Scan Primitives for GPU Computing

Sengupta, Harris*, Zhang, Owens
UC Davis
*NVIDIA Corporation

Presentation by Aydin Buluc for CS595H, Winter 2008
NVIDIA’s CUDA Environment

- Allows arbitrary gather and scatter memory access.
- Each multiprocessor has 16 KB on-chip shared memory.
- Computation is structured as blocks each of which contains up to 512 threads.
- Programmer specifies number of blocks and threads per block.
- Hardware maps blocks to multiprocessors

![CUDA Diagram](image_url)
Memory and Execution Model

- Each thread block is executed by one multiprocessor.
- But a multiprocessor can execute multiple blocks concurrently.
- Increasing the shared memory usage decreases the number of blocks that can be run concurrently.
Scan (All-Prefix-Sums)

- Inputs: A binary associative operator $\oplus$ and a vector of n elements $[a_0, a_1, ..., a_{n-1}]$
- Output: $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$

The algorithms in this paper find the prescan, which is defined as $[i, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$, then add the input vector to get the scan.

Scans are very important parallel building blocks. Has been used to evaluate polynomials, implement radix / quick sort, solve triangular linear systems.
Unsegmented Scan
Phase 1: Up-Sweep

\[
\text{sum}[^v] = \text{sum}[^L[^v]] + \text{sum}[^R[^v]]
\]

1: \textbf{for} \( d = 0 \) to \( \log_2 n - 1 \) \textbf{do}

2: \textbf{for all} \( k = 0 \) to \( n - 1 \) by \( 2^{d+1} \) in parallel \textbf{do}

3: \( x[k + 2^{d+1} - 1] \leftarrow x[k + 2^d - 1] + x[k + 2^{d+1} - 1] \)
What if the number of elements exceeds the maximum number of that a single thread block can process (currently 512)?

- The array is divided across multiple thread blocks, and the partial sums are used as input to the second level scan.
Phase 2: Down-Sweep

\[
\text{sum}[v] = \text{sum}[\text{L}[v]] + \text{sum}[\text{R}[v]]
\]

\[
\text{prescan}[\text{L}[v]] = \text{prescan}[v]
\]
\[
\text{prescan}[\text{R}[v]] = \text{sum}[\text{L}[v]] + \text{prescan}[v]
\]

1: \(x[n - 1] \leftarrow 0\)

2: \(\text{for } d = \log_2 n \text{ down to } 0 \text{ do}\)

3: \(\text{for all } k = 0 \text{ to } n - 1 \text{ by } 2^{d+1} \text{ in parallel do}\)

4: \(t \leftarrow x[k + 2^d - 1]\)

5: \(x[k + 2^d - 1] \leftarrow x[k + 2^{d+1} - 1]\)

6: \(x[k + 2^{d+1} - 1] \leftarrow t + x[k + 2^{d+1} - 1]\)
Array Implementation

<table>
<thead>
<tr>
<th>Step</th>
<th>Array in Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[3 1 7 0 4 1 6 3]</td>
</tr>
<tr>
<td>up</td>
<td>[3 4 7 7 4 5 6 9]</td>
</tr>
<tr>
<td>2</td>
<td>[3 4 7 11 4 5 6 14]</td>
</tr>
<tr>
<td>3</td>
<td>[3 4 7 11 4 5 6 25]</td>
</tr>
<tr>
<td>clear</td>
<td>[3 4 7 11 4 5 6 0]</td>
</tr>
<tr>
<td>down</td>
<td>[3 4 7 0 4 5 6 11]</td>
</tr>
<tr>
<td>6</td>
<td>[3 0 7 4 4 11 6 16]</td>
</tr>
<tr>
<td>7</td>
<td>[0 3 4 11 11 15 16 22]</td>
</tr>
</tbody>
</table>

- This formulation is work-efficient
- Total number of operations ~ 4n = O(n)
Segmented Scans

- A convenient way to execute a scan independently over many sets of values
- Inputs: A data vector and a flag vector
- A flag marks the first element of a segment

\[
\begin{align*}
a &= [5 \ 1 \ 3 \ 4 \ 3 \ 9 \ 2 \ 6] \\
f &= [1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0] \\
\text{segmented +-scan} &= [5 \ 6 \ 3 \ 7 \ 10 \ 19 \ 2 \ 8] \\
\text{segmented max-scan} &= [5 \ 5 \ 3 \ 4 \ 4 \ 9 \ 2 \ 6]
\end{align*}
\]
**Segmented Scan: Up-Sweep**

<table>
<thead>
<tr>
<th>DATA</th>
<th></th>
<th>DATA</th>
<th></th>
<th>FLAGs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 5 1 3 4 3 9 2 6 ]</td>
<td>[ 1 0 1 0 0 0 1 0 ]</td>
<td>[ 5 6 3 7 3 12 2 8 ]</td>
<td>[ 1 1 1 1 0 0 1 1 ]</td>
<td>[ 1 1 1 1 0 0 1 1 ]</td>
<td>[ 1 1 1 1 0 0 1 1 ]</td>
</tr>
<tr>
<td>[ 5 6 3 7 3 12 2 8 ]</td>
<td>[ 1 1 1 1 0 0 1 1 ]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ 5 6 3 7 3 12 2 8 ]</td>
<td>[ 1 1 1 1 0 0 1 1 ]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
d=0
\]

1: \textbf{for } d = 1 \textbf{ to } \log_2 n - 1 \textbf{ do}

2: \textbf{for all } k = 0 \textbf{ to } n - 1 \textbf{ by } 2^{d+1} \textbf{ in parallel do}

3: \textbf{if } f[k + 2^{d+1} - 1] \textbf{ is not set then}

4: \hspace{1cm} x[k + 2^{d+1} - 1] \leftarrow x[k + 2^d - 1] + x[k + 2^{d+1} - 1]

5: \hspace{1cm} f[k + 2^{d+1} - 1] \leftarrow f[k + 2^d - 1] \mid f[k + 2^{d+1} - 1]
Segmented Scan: Down-Sweep

1: \( x[n - 1] \leftarrow 0 \)
2: \( \text{for } d = \log_2 n - 1 \text{ down to } 0 \text{ do} \)
3: \( \text{for all } k = 0 \text{ to } n - 1 \text{ by } 2^{d+1} \text{ in parallel do} \)
4: \( t \leftarrow x[k + 2^d - 1] \)
5: \( x[k + 2^d - 1] \leftarrow x[k + 2^{d+1} - 1] \)
6: \( \text{if } f_i[k + 2^d] \text{ is set then} \)
7: \( x[k + 2^{d+1} - 1] \leftarrow 0 \) \hspace{1cm} \text{If my left most child is a head, then set me to zero}
8: \( \text{else if } f[k + 2^d - 1] \text{ is set then} \)
9: \( x[k + 2^{d+1} - 1] \leftarrow t \)
10: \( \text{else} \)
11: \( x[k + 2^{d+1} - 1] \leftarrow t + x[k + 2^{d+1} - 1] \)
12: \( \text{Unset flag } f[k + 2^d - 1] \)
Segmented Scan – Tree View

Up Sweep
[Green vertices are *flagged*]

Down Sweep
[Red vertices are *left subroots*]
GPU Specific Challenges

- Representation of the flags vector
  - Be as space-efficient as possible
  - Reduce shared memory bank conflicts
- Multi-block segmented scan
  - Not as simple as multi-block unsegmented scan

1: Perform reduce on all blocks in parallel
2: Save partial sum and partial OR trees to global memory
3: Do second-level segmented scans with final sums
4: Load partial sum and partial OR trees from global memory to shared memory
5: Set last element of each block to corresponding element in the output of second-level segmented scan
6: Perform down-sweep on all blocks in parallel
Primitives Built Atop Scan

- Split & Split-And-Segment [great for Radix-Sort]
  - Divide input vector into two pieces, with all the elements marked *false* (0) to the left side of the output, all elements marked *true* (1) to the right side.

  - \[[1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1]\]
  - \[\text{Invert them}\]
  - \[\text{f = prescan (find new false indices)}\]
  - \[\text{NF = 4}\]
  - \[\text{Total number of falses}\]
  - \[\text{Thread ids}\]
  - \[\text{t = id – f + NF}\]
Applications – Quicksort

- Challenge: How to keep track of recursion in quicksort without recursion?
- Answer: Split-and-segment!
  - Choose a pivot in each segment (say, the first element)
  - Distribute the pivot across the segment
  - Split-and-segment on the result of `input op? pivot`
  - Set new segment flags
  - Loop with the other comparison operator this time

- Alternating comparison operators: > and >=
## Quicksort Example

<table>
<thead>
<tr>
<th></th>
<th>Key</th>
<th>Flags</th>
<th>Pivot</th>
<th>&gt;=?</th>
<th>Split</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 8 3 1 7 6 9 3</td>
<td>1 0 0 0 0 0 0 0</td>
<td>4 4 4 4 4 4 4 4</td>
<td>0 1 0 0 1 1 1 0</td>
<td>4 3 1 3 8 7 6 9</td>
<td>1 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td></td>
<td>4 4 4 4 8 8 8 8</td>
<td>1 0 0 0 1 0 0 1</td>
<td>4 4 4 4 8 8 8 8</td>
<td>1 0 0 1 1 0 1 0</td>
<td>3 1 3 4 7 6 8 9</td>
<td>1 0 0 1 1 0 1 0</td>
</tr>
</tbody>
</table>
Applications: SpMV $(y \leftarrow y + Ax)$
(Sparse Matrix Times Dense Vector)

- CSR Representation of matrix A

<table>
<thead>
<tr>
<th>a</th>
<th>0</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(Col) Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 5 6</td>
<td></td>
</tr>
</tbody>
</table>

These are not real pointers, just integer indices to the col-index vector.
SpMV Execution

\[
\text{product} = [x_0a, x_2b, x_0c, x_1d, x_2e, x_2f] \quad (1)
\]
\[
= [[x_0a, x_2b][x_0c, x_1d, x_2e][x_2f]] \quad (2)
\]
\[
= [[x_0a + x_2b, x_2b]
\quad\quad [x_0c + x_1d + x_2e, x_1d + x_2e + x_2e][x_2f]] \quad (3)
\]
\[
y = y + [[x_0a + x_2b, x_0c + x_1d + x_2e, x_2e][x_2f]] \quad (4)
\]

1) Form the products
2) Examine row-pointers to set flags (one segment per row)
3) Perform backwards segmented-scan
4) Use the values on the segment heads to update y
Results - Primitives

- Segmented is 3x slower due to more reads from global memory (partial OR tree) and more computation to keep track of flags.
- Backwards is slower since data has been read to shared memory in reverse order – Why is this bad?
Results – Applications - Sort

- Sort is terrible!
- More than 10 times slower than radix sort.
- Even 4 times slower than CPU implementations of quick sort.
- Explanations for the phenomena is superficial.
- What’s missing?
  - Asymptotical parallel complexity analysis of their quicksort implementation
  - Performance comparison of sorts for different numbers of elements
Results – Applications – SpMV

- 2x slower than best CPU implementation
  - Tuned by OSKI (state of the art)
- Still promising if they can make backward segmented scans as fast as forward ones.
- What’s missing?
  - Comparison with existing formulations on GPU
  - They should have sat down and implemented the old algorithms in CUDA for fair comparison.
  - Remember: old algorithms are not based on “scan”
Conclusions

- For GPUs, vector primitives are much more important than scalar primitives.
  - GPUs are massively data parallel
  - Scans were even implemented on the CM
- CUDA is really productive.
- Great Paper 😊