CENG443
Heterogeneous Parallel Programming

Histogram

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

Show the frequency of alphabets in the phrase “programming massively parallel processors”

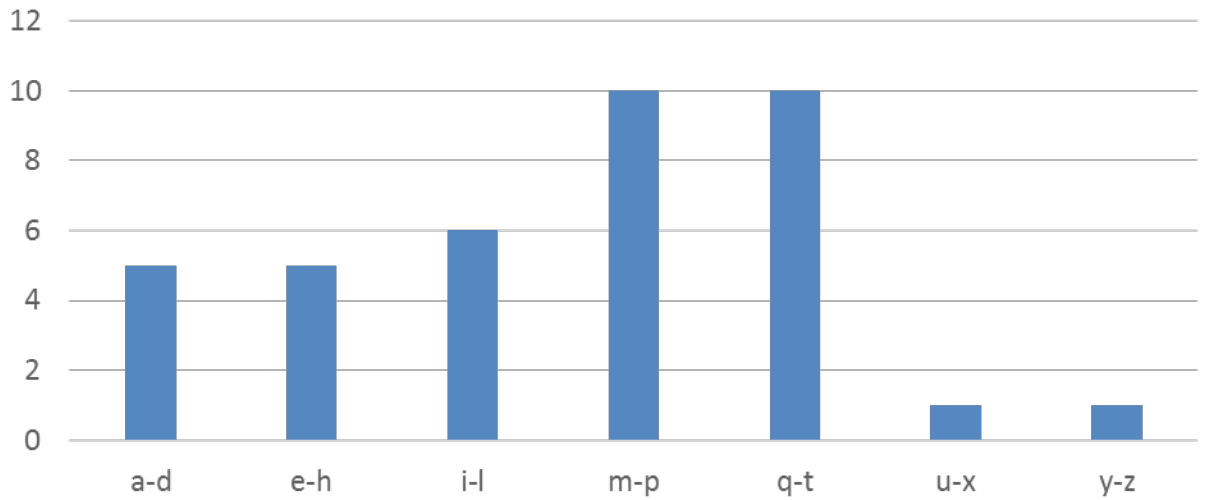
Four “a” letters, zero ‘b” letters, one “c” letter, and so on

Define each value interval (bin) as a continuous range of four alphabets: a-d, e-h, i-l, m-p, q-t, u-x, y-z; 7 bins total

For each character in an input string, increment the appropriate bin counter

A Text Histogram Example

Histogram representation of “programming massively parallel processors”



C Function for Text Histogram

```
sequential_Histogram(char *data, int length, int *histo) {
    for (int i = 0; i < length; i++) {
      int alphabet_position = data[i] – ‘a’;
      if (alphabet_position >= 0 && alphabet_position < 26) {
            histo[alphabet_position/4]++;
      }
   }
}
```

Parallel Algorithm

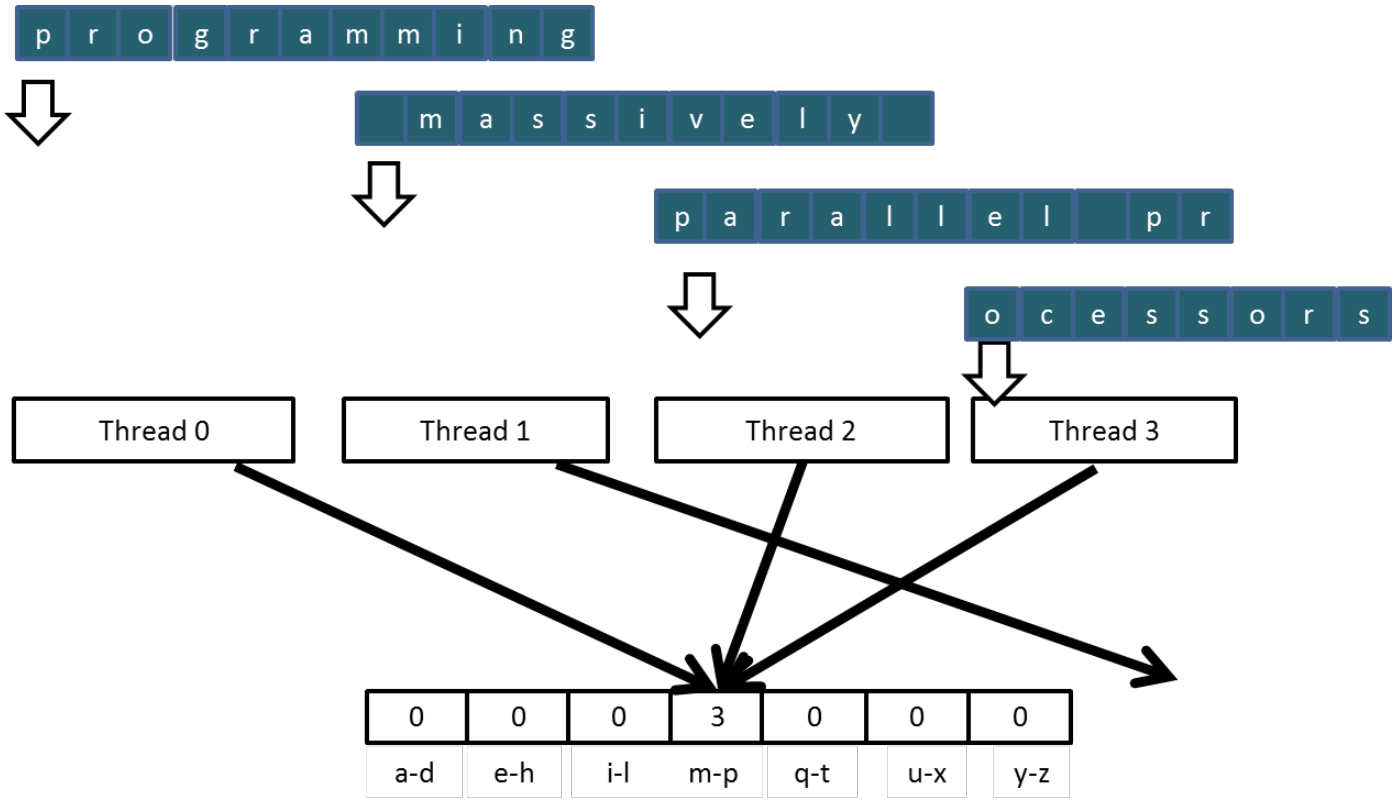
Partition the input into sections

Have each thread to take a section of the input

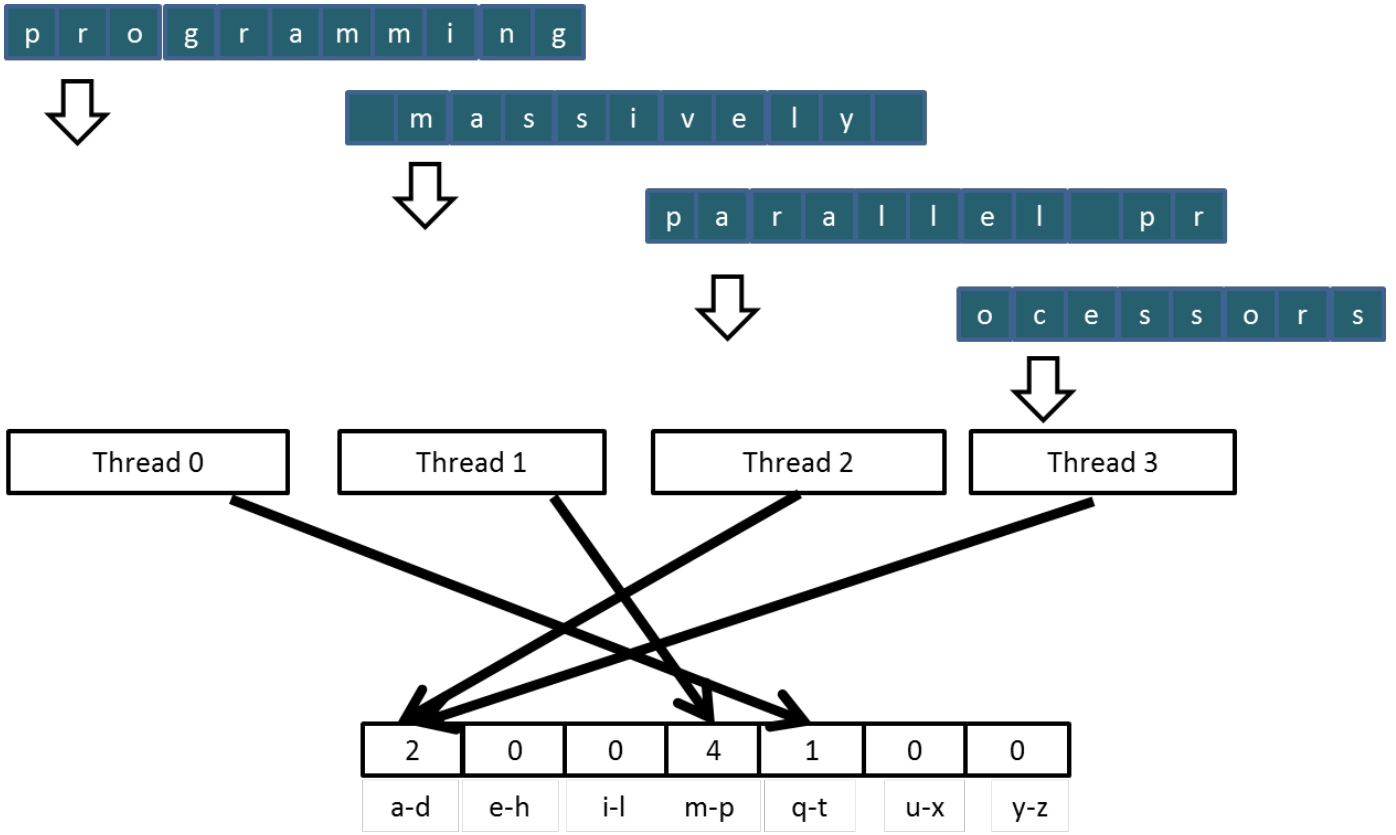
Each thread iterates through its section

For each letter, increment the appropriate bin counter

Sectioned Partitioning (Iteration#1)



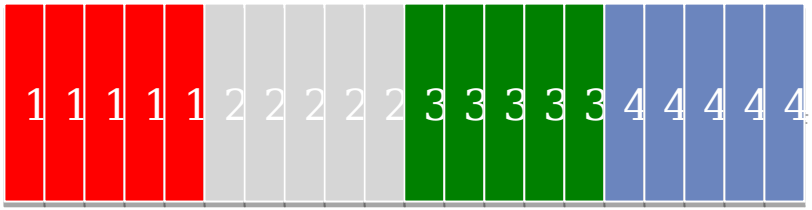
Sectioned Partitioning (Iteration#2)



Sectioned Partitioning

Results in poor memory access efficiency

Adjacent threads do not access adjacent memory locations, accesses are not coalesced



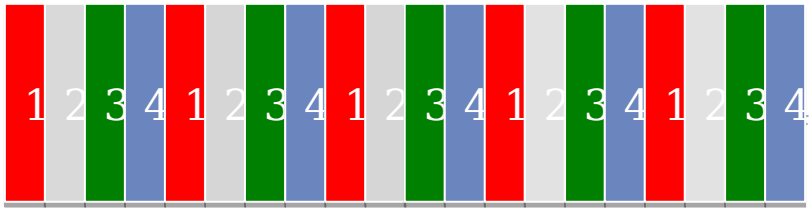
Change to interleaved partitioning

Interleaved Partitioning

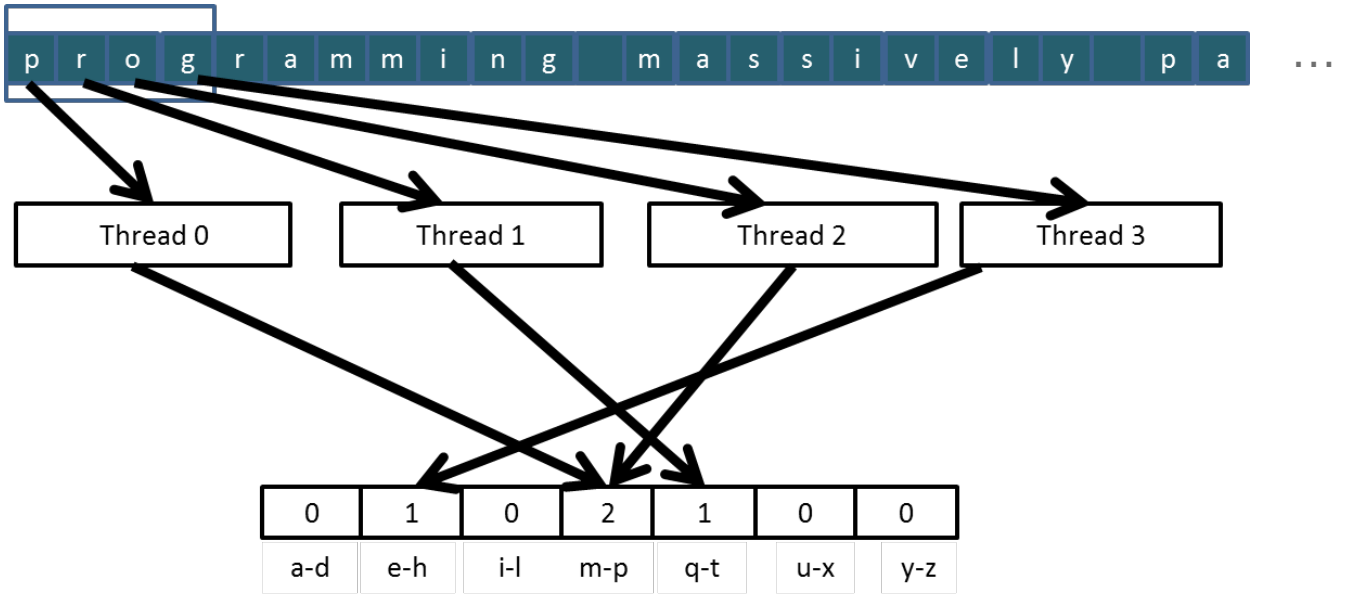
All threads process a contiguous section of elements

They all move to the next section and repeat

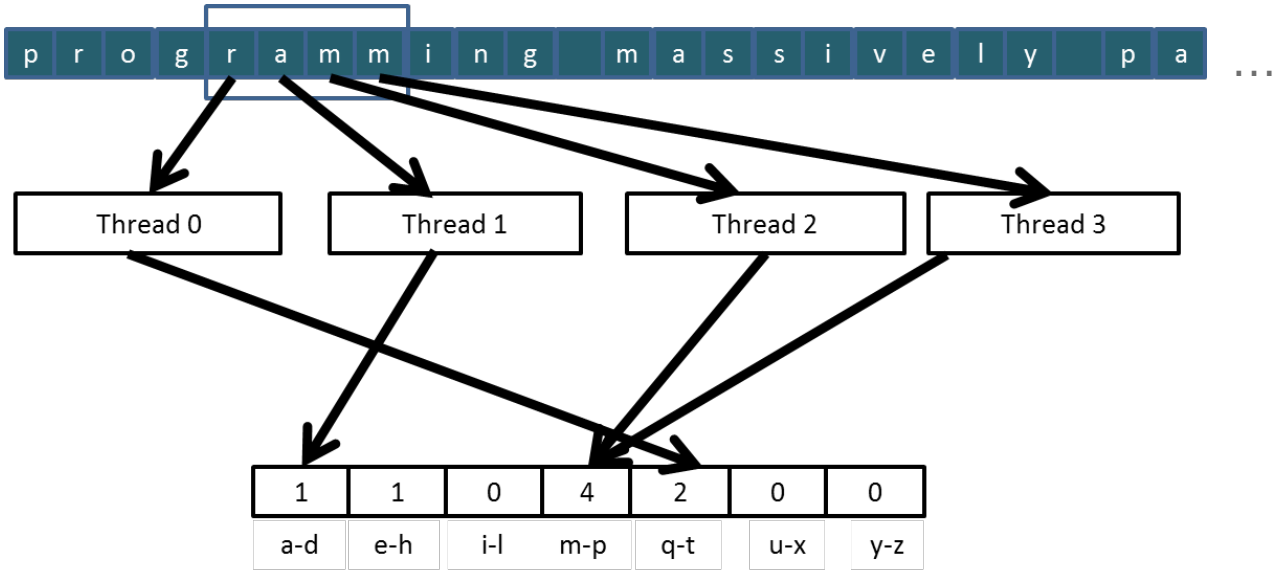
The memory accesses are coalesced



Interleaved Partitioning (Iteration#1)



Interleaved Partitioning (Iteration#2)



Data Race in Parallel Thread Execution

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

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

Old and New are per-thread register variables

Question 1: If Mem[x] was initially 0, what would the value of 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 ← Mem[x]	
2	(1) New ← Old +1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old +1
6		(2) Mem[x] ← 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 ← Mem[x]
2		(1) New ← Old + 1
3		(1) Mem[x] ← New
4	(1) Old ← Mem[x]	
5	(2) New ← Old + 1	
6	(2) Mem[x] ← 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 ← Mem[x]	
2	(1) New ← Old +1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old +1
6		(1) Mem[x] ← 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 ← Mem[x]
2		(1) New ← Old + 1
3	(0) Old ← Mem[x]	
4		(1) Mem[x] ← New
5	(1) New ← Old + 1	
6	(1) Mem[x] ← New	

Thread 1 Old = 0

Thread 2 Old = 0

Mem[x] = 1 after the sequence

Data Race for Histogram Array

Time	Thread 1	Thread 2
1		(0) Old ← histo[x]
2		(1) New ← Old + 1
3		(1) histo[x] ← New
4	(1) Old ← histo[x]	
5	(2) New ← Old + 1	
6	(2) histo[x] ← New	

Time	Thread 1	Thread 2
1		(0) Old ← histo[x]
2		(1) New ← Old + 1
3	(0) Old ← histo[x]	
4		(1) histo[x] ← New
5	(1) New ← Old + 1	
6	(1) histo[x] ← New	

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

Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)

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.

Text Histogram Kernel

```
__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;
   }
}
```

Atomic Operations on DRAM

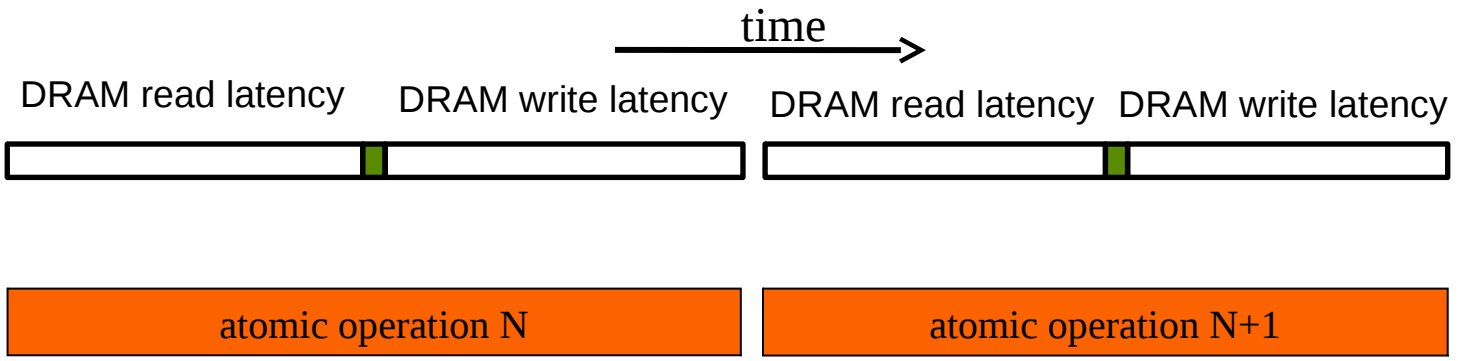
An atomic operation on a DRAM location starts with a read, which has a latency of a few hundred cycles

The atomic operation ends with a write to the same location, with a latency of a few hundred cycles

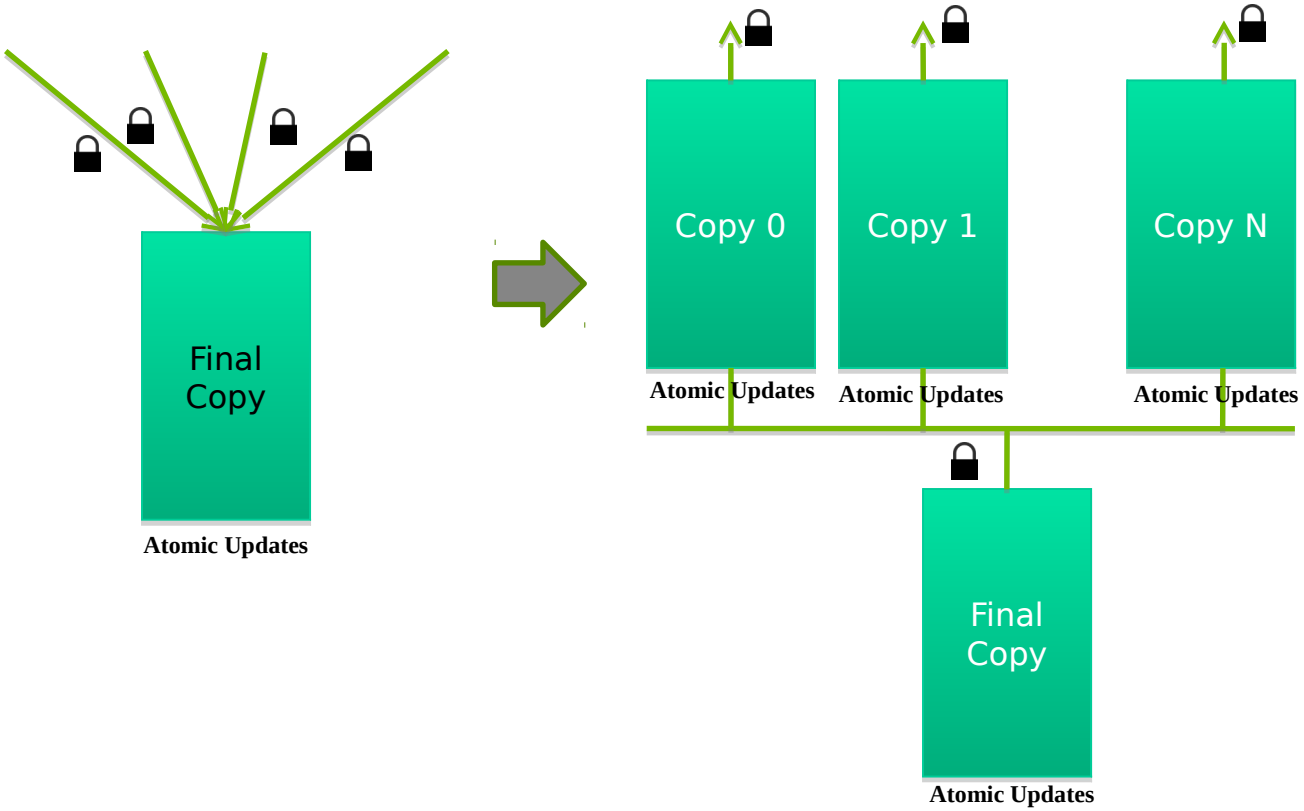
During this whole time, no one else can access the location

Atomic Operations on DRAM

Each Read-Modify-Write has two full memory access delays



Privatization



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

Private Copies of Histogram

```
__global__ void private_histo_kernel(unsigned char *buffer, long size,
unsigned int *histo)
{
    __shared__ unsigned int private_histo[7];
   //The private histogram size needs to be small, Fits into shared memory

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

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) {
     int alphabet_position = buffer[i] - 'a';
     if (alphabet_position >= 0 && alphabet_position < 26)
        atomicAdd( &(private_histo[alphabet_position/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] );
  }

}
```

Aggregation

Some data sets have a large concentration of identical data values in localized areas

- For example, in pictures of the sky, there can be large patches of pixels of identical value

Such high concentration of identical values causes heavy contention and reduced throughput of parallel histogram computation

Simple optimization: For each thread to aggregate consecutive updates into a single update if they are updating the same element of the histogram

Aggregation Code

```
int alphabet_position = buffer[i] - 'a';
if (alphabet_position >= 0 && alpha_position < 26) {
  unsigned int curr_index = alphabet_position/4;
  if(curr_index != prev_index) {
    if (accumulator >= 0)
       atomicAdd(&(histo_s[alphabet_position/4]), accumulator);
    accumulator = 1;
    prev_index = curr_index;
 }else {
    accumulator++;
  }
}
```