ECE408 Spring 2018

Applied Parallel Programming

Lecture 14
Parallel Computation Patterns – Parallel Scan (Prefix Sum)

Objective

- To master parallel Scan (Prefix Sum) algorithms
  - frequently used for parallel work assignment and resource allocation
  - A key primitive used in many parallel algorithms to convert serial computation into parallel computation
  - Based on reduction tree and reverse reduction tree
- To learn the concept of double buffering

(Inclusive) 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 Inclusive 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
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms
  - Etc.

Other Applications

- Assigning camp slots
- Assigning farmer market space
- Allocating memory to parallel threads
- Allocating memory buffer to communication channels
- …

An Inclusive Sequential Scan

Given a sequence $[x_0, x_1, x_2, \ldots]$

Calculate output $[y_0, y_1, y_2, \ldots]$

Such that

$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$
$$\ldots$$

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 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."

Parallel Inclusive Scan using Reduction Trees

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

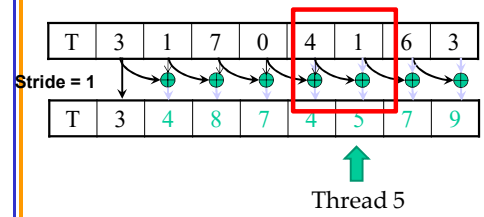
A Kogge-Stone Parallel Scan Algorithm

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

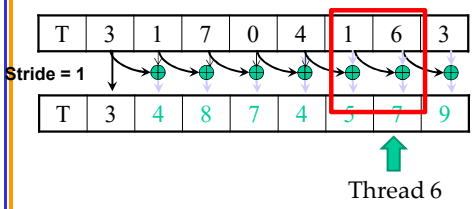


1. (previous slide)
2. Assuming n is a power of 2. Iterate log(n) times, stride from 1 to n/2. Threads *stride* to *n-1 active:* add pairs of elements that are *stride* elements apart.

Iteration #1
Stride = 1

- Active threads: *stride* to *n*-1 (*n* - *stride* active threads)
- Thread *j* adds elements T[*j*] and T[*j-stride*] and writes result into element T[*j*]
- Each iteration requires two synchthreads
  - make sure that input is in place
  - make sure that all input elements have been used

A Kogge-Stone Parallel Scan Algorithm

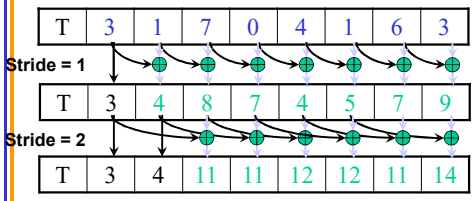


1. (previous slide)
2. Assuming n is a power of 2. Iterate log(n) times, stride from 1 to n/2. Threads *stride* to *n-1 active:* add pairs of elements that are *stride* elements apart.

- Active threads: *stride* to *n*-1 (*n - stride* active threads)
- Thread *j* adds elements T[*j*] and T[*j-stride*] and writes result into element T[*j*]
- Each iteration requires two synchthreads
  - syncthreads(); // make sure that input is in place
  - float temp = T[*j*] + T[*j-stride*];
  - syncthreads(); // make sure that previous output has been consumed
  - T[*j*] = temp;

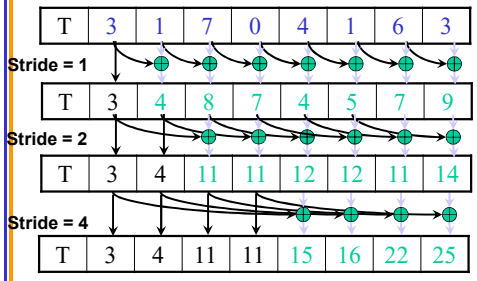
Iteration #1
Stride = 1

A Kogge-Stone Parallel Scan Algorithm



Iteration #2
Stride = 2

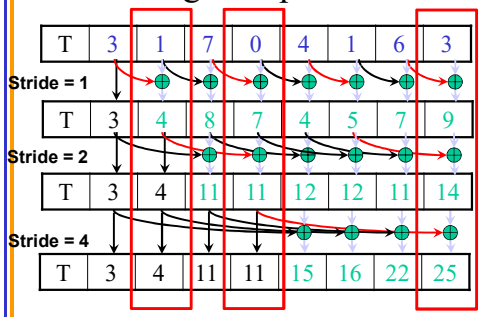
A Kogge-Stone Parallel Scan Algorithm



1. …
2. …
3. Write output from shared memory to device memory

Iteration #3
Stride = 4

Sharing Computation in Kogge-Stone

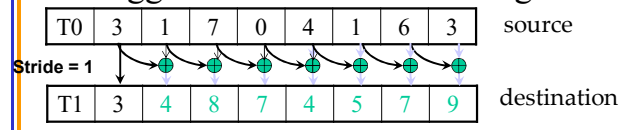


Iteration #3
Stride = 4

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() call

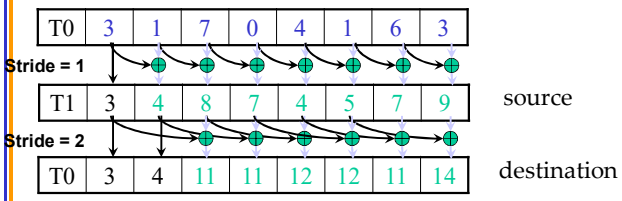
A Double-Buffered Kogge-Stone Parallel Scan Algorithm



- source = &T0[0]; destination = &T1[0];
- 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;
- After the loop, write destination contents to global memory

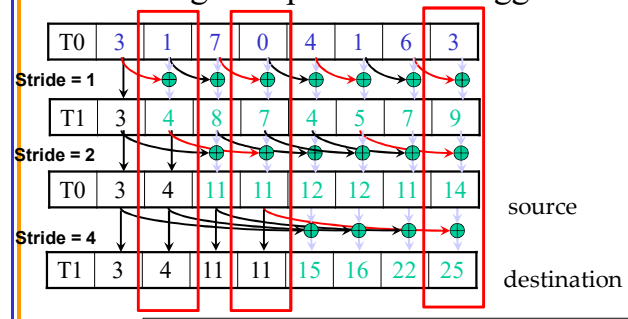
Iteration #1
Stride = 1

A Kogge-Stone Parallel Scan Algorithm



Iteration #2
Stride = 2

Sharing Computation in Kogge-Stone



- 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;
- After the loop, write destination contents to global memory

Iteration #3
Stride = 4

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

ANY MORE QUESTIONS?