CSE 599 I
Accelerated Computing - Programming GPUS

Parallel Patterns: Prefix Sum (Scan)

GPU Teaching Kit

Accelerated Computing

Module 9.1 – Parallel Computation Patterns (Reduction)

Parallel Reduction

Objective

- To learn the parallel reduction pattern
  - An important class of parallel computation
  - Work efficiency analysis
  - Resource efficiency analysis

Partition and Summarize

- A commonly used strategy for processing large input data sets
  - There is no required order of processing elements in a data set (associative and commutative)
  - Partition the data set into smaller chunks
  - Have each thread to process a chunk
  - Use a reduction tree to summarize the results from each chunk into the final answer
- Google and Hadoop MapReduce frameworks support this strategy
- We will focus on the reduction tree step for now

Reduction enables other techniques

- Reduction is also needed to clean up after some commonly used parallelizing transformations
- Privatization
  - Multiple threads write into an output location
  - Replicate the output location so that each thread has a private output location
  - Use a reduction tree to combine the values of private locations into the original output location

What is a reduction computation?

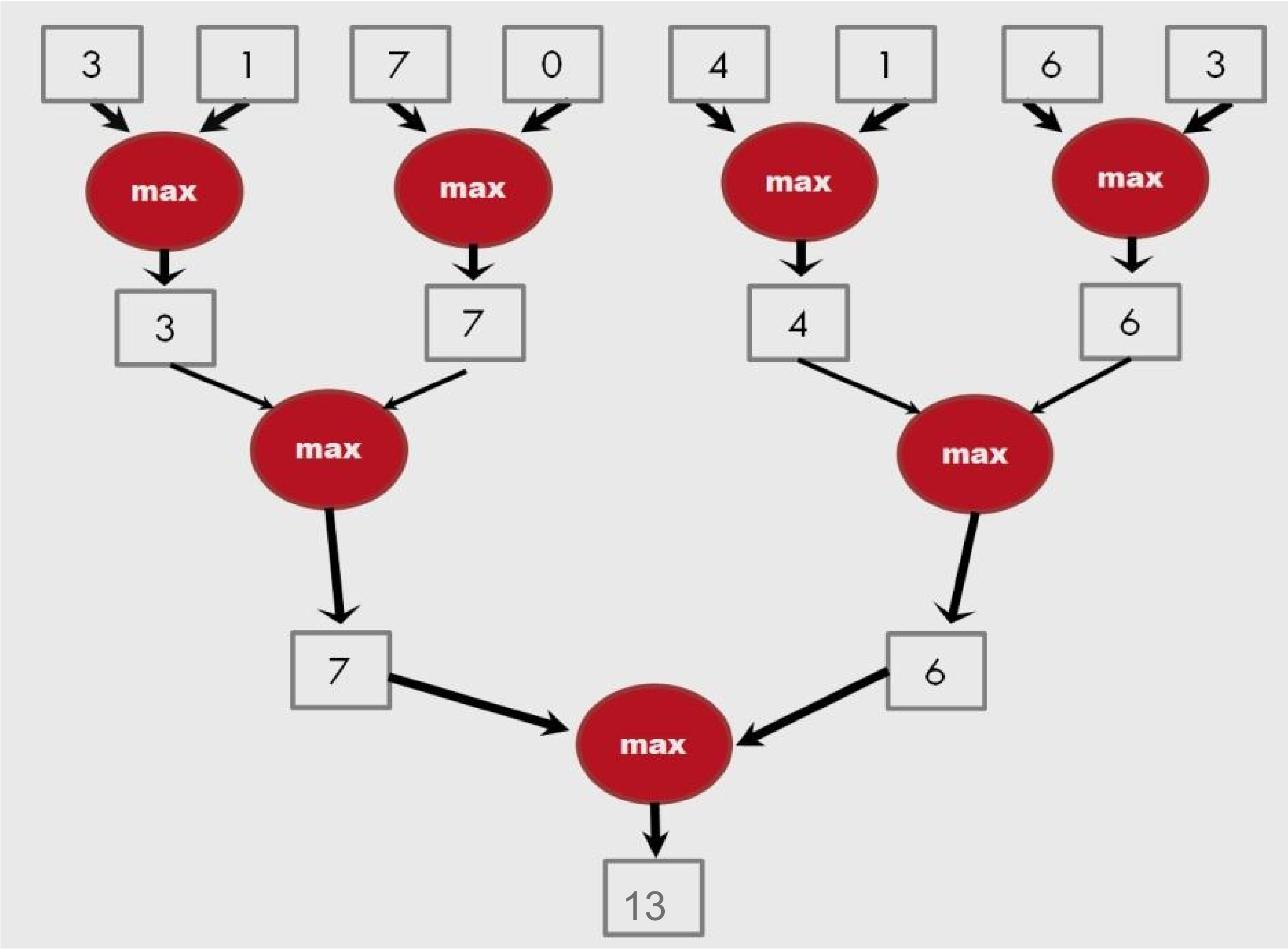
- Summarize a set of input values into one value using a “reduction operation”
  - Max
  - Min
  - Sum
  - Product
- Often used with a user defined reduction operation function as long as the operation
  - Is associative and commutative
  - Has a well-defined identity value (e.g., 0 for sum)
  - For example, the user may supply a custom “max” function for 3D coordinate data sets where the magnitude for the each coordinate data tuple is the distance from the origin.

An example of “collective operation”

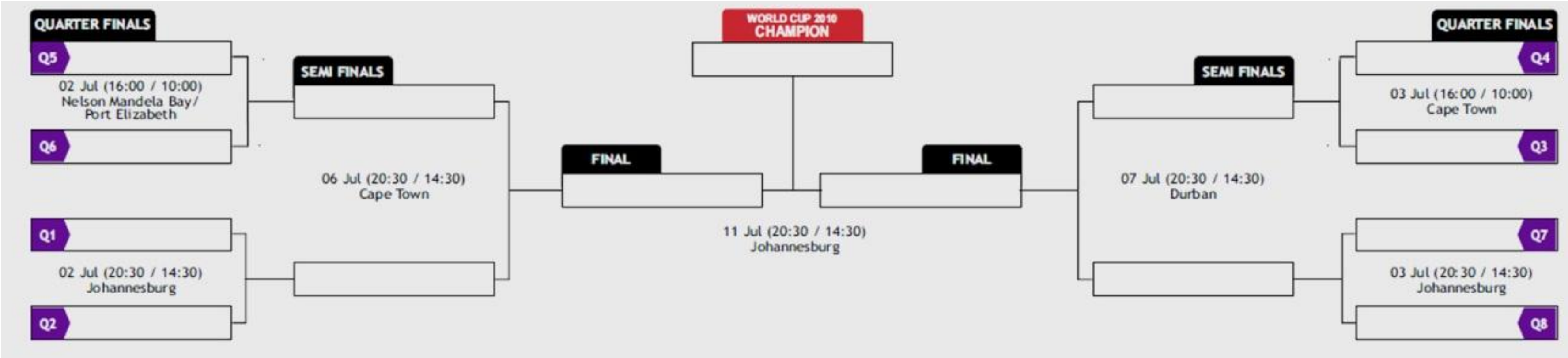
An Efficient Sequential Reduction O(N)

– Initialize the result as an identity value for the reduction operation
  – Smallest possible value for max reduction
  – Largest possible value for min reduction
  – 0 for sum reduction
  – 1 for product reduction
– Iterate through the input and perform the reduction operation between the result value and the current input value
  – N reduction operations performed for N input values
  – Each input value is only visited once – an O(N) algorithm
  – This is a computationally efficient algorithm.

A parallel reduction tree algorithm performs N-1 operations in log(N) steps



A tournament is a reduction tree with “max” operation



A Quick Analysis

- For N input values, the reduction tree performs
  - (1/2)N + (1/4)N + (1/8)N + … (1)N = (1- (1/N))N = N-1 operations
  - In Log (N) steps – 1,000,000 input values take 20 steps
    - Assuming that we have enough execution resources
  - Average Parallelism (N-1)/Log(N))
    - For N = 1,000,000, average parallelism is 50,000
    - However, peak resource requirement is 500,000
    - This is not resource efficient
- This is a work-efficient parallel algorithm
  - The amount of work done is comparable to an efficient sequential algorithm
  - Many parallel algorithms are not work efficient

GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

Module 9.2 – Parallel Computation Patterns (Reduction)

A Basic Reduction Kernel

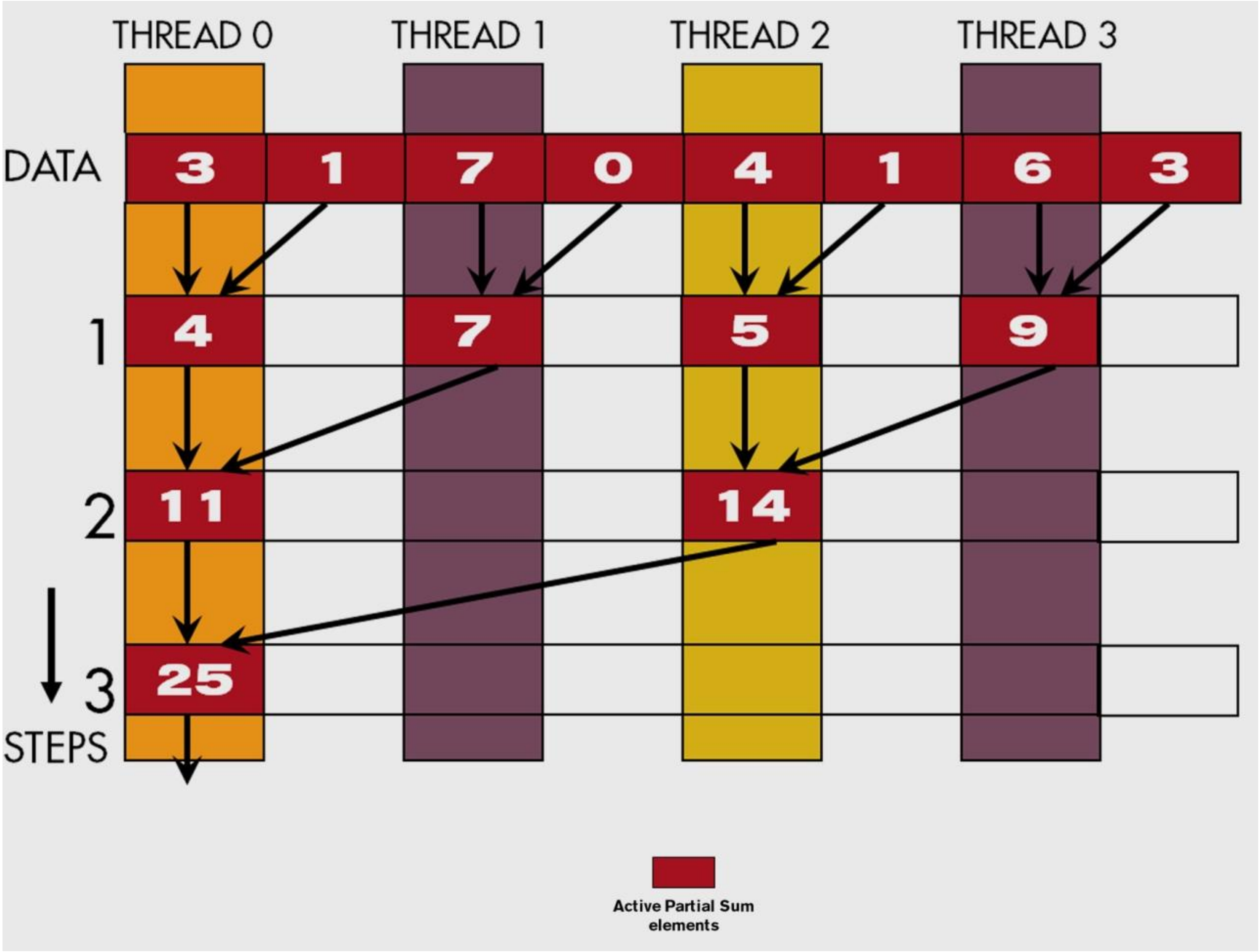
Objective

- To learn to write a basic reduction kernel
  - Thread to data mapping
  - Turning off threads
  - Control divergence

Parallel Sum Reduction

- Parallel implementation
  - Recursively halve # of threads, add two values per thread in each step
  - Takes log(n) steps for n elements, requires n/2 threads
- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory is used to hold a partial sum vector
  - Each step brings the partial sum vector closer to the sum
  - The final sum will be in element 0 of the partial sum vector
  - Reduces global memory traffic due to partial sum values
  - Thread block size limits n to be less than or equal to 2,048

A Parallel Sum Reduction Example



A Naive Thread to Data Mapping

– Each thread is responsible for an even-index location of the partial sum vector (location of responsibility)
– After each step, half of the threads are no longer needed
– One of the inputs is always from the location of responsibility
– In each step, one of the inputs comes from an increasing distance away

A Simple Thread Block Design

– Each thread block takes 2*BlockDim.x input elements
– Each thread loads 2 elements into shared memory

```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
partialSum[2*t] = input[2*t];
partialSum[2*t + 1] = input[2*t + 1];
```

A Simple Thread Block Design

- Each thread block takes 2*BlockDim.x input elements
- Each thread loads 2 elements into shared memory

```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
partialSum[t] = input[t];
partialSum[BLOCK_SIZE+t] = input[BLOCK_SIZE+t];
```

The Reduction Steps

```
for (unsigned int stride = 1;
     stride <= blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % stride == 0)
    partialSum[2*t]+= partialSum[2*t+stride];
}
```

Why do we need __syncthreads()?

Barrier Synchronization

- __syncthreads() is needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step

Back to the Global Picture

- At the end of the kernel, Thread 0 in each thread block writes the sum of the thread block in partialSum[0] into a vector indexed by the blockIdx.x
- There can be a large number of such sums if the original vector is very large
  - The host code may iterate and launch another kernel
- If there are only a small number of sums, the host can simply transfer the data back and add them together

GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

Module 9.3 – Parallel Computation Patterns (Reduction)

A Better Reduction Kernel

Objective

- To learn to write a better reduction kernel
  - Resource efficiency analysis
  - Improved thread to data mapping
  - Reduced control divergence

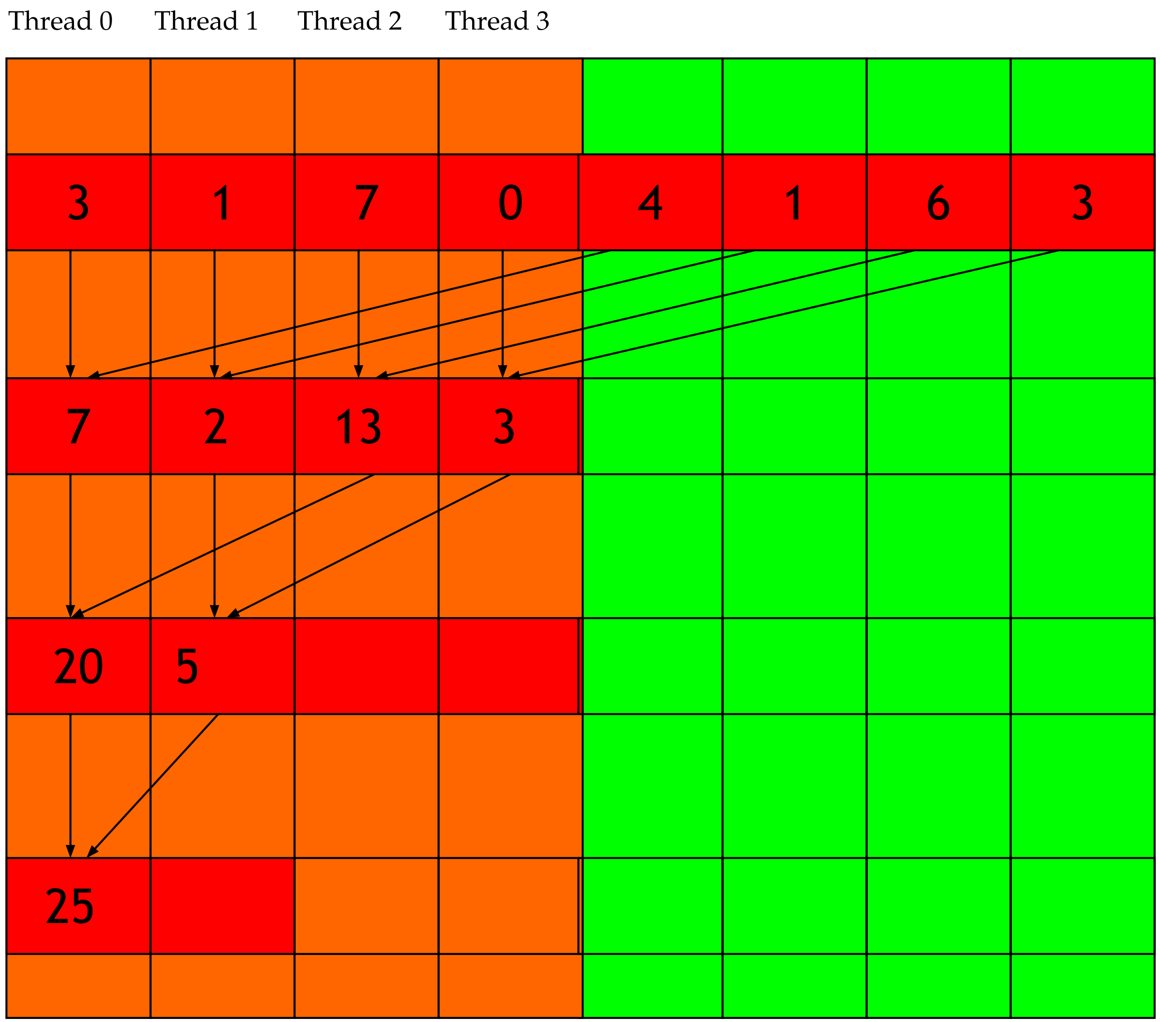
Some Observations on the naïve reduction kernel

- In each iteration, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition still consume execution resources
- Half or fewer of threads will be executing after the first step
  - All odd-index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the `if` test, poor resource utilization but no divergence
    - This can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire

Thread Index Usage Matters

- In some algorithms, one can shift the index usage to improve the divergence behavior
  - Commutative and associative operators
- Always compact the partial sums into the front locations in the partialSum[ ] array
- Keep the active threads consecutive

An Example of 4 threads



A Better Reduction Kernel

```
for (unsigned int stride = blockDim.x;
     stride > 0;  stride /= 2)
{
   __syncthreads();
   if (t < stride)
      partialSum[t] += partialSum[t+stride];
}
```

A Quick Analysis

- For a 1024 thread block
  - No divergence in the first 5 steps
    - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
    - All threads in each warp either all active or all inactive
  - The final 5 steps will still have divergence

GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

Module 10.1 – Parallel Computation Patterns (scan)

Prefix Sum

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

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 $[x_0, x_1, x_2, \ldots]$

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 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."

GPU Teaching Kit

GPU Teaching Kit

Accelerated Computing

Module 10.2 – Parallel Computation Patterns (scan)

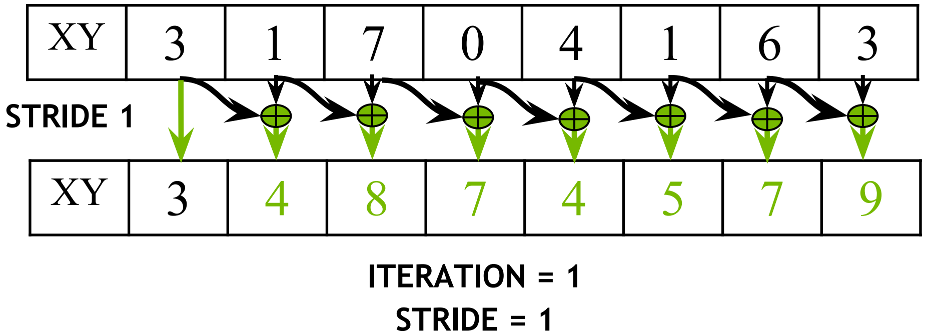
A Work-inefficient Scan Kernel

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

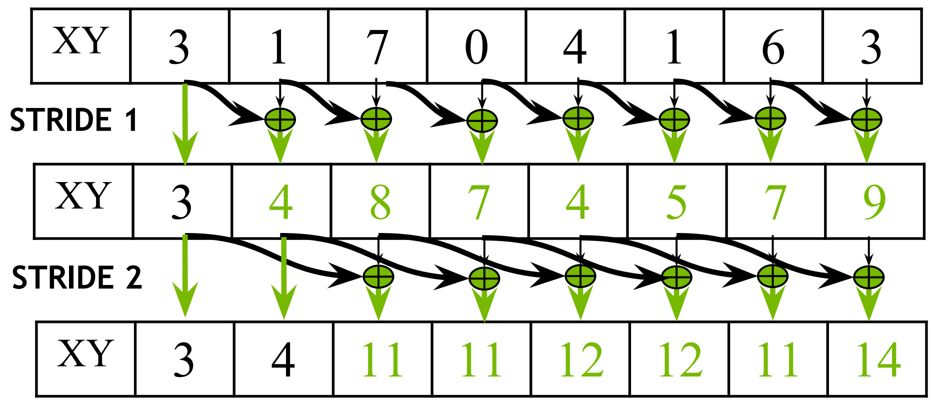
1. Read input from device global memory to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration



- 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

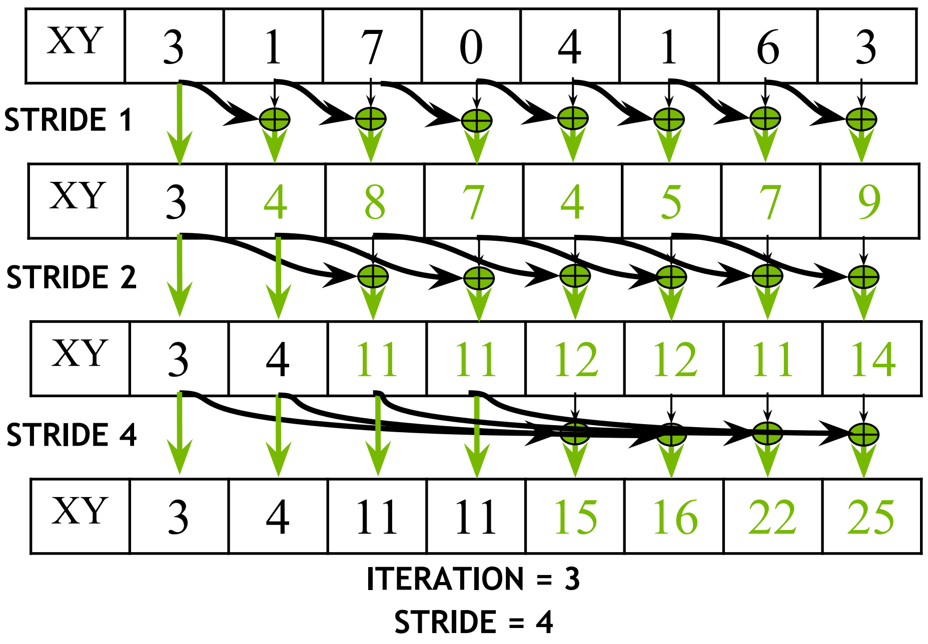
1. Read input from device to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration.



ITERATION = 2
STRIDE = 2

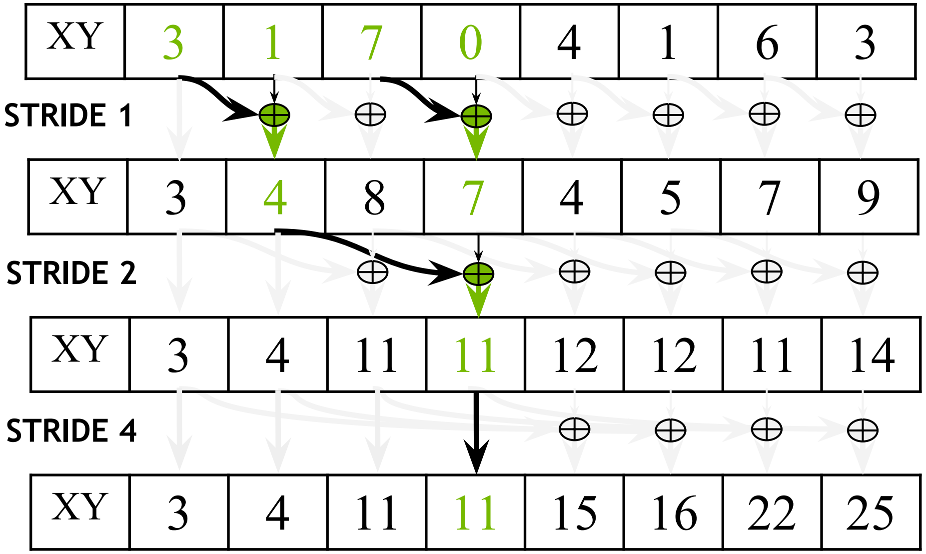
A Better Parallel Scan Algorithm

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



A Better Parallel Scan Algorithm

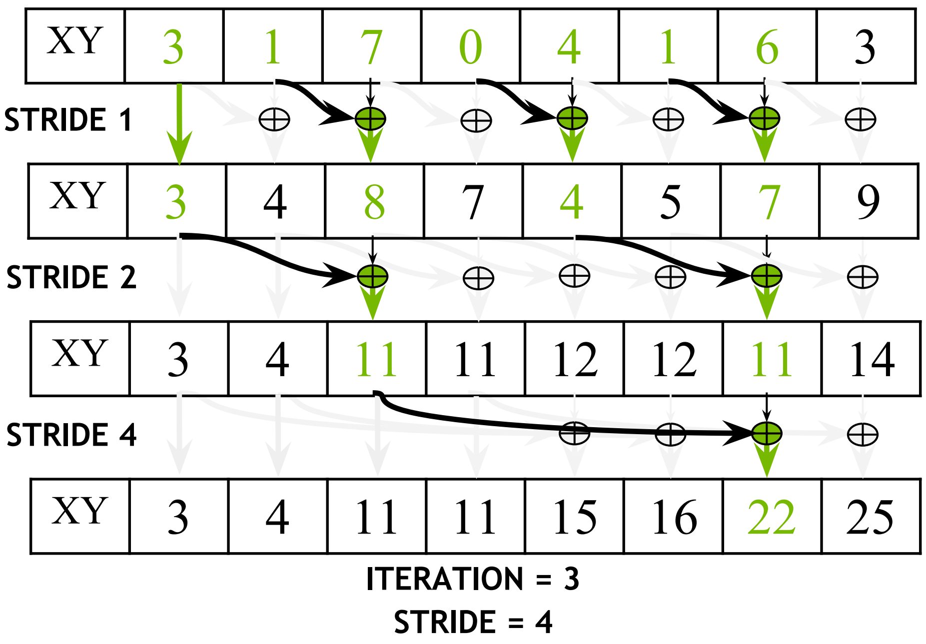
1. Read input from device to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration
3. Write output from shared memory to device memory



ITERATION = 3
STRIDE = 4

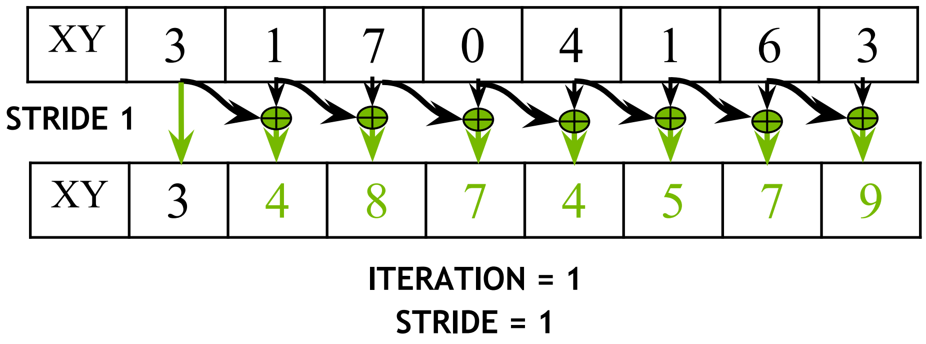
A Better Parallel Scan Algorithm

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



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) → 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

GPU Teaching Kit

ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

GPU Teaching Kit

Accelerated Computing

Lecture 10.3 – Parallel Computation Patterns (scan)

A Work-Efficient Parallel Scan Kernel

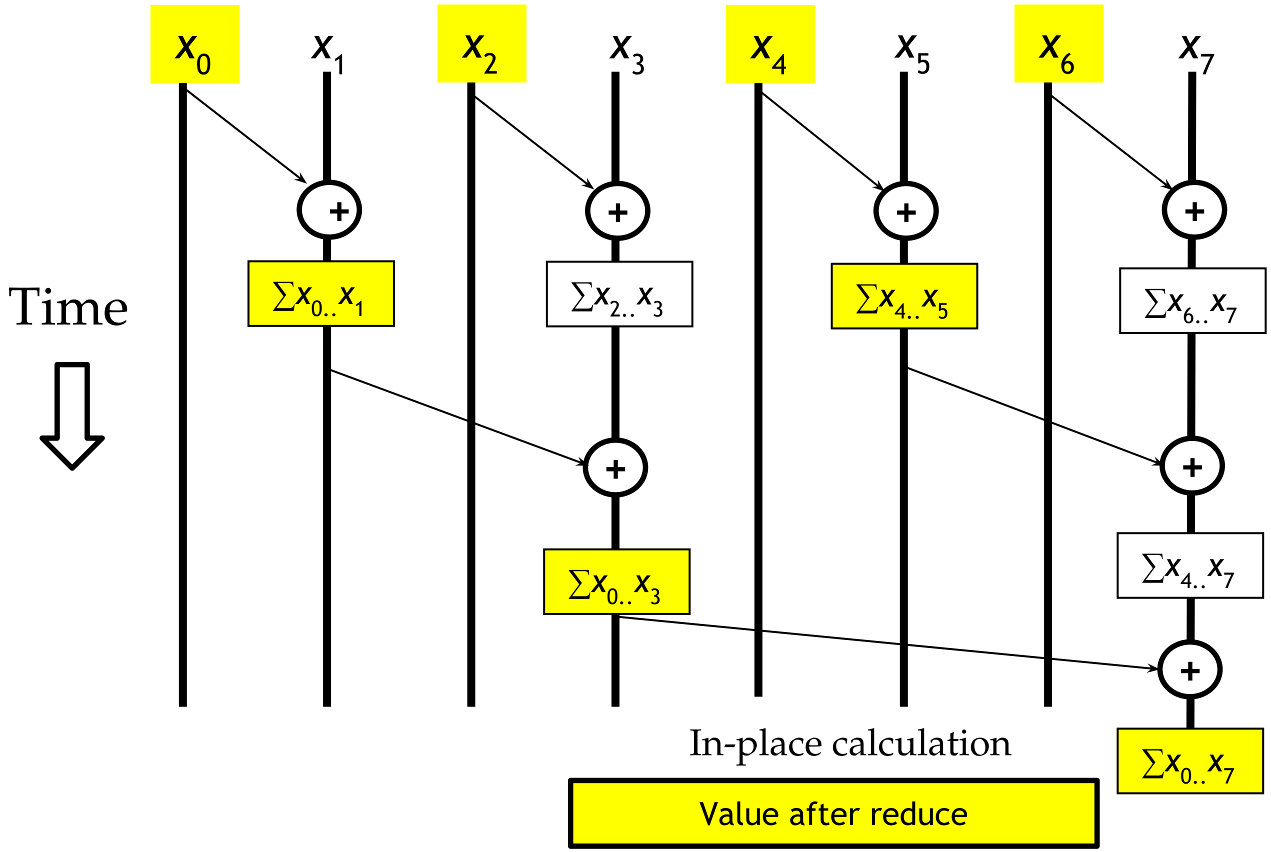
Objective

- 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

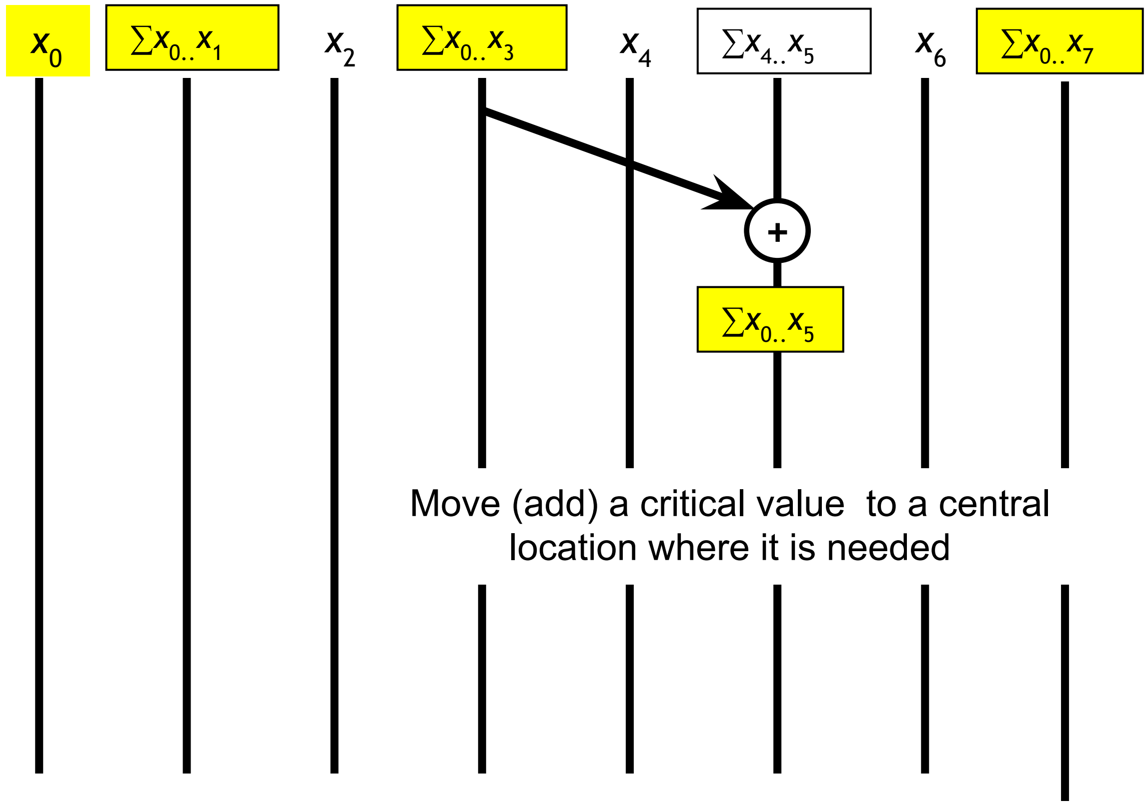


Reduction Phase Kernel Code

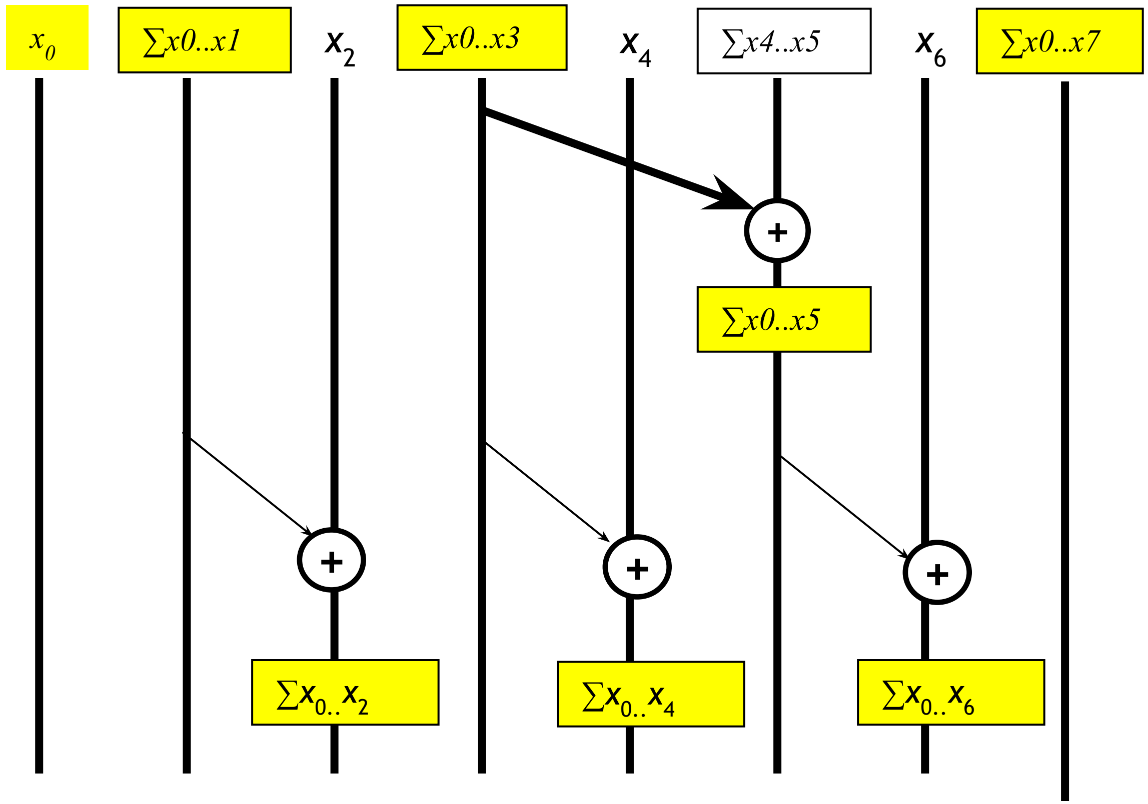
```
// XY[2*BLOCK_SIZE] is in shared memory

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, …

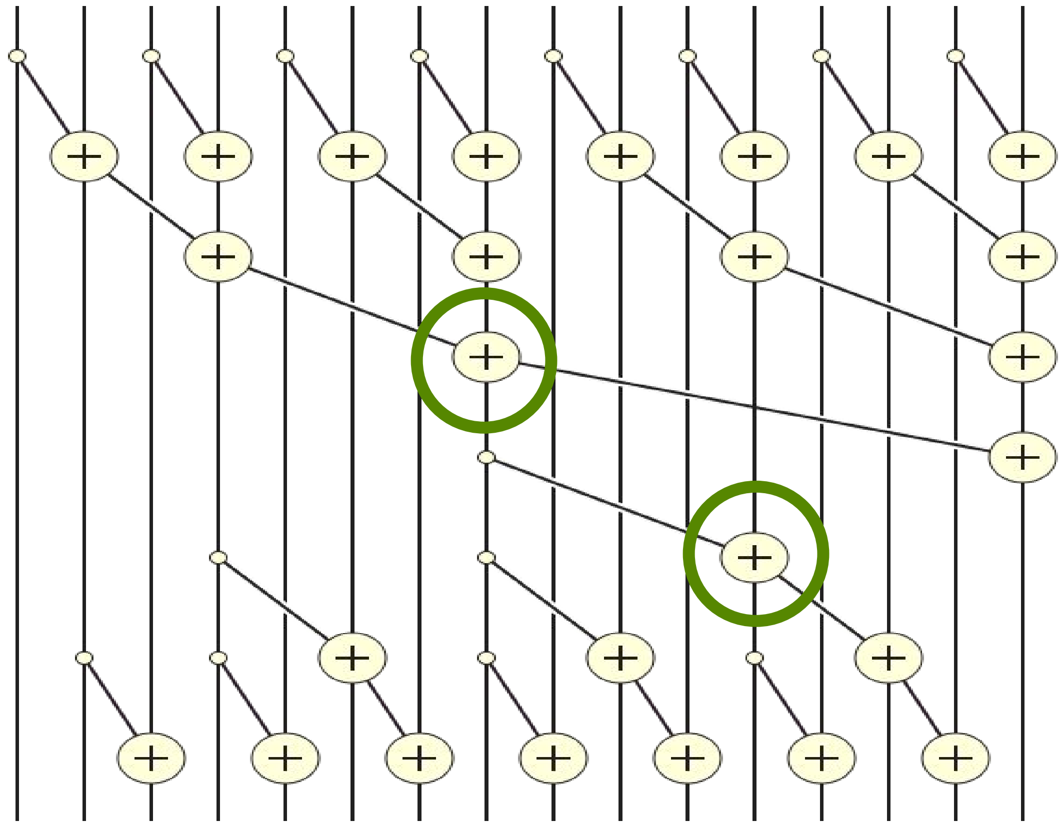
Parallel Scan - Post Reduction Reverse Phase



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

GPU Teaching Kit

GPU Teaching Kit

Accelerated Computing

ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Module 10.4 – Parallel Computation Patterns (scan)

More on Parallel Scan

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

Can We do Even Better?

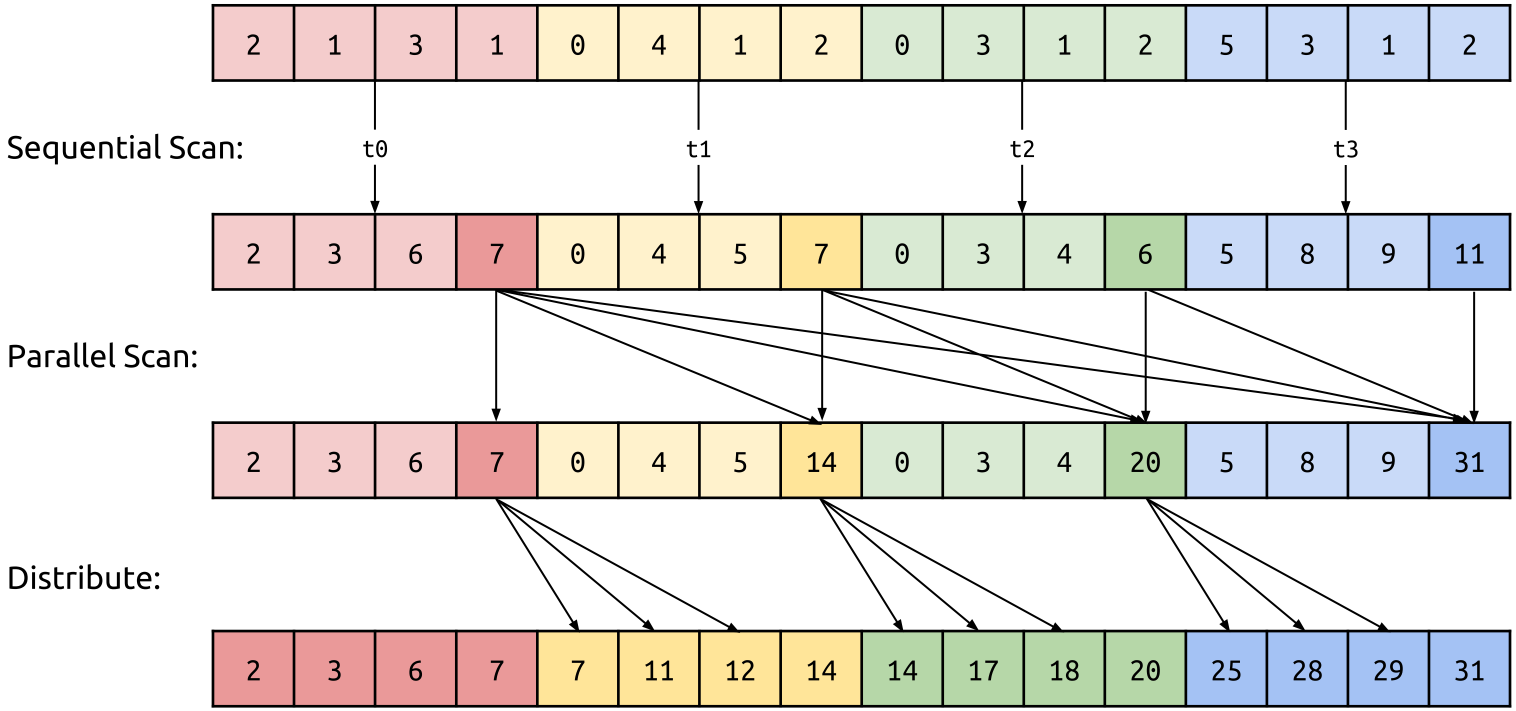
- There are still many inactive threads in many iterations of the work-efficient scan
- Inactive threads still require resources (registers, PC, etc.) to remain resident in a SM
- For large inputs, the performance of the work-efficient kernel may start to resemble O(nlog(n)) rather than O(n)

Thread Granularity Adjustment

- A thread granularity adjustment will make better use of computational resources
- Each thread is assigned a contiguous section of the input
- The scan proceeds in three steps:

  1. Each thread performs a sequential scan of its assigned section
  2. Threads collaborate to perform a parallel scan of the partial sums
  3. Each thread adds the previous thread's prefix sum to all scan values in its assigned section

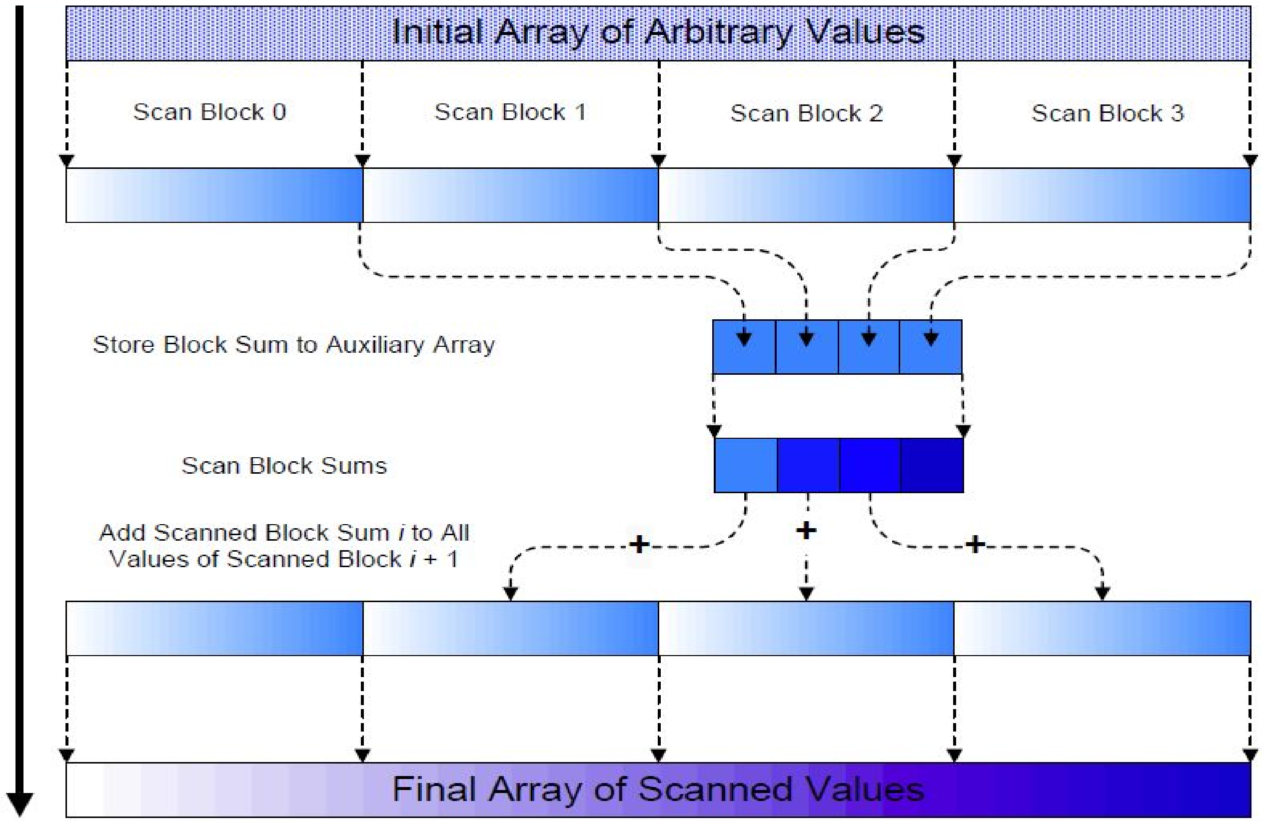
Thread Granularity Adjustment



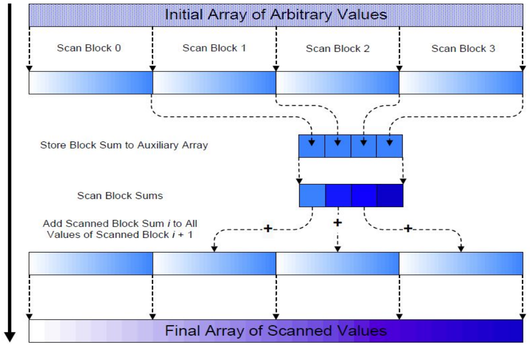
Handling Large Input Vectors

- 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

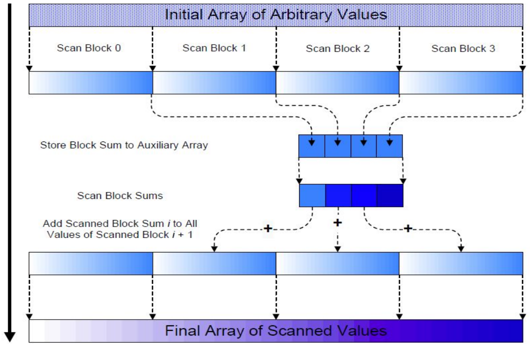


Multi-block Scan (Part 1)




```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start + blockDim.x+t];

…

if (t == 0)
  aux[blockIdx.x] = partialSum[2*BLOCK_SIZE-1];
```

Multi-block Scan (Part 1)




```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start + blockDim.x+t];

…

if (t == 0)
   aux[blockIdx.x] = partialSum[2*BLOCK_SIZE-1];
```

Multi-block Inefficiencies

- Intermediate results are computed in shared memory, then saved in global memory

- Phase 2 reads a subset of the intermediate results from global memory, performs a scan in shared memory, and saves the result back to the global memory

- Phase 3 reads the Phase 2 results from global memory and updates (almost) all global memory values

“Streaming” Scan

- These inefficiencies can be overcome with message passing.
- After computing the local sum, one thread from the block waits for a message from the previous block containing the prefix sum up to that point
- This thread then adds the local sum and passes the result to the next block
- Finally, the prefix sum from the previous block is added to all local results
- This multi-block scan can be done with one kernel launch, reducing the need for round trips to global memory

“Streaming” Scan

```
__shared__ float previous_sum;

// perform local scan (Phase 1)
...

if (threadIdx.x == 0) {
  // Wait for the previous flag
  while (atomicAdd(&flags[bid], 0) == 0) {;}
  // Read previous partial sum
  previous_sum = scan_value[bid];
  // Propagate partial sum
  scan_value[bid+1] = previous_sum + local_sum;
  // Memory fence
  __threadfence();
  // Set flag
  atomicAdd(&flags[bid + 1], 1);
}
__syncthreads();

// perform local distribution (Phase 3)
...
```

“Streaming” Scan

```
const int bid = blockIdx.x;
__shared__ float previous_sum;

// perform local scan (Phase 1)
...

if (threadIdx.x == 0) {
  // Wait for the previous flag
  while (atomicAdd(&flags[bid], 0) == 0) {;}
  // Read previous partial sum
  previous_sum = scan_value[bid];
  // Propagate partial sum
  scan_value[bid+1] = previous_sum + local_sum;
  // Memory fence
  __threadfence();
  // Set flag
  atomicAdd(&flags[bid + 1], 1);
}
__syncthreads();

// perform local distribution (Phase 3)
...
```

This code is susceptible to deadlock!

Dynamic Block ID Assignment

```
__shared__ int bid;

if (threadIdx.x == 0) {
  bid = atomicAdd(DCounter, 1);
}
__syncthreads();
```

```
__shared__ float previous_sum;

// perform local scan (Phase 1)
...

if (threadIdx.x == 0) {
  // Wait for the previous flag
  while (atomicAdd(&flags[bid], 0) == 0) {;}
  // Read previous partial sum
  previous_sum = scan_value[bid];
  // Propagate partial sum
  scan_value[bid+1] = previous_sum + local_sum;
  // Memory fence
  __threadfence();
  // Set flag
  atomicAdd(&flags[bid + 1], 1);
}
__syncthreads();

// perform local distribution (Phase 3)
...
```

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.

GPU Teaching Kit

Stream Compaction

- A common use case for parallel scans
- Stream compaction is the removal of unwanted or irrelevant elements from an input stream based on some predicate
- The elements which pass the predicate test are placed in contiguous memory

Stream Compaction

0	7	0	0	4	0	1	0	0	0	8	4	0	0	6	0

Stream Compaction

0	7	0	0	4	0	1	0	0	0	8	4	0	0	6	0

Predicate: x > 0

Stream Compaction

0	7	0	0	4	0	1	0	0	0	8	4	0	0	6	0

Predicate: x > 0

0	1	0	0	1	0	1	0	0	0	1	1	0	0	1	0

Stream Compaction

0	7	0	0	4	0	1	0	0	0	8	4	0	0	6	0

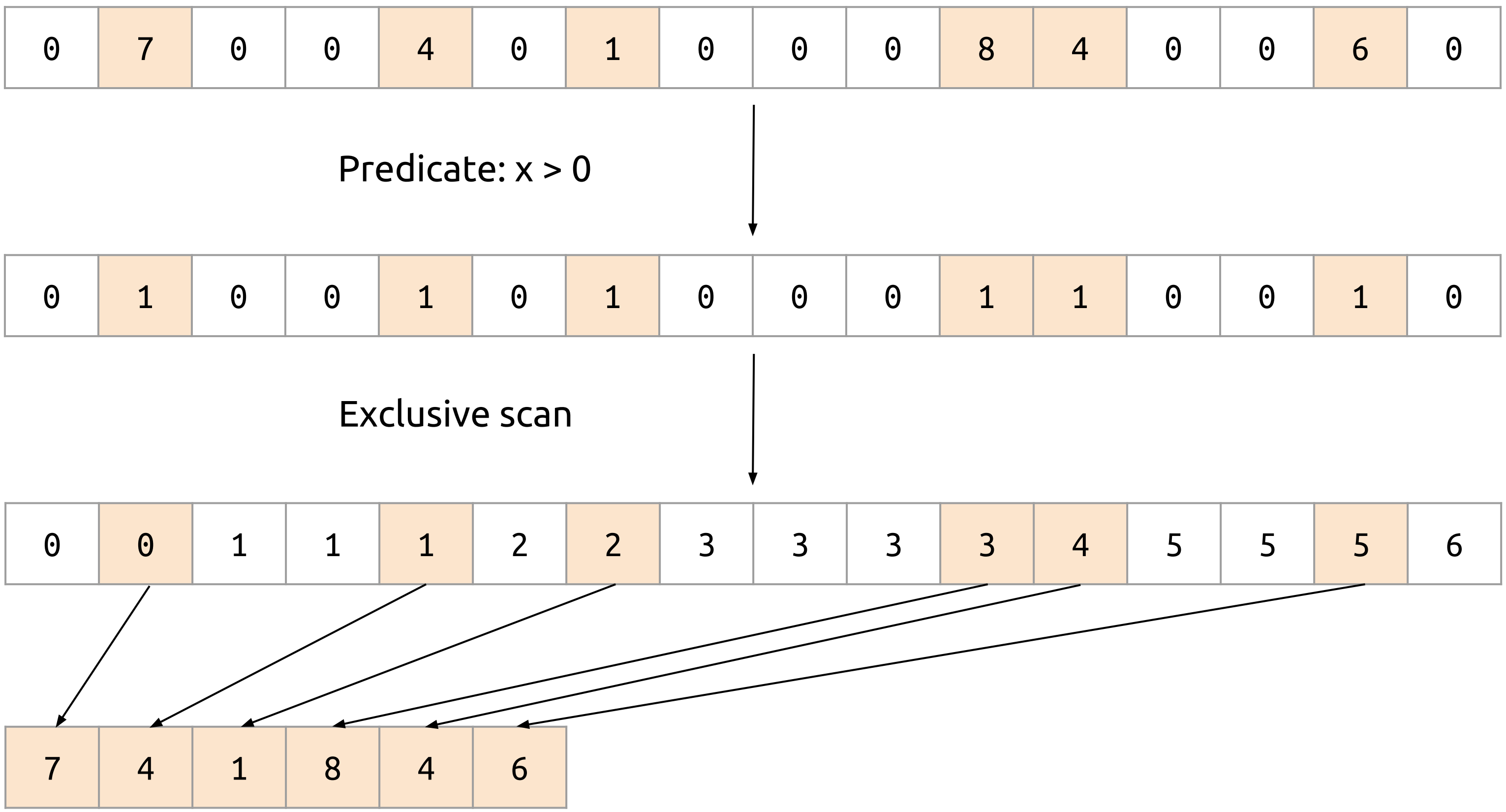
Predicate: x > 0

0	1	0	0	1	0	1	0	0	0	1	1	0	0	1	0

Exclusive scan

0	0	1	1	1	2	2	3	3	3	3	4	5	5	5	6

Stream Compaction




```
if (predicate(input[x])) {
  output[scan[x]] = input[x];
}
```