

# 第6讲 并行计算模式（扫描）



前缀和

低效扫描内核

高效扫描内核

并行扫描的延伸讨论



# 目标

- 掌握并行扫描（前缀和）算法
  - 常用于并行工作分配和资源分配
  - **许多将串行计算转换为并行计算的并行算法的关键原型（key primitive）**
  - 基本的并行计算模式
  - 高效的并行编码/算法
- **阅读材料-** Mark Harris, Parallel Prefix Sum with CUDA
  - [http://http.developer.nvidia.com/GPUGems3/gpugems3\\_ch39.html](http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html)



# 全面(inclusive)扫描（前缀和）定义

- **定义：**扫描操作接受一个二元关联运算符（binary associative operator） $\oplus$  和一个包含  $n$  个元素的数组

$$[x_0, x_1, \dots, x_{n-1}]$$

并返回数组

$$[x_0, (x_0 \oplus x_1), \dots, (x_0 \oplus x_1 \oplus \dots \oplus x_{n-1})]$$

- **举例：**如果 $\oplus$ 是加号，则在数组

$$[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]$$

上进行扫描操作将返回

$$[3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22 \ 25].$$

# 全面扫描应用举例

- 假设有一个 100 寸的三明治要分给 10 个人
- 已知每人想要的三明治尺寸
  - [3 5 2 7 28 4 3 0 8 1]
- 如何快速切分三明治？
- 最终会剩下多少？
- 法1：按顺序切：首先切出3寸，接着切5寸，然后2寸，如此等等
- 法2：计算前缀和：
  - [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (剩下39寸)



# 扫描的典型应用

- 扫描是一个简单而有用的并行构建块(building block)

- 将循环从顺序执行：

```
for (j=1; j<n; j++)  
    out[j] = out[j-1] + f(j);
```

- 转换为并行执行：

```
forall(j) { temp[j] = f(j) };  
scan(out, temp);
```



# 其他应用

- 分配露营地
- 分配农贸市场空间
- 为并行线程分配内存
- 为通信通道分配内存缓冲区空间
- ...



# 全面顺序加法扫描

- 给定一个序列  $[x_0, x_1, x_2, \dots]$
- 按下面的方法计算输出  $[y_0, y_1, y_2, \dots]$

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

...

- 递归定义

$$y_i = y_{i-1} + x_i$$



# 一个高效的C实现

```
y[0] = x[0];
```

```
for (i = 1; i < Max_i; i++) y[i] = y [i-1] + x[i];
```

计算效率：

N个元素需要N次加法—  $O(N)$  ！

仅比顺序归约(sequential reduction)的开销稍多一点



# 朴素的全面并行扫描

- 为每个y元素的计算分配一个线程
- 让每个线程将 y 元素所需的所有 x 元素相加

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

“只要不在乎性能，并行编程很容易的啦。”



前缀和

低效扫描内核

高效并行扫描内核

并行扫描的延伸讨论



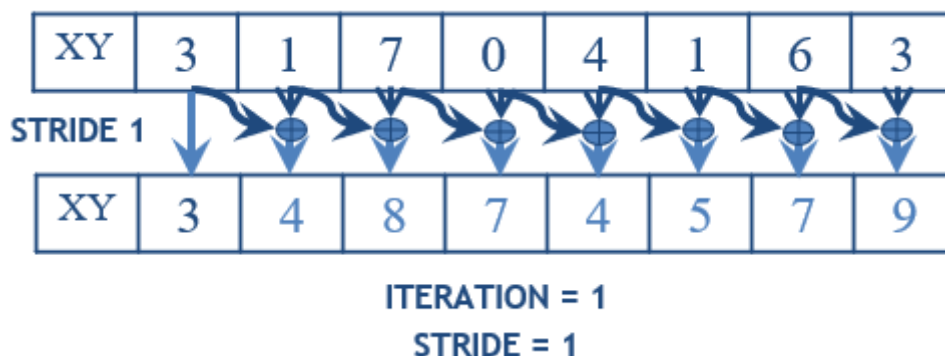
# 目标

- 学习编写和分析高性能扫描内核
  - 交错归约树
  - 线程索引到数据映射
  - 屏障同步(Barrier Synchronization)
  - 工作效率分析



# 更好的并行扫描算法

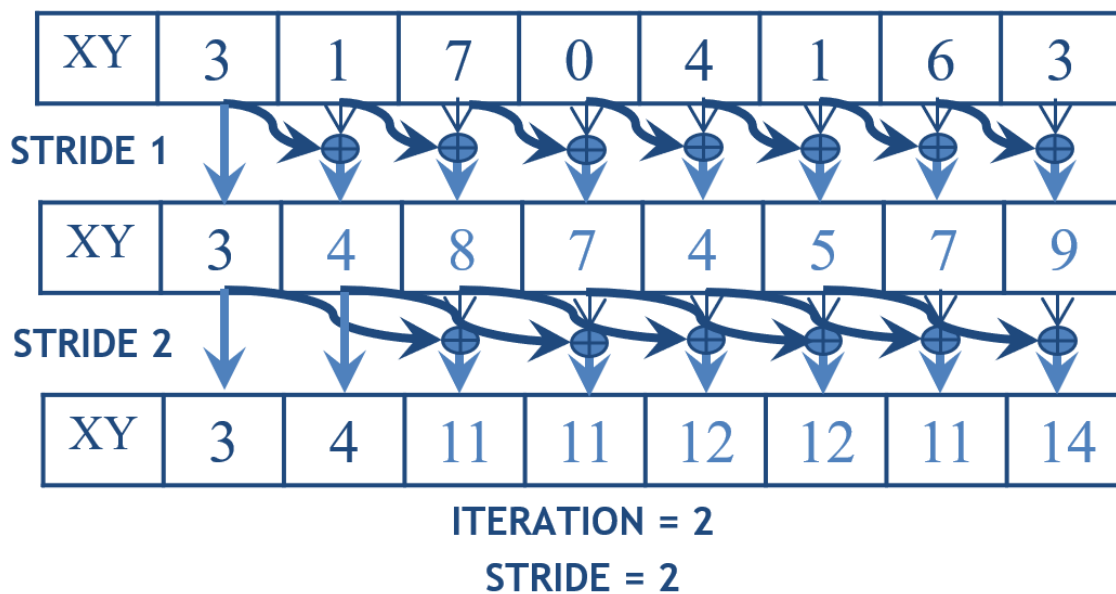
1. 从设备全局内存读取输入到共享内存
2. 迭代 $\log(n)$ 次；步长(stride)从1到 $n-1$ ：每次迭代步长翻倍



- 活动线程步长不超过 $n-1$ （有 $n$ -stride个线程）  
Active threads stride to  $n-1$  ( $n$ -stride threads)
- 线程  $j$  将共享内存中的元素  $j$  和  $j - stride$  相加，并将结果写入共享内存的元素  $j$  中
- 需要屏障同步，读取前一次，写入前一次

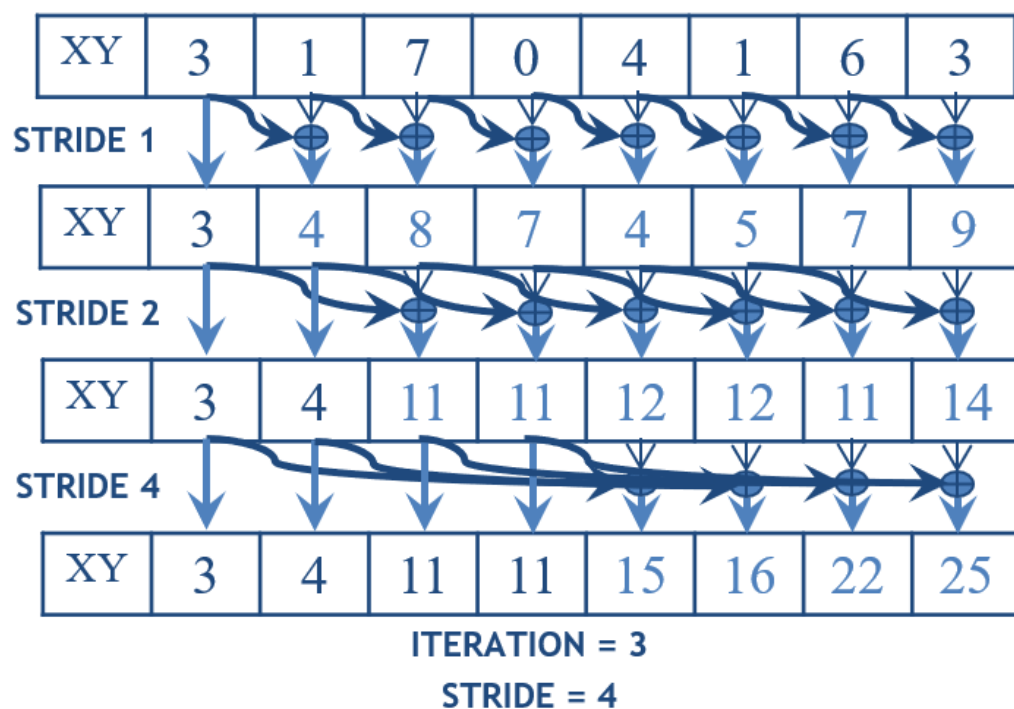
# 更好的并行扫描算法

1. 从设备全局内存读取输入到共享内存
2. 迭代 $\log(n)$ 次；步长(stride)从1到 $n-1$ ：每次迭代步长翻倍



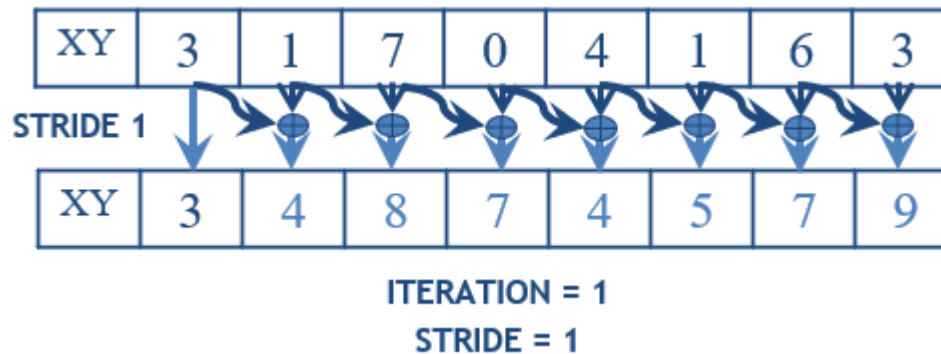
# 更好的并行扫描算法

1. 从设备全局内存读取输入到共享内存
2. 迭代 $\log(n)$ 次；步长(stride)从1到 $n-1$ ：每次迭代步长翻倍
3. 将输出从共享内存写到设备内存



# 处理依赖

- 每次迭代的过程中，每个线程都可能改写另一线程的输入
  - 屏障同步用于保证所有输入都正确生成
  - 所有线程共同保护可能被另一个线程改写的输入操作数
  - 需要屏障同步以确保所有线程都已保护其输入
  - 所有线程都执行加法和写输出



# 一个低效扫描内核

```
__global__ void work_inefficient_scan_kernel(float *X, float *Y, int InputSize) {  
    __shared__ float XY[SECTION_SIZE];  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < InputSize) {XY[threadIdx.x] = X[i];}  
    // the code below performs iterative scan on XY  
    for (unsigned int stride = 1; stride <= threadIdx.x; stride *= 2) {  
        __syncthreads();  
        float in1 = XY[threadIdx.x - stride];  
        __syncthreads();  
        XY[threadIdx.x] += in1;  
    }  
    __syncthreads();  
    If (i < InputSize) {Y[i] = XY[threadIdx.x];}  
}
```



# 工作效率分析

- 此扫描算法执行 $\log(n)$ 次并行迭代
  - 迭代过程分别进行了 $(n-1), (n-2), (n-4), \dots, (n - n/2)$ 次加法
  - 共计加法次数： $n * \log(n) - (n-1) \rightarrow O(n * \log(n))$ 复杂度
- 此扫描算法运行效率不高
  - 顺序扫描算法执行 $n$ 次加法
  - $\log(n)$ 因子的坏处：1024个元素就是10倍的复杂度
- 当执行资源因工作效率低而饱和时，并行算法可能比顺序算法慢



前缀和

低效扫描内核

高效并行扫描内核

并行扫描的延伸讨论



# 目标

- 学习编写高性能扫描内核
  - 两阶段平衡树遍历
  - 积极重用中间结果
  - 通过更复杂的线程索引到数据索引映射来减少控制分支

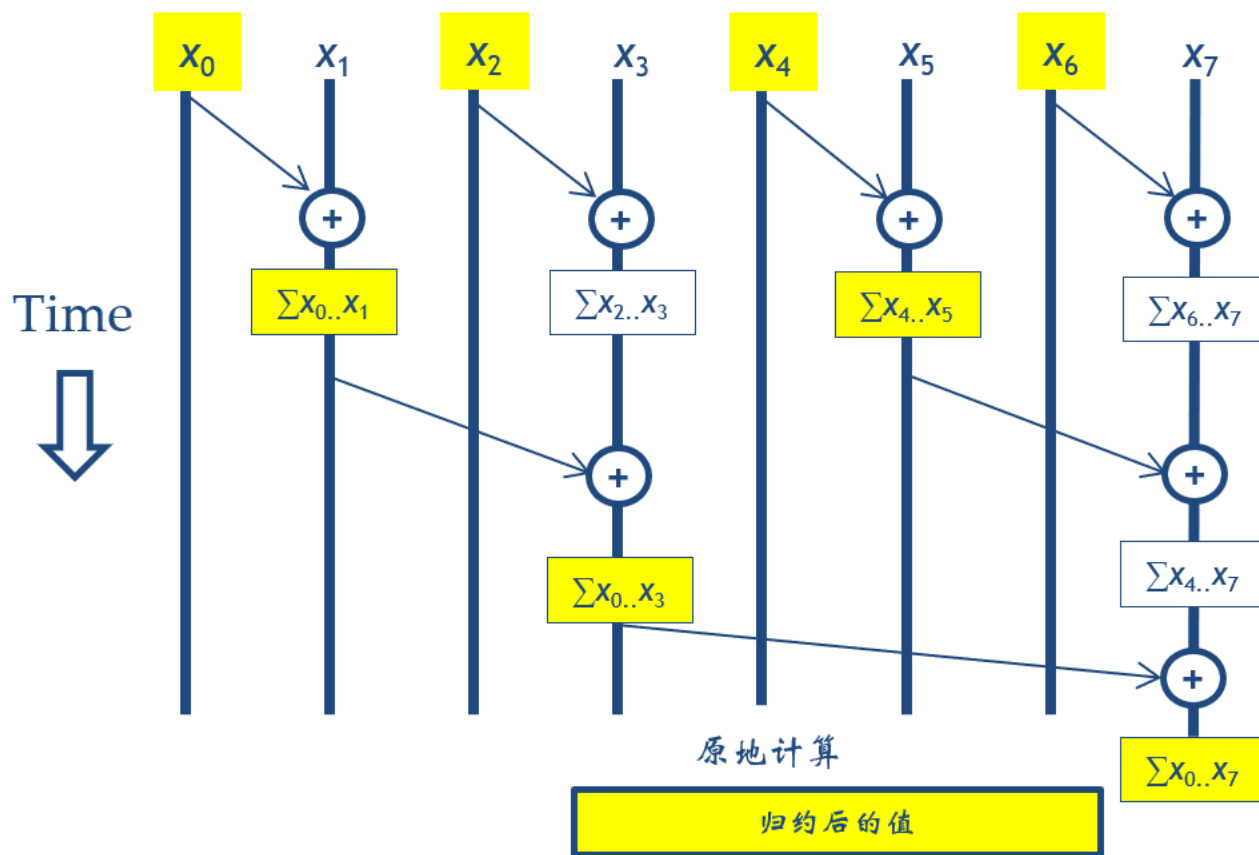


# 提高效率

- 平衡树
  - 在输入数据上形成平衡二叉树并将其扫描到根部和从根部扫描
  - 树不是一个实际的数据结构，而是一个确定每个线程在每个步骤中做什么的概念
- 对于扫描：
  - 从叶子向下遍历到根，在树的内部节点处构建部分和
    - 根包含所有叶子的总和
  - 向上遍历树从部分和构建输出



# 并行扫描 – 归约阶段(Reduction Phase)



# 归约阶段内核代码

```
// XY[2*BLOCK_SIZE] is in shared memory

for (unsigned int stride = 1; stride <= BLOCK_SIZE; stride
    *= 2) {
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE)
        XY[index] += XY[index-stride];
    __syncthreads();
}
```

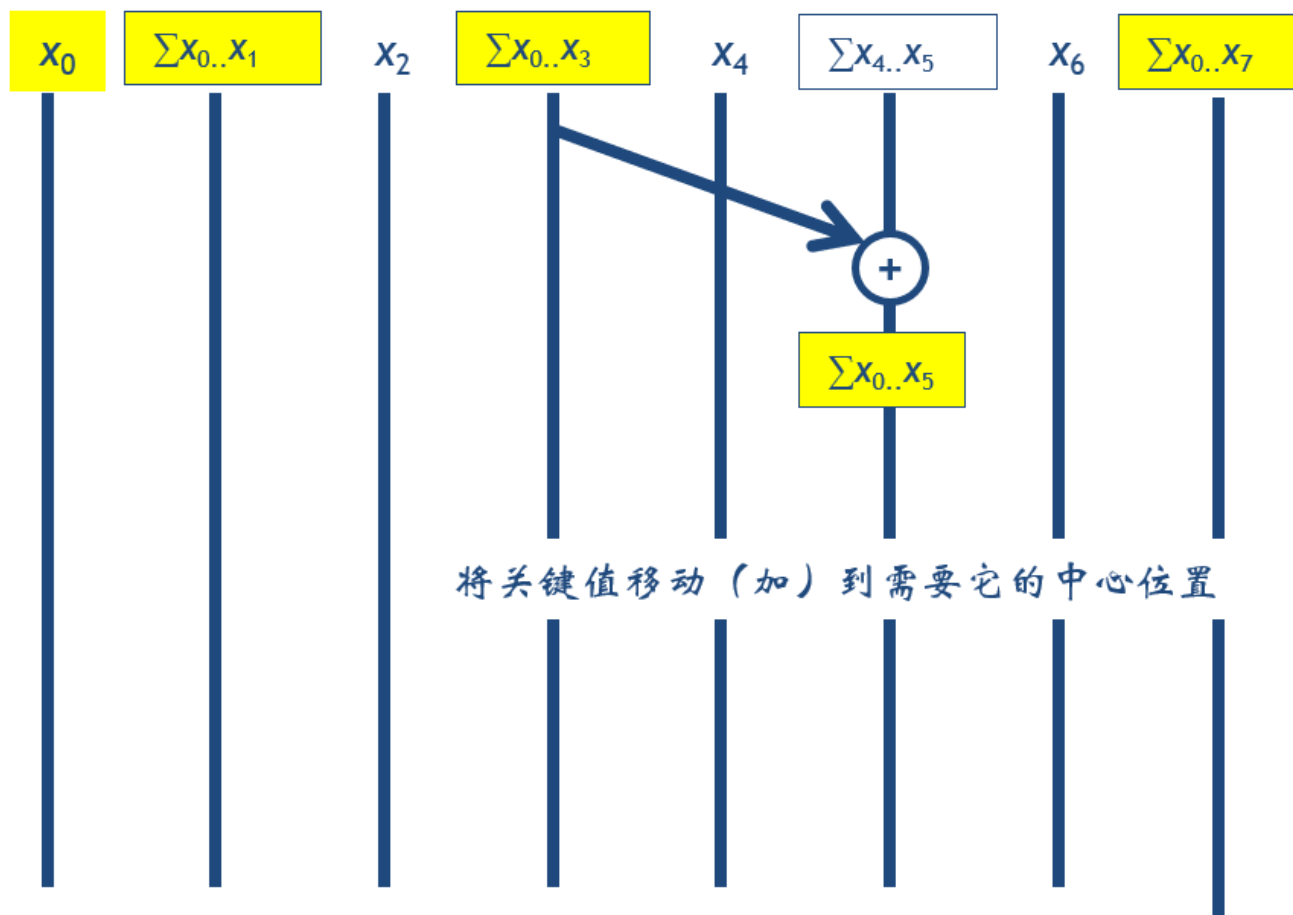
threadIdx.x+1 = 1, 2, 3, 4....

stride = 1,

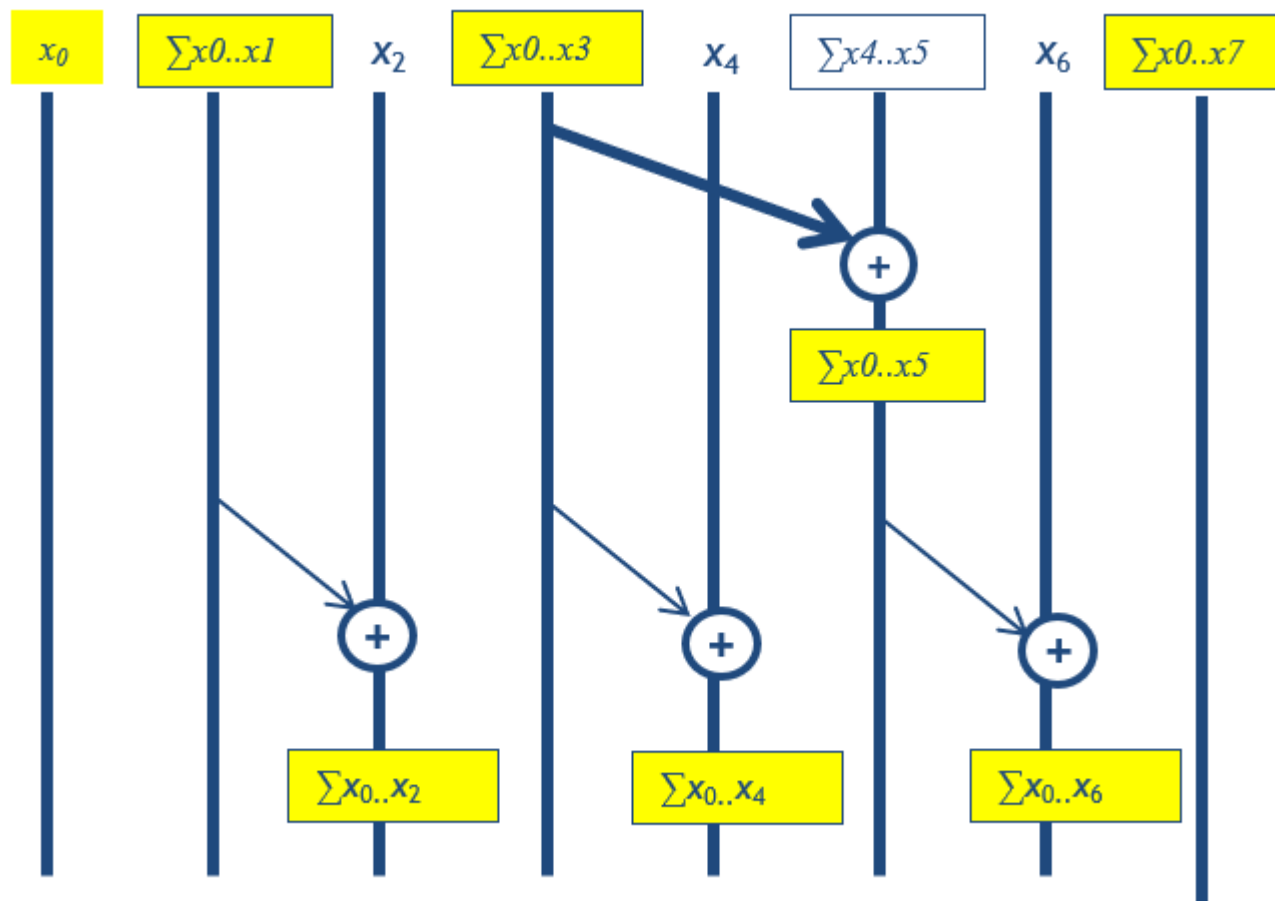
index = 1, 3, 5, 7, ...



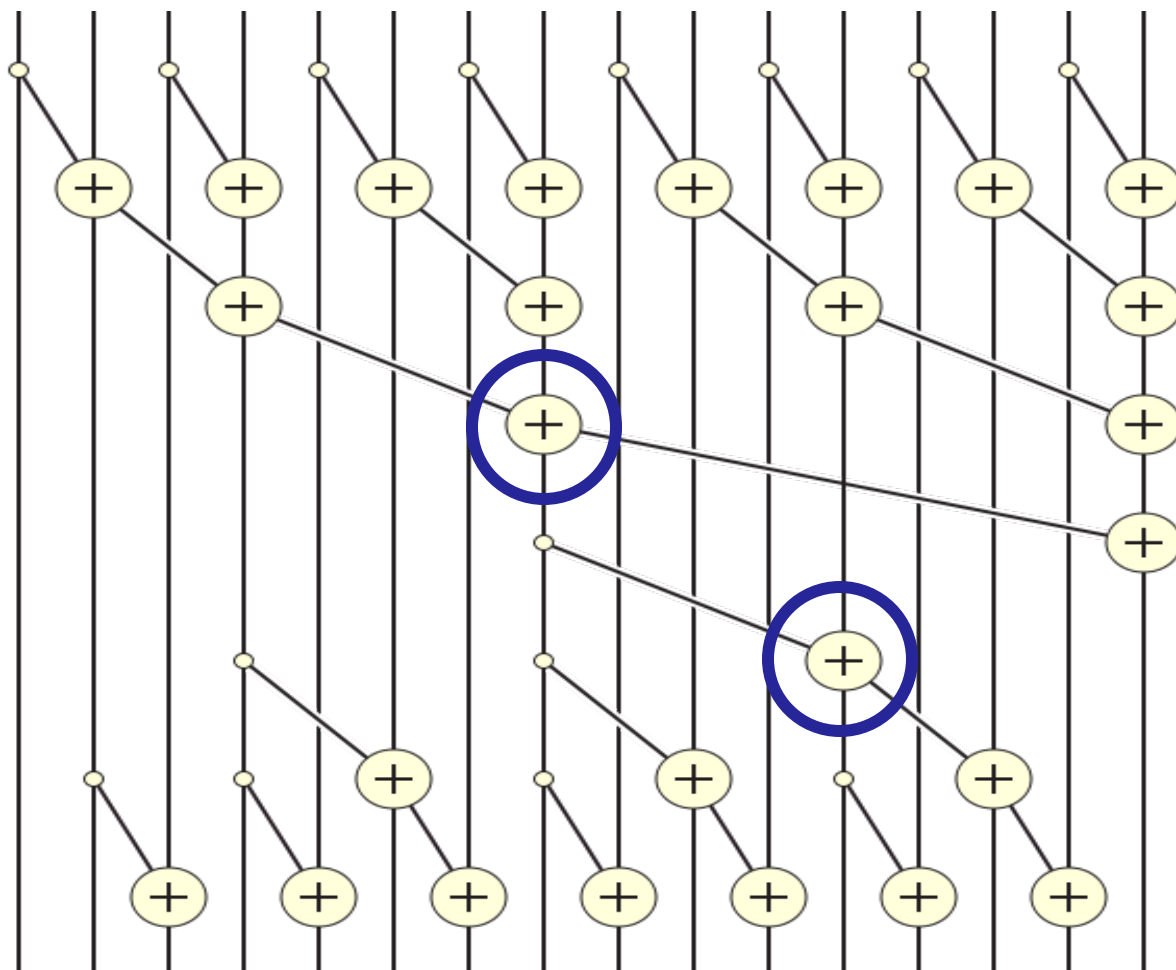
# 并行扫描 – 后归约反向阶段(Post Reduction Reverse Phase)



# 并行扫描 – 后归约反向阶段



# 汇总



# 归约后逆向阶段内核代码

```
for (unsigned int stride = BLOCK_SIZE/2; stride > 0; stride /= 2)
{
    __syncthreads();
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index+stride < 2*BLOCK_SIZE) {
        XY[index + stride] += XY[index];
    }
}

__syncthreads();
if (i < InputSize) Y[i] = XY[threadIdx.x];

//16个元素，第一次
threadIdx.x = 0
stride = BLOCK_SIZE/2 = 8/2 = 4
index = 8-1 = 7
```



前缀和

低效扫描内核

高效扫描内核

并行扫描的延伸讨论



# 目标

- 了解有关并行扫描的更多内容
  - 高效内核的分析
  - 独占扫描(Exclusive scan)
  - 处理超大输入向量



# 高效内核的工作分析

- 高效内核在归约步骤中执行了 $\log(n)$ 次并行迭代
  - 每次迭代分别做了  $n/2, n/4, \dots, 1$  次加法
  - 总共的加法次数： $(n-1) \rightarrow O(n)$  复杂度
- 在归约后反向步骤中执行了 $\log(n)-1$ 次并行迭代
  - 每次迭代分别做了  $2-1, 4-1, \dots, n/2-1$  次加法
  - 总共的加法次数： $(n-2)-(\log(n)-1) \rightarrow O(n)$  复杂度
- 各阶段均执行不超过 $2 \times (n-1)$ 次加法
- 加法的总次数不超过高效顺序算法的两倍
  - 当有足够的硬件时，并行化的收益可以轻松抵消其带来的2倍计算量的开销

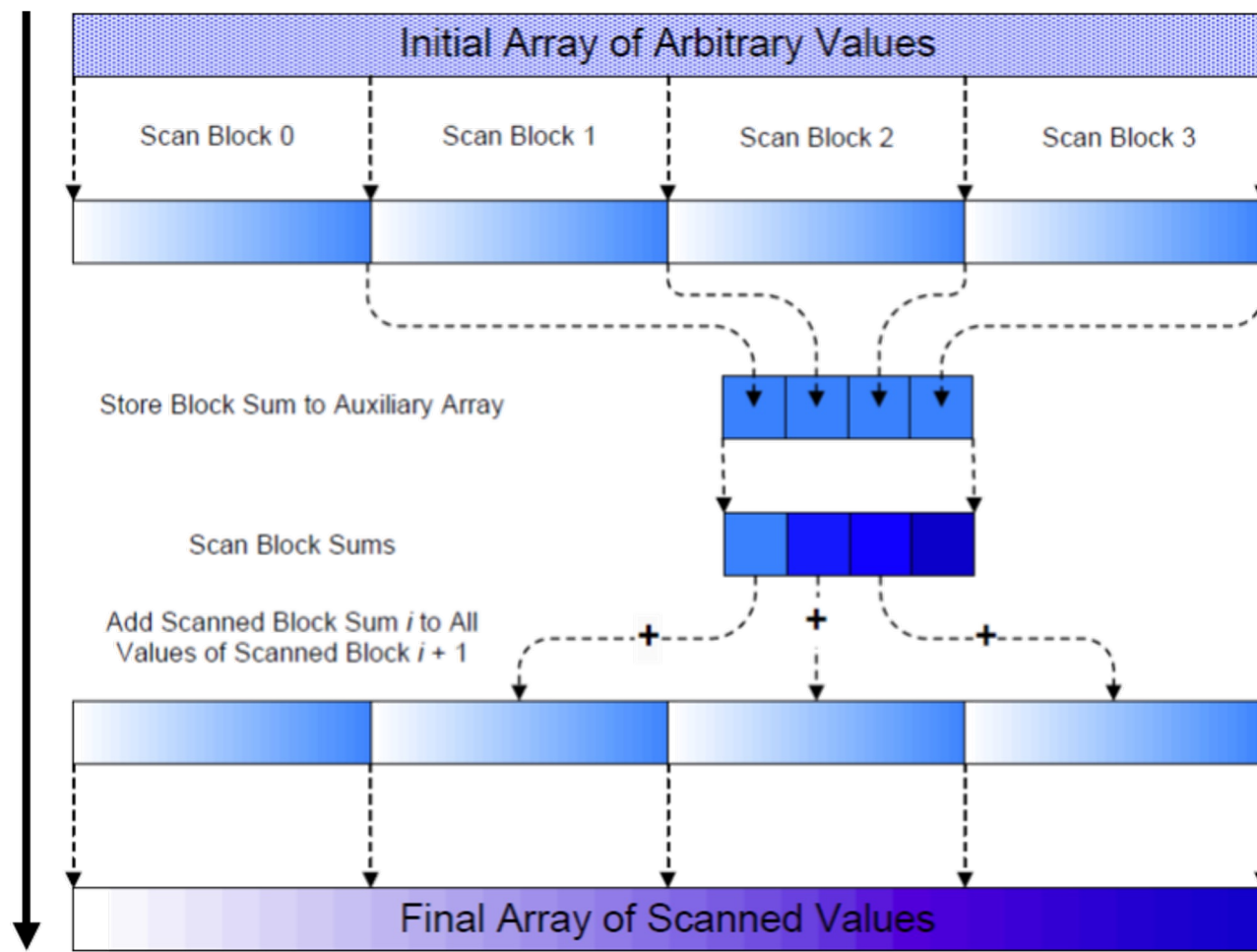
# 一些权衡

- 高效扫描内核通常更可取
  - 更好的能耗比
  - 更少的执行资源需求
- 然而，低效内核由于其单阶段的性质（仅有正向阶段）可能会获得更好的绝对性能
  - 有足够的执行资源时

# 处理超大输入向量

- 基于高效扫描内核
- 将  $2 \times \text{blockDim.x}$  元素的每个部分(section)分配给一个块
  - 在各部分执行并行扫描
- 让每个块将其部分的总和写入由  $\text{blockIdx.x}$  索引的  $\text{Sum}[]$  数组
- 在  $\text{Sum}[]$  数组上运行扫描内核
- 将扫描的  $\text{Sum}[]$  数组值加到相应部分的所有元素上
- 类似地，也可采用低效内核

# 完整扫描流程总览



# 独占扫描(Exclusive Scan)定义

- 定义：**独占扫描操作接受一个二元关联运算符  $\oplus$  和一个包含  $n$  个元素的数组

$$[x_0, x_1, \dots, x_{n-1}]$$

并返回数组

$$[0, x_0, (x_0 \oplus x_1), \dots, (x_0 \oplus x_1 \oplus \dots \oplus x_{n-2})]$$

- 举例：**如果 $\oplus$ 是加号，则在数组

$$[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]$$

上进行独占扫描操作将返回

$$[0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22].$$



# 为何使用独占扫描？

- 查找分配的缓冲区的起始地址
- 全面(inclusive)和独占(exclusive)扫描很容易相互推导；如何选择取决于哪个更便于使用

[3 1 7 0 4 1 6 3]

独占 [0 3 4 11 11 15 16 22]

全面 [3 4 11 11 15 16 22 25]



# 一个简单的独占扫描内核

- 采用一个全面的、低效的扫描内核
- 0号线程块(Block 0):
  - 0号线程将0载入XY[0]
  - 其他线程将 $X[\text{threadIdx.x}-1]$ 载入 $XY[\text{threadIdx.x}]$
- 所有其他线程块：
  - 所有线程将 $X[\text{blockIdx.x}*\text{blockDim.x}+\text{threadIdx.x}-1]$ 载入 $XY[\text{threadIdx.x}]$
- 类似地可采用高效扫描内核，但需要确保每个线程载入两个元素
  - 只有一个0应该被载入
  - 所有元素应仅向右移动一个位置

