CHAPTER 11

Prefix sum (scan)
An introduction to work efficiency in parallel algorithms

```
void sequential_scan(float *x, float *y, unsigned int N) {
    y[0] = x[0];
    for(unsigned int i = 1; i < N; ++i) {
        y[i] = y[i - 1] + x[i];
    }
}
```

FIGURE 11.1

A simple sequential implementation of inclusive scan based on addition.

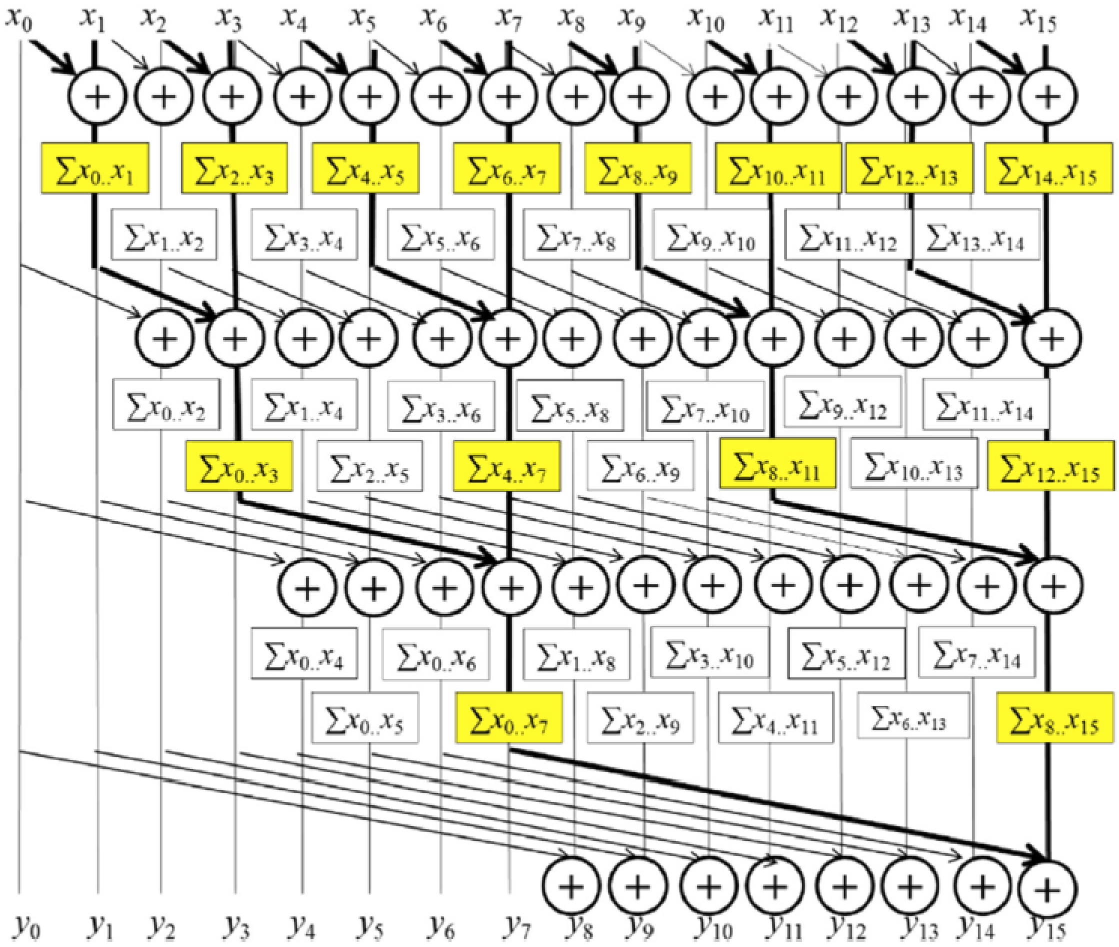


FIGURE 11.2

A parallel inclusive scan algorithm based on Kogge-Stone adder design.

```
 global   void Kogge Stone scan kernel(float *X, float *Y, unsigned int N){
  __shared__ float XY[SECTION SIZE];
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  if(i < N) {
    XY[threadIdx.x] = X[i];
  } else {
    XY[threadIdx.x] = 0.0f;
  }
  for(unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
    __syncthreads();
    float temp;
    if(threadIdx.x >= stride)
      temp = XY[threadIdx.x] + XY[threadIdx.x-stride];
    __ syncthreads();
    if(threadIdx.x >= stride)
      XY[threadIdx.x] = temp;
  }
  if(i < N) {
    Y[i] = XY[threadIdx.x];
  }
}
```

FIGURE 11.3

A Kogge-Stone kernel for inclusive (segmented) scan.

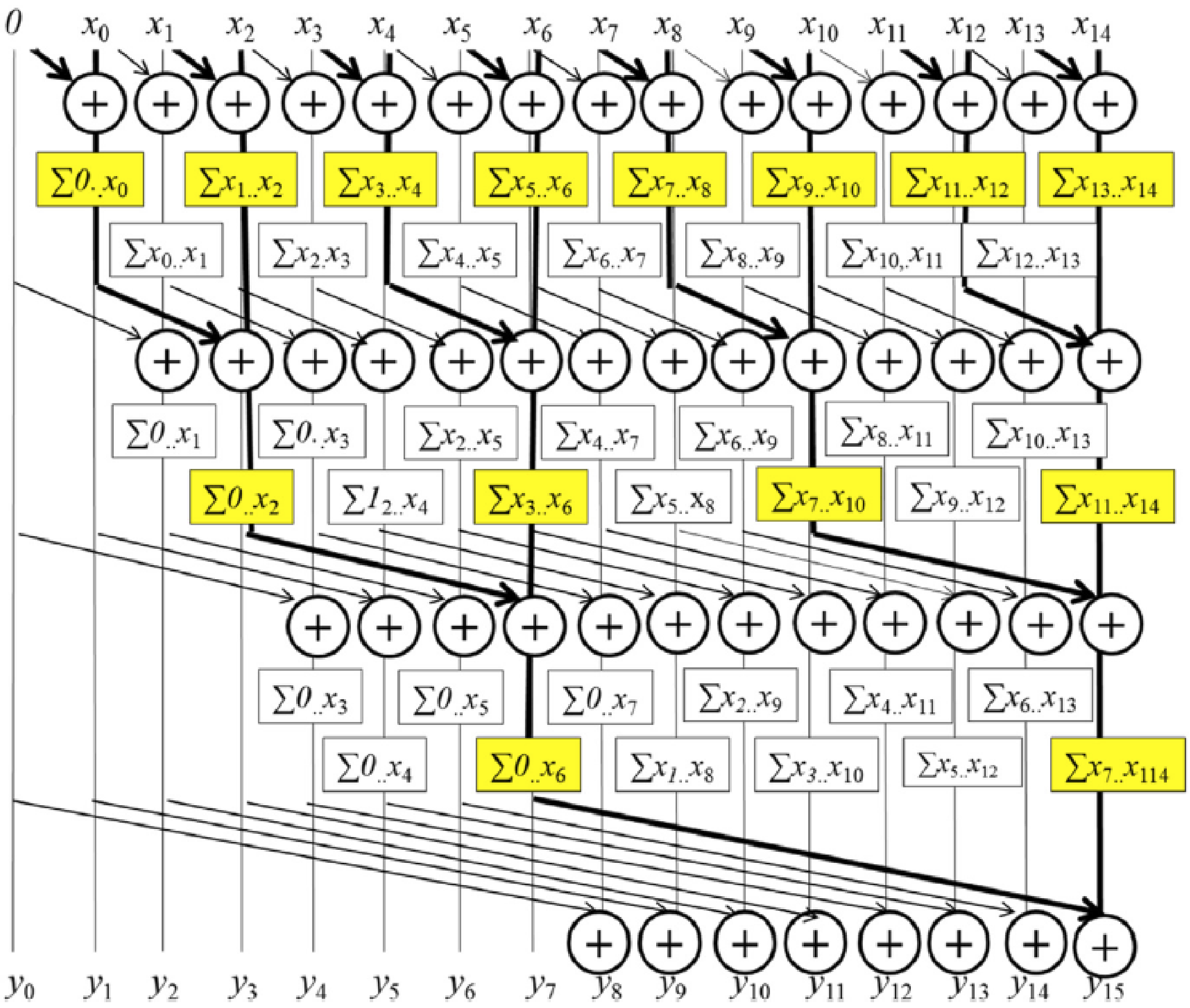


FIGURE 11.4

A parallel exclusive scan algorithm based on Kogge-Stone adder design.

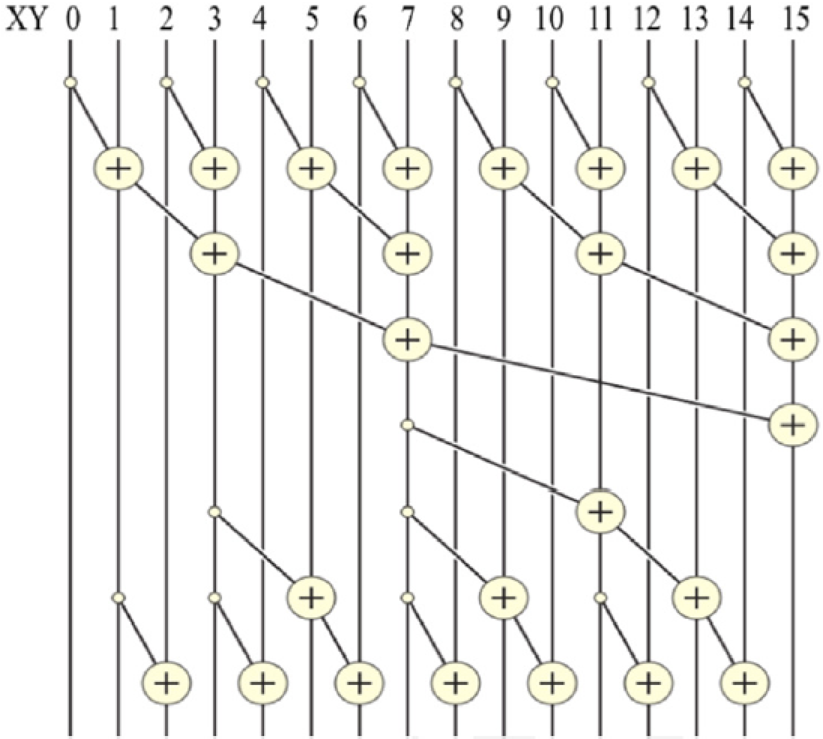
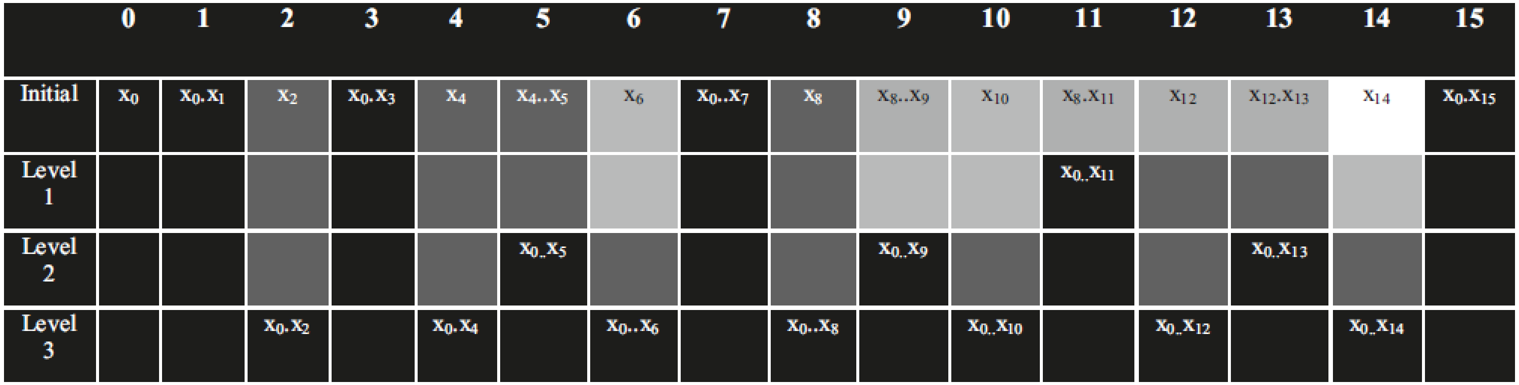


FIGURE 11.5

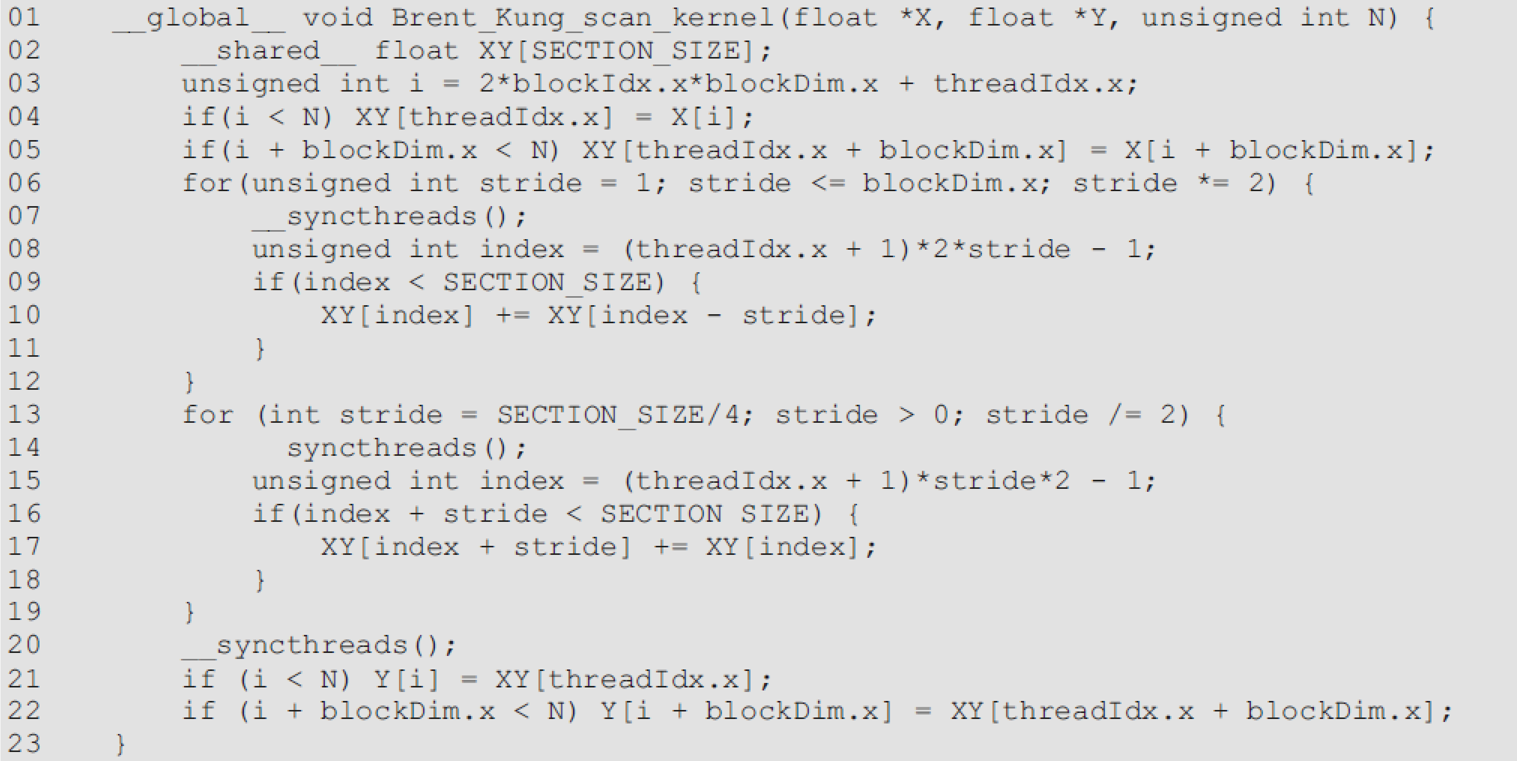
A parallel inclusive scan algorithm based on Brent-Kung adder design.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Initial	x_0	$x_0.x_1$	x_2	$x_0.x_3$	x_4	$x_4..x_5$	x_6	$x_0..x_7$	x_8	$x_8..x_9$	x_{10}	$x_8.x_{11}$	x_{12}	$x_{12}.x_{13}$	x_{14}	$x_0.x_{15}$
Level 1												$x_0..x_{11}$				
Level 2						$x_0..x_5$				$x_0..x_9$				$x_0..x_{13}$		
Level 3			$x_0.x_2$		$x_0.x_4$		$x_0..x_6$		$x_0..x_8$		$x_0..x_{10}$		$x_0..x_{12}$		$x_0..x_{14}$	

FIGURE 11.6

Progression of values in XY after each level of additions in the reverse tree.



```
__global__ void Brent_Kung_scan_kernel(float *X, float *Y, unsigned int N) {
    __shared__ float XY[SECTION_SIZE];
    unsigned int i = 2*blockIdx.x*blockDim.x + threadIdx.x;
    if(i < N) XY[threadIdx.x] = X[i];
    if(i + blockDim.x < N) XY[threadIdx.x + blockDim.x] = X[i + blockDim.x];
    for(unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
        __syncthreads();
        unsigned int index = (threadIdx.x + 1)*2*stride - 1;
        if(index < SECTION_SIZE) {
            XY[index] += XY[index - stride];
        }
    }
    for (int stride = SECTION_SIZE/4; stride > 0; stride /= 2) {
          syncthreads();
        unsigned int index = (threadIdx.x + 1)*stride*2 - 1;
        if(index + stride < SECTION SIZE) {
            XY[index + stride] += XY[index];
        }
    }
    __syncthreads();
    if (i < N) Y[i] = XY[threadIdx.x];
    if (i + blockDim.x < N) Y[i + blockDim.x] = XY[threadIdx.x + blockDim.x];
}
```

FIGURE 11.7

A Brent-Kung kernel for inclusive (segmented) scan.

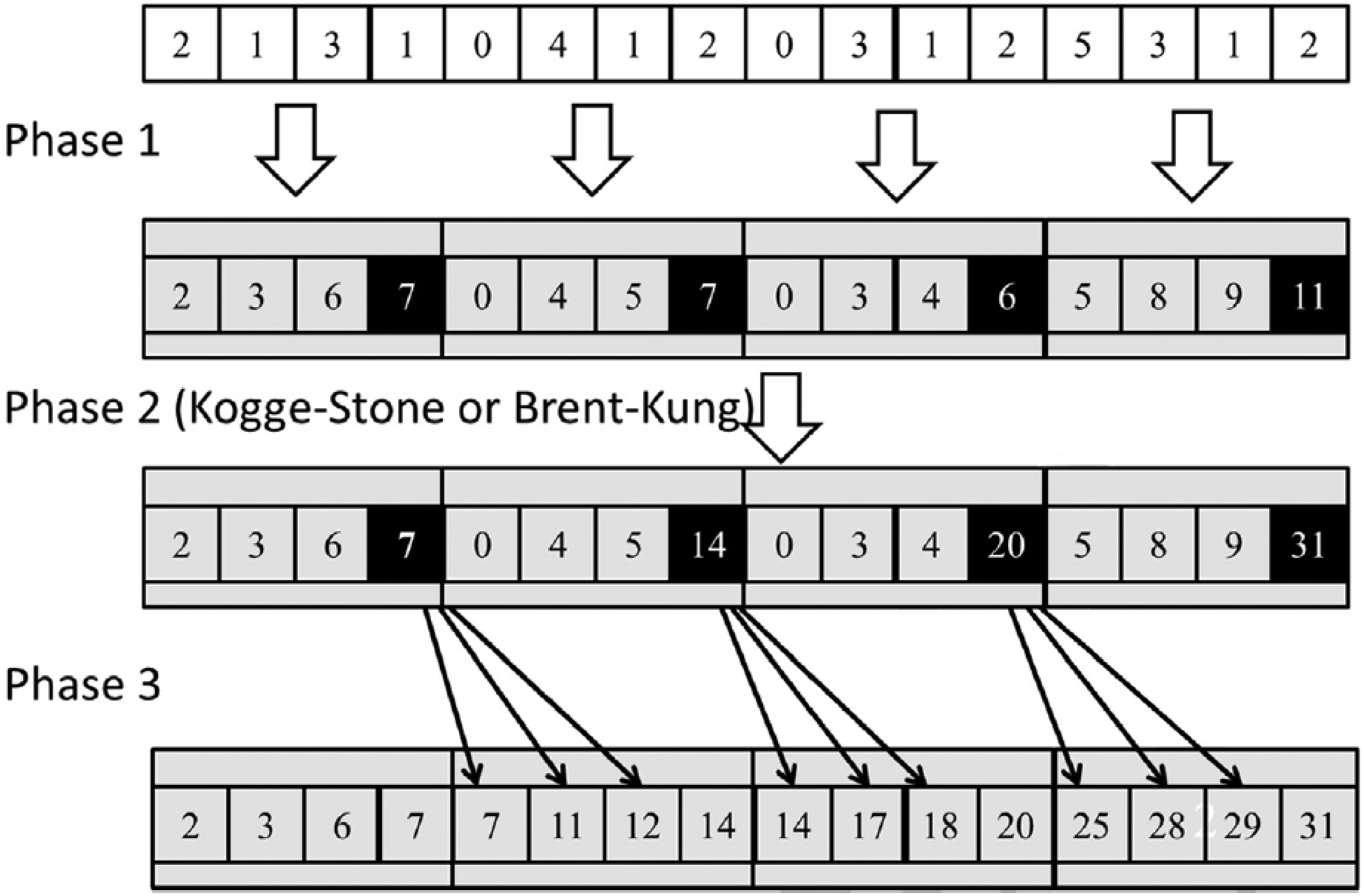


FIGURE 11.8

A three-phase parallel scan for higher work efficiency.

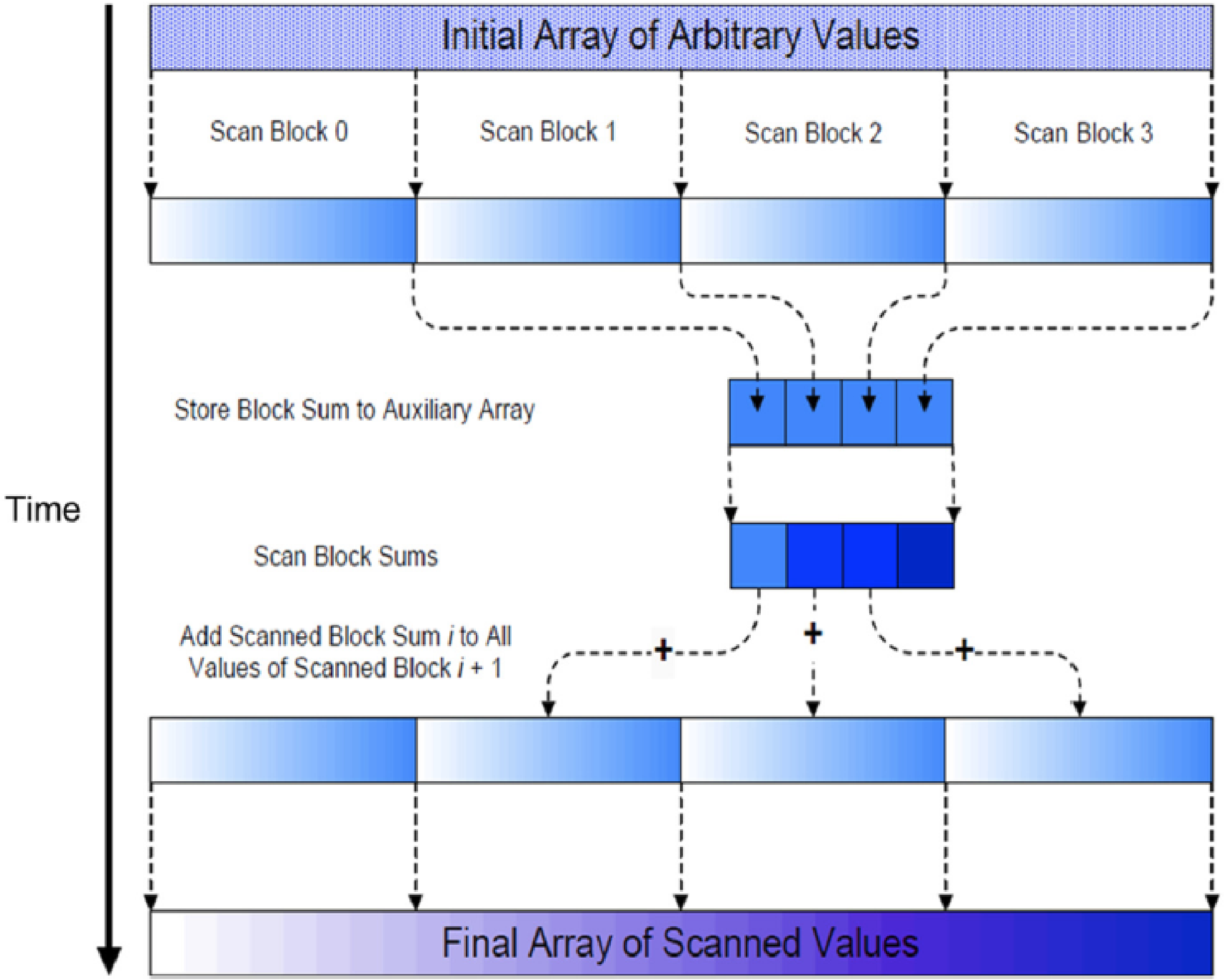

Initial Array of Arbitrary Values
Scan Block 0
Scan Block 1
Scan Block 2
Scan Block 3
Store Block Sum to Auxiliary Array
Time
Scan Block Sums
Add Scanned Block Sum *i* to All Values of Scanned Block *i* + 1
+
+
+
Final Array of Scanned Values

FIGURE 11.9

A hierarchical scan for arbitrary length inputs.

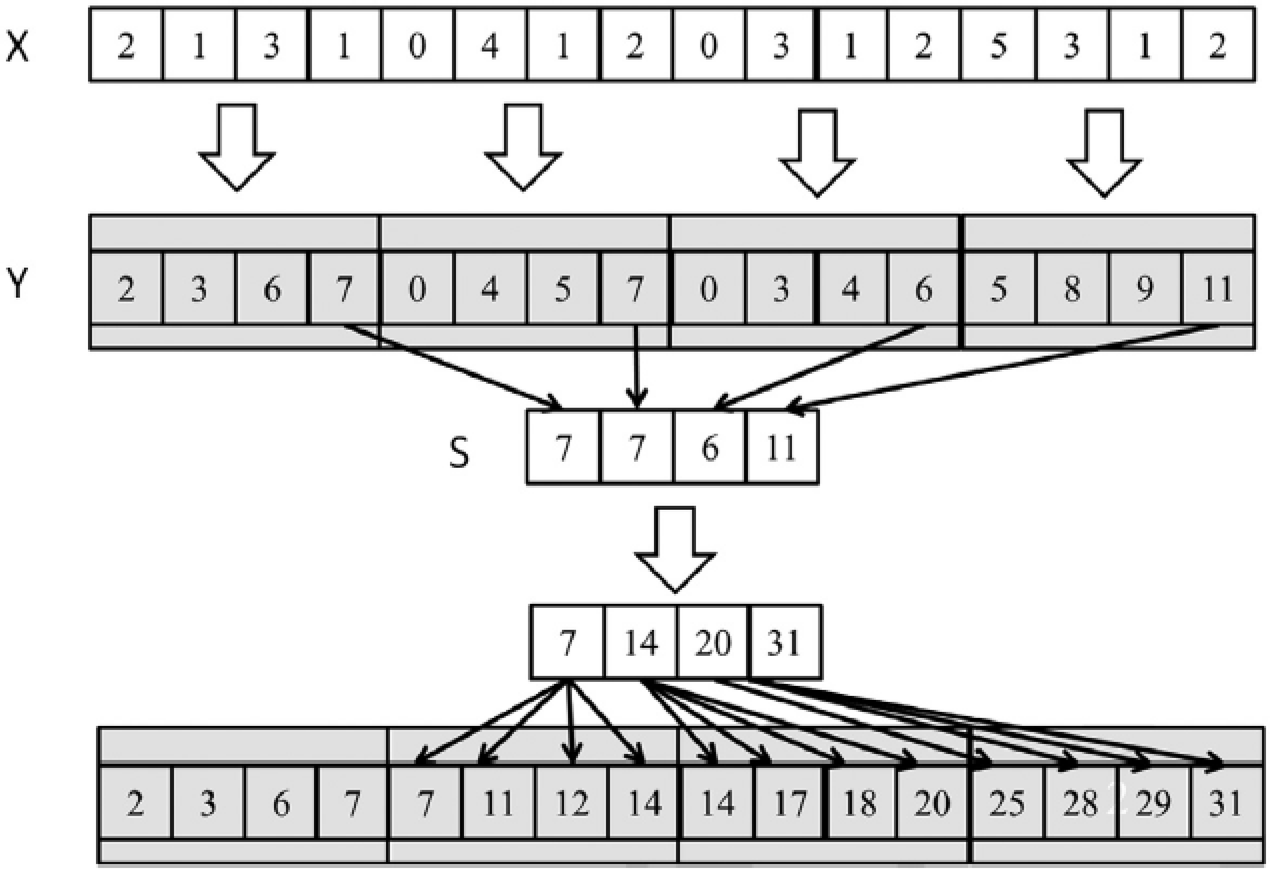
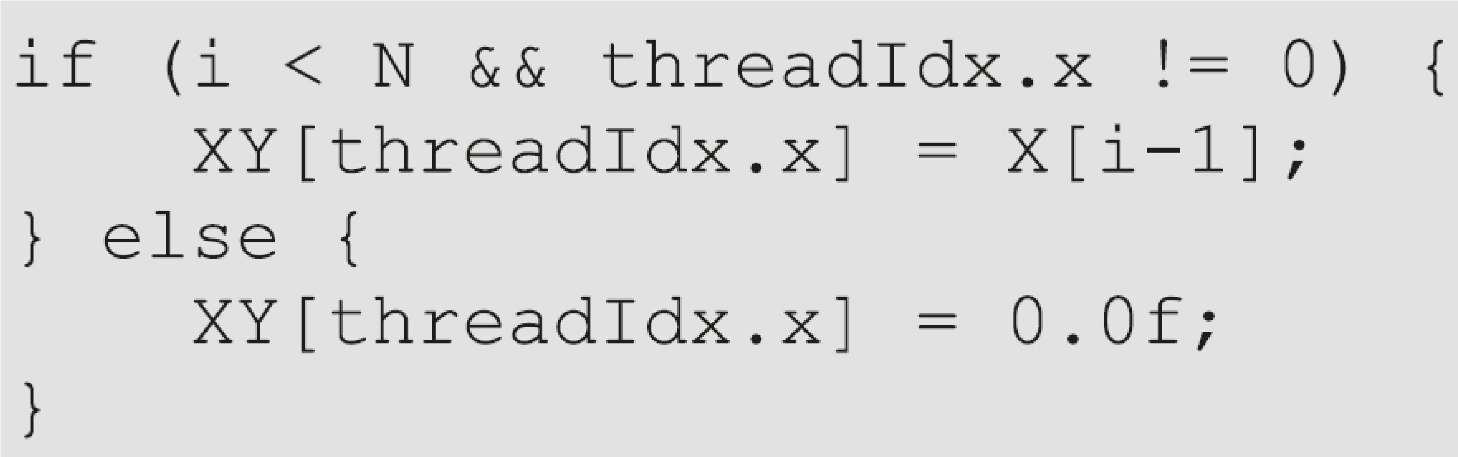


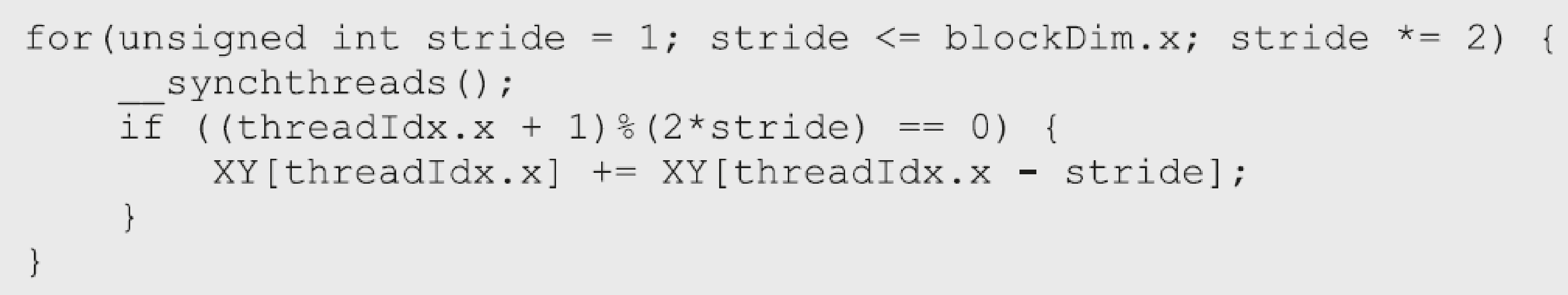
FIGURE 11.10

An example of hierarchical scan.



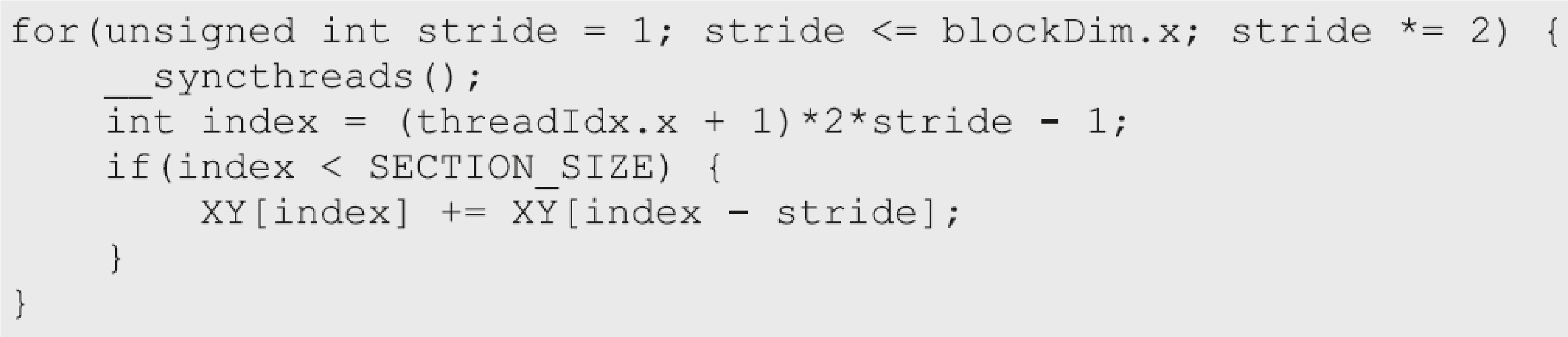
```
if (i < N && threadIdx.x != 0) {
    XY[threadIdx.x] = X[i-1];
} else {
    XY[threadIdx.x] = 0.0f;
}
```

In-text figure 1



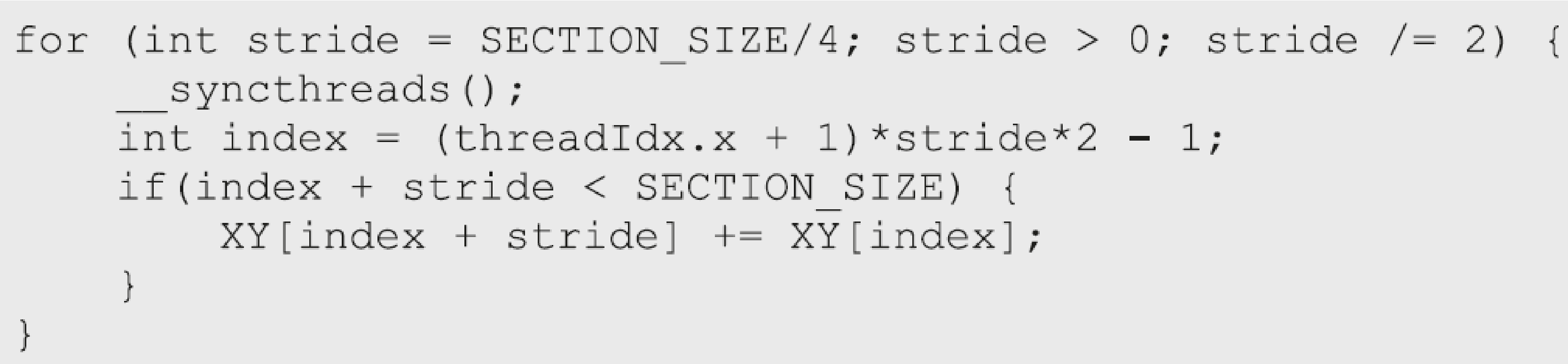
```
for(unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
    __synchthreads();
    if ((threadIdx.x + 1)%(2*stride) == 0) {
        XY[threadIdx.x] += XY[threadIdx.x - stride];
    }
}
```

In-text figure 2



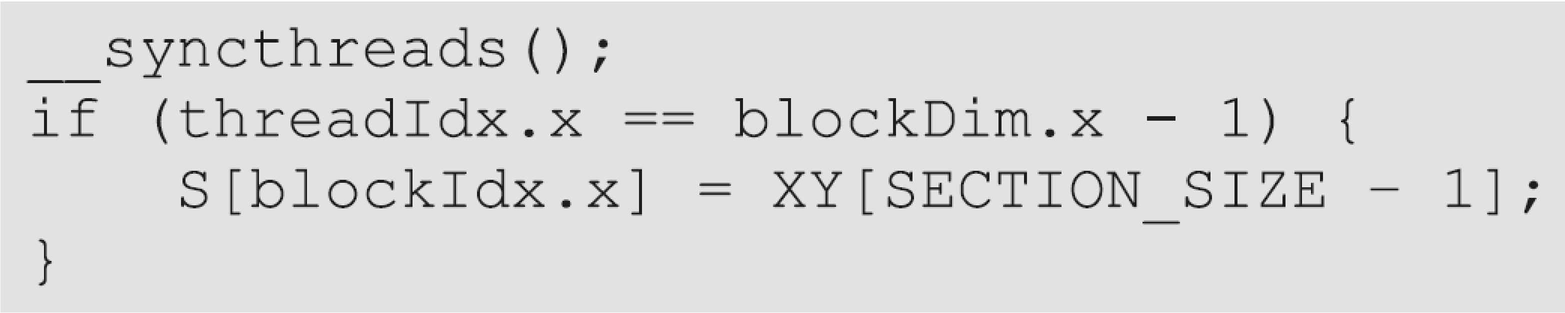
```
for(unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
    __syncthreads();
    int index = (threadIdx.x + 1)*2*stride - 1;
    if(index < SECTION_SIZE) {
        XY[index] += XY[index - stride];
    }
}
```

In-text figure 3



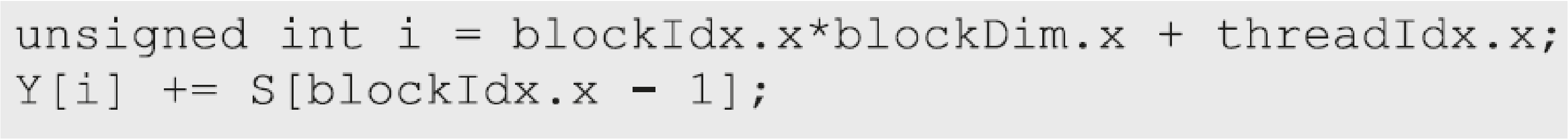
```
for (int stride = SECTION_SIZE/4; stride > 0; stride /= 2) {
    __syncthreads();
    int index = (threadIdx.x + 1)*stride*2 - 1;
    if(index + stride < SECTION_SIZE) {
        XY[index + stride] += XY[index];
    }
}
```

In-text figure 4



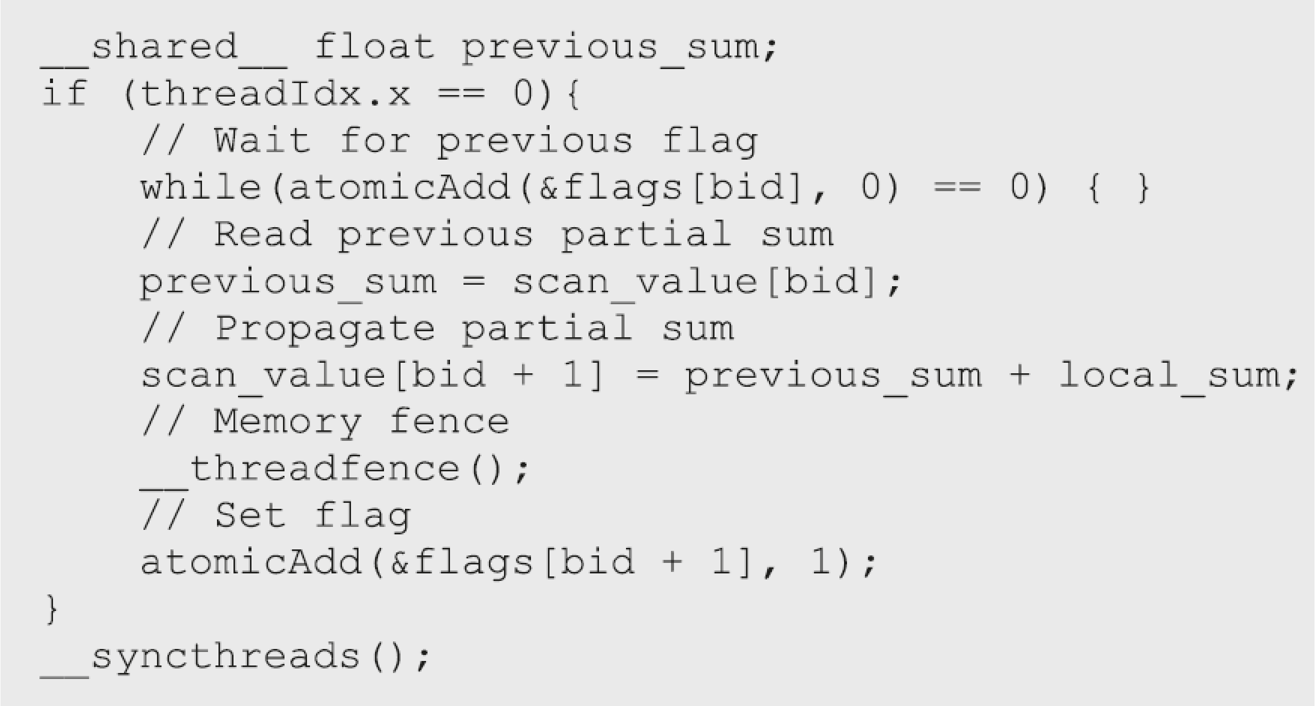
```
__syncthreads();
if (threadIdx.x == blockDim.x - 1) {
    S[blockIdx.x] = XY[SECTION_SIZE - 1];
}
```

In-text figure 5



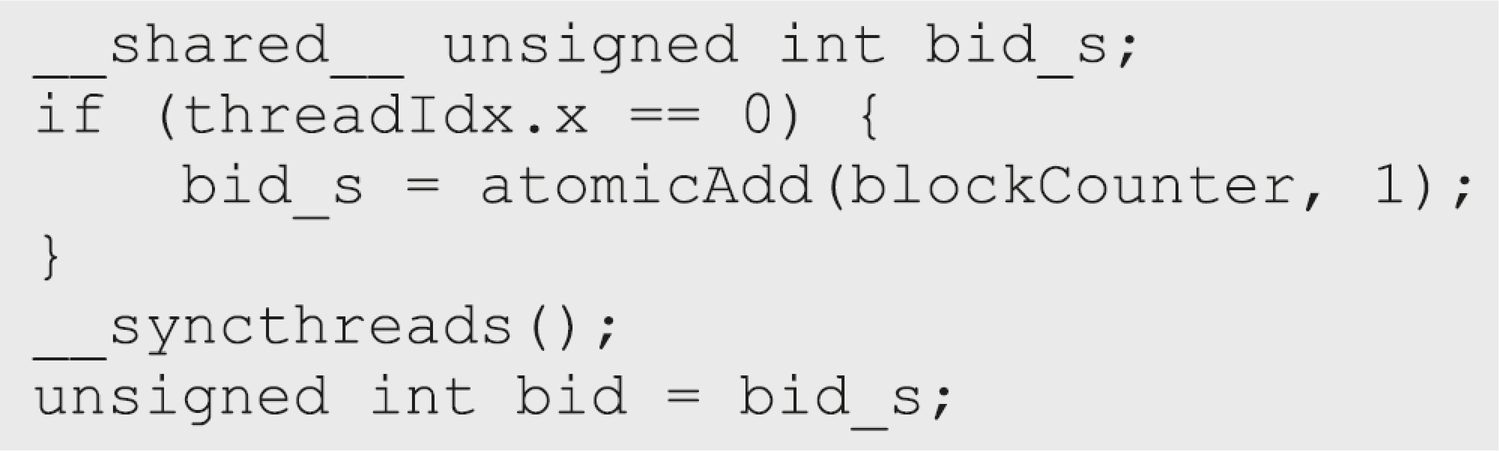
```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
Y[i] += S[blockIdx.x - 1];
```

In-text figure 6



```
__shared__ float previous_sum;
if (threadIdx.x == 0){
    // Wait for previous flag
    while(atomicAdd(&flags[bid], 0) == 0) { }
    // Read previous partial sum
    previous_sum = scan_value[bid];
    // Propagate partial sum
    scan_value[bid + 1] = previous_sum + local_sum;
    // Memory fence
    __threadfence();
    // Set flag
    atomicAdd(&flags[bid + 1], 1);
}
__syncthreads();
```

In-text figure 7



```
__shared__ unsigned int bid_s;
if (threadIdx.x == 0) {
    bid_s = atomicAdd(blockCounter, 1);
}
__syncthreads();
unsigned int bid = bid_s;
```

In-text figure 8