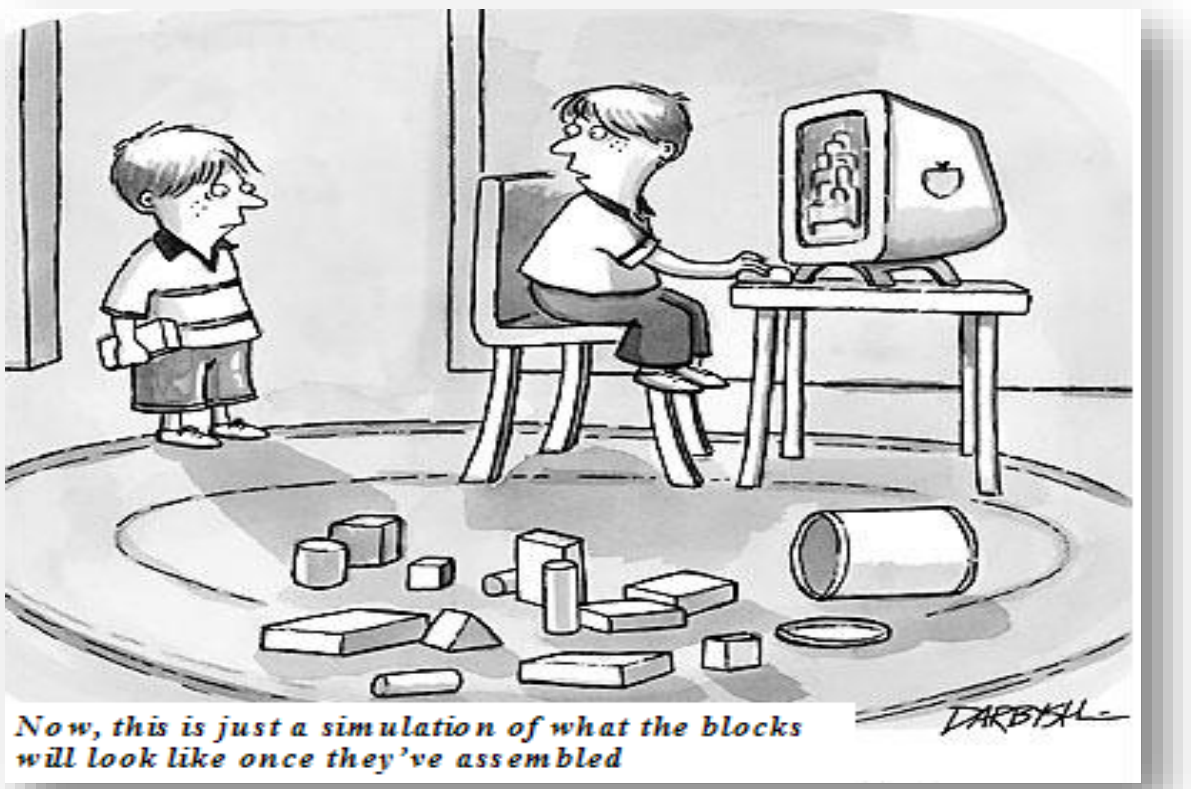
ECE569
Module 29



- Histogram with Atomic Add

Text Histogram Generation Example

- How do you do this in parallel?
  - Partition the input into sections
  - Have each thread to take a section of the input
  - Each thread iterates through its section.
  - For each letter, increment the appropriate bin counter
  - Use atomic operations to build the histogram

Basic Text Histogram Kernel

```
__global__ void histo_kernel(unsigned char *buffer,
    long size, unsigned int *histo)
{                          // thread mapping to a index
    int i =

                           // section size is workload per thread
```


```
int section_size =

// start is the starting index for each thread
int start =
for(                                          ){
   if(                                    )//check boundary
                                           //get position in alphabet

                                           //call atomicAdd


 }
}
```

Basic Text Histogram Kernel: Memory inefficient, why?

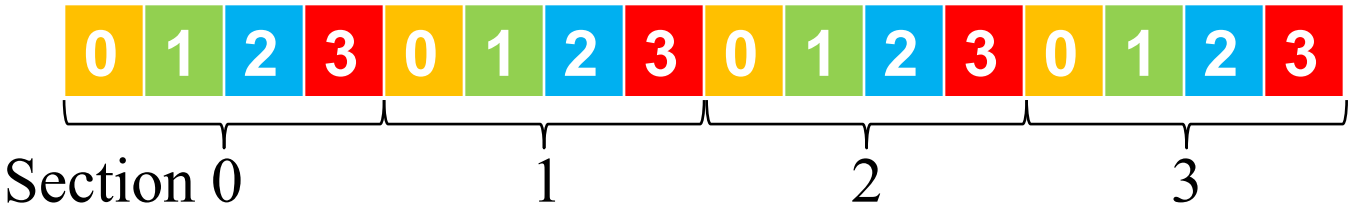
```
__global__ void histo_kernel(unsigned char *buffer,
    long size, unsigned int *histo)
{   // thread mapping to a index
    int i = threadIdx.x + blockIdx.x * blockDim.x;

// section size is workload per thread
int section_size=(size-1)/ (blockDim.x * gridDim.x + 1;
// start is the starting index for each thread
int start = i*section_size;
 for(k=0;k<section_size;k++){
   if(start+k<size)
      int alphabet_position = buffer[start+k] - "a";
   if (alphabet_position >= 0 && alpha_position < 26)
      atomicAdd(&(histo[alphabet_position/4]), 1);
 }
}
```

Partitioning and Memory Access Efficiency

- **partitioning results in poor memory access efficiency**
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized

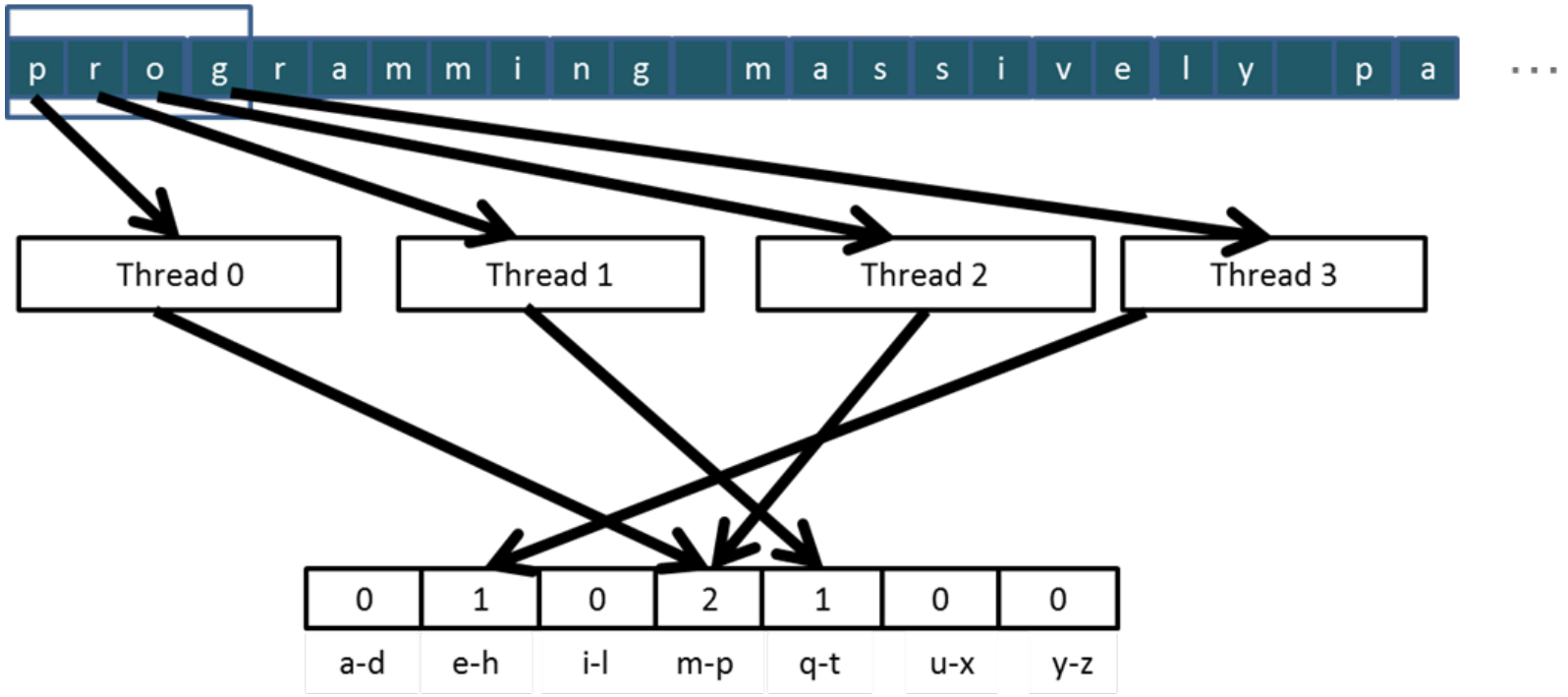
Alternative Access Pattern

- **Change to interleaved partitioning**
  - All threads process a contiguous section of elements
  - They all move to the next section and repeat
  - The memory accesses are coalesced



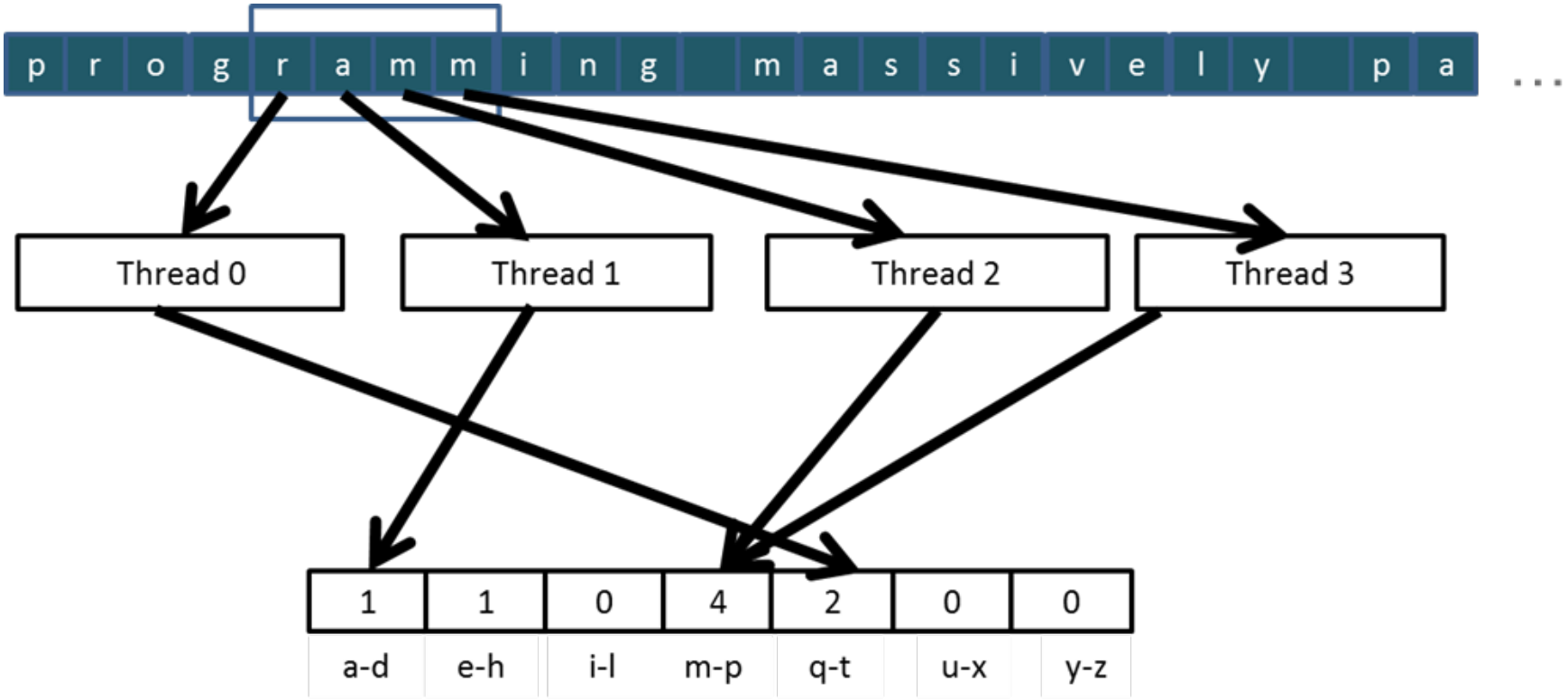
Interleaved Partitioning

- For coalescing and better memory access performance
- Iteration 1



Interleaved Partitioning

- Iteration 2



Basic Text Histogram Kernel –Stride Access

```
__global__ void histo_kernel(unsigned char *buffer,
    long size, unsigned int *histo)
{   // thread mapping to a index
    int i =

// stride is total number of threads
    int stride =

 // All threads handle blockDim.x * gridDim.x
   // consecutive elements
                                { // Loop
   int alphabet_position = buffer[i] - "a";
   if (alphabet_position >= 0 && alpha_position < 26)
      atomicAdd(&(histo[alphabet_position/4]), 1);

 } // end of loop
}
```

Basic Text Histogram Kernel –Stride Access

```
__global__ void histo_kernel(unsigned char *buffer,
    long size, unsigned int *histo)
{   // thread mapping to a index
    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;
 }
}
```

Analysis of the Histogram Generation

- Atomic operations in a parallel environment present a real challenge because they serialize execution.
  - algorithm should be designed to keep the number of threads that must wait for the lock to be released to a minimum.
- **Under what circumstance would the histogram generation performs**
  - The worst?
  - The best?

Analysis of the Histogram Generation

- Each element in the vector contains the count for a single bin in the histogram.
- For uniformly distributed data,
  - will keep a number of threads equivalent to the number of active bins.
  - When this number is large, the histogram will demonstrate high performance because many threads will be actively incrementing histogram counts.
- Performance suffers when the data is not uniformly distributed, causing many of the items fall into a few bins.
  - A pathological case occurs when all the histogram data fits into a single bin.

Synchronization vs. Coordination

- As far as CUDA is concerned, there is a **qualitative difference** between a __syncthreads() function and an atomic operation
- __syncthreads() : barrier
  - establishes a point in the execution of the kernel that every thread in the block needs to reach before the execution continues beyond that point
- The “atomic operation” concept tied to the idea of coordination
  - Threads in a grid of blocks coordinate their execution so that a certain operation invoked in a kernel is conducted in an atomic fashion

Next

- **Atomic operations and performance considerations (DRAM)**