Parallel Programming

Data-Parallel Primitives:

Prefix Scan (Prefix Sum)

Overview

- Prefix Scan the Primitive
- Sequential implementation
- Work-Inefficient parallel implementation
- Work-efficient parallel implementation

Prefix Scan

- Frequently used for parallel work assignment and resource allocation, e.g., allocating memory to parallel threads or for communication channels
- A key primitive in many parallel algorithms to convert serial computation into parallel computation
- A foundational parallel computation pattern

Definition of Prefix Scan

Definition: *The* all-prefix-sums *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

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

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

Example of Prefix Scan

- Assume that we have a 100-inch sausage to feed 10 people
- We know how much each person wants in inches:

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

- How do we cut the sausage 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 and cut in parallel:

[3, 8, 10, 17, 45, 49, 52, 52, 60, 61]

(39 inches left)

Building Block for Parallel Algorithms

- Scan is a simple and useful parallel building block
- Sequential

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

- Parallel:

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

- Useful for many parallel algorithms:
  - Radix sort, Quicksort, String comparison, Lexical analysis
  - Stream compaction, Polynomial evaluation
  - Solving recurrences, Tree operations
  - Histograms, ….

Inclusive Sequential Addition

- Given a sequence [x0, x1, x2, ...],
- Calculate output [y0, y1, y2,...]
- Such that
  - y0 = x0
  - y1 = x0 + x1
  - y2 = x0 + x1+ x2
  - …
- Recursive definition: $y_i = y_{i-1} + x_i$

A Work-Efficient 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)

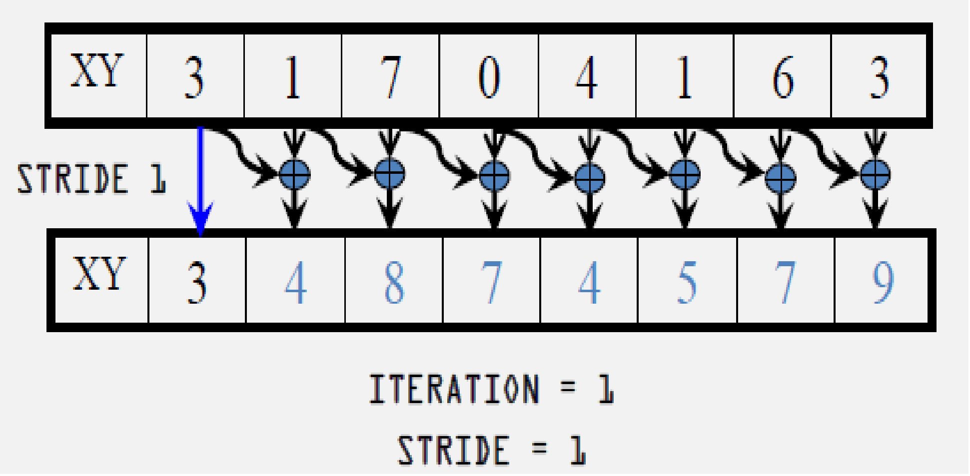
A Simple Parallel Algorithm

- Assign one thread to calculate each y element
- Have every thread to add up all x elements needed for the y element
  - y0 = x0
  - y1 = x0 + x1
  - y2 = x0 + x1+ x2
  - …

A Better Parallel Algorithm

1. Read input from device global memory to shared memory

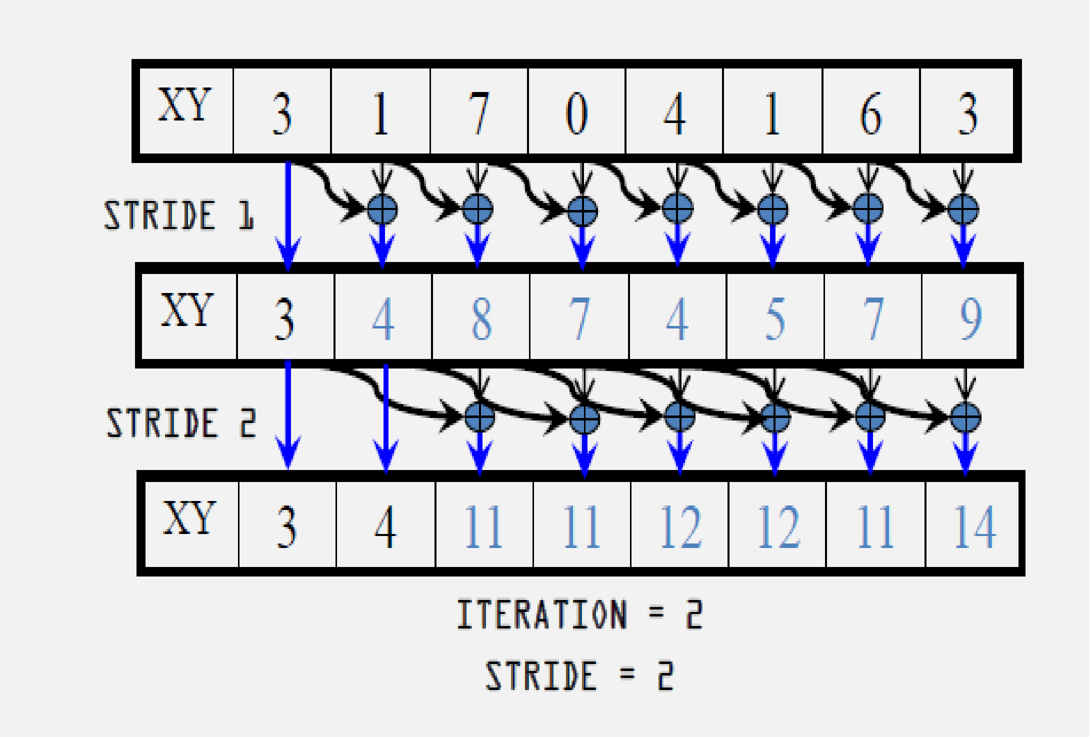
2. Iterate log(n) times; *stride* from 1 to n-1: double *stride* each iteration



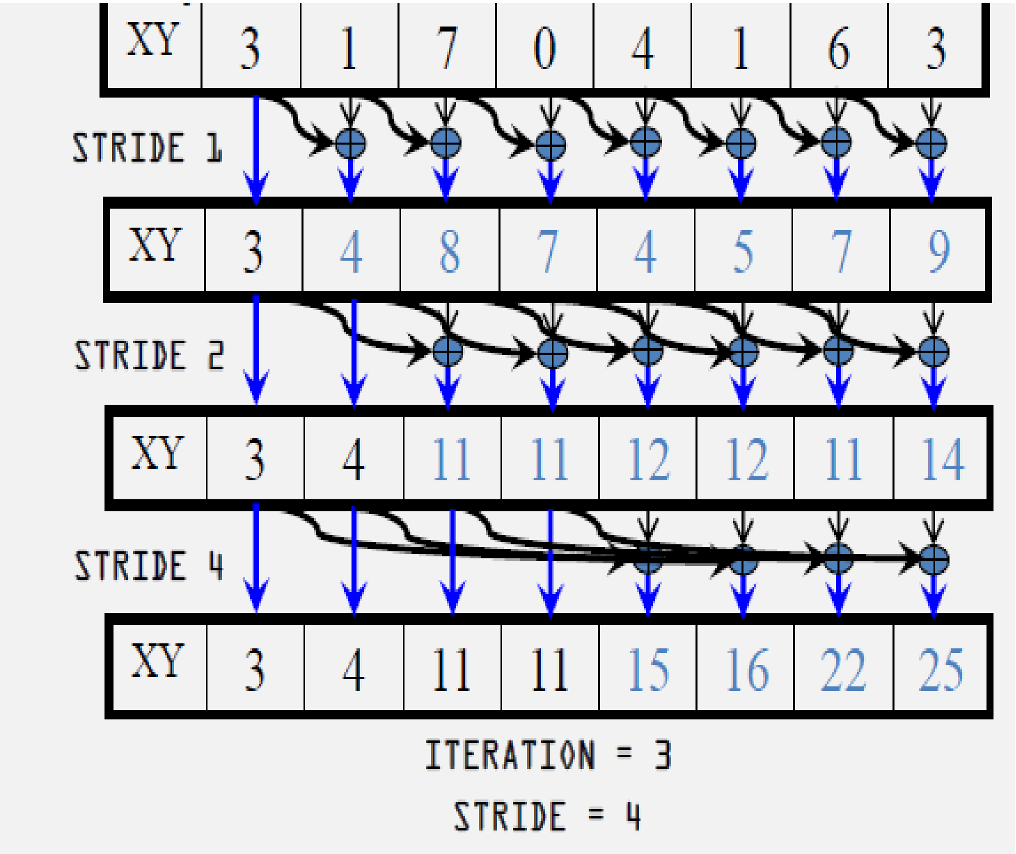
A Better Parallel Algorithm (cont.)

- In each iteration,
  - For each thread *j* from *stride* to *n-1* (*n-stride* threads)
    - Thread *j* adds elements *j* and *j-stride* from shared memory and writes result into element *j* in shared memory
    - Requires barrier synchronization, once before read and once before write

A Better Parallel Algorithm (cont.)



A Better Parallel Algorithm (cont.)



Handling Dependencies

- During every iteration, each thread can overwrite the input of another thread
- Barrier synchronization to ensure all input have been properly generated
- All threads secure input operand that can be overwritten by another thread
- Barrier synchronization to ensure that all threads have secured their input
- All threads perform addition and write output

The Better Scan Kernel

```
__global__ void work_inefficient_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];}
 // the code below performs iterative scan on XY
  for (unsigned int stride = 1; stride <= threadIdx.x; stride *= 2) {
    __syncthreads();
    float in1 = XY[threadIdx.x - stride];
    __syncthreads();
    XY[threadIdx.x] += in1;
  }
```

Work Efficiency Considerations

- This scan executes log(n) parallel iterations
  - An iteration performs (n-1), (n-2), (n-4),..,n/2 adds
  - Total adds: n * log(n) - (n-1) → O(n*log(n)) work
- This scan algorithm is not work efficient
  - Sequential scan algorithm does n adds
  - A factor of log(n) can hurt: 10x for 1024 elements!
- A parallel algorithm can be slower than a sequential one when execution resources are saturated from low work efficiency

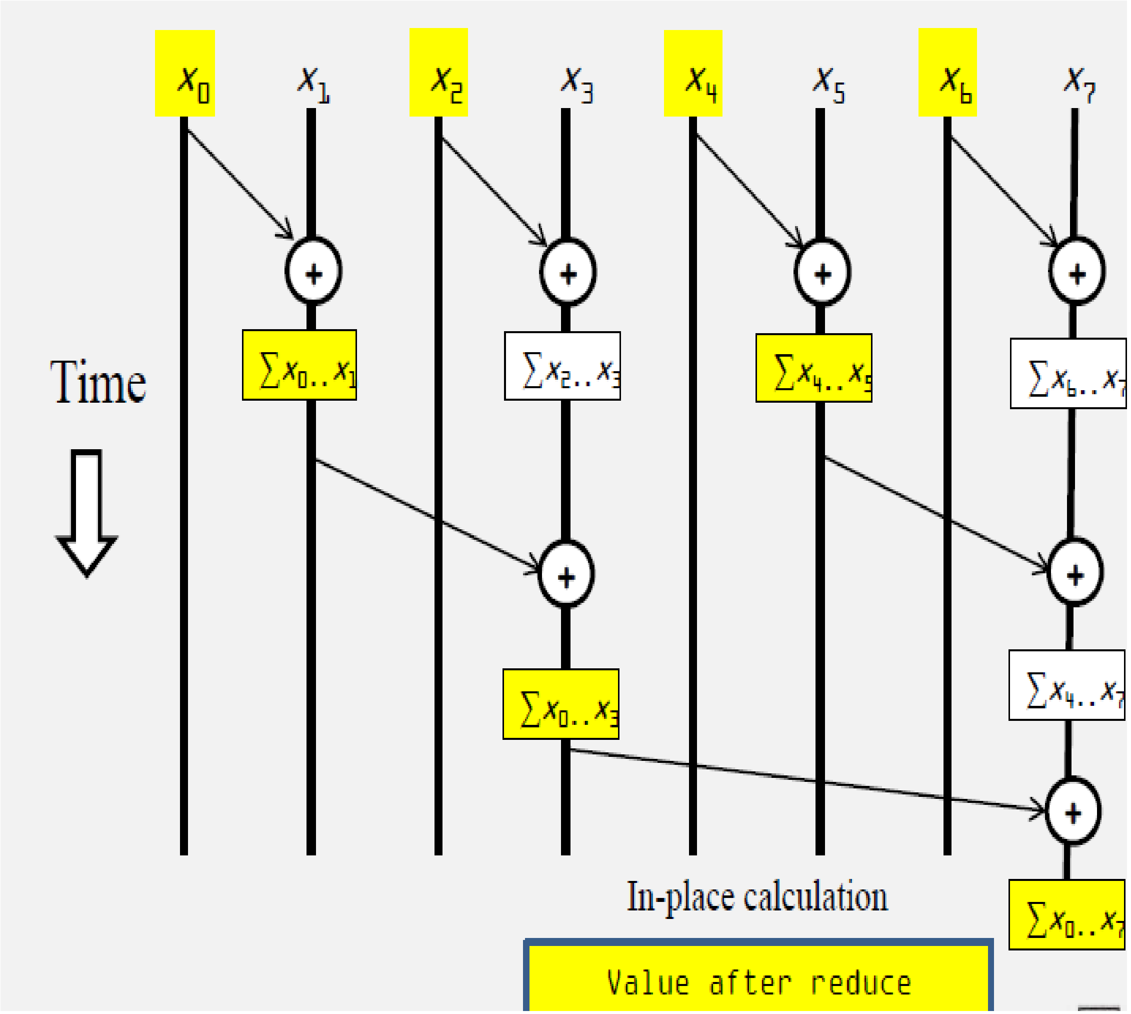
How To Improve Efficiency

- Two-phase balanced tree traversal
- Aggressive reuse of computation results
- Reducing control divergence with more complex thread index to data index mapping

Balanced Tree for Scan

- Form a balanced binary tree on the input data and sweep it to and from the root
- Here the tree is not an actual data structure, but a concept to determine what each thread does at each step
- 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 output from the partial sums

Parallel Scan – Reduction Phase

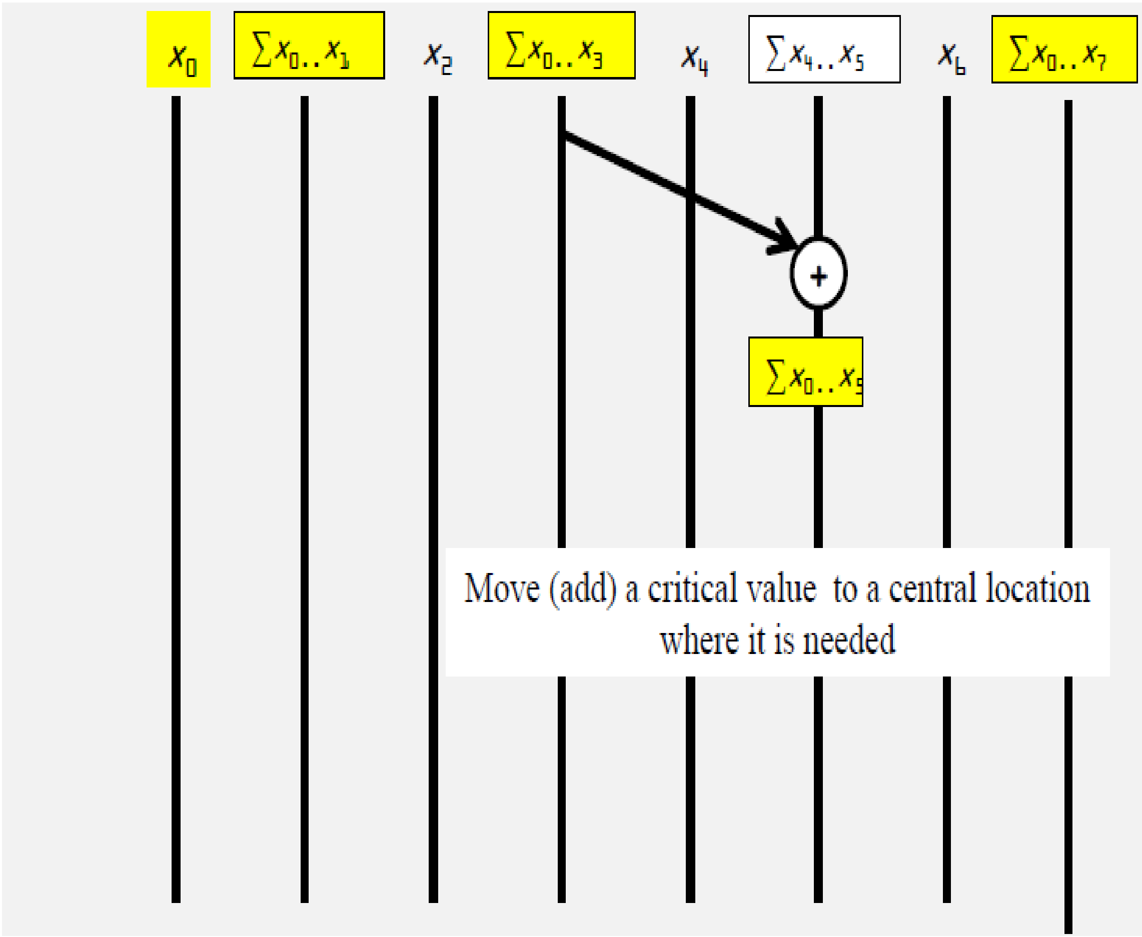


Reduction Phase Kernel Code

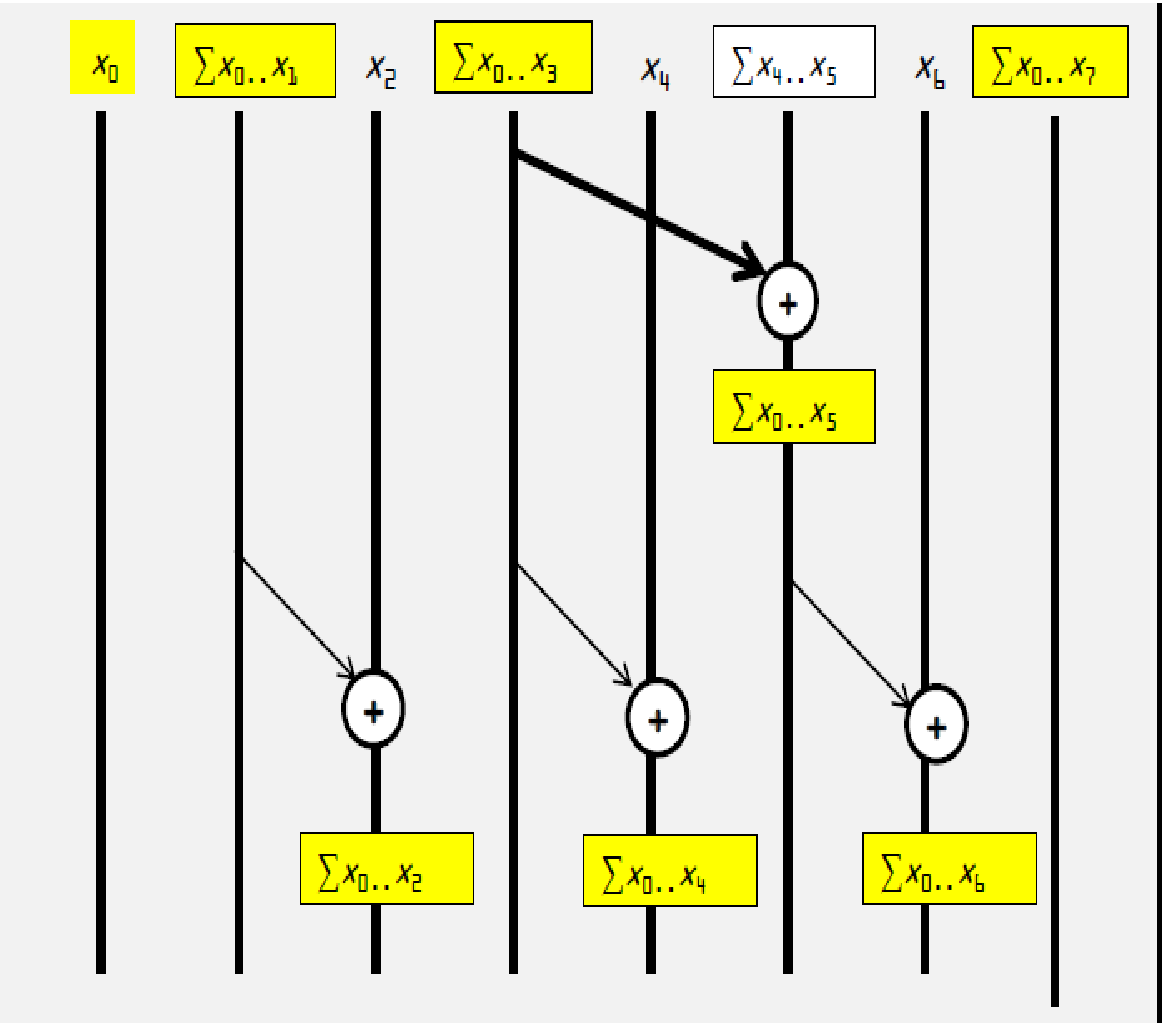
```
// XY[2*BLOCK_SIZE] is in shared memory

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

Parallel Scan –
Post Reduction Reverse Phase



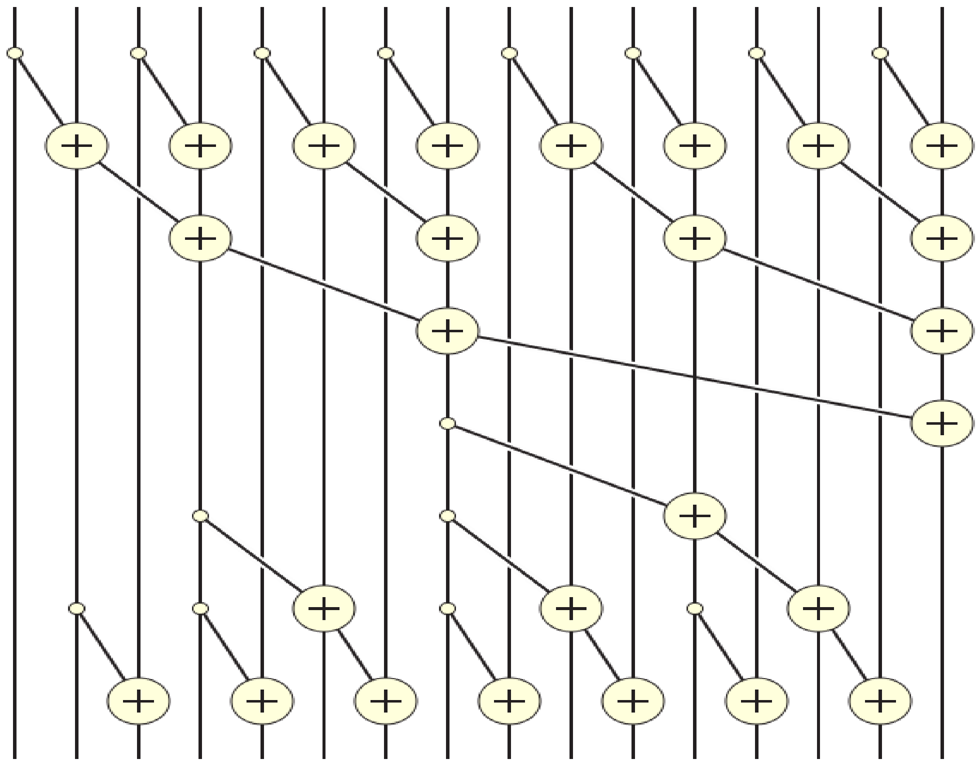
Parallel Scan –
Post Reduction Reverse Phase



Post Reduction Reverse Phase Kernel Code

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

Putting it all together



Efficiency Analysis

- The work efficient kernel executes log(n) parallel iterations in the reduction step
  - The iterations do n/2, n/4,..1 adds
  - Total adds: (n-1) → O(n) work
- It executes log(n)-1 parallel iterations in the post reduction reverse step
  - The iterations do 2-1, 4-1, …. n/2-1 adds
  - Total adds: (n-2) – (log(n)-1) → O(n) work
- Both phases perform up to no more than 2*(n-1) adds
- The total number of adds is no more than twice of that done in the efficient sequential algorithm

Tradeoffs

- The work efficient scan kernel is normally more desirable
  - Better Energy efficiency
  - Less execution resource requirement
- However, the work inefficient kernel could be better for absolute performance due to its single-step nature if there is sufficient execution resource

Summary

- Prefix scan is a common data-parallel primitive.
- A naïve parallel implementation is work inefficient.
- A work-efficient implementation takes a two phase balanced tree approach.
- There are tradeoffs between the two implementations.