

Contemporary Computer Architecture TDSN13

LECTURE 8 – CUDA HISTOGRAM & REDUCTION

ANDREAS AXELSSON (ANDREAS.AXELSSON@JU.SE)

Key Concept 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 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. These three operations are performed in one atomic transaction. The function returns **old**.

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions>

# 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.x+)

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

- Double-precision floating-point atomic add (Compute capability 6.x+)

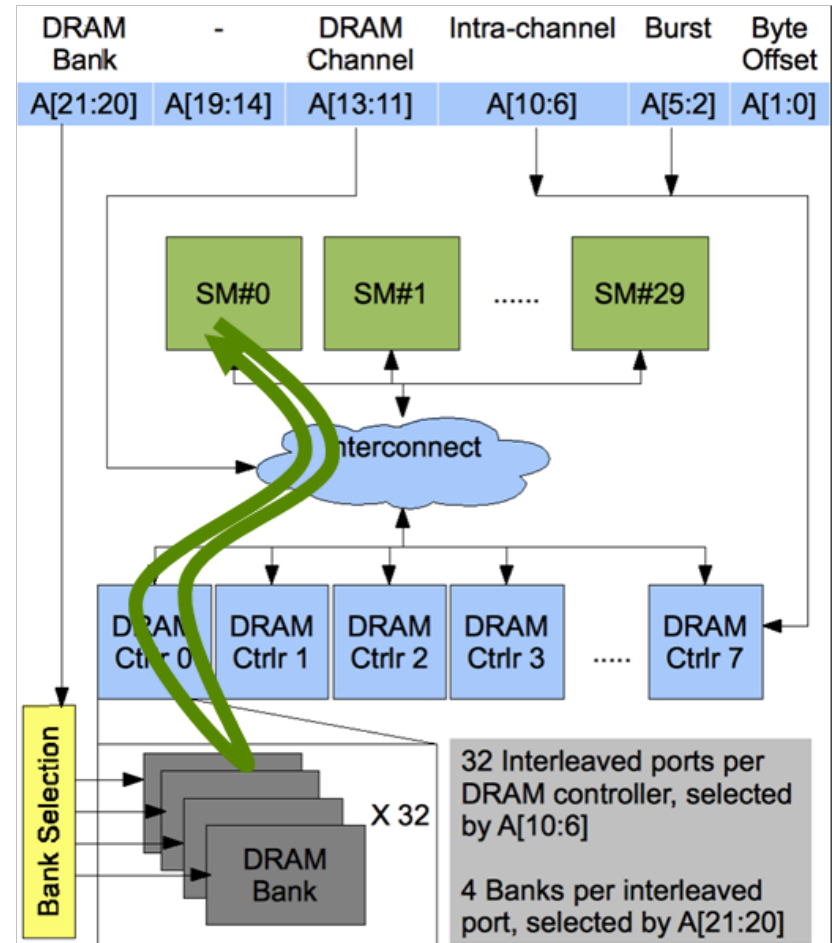
```
double atomicAdd(double* address, double val);
```

- 16-bit floating-point atomic add (Compute capability 7.x+)

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

# Atomic Operation on Global Mem

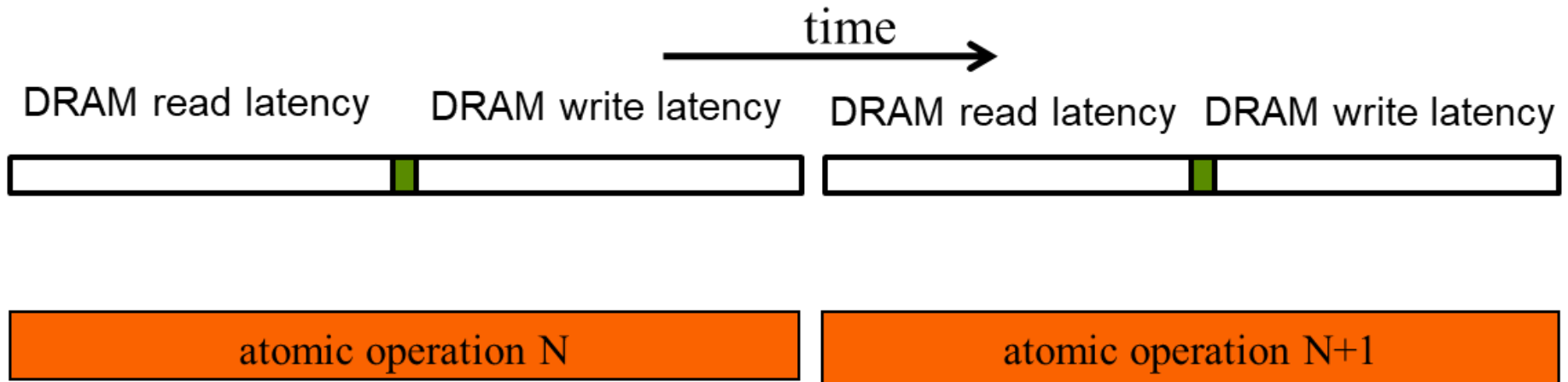
- An atomic operation on a DRAM location starts with a read, which has a latency of a few hundred cycles
- The atomic operation ends with a write to the same location, with a latency of a few hundred cycles
- During this whole time, no one else can access the location



# Atomic Operations on DRAM

---

- Each Read-Modify-Write has two full memory access delays
  - All atomic operations on the same variable (DRAM location) are serialized



# Latency determines throughput

---

- Throughput of atomic operations on the same DRAM location is the rate at which the application can execute an atomic operation.
- The rate for atomic operation on a particular location 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 throughput is reduced to  $< 1/1000$  of the peak bandwidth of one memory channel!

# Supermarket checkout experience

---

- 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 drastically 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  $1 / (\text{entire shopping time of each customer})$

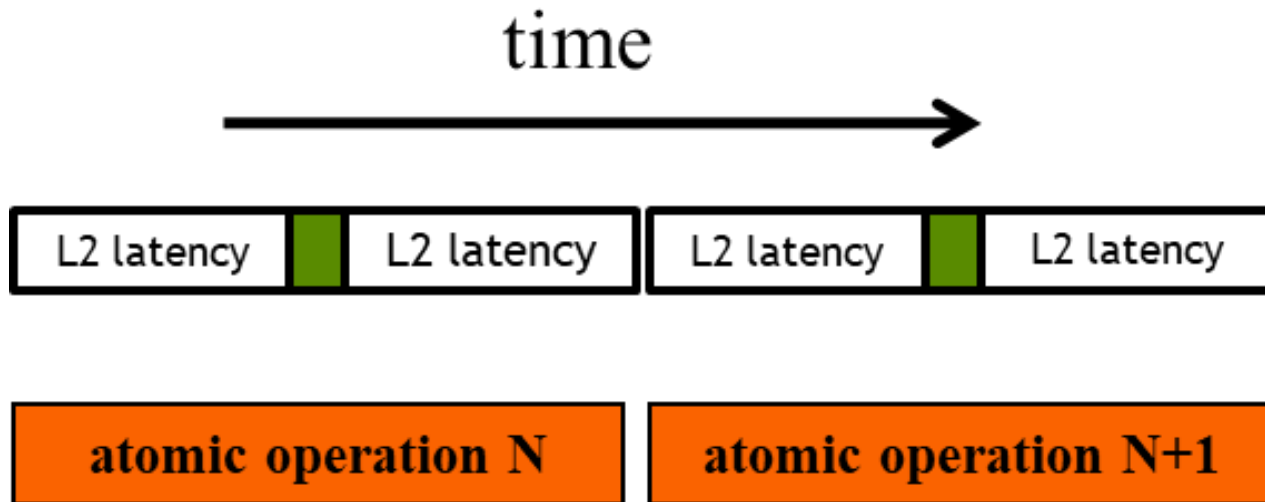




# Hardware Improvements

---

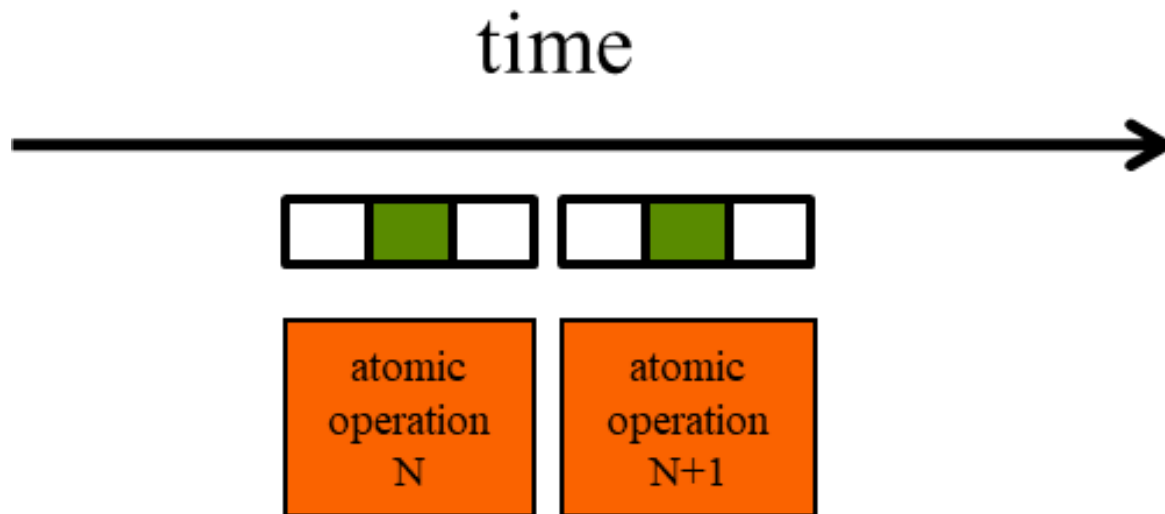
- Atomic operations on Fermi L2 cache
  - Medium latency, about 1/10 of the DRAM latency
  - Shared among all blocks
  - “Free improvement” on Global Memory atomics



# Hardware Improvements

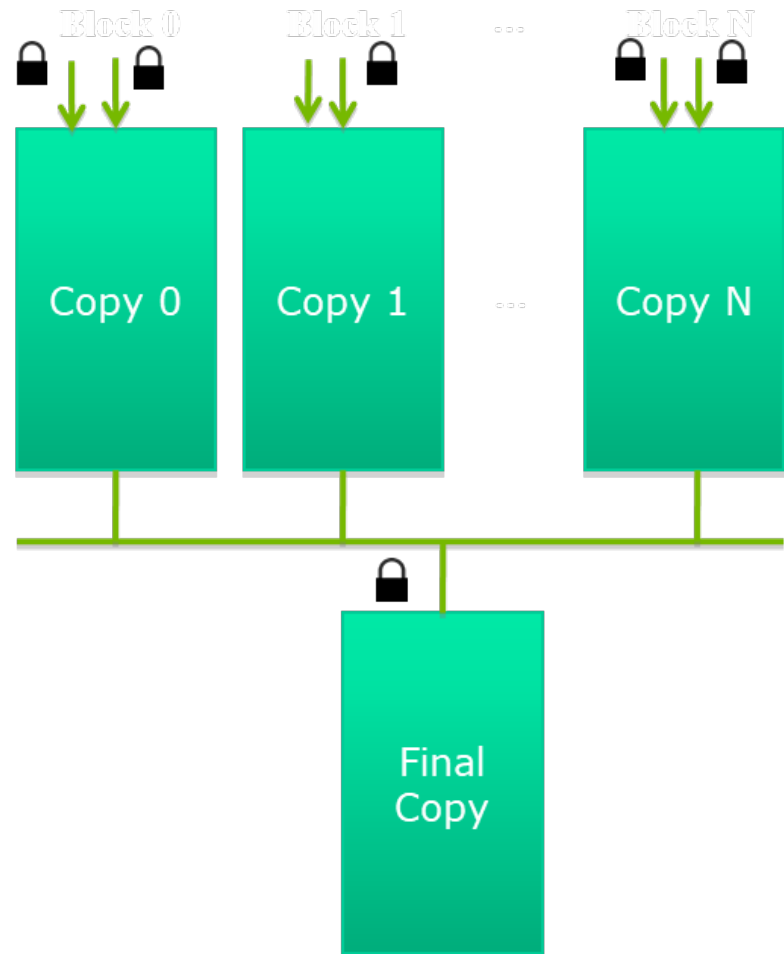
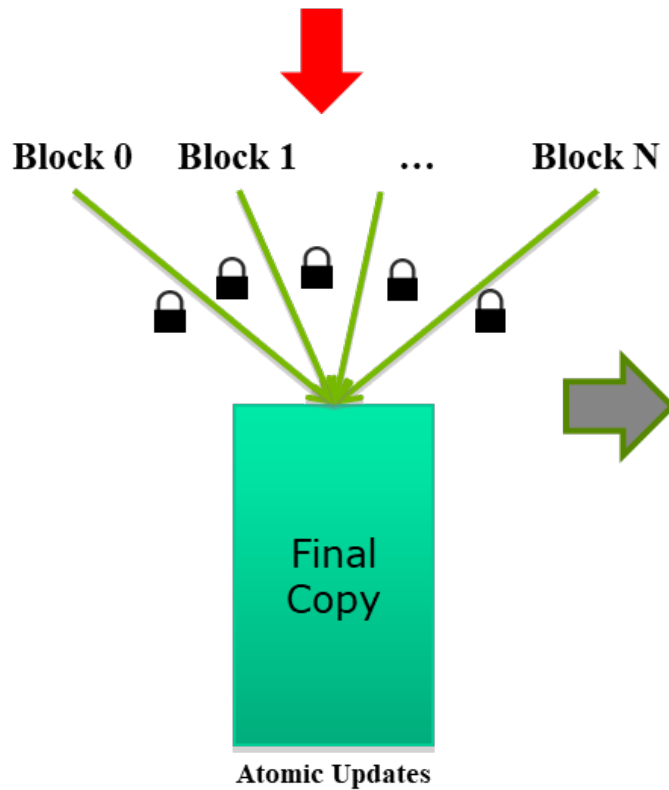
---

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

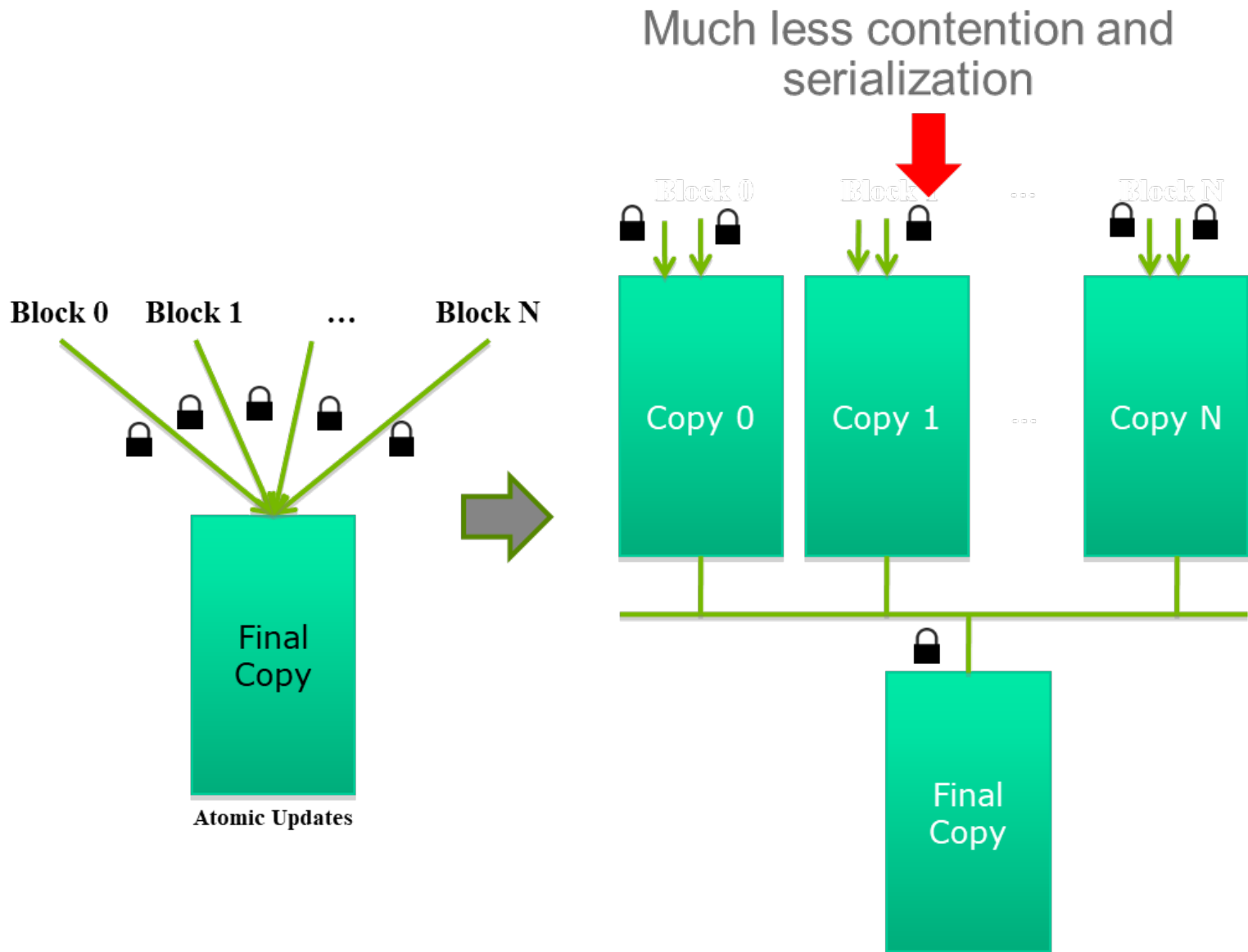


# Privatization

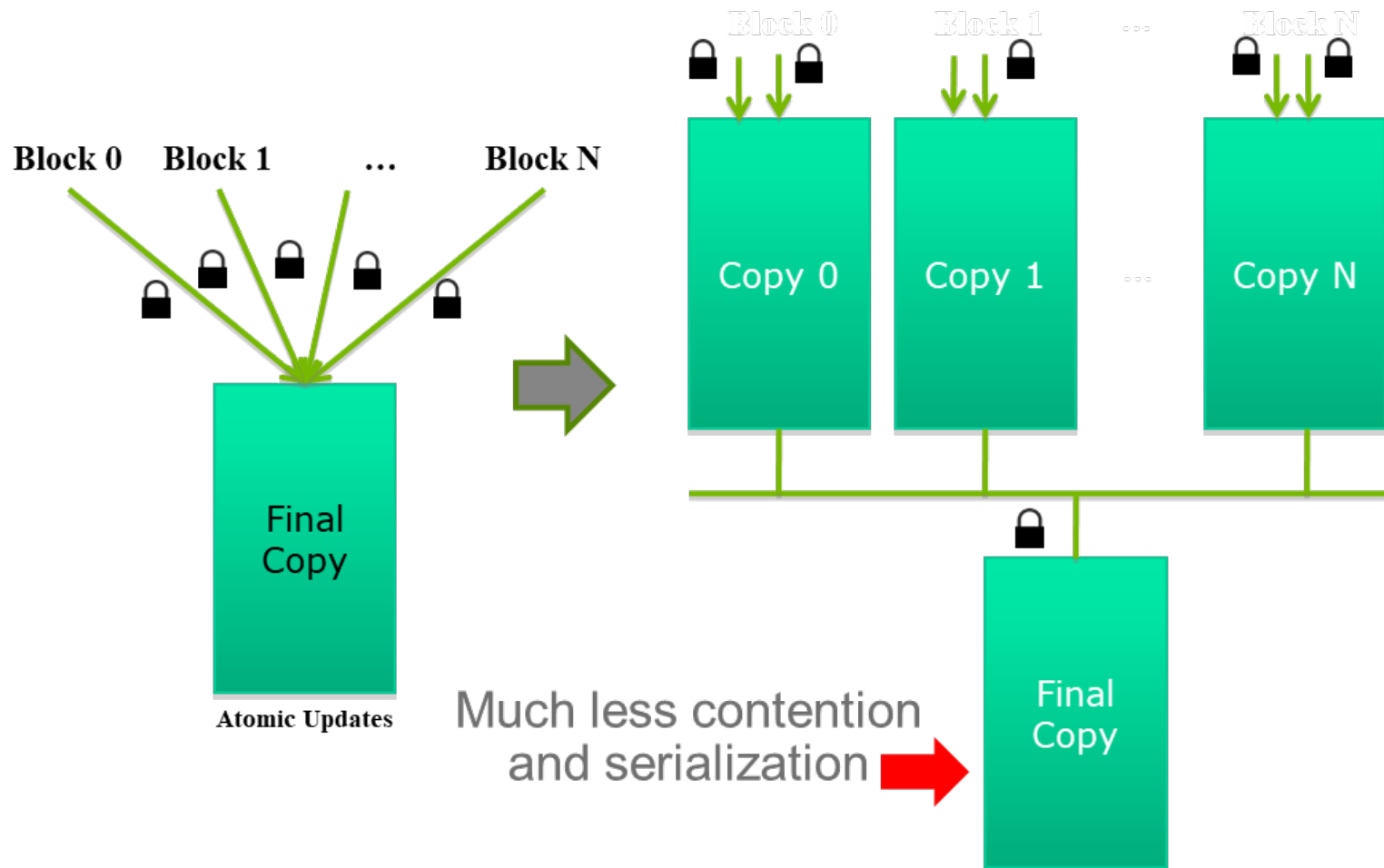
Heavy contention and  
serialization



# Privatization (cont.)



# 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 mem atomics in histogram

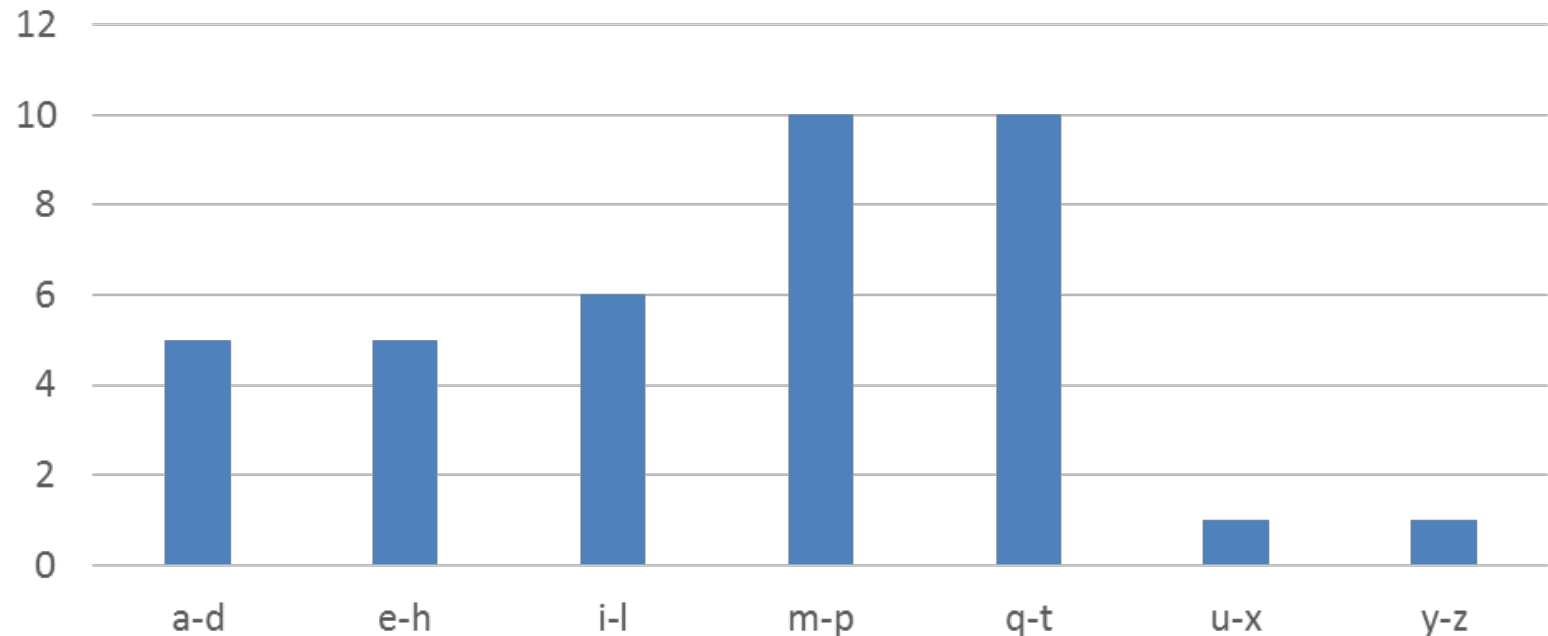
---

- 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!

# 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:





# Our Text 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;
 }
}
```

# Shared mem atomics privatization

---

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

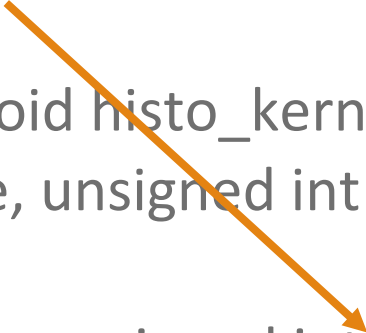
```
__global__ void histo_kernel(unsigned char *buffer,
 long size, unsigned int *histo)
{
 __shared__ unsigned int histo_private[7];
```

# Shared mem atomics 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];
```



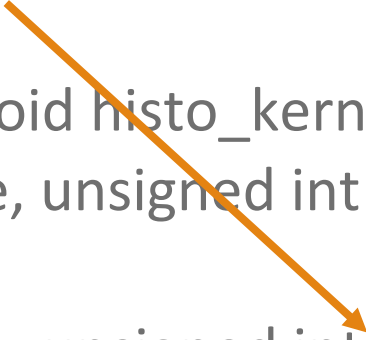
# Shared mem atomics privatization

---

- 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();
```



# Shared mem atomics 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(&(amp;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



# What is a reduction computation?

---

- Summarize a set of input values into one value using a “reduction operation”
  - Max
  - Min
  - Sum
  - Product
- Often used with a user defined reduction operation function as long as the operation
  - Is associative and commutative
  - Has a well-defined identity value (e.g., 0 for sum)
  - For example, the user may supply a custom “max” function for 3D coordinate data sets where the magnitude for the each coordinate data tuple is the distance from the origin.

An example of “collective operation”

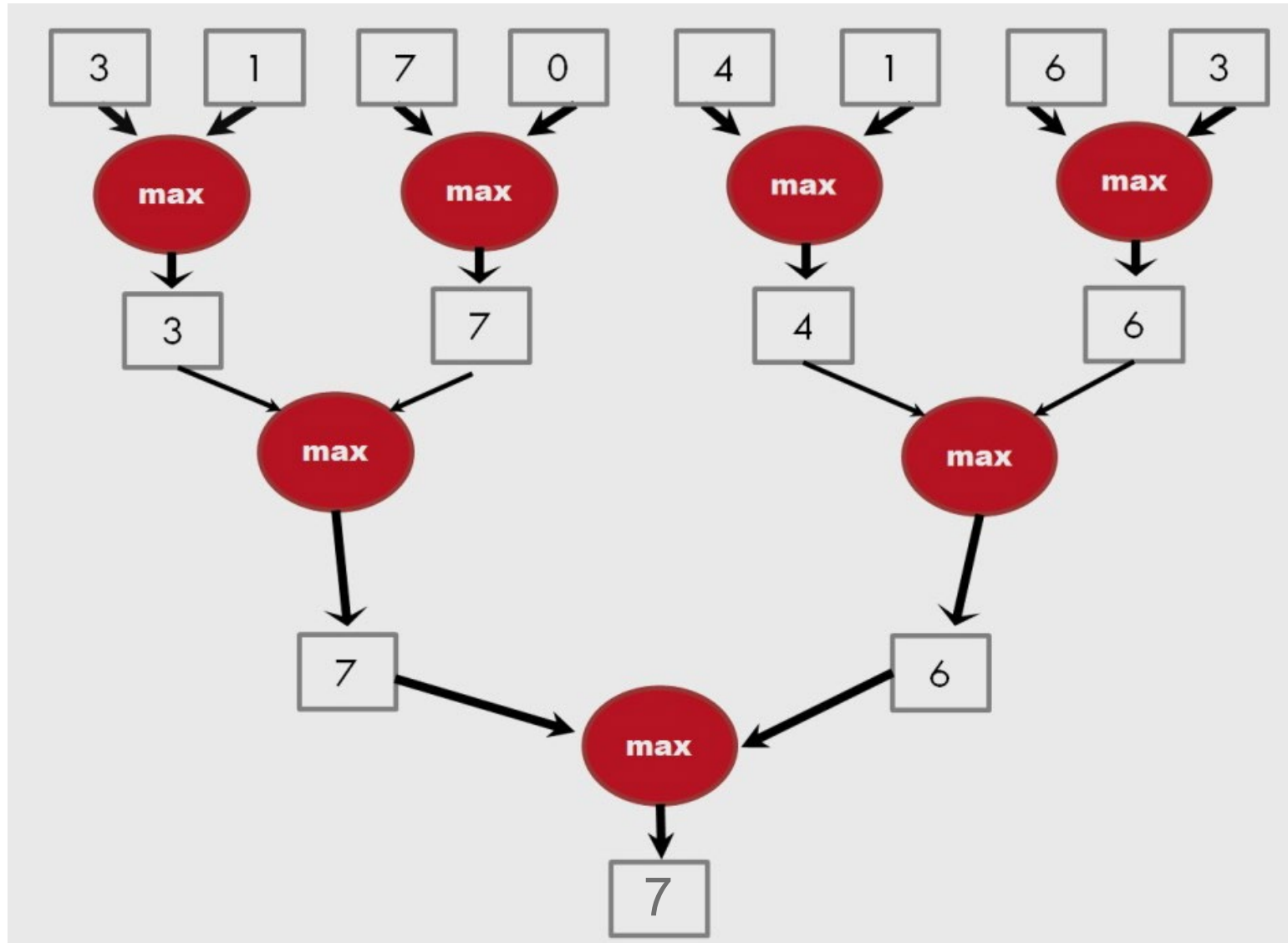
# Efficient Sequential Reduct. $O(N)$

---

- Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction
- Iterate through the input and perform the reduction operation between the result value and the current input value
  - $N$  reduction operations performed for  $N$  input values
  - Each input value is only visited once – an  $O(N)$  algorithm
  - This is a computationally efficient algorithm.

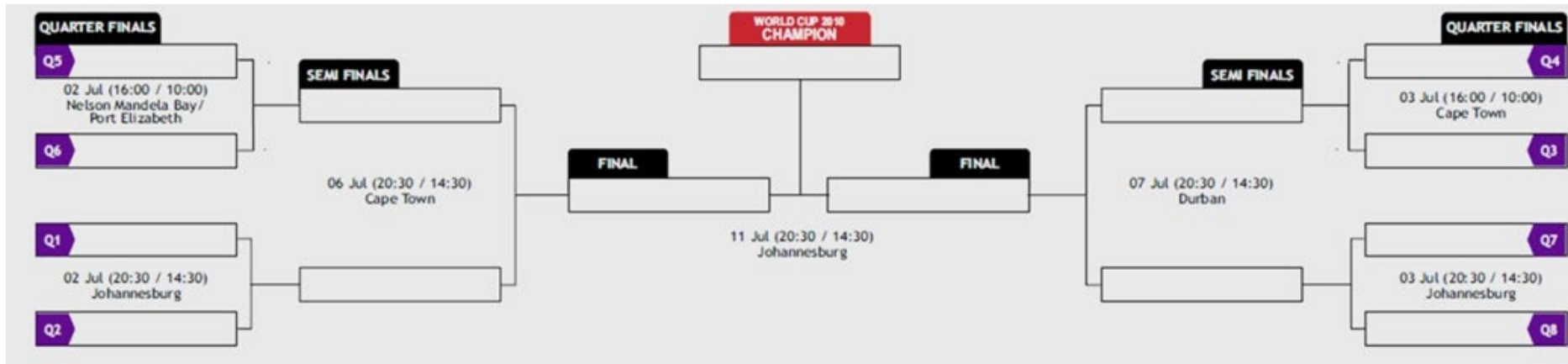
# A parallel reduction tree algorithm

---



# A tournament

---



A reduction tree with “max” operation

# A Quick Analysis

---

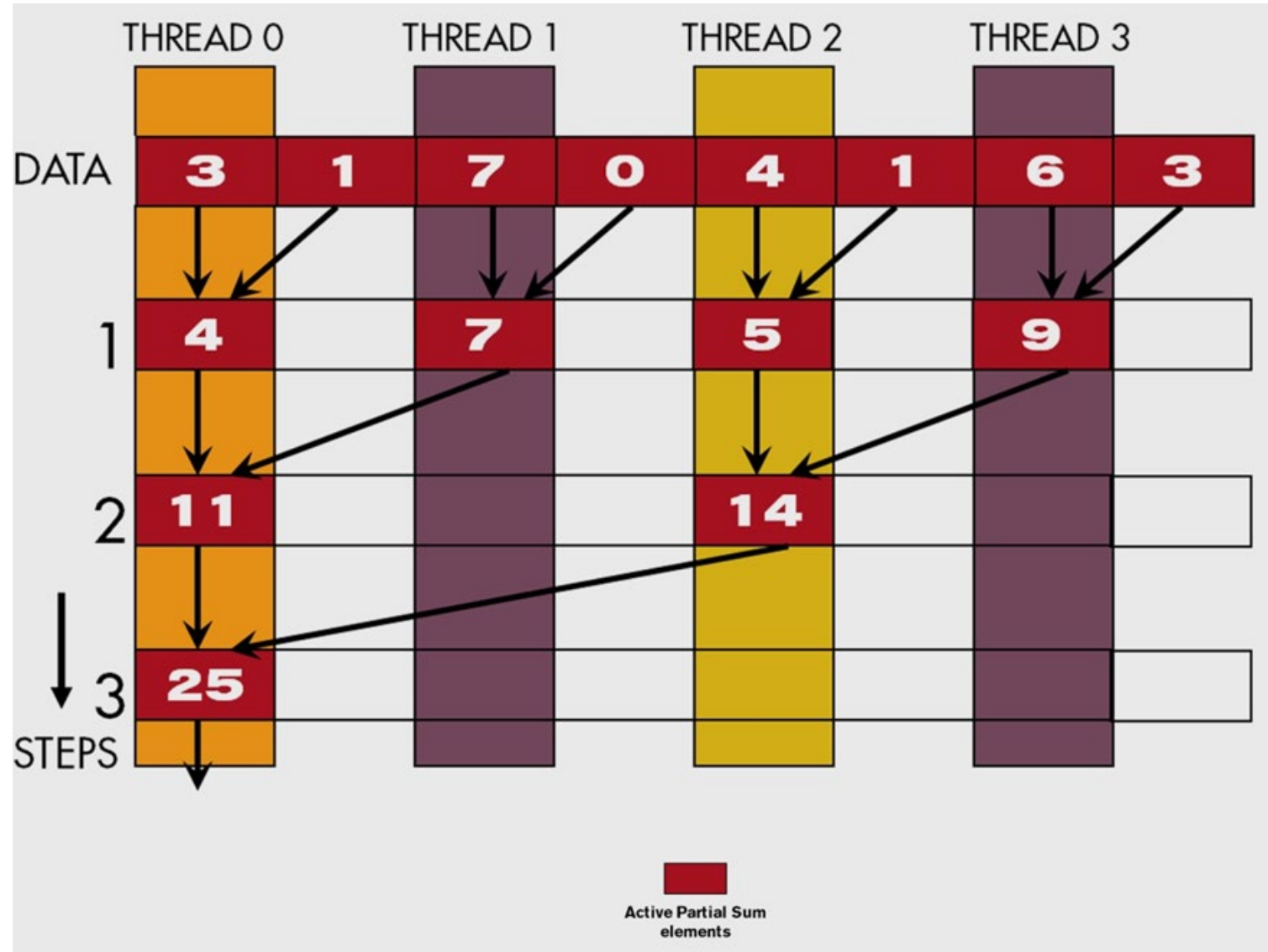
## – For $N$ input values, the reduction tree performs

- $(1/2)N + (1/4)N + (1/8)N + \dots (1)N = (1 - (1/N))N = N-1$  operations
- In  $\log(N)$  steps – 1,000,000 input values take 20 steps
  - Assuming that we have enough execution resources
- Average Parallelism  $(N-1)/\log(N)$ 
  - For  $N = 1,000,000$ , average parallelism is 50,000
  - However, peak resource requirement is 500,000
  - This is not resource efficient

## – This is a work-efficient parallel algorithm

- The amount of work done is comparable to the efficient sequential algorithm
- Many parallel algorithms are not work efficient

# A Parallel Sum Reduction Example



# A Naive Thread to Data Mapping

---

- Each thread is responsible for an even-index location of the partial sum vector (location of responsibility)
- After each step, half of the threads are no longer needed
- One of the inputs is always from the location of responsibility
- In each step, one of the inputs comes from an increasing distance away

# A Simple Thread Block Design

---

- Each thread block takes  $2 \times \text{BlockDim.x}$  input elements
- Each thread loads 2 elements into shared memory

```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start + blockDim.x+t];
```



# The Reduction Steps

---

```
for (unsigned int stride = 1;
 stride <= blockDim.x; stride *= 2)
{
 __syncthreads();
 if (t % stride == 0)
 partialSum[2*t] += partialSum[2*t+stride];
}
```

Why do we need `__syncthreads()`?

# Barrier Synchronization

---

- `__syncthreads()` is needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step

# Back to the Global Picture

---

- At the end of the kernel, Thread 0 in each block writes the sum of the thread block in `partialSum[0]` into a vector indexed by the `blockIdx.x`
- There can be a large number of such sums if the original vector is very large
  - The host code may iterate and launch another kernel
- If there are only a small number of sums, the host can simply transfer the data back and add them together
- Alternatively, Thread 0 of each block could use atomic operations to accumulate into a global sum variable.

# The naïve reduction kernel

---

- In each iteration, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition still consume execution resources
- Half or fewer of threads will be executing after the first step
  - All odd-index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the `if` test, poor resource utilization but no divergence
  - This can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire

# Thread Index Usage Matters

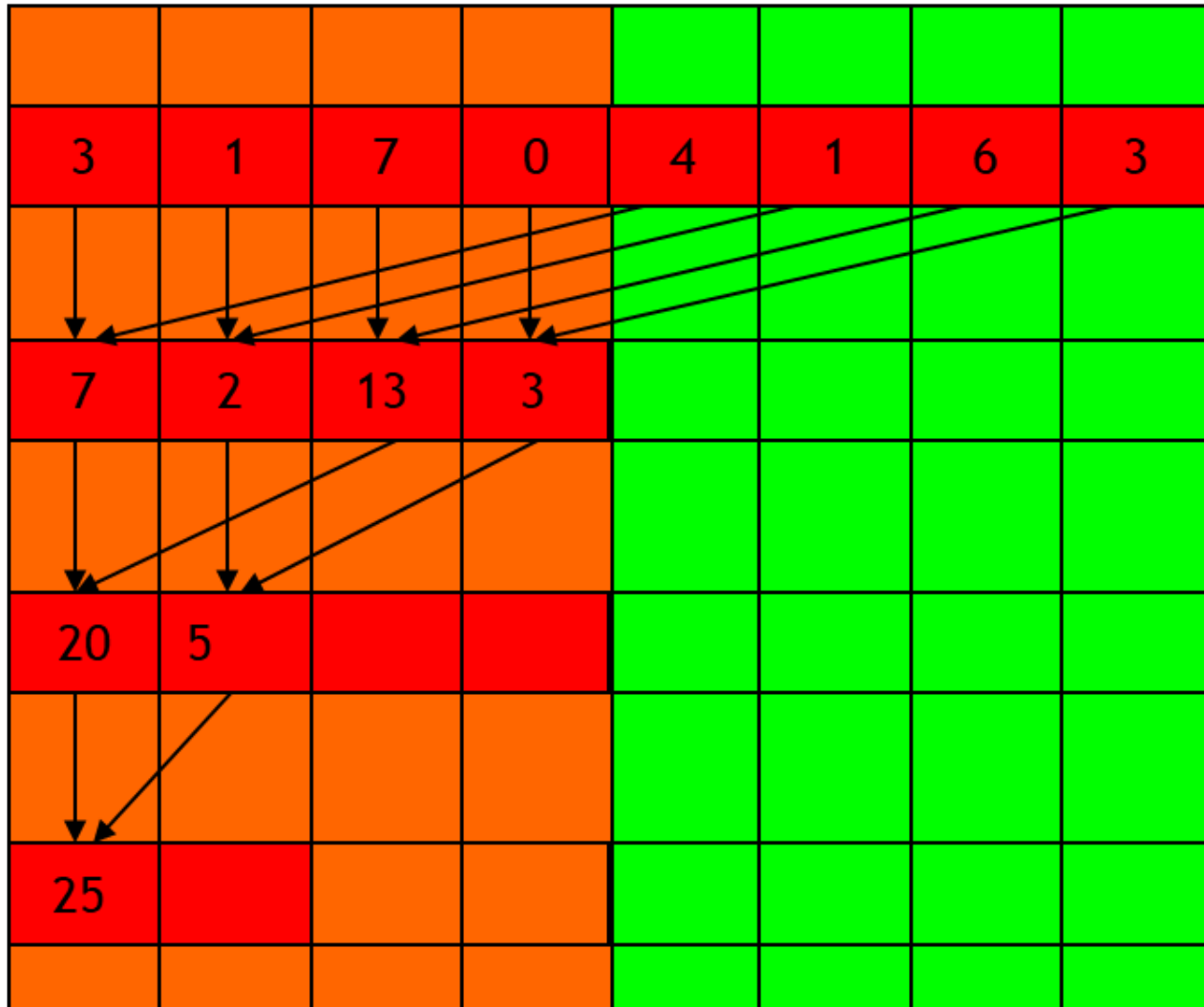
---

- In some algorithms, one can shift the index usage to improve the divergence behavior
  - Commutative and associative operators
- Always compact the partial sums into the front locations in the `partialSum[ ]` array
- Keep the active threads consecutive

# An Example of 4 threads

---

Thread 0 Thread 1 Thread 2 Thread 3



# A Better Reduction Kernel

---

```
for (unsigned int stride = blockDim.x;
 stride > 0; stride /= 2)
{
 __syncthreads();
 if (t < stride)
 partialSum[t] += partialSum[t+stride];
}
```

Takeaway: As long as stride is bigger than a warp size,  
either all t in a warp is less than stride or bigger than stride.  
Thus no warp divergence occurs.

# A Quick Analysis

---

- For a 1024 thread block
  - No divergence in the first 5 steps
    - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
    - All threads in each warp either all active or all inactive
  - The final 5 steps will still have divergence



# Introduction to CUDA

---

## Questions?

Contact information

Andreas Axelsson

Email: [andreas.axelsson@ju.se](mailto:andreas.axelsson@ju.se)

Mobile: 0709-467760