source: NVIDIA GPU Teaching Kit

IF3230
Sistem Paralel dan Terdistribusi
CUDA

Achmad Imam Kistijantoro (imam@staff.stei.itb.ac.id)
Robithoh Annur (robithoh@stafff.stei.itb.ac.id)
Andreas Bara Timur (bara@staff.stei.itb.ac.id)

February 2023   3/1/2023

Objective

– To master parallel scan (prefix sum) algorithms
  – Frequently used for parallel work assignment and resource allocation
  – A key primitive in many parallel algorithms to convert serial computation into parallel computation
  – A foundational parallel computation pattern
  – Work efficiency in parallel code/algorithms

– Reading –Mark Harris, Parallel Prefix Sum with CUDA
  – http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Inclusive Scan (Prefix-Sum) Definition

Definition: *The* scan *operation takes a binary associative operator* $\oplus$ (pronounced as circle plus), *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 scan operation on the array would return

$$[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3],$$
$$[3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22 \ 25].$$

An Inclusive Scan Application Example

– Assume that we have a 100-inch sandwich 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 sandwich 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:

  – [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   • Polynomial evaluation
  • Quicksort   • Solving recurrences
  • String comparison   • Tree operations
  • Lexical analysis   • Histograms, ….
  • Stream compaction

Other Applications

– Assigning camping spots
– Assigning Farmer's Market spaces
– Allocating memory to parallel threads
– Allocating memory buffer space for communication channels
– …

An Inclusive Sequential Addition 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

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

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)!
Only slightly more expensive than sequential reduction.

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."

Objective

– To learn to write and analyze a high-performance scan kernel

- – Interleaved reduction trees
- – Thread index to data mapping
- – Barrier Synchronization
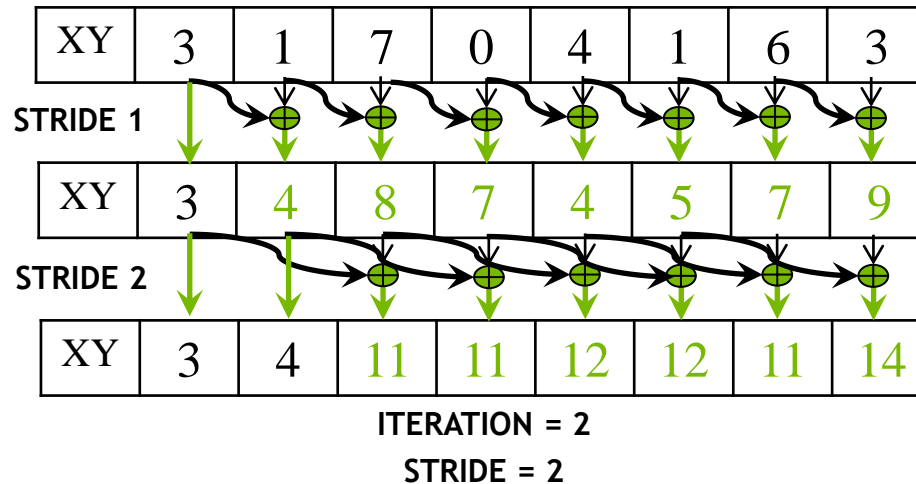- – Work efficiency analysis

A Better Parallel Scan 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

ITERATION = 1
STRIDE = 1
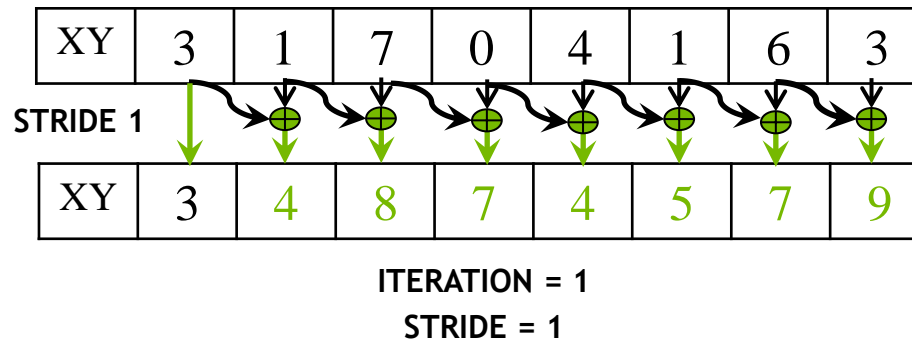
- Active threads *stride* to n-1 (n-stride threads)
- Thread *j* adds elements *j* and *j-stride* from shared memory and writes result into element j in shared memory
- Requires barrier synchronization, once before read and once before write
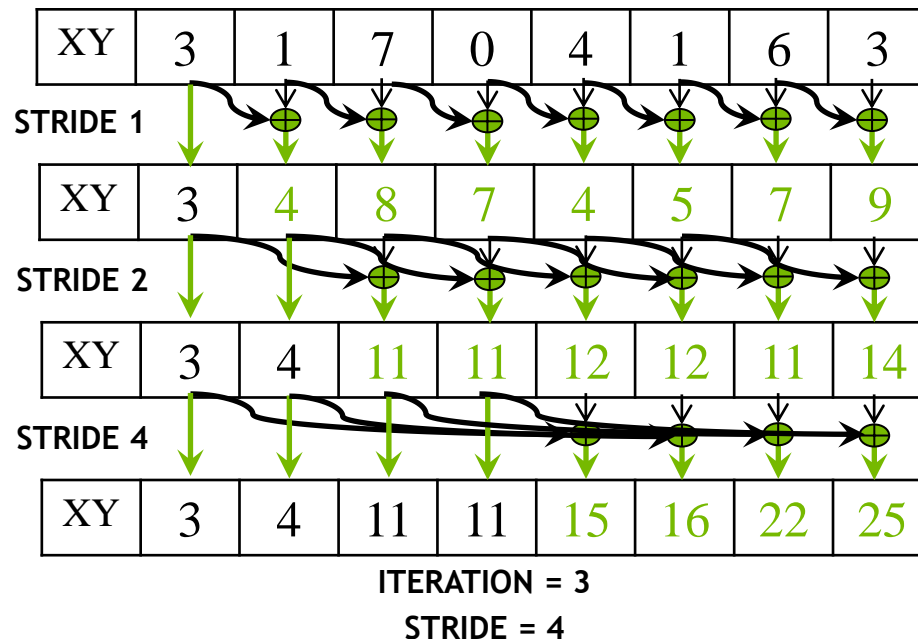
A Better Parallel Scan Algorithm

1. Read input from device to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration.

ITERATION = 2
STRIDE = 2

# A Better Parallel Scan Algorithm

1. Read input from device to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration
3. Write output from shared memory to device memory

Handling Dependencies

– During every iteration, each thread can overwrite the input of another thread

- Barrier synchronization to ensure all inputs have been properly generated
- All threads secure input operand that can be overwritten by another thread
- Barrier synchronization is required to ensure that all threads have secured their inputs
- All threads perform addition and write output

XY	3	1	7	0	4	1	6	3

STRIDE 1

XY	3	4	8	7	4	5	7	9
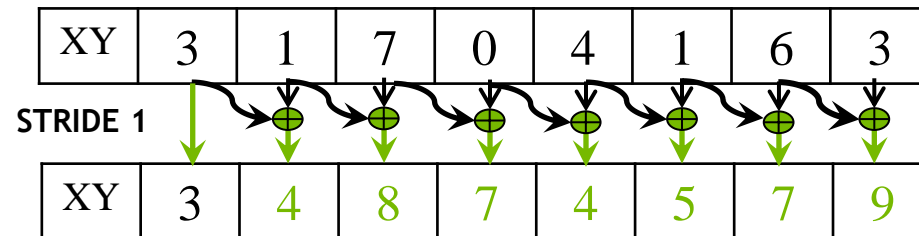
ITERATION = 1
STRIDE = 1

A Work-Inefficient 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;
        }
        __ syncthreads();
        If (i < InputSize) {Y[i] = XY[threadIdx.x];}
}
```

Work Efficiency Considerations

– This Scan executes log(n) parallel iterations
  – The iterations do (n-1), (n-2), (n-4),..(n- n/2) adds each
  – Total adds: n * log(n) - (n-1) $\rightarrow$ 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

Objective

– To learn to write a work-efficient scan kernel
  – Two-phased balanced tree traversal
  – Aggressive re-use of intermediate results
  – Reducing control divergence with more complex thread index to data index mapping
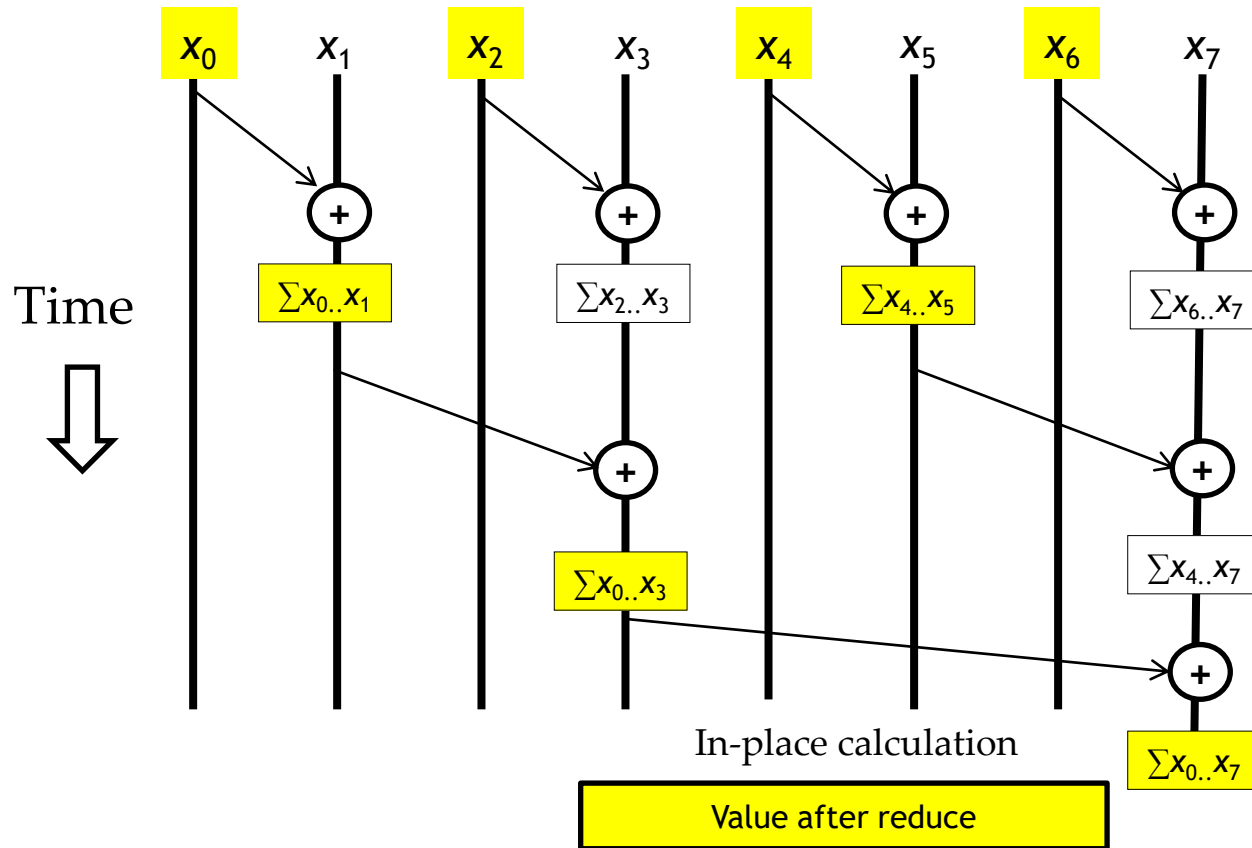
Improving Efficiency

- *Balanced Trees*
  - Form 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 concept to determine what each thread does at each step
- For scan:
  - Traverse down from leaves to the root building partial sums at internal nodes in the tree
    - The root holds the sum of all leaves
  - Traverse back up the tree building the output from the partial sums

Parallel Scan - Reduction Phase

Time

x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7

$\sum x_{0..}x_1$ $\sum x_{2..}x_3$ $\sum x_{4..}x_5$ $\sum x_{6..}x_7$

$\sum x_{0..}x_3$ $\sum x_{4..}x_7$

In-place calculation

$\sum x_{0..}x_7$

Value after reduce

Reduction Phase Kernel Code

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

for (unsigned 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();
}
```
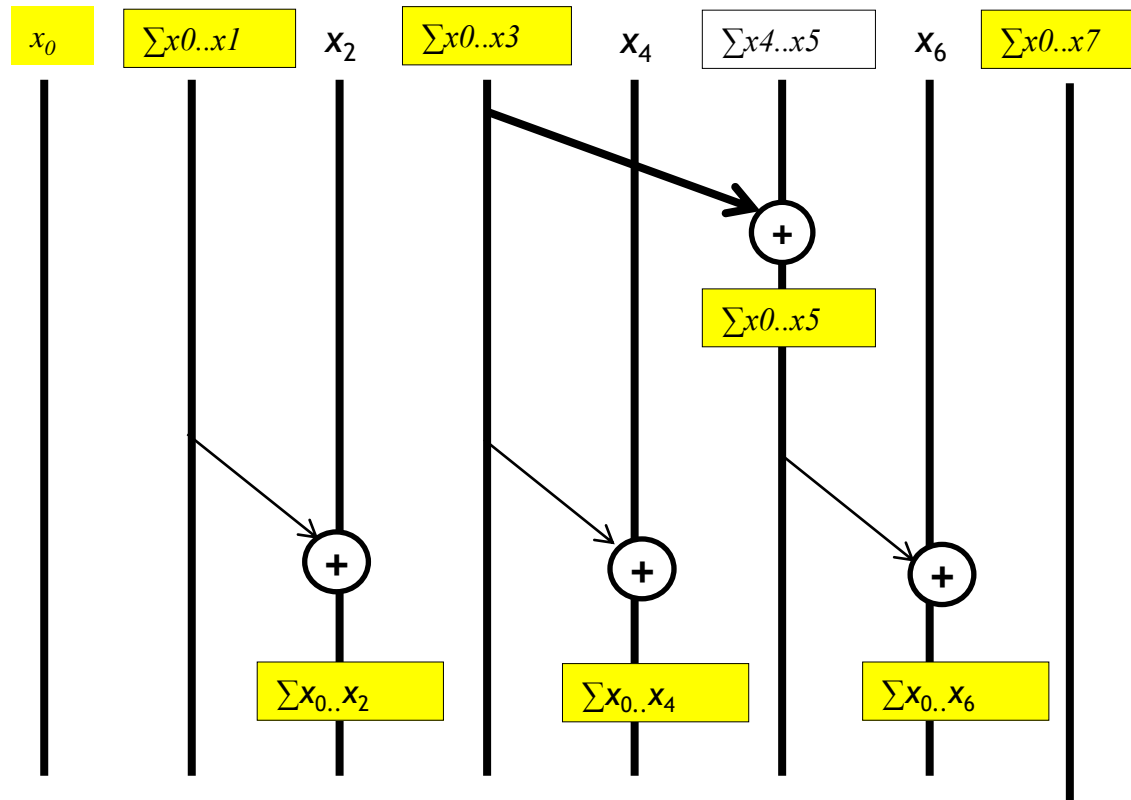
threadIdx.x+1　　= 1, 2, 3, 4….

stride = 1,
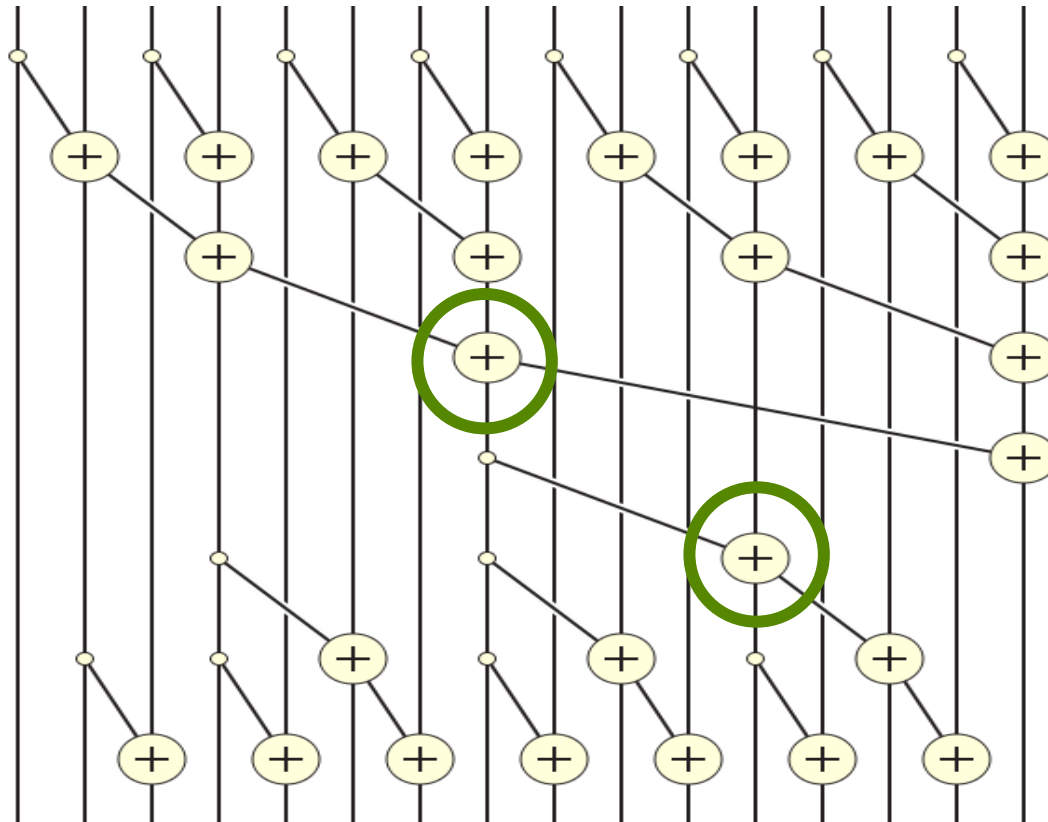
index = 1, 3, 5, 7, …

Parallel Scan - Post Reduction Reverse Phase
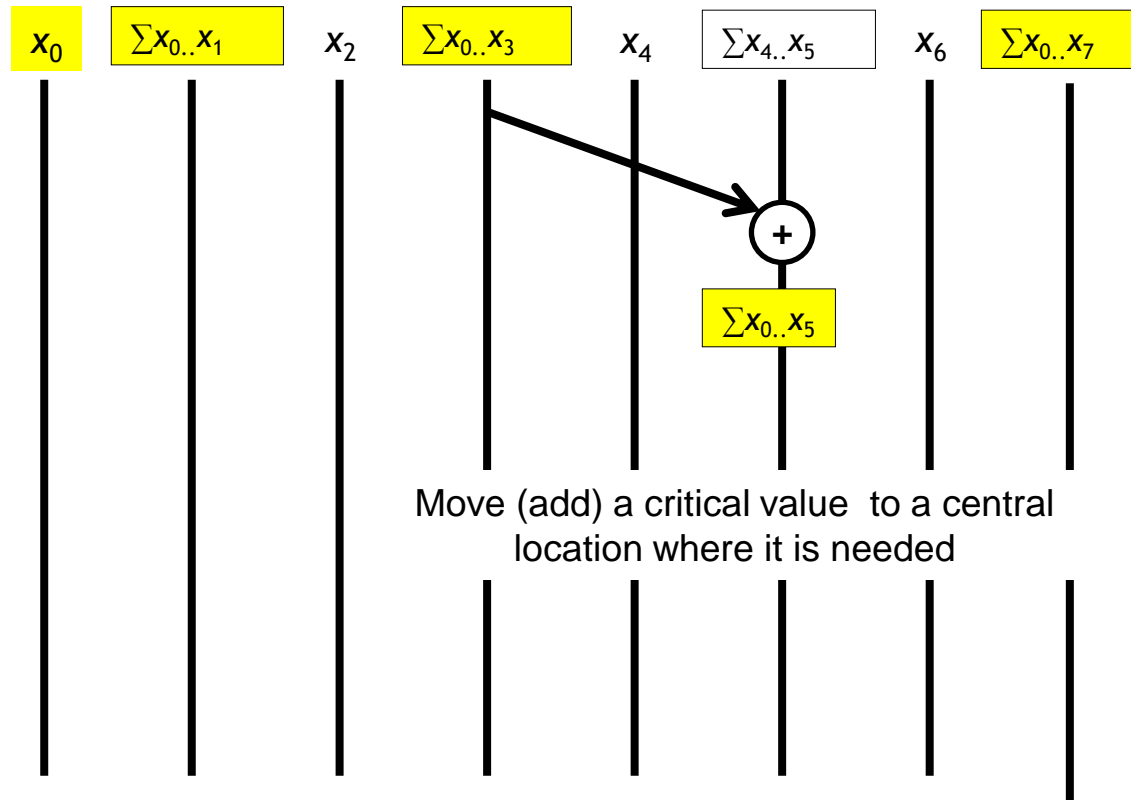
Parallel Scan - Post Reduction Reverse Phase

Putting it Together

Post Reduction Reverse Phase Kernel Code

```
for (unsigned 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];
```

First iteration for 16-element section
threadIdx.x = 0
stride = BLOCK_SIZE/2 = 8/2 = 4
index = 8-1 = 7

Objective

– To learn more about parallel scan
  – Analysis of the work efficient kernel
  – Exclusive scan
  – Handling very large input vectors

Work Analysis of the Work Efficient Kernel

– 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 2x(n-1) adds

– 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

Some 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-phase nature (forward phase only)
  - There is sufficient execution resource

Handling Large Input Vectors

– Build on the work efficient scan kernel

– Have each section of 2*blockDim.x elements assigned to a block

    – Perform parallel scan on each section

– Have each block write the sum of its section into a Sum[] array indexed by blockIdx.x

– Run the scan kernel on the Sum[] array

– Add the scanned Sum[] array values to all the elements of corresponding sections

– Adaptation of work inefficient kernel is similar.

Overall Flow of Complete Scan
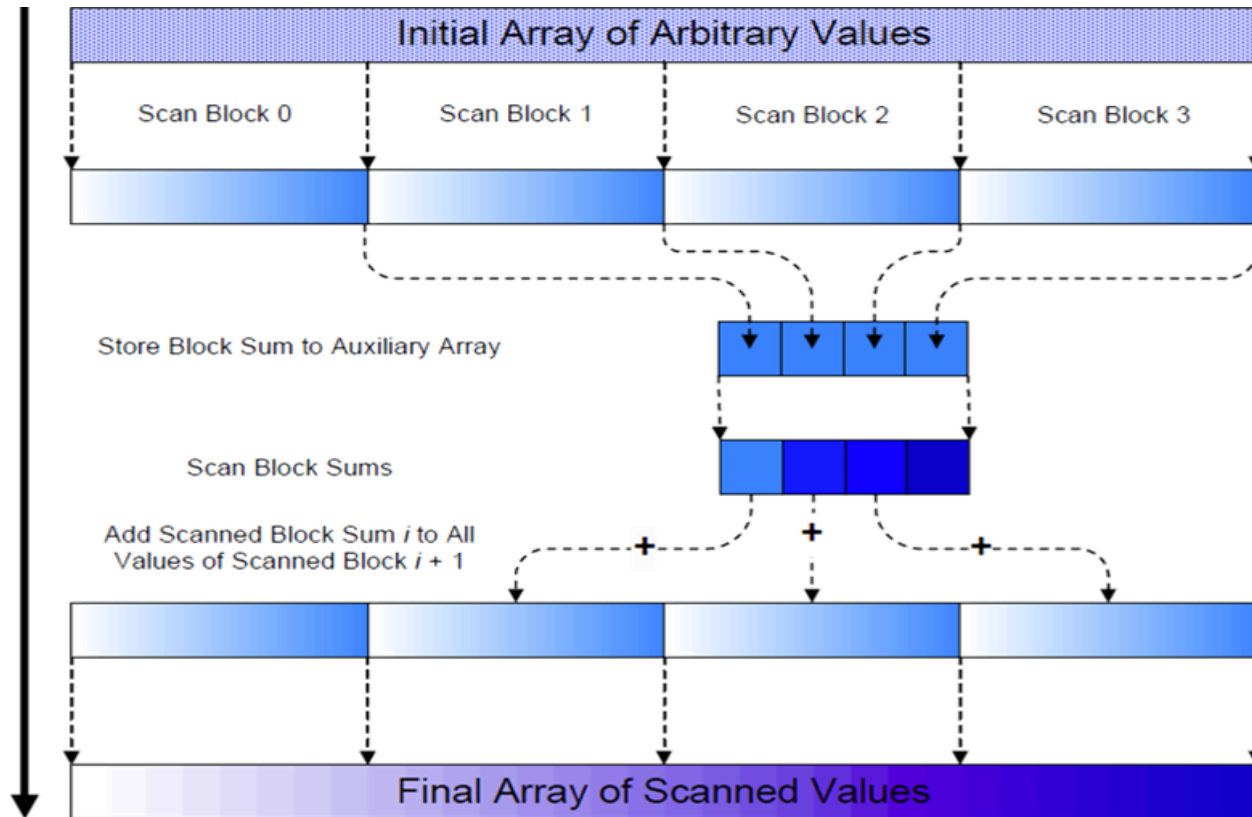
Exclusive Scan Definition

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 the array   [3 1 7   0   4   1   6   3],
would return [0 3 4 11   11 15 16 22].

Why Use 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, work inefficient scan kernel

– Block 0:
  – Thread 0 loads 0 into XY[0]
  – Other threads load 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 work efficient scan kernel but ensure that each thread loads two elements
  – Only one zero should be loaded
  – All elements should be shifted to the right by only one position

Read the Harris article (Parallel Prefix Sum with CUDA) for a more intellectually interesting approach to exclusive scan kernel implementation.