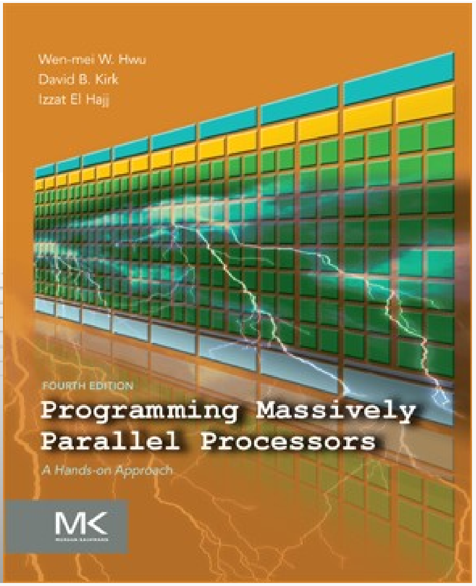
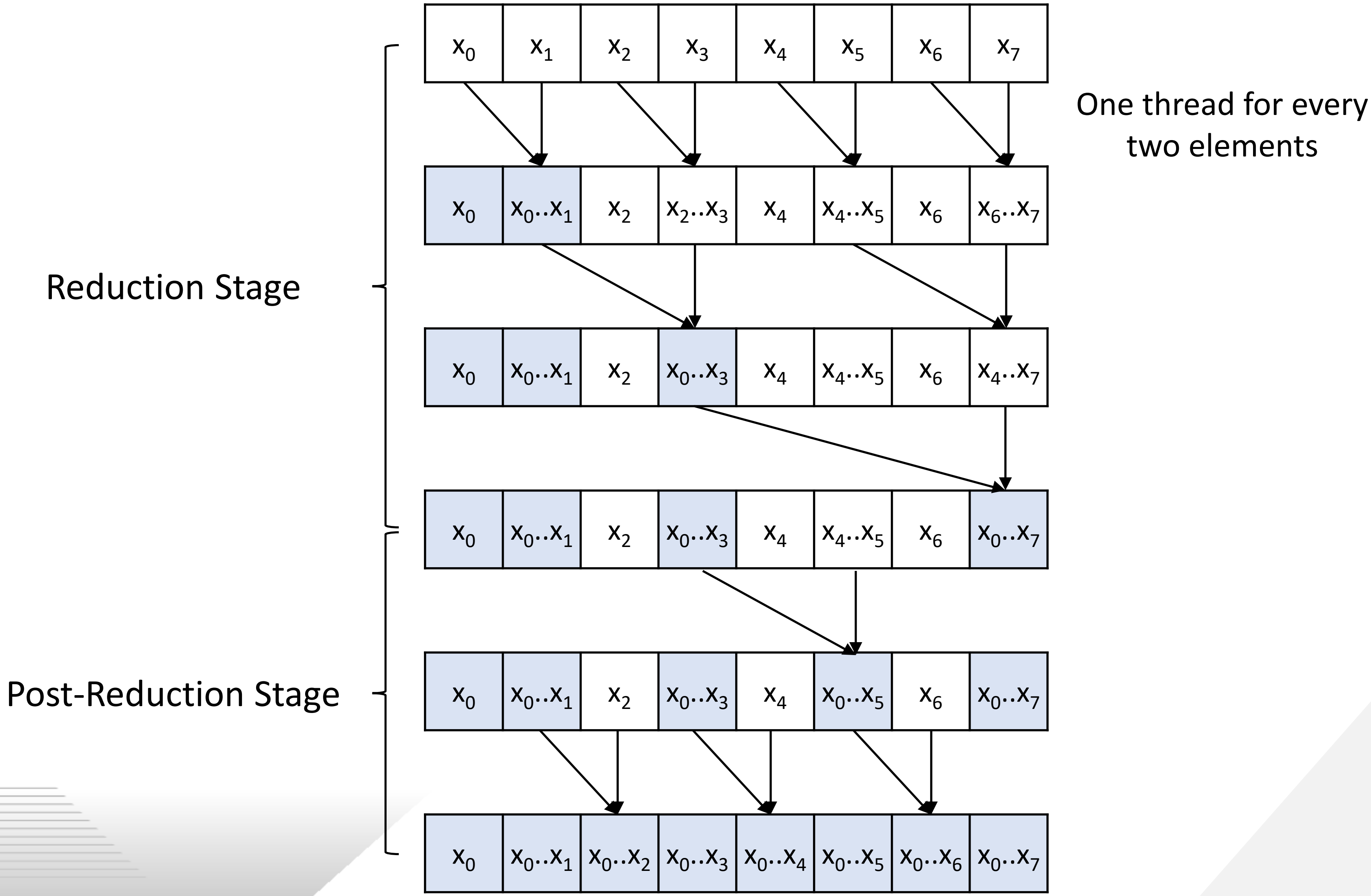
Programming Massively Parallel Processors

A Hands-on Approach

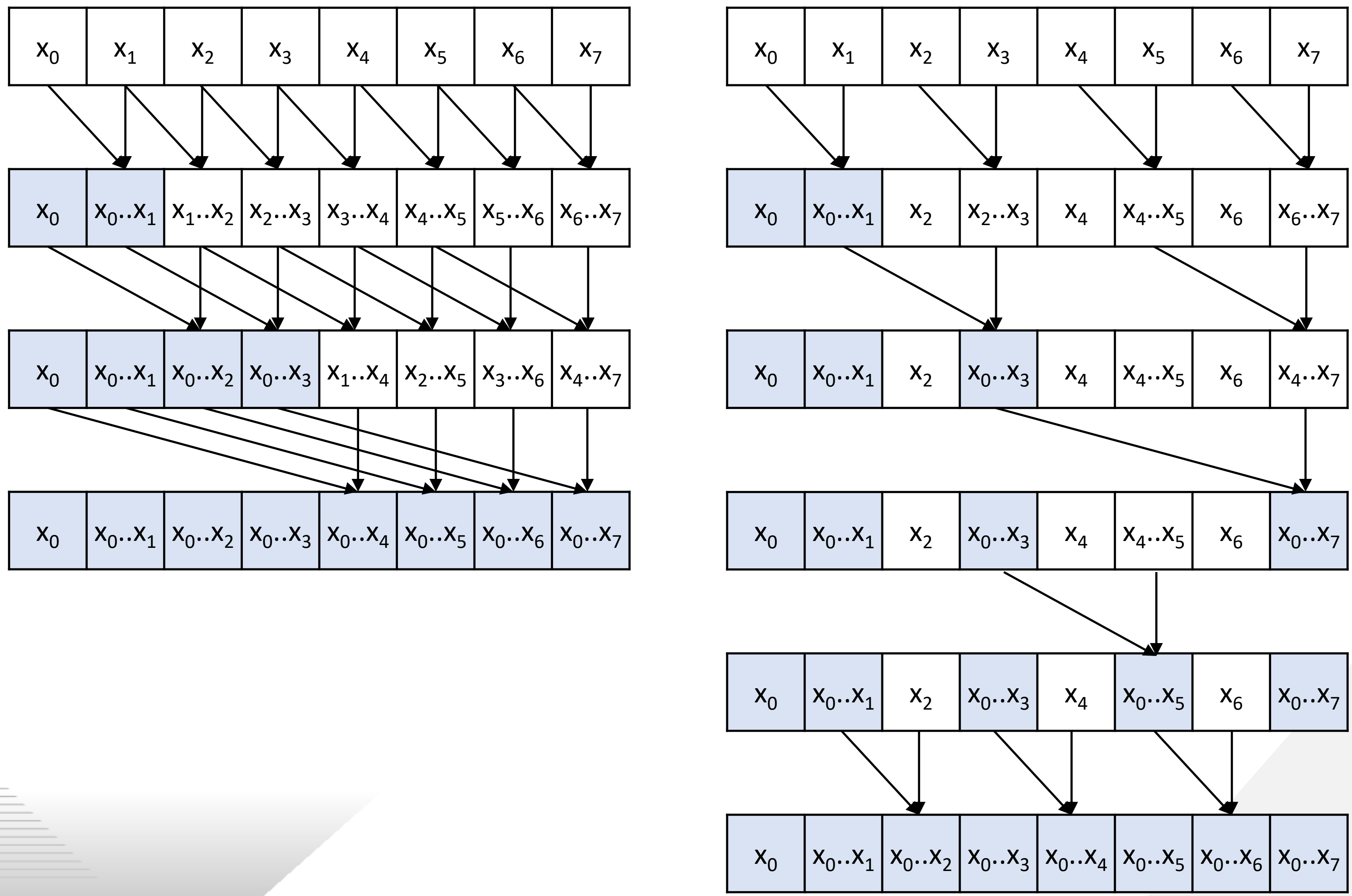
CHAPTER 11 (PART 2) › Prefix Sum (Scan)



Brent-Kung Parallel (Inclusive) Scan



Kogge-Stone vs. Brent-Kung



Work Efficiency

- Recall: Kogge-Stone
  - **log(N) steps**
  - **O(N*log(N)) operations**
- Brent-Kung
  - Reduction stage:
    - log(N) steps
    - N/2 + N/4 + … + 4 + 2 + 1 = N-1 operations
  - Post-Reduction stage:
    - log(N)-1 steps
    - (2-1) + (4-1) + … + (N/2-1) = (N-2) - (log(N)-1)
  - Total:
    - **2*log(N)-1 steps**
    - (N-1) + (N-2) - (log(N)-1) = 2*N – log(N) – 2 = **O(N) operations**
- Brent-Kung takes **more steps** but is **more work-efficient**

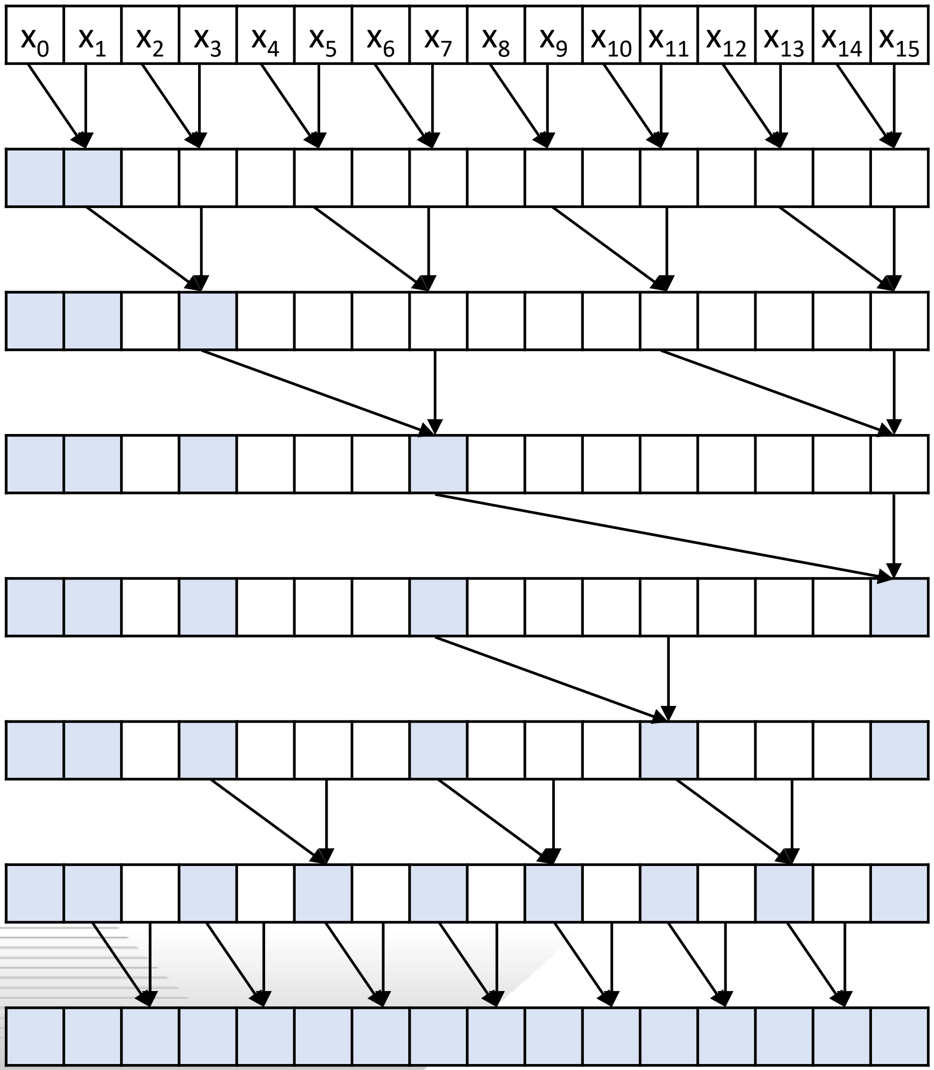
Optimizations

- Using **shared memory**
  - Similar to Kogge-Stone
  - Also enables **coalescing** of global memory loads
    - In Kogge-Stone, they were already coalesced

- No need for **double-buffering**
  - Unlike Kogge-Stone, no data element is read and written by different threads on the same iteration

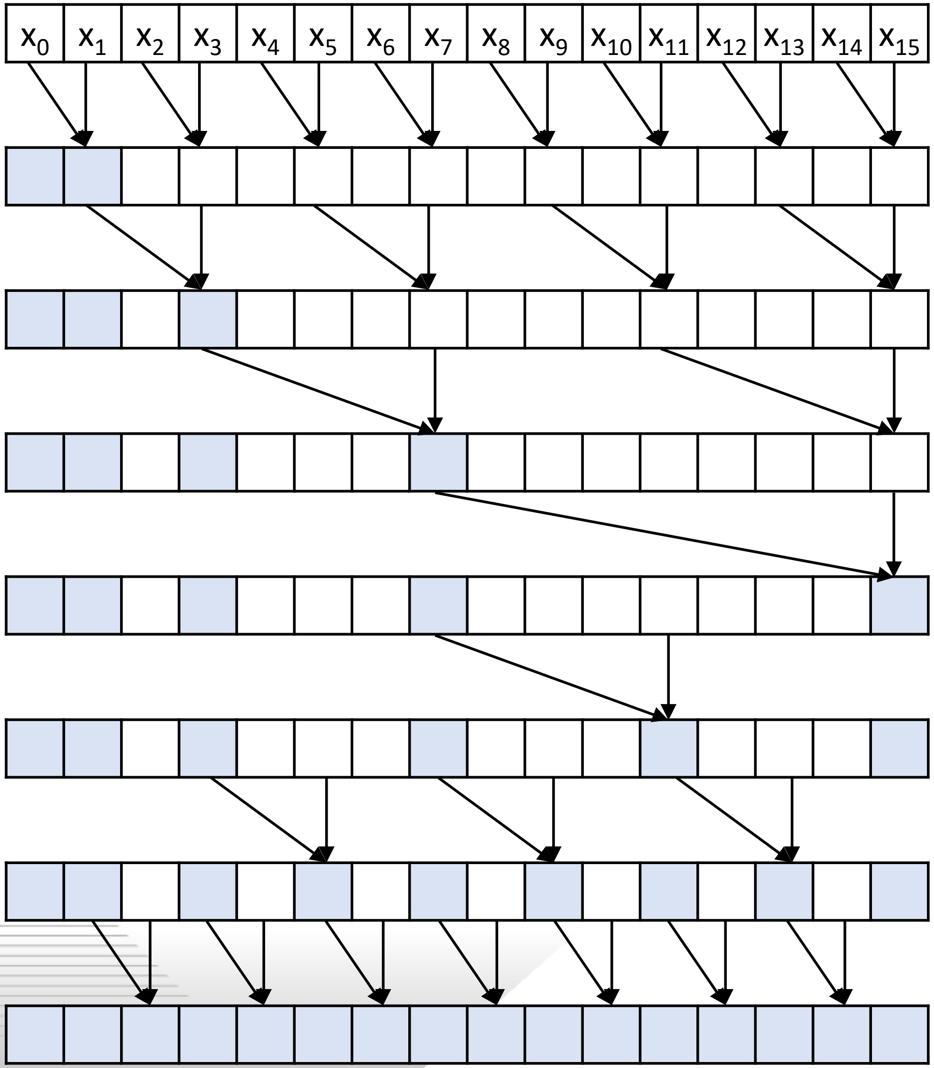
- Minimizing **control divergence**
  - Do not assign threads to specific data elements
  - Re-index threads on every iteration to different elements
    - If there are M operations, assign them to the first M threads based on the thread index and stride value

Re-index Threads to Minimize Divergence



stride (s) \ thread (t)	0	1	2	3	...
1					
2					
4					
8					
4					
2					
1					

Re-index Threads to Minimize Divergence

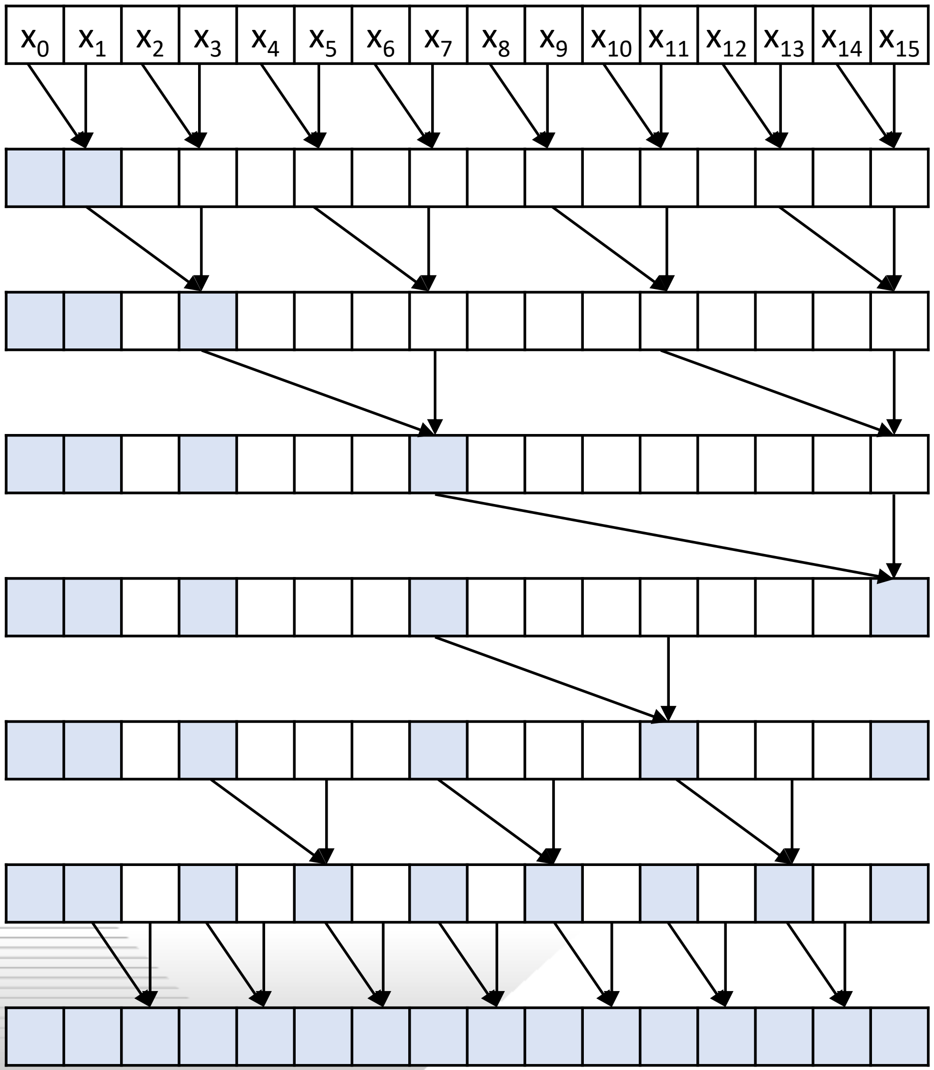


thread (t)

stride (s)	**0**	**1**	**2**	**3**	**...**
1	1	3	5	7	...
2	3	7	11	15	
4	7	15			
8	15				
4					
2					
1					

index of thread's right source element and destination element

Re-index Threads to Minimize Divergence

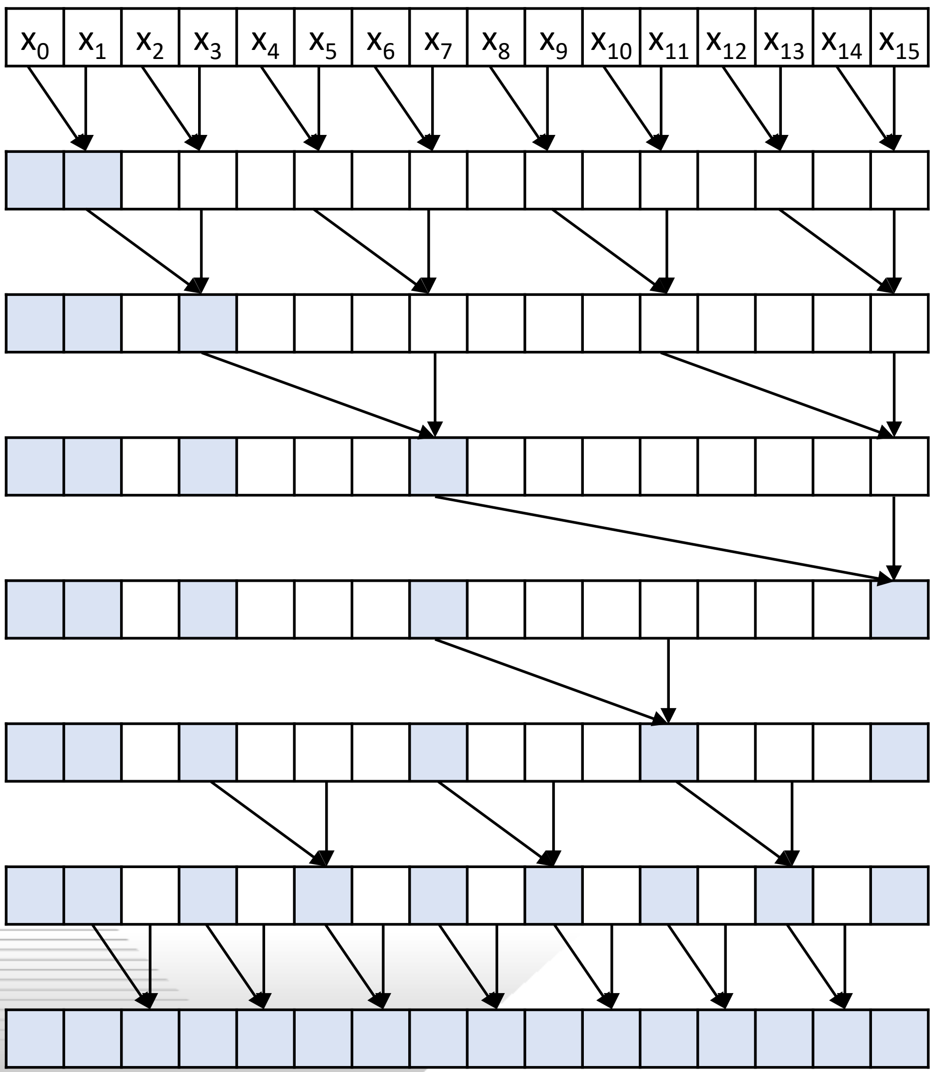


thread (t)

stride (s)	0	1	2	3	...
1	1	3	5	7	...
2	3	7	11	15	
4	7	15			
8	15				
4	7				
2	3	7	11		
1	1	3	5	7	...

index of thread’s left source element

Re-index Threads to Minimize Divergence

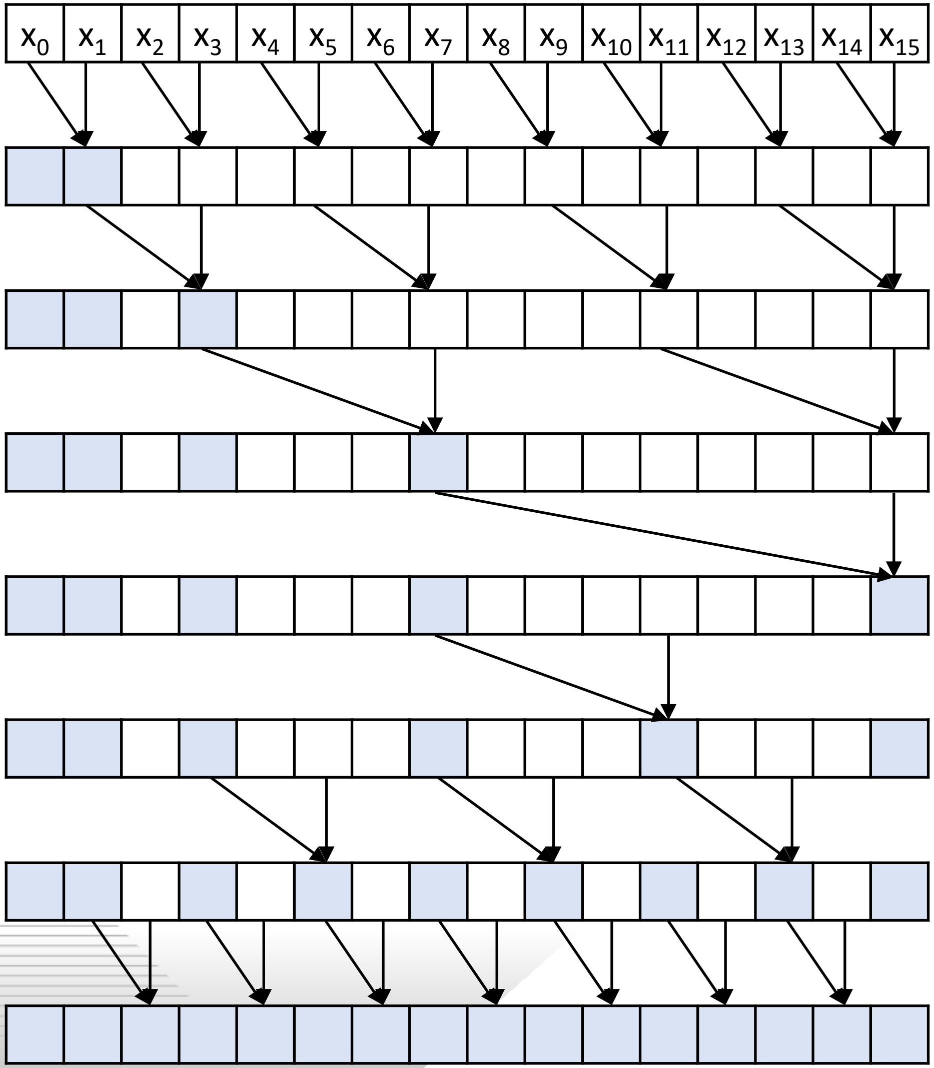


thread (t)

stride (s)	0	1	2	3	...
1	1	3	5	7	...
2	3	7	11	15	
4	7	15			
8	15				
4	7				
2	3	7	11		
1	1	3	5	7	...

How to express index in terms of s and t?

Re-index Threads to Minimize Divergence

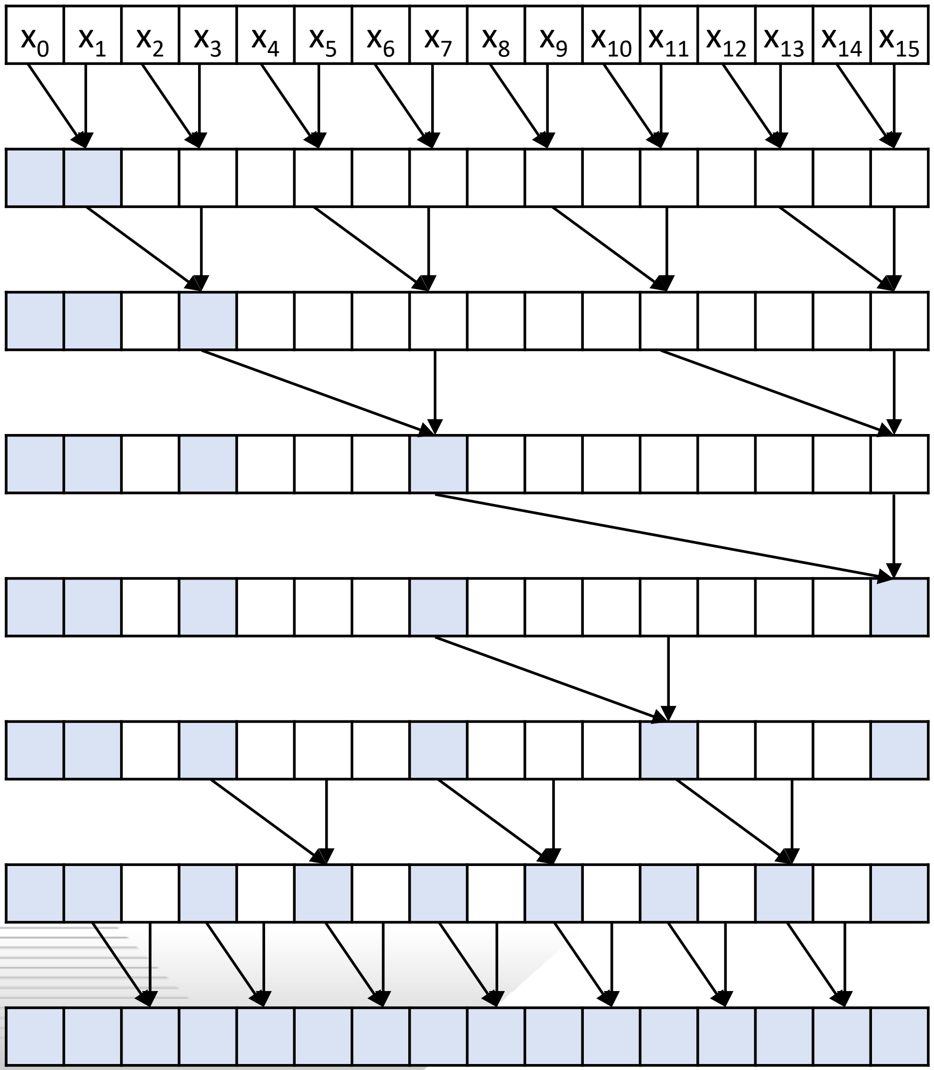


thread (t)

stride (s)	0	1	2	3	...
1	2-1	4-1	6-1	8-1	...
2	4-1	8-1	12-1	16-1	
4	8-1	16-1			
8	16-1				
4	8-1				
2	4-1	8-1	12-1		
1	2-1	4-1	6-1	8-1	...

Pattern: indexes are multiples of 2*s minus 1

Re-index Threads to Minimize Divergence

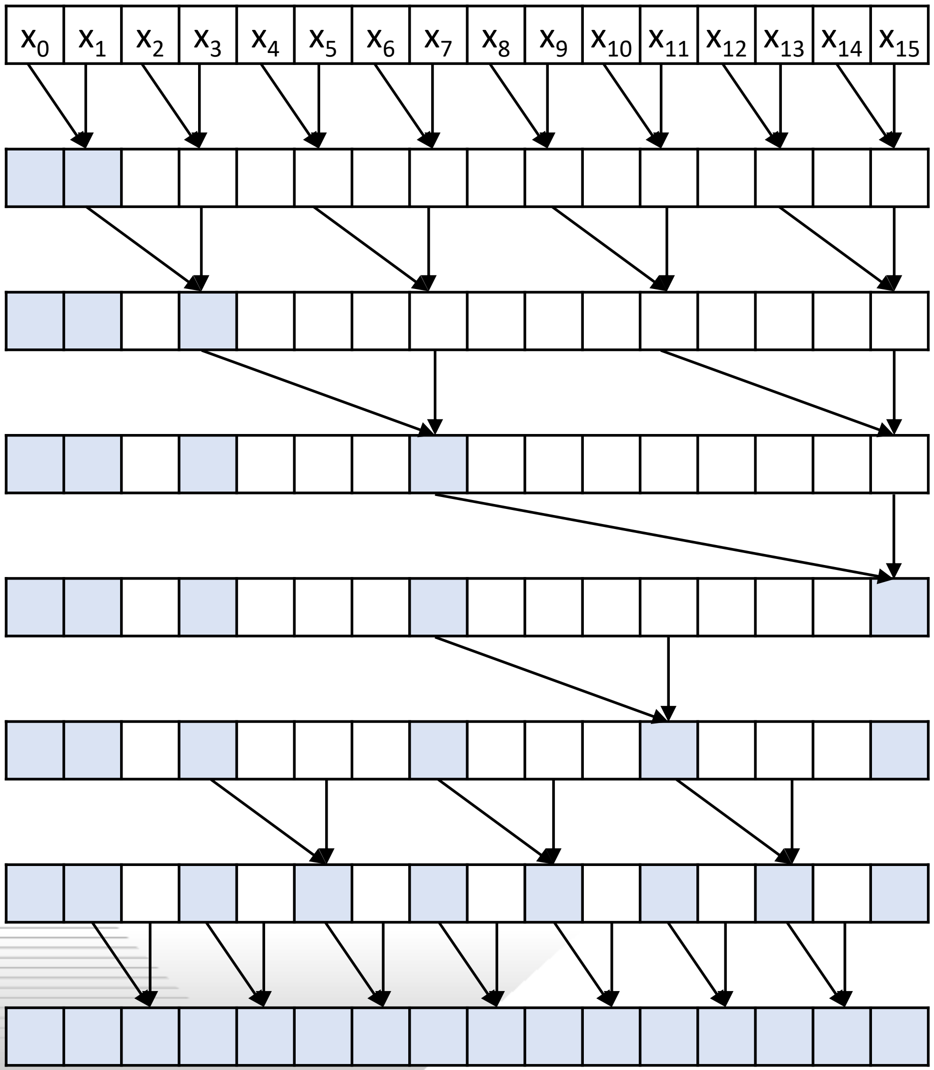


thread (t)

stride (s)	0	1	2	3	...
1	1*2*s-1	2*2*s-1	3*2*s-1	4*2*s-1	...
2	1*2*s-1	2*2*s-1	3*2*s-1	4*2*s-1	
4	1*2*s-1	2*2*s-1			
8	1*2*s-1				
4	1*2*s-1				
2	1*2*s-1	2*2*s-1	3*2*s-1		
1	1*2*s-1	2*2*s-1	3*2*s-1	4*2*s-1	...

Pattern: indexes are the **(t+1)th** multiple of **2*s**

Re-index Threads to Minimize Divergence



thread (t)

stride (s)	0	1	2	3	…
1	(t+1)*2*s-1	(t+1)*2*s-1	(t+1)*2*s-1	(t+1)*2*s-1	…
2	(t+1)*2*s-1	(t+1)*2*s-1	(t+1)*2*s-1	(t+1)*2*s-1	
4	(t+1)*2*s-1	(t+1)*2*s-1			
8	(t+1)*2*s-1				
4	(t+1)*2*s-1				
2	(t+1)*2*s-1	(t+1)*2*s-1	(t+1)*2*s-1		
1	(t+1)*2*s-1	(t+1)*2*s-1	(t+1)*2*s-1	(t+1)*2*s-1	…

Every thread processes element (t+1)*2*s-1…

…if the destination is in bounds

Brent Kung Inclusive Scan Code

```
__global__ void scan_kernel(float* input, float* output, float* partialSums, unsigned int N) {

    unsigned int segment = 2*blockIdx.x*blockDim.x;

    __shared__ float buffer_s[2*BLOCK_DIM];
    buffer_s[threadIdx.x] = input[segment + threadIdx.x];
    buffer_s[threadIdx.x + BLOCK_DIM] = input[segment + threadIdx.x + BLOCK_DIM];
    __syncthreads();

    // First tree
    for(unsigned int stride = 1; stride <= BLOCK_DIM; stride *= 2) {
        unsigned int i = (threadIdx.x + 1)*2*stride - 1;
        if(i < 2*BLOCK_DIM) {
            buffer_s[i] += buffer_s[i - stride];
        }
        __syncthreads();
    }

    // Second tree
    for(unsigned int stride = BLOCK_DIM/2; stride >= 1; stride /= 2) {
        unsigned int i = (threadIdx.x + 1)*2*stride - 1;
        if(i + stride < 2*BLOCK_DIM) {
            buffer_s[i + stride] += buffer_s[i];
        }
        __syncthreads();
    }

    // Store partial sum
    if(threadIdx.x == 0) {
        partialSums[blockIdx.x] = buffer_s[2*BLOCK_DIM - 1];
    }

    // Store output
    output[segment + threadIdx.x] = buffer_s[threadIdx.x];
    output[segment + threadIdx.x + BLOCK_DIM] = buffer_s[threadIdx.x + BLOCK_DIM];

}
```

Work Efficiency (the Reality)

- While Brent-Kung has higher theoretical work-efficiency than Kogge-Stone, in practice, its actual resource consumption on GPUs after accounting for inactive threads is O(N*log(N))

- Performance of Brent-Kung on GPUs is similar or may even be worse than Kogge-Stone

- Still is an interesting case to study

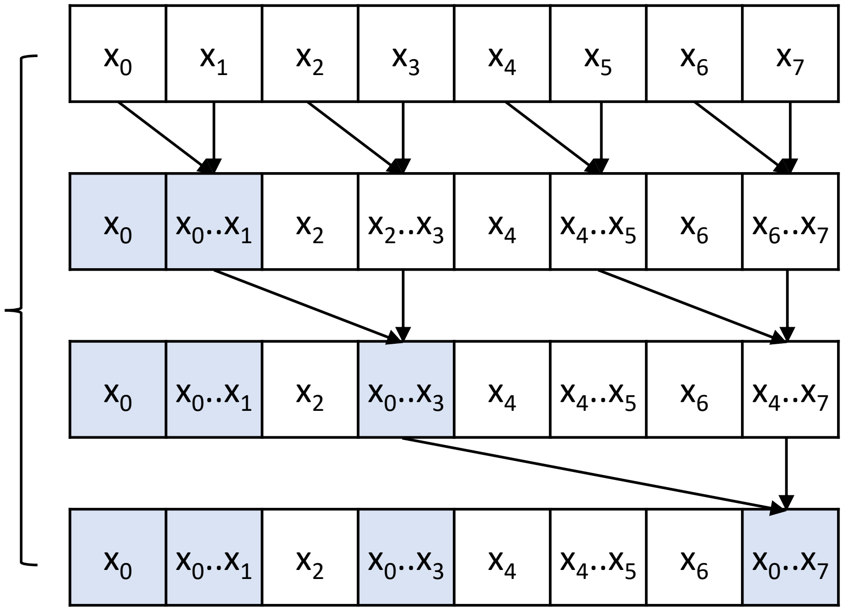
Exclusive Scan

- First approach: formulate as an inclusive scan by shifting input elements when loading them
  - Similar to what we did with the Kogge-Stone approach

- Second approach: use a different post-reduction stage
  - Incurs one more step compared to the inclusive scan post-reduction stage

Brent-Kung Parallel (Exclusive) Scan

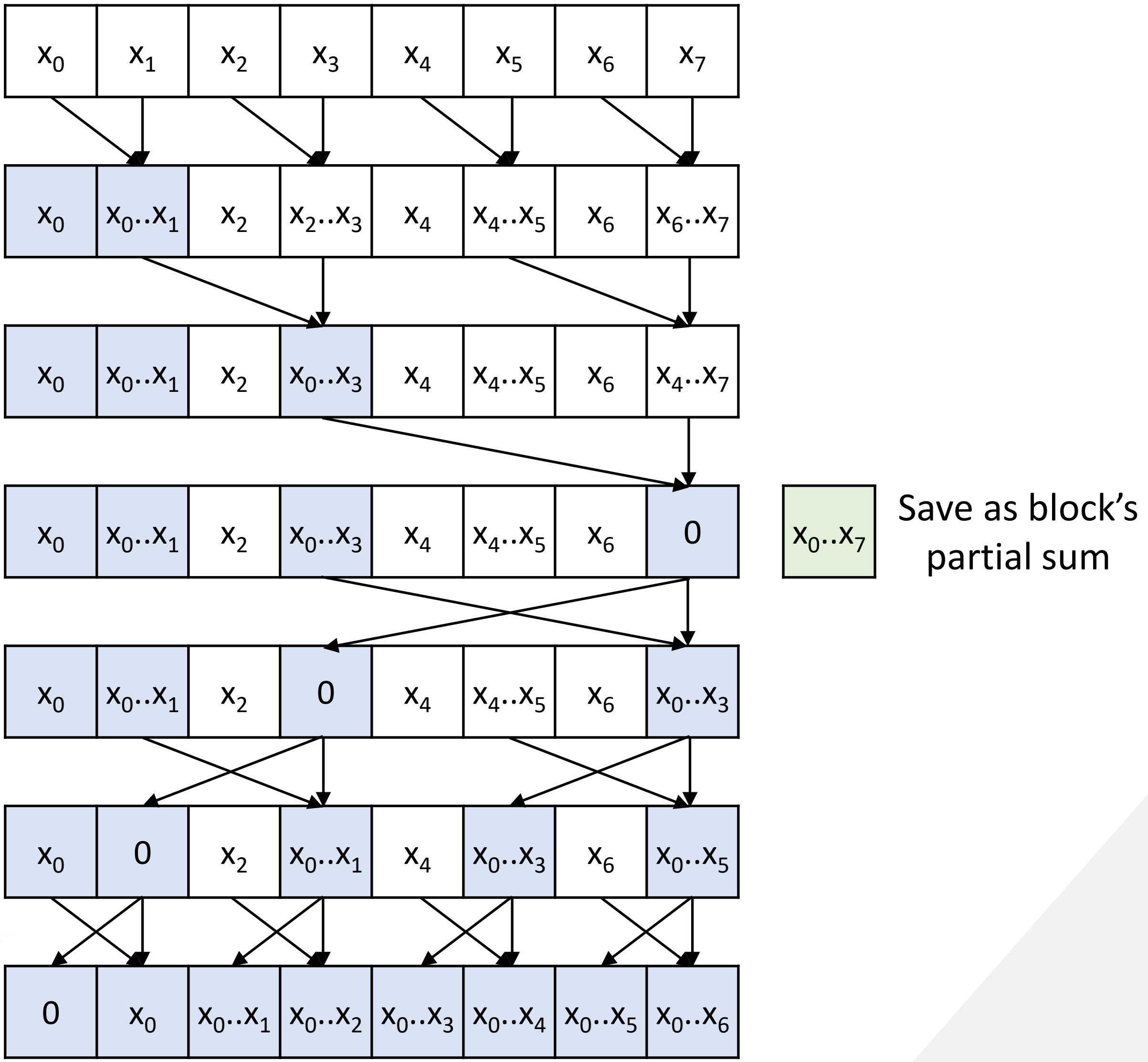
Reduction Stage
(same as inclusive)



Brent-Kung Parallel (Exclusive) Scan

Reduction Stage
(same as inclusive)

Post-Reduction Stage
(for exclusive)



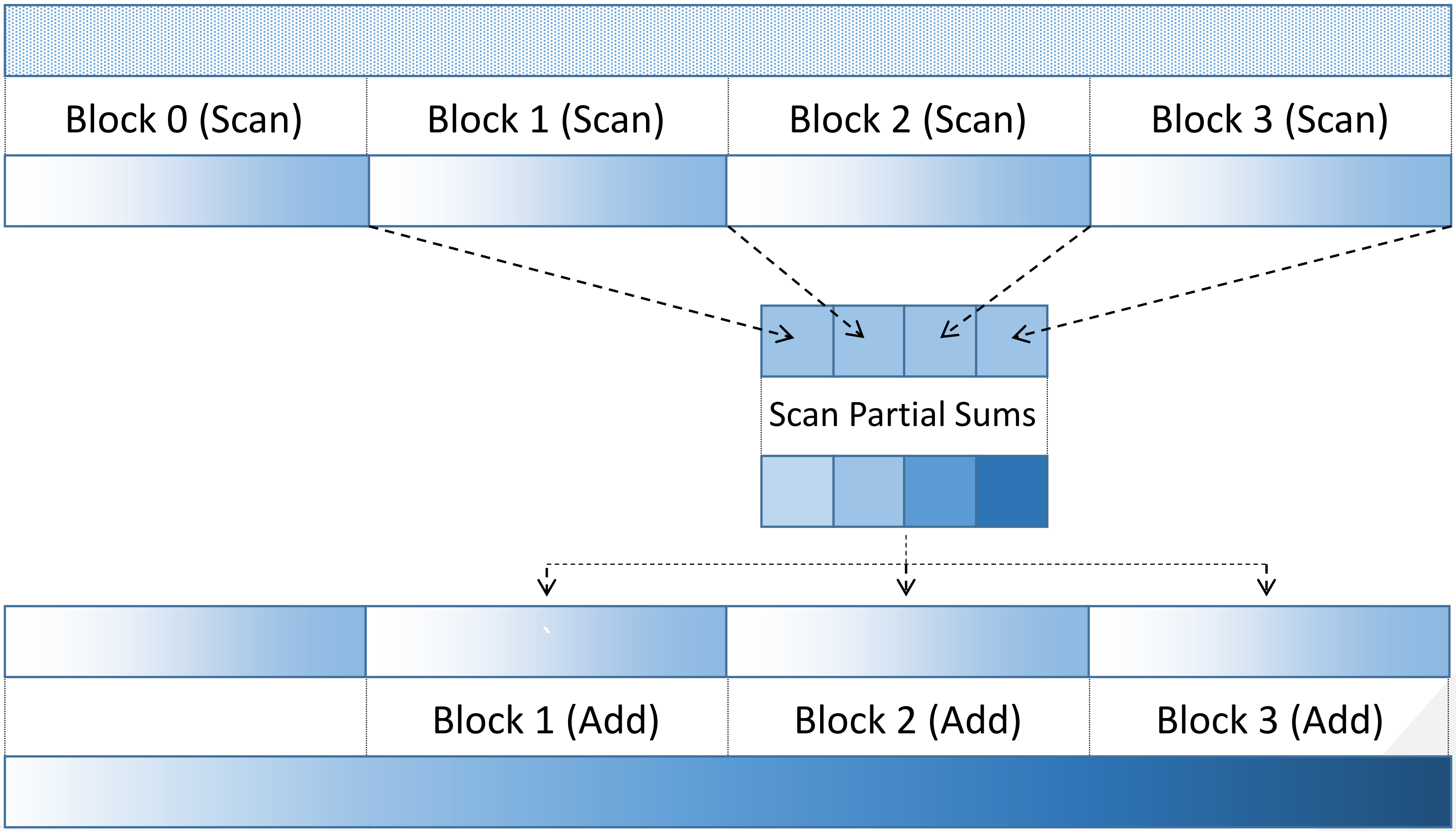
Thread Coarsening

- Parallelizing scan incurs the overhead of lowering work efficiency
  - It also incurs the overhead of barrier synchronization every step and control divergence in the final steps (similar to reduction)

- If resources are insufficient, the hardware will serialize the thread blocks, incurring overhead unnecessarily

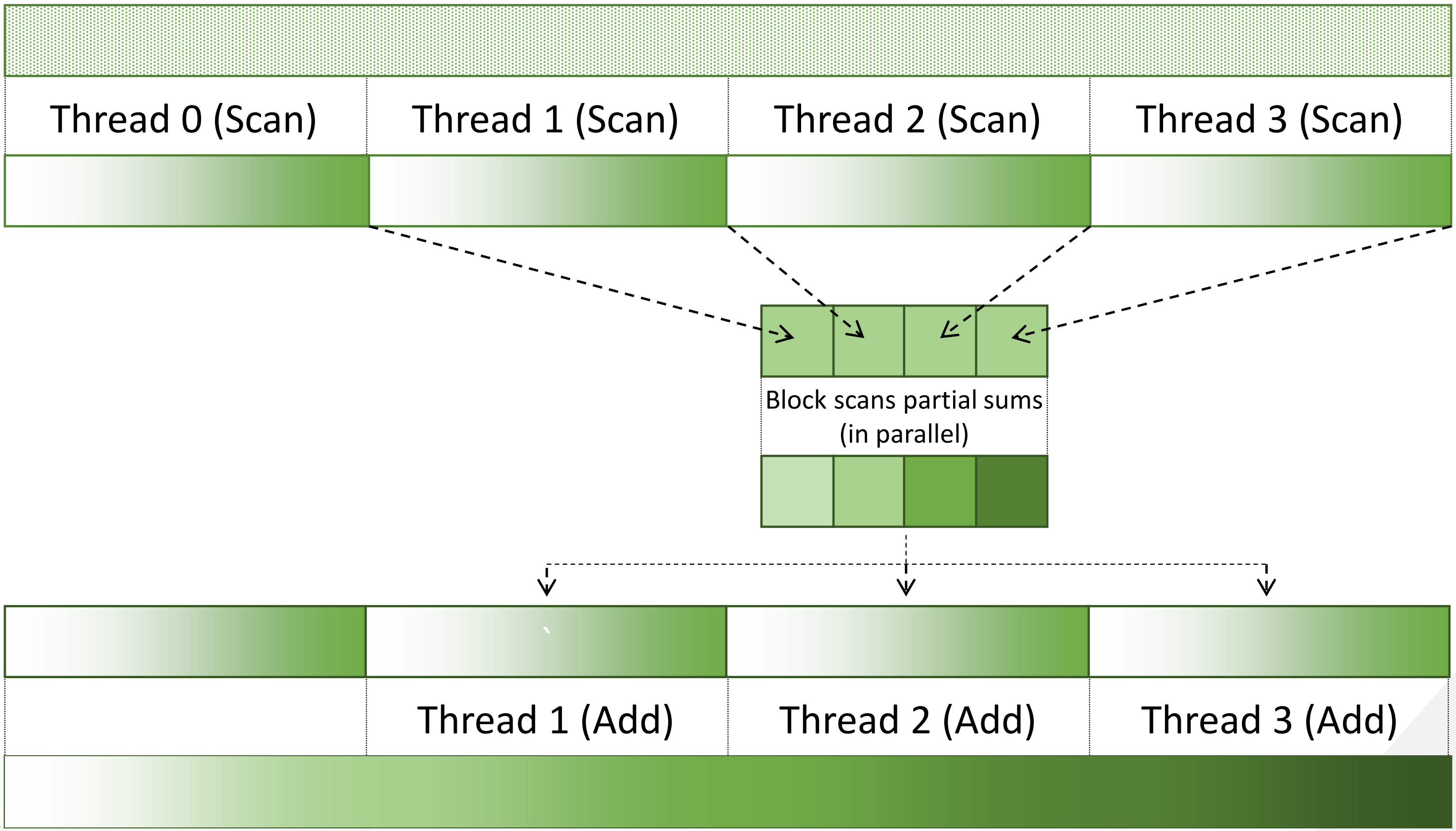
- Apply **thread coarsening** via segmented scan
  - Each thread scans a segment sequentially
    - Sequential scan is work efficient
  - Only scan the partial sums of each segment in parallel

Recall: Segmented Scan Example



Segment **grid** scan across **blocks**

Segmented Scan Example



Segment **block** scan across **threads**

Kogge-Stone Parallel (Inclusive) Scan with Thread Coarsening Code

```
unsigned int segment = COARSE_FACTOR*blockIdx.x*blockDim.x;

// Load data to shared memory
__shared__ float buffer_s[COARSE_FACTOR*BLOCK_DIM];
for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
    buffer_s[c*BLOCK_DIM + threadIdx.x] = input[segment + c*BLOCK_DIM + threadIdx.x];
}
__syncthreads();

// Scan thread subsegment
unsigned int threadSegment = threadIdx.x*COARSE_FACTOR;
for(unsigned int c = 1; c < COARSE_FACTOR; ++c) {
    buffer_s[threadSegment + c] += buffer_s[threadSegment + c - 1];
}

// Allocate and initialize double buffers for partial sums
__shared__ float buffer1_s[BLOCK_DIM];
__shared__ float buffer2_s[BLOCK_DIM];
float* inBuffer_s = buffer1_s;
float* outBuffer_s = buffer2_s;
unsigned int partialSumIdx = threadSegment + COARSE_FACTOR - 1;
inBuffer_s[threadIdx.x] = buffer_s[partialSumIdx];
__syncthreads();

// Parallel scan of partial sums
for(unsigned int stride = 1; stride <= BLOCK_DIM/2; stride *= 2) {
    if(threadIdx.x >= stride) {
        outBuffer_s[threadIdx.x] = inBuffer_s[threadIdx.x] + inBuffer_s[threadIdx.x - stride];
    } else {
        outBuffer_s[threadIdx.x] = inBuffer_s[threadIdx.x];
    }
    __syncthreads();
    float* tmp = inBuffer_s;
    inBuffer_s = outBuffer_s;
    outBuffer_s = tmp;
}

...
```

Kogge-Stone Parallel (Inclusive) Scan with Thread Coarsening Code

```
...

// Add previous thread's partial sums
if(threadIdx.x > 0) {
    float prevPartialSum = inBuffer_s[threadIdx.x - 1];
    for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
        buffer_s[threadSegment + c] += prevPartialSum;
    }
}
__syncthreads();

// Save block's partial sum
if(threadIdx.x == BLOCK_DIM - 1) {
    partialSums[blockIdx.x] = buffer_s[COARSE_FACTOR*BLOCK_DIM - 1];
}

// Write output
for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
    output[segment + c*BLOCK_DIM + threadIdx.x] = buffer_s[c*BLOCK_DIM + threadIdx.x];
}
```

References

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