



GPU Teaching Kit  
Accelerated Computing



## Module 7 – Parallel Computation Patterns (Histogram) Lecture 7.1 - Histogramming



# Objective

- To learn the parallel histogram computation pattern
  - An important, useful computation
  - Very different from all the patterns we have covered so far in terms of output behavior of each thread
  - A good starting point for understanding output interference in parallel computation



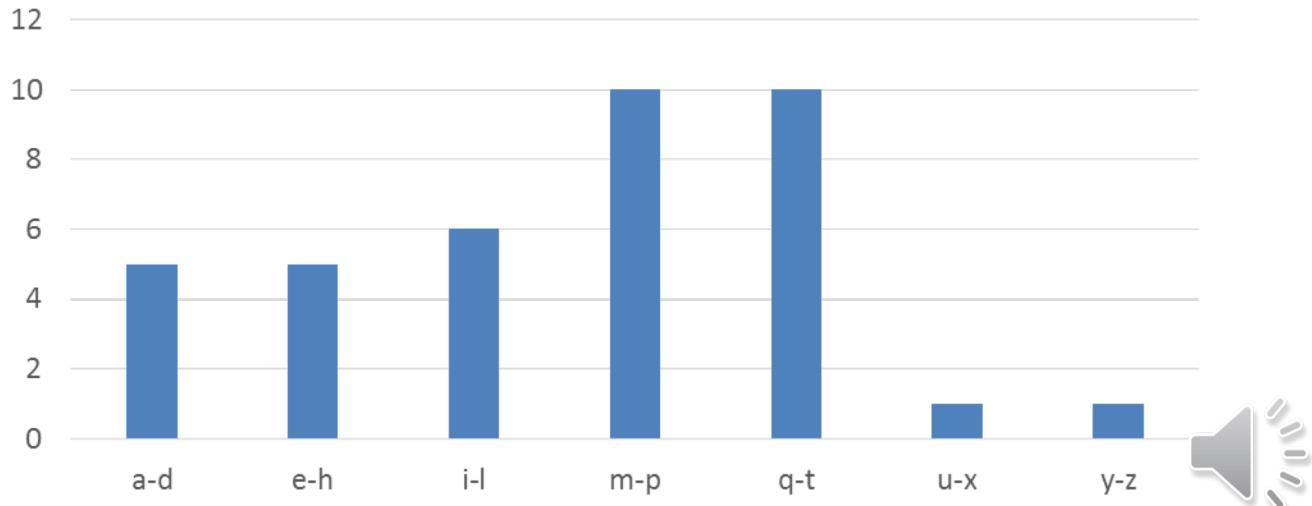
# Histogram

- A method for extracting notable features and patterns from large data sets
  - Feature extraction for object recognition in images
  - Fraud detection in credit card transactions
  - Correlating heavenly object movements in astrophysics
  - ...
- Basic histograms - for each element in the data set, use the value to identify a “bin counter” to increment



# A Text Histogram Example

- Define the bins as four-letter sections of the alphabet: a-d, e-h, i-l, n-p, ...
- For each character in an input string, increment the appropriate bin counter.
- In the phrase “Programming Massively Parallel Processors” the output histogram is shown below:

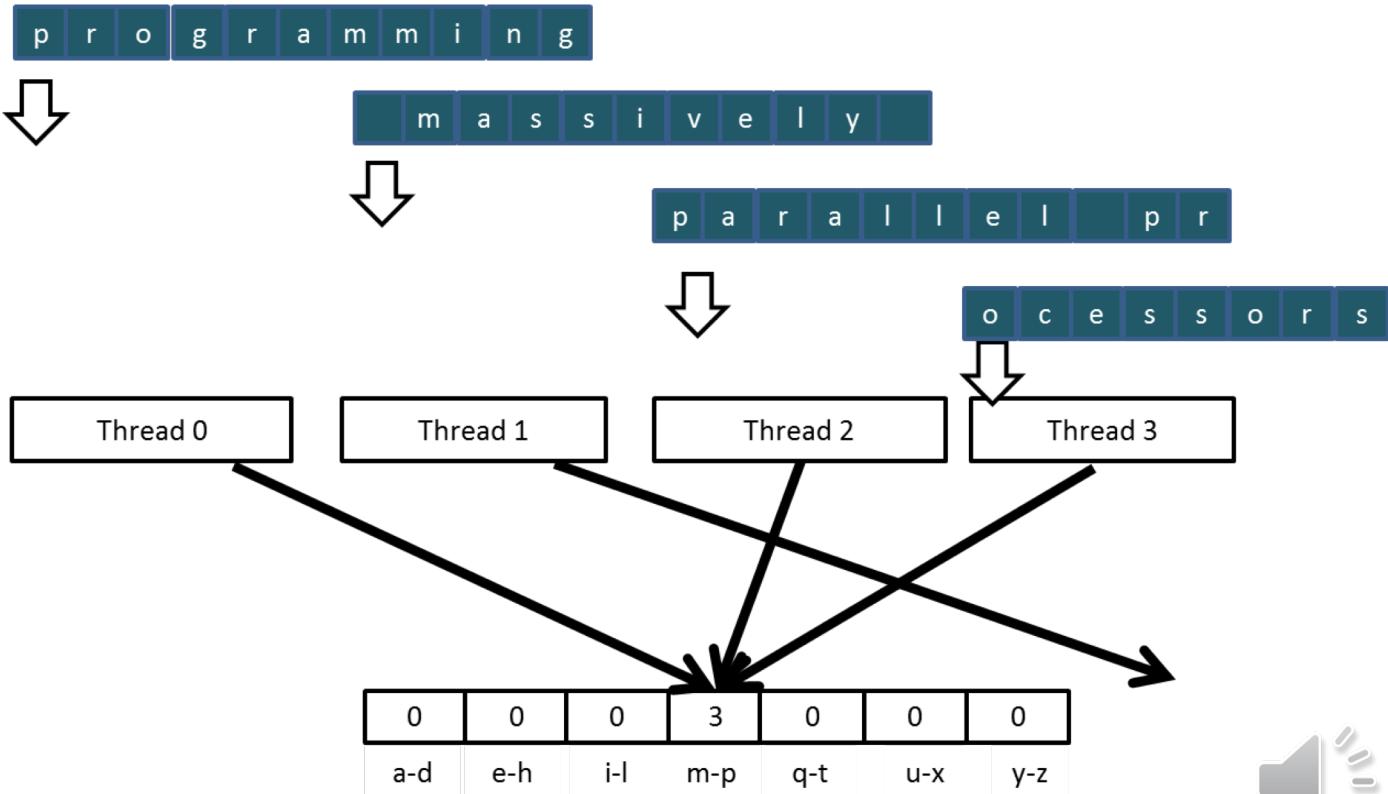


# A simple parallel histogram algorithm

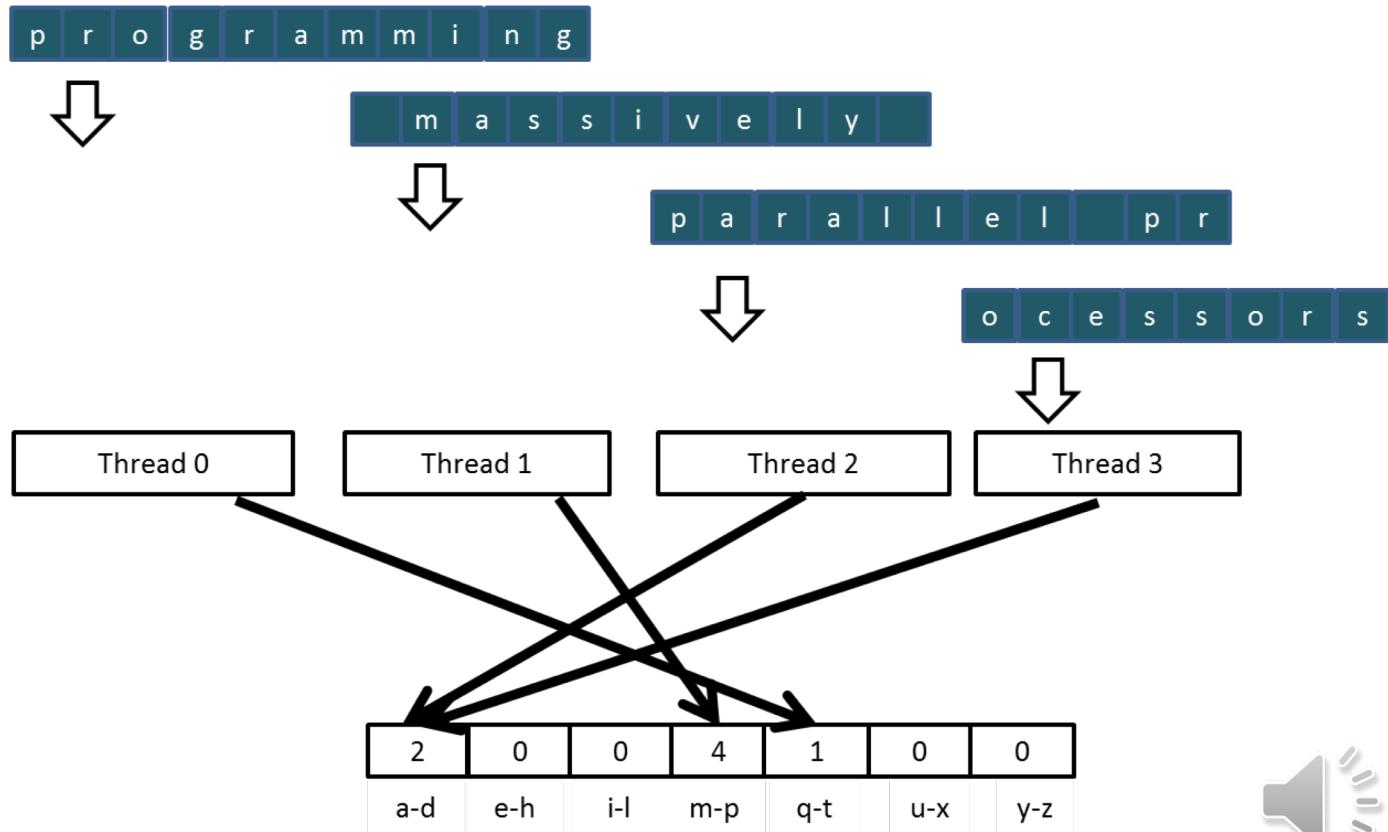
- Partition the input into sections
- Have each thread take a section of the input
- Each thread iterates through its section.
- For each letter, increment the appropriate bin counter



# Sectioned Partitioning (Iteration #1)

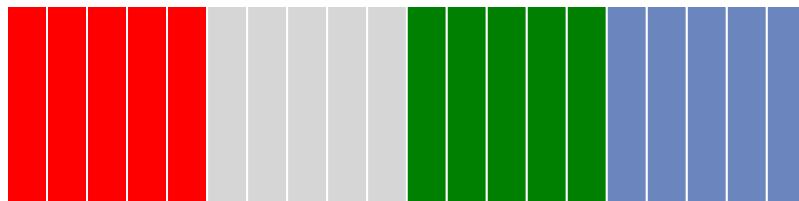


# Sectioned Partitioning (Iteration #2)



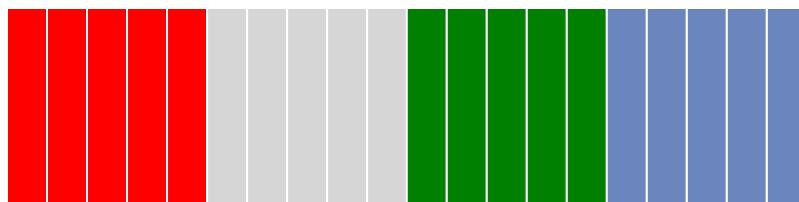
# Input Partitioning Affects Memory Access Efficiency

- Sectioned partitioning results in poor memory access efficiency
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized

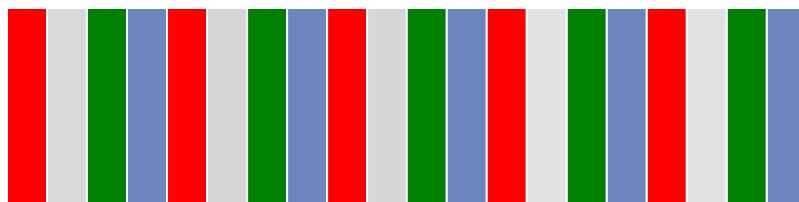


# Input Partitioning Affects Memory Access Efficiency

- Sectioned partitioning results in poor memory access efficiency
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized

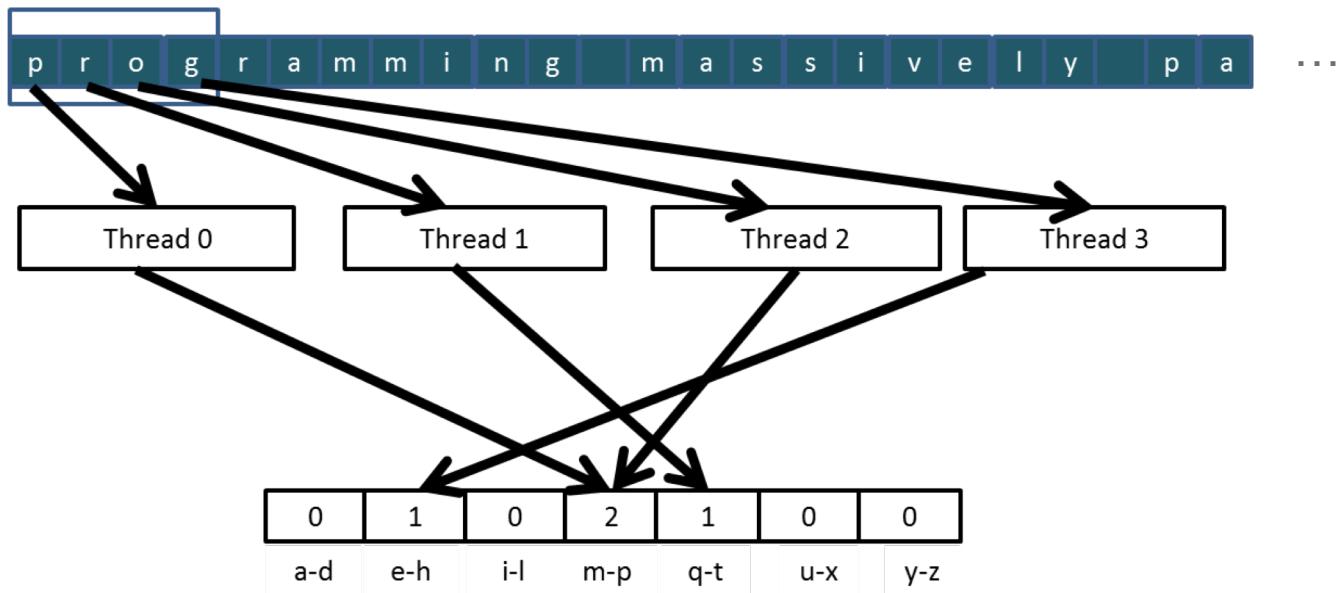


- Change to interleaved partitioning
  - All threads process a contiguous section of elements
  - They all move to the next section and repeat
  - The memory accesses are coalesced

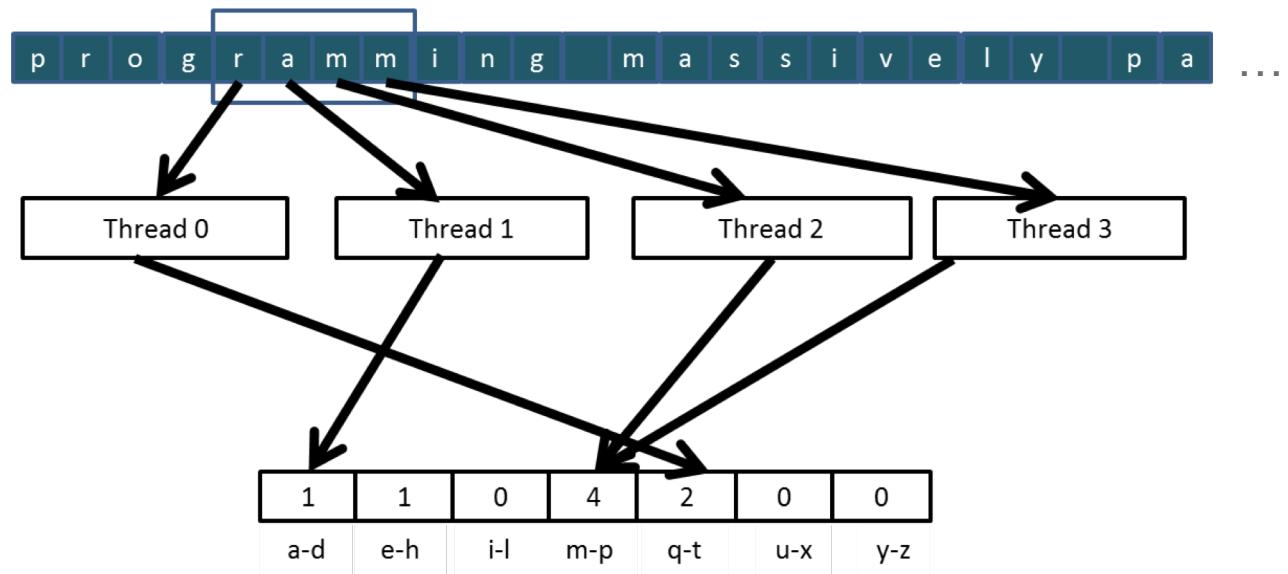


# Interleaved Partitioning of Input

- For coalescing and better memory access performance



# Interleaved Partitioning (Iteration 2)



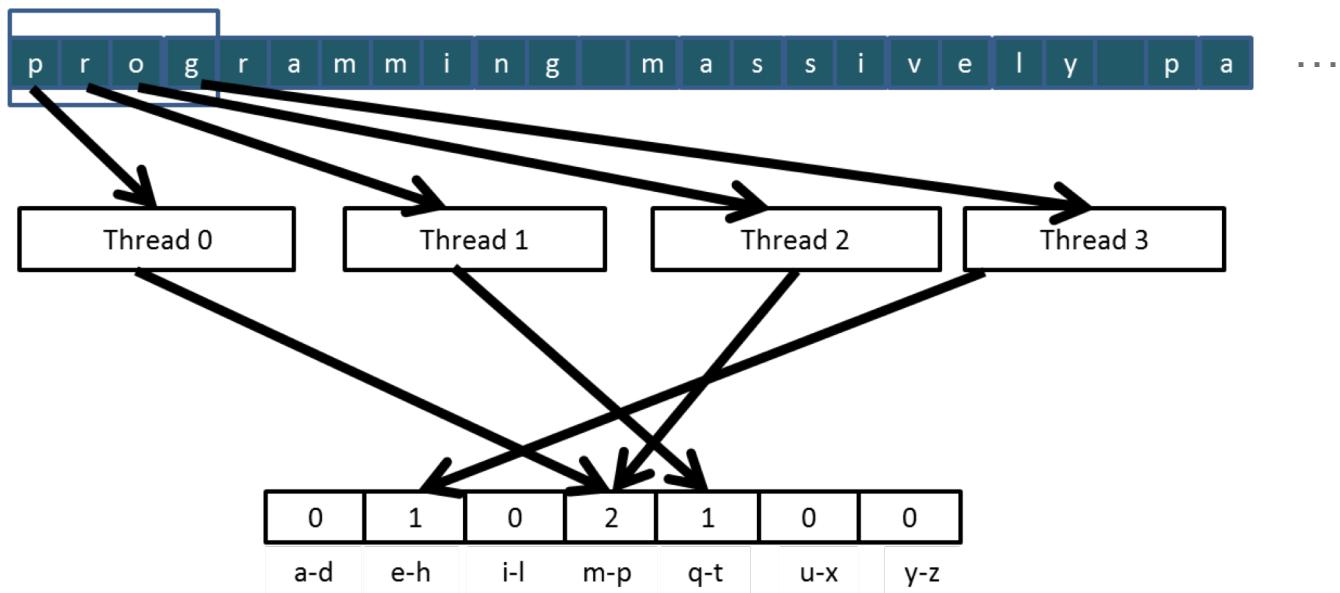
# Objective

- To understand data races in parallel computing
  - Data races can occur when performing read-modify-write operations
  - Data races can cause errors that are hard to reproduce
  - Atomic operations are designed to eliminate such data races



# Read-modify-write in the Text Histogram Example

- For coalescing and better memory access performance



# Read-Modify-Write Used in Collaboration Patterns

- For example, multiple bank tellers count the total amount of cash in the safe
- Each grab a pile and count
- Have a central display of the running total
- Whenever someone finishes counting a pile, read the current running total (read) and add the subtotal of the pile to the running total (modify-write)
- A bad outcome
  - Some of the piles were not accounted for in the final total



# A Common Parallel Service Pattern

- For example, multiple customer service agents serving waiting customers
- The system maintains two numbers,
  - the number to be given to the next incoming customer (I)
  - the number for the customer to be served next (S)
- The system gives each incoming customer a number (read I) and increments the number to be given to the next customer by 1 (modify-write I)
- A central display shows the number for the customer to be served next
- When an agent becomes available, he/she calls the number (read S) and increments the display number by 1 (modify-write S)
- Bad outcomes
  - Multiple customers receive the same number, only one of them receives service
  - Multiple agents serve the same number



# A Common Arbitration Pattern

- For example, multiple customers booking airline tickets in parallel
- Each
  - Brings up a flight seat map (read)
  - Decides on a seat
  - Updates the seat map and marks the selected seat as taken (modify-write)
- A bad outcome
  - Multiple passengers ended up booking the same seat



# Data Race in Parallel Thread Execution

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 and New are per-thread register variables.

Question 1: If  $\text{Mem}[x]$  was initially 0, what would the value of  $\text{Mem}[x]$  be after threads 1 and 2 have completed?

Question 2: What does each thread get in their Old variable?

Unfortunately, the answers may vary according to the relative execution timing between the two threads, which is referred to as a **data race**.



# Timing Scenario #1

Time	Thread 1	Thread 2
1	(0) Old $\leftarrow$ Mem[x]	
2	(1) New $\leftarrow$ Old + 1	
3	(1) Mem[x] $\leftarrow$ New	
4		(1) Old $\leftarrow$ Mem[x]
5		(2) New $\leftarrow$ Old + 1
6		(2) Mem[x] $\leftarrow$ New

- Thread 1 Old = 0
- Thread 2 Old = 1
- Mem[x] = 2 after the sequence



# Timing Scenario #2

Time	Thread 1	Thread 2
1		(0) Old $\leftarrow$ Mem[x]
2		(1) New $\leftarrow$ Old + 1
3		(1) Mem[x] $\leftarrow$ New
4	(1) Old $\leftarrow$ Mem[x]	
5	(2) New $\leftarrow$ Old + 1	
6	(2) Mem[x] $\leftarrow$ New	

- Thread 1 Old = 1
- Thread 2 Old = 0
- Mem[x] = 2 after the sequence



# Timing Scenario #3

Time	Thread 1	Thread 2
1	(0) Old $\leftarrow$ Mem[x]	
2	(1) New $\leftarrow$ Old + 1	
3		(0) Old $\leftarrow$ Mem[x]
4	(1) Mem[x] $\leftarrow$ New	
5		(1) New $\leftarrow$ Old + 1
6		(1) Mem[x] $\leftarrow$ New

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence



# Timing Scenario #4

Time	Thread 1	Thread 2
1		(0) Old $\leftarrow$ Mem[x]
2		(1) New $\leftarrow$ Old + 1
3	(0) Old $\leftarrow$ Mem[x]	
4		(1) Mem[x] $\leftarrow$ New
5	(1) New $\leftarrow$ Old + 1	
6	(1) Mem[x] $\leftarrow$ New	

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence



# Purpose of Atomic Operations – To Ensure Good Outcomes

```
thread1: Old ← Mem[x]  
        New ← Old + 1  
        Mem[x] ← New
```

```
thread2: Old ← Mem[x]  
        New ← Old + 1  
        Mem[x] ← New
```

Or

```
thread1: Old ← Mem[x]  
        New ← Old + 1  
        Mem[x] ← New
```

```
thread2: Old ← Mem[x]  
        New ← Old + 1  
        Mem[x] ← New
```



# Objective

- To learn to use atomic operations in parallel programming
  - Atomic operation concepts
  - Types of atomic operations in CUDA
  - Intrinsic functions
  - A basic histogram kernel



# Data Race Without Atomic Operations

Mem[x] initialized to 0

thread1: Old  $\leftarrow$  Mem[x]

time  
↓  
New  $\leftarrow$  Old + 1

Mem[x]  $\leftarrow$  New

thread2: Old  $\leftarrow$  Mem[x]

New  $\leftarrow$  Old + 1

Mem[x]  $\leftarrow$  New

- Both threads receive 0 in Old
- Mem[x] becomes 1



# Key Concepts of Atomic Operations

- A read-modify-write operation performed by a single hardware instruction on a memory location *address*
  - Read the old value, calculate a new value, and write the new value to the location
- The hardware ensures that no other threads can perform another read-modify-write operation on the same location until the current atomic operation is complete
  - Any other threads that attempt to perform an atomic operation on the same location will typically be held in a queue
  - All threads perform their atomic operations **serially** on the same location



# Atomic Operations in CUDA

- Performed by calling functions that are translated into single instructions (a.k.a. *intrinsic functions* or *intrinsics*)
  - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
  - Read CUDA C programming Guide 4.0 or later for details
- Atomic Add

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

  - reads the 32-bit word **old** from the location pointed to by **address** in global or shared memory, computes (**old + val**), and stores the result back to memory at the same address. The function returns **old**.



# More Atomic Adds in CUDA

- 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);



# A Basic Text Histogram Kernel

- The kernel receives a pointer to the input buffer of byte values
- Each thread process the input in a strided pattern

```
__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) {
        atomicAdd( &(histo[buffer[i]]), 1);
        i += stride;
    }
}
```



# A Basic Histogram Kernel (cont.)

- The kernel receives a pointer to the input buffer of byte values
- Each thread process the input in a strided pattern

```
__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 && alpha_position < 26)
            atomicAdd(&(histo[alphabet position/4]), 1);
        i += stride;
    }
}
```



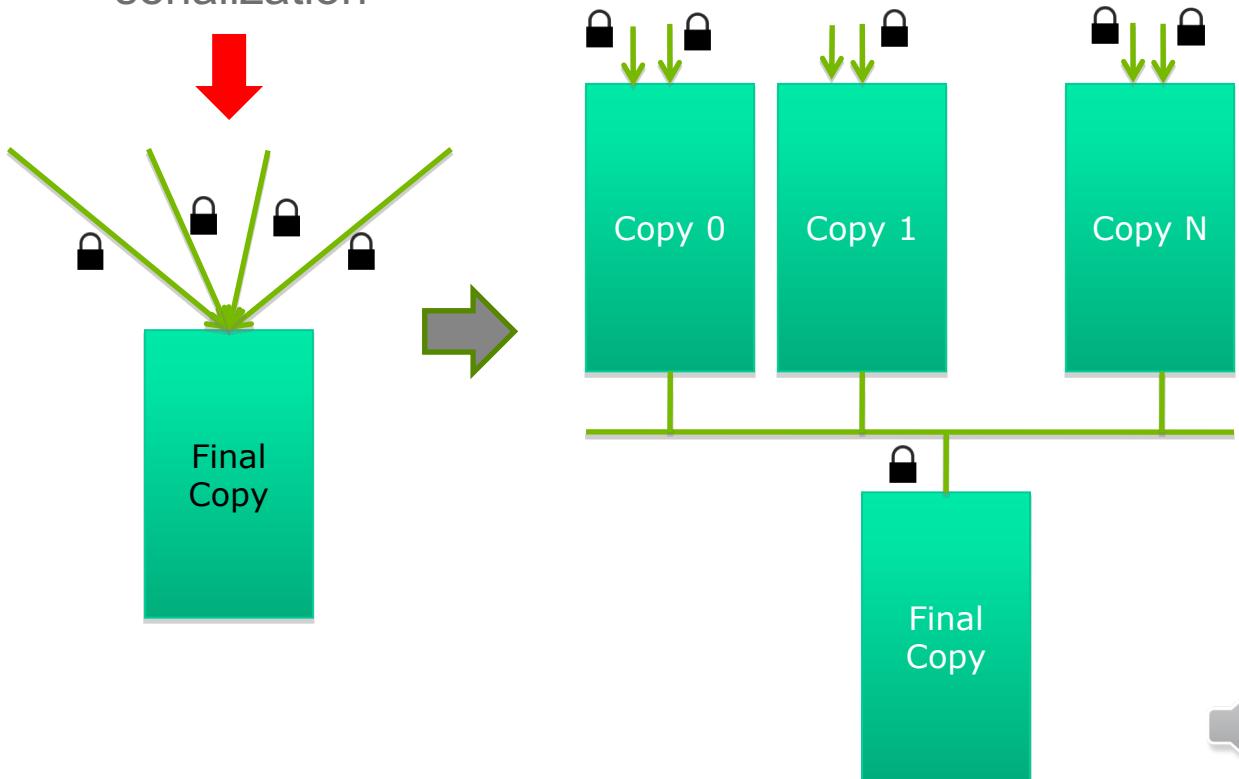
# Objective

- Learn to write a high performance kernel by privatizing outputs
  - Privatization as a technique for reducing latency, increasing throughput, and reducing serialization
  - A high performance privatized histogram kernel
  - Practical example of using shared memory and L2 cache atomic operations



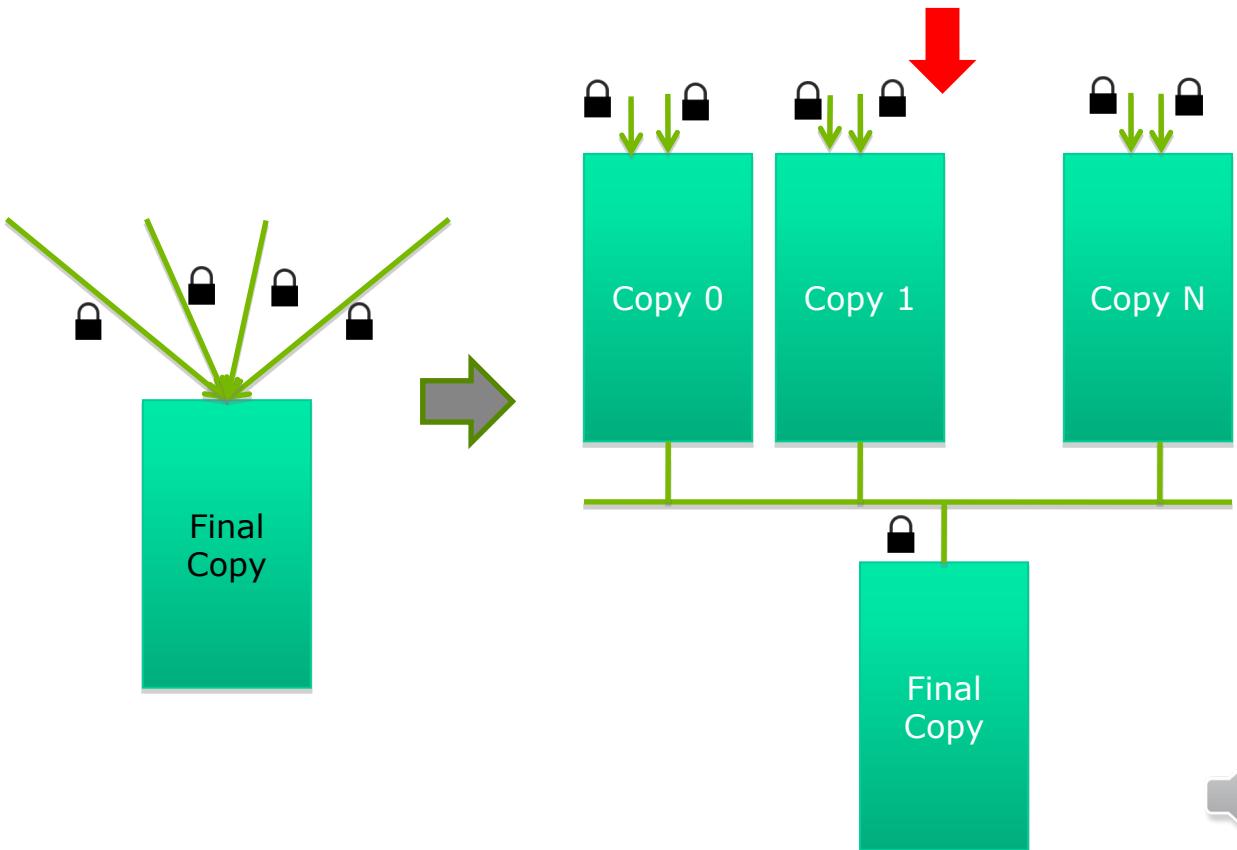
# Privatization

Heavy contention and serialization

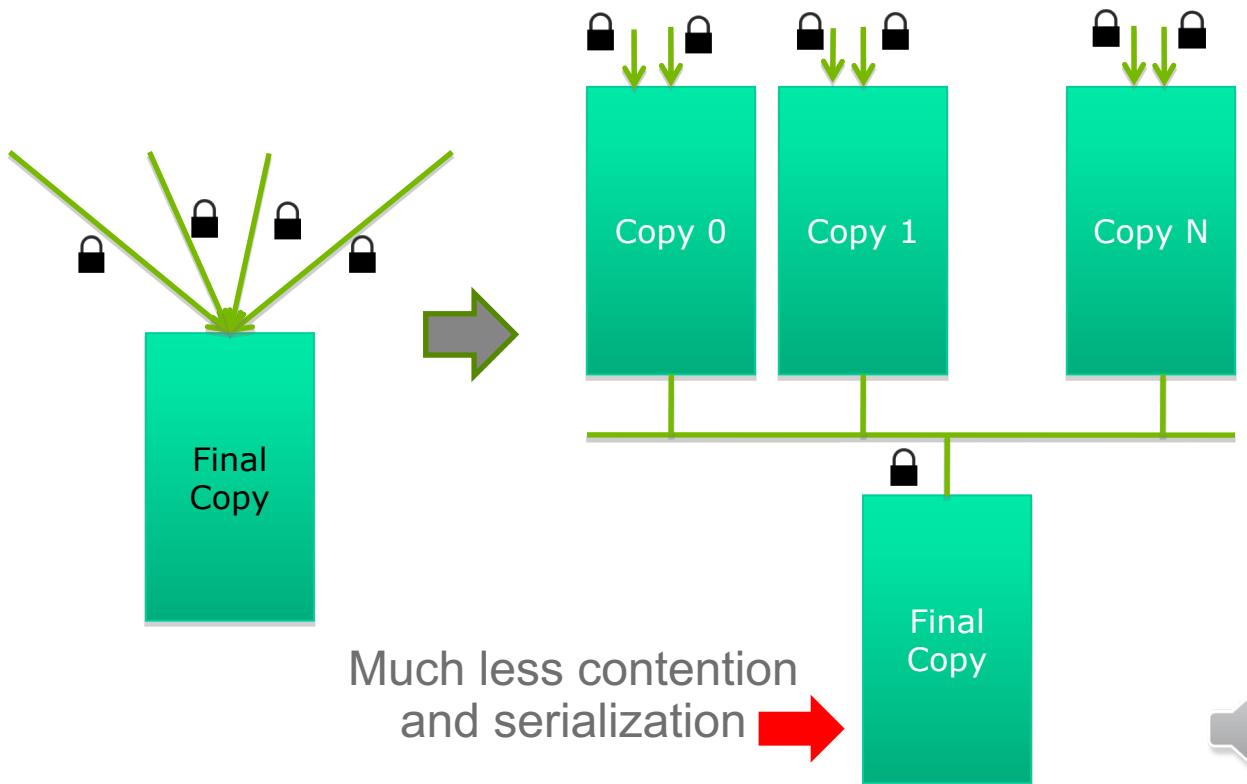


# Privatization (cont.)

Much less contention and serialization



# Privatization (cont.)



# Cost and Benefit of Privatization

- Cost
  - Overhead for creating and initializing private copies
  - Overhead for accumulating the contents of private copies into the final copy
- Benefit
  - Much less contention and serialization in accessing both the private copies and the final copy
  - The overall performance can often be improved more than 10x



# Shared Memory Atomics for Histogram

- Each subset of threads are in the same block
- Much higher throughput than DRAM (100x) or L2 (10x) atomics
- Less contention – only threads in the same block can access a shared memory variable
- This is a very important use case for shared memory!



# Shared Memory Atomics Requires Privatization

- Create private copies of the histo[] array for each thread block

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



# Shared Memory Atomics Requires Privatization

- Create private copies of the histo[] array for each thread block

```
__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();
```

Initialize the bin counters in  
the private copies of histo[]



# Build Private Histogram

```
int i = threadIdx.x + blockIdx.x * blockDim.x;  
// stride is total number of threads  
int stride = blockDim.x * gridDim.x;  
while (i < size) {  
    atomicAdd( &(private_histo[buffer[i]/4], 1);  
    i += stride;  
}
```



# Build Final Histogram

```
// wait for all other threads in the block to finish  
__syncthreads();  
  
if (threadIdx.x < 7) {  
    atomicAdd(&(histo[threadIdx.x]), private_histo[threadIdx.x] );  
}  
  
}
```



# More on Privatization

- Privatization is a powerful and frequently used technique for parallelizing applications
- The operation needs to be associative and commutative
  - Histogram add operation is associative and commutative
  - No privatization if the operation does not fit the requirement
- The private histogram size needs to be small
  - Fits into shared memory
- What if the histogram is too large to privatize?
  - Sometimes one can partially privatize an output histogram and use range testing to go to either global memory or shared memory

