Parallel Patterns: Scan (Prefix Sum)

Prof. Seokin Hong

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

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

Example: If $\oplus$ is addition, then the scan operation on the array [3 1 7 0 4 1 6 3], would return [3 4 11 11 15 16 22 25].

A Scan Application Example

- Assume that we have a 100-inch bread to feed 10
- We know how much each person wants in inches
  - [3 5 2 7 28 4 3 0 8 1]
- How do we cut the bread quickly?
- How much will be left

- Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.
- Method 2: calculate prefix-sum array
  - [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)

Typical Applications of Scan

- Scan is a simple and useful parallel building block
  - Convert recurrences from sequential :

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

  - into parallel:

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

- Useful for many parallel algorithms:
  - radix sort
  - quicksort
  - String comparison
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations

- Other Applications
  - Allocating memory to parallel threads
  - Allocating memory buffer to communication channels

An Inclusive Sequential Scan

Given a sequence $[x_0, x_1, x_2, ...]$

Calculate output $[y_0, y_1, y_2, ...]$

Such that

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

...

Using a recursive definition

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

An 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 Parallel Scan

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

$$y_0 = x_0$$

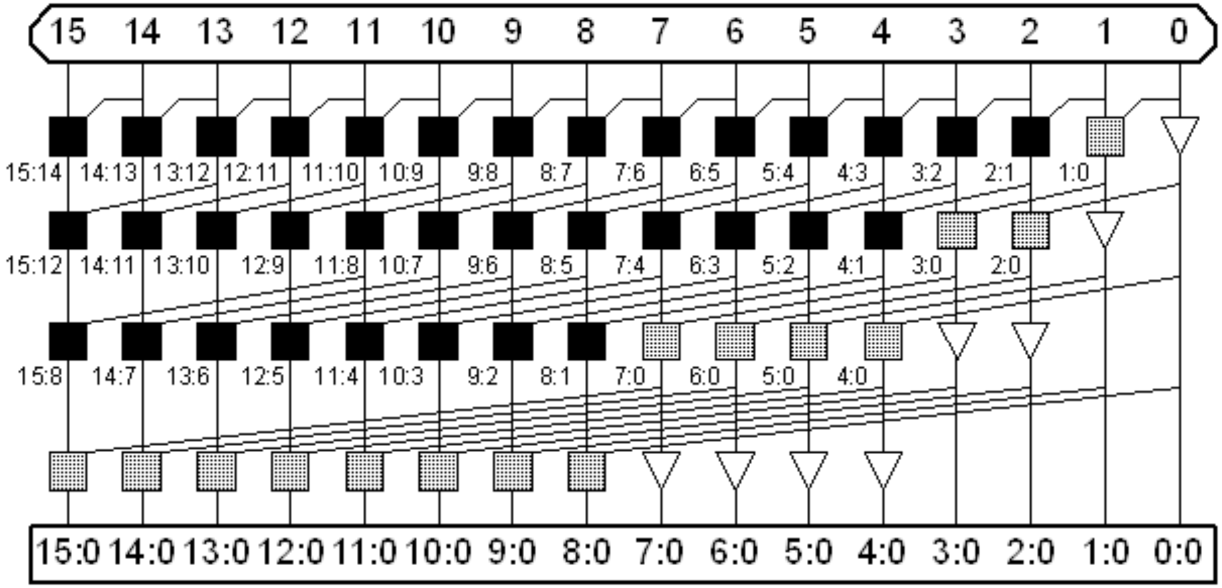
$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

“Parallel programming is easy as long as you do not care about performance.” ☹

Parallel Scan using Reduction Trees

- Calculate each output element as the reduction of all previous elements
  - Some partial sums will be shared among the calculation of output elements
  - Based on hardware adder design by Peter Kogge and Harold Stone at IBM in the 1970s – Kogge-Stone Trees



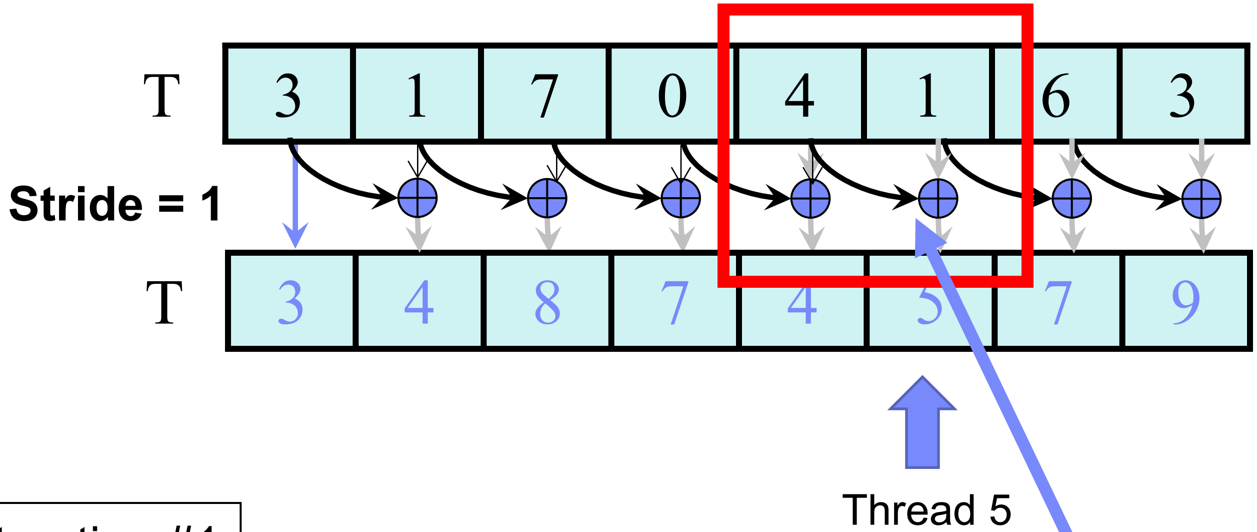
A Kogge-Stone Parallel Scan Algorithm

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

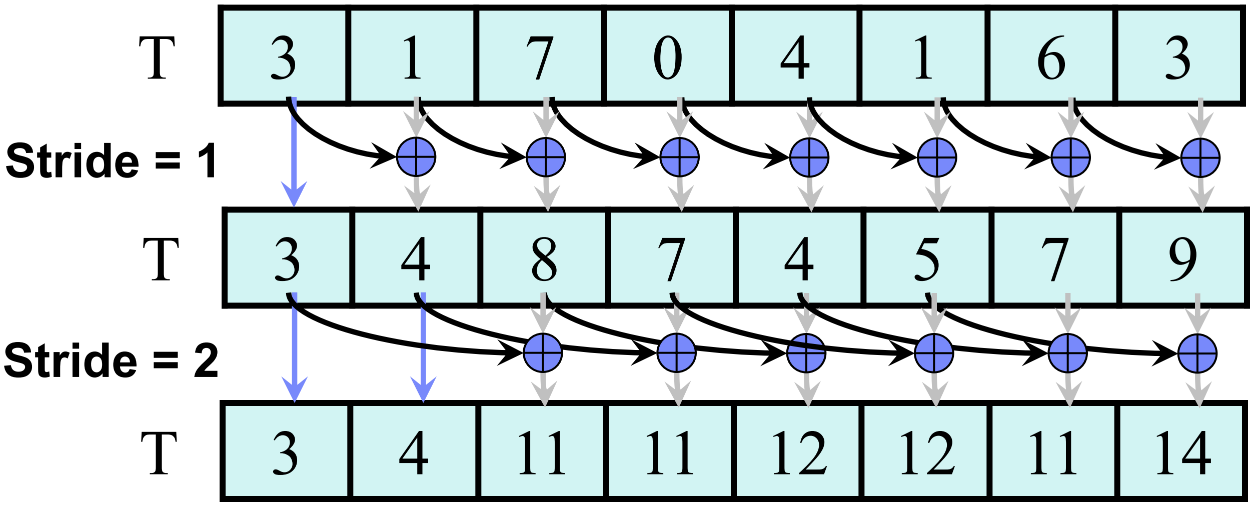


1. Load input from global memory into shared memory array T
2. Iterate log(n) times, stride from 1 to ceil(n/2.0).

Iteration #1
Stride = 1

- Thread *j* adds elements *j* and *j-stride* from T and writes result into shared memory buffer T
- Each iteration requires two syncthreads
  - syncthreads(); // make sure that input is in place
  - float temp = T[j] + T[k - stride];
  - syncthreads(); // make sure that previous output has been consumed
  - T[j] = temp;

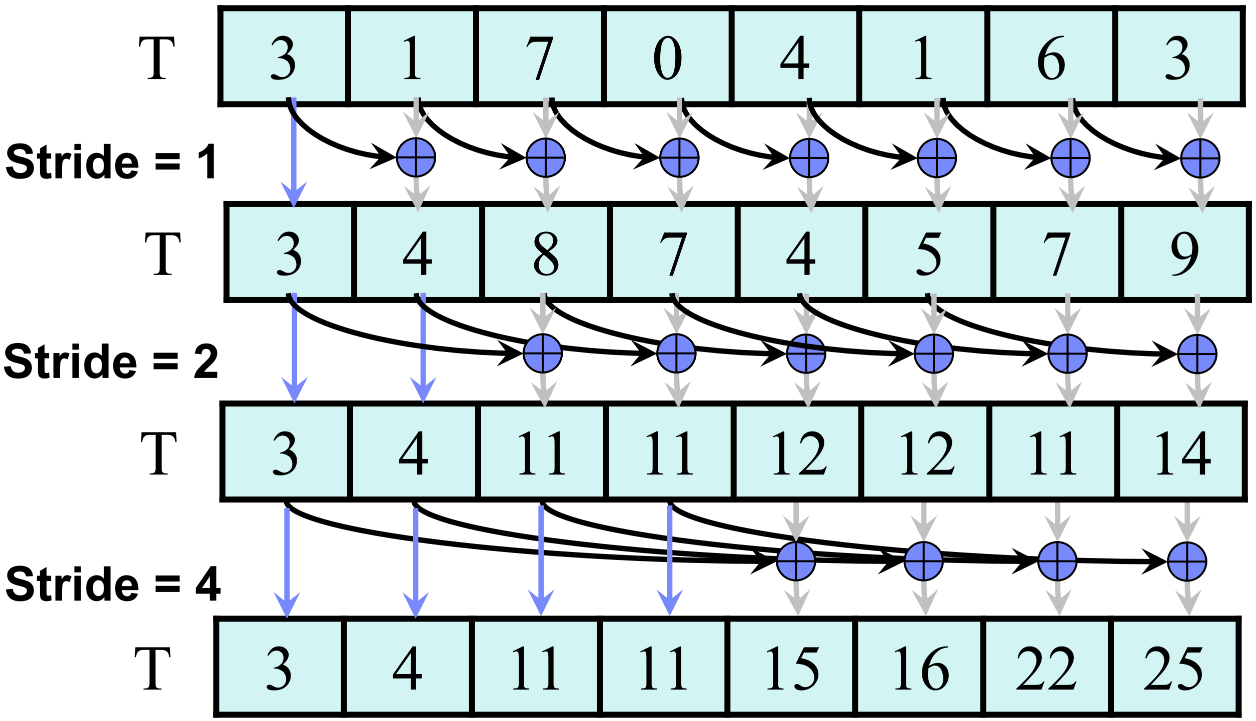
A Kogge-Stone Parallel Scan Algorithm



1. Load input from global memory into shared memory array T
2. Iterate log(n) times, stride from 1 to ceil(n/2.0).

Iteration #2
Stride = 2

A Kogge-Stone Parallel Scan Algorithm



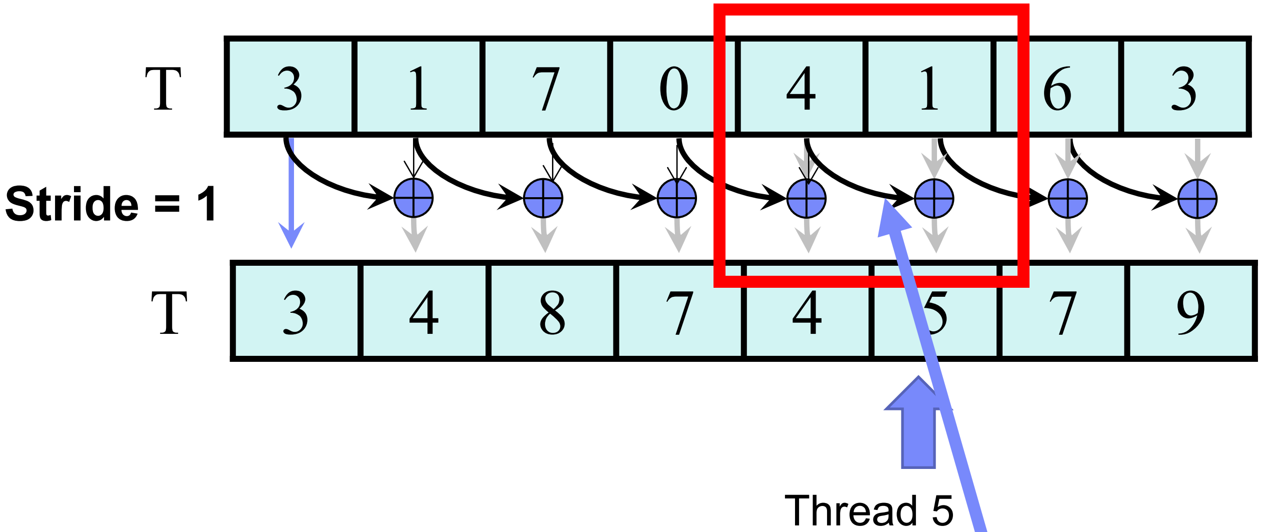
1. Load input from global memory to shared memory.
2. Iterate log(n) times, stride from 1 to ceil(n/2.0).
3. Write output from shared memory to device memory

Iteration #3
Stride = 4

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

A Double-Buffered Kogge-Stone Parallel Scan Algorithm



1. ……
2. Iterate log(n) times, stride from 1 to ceil(n/2.0).

- Thread *j* adds elements *j* and *j-stride* from T and writes result into shared memory buffer T
- 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;

Iteration #1
Stride = 1

Work Efficiency Analysis

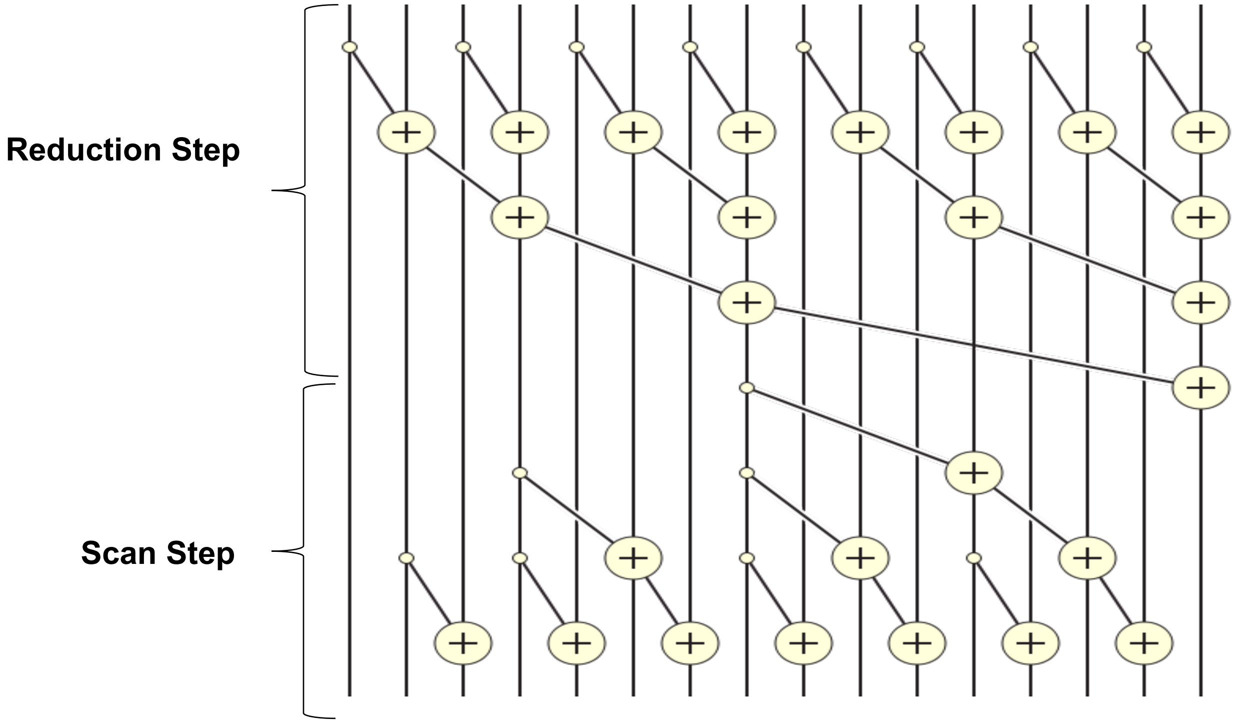
- A Kogge-Stone scan kernel executes log(n) parallel iterations
  - The steps do (n-1), (n-2), (n-4),..(n- n/2) add operations each
  - Total # of add operations: n * log(n) - (n-1) → O(n*log(n)) work

- 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

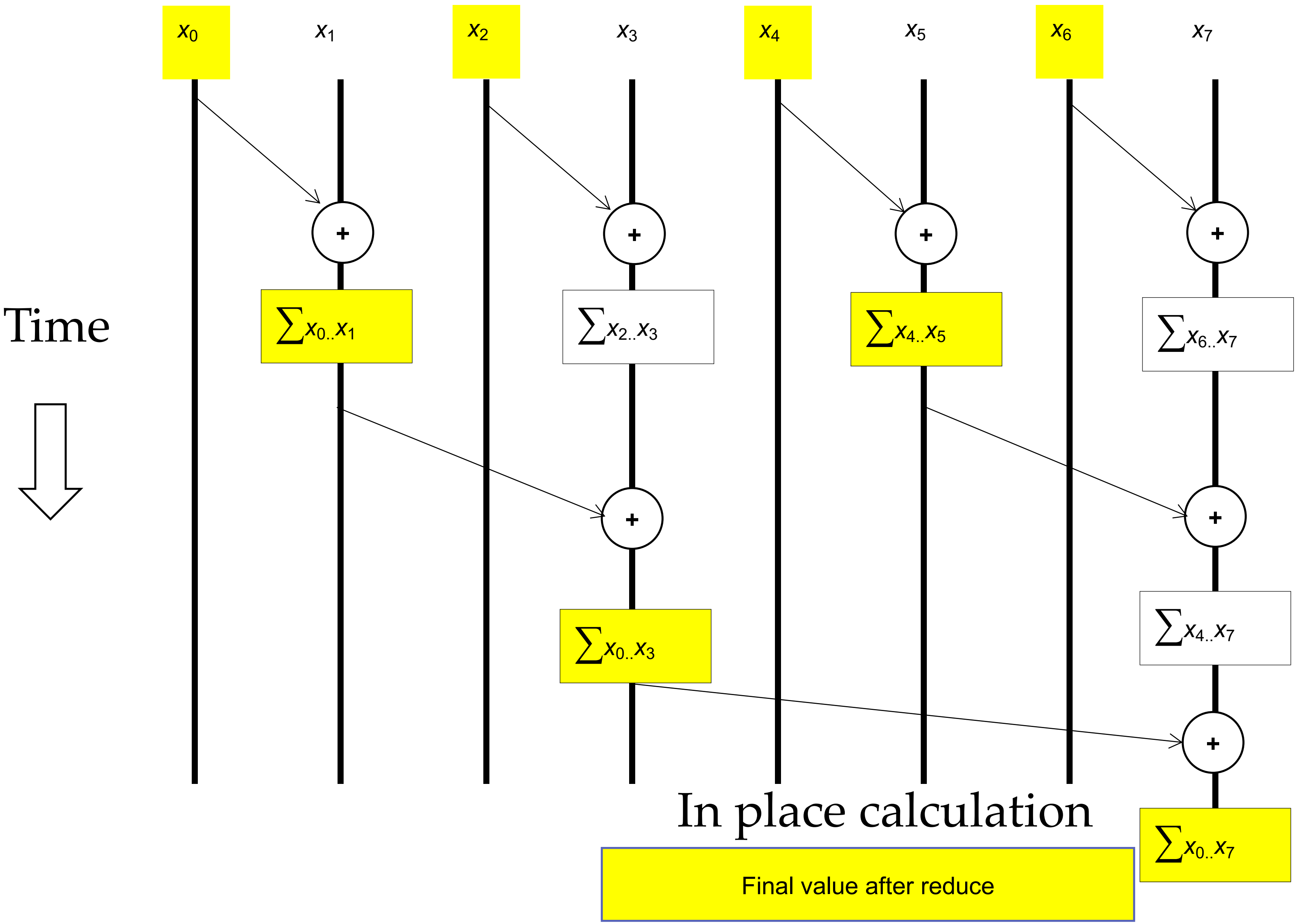
- A parallel algorithm can be slow when execution resources are saturated due to low work efficiency

Brent-Kung Parallel Scan

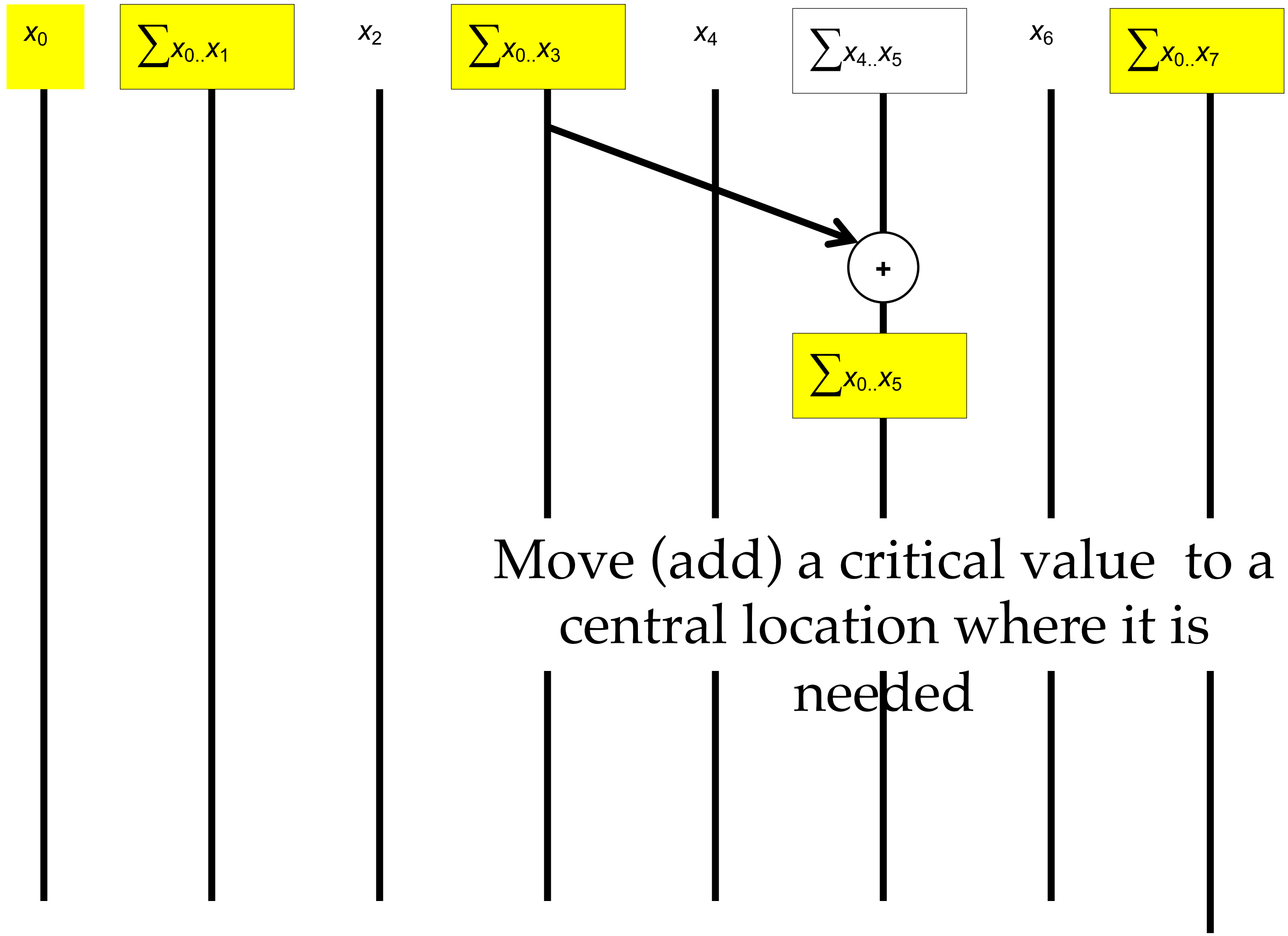
- **Brent-Kung Parallel Scan**
  - **Reduction Step :** Traverse down from leaves to root for building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  - **Scan Step :** Traverse back up the tree for building the scan from the partial sums



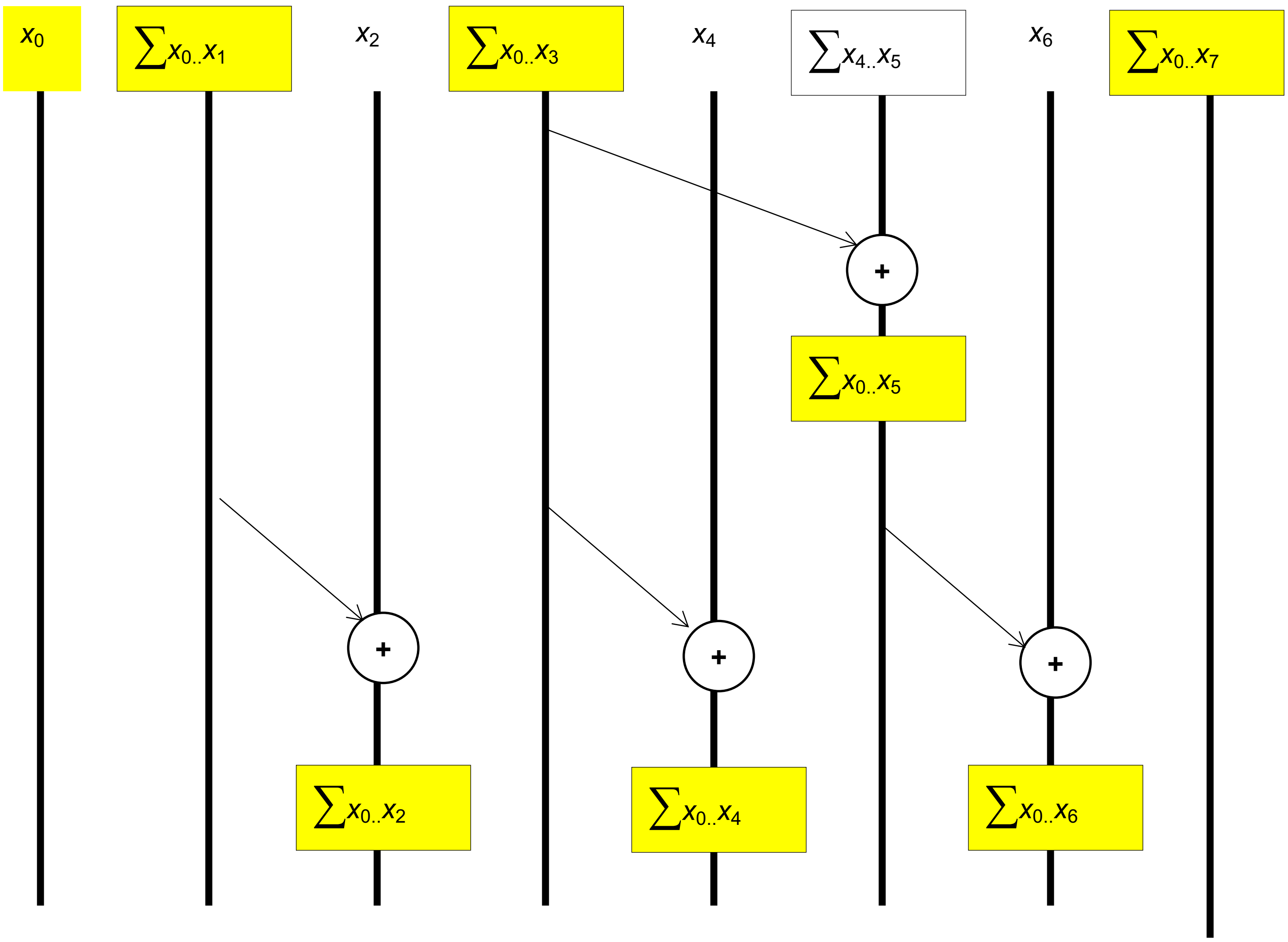
Brent-Kung Parallel Scan - Reduction Step



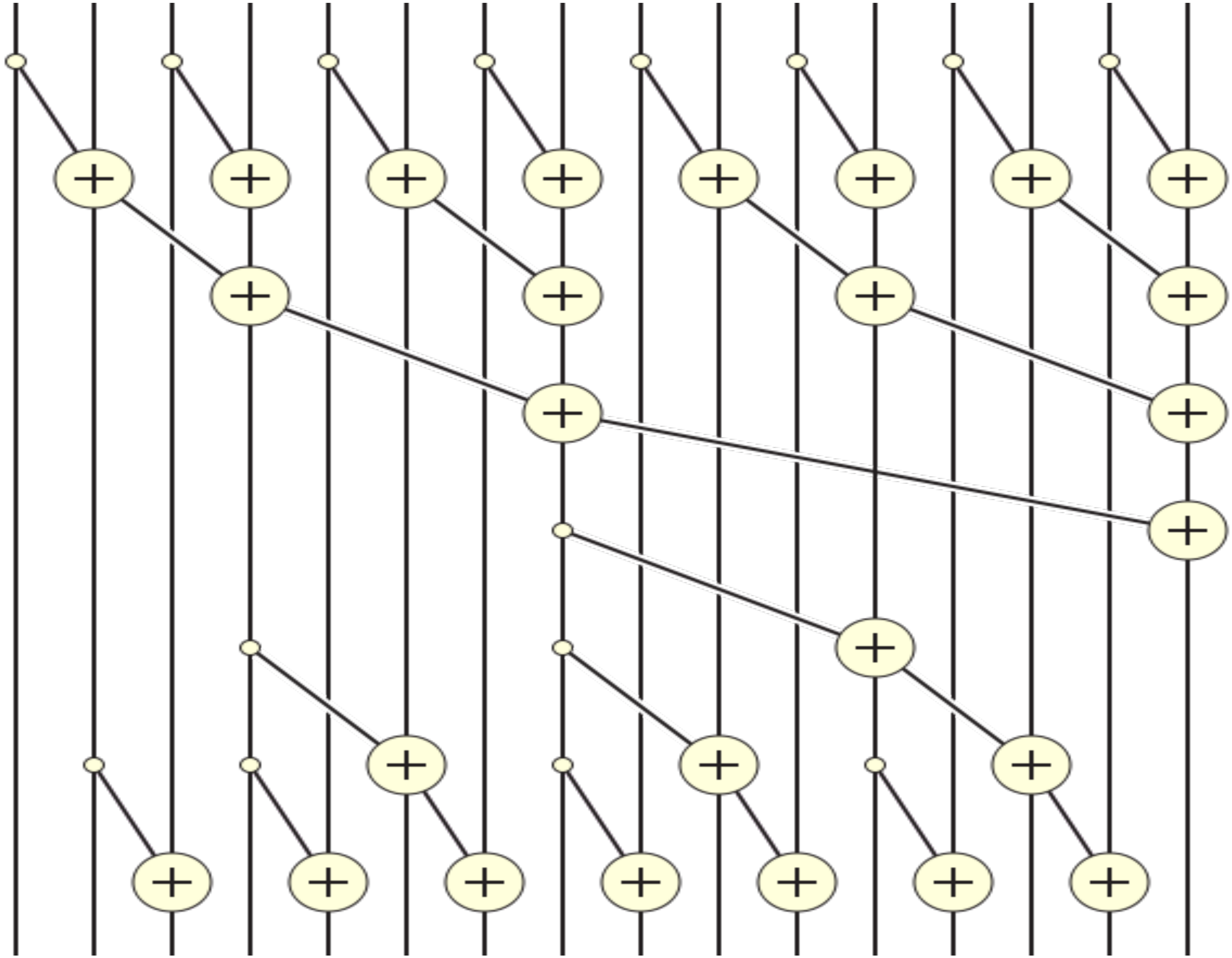
Brent-Kung Parallel Scan - Scan Step



Brent-Kung Parallel Scan - Scan Step

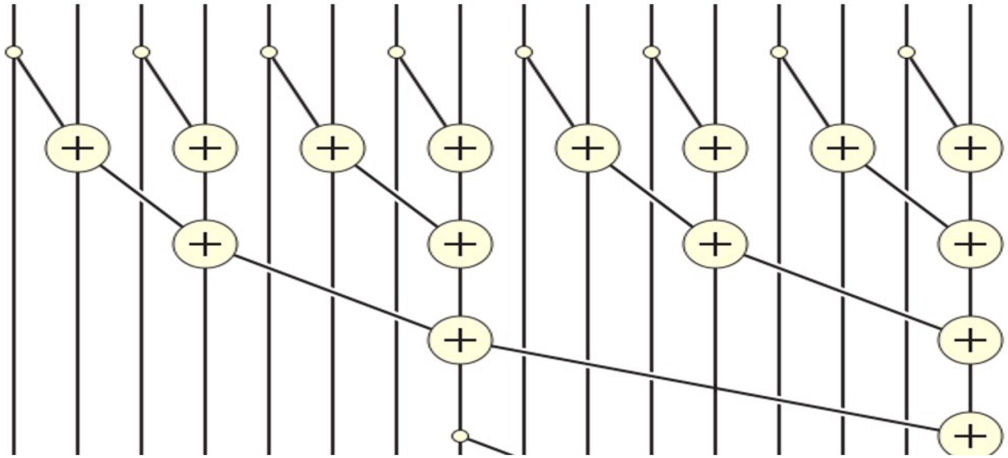


Putting it Together



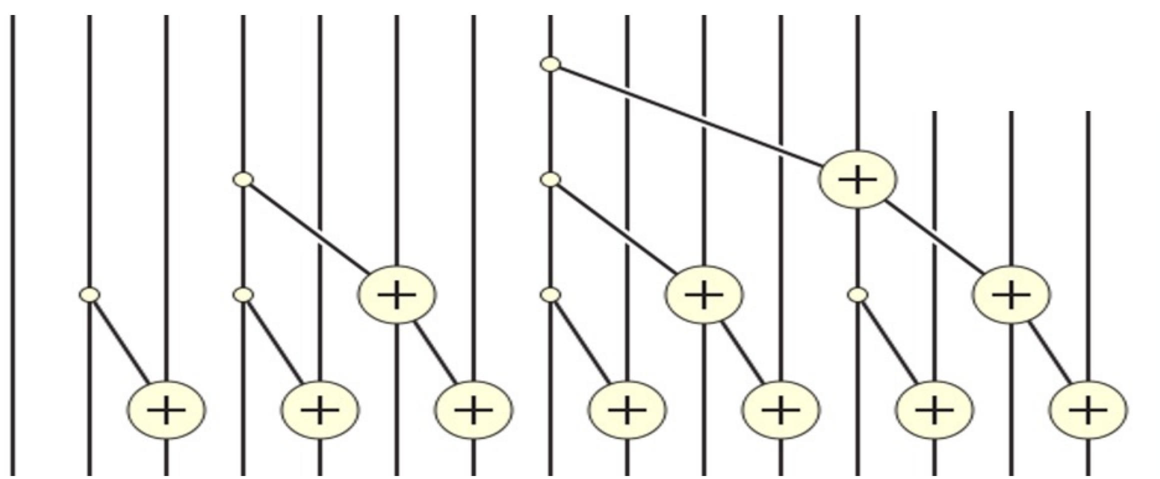
Reduction Step Kernel Code

```
// float T[BLOCK_SIZE] is in shared memory

int stride = 1;
while(stride < BLOCK_SIZE)
  {
     int index = (threadIdx.x+1)*stride*2 - 1;
     if(index < BLOCK_SIZE)
        T[index] += T[index-stride];
     stride = stride*2;

     __syncthreads();
  }
```



Scan Step Kernel Code

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



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 adds 1, …., n/4, n/2 adds
  - Total adds: 2* (n-1) → 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 2X work when there is sufficient hardware

- **Brent-Kung vs Kogge-Stone**
  - 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