Prefix Sum

Outline

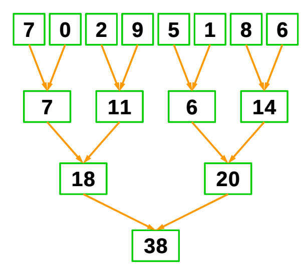
- Reduction operations
  - Parallel reduction
- Prefix sum
  - Parallel scan
  - Work-efficient scan
- Applications of scan
  - Stream compaction

Reduction

- A class of operations involves:
  - A ordered set $S=\{a_0, a_1, a_2, \dots, a_{n-1}\}$ of n numbers
  - A binary associative operator
- Examples of reduction operations:
  - Sum = Reduce(+, S) = $a_0 + a_1 + a_2 + \dots + a_{n-1}$
  - Product = Reduce(×, S) = $a_0 \times a_1 \times a_2 \times \dots \times a_{n-1}$
  - Min = Reduce(min, S) = $\min(a_0, a_1, a_2, \dots, a_{n-1})$
- The output is a single number
  - Require O(N) time to computer on a sequential computer

Parallel Reduction

- Technique for performing reduction on parallel computers
  - Compute the sum of 2 numbers in each step
  - Reduce the numbers in the set by half
- Require $\log_2 N$ steps on a N-processor computer
  - Require $N/M+\log_2 M$ on M processors (M<N)

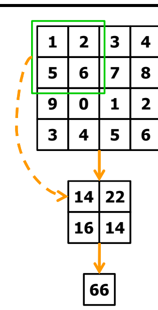


Speedup & Efficiency

- Speedup is the time it takes to complete an algorithm on 1 processor divided by the time it takes on N processors
  - Measures the gain of parallelizing an algorithm
- Speedup of parallel reduction is $N/\log_2 N$
  - With M processors M<N, the speedup is $N/(N/M+\log_2 M)$
- Efficiency is defined as the speedup divided by the number of processors used
  - Measures how well the processors are unitized
- Efficiency of parallel reduction is $1/\log_2 N$
  - With M processors M<N, the speedup is $1/(N/M+\log_2 M)$

Parallel Reduction on GPU

- Put data in a square texture & perform 2D reduction
  - Render a quarter-sized texture each pass
- Use shader:
  - Fetch nearby 4 values & calculate the sum
- Use build-in texture sampling functionality:
  - Set sampler to linear
  - Fetch the value at the center of the 4 pixels

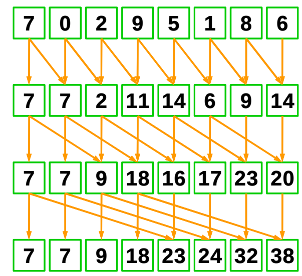


Prefix Sum (a.k.a. Scan)

- Given a list of n numbers, compute the partial sums using only numbers on the left sides
  - Input: $a_0 , a_1 , a_2 , \ldots , a_{n-1}$
  - Output: $a_0 , a_0+a_1 , a_0+a_1+a_2 , \ldots, a_0+a_1+a_2+\ldots+a_{n-1}$
  - Require O(N) on a sequential computer
- Two variants of scan:
  - Inclusive scan: add all numbers on the left and the number itself
  - Exclusive scan: only add numbers on the left
    - *The first output is zero*
    - *The last number in the input list is not used*

Parallel Scan

- A commonly used building block for parallel algorithms
  - Require $\log_2 N$ steps on a N-processor computer
- In each step k, k from 0 to $\log_2 N$:
  - if $i > 2^k$, add number a[i] with $a[i-2^k]$



Algorithm Complexity

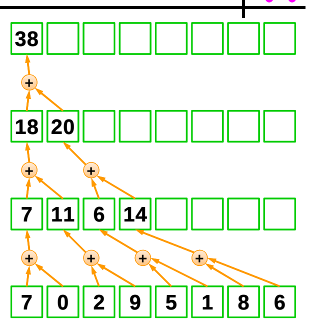
- On a computer with N processors:
  - The total time needed to complete is $\log_2 N$
  - The speedup is $N/\log_2 N$
  - The efficiency is $1/\log_2 N$
- On a computer with M processors (M<N):
  - The total number of addition operations needed is $N \times \log_2 N$
  - The total time needed to complete the additions is $(N \times \log_2 N)/M$
  - Reduce the redundant add operations can further improve processing speed

Work Efficient Parallel Scan

- Based on the balanced tree data structure
  - Build a balanced binary tree on the input data, then traverse the tree to and from the root
  - Perform one add per tree node, resulting a total of O(N) addition operations
- The algorithm consists of 2 phases
  - Upsweep phase traverses the tree from leaves to root computing partial sums
  - Down-sweep phase traverses from the root to leaves, using the partial sums to build the scan

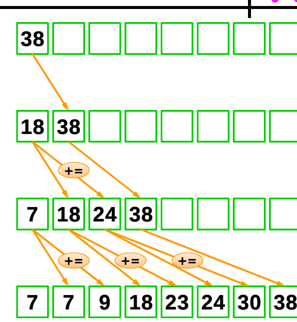
Upsweep Phase

```
// same operation as parallel reduction
for ( d = 1 to log2n ) {
    for ( i = 1 to n/2^d -1 ) do in parallel {
        a_d[i] = a_(d-1)[2i] + a_(d-1)[2i+1]
    }
}
```



Down-sweep Phase

```
for (d = (log2n)-1 downto 0) {
    for ( i = 0 to n/2^d -1 ) do in parallel {
        if ( i > 0 ) {
            if ( (i mod 2) ≠ 0 )
                a_d[i] = a_(d+1)[i/2]
            else
                a_d[i] += a_(d+1)[(i/2)-1]
        }
    }
}
```



Applications of Scan

- Radix sort
- Quicksort
- String comparison
- Lexical analysis
- Stream compaction
- Sparse matrices
- Polynomial evaluation
- Solving recurrences
- Tree operations
- Histograms

Stream Compaction

- Generate a compact stream by removing unwanted items from the original stream
  - Input: an ordered set S & a predicate p
  - Output: only elements v for which p(v) is true, preserving the ordering of the input elements
- Applications:
  - An important operation in collision detection & sparse matrix compression
  - Can be used to transform a heterogeneous vector, with elements of many types, into homogeneous vectors, in which each element has the same type

Stream Compaction Example

- Remove ≤4 numbers from the input stream
- Create a bit stream
  - Label >4 with 1
  - Label ≤4 with 0
- Apply exclusive prefix sum on the bit stream
- Store numbers into the addresses specified by the result of prefix sum
  - Require scatter support

| 7 | 0 | 2 | 9 | 5 | 1 | 8 | 6 |

Selection

1 0 0 1 1 0 1 1

Exclusive scan

0 1 1 1 2 3 3 4

Scatter

| 7 | 9 | 5 | 8 | 6 |