Chapter-9
Parallel patterns—parallel histogram computation
An introduction to atomic operations and privatization

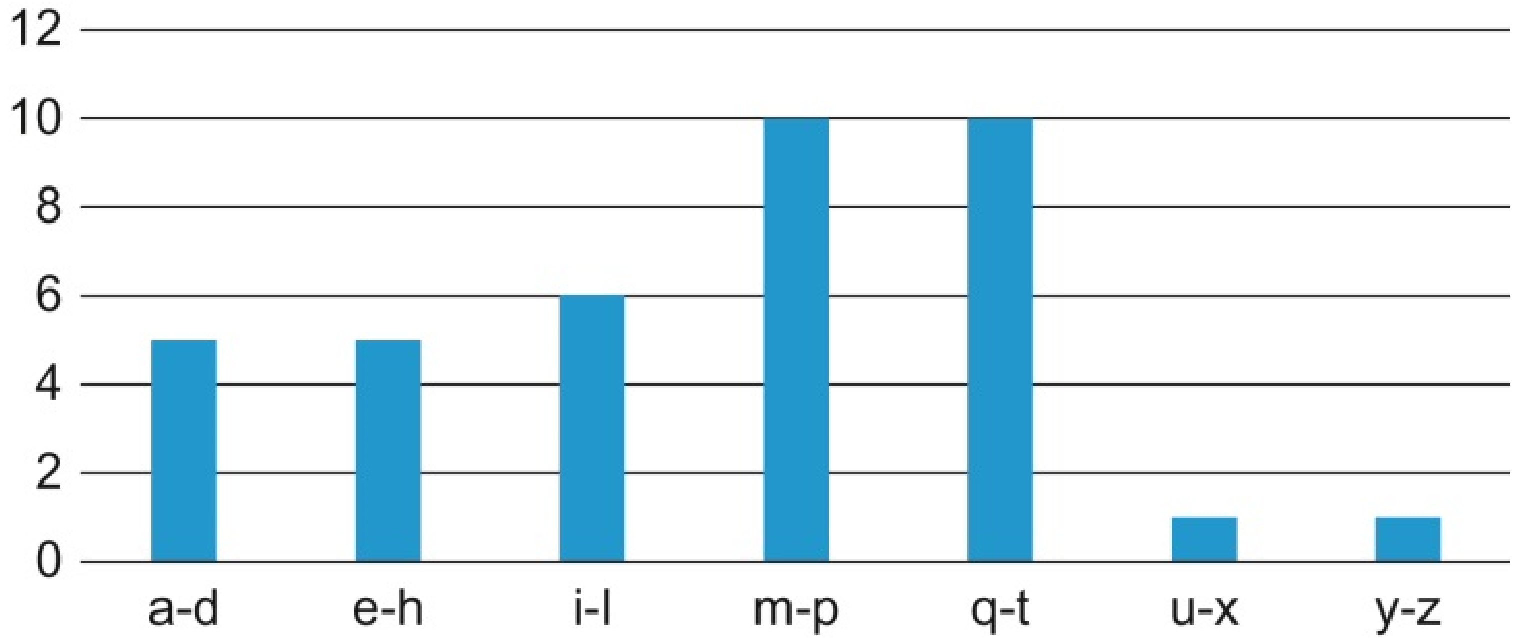

FIGURE 9.1: A histogram representation of “programming massively parallel processors.”

```
sequential_Histogram(char *data, int length, int *histo) {
   for (int i = 0; i < length; i++) {
       int alphabet_position = data[i] - 'a';
       if (alphabet_position >= 0 && alphabet_position < 26) {
            histo[alphabet_position/4]++
       }
   }
}
```

FIGURE 9.2: A simple C function for calculating histogram for an input text string.

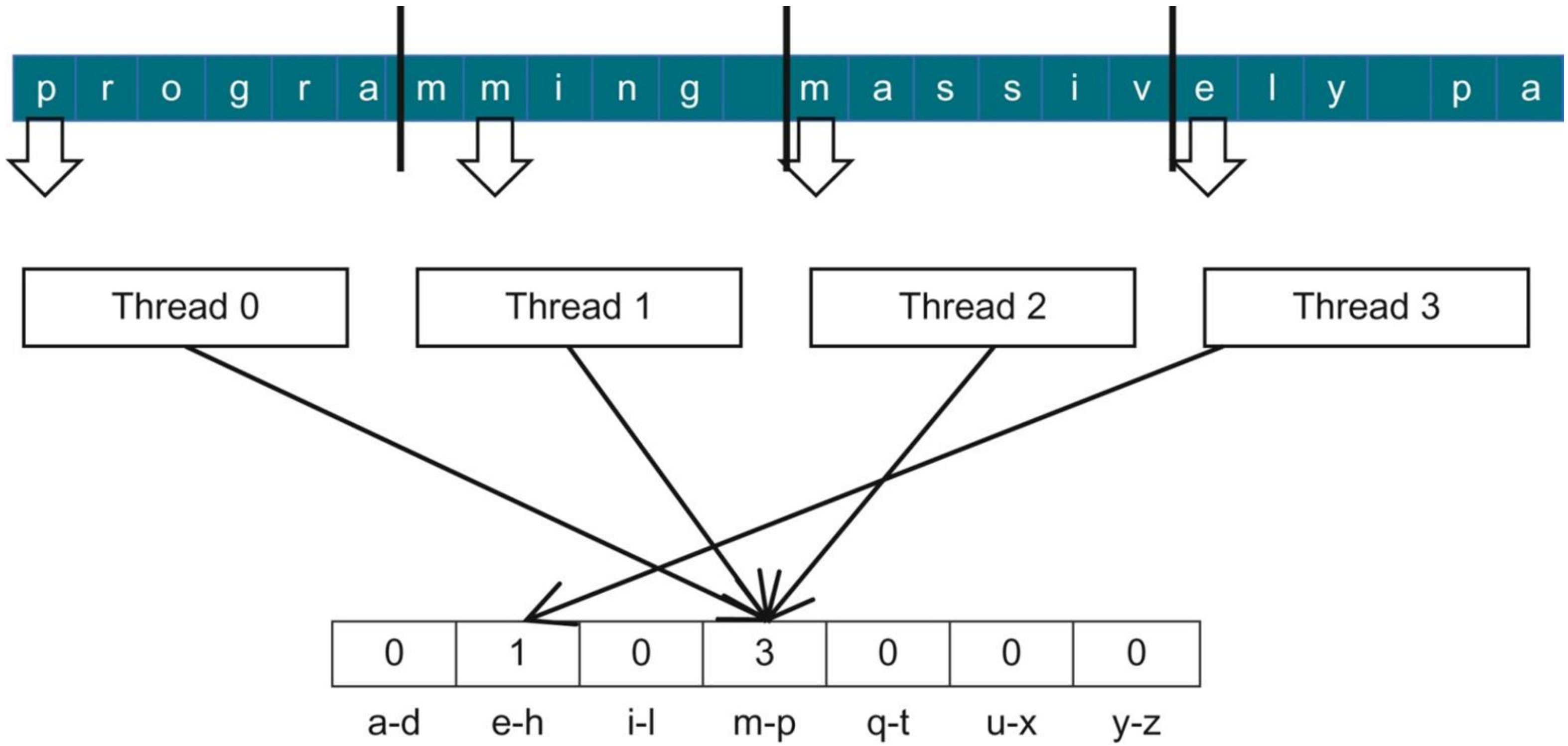

FIGURE 9.3: Strategy I for parallelizing histogram computation.

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.

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 where Thread 2 runs ahead of Thread 1.

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
1.   int i = threadIdx.x + blockIdx.x * blockDim.x;
2.   int section_size = (size-1)  / (blockDim.x * gridDim.x) +1;
3.   int start = i*section_size;

 // All threads handle blockDim.x * gridDim.x
  // consecutive elements
4.  for (k = 0; k < section_size; k++) {
5.    if (start+k < size) {
6.      int alphabet_position = buffer[start+k] – ‘a’;
7.      if (alphabet_position >= 0 && alpha_position < 26) atomicAdd(&(histo[alphabet_position/4]), 1);
      }
    }
}
```

FIGURE 9.6: A CUDA kernel for calculation histogram based on Strategy I.

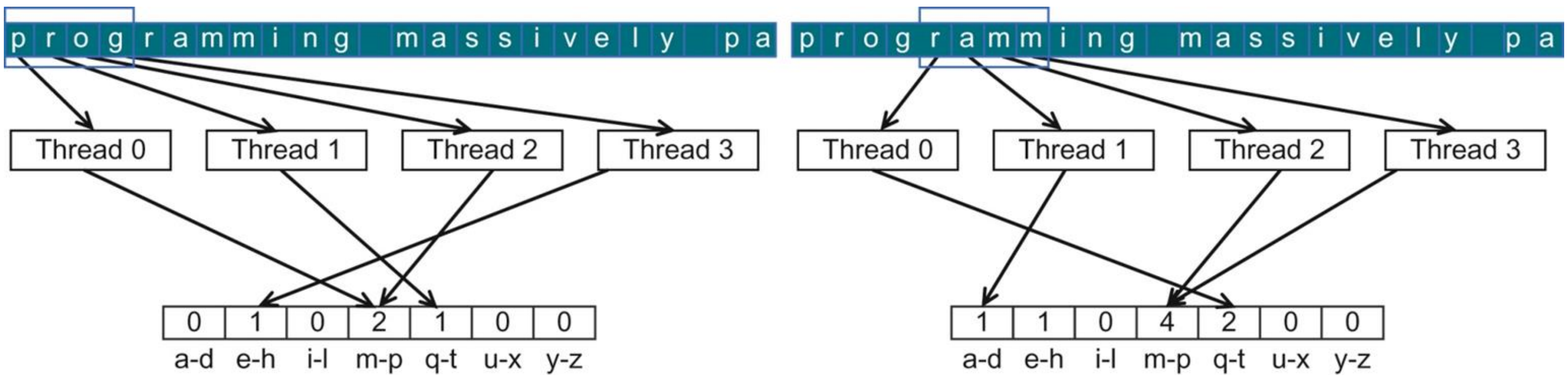

FIGURE 9.7: Desirable access pattern to the input buffer for memory coalescing—Strategy II.

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{

1.  unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;

// All threads handle blockDim.x * gridDim.x consecutive elements in each iteration
2.   for (unsigned int i = tid;  i < size; i += blockDim.x*gridDim.x ) {
3.       int alphabet_position = buffer[i] – 'a';
4.       if (alphabet_position >= 0 && alpha_position < 26) atomicAdd(&(histo[alphabet_position/4]), 1);
     }
}
```

FIGURE 9.8: A CUDA kernel for calculating histogram based on Strategy II.

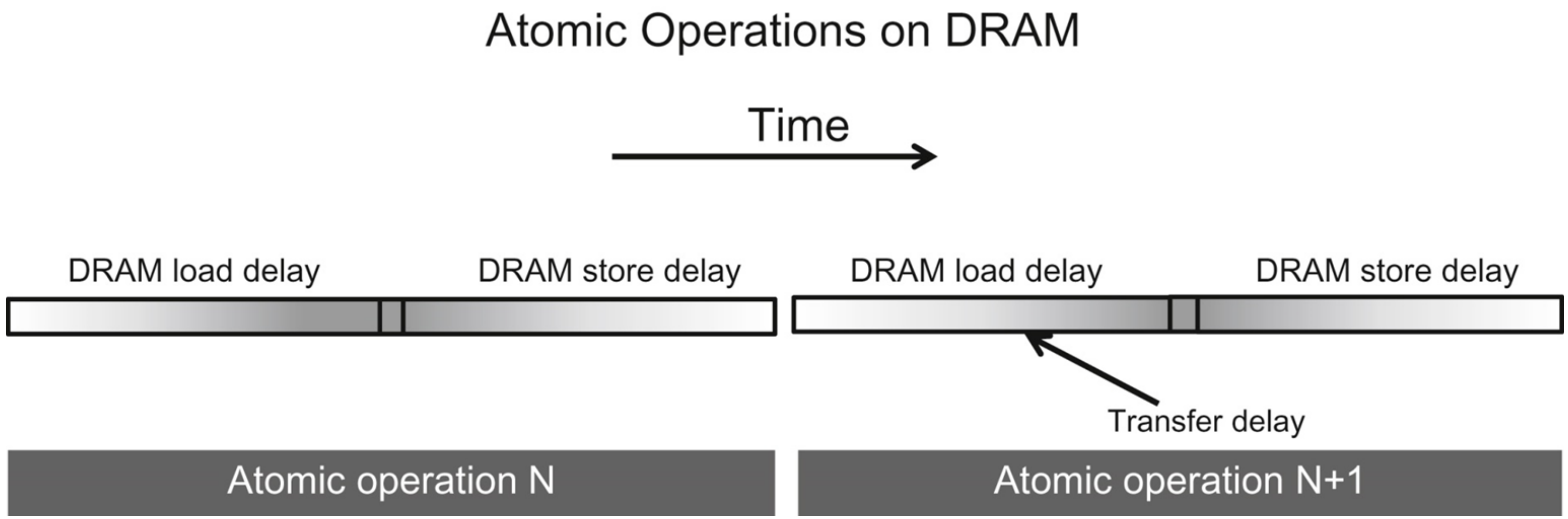

FIGURE 9.9: Throughput of atomic operation is determined by the memory access latency.

```
__global__ void histogram_privatized_kernel(unsigned char* input, unsigned int* bins,
   unsigned int num_elements, unsigned int num_bins) {

1.    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;

      // Privatized bins
2.    extern __shared__ unsigned int histo_s[];
3.    for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx +=blockDim.x) {
4.       histo_s[binIdx] = 0u;
      }
5.    __syncthreads();

      // Histogram
6.    For (unsigned int i = tid; i < num_elements; i += blockDim.x*gridDim.x) {
        int alphabet_position = buffer[i] – "a";
7.      if (alphabet_position >= 0 && alpha_position < 26) atomicAdd(&(histo_s[alphabet_position/4]), 1);
      }
8.    __syncthreads();

   // Commit to global memory
9.    for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {
10.      atomicAdd(&(histo[binIdx]), histo_s[binIdx]);
      }
}
```

FIGURE 9.10: A privatized text histogram kernel.

```
__global__ void histogram_privatized_kernel(unsigned char* input, unsigned int* bins,
  unsigned int num_elements, unsigned int num_bins) {

  unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;

  // Privatized bins
  extern __shared__ unsigned int histo_s[];
  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx +=blockDim.x) {
     histo_s[binIdx] = 0u;
  }
  __syncthreads();

  unsigned int prev_index = -1;
  unsigned int accumulator = 0;

  for(unsigned int i = tid; i < num_elements; i += blockDim.x*gridDim.x) {
    int alphabet_position = buffer[i] – "a";
    if (alphabet_position >= 0 && alpha_position < 26) {
          unsigned int curr_index = alphabet_position/4;
          if (curr_index != prev_index) {
             if (accumulator >= 0) atomicAdd(&(histo_s[alphabet_position/4]), accumulator);
             accumulator = 1;
             prev_index = curr_index;
          }
          else {
             accumulator++;
          }
      }
    }
  __syncthreads();

  // Commit to global memory
  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {
      atomicAdd(&(histo[binIdx]), histo_s[binIdx]);
  }
}
```

FIGURE 9.11: An aggregated text histogram kernel.