GPU Primitives
- Case Study: Hair Rendering

Ulf Assarsson, Markus Billeter, Ola Olsson, Erik Sintorn

Chalmers University of Technology

Gothenburg, Sweden
Parallelism

- Programming massively parallel systems
Parallelism

- Programming massively parallel systems
- Parallelizing algorithms
Parallelism

- Programming massively parallel systems
- Parallelizing algorithms

- Our research on 3 key components:
  1. Stream compaction
  2. Prefix Sum
  3. Sorting
Parallelism

- Programming massively parallel systems
- Parallelizing algorithms

Our research on 3 key components:
1. Stream compaction \(3x\) faster than any other implementation we know of
2. Prefix Sum – 30% faster than CUDPP 1.1
3. Sorting – faster than newest CUDPP 1.1 July 2009
Parallelism

- Programming massively parallel systems
- Parallelizing algorithms
- Our research on 3 key components:
  1. Stream compaction
  2. Prefix Sum
  3. Sorting
Parallelism

- Programming massively parallel systems
- Parallelizing algorithms
- Our research on 3 key components:
  1. Stream compaction
  2. Prefix Sum
  3. Sorting

input

| 1 | 3 | 9 | 4 | 2 | 5 | 7 | 1 | 8 | 4 | 5 | 9 | 3 |

output

| 0 | 1 | 4 | 13 | 15 | ... | ... | ... | ... | ... | ... | ... | ... |

Each output element is sum of all preceding input elements
Parallelism

- Programming massively parallel systems
- Parallelizing algorithms
- Our research on 3 key components:
  1. Stream compaction
  2. Prefix Sum
  3. Sorting
1. Stream Compaction

- Used for:
  - Load balancing & load distribution
    - Alternative to global task queue
  - Parallel Tree Traversal

1Stream reduction operations for GPGPU applications, Horn, GPU Gems 2, 2005.
1. Stream Compaction

- **Used for:**
  - Load balancing & load distribution
    - Alternative to global task queue
  - Parallel Tree Traversal
  - Constructing spatial hierarchies
  - Radix Sort
  - Ray Tracing
    - Aila and Laine, *Understanding the Efficiency of Ray Traversal on GPUs*, HPG 2009
1. Stream Compaction - shadows

Alias Free Hard Shadows

  - Prefix sum, stream compaction, sorting

  - Prefix sum
2. Prefix Sum

- Good for
  - Solving recurrence equations
  - Sparse Matrix Computations
  - Tri-diagonal linear systems
  - Stream-compaction

Each output element is sum of all preceding input elements

<table>
<thead>
<tr>
<th>input</th>
<th>1</th>
<th>3</th>
<th>9</th>
<th>4</th>
<th>2</th>
<th>5</th>
<th>7</th>
<th>1</th>
<th>8</th>
<th>4</th>
<th>5</th>
<th>9</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>13</td>
<td>15</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
3. Sorting

Radix Sort:

– Nadathur Satish, Mark Harris, Michael Garland


– Markus Billeter, Ola Olsson, Ulf Assarsson

Stream Compaction

- Parallel algorithms often targets unlimited \#proc and have complexity $O(n \log n)$

- E.g.:

```
A x B x C D E x x x F x x G H x
```

Evaluate Predicate => Flag Array

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

Exclusive Prefix Sum

```
0 1 1 2 2 3 4 5 5 5 5 6 6 6 7 8
```

Scatter/Gather Valid Elements

```
A B C D E F G H - - - - - - - -
```

- But actual \#proc are far from unlimited
Stream Compaction

- More efficient option (~Blelloch 1990):
  - Split input among processors and work sequentially on each part
  - E.g.: Each stream processor sequentially compacts one part of stream

  - Input
    - Split input among processors
    - Work sequentially on each part
    - Each stream processor compacts one part of stream
    - Removing the unwanted elements inside each part
    - Then concatenate parts

  - Output
    - Concatenated output
Stream Compaction

- BUT:
  - Naïvely treating each SIMD-lane as one processor gives horrible memory access pattern

Many versions of algorithms improving access pattern

- We suggest treating hardware as a
  - Limited number of processors with a specific SIMD width
    - GTX280: 30 processors, logical SIMD width = 32 lanes (CUDA 2.1/2.2 API)
Stream Compaction

- Our basic idea:
  - Split input among processors and work sequentially on each part
  - Each (multi-)processor sequentially compacts one part of stream

Start by computing output offsets for each processor

...removing the unwanted elements inside each part

...then concatenate parts
Stream Compaction

- Computing the processors’ output offsets:
  - Each processor counts its number of valid elements (i.e., output length)
  - Compute Prefix Sum array for all counts
  - This array tells the output position for each processor

Input:

Counts = \{ \#valids, \#valids, \#valids, \ldots \}

Prefix sum = \{ 0, \#valids for \ p_0, \#valids for \ p_0+p_1, \#valids for \ p_0+p_1+p_2, \ldots \}

Output:
**Stream Compaction**

- Computing the processors’ output offsets:
  - Each processor counts its number of valid elements (i.e., output length)
  - Compute Prefix Sum array for all counts
  - This array tells the output position for each processor

\[
\text{Counts} = \{ \#\text{valids}, \#\text{valids}, \#\text{valids}, \ldots, \#\text{valids} \}
\]

\[
\text{Prefix sum} = \{ 0, \#\text{valids for } p_0, \#\text{valids for } p_0+p_1, \#\text{valids for } p_0+p_1+p_2, \ldots, \#\text{valids for } p_0+\ldots+p_{#p-1} \}
\]
Stream Compaction

- Each processor counts its number of valid elements

\[ w = \text{SIMD width} \]

Each processor:
- Loop through its input list:
  - Reading \( w \) elements each iteration
    - Perfectly coalesced (i.e., each thread reads 1 element)
  - Each lane (thread / stream processor) increases its counter if its element is valid
- Finally, sum the \( w \) counters
Stream Compaction

- Our basic idea:

Split input among processors and work sequentially on each part

Each processor sequentially compacts one part of stream

...removing the unwanted elements inside each part

...then concatenate parts
Stream Compaction

- Compacting the input list for each SIMD-processor

  \( w = \text{SIMD width} \)

Each processor:
- Loop through its input list:
  - Reading \( w \) elements each iteration
    - Perfectly coalesced (i.e., each thread reads 1 element)
  - Use a standard parallel compaction for \( w \) elements
  - Write to output list and update output position by \#valid elements
Stream Compaction

Stream compaction with

- Optimal coalesced reads
- Good write pattern
Steam Compaction

- In reality we use:
  - GTX280:
    - $P = 480$ to increase occupancy and hide mem latency
      - 30x4 blocks à 4 warps à 32 threads
        - Hardware specific
    - Highest memory bandwidth if each lane fetches 32 bit data in 64 bit units (i.e., 2 floats instead of 1).
      - Hardware specific

<table>
<thead>
<tr>
<th></th>
<th>32 bit fetches</th>
<th>64 bit fetches</th>
<th>128 bit fetches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth (GB/s)</td>
<td>77.8</td>
<td>102.5</td>
<td>73.4</td>
</tr>
</tbody>
</table>
Stream Compaction

- Our Trick:
  - Avoiding algorithms designed for unlimited #processors
  - Sequential algorithm – very simple
  - Split input into many independent pieces, apply sequential algorithm to each piece and combine the results later
    - Divide work among independent processors
    - Use SIMD-sequential algorithm on a processor
      - i.e., fetch block of $w$ elements
      - Use parallel algorithm when working with the $w$ elements
        - Work in fast shared memory
Stream Compaction

- The evolution of stream compaction algorithms:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>4M elements</th>
<th>NVIDIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horn (2005)</td>
<td>60 ms</td>
<td>GeForce 8800</td>
</tr>
<tr>
<td>..modified with Blelloch’s prefix sum</td>
<td>37.2 ms</td>
<td>GeForce 8800</td>
</tr>
<tr>
<td>Roger, Assarsson, Holzschuch (2007)</td>
<td>13.7 ms</td>
<td>GeForce 8800</td>
</tr>
<tr>
<td></td>
<td>2.54 ms</td>
<td>GTX280</td>
</tr>
<tr>
<td>CUDPP¹ (2009)</td>
<td>1.81 ms</td>
<td>GTX280</td>
</tr>
<tr>
<td>Billeter, Olsson, Assarsson (2009)</td>
<td>0.56 ms</td>
<td>GTX280</td>
</tr>
</tbody>
</table>

What will be next...?

¹CUDPP: Mark Harris, John D. Owens, Shubhabrata Sengupta, Yao Zhang, Andrew Davidson.

- Harris, Sengupta, and Owens. "Parallel Prefix Sum (Scan) with CUDA". GPU Gems 3, 2007.
Our Stream Compaction


Code downloadable here: www.cse.chalmers.se/~billeter/pub/pp

The error bars display variations in time as the proportion of valid elements is changed. The graphs represent the average time for varying proportions of valid elements. Also shown are curves for compaction of 64 bit and 128 bit elements.
### Making a fast Prefix Sum

- **Simple modification:**

  Split input among processors

  ![Diagram of processors](image)

  \[
  \text{Sum} = \begin{cases} 
  \Sigma, & \text{Proc 0} \\
  \Sigma, & \text{Proc 1} \\
  \Sigma, & \text{Proc 2} \\
  \vdots & \vdots \\
  \Sigma & \text{Proc } \ldots 
  \end{cases}
  \]

  \[
  \text{Prefix Sum} = \begin{cases} 
  \Sigma p_0, & \text{Proc 0} \\
  \Sigma p_{0+1}, & \text{Proc 1} \\
  \Sigma p_{0+1+2}, & \text{Proc 2} \\
  \Sigma p_{0+1+\ldots+p-1} & \text{Proc } \ldots 
  \end{cases}
  \]

  1. Each processor computes the **sum** of all its elements
  2. Compute a **prefix sum** over the \( p \) sums
  3. Each proc executes a **SIMD-sequential prefix sum** for its elements
     - and simultaneously adds the sum in all previous sublists
Making a fast Prefix Sum

- **Simple modification:**

  Split input among processors

  1. Each processor computes the **sum** of all its elements
  2. Compute a **prefix sum** over the $p$ sums
  3. Each proc executes a *SIMD-sequential prefix sum* for its elements
      - and simultaneously adds the sum in all previous sublists
Making a fast Prefix Sum

- Simple modification:

  Split input among processors

  \[\text{Proc 0} \quad \text{Proc 1} \quad \text{Proc 2} \quad \ldots\]

  \[
  \text{Sum} = \{ \Sigma, \Sigma, \Sigma, \ldots, \Sigma \}
  \]

  \[
  \text{Prefix Sum} = \{ \Sigma p_0, \Sigma p_{0+1}, \Sigma p_{0+1+2}, \ldots, \Sigma p_{0+1+\ldots+#p-1} \}
  \]

  1. Each processor computes the \textbf{sum} of all its elements
  2. Compute a \textbf{prefix sum} over the \( p \) sums
  3. Each proc executes a \textit{SIMD-sequential prefix sum} for its elements
     - and simultaneously adds the sum in all previous sublists
Simple modification:

- Split input among processors

1. Each processor computes the sum of all its elements
2. Compute a prefix sum over the $p$ sums
3. Each proc executes a *SIMD-sequential* prefix sum for its elements
   - and simultaneously adds the sum in all previous sublists
Making a fast Prefix Sum

- Simple modification:
  - Split input among processors

\[
\text{Sum} = \{ \Sigma, \Sigma, \Sigma, \ldots \Sigma \}
\]

\[
\text{Prefix Sum} = \{ \Sigma p_0, \Sigma p_{0+1}, \Sigma p_{0+1+2}, \ldots \Sigma p_{0+1+\ldots+#p-1} \}
\]

1. Each processor computes the sum of all its elements
2. Compute a prefix sum over the \( p \) sums
3. Each proc executes a SIMD-sequential prefix sum for its elements
   - and simultaneously adds the sum in all previous sublists
Making a fast Prefix Sum

- Simple modification:
  
  Split input among processors

  1. Each processor computes the **sum** of all its elements
  2. Compute a **prefix sum** over the $p$ sums
  3. Each proc executes a **SIMD-sequential prefix sum** for its elements
     - and simultaneously **adds the sum** in all previous sublists
Making a fast Prefix Sum

- Simple modification:
  - Split input among processors
    - Proc 0
    - Proc 1
    - Proc 2
    - ...

  \[
  \text{Sum} = \{ \sum, \sum, \sum, \ldots, \sum \}
  \]

  \[
  \text{Prefix Sum} = \{ \sum_{p_0}, \sum_{p_0+1}, \sum_{p_0+1+2}, \ldots, \sum_{p_0+1+\ldots+#-1} \}
  \]

1. Each processor computes the sum of all its elements
2. Compute a prefix sum over the \( p \) sums
3. Each proc executes a SIMD-sequential prefix sum for its elements
   - and simultaneously adds the sum in all previous sublists
1. Each processor computes sum of its elements:

- Loop through its input list:
  - Reading $w$ elements each iteration
    - *Perfectly coalesced* (i.e., each thread reads 1 element)
  - Each lane adds its element to its own counter
- *Finally, sum the $w$ counters*
### Making a fast Prefix Sum

- **Simple modification:**

  Split input among processors

  - Proc 0
  - Proc 1
  - Proc 2
  - ...

  \[
  \text{Sum} = \{ \Sigma, \Sigma , \Sigma, \ldots \Sigma \} \\
  \text{Prefix Sum} = \{ \Sigma p_0, \Sigma p_{0+1}, \Sigma p_{0+1+2}, \ldots \Sigma p_{0+1+\ldots+#p-1} \}
  \]

1. Each processor computes the **sum** of all its elements
2. Compute a **prefix sum** over the \( p \) sums
3. Each proc executes a **SIMD-sequential prefix sum** for its elements
   - and simultaneously adds the sum in all previous sublists
Making a fast Prefix Sum

- Simple modification:
  Split input among processors

1. Each processor computes the sum of all its elements
2. Compute a prefix sum over the $p$ sums
3. Each proc executes a *SIMD-sequential* prefix sum for its elements
   - and simultaneously adds the sum in all previous sublists
Making a fast Prefix Sum

3. Each processor executes a SIMD-sequential prefix sum of all its elements:

\[ w = \text{SIMD width} \]

- Each processor:
  - Loop through its input list:
    - Reading \( w \) elements each iteration
      - Perfectly coalesced (i.e., each thread reads 1 element)
    - Compute a standard parallel prefix sum for \( w \) elements
    - Write to output list
      - Perfectly coalesced
Results: Prefix Sum

- Easier than compaction
  - Number of output elements is equal to inputs
    ⇒ perfect coalescing when reading and writing!

- Results: 32bit elements

<table>
<thead>
<tr>
<th></th>
<th>GPU</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our</td>
<td>GTX280</td>
<td>3.7 ms</td>
</tr>
<tr>
<td>CUDPP</td>
<td>GTX280</td>
<td>5.3 ms</td>
</tr>
</tbody>
</table>
Radix Sort

- “Stream split”
  - Like compaction
  - Place invalid elements in second half of the output buffer

- Radix Sort
  - Apply stream split once for each bit in the key
Radix Sort

- Radix sorting a stream of n 32-bits elements:

\[ S = \text{Stream of n elements} \]

For \( i = 0 \ldots 31 \)

\[ S = \text{Elements in S with bit}<i> = 0 + \text{Elements in S with bit}<i> = 1 \]

Using stream split

Only 32 invocations of the stream split function
- Internal order of valid/invalid elements must be preserved in each split

Result: sorting 4M 32-bits elements (key/value) in 29 ms, GTX280.
Radix Sort

32bit keys only

32bit key/value pairs

Code downloadable here: www.cse.chalmers.se/~billeter/pub/pp
Mirrors Edge

Electronic Arts / DICE
Hair Rendering - state of the art (realtime)

In recent games

In recent research
State of the art (realtime)

A few hundred textured polygons

Half a million individual line segments
Hair Rendering

- Refreshed version of:
Hair Rendering

- Hair is challenging to render in realtime because:
  - For realistic results, hair geometry must be hundreds of thousands of very thin primitives (in realtime, lines)
  - Good looking images have been produced using textured patches, but these look bad when animated (viewed from the wrong angle)
  - The often subpixel sized, fairly transparent, primitives must be alpha blended
  - The self shadowing effects are crucial to realism, and cannot be handled by standard shadowmapping / stencil shadow techniques
Real time hair rendering

Two main challenges

Self shadowing
• Standard shadowing techniques fail
  • Shadow Maps => aliasing at silhouette edges
  • Shadow Volumes => overdraw proportional to the number of silhouette edges
  • Hair is ALL silhouette edges
• Neither technique handles transparency

Transparency
• Each strand should contribute very little to a pixel (~1%)
• Hair strands are actually refractive and at least some transparency effect is required
• Alpha blending works very well to handle this
Transparency is order dependent

Standard solution: sort transparent primitives and render back-to-front.

Standard alpha blending equation for transparency:

\[ f = \alpha c + (1-\alpha)b \]
Figure 2: Variance contributions to stratified sampling. (a) When a single silhouette edge passes through the filter region, $O(N^{1/2})$ samples contribute to the variance. (b) When the filter region is covered with fine geometry, all $N$ samples contribute to the variance, resulting in a much larger expected error.

Importance of Shadows

Importance of Shadows

- The need for selfshadowing
Importance of Shadows

With hair self shadowing

Without hair self shadowing
Hair is sub-pixel sized and transparent, alpha blending is absolutely necessary.

**Importance of Transparency**

- Without alpha blending
- With alpha blending
Importance of Transparency

Hair rendered without alpha blending.

Hair rendered with alpha blending ($=0.2$).
Real time hair rendering

The two problems are quite similar

For shadows, we want to know how much the hair fragments, in front, blocks the light
- Can be solved by sorting

For alpha blending, we need the fragments sorted on their depth
Related Work

KAJIYA AND KAY. Rendering fur with three-dimensional textures, SIGGRAPH 1989.


Related Work

- Opacity Shadow Maps
  - by Tae-yong Kim and Ulrich Neumann, Rendering Techniques 2001

- Deep Opacity Maps
  - by Cem Yuksel and John Keyser, Eurographics 2008
Opacity Maps

- Build a 3d texture where each slice represents the hair opacity at a certain distance from light
  \[ \Rightarrow \text{Each texel} = \text{amount of shadow} \]
  - Two classic options
    - Advance far plane per slice
    - Advance both near + far plane and copy result from previous slice.

Essentially a 3D-texture with shadow values.
Each slice: 512x512 texels
256 slices
Opacity Maps

- Build a 3d texture where each slice represents the hair opacity at a certain distance from light

  ⇒ Each texel = amount of shadow

- Two classic options
  - Advance far plane per slice
  - Advance both near + far plane and copy result from previous slice.

Essentially a 3D-texture with shadow values.
Each slice: 512x512 texels
256 slices
Opacity Maps

- Build a 3d texture where each slice represents the hair opacity at a certain distance from light

  ⇒ Each texel = amount of shadow

- Two classic options
  - Advance far plane per slice
  - Advance both near + far plane and copy result from previous slice.

Essentially a 3D-texture with shadow values.
Each slice: 512x512 texels
256 slices
Opacity Maps

- Build a 3d texture where each slice represents the hair opacity at a certain distance from light

  ⇒ Each texel = amount of shadow

- Two classic options
  - Advance far plane per slice
  - Advance both near + far plane and copy result from previous slice.

- Disadvantage
  - All geometry sent for rendering for each slice

Essentially a 3D-texture with shadow values.
Each slice: 512x512 texels
256 slices
Opacity Maps

- Build a 3d texture where each slice represents the hair opacity at a certain distance from light
  → Each texel = amount of shadow

- Wish:
  - Know which hair strands that should be rendered into which slice

- Advantage
  - All hair strands just rendered once.
Opacity Maps

- In NVIDIA’s Nalu demo, an implementation is suggested that renders 16 slices in one pass, by:
  - rendering opacity to four channels of four rendertargets.
Opacity Maps

- In general, 16 slices is not enough:
  - Today 32 rendertargets possible
    - But generates 32 writes per hair fragment which is slow!
Partial-Radixsort of hair strands

- Our original method used a partial quicksort algorithm based on geometry shaders

- Partial radix sort is much faster...
  - Sort on the lines’ center points. Divide into 256 buckets
Partial-Radixsort

- Quick sort would require \( n-1 \) stream split calls
  - (not feasible)
  - Partial quick sort into 256 buckets requires 255 calls
    - Still quite expensive

- Radix sort of 32 bit numbers requires 32 stream split calls

- **Partial radix sort into 256 buckets requires 8 calls**
  - Fast
Building the Opacity Map Texture

- Now, it is easy to build the opacity map texture by:
  - Enabling additive blending
  - Set up camera from lights viewpoint
  - For each slice $s$
    - Render bucket $s$ into the texture-slice
    - Copy the texture-slice to texture-slice $s+1$
Building the Opacity Map Texture

Now, it is easy to build the opacity map texture by:

- Enabling additive blending
- Setting up camera from lights viewpoint
- For each slice:
  - Render buckets into texture slice
  - Copy texture slice to texture slice + 1
With radix sorting, alpha blending is easy

- Simply sort geometry into sublists for each slice of the viewing frustrum from the cameras viewpoint
- This time, sort back to front
- Render the generated VBO
About half a million line segments rendered with 256 Opacity Map slices and approximate alpha sorting at 37 fps (GTX280)
Results

27 fps using 400k hair strands (1.8M line segments)
Demo

Movie
Drawback

- Working memory consumption:
  - e.g. $512 \times 512 \times 256 = 64 \text{Mb}$
  - independent of #objects

Solutions

- NVIDIA’ GDC-presentation 2009:
  *Advanced Visual Effects with Direct3D for PC*, Cem Cebenoyan, Sarah Tariq, and Simon Green

Or

**Solution 1:**

- Use only one sorting pass
  - that sorts into slices along vector half way between light and view direction
  - This allows 2D-shadow texture instead of 3D-texture

Solution 1:

- Alpha blending either back-to-front or front-to-back
  - Render slices to screen using the 2D-shadow texture
    - If $\theta<90^\circ$, in front-to-back order
    - Else, in back-to-front order
  - Render slice into shadow texture

\[ f = \alpha c + (1-\alpha)b \]

Solution 1: Caveat

- Possible caveat for rectangles and lines

   ![Diagram](image)

   Incorrect front-to-back order

   But the sorting only needs to be approximate anyway
Solution 2:

- If all hair strands have identical alpha-value:
## Timings

<table>
<thead>
<tr>
<th>Algorithm steps</th>
<th>(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorting -incl both:</td>
<td></td>
</tr>
<tr>
<td>Create key/value-pairs</td>
<td>3.2</td>
</tr>
<tr>
<td>Sort into buckets</td>
<td>8.1</td>
</tr>
<tr>
<td>Shadows:</td>
<td></td>
</tr>
<tr>
<td>Create shadow-map</td>
<td>~0.1</td>
</tr>
<tr>
<td>Create opacity maps</td>
<td>12</td>
</tr>
<tr>
<td>Render:</td>
<td></td>
</tr>
<tr>
<td>body with hair</td>
<td>0.16</td>
</tr>
<tr>
<td>hair with shadows</td>
<td>13.5</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td>36.3  = 27 fps</td>
</tr>
</tbody>
</table>
Implementation of stream compaction, prefix sum and radix sort available at

• http://www.cse.chalmers.se/~billeter/pub/pp

Thank you for your attention.

Questions?

These slides are available at:
http://www.cse.chalmers.se/~uffe/publications.htm
Figure 2: Variance contributions to stratified sampling. (a) When a single silhouette edge passes through the filter region, \( O(N^{1/2}) \) samples contribute to the variance. (b) When the filter region is covered with fine geometry, all \( N \) samples contribute to the variance, resulting in a much larger expected error.