Announcements

- Presentation topics due 02/07
- Homework 2 due 02/13

Agenda

- Finish atomic functions from Monday
- Parallel Algorithms
  - Parallel Reduction
  - Scan
  - Stream Compression
  - Summed Area Tables

Parallel Reduction

- Given an array of numbers, design a parallel algorithm to find the sum.
- Consider:
  - *Arithmetic intensity*: compute to memory access ratio
Parallel Reduction

- Given an array of numbers, design a parallel algorithm to find:
  - The sum
  - The maximum value
  - The product of values
  - The average value
- How different are these algorithms?

Parallel Reduction

- Reduction: An operation that computes a single result from a set of data
- Examples:
  - Minimum/maximum value
  - Average, sum, product, etc.
- Parallel Reduction: Do it in parallel. Obviously

Example. Find the sum:

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

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

```
1  5  9  13
```
Parallel Reduction

- Similar to brackets for a basketball tournament
- log(n) passes for n elements

All-Prefix-Sums

- **All-Prefix-Sums**
  - **Input**
    - Array of n elements: \([a_0, a_1, ..., a_{n-1}]\)
    - Binary associate operator: \(\oplus\)
    - Identity: \(I\)
  - **Outputs the array**:
    - \([I, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, ..., a_0 \oplus a_1 \oplus ... \oplus a_{n-1}]\)
All-Prefix-Sums

- Example
  - If $\oplus$ is addition, the array $[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]$ is transformed to $[0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22]$
  - Seems sequential, but there is an efficient parallel solution

Scan

- **Scan**: all-prefix-sums operation on an array of data
- **Exclusive Scan**: Element $j$ of the result does not include element $j$ of the input:
  - In: $[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]$  
  - Out: $[0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22]$
- **Inclusive Scan (Prescan)**: All elements including $j$ are summed
  - In: $[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]$  
  - Out: $[3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22 \ 25]$

Scan

- How do you generate an exclusive scan from an inclusive scan?
  - Input: $[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]$  
  - Inclusive: $[3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22 \ 25]$  
  - Exclusive: $[0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22]$
  - // Shift right, insert identity

- How do you go in the opposite direction?

Scan

- **Use cases**
  - **Stream compaction**
  - **Summed-area tables** for variable width image processing
  - **Radix sort**
  - ...
Scan

- Used to convert certain sequential computation into equivalent parallel computation

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: \text{seq}(e) = e</td>
<td>01: \text{seq}(e) = e</td>
</tr>
<tr>
<td>02: \text{seq}(e) = \text{seq}(e) \cdot \text{seq}(e)</td>
<td>02: \text{seq}(e) = \text{seq}(e) \cdot \text{seq}(e)</td>
</tr>
<tr>
<td>03: \text{seq}(e) = \text{seq}(e) \cdot \text{seq}(e)</td>
<td>03: \text{seq}(e) = \text{seq}(e) \cdot \text{seq}(e)</td>
</tr>
</tbody>
</table>

Design a parallel algorithm for exclusive scan

- In: [3 1 7 0 4 1 6 3]
- Out: [0 3 4 11 11 15 16 22]

Consider:
- Total number of additions

Sequential Scan: single thread, trivial

- \( n \) adds for an array of length \( n \)
- Work complexity: \( O(n) \)
- How many adds will our parallel version have?

Naive Parallel Scan

- Is this exclusive or inclusive?
- Each thread
  - Writes one sum
  - Reads two values


Naive Parallel Scan: Input

Naive Parallel Scan: \( d = 1, 2^{d-1} = 1 \)

for \( d = 1 \) to \( \log_2 n \)
for all \( k \) in parallel
if \( k \geq 2^{d-1} \)
\( x[k] = x[k - 2^{d-1}] + x[k] \)
**Naive Parallel Scan**: $d = 1, 2^{d-1} = 1$

For $d = 1$ to $\log_2 n$
for all $k$ in parallel
if ($k \geq 2^{d-1}$)
    $x[k] = x[k-2^{d-1}] + x[k]$;
Scan

- **Naive Parallel Scan**: $d = 1$, $2^{d-1} = 1$

  - Recall, it runs in parallel!

  
  for $d = 1$ to $\log_2 n$
  
  for all $k$ in parallel
  
  if ($k \geq 2^{d-1}$)
  
  $a(k) = a(k - 2^{d-1}) + a(k)$;
Scan

**Naive Parallel Scan: d = 2, 2^{d-1} = 2**

Consider only $k = 7$

for $d = 1$ to $\log_2 n$
for all $k$ in parallel
if ($k \geq 2^{d-1}$)
   $x[k] = x[k - 2^{d-1}] + x[k]$;

Scan

**Naive Parallel Scan: d = 3, 2^{d-1} = 4**

Consider only $k = 7$

for $d = 1$ to $\log_2 n$
for all $k$ in parallel
if ($k \geq 2^{d-1}$)
   $x[k] = x[k - 2^{d-1}] + x[k]$;
Scan

- **Naive Parallel Scan**: Final

```
0 1 2 3 4 5 6 7
0 1 3 5 7 9 11 13
0 1 3 6 10 14 18 22
0 1 3 6 10 15 21 28
```

- **Naive Parallel Scan**
  - What is naive about this algorithm?
    - What was the work complexity for sequential scan?
    - What is the work complexity for this?

---

Stream Compaction

- **Stream Compaction**
  - Given an array of elements
    - Create a new array with elements that meet a certain criteria, e.g. non null
    - Preserve order

```
a b c d e f g h
```

---

Stream Compaction

- **Stream Compaction**
  - Given an array of elements
    - Create a new array with elements that meet a certain criteria, e.g. non null
    - Preserve order

```
a b c d e f g h
```

```
a c d g
```
Stream Compaction

- Stream Compaction
  - Used in collision detection, sparse matrix compression, etc.
  - Can reduce bandwidth from GPU to CPU

Step 1: Compute temporary array containing
- 1 if corresponding element meets criteria
- 0 if element does not meet criteria
Stream Compaction

- Stream Compaction
  - Step 1: Compute temporary array

\[
\begin{array}{cccccccc}
  a & b & c & d & e & f & g & h \\
  l & 0 & 1 & 0 & 1 & 0 & 1 & l \\
\end{array}
\]
Stream Compaction

- Stream Compaction
  - Step 1: Compute temporary array

  \[
  \begin{array}{cccccccc}
  a & b & c & d & e & f & g & h \\
  \hline
  1 & 0 & 1 & 1 & 0 & 0 & 1 & 1
  \end{array}
  \]

  It runs in parallel!
Stream Compaction

**Step 2:** Run exclusive scan on temporary array

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

Scan result:

- Scan runs in parallel
- What can we do with the results?

**Step 3:** Scatter

Result of scan is index into final array
- Only write an element if temporary array has a 1

Final array:

| 0 | 1 | 2 | 3 |
Stream Compaction

Step 3: Scatter

Scan result:

Final array:

Stream Compaction

Step 3: Scatter

Scan result:

Final array:
Stream Compaction

- **Step 3**: Scatter

```
  a b c d e f g h
  1 0 1 1 0 0 1 0
```

Scan result:
```
  0 1 1 2 3 3 3 4
```

Final array:
```
  0 1 2 3
```

Scatter runs in parallel!

Summed Area Table

- **Summed Area Table (SAT)**: 2D table where each element stores the sum of all elements in an input image between the lower left corner and the entry location.

```
Input image:

   2 1 0 0
   0 1 2 0
   1 2 1 0
   1 1 0 2

SAT:

   4 9 12 14
   2 6 9 11
   2 5 6 8
   1 2 2 4
```

Example:

\[(1 + 1 + 0) + (1 + 2 + 1) + (0 + 1 + 2) = 9\]
Summed Area Table

- Benefit
  - Used to perform different width filters at every pixel in the image in constant time per pixel
  - Just sample four pixels in SAT:

\[
s_{\text{filter}} = \frac{s_{ur} - s_{ul} - s_{br} + s_{bl}}{w \times h},
\]

- Uses
  - Glossy environment reflections and refractions
  - Approximate depth of field

Summed Area Table

Input image

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

SAT

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summed Area Table

Input image

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

SAT

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Summed Area Table

Input image

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

SAT

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Summed Area Table

Input image

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

SAT

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Summed Area Table

Input image

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

SAT

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

...
Summed Area Table

Input image

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

SAT

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

How would implement this on the GPU?

How would compute a SAT on the GPU using inclusive scan?

Step 1 of 2:

Input image

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Partial SAT

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

One inclusive scan for each row
Summed Area Table

- Step 2 of 2:

<table>
<thead>
<tr>
<th>Partial SAT</th>
<th>Final SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 3 3 3</td>
<td>4 9 12 14</td>
</tr>
<tr>
<td>0 1 3 3 3</td>
<td>2 6 9 11</td>
</tr>
<tr>
<td>1 3 4 4</td>
<td>2 5 6 8</td>
</tr>
<tr>
<td>1 2 2 2 4</td>
<td>1 2 2 4</td>
</tr>
</tbody>
</table>

One inclusive scan for each column, bottom to top

Summary

- Parallel reductions and scan are building blocks for many algorithms
- An understanding of parallel programming and GPU architecture yields efficient GPU implementations