CS 380 - GPU and GPGPU Programming
Lecture 25: Parallel Prefix Sum / Scan

Markus Hadwiger, KAUST
Reading Assignment #14 (until Dec 2)

Read (required):

- CUDA Cooperative Groups (Volta + Turing)
  - https://devblogs.nvidia.com/cooperative-groups/

- Programming Tensor Cores in CUDA 10/9

- CUDA 10/9 Features Revealed: Volta, Turing, cooperative groups, and more

Read (optional):

- Warp-aggregated atomics

- Fast Histograms Using Shared Atomics on Maxwell
Semester Project Presentation Event

Thursday, Dec 12; time tbd
Presentation slots < 10 min
Parallel Prefix Sum (Scan)

• Definition:
The all-prefix-sums operation takes a binary associative operator \( \oplus \) with identity \( I \), and an array of \( n \) elements
\[
[a_0, a_1, \ldots, a_{n-1}]
\]
and returns the ordered set
\[
[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})].
\]

• Example:
if \( \oplus \) is addition, then scan on the set
\[
[3, 1, 7, 0, 4, 1, 6, 3]
\]
returns the set
\[
[0, 3, 4, 11, 11, 15, 16, 22]
\]

(From Blelloch, 1990, “Prefix Sums and Their Applications”)
Applications of Scan

- **Scan is a simple and useful parallel building block**
  - Convert recurrences from sequential:
    
    ```
    for (j=1; j<n; j++)
        out[j] = out[j-1] + f(j);
    ```

  - into parallel:
    
    ```
    forall (j) { temp[j] = f(j) }
    scan(out, temp);
    ```

- **Useful for many parallel algorithms:**
  - radix sort
  - quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Range Histograms
  - Etc.
Scan on the CPU

```c
void scan( float* scanned, float* input, int length)
{
    scanned[0] = 0;
    for(int i = 1; i < length; ++i)
    {
        scanned[i] = input[i-1] + scanned[i-1];
    }
}
```

- Just add each element to the sum of the elements before it
- Trivial, but sequential
- Exactly $n$ adds: optimal in terms of work efficiency
Prefix Sum Application
- Compaction -
Parallel Data Compaction

- Given an array of marked values

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>7</th>
<th>4</th>
<th>2</th>
<th>1</th>
<th>5</th>
<th>6</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Output the compacted list of marked values

  | 3 | 7 | 6 |
Using Prefix Sum

- Calculate prefix sum on index array

```
3 1 7 4 2 1 5 6 3 1
1 0 1 0 0 0 0 1 0 0
0 1 1 2 2 2 2 3 3 3
```

- For each marked value lookup the destination index in the prefix sum

```
3 1 7 4 2 1 5 6 3 1
1 0 1 0 0 0 0 1 0 0
0 1 1 2 2 2 2 3 3 3
```

- Parallel write to separate destination elements
Prefix Sum Application
- Range Histogram -
Range Histogram

- A histogram calculate the occurrence of each value in an array.
  \[ h[i] = |J| \quad J = \{j | v[j] = i \} \]

- Range query: number over elements in interval \([a, b]\).

- Slow answer:
  ```
  hrange = 0;
  for (i = a; i<=b; ++i)
      hrange += h[i];
  ```
Fast Range Histogram

- Compute prefix sum of histogram
- Fast answer:
  
  \[
  \text{hrange} = \text{pref}[B] - \text{pref}[A];
  \]

  \[
  = \sum_{0}^{B} h[i] - \sum_{0}^{A} h[i] = \sum_{A}^{B} h[i]
  \]
Prefix Sum Application
- Summed Area Tables -
Summed Area Tables

- Per texel, store sum from (0, 0) to (u, v)

- Many bits per texel (sum !)
- Evaluation of 2D integrals in constant time!

\[
\int\int_{B \times C} I(x, y) \, dx \, dy = A - B - C + D
\]
Summed Area Table with Prefix Sums

- One possible way:
  - Compute prefix sum horizontally

- ... then vertically on the result
In designing a parallel algorithm, it is more important to make it efficient than to make it asymptotically fast. The efficiency of an algorithm is determined by the total number of operations, or work that it performs. On a sequential machine, an algorithm's work is the same as its time. On a parallel machine, the work is simply the processor-time product. Hence, an algorithm that takes time \( t \) on a \( P \)-processor machine performs work \( W = Pt \). In either case, the work roughly captures the actual cost to perform the computation, assuming that the cost of a parallel machine is proportional to the number of processors in the machine.

We call an algorithm work-efficient (or just efficient) if it performs the same amount of work, to within a constant factor, as the fastest known sequential algorithm.

For example, a parallel algorithm that sorts \( n \) keys in \( O( \sqrt{n} \log(n) ) \) time using \( \sqrt{n} \) processors is efficient since the work, \( O( n \log(n) ) \), is as good as any (comparison-based) sequential algorithm.

However, a sorting algorithm that runs in \( O( \log(n) ) \) time using \( n^2 \) processors is not efficient.

The first algorithm is better than the second – even though it is slower – because it's work, or cost, is smaller. Of course, given two parallel algorithms that perform the same amount of work, the faster one is generally better.
Vector Reduction

Array elements

0 1 2 3 4 5 6 7 8 9 10 11

1 0+1 2+3 4+5 6+7 8+9 10+11

2 0...3 4..7 8..11

3 0..7 8..15

iterations

0..15
A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations, Kogge and Stone, 1973

See “carry lookahead” adders vs. “ripple carry” adders
**$O(n \log n)$ Scan**

<table>
<thead>
<tr>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$x_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma(x_0..x_0)$</td>
<td>$\Sigma(x_0..x_1)$</td>
<td>$\Sigma(x_1..x_2)$</td>
<td>$\Sigma(x_2..x_3)$</td>
<td>$\Sigma(x_3..x_4)$</td>
<td>$\Sigma(x_4..x_5)$</td>
<td>$\Sigma(x_5..x_6)$</td>
<td>$\Sigma(x_6..x_7)$</td>
</tr>
</tbody>
</table>

$d=1$

$\Sigma(x_0..x_0)$ | $\Sigma(x_0..x_1)$ | $\Sigma(x_0..x_2)$ | $\Sigma(x_0..x_3)$ | $\Sigma(x_1..x_4)$ | $\Sigma(x_2..x_5)$ | $\Sigma(x_3..x_6)$ | $\Sigma(x_4..x_7)$ |

$d=2$

$\Sigma(x_0..x_0)$ | $\Sigma(x_0..x_1)$ | $\Sigma(x_0..x_2)$ | $\Sigma(x_0..x_3)$ | $\Sigma(x_0..x_4)$ | $\Sigma(x_0..x_5)$ | $\Sigma(x_0..x_6)$ | $\Sigma(x_0..x_7)$ |

$d=3$

- Step efficient ($\log n$ steps)
- Not work efficient ($n \log n$ work)
- Requires barriers at each step (WAR dependencies)
Hillis-Steele Scan Implementation

No WAR conflicts, $O(2N)$ storage
A First-Attempt Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

Each thread reads one value from the input array in device memory into shared memory array T0. Thread 0 writes 0 into shared memory array.
A First-Attempt Parallel Scan Algorithm

1. (previous slide)

2. Iterate log(n) times: Threads \textit{stride} to \textit{n}: Add pairs of elements \textit{stride} elements apart. Double \textit{stride} at each iteration. (note must double buffer shared mem arrays)

Iteration #1
Stride = 1

- Active threads: \textit{stride} to \textit{n}-1 (\textit{n-stride} threads)
- Thread \textit{j} adds elements \textit{j} and \textit{j-stride} from T0 and writes result into shared memory buffer T1 (ping-pong)
A First-Attempt Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate \( \log(n) \) times: Threads \textit{stride} to \( n \): Add pairs of elements \textit{stride} elements apart. Double \textit{stride} at each iteration. (note must double buffer shared mem arrays)

Iteration #2
Stride = 2
A First-Attempt Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate \( \log(n) \) times: Threads stride to \( n \): Add pairs of elements stride elements apart. Double stride at each iteration. (note must double buffer shared mem arrays)

Iteration #3
Stride = 4
A First-Attempt Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate log(n) times: Threads stride to n: Add pairs of elements stride elements apart. Double stride at each iteration. (note must double buffer shared mem arrays)

3. Write output to device memory.
Work Efficiency Considerations

- The first-attempt Scan executes $\log(n)$ parallel iterations
  - Total adds: $n \times (\log(n) - 1) + 1 \rightarrow O(n \times \log(n))$ work

- This scan algorithm is not very work efficient
  - Sequential scan algorithm does $n$ adds
  - A factor of $\log(n)$ hurts: 20x for $10^6$ elements!

- A parallel algorithm can be slow when execution resources are saturated due to low work efficiency
Balanced Trees

• For improving efficiency
• A common parallel algorithm pattern:
  – Build a balanced binary tree on the input data and sweep it to and from the root
  – Tree is not an actual data structure, but a concept to determine what each thread does at each step

• For scan:
  – Traverse down from leaves to root building partial sums at internal nodes in the tree
    • Root holds sum of all leaves
  – Traverse back up the tree building the scan from the partial sums
Typical Parallel Programming Pattern

- 2 log(n) steps
Typical Parallel Programming Pattern

- 2 log(n) steps
A Regular Layout for Parallel Adders, Brent and Kung, 1982
\( O(n) \) Scan [Bleloch]

- Work efficient (\( O(n) \) work)
- Bank conflicts, and lots of ‘em

Courtesy John Owens
Build the Sum Tree

Assume array is already in shared memory
Build the Sum Tree

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

Stride 2

Iteration 1, \( n/2 \) threads

Each \( + \) corresponds to a single thread.

Iterate \( \log(n) \) times. Each thread adds value \( \text{stride} / 2 \) elements away to its own value.
Build the Sum Tree

Stride 2

Stride 4

Iteration 2, \( n/4 \) threads

Each \( \oplus \) corresponds to a single thread.

Iterate \( \log(n) \) times. Each thread adds value \( \text{stride} / 2 \) elements away to its own value.
Build the Sum Tree

Iterate $\log(n)$ times. Each thread adds value $\text{stride} / 2$ elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering.
Down-Sweep Variant 1: Exclusive Scan

We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.
Build Scan From Partial Sums
Build Scan From Partial Sums

<table>
<thead>
<tr>
<th>T</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>11</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
</table>

Stride 8

<table>
<thead>
<tr>
<th>T</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>0</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>11</th>
</tr>
</thead>
</table>

Iteration 1
1 thread

Each ⦿ corresponds to a single thread.

Iterate log(n) times. Each thread adds value \( \text{stride} / 2 \) elements away to its own value and sets the value \( \text{stride} \) elements away to its own previous value.
Build Scan From Partial Sums

Stride 8

Stride 4

Iteration 2
2 threads

Iterate log(n) times. Each thread adds value \( \text{stride} / 2 \) elements away to its own value and sets the value \( \text{stride} / 2 \) elements away to its own previous value.

Each \( + \) corresponds to a single thread.
Build Scan From Partial Sums

Done! We now have a completed scan that we can write out to device memory.

Total steps: $2 \times \log(n)$.
Total work: $2 \times (n-1)$ adds = $O(n)$  Work Efficient!
Down-Sweep Variant 2: Inclusive Scan

We now have an array of partial sums. Let’s propagate the sums back.
Build Scan From Partial Sums

Stride 8

\[ \begin{array}{cccccccc}
T & 3 & 4 & 7 & 11 & 4 & 5 & 6 & 25 \\
\end{array} \]

no operation

\[ \begin{array}{cccccccc}
T & 3 & 4 & 7 & 11 & 4 & 5 & 6 & 25 \\
\end{array} \]

Each \( \oplus \) corresponds to a single thread.

Iterate \( \log(n) \) times. Each thread adds value \( \text{stride} / 2 \) elements away to its own value. First element adds zero.
Build Scan From Partial Sums

Iterate $\log(n)$ times. Each thread adds value $\text{stride} / 2$ elements away to its own value.
First element adds zero.

Each $\oplus$ corresponds to a single thread.
Build Scan From Partial Sums

Done! We now have a completed scan that we can write out to device memory.

Total steps: $2 \times \log(n)$.
Total work: $< 2 \times (n-1)$ adds = $O(n)$  Work Efficient!
Application to Large Arrays

[Diagram showing the process of applying the algorithm to large arrays, with steps including scanning blocks, storing block sums, and adding scanned block sums to the next block.]
Scan papers


- Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, GPU Gems 3, chapter 39, pages 851–876. Addison Wesley, August 2007.


Thank you.

- NVIDIA