子：数组求和

- 将前例中CPU中二元操作符  $\oplus$  定义为整型相加

```
int sum(int* data){  
    int r = 0;  
    for(int i = 0; i < N; ++i){  
        r = reduce(r, data[i]);  
    }  
  
    int reduce(int r, int i){  
        return (r+i);  
    }  
}
```

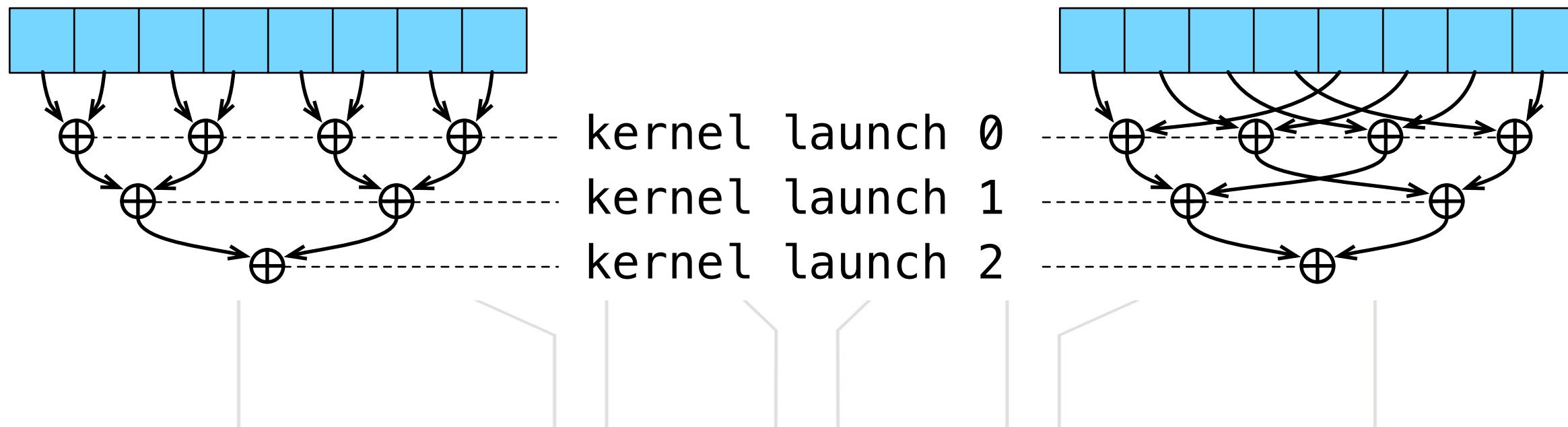


## ● 例子：数组求和

- GPU并行求和

```
__global__ void sum_reduction(float* out, float* data){  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    out[tid] = data[tid*2] + data[tid*2+1];  
}
```

out可以等于data吗？



- 例子：数组求和

- GPU并行求和

```
__global__ void sum_reduction(float* out, float* data){  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    out[tid] = data[tid*2] + data[tid*2+1];  
}
```

- 以上代码存在问题：
  - 需要在全局内存中额外分配内存存储中间结果（out）
    - » 大小为 $0.5*n+0.25*n$
    - » 全局内存访问慢
  - 线程工作量小（需产生大量线程）

## ● 例子：数组求和

## – GPU并行求和（共享内存）

```
__global__ void sum_reduction(float *out, float* in, int N){  
    extern __shared__ int sdata[];  
  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    sdata[threadIdx.x] = in[tid]; ←  
    __syncthreads();  
  
    for (int stride=1; stride<blockDim.x; stride*=2){  
        int sid = threadIdx.x*2*stride;  
        if (sid<blockDim.x){  
            sdata[sid] += sdata[sid+stride];  
        }  
        __syncthreads();  
    }  
  
    if (threadIdx.x==0){  
        out[blockIdx.x] = sdata[0];  
    }  
}
```

out大小为blockDim.x

数据量大时也可以先进行一次简单的串行归约  
for(int i = tid;  
 i < N;  
 i+=blockDim.x\*gridDim.x)  
{  
 sum += in[i];  
}  
sdata[threadIdx.x] = sum;

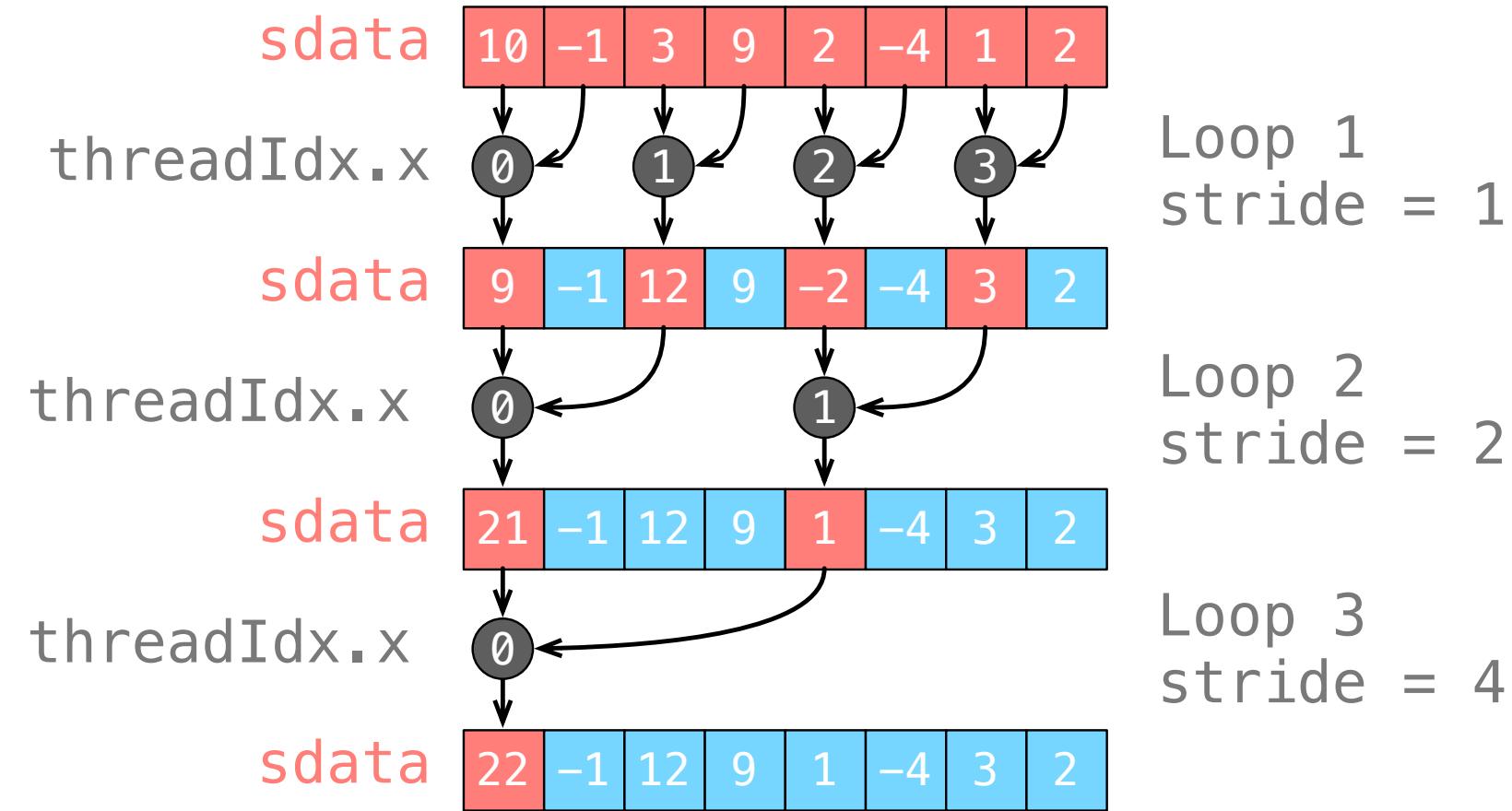
# 归约算法



- 例子：数组求和

- GPU并行求和

- 共享内存
- 线程块reduction



```
for (int stride=1; stride<blockDim.x; stride<<=1){  
    int sid = threadIdx.x*2*stride;  
    if (sid<blockDim.x){  
        sdata[sid] += sdata[sid+stride];  
    }  
    __syncthreads();  
}
```

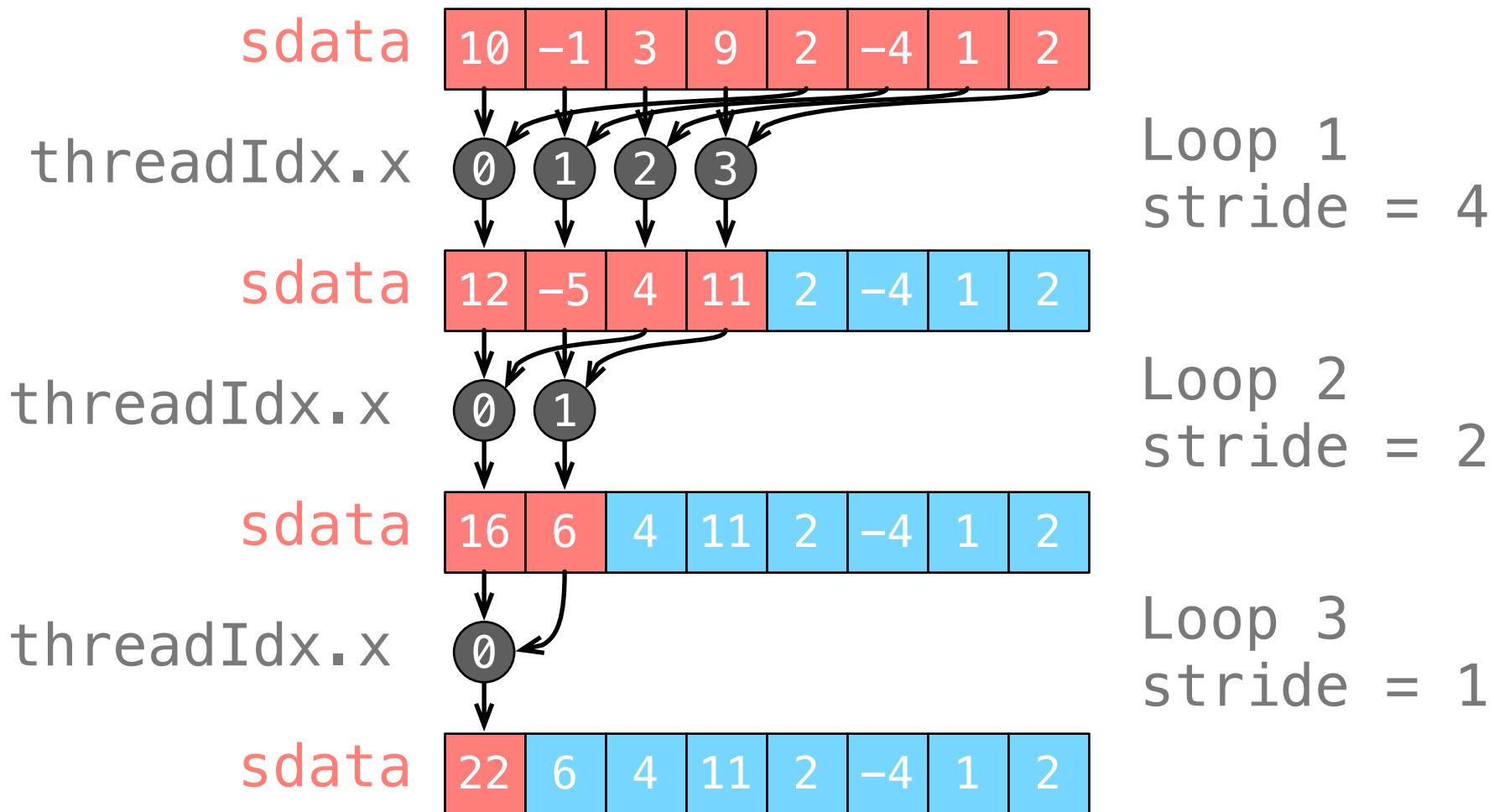
# 归约算法



- 例子：数组求和

- GPU并行求和

- 共享内存
- 线程块reduction



```
for (int stride=blockDim.x/2; stride>0; stride>>=1){  
    if (threadIdx.x<stride){  
        sdata[threadIdx.x] += sdata[threadIdx.x+stride];  
    }  
    __syncthreads();  
}
```

## ● 例子：数组求和

## – GPU并行求和

- 注意：以下两种写法哪种正确？

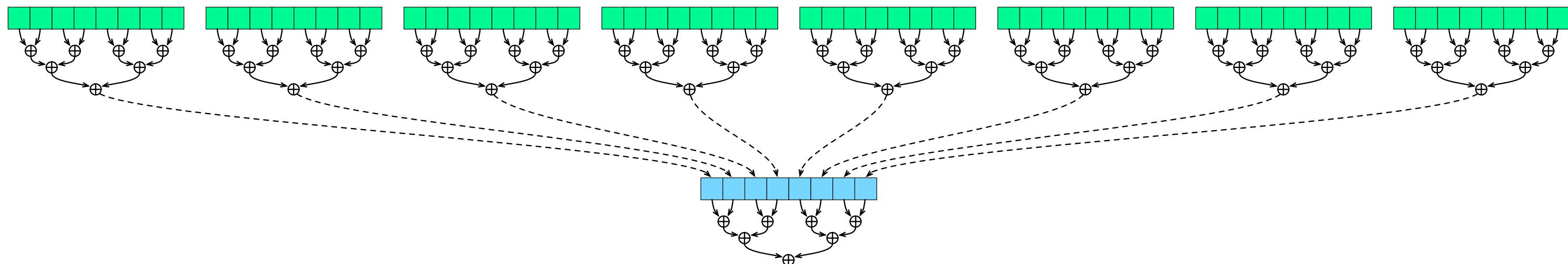
```
for (int stride=blockDim.x/2; stride>0; stride>>=1){  
    if (threadIdx.x<stride){  
        sdata[threadIdx.x] += sdata[threadIdx.x+stride];  
    }  
    __syncthreads();  
}
```

```
for (int stride=blockDim.x/2; stride>0; stride>>=1){  
    if (threadIdx.x<stride){  
        sdata[threadIdx.x] += sdata[threadIdx.x+stride];  
        __syncthreads();  
    }  
}
```

- 例子：数组求和

- GPU并行求和

- 对线程块产生的结果迭代调用归约核函数
    - 当n不大时，用CPU串行计算可能更有效率
      - 减少创建线程、同步等开销
    - 也可以使用**atomicAdd**完成归约
      - 只需调用一次核函数（或数次无atomic的核函数加一次带atomic的核函数）



- 例子：数组求和
  - GPU并行求和
    - 使用**atomicAdd**完成归约

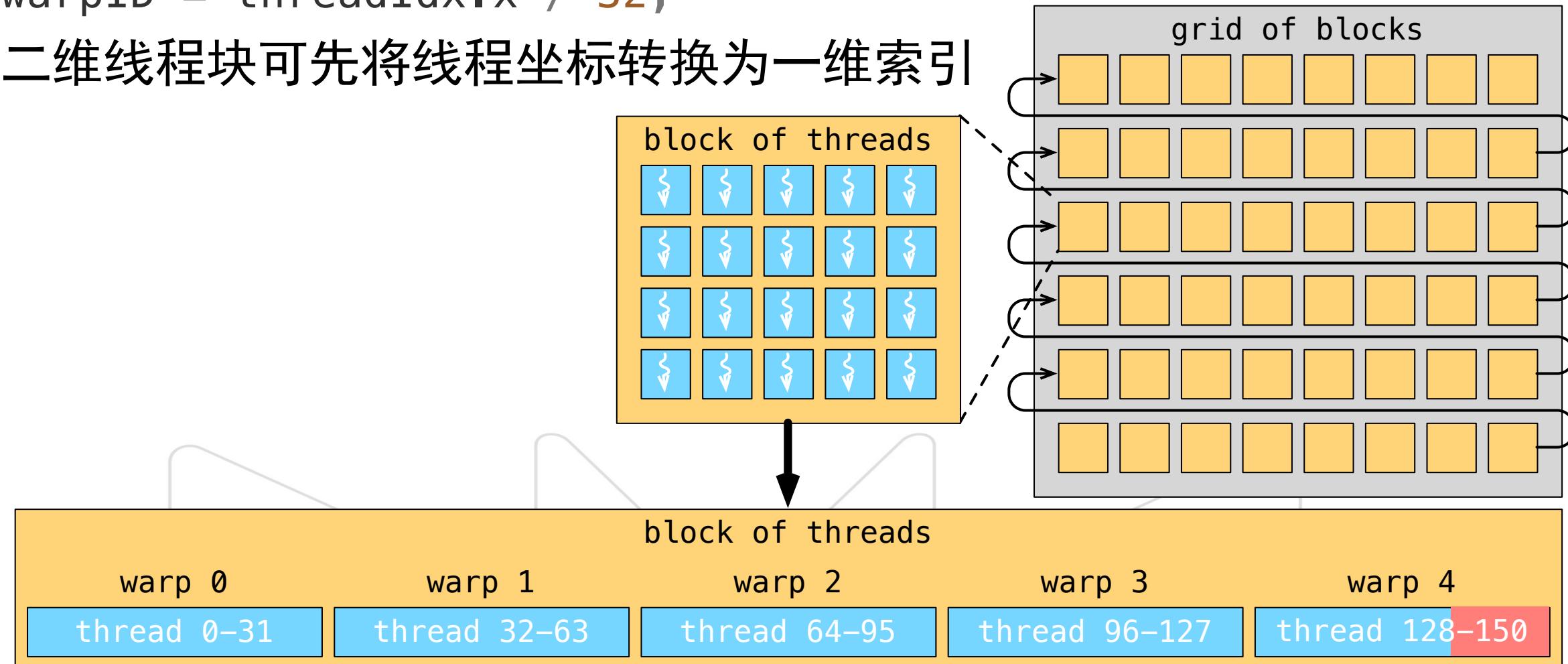
```
__global__ void sum_reduction(float *out, float* in){  
    extern __shared__ int sdata[];  
  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    sdata[threadIdx.x] = in[tid];  
    __syncthreads();  
  
    for (int stride=blockDim.x/2; stride>0; stride>>=1){  
        if (threadIdx.x<stride){  
            sdata[threadIdx.x] += sdata[threadIdx.x+stride];  
        }  
        __syncthreads();  
    }  
  
    if (threadIdx.x==0)  
        atomicAdd(out, sdata[0]);  
}
```

- 使用线程束洗牌指令
  - 共享内存使线程块中线程快速交换数据
    - 必要时需要通过 `_syncthreads()` 同步
  - 洗牌指令使线程束中线程直接交换数据
    - 允许线程直接读取束内另一线程的寄存器
    - 可取代atomic操作
      - 不可置于控制流分支分流的语句中（参见课件6）
    - 不通过共享内存或全局内存
      - 比共享内存延迟更低
      - 交换不消耗额外内存
    - 隐式同步（由线程束执行模式决定）
      - 不需要使用 `_syncthreads()` 同步
    - 需要计算能力3.0或以上（Kepler或之后架构）

- 使用线程束洗牌指令

- 束内线程 (lane)

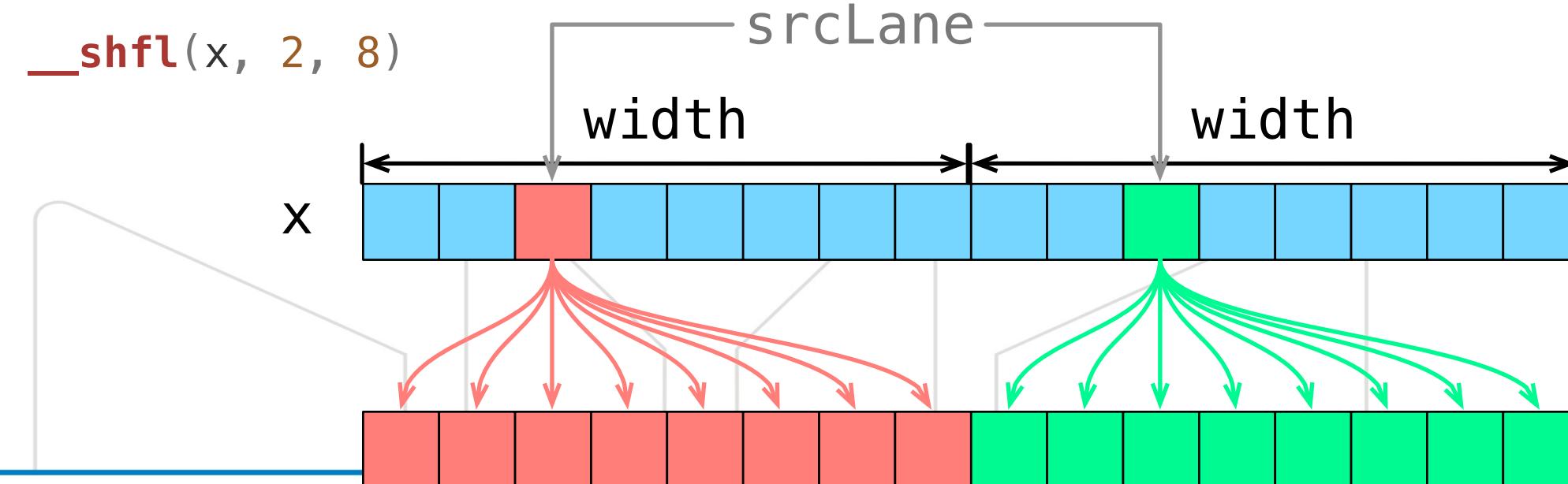
- laneID = threadIdx.x % 32;
    - warpID = threadIdx.x / 32;
    - 二维线程块可先将线程坐标转换为一维索引



## ● 使用线程束洗牌指令

### – 四种形式的洗牌指令

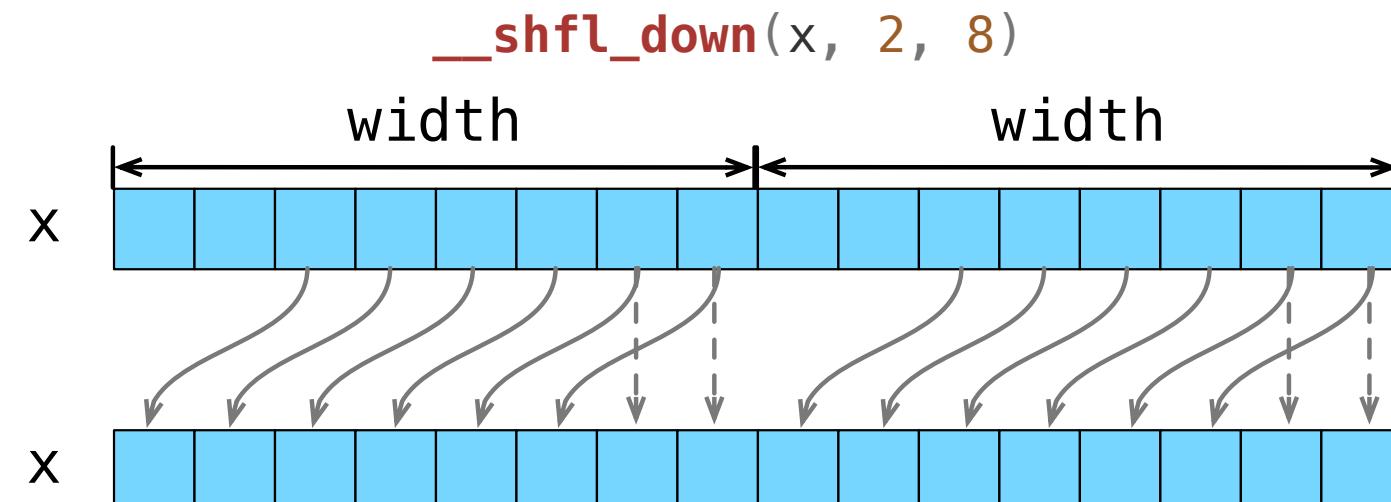
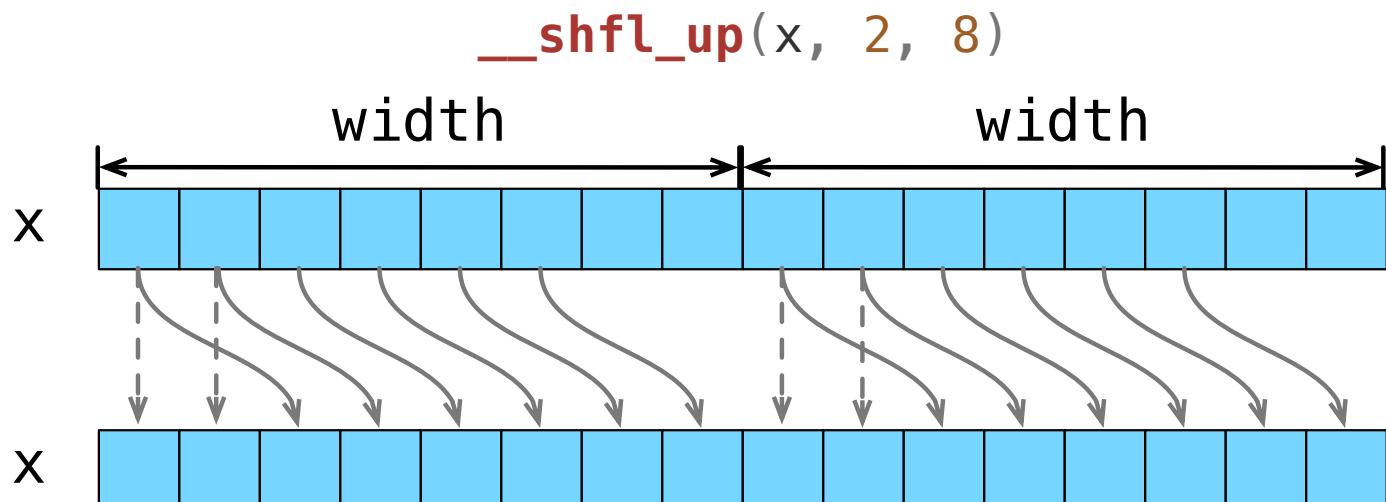
- `_shfl()`, `_shfl_up()`, `_shfl_down()`, `_shfl_xor()`
- 支持`int`, `float`, `half`
- T `_shfl(T var, int srcLane, int width=warpSize)`
  - var: 需要交换的变量
  - srcLane: 数据来源的束内线程 (具体含义依赖于width)
  - Width: 交换数据的线程束的分段宽度 (需要是不超过32的2的指数)
  - 如, `_shfl(x, 2, 8)`



- 使用线程束洗牌指令

- 四种形式的洗牌指令

- T **\_shfl\_up**(T var, int delta, int width=warpSize)
    - T **\_shfl\_down**(T var, int delta, int width=warpSize)
    - delta: 移位的偏移量



## ● 使用线程束洗牌指令

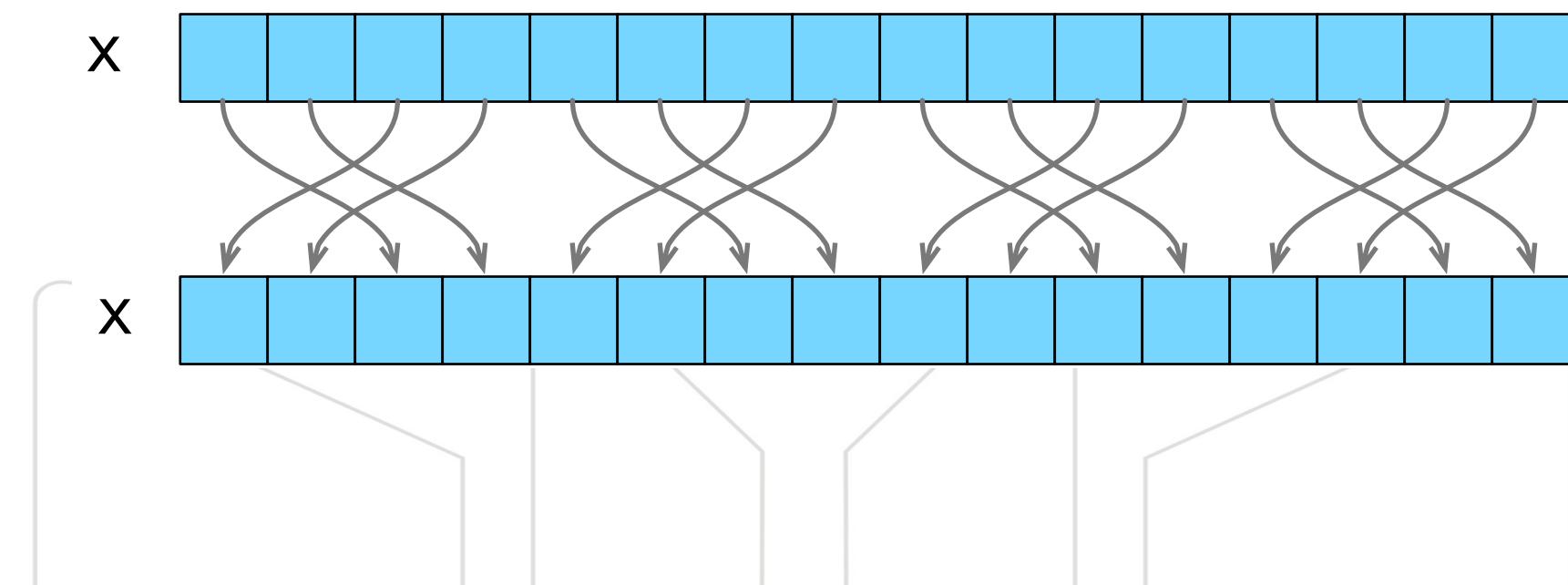
### – 四种形式的洗牌指令

- T **\_shfl\_xor**(T var, int laneMask, int width=warpSize)

- 根据束内线程自身索引指按位异或来传输数据（蝴蝶交换）
- laneMask: 指明异或的bit

» **\_shfl(x, 2)**

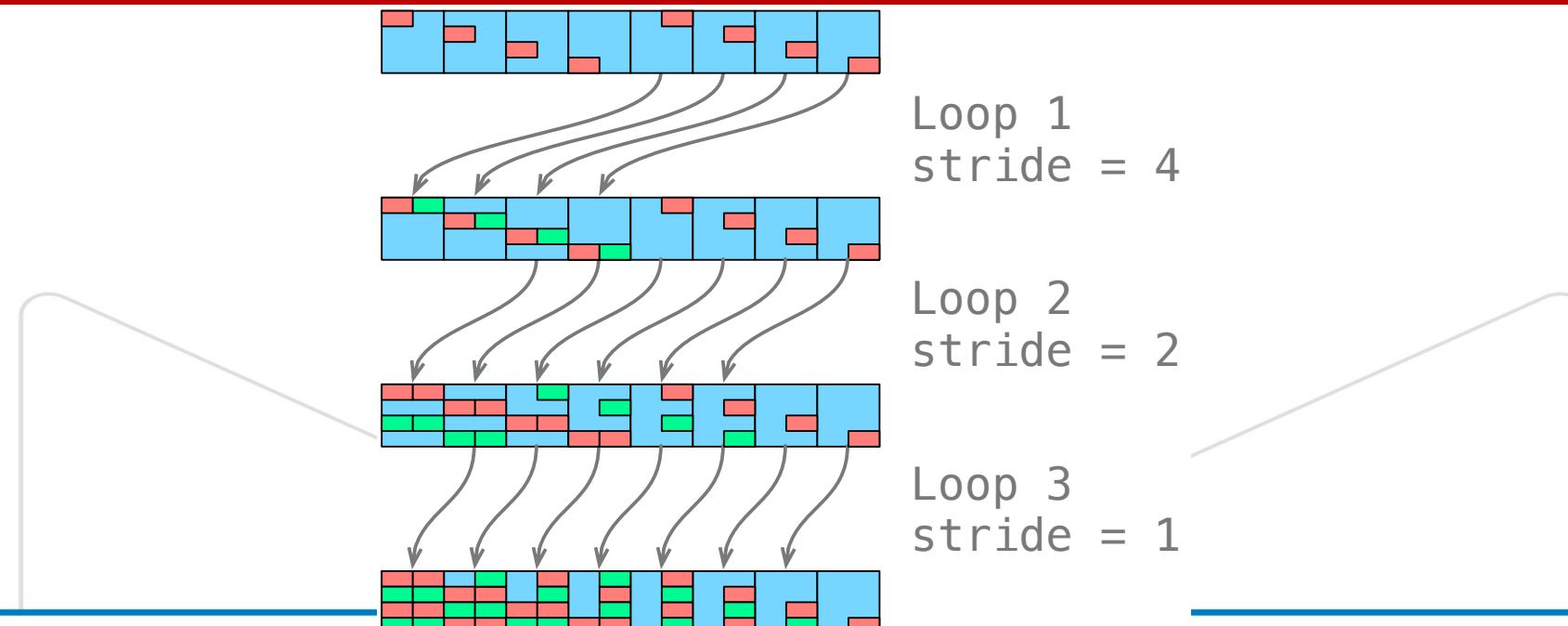
- 指明用于求异或的bit为010b
- 0 (000b) 与 2 (010b) 交换, 1 (001b) 与 3 (011b) 交换, 以此类推



## ● 使用线程束洗牌指令

### – 使用 `_shfl_down()` 的数组求和

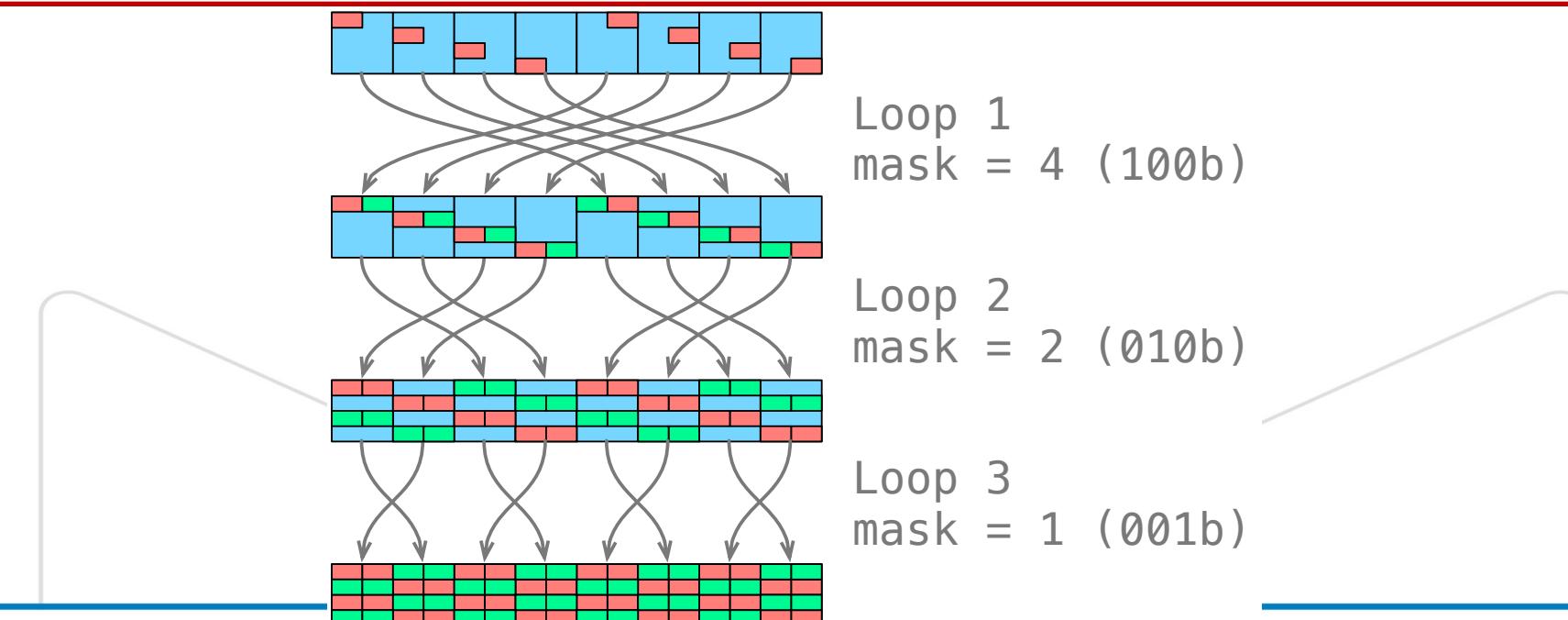
```
__global__ void sum_reduction_warp(int* out, int* data, int N){  
    int local_sum = data[blockIdx.x*blockDim.x+threadIdx.x];  
  
    for (int stride = WARP_SIZE/2; stride>0; stride>>=1)  
        local_sum += _shfl_down(local_sum, stride);  
  
    if (threadIdx.x%32==0)  
        out[blockIdx.x*blockDim.x/WARP_SIZE+threadIdx.x/WARP_SIZE];  
}
```



- 使用线程束洗牌指令

- 使用 `_shfl_xor()` 的数组求和

```
__global__ void sum_reduction_warp(int* out, int* data, int N){  
    int local_sum = data[blockIdx.x*blockDim.x+threadIdx.x];  
  
    for (int mask = WARP_SIZE/2; mask >0; mask >>=1)  
        local_sum += _shfl_xor(local_sum, mask);  
  
    if (threadIdx.x%32==0)  
        out[blockIdx.x*blockDim.x/WARP_SIZE+threadIdx.x/WARP_SIZE];  
}
```



- 归约算法
- 扫描算法



- 以下名称通常均指扫描算法

- 扫描 (scan)、前缀扫描 (prefix scan)、前缀求和 (prefix sum)、并行前缀求和 (parallel prefix sum)

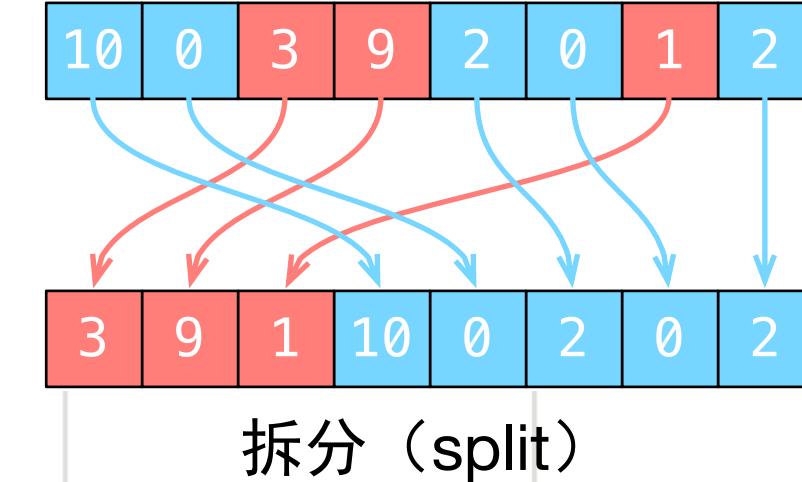
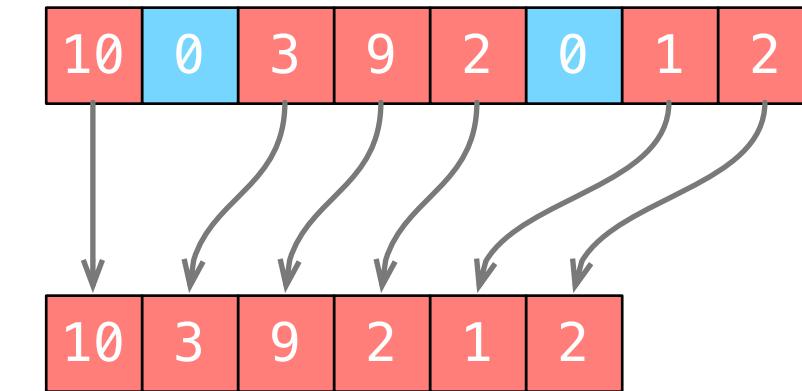
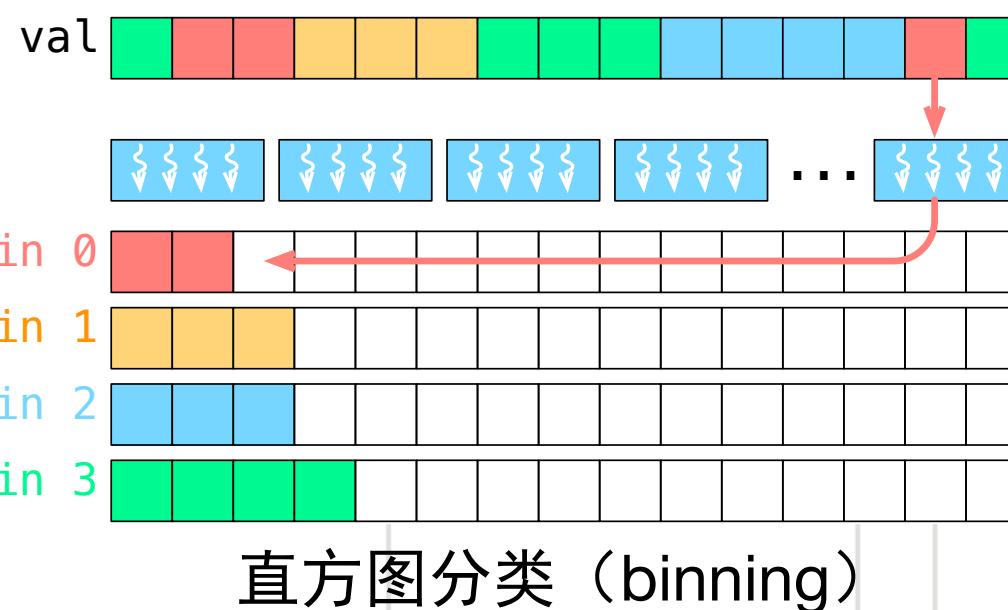
- 扫描是许多并行算法的基本组成模块

- 基数排序 (radix sort)
  - 快速排序 (quick sort)
  - 流压缩 (stream compaction) 和流拆分 (stream split)
  - 计算直方图 (histogram)
  - 尽管扫描在串行算法中往往 是不必要的！

## ● 为什么需要扫描？

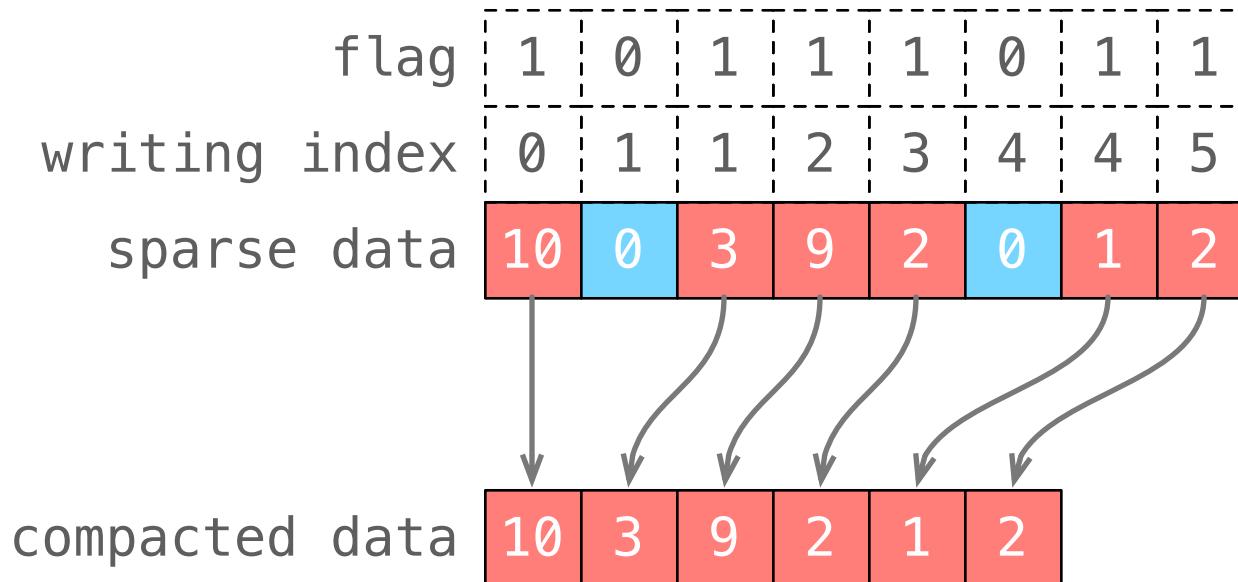
– 以下算法都面临同一个问题：数据输出的位置

- 直方图分类例子
  - 分配足够大的空间
  - 使用atomic决定输出位置
- 存在问题：
  - 浪费空间、不保留元素在原数组中顺序



## ● 使用扫描解决输出位置问题

– 以压缩为例



– 输出位置 (`writing index`) 可由对数组 `flag` 的前缀求和得到

- 前缀求和定义

- 对于给定数组  $A = [a_0, a_1, \dots, a_{n-1}]$  及二元操作符  $\oplus$ , 求:

$$\text{Scan}(A) = [0, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-1})]$$

## ● 串行前缀求和

– 例子：给定数组  $A = [2, 4, 5, 1, 3]$  并使用整型求和

- $\text{Scan}(A) = [0, 2, 2+4, 2+4+5, 2+4+5+1]$
- $\text{Scan}(A) = [0, 2, 6, 11, 12]$

– 代码样例：

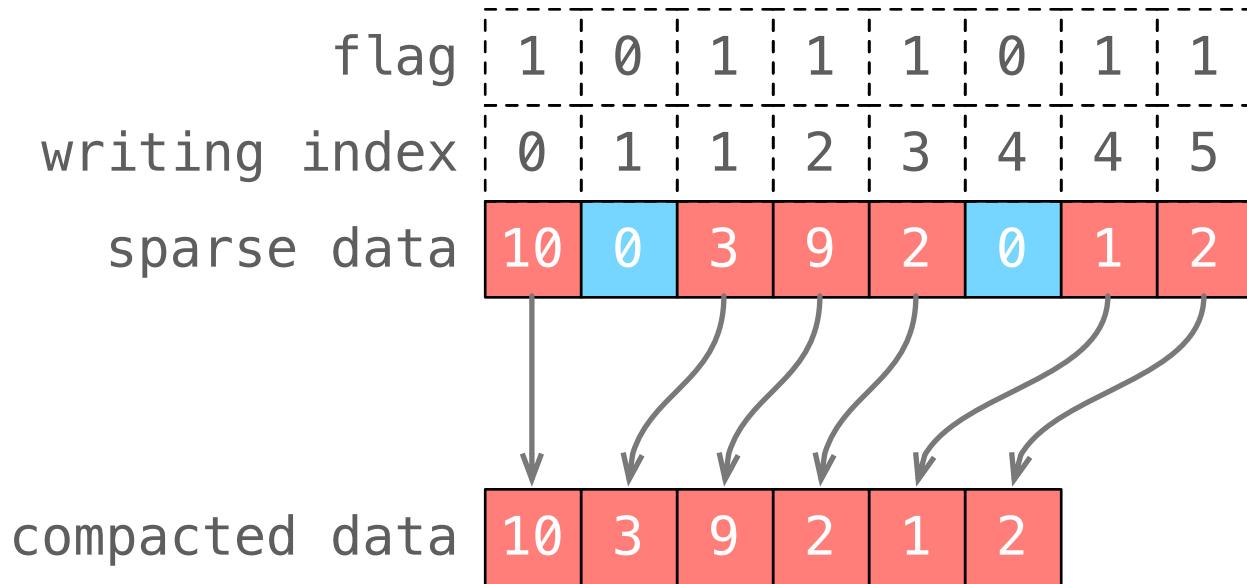
```
int A[5] = {2, 4, 5, 1, 3}
int scan_A[5];

int running_sum = 0;
for(int i = 0; i < 5; ++i){
    scan_A[i] = running_sum;
    running_sum += A[i];
}
```

- 使用扫描解决压缩算法输出位置问题

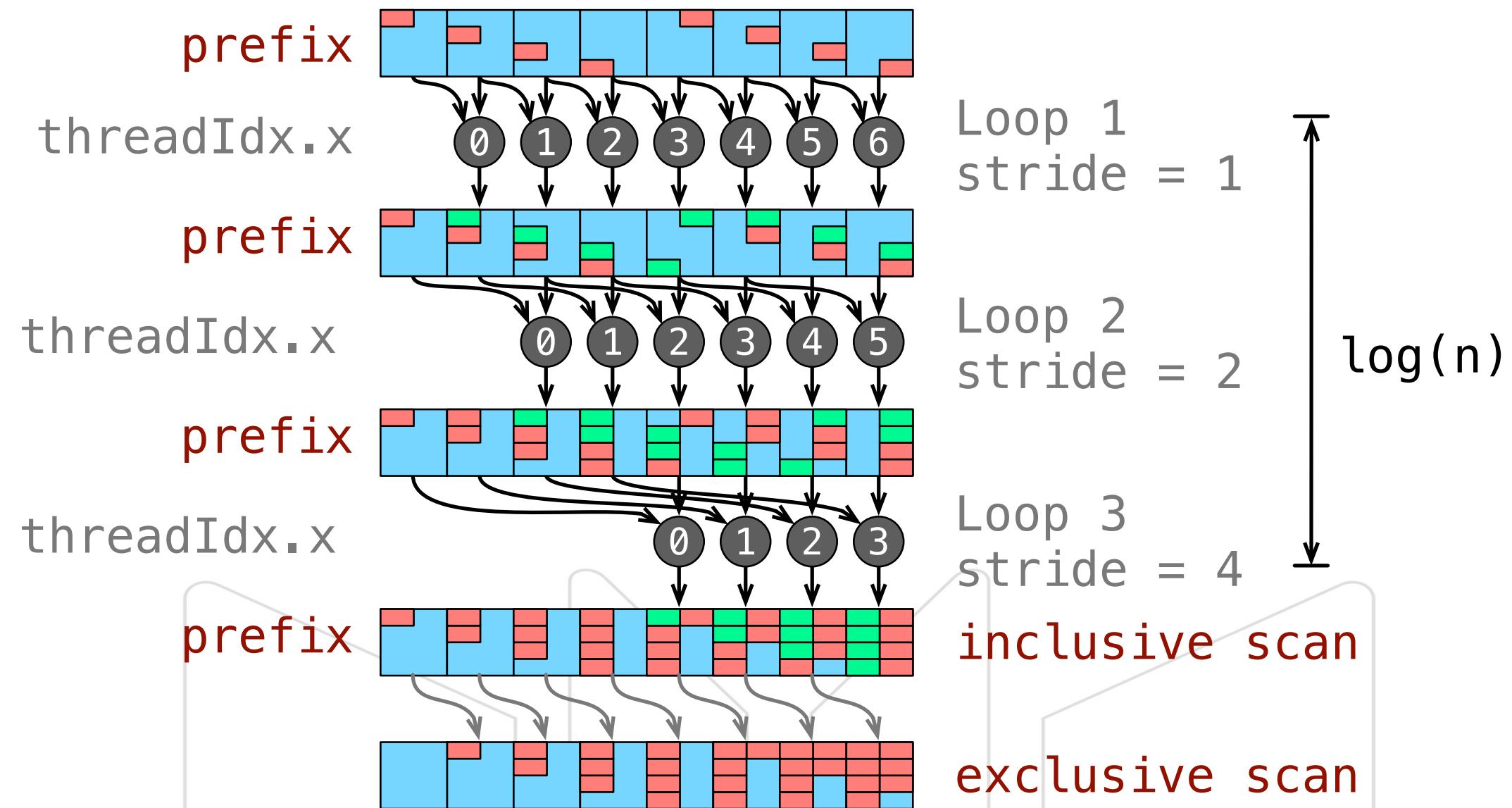
- 串行示意代码

```
int data[8] = {10, 0, 3, 9, 2, 0, 1, 2};  
int flag[8];  
int writing_index[8];  
int running_sum = 0;  
  
for(int i = 0; i < 8; ++i){  
    flag[i] = (data[i] != 0);  
}  
  
for(int i = 0; i < 8; ++i){  
    writing_index[i] = running_sum;  
    running_sum += flag[i];  
}  
  
int* compacted_data = new int[running_sum];  
for(int i = 0; i < 8; ++i){  
    if (flag[i]){  
        compacted_data[writing_index[i]] = data[i];  
    }  
}
```



- 并行扫描算法（线程块）

- 包容性扫描与排他性扫描



- 并行扫描算法（线程块）

- 使用 `_syncthreads()` 同步线程块内的线程
- 无存储体冲突

```
__global__ void scan(int* data){  
    extern __shared__ int s_data[];  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    s_data[threadIdx.x] = data[tid];  
  
    for(int stride = 1; stride < blockDim.x; stride<<=1){  
        __syncthreads();  
        int val = (threadIdx.x>=stride)?s_data[threadIdx.x-stride]:0;  
        __syncthreads();  
        s_data[threadIdx.x] += val;  
    }  
    //output  
}
```

## ● 并行扫描算法（线程块）

- 使用 `__syncthreads()` 同步线程块内的线程
- 无存储体冲突

```
__global__ void scan(int* data){  
    extern __shared__ int s_data[];  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    s_data[threadIdx.x] = data[tid];  
  
    for(int stride = 1; stride < blockDim.x; stride<<=1){  
        __syncthreads();  
        int val = (threadIdx.x>=stride)?s_data[threadIdx.x-stride]:0;  
        __syncthreads();  
        s_data[threadIdx.x] += val;  
    }  
    //output  
}
```

是否可用以下代码替代？

```
if (threadIdx.x>=stride){  
    s_data[threadIdx.x]+=s_data[threadIdx.x-stride];  
}
```

## ● 并行扫描算法（线程块）

- 使用 `__syncthreads()` 同步线程块内的线程
- 无存储体冲突

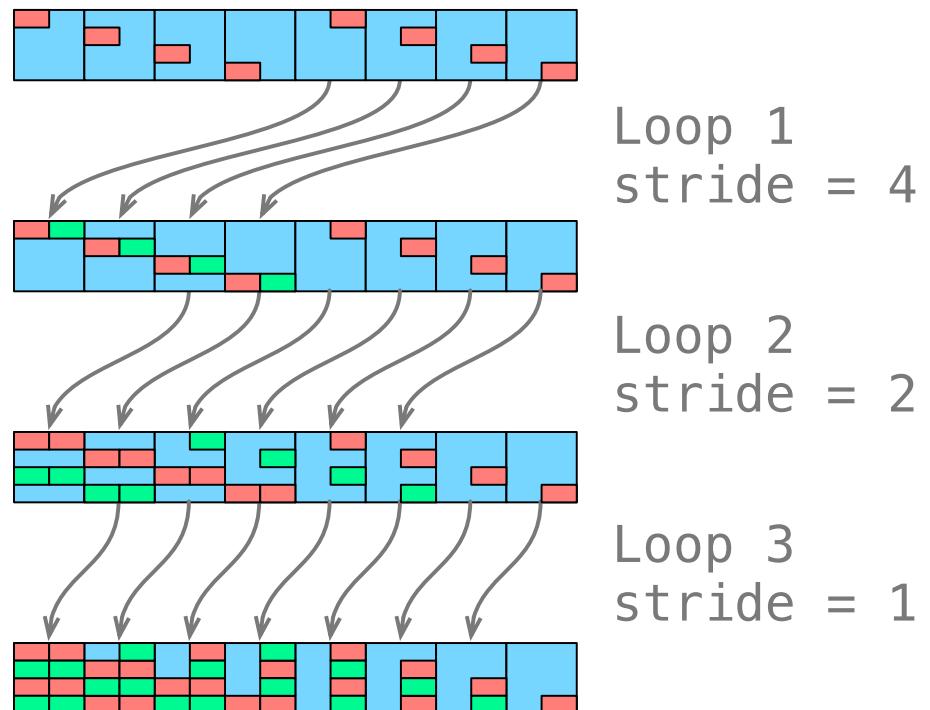
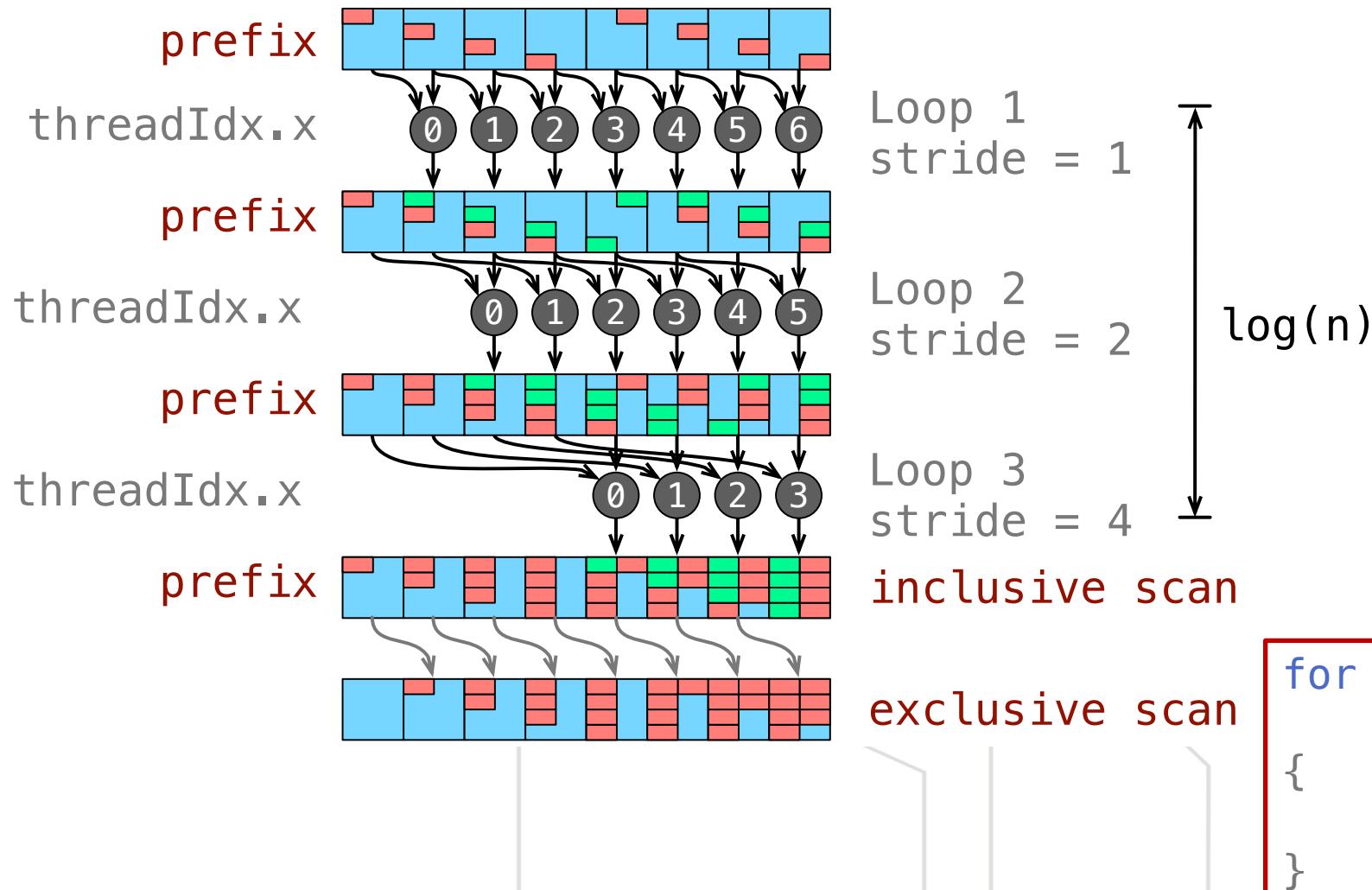
```
__global__ void scan(int* data){  
    extern __shared__ int s_data[];  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    s_data[threadIdx.x] = data[tid];  
  
    for(int stride = 1; stride < blockDim.x; stride<<=1){  
        __syncthreads();  
        int val = (threadIdx.x>=stride)?s_data[threadIdx.x-stride]:0;  
        __syncthreads();  
        s_data[threadIdx.x] += val;  
    }  
    //output  
}
```

是否可用以下代码替代？

```
if (threadIdx.x>=stride){  
    s_data[threadIdx.x]+=s_data[threadIdx.x-stride];  
    __syncthreads();  
}
```

## ● 并行扫描算法（线程束）

- 对比扫描算法与之前的使用线程束的归约算法
  - stride变化的顺序是否重要？



```

for (int stride = WARP_SIZE/2;
    stride > 0; stride>>=1)
{
    local_sum += __shfl_down(local_sum, stride);
}
  
```

## ● 并行扫描算法（线程束）

- 不需要分配共享内存
- 不需要同步

```
__global__ void scan(int* data){  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    int val1 = data[tid], val2;  
  
    for(int stride = 1; stride < 32; stride<<=1){  
        val2 = __shfl_up(val1, stride);  
        if (threadIdx.x % 32 >= stride)  
            val1 += val2;  
    }  
    //output  
}
```

## ● 并行扫描算法（线程束）

- 不需要分配共享内存
- 不需要同步

```
__global__ void scan(int* data){  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    int val1 = data[tid], val2;  
  
    for(int stride = 1; stride < 32; stride<<=1){  
        val2 = __shfl_up(val1, stride);  
        if (threadIdx.x % 32 >= stride)  
            val1 += val2;  
    }  
    //output  
}
```

是否可用以下代码替代？

```
if (threadIdx.x % 32 >= stride){  
    val1 += __shfl_up(val1, stride);  
}
```

## ● 并行扫描算法（线程束）

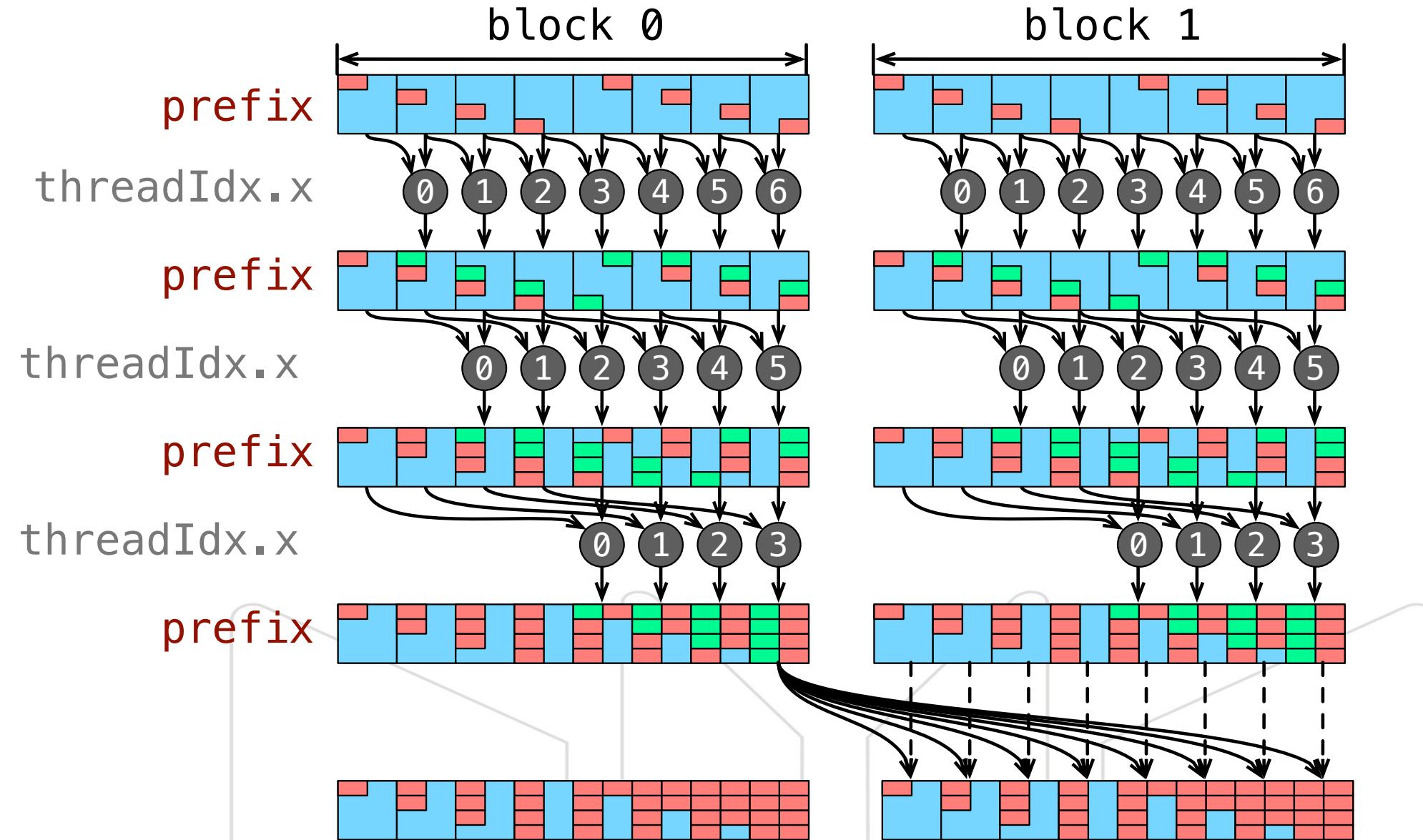
- 不需要分配共享内存
- 不需要同步

```
__global__ void scan(int* data){  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    int val1 = data[tid], val2;  
  
    for(int stride = 1; stride < 32; stride<<=1){  
        val2 = __shfl_up(val1, stride);  
        if (threadIdx.x % 32 >= stride)  
            val1 += val2;  
    }  
    //output  
}
```

CUDA 9.0后可使用  
`__shfl_up_sync(mask, val1, stride);`  
mask指明参与交换的线程  
0xffffffff表明所有线程都参与交换

## ● 并行扫描算法（全局）

- 需要将每个线程块的局部总和加到其后的所有线程块的计算结果中



## ● 并行扫描算法（全局）

- 需要将每个线程块的局部总和加到其后的所有线程块的计算结果中
  - 同样需要一次前缀求和
  - 可迭代调用前缀求和核函数完成
  - 使用原子操作完成串行前缀求和

```
__global__ void scan(int* out, int* block_sums, int* data){  
    extern __shared__ int s_data[];  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    s_data[threadIdx.x] = data[tid];  
  
    for(int stride = 1; stride < blockDim.x; stride<<=1){  
        __syncthreads();  
        int val = (threadIdx.x>=stride)?s_data[threadIdx.x-stride]:0;  
        __syncthreads();  
        s_data[threadIdx.x] += val;  
    }  
  
    out[tid] = s_data[threadIdx.x];  
    if (threadIdx.x==0)  
        for(int i=blockIdx.x+1; i<gridDim.x; ++i)  
            atomicAdd(&block_sums[i], s_data[blockDim.x-1])  
}
```

## ● 并行扫描算法（全局）

- 需要将每个线程块的局部总和加到其后的所有线程块的计算结果中
  - 使用原子操作：更新局部结果为全局结果
    - 加入block\_sums

```
__global__ void scan_update(int* out, int* block_sums){  
    extern __shared__ int block_sum;  
    int idx = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if (threadIdx.x == 0)  
        block_sum = block_sums[blockIdx.x];  
  
    __syncthreads();  
  
    out[idx] += block_sum;  
}
```

- 为什么要学习并行模式？
- 两种常见的并行模式
  - 归约：将数组中数据归约为一个元素
    - 迭代调用核函数
    - 在核函数内添加循环完成
      - 减少数据在缓存与全局内存中的移动
  - 扫描：前缀求和
    - 多种算法的基础
    - 同样可以迭代调用核函数或在核函数内添加循环完成
  - 高效的程序需要从网格、线程块、线程束三个层面进行优化
  - 原子操作可用于对线程块的结果进行归约/扫描

# Questions?

