Prefix Sum
Outline

• Reduction operations
  • Parallel reduction

• Prefix sum
  • Parallel scan
  • Work-efficient scan

• Applications of scan
  • Line of light
  • Stream compaction
Reduction

- A class of operations involves:
  - A ordered set $S=\{a_0, a_1, a_2, \ldots, a_{n-1}\}$ of $n$ numbers
  - A binary associative operator

- Examples of reduction operations:
  - Sum = Reduce($+$, $S$) = $a_0 + a_1 + a_2 + \ldots + a_{n-1}$
  - Product = Reduce($\times$, $S$) = $a_0 \times a_1 \times a_2 \times \ldots \times a_{n-1}$
  - Min = Reduce($\text{min}$, $S$) = $\text{min}(a_0, a_1, a_2, \ldots, a_{n-1})$

- The output is a single number
  - Require $O(N)$ time to compute 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 _____ 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 \( \frac{N}{\log_2 N} \)
    - With \( M \) processors \( M < N \), the speedup is \( \frac{N}{______} \)

- 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 \( \frac{1}{\log_2 N} \)
    - With \( M \) processors \( M < N \), the speedup is \( \frac{1}{______} \)
Inherent 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
MIPMAP

point sampling

mipmaps & linear interpolation
CUDA Implementation

- Need to use multiple thread blocks
  - For very large arrays
- How to communicate partial results between thread blocks?
  - CUDA has no global sync (among blocks)
  - Solution: use multiple kernels
    - Kernel launch as global sync point
    - Kernel launch has no low overhead

Mark Harris, from
Solution: Kernel Deposition

- Avoid global sync by decomposing computation into multiple kernel invocations

- In the case of reduction, code for all levels are the same
Reduction #1

```c
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```
Interleaved Addressing

![Diagram of Interleaved Addressing](image)

1. **Step 1**: Thread IDs (0, 2, 4, 6, 8, 10, 12, 14) are applied with a stride of 1.
   - Values: 
     - Step 1: 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2

2. **Step 2**: Thread IDs (0, 4, 8, 12) are applied with a stride of 2.
   - Values: 
     - Step 2: 11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2

3. **Step 3**: Thread IDs (0, 8) are applied with a stride of 4.
   - Values: 
     - Step 3: 18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2

4. **Step 4**: Thread IDs (0) are applied with a stride of 8.
   - Values: 
     - Step 4: 24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

5. **Step 5**: Thread IDs (0) are applied with a stride of 16.
   - Values: 
     - Step 5: 41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

The diagram illustrates how values are accessed using interleaved addressing with different strides.
**Performance for 4M elements**

<table>
<thead>
<tr>
<th>Kernel 1: interleaved addressing with divergent branching</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
</tr>
</tbody>
</table>

Block Size = 128 threads, on G80 (GeForce 8800 GTX)
Reduction #2

Just replace divergent branch in inner loop:

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2^s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

With strided index and non-divergent branch:

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```
Parallel Reduction
## Performance

<table>
<thead>
<tr>
<th></th>
<th>Time ($2^{22}$ ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel 1:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Kernel 2:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
</tbody>
</table>
More optimization

<table>
<thead>
<tr>
<th>Kernel 1: interleaved addressing with divergent branching</th>
<th>Time ((2^{22} \text{ ints}))</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Kernel 2: interleaved addressing with bank conflicts   | 3.456 ms                      | 4.854 GB/s | 2.33x        | 2.33x              |

| Kernel 3: sequential addressing                        | 1.722 ms                      | 9.741 GB/s | 2.01x        | 4.68x              |

| Kernel 4: first add during global load                 | 0.965 ms                      | 17.377 GB/s| 1.78x        | 8.34x              |

| Kernel 5: unroll last warp                            | 0.536 ms                      | 31.289 GB/s| 1.8x         | 15.01x             |

| Kernel 6: completely unrolled                          | 0.381 ms                      | 43.996 GB/s| 1.41x        | 21.16x             |

| Kernel 7: multiple elements per thread                 | 0.268 ms                      | 62.671 GB/s| 1.42x        | 30.04x             |
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 ___
  - The speedup is _____
  - The efficiency is ______

- On a computer with M processors (M<N):
  - The total number of addition operations needed is ______
  - The total time needed to complete the additions is ______
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]
    }
- }
- }

38

18 20

7 11 6 14

7 0 2 9 5 1 8 6
Down-sweep Phase

- for \( d = (\log_2 n) - 1 \) downto 0
  
  \[
  \begin{array}{c}
  \text{for } (i = 0 \text{ to } n/2^d - 1) \\
  \text{do in parallel} \\
  \text{if } (i > 0) \\
  \text{if } ((i \mod 2) \neq 0) \\
  a_d[i] = a_{d+1}[i/2] \\
  \text{else} \\
  a_d[i] += a_{d+1}[(i/2) - 1]
  \end{array}
  \]
Applications of Scan

- Radix sort
- Quicksort
- String comparison
- Lexical analysis
- Stream compaction
- Sparse matrices
- Polynomial evaluation
- Solving recurrences
- Tree operations
- Histograms
Sample App – Line of Sight

Altitude Map

Altitude Vector

Angle Vector

Max-Scan of Angle Vector

Ray Vectors
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