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

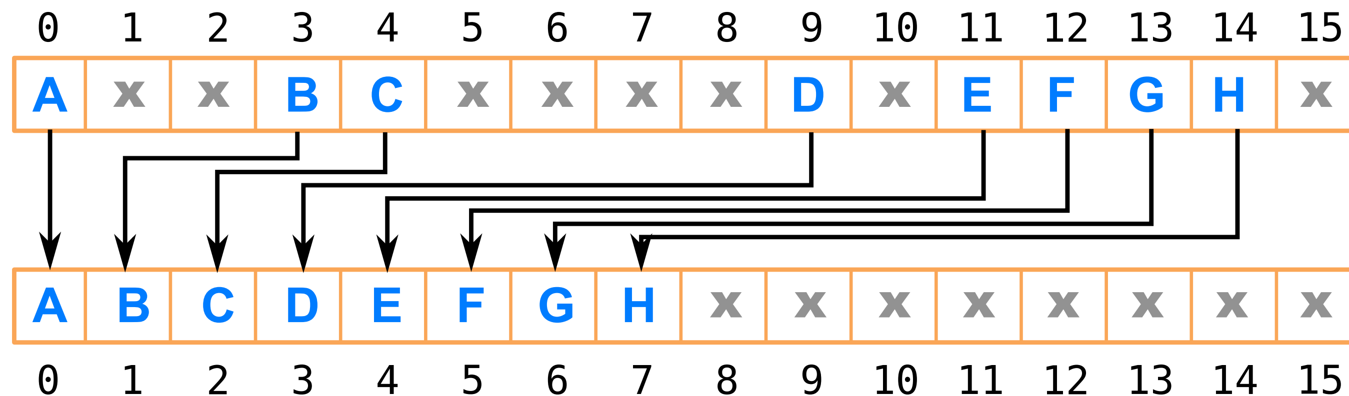
- Programming massively parallel systems
- Parallelizing algorithms
- Our research on 3 key components:

1. Stream compaction
2. Prefix Sum – 30% faster than CUDPP 1.1
3. Sorting – faster than newest CUDPP 1.1 July 2009

3x faster than any other implementation we know of

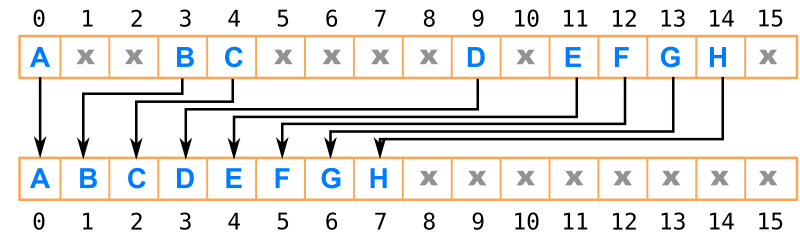
Parallelism

- 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

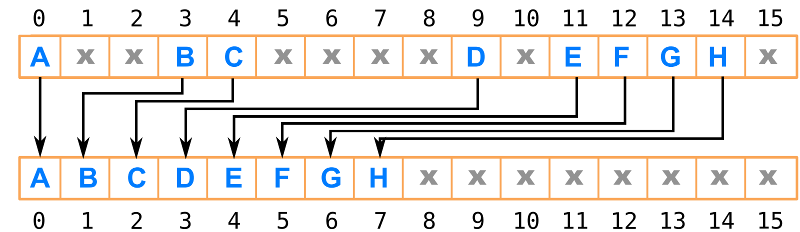


input	1	3	9	4	2	5	7	1	8	4	5	9	3
output	0	1	4	13	15	…	…	…	…	…	…	…	…

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



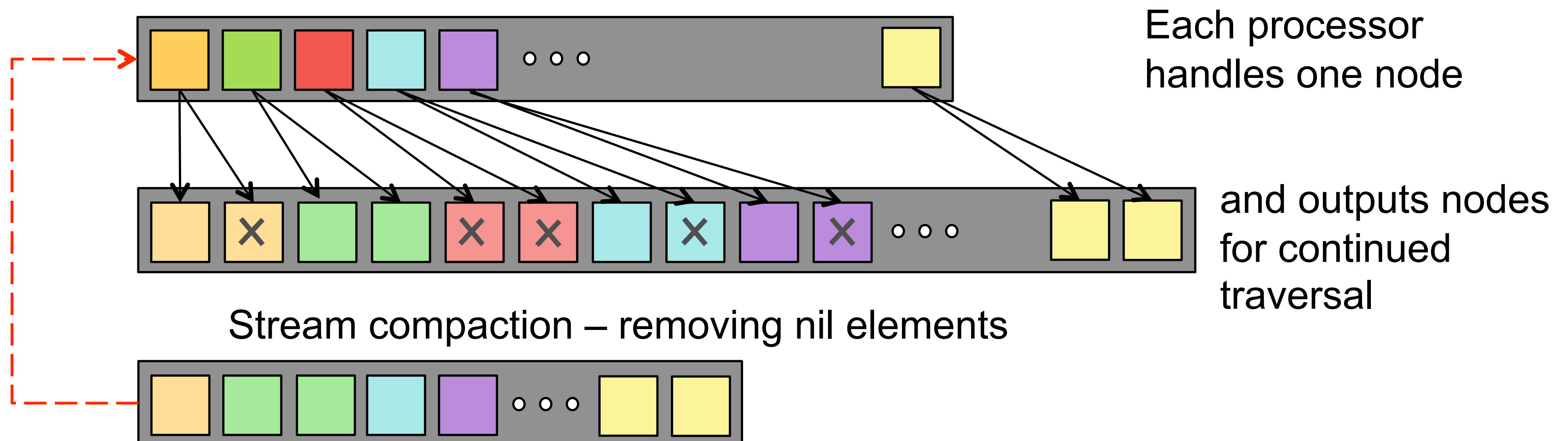
input	1	3	9	4	2	5	7	1	8	4	5	9	3
output	0	1	4	13	15	…	…	…	…	…	…	…	…

19 5 100 1 63 79

1 5 19 63 79 100

1. Stream Compaction

- Used for:
  - Load balancing & load distribution
    - Alternative to global task queue
  - Parallel Tree Traversal
    - Collision Detection - Horn, GPUGems 2, 2005.[1]



[1]*Stream reduction operations for GPGPU applications,* Horn, GPU Gems 2, 2005.

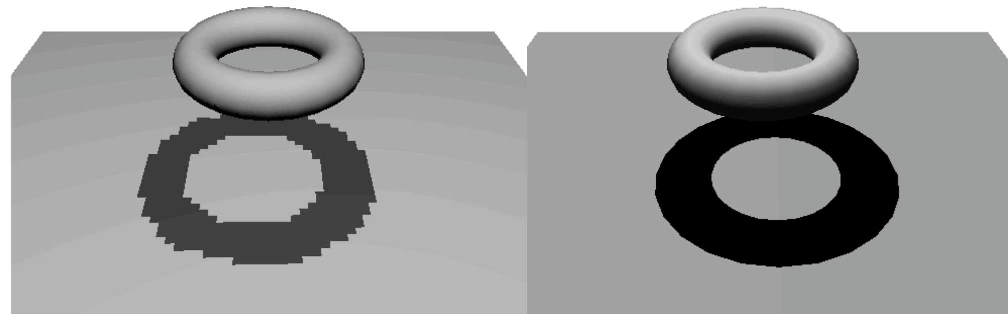
1. Stream Compaction

- Used for:
  - Load balancing & load distribution
    - Alternative to global task queue
  - Parallel Tree Traversal
    - Collision Detection - Horn, GPUGems 2, 2005.
  - Constructing spatial hierarchies
    - Lauterbach, Garland, Sengupta, Luebke, Manocha, *Fast BVH Construction on GPUs*, EGSR 2009
  - Radix Sort
    - Satish, Harris, Garland, *Designing efficient sorting algorithms for manycore GPUs*, IEEE Par. & Distr. Processing Symp., May 2009
  - Ray Tracing
    - Aila and Laine, *Understanding the Efficiency of Ray Traversal on GPUs*, HPG 2009
    - Roger, Assarsson, Holzschuch, *^{2}Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU,* EGSR 2007.

1. Stream Compaction - shadows

Alias Free Hard Shadows

- *Resolution Matched Shadow Maps*, by Aaron Lefohn, Shubhabrata Sengupta, John Owens, Siggraph 2008
  - Prefix sum, stream compaction, sorting
- *Sample Based Visibility for Soft Shadows using Alias-free Shadow Maps*, by Erik Sintorn, Elmar Eisemann, Ulf Assarsson, EGSR 2008
  - Prefix sum



2. Prefix Sum

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

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

3. Sorting

Radix Sort:

- Nadathur Satish, Mark Harris, Michael Garland

  Designing Efficient Sorting Algorithms for Manycore GPUs, IEEE Parallel & Distributed Processing Symposium, May 2009.

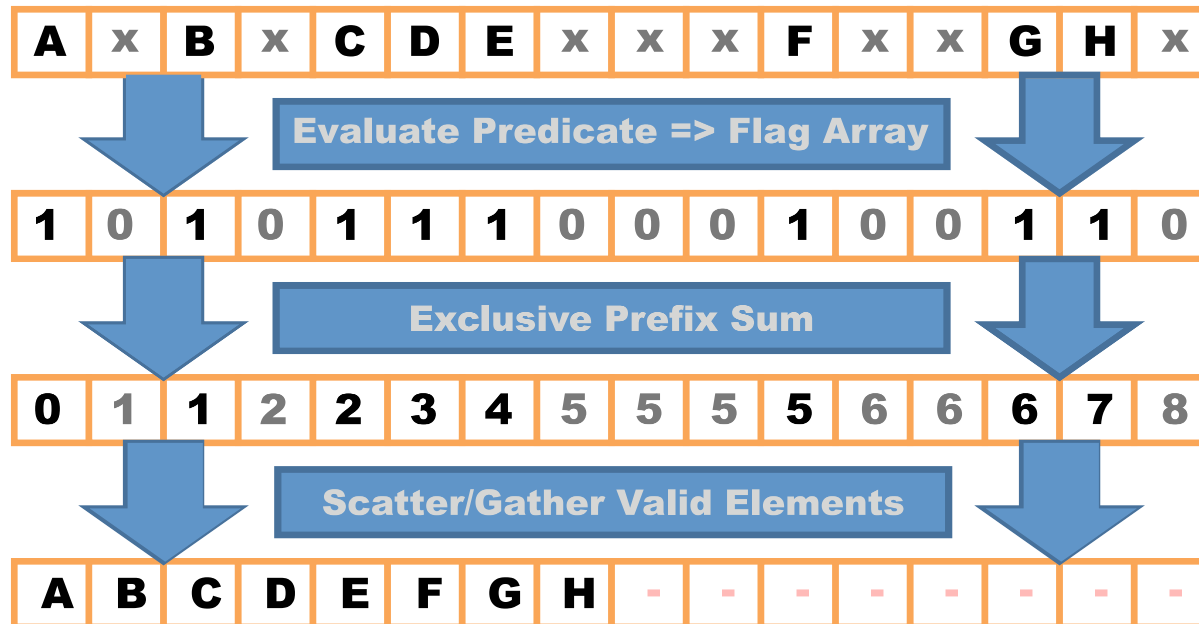
- Markus Billeter, Ola Olsson, Ulf Assarsson

  Efficient Stream Compaction on Wide SIMD Many-Core Architectures", HPG, 2009.

Stream Compaction

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

- E.g.:



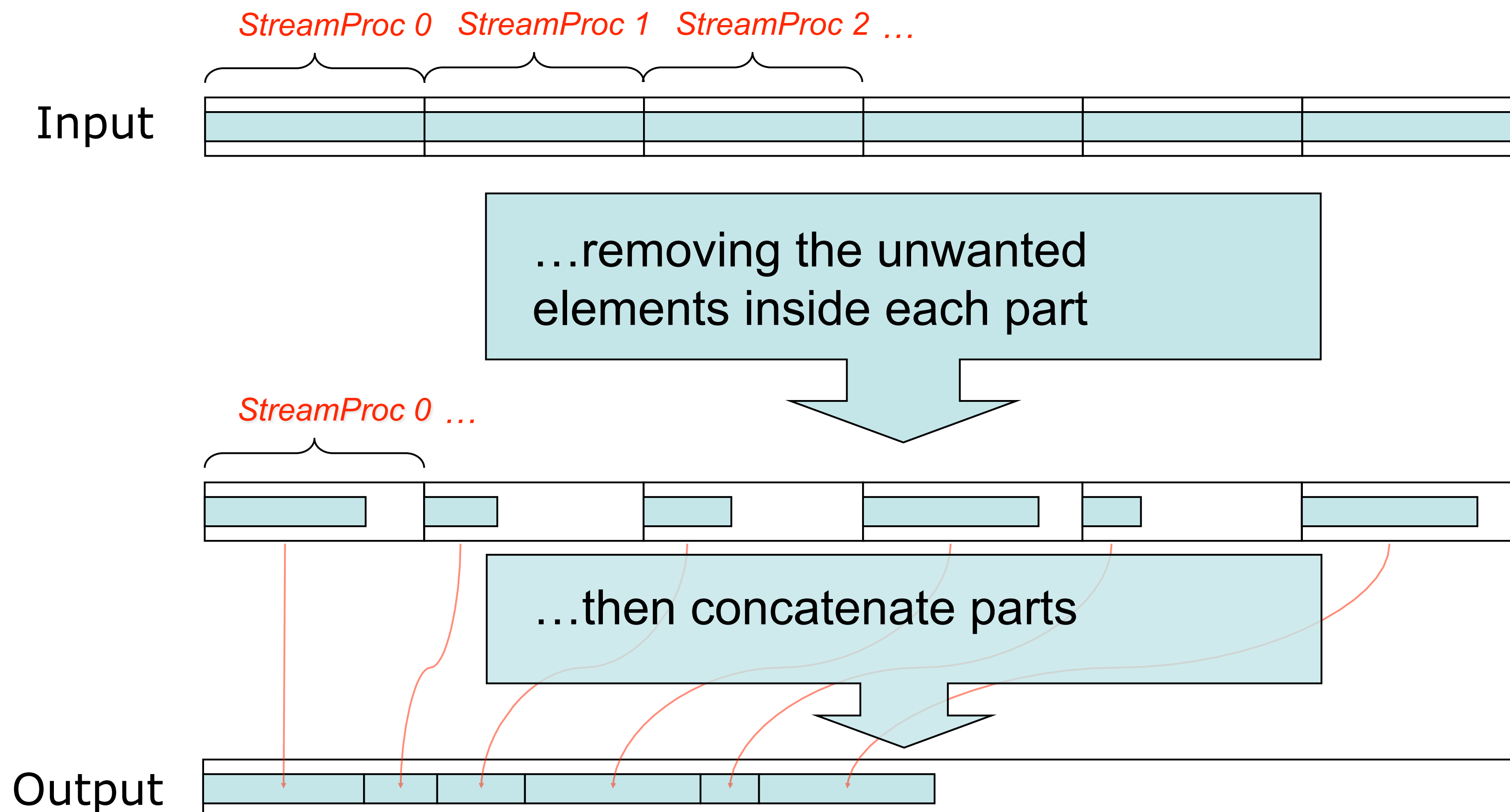
- But actual #proc are far from unlimited

Stream Compaction

- More efficient option (~Blelloch 1990):

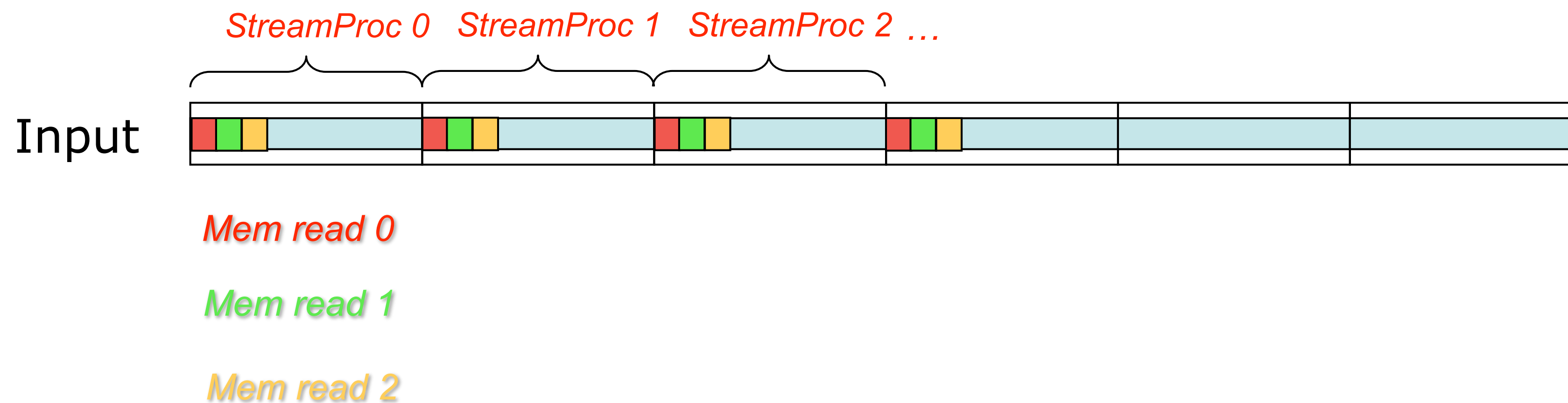
  Split input among processors and work sequentially on each part

  E.g.: Each stream processor sequentially compacts one part of stream



Stream Compaction

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



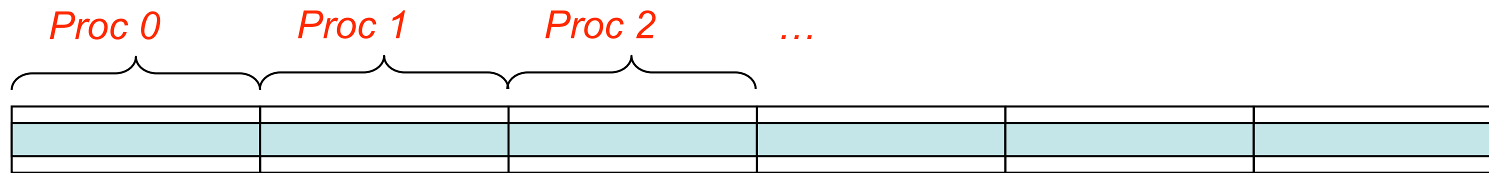
  - Many versions of algorithms improving access pattern
  - We suggest treating hardware as a
    - Limited number of processors with a specific SIMD width
      - GTX280: 30 processors, logical SIMD width = 32 lanes
        (CUDA 2.1/2.2 API)

Stream Compaction

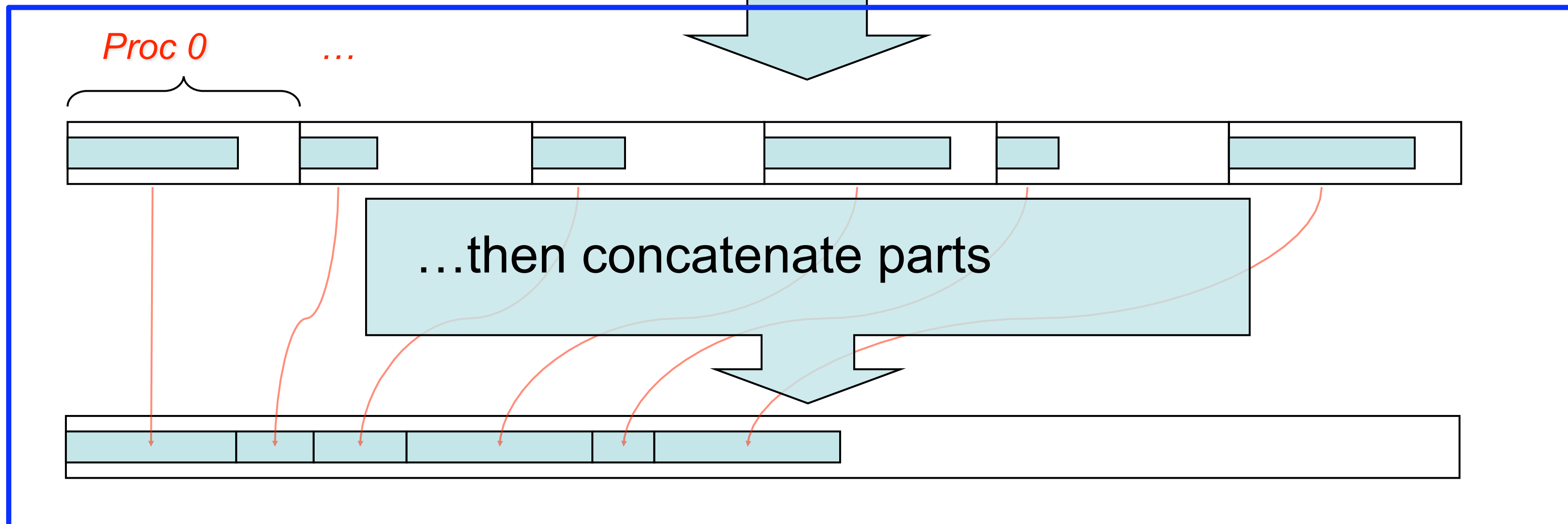
- Our basic idea:

  Split input among processors and work sequentially on each part

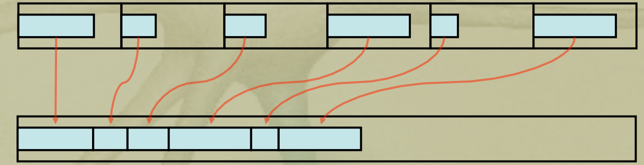
  Each (multi-)processor sequentially compacts one part of stream



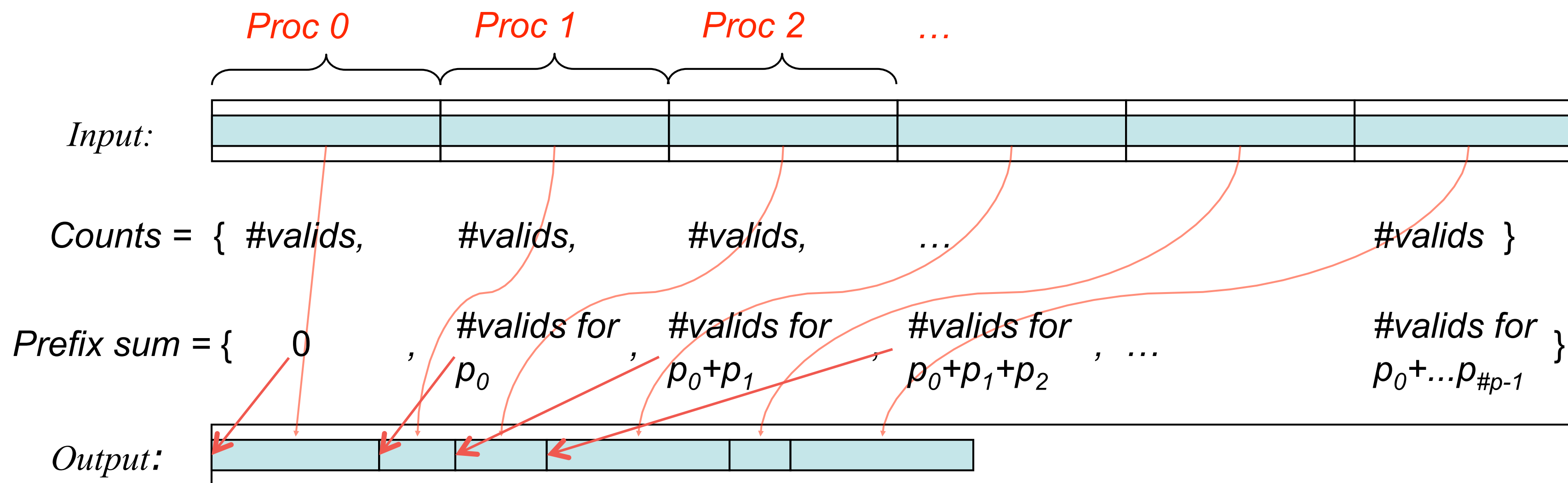
Start by computing output offsets for each processor



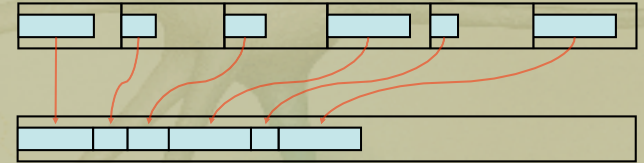
Stream Compaction



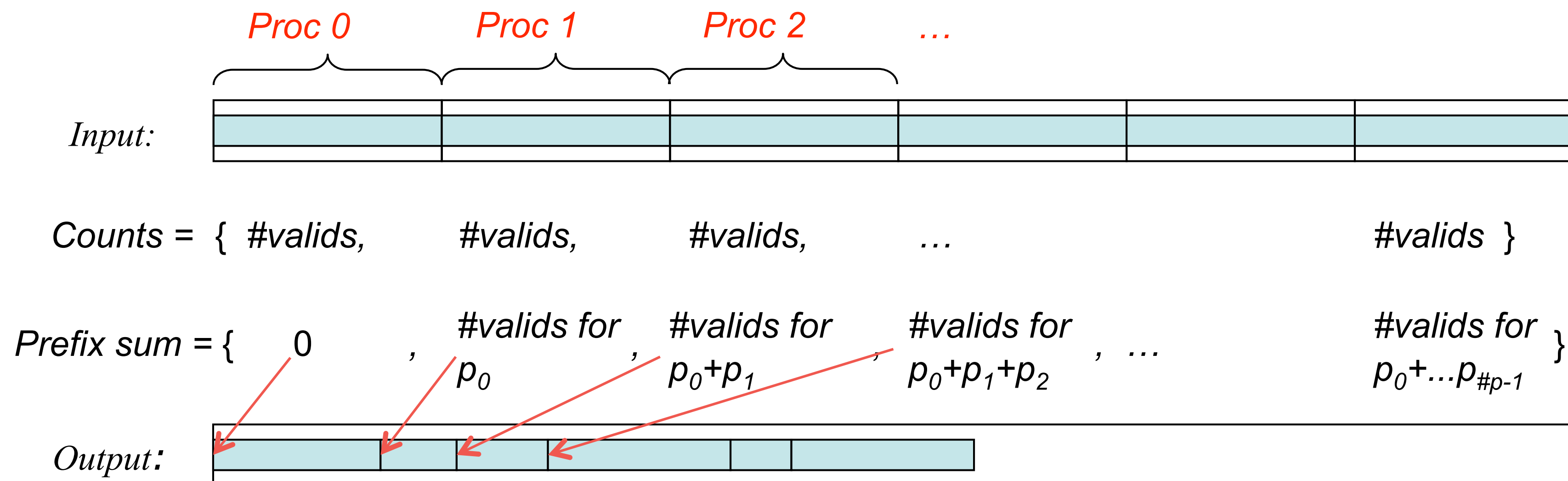
- Computing the processors' output offsets:
  - Each processor counts its number of valid elements (i.e., output length)
  - Compute Prefix Sum array for all counts
  - This array tells the output position for each processor



Stream Compaction

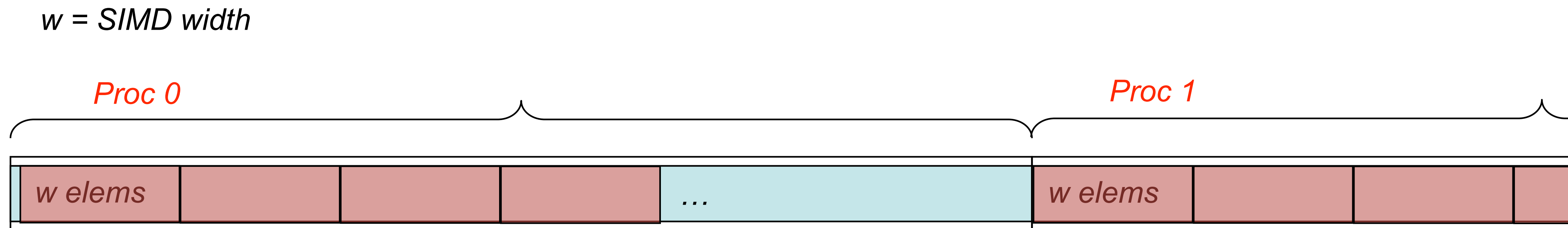


- Computing the processors' output offsets:
  - Each processor counts its number of valid elements (i.e., output length)
  - Compute Prefix Sum array for all counts
  - This array tells the output position for each processor



Stream Compaction

- Each processor counts its number of valid elements



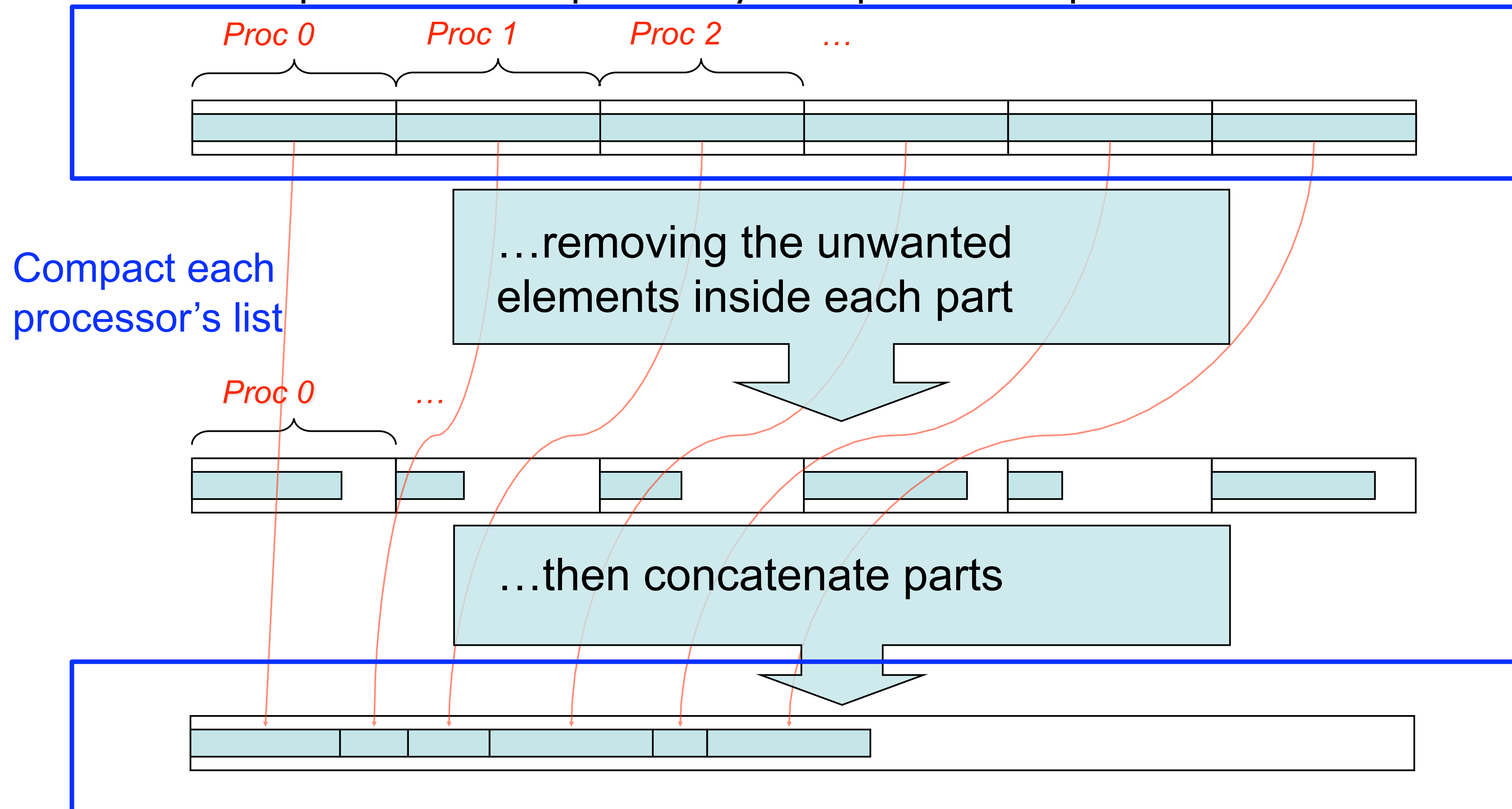
- Each processor:
  - Loop through its input list:
    - Reading *w* elements each iteration
      - *Perfectly coalesced (i.e., each thread reads 1 element)*
    - Each lane (thread / stream processor) increases its counter if its element is valid
  - *Finally, sum the w counters*

Stream Compaction

- Our basic idea:

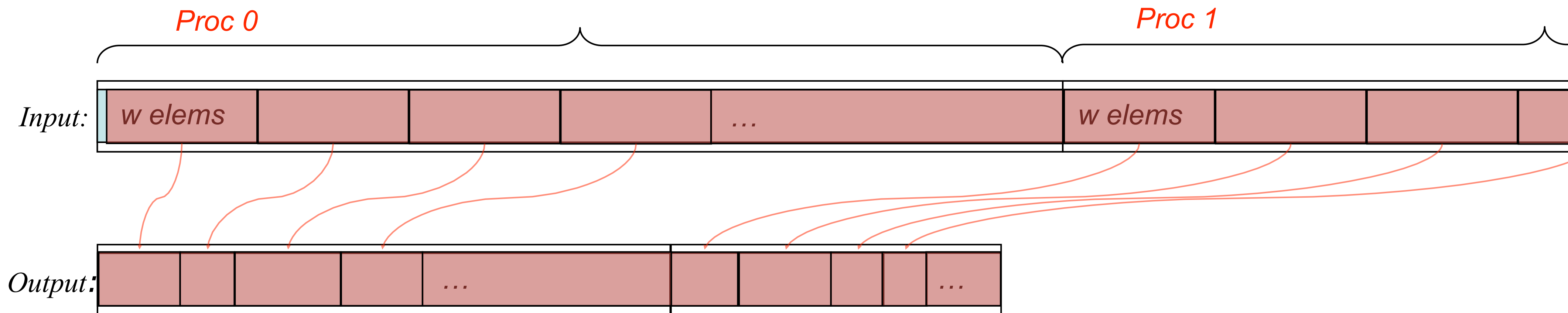
  Split input among processors and work sequentially on each part

  Each processor sequentially compacts one part of stream



Stream Compaction

- Compacting the input list for each SIMD-processor



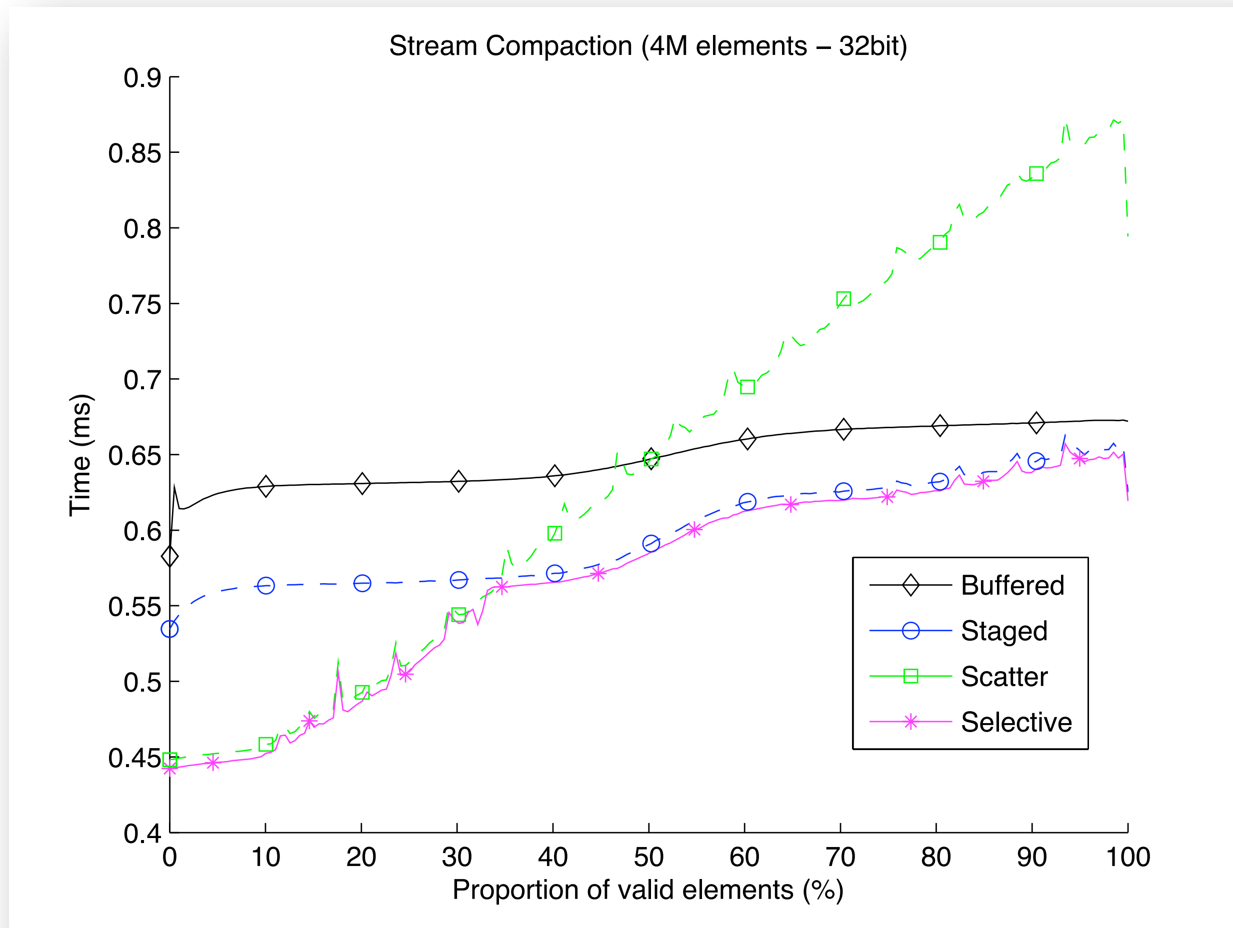
- Each processor:
  - Loop through its input list:
    - Reading *w* elements each iteration
      - *Perfectly coalesced (i.e., each thread reads 1 element)*
    - Use a standard parallel compaction for *w* elements
    - Write to output list and update output position by #valid elements

POPC
SSE-Movmask
Any/All

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

32x	32 bit fetches	64 bit fetches	128 bit fetches
Bandwidth (GB/s)	77.8	102.5	73.4

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:

	Algorithm	4M elements	NVIDIA
shaders	*Horn (2005)*	60 ms	Geforce 8800
	..modified with Blelloch's prefix sum	37.2 ms	Geforce 8800
	Roger, Assarsson, Holzschuch (2007)	13.7 ms	Geforce 8800
CUDA	*Ziegler, Tevs, Theobalt, Seidel (2006)*	3.56	Geforce 8800
		2.54 ms	GTX280
	CUDPP[1] (2009)	1.81 ms	GTX280
	Billeter, Olsson, Assarsson (2009)	0.56 ms	GTX280
	What will be next… ?		

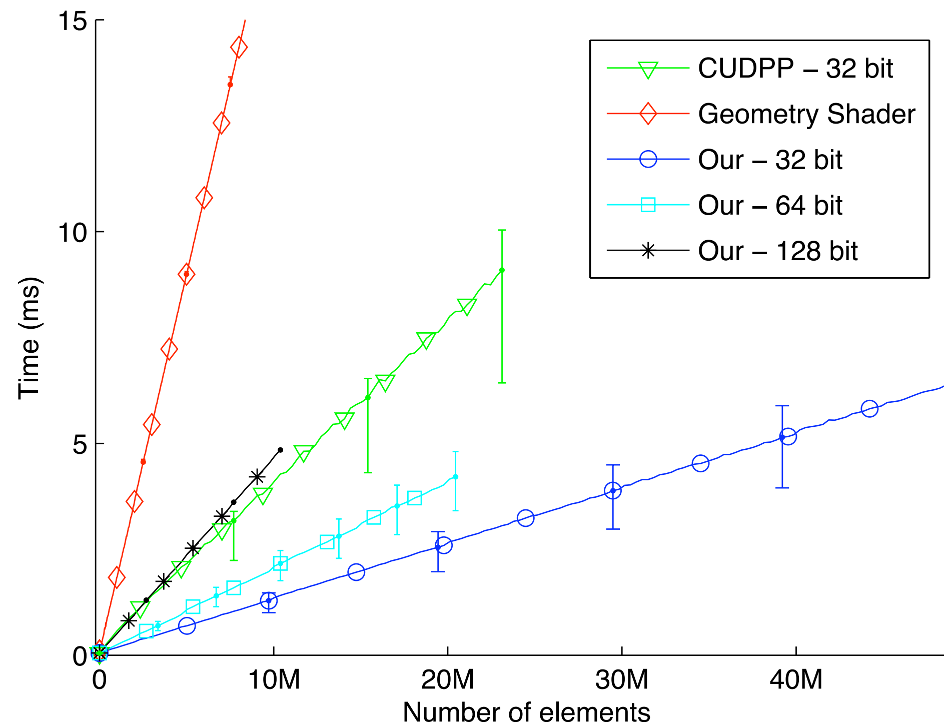
[1]CUDPP: Mark Harris, John D. Owens, Shubhabrata Sengupta, Yao Zhang, Andrew Davidson.

- *Harris, Sengupta, and Owens. "Parallel Prefix Sum (Scan) with CUDA". GPU Gems 3, 2007.*
- *Sengupta, Harris, Zhang, and Owens. "Scan Primitives for GPU Computing". Graphics Hardware 2007.*

Our Stream Compaction

Markus Billeter, Ola Olsson, Ulf Assarsson, “*Efficient Stream Compaction on Wide SIMD Many-Core Architectures*”, HPG, 2009.

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

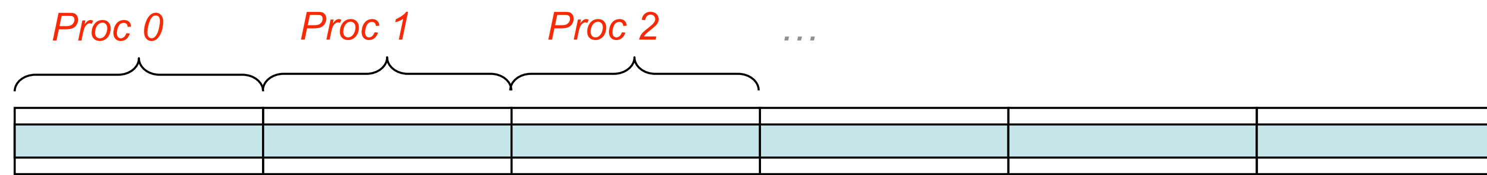


The error bars display variations in time as the proportion of valid elements is changed. The graphs represent the average time for varying proportions of valid elements. Also shown are curves for compaction of 64 bit and 128 bit elements.

Making a fast Prefix Sum

- Simple modification:

  Split input among processors



Sum = { Σ , Σ , Σ , … Σ }

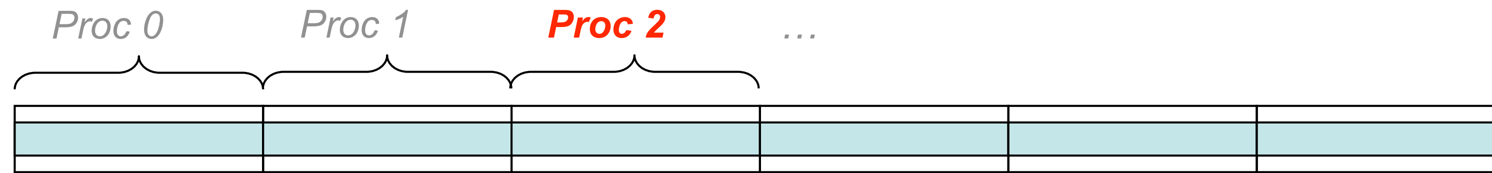
Prefix Sum = { Σp_0 , Σp_{0+1} , Σp_{0+1+2} , $\Sigma p_{0+1+\ldots+\#p-1}$

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



Sum = { Σ , Σ , Σ , … Σ }

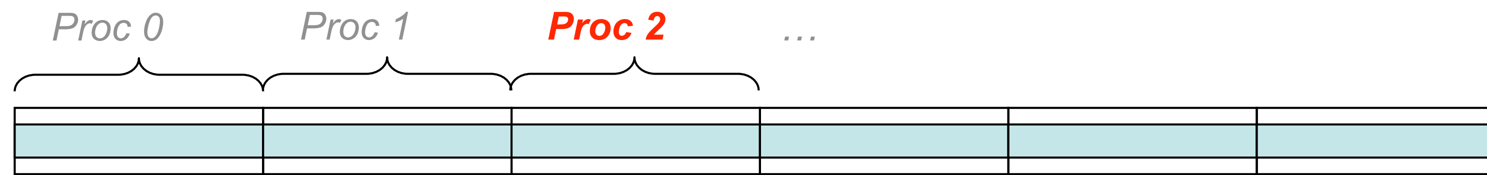
Prefix Sum = { Σp_0 , Σp_{0+1} , Σp_{0+1+2} , $\Sigma p_{0+1+\ldots+\#p-1}$

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



Sum = $\{ \Sigma, \Sigma, \Sigma, \ldots \Sigma \}$

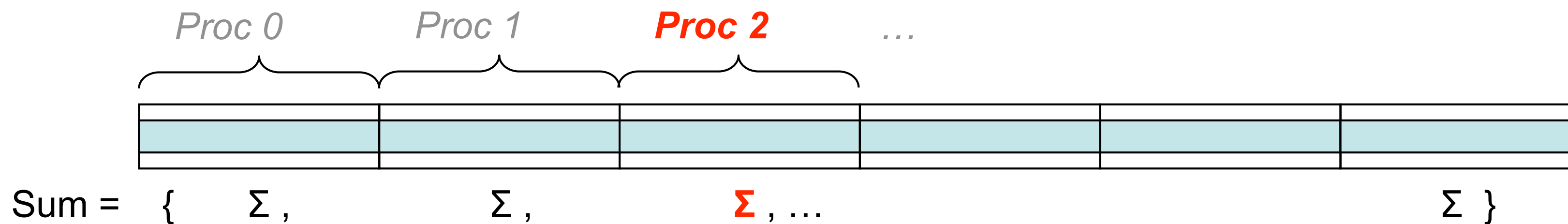
Prefix Sum = $\{ \Sigma p_0, \Sigma p_{0+1}, \Sigma p_{0+1+2}, \Sigma p_{0+1+\ldots+\#p-1}$

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

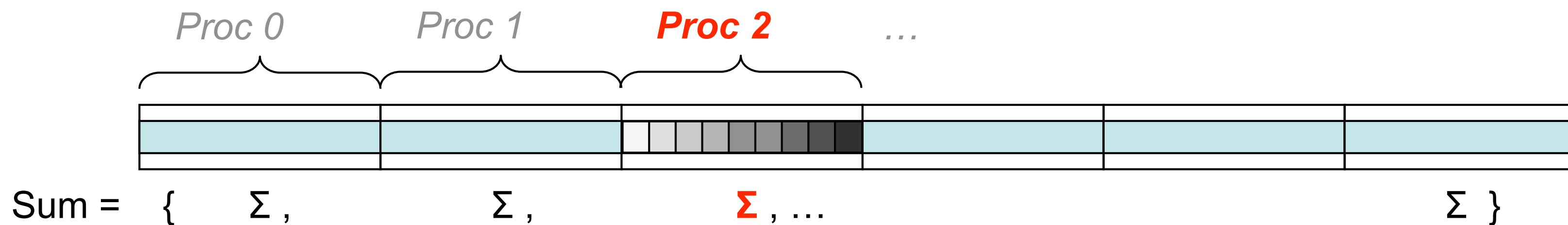


1. Each processor computes the **sum** of all its elements
2. Compute a **prefix sum** over the *p* sums
3. Each proc executes a *SIMD-sequential* **prefix sum** for its elements
   - and simultaneously adds the sum in all previous sublists

Making a fast Prefix Sum

- Simple modification:

  Split input among processors



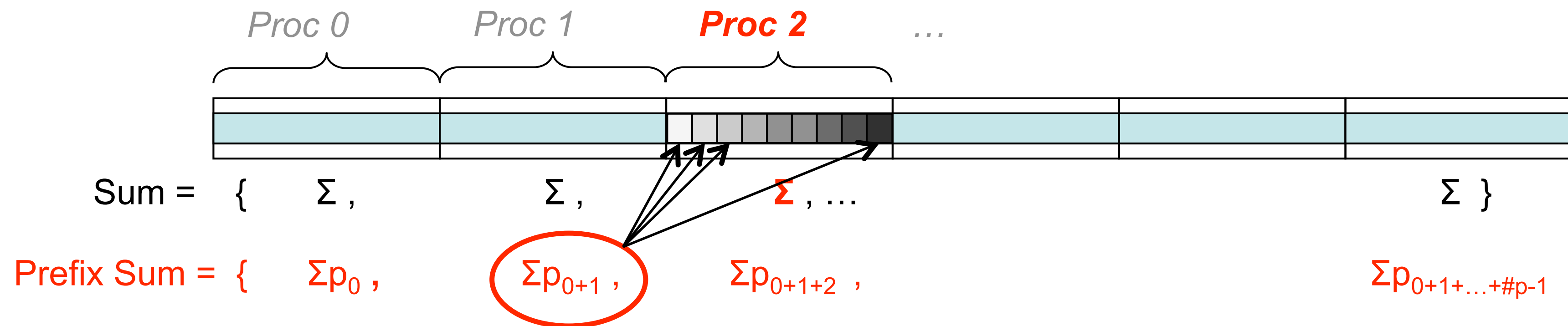
Prefix Sum = { Σp_0 , Σp_{0+1} , Σp_{0+1+2} , $\Sigma p_{0+1+\ldots+\#p-1}$

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

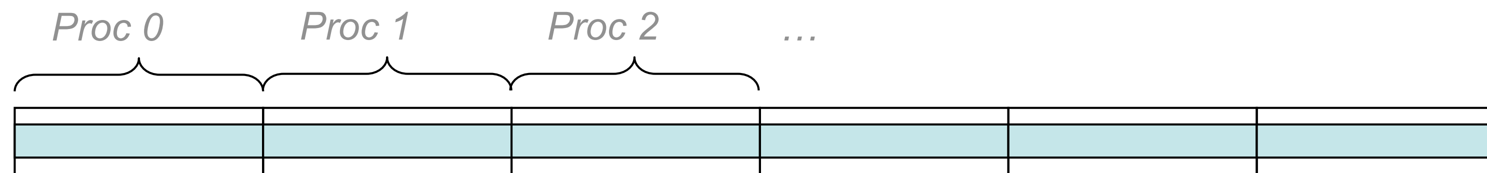


1. Each processor computes the **sum** of all its elements
2. Compute a **prefix sum** over the *p* sums
3. Each proc executes a *SIMD-sequential* **prefix sum** for its elements
   – and simultaneously **adds the sum** in all previous sublists

Making a fast Prefix Sum

- Simple modification:

Split input among processors



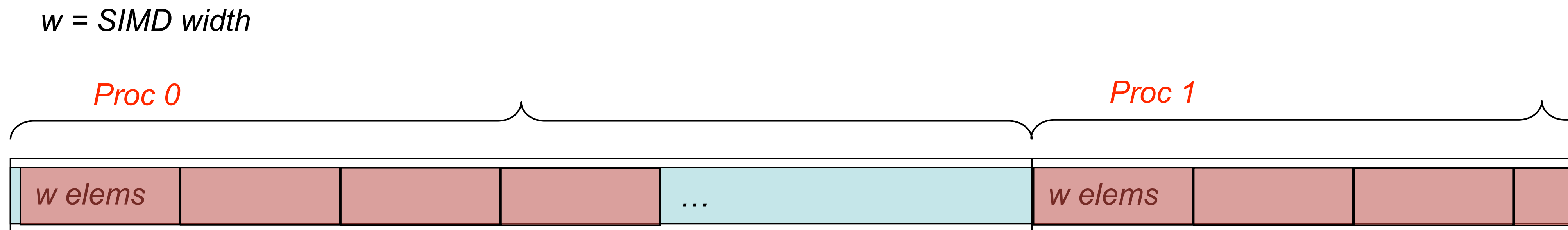
Sum = { Σ , Σ , Σ , … Σ }

Prefix Sum = { Σp_0 , Σp_{0+1} , Σp_{0+1+2} , $\Sigma p_{0+1+\ldots+\#p-1}$

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

1. Each processor computes sum of its elements:

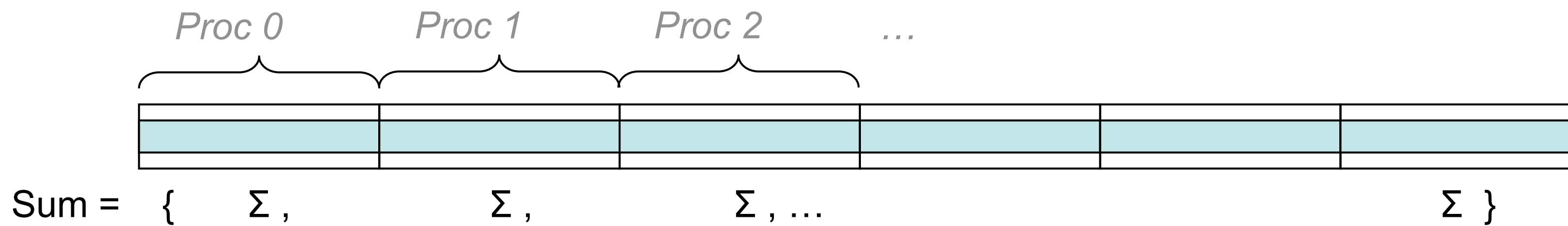


- Each processor:
  - Loop through its input list:
    - Reading *w* elements each iteration
      - *Perfectly coalesced (i.e., each thread reads 1 element)*
    - Each lane adds its element to its own counter
  - *Finally, sum the w counters*

Making a fast Prefix Sum

- Simple modification:

Split input among processors

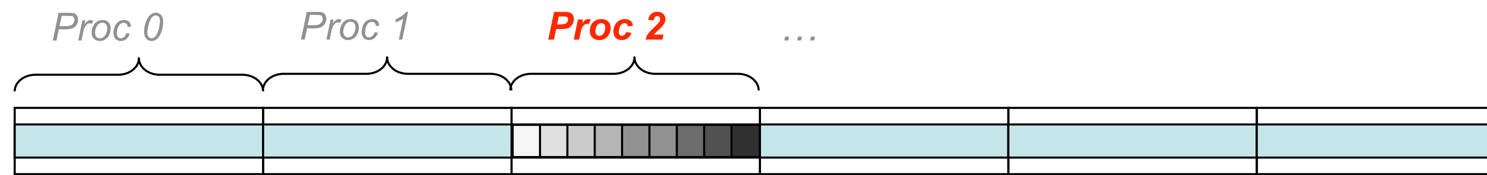


1. Each processor computes the **sum** of all its elements
2. Compute a **prefix sum** over the *p* sums
3. Each proc executes a *SIMD-sequential* **prefix sum** for its elements
   – and simultaneously adds the sum in all previous sublists

Making a fast Prefix Sum

- Simple modification:

Split input among processors



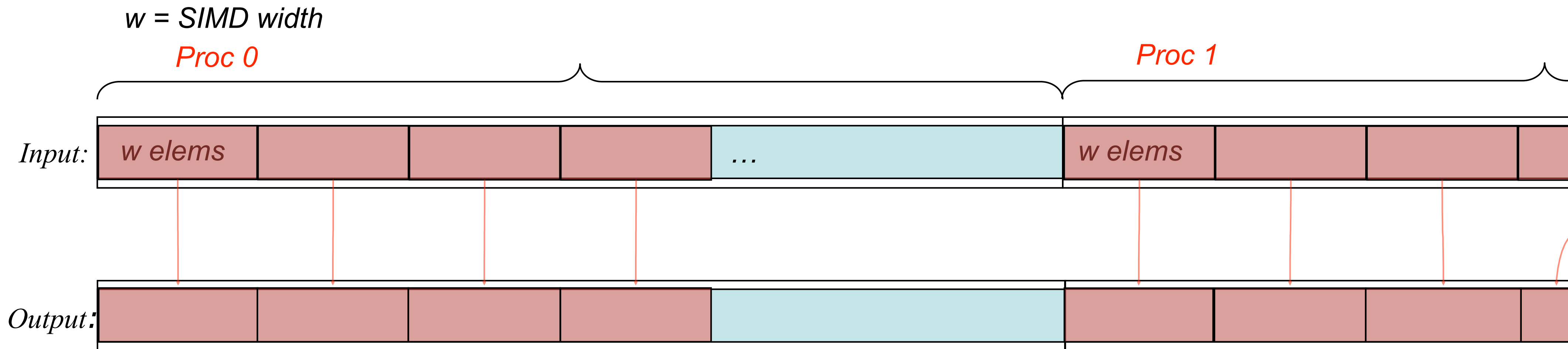
Sum = { Σ , Σ , Σ , … Σ }

Prefix Sum = { Σp_0 , Σp_{0+1} , Σp_{0+1+2} , $\Sigma p_{0+1+\ldots+\#p-1}$

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

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



- 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

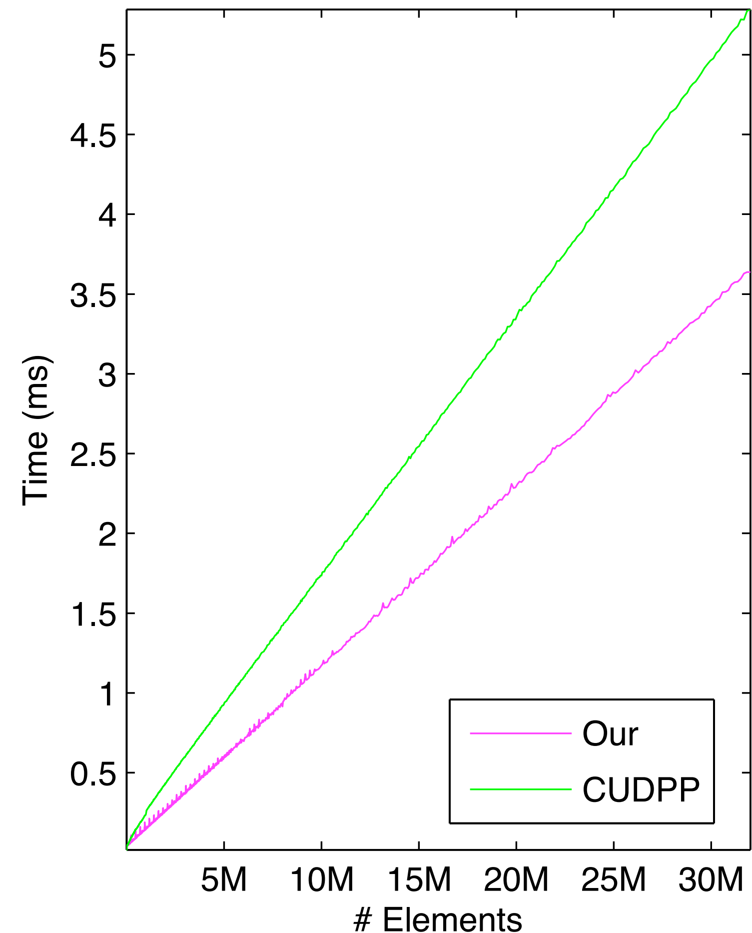
Results: Prefix Sum

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

- Results: 32bit elements

4M elements:

	GPU	Time
Our	GTX280	3.7 ms
CUDPP	GTX280	5.3 ms



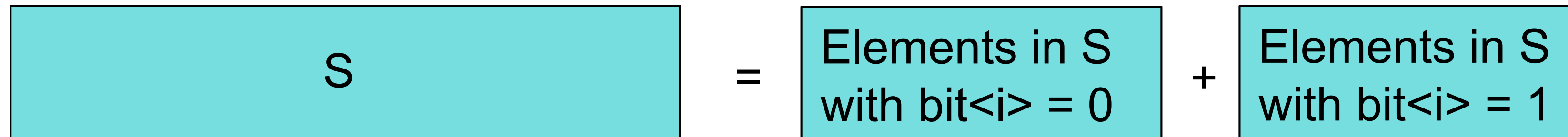
Radix Sort

- "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 = Stream of n elements
For i=0..31



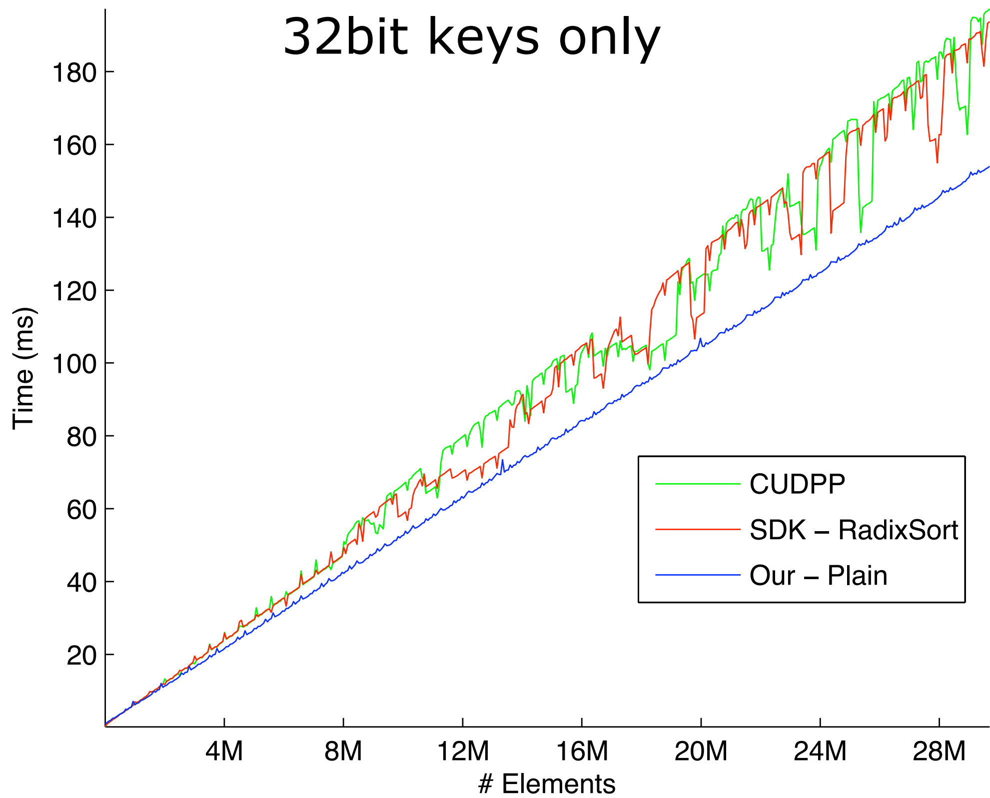
Using stream split

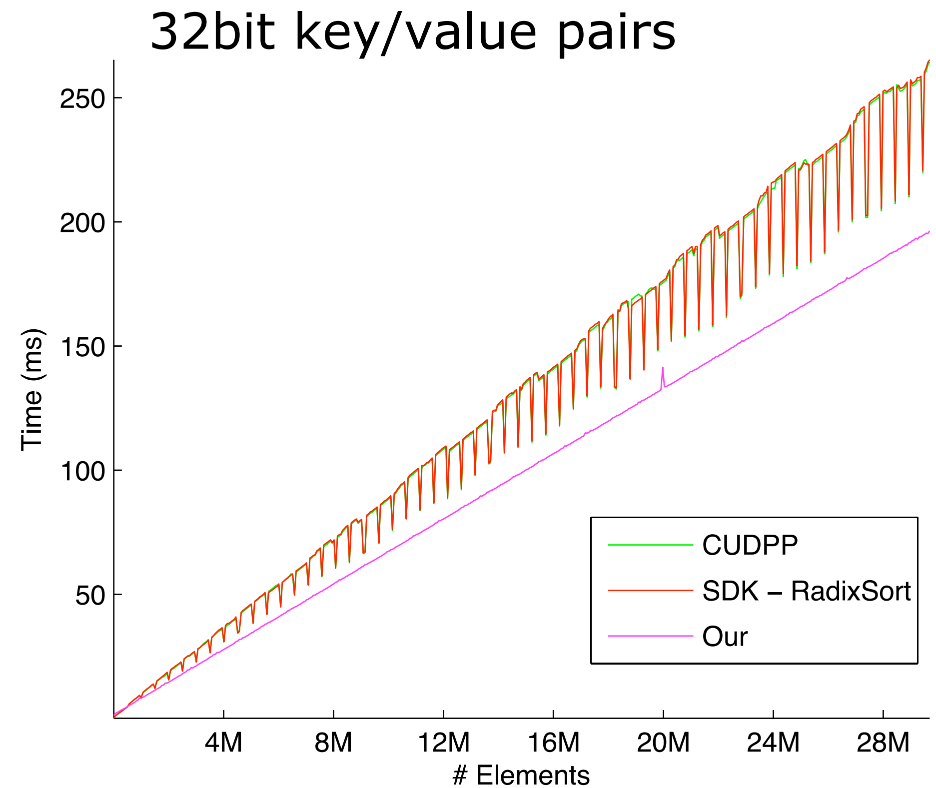
Only 32 invocations of the stream split function

- 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





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

42

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:
  - Erik Sintorn, Ulf Assarson. *Real-Time Approximate Sorting for Self Shadowing and Transparency in Hair Rendering.* I3D 2008*.*

Hair Rendering

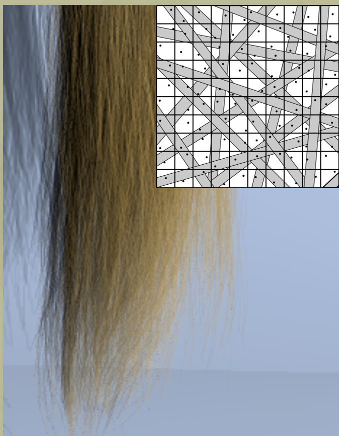
- Hair is challenging to render in realtime because:
  - For realistic results, hair geometry must be hundreds of thousands of very thin primitives (in realtime, lines)
  - Good looking images have been produced using textured patches, but these look bad when animated (viewed from the wrong angle)
  - 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

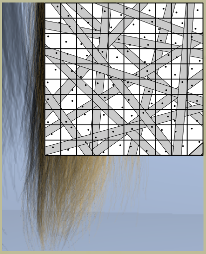
Self shadowing

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

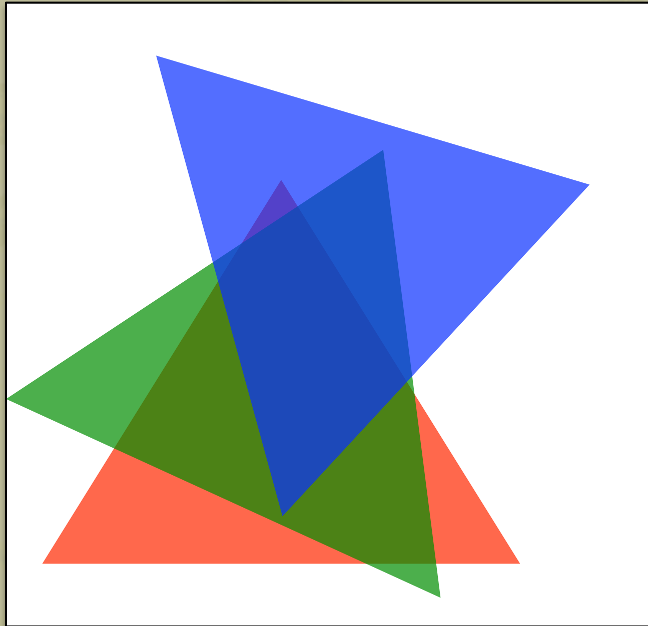


Transparency

- 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:

$$f = \alpha \mathbf{c} + (1-\alpha)\mathbf{b}$$

Aliasing

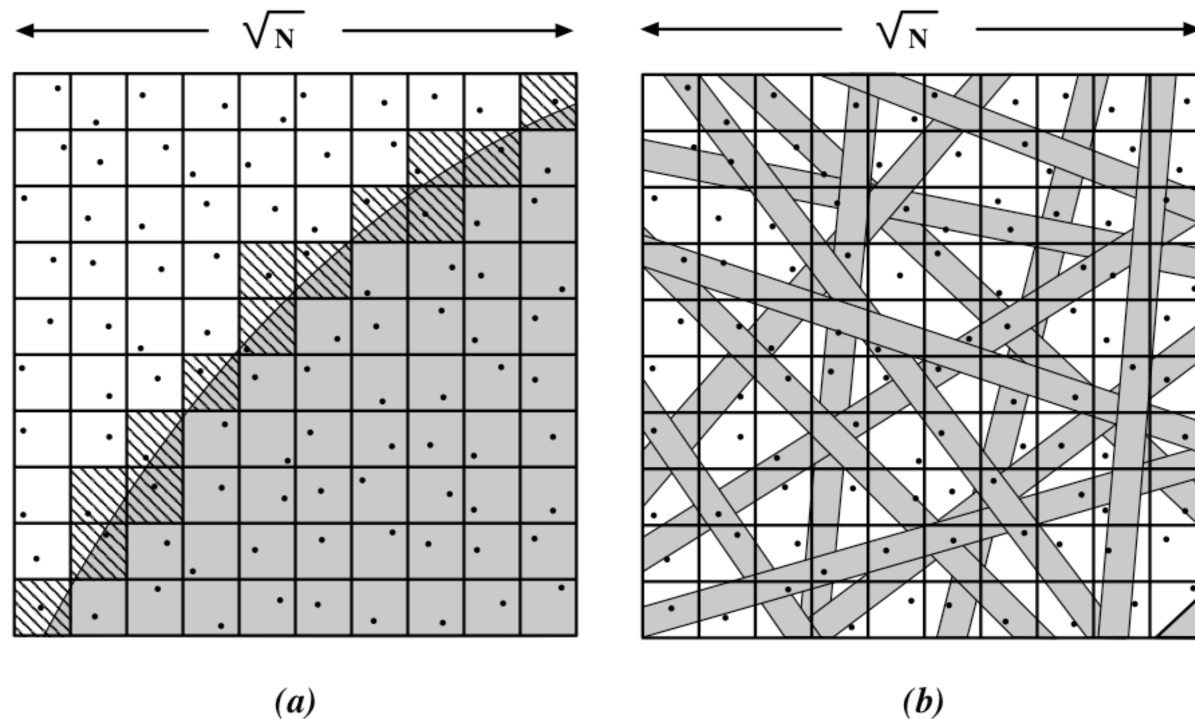
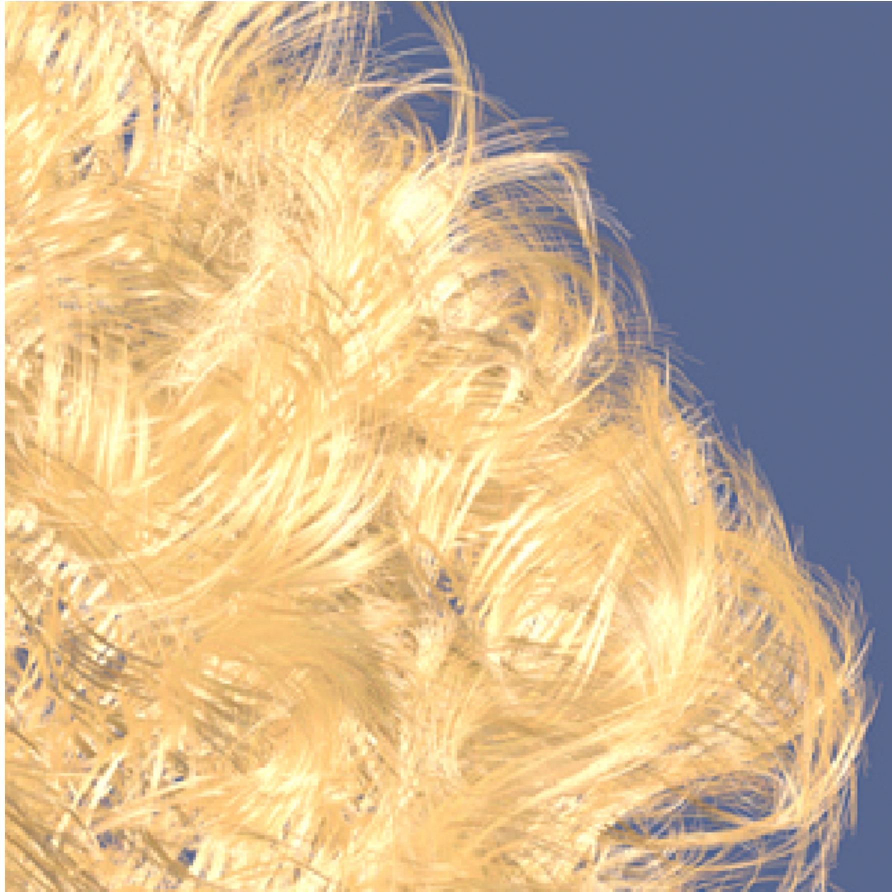


Figure 2: *Variance contributions to stratified sampling.* ***(a)*** *When a single silhouette edge passes through the filter region, $O(N^{1/2})$ samples contribute to the variance.* ***(b)*** *When the filter region is covered with fine geometry, all N samples contribute to the variance, resulting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

From: Tom Lokovic and Erich Veach, “*Deep Shadow Maps”, pp 385-392, Siggraph 2000.*

Importance of Shadows

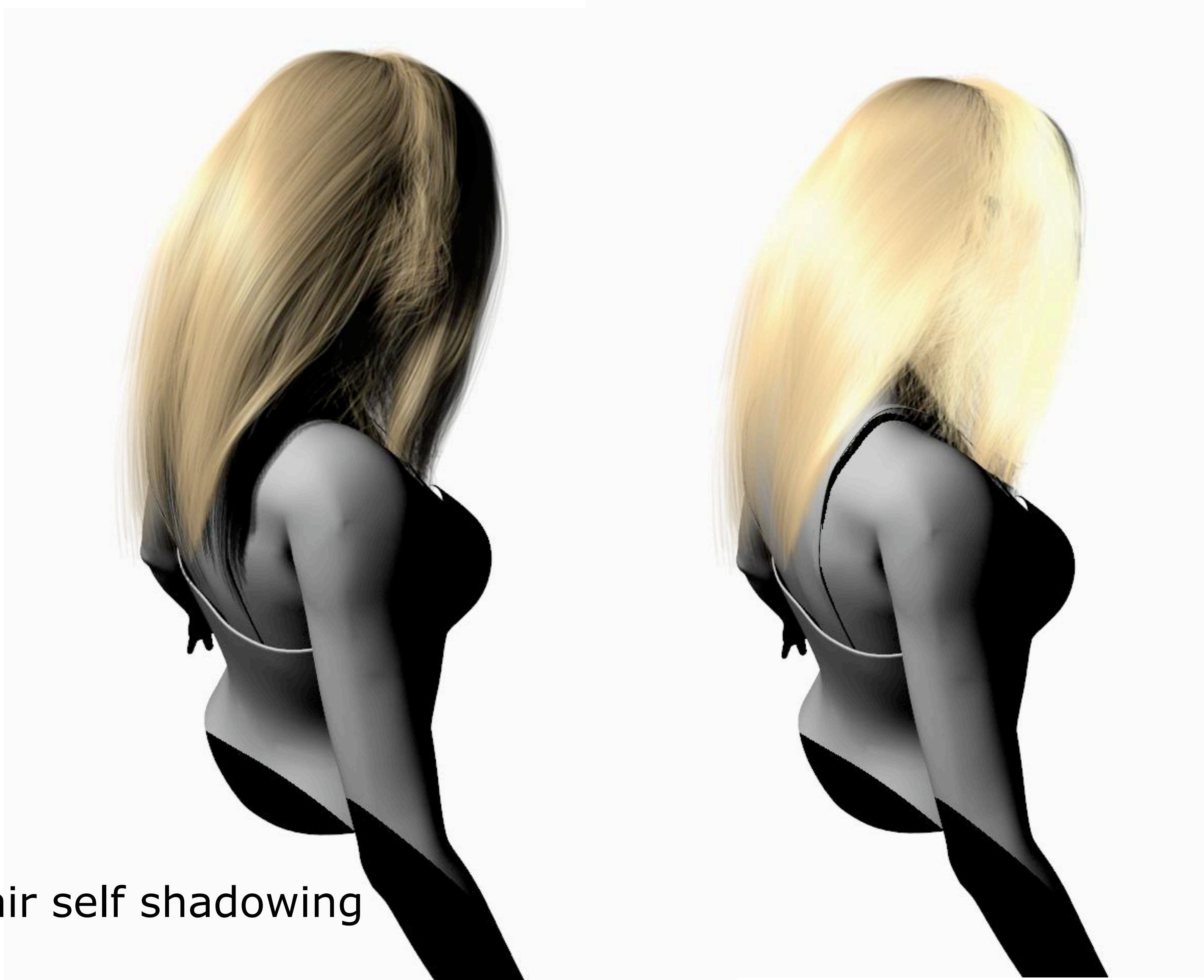


Images from: Tom Lokovic and Erich Veach, "*Deep Shadow Maps*", *pp 385-392, Siggraph 2000.*

Importance of Shadows

- The need for selfshadowing

Importance of Shadows



With hair self shadowing

Without hair self shadowing

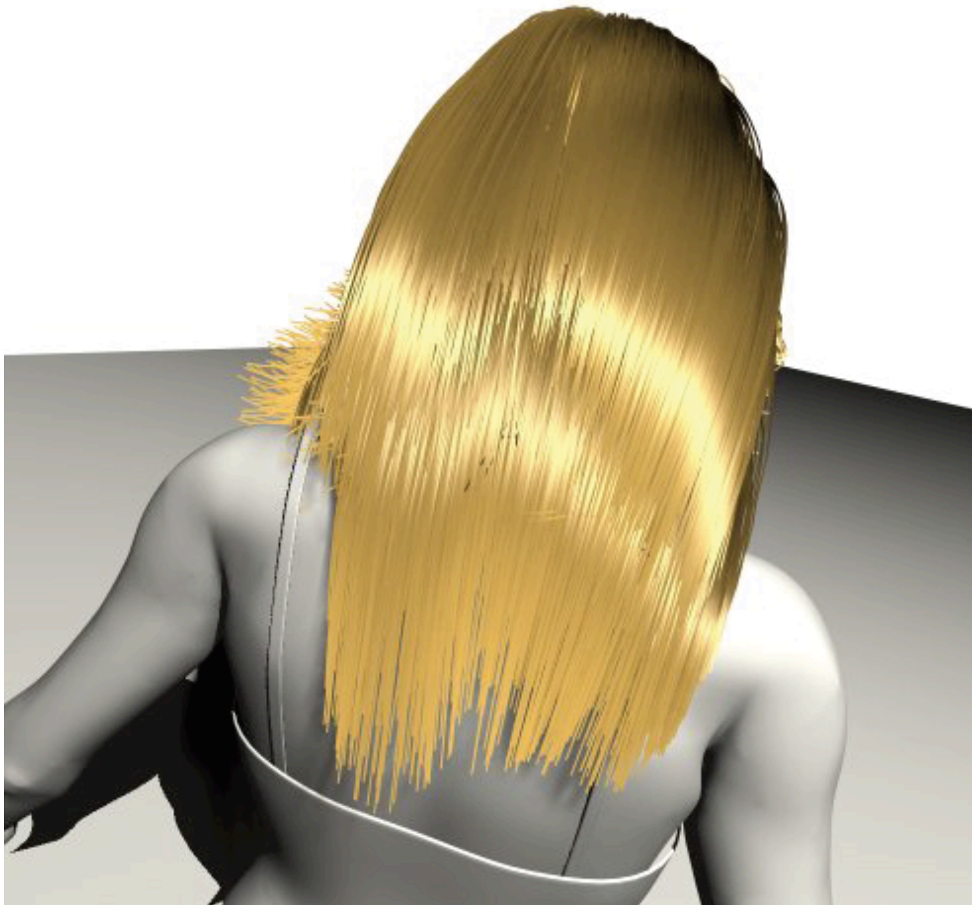
Importance of Transparency

- Hair is sub-pixel sized and transparent, alpha blending is absolutely necessary

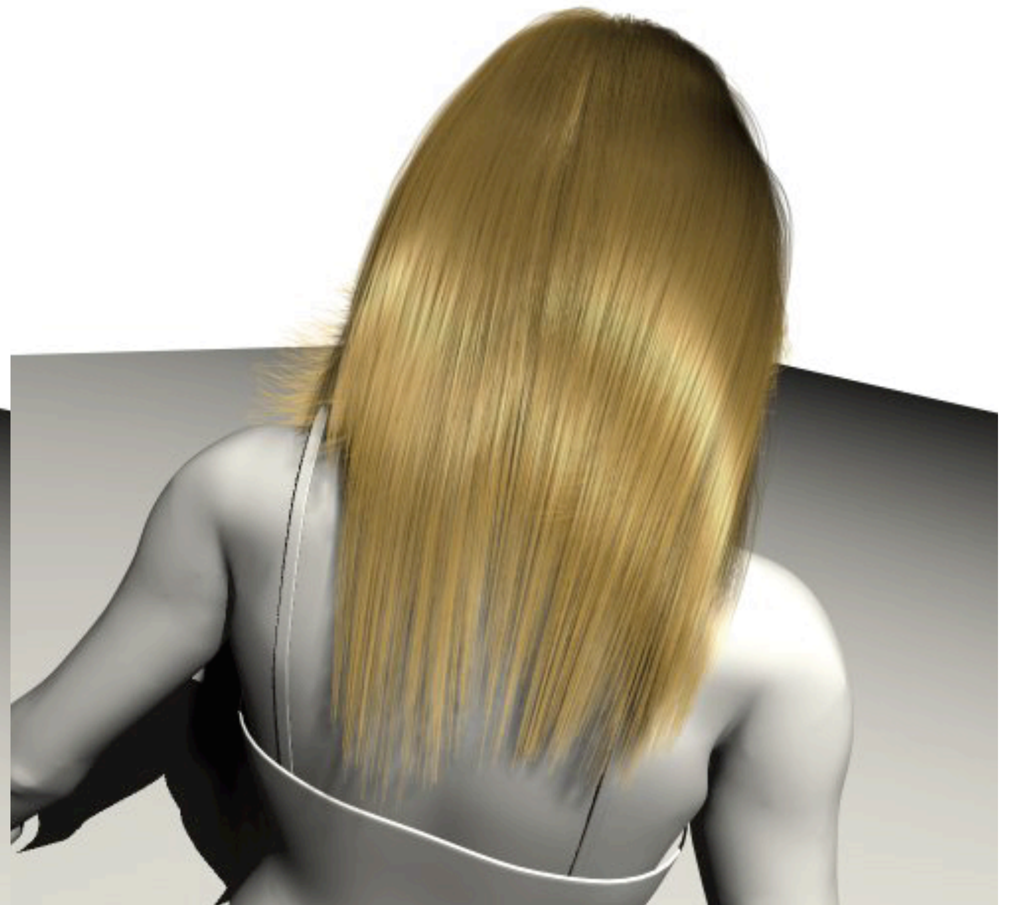
Without alpha blending

With alpha blending

Importance of Transparency



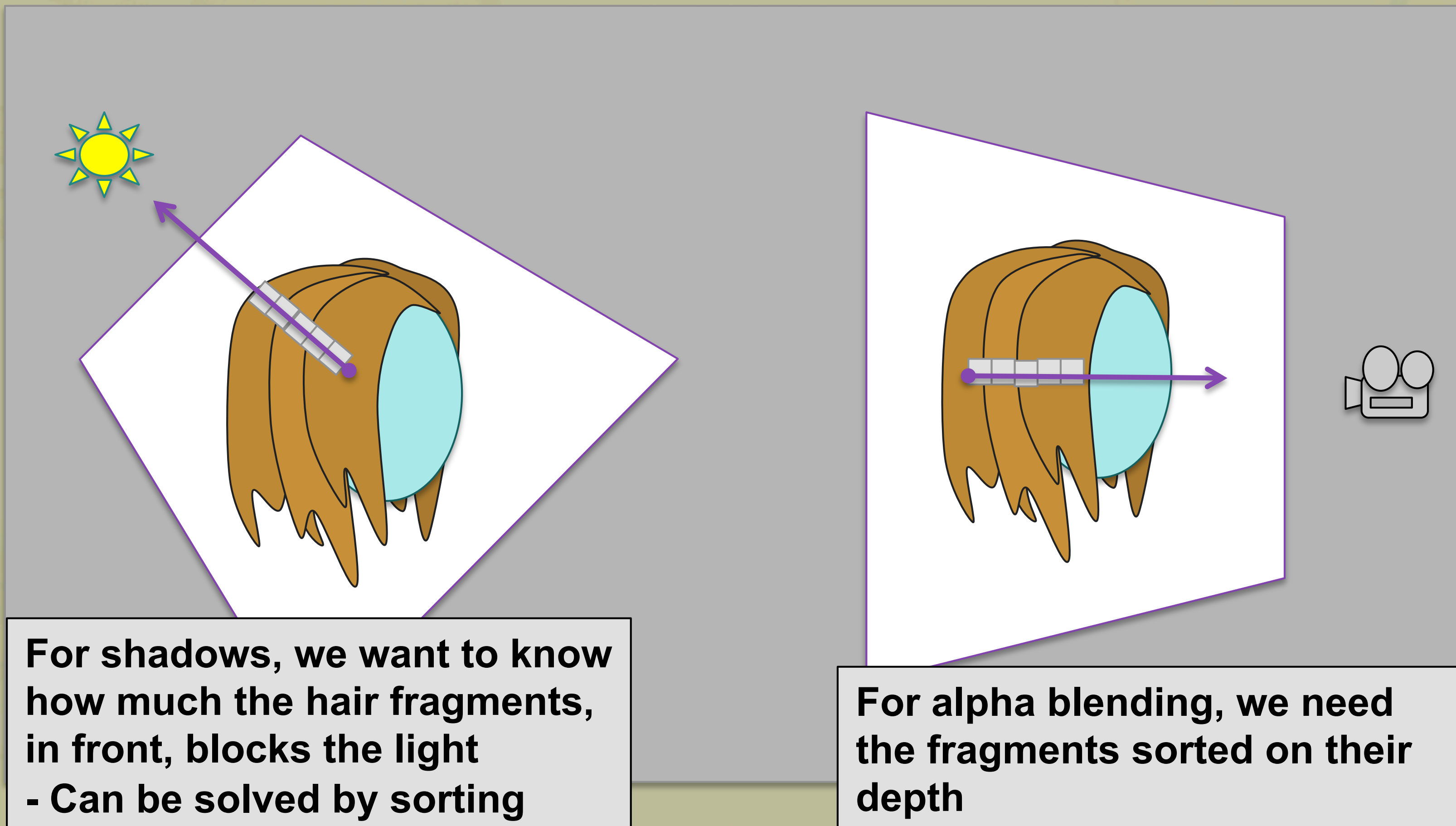
Hair rendered without alpha blending.



Hair rendered with alpha blending (= 0.2).

Real time hair rendering

The two problems are quite similar



Related Work

KAJIYA AND KAY. Rendering fur with threedimensional textures, SIGGRAPH 1989.

LOKOVIC AND VEACH. 2000. Deep shadow maps. In SIGGRAPH 2000.

MARSCHNER, JENSEN, CAMMARANO, WORLEY, AND HANRAHAN. Light scattering from human hair fibers. ACM Trans. Graph. 2003.

MERTENS, KAUTZ, BEKAERT, AND REETH. A self-shadow algorithm for dynamic hair using density clustering. In SIGGRAPH 2004 Sketches. 2004.

ZINKE, SOBOTTKA, AND WEBER. Photorealistic rendering of blond hair. In Vision, Modeling, and Visualization (VMV04), 2004.

NGUYEN AND DONELLY. Hair animation and rendering in the nalu demo. GPU Gems 2. 2005.

WARD, BERTAILS, KIM, MARSCHNER, CANI, AND LIN. A survey on hair modeling: Styling, simulation, and rendering. IEEE Transactions on Visualization and Computer Graphics 2007.

SINTORN, E., AND ASSARSSON, U. 2008. Real-time approximate sorting for self shadowing and transparency in hair rendering. In I3D 2008.

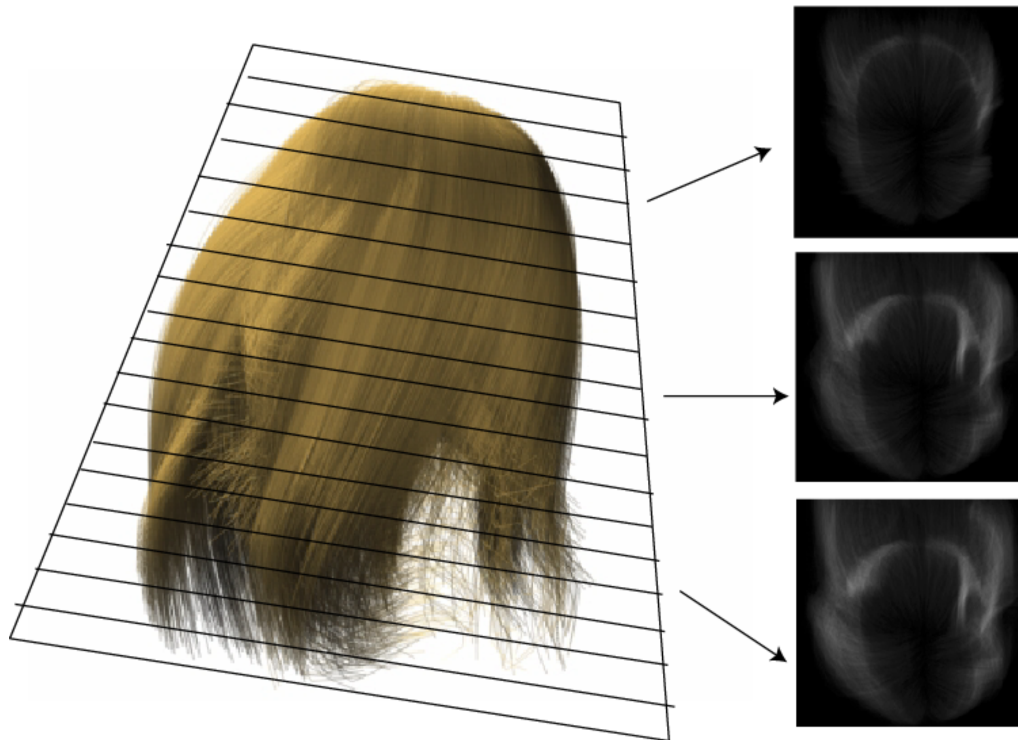
ZINKE, YUKSEL, WEBER, AND KEYSER. Dual scattering approximation for fast multiple scattering in hair. ACM Trans. Graph. 2008.

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
  - ⇒ Each texel = amount of shadow



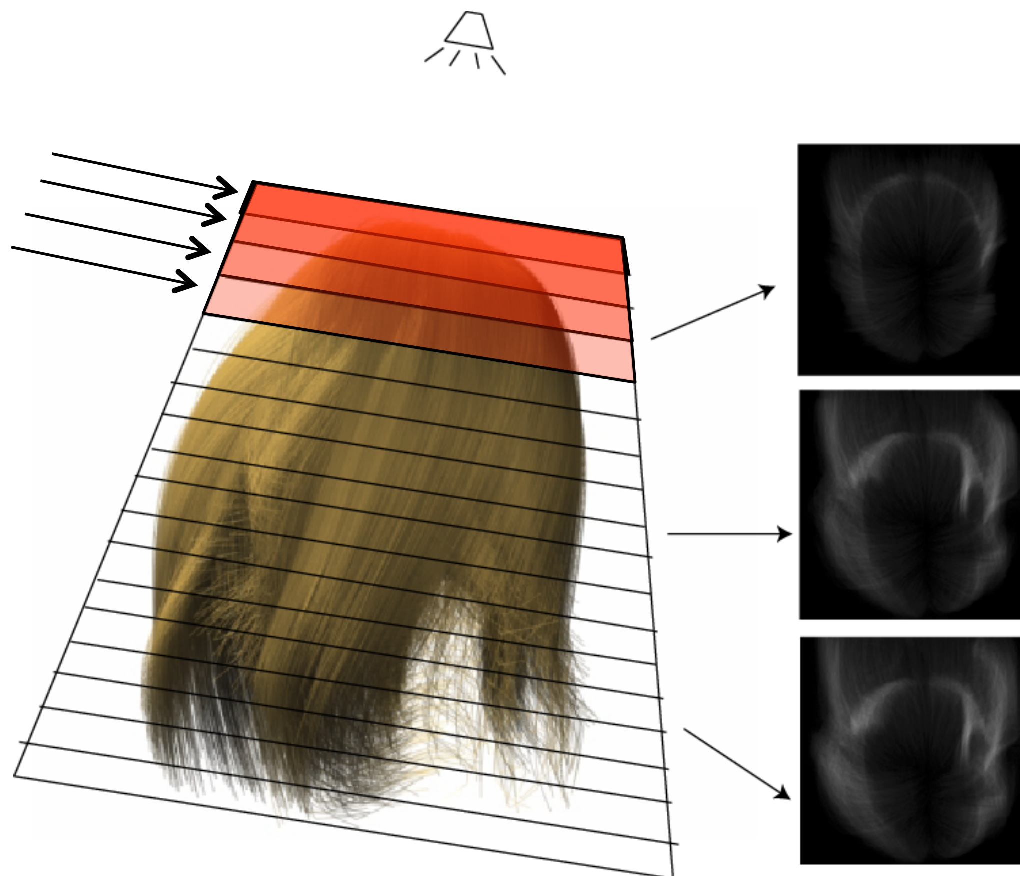
- Two classic options
  - Advance far plane per slice
  - Advance both near + far plane and copy result from previous slice.

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

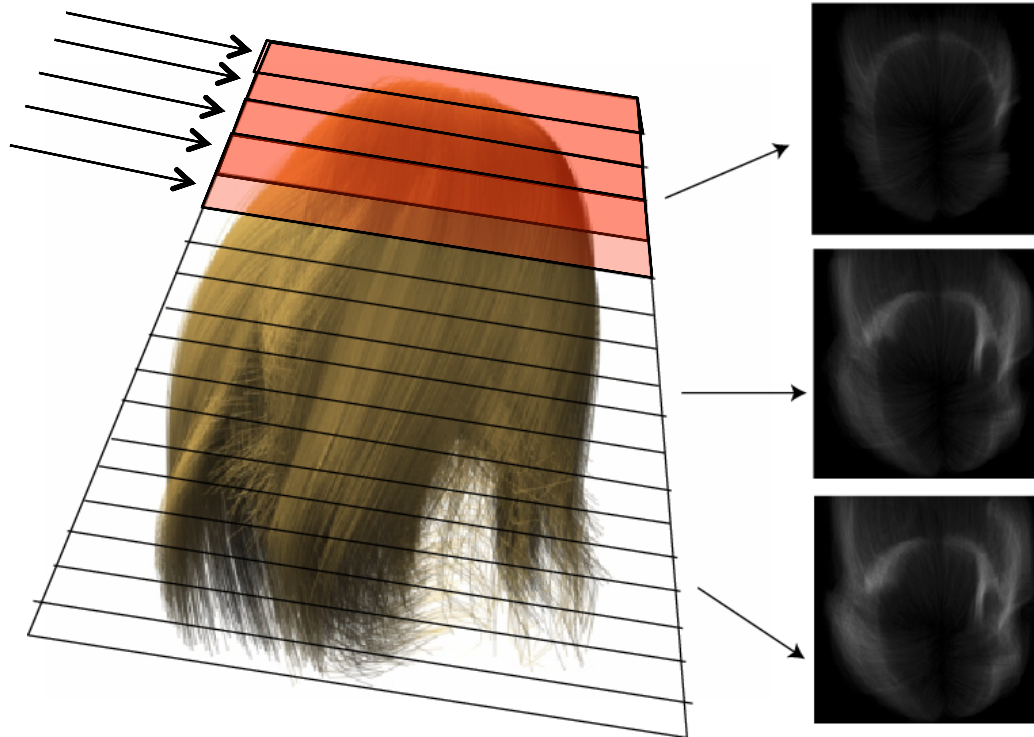


- Two classic options
  - Advance far plane per slice
  - Advance both near + far plane and copy result from previous slice.

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

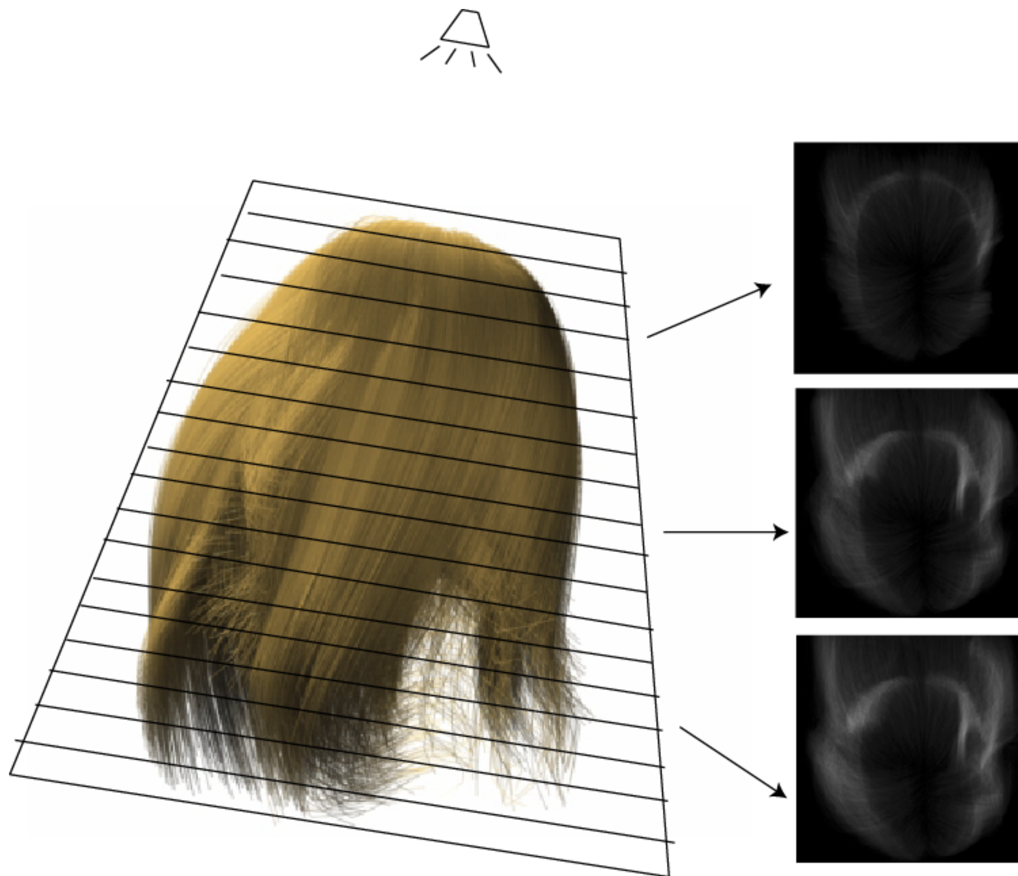


- Two classic options
  - Advance far plane per slice
  - Advance both near + far plane and copy result from previous slice.

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

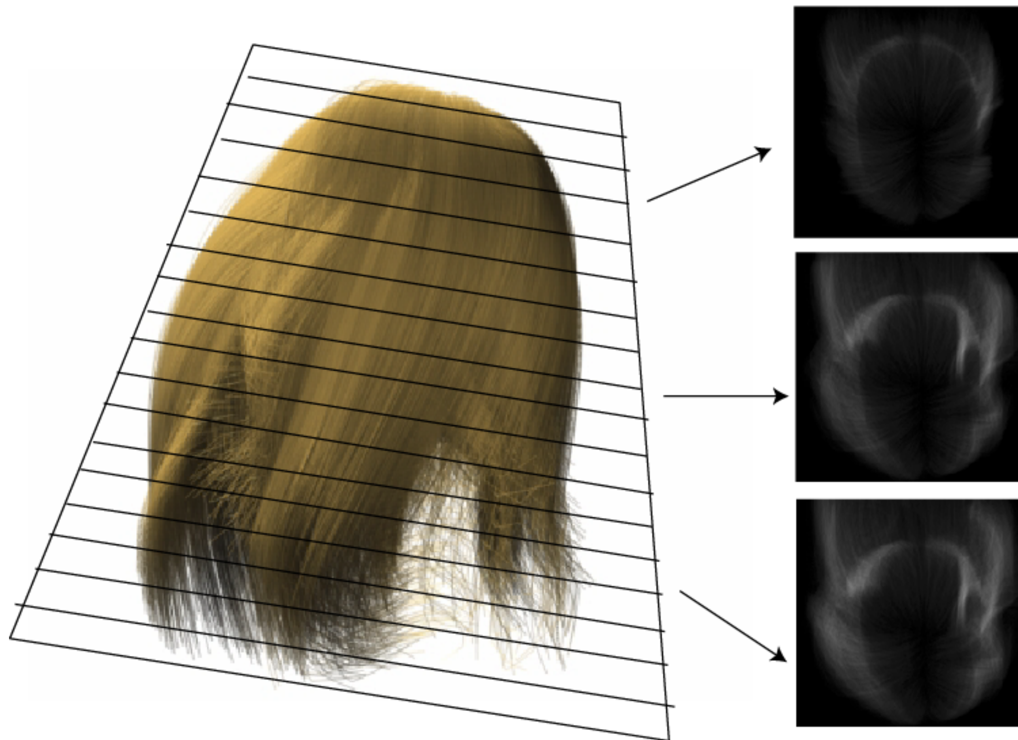


- Two classic options
  - Advance far plane per slice
  - Advance both near + far plane and copy result from previous slice.
- Disadvantage
  - All geometry sent for rendering for each slice

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.

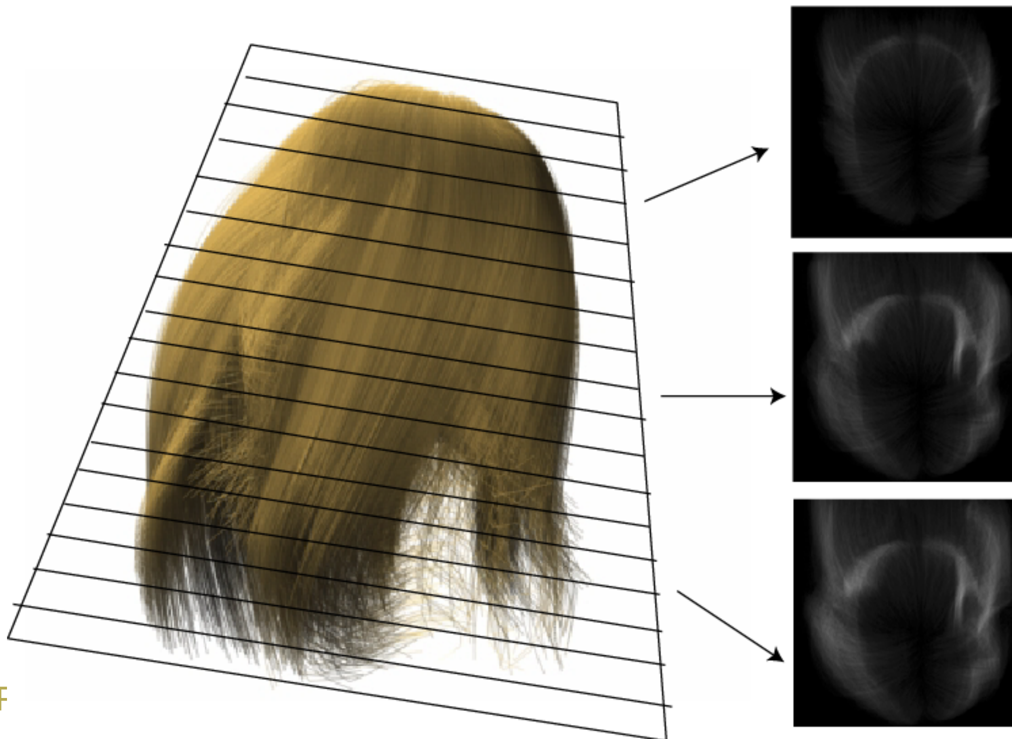
Opacity Maps

- 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

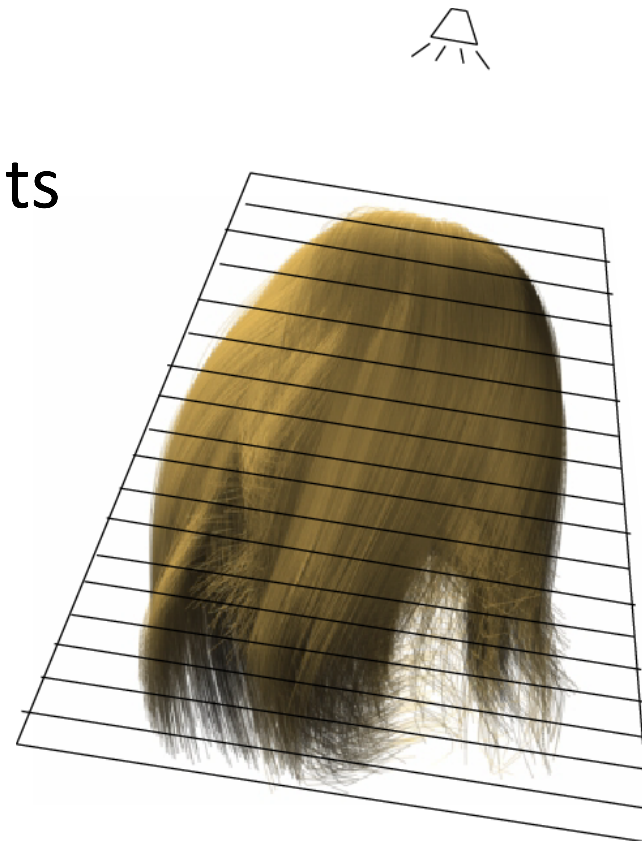


Partial-Radixsort

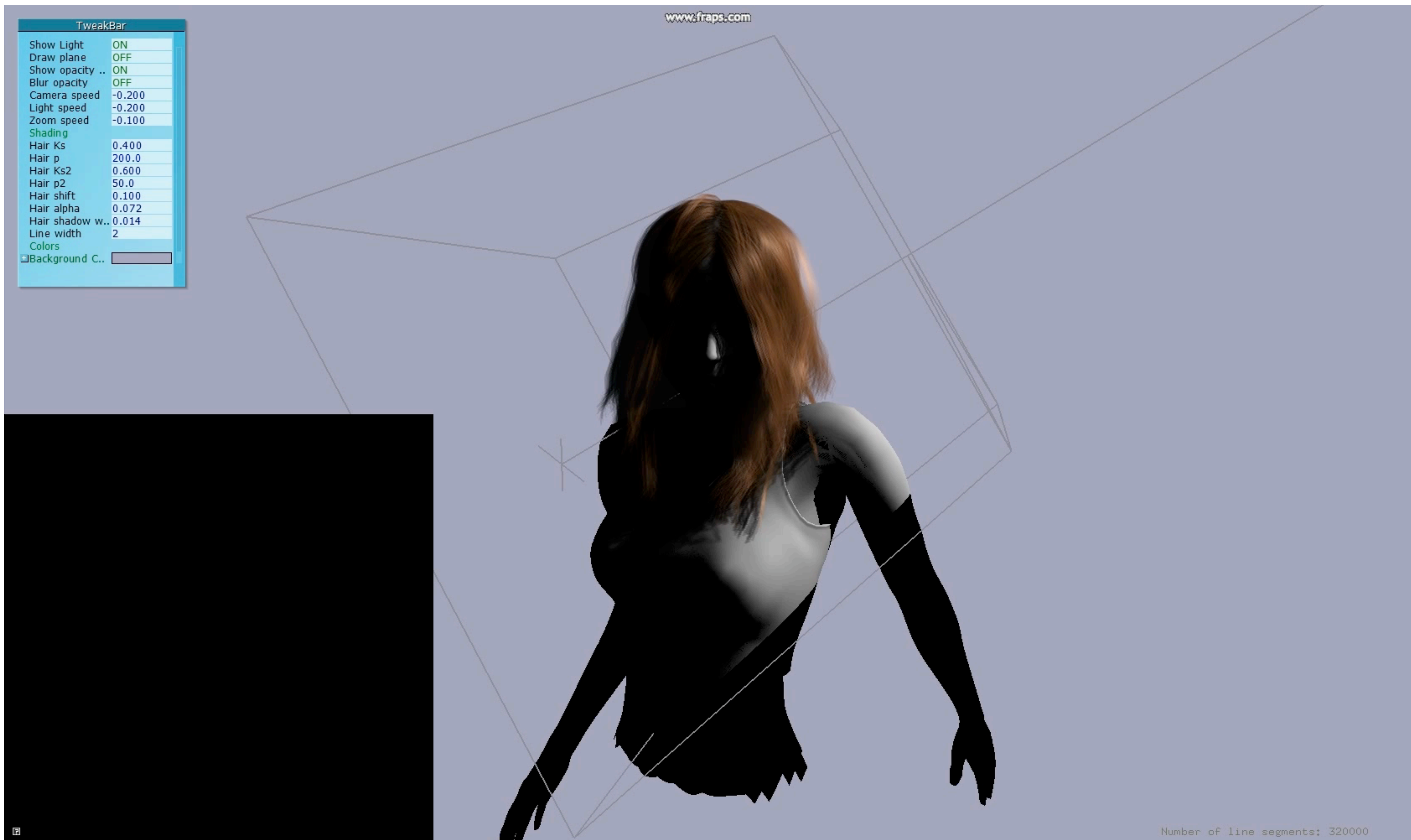
- Quick sort would require n-1 stream split calls
  - (not feasible)
  - Partial quick sort into 256 buckets requires 255 calls
    - Still quite expensive
- Radix sort of 32 bit numbers requires 32 stream split calls
- Partial radix sort into 256 buckets requires 8 calls
  - Fast

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*

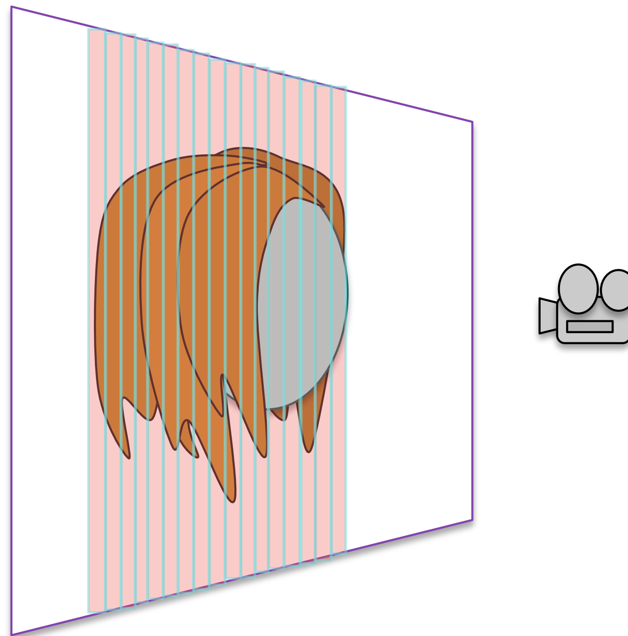


Building the Opacity Map Texture

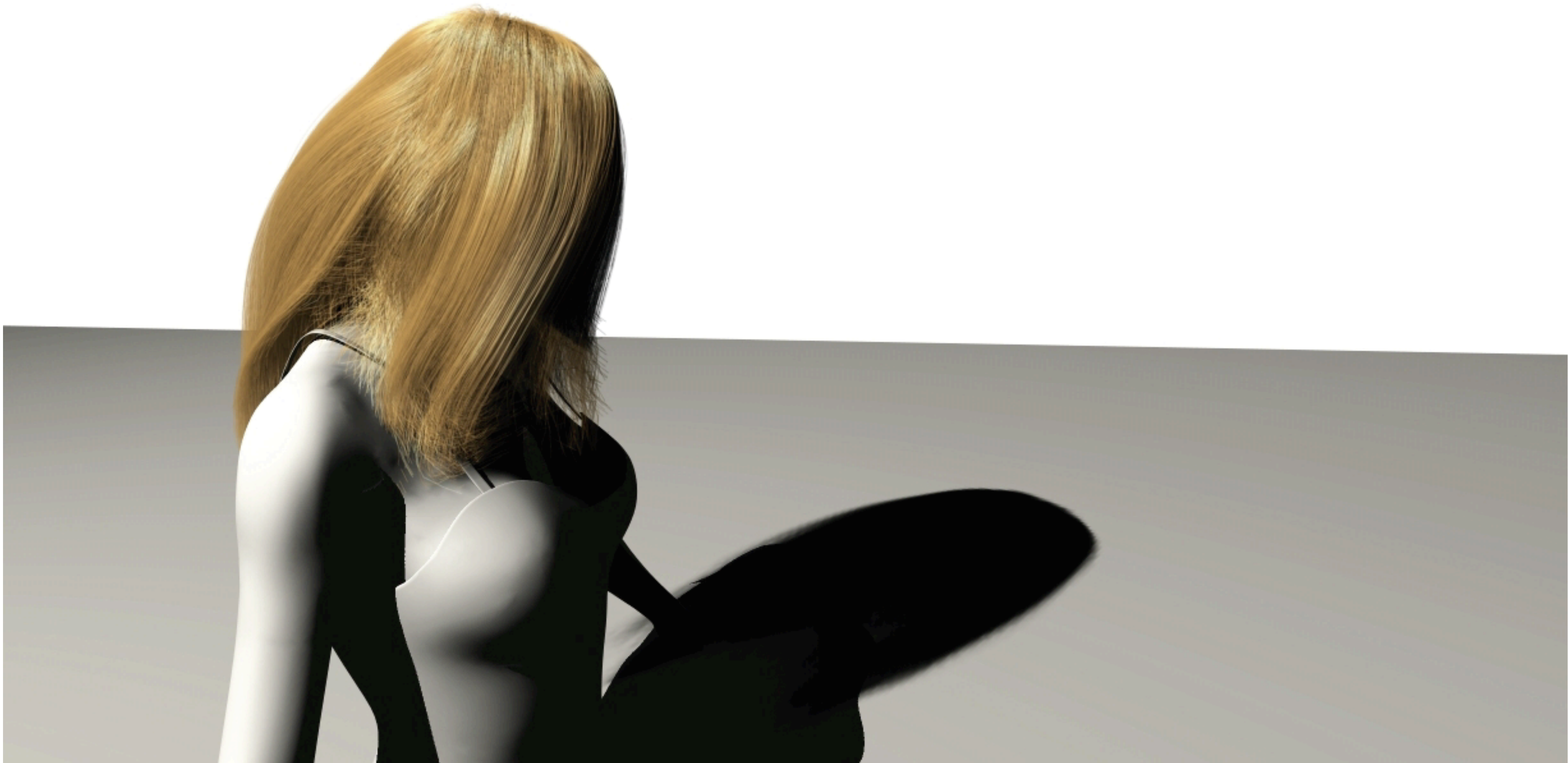


Alpha Sorting

- 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



Results



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 consumtion:
  - e.g. 512x512x256 = 64Mb
  - independent of #objects

- Solutions
  - NVIDIA’ GDC-presentation 2009:

    Advanced Visual Effects with Direct3D for PC, Cem Cebenoyan, Sarah Tariq, and Simon Green

Or

  - Erik Sintorn, Ulf Assarson. *Hair Self Shadowing and Transparency Depth Ordering Using Occupancy maps. I3D 2009.*

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

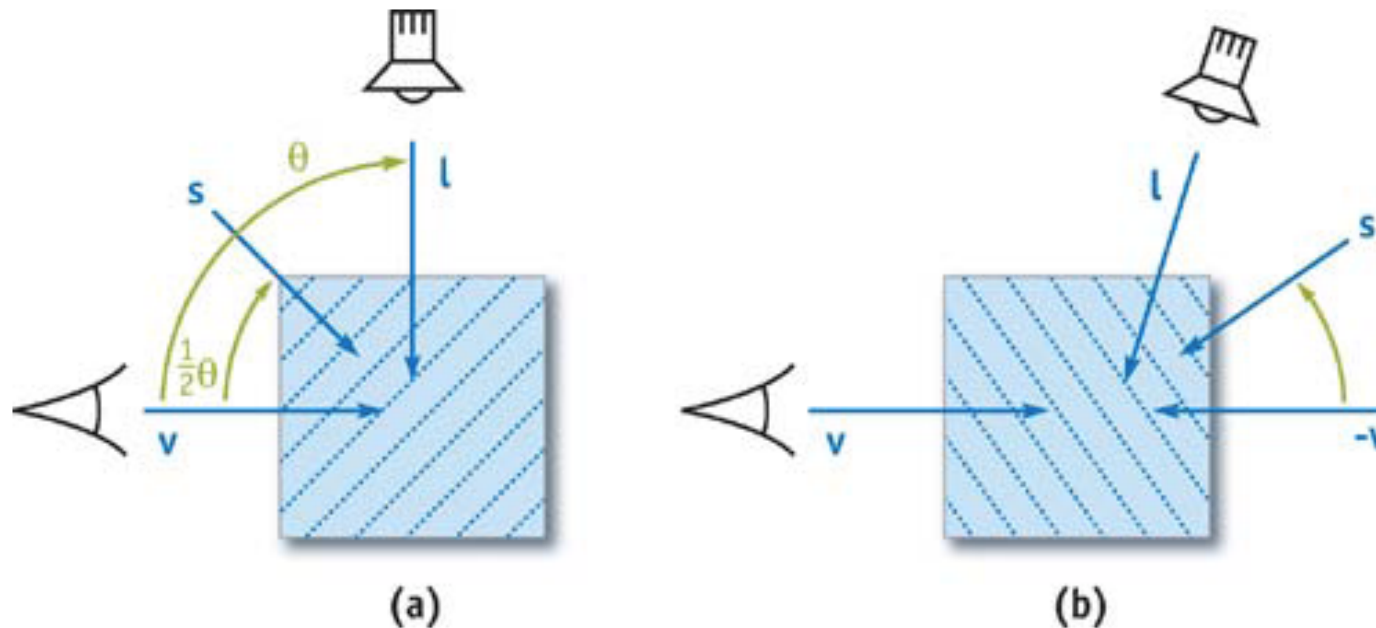


Image from: Volume Rendering Techniques, Milan Ikits, Joe Kniss, Aaron Lefohn, Charles Hansen. Chapter 39, section 39.5.1, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics(2004).*

Solution 1:

$f = \alpha\mathbf{c} + (1-\alpha)\mathbf{b}$

- Alpha blending either back-to-front **or** front-to-back
  - Render slices to screen using the 2D-shadow texture
    - If θ<90°, in front-to-back order
    - Else, in back-to-front order
  - Render slice into shadow texture

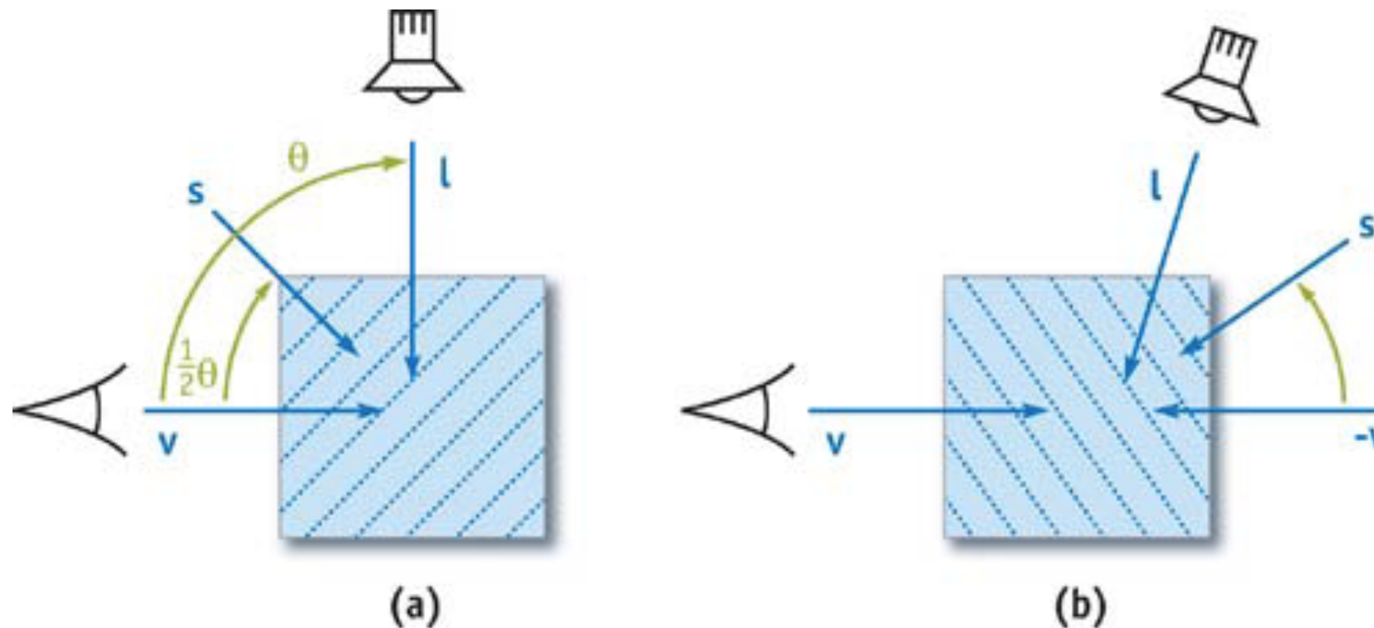
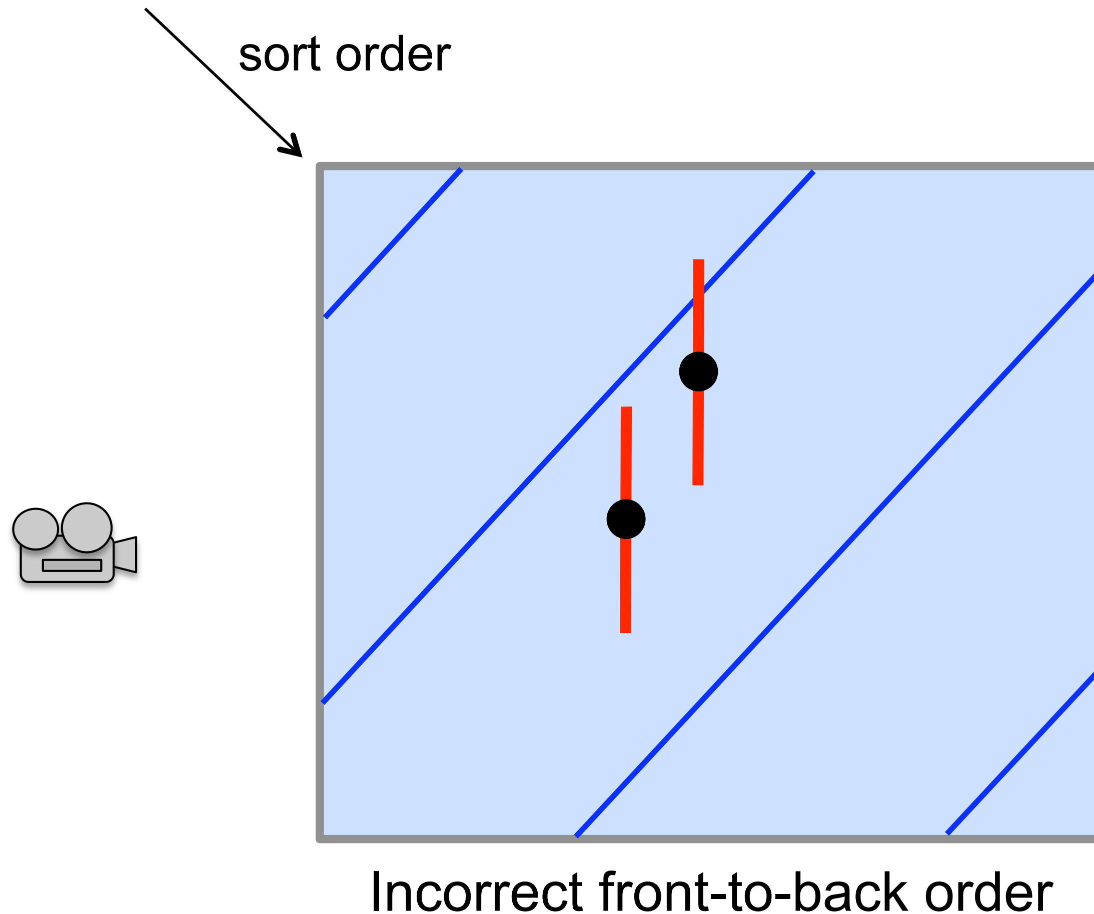


Image from: Volume Rendering Techniques, Milan Ikits, Joe Kniss, Aaron Lefohn, Charles Hansen. Chapter 39, section 39.5.1, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics(2004).*

Solution 1: Caveat

- Possible caveat for rectangles and lines

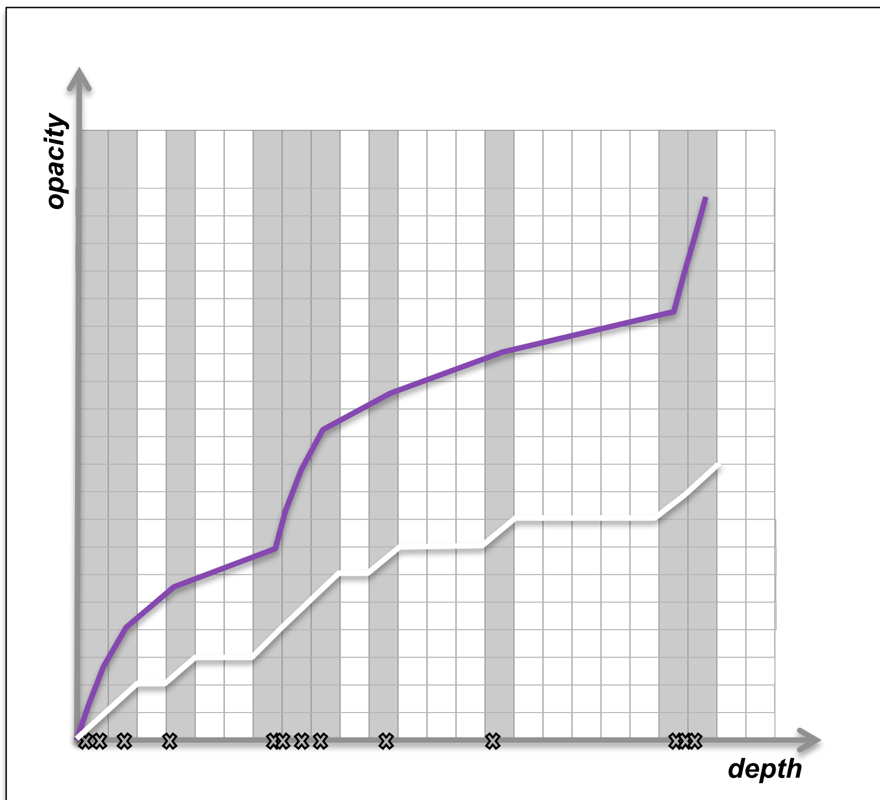


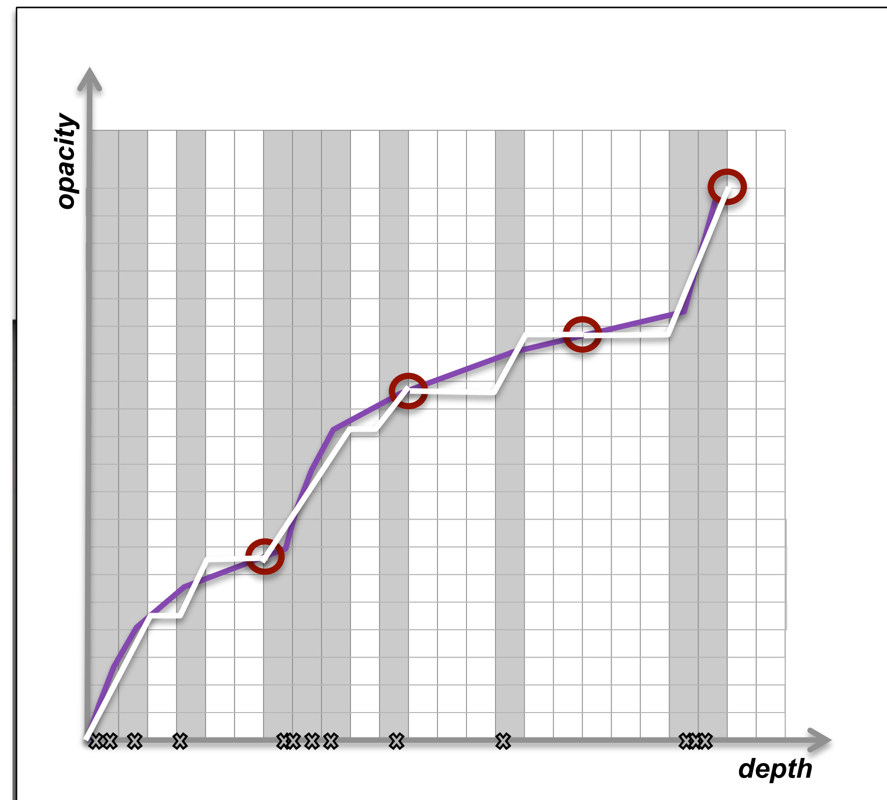
Incorrect front-to-back order

But the sorting only needs to be approximate anyway

Solution 2:

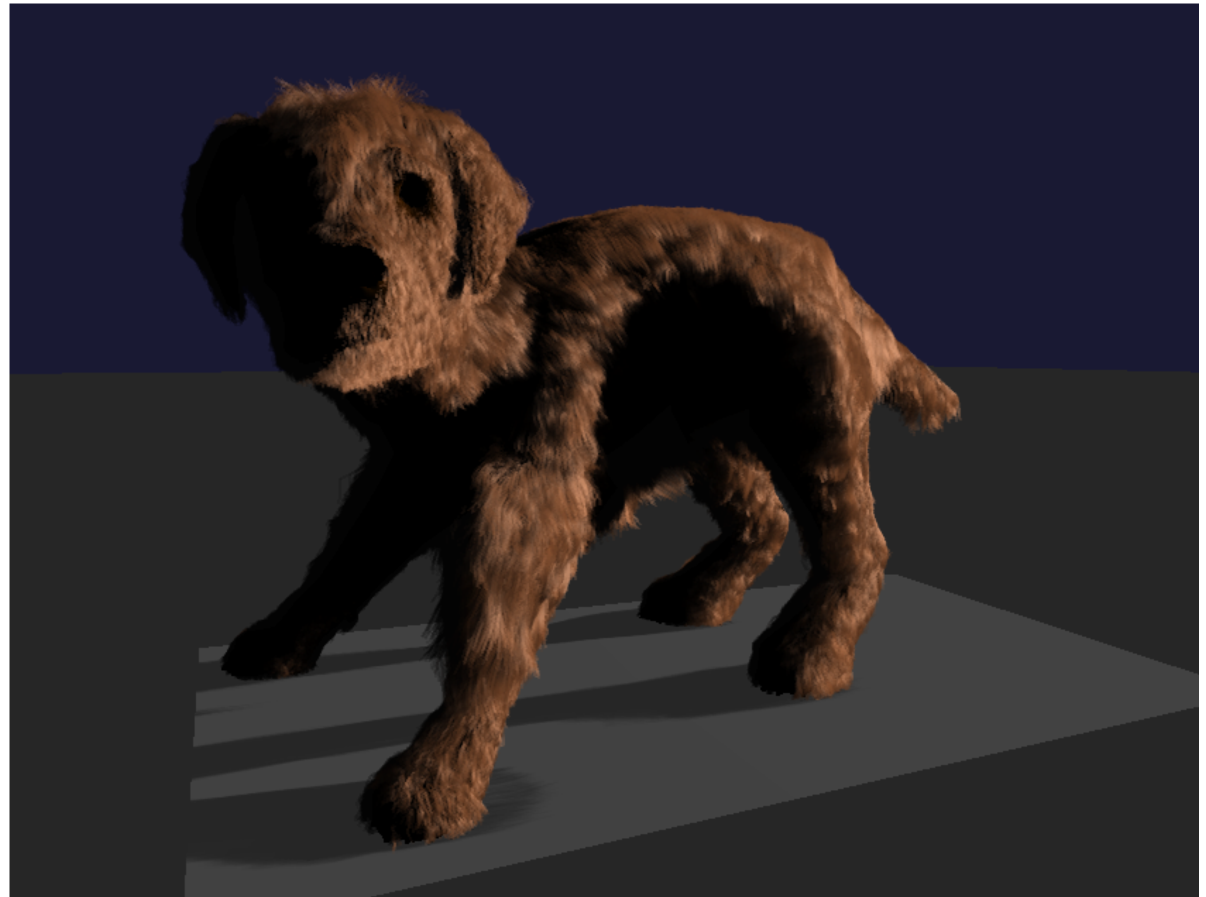
- If all hair strands have identical alpha-value:
  – Erik Sintorn, Ulf Assarson. *Hair Self Shadowing and Transparency Depth Ordering Using Occupancy maps. I3D 2009.*





Timings

Algorithm steps	*(ms)*
Sorting -incl both:	
Create key/value-pairs	3.2
Sort into buckets	8.1
Shadows:	
Create shadow-map	~0.1
Create opacity maps	12
Render:	
body with hair	0.16
hair with shadows	13.5
Total:	36.3 = 27 fps



The End

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

Aliasing

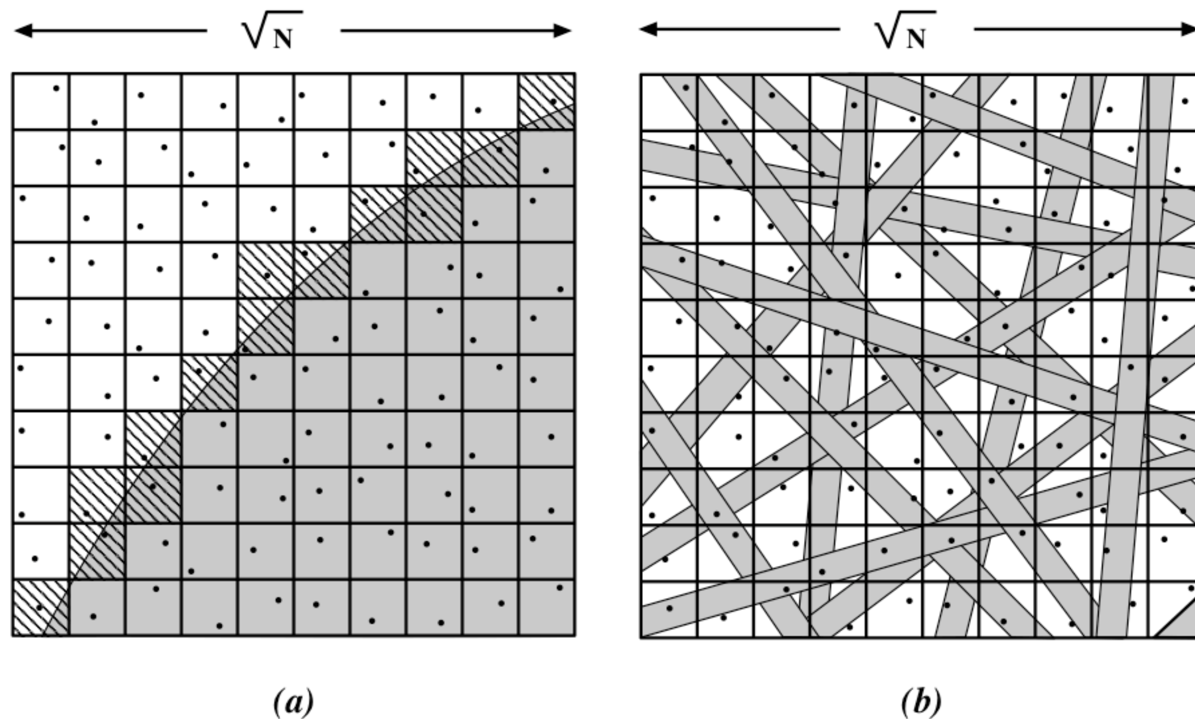
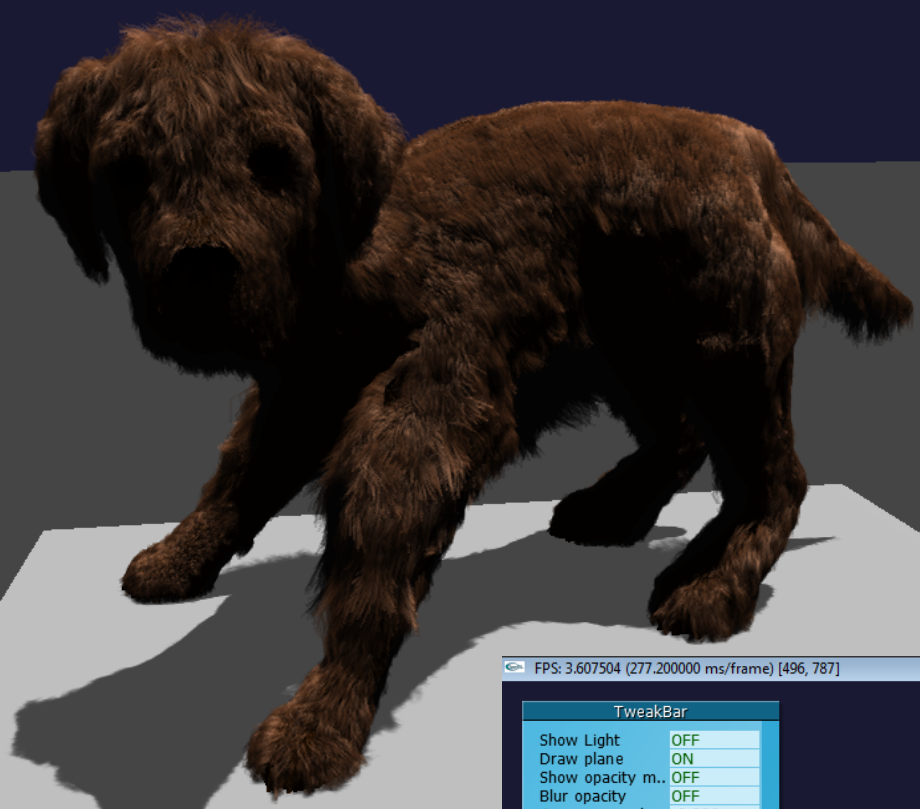


Figure 2: *Variance contributions to stratified sampling.* ***(a)*** *When a single silhouette edge passes through the filter region,* $O(N^{1/2})$ *samples contribute to the variance.* ***(b)*** *When the filter region is covered with fine geometry, all* N *samples contribute to the variance, resulting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



FPS: 3.607504 (277.200000 ms/frame) [496, 787]
TweakBar
Show Light OFF
Draw plane ON
Show opacity m.. OFF
Blur opacity OFF
Camera speed -0.200
Light speed -0.400
Zoom speed -0.100
Shading
Hair Ks 0.300
Hair p 50.0
Hair Ks2 0.500
Hair p2 20.0
Hair shift 0.100
Hair alpha 0.150
Hair shadow we.. 0.000
Line width 2
Colors
Background Co..
Hair Color
Model another.e..