CHAPTER 10

Reduction
And minimizing divergence

```
01    sum = 0.0f;
02    for(i = 0; i < N; ++i) {
03        sum += input[i];
04    }
```

FIGURE 10.1

A simple sum reduction sequential code.

```
01    acc = IDENTITY;
02    for(i = 0; i < N; ++i) {
03        acc = Operator(acc, input[i]);
04    }
```

FIGURE 10.2

The general form of a reduction sequential code.

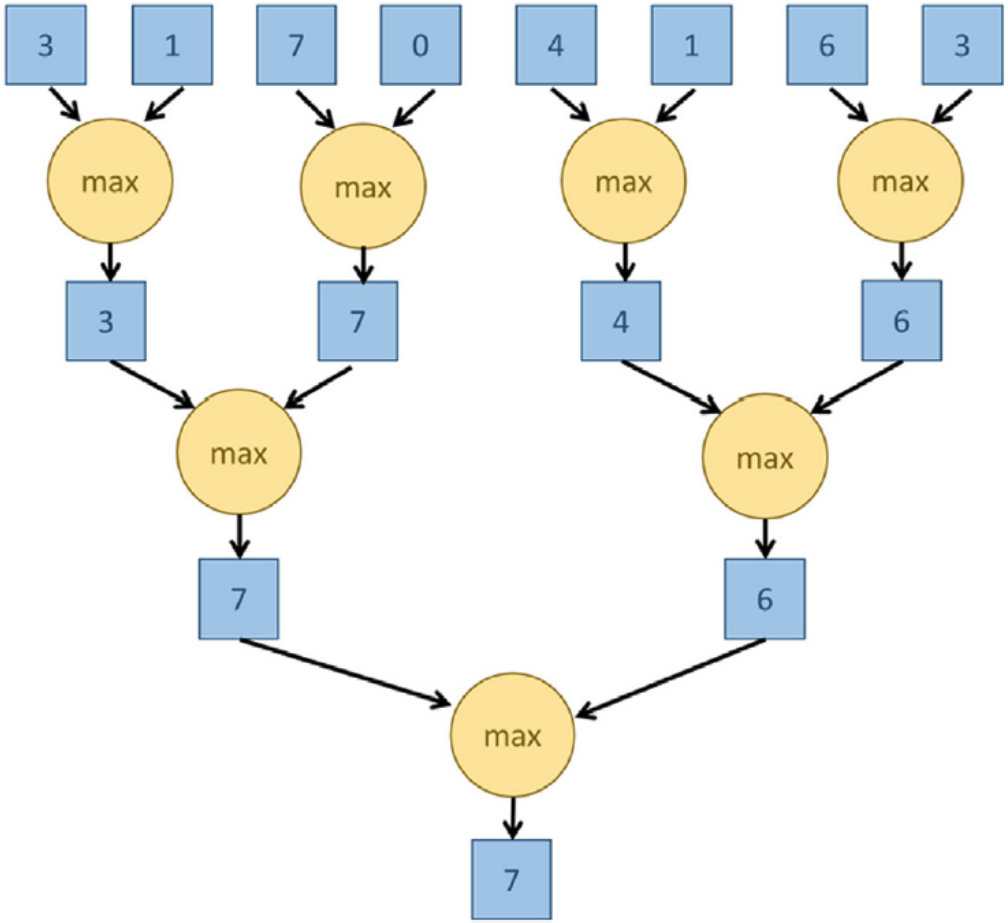


FIGURE 10.3

A parallel max reduction tree.

FIGURE 10.4

The 2010 World Cup Finals as a reduction tree.

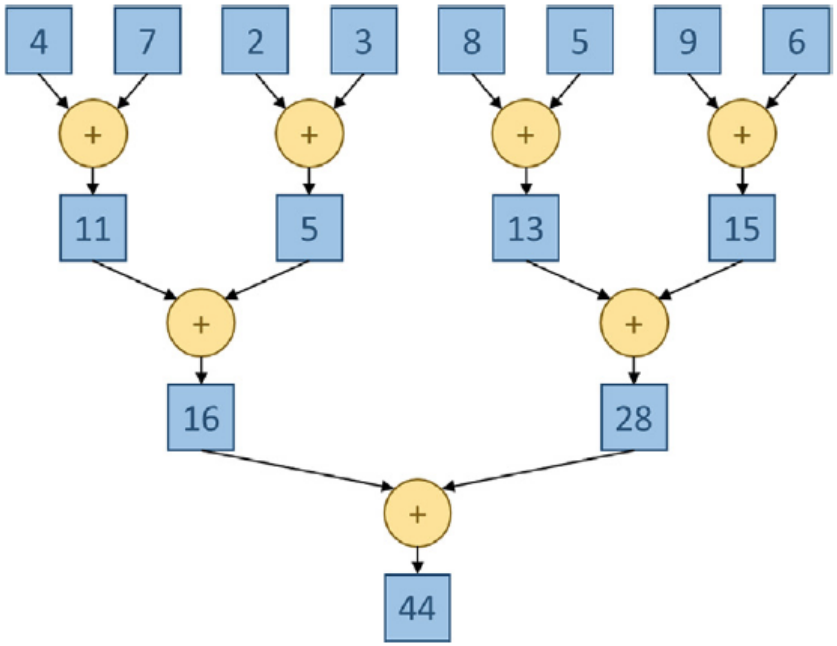


FIGURE 10.5

A parallel sum reduction tree.

```
01    __global__ void SimpleSumReductionKernel(float* input, float* output) {
02        unsigned int i = 2*threadIdx.x;
03        for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
04            if (threadIdx.x % stride == 0) {
05                input[i] += input[i + stride];
06            }
07            __syncthreads();
08        }
09        if(threadIdx.x == 0) {
10            *output = input[0];
11        }
12    }
```

FIGURE 10.6

A simple sum reduction kernel.

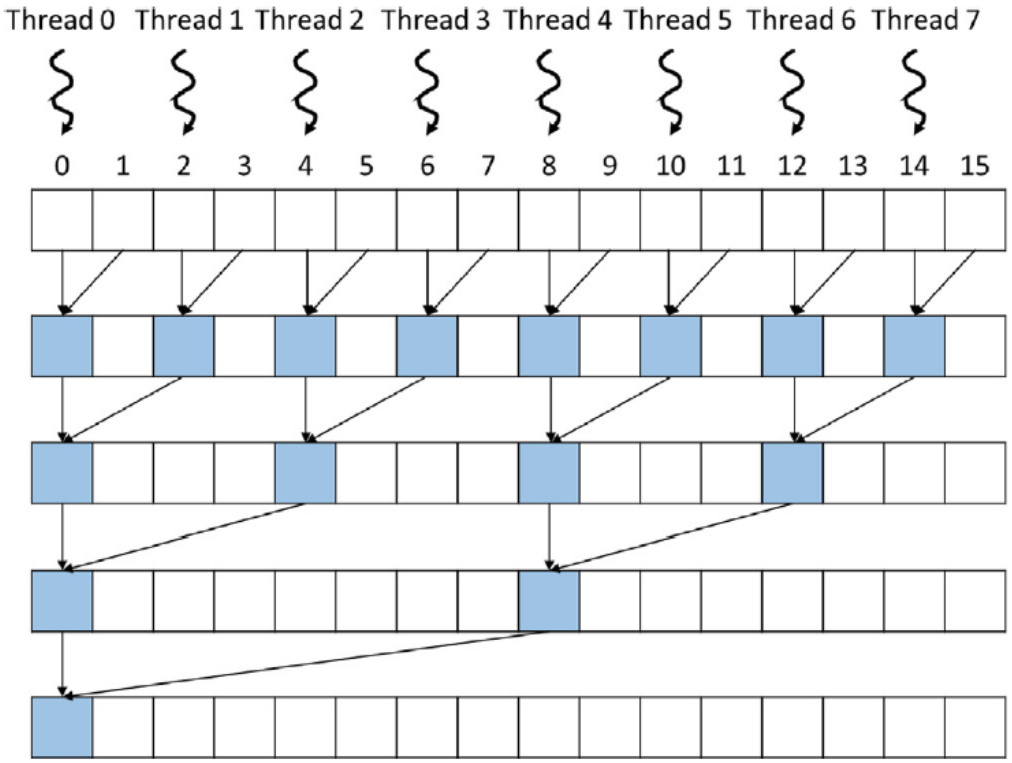


FIGURE 10.7

The assignment of threads ("owners") to the `input` array locations and progress of execution over time for the `SimpleSumReudctionKernel` in Fig. 10.6. The time progresses from top to bottom, and each level corresponds to one iteration of the for-loop.

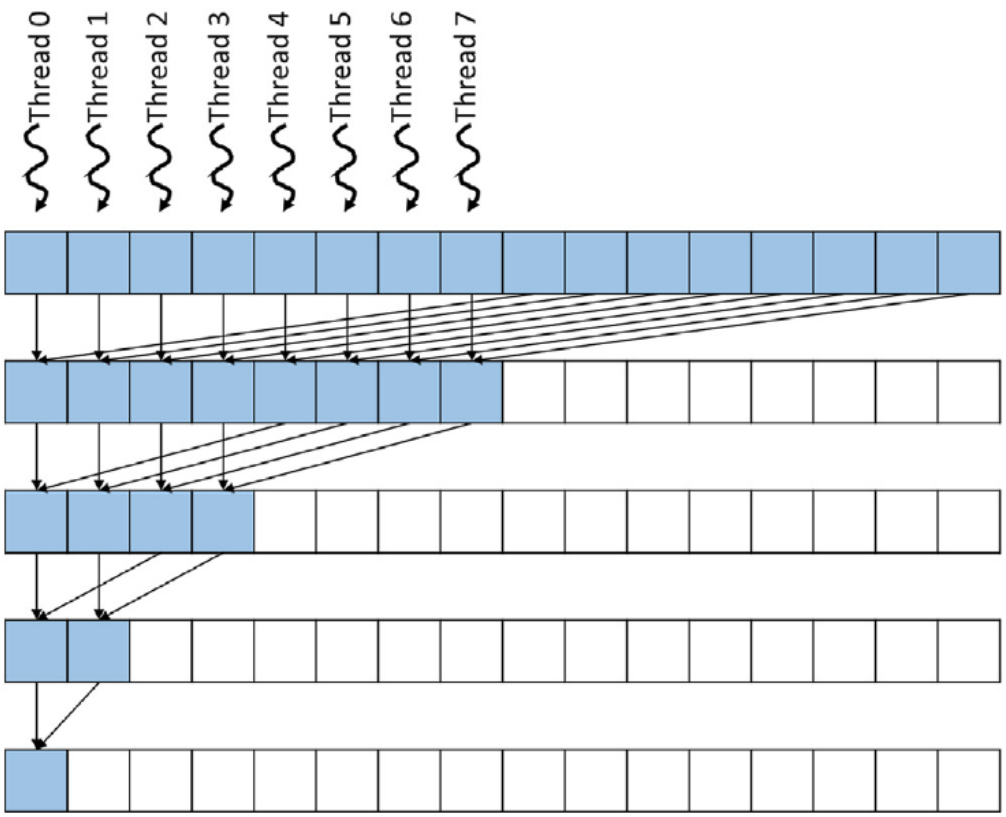


FIGURE 10.8

A better assignment of threads to input array locations for reduced control divergence.

```
01    __global__ void ConvergentSumReductionKernel(float* input, float* output) {
02        unsigned int i = threadIdx.x;
03        for (unsigned int stride = blockDim.x; stride >= 1; stride /= 2) {
04            if (threadIdx.x < stride) {
05                input[i] += input[i + stride];
06            }
07            __syncthreads();
08        }
09        if(threadIdx.x == 0) {
10            *output = input[0];
11        }
12    }
```

FIGURE 10.9

A kernel with less control divergence and improved execution resource utilization efficiency.

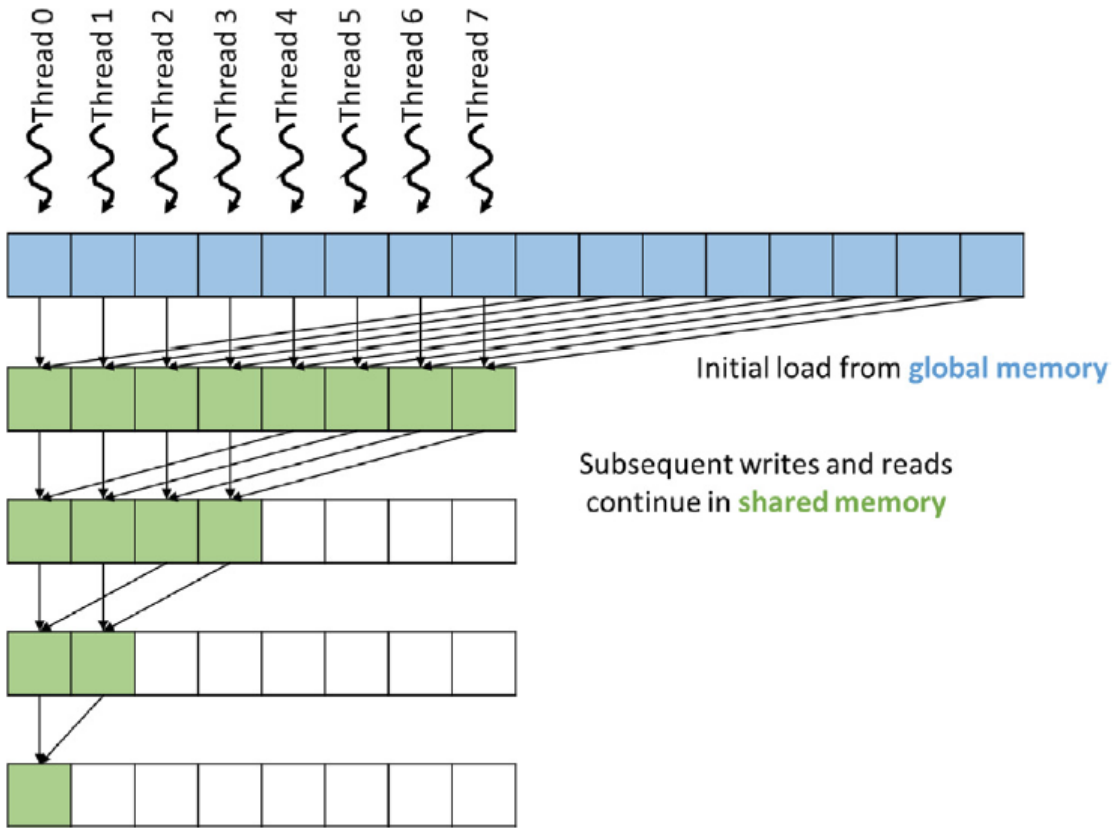


FIGURE 10.10

Using shared memory to reduce accesses to the global memory.

```
01    __global__ void SharedMemorySumReductionKernel(float* input) {
02        __shared__ float input_s[BLOCK_DIM];
03        unsigned int t = threadIdx.x;
04        input_s[t] = input[t] + input[t + BLOCK_DIM];
05        for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2) {
06            __syncthreads();
07            if (threadIdx.x < stride) {
08                input_s[t] += input_s[t + stride];
09            }
10        }
11        if (threadIdx.x == 0) {
12            *output = input_s[0];
13        }
14    }
```

FIGURE 10.11

A kernel that uses shared memory to reduce global memory accesses.

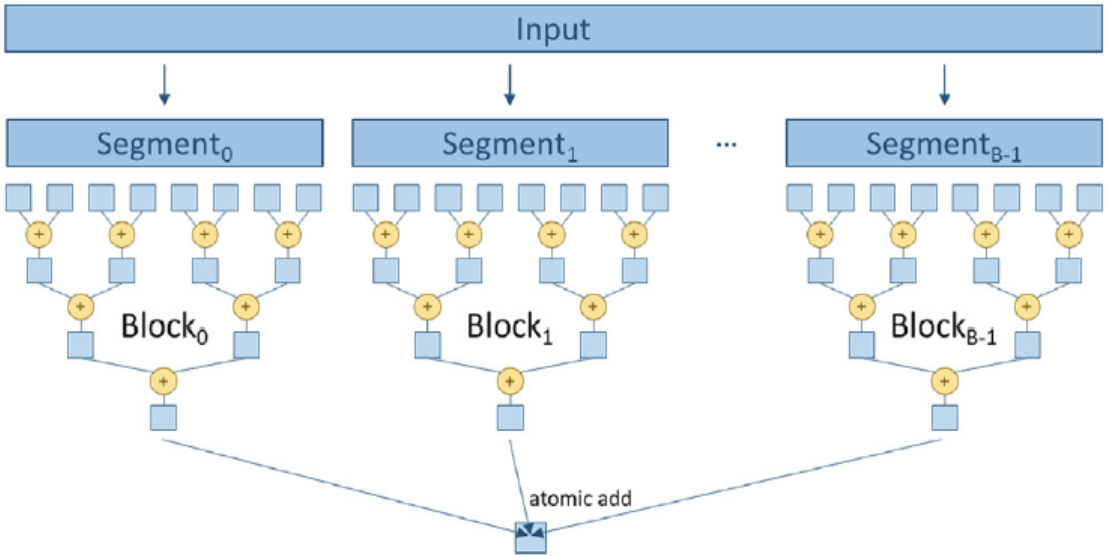


FIGURE 10.12

Segmented multiblock reduction using atomic operations.

```
01    __global__ SegmentedSumReductionKernel(float* input, float* output) {
02        __shared__ float input_s[BLOCK_DIM];
03        unsigned int segment = 2*blockDim.x*blockIdx.x;
04        unsigned int i = segment + threadIdx.x;
05        unsigned int t = threadIdx.x;
06        input_s[t] = input[i] + input[i + BLOCK_DIM];
07        for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2){
08            __syncthreads();
09            if (t < stride) {
10                input_s[t] += input_s[t + stride];
11              }
12        }
13        if (t == 0) {
14              atomicAdd(output, input_s[0]);
15          }
16    }
```

FIGURE 10.13

A segmented multiblock sum reduction kernel using atomic operations.

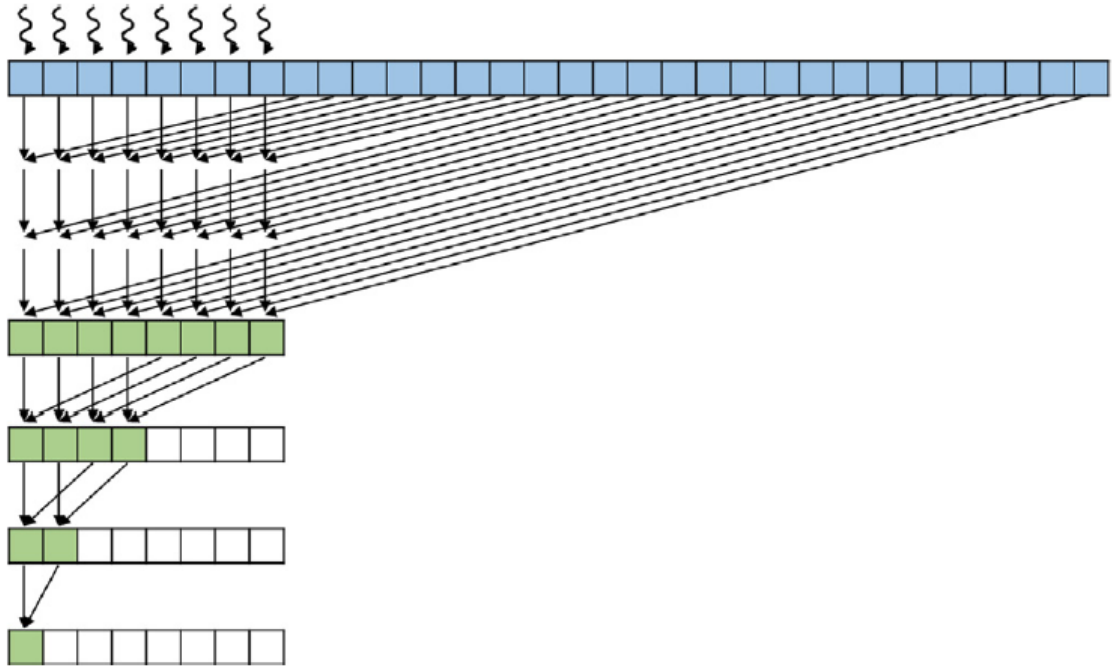


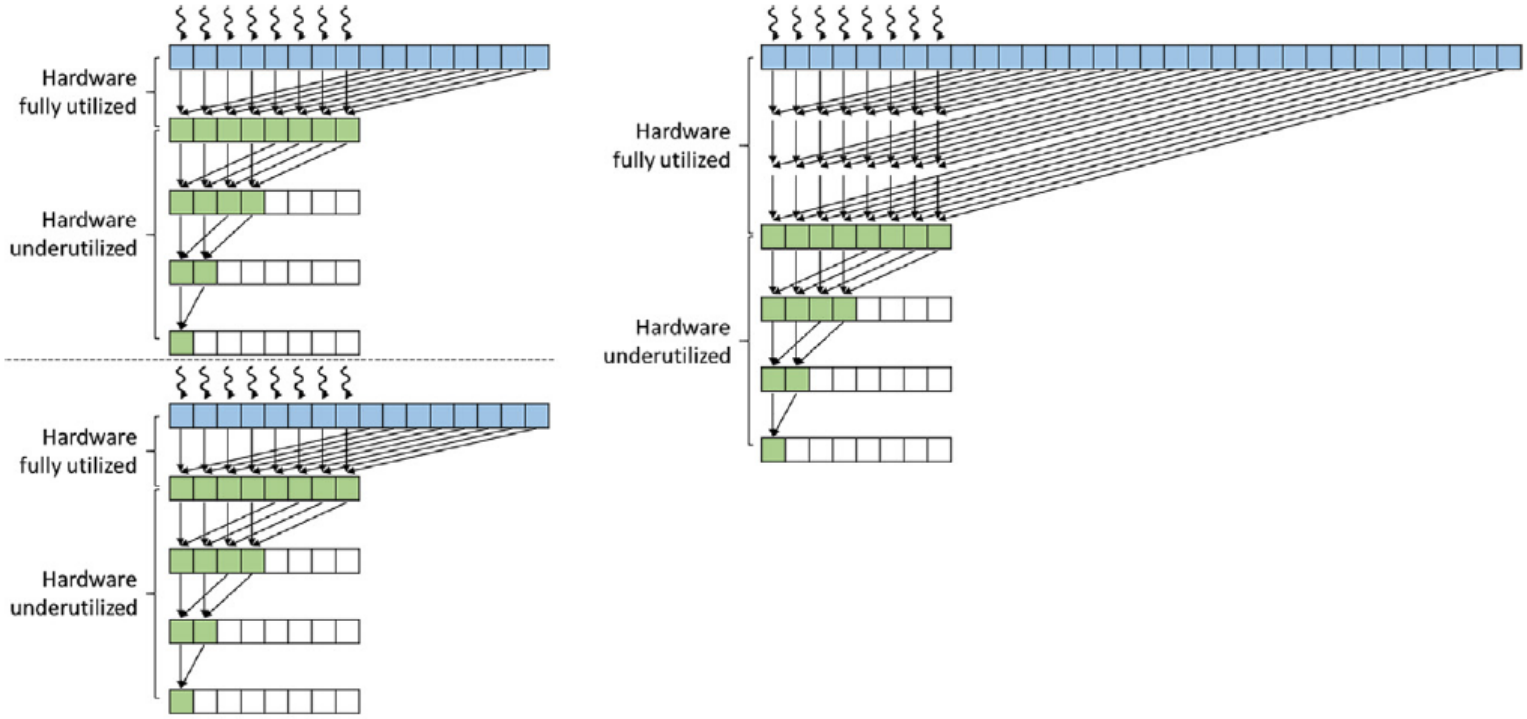
FIGURE 10.14

Thread coarsening in reduction.

```
01    __global__ CoarsenedSumReductionKernel(float* input, float* output) {
02        __shared__ float input_s[BLOCK_DIM];
03        unsigned int segment = COARSE_FACTOR*2*blockDim.x*blockIdx.x;
04        unsigned int i = segment + threadIdx.x;
05        unsigned int t = threadIdx.x;
06        float sum = input[i];
07        for(unsigned int tile = 1; tile < COARSE_FACTOR*2; ++tile) {
08            sum += input[i + tile*BLOCK_DIM];
09        }
10        input_s[t] = sum;
11        for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2){
12            __syncthreads();
13            if (t < stride) {
14                input_s[t] += input_s[t + stride];
15            }
16        }
17        if (t == 0) {
18              atomicAdd(output, input_s[0]);
19          }
20    }
```

FIGURE 10.15

Sum reduction kernel with thread coarsening.

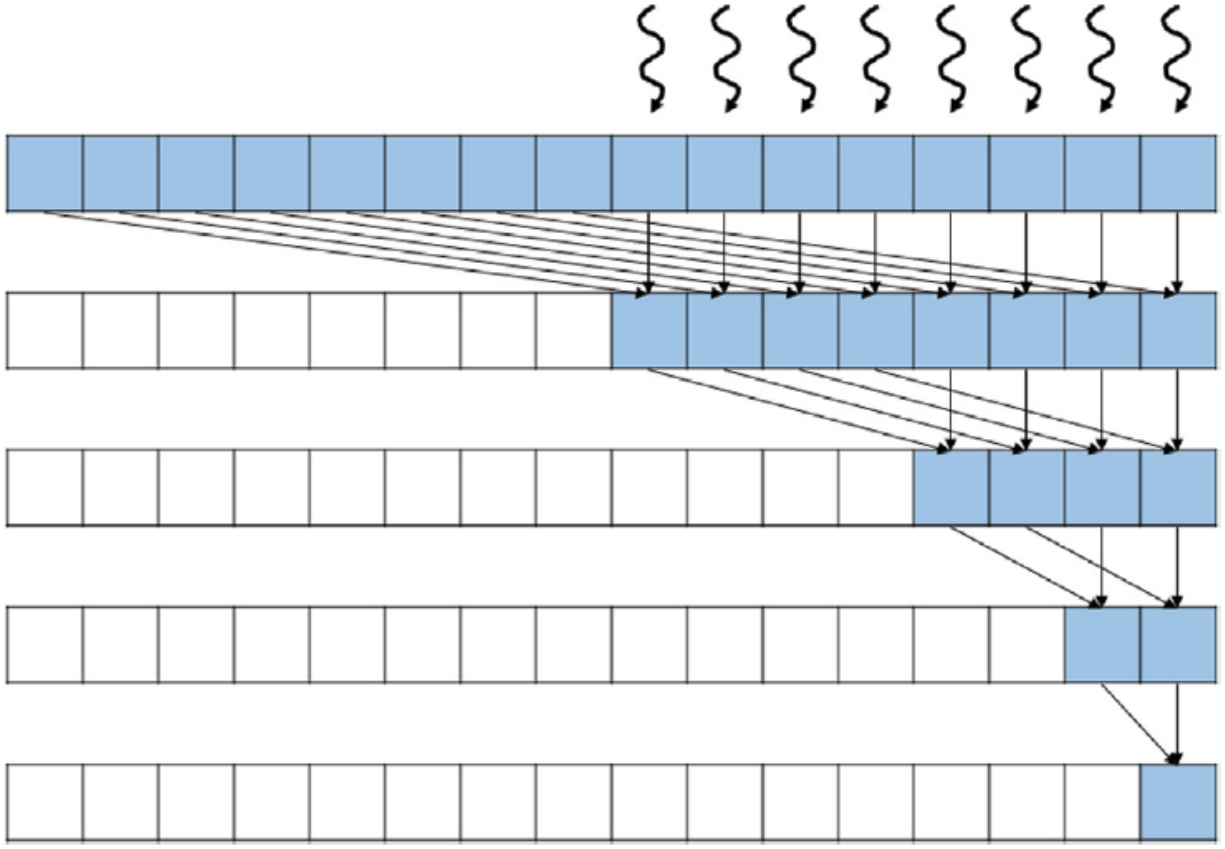


(A) Execution of two original thread blocks serialized by the hardware

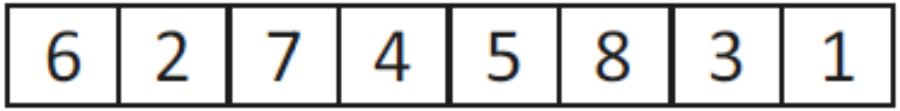
(B) Execution of one coarsened thread block doing the work of two original thread blocks

FIGURE 10.16

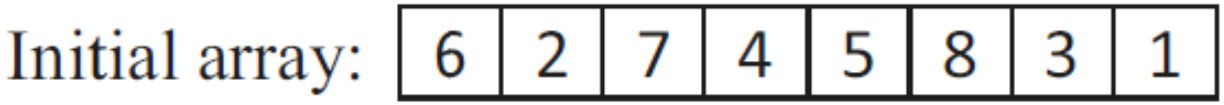
Comparing parallel reduction with and without thread coarsening.



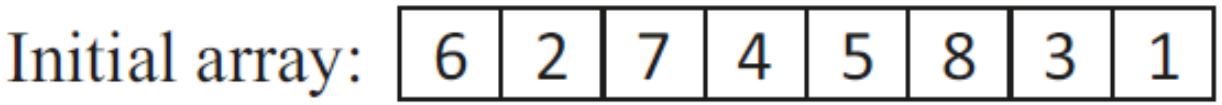
In-text figure 1



In-text figure 2



In-text figure 3



In-text figure 4