Parallel Prefix Scan

Nikos Hardavellas

Some slides/material from:
UIUC course by Wen-Mei Hwu and David Kirk
IISC-SERC course by Mike Giles, Oxford University Mathematical Institute

1

Parallel Prefix Scan: Why Do We Care?

- Parallel Prefix Sum (Scan) algorithms
  - One of the most frequently-used parallel patterns
  - frequently used for parallel work assignment and resource allocation
  - A key primitive in many parallel algorithms to covert serial computation into parallel computation
  - Based on reduction tree and reverse reduction tree
- Reading – Mark Harris, Parallel Prefix Sum with CUDA
  - http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf

2

(Inclusive) Prefix-Sum (Scan) Definition

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].

3

A Inclusive Scan Application Example

World's longest sausage (38.99 miles)

World's longest hot dog (60m)

- Assume a 100-inch sausage 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 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 Scan
  - [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)

4

Other Applications

- Allocating memory to parallel threads
  - Think of supercomputers with 1-million-thread workloads
  - Not science fiction:
    - Sunway TaihuLight @ NSC, China, 2016: 10,649,600 SW26010 cores
    - Titan @ ORNL, 2012: 299,008 x86 cores + 50,233,344 CUDA cores
    - Summit @ ORNL, 2018: 221,184 Power9 cores + 141,557,760 CUDA cores
- Allocating memory buffer to communication channels
- …

5

An Inclusive Sequential Prefix-Sum

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

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

6

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)!

7

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

$$y_0 = x_0$$

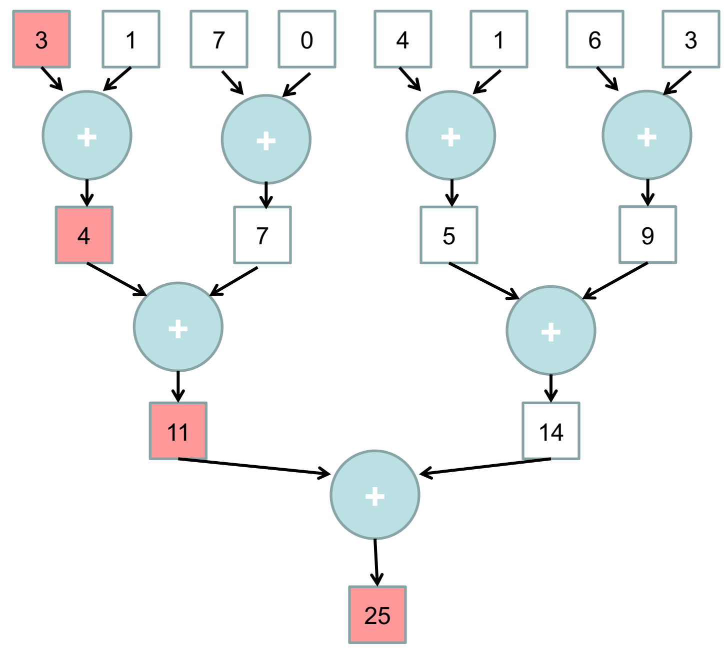
$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

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

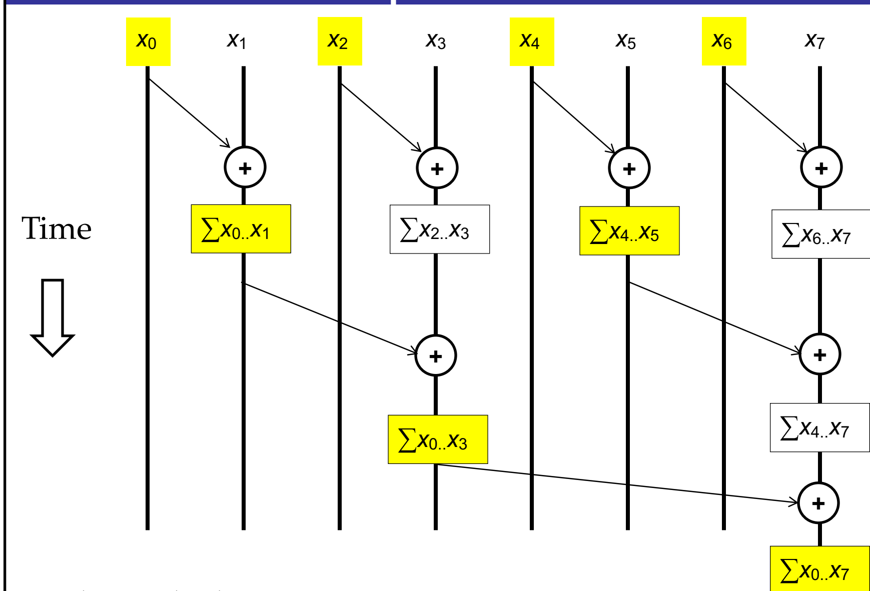
8

Let’s Look at the Reduction Tree Again



9

Reduction Scan Step

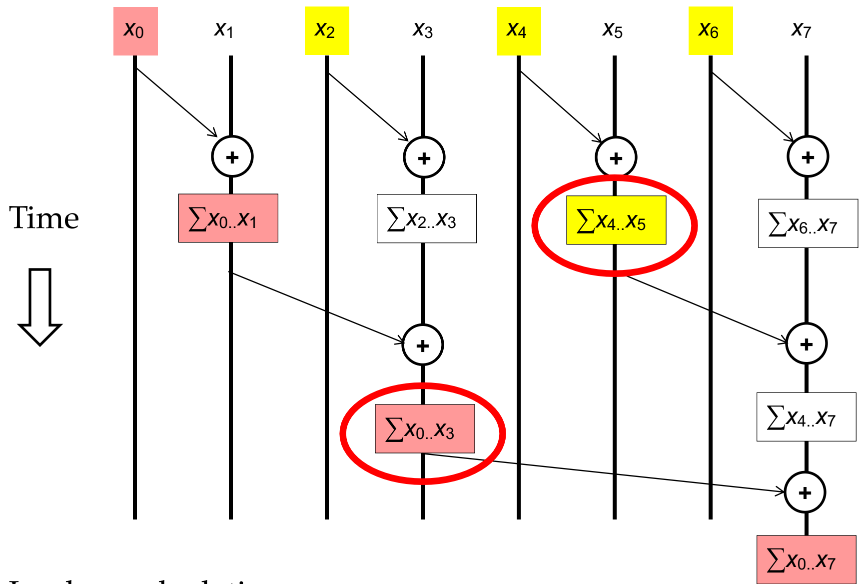


In place calculation:

Value after reduce

10

Reduction Scan Step



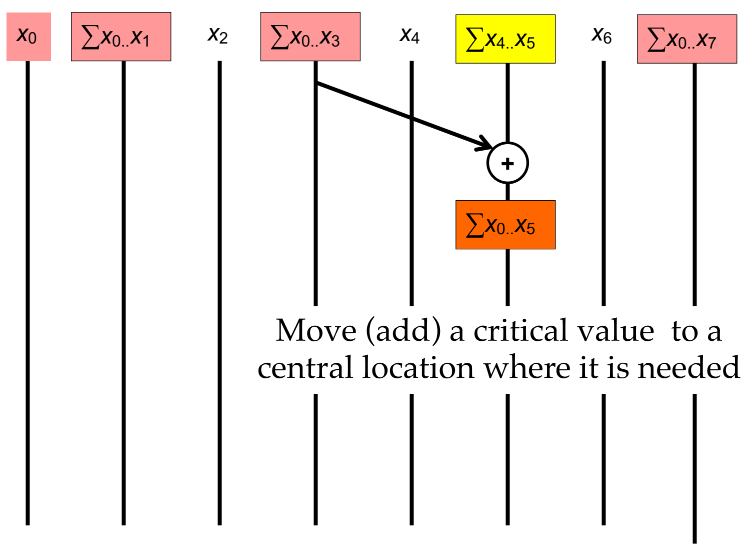
In place calculation:

Value after reduce

Value after reduce matches desired final value

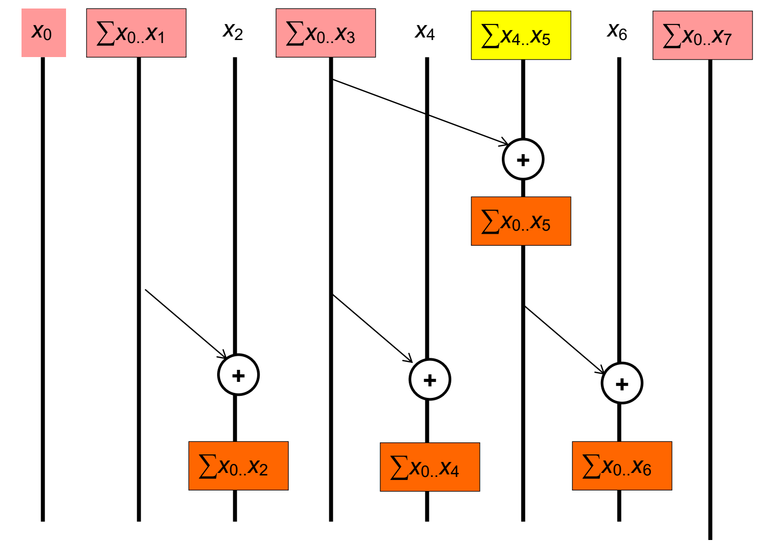
11

Inclusive Post Scan Step



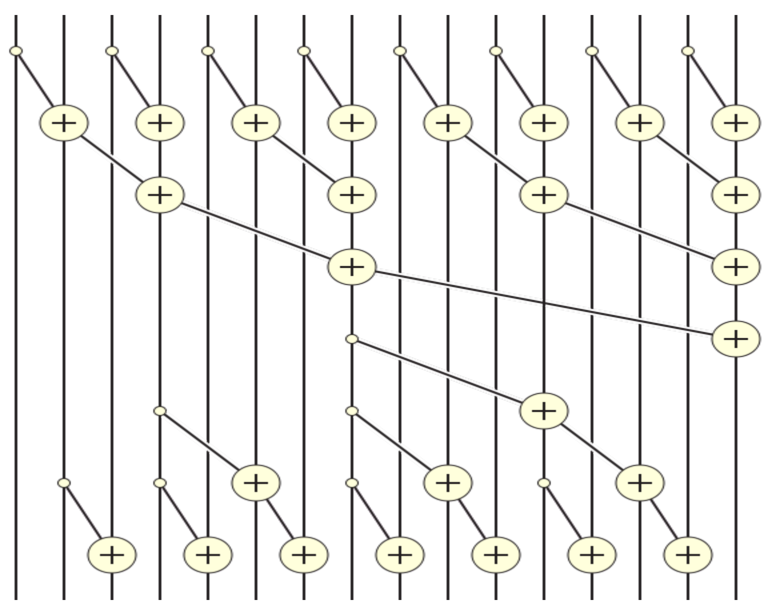
12

Inclusive Post Scan Step



13

Putting it Together



14

Reduction Step Kernel Code

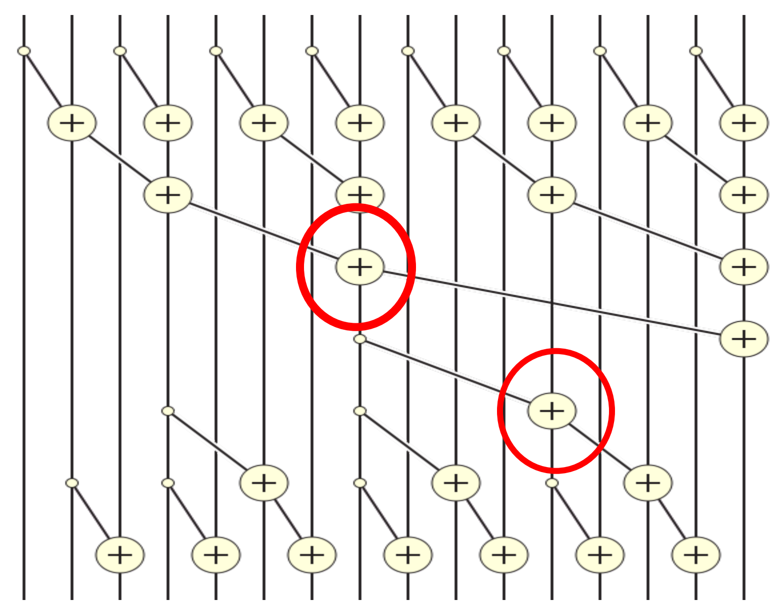
```
// scan_array[BLOCK_SIZE] is in shared memory

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

    __syncthreads();
}
```

15

Putting it Together



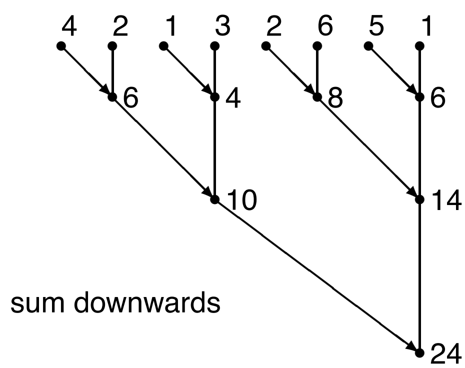
16

Post Scan Step

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

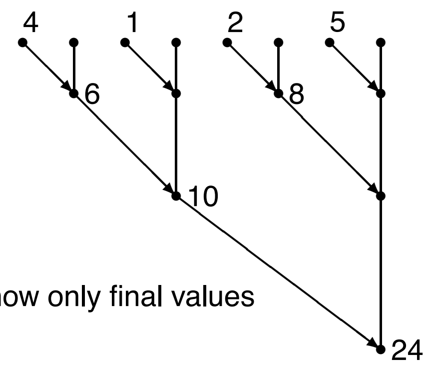
17

Parallel Scan: Another View



18

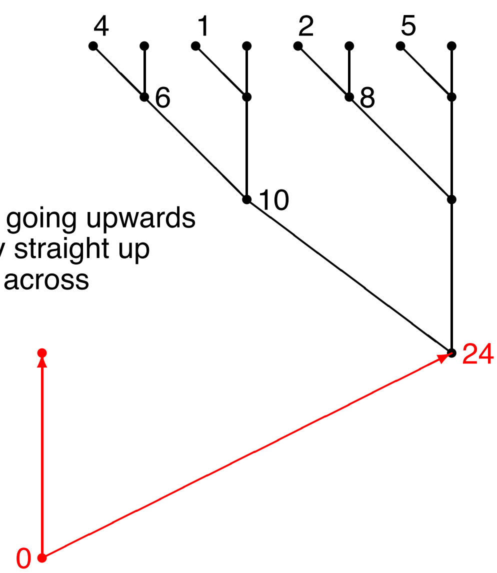
Parallel Scan: Another View



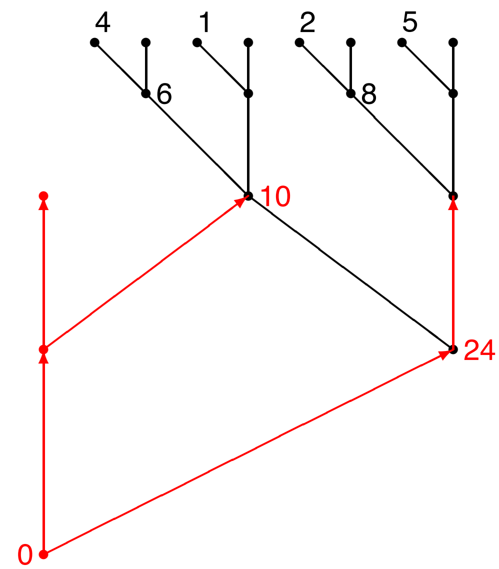
show only final values

Parallel Scan: Another View

start going upwards
carry straight up
sum across

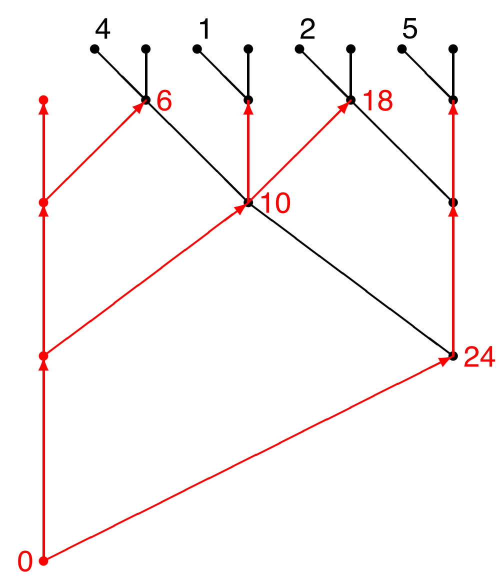


Parallel Scan: Another View



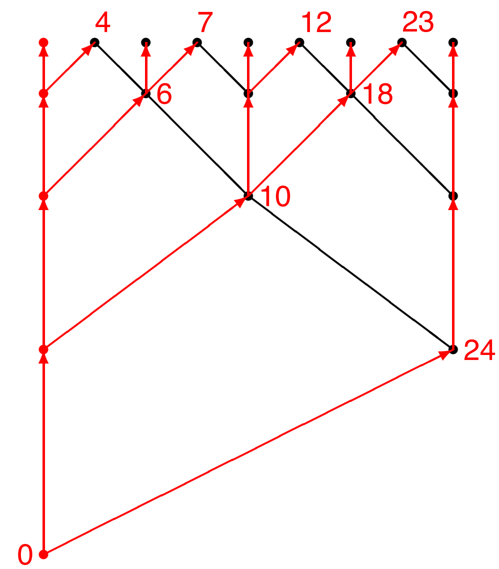
21

Parallel Scan: Another View



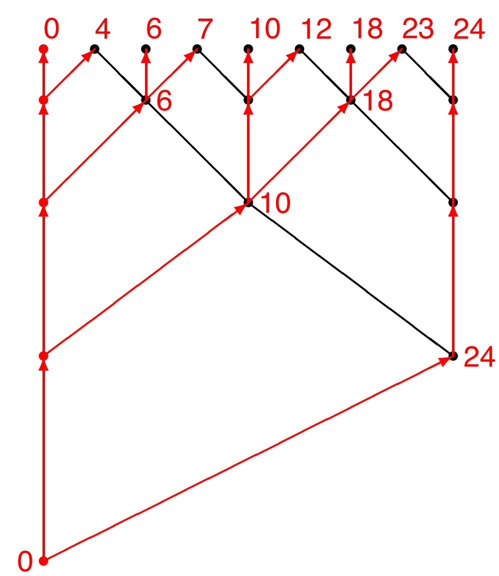
22

Parallel Scan: Another View



23

Parallel Scan: Another View



24

Single-Kernel Parallel Scan: Potential for Deadlock

- However, this needs the sum of all preceding blocks to add to the local scan values
  - replace initial value 0 at the start of the upward sweep
- Problem: blocks are not necessarily processed in order, so could end up in deadlock waiting for results from a block which doesn't get a chance to start.
- Could launch multiple kernels on multiple streams, enforce dependency order using events
  - Dependency encoding might be tricky to get right
  - Switching often to CPU degrades performance
  - Any other solutions?
- **Solution: use atomic increments**

25

Enforcing Block Processing Order

- Declare a global device variable

```
__device__ int my_block_count = 0;
```

- At the beginning of the kernel code use

```
__shared__ unsigned int my_blockId;
if (threadIdx.x==0) {
      my_blockId = atomicInc( &my_block_count,
                              UINT_MAX);
}
__syncthreads();
```

- This returns the old value of my_block_count and increments it, all in one operation. The UINT_MAX ensures atomicInt always increments the counter.
- This gives us a way of launching blocks in strict order.

26

Block-Ordered Global Scan

- Use a new global counter, `block_sums_completed`, to indicate which blocks have completed their downward sums
- For a single-kernel global scan, the kernel does the following:
  - get in-order block ID; let's call it B_i
  - do downward pass
  - B_i waits until `block_sums_completed` $= B_i - 1$. When it does, it shows that the preceding block has computed the sum of the blocks so far on a global variable `sum`
  - get `sum`, increment it with the local partial sum
  - increment `block_sums_completed` to signal to block B_{i+1} that you are done
  - do upwards pass and store the results

27

(Exclusive) Prefix-Sum (Scan) Definition

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

$$[a0, a1, \ldots, an\text{-}1],$$

and returns the array

$$[0, a0, (a0 \oplus a1), \ldots, (a0 \oplus a1 \oplus \ldots \oplus an\text{-}2)].$$

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

28

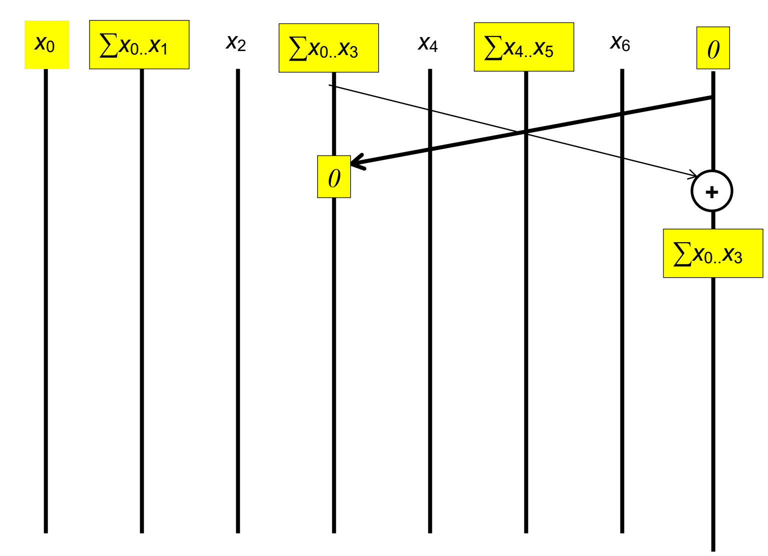
Why Exclusive Scan

- 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]

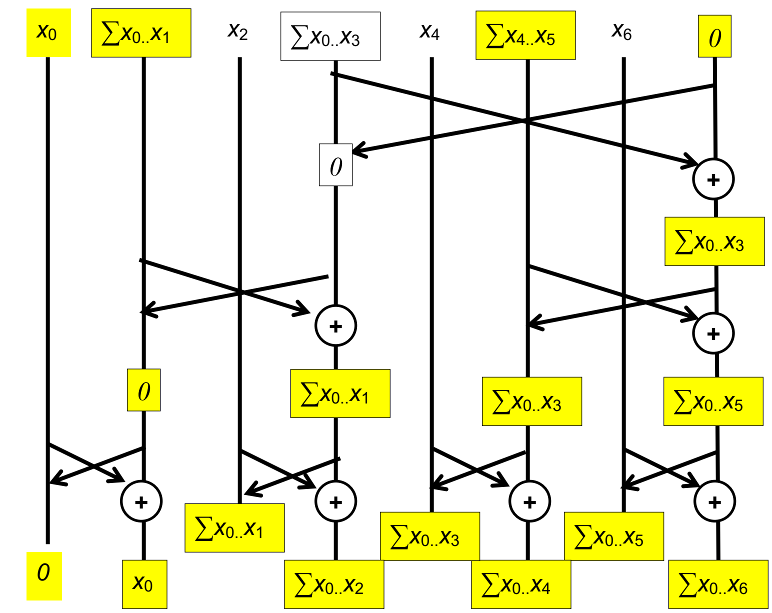
29

Exclusive Post Scan Step



30

Exclusive Post Scan Step



31

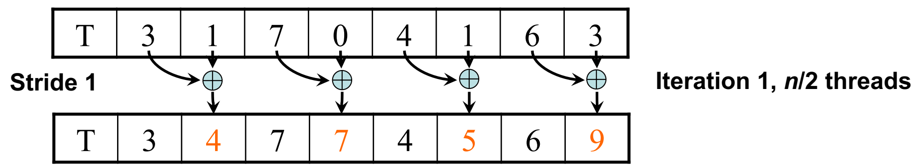
Exclusive Scan Example – Reduction Step

T	3	1	7	0	4	1	6	3

Assume array is already in shared memory

32

Reduction Step (cont.)

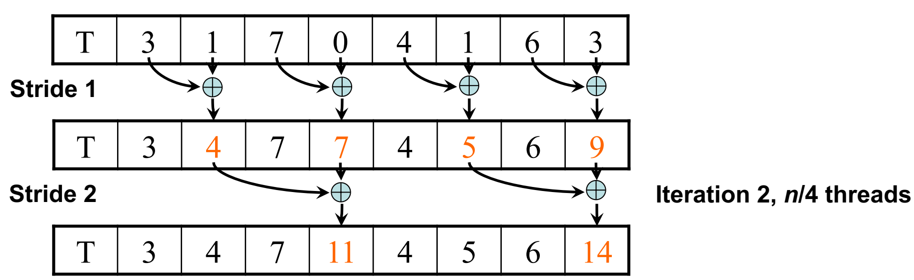


Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* elements away to its own value

33

Reduction Step (cont.)

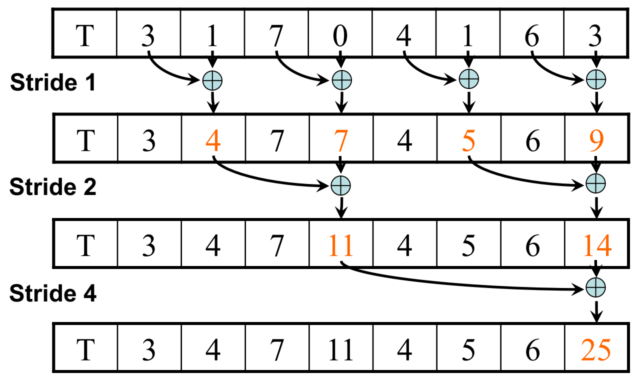


Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* elements away to its own value

34

Reduction Step (cont.)



Iteration log(*n*), 1 thread

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering

35

Zero the Last Element

T	3	4	7	11	4	5	6	0

We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

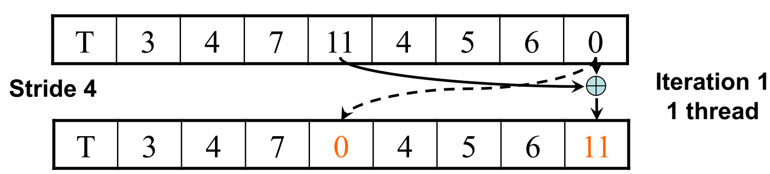
36

Post Scan Step from Partial Sums

T	3	4	7	11	4	5	6	0

37

Post Scan Step from Partial Sums (cont.)

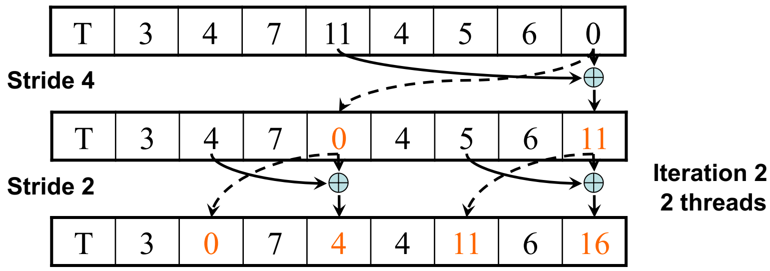


Each $\oplus$ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

38

Post Scan From Partial Sums (cont.)

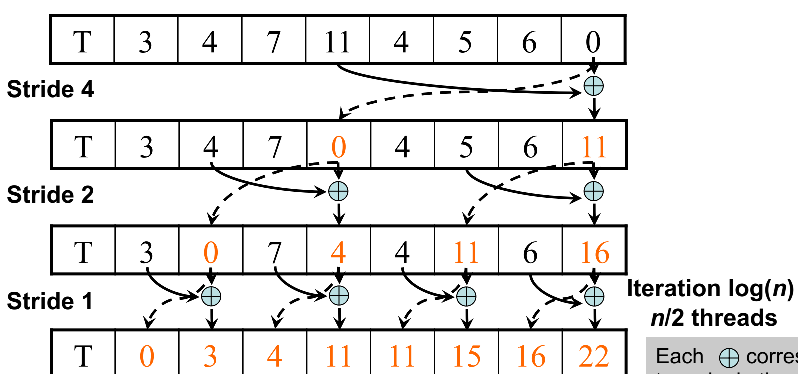


Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

39

Post Scan Step From Partial Sums (cont.)



Each ⊕ corresponds to a single thread.

Done! We now have a completed scan that we can write out to device memory.

Total steps: 2 * log(*n*).
Total work: 2 * (*n*-1) adds = *O*(*n*) **Work Efficient!**

40

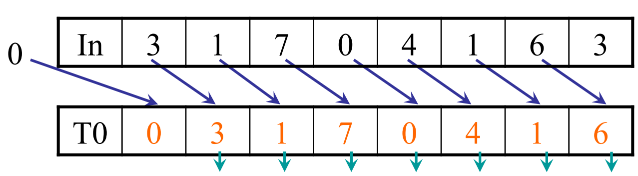
Work Analysis

- The Parallel Inclusive 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, 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

Compare to a work-inefficient parallel scan

41

A Plausible Parallel Scan Algorithm

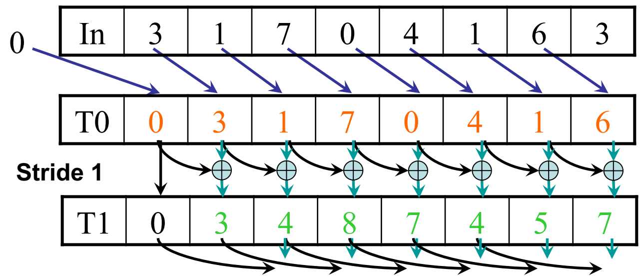


1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

Each thread reads one value from the input array in device memory into shared memory array T0. Thread 0 writes 0 into shared memory array.

42

A Plausible Parallel Scan Algorithm



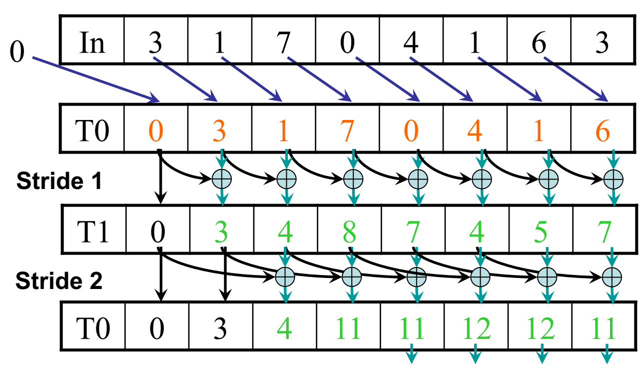
1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements that are *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #1
Stride = 1

• Active threads: *stride* to *n*-1 (*n-stride* threads)
• Thread *j* adds elements *j* and *j-stride* from T0 and writes result into shared memory buffer T1 (ping-pong)

43

A Plausible Parallel Scan Algorithm

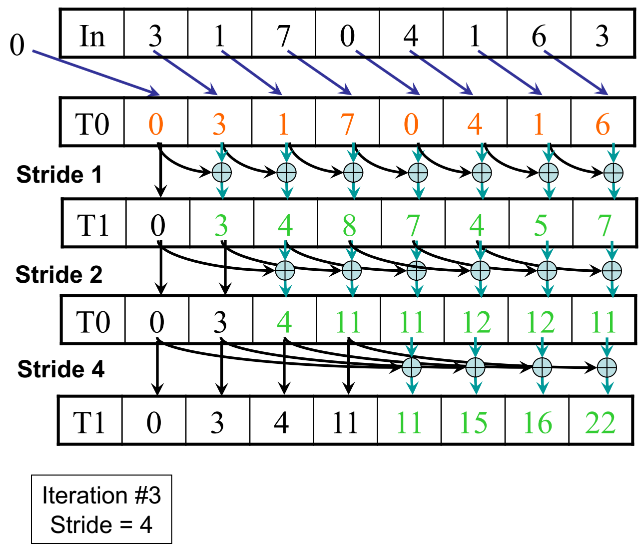


1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements that are *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #2
Stride = 2

44

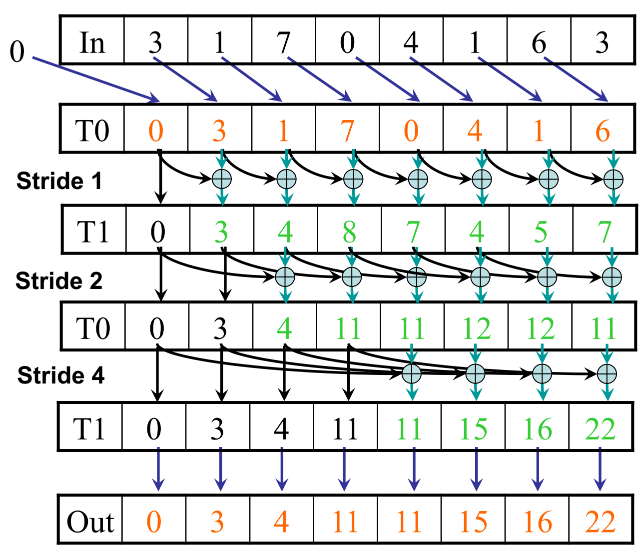
A Plausible Parallel Scan Algorithm



1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

45

A Plausible Parallel Scan Algorithm



1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)
3. Write output to device memory.

46

Sample Code for Plausible Parallel Scan Algorithm

```
__global__ void scan(float *d_sum,float *d_data)
{
    extern __shared__ float temp[];
    int tid = threadIdx.x;
    temp[tid] = d_data[tid+blockIdx.x*blockDim.x];
    for (int d=1; d<blockDim.x; d<<=1) {
        __syncthreads();
        float temp2 = (tid >= d) ? temp[tid-d] : 0;
        __syncthreads();
        temp[tid] += temp2;
    }
    ...
}
```

47

Work Efficiency Considerations

- The plausible parallel Scan executes log(n) parallel iterations
  - The steps do (n-1), (n-2), (n-4),..(n- n/2) adds
  - Total adds: n * $\log_2(n)$ + (n-1) → O(n*$\log_2(n)$) work
- This scan algorithm is not very work efficient
  - Sequential scan algorithm does *n* adds
  - A factor of $\log_2(n)$ hurts: 20x for 10^6 elements!
- A parallel algorithm can be slow when execution resources are saturated due to low work efficiency

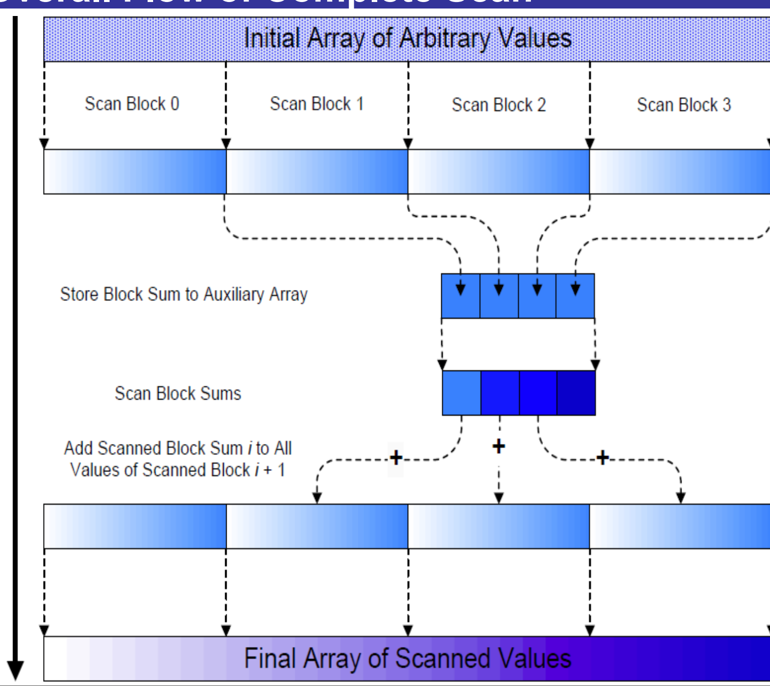
48

Working on Arbitrary Length Input

- Build on the scan kernel that handles up to 2*blockDim elements
- Have each section of 2*blockDim elements assigned to each block
- Have each block write the sum of its section into a Sum array indexed by blockIdx.x
- 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

49

Overall Flow of Complete Scan



50