

# 第五讲 – 并行计算算法 Histogram



# 目录

- 并行直方图 (Parallel Histogram)
- 数据竞态条件 (Data Racing Condition)
- 私有化直方图内核 (Privatized Histogram Kernel)



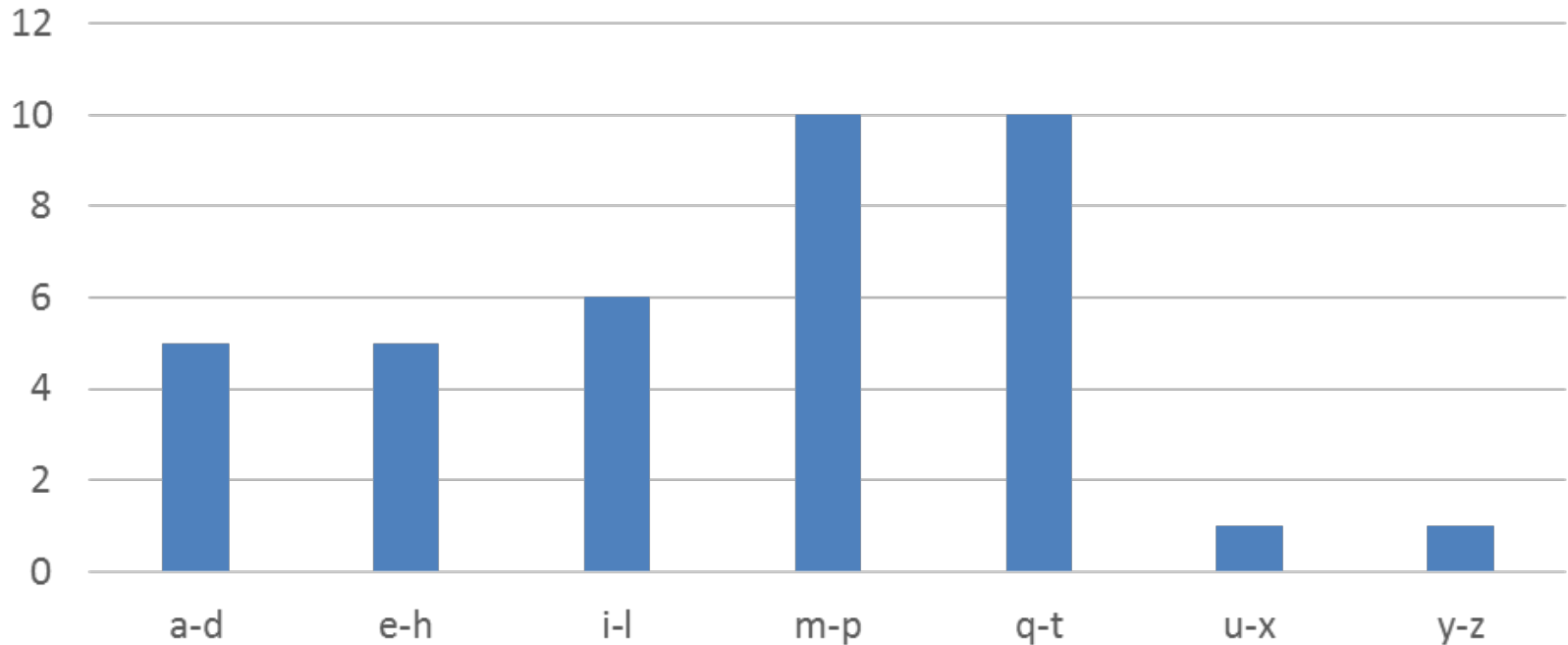
## 小节目标

- 学习并行直方图计算模式
  - 一种重要且有效的计算方法
  - 就每个线程的输出行为而言，与我们目前介绍的所有模式都有很大不同：输出可以被所有参与的线程修改。



## 示例：文本直方图

- 定义 bins 为划分字母表的若干个部分，每个部分包含四个字母：a-d、e-h、i-l、m-p、...
- 对于输入字符串中的每个字符，相应的 bin 计数增加。
- 在短语 “Programming Massively Parallel Processors” 中，输出直方图如下所示：

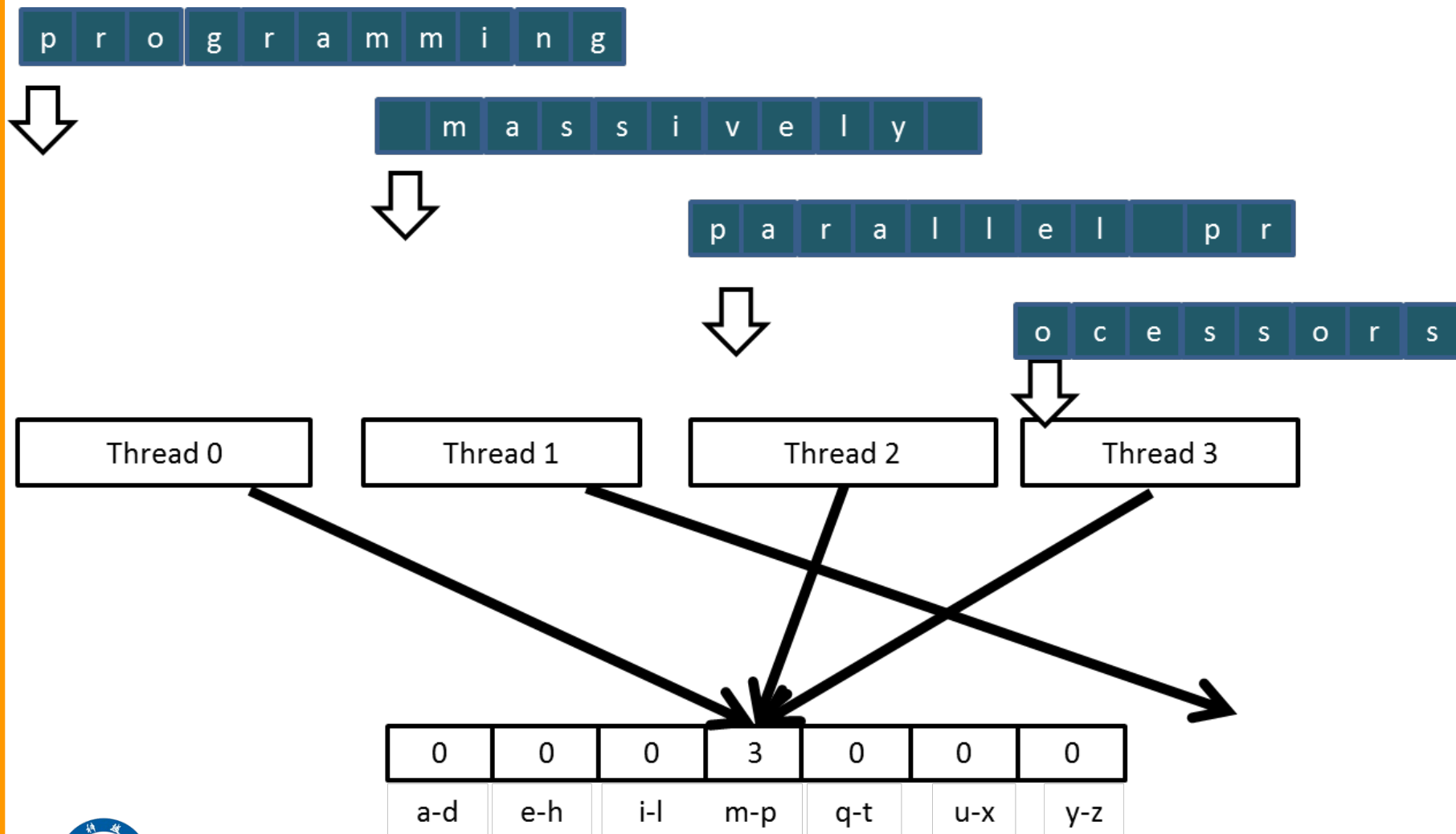


# 一个简单的并行直方图算法

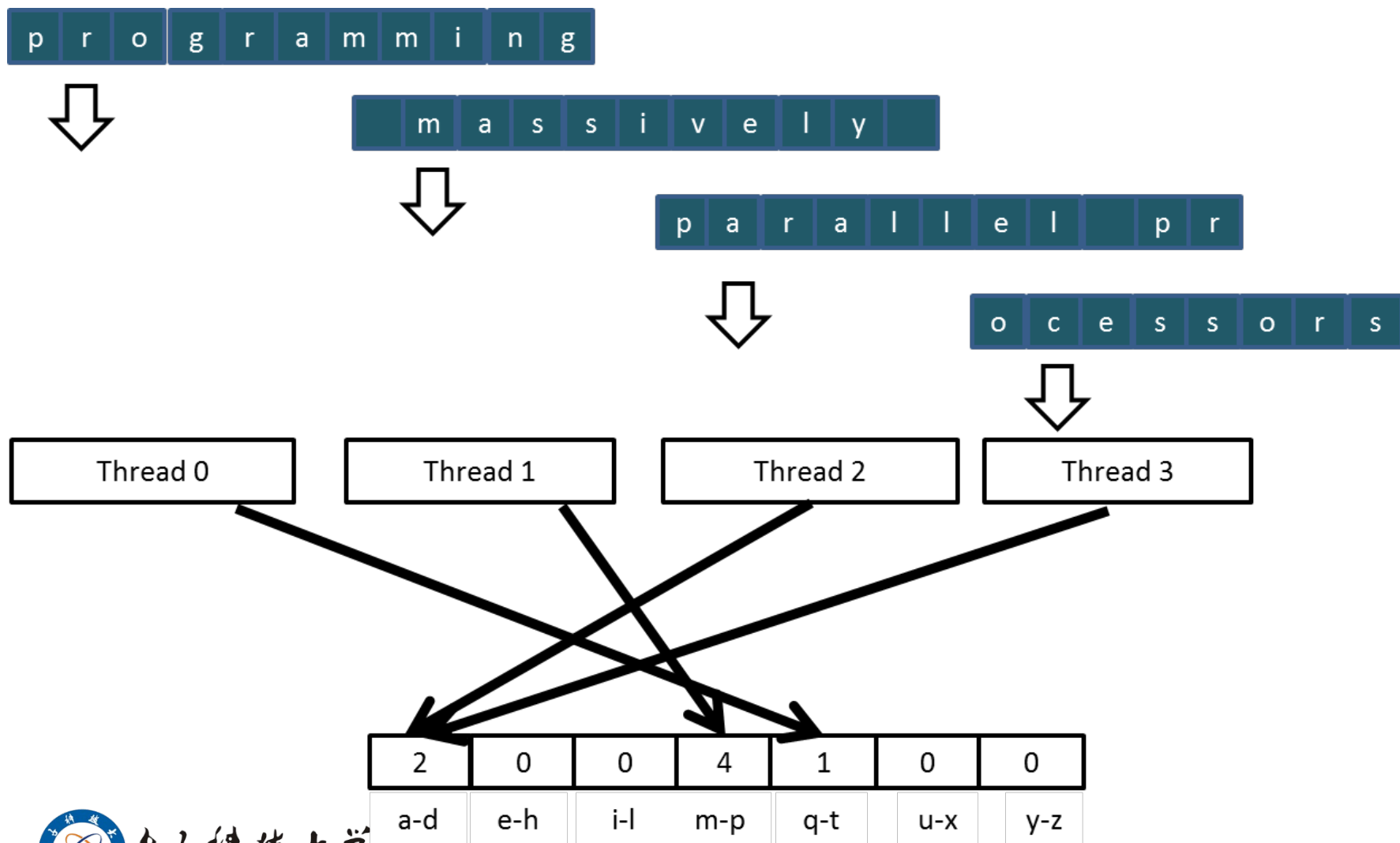
- 将输入划分为若干个部分
- 让每个线程处理一个部分的输入
- 每个线程遍历对应的部分
- 对于每个字母，增加相应的 bin 计数器



# 分段后的分区 (第 1 步迭代)

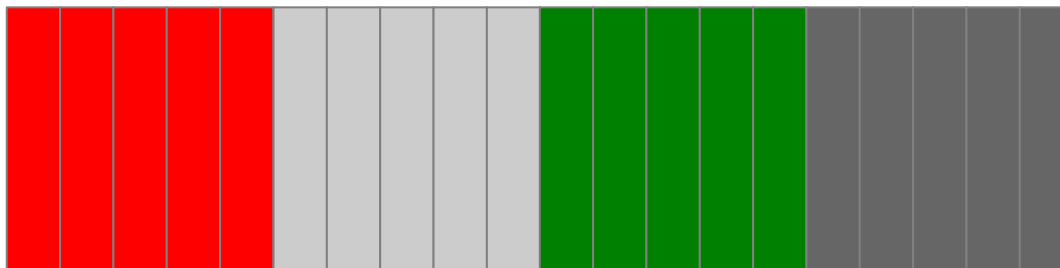


## 分段后的分区 (第 2 步迭代)



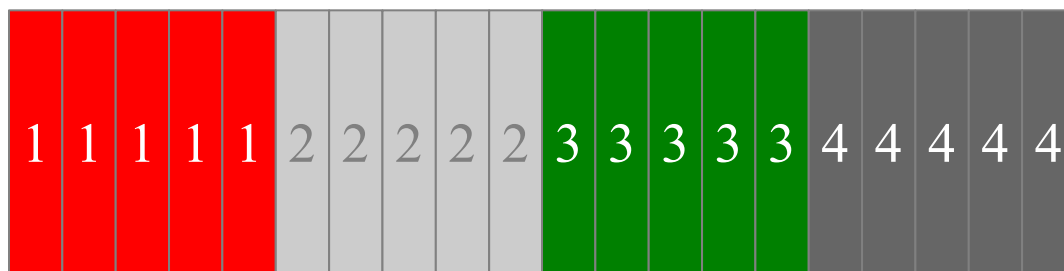
# 输入分区影响内存访问效率

- 分段分区（Sectioned Partitioning）导致内存访问效率低下
  - 相邻线程不访问相邻内存位置
  - 访问未被合并
  - DRAM 带宽利用率低

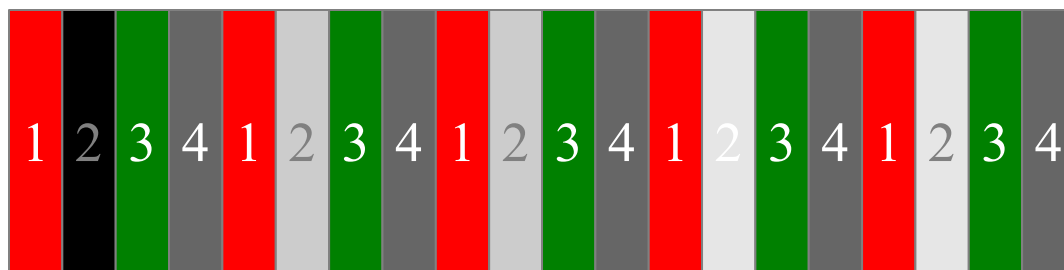


# 输入分区影响内存访问效率

- 分段分区 ( Sectioned Partitioning ) 导致内存访问效率低下
  - 相邻线程不访问相邻内存位置
  - 访问未被合并
  - DRAM 带宽利用率低

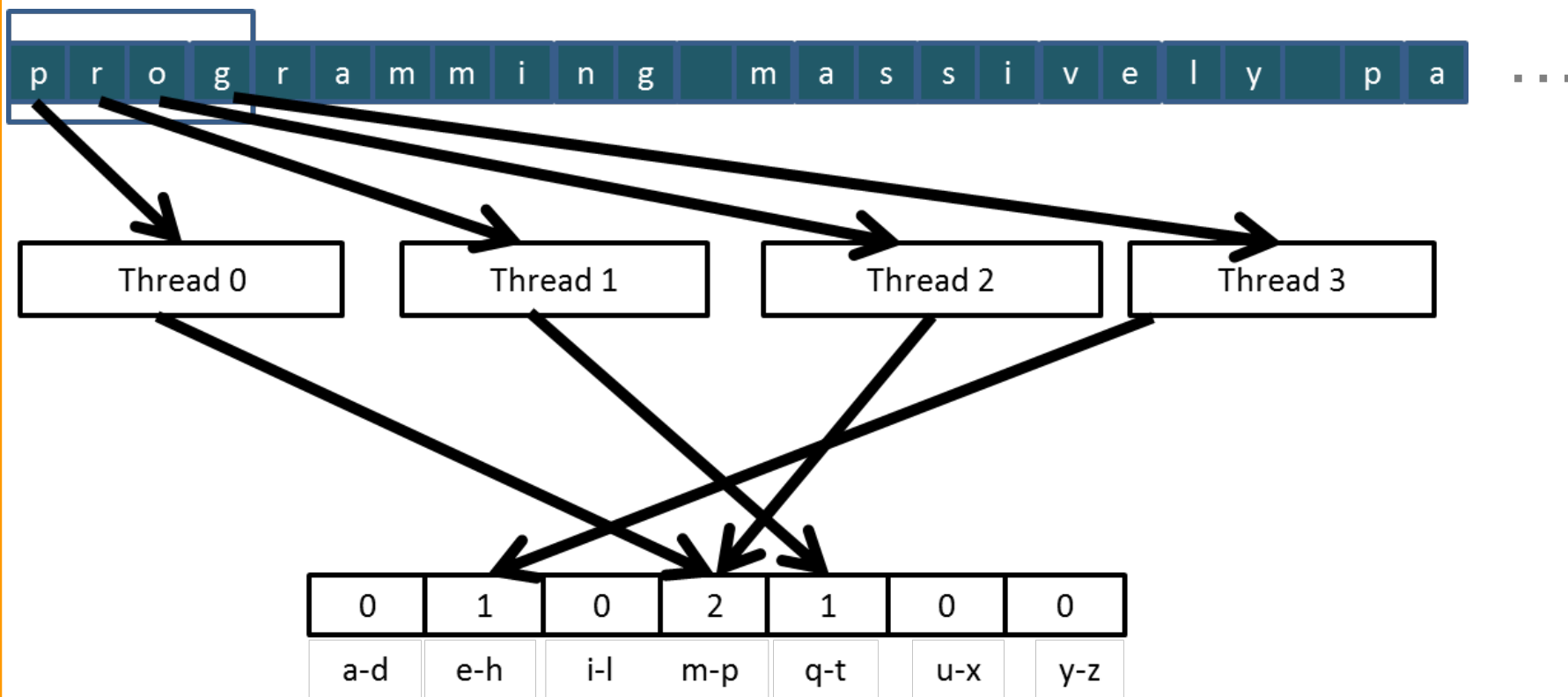


- 更改为交错分区
  - 所有线程处理元素的连续部分
  - 他们都移动到下一部分并重复
  - 内存访问被合并

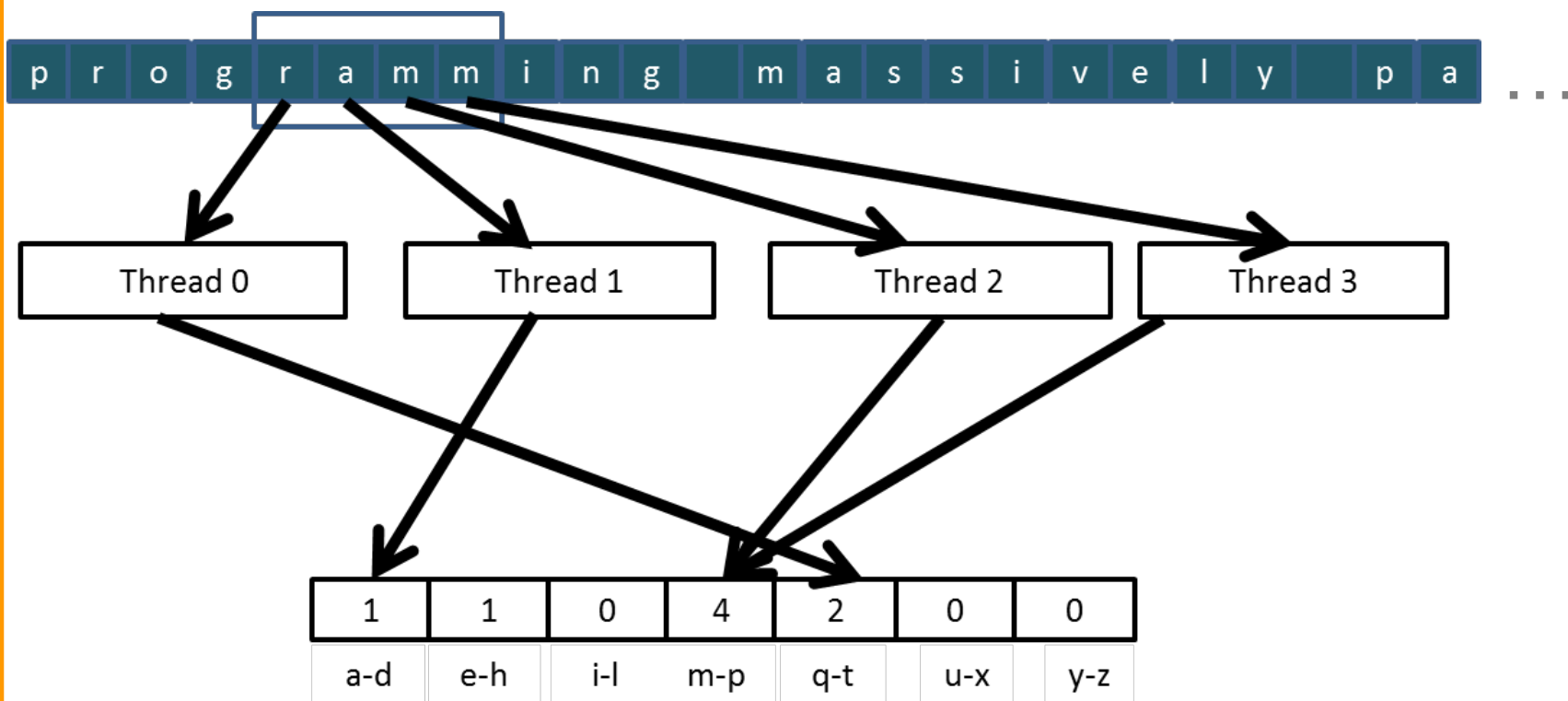


# 输入的交错分区

— 用于合并和更好的内存访问性能



## 交错分区 (第 2 步迭代)



# 目录

- 并行直方图 (Parallel Histogram)
- **数据竞态条件 (Data Racing Condition)**
- 私有化直方图内核 (Privatized Histogram Kernel)



## 小节目标

- 了解并行计算中的数据争用 (Data Race)
  - 执行**读取-修改-写入**操作时可能发生数据争用
  - 数据争用可能导致难以重现的错误
  - **原子操作**旨在消除此类数据争用
    - 原子操作概念
    - CUDA 中的原子操作类型
    - 内在函数
- 一个基本的直方图内核



# 并行线程执行中的数据争用

thread1:  $\text{Old} \leftarrow \text{Mem}[x]$   
           $\text{New} \leftarrow \text{Old} + 1$   
           $\text{Mem}[x] \leftarrow \text{New}$

thread2:  $\text{Old} \leftarrow \text{Mem}[x]$   
           $\text{New} \leftarrow \text{Old} + 1$   
           $\text{Mem}[x] \leftarrow \text{New}$

Old 和 New 是每个线程的寄存器变量。

问题 1：如果  $\text{Mem}[x]$  最初为 0，那么线程 1 和 2 执行完毕后  $\text{Mem}[x]$  的值是多少？

问题 2：每个线程在它们的 Old 变量中得到什么？

不幸的是，答案可能会根据两个线程之间的相对执行时序而有所不同，这被称为数据争用。

竞态条件 (*race condition*)，其中两个或多个同时更新操作的结果根据所涉及操作的相对时序而变化。



# 时序场景 #1

时间	线程 1	线程 2
1	(0) $\text{Old} \leftarrow \text{Mem}[x]$	
2	(1) $\text{New} \leftarrow \text{Old} + 1$	
3	(1) $\text{Mem}[x] \leftarrow \text{New}$	
4		(1) $\text{Old} \leftarrow \text{Mem}[x]$
5		(2) $\text{New} \leftarrow \text{Old} + 1$
6		(2) $\text{Mem}[x] \leftarrow \text{New}$

- Thread 1 Old = 0
- Thread 2 Old = 1
- $\text{Mem}[x] = 2$



## 时序场景 #2

时间	线程 1	线程 2
1		(0) $\text{Old} \leftarrow \text{Mem}[x]$
2		(1) $\text{New} \leftarrow \text{Old} + 1$
3		(1) $\text{Mem}[x] \leftarrow \text{New}$
4	(1) $\text{Old} \leftarrow \text{Mem}[x]$	
5	(2) $\text{New} \leftarrow \text{Old} + 1$	
6	(2) $\text{Mem}[x] \leftarrow \text{New}$	

- Thread 1 Old = 1
- Thread 2 Old = 0
- $\text{Mem}[x] = 2$



## 时序场景 #3

时间	线程 1	线程 2
1	(0) $\text{Old} \leftarrow \text{Mem}[x]$	
2	(1) $\text{New} \leftarrow \text{Old} + 1$	
3		(0) $\text{Old} \leftarrow \text{Mem}[x]$
4	(1) $\text{Mem}[x] \leftarrow \text{New}$	
5		(1) $\text{New} \leftarrow \text{Old} + 1$
6		(1) $\text{Mem}[x] \leftarrow \text{New}$

- Thread 1 Old = 0
- Thread 2 Old = 0
- $\text{Mem}[x] = 1$



## 时序场景 #4

时间	线程 1	线程 2
1		(0) $\text{Old} \leftarrow \text{Mem}[x]$
2		(1) $\text{New} \leftarrow \text{Old} + 1$
3	(0) $\text{Old} \leftarrow \text{Mem}[x]$	
4		(1) $\text{Mem}[x] \leftarrow \text{New}$
5	(1) $\text{New} \leftarrow \text{Old} + 1$	
6	(1) $\text{Mem}[x] \leftarrow \text{New}$	

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1



# 原子操作的目的——确保好的结果

thread1:  $\text{Old} \leftarrow \text{Mem}[x]$   
 $\text{New} \leftarrow \text{Old} + 1$   
 $\text{Mem}[x] \leftarrow \text{New}$

thread2:  $\text{Old} \leftarrow \text{Mem}[x]$   
 $\text{New} \leftarrow \text{Old} + 1$   
 $\text{Mem}[x] \leftarrow \text{New}$

或

thread1:  $\text{Old} \leftarrow \text{Mem}[x]$   
 $\text{New} \leftarrow \text{Old} + 1$   
 $\text{Mem}[x] \leftarrow \text{New}$

thread2:  $\text{Old} \leftarrow \text{Mem}[x]$   
 $\text{New} \leftarrow \text{Old} + 1$   
 $\text{Mem}[x] \leftarrow \text{New}$



# 原子操作中的关键概念

- 由单个硬件指令对内存位置地址执行的读取-修改-写入操作
  - 读取旧值，计算新值，并将新值写入位置
- 硬件确保在当前原子操作完成之前，没有其他线程可以在同一位置执行另一个读取-修改-写入操作
  - Any other threads that attempt to perform an atomic operation on the same location will typically be held in a queue尝试在同一位置执行原子操作的任何其他线程通常都将保留在**队列**中
  - 所有线程**串行**执行针对同一位置的原子操作



# CUDA 中的原子操作

- 通过调用转换为单个指令的函数（也称为内在函数）来执行
  - 原子级的 add, sub, inc, dec, min, max, exch (交换), CAS (比较和交换) 操作
  - 阅读 CUDA C 编程指南 4.0 或更高版本了解详细信息

- 原子级的 Add 操作

*int atomicAdd(int\* address, int val);*

- 从全局或共享内存中地址指向的位置读取 32 位变量 old，计算 (old + val)，并将结果存储回同一地址的内存中。该函数返回变量 old。



# CUDA 中更多的原子级 Add 操作

## – Unsigned 32-bit integer atomic add

```
unsigned int atomicAdd(unsigned int* address,  
    unsigned int val);
```

## – Unsigned 64-bit integer atomic add

```
unsigned long long int atomicAdd(unsigned long long  
    int* address, unsigned long long int val);
```

## – Single-precision floating-point atomic add (capability > 2.0)

```
float atomicAdd(float* address, float val);
```



## 一个基本的直方图内核（续）

- 内核接收一个指向字节值输入缓冲区 (the input buffer of byte values) 的指针
- 每个线程以 strided 模式处理输入

```
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;

    // All threads handle blockDim.x * gridDim.x
    // consecutive elements
    while (i < size) {
        int alphabet_position = buffer[i] - "a";
        if (alphabet_position >= 0 && alphabet_position < 26)
            atomicAdd(&(histo[alphabet_position/4]), 1);
        i += stride;
    }
}
```



# 目录

- 并行直方图 ( Parallel Histogram )
- 数据竞态条件 ( Data Racing Condition )
- 私有化直方图内核  
( Privatized Histogram Kernel )



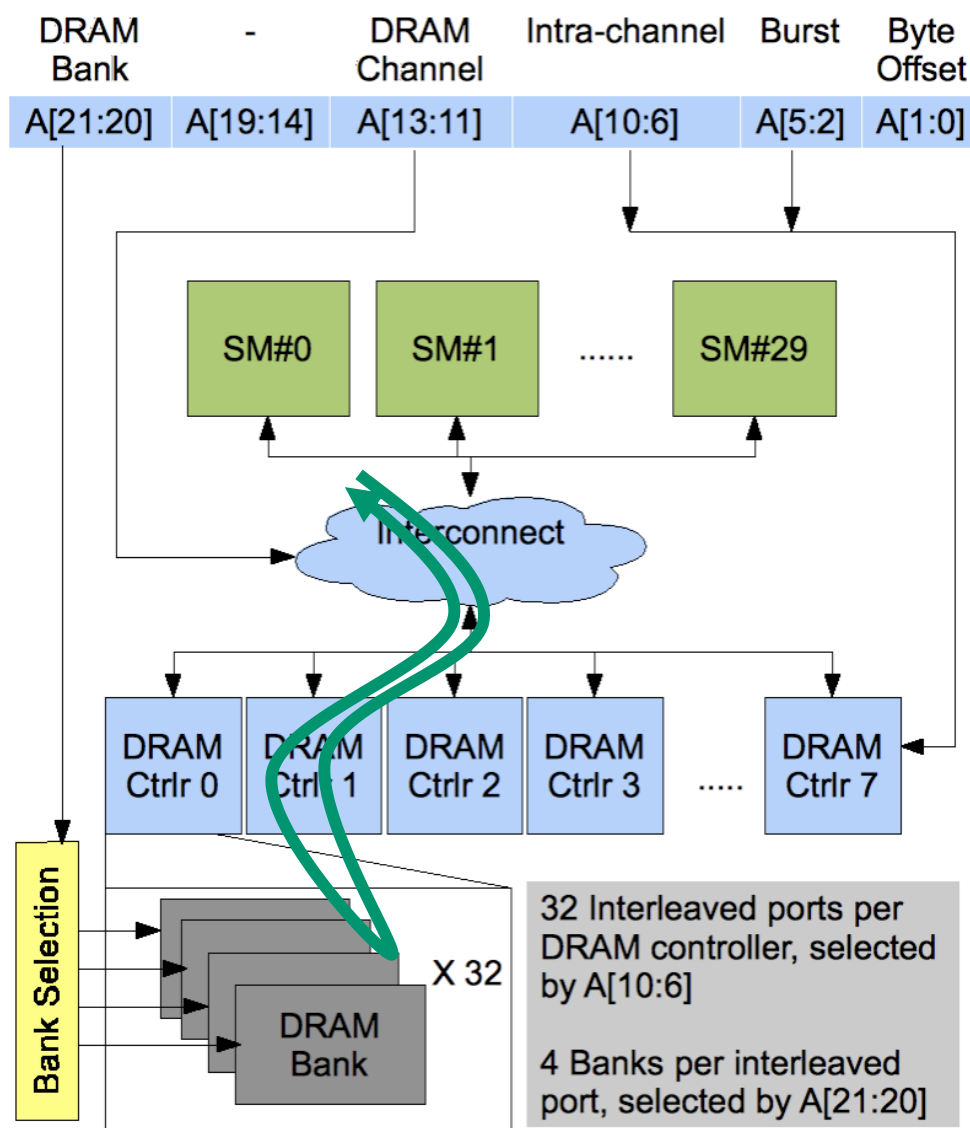
## 小节目标

- 了解原子操作的主要性能注意事项
  - 原子操作的延迟和吞吐量
  - 全局内存上的原子操作
  - 共享 L2 级缓存上的原子操作
  - 共享内存上的原子操作



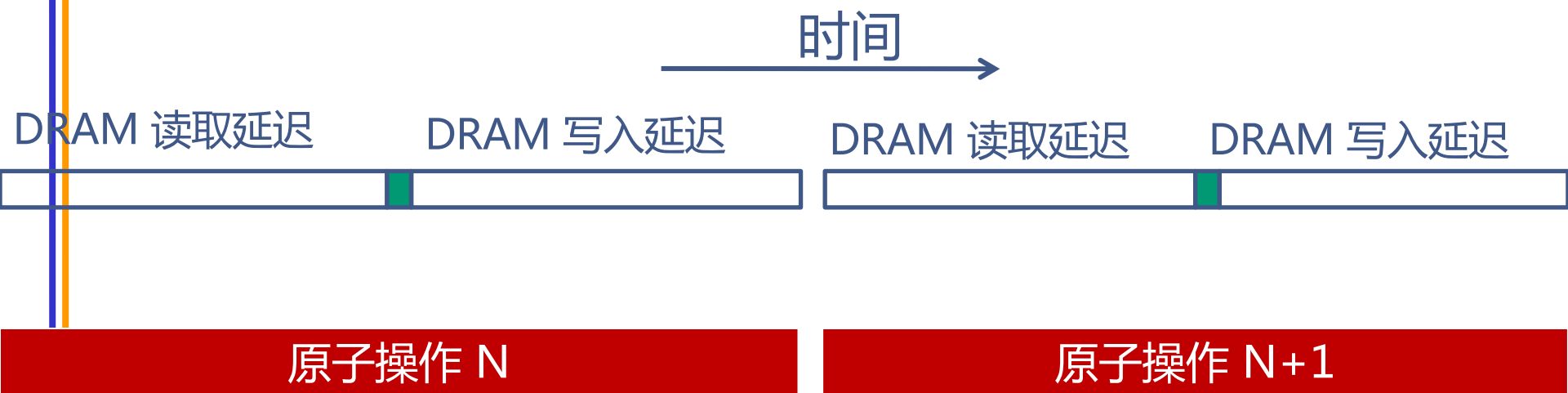
# 全局内存 (DRAM) 上的原子操作

- DRAM 位置上的原子操作从读取开始，其延迟为几百个周期
- 原子操作以在同一位置进行写入操作为结束，延迟为几百个周期
- 在此期间，没有其他线程可以访问该位置



# DRAM 上的原子操作

- 每个读取-修改-写入操作都有两个完整的内存访问延迟
  - 对同一变量（DRAM 位置）的所有原子操作都被串行化



# 延迟决定吞吐量

- 同一 DRAM 位置上的原子操作的吞吐量是应用程序可以执行原子操作的速率。
- 特定位置的原子操作速率受读取-修改-写入序列的总延迟限制，对于全局内存 (DRAM) 位置，通常超过 1000 个周期。
- 这意味着，如果许多线程尝试在同一位置进行原子操作（争用），则内存吞吐量将降低到小于一个内存通道峰值带宽的 1/1000！



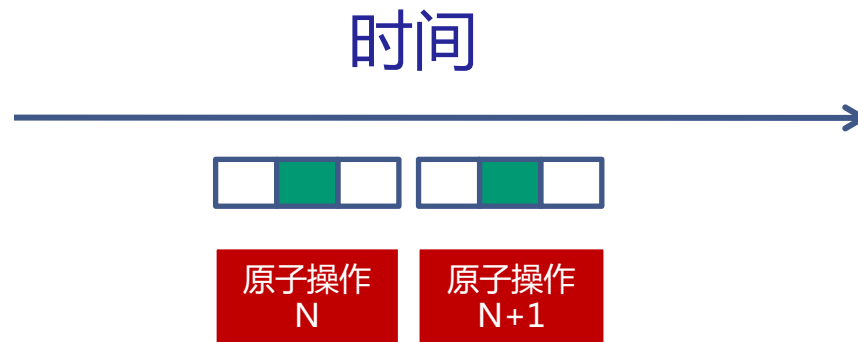
# 硬件改进

- Fermi L2 缓存上的原子操作
  - 中等延迟，大约是 DRAM 延迟的 1/10
  - 在所有线程块之间共享
  - 全局内存原子的“免费改进” (Free Improvement)



# 硬件改进

- 共享内存上的原子操作
  - 非常短的延迟
  - 每个线程块私有
  - 需要程序员的算法工作（稍后介绍更多）



# 目录

- 并行直方图 ( Parallel Histogram )
- 数据竞态条件 ( Data Racing Condition )
- 私有化直方图内核  
( Privatized Histogram Kernel )



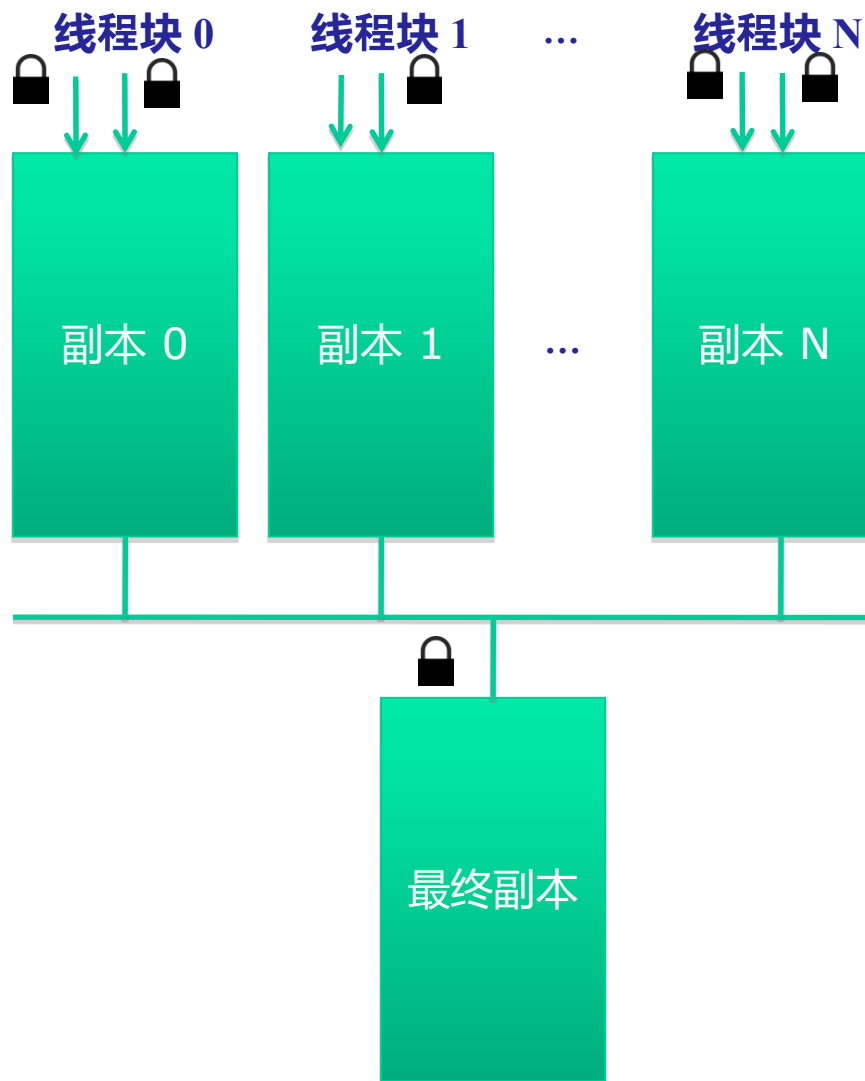
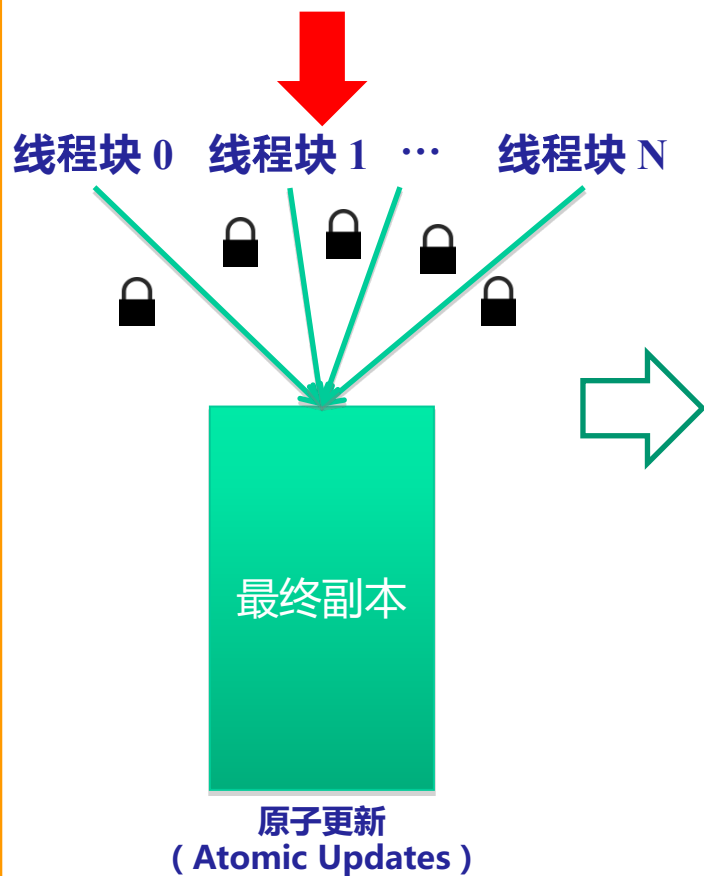
## 小节目标

- 通过私有化输出来学习编写高性能内核
  - 私有化作为一种减少延迟、增加吞吐量和减少串行化的技术
  - 一个高性能的私有直方图内核
  - 使用共享内存和 L2 缓存原子操作的实际示例



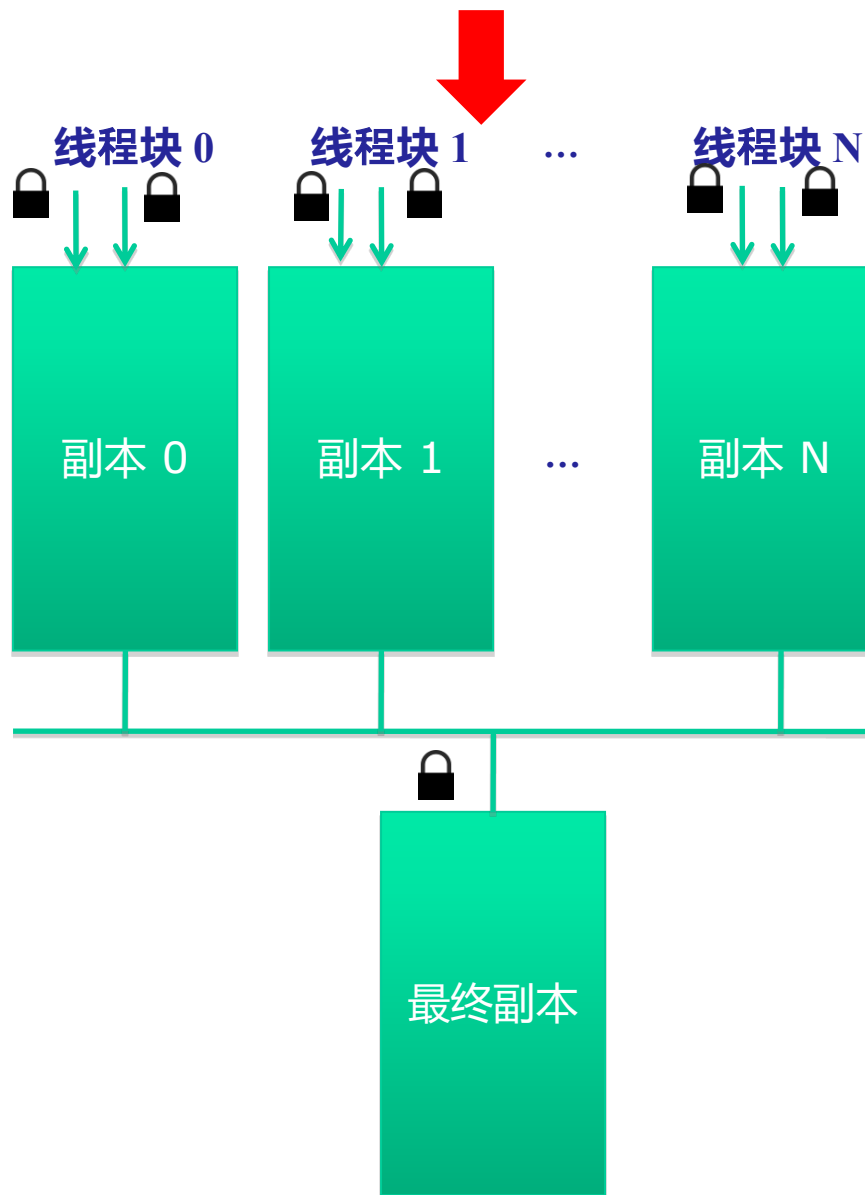
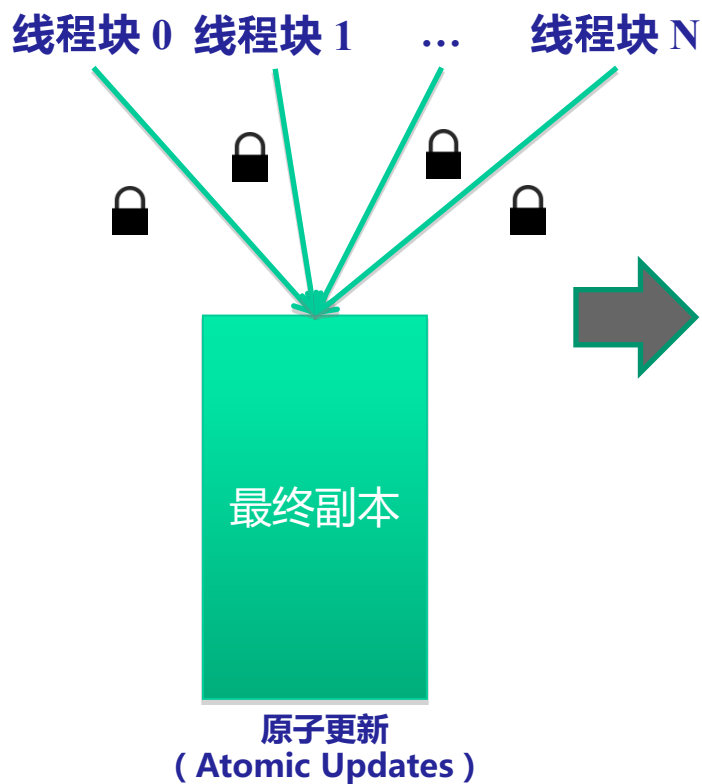
# 私有化

## 重度争用和串行化

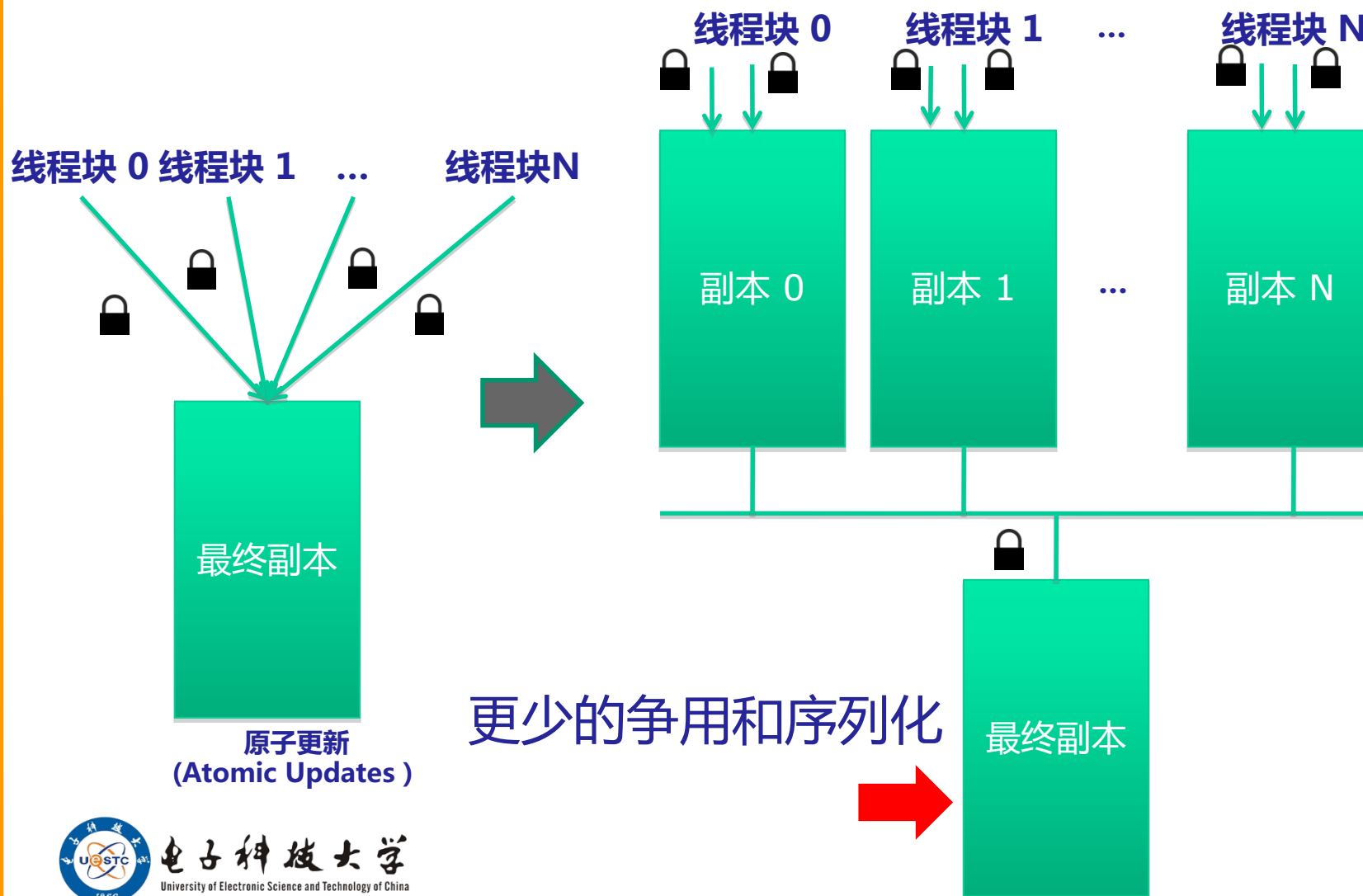


# 私有化 ( 续 )

更少的争用和序列化



# 私有化 ( 续 )



# 私有化的成本和收益

## — 成本

- 创建和初始化私有副本的开销
- 将私有副本的内容累积到最终副本的开销

## — 收益

- 访问私有副本和最终副本时的数据争用和串行化要少得多
- 整体性能往往可以提升10倍以上



# 直方图的共享内存原子

- 每个线程子集都在同一个线程块中
- 吞吐量远高于 DRAM (100x) 或 L2 (10x) 原子
- 更少的数据争用——只有同一线程块中的线程可以访问共享内存变量
- 这是共享内存的一个非常重要的用例！



# 共享内存原子需要私有化

- 为每个线程块创建 histo[] 数组的私有副本

```
__global__ void histo_kernel(unsigned char *buffer,  
                             long size, unsigned int *histo)  
{  
    __shared__ unsigned int histo_private[7];
```



# 共享内存原子需要私有化

- 为每个线程块创建 histo[] 数组的私有副本

```
__global__ void histo_kernel(unsigned char *buffer,  
                             long size, unsigned int *histo)  
{  
    __shared__ unsigned int histo_private[7];
```

```
    if (threadIdx.x < 7) histo_private[threadIdx.x] = 0;  
    __syncthreads();
```

在 histo[] 的私有副本中  
初始化 bin 计数器



# 建立私有直方图

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
// stride is total number of threads
int stride = blockDim.x * gridDim.x;

int alphabet_position = 0;
while (i < size) {
    alphabet_position = buffer[i] - "a";

    if (alphabet_position >= 0 && alphabet_position < 26)
        atomicAdd(&(private_histo[alphabet_position/4]), 1);

    i += stride;
}
```



# 构建最终直方图

```
// wait for all other threads in the block to finish
__syncthreads();

if (threadIdx.x < 7) {
    atomicAdd(&(histo[threadIdx.x]), private_histo[threadIdx.x] );
}
}
```



# 关于私有化

- 私有化是一种用于并行化应用程序的强大且常用的技术
- 私有直方图大小需要很小
  - 适合共享内存
- 如果直方图太大而无法私有化怎么办？
  - 有时可以将输出直方图部分私有化并使用范围测试 (range testing) 来访问全局内存或共享内存
- 一些数据集在局部区域具有大量相同的数据值
  - 一个简单而有效的优化方法是每个线程在更新直方图的相同元素时将连续更新聚合为单个更新



# ANY QUESTIONS?



电子科技大学  
University of Electronic Science and Technology of China