ECE408/CS483/CSE408 Spring 2025

Applied Parallel Programming

Lecture 15
Parallel Computation Patterns – Parallel Scan (Prefix Sum)

Course Reminders

- Project
  - We are grading milestone 1
  - Milestone 2 will be released soon
- Lab 5 – due this Friday
  - Implement a kernel and associated host code that performs reduction of a 1D list stored in an array. The reduction should give the sum of the list. You should implement the improved kernel discussed in the lecture. Your kernel should be able to handle input lists of **arbitrary length**.
- MT1
  - Regrade requests are due this Friday

Objective

- To learn parallel scan (prefix sum) algorithms based on reductions
  - Kogge-Stone Parallel Scan
  - Brent-Kung Parallel Scan
  - Hierarchical algorithms
- To learn the concept of double buffering
- To understand tradeoffs between work efficiency and latency

Scan Includes all Partial Results

Reductions are a simplified form of scans,

- we reduce the entire dataset to just one number.

In scan / parallel prefix,

- we need all of the partial sums
- (or whatever the operator might be).

(Inclusive) Scan (Prefix-Sum) Definition

Definition: *The scan operation takes a binary associative operator* $\oplus$*, and an array of n elements*

$$[x_0, x_1, \ldots, x_{n-1}],$$

and returns the prefix-sum array

$$[x_0, (x_0 \oplus x_1), (x_0 \oplus x_1 \oplus x_2), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})].$$

Example: If $\oplus$ is addition, the scan operation

on the array [3 1 7 0 4 1 6 3],

returns [3 4 11 11 15 16 22 25].

Example: Sharing a Big Sandwich

You order a 100-inch sandwich to feed 10 people, and you know how much each person wants in inches:

[3 5 2 7 28 4 3 0 8 1] .

How do you cut the bread quickly?

How much of the sandwich is left over?

Method 1: sequentially!

Cut 3 inches, then cut 5 inches, then …

Method 2: **calculate cutting offsets with prefix-sum**

[3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)

Typical Applications of Scan

A simple and useful parallel building block.

Convert sequential recurrences

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

into parallel:

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

Typical Applications of Scan

- Useful for many parallel algorithms:
  - radix sort
  - quicksort
  - string comparison
  - lexical analysis
  - stream compaction
  - polynomial evaluation
  - solving recurrences
  - tree operations
  - histograms
  - …

Other Applications

- Assigning camp slots
- Assigning farmer market space
- Allocating memory to parallel threads
- Allocating memory buffer to communication channels
- …

An Inclusive Sequential Scan

Given a sequence $[x_0, x_1, x_2, \ldots]$

Calculate output $[y_0, y_1, y_2, \ldots]$

Such that

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

...

Using a recursive definition

$$y_i = y_{i-1} + x_i$$

A Sequential C Implementation

```
y[0] = x[0];
for (i = 1; i < Max_i; i++)
    y[i] = y[i-1] + x[i];
```

Computationally efficient:

N additions needed for N elements - O(N).

A Naïve Inclusive Parallel Scan

- Assign one thread to calculate each y element
- Have every thread to add up all x elements needed for the y element

$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$

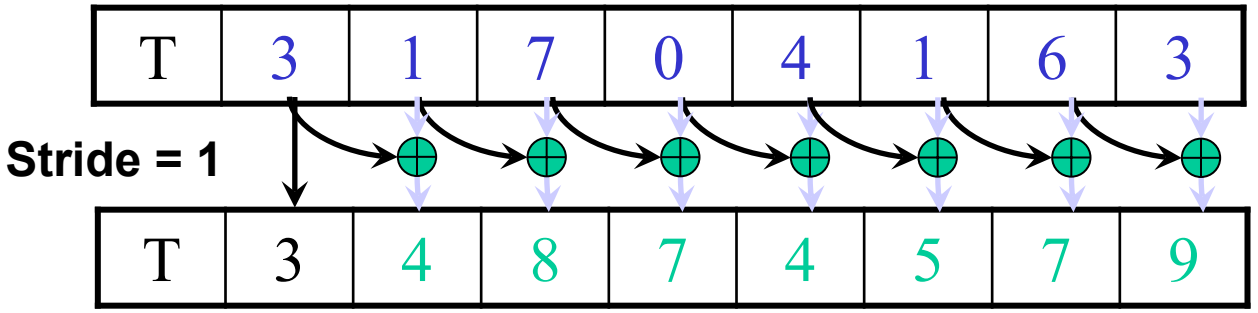
"Parallel programming is easy as long as you do not care about performance."

Parallel Inclusive Scan using Reduction Trees

Calculate each output element as the reduction of all previous elements

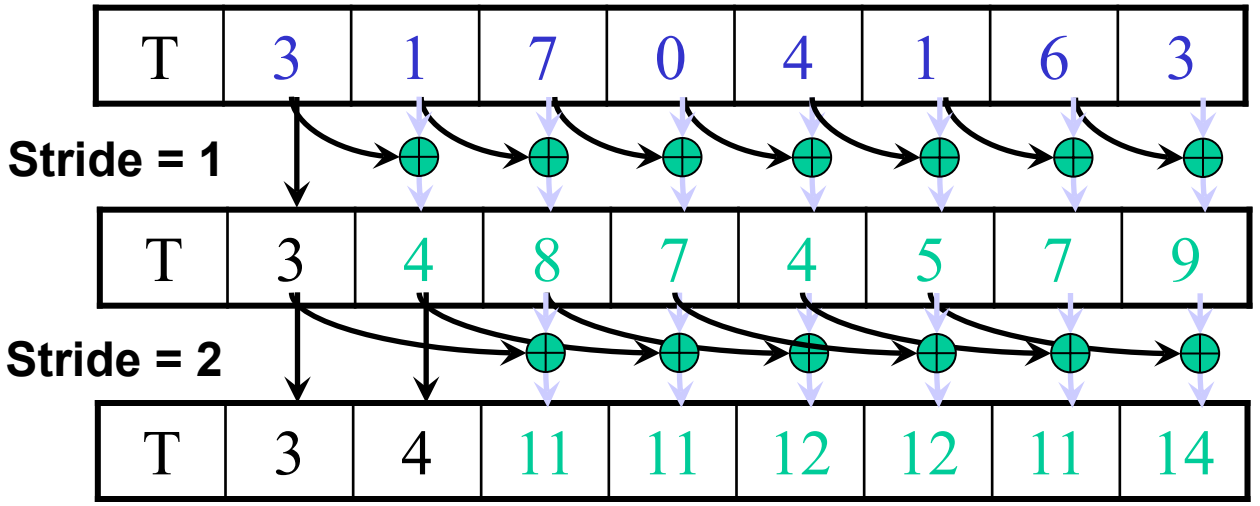
- Some reduction partial sums will be shared among the calculation of output elements
- Based on hardware added design by Peter Kogge and Harold Stone at IBM in the 1970s – Kogge-Stone Trees
- Goal: low latency

A Kogge-Stone Parallel Scan Algorithm



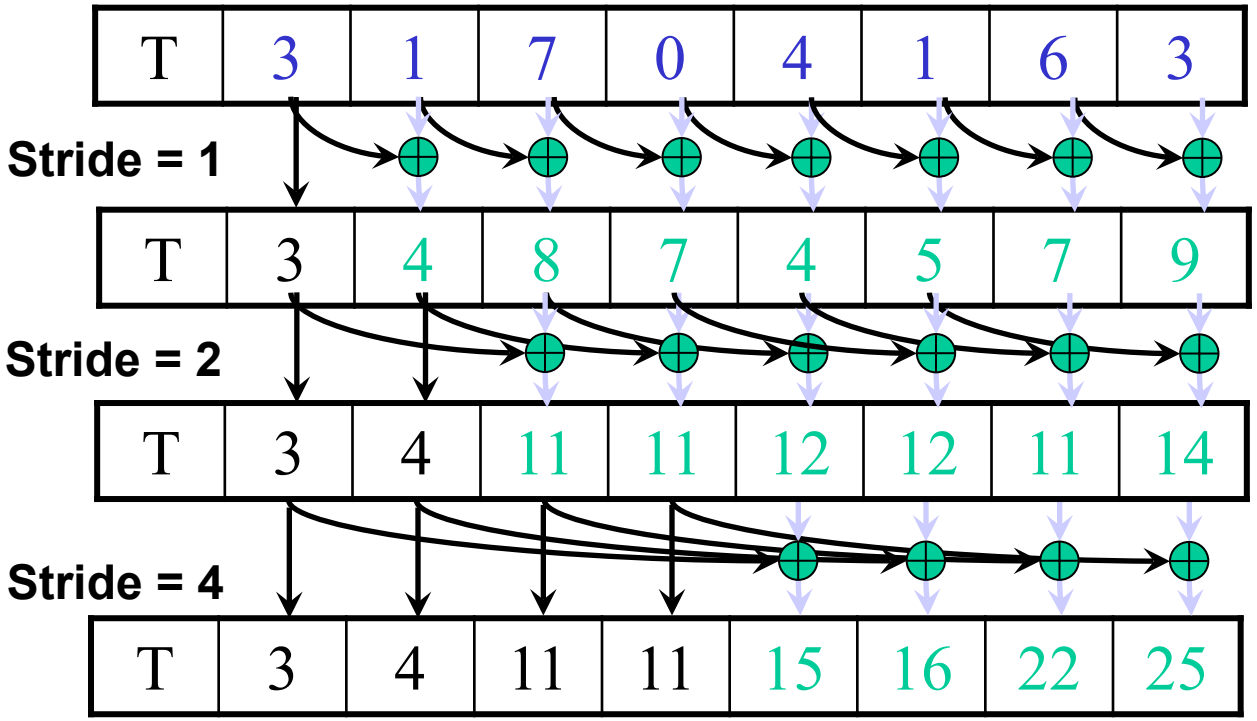
Iteration #1
Stride = 1

A Kogge-Stone Parallel Scan Algorithm



Iteration #2
Stride = 2

A Kogge-Stone Parallel Scan Algorithm



Iteration #3
Stride = 4

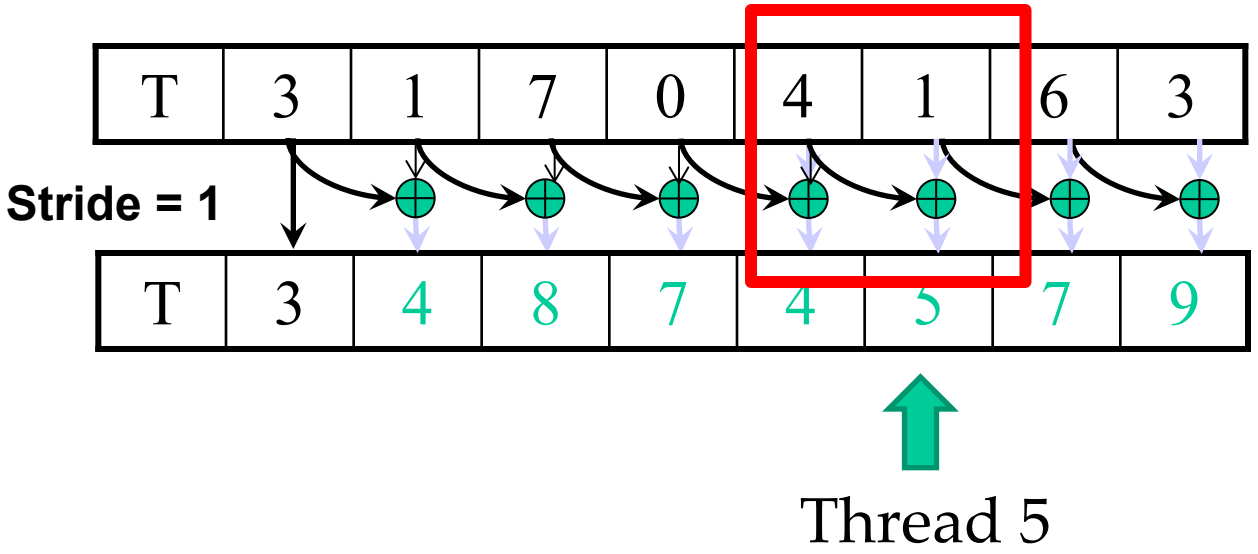
A Kogge-Stone Parallel Scan Algorithm

T	3	1	7	0	4	1	6	3

1. Load input from global memory into shared memory array T

Each thread loads one value from the input (global memory) array into shared memory array T.

A Kogge-Stone Parallel Scan Algorithm

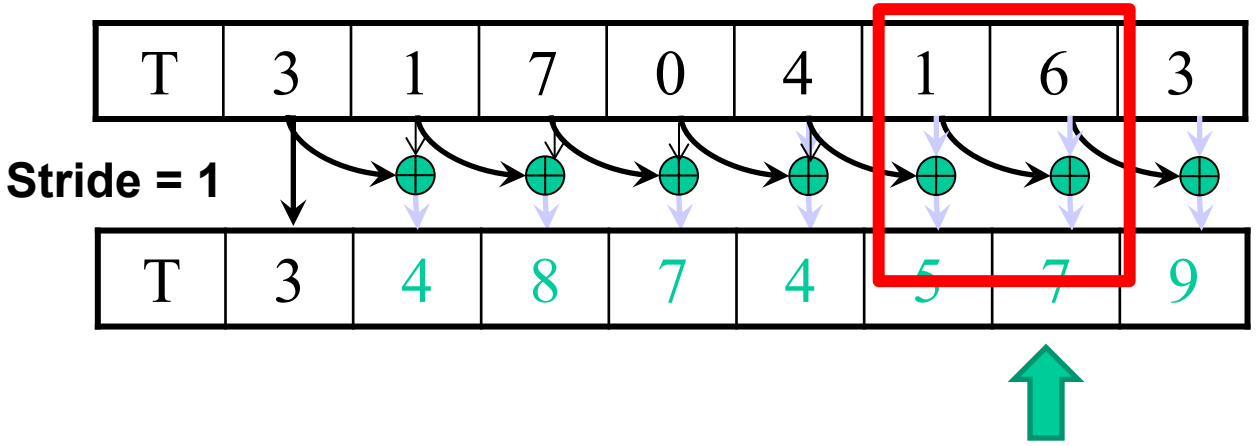


1. (previous slide)
2. Assuming n is a power of 2. Iterate log(n) times, stride from 1 to n/2. Threads *stride* to *n-1 active:* add pairs of elements that are s*tride* elements apart.

Iteration #1
Stride = 1

- Active threads: *stride* to *n*-1 (*n* - *stride* active threads)
- Thread *j* adds elements T[*j*] and T[*j-stride*] and writes result into element T[*j*]
- Each iteration requires two syncthreads
  - make sure that input is in place
  - make sure that all input elements have been used

A Kogge-Stone Parallel Scan Algorithm

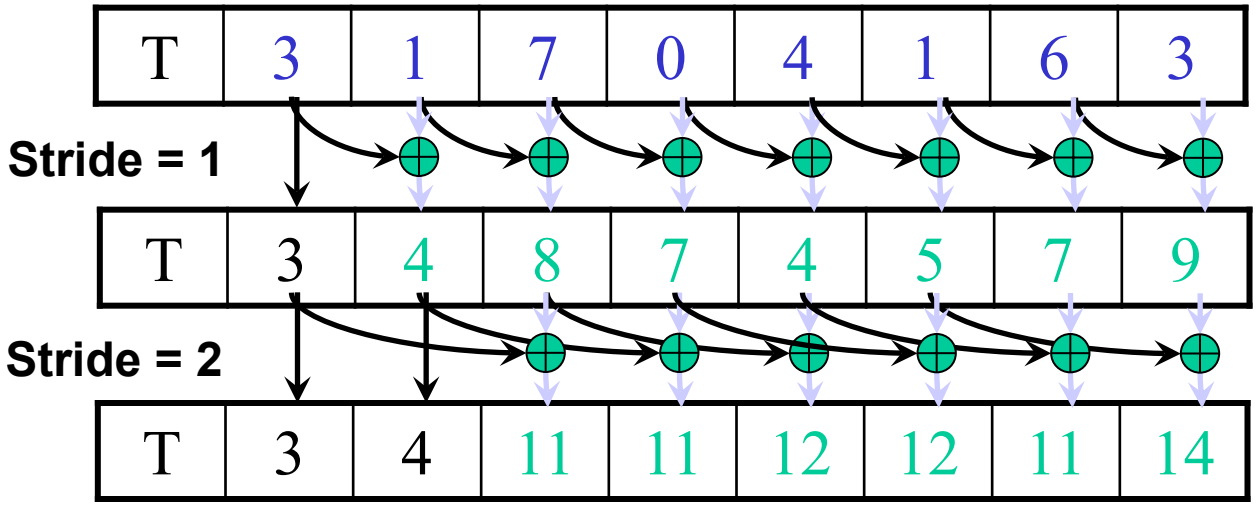


1. (previous slide)
2. Assuming n is a power of 2. Iterate log(n) times, stride from 1 to n/2. Threads *stride* to *n-1* *active:* add pairs of elements that are s*tride* elements apart.

Iteration #1
Stride = 1

- Active threads: *stride* to *n*-1 (*n* - *stride* active threads)
- Thread *j* adds elements T[*j*] and T[*j-stride*] and writes result into element T[*j*]
- Each iteration requires two syncthreads
  - syncthreads(); // make sure that input is in place
  - float temp = T[*j*] + T[*j-stride*];
  - syncthreads(); // make sure that previous output has been consumed
  - T[*j*] = temp;

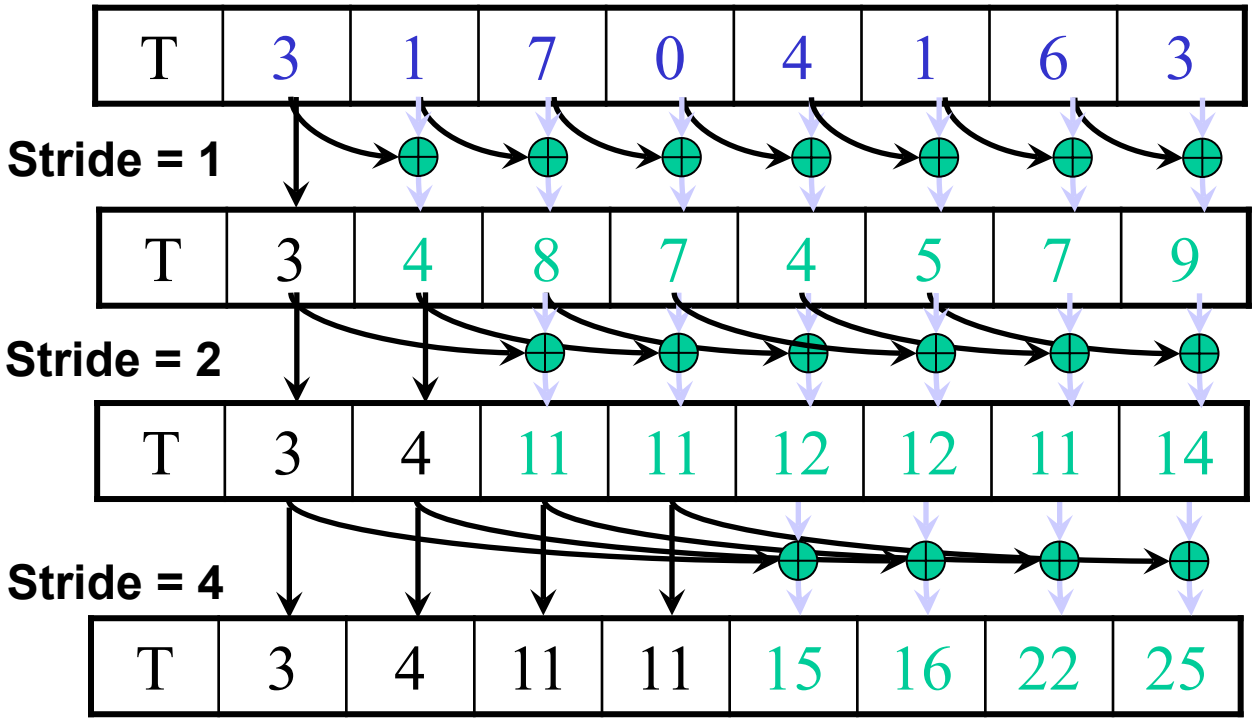
A Kogge-Stone Parallel Scan Algorithm



1. …
2. Assuming n is a power of 2. Iterate log(n) times, stride from 1 to n/2. Threads *stride* to *n-1* *active:* add pairs of elements that are s*tride* elements apart.

Iteration #2
Stride = 2

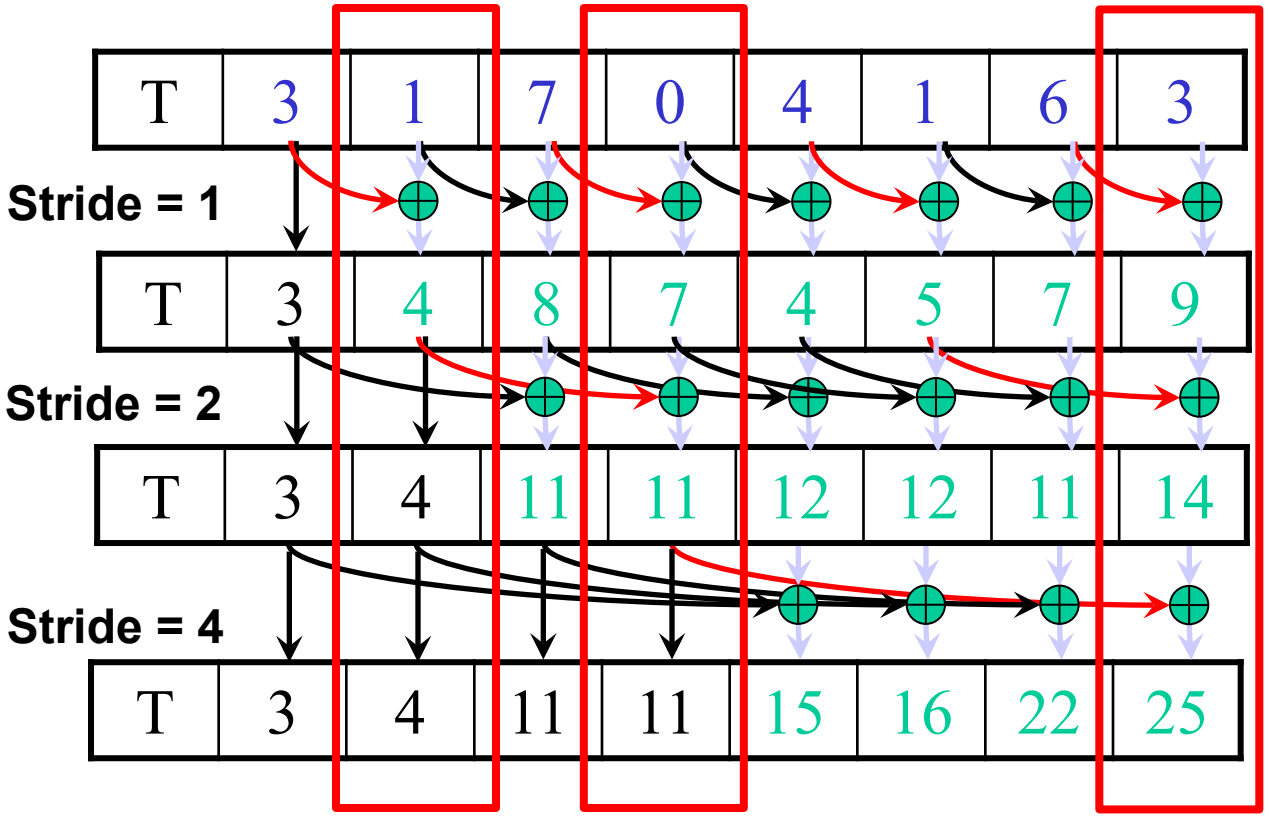
A Kogge-Stone Parallel Scan Algorithm



1. ...
2. …
3. Write output from shared memory to device memory

Iteration #3
Stride = 4

Sharing Computation in Kogge-Stone



Iteration #3
Stride = 4

(*Incomplete*) Implementation

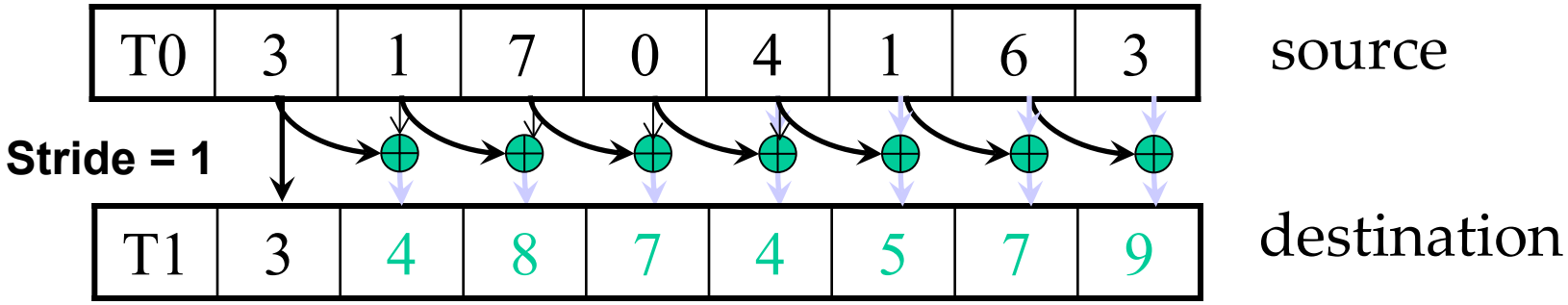
```
__global__
void Kogge_Stone_scan_kernel(float *X, float *Y, int InputSize)
{
 __shared__ float XY[SECTION_SIZE];
 int i = blockIdx.x*blockDim.x + threadIdx.x;
 if (i < InputSize) XY[threadIdx.x] = X[i];

 for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
   __syncthreads();
   if (threadIdx.x >= stride) // This code has a data race condition
      XY[threadIdx.x] += XY[threadIdx.x-stride];
 }
 Y[i] = XY[threadIdx.x];
}
```

Double Buffering

- Use two copies of data T0 and T1
- Start by using T0 as input and T1 as output
- Switch input/output roles after each iteration
  - Iteration 0: T0 as input and T1 as output
  - Iteration 1: T1 as input and T0 and output
  - Iteration 2: T0 as input and T1 as output
- This is typically implemented with two pointers, *source* and *destination* that swap their contents from one iteration to the next
- This eliminates the need for the second __syncthreads() call

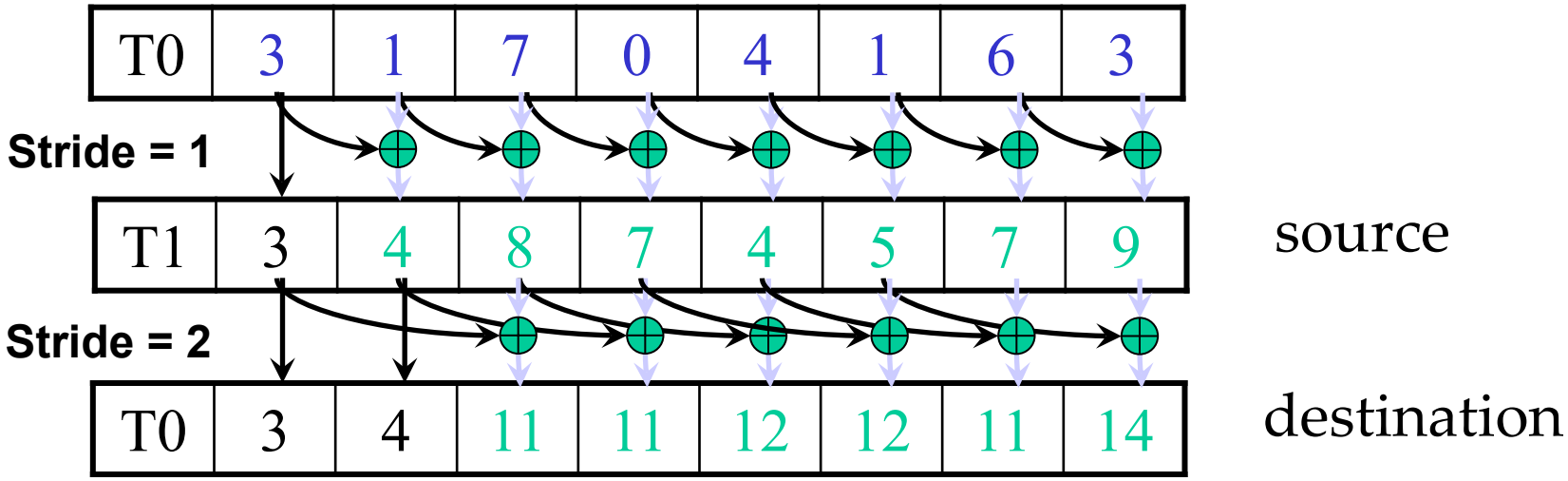
A Double-Buffered Kogge-Stone Parallel Scan Algorithm



Iteration #1
Stride = 1

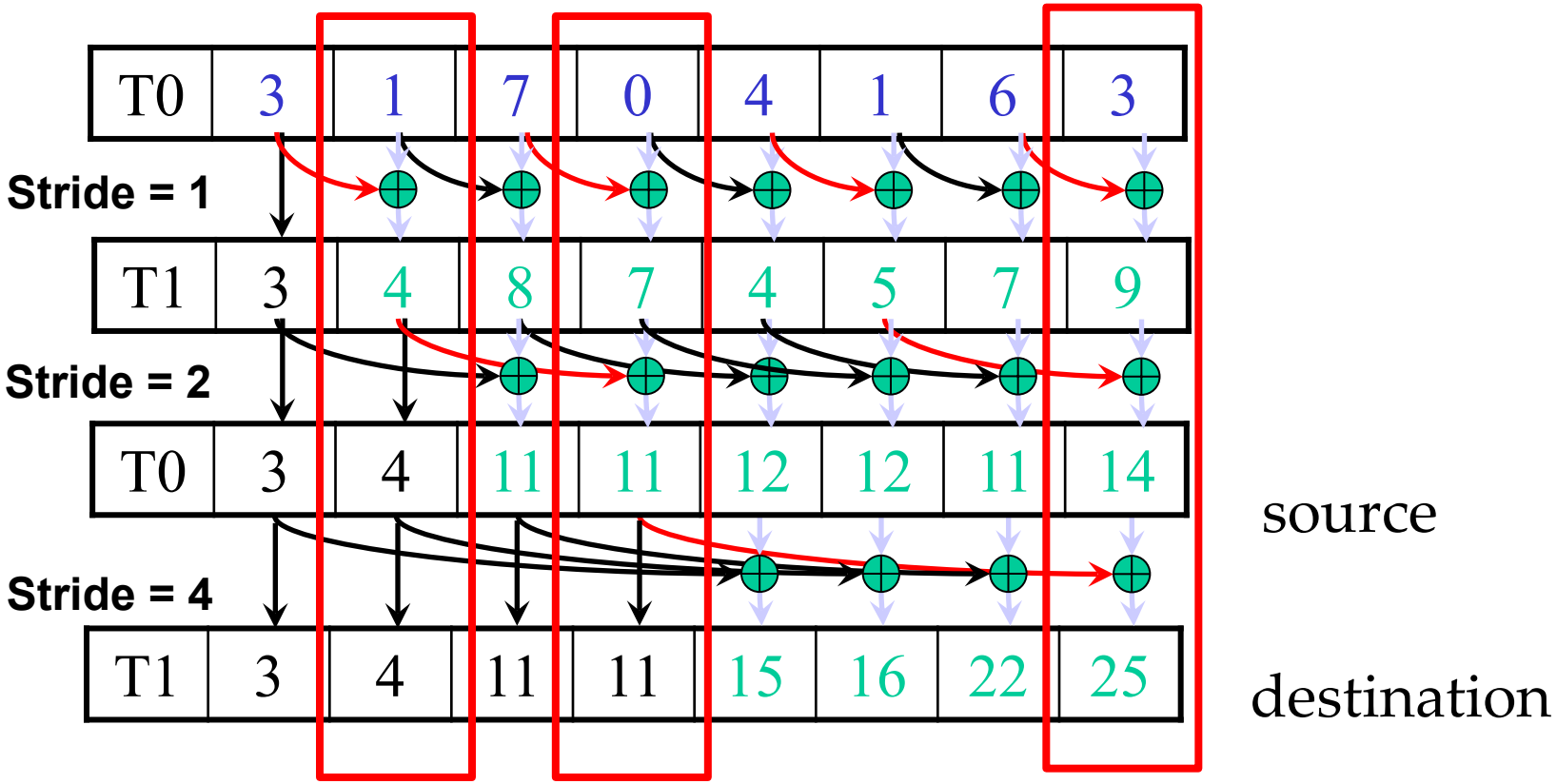
- source = &T0[0]; destination = &T1[0];
- Each iteration requires only one syncthreads()
  - syncthreads(); // make sure that input is in place
  - float destination[*j*] = source[*j*] + source[*j-stride*];
  - temp = destination; destination = source; source = temp;
- After the loop, write destination contents to global memory

A Double-Buffered Kogge-Stone Parallel Scan Algorithm



Iteration #2
Stride = 2

Sharing Computation in a Double-Buffered Kogge-Stone



Iteration #3
Stride = 4

Work Efficiency Analysis

- A Kogge-Stone scan kernel executes log(n) parallel iterations
  - The steps do (n-1), (n-2), (n-4),..(n- n/2) add operations each
  - Total # of add operations: n * log(n) - (n-1) → O(n*log(n)) work

- This scan algorithm is not very work efficient
  - Sequential scan algorithm does *n* adds
  - A factor of log(n) hurts: 20x for 1,000,000 elements!
  - Typically used within each block, where n ≤ 1,024

- A parallel algorithm can be slow when execution resources are saturated due to low work efficiency

Improving Efficiency

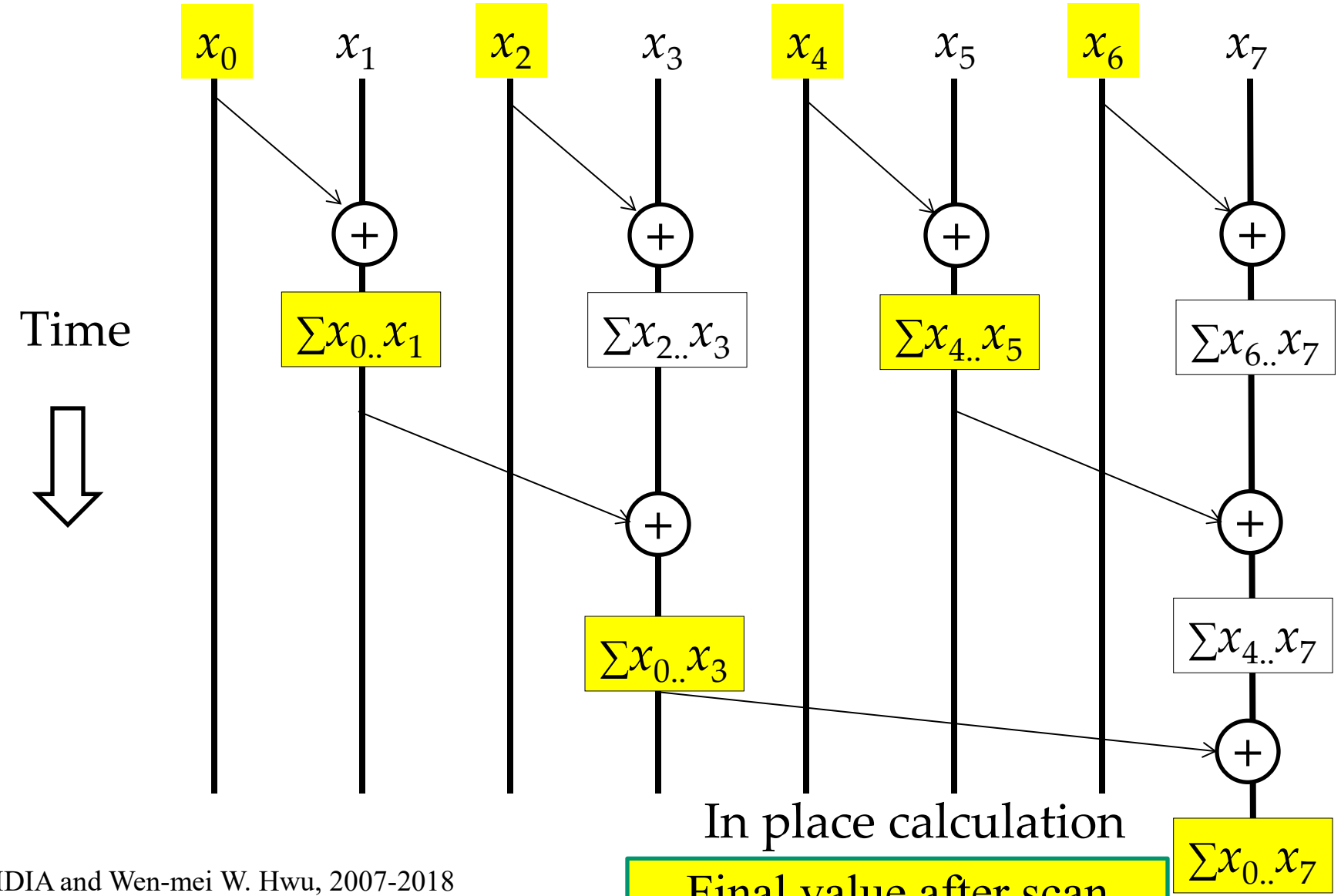
- A common parallel algorithm pattern:

  Balanced Trees

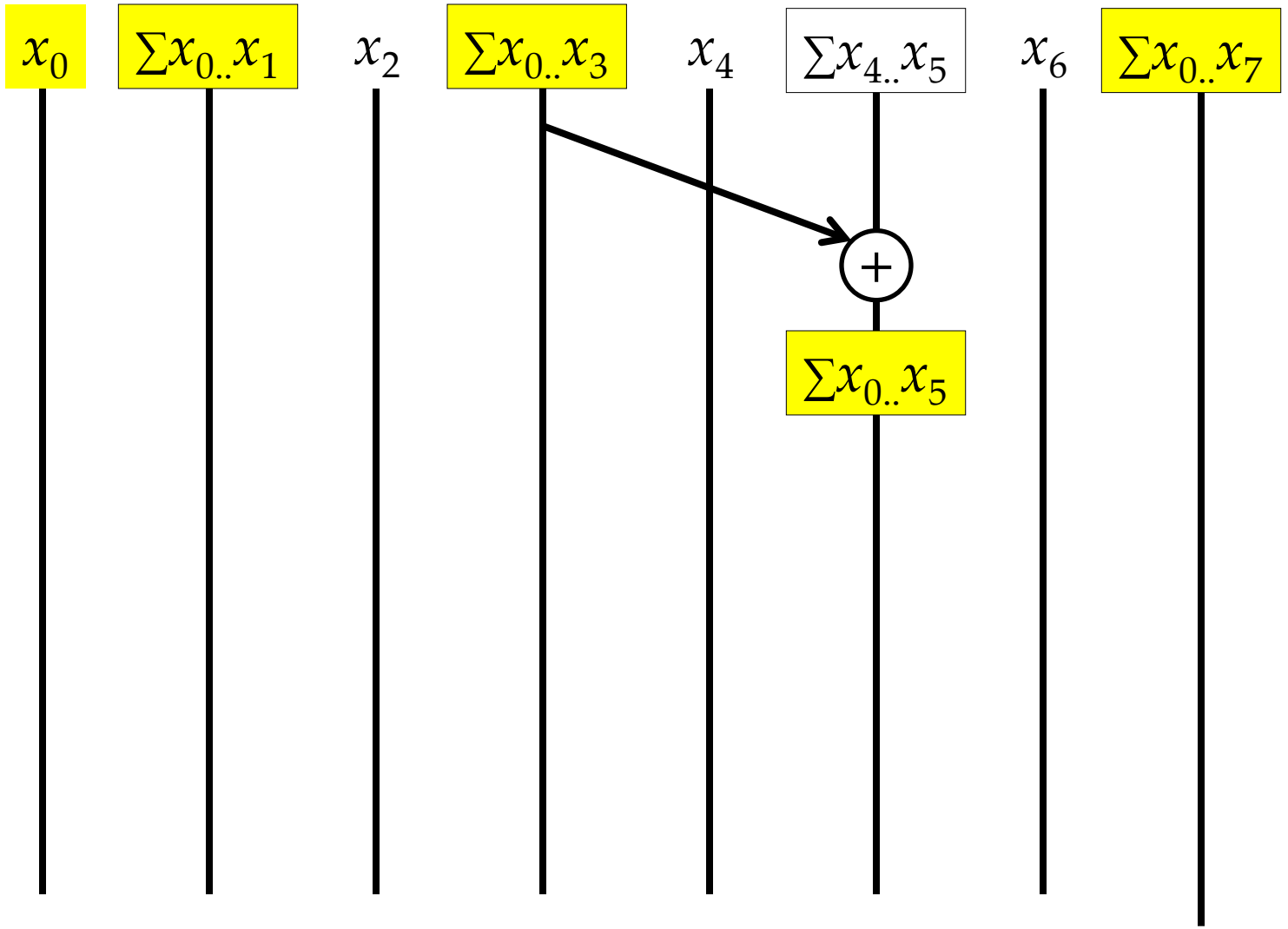
  - Build a balanced binary tree on the input data and sweep it to and from the root
  - Tree is not an actual data structure, but a conceptual pattern

- For scan:
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  - Traverse back up the tree building the scan from the partial sums

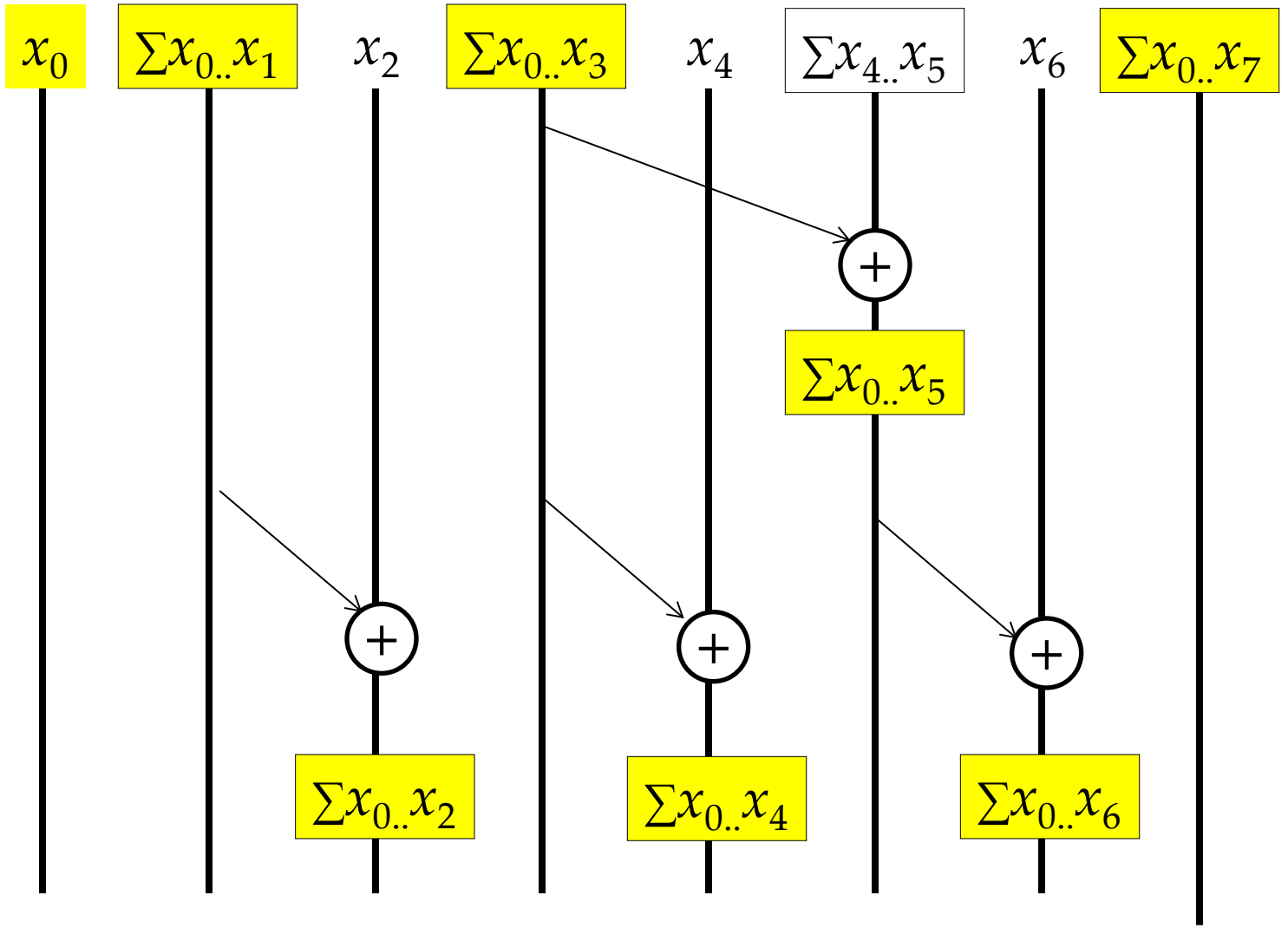
Brent-Kung Parallel Scan Step



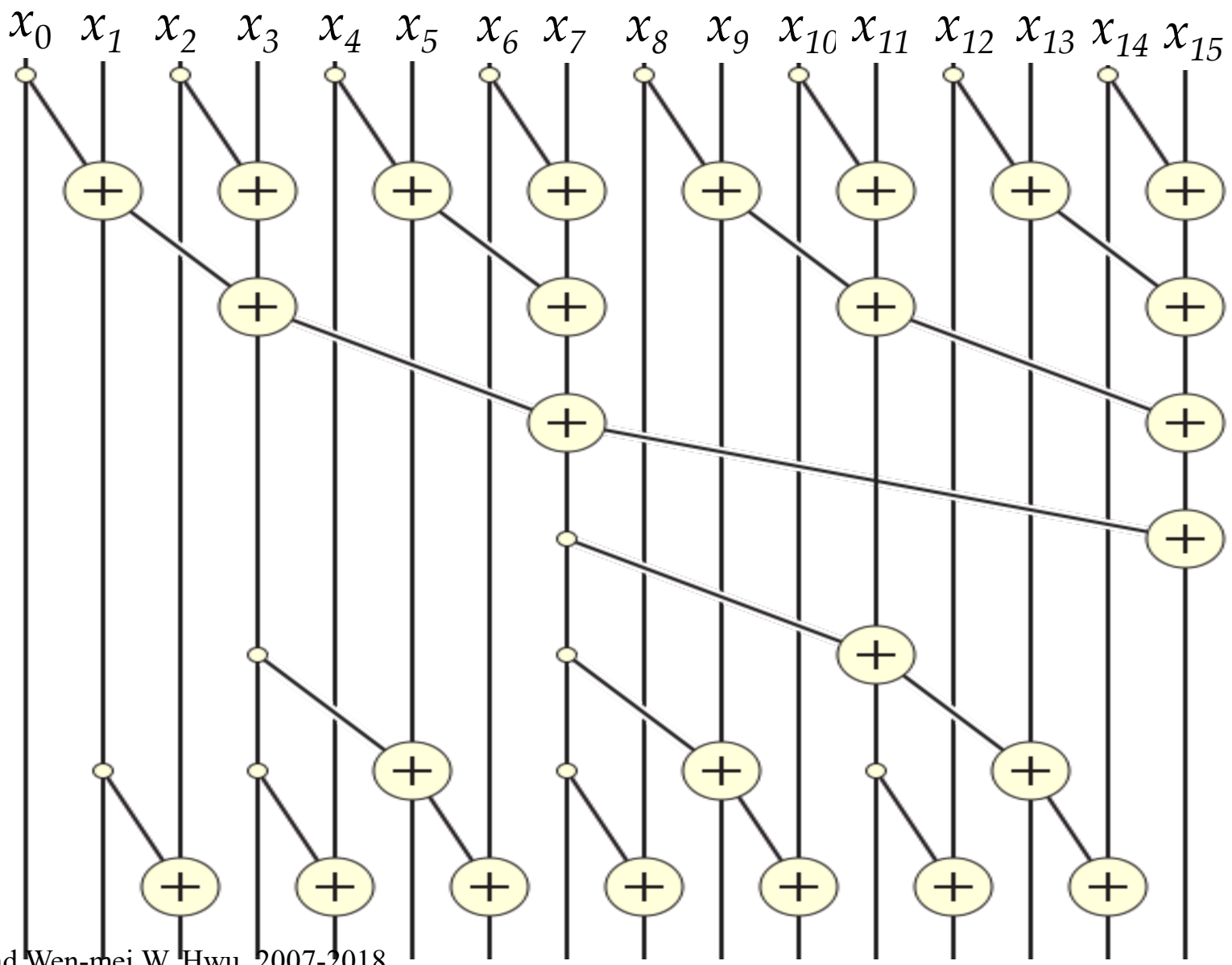
Inclusive Post-Scan Step



Inclusive Post Scan Step



Putting it Together (Data View)



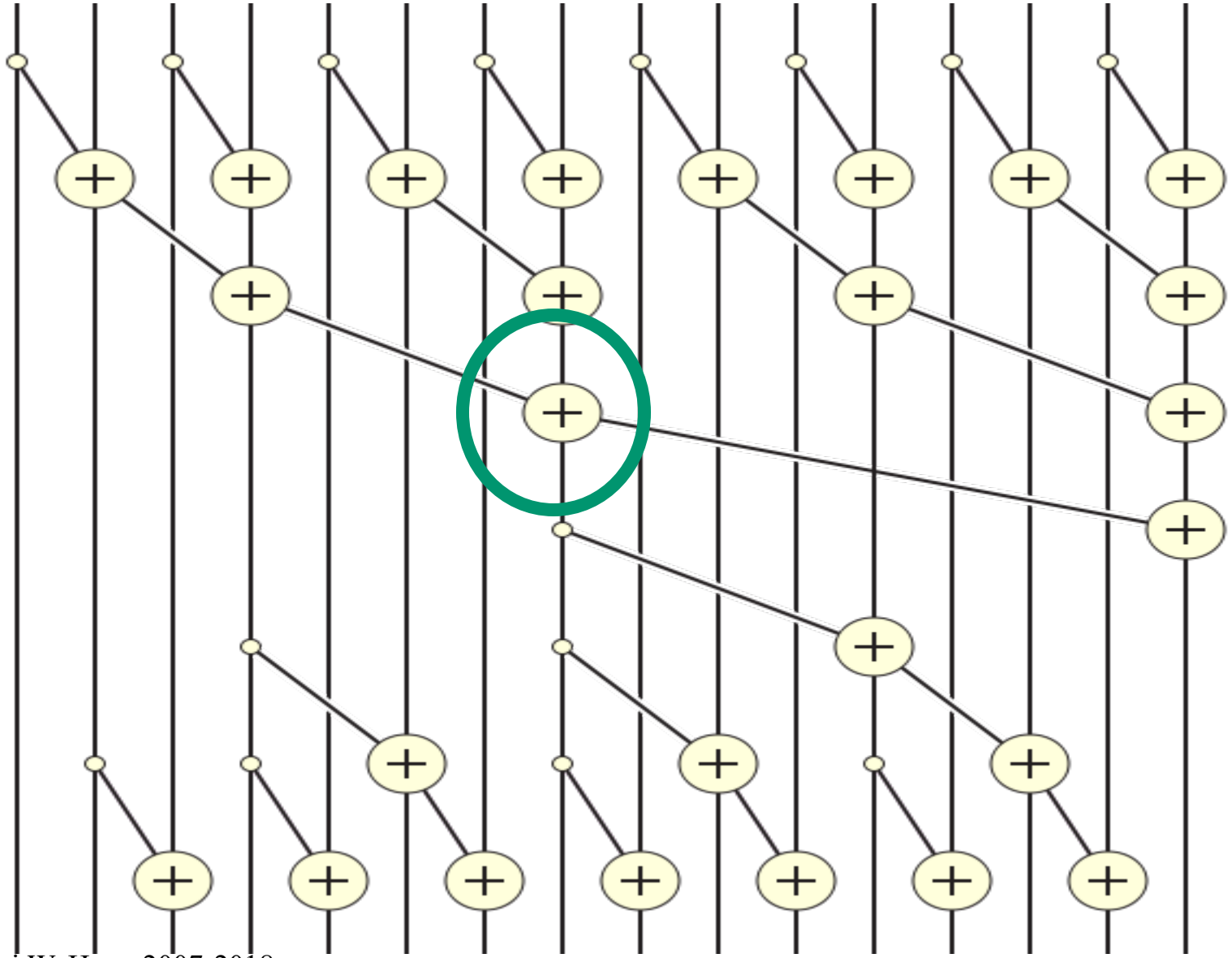
Reduction Step Kernel Code

```
// float T[2*BLOCK_SIZE] is in shared memory
// for previous slide, BLOCK_SIZE is 8

int stride = 1;
while(stride < 2*BLOCK_SIZE) {
    __syncthreads();
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE && (index-stride) >= 0)
        T[index] += T[index-stride];
    stride = stride*2;
}
```

```
// In our example,
// threadIdx.x+1    = 1, 2, 3, 4, 5, 6, 7, 8
// stride = 1, index = 1, 3, 5, 7, 9, 11, 13, 15
```

Putting it Together



Post Scan Step (Distribution Tree)

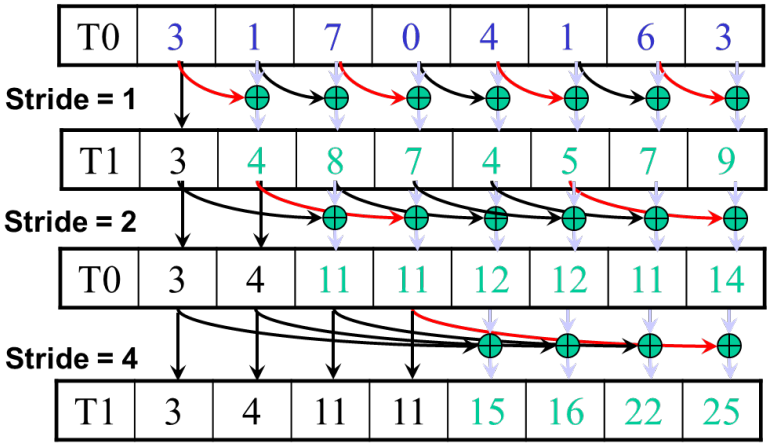
```
int stride = BLOCK_SIZE/2;
while(stride > 0) {
    __syncthreads();
    int index = (threadIdx.x+1)*stride*2 - 1;
    if ((index+stride) < 2*BLOCK_SIZE)
        T[index+stride] += T[index];
    stride = stride / 2;
 }
```

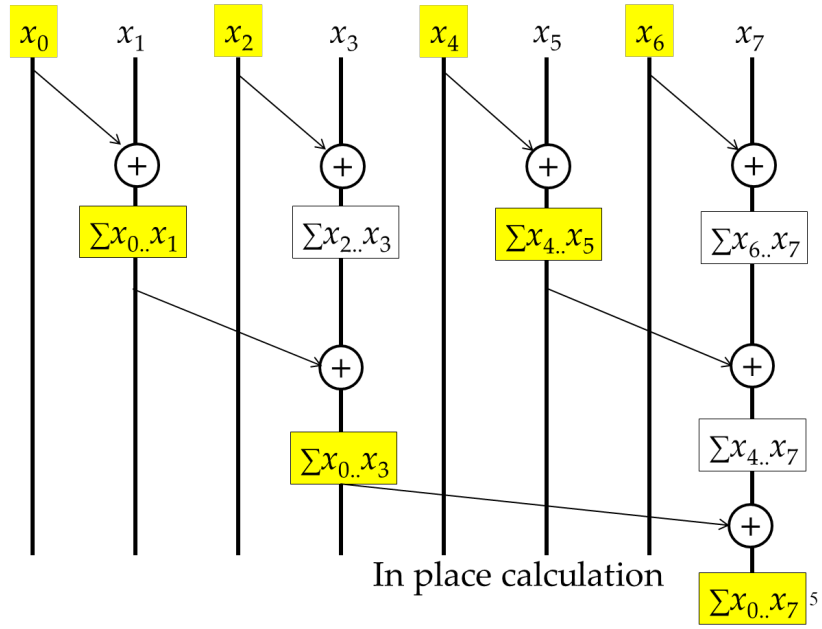
```
// In our example,
// BLOCK_SIZE=8 stride=4, 2, 1
// for first iteration, active thread = 0 index = 7, stride = 11
```

Work Analysis

- The parallel Scan executes 2*log(n) parallel iterations
  - log(n) in reduction and log(n) in post scan
  - The iterations do n/2, n/4,..1, (2-1), …., (n/4-1), (n/2-1) useful adds
  - In our example, n = 16, the number of useful adds is 16/2 + 16/4 + 16/8 + 16/16 + (16/8-1) + (16/4-1) + (16/2-1)
  - Total adds: (n-1) + (n-2) – (log(n) -1) = 2*(n-1) – log(n)→ O(n) work
- The total number of adds is no more than twice of that done in the efficient sequential algorithm
  - The benefit of parallelism can easily overcome the 2× work when there is sufficient hardware

Kogge-Stone vs. Brent-Kung

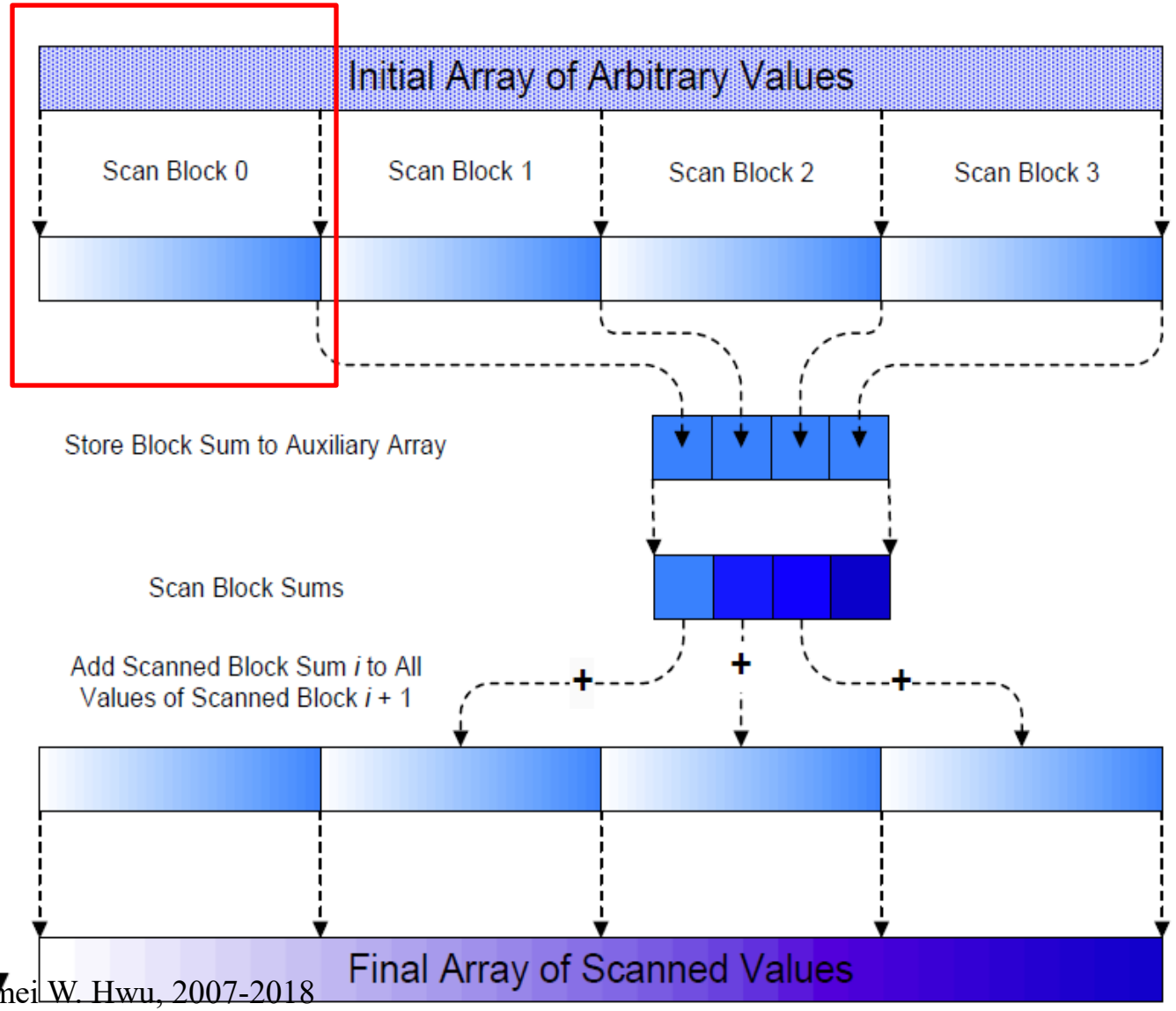




- Brent-Kung uses half the number of threads compared to Kogge-Stone
  - Each thread should load two elements into the shared memory
- Brent-Kung takes twice the number of steps compared to Kogge-Stone
  - Kogge-Stone is more popular for parallel scan with blocks in GPUs

Overall Flow of Complete Scan
A Hierarchical Approach

Scan kernel



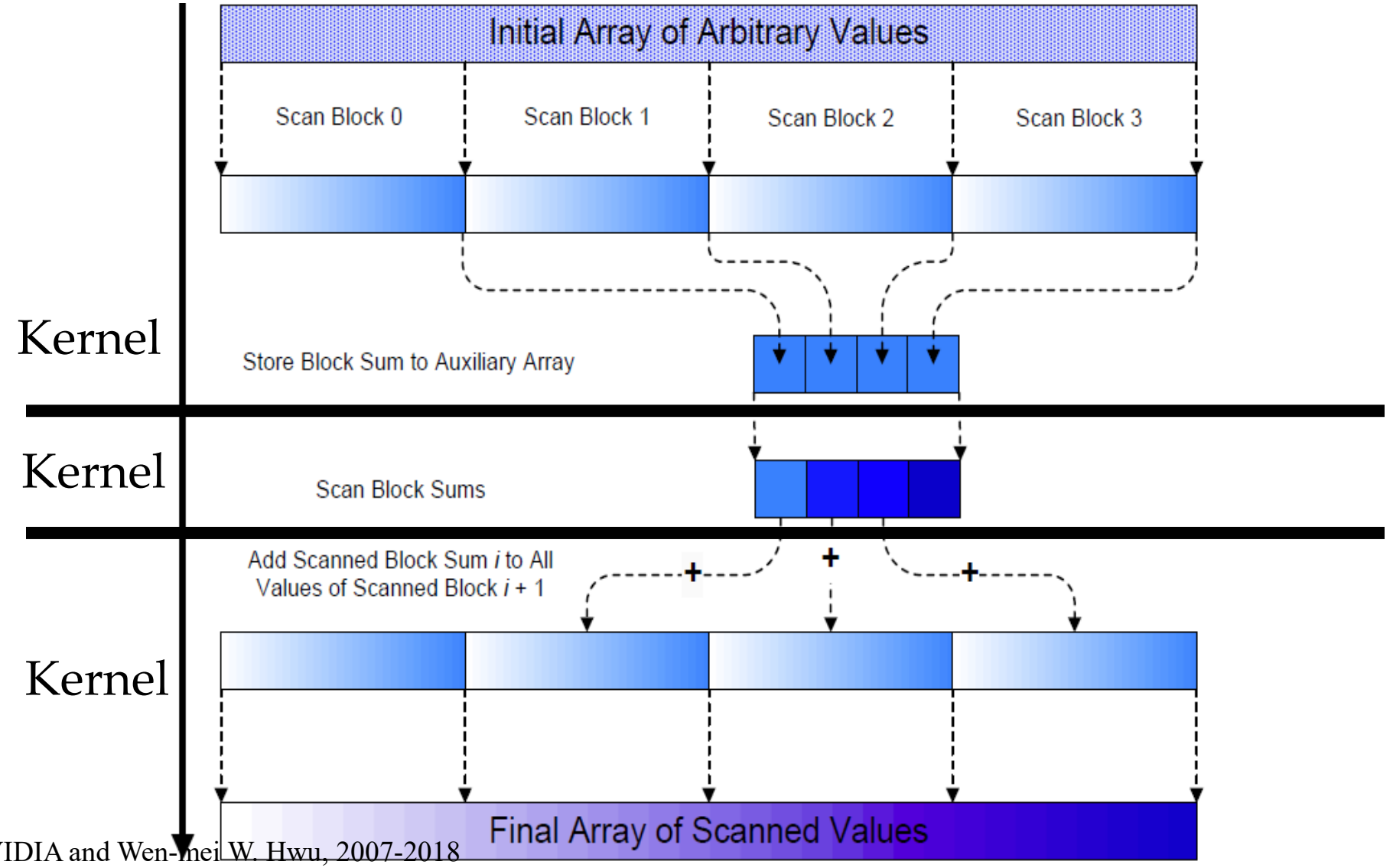
Using Global Memory Contents in CUDA

- Data in registers and shared memory of one thread block are not visible to other blocks
- To make data visible, the data has to be written into global memory
- However, any data written to the global memory are not visible until a memory fence. This is typically done by terminating the kernel execution
- Launch another kernel to continue the execution. The global memory writes done by the terminated kernels are visible to all thread blocks.

Scan of Arbitrary Length Input

- Build on the scan kernel that handles up to 2*blockDim.x elements from Brent-Kung
  - For Kogge-Stone, have each section of blockDim.x elements assigned to a block
- Have each block write the sum of its section into a Sum array using its blockIdx.x as index
- Run parallel scan on the Sum array
  - May need to break down Sum into multiple sections if it is too big for a block
- Add the scanned Sum array values to the elements of corresponding sections

Overall Flow of Complete Scan A Hierarchical Approach



(Exclusive) Scan Definition

Definition: *The exclusive* scan *operation takes a binary associative operator* $\oplus$*, and an array of n elements*

$$[x_0, x_1, \ldots, x_{n-1}]$$

and returns the array

$$[0, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2})].$$

Example: If $\oplus$ is addition, then the exclusive scan operation on

$$[3\ \ 1\ \ 7\ \ 0\ \ 4\ \ 1\ \ 6\ \ 3],$$

would return

$$[0\ \ 3\ \ 4\ 11\ \ 11\ 15\ 16\ 22].$$

Why Exclusive Scan

- To find the beginning address of allocated buffers

- Inclusive and Exclusive scans can be easily derived from each other; it is a matter of convenience

	[3 1 7 0 4 1 6 3]
Exclusive	[0 3 4 11 11 15 16 22]
Inclusive	[3 4 11 11 15 16 22 25]

A simple exclusive scan kernel

- Adapt an inclusive, Kogge-Stone scan kernel
  - Block 0:
    - Thread 0 loads 0 into (shared) XY[0]
    - Other threads load (global) X[threadIdx.x-1] into XY[threadIdx.x]
  - All other blocks:
    - All thread load X[blockIdx.x*blockDim.x+threadIdx.x-1] into XY[threadIdex.x]

- Similar adaption for Brent-Kung kernel but pay attention that each thread loads two elements
  - Only one zero should be loaded
  - All elements should be shifted by only one position

ANY MORE QUESTIONS?
READ CHAPTER 8

Problem Solving

- Q: Suppose we have a kernel function that performs partial sum reduction. The block dimension is (64,1,1). During the **second** iteration of the for loop, is there any warp that has a control divergence?

```
__shared__ int partialSum[2*BLOCK_SIZE];
for (int stride = blockDimx.x; stride >= 1; stride = stride/2)  {
   __syncthreads();
   if (threadIdx.x < stride) partialSum[t] += partialSum[t + stride];
}
```

- A:
  - No

Problem Solving

- Q: Consider a kernel that performs Brent-Kung scan algorithm and assume that there are 1024 elements in a section, and a warp size is 32. In which iteration will there be at least one warp that has a control divergence? The stride is 1 for the first iteration.

- A:
  - In 6th iteration

Problem Solving

- Q: Suppose we use Brent-Kung in a hierarchical approach to perform parallel scan on a 1D input array of 2^{42} elements. We use 1024 threads per block in all our Brent-Kung kernels and our GPU supports at most 2048 blocks per grid. Each block processes 2048 elements. What is the best approximation of the number of floating-point add operations performed per thread block in the reduction and post scan steps in the Brent-Kung kernel (summed together)?

- A:
  - 2048 x 2