NVIDIA

GPU Teaching Kit

Accelerated Computing

ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

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

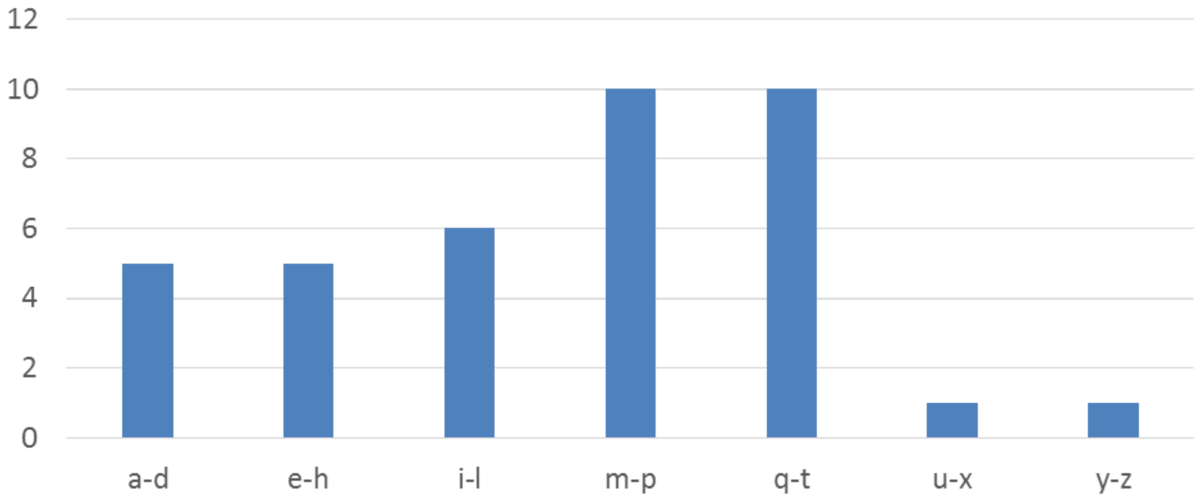
Histogram

- 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

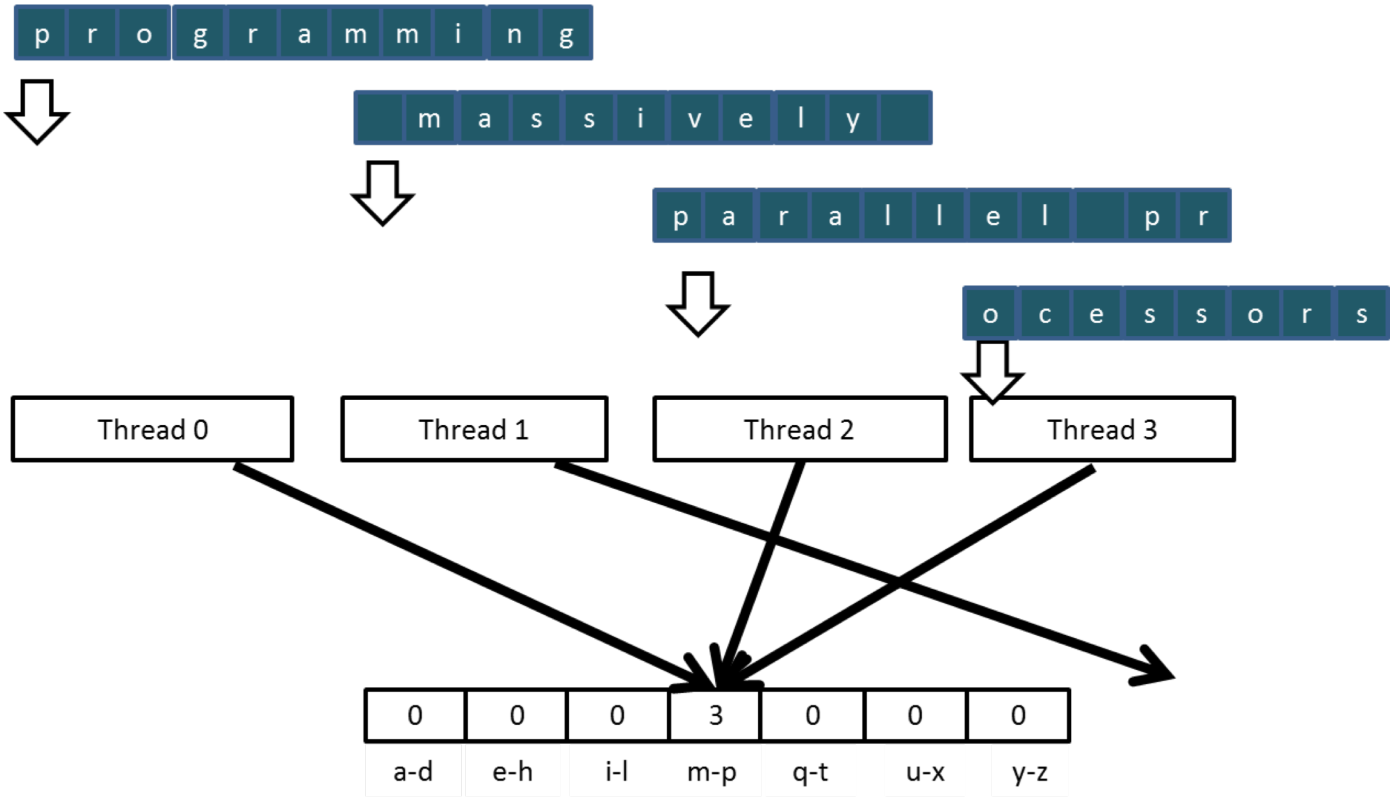
- Define the bins as four-letter sections of the alphabet: a-d, e-h, i-l, n-p, …
- For each character in an input string, increment the appropriate bin counter.
- In the phrase "Programming Massively Parallel Processors" the output histogram is shown below:



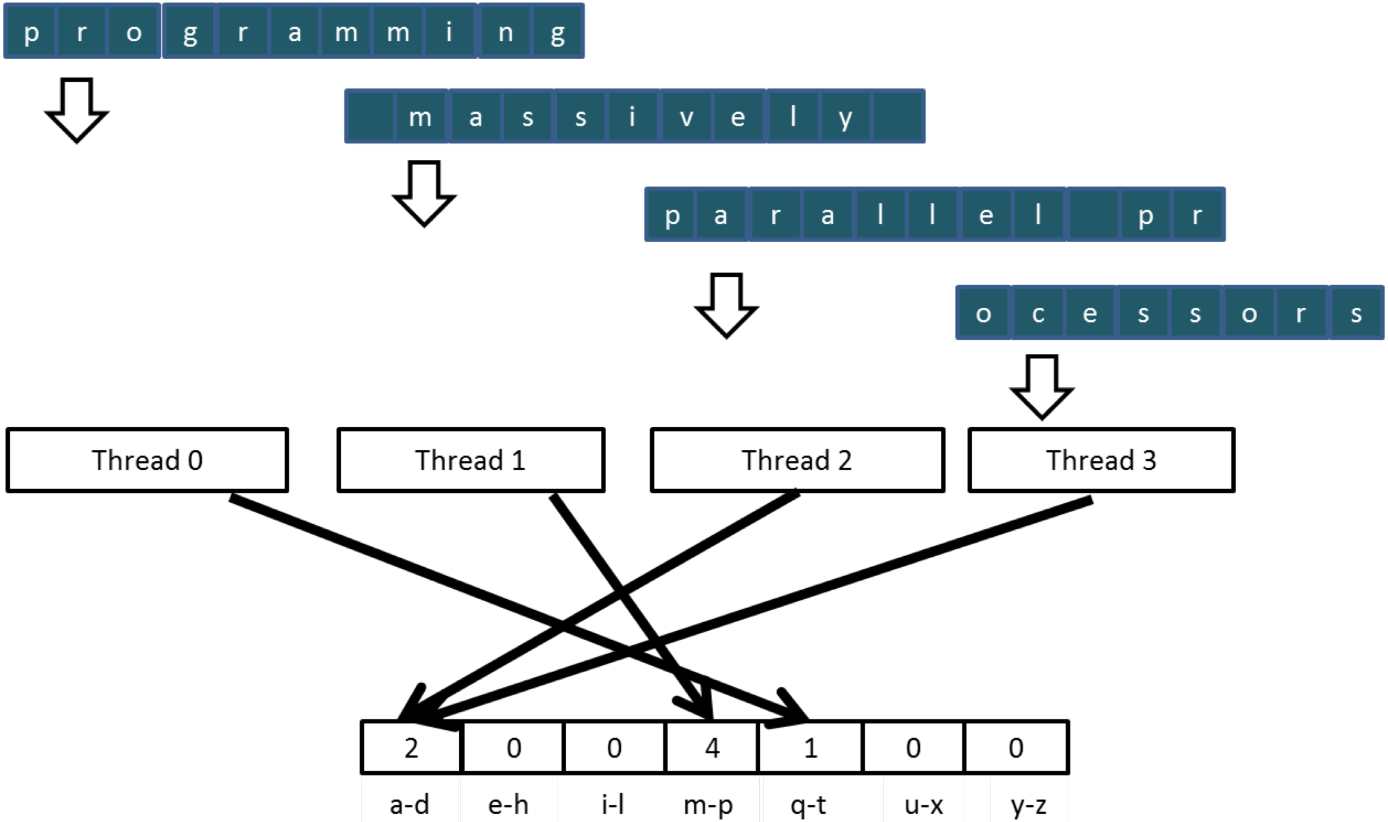
A simple parallel histogram algorithm

- Partition the input into sections
- Have each thread to 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)



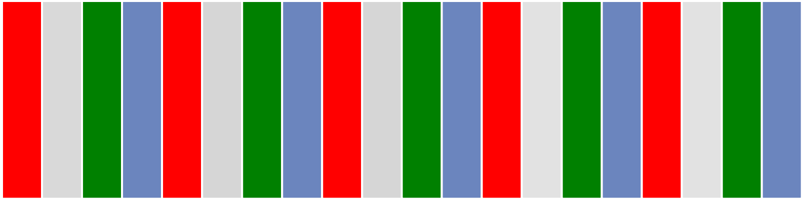
Input Partitioning Affects Memory Access Efficiency

- 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

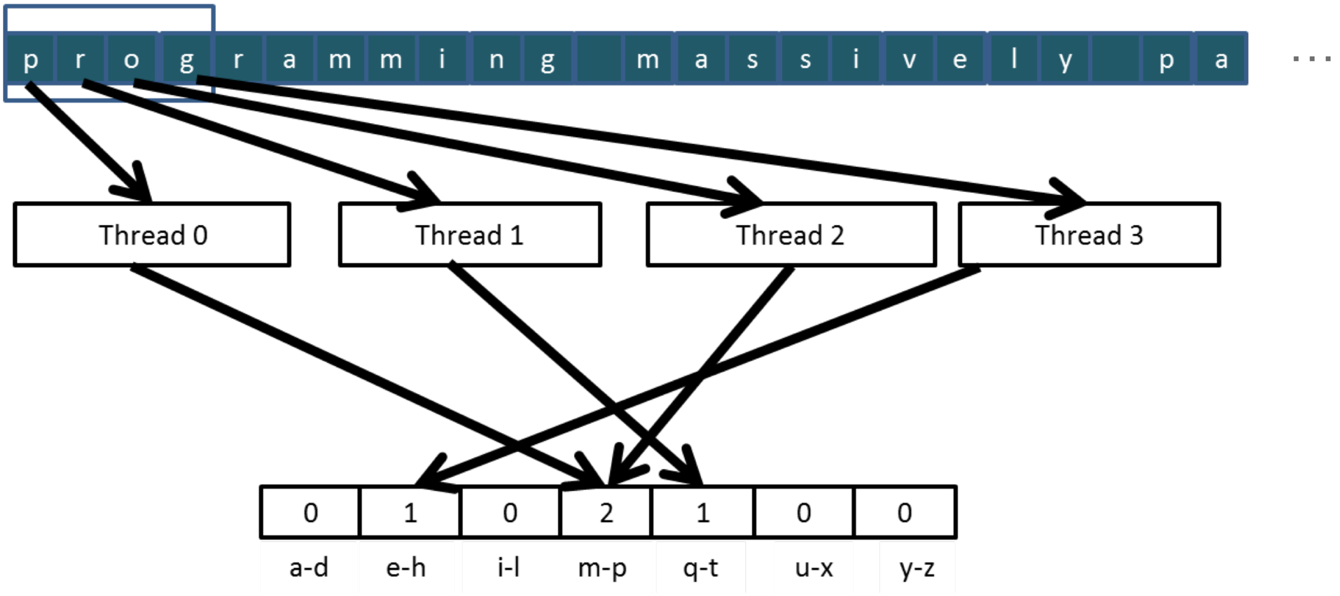
- Sectioned partitioning results in poor memory access efficiency
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized

- 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

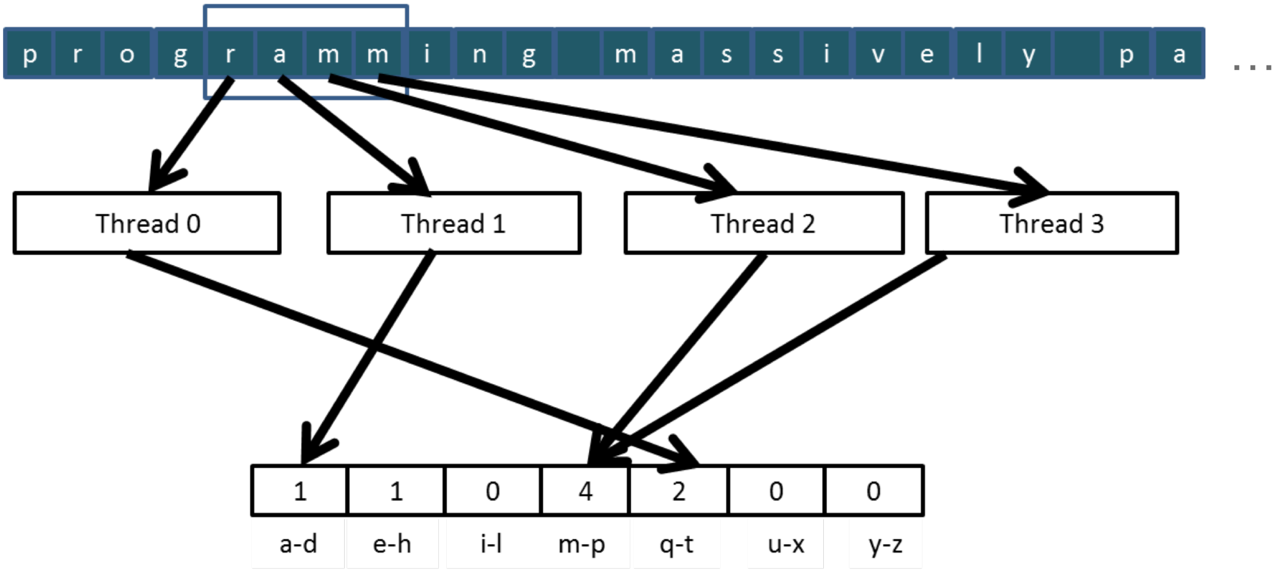


Interleaved Partitioning of Input

– For coalescing and better memory access performance



Interleaved Partitioning (Iteration 2)

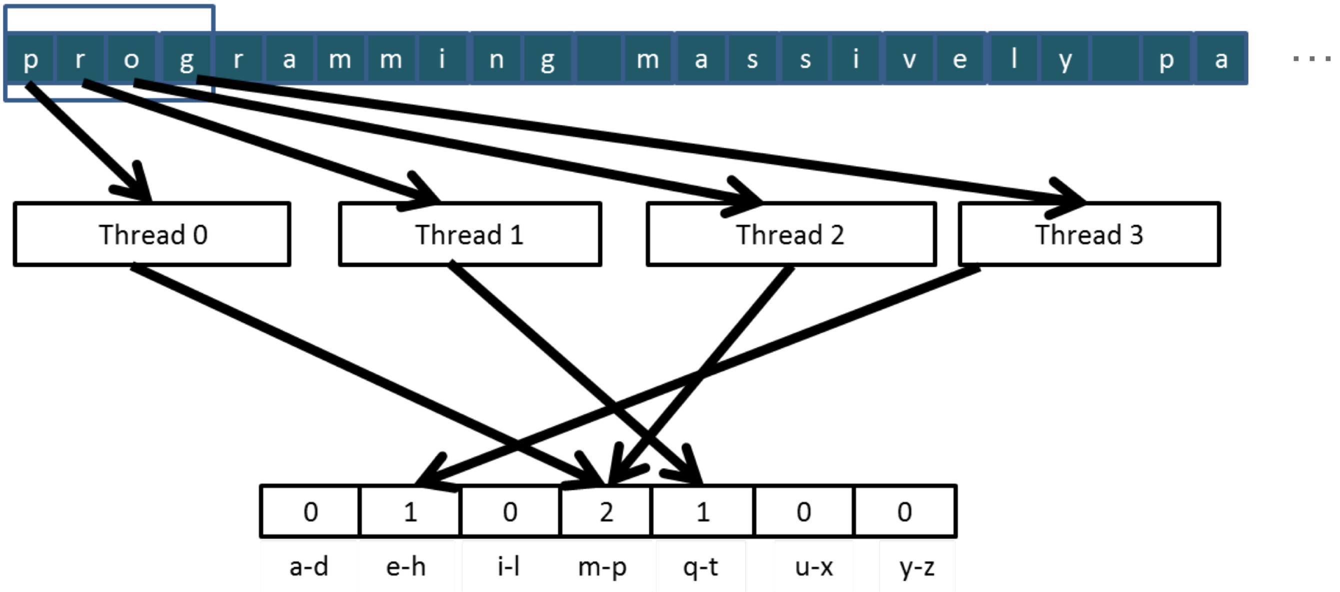


Objective

- 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



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

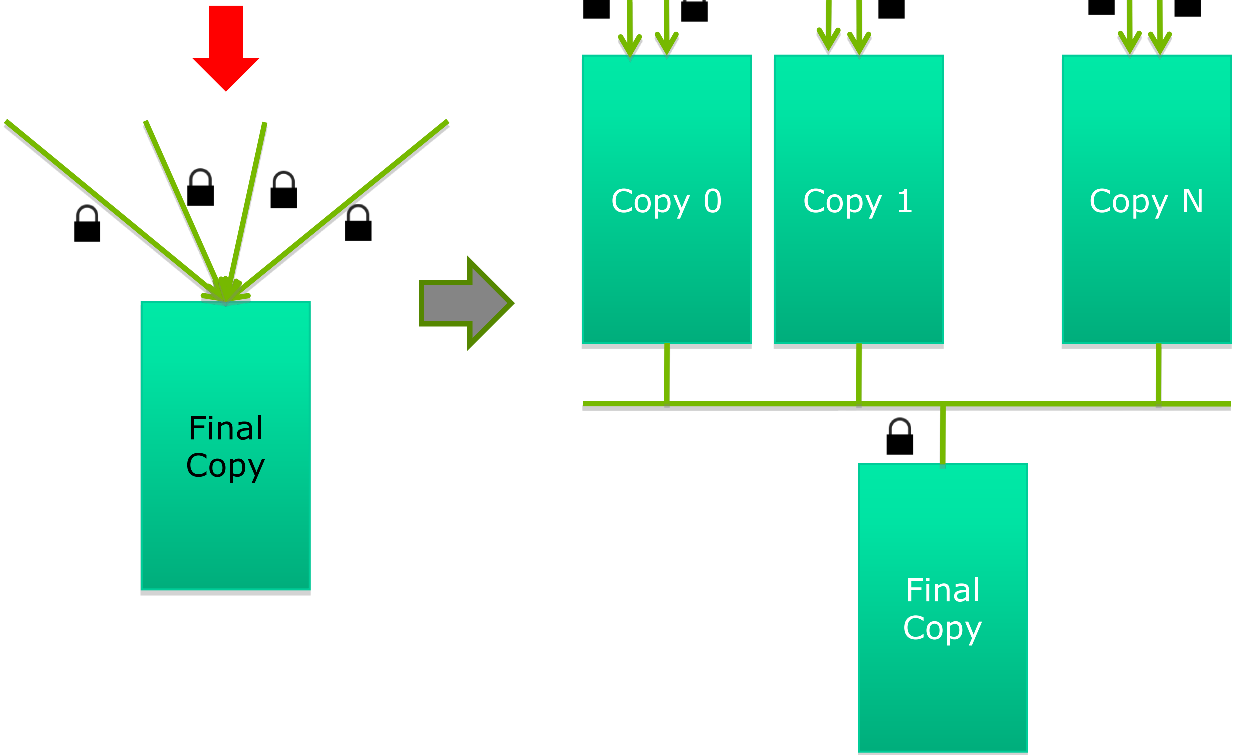
```
__global__ void histo_kernel(unsigned char *buffer,
       long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

// 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

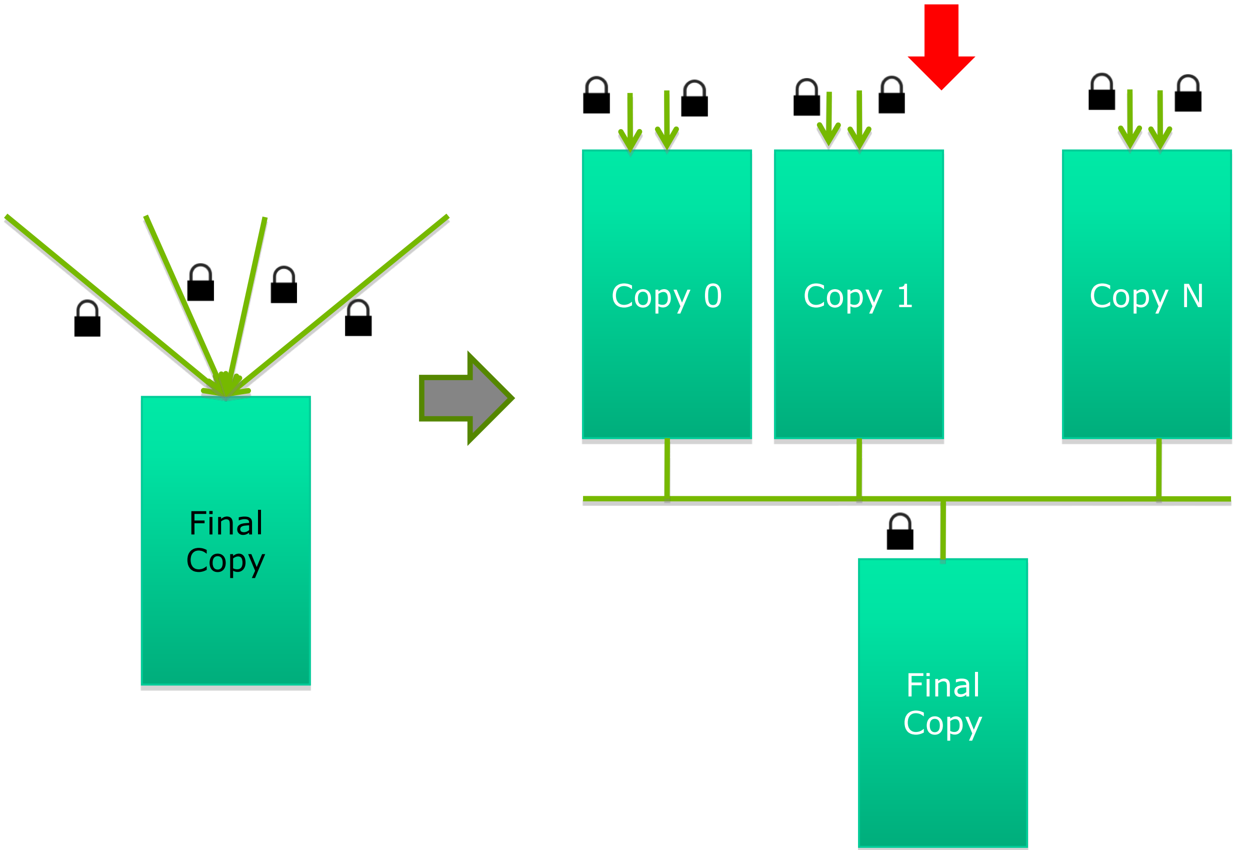
Privatization

Heavy contention and serialization

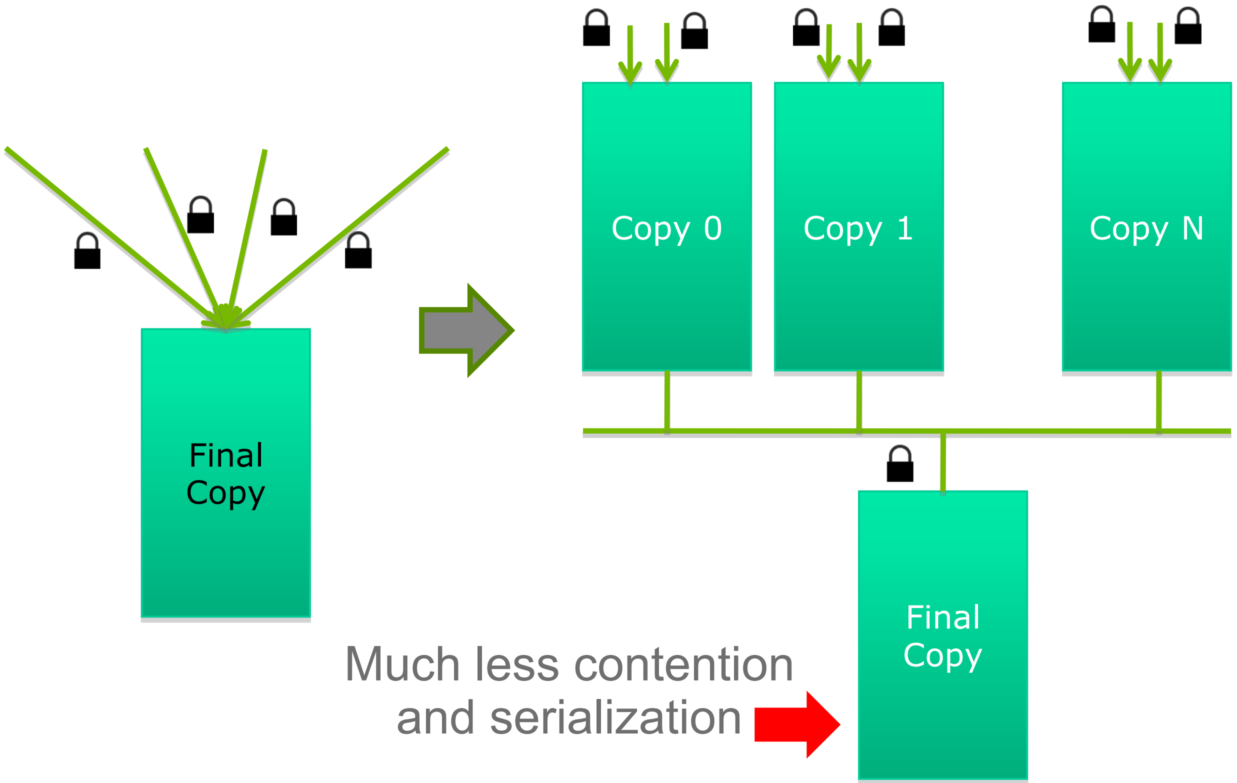


Privatization (cont.)

Much less contention and serialization



Privatization (cont.)



Much less contention and serialization

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] );
 }

}
```