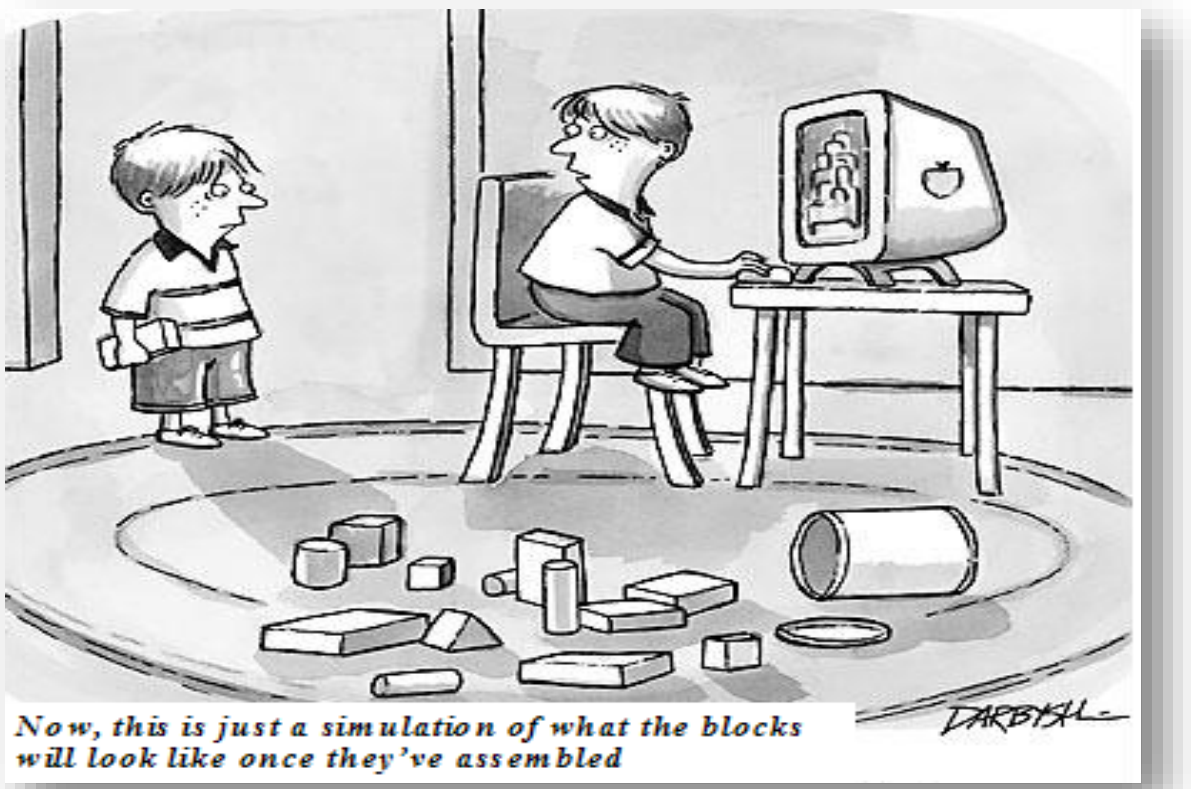
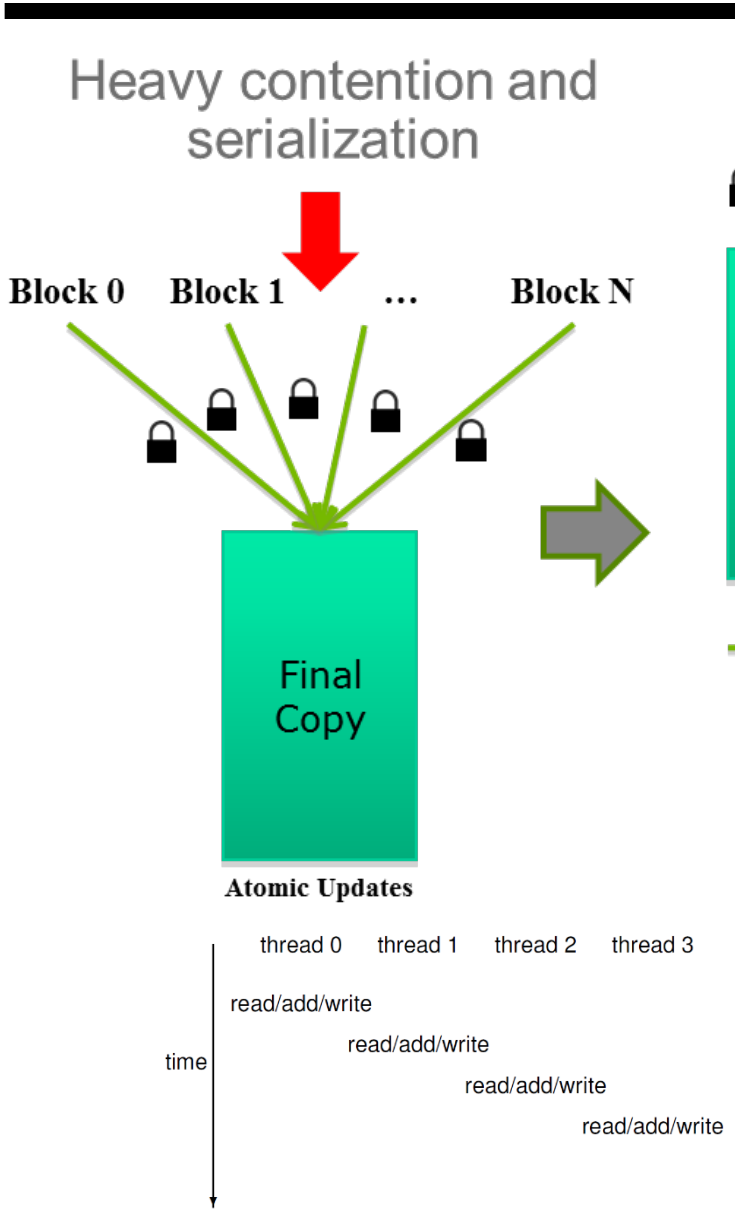
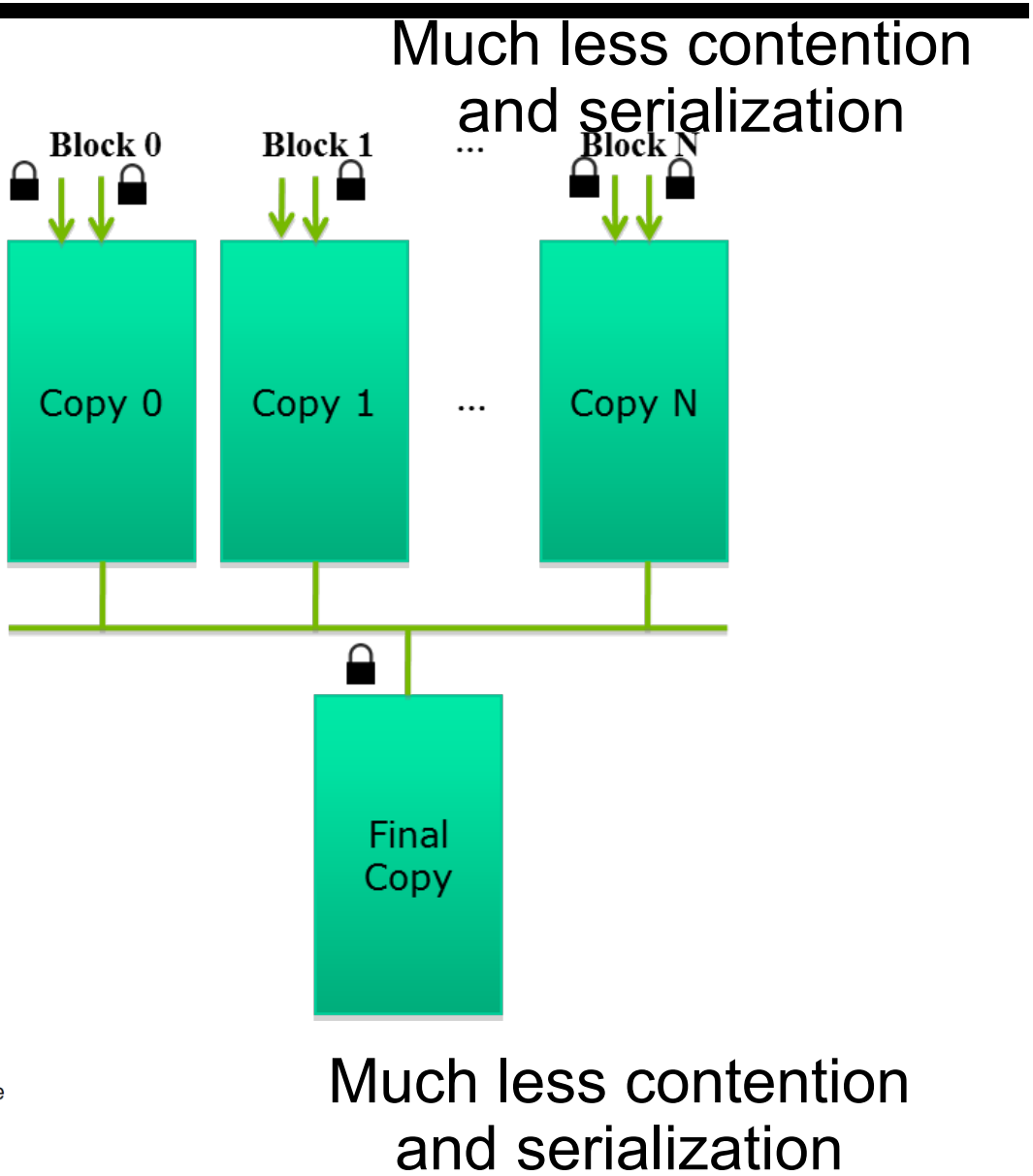
ECE569
Module 31



- Shared Memory based Histogram

Histogram with Privatization





Histogram with 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
  - This is a very important use case for shared memory!

Histogram with Privatization – Shared Memory

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

// Note: 7 bins,  4 characters per bin
// one copy per thread block in the grid
// initialize the bin counters to 0 in private copies
// each thread in a block writes 0 to shared memory
```

Histogram with Privatization – Shared Memory

```
__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();
   // build the private histogram
```

Will this initialization work correctly for the case where:

a) 7 bins, 32 threads/block

b) 64 bins 32 threads/block

Histogram with Privatization – Shared Memory

```
__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();
int i = threadIdx.x + blockIdx.x * blockDim.x;
int stride = blockDim.x * gridDim.x;
    while (i < size) {
         position = buffer[i] - “a”;
         atomicAdd( &histo_private[position/4], 1);
         i += stride;    }
// build final histogram assuming size < block size
```

Histogram with Privatization – Shared Memory

```
__global__ void histo_kernel(unsigned char *buffer,
    long size, unsigned int *histo)
{
    __shared__ unsigned int histo_private[6];

if (threadIdx.x < 7) histo_private[threadidx.x] = 0;
   __syncthreads();
int i = threadIdx.x + blockIdx.x * blockDim.x;
int stride = blockDim.x * gridDim.x;
    while (i < size) {
         position = buffer[i] – “a”;
         atomicAdd( &histo_private[position/4], 1);
         i += stride;    }
// wait for all other thread in the block to finish
__syncthreads();
if (threadIdx.x < 7) {
atomicAdd(&(histo[threadIdx.x]), histo_private[threadIdx.x] );
}
} //Assumes that number if bins is smaller than the block size
```

Histogram with Privatization – Shared Memory 640M atomic operations/sec

```
__global__ void histo_kernel(unsigned char *buffer,
    long size, unsigned int *histo)
{
    __shared__ unsigned int histo_private[6];

if (threadIdx.x < 7) histo_private[threadidx.x] = 0;
   __syncthreads();
int i = threadIdx.x + blockIdx.x * blockDim.x;
int stride = blockDim.x * gridDim.x;
    while (i < size) {
         position = buffer[i] – “a”;
         atomicAdd( &histo_private[position/4], 1);
         i += stride;    }
__syncthreads();
if (threadIdx.x < 7) {
atomicAdd(&(histo[threadIdx.x]), histo_private[threadIdx.x] );
}
}
```

How many arithmetic operations per second?

Histogram with Privatization – Shared Memory 640M atomic operations/sec

```
__global__ void histo_kernel(unsigned char *buffer,
    long size, unsigned int *histo) {
    __shared__ unsigned int histo_private[6];
 if (threadIdx.x < 7) histo_private[threadidx.x] = 0;
   __syncthreads();
int i = threadIdx.x + blockIdx.x * blockDim.x;
int stride = blockDim.x * gridDim.x;
    while (i < size) {
         position = buffer[i] – “a”;
         atomicAdd( &histo_private[position/4], 1);
         i += stride;    }
__syncthreads();
if (threadIdx.x < 7) {
atomicAdd(&(histo[threadIdx.x]), histo_private[threadIdx.x] );
}
}
```

How many arithmetic operations per second?

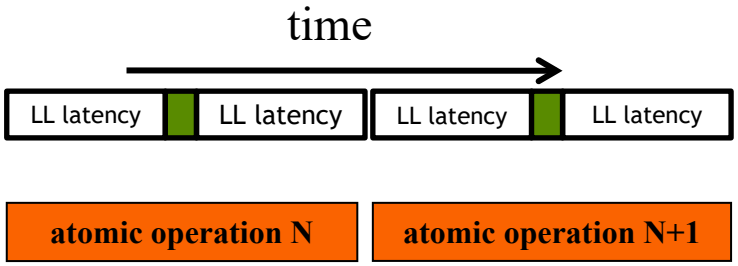
7*17.5M = 122.5Mops/sec

GPUs more than 10^12 ops/second

Motivated several optimization strategies

Hardware Improvements

- Atomic operations in last level cache (LLC)
  - Medium latency, about **1/10** of the DRAM latency
  - Shared among all SMs
  - if updated variable found in LLC, it is updated in the LLC
    - If not in LLC: cache miss, bring into the cache where it is updated
  - **Very high hit rate for atomic operations!**
    - variables updated by atomic operations tend to be heavily accessed by many threads
  - "Free improvement" on Global Memory atomics
    - Programmer doesn't need to make any change in the program to benefit
  - **Histogram**: 10x is improvement
    - 51200M operations/second
      - Still far from 10^12



Scalability

- **In general algorithms that rely on atomics have limited scalability.**
  - # of bins !
  - Need ways to improve scalability with a potential increase in complexity of the algorithm

Privatization - Local Histogram – Other Ideas

- **Case study: Assume 128 items, 8 threads to generate a histogram and 3 bins to target**
  - Each thread will be responsible for 16 items
  - Rather than only one single set of 3 bins in memory, what if we launch 8 threads and give each thread its own set of bins (local histogram)
    - **Do we need atomic operations to manage access to these local per-thread histograms?**
  - Then establish an adder tree structure to merge all 8 local histograms
  - Finally update the global histogram

Another Idea (Sort and Reduce by Key)

- **Sorting Based**

Bin	1	0	2	0	1	0	2	2	1	0	0	1	2	2	…
Val	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…

														…
														…

Another Idea (Sort and Reduce by Key)

- **Sort and Reduce**

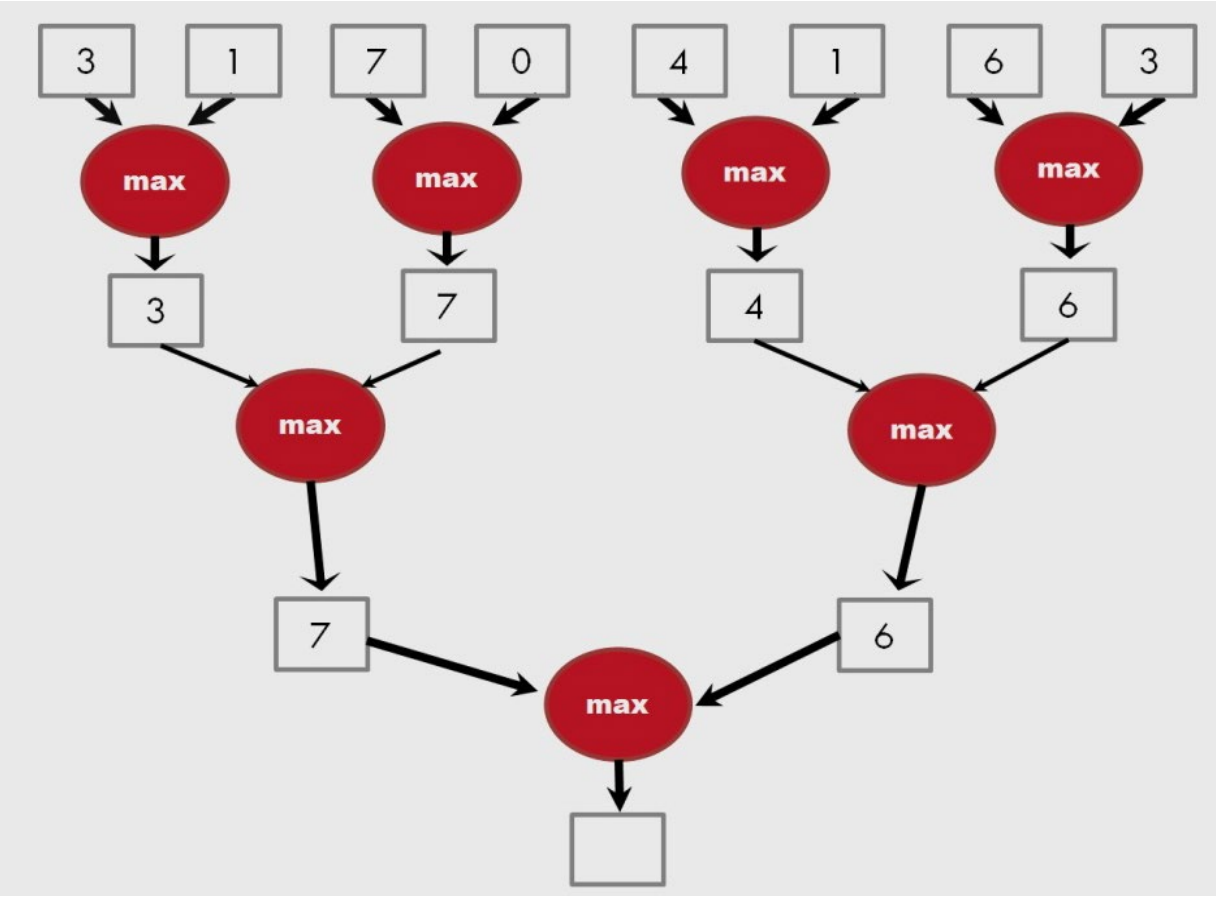
Bin	1	0	2	0	1	0	2	2	1	0	0	1	2	2	…
Val	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…

0	0	0	0	0	1	1	1	1	2	2	2	2	2	…
1	1	1	1	1	1	1	1	1	1	1	1	1	1	…

0	1	2
5	4	5

Another Idea

- **Reduction**
  - Coming soon



Next

- **Computation patterns**