

EE5351

Applied Parallel Programming

Lecture 14, 16 Atomic Operations and Histogramming

But first, some review...

- GeForce GTX 280 GPU
 - 30 Streaming Multiprocessors
 - Maximum of 1024 threads per SM
 - 16 kB shared memory per SM
 - 16k registers per SM

```
$ nvcc --ptxas-options="-v"  
matrixmul_kernel.cu  
ptxas info: Compiling entry function  
'_Z15MatrixMulKernel6MatrixS_S_' for 'sm_10'  
ptxas info: Used 12 registers, 2120+16 bytes  
smem, 8 bytes cmem[1]
```

- 4 Resource Limitations
 - Thread contexts: $1024 / 256 = 4$ blocks per SM
 - Shared memory: $16384 / 2136 = 7$ blocks per SM
 - Registers: $16384 \text{ registers} / (12 * 256) \text{ registers per block} = 5$ blocks per SM
 - Blocks: Max of 8 blocks per SM
- Limited to 4 blocks per SM by thread context limitation
- Maximum threads simultaneously scheduled:
 $4 \text{ blocks/SM} * 256 \text{ threads/block} * 30 \text{ SMs} = 30720 \text{ threads}$
- In general, answer is implementation dependent

Objective

- To understand **atomic** operations
 - **Read-modify-write** in parallel computation
 - 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 Collaboration Pattern

- Multiple bank tellers count the total cash assets in the safe
- Each grab a pile and count
- Have a central display of the running total
- Whenever someone finishes counting a pile, add the subtotal of the pile to the running total
- A bad outcome
 - Some of the piles were not accounted for

A Common Parallel Coordination Pattern

- Multiple customer service agents serving customers
- Each customer gets a number
- A central display shows the number of the next customer who will be served
- When an agent becomes available, he/she calls the number and adds 1 to the display
- **Bad outcomes**
 - Multiple customers get the same number
 - Multiple agents serve the same number
 - Some number doesn't get served

A Common Arbitration Pattern

- Multiple customers booking air tickets
- Each:
 - Brings up a flight seat map
 - Decides on a seat
 - Updates the seat map, marking the seat as taken
- A bad outcome
 - Multiple passengers ended up booking the same seat

Atomic Operations

<code>thread1:</code>	$\text{Old} \leftarrow \text{Mem}[x]$	<code>thread2:</code>	$\text{Old} \leftarrow \text{Mem}[x]$
	$\text{New} \leftarrow \text{Old} + 1$		$\text{New} \leftarrow \text{Old} + 1$
	$\text{Mem}[x] \leftarrow \text{New}$		$\text{Mem}[x] \leftarrow \text{New}$

If $\text{Mem}[x]$ was initially 0, what would the value of $\text{Mem}[x]$ 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, you should use **atomic operations**.

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

Race condition = when computed values depend on ordering

Atomic Operations – To Ensure Good Outcomes

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

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

Or

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

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

Without Atomic Operations

Mem[x] initialized to 0

thread1: Old \leftarrow Mem[x]

New \leftarrow Old + 1

Mem[x] \leftarrow New

thread2: Old \leftarrow Mem[x]

New \leftarrow Old + 1

Mem[x] \leftarrow New

- Both threads read Mem[x] = 0
- Mem[x] becomes 1

Atomic Operations in General

- Performed by a **single ISA instruction** on a **memory location *address***
 - **Read** the old value at the location, **calculate** a new value, and **write** the new value to the same location
- The hardware ensures that **no other threads can access the memory location until the atomic operation is complete**
 - Any other thread that accesses the location will typically be held in a **queue** until its turn
 - All threads perform the atomic operation **serially**
 - Atomic operations do not give any guarantees on the **order** in which requests are processed

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 details

- Atomic Add

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

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

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
(compute capability > 2.0)

float atomicAdd(float address, float val);*

Custom Atomic Operations

- **Any atomic operation** can be implemented based on **atomicCAS()** (Compare And Swap)
- `int atomicCAS(int* address, int compare, int val)`
 - Reads 32-bit or 64-bit word `old` located at `address` in global or shared memory, computes `(old == compare ? val : old)`, and stores result back to `address`, returns `old`

Custom Atomic Operations

```
__device__ double atomicAdd(double* address,
                             double val){
    unsigned long long int* address_as_ull =
        (unsigned long long int*)address;
    unsigned long long int old =
        *address_as_ull, assumed;
    do{
        assumed = old;          // READ
        old = atomicCAS(address_as_ull, assumed,
            val + assumed);      // MODIFY + WRITE
    } while (assumed != old);
    return old;
}
```

Histogramming

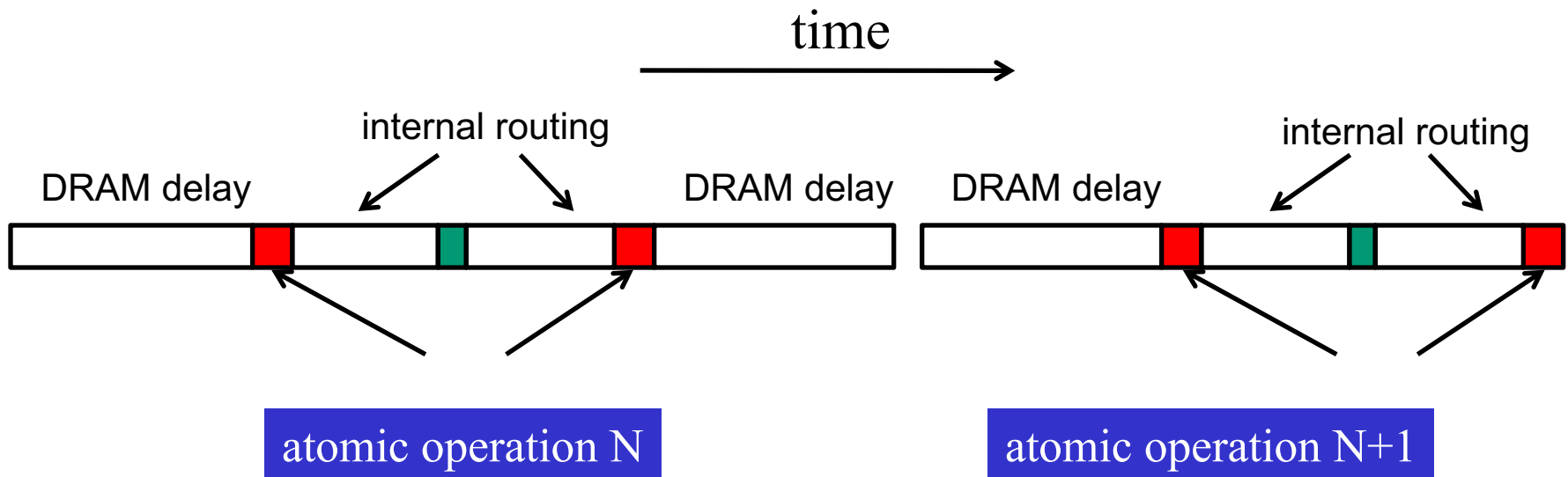
- A method for extracting notable features and patterns from large data sets
 - **Feature extraction** for object recognition in images
 - **Fraud detection** in credit card transactions
 - **Correlating** heavenly object movements in astrophysics
 - ...
- Basic histograms - for each element in the data 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(3), G(2), ...
- How do you do this in parallel?

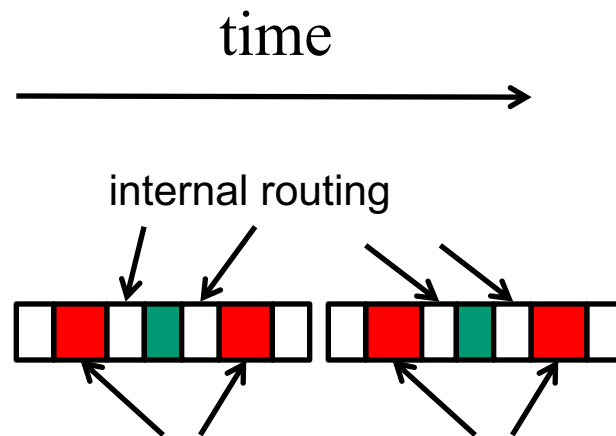
Atomic Operations on DRAM

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



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)

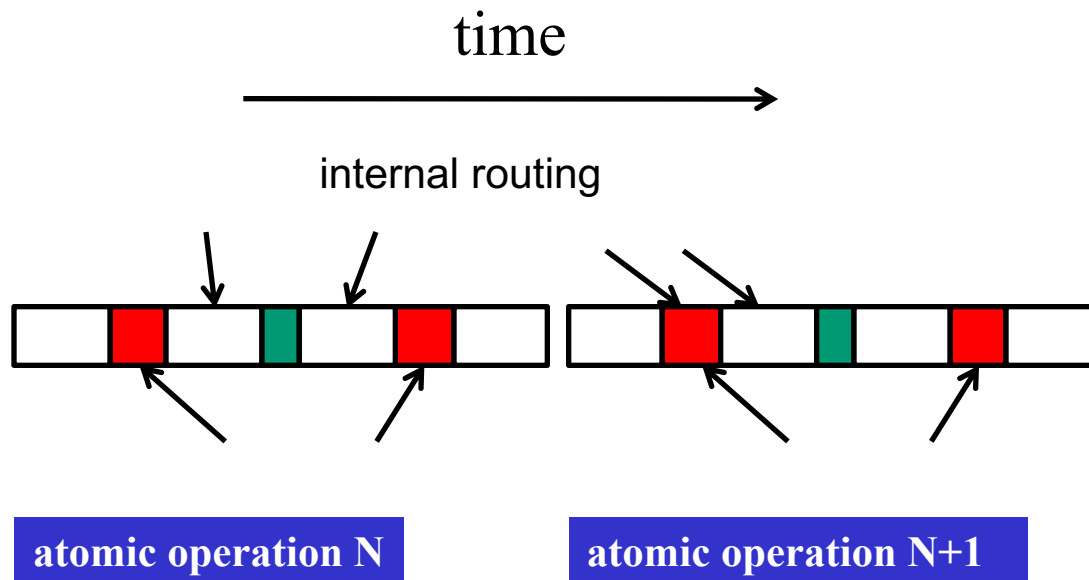


atomic operation N

atomic operation N+1

Hardware Improvements (cont.)

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



Objective

- To learn practical **histogram** programming techniques
 - Basic histogram algorithm using **atomic operations**
 - **Privatization**

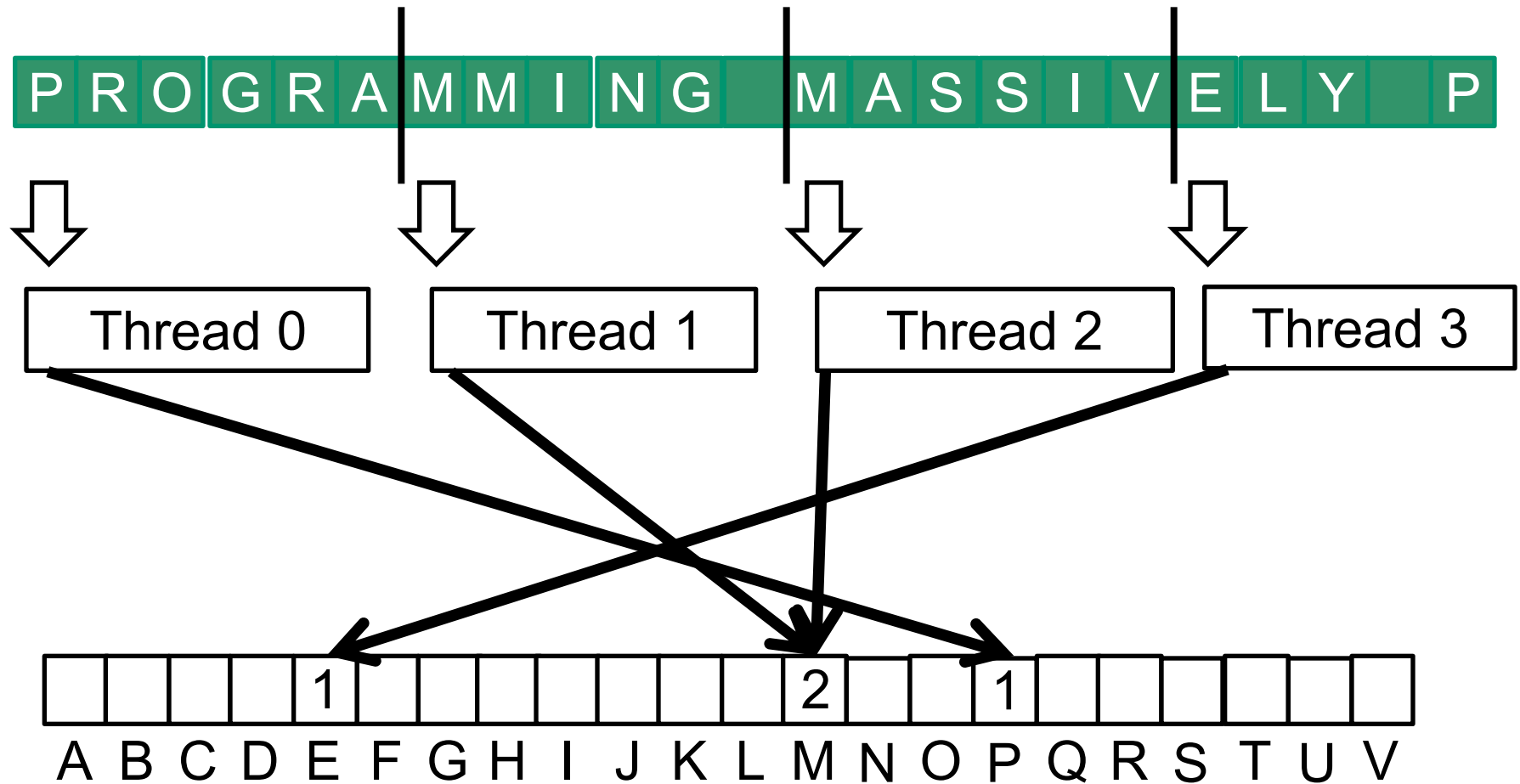
Atomic Operations (Review)

- Performed by a single ISA instruction on a memory location *address*
 - Read the old value at the location, calculate a new value, and write the new value to the same location
- The hardware ensures that no other threads can access the location until the atomic operation is complete
 - Any other thread that accesses the location will be held in a queue until its turn
 - All threads perform the atomic operation serially
 - Atomic operations do not give any guarantees on the order in which requests are processed

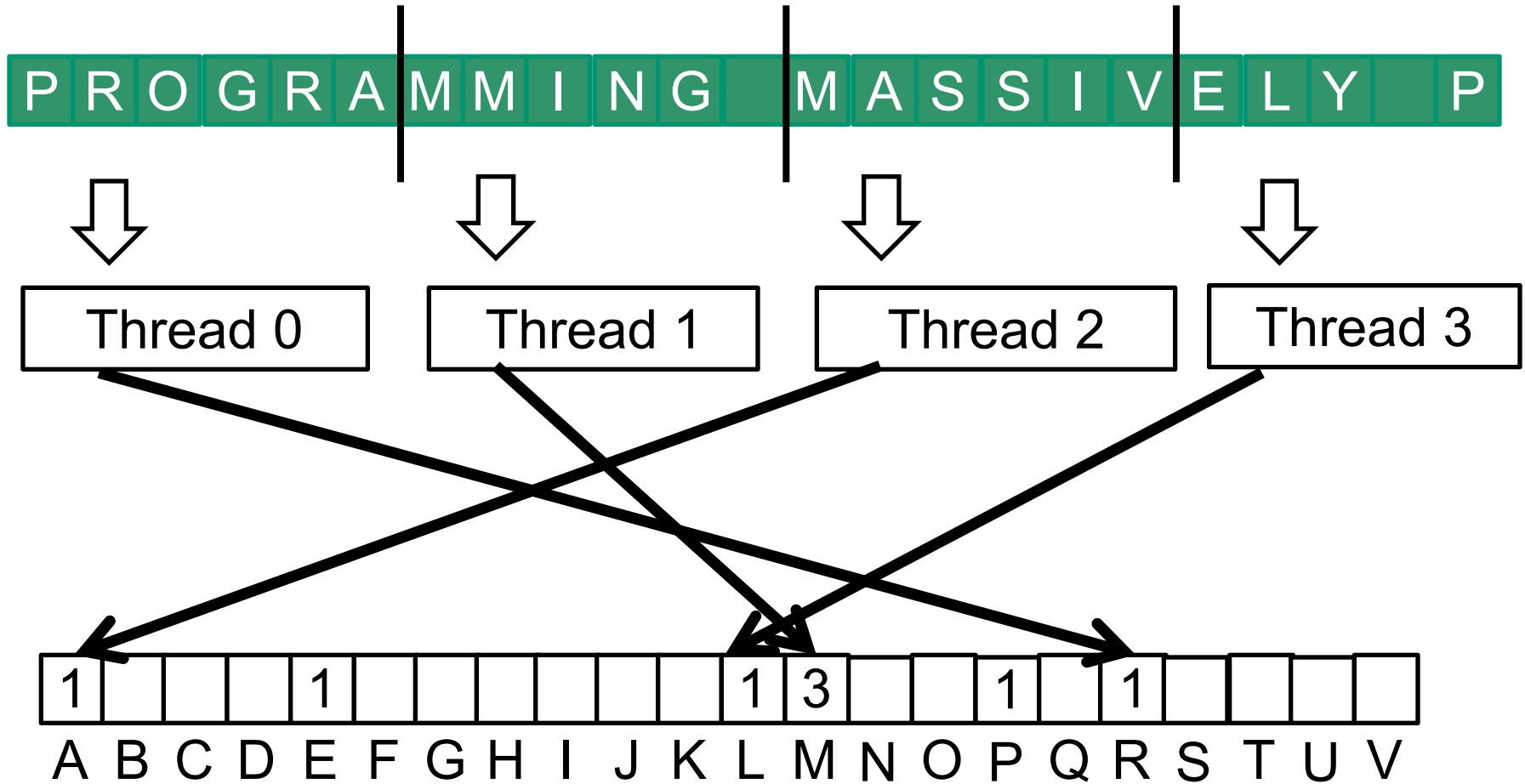
A Histogram Example (Review)

- In sentence “Programming Massively Parallel Processors” build a histogram of frequencies of each letter
→ A(4), B(0), C(1), D(0), E(3), F(0), G(2), ...
- **How do you do this in parallel?**
 - Have each thread take a section of the input
 - Process each input letter, using atomic operations to build the histogram

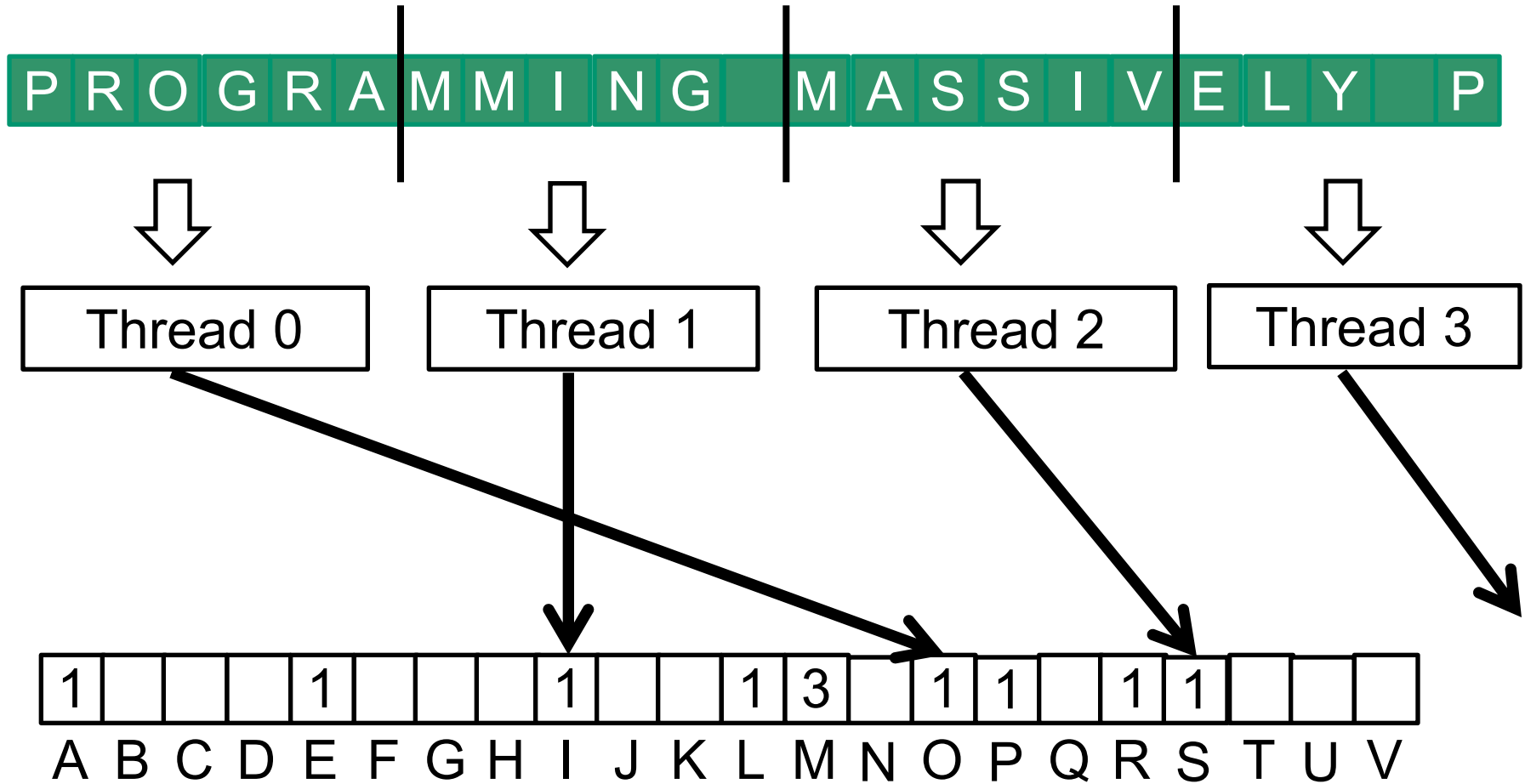
Iteration #1 – 1st letter in each section



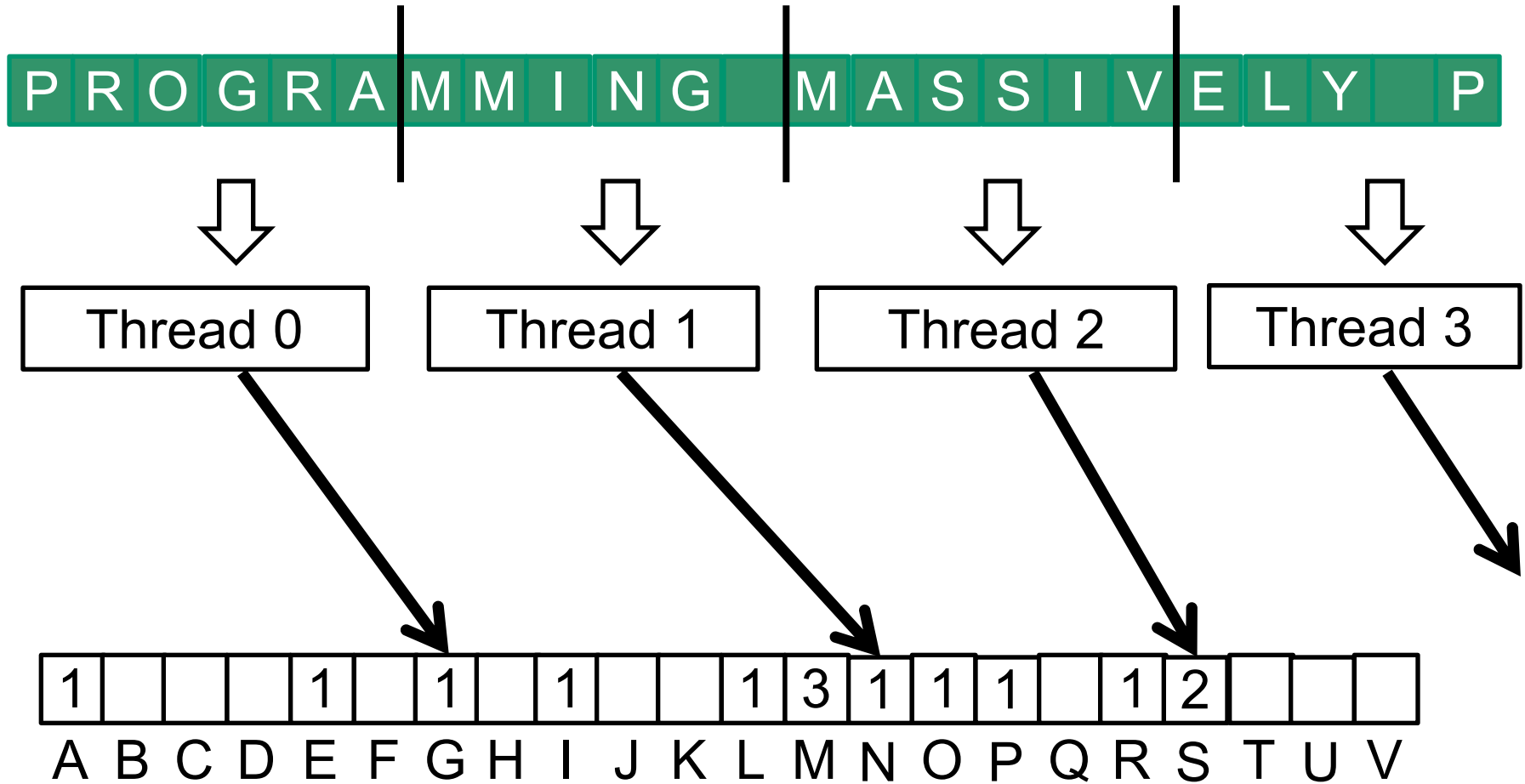
Iteration #2



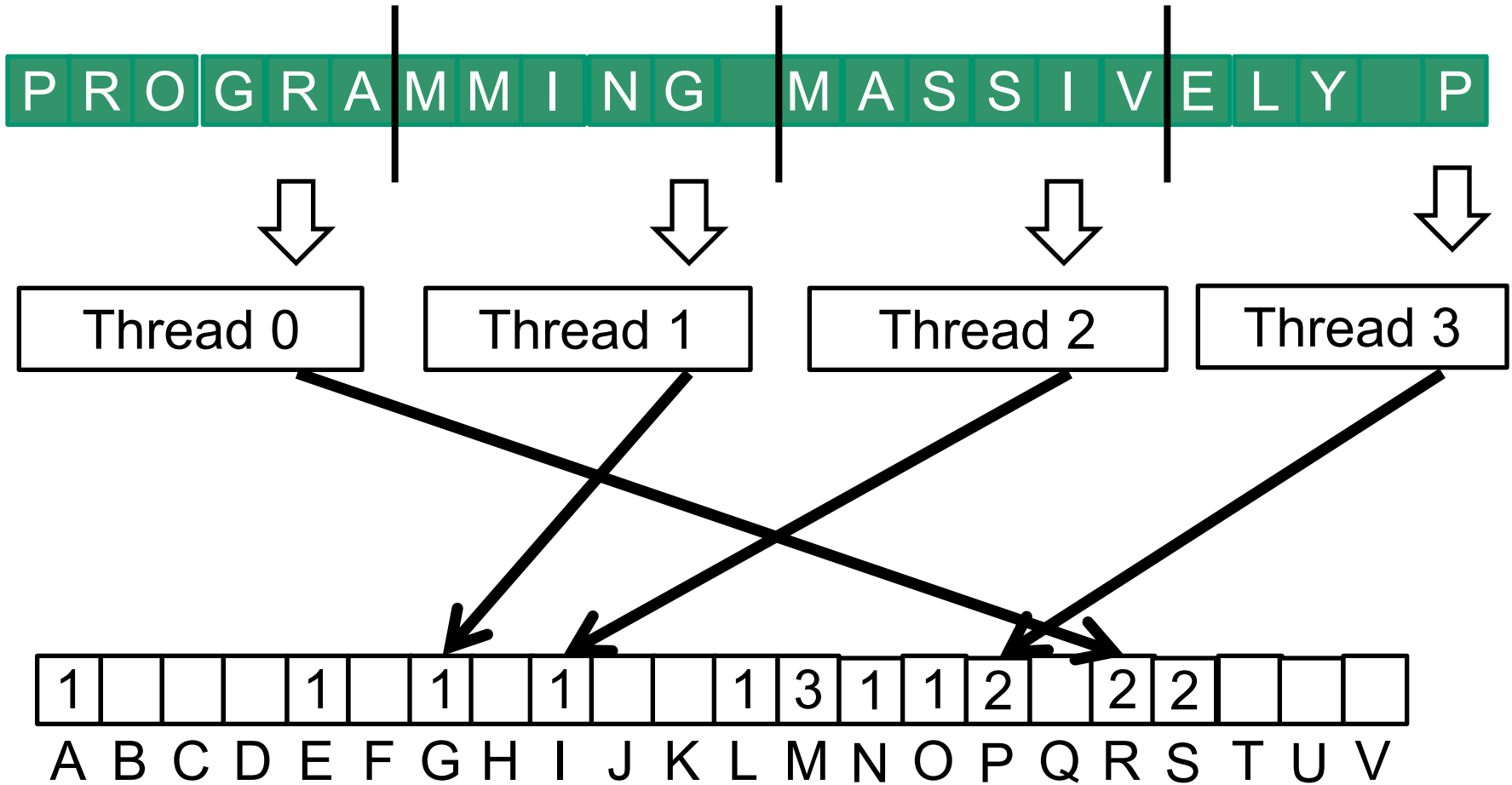
Iteration #3




Iteration #4



Iteration #5

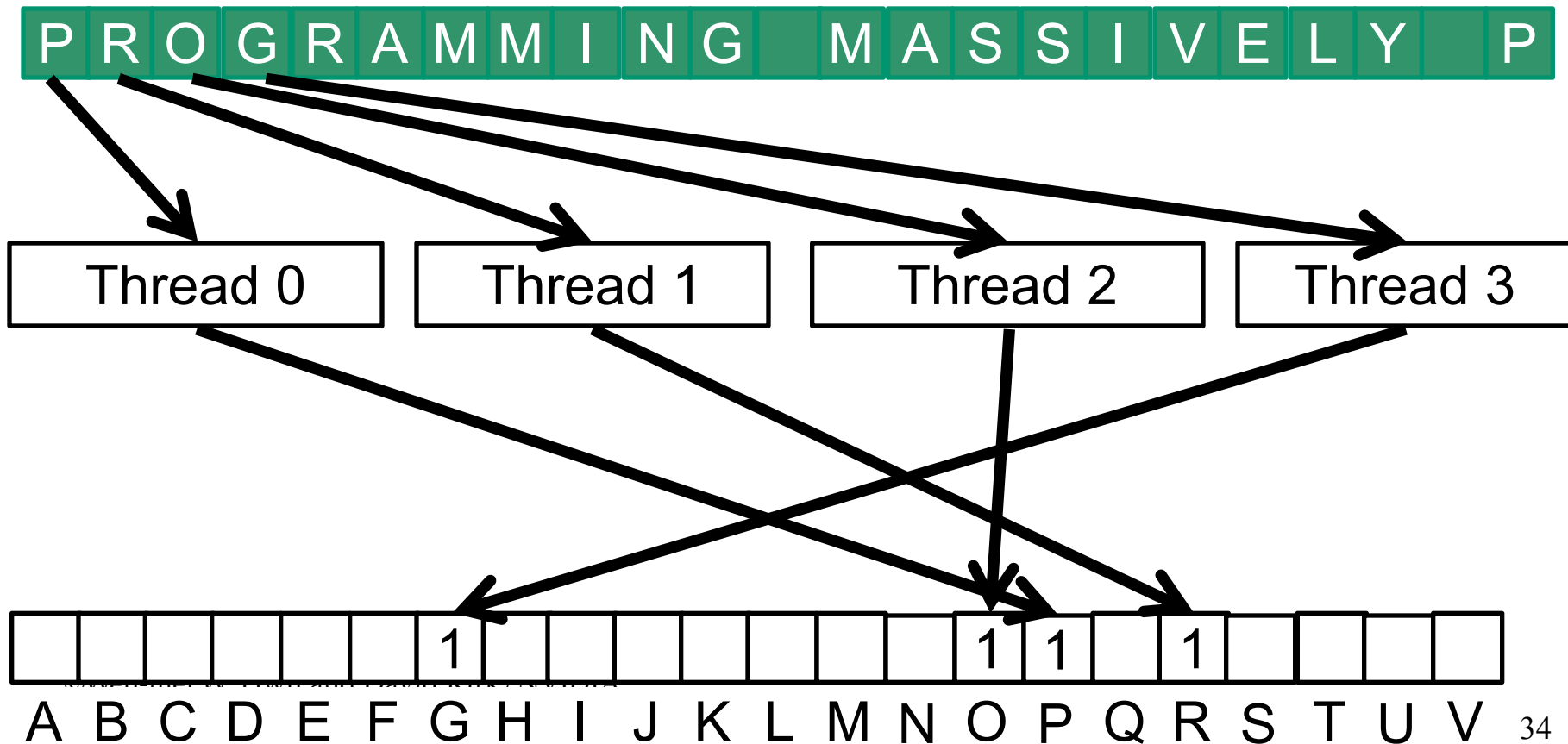


Two vertical bars, one red and one yellow, are positioned on the left side of the slide.

What is **wrong
with the algorithm?**

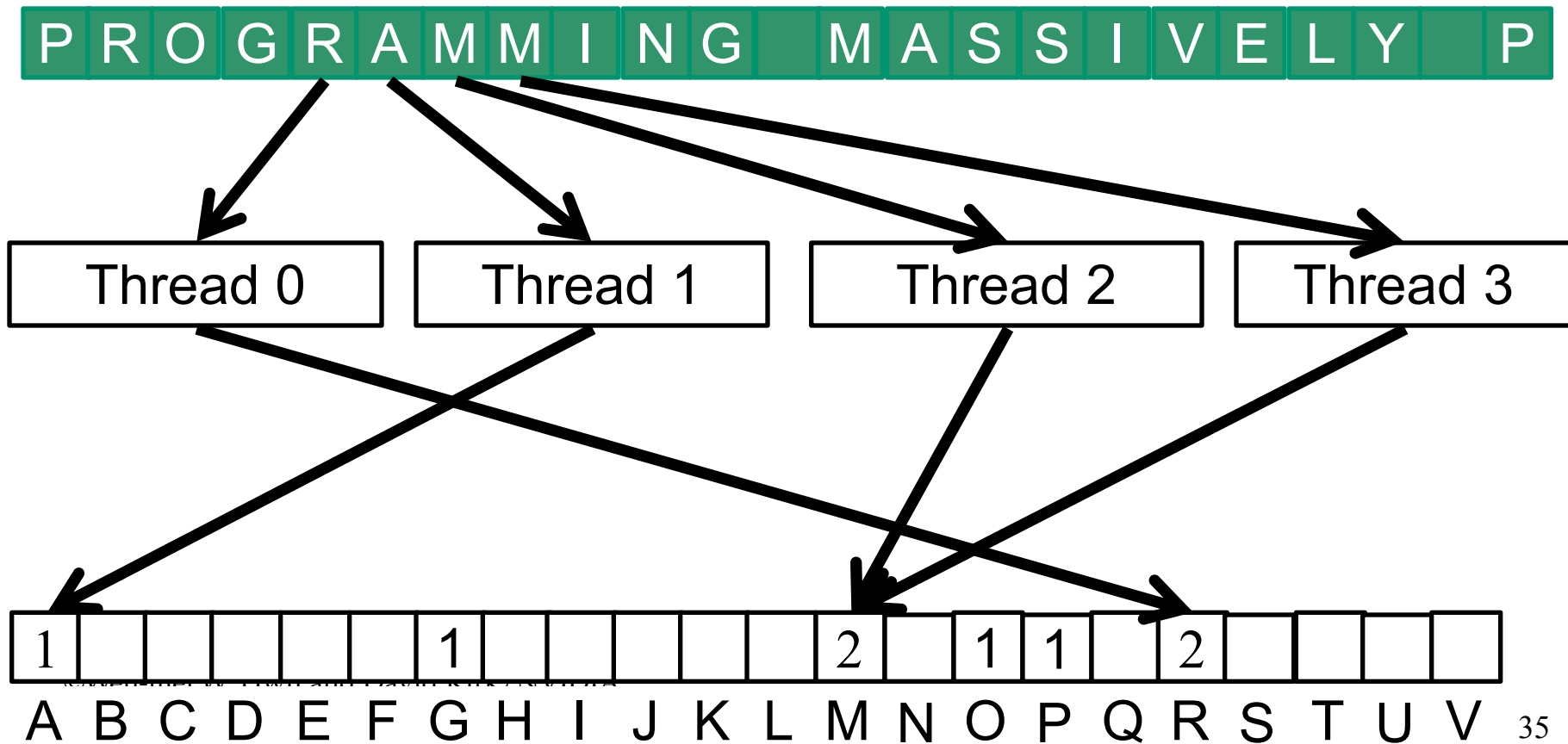
What is wrong with the algorithm?

- Reads from the input array are **not coalesced**
 - Instead, assign inputs to each thread in a strided pattern
 - Adjacent threads should process adjacent input letters



Iteration 2

- All threads move to the next section of input



A Histogram Kernel

- The kernel receives a pointer to the input buffer
- Each **thread** processes input in a **strided pattern**
 - but **warps** process input in a **coalesced 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;
```

More on the Histogram Kernel

```
// All threads process blockDim.x * gridDim.x
```

```
// consecutive elements
```

```
while (i < size) {
```

```
    atomicAdd( &(histo[buffer[i]]), 1);
```

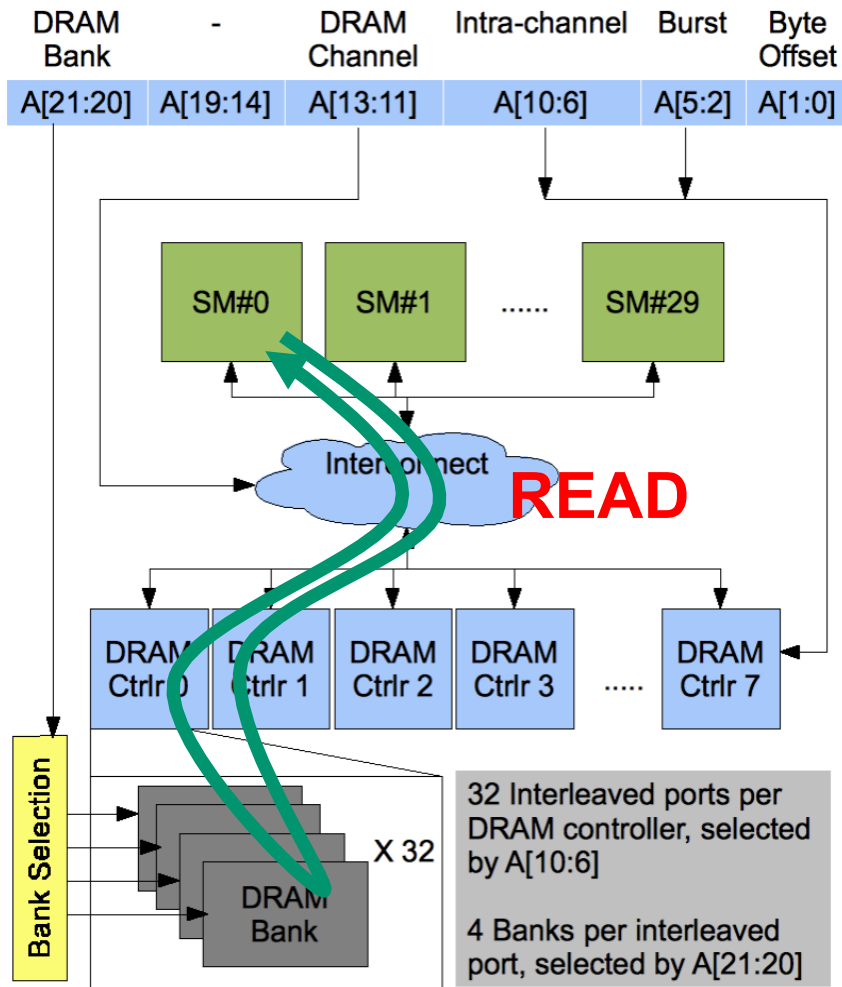
```
    i += stride;
```

```
}
```

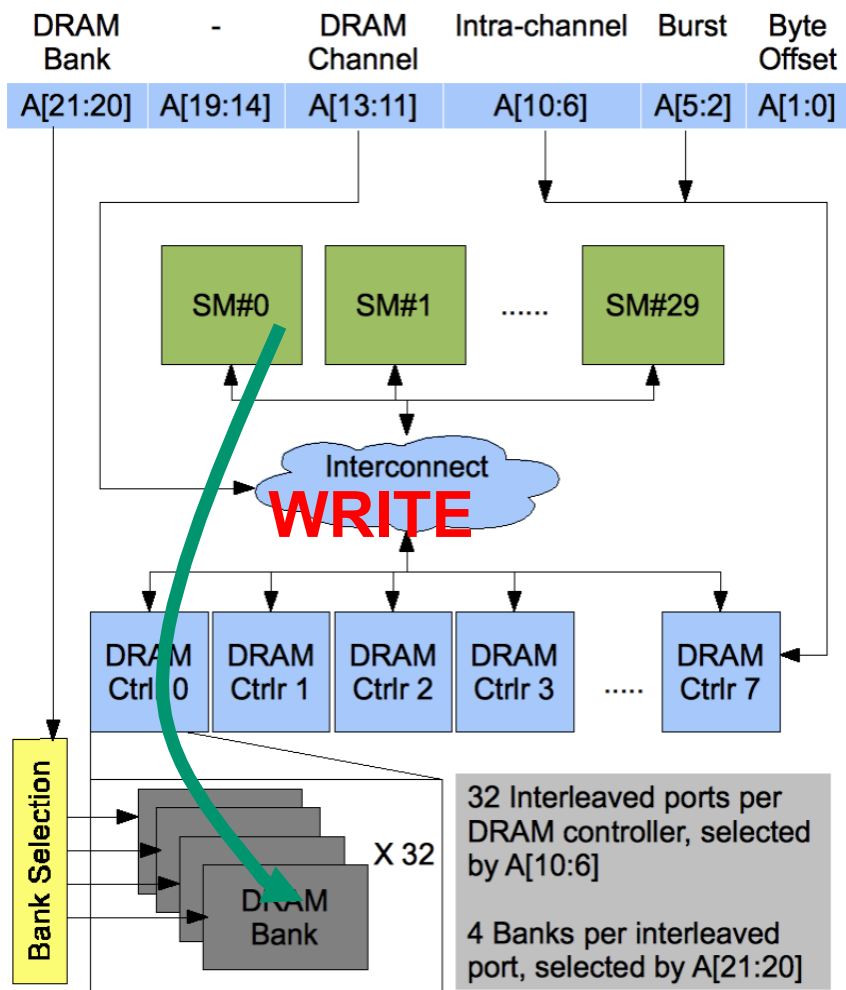
```
}
```

Atomic Operations on DRAM

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



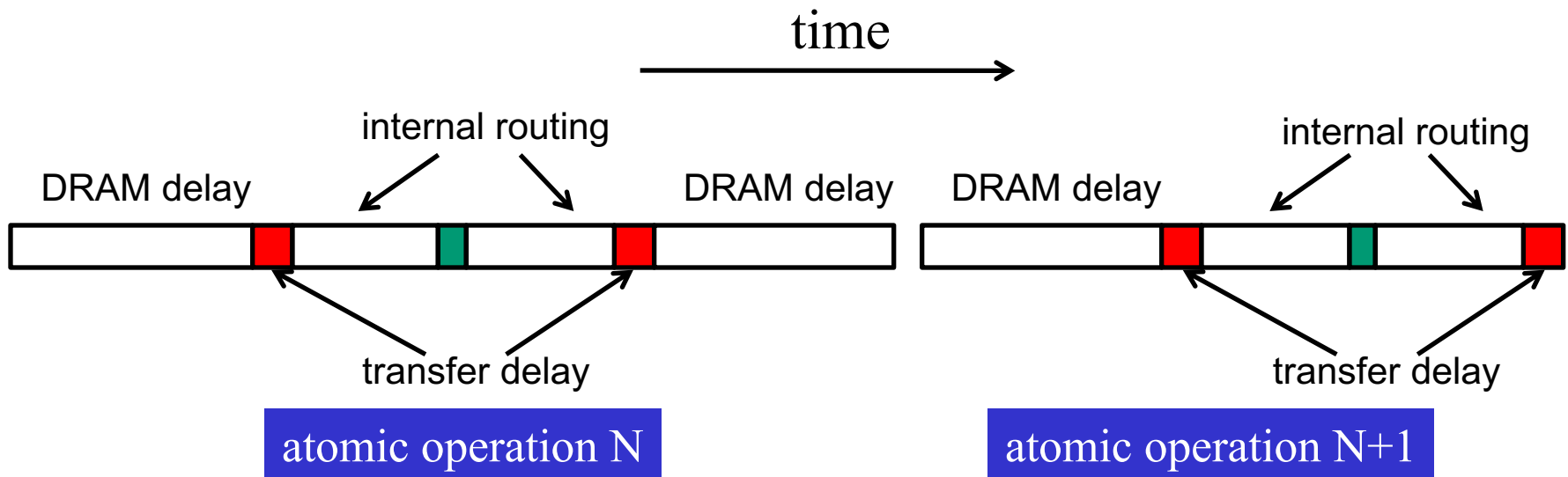
Atomic Operations on DRAM



- An atomic operation starts with a read, having a latency of a **few hundred cycles**
- The atomic operation ends with a write, having 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

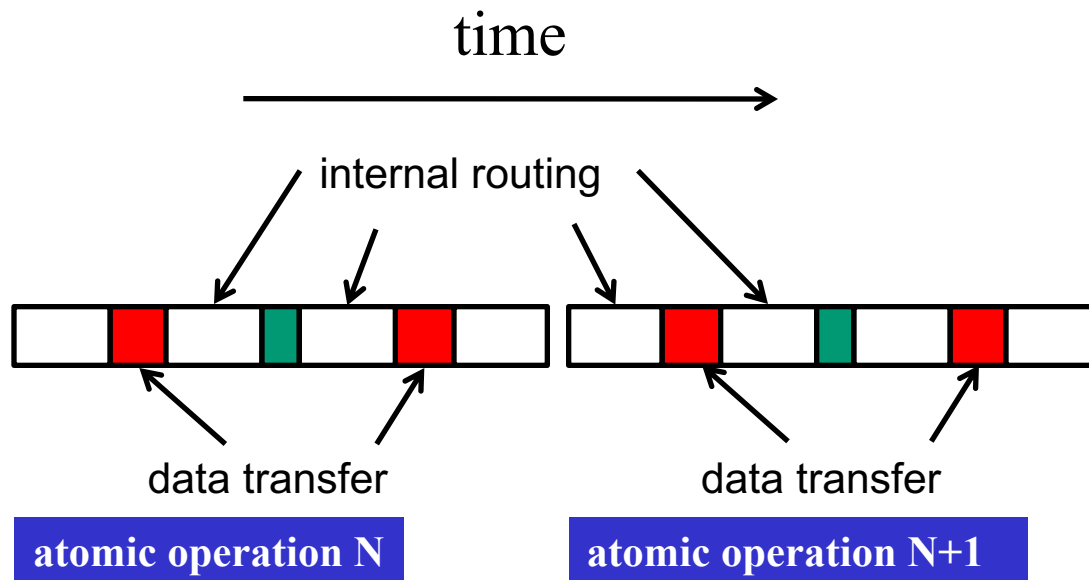
- **Throughput** of an atomic operation is the rate at which the application can execute an atomic operation on a **particular location**
- The rate is limited by the total latency of the read-modify-write sequence, typically **more than 1000 cycles for global memory** (DRAM) locations
- This means that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

You may have a similar experience in a supermarket checkout

- Some customers realize that they missed an item after they started to check out
- They run to the aisle and get the item while the line waits
 - The rate of checkout is reduced due to the long latency of running to the aisle and back.
- Imagine a store where **every customer starts the check out before they even fetch any of the items**
 - The rate of the checkout will be:
 $1 / (\text{entire shopping time of each customer})$

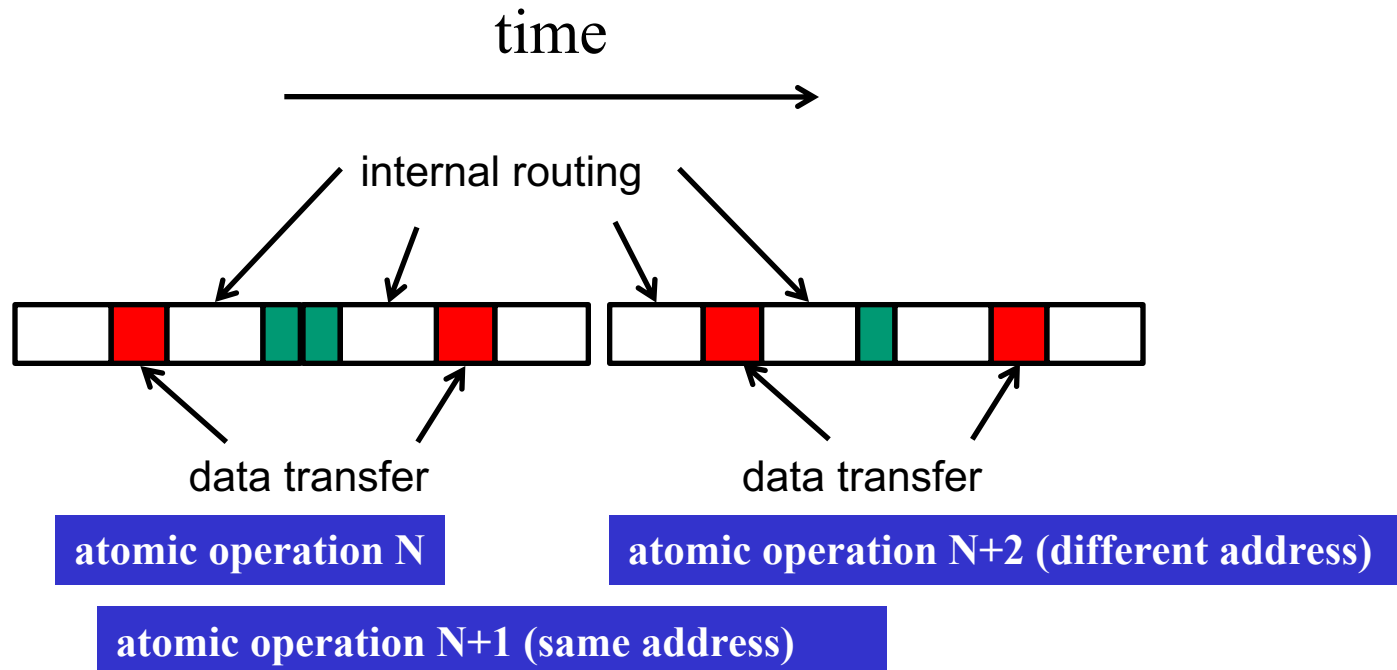
Hardware Improvements: Fermi

- Atomic operations on **Fermi L2 cache**
 - medium latency, but still serialized
 - Global to all blocks
 - “Free improvement” on Global Memory atomics if the data is in the L2 cache



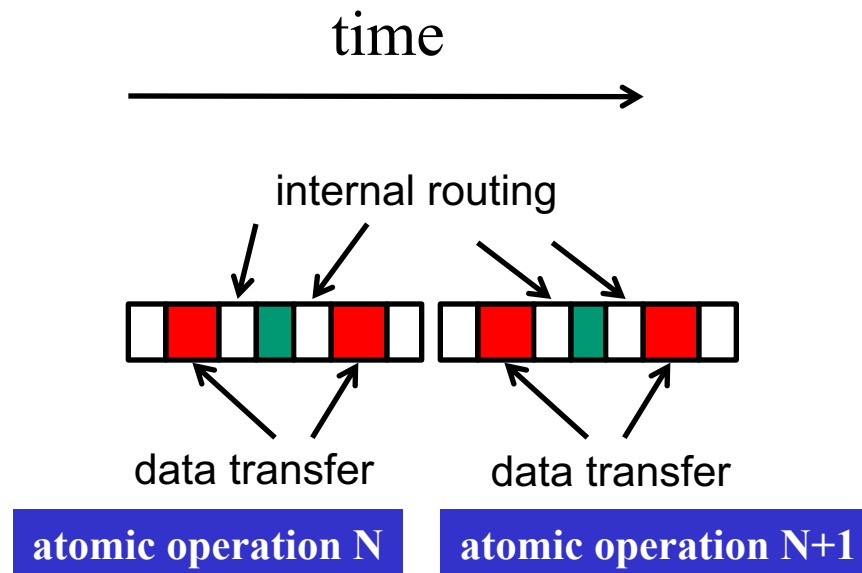
Hardware Improvements: Kepler

- Atomic operations on **Kepler L2 cache**
 - Much shorter latency for repeated updates to the same address, by avoiding unnecessary re-transfers and re-routings (9x throughput improvement)
 - Still slower for a group of different addresses



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)



Atomics in Shared Memory 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[256];  
    if (threadIdx.x < 256) histo_private[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) {  
    atomicAdd( &(amp;private_histo[buffer[i]]), 1);  
    i += stride;  
}
```

What comes next?

Build Final Histogram

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

```
if (threadIdx.x < 256)  
    atomicAdd( &(amp;histo[threadIdx.x]),  
               private_histo[threadIdx.x] );  
}
```

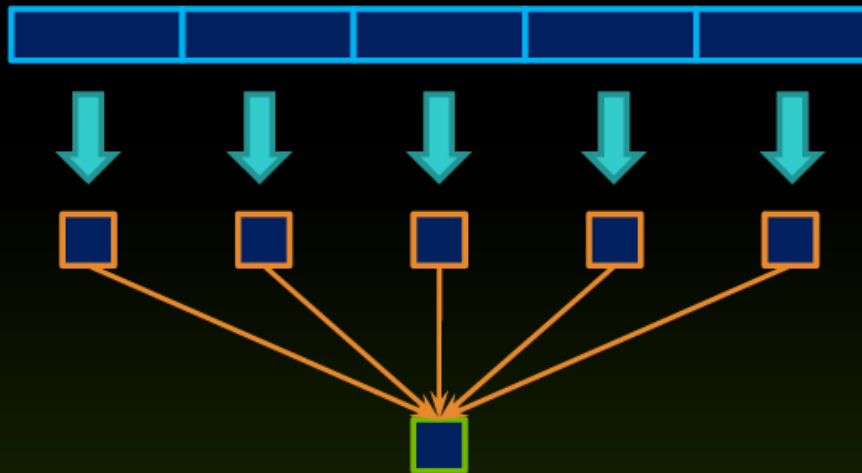
How else might we perform the update?

More on Privatization

- **Privatization** is a powerful and frequently-used technique for parallelizing applications
- The operation needs to be **associative and commutative**
 - True for all valid uses of **atomic operations**, because they **do not guarantee ordering**
 - Histogram add operation is associative and commutative
- The histogram size needs to be small
 - How small does it need to be? How small should it be?

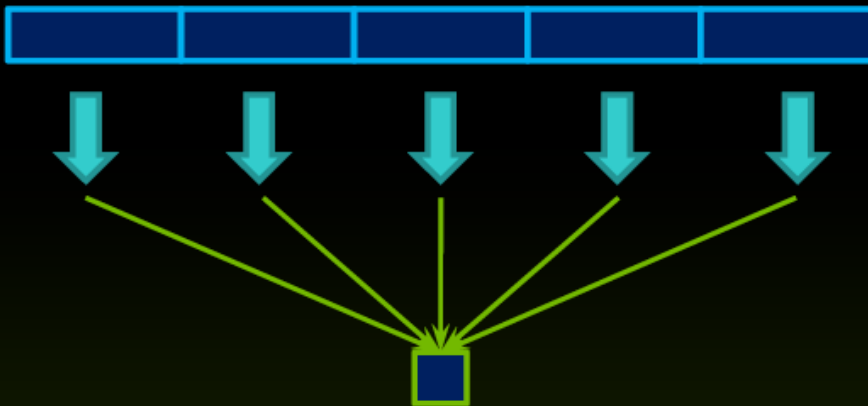
Revisiting Reduction

Without Atomics



1. Divide input data array into N sections
2. Launch N blocks, each reduces one section
3. Output is N values
4. Second launch of N threads, reduces outputs to single value

With Atomics



1. Divide input data array into N sections
2. Launch N blocks, each reduces one section
3. Write output directly via atomic. No need for second kernel launch.

QUESTIONS?

MP5 due tonight

MP6 released today

Remember: Project Proposals

Due Wednesday at noon