ECE408

Applied Parallel Programming

Lecture 18
Atomic Operations and Histogramming

Objective

- To understand atomic operations
  - Read-modify-write in parallel computation
  - A primitive form of “critical regions” in parallel programs
  - Use of atomic operations in CUDA
  - Why atomic operations reduce memory system throughput
  - How to avoid atomic operations in some parallel algorithms

- Histogramming as an example application of atomic operations
  - Basic histogram algorithm
  - Privatization

A Common Arbitration Pattern

- Multiple customers booking airline tickets
- Each one:
  - Brings up a flight seat map
  - Decides on a seat
  - Update the seat map, mark the seat as taken

- A bad outcome: Multiple passengers ended up booking the same seat

Read-Modify-Write Operations

	Old ← Mem[x]		Old ← Mem[x]
thread1:	New ← Old + 1	thread2:	New ← Old + 1
	Mem[x] ← New		Mem[x] ← New

If Mem[x] was **initially 0**, what would the value of Mem[x] be after threads 1 and 2 have completed?

– What does each thread get in their Old variable?

The answer may vary due to data races. To avoid data races, we need to use **atomic operations**

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

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

```
                                  thread2: Old ← Mem[x]
                                           New ← Old + 1
                                           Mem[x] ← New
thread1: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

Atomic Operations in General

- Typically performed by a single instruction in the processor's instruction set 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 access the location until the atomic operation is complete
  - Any other threads that access the location will typically be stalled or held in a queue until its turn
  - All threads perform the atomic operation **serially**

Atomic Operations in CUDA

- Function calls that are translated into single instructions (a.k.a. *intrinsics*)
  - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
  - Read CUDA C programming Guide for more details

- Atomic Add

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

reads the 32-bit word **old** 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

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

Building synchronization with atomics

- How would we build `__syncthreads()` for block
- How would we create `__syncthreads()` for entire grid?
  - And why would this not be a good idea?

- How would we create a critical section? I.e., one thread per block executing a particular section of code?

- How would we create a critical section per grid?
  - Why doesn't this have the same issue as `__syncthreads()` for grid?

Building synchronization with atomics

- How would we build `__syncthreads()` for block

Building synchronization with atomics

- How would we create `__syncthreads()` for entire grid?
  - And why would this not be a good idea?

Building synchronization with atomics

- How would we create a critical section? I.e., one thread per block executing a particular section of code?

- How would we create a critical section per grid?
  - Why doesn't this have the same issue as `__syncthreads()` for grid?

Atomic Compare and Swap

```
int atomicCAS(int *address, int compare, int val)
{
    int old = *address;
    if  (old == compare)
      *address = val;
    return old;
}
```

Histogramming

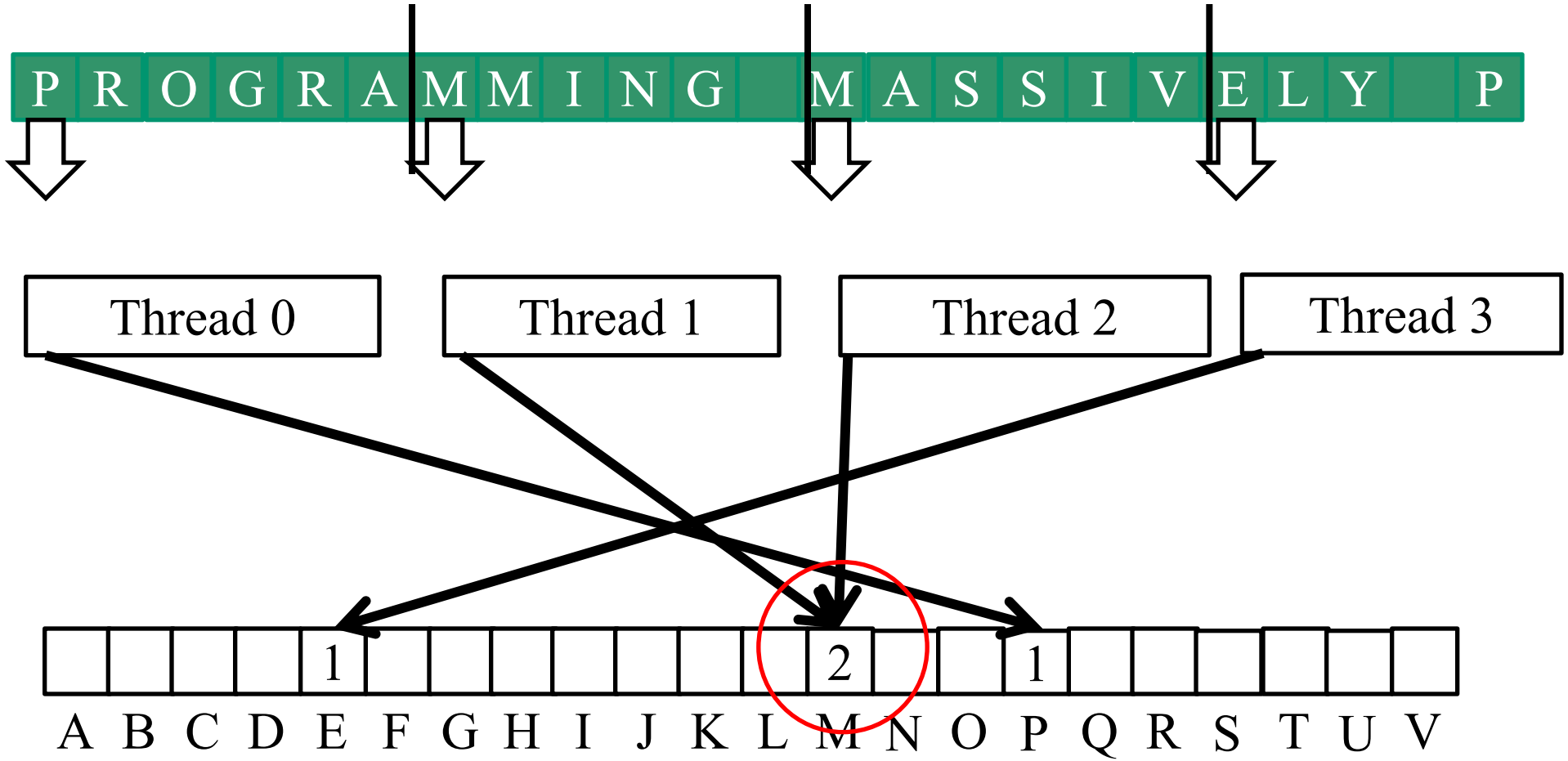
- A method for extracting notable features and patterns from large data sets
  - Feature extraction for object recognition in images
  - Fraud detection in credit card transactions
  - Correlating stellar object movements in astrophysics
  - …
- Basic histograms - for each element in the data set, use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

A Histogram Example

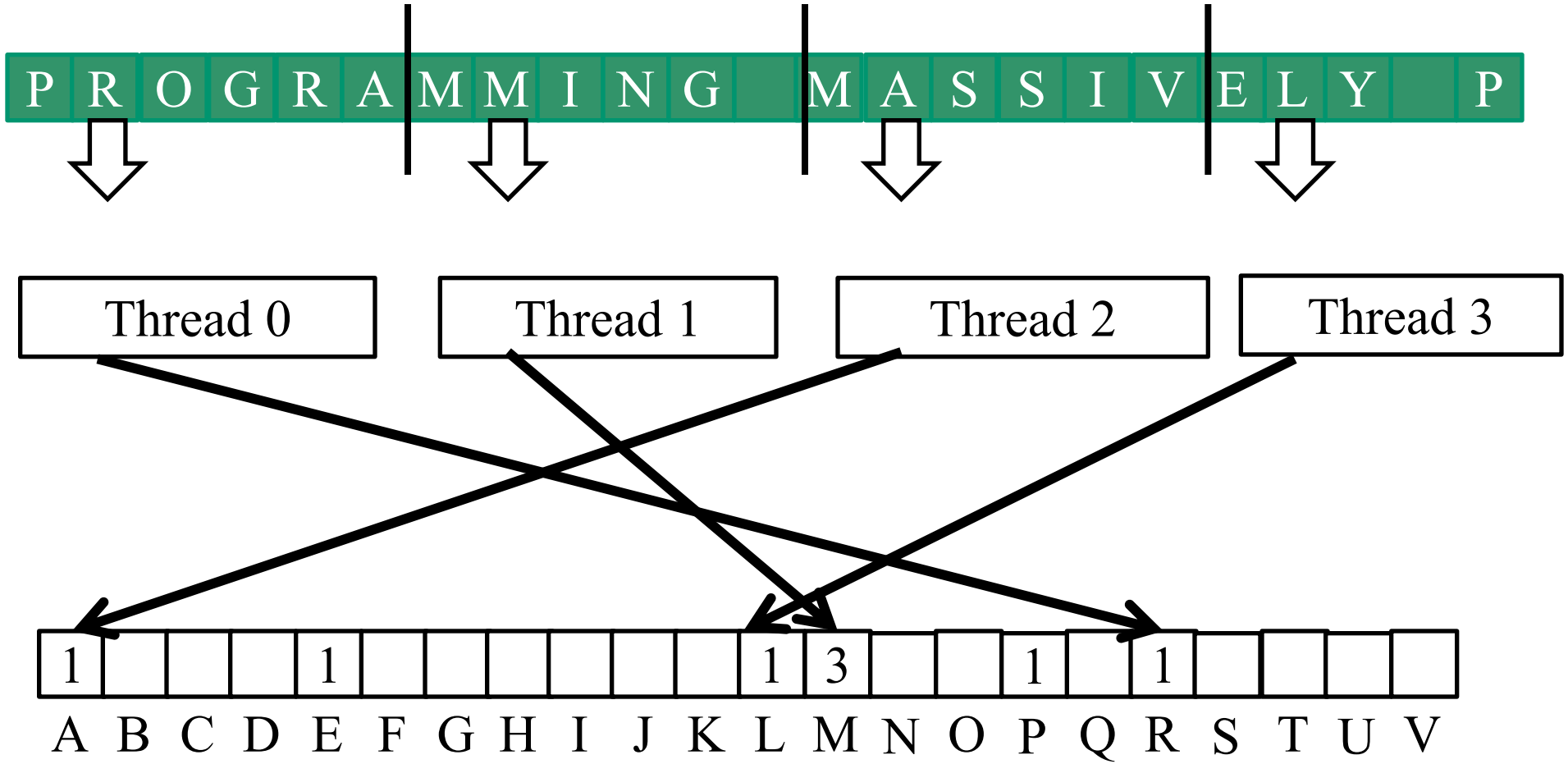
- In sentence “Programming Massively Parallel Processors” build a histogram of frequencies of each letter
- A(4), C(1), E(1), G(1), …

- How do you do this in parallel?

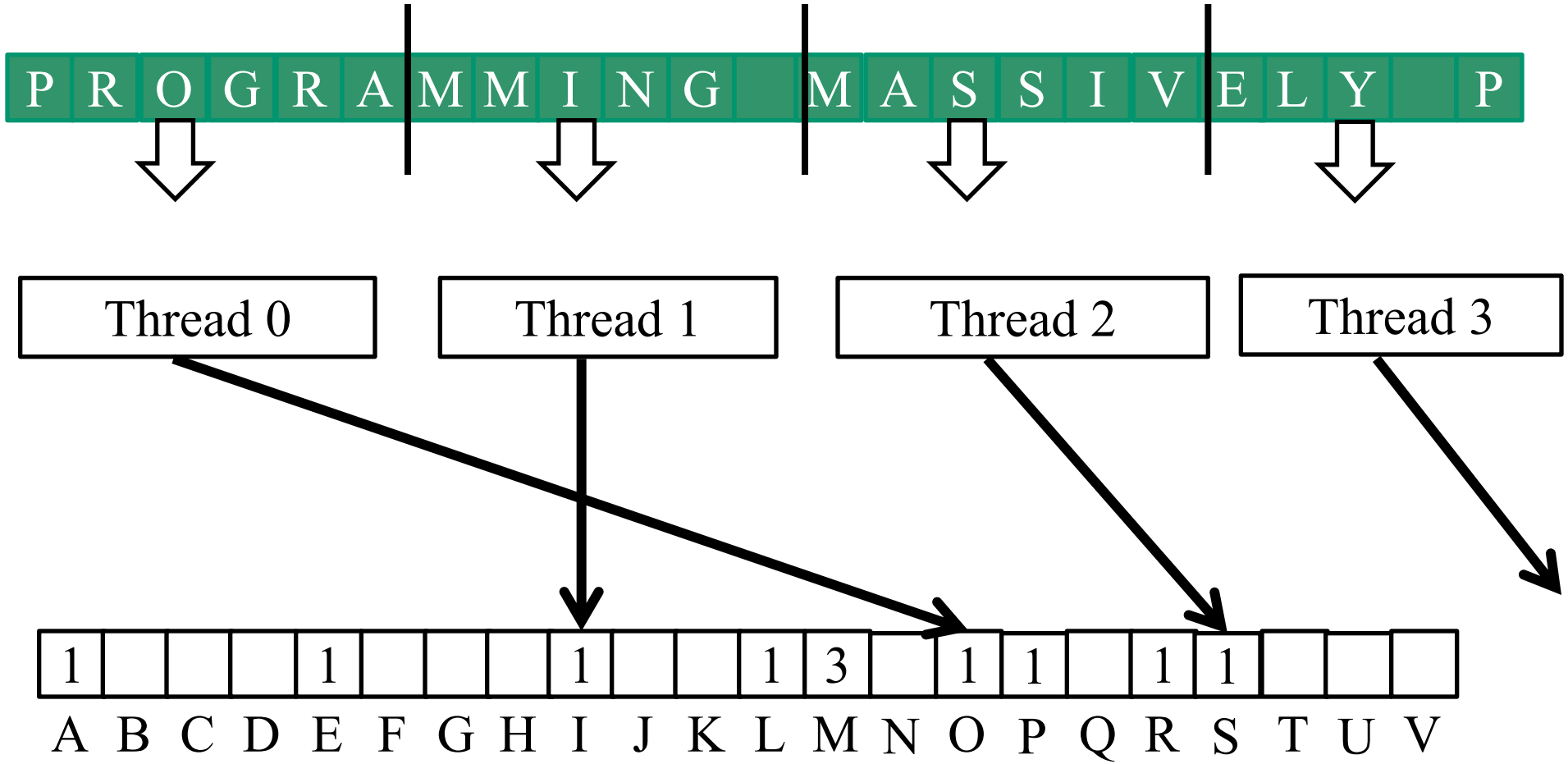
Iteration #1 – 1st letter in each section



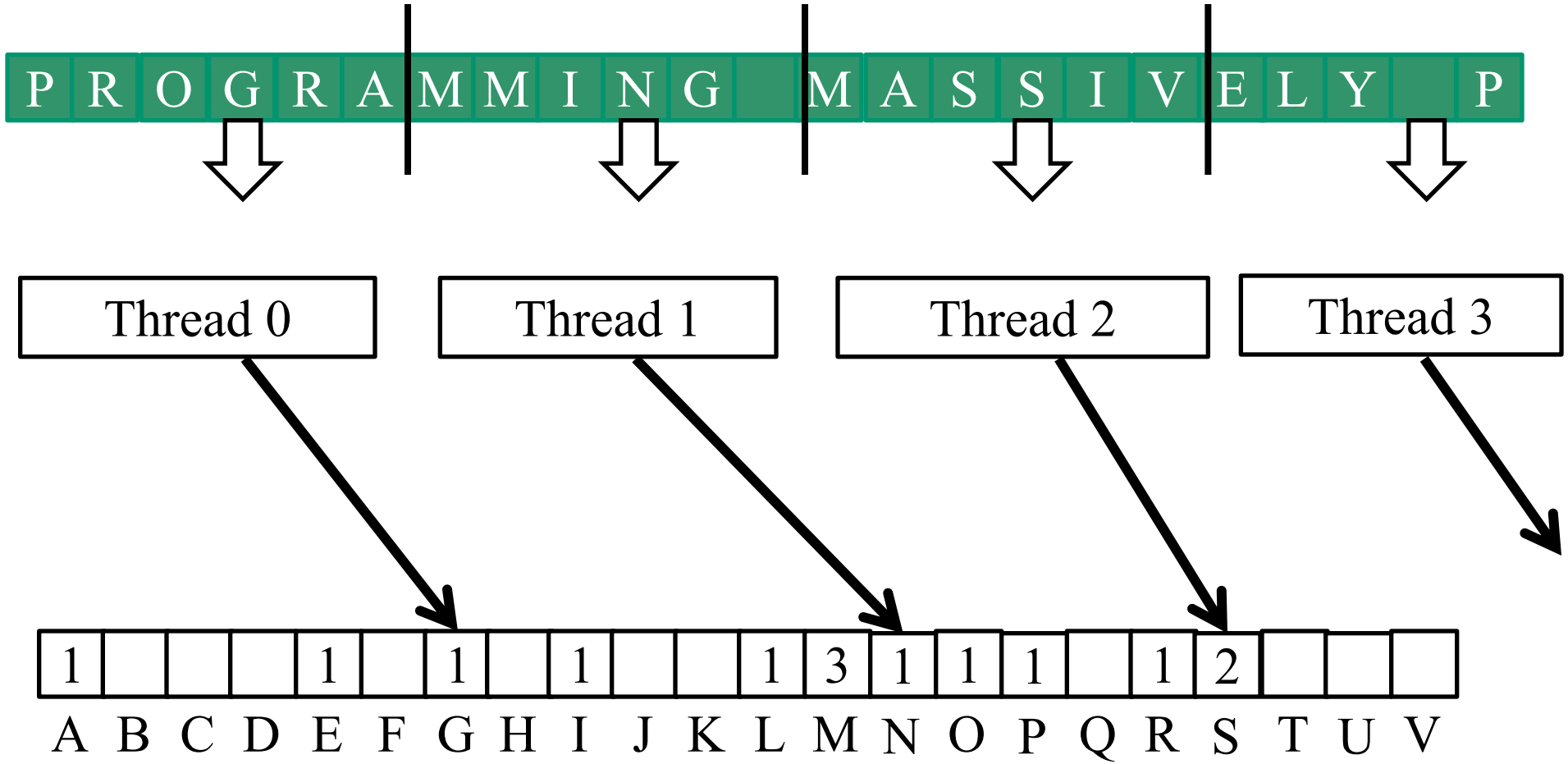
Iteration #2 – 2nd letter in each section



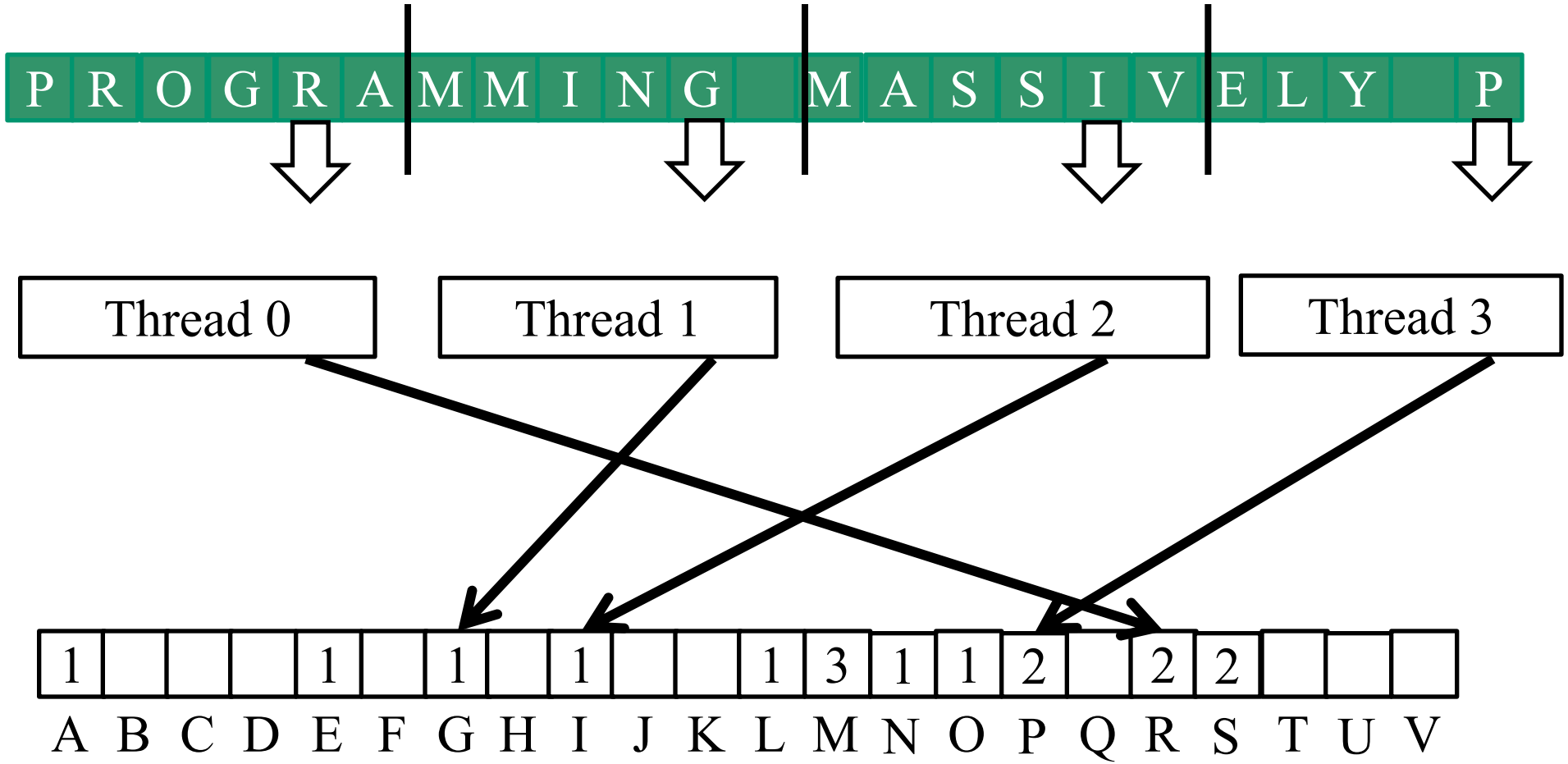
Iteration #3


P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1 1 1 1 3 1 1 1 1
A B C D E F G H I J K L M N O P Q R S T U V

Iteration #4

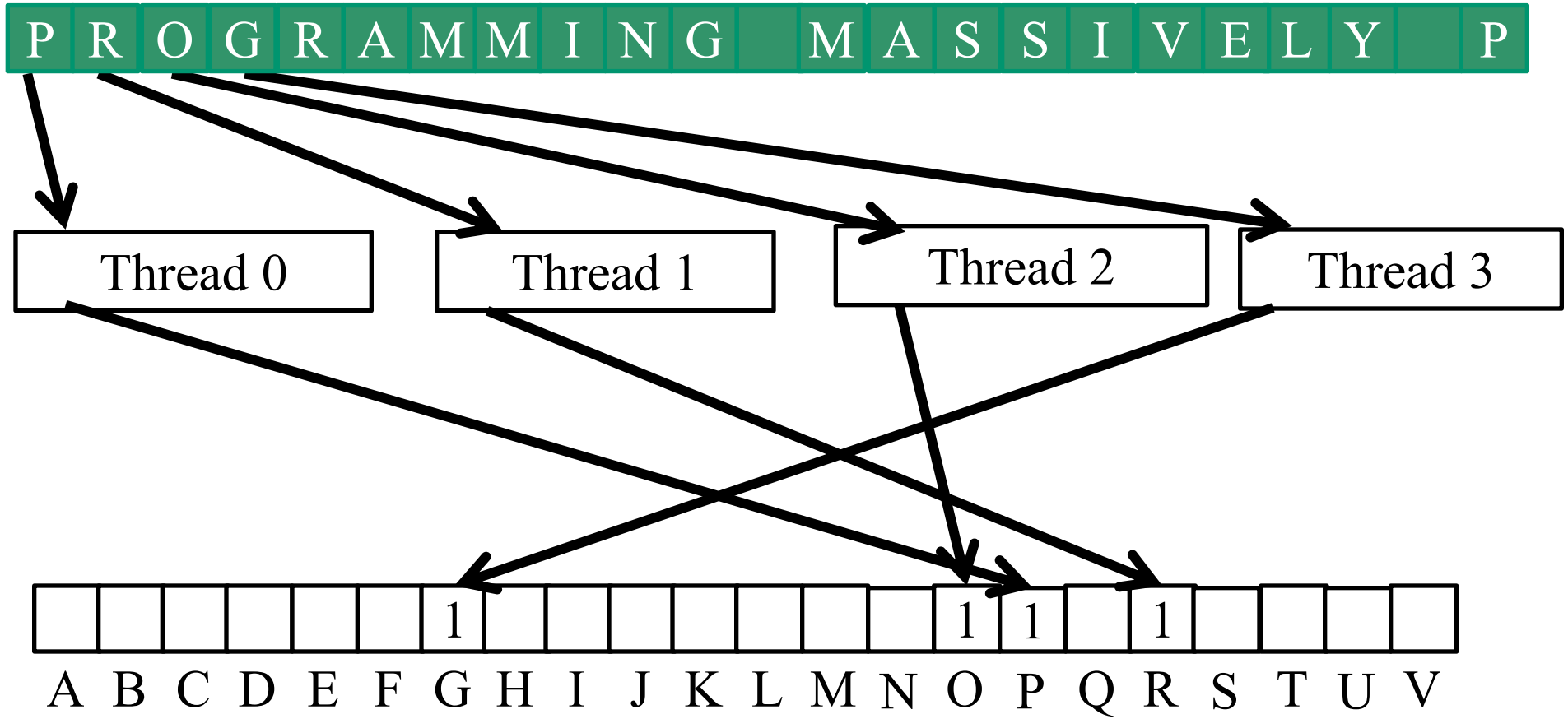

P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1 1 1 1 1 1 3 1 1 1 1 2
A B C D E F G H I J K L M N O P Q R S T U V

Iteration #5


P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1 1 1 1 1 1 3 1 1 2 2 2
A B C D E F G H I J K L M N O P Q R S T U V

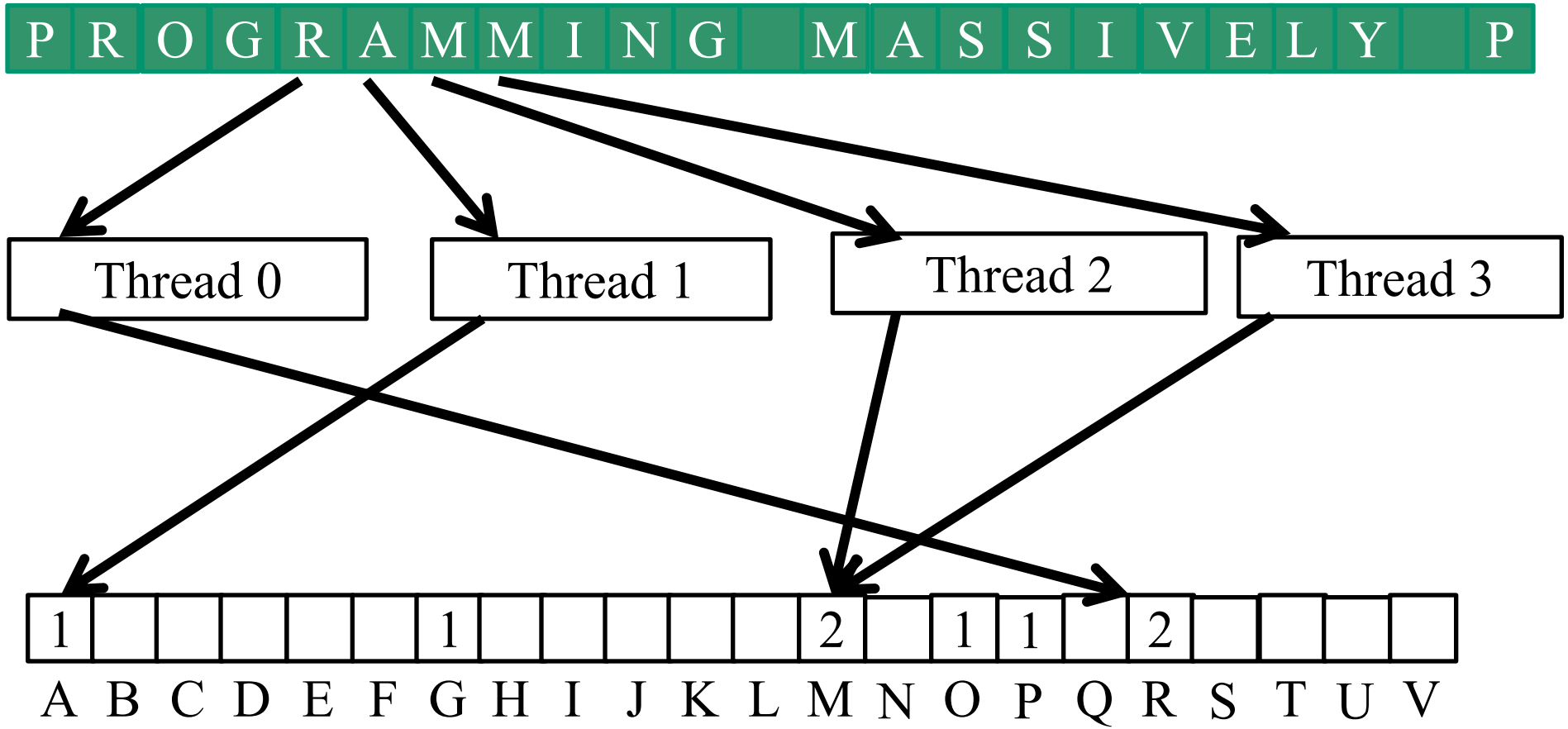
A better approach

- Coalesce the Reads
  - Assign inputs to each thread in a strided pattern
  - Adjacent threads process adjacent input letters



Iteration 2

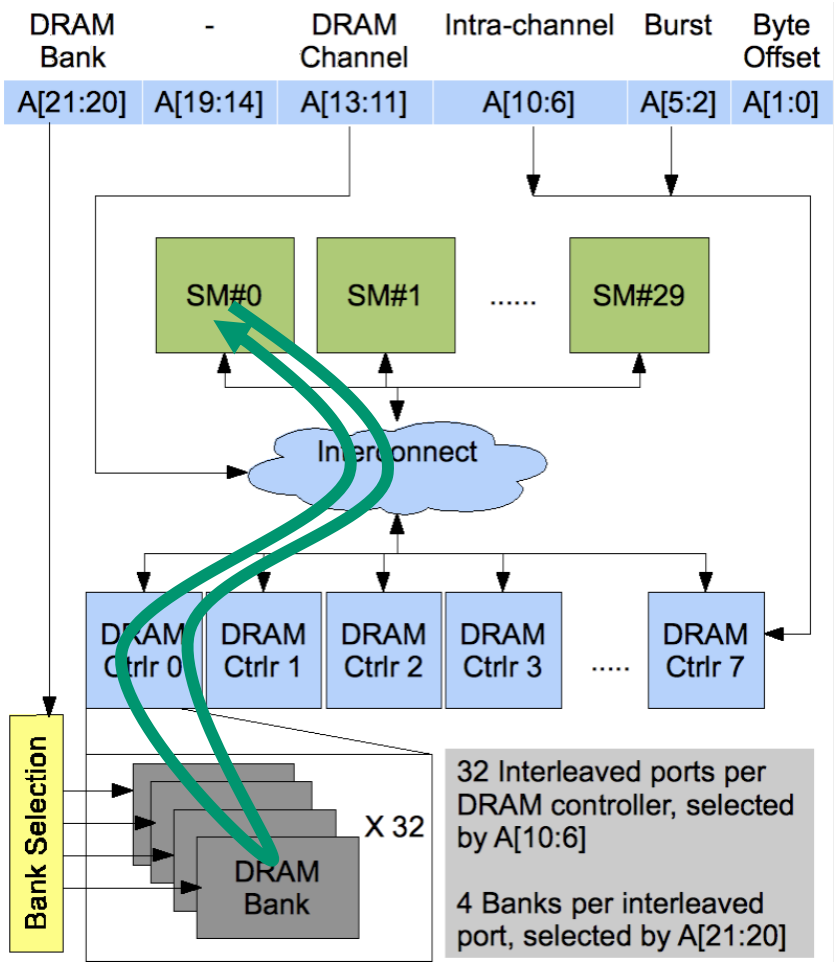
- All threads move to the next section of input



A Histogram Kernel

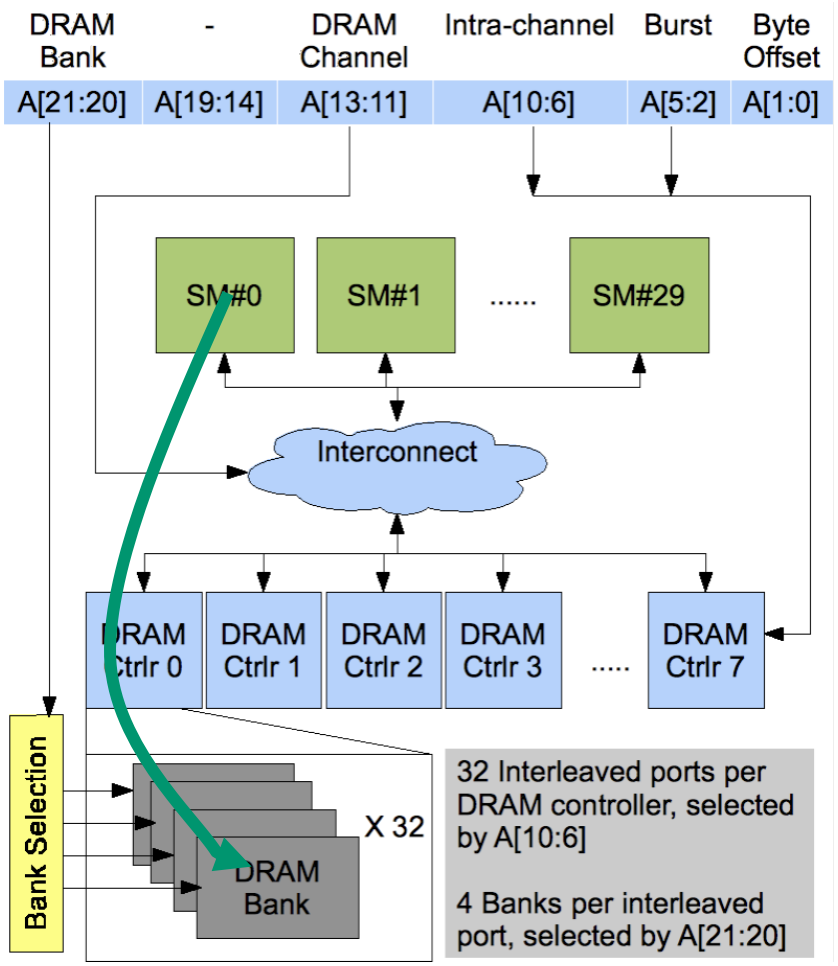
```
__global__
void histo_kernel(unsigned char *buffer,int size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

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

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

Atomic Operations on DRAM



- An atomic operation starts with a read, with a latency of a few hundred cycles

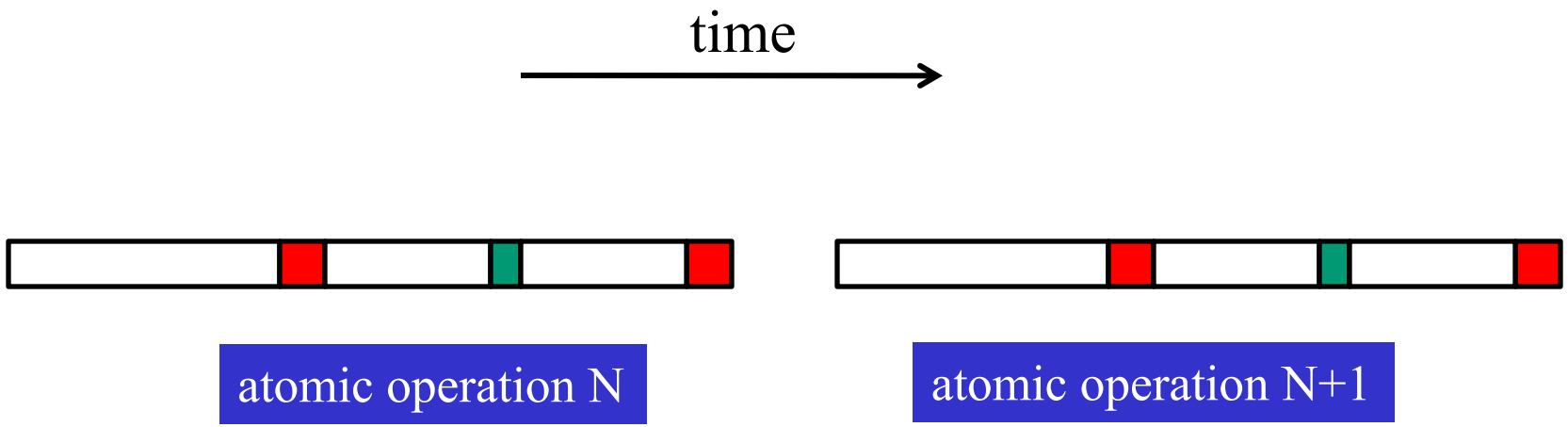
Atomic Operations on DRAM



- An atomic operation starts with a read, with a latency of a few hundred cycles
- The atomic operation ends with a write, with a latency of a few hundred cycles
- During this whole time, no one else can access the location

Atomic Operations on DRAM

- Each Load-Modify-Store has two full memory access delays
  - All atomic operations on the same variable (RAM location) are serialized

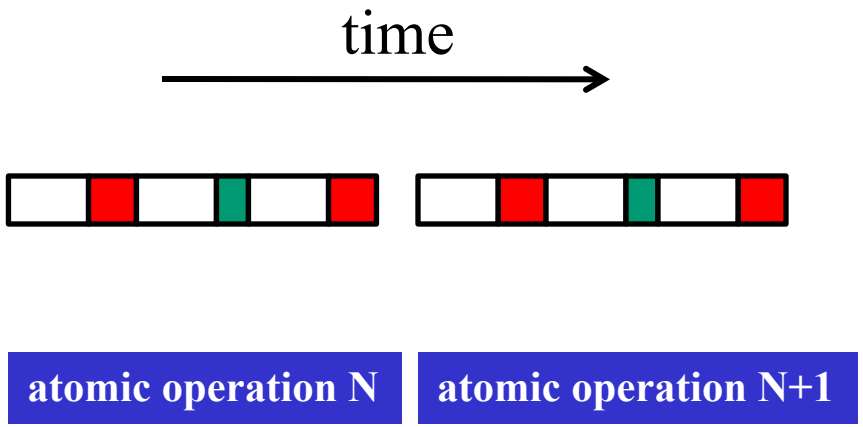


Latency determines throughput of atomic operations

- Throughput of an atomic operation is the rate at which the application can execute an atomic operation on a particular location.
- The rate is limited by the total latency of the read-modify-write sequence, typically more than 1000 cycles for global memory (DRAM) locations.
- This means that if many threads attempt to do atomic operation on the same location (contention), the memory bandwidth is reduced to < 1/1000!

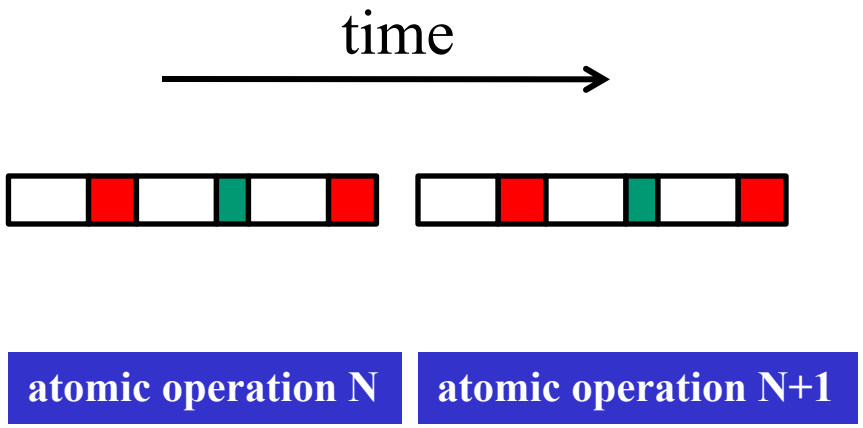
Hardware Improvements

- Atomic operations on L2 cache
  - medium latency, but still serialized
  - Global to all blocks
  - “Free improvement” on Global Memory atomics



Hardware Improvements

- Atomic operations on Shared Memory
  - Very short latency, but still serialized
  - Private to each thread block
  - Need algorithm work by programmers (more later)



ANY MORE QUESTIONS
READ CHAPTER 9