A New Parallel Prefix-Scan Algorithm for GPUs

Sepideh Maleki*, Annie Yang, and Martin Burtscher

Department of Computer Science

TEXAS STATE UNIVERSITY

The rising STAR of Texas

Efficient Computing Laboratory
Highlights

- GPU-friendly algorithm for prefix scans called **SAM**

Novelties and features

- Higher-order support that is communication optimal
- Tuple-value support with constant workload per thread
- Carry propagation scheme with $O(1)$ auxiliary storage
- Implemented in unified 100-statement CUDA kernel

Results

- Outperforms CUB by up to **2.9-fold** on higher-order and
  by up to **2.6-fold** on tuple-based prefix sums
Prefix Sums

- Each value in the output sequence is the sum of all prior elements in the input sequence
  - Input
    
  - Output
    
- Can be computed efficiently in parallel

Applications

- Sorting, lexical analysis, polynomial evaluation, string comparison, stream compaction, & data compression
Data Compression

- Data compression algorithms
  - **Data model** predicts next value in input sequence and emits difference between actual and predicted value
  - **Coder** maps frequently occurring values to produce shorter output than infrequent values

- Delta encoding
  - Widely used data model
  - Computes **difference sequence** (i.e., predicts current value to be the same as previous value in sequence)
  - Used in image compression, speech compression, etc.
Delta Coding

- Delta **encoding** is embarrassingly parallel
- Delta **decoding** appears to be sequential
  - Decoded prior value needed to decode current value
- **Prefix sum** decodes delta encoded values
  - Decoding can also be done in parallel

<table>
<thead>
<tr>
<th>Input sequence</th>
<th>1, 2, 3, 4, 5, 2, 4, 6, 8, 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Difference sequence</td>
<td>1, 1, 1, 1, 1, -3, 2, 2, 2, 2</td>
</tr>
<tr>
<td>(encoding)</td>
<td></td>
</tr>
<tr>
<td>Prefix sum (decoding)</td>
<td>1, 2, 3, 4, 5, 2, 4, 6, 8, 10</td>
</tr>
</tbody>
</table>
Extensions of Delta Coding

- Higher orders
  - Higher-order predictions are often more accurate
    - First order \( \text{out}_k = \text{in}_k - \text{in}_{k-1} \)
    - Second order \( \text{out}_k = \text{in}_k - 2 \cdot \text{in}_{k-1} + \text{in}_{k-2} \)
    - Third order \( \text{out}_k = \text{in}_k - 3 \cdot \text{in}_{k-1} + 3 \cdot \text{in}_{k-2} - \text{in}_{k-3} \)

- Tuple values
  - Data frequently appear in tuples
    - Two-tuples \( x_0, y_0, x_1, y_1, x_2, y_2, ..., x_{n-1}, y_{n-1} \)
    - Three-tuples \( x_0, y_0, z_0, x_1, y_1, z_1, ..., x_{n-1}, y_{n-1}, z_{n-1} \)
Problem and Solution

- Conventional prefix sums are insufficient
  - Do not decode higher-order delta encodings
  - Do not decode tuple-based delta encodings

- Prior work
  - Requires inefficient workarounds to handle higher-order and tuple-based delta encodings

- SAM algorithm and implementation
  - Directly and efficiently supports these generalizations
  - Even supports combination of higher orders and tuples
Work Efficiency of Prefix Sums

- Sequential prefix sum requires only a single pass
  - $2n$ data movement through memory
  - Linear $O(n)$ complexity

- Parallel algorithm should have same complexity
  - $O(n)$ applications of the sum operator

```plaintext
1. out[0] = 0
2. for i from 1 to n do
3.   out[i] = out[i - 1] + in[i - 1]
```
Hierarchical Parallel Prefix Sum

1. Initial Array of Arbitrary Values
2. Break Array into Chunks
3. Compute Local Prefix Sums
4. Gather Top Most Values
5. Compute Prefix Sum
6. Add Resulting Carry \( i \) to all Values of Chunk \( i \)
7. Final Values

A New Parallel Prefix-Scan Algorithm for GPUs
Standard Prefix-Sum Implementation

- Based on 3-phase approach
- Reads and writes every element twice
  - $4n$ main-memory accesses
- Auxiliary array is stored in global memory
  - Calculation is performed across blocks
- High-performance implementations
  - Allocate and process several values per thread
- Thrust and CUDPP use this hierarchical approach
SAM Base Implementation

- Intra-block prefix sums
  - Computes prefix sum of each chunk conventionally
  - Writes *local sum* of each chunk to auxiliary array
  - Writes *ready flag* to second auxiliary array

- Inter-block prefix sums
  - Reads local sums of *all* prior chunks
  - Adds up local sums to calculate carry
  - Updates all values in chunk using carry
  - Writes final result to global memory
Pipelined Processing of Chunks

A New Parallel Prefix-Scan Algorithm for GPUs
Carry Propagation Scheme

- Persistent-block-based implementation
  - Same block processes every $k^{th}$ chunk
  - Carries require only $O(1)$ computation per chunk

- Circular-buffer-based implementation
  - Only $3k$ elements needed at any point in time
  - Local sums and ready flags require $O(1)$ storage

- Redundant computations for latency hiding
  - Write-followed-by-independent-reads pattern
  - Multiple values processed per thread (fewer chunks)
Higher-order Prefix Sums
Higher-order Prefix Sums

- Higher-order difference sequences can be computed by repeatedly applying first order

<table>
<thead>
<tr>
<th>Input values</th>
<th>1, 2, 3, 4, 5, 2, 4, 6, 8, 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>First order</td>
<td>1, 1, 1, 1, 1, -3, 2, 2, 2, 2</td>
</tr>
<tr>
<td>Second order</td>
<td>1, 0, 0, 0, 0, 0, -4, 5, 0, 0, 0</td>
</tr>
</tbody>
</table>

- Prefix sum is the inverse of order-1 differencing
  - $K$ prefix sums will decode an order-$k$ sequence
- No direct solution for computing higher orders
  - Must use iterative approach
  - Other codes’ memory accesses proportional to order
Higher-order Prefix Sums (cont.)

- SAM is more efficient
  - Internally iterates only the computation phase
  - Does not read and write data in each iteration
  - Requires only $2n$ main-memory accesses for any order

- SAM’s higher-order implementation
  - Does not require additional auxiliary arrays
    - Both sum array and ‘flag’ array are $O(1)$ circular buffers
  - Only needs non-Boolean ready ‘flags’
    - Uses counts to indicate iteration of current local sum
Tuple-based Prefix Sums
Tuple-based Prefix Sums

- Data may be tuple based \( x_0, y_0, x_1, y_1, ..., x_{n-1}, y_{n-1} \)
- Other codes have to handle tuples as follows
  - Reordering elements, compute, undo reordering
    - Slow due to reordering and may require extra memory
  - Defining a tuple data type as well as the plus operator
    - Slow for large tuples due to excessive register pressure
Tuple-based Prefix Sums (cont.)

- SAM is more efficient
  - No reordering
  - No special data types or overloaded operators
  - Always same amount of data per thread

- SAM’s tuple implementation
  - Employs multiple sum arrays, one per tuple element
    - Each sum array is an O(1) circular buffer
    - Uses modulo operations to determine which array to use
  - Still employs single O(1) flag array
Experimental Methodology

- Evaluate following prefix sum implementations
  - Thrust library (from CUDA Toolkit 7.5)
    - $4n$
  - CUDPP library 2.2
    - $4n$
  - CUB library 1.5.1
    - $2n$
  - SAM 1.1
    - $2n$

<table>
<thead>
<tr>
<th>GPU</th>
<th>GeForce Titan X</th>
<th>Tesla K40c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Maxwell</td>
<td>Kepler</td>
</tr>
<tr>
<td>PE</td>
<td>3072</td>
<td>2880</td>
</tr>
<tr>
<td>Multiprocessors</td>
<td>24</td>
<td>15</td>
</tr>
<tr>
<td>Persistent Blocks</td>
<td>48</td>
<td>30</td>
</tr>
<tr>
<td>Global Memory</td>
<td>12 GB</td>
<td>12 GB</td>
</tr>
<tr>
<td>Peak Bandwidth</td>
<td>336 GB/s</td>
<td>288 GB/s</td>
</tr>
</tbody>
</table>
Performance Evaluation
Prefix Sum Throughputs (Titan X)

- SAM and CUB outperform the other approaches ($2n$ vs. $4n$)
- For 64-bit values, throughputs are about half (but same GB/s)
- SAM matches cudaMemcpy throughput at high end (264 GB/s)
  - Surprising since SAM was designed for higher orders and tuples
A New Parallel Prefix-Scan Algorithm for GPUs

- K40 throughputs are lower for all algorithms
- SAM is faster than Thrust/CUDPP on medium and large inputs
- CUB outperforms SAM by 50% on large inputs on 32-bits ints
  - SAM’s implementation is not a particularly good fit for this older GPU
Higher-order Throughput (Titan X)

- Throughputs decrease as order increases due to more iterations
- SAM’s performance advantage increases with higher orders
  - Always executes $2^n$ global memory accesses
  - Outperforms CUB by 52% on order 2, 78% on order 5, and 87% on order 8
Higher-order Throughputs (K40)

- CUB outperforms SAM on orders 2 and 5, but not on order 8
  - Again, SAM’s relative performance increases with higher orders
- SAM’s relative performance over CUB is higher on 64-bit values
  - Baseline advantage of CUB over SAM is smaller for 64-bit values
Tuple-based Throughputs (Titan X)

- Throughputs decrease with larger tuple sizes due to extra work
- SAM’s performance advantage increases with larger tuple sizes
  - Larger tuples increase register pressure in CUB but not in SAM
  - SAM is 17% slower on 2-tuples but 20% faster on 5-tuples and 34% faster on 8-tuples
A New Parallel Prefix-Scan Algorithm for GPUs

- SAM outperforms CUB on 8-tuples (and larger tuples)
  - Again, SAM’s relative performance increases with larger tuple sizes
- The benefit of SAM over CUB is higher with 64-bit values
  - SAM already outperforms CUB on 5-tuples
Summary

- SAM directly supports prefix scans
  - Higher-order prefix scans
  - Tuple-based prefix scans
- SAM performance on Maxwell and Kepler GPUs
  - Reaches cudaMemcpy throughput on large inputs
  - Outperforms all alternatives by up to 2.9x on higher orders and by up to 2.6x on tuple-based prefix sums
- SAM’s relative performance increases with higher orders and larger tuple sizes
Question?

- Contact Info: Smaleki@txstate.edu

http://cs.txstate.edu/~burtscher/research/SAM/

- Acknowledgments
  - National Science Foundation
  - NVIDIA Corporation
  - Texas Advanced Computing Center