

P&S Heterogeneous Systems

Parallel Patterns: Prefix Sum (Scan)

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2021

2 December 2021

Parallel Patterns

Reduction Operation

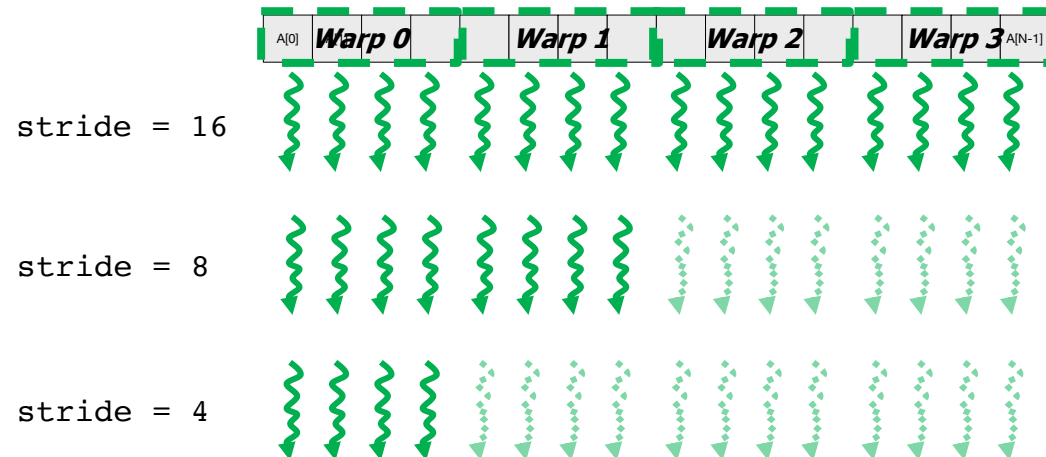
- A **reduction** operation reduces a set of values to a single value
 - Sum, Product, Minimum, Maximum are examples
- **Properties of reduction**
 - Associativity
 - Commutativity
 - Identity value
- Reduction is a key primitive for parallel computing
 - E.g., MapReduce programming model

Divergence-Free Mapping (II)

■ Program with high SIMD utilization

```
__shared__ float partialSum[]  
  
unsigned int t = threadIdx.x;  
  
for(int stride = blockDim.x; stride > 0; stride >> 1){  
  
    __syncthreads();  
  
    if (t < stride)  
        partialSum[t] += partialSum[t + stride];  
  
}
```

Warp utilization
is maximized

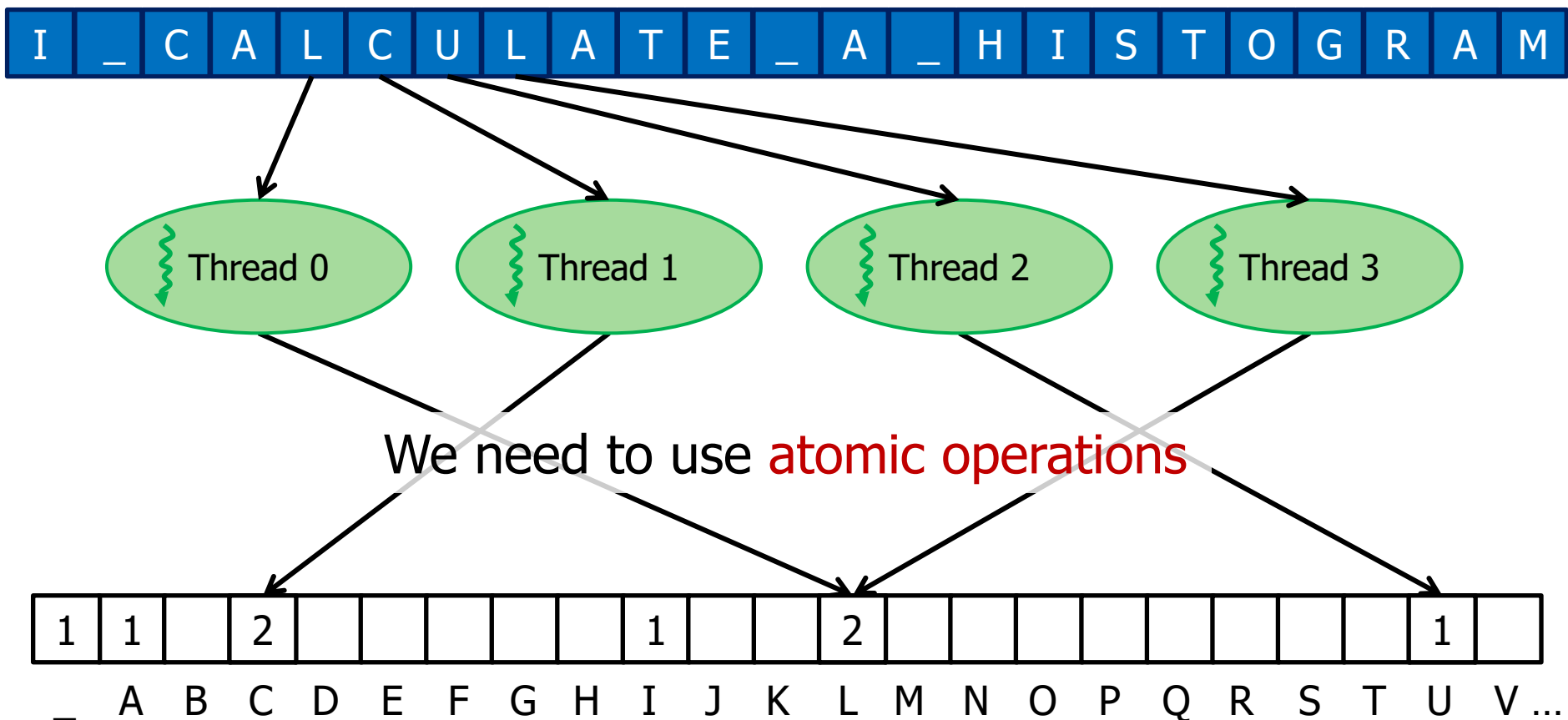


Histogram Computation

- Histogram is a frequently used computation for **reducing the dimensionality and extracting notable features** and patterns from large data sets
 - Feature extraction for object recognition in images
 - Fraud detection in credit card transactions
 - Correlating heavenly object movements in astrophysics
 - ...
- Basic histograms - for **each element in the data set, use the value to identify a "bin" to increment**
 - Divide possible input value range into "bins"
 - Associate a counter to each bin
 - For each input element, examine its value and determine the bin it falls into and increment the counter for that bin

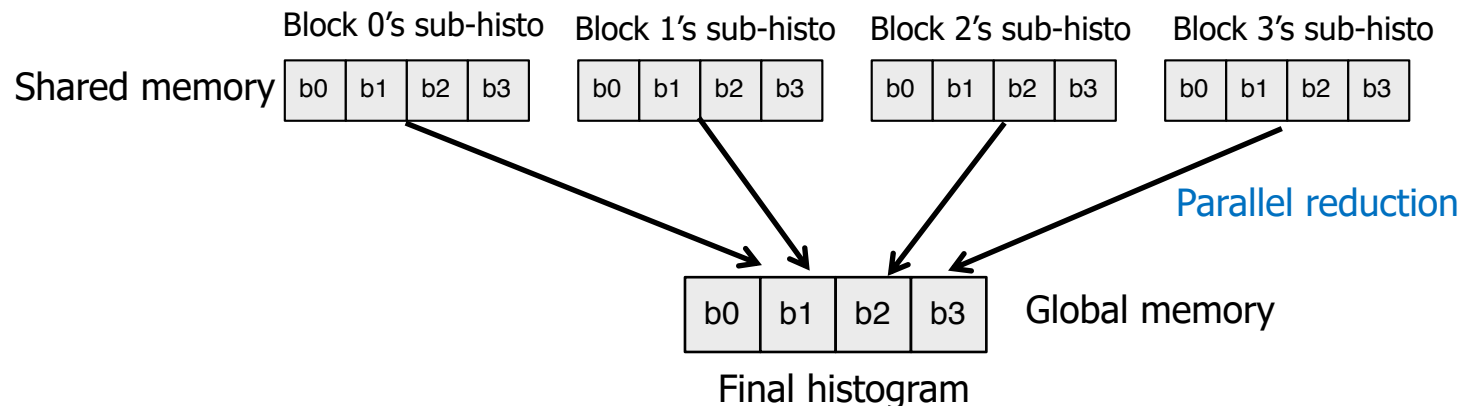
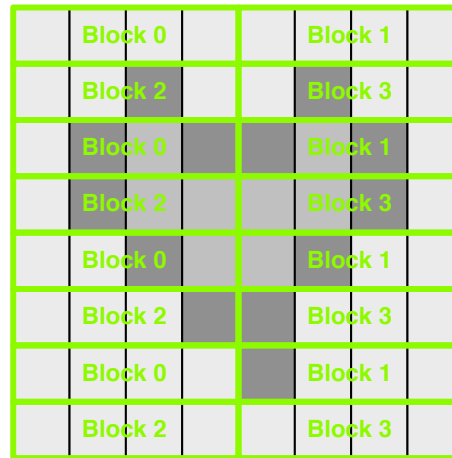
Parallel Histogram Computation: Iteration 2

- All threads move to the next section of the input
 - Each thread moves to element $\text{threadID} + \#\text{threads}$



Histogram Privatization

- **Privatization:** Per-block sub-histograms in shared memory
 - Threads use atomic operations in shared memory



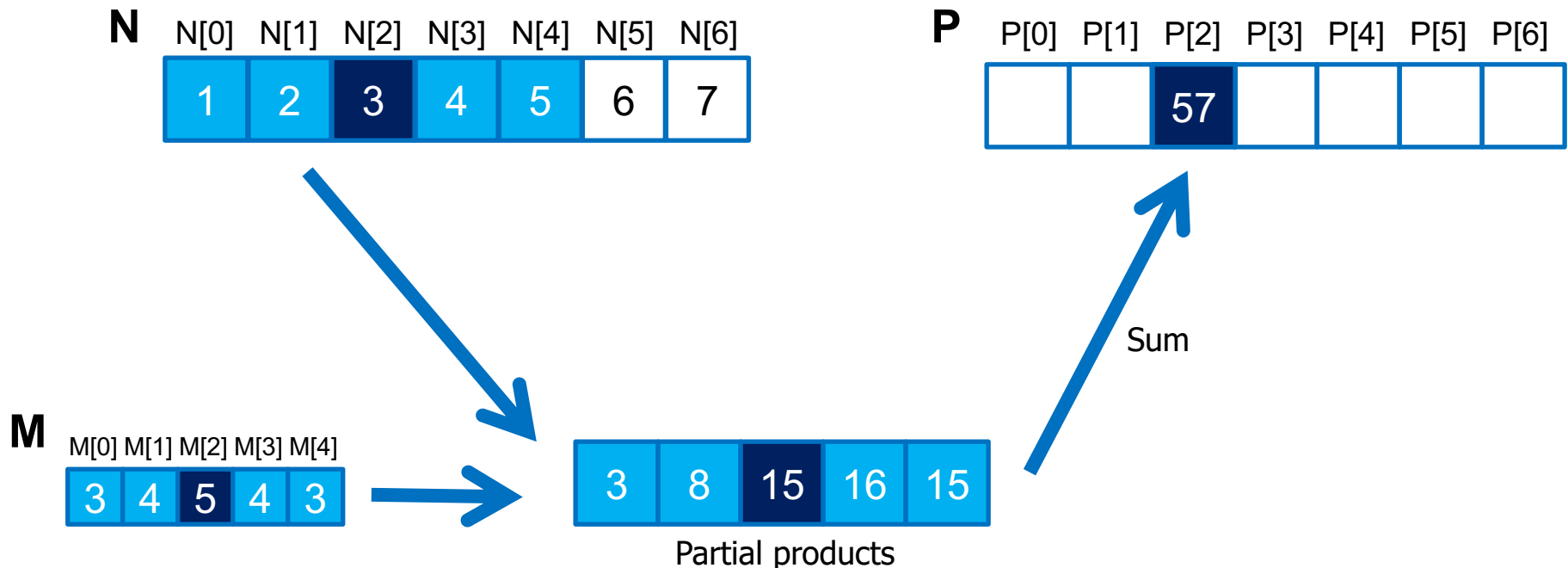
Convolution Applications

- **Convolution** is a widely-used operation in signal processing, image processing, video processing, and computer vision
- Convolution applies a **filter** or **mask** or **kernel*** on each element of the input (e.g., a signal, an image, a frame) to obtain a new value, which is a **weighted sum of a set of neighboring input elements**
 - Smoothing, sharpening, or blurring an image
 - Finding edges in an image
 - Removing noise, etc.
- Applications in machine learning and artificial intelligence
 - **Convolutional Neural Networks** (CNN or ConvNets)

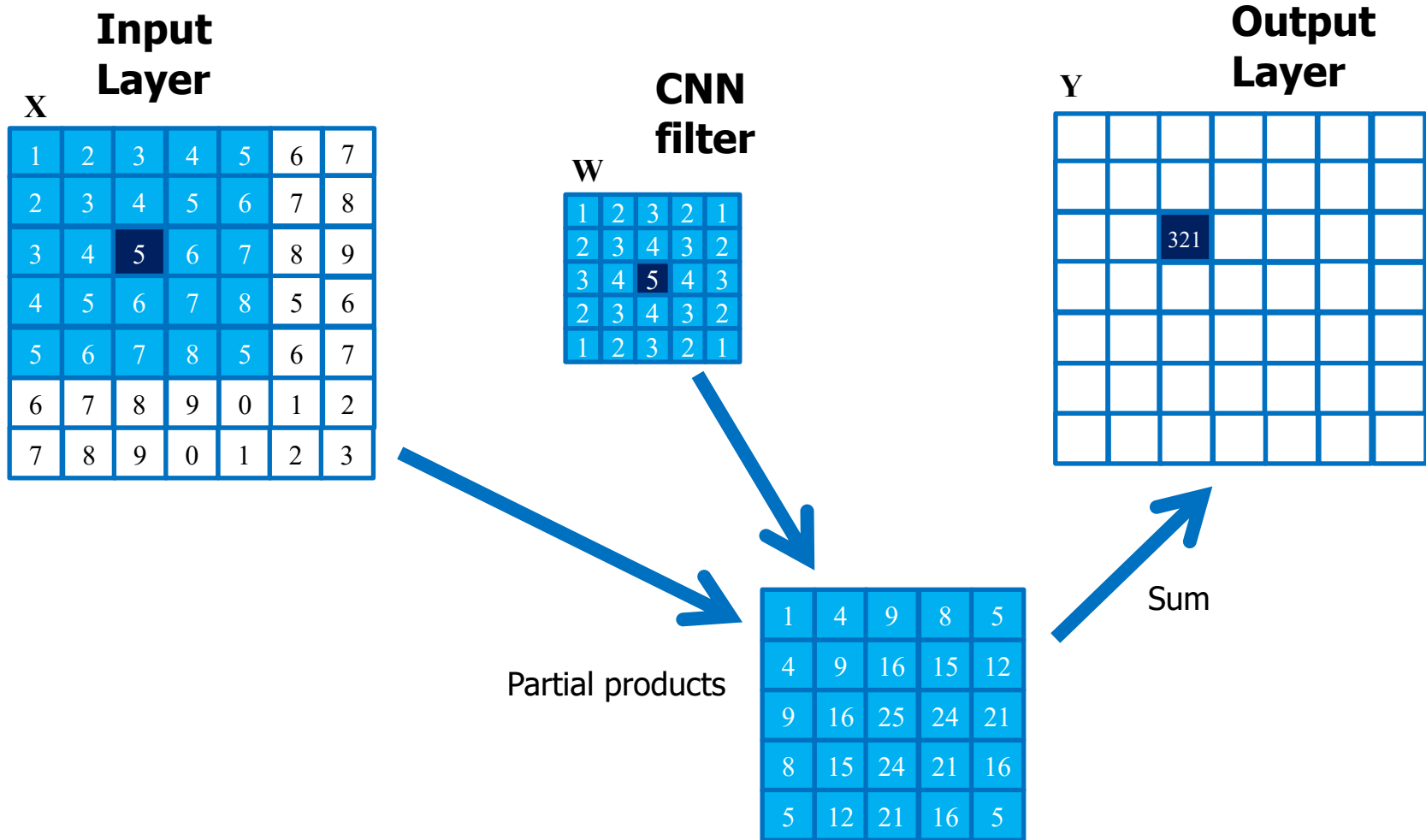
* The term "kernel" may create confusion in the context of GPUs (recall a CUDA/GPU kernel is a function executed by a GPU)

1D Convolution Example

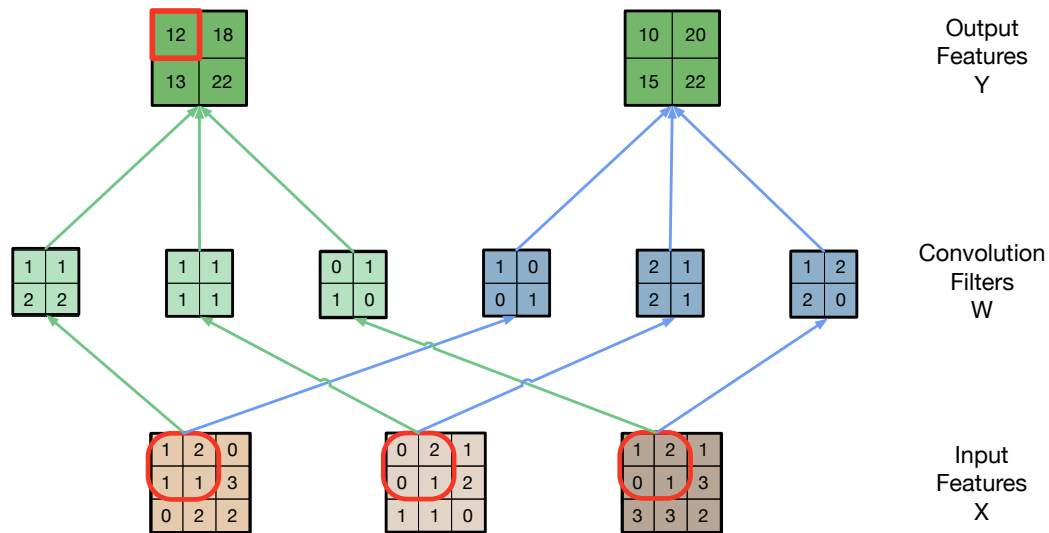
- Commonly used for audio processing
- Mask size is usually **an odd number of elements** for symmetry (5 in this example)
- Calculation of $P[2]$:



Another Example of 2D Convolution



Implementing a Convolutional Layer with Matrix Multiplication



$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 2 & 2 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 & 2 & 1 & 2 & 1 & 1 & 2 & 2 & 0 \\ \hline \end{array} * \begin{array}{|c|c|c|c|} \hline 1 & 2 & 1 & 1 \\ \hline 2 & 0 & 1 & 3 \\ \hline 1 & 1 & 0 & 2 \\ \hline 1 & 3 & 2 & 2 \\ \hline 0 & 2 & 0 & 1 \\ \hline 2 & 1 & 1 & 2 \\ \hline 0 & 1 & 1 & 1 \\ \hline 1 & 2 & 1 & 0 \\ \hline 1 & 2 & 0 & 1 \\ \hline 2 & 1 & 1 & 3 \\ \hline 0 & 1 & 3 & 3 \\ \hline 1 & 3 & 3 & 2 \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline 12 & 18 & 13 & 22 \\ \hline 10 & 20 & 15 & 22 \\ \hline \end{array}$$

Convolution Filters W' Input Features X (unrolled) Output Features Y

Prefix Sum (Scan)

Prefix Sum (Scan)

- **Prefix sum** or **scan** is an operation that takes an input array and an associative operator,
 - E.g., addition, multiplication, maximum, minimum
- And returns an output array that is the result of recursively applying the associative operator on the elements of the input array

- Input array $[x_0, x_1, \dots, x_{n-1}]$
- Associative operator \oplus

- An output array $[y_0, y_1, \dots, y_{n-1}]$ where
 - **Exclusive** scan: $y_i = x_0 \oplus x_1 \oplus \dots \oplus x_{i-1}$
 - **Inclusive** scan: $y_i = x_0 \oplus x_1 \oplus \dots \oplus x_i$

Scan Applications

- Scan is a key parallel primitive that can

- convert recurrences from sequential

```
for(int i=1; i<n; i++)  
    out[i] = out[i-1] + f(i);
```

- into parallel

```
forall(i) {temp[i] = f(i)};  
scan(out, temp);
```

- Scan is a **basic building block of many parallel algorithms**
 - E.g., stream compaction, partition, select, unique, radix sort, quicksort, string comparison, lexical analysis, polynomial evaluation, solving recurrences, tree operations, histograms, etc.

Examples of Exclusive and Inclusive Scan

Input

1	2	3	4	1	1	1	1	0	1	2	3	2	2	2	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Output (Exclusive Scan)

```
out[0] = 0; // Identity
for(int i=1; i<n; i++)
    out[i] = out[i-1] + in[i-1];
```

0	1	3	6	10	11	12	13	14	14	15	17	20	22	24	26
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----

Output (Inclusive Scan)

```
out[0] = in[0];
for(int i=1; i<n; i++)
    out[i] = out[i-1] + in[i];
```

1	3	6	10	11	12	13	14	14	15	17	20	22	24	26	28
---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

Hierarchical (Inclusive) Scan

Input

<i>Block 0</i>				<i>Block 1</i>				<i>Block 2</i>				<i>Block 3</i>			
1	2	3	4	1	1	1	1	0	1	2	3	2	2	2	2

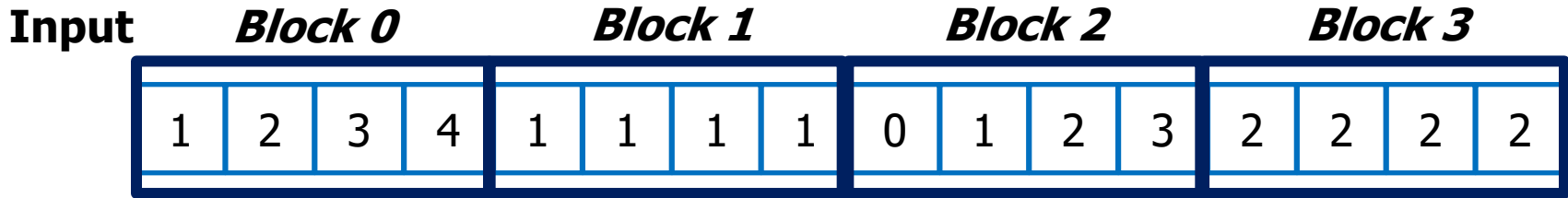
Per-block (Inclusive) Scan

1	3	6	10	1	2	3	4	0	1	3	6	2	4	6	8
---	---	---	----	---	---	---	---	---	---	---	---	---	---	---	---

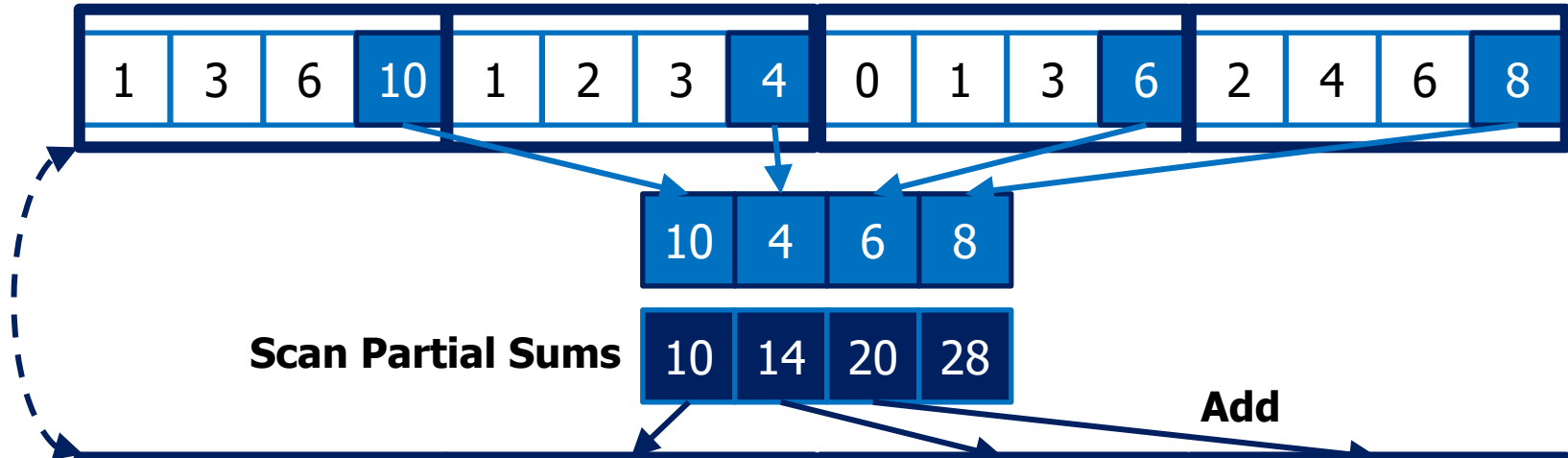
Output (Inclusive Scan)

1	3	6	10	11	12	13	14	14	15	17	20	22	24	26	28
---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

Hierarchical (Inclusive) Scan



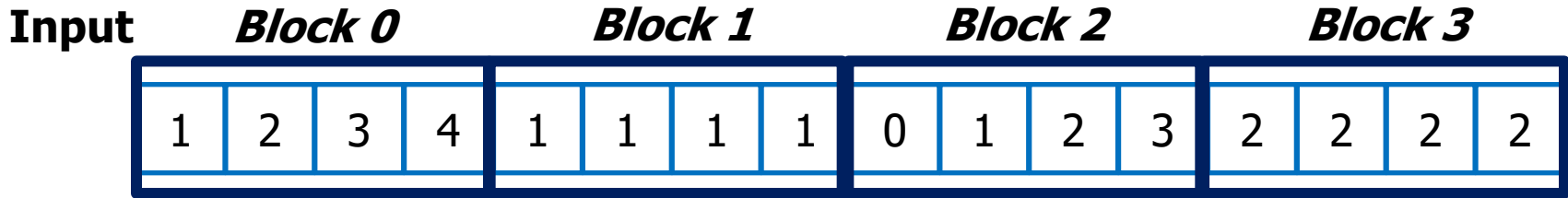
Per-block (Inclusive) Scan



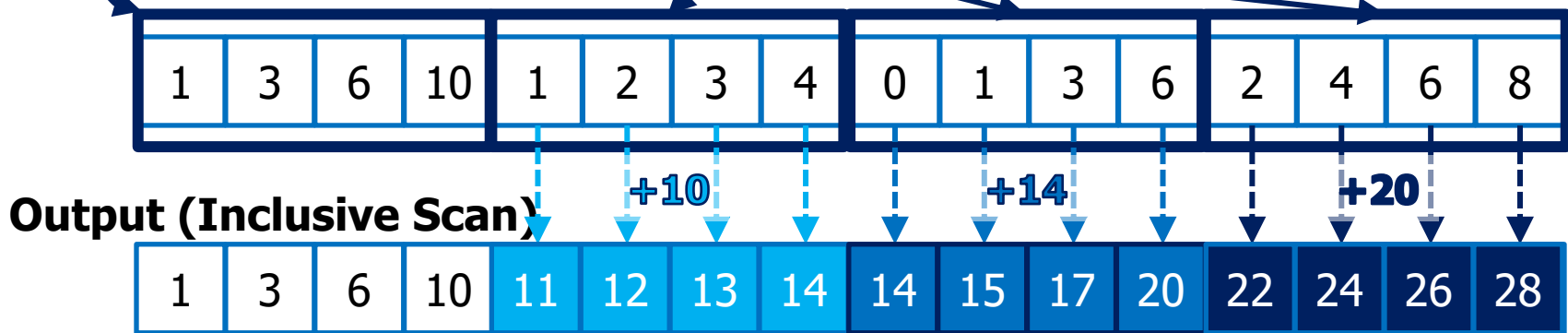
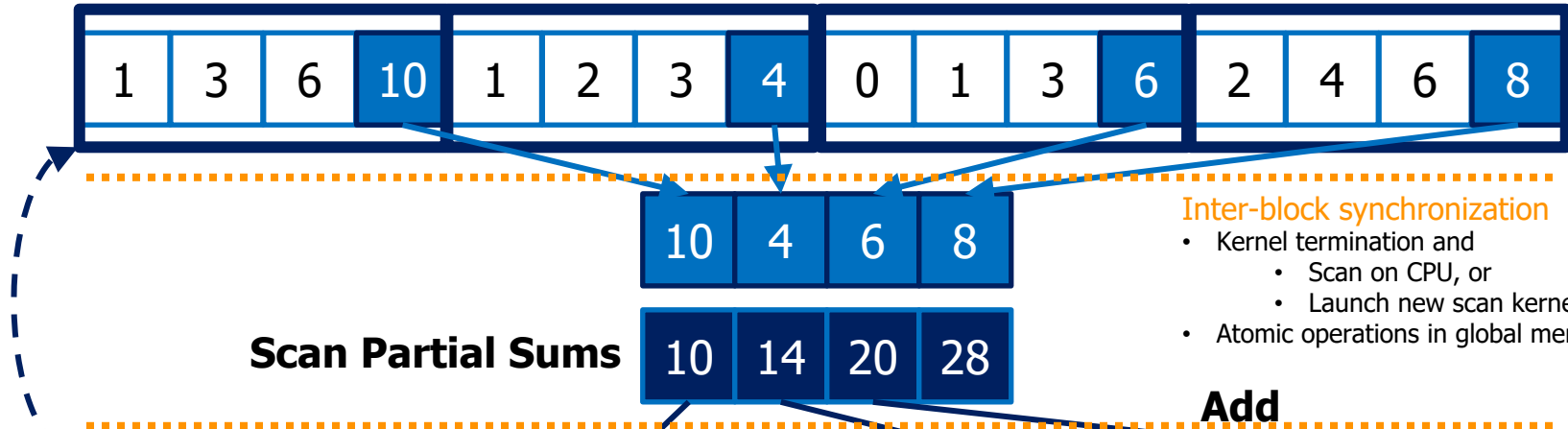
Output (Inclusive Scan)



Hierarchical (Inclusive) Scan



Per-block (Inclusive) Scan

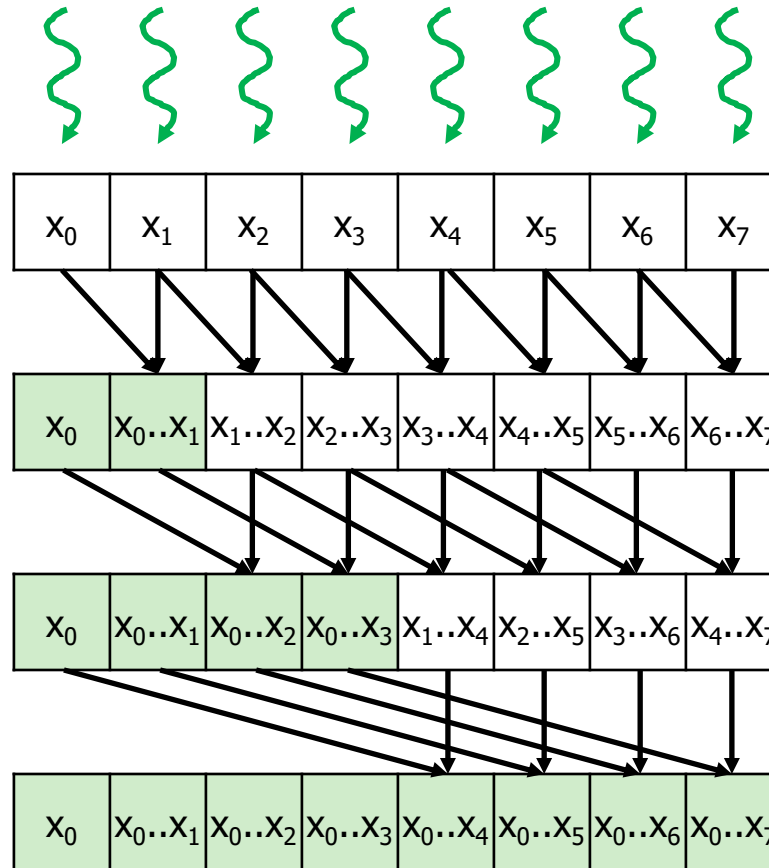


Per-Block (Inclusive) Scan

- Inside a thread block, we can also apply a hierarchical approach
 - Warps
 - Threads

- Let's start with the basic algorithms
 - Kogge-Stone
 - Brent-Kung

Kogge-Stone Parallel (Inclusive) Scan



Kogge-Stone Parallel (Inclusive) Scan Code

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

```
output[i] = input[i];
```

```
__syncthreads();
```

```
for(unsigned int stride = 1; stride <= BLOCK_DIM/2; stride *= 2) {
```

```
    float v;
```

```
    if(threadIdx.x >= stride) {
```

```
        v = output[i - stride];
```

```
    }
```

```
    __syncthreads();
```

Wait for everyone to read before updating

```
    if(threadIdx.x >= stride) {
```

```
        output[i] += v;
```

```
    }
```

```
    __syncthreads();
```

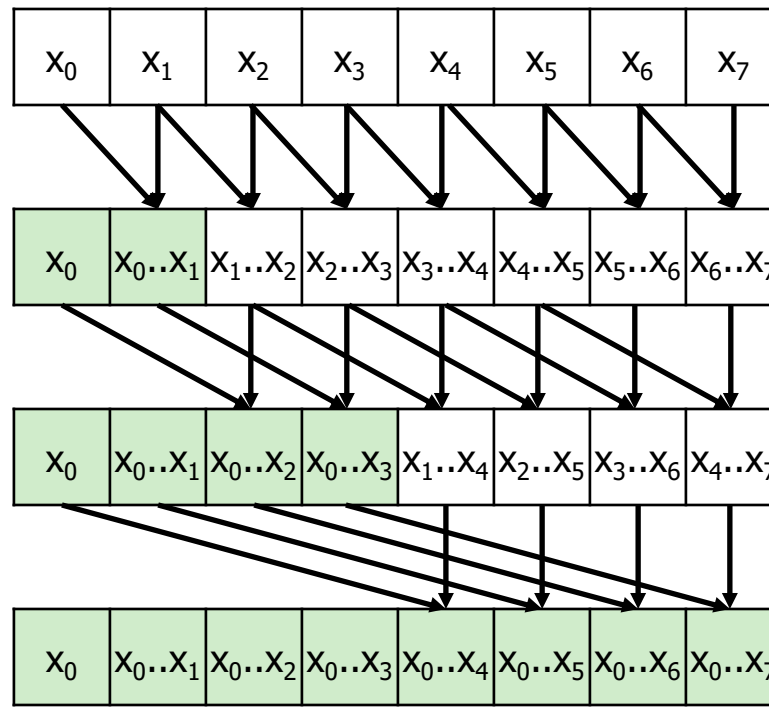
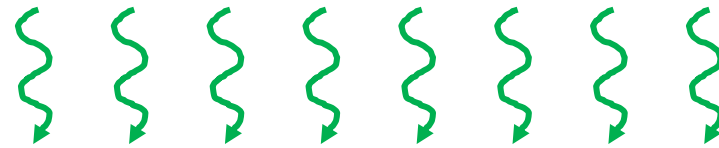
```
}
```

```
if(threadIdx.x == BLOCK_DIM - 1) {
```

```
    partialSums[blockIdx.x] = output[i];
```

```
}
```

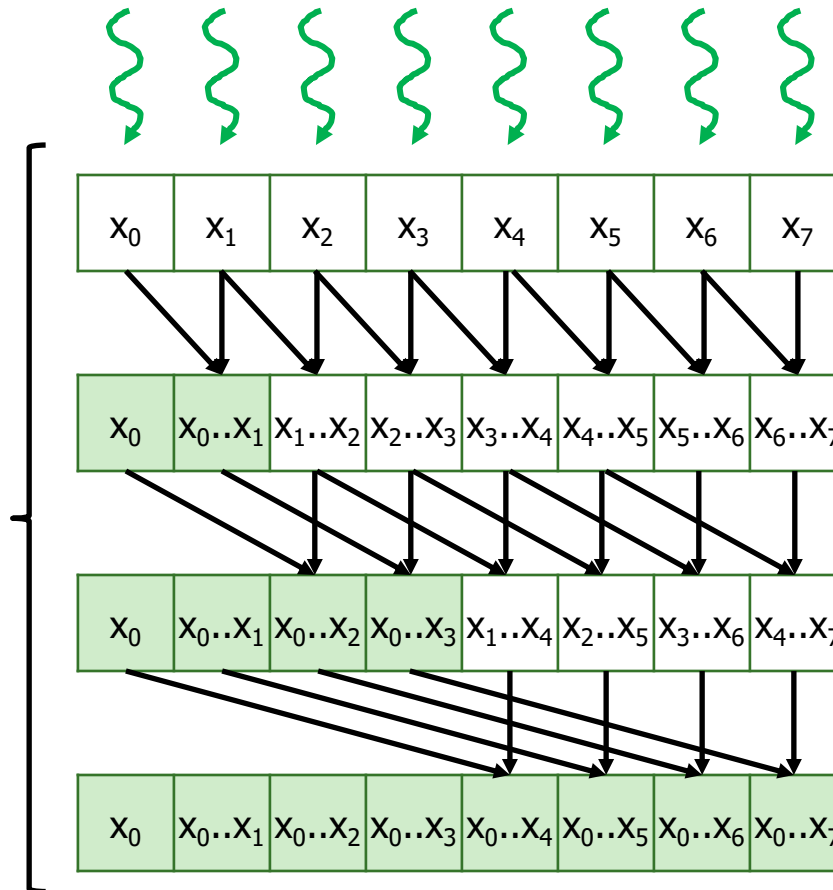
Kogge-Stone Parallel (Inclusive) Scan



Observation:
memory locations
are reused

Using Shared Memory

Optimization: load once to a **shared memory** buffer and perform successive reads and writes to the same array can be done in shared memory



Using Shared Memory Code

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

__shared__ float buffer_s[BLOCK_DIM];
buffer_s[threadIdx.x] = input[i];
__syncthreads();

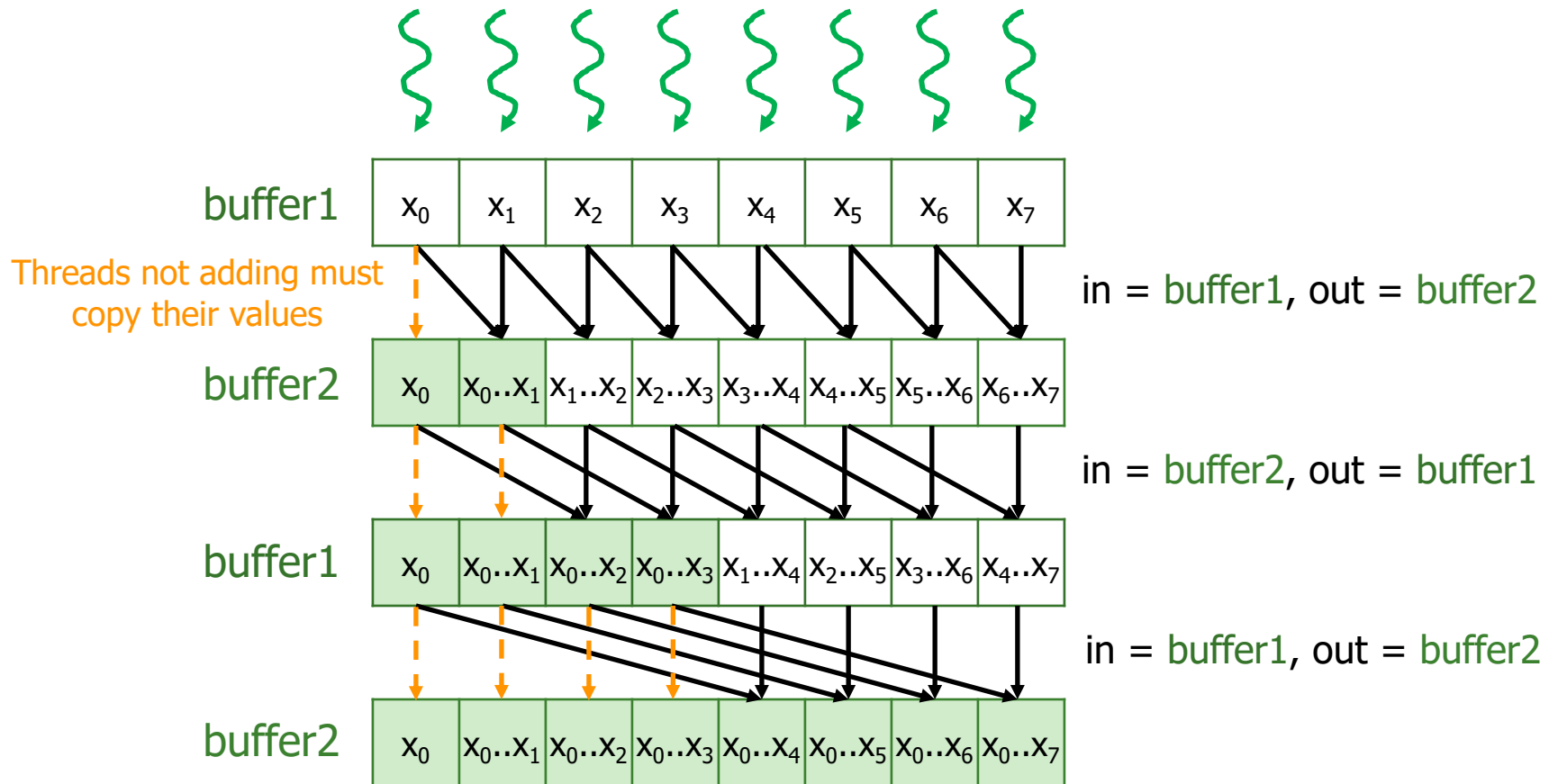
for(unsigned int stride = 1; stride <= BLOCK_DIM/2; stride *= 2) {
    float v;
    if(threadIdx.x >= stride) {
        v = buffer_s[threadIdx.x - stride];
    }
    __syncthreads();

    if(threadIdx.x >= stride) {
        buffer_s[threadIdx.x] += v;
    }
    __syncthreads();
}

if(threadIdx.x == BLOCK_DIM - 1) {
    partialSums[blockIdx.x] = buffer_s[threadIdx.x];
}

output[i] = buffer_s[threadIdx.x];
```

Double Buffering



Optimization: eliminate one synchronization by using two buffers and alternating them as the input/output buffer (called **double buffering**)

Double Buffering

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

__shared__ float buffer1_s[BLOCK_DIM];
__shared__ float buffer2_s[BLOCK_DIM];
float* inBuffer_s = buffer1_s;
float* outBuffer_s = buffer2_s;
inBuffer_s[threadIdx.x] = input[i];
__syncthreads();

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();

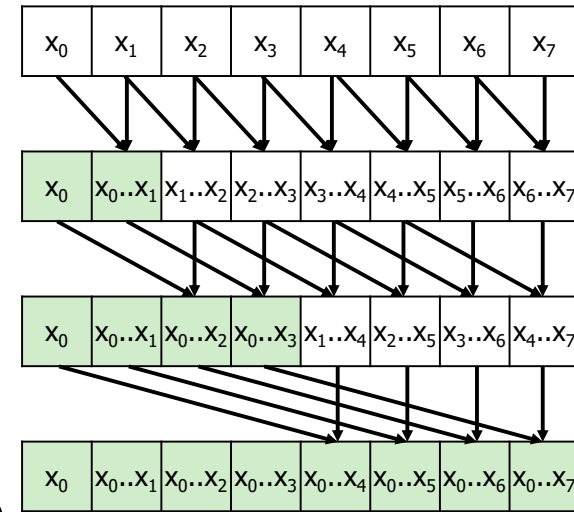
    float* tmp = inBuffer_s;
    inBuffer_s = outBuffer_s;
    outBuffer_s = tmp;
}

if(threadIdx.x == BLOCK_DIM - 1) {
    partialSums[blockIdx.x] = inBuffer_s[threadIdx.x];
}

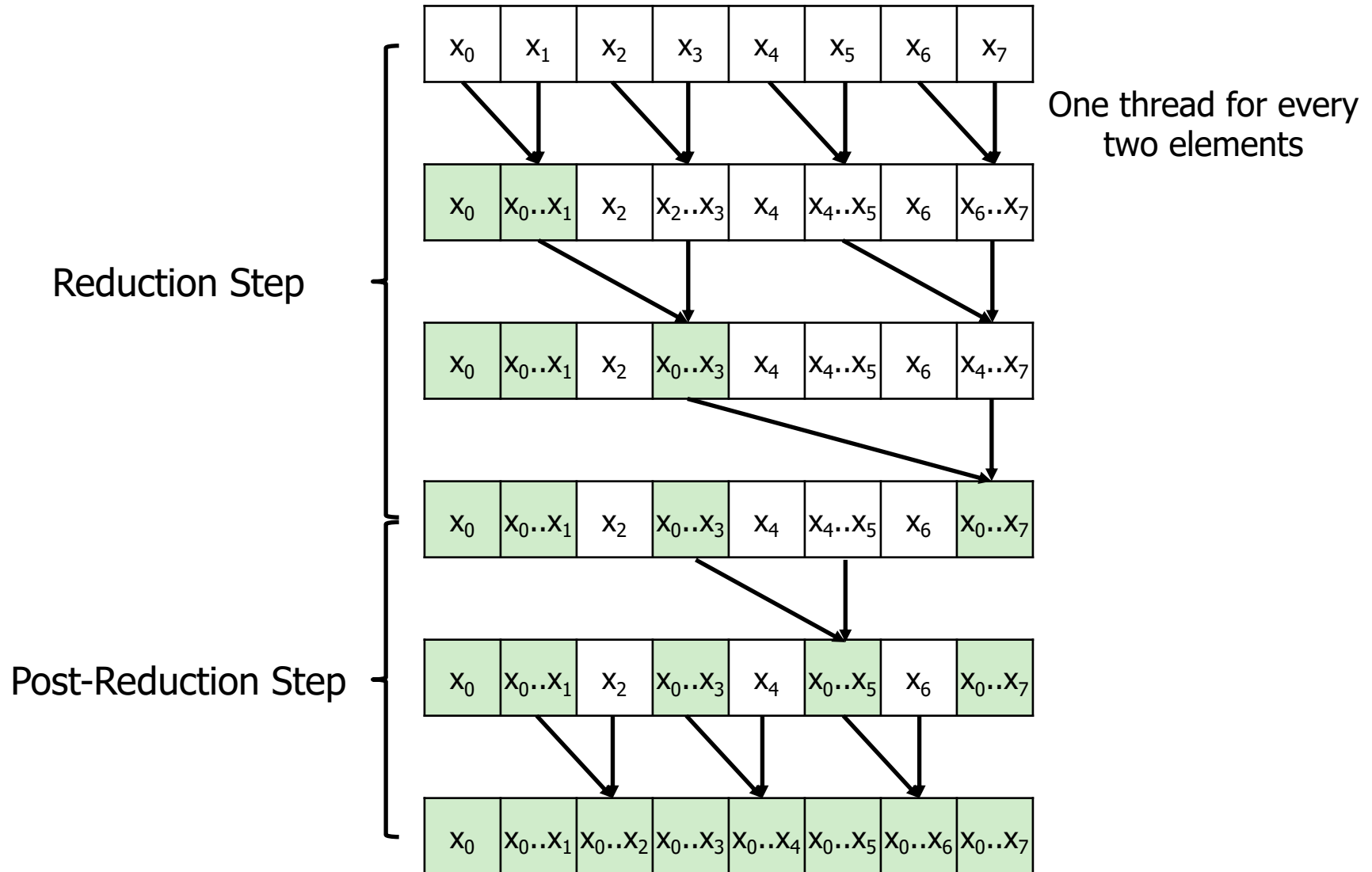
output[i] = inBuffer_s[threadIdx.x];
```

Work Efficiency

- A parallel algorithm is **work-efficient** if it performs the same amount of work as the corresponding sequential algorithm
- Scan work efficiency
 - Sequential scan performs N additions
 - Kogge-Stone parallel scan performs:
 - $\log(N)$ steps, $N - 2^{\text{step}}$ operations per step
 - Total: $(N-1) + (N-2) + (N-4) + \dots + (N-N/2)$
 $= N \cdot \log(N) - (N-1) = O(N \cdot \log(N))$ operations
 - Algorithm is not work efficient
- If resources are limited, parallel algorithm will be slow because of low work efficiency



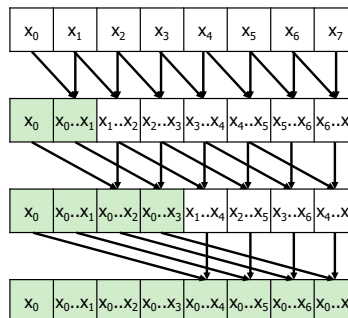
Brent-Kung Parallel (Inclusive) Scan



Work Efficiency

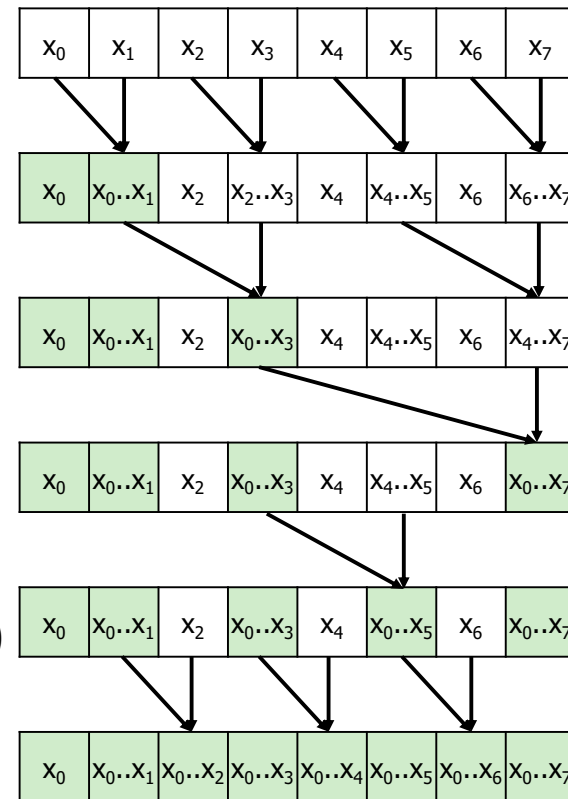
- Recall: Kogge-Stone

- $\log(N)$ steps
- $O(N \cdot \log(N))$ operations



- Brent-Kung

- Reduction step:
 - $\log(N)$ steps
 - $1 + 2 + 4 + \dots + N/2 = N-1$ operations
- Post-Reduction step:
 - $\log(N)-1$ steps
 - $(2-1) + (4-1) + \dots + (N/2-1) = (N-2) - (\log(N)-1)$
- Total:
 - $2 \cdot \log(N) - 1$ steps
 - $(N-1) + (N-2) - (\log(N)-1) = 2 \cdot N - \log(N) - 2 = O(N)$ operations



- Brent-Kung takes **more steps** but is **more work-efficient**

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 $O(N \cdot \log(N))$**
- Performance of Brent-Kung on GPUs is similar or may even be worse than Kogge-Stone
- Still is an interesting case to study

Recall: Warp Shuffle Functions

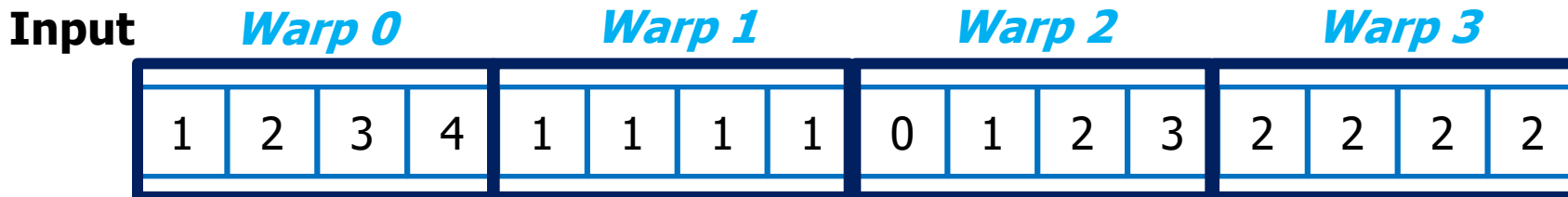
- Built-in **warp shuffle functions** enable threads to share data with other threads in the same warp
 - Faster than using shared memory and `__syncthreads()` to share across threads in the same block
- Variants:
 - `__shfl_sync(mask, var, srcLane)`
 - Direct copy from indexed lane
 - `__shfl_up_sync(mask, var, delta)`
 - Copy from a lane with lower ID relative to caller
 - `__shfl_down_sync(mask, var, delta)`
 - Copy from a lane with higher ID relative to caller
 - `__shfl_xor_sync(mask, var, laneMask)`
 - Copy from a lane based on bitwise XOR of own lane ID

Recall: Per-Block (Inclusive) Scan

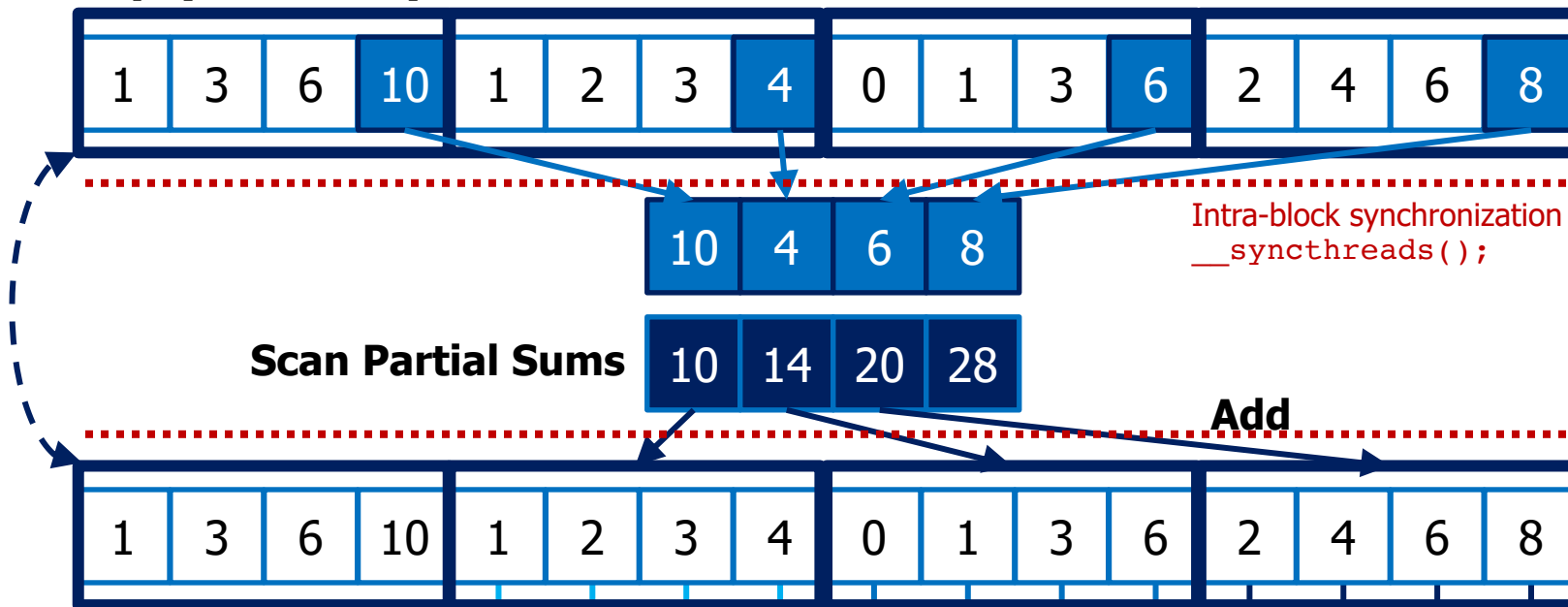
- Inside a thread block, we can also apply a hierarchical approach
 - Warps
 - Threads

- Let's start with the basic algorithms
 - Kogge-Stone
 - Brent-Kung

Per-Block Hierarchical (Inclusive) Scan



Per-warp (Inclusive) Scan



Output (Inclusive Scan)



Warp Scan

```
#define WARP_SIZE 32

// warp ID and lane ID
__device__ inline int lane_id(void) { return threadIdx.x % WARP_SIZE; }
__device__ inline int warp_id(void) { return threadIdx.x / WARP_SIZE; }

// warp scan
__device__ int warp_scan(int val){
    int x = val;

    #pragma unroll
    for(int offset = 1; offset < WARP_SIZE; offset <=< 1){

        int y = __shfl_up_sync(0xffffffff, x, offset);

        if(lane_id() >= offset)
            x += y;
    }

    return x - val;
}
```

Per-Block Hierarchical Scan

```

__device__ int block_scan(int* count, int x){
    __shared__ int sdata[L_DIM];

    // A. Exclusive scan within each warp
    int warpPrefix = warp_scan(x);

    // B. Store in shared memory
    if(lane_id() == WARP_SIZE - 1)
        sdata[warp_id()] = warpPrefix + x;

    __syncthreads();

    // C. One warp scans in shared memory
    if(threadIdx.x < WARP_SIZE)
        sdata[threadIdx.x] = warp_scan(sdata[threadIdx.x]);

    __syncthreads();

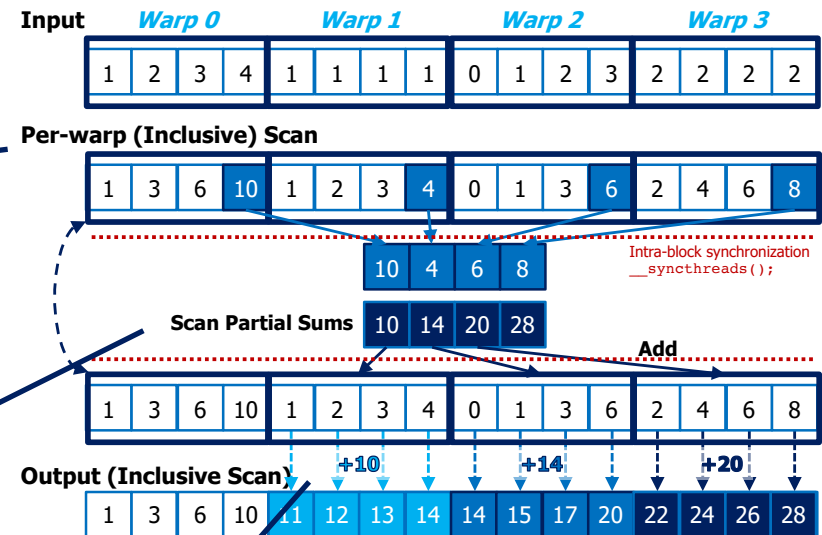
    // D. Each thread calculates its final value
    int thread_out_element = warpPrefix + sdata[warp_id()];
    int output = thread_out_element + *count;

    __syncthreads();

    if(threadIdx.x == blockDim.x - 1)
        *count += (thread_out_element + x);

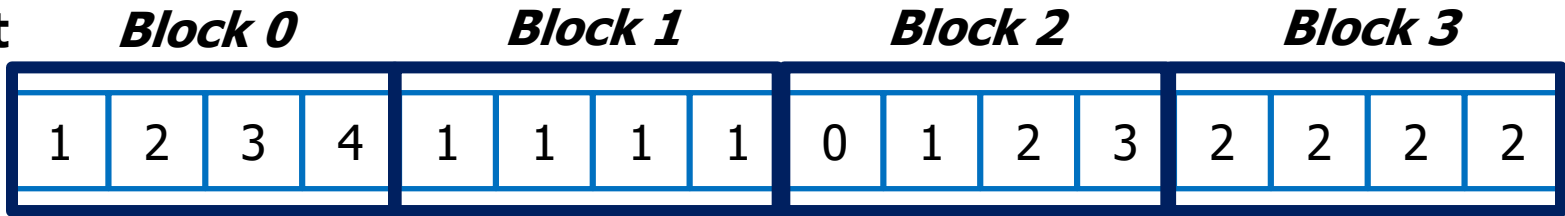
    return output;
}

```

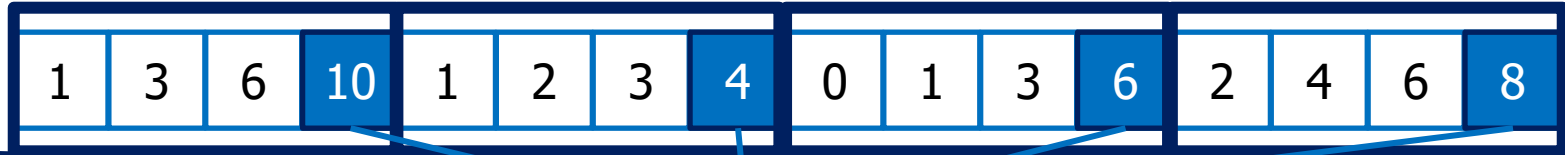


Scan-Scan-Add (SSA)

Input



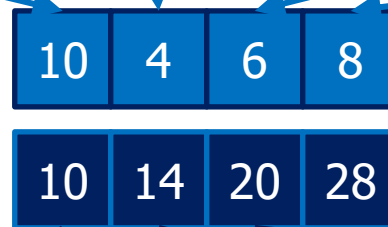
Per-block (Inclusive) Scan



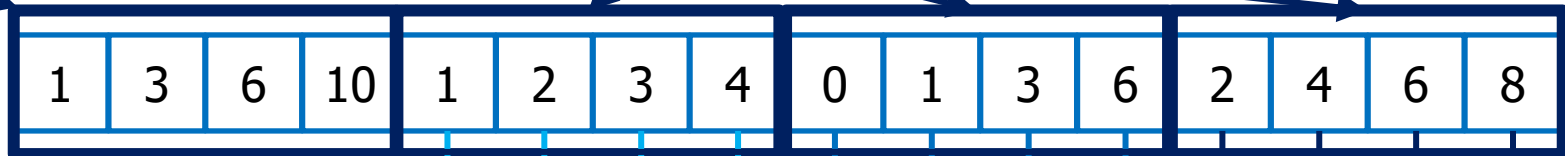
Inter-block synchronization

- Kernel termination and
 - Scan on CPU, or
 - Launch new scan kernel on GPU
- Atomic operations in global memory

Scan Partial Sums



Add

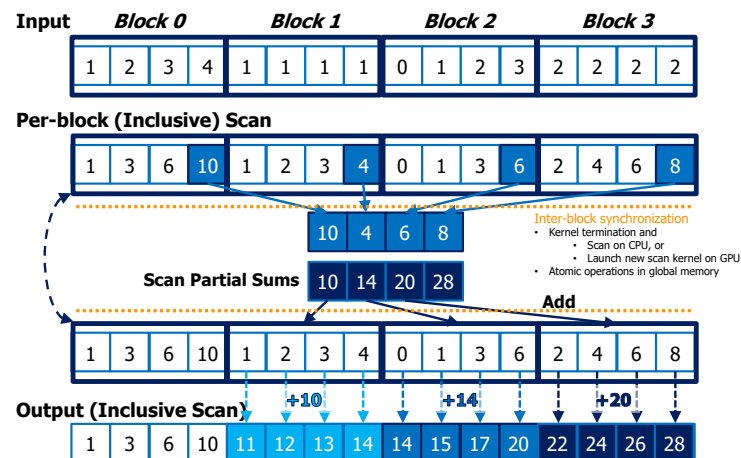


Output (Inclusive Scan)

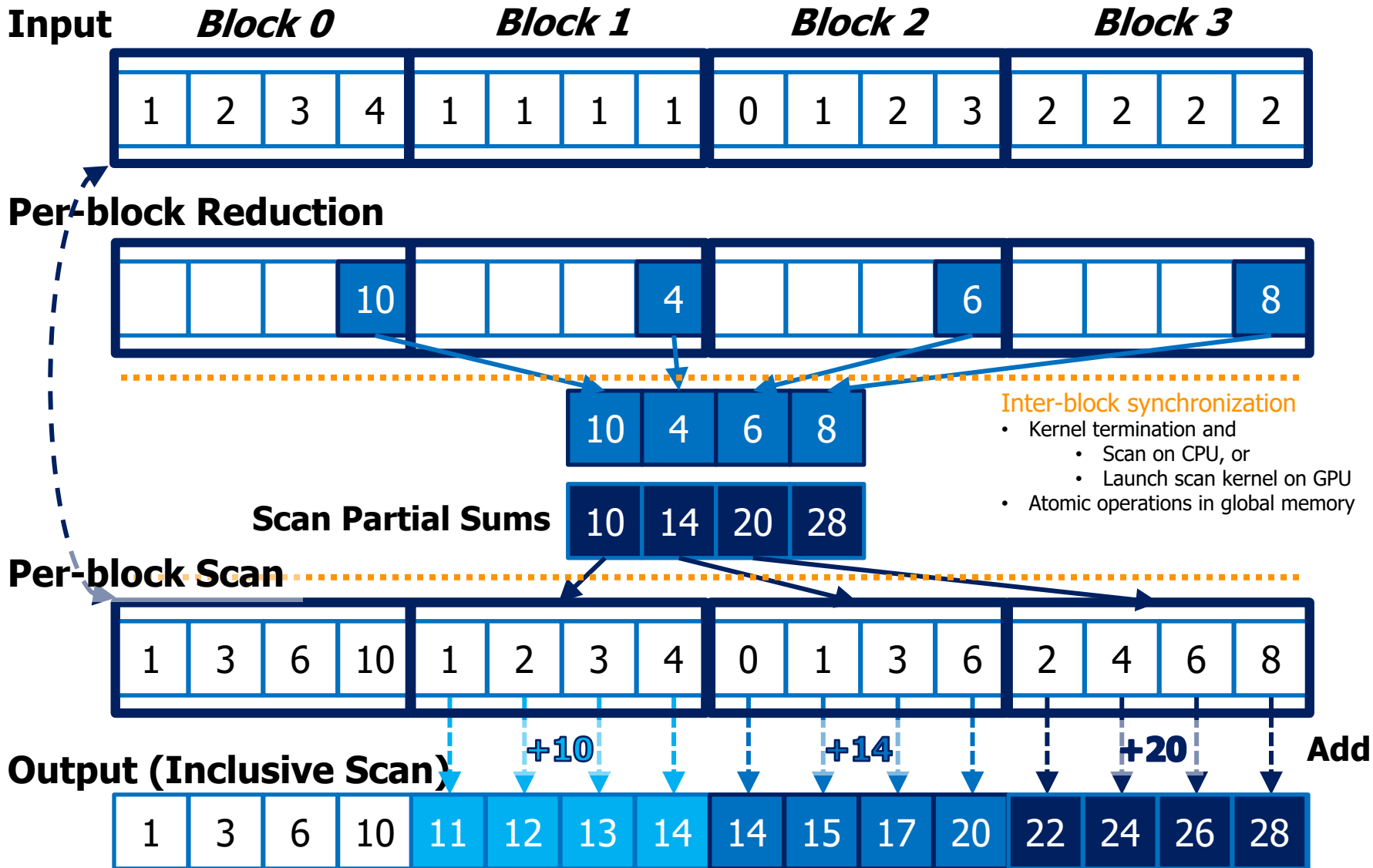


SSA: Global Memory Accesses

- Scan
 - First kernel reads **input array (N elements)** and writes array with **per-block prefix sums (N elements)**
- Scan
 - Second kernel reads and writes $N / \text{BLOCK_SIZE}$ elements
- Add
 - Third kernel reads array with **per-block prefix sums (N elements)** and writes **output (N elements)**
- **4N elements** are read/written

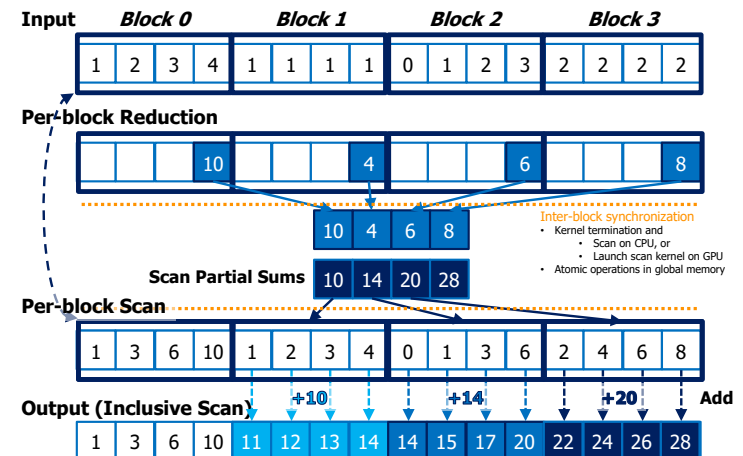


Reduce-Scan-Scan (RSS)



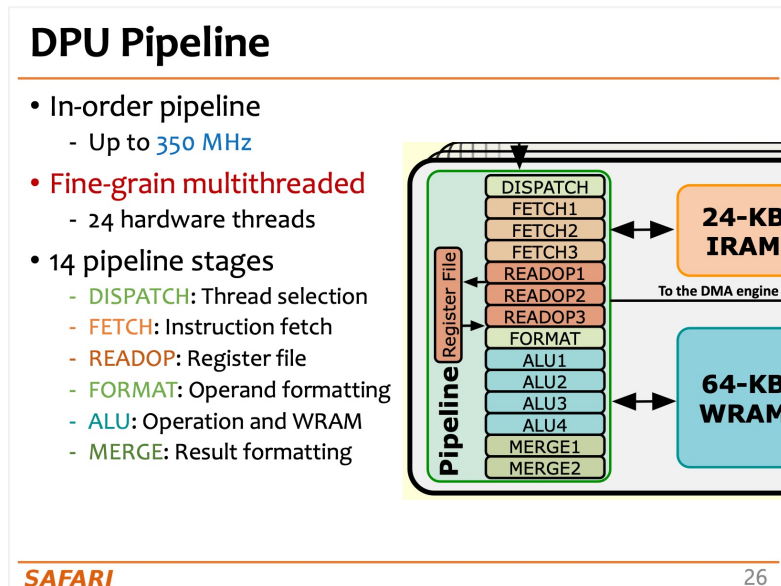
RSS: Global Memory Accesses

- Reduce
 - First kernel reads **input array (N elements)** and writes per-block reduction ($N / \text{BLOCK_SIZE}$ elements)
- Scan
 - Second kernel reads and writes $N / \text{BLOCK_SIZE}$ elements
- Scan
 - Third kernel reads **input array (N elements)** and scan partial sums ($N / \text{BLOCK_SIZE}$ elements), and writes **output (N elements)**
- **$3N$ elements** are read/written



SSA vs. RSS in a FGMT Architecture

- UPMEM Processing-in-Memory cores are fine-grained multithreaded
- Threads (called *tasklets*) can use **handshakes** to communicate pairs of tasklets and **barriers** to synchronize all tasklets



Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture

Juan Gómez-Luna¹ Izzat El Hajj² Ivan Fernandez^{1,3} Christina Giannoula^{1,4}
Geraldo F. Oliveira¹ Onur Mutlu¹

¹ETH Zürich ²American University of Beirut ³University of Malaga ⁴National Technical University of Athens

<https://arxiv.org/pdf/2105.03814.pdf>

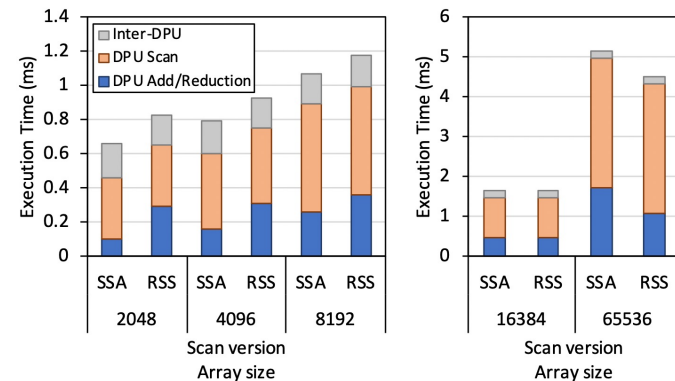
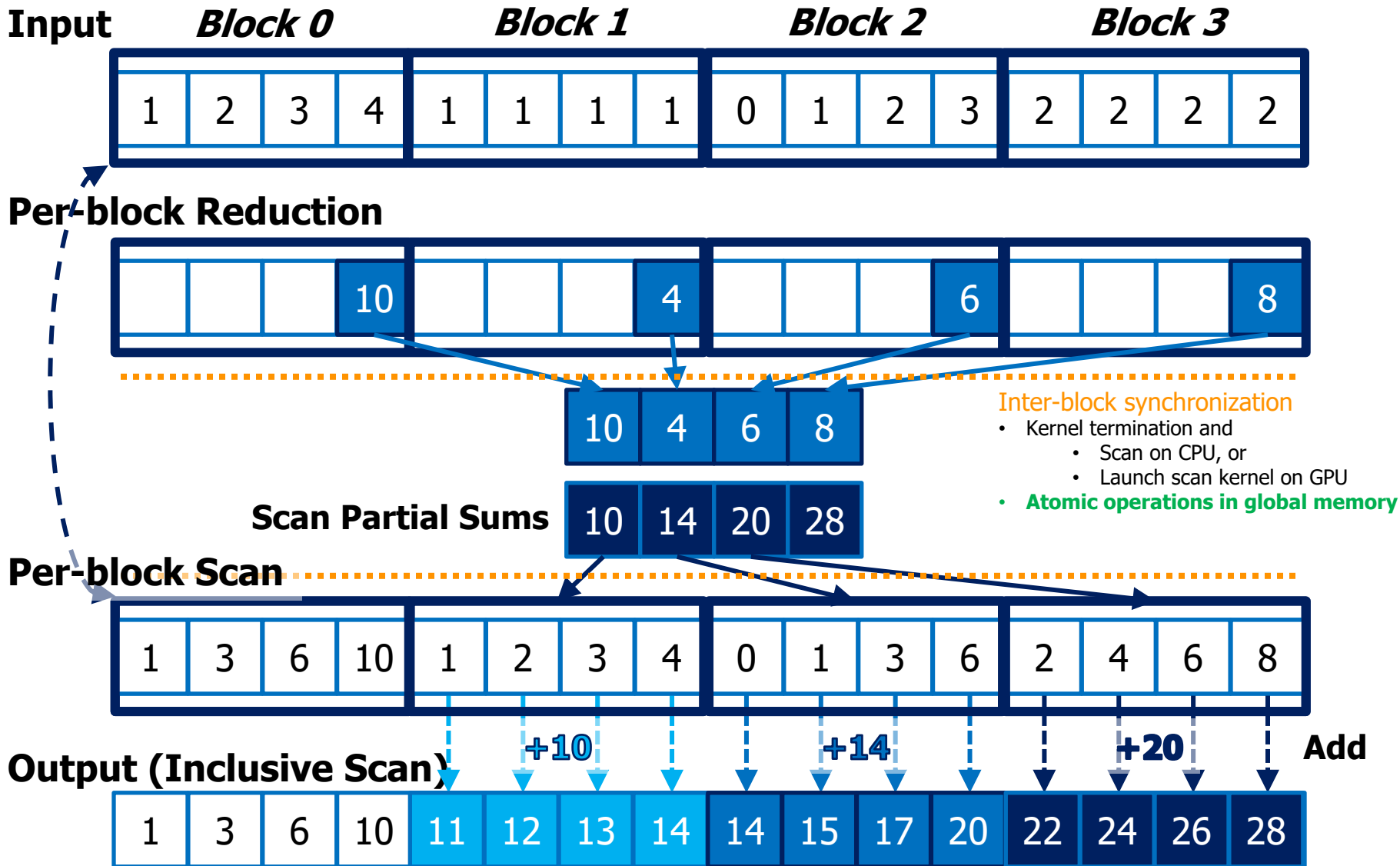


Figure 24: Two versions of scan (SCAN-SSA, SCAN-RSS) on 1 DPU.

Reduce-Scan-Scan (RSS)



Adjacent Block Synchronization (I)

```
__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();
```

flags and scan_value reside in global memory

Adjacent Block Synchronization (II)

```
__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();
```

Adjacent block
synchronization reduces
global memory accesses from
3N (or 4N) to 2N elements

Thread blocks may not be scheduled linearly
(in accordance with their block ID)

In the beginning of the kernel,
we obtain a **dynamic block ID**:

```
__shared__ int sbid;
if (threadIdx.x == 0)
    sbid = atomicAdd(DCounter, 1);
__syncthreads();
const int bid = sbid;
```

DCounter resides in global memory

Recall: Scan Applications

- Scan is a key parallel primitive that can
 - convert recurrences from sequential

```
for(int i=1; i<n; i++)  
    out[i] = out[i-1] + f(i);
```

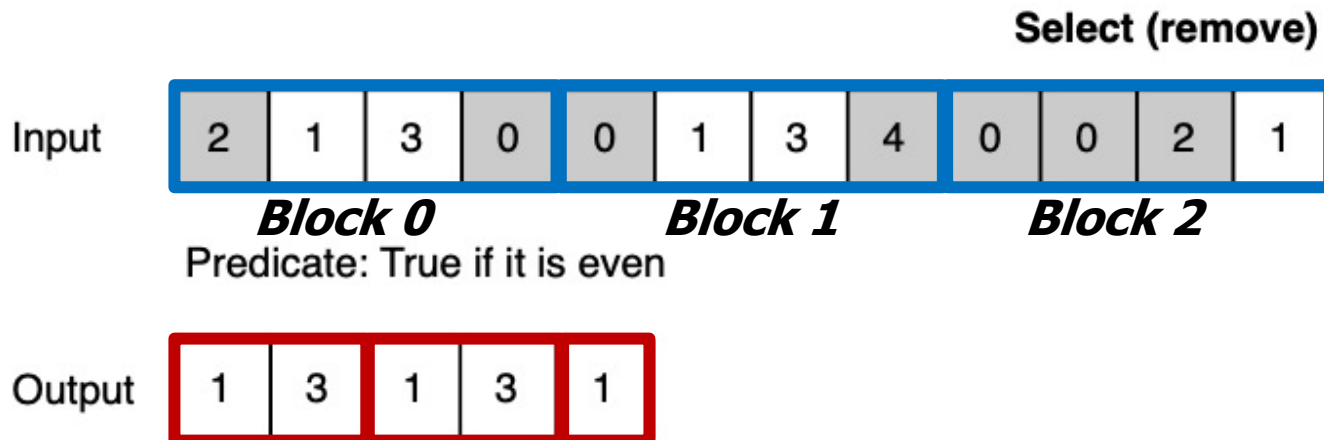
- into parallel

```
forall(i) {temp[i] = f(i)};  
scan(out, temp);
```

- Scan is a **basic building block of many parallel algorithms**
 - E.g., stream compaction, partition, select, unique, radix sort, quicksort, string comparison, lexical analysis, polynomial evaluation, solving recurrences, tree operations, histograms, etc.

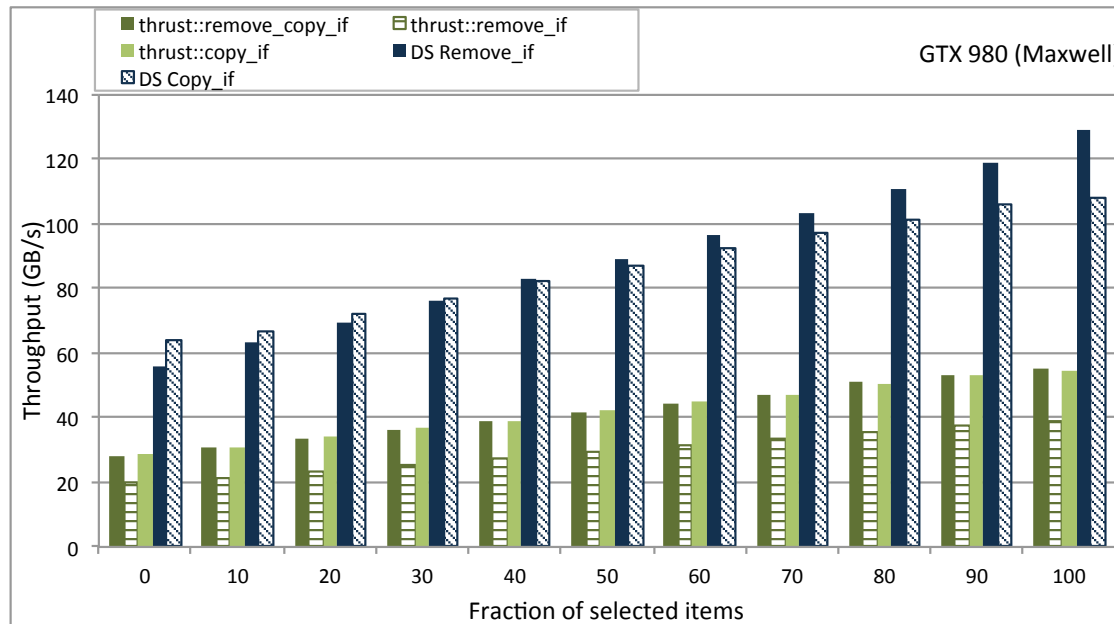
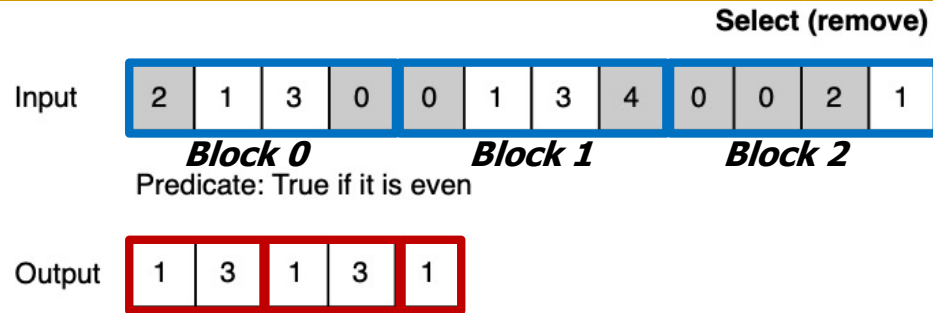
In-Place Data Sliding Algorithms (I)

- Algorithms that move data in memory in one direction
 - Padding / Unpadding
 - Stream compaction
 - SELECT
 - UNIQUE
 - Partitioning
 - Etc.



In-Place Data Sliding Algorithms (II)

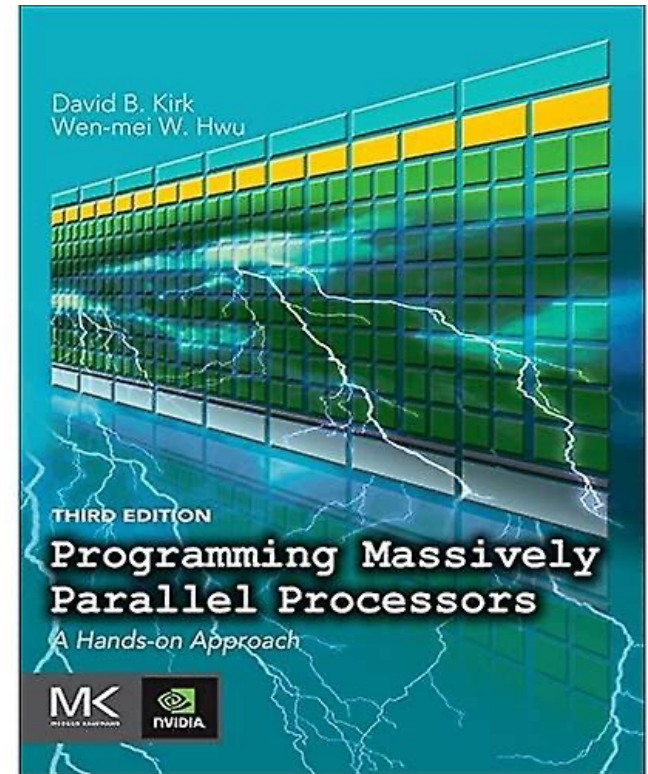
SELECT



~3x speedup

Recommended Readings

- Hwu and Kirk, “**Programming Massively Parallel Processors,**” Third Edition, 2017
 - Chapter 8 - Parallel patterns:
prefix sum: An introduction to work efficiency in parallel algorithms



P&S Heterogeneous Systems

Parallel Patterns: Prefix Sum (Scan)

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2021

2 December 2021