Scan Primitives for GPU Computing

Shubho Sengupta, Mark Harris*, Yao Zhang, John Owens

University of California Davis, *NVIDIA Corporation
Motivation

- Raw compute power and bandwidth of GPUs increasing rapidly
- Programmable unified shader cores
- Ability to program outside the graphics framework
- However lack of efficient data-parallel primitives and algorithms
Motivation

- Current efficient algorithms either have streaming access
- 1:1 relationship between input and output element
Motivation

- Or have small “neighborhood” access
- N:1 relationship between input and output element where N is a small constant
Motivation

- However interesting problems require more general access patterns
  - Changing one element affects everybody
- Stream Compaction
Motivation

- Split

- Needed for Sort
Motivation

- Common scenarios in parallel computing
  - Variable output per thread
  - Threads want to perform a split – radix sort, building trees
- “What came before/after me?”
- “Where do I start writing my data?”
- Scan answers this question
System Overview

Algorithms
Sort, Sparse matrix operations, ...

Higher Level Primitives
Enumerate, Distribute, ...

Low Level Primitives
Scan and variants

Libraries and Abstractions
for data parallel programming
Scan

• Each element is a sum of all the elements to the left of it (Exclusive)

• Each element is a sum of all the elements to the left of it and itself (Inclusive)

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>1</th>
<th>7</th>
<th>0</th>
<th>4</th>
<th>1</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>11</th>
<th>15</th>
<th>16</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exclusive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>4</th>
<th>11</th>
<th>11</th>
<th>15</th>
<th>16</th>
<th>22</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inclusive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scan – the past

- First proposed in APL (1962)
- Used as a data parallel primitive in the Connection Machine (1990)
- Guy Blelloch used scan as a primitive for various parallel algorithms (1990)
Scan – the present

• First GPU implementation by Daniel Horn (2004), $O(n \log n)$

• Subsequent GPU implementations by

• NVIDIA CUDA implementation by Mark Harris (2007), $O(n)$, space efficient
Scan – the implementation

- $O(n)$ algorithm – same work complexity as the serial version
- Space efficient – needs $O(n)$ storage
- Has two stages – reduce and down-sweep
Scan - Reduce

- log n steps
- Work halves each step
- O(n) work
- In place, space efficient
Scan - Down Sweep

- log \( n \) steps
- Work doubles each step
- \( O(n) \) work
- In place, space efficient
Segmented Scan

- Input

```
 3 1 7 0 4 1 6 3
```

- Scan within each segment in parallel

- Output

```
0 3 0 7 7 0 1 7
```
Segmented Scan

• Introduced by Schwartz (1980)
• Forms the basis for a wide variety of algorithms
  • Quicksort, Sparse Matrix-Vector Multiply, Convex Hull
Segmented Scan - Challenges

- Representing segments
- Efficiently storing and propagating information about segments
- Scans over all segments should happen in parallel
  - Overall work and space complexity should be $O(n)$ regardless of the number of segments
Representing Segments

- Possible Representations are
  - Vector of segment lengths
  - Vector of indices which are segment heads
  - Vector of flags: 1 for segment head, 0 if not
- First two approaches hard to parallelize as different size as input
- We use the third as it is the same size as input
Segmented Scan – Flag Storage

- Space-Inefficient to store one flag in an integer
- Store one flag in a byte striped across 32 words
- Reduces bank conflicts
Segmented Scan – implementation

- Similar to Scan
  - $O(n)$ space and work complexity
  - Has two stages – reduce and down-sweep
Segmented Scan – implementation

- Unique to segmented scan
  - Requires an additional flag per element for intermediate computation
    - Additional flags get set in reduce step
    - Additional book-keeping with flags in down-sweep
  - These flags prevent data movement between segments
Platform – NVIDIA CUDA and G80

- Threads grouped into blocks
- Threads in a block can cooperate through fast on-chip memory
- Hence programmer must partition problem into multiple blocks to use fast memory
- Adds complexity but usually much faster code
Segmented Scan – Large Input
Segmented Scan – Advantages

- Operations in parallel over all the segments
- Irregular workload since segments can be of any length
- Can simulate divide-and-conquer recursion since additional segments can be generated
Primitives - Enumerate

• Input: a true/false vector

| F | F | T | F | T | T | T | T | F |

• Output: count of true values to the left of each element

| 0 | 0 | 0 | 1 | 1 | 2 | 3 | 3 |

• Useful in stream compact

• Output for each true element is the address for that element in the compacted array
Primitives - Distribute

• Input: a vector with segments

```
3 1 7 4 0 1 6 3
```

• Output: the first element of a segment copied over all other elements

```
3 3 3 4 4 4 6 6
```
Primitives – Distribute

- Set all elements except the head elements to zero

  3 0 0 4 0 0 6 0

- Do inclusive segmented scan

  3 3 3 4 4 4 6 6

- Used in quicksort to distribute pivot
Primitives – Split and Segment

• Input: a vector with true/false elements. Possibly segmented

```
3 1 7 0 4 1 6 3 False
```

• Output: Stable split within each segment – falses on the left, trues on the right

```
3 0 1 7 1 6 4 3
```
Primitives – Split and Segment

• Can be implemented with Enumerate
  • One enumerate for the falses going left to right
  • One enumerate for the trues going right to left
• Used in quicksort
Applications – Quicksort

• Traditional algorithm GPU unfriendly
• Recursive
• Segments vary in length, unequal workload
• Primitives built on segmented scan solves both problems
  • Allows operations on all segments in parallel
  • Simulates recursion by generating new segments in each iteration
Applications – Quicksort

Input

5 3 7 4 6 8 9 3

Distribute pivot

5 5 5 5 5 5 5 5 5

Input > pivot

F F T F T T T T F

Split and Segment

5 3 4 3 7 6 8 9
### Applications – Quicksort

<table>
<thead>
<tr>
<th>5</th>
<th>3</th>
<th>4</th>
<th>3</th>
<th>7</th>
<th>6</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

- **Distribute pivot**
- **Input \( \geq \) pivot**
- **Split and segment**
Applications – Quicksort

Distribute pivot

Input > pivot

Split and segment
Applications – Sparse M-V multiply

- Dense matrix operations are much faster on GPU than CPU
- However Sparse matrix operations on GPU much slower
- Hard to implement on GPU
  - Non-zero entries in row vary in number
Applications – Sparse M-V multiply

- Three different approaches
  - Rows sorted by number of non-zero entries [Bolz]
  - Stored as diagonals and processed them in sequence [Krüger]
  - Rows computed in parallel but runtime determined by longest row [Brook]
Applications – Sparse M-V multiply

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
\end{bmatrix}
\begin{bmatrix}
a & 0 & b \\
c & d & e \\
0 & 0 & f \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
\end{bmatrix}
\]

Non-zero elements:
\begin{bmatrix}
a \\
\end{bmatrix}
\begin{bmatrix}
b \\
c \\
d \\
\end{bmatrix}
\begin{bmatrix}
e \\
f \\
\end{bmatrix}

Column Index:
\begin{bmatrix}
0 \\
2 \\
0 \\
1 \\
2 \\
2 \\
\end{bmatrix}

Row begin Index:
\begin{bmatrix}
0 \\
2 \\
5 \\
\end{bmatrix}
Applications – Sparse M-V multiply

Column Index: \[0 \ 2 \ 0 \ 1 \ 2 \ 2\]

\[\begin{array}{cccccc}
    a & b & c & d & e & f \\
\end{array}\]
\[\begin{array}{cccccc}
    x_0 & x_2 & x_0 & x_1 & x_2 & x_2 \\
\end{array}\]

\[\begin{array}{cccccc}
    ax_0 & bx_2 & cx_0 & dx_1 & ex_2 & fx_2 \\
\end{array}\]

Backward inclusive segmented scan
Pick first element in segment

\[\begin{array}{cccccc}
    ax_0 + bx_2 & cx_0 + dx_1 + ex_2 & fx_2 \\
\end{array}\]
Applications – Tridiagonal Solver

- Implemented Kass and Miller’s shallow water solver
  - Water surface described as a 2D array of heights
- Global movement of data
  - From one end to the other and back
- Suits the Reduce/Down-sweep structure of scan
Applications – Tridiagonal Solver

- Tridiagonal system of $n$ rows solved in parallel
- Then for each of the $m$ columns in parallel
- Read pattern is similar to but more complex than scan
Results - Scan

- Packing and Unpacking Flags
- Non sequential I/O
- Saving State

- Extra computation for sequential memory access

- 1.1x slower
- 3x slower
- 4.8 x slower

Time (Normalized)

Forward Scan
Backward Scan
Forward Segmented Scan
Backward Segmented Scan
Results – Sparse M-V Multiply

- Input: “raefsky” matrix, 3242 x 3242, 294276 elements
- GPU (215 MFLOPS) half as fast as CPU “oski” (522 MFLOPS)
  - Hard to do irregular computation
- Most time spent in backward segmented scan
Results - Sort

- Slow Merge
- Packing/Unpacking Flags
- Complex Kernel

Time (Normalized)

Radix Sort
- Global
- Block

Quick Sort
- GPU (13x slower)
- CPU (4x slower)

2x slower

13x slower
Results – Tridiagonal solver

- 256 x 256 grid: 367 simulation steps per second
- Dominated by the overhead of mapping and unmapping vertex buffers
- 3x faster than a CPU cyclic reduction solver
- 12x faster when using shared memory
Improved Results Since Publication

- Twice as fast for all variants of scan and sparse matrix-vector multiply
- Scan
  - More work per thread – 8 elements vs 2 before
- Segmented Scan
  - No packing of flags
  - Sequential memory access
- More optimizations possible
**Contribution and Future Work**

- Algorithm and implementation of segmented scan on GPU
- First implementation of quicksort on GPU
- Primitives appropriate for complex algorithms
  - Global data movement, unbalanced workload, recursive
  - Scan never occurs in serial computation
- Tiered approach, standard library and interfaces
Acknowledgments

• Jim Ahrens, Guy Blelloch, Jeff Inman, Pat McCormick
• David Luebke, Ian Buck
• Jeff Bolz
• Eric Lengyel
• Department of Energy
• National Science Foundation
Shallow Water Simulation