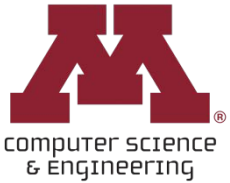
CSCI 5451: Introduction to Parallel Computing

Lecture 25: Histograms in Cuda



University of Minnesota
Driven to Discover®

Announcements (12/03)

- Updated HW4
  - Question 4 in HW4 has been updated as access to Colab is restricted for some
  - Be sure to review this updated file
  - The question is largely the same, but the timings are now given explicitly (you do not need to run separately on an A100 + T4)
- Cuda Machines now fully available except for GPU 2
  - If you are assigned to GPU 2, then select one of the other gpus to use at random

Overview

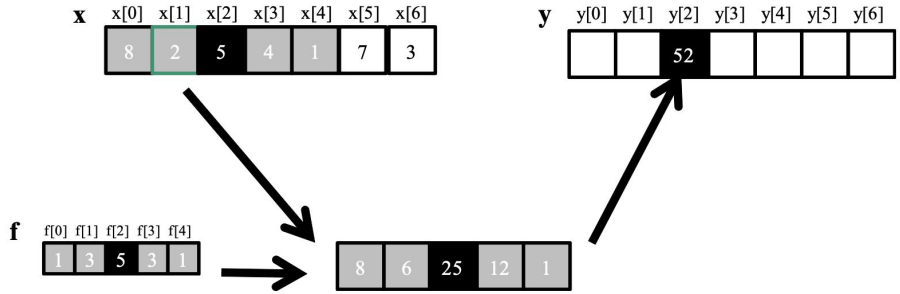
- ❑ Histograms
- ❑ Parallel Histograms
- ❑ Simple histogram Kernel
- ❑ Anatomy of Atomic Updating in Cuda
- ❑ Improved Histogram Kernels (Privatization, Shared Memory, Coarsening, Aggregation)

Overview

- ❑ **Histograms**
- ❑ Parallel Histograms
- ❑ Simple histogram Kernel
- ❑ Anatomy of Atomic Updating in Cuda
- ❑ Improved Histogram Kernels (Privatization, Shared Memory, Coarsening, Aggregation)

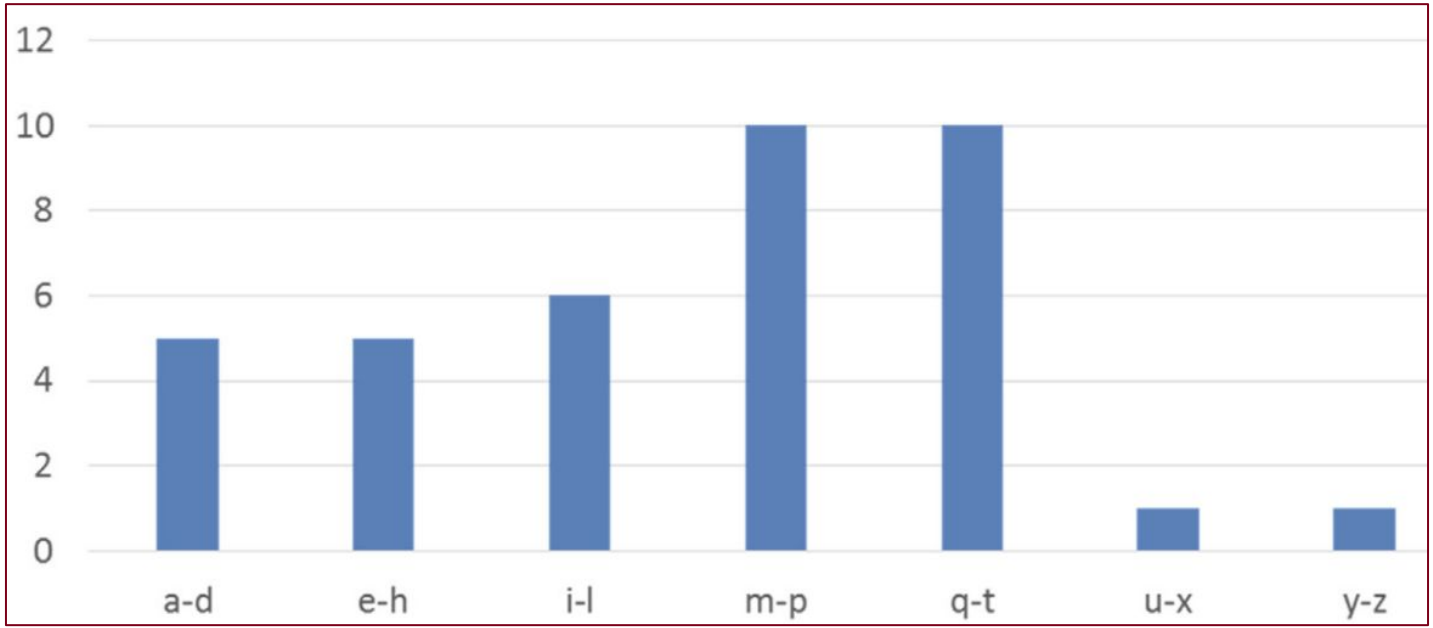
Histograms

- So far, the algorithms we have examined have used an output decomposition
- Each thread writes to a separate location (owner-computes)
- There is no contention for writing to the same memory location between different threads
- We do not need to coordinate writes across threads at all - they are independent



Histograms

We will use histograms to demonstrate a case where multiple threads need to write to the same location at a given time



Histograms

- The below algorithm is serial (CPU), runs in O(N) and assumes *data* is in ACSII format
- The result stores the number of groups of 4 chars ('a-d', 'e-h', 'i-l', etc.) in *histo*
- This takes reasonable advantage of the cache by operating on consecutive entries of *data*

```
01  void histogram_sequential(char *data, unsigned int length,
                              unsigned int *histo) {
02      for(unsigned int i = 0; i < length; ++i) {
03          int alphabet_position = data[i] - 'a';
04          if(alphabet_position >= 0 && alphabet_position < 26)
05              histo[alphabet_position/4]++;
06          }
07      }
08  }
```

Histograms

How to parallelize?

- The below algorithm is serial (CPU), runs in O(N) and assumes *data* is in ACSII format
- The result stores the number of groups of 4 chars ('a-d', 'e-h', 'i-l', etc.) in *histo*
- This takes reasonable advantage of the cache by operating on consecutive entries of *data*

```
01  void histogram_sequential(char *data, unsigned int length,
                              unsigned int *histo) {
02      for(unsigned int i = 0; i < length; ++i) {
03          int alphabet_position = data[i] - 'a';
04          if(alphabet_position >= 0 && alphabet_position < 26)
05              histo[alphabet_position/4]++;
06          }
07      }
08  }
```

Histograms

How to parallelize? Input Decomposition

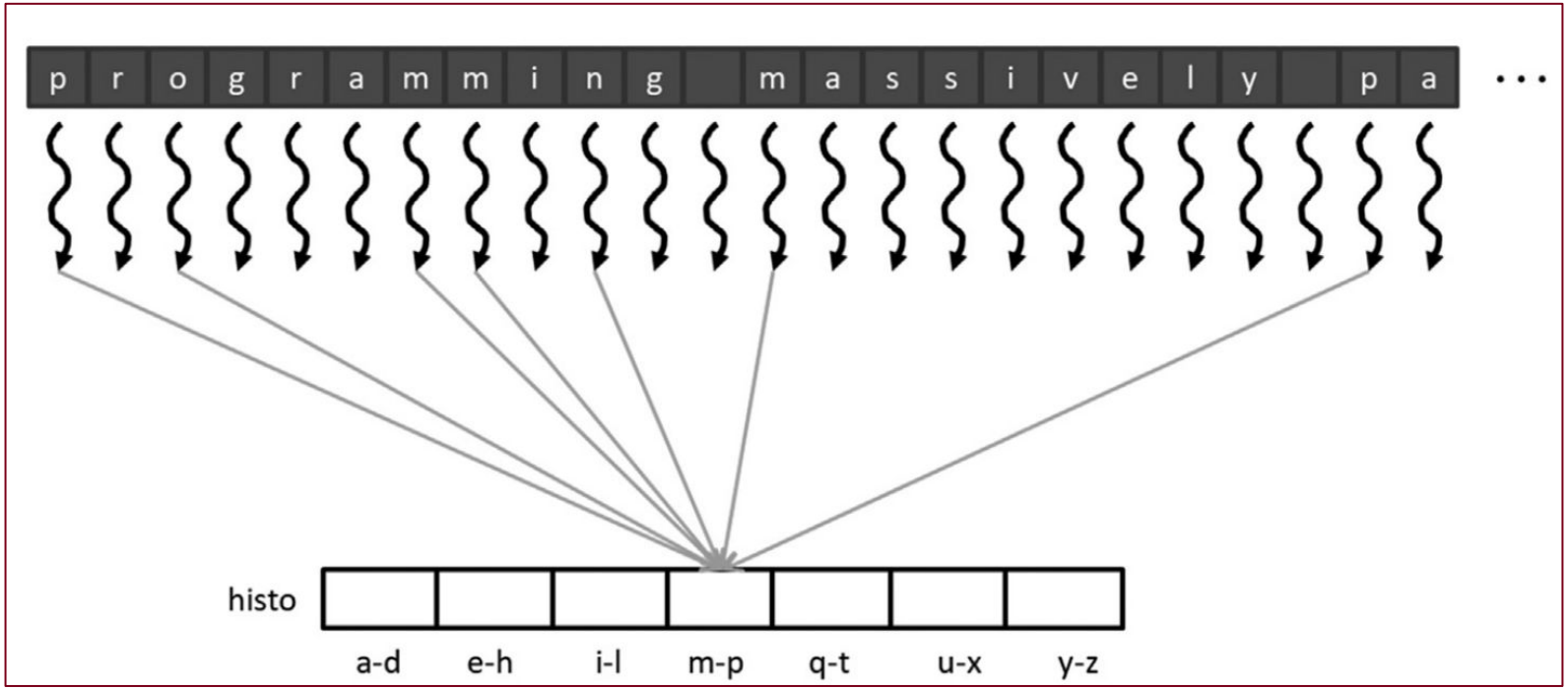
- The below algorithm is serial (CPU), runs in O(N) and assumes *data* is in ACSII format
- The result stores the number of groups of 4 chars ('a-d', 'e-h', 'i-l', etc.) in *histo*
- This takes reasonable advantage of the cache by operating on consecutive entries of *data*

```
01  void histogram_sequential(char *data, unsigned int length,
                              unsigned int *histo) {
02      for(unsigned int i = 0; i < length; ++i) {
03          int alphabet_position = data[i] - 'a';
04          if(alphabet_position >= 0 && alphabet_position < 26)
05              histo[alphabet_position/4]++;
06          }
07      }
08  }
```

Overview

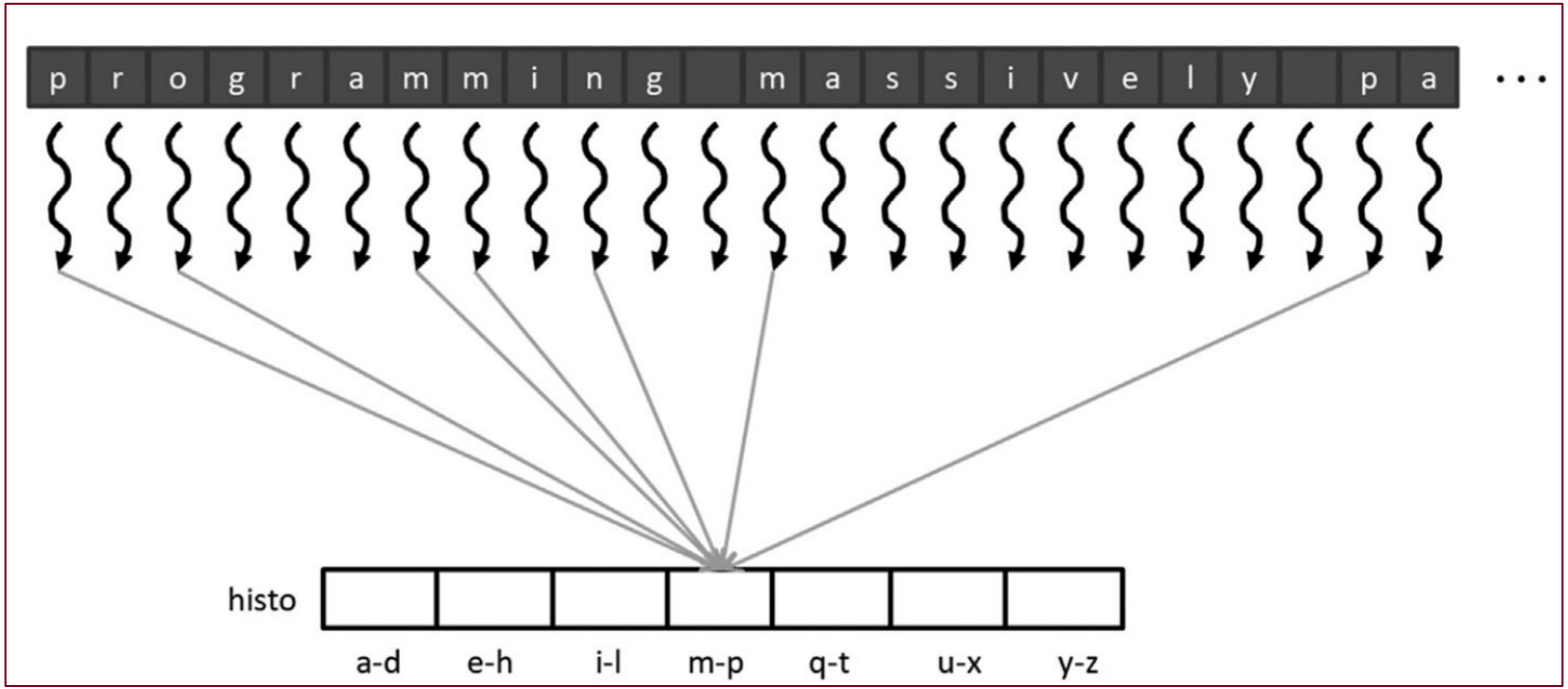
- ❑ Histograms
- ❑ **Parallel Histograms**
- ❑ Simple histogram Kernel
- ❑ Anatomy of Atomic Updating in Cuda
- ❑ Improved Histogram Kernels (Privatization, Shared Memory, Coarsening, Aggregation)

Parallel Histogram



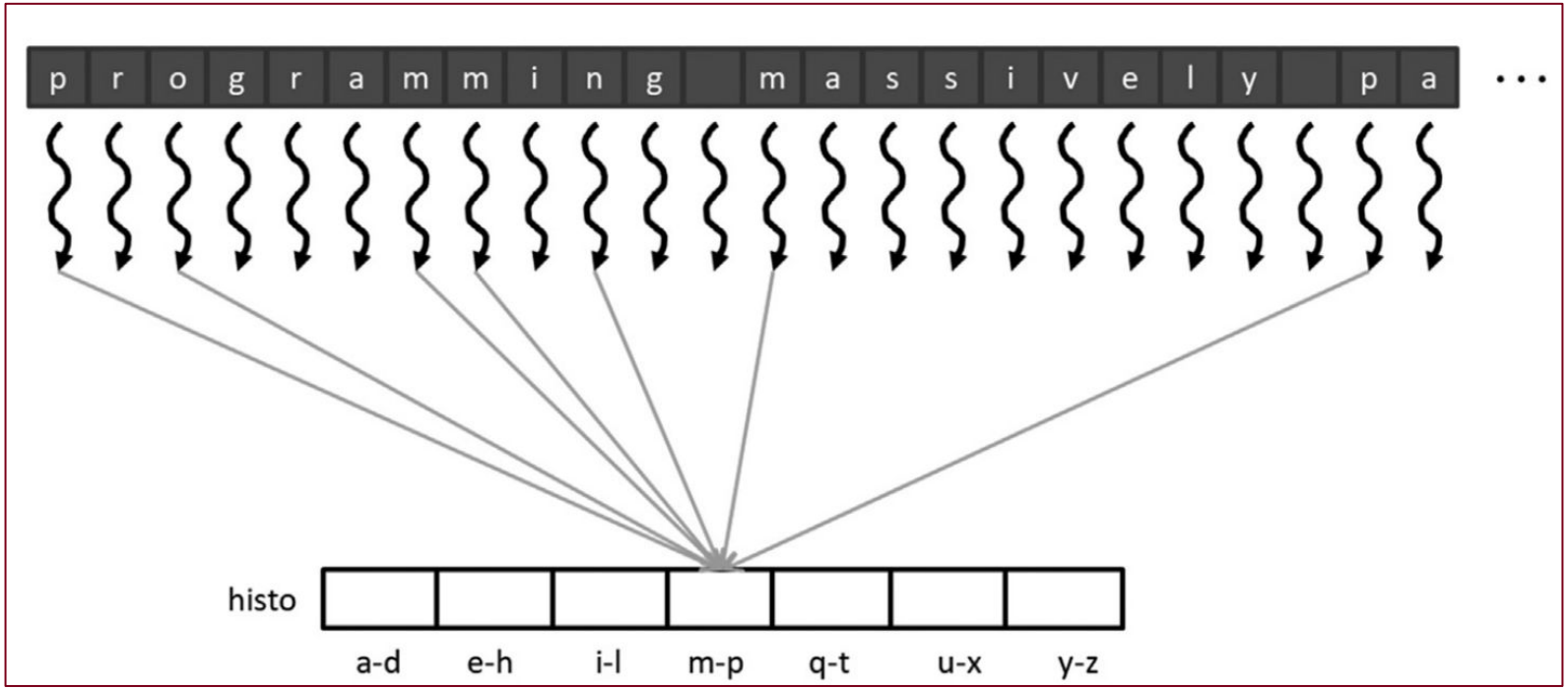
Parallel Histogram

What problem does this introduce?



Parallel Histogram

What problem does this introduce?
Race conditions



Race Conditions

- Recall that race conditions can occur in our program when we are issuing concurrent writes to the same memory location
- The exact ordering of instructions may result in our program not updating variables as a serial version of our program initially would have

Time	Thread 1	Thread 2
1	(0) Old ← histo[x]	
2	(1) New ← Old + 1	
3		(0) Old ← histo[x]
4	(1) histo[x] ← New	
5		(1) New ← Old + 1
6		(1) histo[x] ← New

Overview

- ❑ Histograms
- ❑ Parallel Histograms
- ❑ **Simple histogram Kernel**
- ❑ Anatomy of Atomic Updating in Cuda
- ❑ Improved Histogram Kernels (Privatization, Shared Memory, Coarsening, Aggregation)

Atomic Additions

As is done within OpenMP and pthreads, we can make use of an atomic addition to ensure that our writes will occur sequentially when multiple threads attempt to write to the same location (this eliminates the race condition)

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

Histogram Kernel

```
01  __global__ void histo_kernel(char *data, unsigned int length,
                                 unsigned int *histo)  {
02      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03      if (i < length) {
04          int alphabet_position = data[i] - 'a';
05          if (alphabet_position >= 0 && alpha_position < 26) {
06              atomicAdd(&(histo[alphabet_position/4]), 1);
07          }
08      }
09  }
```

Histogram Kernel

How should we launch this kernel?
Are we coalescing?

```
01  __global__ void histo_kernel(char *data, unsigned int length,
                                 unsigned int *histo)  {
02      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03      if (i < length) {
04          int alphabet_position = data[i] - 'a';
05          if (alphabet_position >= 0 && alpha_position < 26) {
06              atomicAdd(&(histo[alphabet_position/4]), 1);
07          }
08      }
09  }
```

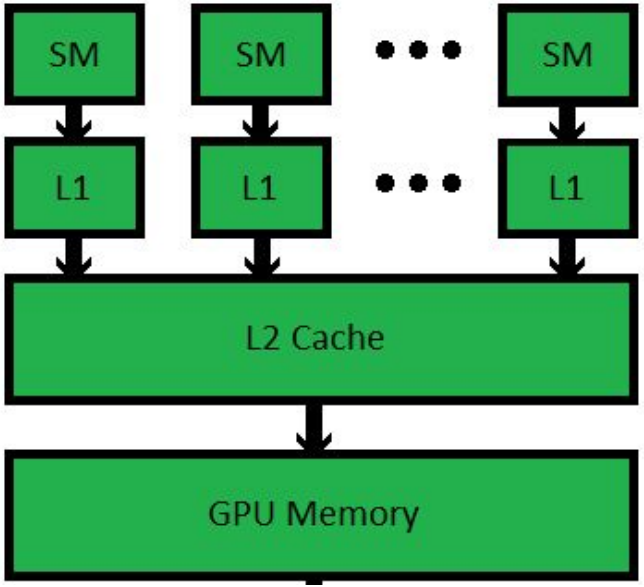
Overview

- ❑ Histograms
- ❑ Parallel Histograms
- ❑ Simple histogram Kernel
- ❑ **Anatomy of Atomic Updating in Cuda**
- ❑ Improved Histogram Kernels (Privatization, Shared Memory, Coarsening, Aggregation)

How do atomic additions work in practice?

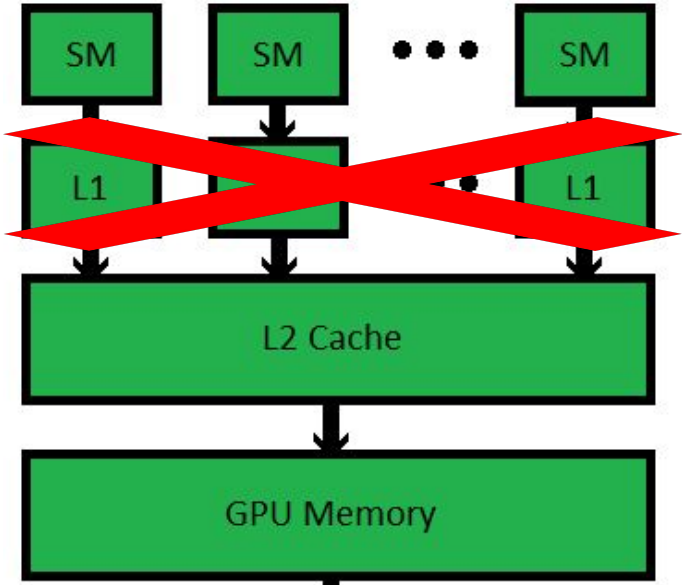
Anatomy of Atomic Additions in Cuda

- All threads within a warp execute the same instruction for the atomicAdd
- Unlike non-atomic memory transactions, atomic memory transactions skip the L1 cache altogether and go straight to the L2 global cache. Why?
  -



Anatomy of Atomic Additions in Cuda

- All threads within a warp execute the same instruction for the atomicAdd
- Unlike non-atomic memory transactions, atomic memory transactions skip the L1 cache altogether and go straight to the L2 global cache. Why?
  - Global variables are shared. We need the transactions to them to be sequential across all threads in the grid, not just a given threadblock



Anatomy of Atomic Additions in Cuda

- Once at the L2 cache, the actual updates take place for the threads within that warp
- If multiple threads within a warp execute an update to the same location, they are *serialized* by the L2 memory controller
- These *serialized* updates are pipelined to ensure there are still some speedups

```
// 32 threads in a warp all execute:
atomicAdd(&shared_counter, 1);
```

Warp Pipeline after reaching L2 controller

[issue]
[read]
[add]
[write]
[complete]

Pipeline of execution for this warp after reaching L2 memory controller (given that all writes are to the same ocation → *shared_counter*)

Thread 1: [–pipeline–]
Thread 0: [–pipeline–]
Thread 2: [–pipeline–]
...

// The operations overlap in the pipeline

Anatomy of Atomic Additions in Cuda

- The warp only continues execution after the final thread finishes executing its pipeline (and the L2 memory controller sends a signal the warp has completed its updates)
- The ordering of sequential updates is unknown

```
// All 32 threads in a warp execute the following
// Assume only one 32 thread block is dispatched
atomicAdd(&counter, threadIdx.x);

// Possible execution orders:
// Run 1: threads execute in order 0,1,2,3,…,31
// Run 2: threads execute in order 5,2,31,0,17,…
// Run 3: threads execute in some other order
// All are valid

// Final result is always: counter += (0+1+2+…+31) = 496
// But intermediate states are unpredictable
```

What about threads in the same threadblock, but different warps? What about between different threadblocks?

Anatomy of Atomic Additions in Cuda

- As within warps, transactions to the same memory locations will always be serial in threads from different warps, and the ordering remains unknown
- In total, we cannot know which threads in a warp, warps in a block, or blocks in a grid will execute their transactions first
- The exact order of atomic additions by individual thread is unknown

```
atomicAdd(&global_counter, 1);
```

Pipeline stage for the entire warp:

[Issue] →
[L2 request] →
[L2 queue] →
[read] →
[modify] →
[write] →
[respond]

Multiple atomics of different warps can be in flight concurrently:

Warp 1, Block 0: [—pipeline—]
Warp 1, Block 2: [—pipeline—]
Warp 0, Block 0: [—pipeline—]

Anatomy of Atomic Additions in Cuda

- Different warps ***do not implicitly synchronize*** their updates
- Likewise, warps across different threadblocks ***do not synchronize their updates***
- We can synchronize the value within a threadblock with __synchthreads()
- There ***is no way to synchronize*** the value between threadblocks at a given point during kernel execution

```
// BAD - race condition within block
atomicAdd(&counter, 1);
if (counter > 50) { ... }  // Undefined behavior!

// GOOD - synchronized within block
//             - Not synchronized between blocks!!
atomicAdd(&counter, 1);
__syncthreads();
if (counter > 50) { ... }  // Now safe within block
```

Utilizing this Anatomy in Practice

In general, atomic operations are useful, provided your kernel has the following properties

- ❑ The variable being atomically updated is write-only, and the operations performed on this variable are independent of the ordering
- ❑ **Or**, if you are reading the atomic variable during program execution:
  - Threads in different threadblocks are accessing different regions
  - **And** you are synchronizing after issuing your writes to ensure coherence within each threadblock

Overview

- ❑ Histograms
- ❑ Parallel Histograms
- ❑ Simple histogram Kernel
- ❑ Anatomy of Atomic Updating in Cuda
- ❑ **Improved Histogram Kernels (Privatization, Shared Memory, Coarsening, Aggregation)**

Revisiting the Original Kernel

Why is this kernel not very fast?

```
01  __global__ void histo_kernel(char *data, unsigned int length,
                                 unsigned int *histo)  {
02      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03      if (i < length) {
04          int alphabet_position = data[i] - 'a';
05          if (alphabet_position >= 0 && alpha_position < 26) {
06              atomicAdd(&(histo[alphabet_position/4]), 1);
07          }
08      }
09  }
```

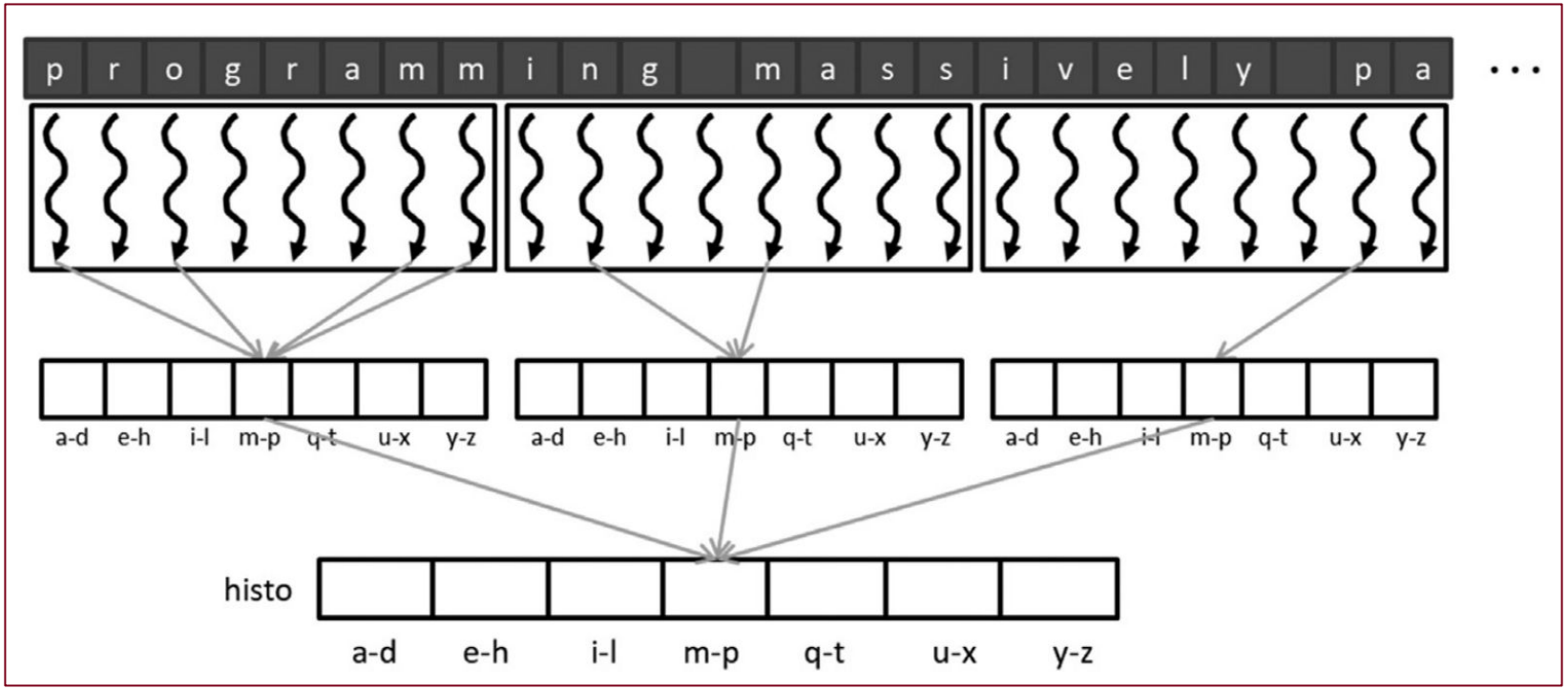
Revisiting the Original Kernel

Why is this kernel not very fast?
Excessive serialization leads to idle warps

```
01  __global__ void histo_kernel(char *data, unsigned int length,
                                 unsigned int *histo)  {
02      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03      if (i < length) {
04          int alphabet_position = data[i] - 'a';
05          if (alphabet_position >= 0 && alpha_position < 26) {
06              atomicAdd(&(histo[alphabet_position/4]), 1);
07          }
08      }
09  }
```

Privatization Kernel

Copy the *histo* array for each block of threads that we will spawn



Privatization Kernel

Copy the *histo* array for each block of threads that we will spawn

```
01  __global__ void histo_private_kernel(char *data, unsigned int length,
                                          unsigned int *histo) {
02    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03    if(i < length) {
04      int alphabet_position = data[i] - 'a';
05      if (alphabet_position >= 0 && alphabet_position < 26) {
06         atomicAdd(&(histo[blockIdx.x*NUM_BINS + alphabet_position/4]), 1);
07      }
08    }
09    if(blockIdx.x > 0) {
10      __syncthreads();
11      for(unsigned int bin=threadIdx.x; bin<NUM_BINS; bin += blockDim.x){
12         unsigned int binValue = histo[blockIdx.x*NUM_BINS + bin];
13         if(binValue > 0) {
14            atomicAdd(&(histo[bin]), binValue);
15         }
16      }
17    }
18  }
```

Privatization Kernel

Does this use memory coalescing?
Why is this advantageous?

```
01  __global__ void histo_private_kernel(char *data, unsigned int length,
                                         unsigned int *histo) {
02    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03    if(i < length) {
04      int alphabet_position = data[i] - 'a';
05      if (alphabet_position >= 0 && alphabet_position < 26) {
06         atomicAdd(&(histo[blockIdx.x*NUM_BINS + alphabet_position/4]), 1);
07      }
08    }
09    if(blockIdx.x > 0) {
10      __syncthreads();
11      for(unsigned int bin=threadIdx.x; bin<NUM_BINS; bin += blockDim.x){
12         unsigned int binValue = histo[blockIdx.x*NUM_BINS + bin];
13         if(binValue > 0) {
14            atomicAdd(&(histo[bin]), binValue);
15         }
16      }
17    }
18  }
```

Privatization Kernel

Do all threads participate in updating the full histogram? Where is this full histogram stored?

```
01  __global__ void histo_private_kernel(char *data, unsigned int length,
                                         unsigned int *histo) {
02    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03    if(i < length) {
04      int alphabet_position = data[i] - 'a';
05      if (alphabet_position >= 0 && alphabet_position < 26) {
06         atomicAdd(&(histo[blockIdx.x*NUM_BINS + alphabet_position/4]), 1);
07      }
08    }
09    if(blockIdx.x > 0) {
10      __syncthreads();
11      for(unsigned int bin=threadIdx.x; bin<NUM_BINS; bin += blockDim.x){
12         unsigned int binValue = histo[blockIdx.x*NUM_BINS + bin];
13         if(binValue > 0) {
14            atomicAdd(&(histo[bin]), binValue);
15         }
16      }
17    }
18  }
```

Shared Memory Kernel

```
01 __global__ void histo_private_kernel(char* data, unsigned int length,
                                        unsigned int* histo) {
02       // Initialize privatized bins
03   __shared__ unsigned int histo_s[NUM_BINS];
04   for(unsigned int bin = threadIdx.x; bin< NUM_BINS; bin += blockDim.x) {
05      histo_s[bin] = 0u;
06   }
07   __syncthreads();
08      // Histogram
09   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
10    if(i < length) {
11      int alphabet_position = data[i] - 'a';
12      if(alphabet_position >= 0 && alphabet_position < 26) {
13         atomicAdd(&(histo_s[alphabet_position/4]), 1);
14      }
15    }
16    __syncthreads();
17      // Commit to global memory
18    for(unsigned int bin=threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
19          unsigned int binValue = histo_s[bin];
20          if(binValue > 0) {
21            atomicAdd(&(histo[bin]), binValue);
22          }
23      }
24  }
```

Shared Memory Kernel

Do all threads update the shared memory values?

```
01 __global__ void histo_private_kernel(char* data, unsigned int length,
                                        unsigned int* histo) {
02       // Initialize privatized bins
03   __shared__ unsigned int histo_s[NUM_BINS];
04   for(unsigned int bin = threadIdx.x; bin< NUM_BINS; bin += blockDim.x) {
05      histo_s[bin] = 0u;
06   }
07   __syncthreads();
08      // Histogram
09   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
10    if(i < length) {
11      int alphabet_position = data[i] - 'a';
12      if(alphabet_position >= 0 && alphabet_position < 26) {
13         atomicAdd(&(histo_s[alphabet_position/4]), 1);
14      }
15    }
16    __syncthreads();
17      // Commit to global memory
18    for(unsigned int bin=threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
19          unsigned int binValue = histo_s[bin];
20          if(binValue > 0) {
21            atomicAdd(&(histo[bin]), binValue);
22          }
23      }
24  }
```

Shared Memory Kernel

Why is updating the shared memory locations better?

```
01 __global__ void histo_private_kernel(char* data, unsigned int length,
                                        unsigned int* histo) {
02       // Initialize privatized bins
03   __shared__ unsigned int histo_s[NUM_BINS];
04   for(unsigned int bin = threadIdx.x; bin< NUM_BINS; bin += blockDim.x) {
05      histo_s[bin] = 0u;
06   }
07   __syncthreads();
08      // Histogram
09   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
10    if(i < length) {
11      int alphabet_position = data[i] - 'a';
12      if(alphabet_position >= 0 && alphabet_position < 26) {
13         atomicAdd(&(histo_s[alphabet_position/4]), 1);
14      }
15    }
16    __syncthreads();
17      // Commit to global memory
18    for(unsigned int bin=threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
19          unsigned int binValue = histo_s[bin];
20          if(binValue > 0) {
21            atomicAdd(&(histo[bin]), binValue);
22          }
23      }
24  }
```

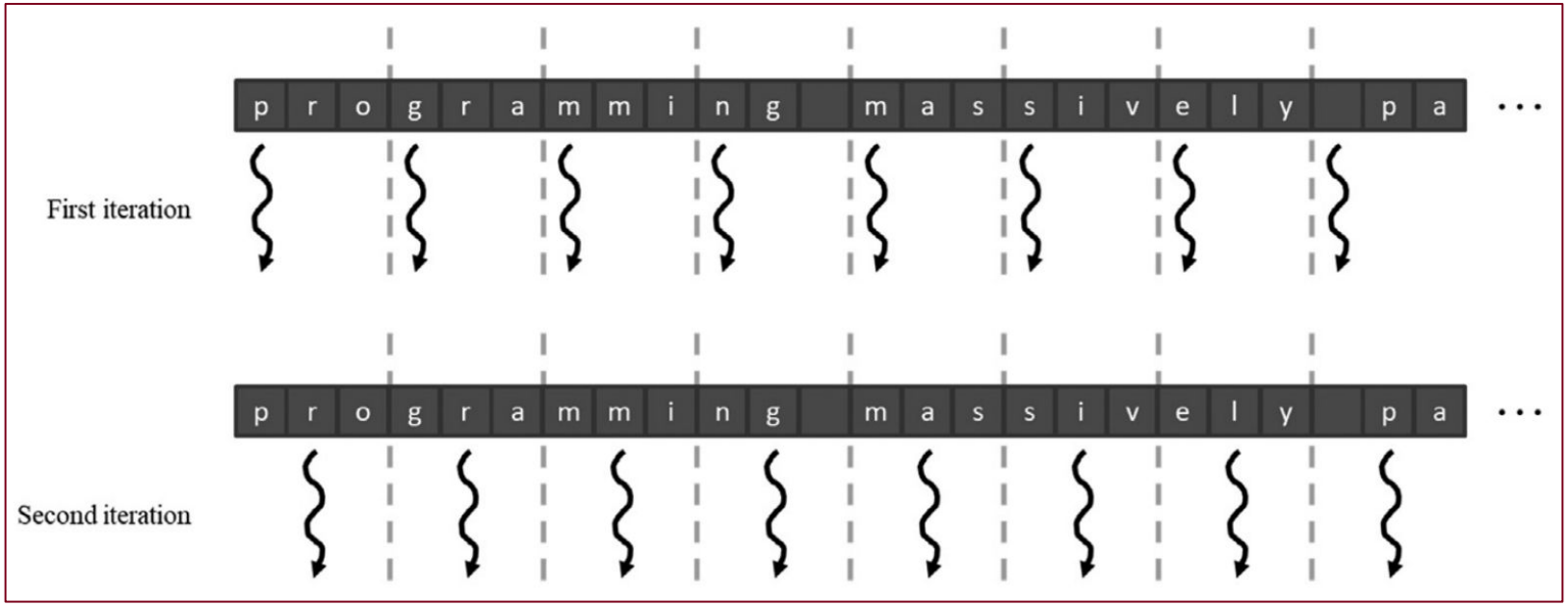
Shared Memory Kernel

Do all threads write to global memory?

```
01 __global__ void histo_private_kernel(char* data, unsigned int length,
                                        unsigned int* histo) {
02       // Initialize privatized bins
03   __shared__ unsigned int histo_s[NUM_BINS];
04   for(unsigned int bin = threadIdx.x; bin< NUM_BINS; bin += blockDim.x) {
05      histo_s[bin] = 0u;
06   }
07   __syncthreads();
08      // Histogram
09   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
10    if(i < length) {
11      int alphabet_position = data[i] - 'a';
12      if(alphabet_position >= 0 && alphabet_position < 26) {
13         atomicAdd(&(histo_s[alphabet_position/4]), 1);
14      }
15    }
16    __syncthreads();
17      // Commit to global memory
18    for(unsigned int bin=threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
19          unsigned int binValue = histo_s[bin];
20          if(binValue > 0) {
21            atomicAdd(&(histo[bin]), binValue);
22          }
23      }
24  }
```

Coarsening Kernel (No 1)

We can add coarsening by giving each thread a set of input elements to work on which are sequential



Coarsening Kernel (No 1)

```
01  __global__ void histo_private_kernel(char* data, unsigned int length,
                                         unsigned int* histo) {
02      // Initialize privatized bins
03    __shared__ unsigned int histo_s[NUM_BINS];
04    for(unsigned int bin = threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
05        histo_s[binIdx] = 0u;
06    }
07     __syncthreads();
08      // Histogram
09    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
10    for(unsigned int i=tid*CFACTOR; i<min((tid+1)*CFACTOR, length); ++i) {
11        int alphabet_position = data[i] - 'a';
12        if(alphabet_position >= 0 && alphabet_position < 26) {
13              atomicAdd(&(histo_s[alphabet_position/4]), 1);
14        }
15    }
16    __syncthreads();
17      // Commit to global memory
18    for(unsigned int bin = threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
19        unsigned int binValue = histo_s[binIdx];
20        if(binValue > 0) {
21          atomicAdd(&(histo[binIdx]), binValue);
22        }
23    }
24  }
```

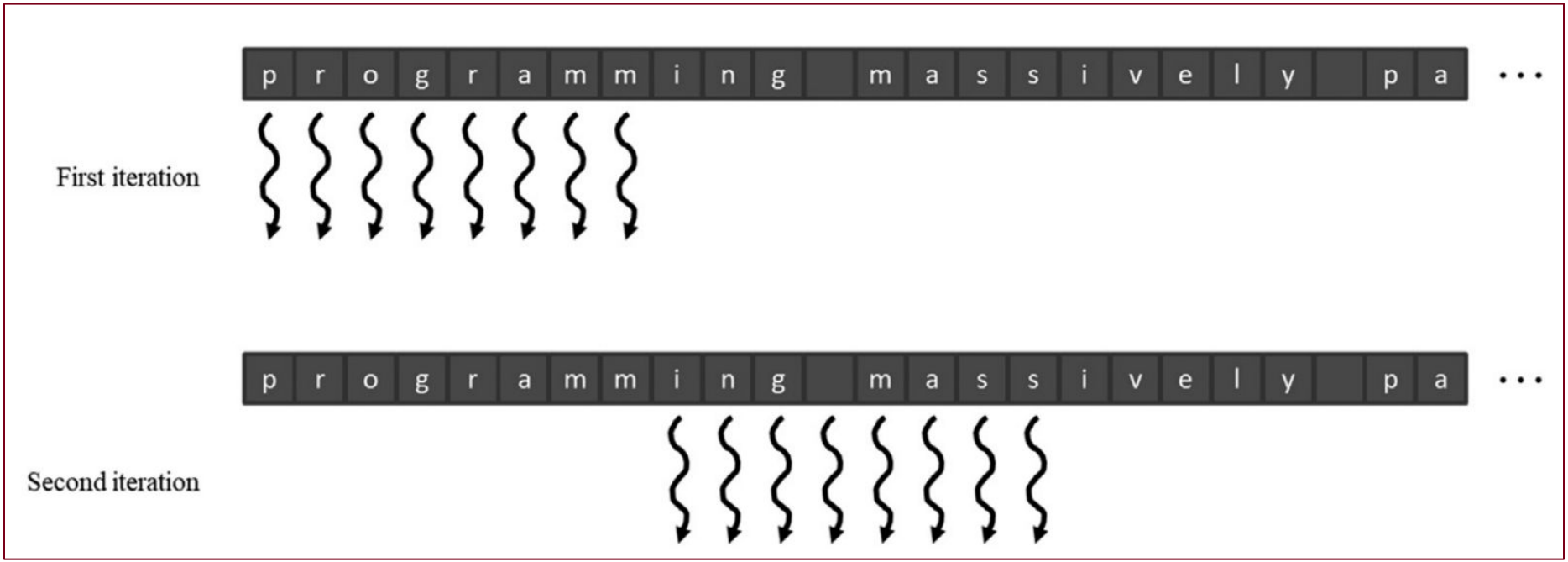
Coarsening Kernel (No 1)

Why is this a poor coarsening implementation?

```
01  __global__ void histo_private_kernel(char* data, unsigned int length,
                                         unsigned int* histo) {
02      // Initialize privatized bins
03    __shared__ unsigned int histo_s[NUM_BINS];
04    for(unsigned int bin = threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
05        histo_s[binIdx] = 0u;
06    }
07     __syncthreads();
08      // Histogram
09    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
10    for(unsigned int i=tid*CFACTOR; i<min((tid+1)*CFACTOR, length); ++i) {
11        int alphabet_position = data[i] - 'a';
12        if(alphabet_position >= 0 && alphabet_position < 26) {
13              atomicAdd(&(histo_s[alphabet_position/4]), 1);
14        }
15    }
16    __syncthreads();
17      // Commit to global memory
18    for(unsigned int bin = threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
19        unsigned int binValue = histo_s[binIdx];
20        if(binValue > 0) {
21          atomicAdd(&(histo[binIdx]), binValue);
22        }
23    }
24  }
```

Coarsening Kernel (No 2)

We now add a coalesced version of coarsening that allows for more efficient memory loading.



Coarsening Kernel (No 2)

```
01  __global__ void histo_private_kernel(char* data, unsigned int length,
                                         unsigned int* histo){
02      // Initialize privatized bins
03    __shared__ unsigned int histo_s[NUM_BINS];
04    for(unsigned int bin=threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
05          histo_s[binIdx] = 0u;
06    }
07    __syncthreads();
08      // Histogram
09    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
10    for(unsigned int i = tid; i < length; i += blockDim.x*gridDim.x) {
11       int alphabet_position = data[i] - 'a';
12       if(alphabet position >= 0 && alphabet position < 26) {
13           atomicAdd(&(histo_s[alphabet_position/4]), 1);
14       }
15    }
16    __syncthreads();
17      // Commit to global memory
18    for(unsigned int bin = threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
19       unsigned int binValue = histo_s[binIdx];
20       if(binValue > 0) {
21          atomicAdd(&(histo[binIdx]), binValue);
22       }
23    }
24  }
```

Aggregation Kernel

```
01  __global__ void histo_private_kernel(char* data, unsigned int length,
                                         unsigned int* histo){
02      // Initialize privatized bins
03    __shared__ unsigned int histo_s[NUM_BINS];
04    for(unsigned int bin = threadIdx.x; bin < NUM_BINS; bin += blockDim.x){
05        histo_s[bin] = 0u;
06    }
07    __syncthreads();
08      // Histogram
09    unsigned int accumulator = 0;
10    int prevBinIdx = -1;
11    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
12    for(unsigned int i = tid; i < length; i += blockDim.x*gridDim.x) {
13       int alphabet_position = data[i] - 'a';
14       if(alphabet_position >= 0 && alphabet_position < 26) {
15          int bin = alphabet_position/4;
16          if(bin == prevBinIdx) {
17               ++accumulator;
18          } else {
19               if(accumulator > 0) {
20                  atomicAdd(&(histo_s[prevBinIdx]), accumulator);
21               }
22               accumulator = 1;
23               prevBinIdx = bin;
24          }
25        }
26     }
27     if(accumulator > 0) {
28          atomicAdd(&(histo_s[prevBinIdx]), accumulator);
29     }
30     __syncthreads();
31      // Commit to global memory
32     for(unsigned int bin = threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
33          unsigned int binValue = histo_s[bin];
34          if(binValue > 0) {
35            atomicAdd(&(histo[bin]), binValue);
36          }
37     }
38  }
```

Aggregation Kernel

When is this kernel more useful?

```
01  __global__ void histo_private_kernel(char* data, unsigned int length,
                                         unsigned int* histo){
02      // Initialize privatized bins
03    __shared__ unsigned int histo_s[NUM_BINS];
04    for(unsigned int bin = threadIdx.x; bin < NUM_BINS; bin += blockDim.x){
05        histo_s[bin] = 0u;
06    }
07    __syncthreads();
08      // Histogram
09    unsigned int accumulator = 0;
10    int prevBinIdx = -1;
11    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
12    for(unsigned int i = tid; i < length; i += blockDim.x*gridDim.x) {
13       int alphabet_position = data[i] - 'a';
14       if(alphabet_position >= 0 && alphabet_position < 26) {
15          int bin = alphabet_position/4;
16          if(bin == prevBinIdx) {
17               ++accumulator;
18          } else {
19               if(accumulator > 0) {
20                  atomicAdd(&(histo_s[prevBinIdx]), accumulator);
21               }
22               accumulator = 1;
23               prevBinIdx = bin;
24          }
25        }
26     }
27     if(accumulator > 0) {
28          atomicAdd(&(histo_s[prevBinIdx]), accumulator);
29     }
30     __syncthreads();
31      // Commit to global memory
32     for(unsigned int bin = threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
33          unsigned int binValue = histo_s[bin];
34          if(binValue > 0) {
35            atomicAdd(&(histo[bin]), binValue);
36          }
37     }
38  }
```

Aggregation Kernel

When is this kernel more useful? When there are a large number of entries in *data* which are the same.

```
01  __global__ void histo_private_kernel(char* data, unsigned int length,
                                         unsigned int* histo){
02      // Initialize privatized bins
03    __shared__ unsigned int histo_s[NUM_BINS];
04    for(unsigned int bin = threadIdx.x; bin < NUM_BINS; bin += blockDim.x){
05        histo_s[bin] = 0u;
06    }
07    __syncthreads();
08      // Histogram
09    unsigned int accumulator = 0;
10    int prevBinIdx = -1;
11    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
12    for(unsigned int i = tid; i < length; i += blockDim.x*gridDim.x) {
13       int alphabet_position = data[i] - 'a';
14       if(alphabet_position >= 0 && alphabet_position < 26) {
15          int bin = alphabet_position/4;
16          if(bin == prevBinIdx) {
17               ++accumulator;
18          } else {
19               if(accumulator > 0) {
20                  atomicAdd(&(histo_s[prevBinIdx]), accumulator);
21               }
22               accumulator = 1;
23               prevBinIdx = bin;
24          }
25        }
26     }
27     if(accumulator > 0) {
28          atomicAdd(&(histo_s[prevBinIdx]), accumulator);
29     }
30     __syncthreads();
31      // Commit to global memory
32     for(unsigned int bin = threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
33          unsigned int binValue = histo_s[bin];
34          if(binValue > 0) {
35            atomicAdd(&(histo[bin]), binValue);
36          }
37     }
38  }
```