Scan Primitives for GPU Computing

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

Presented by Mary Fletcher
Slides adapted from authors’ slides

Motivation

- Raw compute power and bandwidth of GPUs increasing rapidly
- Move to general-purpose applications on GPU
- Lack of efficient, general data-parallel primitives and algorithms
Motivation

• Current efficient algorithms either have streaming access
  • 1:1 relationship between input and output element
• Or have small “neighborhood” access
  • k:1 relationship between input and output element where k is a small constant
  • Example, image convolution

Motivation

• However interesting problems require more general access patterns
  • Output depends on arbitrary number of inputs
• Stream Compaction
Motivation

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

Scan (aka prefix sum)

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

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

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

| 3 | 4 | 11| 11| 15| 16| 22| 25|

Output
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 Stage

- log n steps
- Work halves each step
- O(n) total work
Scan - Reduce Stage

- log n steps
- Work halves each step
- $O(n)$ total work

In place, space efficient
Scan - Down Sweep Stage

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

Scan - Implementation

Image from the CUDA programming guide
Segmented Scan

- Input - array broken into segments

| 3 | 1 | 7 | 2 | 4 | 1 | 6 | 3 |

- Scan within each segment in parallel
- Output

| 0 | 3 | 0 | 7 | 9 | 0 | 1 | 7 |
Segmented Scan - Challenges

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

Representing Segments

- Vector of flags: 1 if segment head, 0 if not

\[
\begin{array}{cccccc}
3 & 1 & 7 & 2 & 4 & 1 & 6 & 3 \\
\end{array}
\]

- 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
- Unique to segmented scan
  - Requires an additional flag per element for intermediate computation
  - These flags prevent data movement between segments

Segmented Scan – Advantages

- Operates in parallel over all the segments
- Good for 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 F

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

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 – Split and Segment**

- Input: a vector with true/false elements, possibly segmented

<table>
<thead>
<tr>
<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>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

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

| 3 | 0 | 1 | 7 | 1 | 6 | 4 | 3 |

**Applications – Quicksort**

- Traditional algorithm GPU unfriendly
- Recursive
- Subarrays vary in length, unequal workload
- Primitives built on segmented scan solve both problems
  - Allow operations on all segments in parallel
  - Simulate recursion by generating new segments in each iteration
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

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

Non-zero elements: a, b, c, d, e, f
Row begin index: 0, 2, 0, 1, 2, 2
Column Index: 0, 2, 5
Applications – Sparse M-V multiply

Column Index: 0 2 0 1 2 2

\[ \begin{bmatrix} a & b & c & d & e & f \end{bmatrix} \times \begin{bmatrix} x_0 & x_2 & x_0 & x_1 & x_2 & x_2 \end{bmatrix} = \]

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

\[ ax_0 + bx_2 \quad bx_2 \quad cx_0 + dx_1 + ex_2 \quad dx_1 + ex_2 \quad ex_2 \quad fx_2 \]

Results - Scan

<table>
<thead>
<tr>
<th>Time (Normalized)</th>
<th>Forward Scan</th>
<th>Backward Scan</th>
<th>Forward Segmented Scan</th>
<th>Backward Segmented Scan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.1x slower</td>
<td>3x slower</td>
<td>4.8x slower</td>
<td></td>
</tr>
</tbody>
</table>

Packing and Unpacking Flags
Non-sequential memory access
Saving State

Extra computation for sequential memory access
Results – Sparse M-V Multiply

- Input: “raefsky” matrix, 3242 x 3242, 294276 elements
- GPU - 215 MFLOPS
- OSKI on Pentium 4 - 522 MFLOPS
- Most time spent in backward segmented scan

Results - Sort

<table>
<thead>
<tr>
<th>Time (Normalized)</th>
<th>Global</th>
<th>Block</th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix Sort</td>
<td></td>
<td></td>
<td>13x slower</td>
<td>4x slower</td>
</tr>
<tr>
<td>Quick Sort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Global
- Block
- GPU
- CPU
- Slow Merge
- Packing/Unpacking Flags
- Complex Kernel
Improved Results Since Publication

• Twice as fast for all variants of scan and sparse matrix-vector multiply
• More optimizations possible

Conclusions

• 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
• CUDPP: CUDA Data Parallel Primitives Library
  • http://www.gpgpu.org/scan-gpugems3