



GPU Teaching Kit

Accelerated Computing



Module 7 – Parallel Computation Patterns (Histogram)

Lecture 7.4 Atomic Operation Performance



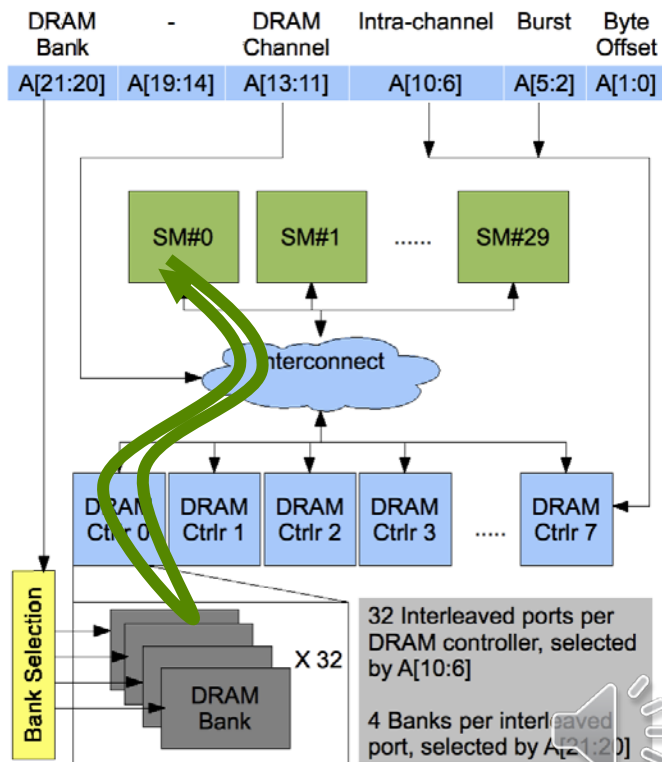
Objective

- To learn about the main performance considerations of atomic operations
 - Latency and throughput of atomic operations
 - Atomic operations on global memory
 - Atomic operations on shared L2 cache
 - Atomic operations on shared memory



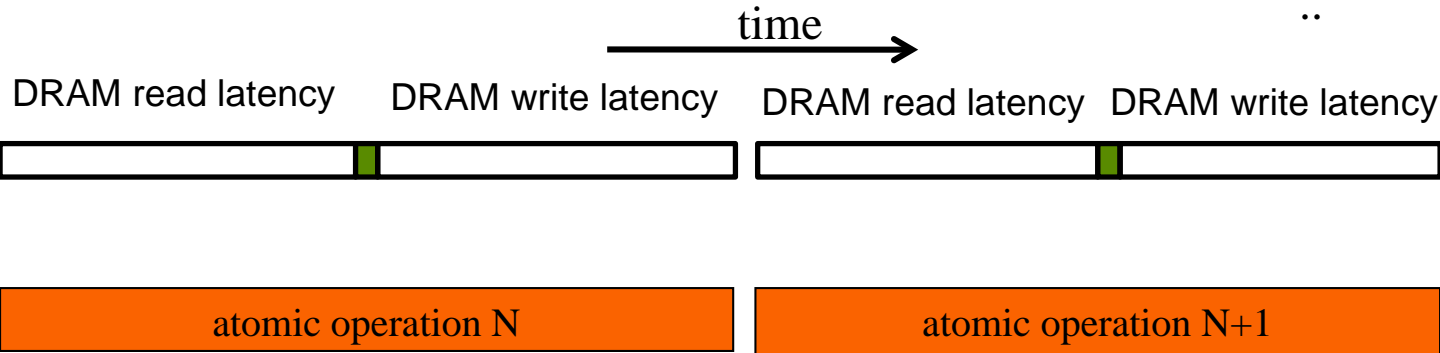
Atomic Operations on Global Memory (DRAM)

- 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
 - 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!



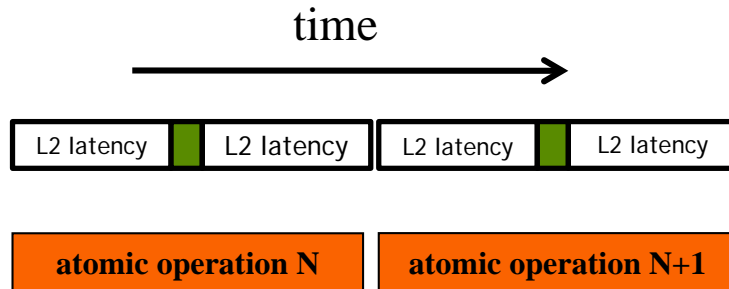
You may have a similar experience in supermarket checkout

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



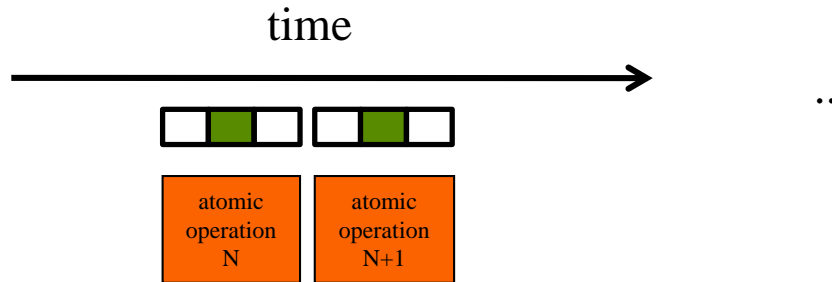
Hardware Improvements

- 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)





GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).





GPU Teaching Kit

Accelerated Computing



Module 7 – Parallel Computation Patterns (Histogram)

Lecture 7.5 – Privatization Technique for Improved Throughput



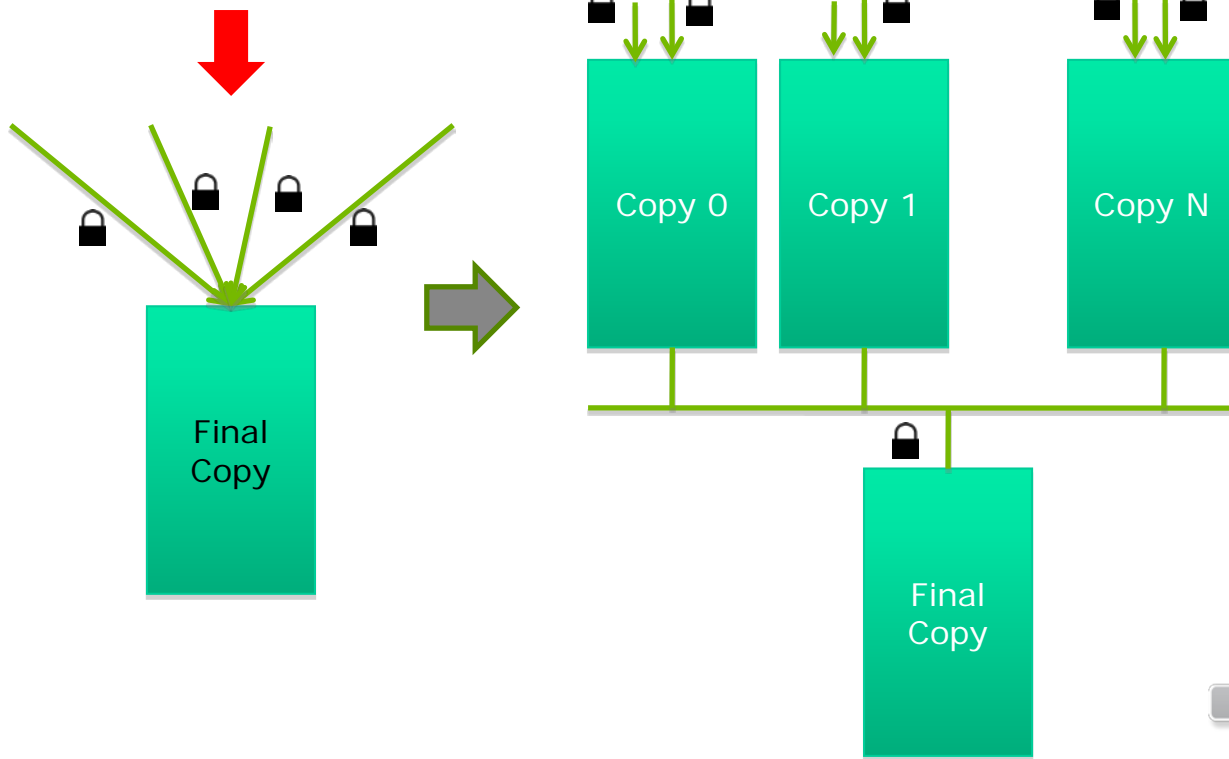
Objective

- 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



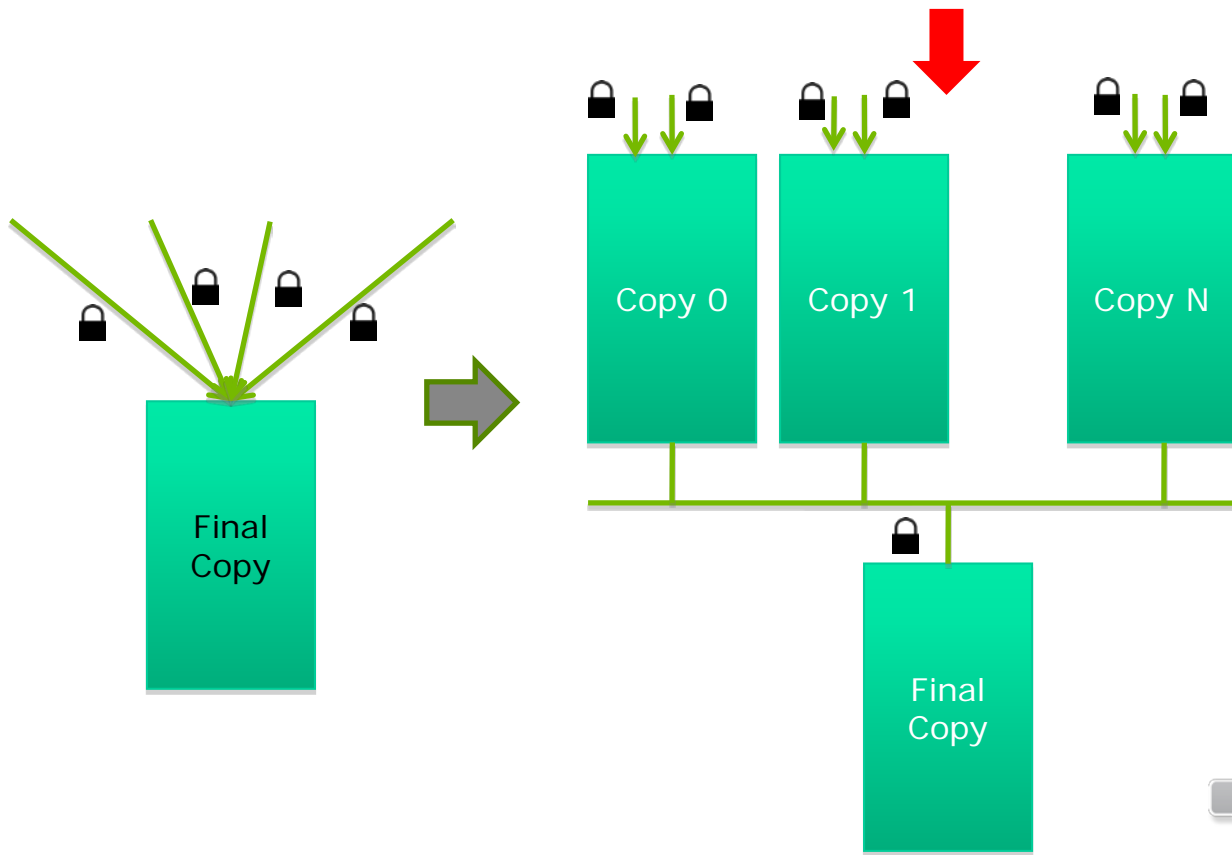
Privatization

Heavy contention and
serialization

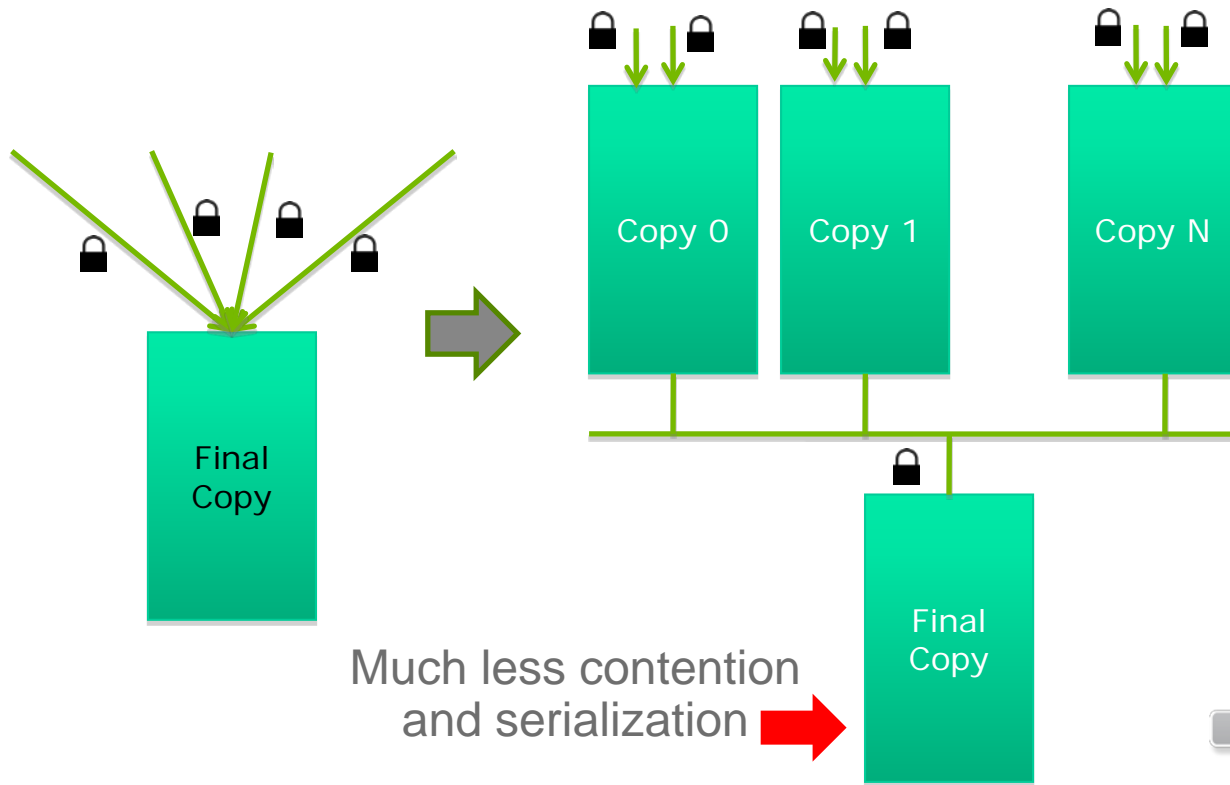


Privatization (cont.)

Much less contention and serialization



Privatization (cont.)



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

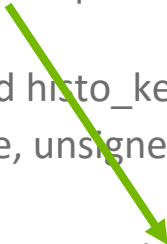
- Each subset of threads are in the same block
- Much higher throughput than DRAM (100x) or L2 (10x) atomics
- Less contention – only threads in the same block can access a shared memory variable
- This is a very important use case for shared memory!



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( &(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





GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

