CHAPTER 9

Parallel histogram
An introduction to atomic operations
and privatization

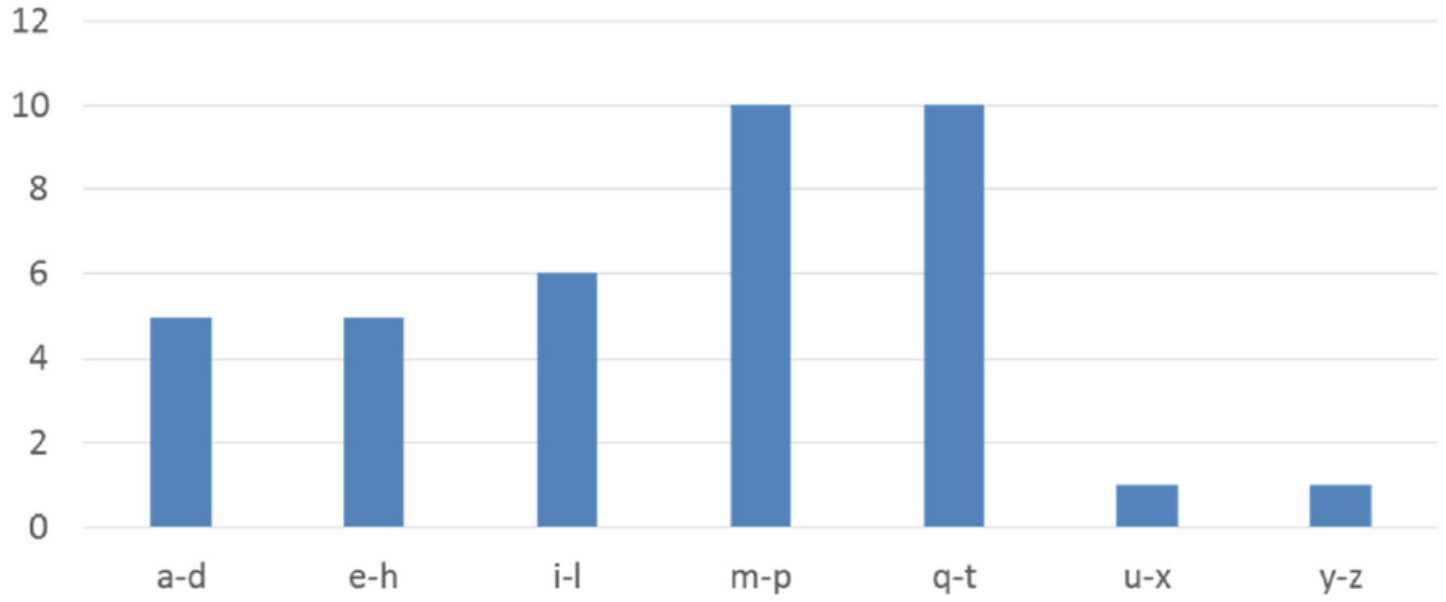

FIGURE 9.1

A histogram representation of the phrase "programming massively parallel processors."

```
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  }
```

FIGURE 9.2

A simple C function for calculating histogram for an input text string.

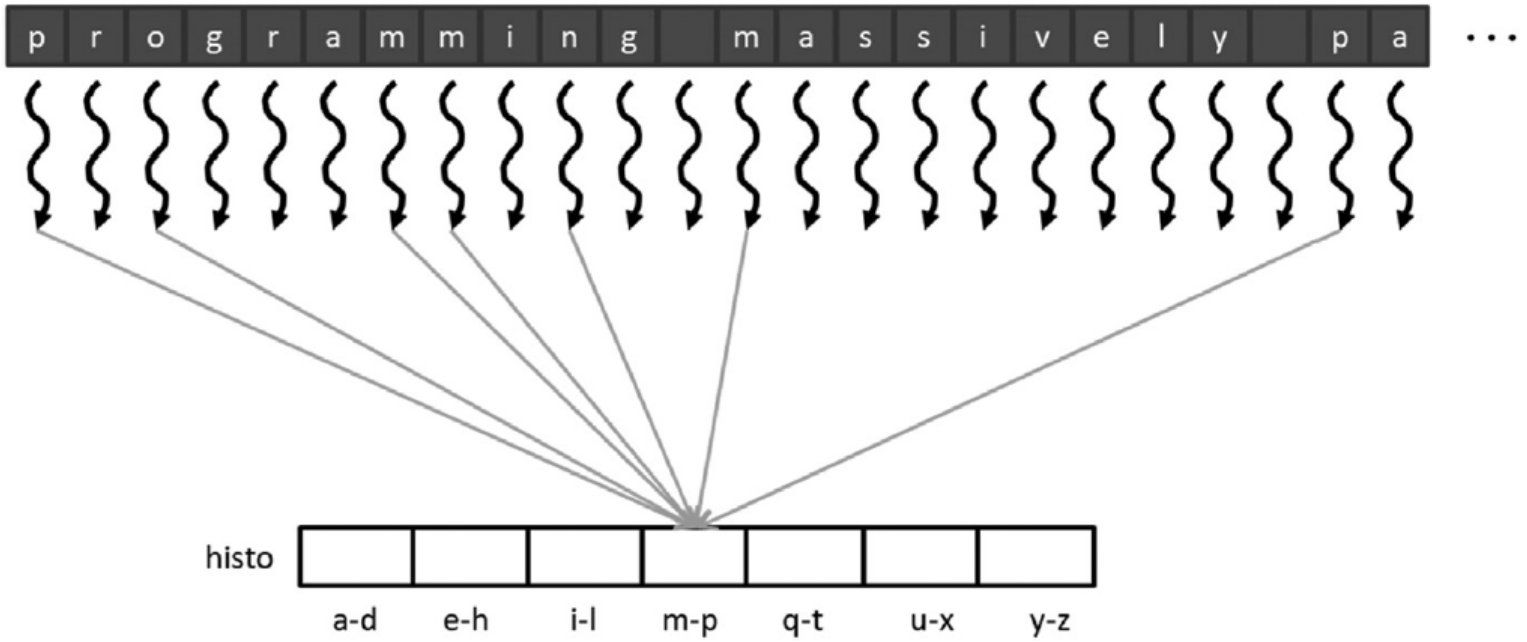

FIGURE 9.3

Basic parallelization of a histogram.

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

(A)

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

(B)

FIGURE 9.4

Race condition in updating a `histo` array element: (A) One possible interleaving of instructions; (B) Another possible interleaving of instructions.

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

(A)

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	

(B)

FIGURE 9.5

Race condition scenarios in which thread 2 runs ahead of thread 1: (A) One possible interleaving of instructions; (B) Another possible interleaving of instructions.

```
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  }
```

FIGURE 9.6

A CUDA kernel for calculation histogram.

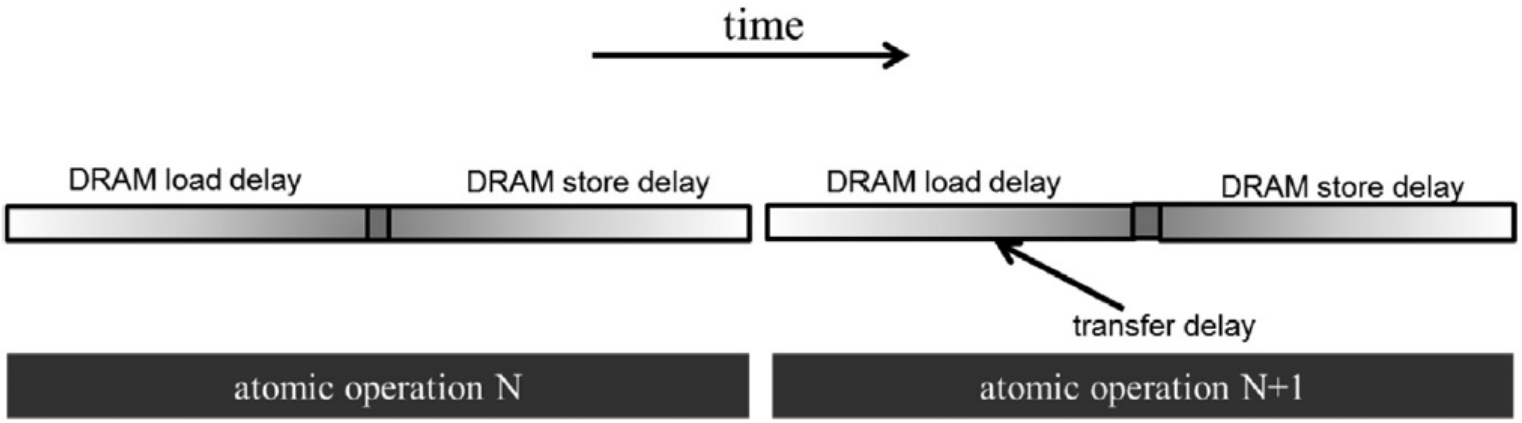

FIGURE 9.7

The throughput of an atomic operation is determined by the memory access latency.

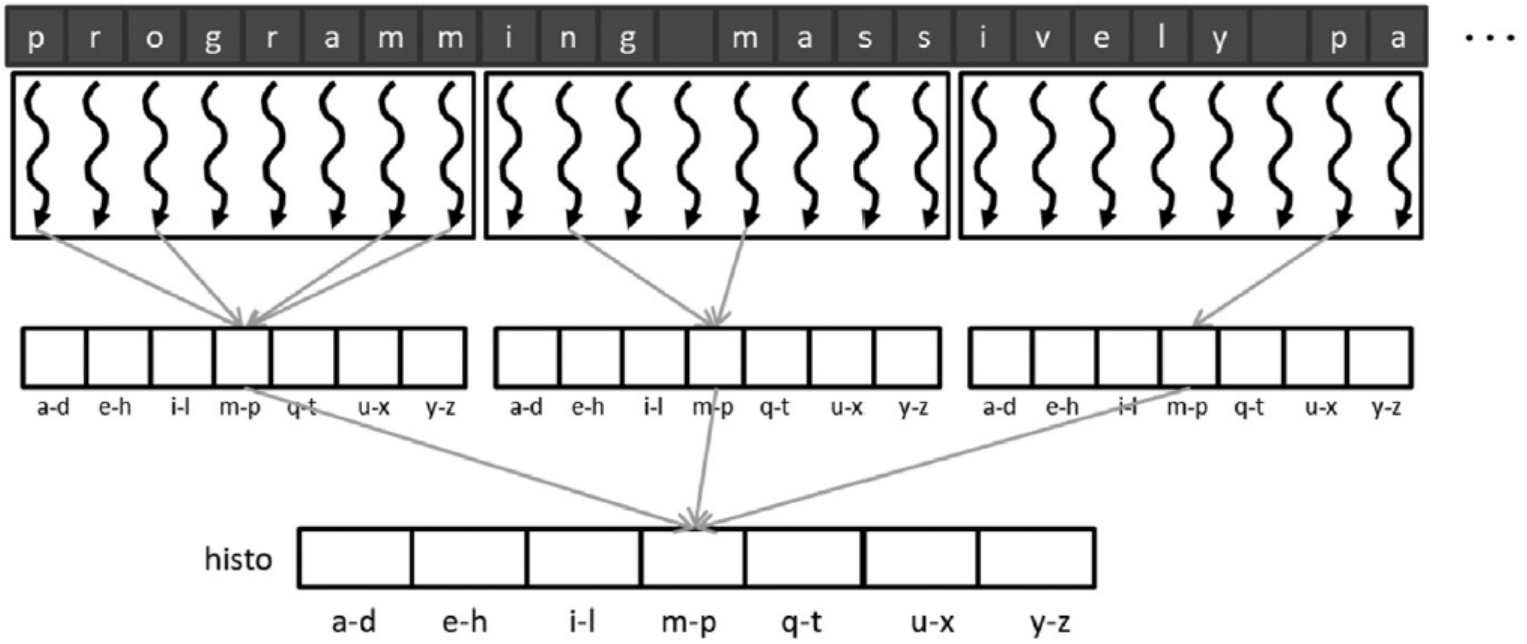

FIGURE 9.8

Private copies of histogram reduce contention of atomic operations.

```
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  }
```

FIGURE 9.9

Histogram kernel with private versions in global memory for thread blocks.

```
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  }
```

FIGURE 9.10

A privatized text histogram kernel using the shared memory.

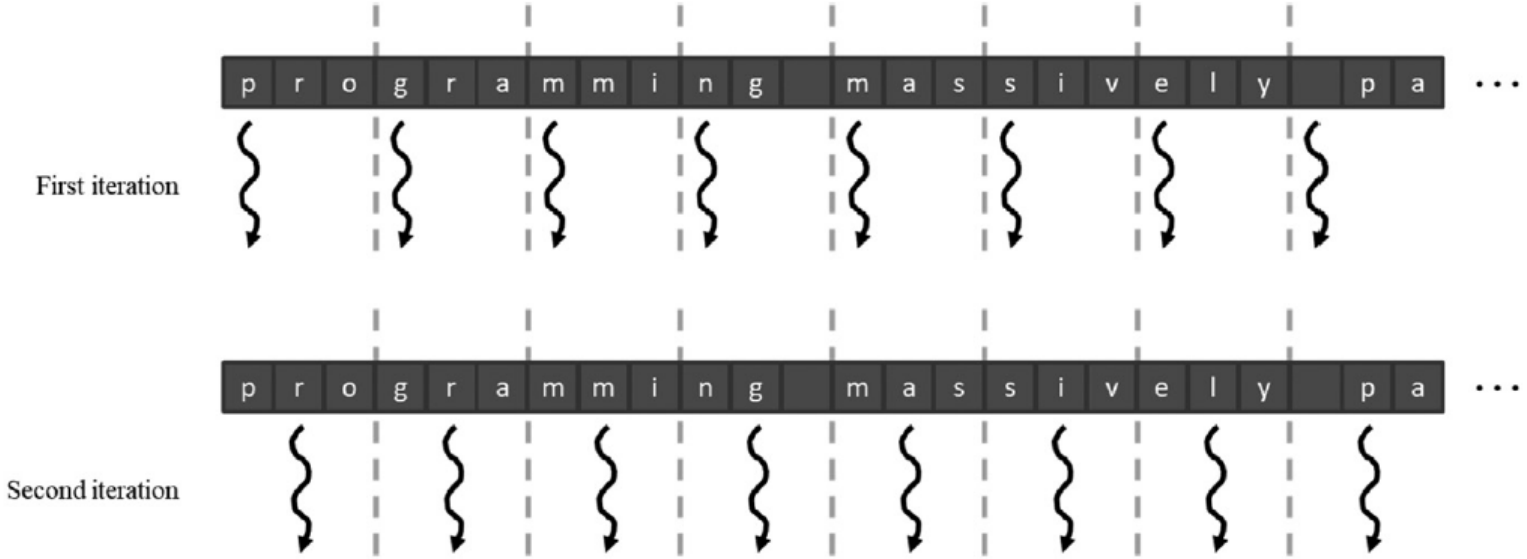

FIGURE 9.11

Contiguous partitioning of input elements.

```
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  }
```

FIGURE 9.12

Histogram kernel with coarsening using contiguous partitioning.

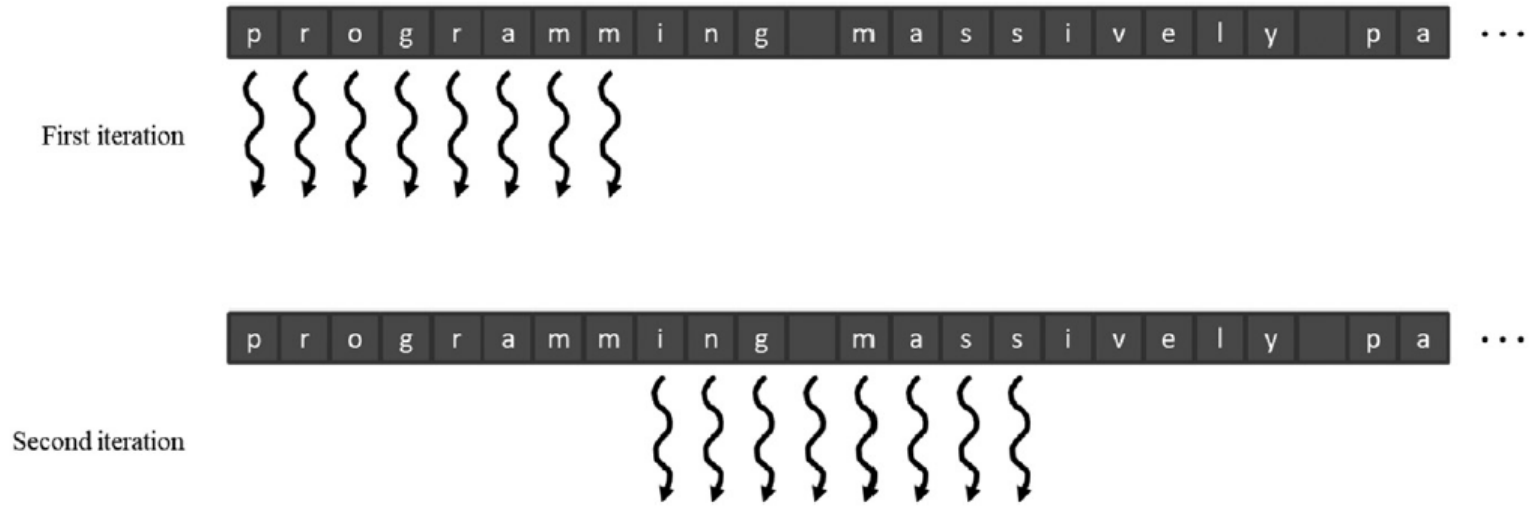

FIGURE 9.13

Interleaved partitioning of input elements.

```
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  }
```

FIGURE 9.14

Histogram kernel with coarsening using interleaved partitioning.

```
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  }
```

FIGURE 9.15

An aggregated text histogram kernel.

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

In-text figure 1

Intrinsic Functions

Modern processors often offer special instructions that either perform critical functionality (such as the atomic operations) or substantial performance enhancement (such as vector instructions). These instructions are typically exposed to the programmers as intrinsic functions, or simply instrinsics. From the programmer's perspective, these are library functions. However, they are treated in a special way by compilers; each such call is translated into the corresponding special instruction. There is typically no function call in the final code, just the special instructions in line with the user code. All major modern compilers, such as the GNU Compiler Collection (gcc), Intel C Compiler, and Clang/LLVM C Compiler support intrinsics.

In-text figure 2

```
histo[alphabet_position/4]++
```

In-text figure 3