Lecture No.7: Histogram

Muhammad Osama Mahmoud, TA

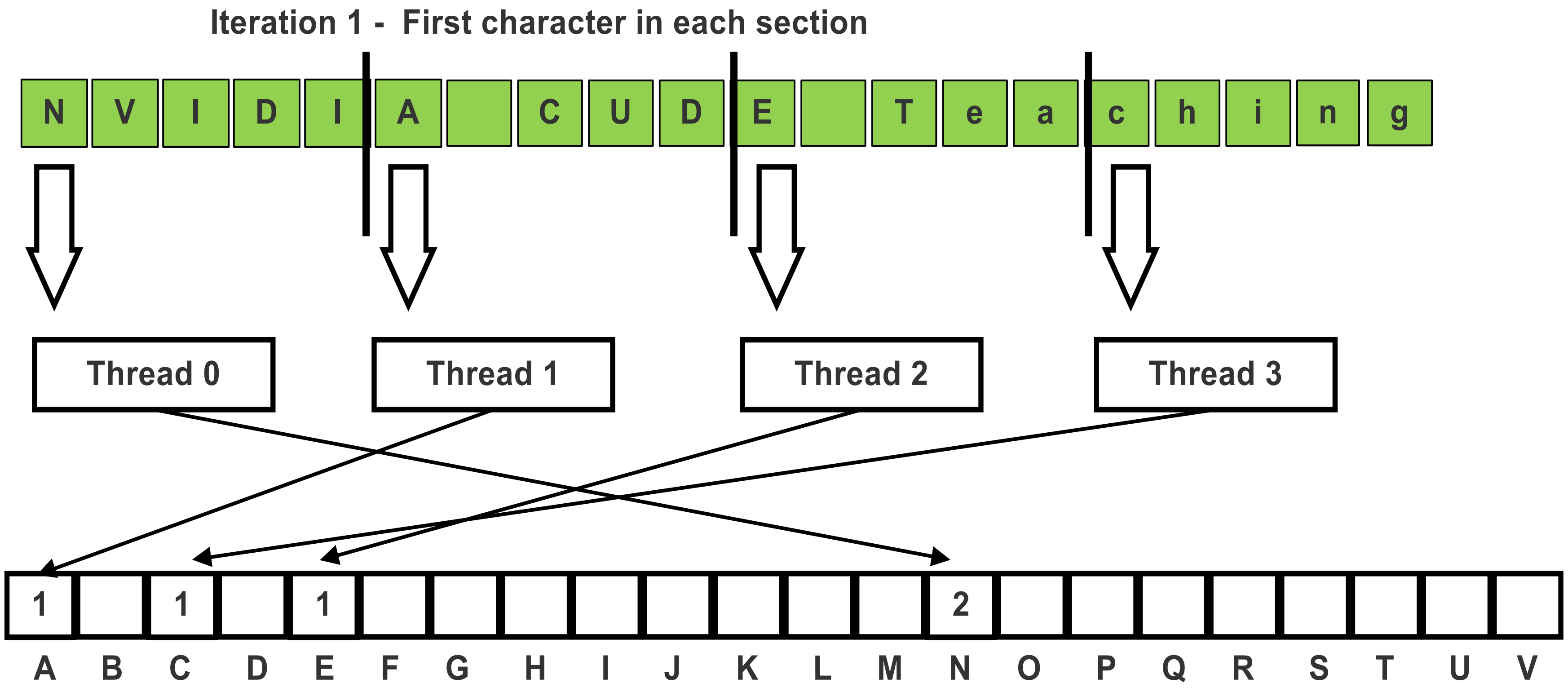
Histogram

- A method of extracting notable information and features from large data sets
- Applications:
  - Feature extraction in images
  - Fraud detection in credit card transaction
- A typical histogram is a representation of tabulated frequencies (count the occurrence of an input value)
- Example: compute the histogram for "NVIDIA CUDA Teaching Center "

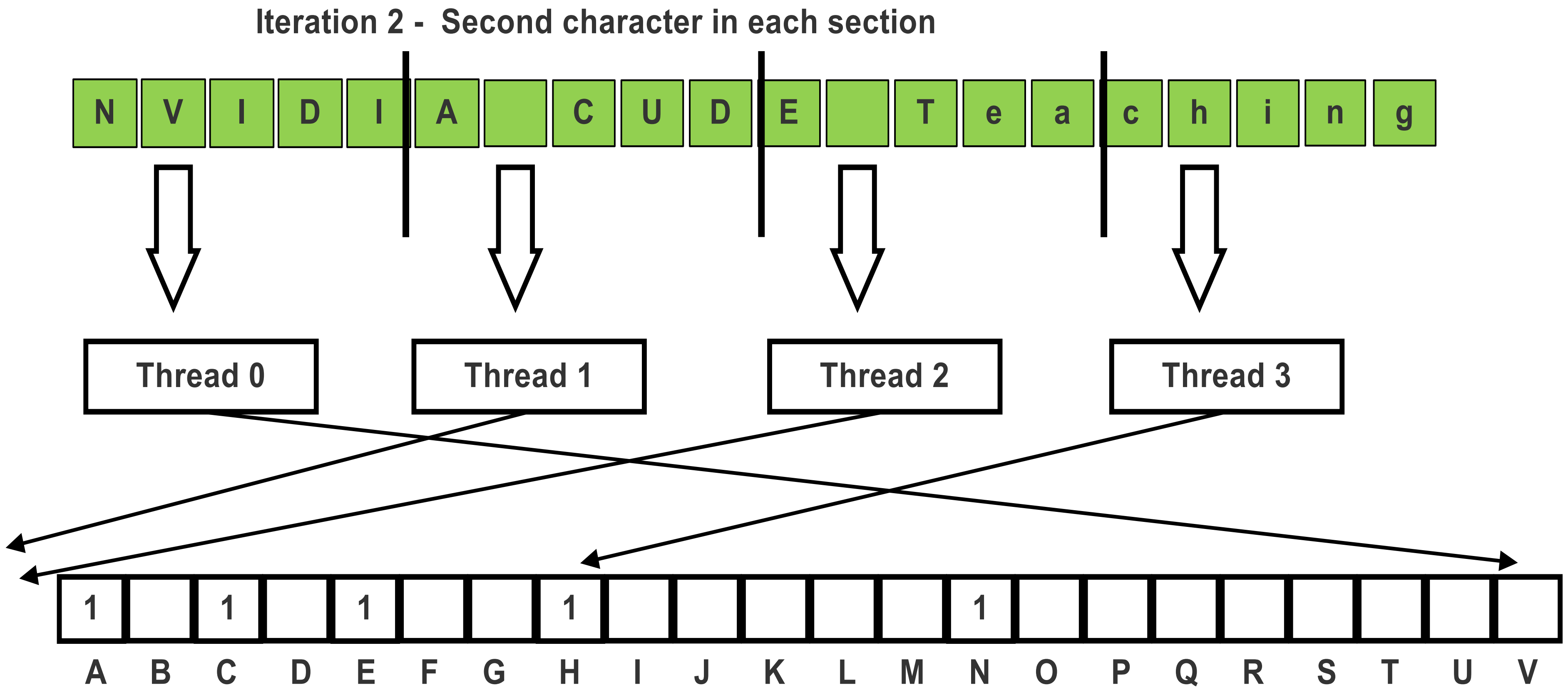
  N(2) , V(1) , I(3) , …

Parallel Histogram

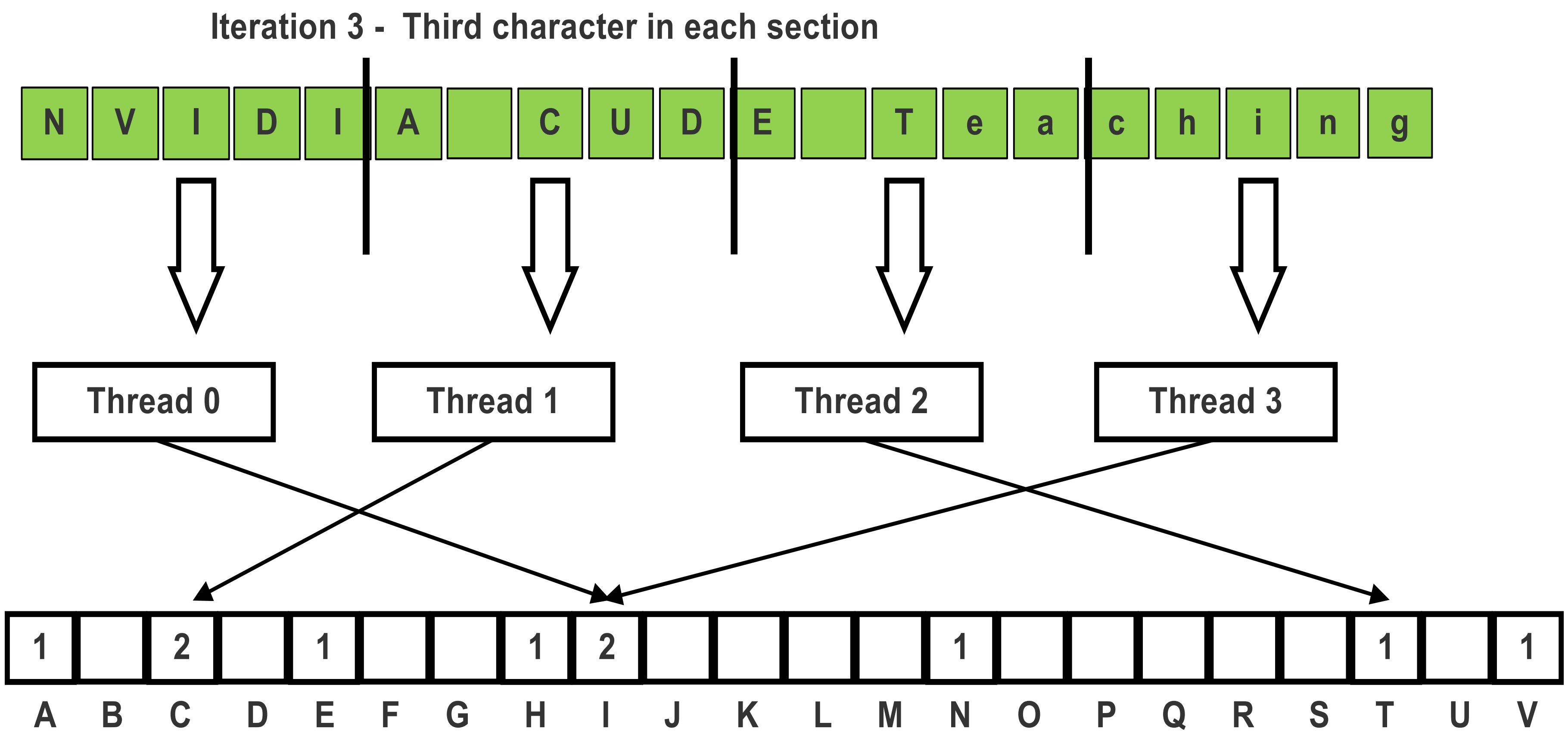
- Assign a section of input to each thread
- For each input character, use atomic operations to build the histogram



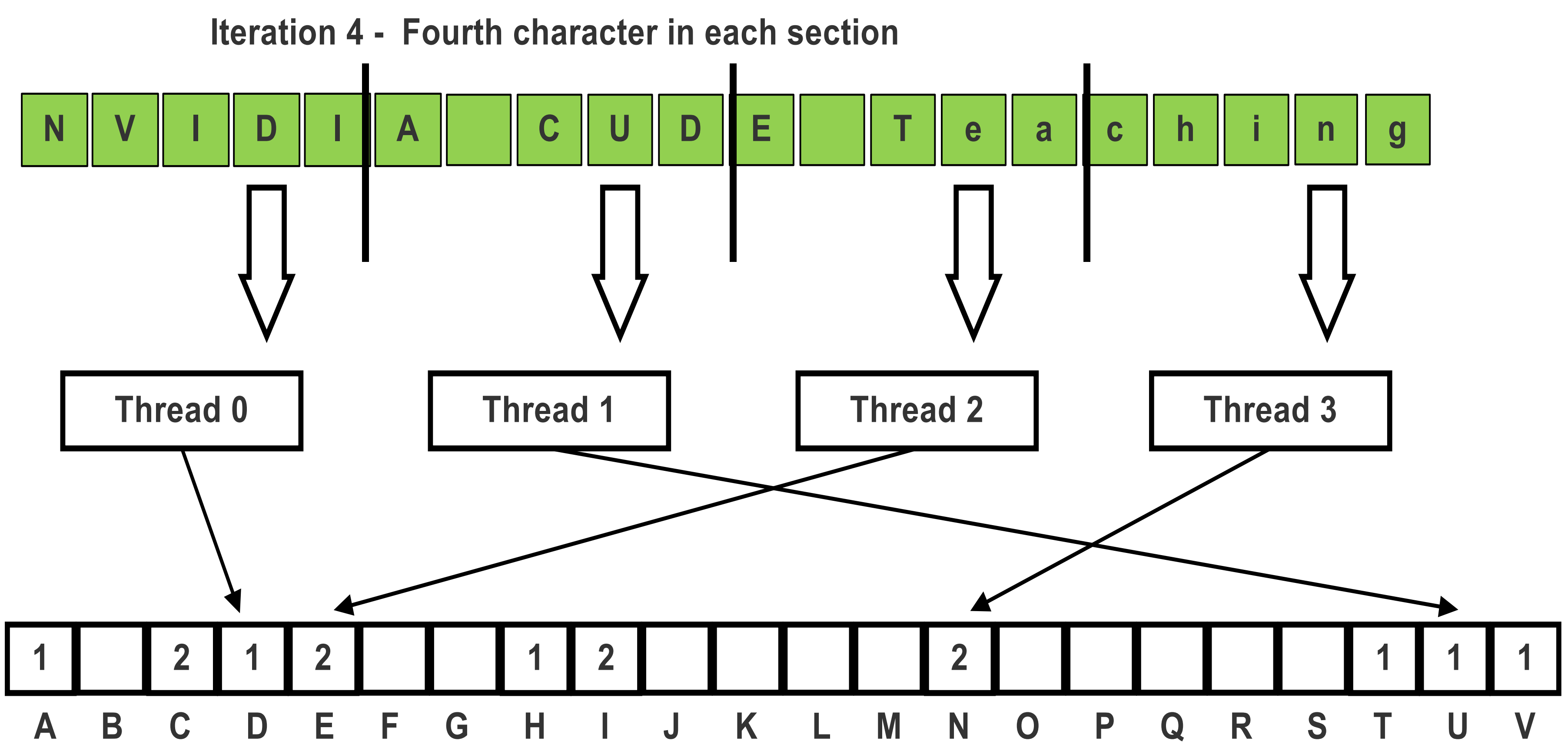
Parallel Histogram (Cont.)



Parallel Histogram (Cont.)



Parallel Histogram (Cont.)



Atomic Operations

- Atomic means locking a memory location for few cycles until all threads finish their operation on that location (operations are serialized)

- Done in CUDA using atomic functions (*atomicADD()*, *atomicSub*, etc.)

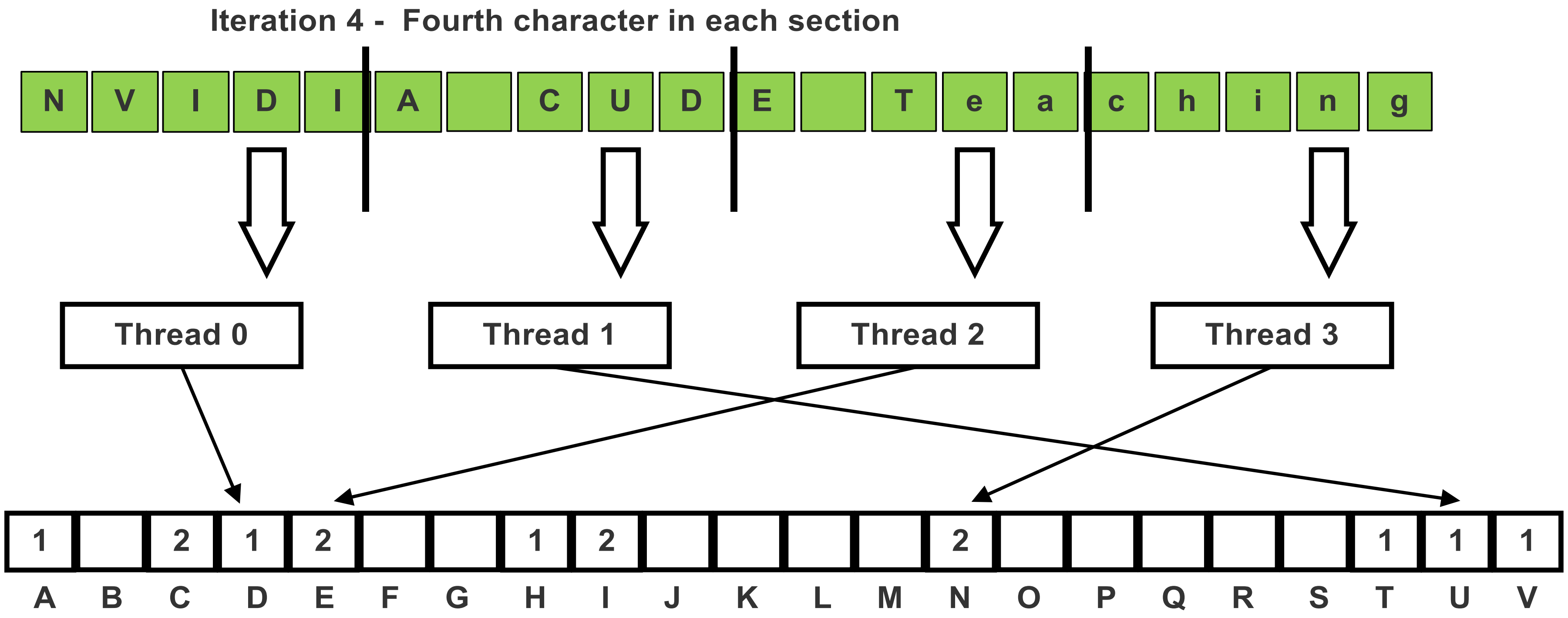
- An atomic function performs a read-modify-write sequence of atomic operations on 1 32-bit or 64-bit word residing in global or shared memory.

Atomic Operations (Cont.)

- For example, *atomicAdd()* reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address.
- Provide the guarantee that all the operations executed without any interference from other threads. In other words, no other thread can access this address until the operation is complete
- Atomic operations on Shared Memory have very short latency, but still serialized
- Private for each thread block

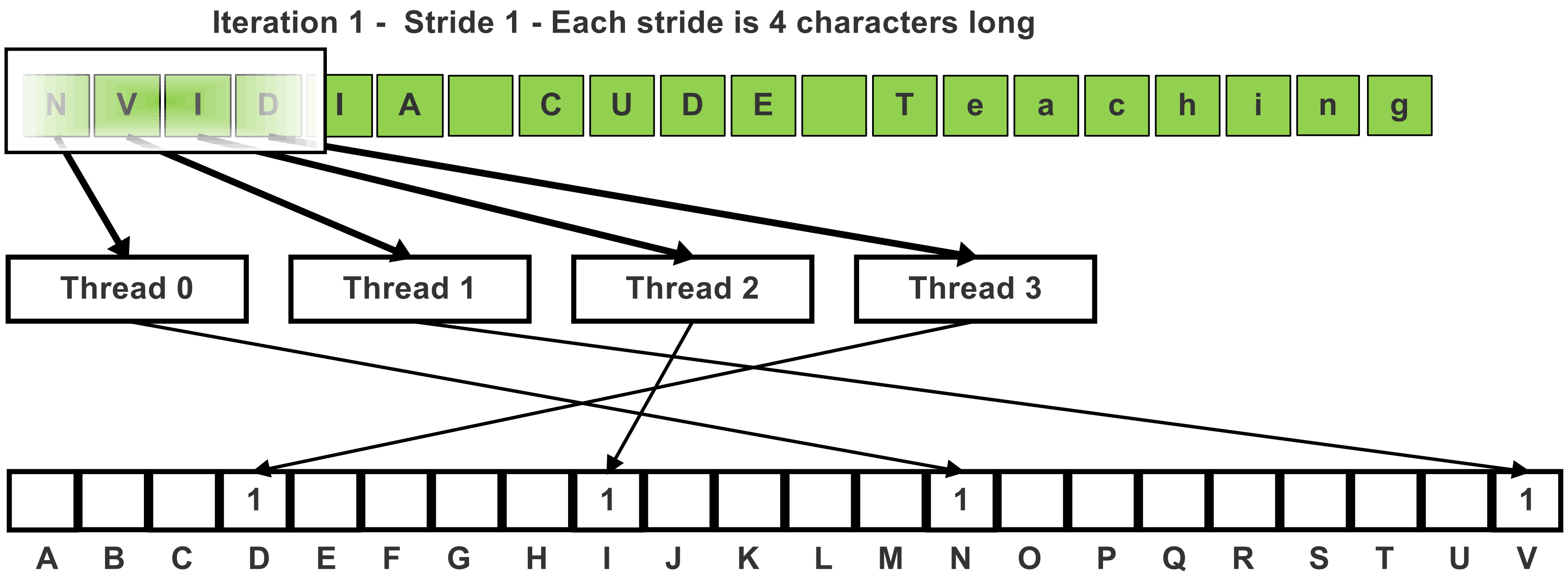
Non-coalesced Memory Access

- The previous parallel algorithm has a non-coalesced memory access for all threads (threads not accessing data in consecutive four words are not memory coalesced)



Coalesced Memory Access

- Section the input array into equal-sized strided patterns, and assign all threads to process a pattern



Basic Histogram Kernel

```
__global__ void hist_kernel(unsigned char *buffer, int size, unsigned int *hist)
{
        int i = threadIdx.x + blockIdx.x * blockDim.x;
        // stride is the total number of threads
        int stride = blockDim.x * gridDim.x;
        // All threads handle blockDim.x * gridDim.x consecutive elements
        while (i < size)
        {
                atomicAdd( &(hist[buffer[i]]), 1);
                i += stride;
        }
}
```

Shared Memory

- Have each thread block works on its private version of the histogram on shared memory
- Pros:
  - Reduces latency of GDRAM (global memory), maximize throughput
  - Reduces the serialization effect of atomic operations
- Cons:
  - Tiling needs to be handled carefully
  - Shared memory size is limited

Histogram Kernel with Shared Memory

```
__global__ void hist_kernel(unsigned char *buffer, int size, unsigned int *hist)
{
        int  i = threadIdx.x + blockIdx.x * blockDim.x;
        int  stride = blockDim.x * gridDim.x;
        extern __shared__ unsigned int   sh_hist_block[ ];
        if (threadIdx.x < blockDim.x) sh_hist_block[threadidx.x] =  0;
        __syncthreads();
        while (i  <  size)  {
                 atomicAdd(&(sh_hist_block[buffer[i]]), 1);
                i += stride;  }
        // wait for all other threads in the block to finish
        __syncthreads();
        if (threadIdx.x < blockDim.x) {
                atomicAdd( &(hist[threadIdx.x]), sh_hist_block[threadIdx.x] );
        }
}
```

References

[1] Wen-mei W. Hwu, “Heterogeneous Parallel Programming”. Online course, 2014. Available: https://class.coursera.org/hetero-002