Efficient Stream Reduction on the GPU

David Roger, Ulf Assarsson, Nicolas Holzschuch

Grenoble University  Chalmers University of Technology  Cornell University
Stream Reduction

Removing unwanted elements from a stream

Input stream

Reduced stream
Applications

• Tree traversal:
  – Ray tracing
  – Collision detection

• Often the bottleneck
Sequential Algorithm

- Algorithm:
  
  ```
  i=0
  for j=0 to n-1 do
    if x[j] is valid then
      x[i]=x[j]
      i=i+1
  ```

- Easy: one single loop

- Linear complexity
On GPU

- Parallelism

- We assume no scatter
  - We will speak about scatter later
Talk Structure

● Previous Works
● Algorithm Overview
● Details and Implementation
● Results
● Future Works & Conclusion
Previous works: Horn's Method

Input stream

Prefix sum scan: computes the displacements

Prefix sum

Dichotomic search: performs the displacements

Reduced stream
Previous works

- **Prefix sum scan**
  - Hillis and Steele, Horn: $O(n \log n)$
  - Blelloch, Sengupta *et al.*, Harris *et al.*: $O(n)$
  - Sengupta *et al.* Hybrid: $O(n)$

- **Dichotomitic search**: $O(n \log n)$

- **Overall complexity**: $O(n \log n)$
Other approaches

- Geometry shader + stream output
  - NV_transform_feedback
  - Input stream: vertices in a VBO
  - Geometry shader discards NULL elements
  - Output stream: vertices in a VBO
    - No fragments, no fragment shader

- Bitonic sort
  - Slow

- Sum scan + Scatter with vertex engine
Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Future Works & Conclusion
Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Future Works & Conclusion
Our approach

Input stream, split in blocks

Reduction of the blocks

Concatenation

Reduced stream
Reduction of the blocks

• In parallel

• Using previous works
  – Prefix sum scan
  – Dichotomtic search

• Complexity
  – $s$: size of a block
  – One block: $O(s \log s)$
  – $n/s$ blocks: $O(n \log s)$
Concatenation of the blocks

• **Prefix sum scan**
  - Computes displacements of the blocks in parallel

• **Line drawing**
  - Segments extremities moved by scattering (vertex engine)
  - Other elements linearly interpolated (rasterization)

• **Complexity:** $O(n)
Concatenation of the blocks

Reduced blocks

Reduced stream
Concatenation of the blocks

Reduced blocks

Reduced stream

Move the extremities with the vertex shader
Concatenation of the blocks

Reduced blocks

Move the extremities with the vertex shader

Rasterization
Algorithmic complexity

- All previous works: $O(n \log n)$
- Our algorithm: $O(n \log s)$
  - $s$ is the size of the blocks
  - $s$ is a constant!
Overview

Input stream, split in blocks

Prefix sum scan + Dichotomic search

Prefix sum scan + Line drawing

Reduced stream
Why is it efficient?

The key is block concatenation:

- Dichotomic search is avoided
- Vertex engine: scatter ... but lesser efficiency
  - Use it for a few elements (segment extremities)
  - Interpolate the other elements
Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Future Works & Conclusion
Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Future Works & Conclusion
Overview

Input stream, split in blocks

Prefix sum scan + Dichotomic search

Prefix sum scan + Line drawing

Reduced stream
Overview

Input stream, split in blocks

Prefix sum scan + Dichotomic search

Prefix sum scan + Line drawing

Reduced stream
**Dichotomistic search details**

**Input block**

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

**Prefix sum**

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

\[ \text{sum}[j] = 3 \]

**Reduced block**

```
0 1 2 3 4 ? 6 7 8 9 10 11 12 13 14 15
```

**Gather:**
At output position \( i \)

**Search** \( j \) in input such as:

\[ i = j - \text{sum}[j] \]

**Search bounds:**

\[ i + \text{sum}[i] \leq j \leq i + \text{sum}[15] \]

**Example:**

- \( i = 5 \)
- \( 6 \leq j \leq 11 \)
- Search result \( j = 8 \)
Dichotomic search pseudo-code

Search \( j_0 \) such as \( i = j_0 - \text{sum}[j_0] \):

\[
\begin{align*}
\text{lowBound} & = i + \text{sum}[i] \\
\text{upBound} & = i + \text{sum}[n-1] \\
\text{if}(\text{upBound} > n-1) & \text{ discard} \\
\text{j} & = (\text{lowBound} + \text{upBound})/2 \\
\text{found} & = \text{j} - \text{sum}[\text{j}] - i
\end{align*}
\]

\[
\begin{align*}
\text{while}(\text{found} \neq 0) & \{
\text{if} (\text{found} < 0) \text{ lowBound} = \text{j} \\
\text{else} & \text{ upBound} = \text{j} \\
\text{j} & = (\text{lowBound} + \text{upBound})/2 \\
\text{found} & = \text{j} - \text{sum}[\text{j}] - i
\} 
\end{align*}
\]
Dichotomic search improvement

Search $j_0$ such as $i = j_0 - \text{sum}[j_0]$:

lowBound = $i + \text{sum}[i]$

upBound = $i + \text{sum}[n-1]$

if(upBound > n-1) discard

j = (lowBound + upBound)/2

found = $j - \text{sum}[j] - i$

while(found $\neq 0$) {

if (found < 0) lowBound = $j - \text{found}$
else upBound = $j - \text{found}$

j = (lowBound + upBound) / 2

found = $j - \text{sum}[j] - i$

}

Because $j - \text{sum}[j]$ is contracting!
Overview

Input stream, split in blocks

Prefix sum scan +
Dichotomic search

Prefix sum scan +
Line drawing

Reduced stream
Overview

Input stream, split in blocks

Prefix sum scan + Dichotomic search

Prefix sum scan + Line drawing

Reduced stream
Lines wrapping

- We use 2D textures: wrap line segments
  - Split all segments in two
    - Or
  - Use geometry engine to split only when necessary

[Diagram showing concatenation]
Lines wrapping

- We use 2D textures: wrap line segments
  - Split all segments in two
    - Or
  - Use geometry engine to split only when necessary
Lines wrapping

- We use 2D textures: wrap line segments
  - Split all segments in two
    - Or
  - Use geometry engine to split only when necessary

Concatenation
Lines wrapping

- We use 2D textures: wrap line segments
  - Split all segments in two
    - Or
  - Use geometry engine to split only when necessary

Concatenation
Lines wrapping

• We use 2D textures: wrap line segments
  – Split all segments in two
    • Or
  – Use geometry engine to split only when necessary
Lines wrapping

- We use 2D textures: wrap line segments
  - Split all segments in two
    - Or
  - Use geometry engine to split only when necessary

[Diagram of concatenation]
Lines wrapping

- We use 2D textures: wrap line segments
  - Split all segments in two
    - Or
  - Use geometry engine to split only when necessary

<table>
<thead>
<tr>
<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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Concatenation
Lines wrapping

- We use 2D textures: wrap line segments
  - Split all segments in two
    - Or
  - Use geometry engine to split only when necessary
Lines wrapping

- We use 2D textures: wrap line segments
  - Split all segments in two
    - Or
  - Use geometry engine to split only when necessary
Lines wrapping

- We use 2D textures: wrap line segments
  - Split all segments in two
    - Or
  - Use geometry engine to split only when necessary

Concatenation
Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Future Works & Conclusion
Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Future Works & Conclusion
Behavior: linear complexity
Behavior: block size

![Graph showing time vs. block size for 1M and 4M elements.](image-url)
Behavior: fill ratio

![Graph showing the behavior of fill ratio for different block sizes. The graph plots time (ns) against the number of valid elements (%). Two lines are shown: one for block size 512 and another for block size 64. The graph indicates that as the number of valid elements increases, the time also increases, reaching a peak and then decreasing.]
Comparison with previous works

![Graph comparing time (ns) vs. stream elements (M) for different methods: Horn, Blelloch, Geometric shaders, Blelloch + Vertex engine scatter, and Our algorithm.]
Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Future Works & Conclusion
Talk Structure

• Previous Works
• Algorithm Overview
• Details and Implementation
• Results

• Future Works & Conclusion
Scatter? (future work)

- Scatter available in CUDA
- Possible improvements
Scatter? (future work)

Input stream, split in blocks

Reduction of the blocks:
- without scatter:
  sum scan + search
  $O(n \log s)$

- with scatter:
  sequential algo
  (loop over the block)
  $O(n)$

Concatenation:
- Simpler
- No wrapping
Scatter? (future work)

• Overall complexity: \(O(n)\)

• ... but other techniques in \(O(n)\)
  – Sum scan (Harris et al. or Sengupta et al.) + scatter

• Future work: tests with CUDA
  – Expected speed up \(\geq 2.5\)
Conclusion

• Orthogonal to previous works:
  – We don't compete with them, we use them!

• Better asymptotic complexity
  – $O(n)$ Vs $O(n \log n)$

• Significant speed up

• Does not require scatter
Thank you