Activity
- Given a set of numbers, design a GPU algorithm for:
  - Team 1: Sum of values
  - Team 2: Maximum value
  - Team 3: Product of values
  - Team 4: Average value
- Consider:
  - Bottlenecks
  - **Arithmetic intensity**: compute to memory access ratio
  - Optimizations
  - Limitations

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

Parallel Reduction
- Store data in 2D texture
- Render viewport-aligned quad ¼ of texture size
  - Texture coordinates address every other texel
  - Fragment shader computes operation with four texture reads for surrounding data
- Use output as input to the next pass
- Repeat until done

uniform sampler2D u_Data;
in vec2 fs_Texcoords;
out float out_MaxValue;

void main(void) {
  float v0 = texture(u_Data, fs_Texcoords).r;
  float v1 = textureOffset(u_Data, fs_Texcoords, ivec2(0, 1)).r;
  float v2 = textureOffset(u_Data, fs_Texcoords, ivec2(1, 0)).r;
  float v3 = textureOffset(u_Data, fs_Texcoords, ivec2(1, 1)).r;
  out_MaxValue = max(max(v0, v1), max(v2, v3));
}

Parallel Reduction

- Reduces $n^2$ elements in log(n) time
- 1024x1024 is only 10 passes

**Bottlenecks**
- Read back to CPU, recall `glReadPixels`
- Each pass depends on the previous
  - How does this affect pipeline utilization?
- Low arithmetic intensity
Parallel Reduction

- **Optimizations**
  - Use just red channel or rgba?
  - Read 2x2 areas or nxn? What is the trade off?
  - When do you read back? 1x1?
  - How many textures are needed?

- **Ping Ponging**
  - Use two textures: X and Y
  - First pass: X is input, Y is output
  - Additional passes swap input and output
  - Implement with FBOs

```
Input X Y
Output Y X
```

Parallel Reduction

- **Limitations**
  - Maximum texture size
  - Requires a power of two in each dimension
  - How do you work around these?

All-Prefix-Sums

- **All-Prefix-Sums**
  - Input
    - Array of n elements: \([a_1, a_2, \ldots, a_n]\)
    - Binary associate operator: \(@\)
    - Identity: \(I\)
  - Outputs the array: \([a_1, a_1 @ a_2, a_1 @ a_2 @ a_3, \ldots, a_1 @ a_2 @ \ldots @ a_n]\)

All-Prefix-Sums

- **Example**
  - If \(@\) 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: \([3 4 11 11 15 16 22 25]\)
How do you generate an exclusive scan from an inclusive scan?

- **In:** \[3 1 7 0 4 1 6 3\]
- **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: 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

---

**Scan: 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

---

**Scan: Stream Compaction**

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

- Stream Compaction
  - Step 1: Compute temporary array

\[ a \ b \ c \ d \ e \ f \ g \ h \ 1 \ 0 \]

Scan: Stream Compaction

- Stream Compaction
  - Step 1: Compute temporary array

\[ a \ b \ c \ d \ e \ f \ g \ h \ 1 \ 0 \ 1 \]

Scan: Stream Compaction

- Stream Compaction
  - Step 1: Compute temporary array

\[ a \ b \ c \ d \ e \ f \ g \ h \ 1 \ 0 \ 1 \ 0 \]

Scan: Stream Compaction

- Stream Compaction
  - Step 1: Compute temporary array

\[ a \ b \ c \ d \ e \ f \ g \ h \ 1 \ 1 \ 0 \]
Scan: Stream Compaction

- Stream Compaction
  - **Step 1**: Compute temporary array

```
1 0 0 1 0 0 1
1 0 0 1 0 0 1
```

- It runs in parallel!

Scan: Stream Compaction

- Stream Compaction
  - **Step 1**: Compute temporary array

```
1 0 0 1 0 0 1
1 0 0 1 0 0 1
```

- It runs in parallel!

Scan: Stream Compaction

- Stream Compaction
  - **Step 2**: Run exclusive scan on temporary array

```
0 1 1 1 2 2 2
0 1 1 1 2 2 2
```

Scan result:

Scan runs in parallel

What can we do with the results?
Stream Compaction

**Step 3: Scatter**

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

Scan result:

<table>
<thead>
<tr>
<th></th>
<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></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Final array:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Scan: Stream Compaction**

- **Stream Compaction**
  - **Step 3:** Scatter

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

  Scan result: 0 1 2 3 4 5 6 7 8 9

  Final array: 0 1 2 3 4 5 6 7 8 9  
  - Scatter runs in parallel!

---

**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></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Activity**

- 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
  - Ignore GLSL constraints

---

**Scan**

- **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?

---

**Scan**

- **Naive Parallel Scan**

  - for $d = 1$ to $\log_2 n$
    - for all $k$ in parallel
      - if ($k >= 2^d$)
        - $x(k) = x(k - 2^d) + x(k)$

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

- **Naive Parallel Scan**: Input

0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

for \( d = 1 \) to \( \log_2 n \)

for all \( k \) in parallel

if \( k \geq 2d-1 \)

\[ x[k] = x[k - 2d + 1] + x[k]; \]

Scan

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

0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

for \( d = 1 \) to \( \log_2 n \)

for all \( k \) in parallel

if \( k \geq 2d-1 \)

\[ x[k] = x[k - 2d + 1] + x[k]; \]

Scan

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

0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

for \( d = 1 \) to \( \log_2 n \)

for all \( k \) in parallel

if \( k \geq 2d-1 \)

\[ x[k] = x[k - 2d + 1] + x[k]; \]

Scan

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

0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

for \( d = 1 \) to \( \log_2 n \)

for all \( k \) in parallel

if \( k \geq 2d-1 \)

\[ x[k] = x[k - 2d + 1] + x[k]; \]

Scan

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

0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

for \( d = 1 \) to \( \log_2 n \)

for all \( k \) in parallel

if \( k \geq 2d-1 \)

\[ x[k] = x[k - 2d + 1] + x[k]; \]
Recall, it runs in parallel!
Consider only $k = 7$ for $d = 1$ to log$_2 n$
  for all $k$ in parallel
  if ($k \geq 2^{d-1}$)
  $x[k] = x[k - 2^{d-1}] + x[k]$;

What is naive about this algorithm?

What was the work complexity for sequential scan?

What is the work complexity for this?
Summary

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