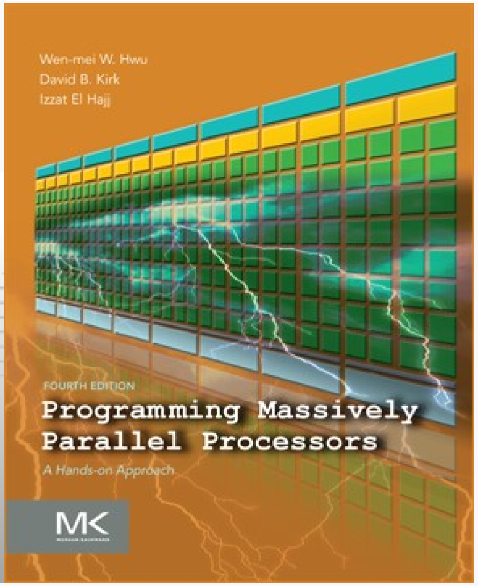
Programming Massively Parallel Processors

A Hands-on Approach

CHAPTER 10 › Reduction



Reduction

- A **reduction** operation reduces a set of input values to one value
  - e.g., sum, product, min, max
- Reduction operations are:
  - Associative
  - Commutative
  - Have a well-define identity value
- We will use sum as an example

Sequential Reduction

- Sequential reduction for sum:

```
sum = 0.0f;
for(i = 0; i < N; ++i) {
    sum += input[i];
}
```

- In general:

```
acc = IDENTITY;
for(i = 0; i < N; ++i) {
    acc = f(acc, input[i]);
}
```

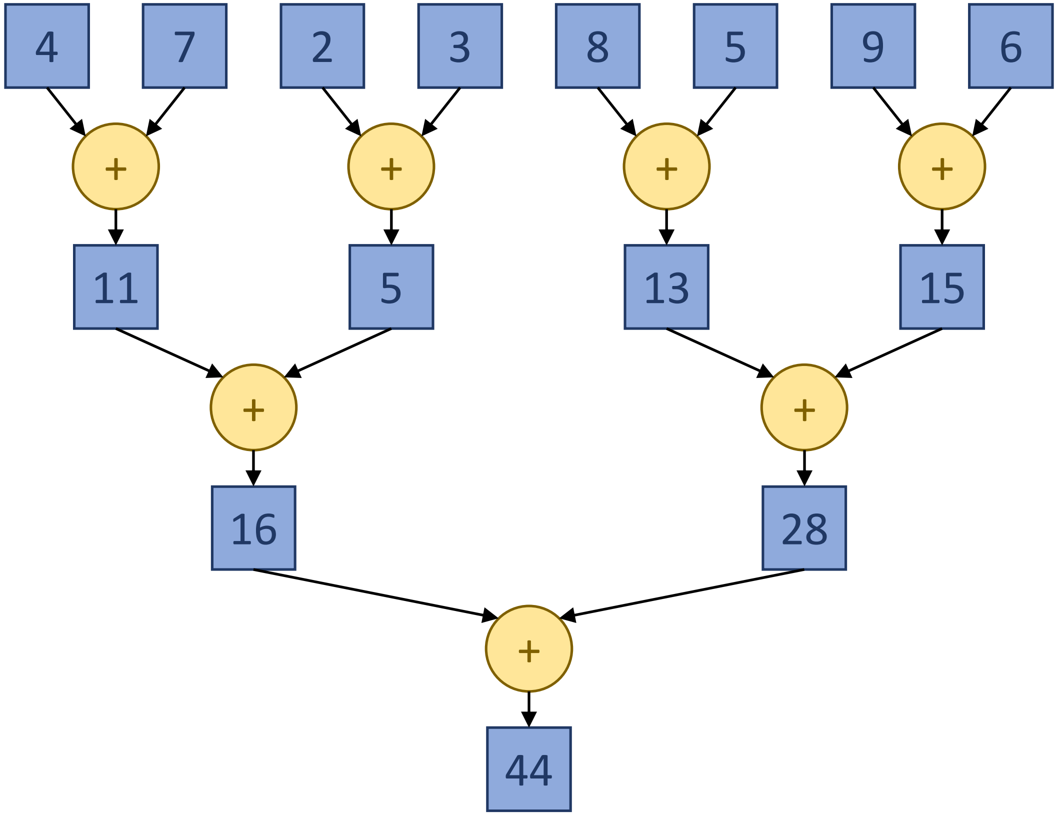
Parallel Reduction with Atomics

- Parallel reduction for sum using atomics:

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
if(i < N) {
    atomicAdd(sum, input[i]);
}
```

- Poor performance:
  - All additions are serialized by the hardware

Parallel Reduction



<u>Approach:</u> Every thread adds two elements in each step

Takes *log(N)* steps and half the threads drop out every step

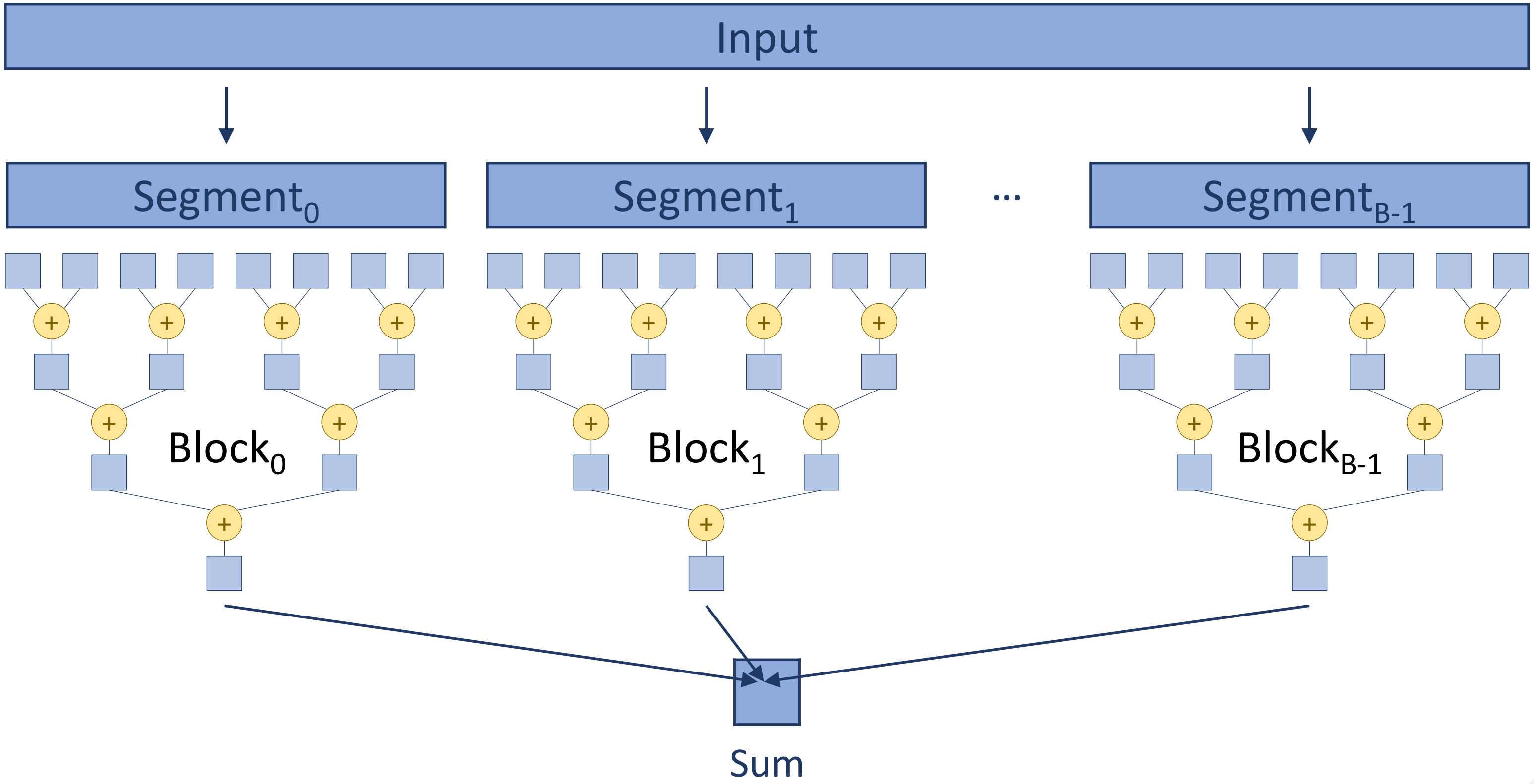
Pattern is called a **reduction tree**

Segmented Reduction

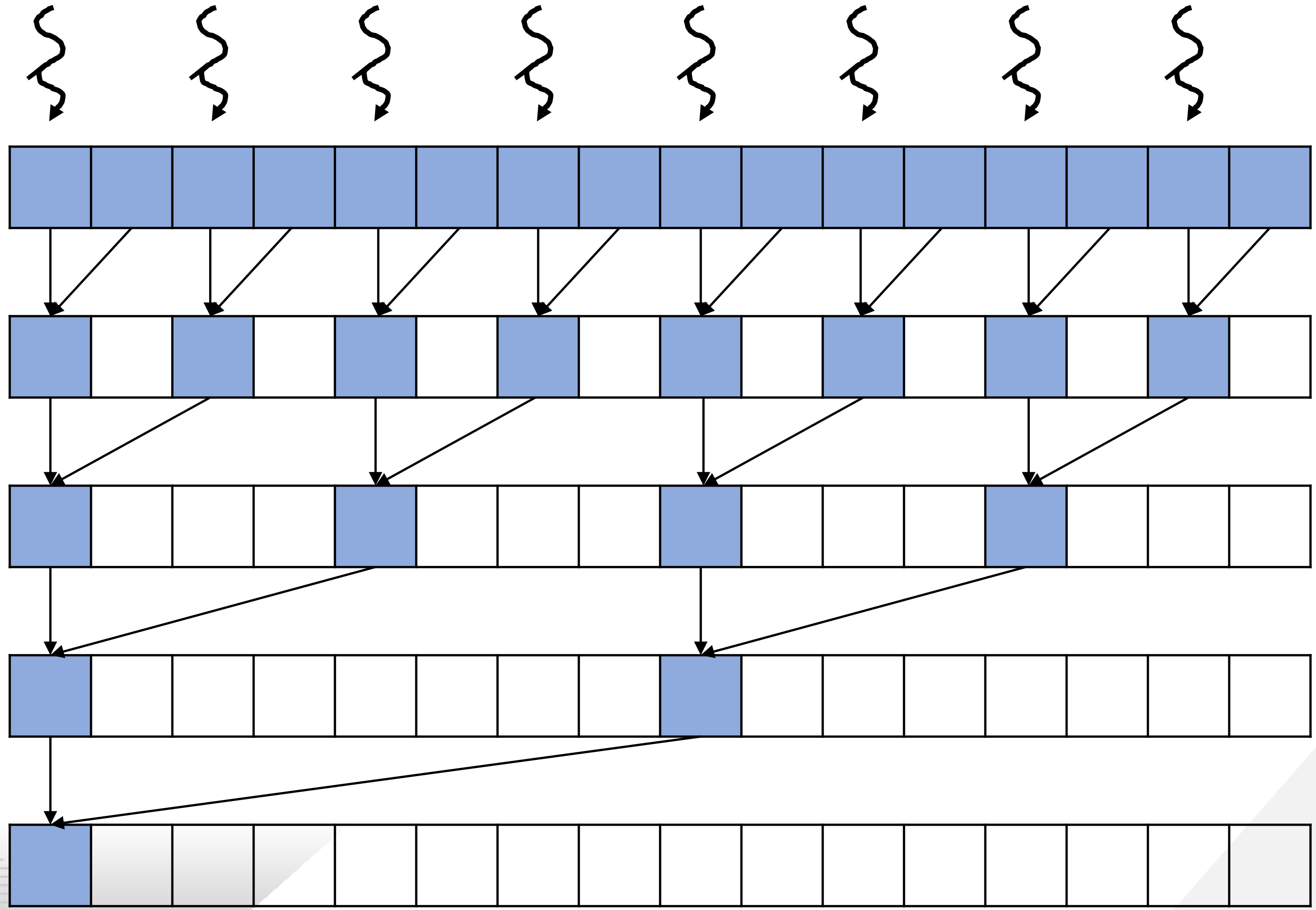
- Threads must synchronize between steps
  - Cannot synchronize across threads in different blocks

- Solution: **segmented reduction**
  - Every thread block reduces a segment of the input and produces a partial sum
  - The partial sum is atomically added to the final sum

Segmented Reduction Example



Reduction Tree (Per Block)



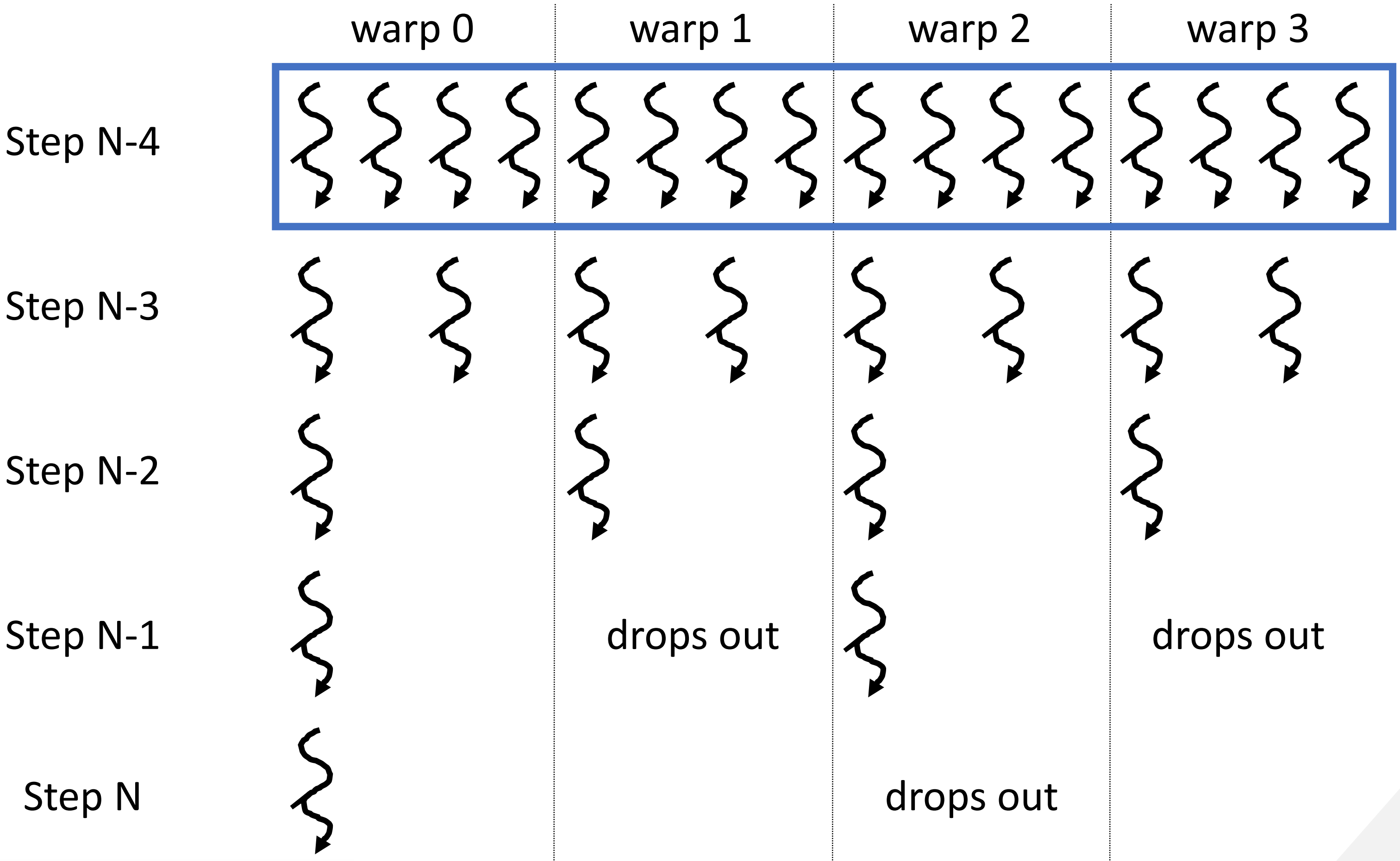
Reduction Code

```
unsigned int segment = 2*blockDim.x*blockIdx.x;
unsigned int i = segment + 2*threadIdx.x;
for(unsigned int stride = 1; stride <= BLOCK_DIM; stride *= 2) {
    if(threadIdx.x%stride == 0) {
        input[i] += input[i + stride];
    }
    __syncthreads();
}
```

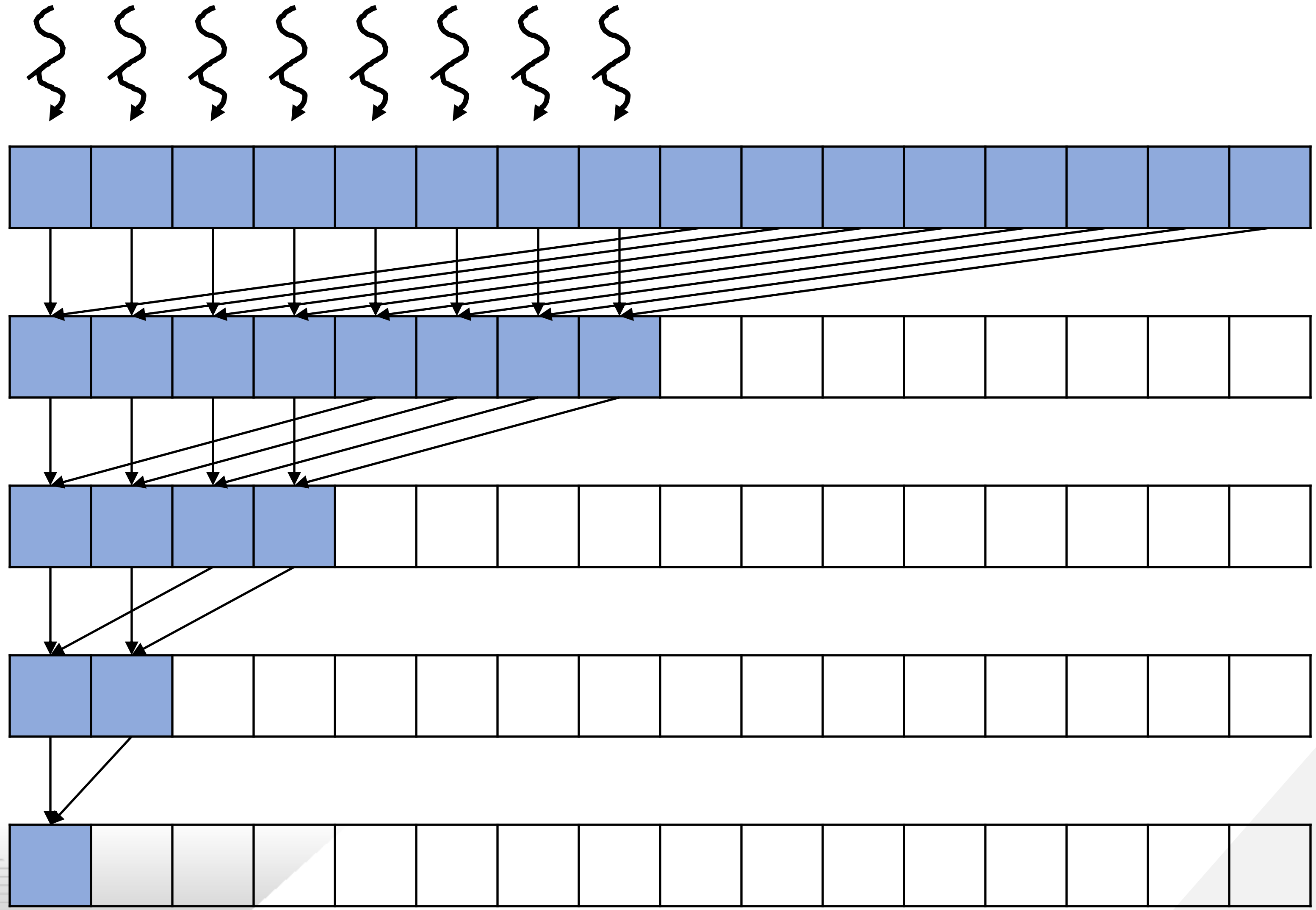
- Problems:
  - Accesses to `input` are not coalesced
  - Control divergence

```
__global__ void SimpleSumReductionKernel(float* input, float* output) {
    unsigned int i = 2*threadIdx.x;
    for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
        if (threadIdx.x % stride == 0) {
            input[i] += input[i + stride];
        }
        __syncthreads();
    }
    if(threadIdx.x == 0) {
        *output = input[0];
    }
}
```

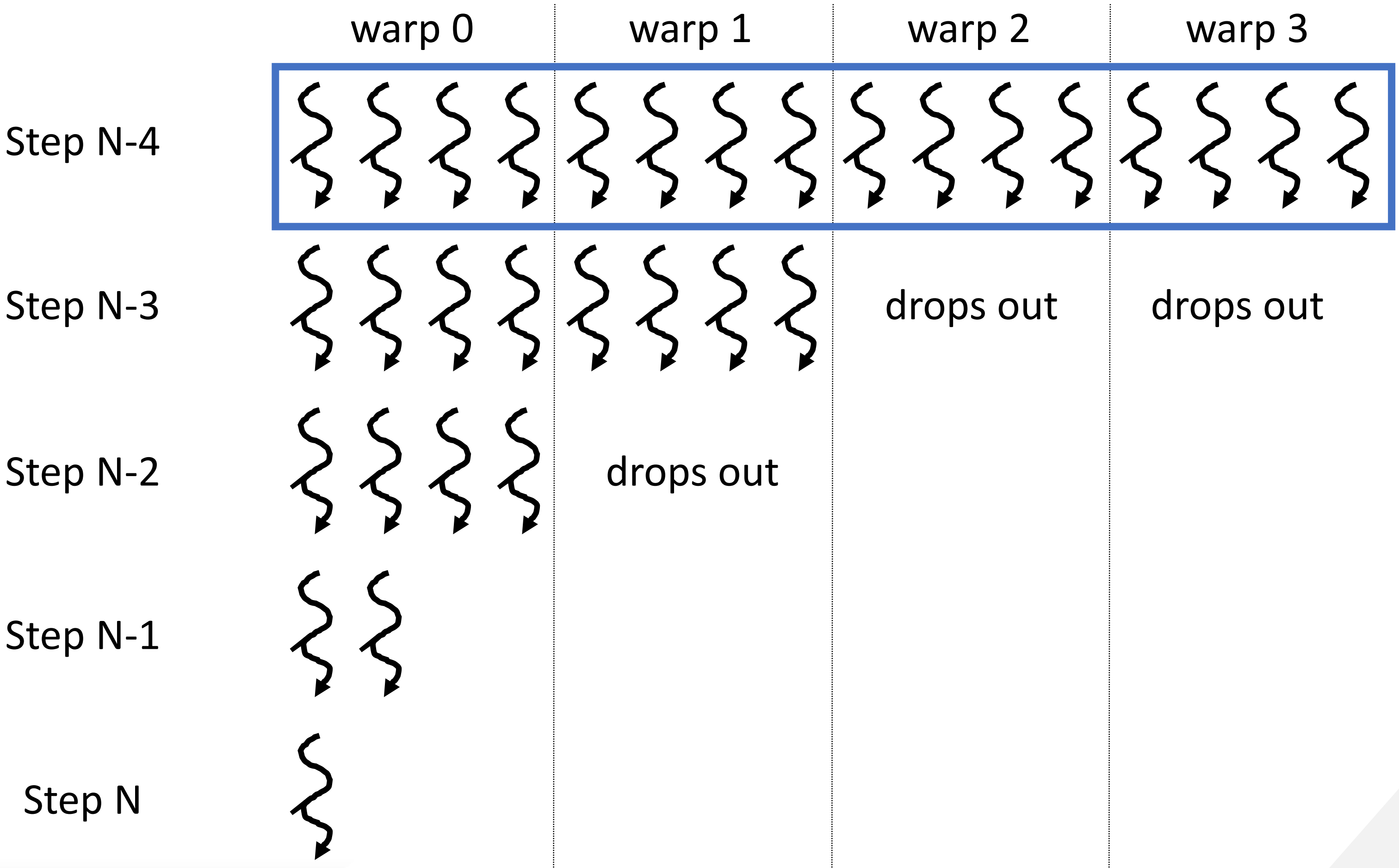
Control Divergence Problem



Coalescing and Minimizing Divergence



Control Divergence Minimized



Reduction Code with Coalescing and Minimizing Divergence

```
unsigned int segment = 2*blockDim.x*blockIdx.x;
unsigned int i = segment + threadIdx.x;
for(unsigned int stride = BLOCK_DIM; stride > 0; stride /= 2) {
    if(threadIdx.x < stride) {
        input[i] += input[i + stride];
    }
    __syncthreads();
}
```

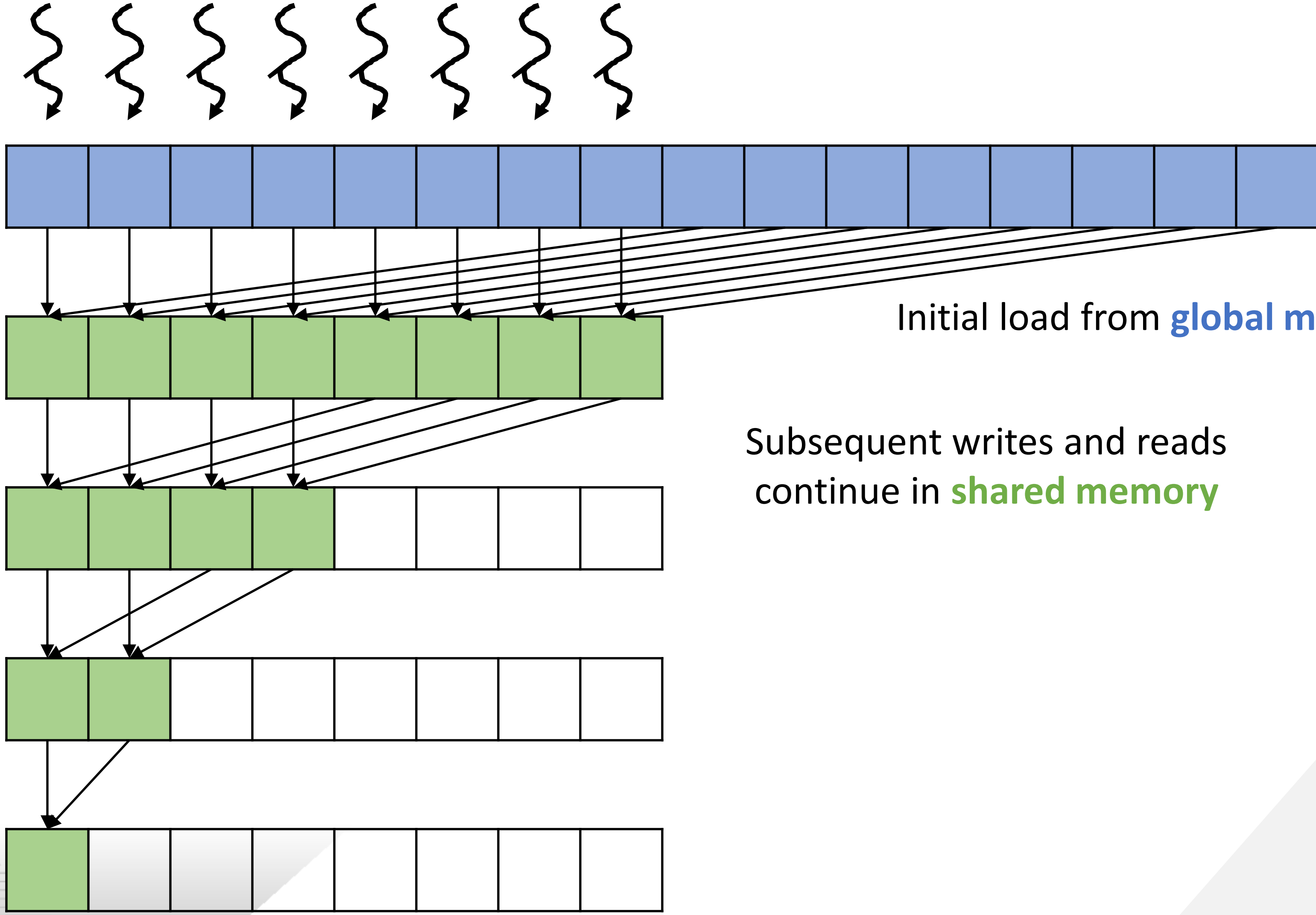
```
__global__ void ConvergentSumReductionKernel(float* input, float* output) {
    unsigned int i = threadIdx.x;
    for (unsigned int stride = blockDim.x; stride >= 1; stride /= 2) {
        if (threadIdx.x < stride) {
            input[i] += input[i + stride];
        }
        __syncthreads();
    }
    if(threadIdx.x == 0) {
        *output = input[0];
    }
}
```

Data Reuse

- While specific data values are not reused, the same memory locations are repeatedly read and written

- Optimization: load input to shared memory first and perform reduction tree on shared memory
    - Also avoids modifying the input if needed in the future

Using Shared Memory



Initial load from **global memory**

Subsequent writes and reads continue in **shared memory**

Reduction Code with Shared Memory

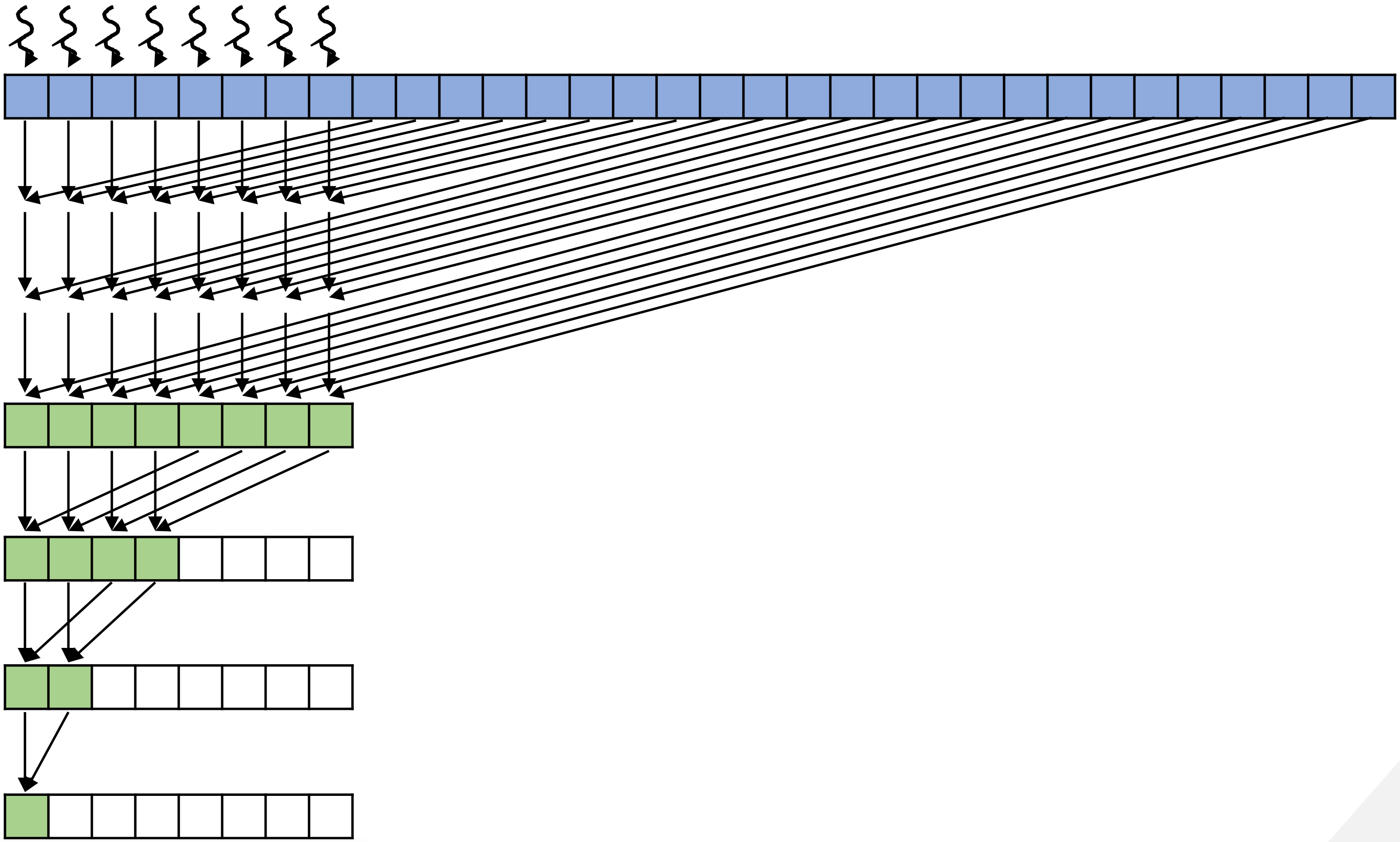
```
unsigned int segment = 2*blockDim.x*blockIdx.x;
unsigned int i = segment + threadIdx.x;

// Load data to shared memory
__shared__ float input_s[BLOCK_DIM];
input_s[threadIdx.x] = input[i] + input[i + BLOCK_DIM];
__syncthreads();

// Reduction tree in shared memory
for(unsigned int stride = BLOCK_DIM/2; stride > 0; stride /= 2) {
    if(threadIdx.x < stride) {
        input_s[threadIdx.x] += input_s[threadIdx.x + stride];
    }
    __syncthreads();
}
```

Thread Coarsening

- Cost of parallelization:
  - Synchronization every step
  - Control divergence in the final steps

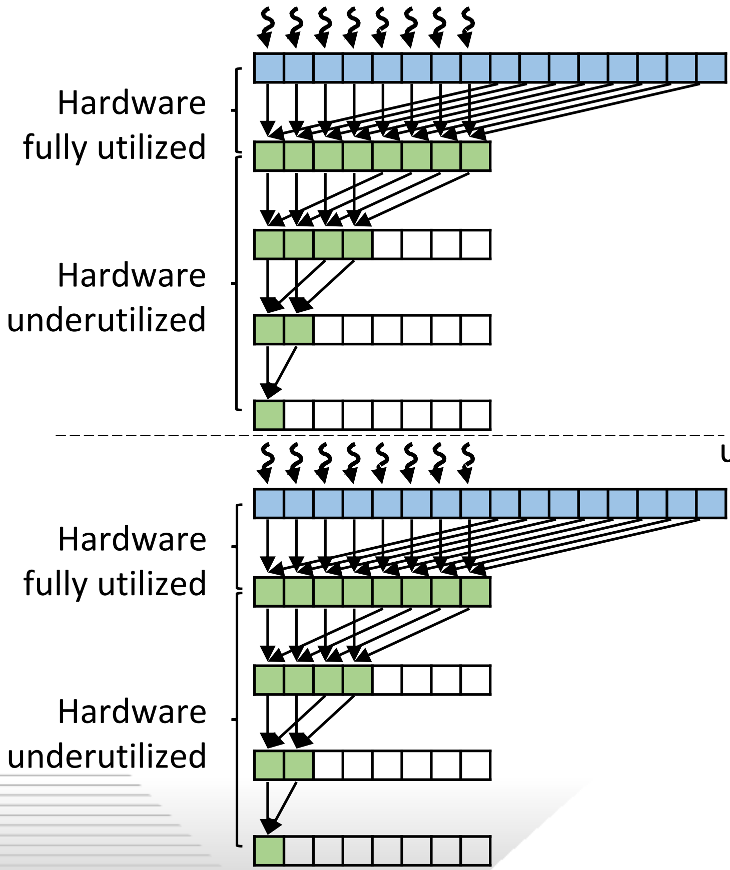
- Better to coarsen threads if there are many more blocks than resources available

Applying Thread Coarsening

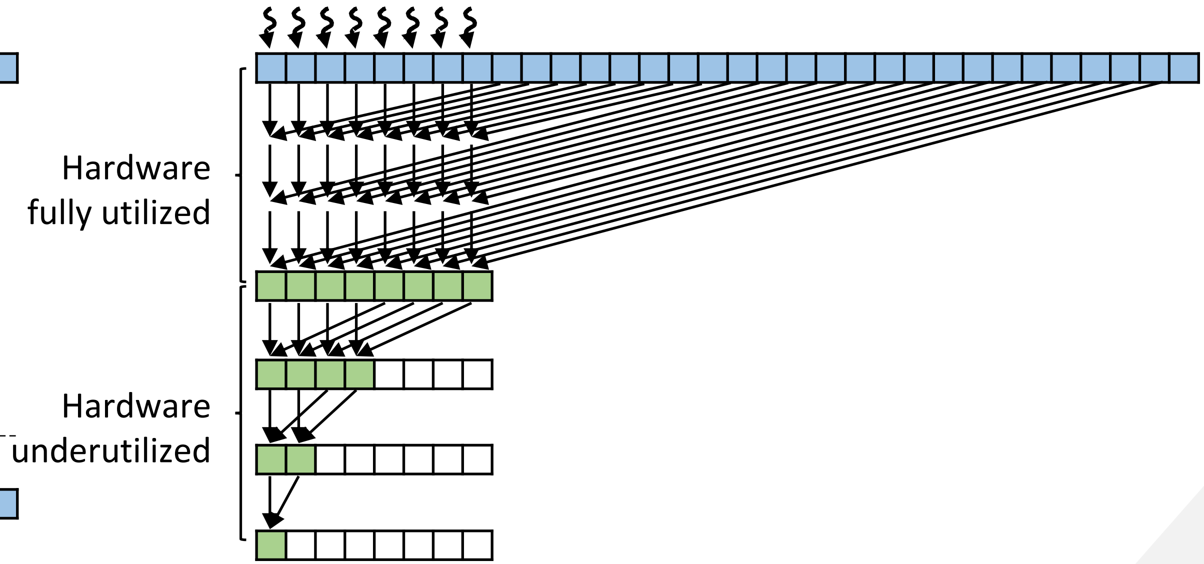


Before vs. After Coarsening

Execution of two non-coarsened thread blocks serialized by the hardware



Execution of one coarsened thread block doing the work of two original non-coarsened thread blocks



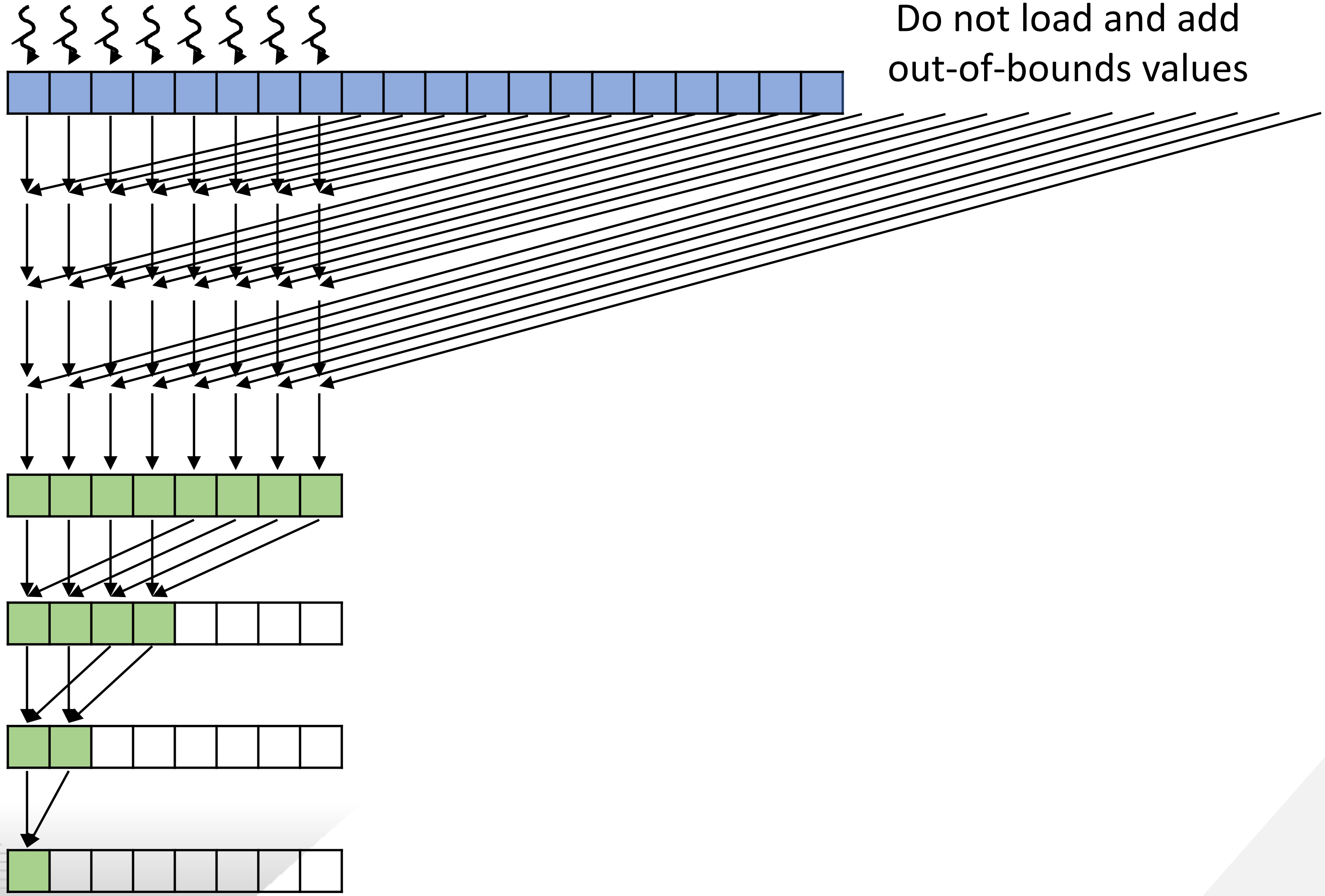
Reduction Code with Thread Coarsening

```
unsigned int segment = COARSE_FACTOR*2*blockDim.x*blockIdx.x;
unsigned int i = segment + threadIdx.x;

// Load data to shared memory
__shared__ float input_s[BLOCK_DIM];
float threadSum = 0.0f;
for(unsigned int c = 0; c < COARSE_FACTOR*2; ++c) {
    threadSum += input[i + c*BLOCK_DIM];
}
input_s[threadIdx.x] = threadSum;

__syncthreads();

// Reduction tree in shared memory
for(unsigned int stride = BLOCK_DIM/2; stride > 0; stride /= 2) {
    if(threadIdx.x < stride) {
        input_s[threadIdx.x] += input_s[threadIdx.x + stride];
    }
    __syncthreads();
}
```

Coarsening Benefits

- Let *N* be the number of elements per original block
  - i.e., *N* = 2*blockDim.x
- If blocks are all executed in parallel:
  - *log(N)* steps, *log(N)* synchronizations
- If blocks serialized by the hardware by a factor of C:
  - *C*log(N)* steps, *C*log(N)* synchronizations
- If blocks are coarsened by a factor of C:
  - *2*(C – 1) + log(N)* steps, *log(N)* synchronizations

Boundary Conditions



Do not load and add out-of-bounds values

References

- Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.