GPGPU: Parallel Reduction and Scan

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2011

Administrivia
- Assignment 3 due Wednesday
  - 11:59pm on Blackboard
- Assignment 4 handed out Monday, 02/14
- Final
  - Wednesday 05/04, 12:00-2:00pm
  - Review session probably Friday 04/29

Agenda

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

---

```cpp
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

- 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
Parallel Reduction

```cpp
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));
}
```

---

Read four values in a 2x2 area of the texture

Read four values in a 2x2 area of the texture

---

Output the maximum value

Output the maximum value

---

Parallel Reduction

```cpp
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));
}
```

---

Only using the red channel.

How efficient is this?

Only using the red channel.

How efficient is this?
Parallel Reduction

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

- 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?

---

Parallel Reduction

- 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

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
<td>X</td>
</tr>
</tbody>
</table>

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

- **Input**
  - Array of \( n \) elements: \( a_0, a_1, \ldots, a_{n-1} \)
  - Binary associate operator: \( \oplus \)
  - Identity: \( I \)

- **Outputs the array**
  \[ [a_0 \oplus 0, a_0 \oplus a_1, \ldots, a_0 \oplus a_{n-1}] \]

**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: \( [3, 4, 11, 11, 15, 16, 22, 25] \)

**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

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

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

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

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

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

Scan: Stream Compaction

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

```plaintext
a b c d e f g h
```
Scan: Stream Compaction

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

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

Scan: Stream Compaction

- Stream Compaction
  - Step 1: Compute temporary array

```
  a  b  c  d  e  f  g  h
  1  0  1  0  0  1  0  0
  ```
Scan: Stream Compaction

- Stream Compaction
  - Step 1: Compute temporary array

```
  a b c d e f g h
  1 0 1 1 0 0 0 1
```
Scan: Stream Compaction

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

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

- It runs in parallel!

Scan: Stream Compaction

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

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

- It runs in parallel!

Scan: Stream Compaction

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

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

- It runs in parallel!

Scan: Stream Compaction

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

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

- Scan result:
Scan: Stream Compaction

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

Scan result:

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

Scan result:

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

**Step 3:** Scatter

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

Scan result:

```
Final array: a
```

Scan: Stream Compaction

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

Scan result:

```
Final array: a
```
Scan: Stream Compaction

- Stream Compaction
  - Step 3: Scatter

<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:

| 0 | 1 | 1 | 2 | 3 | 3 | 3 | 4 |

Final array:

<table>
<thead>
<tr>
<th>a</th>
<th>C</th>
<th>d</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Scan result:

| 0 | 1 | 1 | 2 | 3 | 3 | 3 | 4 |

Final array:

<table>
<thead>
<tr>
<th>a</th>
<th>C</th>
<th>d</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Scatter runs in parallel!
Scan: Stream Compaction

Stream Compaction

**Step 3:** Scatter

```
<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:

```
| 0 | 1 | 2 | 3 |
```

Final array:

```
| a | c | d | g |
```

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>
</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?
**Naive Parallel Scan**

For $d = 1$ to $\log_2 n$

For all $k$ in parallel

if $(k \geq 2^{d-1})$

$x[k] = x[k - 2^{d-1}] + x[k];$

Is this exclusive or inclusive?

Each thread

- Writes one sum
- Reads two values

---

**Naive Parallel Scan: Input**

0 1 2 3 4 5 6 7

---

**Scan**

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

0 1 2 3 4 5 6 7

---

**Scan**

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

0 1 2 3 4 5 6 7
Scan

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

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

Scan

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

For \( d = 1 \), \( 2^{d-1} = 1 \)

<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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
</tbody>
</table>

Recall, it runs in parallel!
Scan

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

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

  after $d = 1$

  Consider only $k = 7$

  after $d = 2$

Scan

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

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

  after $d = 1$

  0 1 3 5 7 9 11 13

  after $d = 1$

  0 1 3 5 7 9 11 13

  after $d = 2$

Scan

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

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

  after $d = 1$

  0 1 3 5 7 9 11 13

  after $d = 1$

  0 1 3 5 7 9 11 13

  after $d = 2$
Scan

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

Consider only $k = 7$

Scan

- Naive Parallel Scan: Final

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