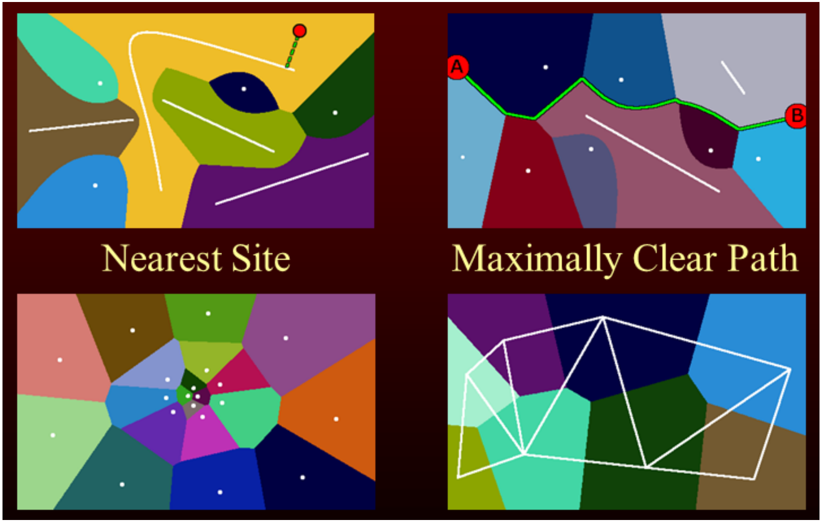
GPU Algorithms I
The Early Years

MADALGO Summer School on Algorithms for Modern Parallel and Distributed Models

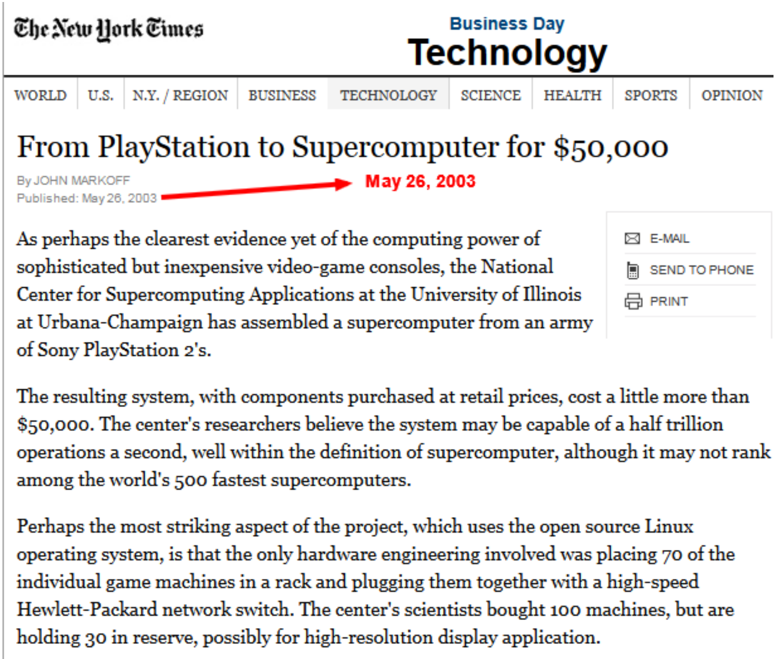
Suresh Venkatasubramanian
University of Utah

Motivation



Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware [SIGGRAPH 1999]

Motivation



The New York Times

Business Day

Technology

WORLD | U.S. | N.Y. / REGION | BUSINESS | TECHNOLOGY | SCIENCE | HEALTH | SPORTS | OPINION

From PlayStation to Supercomputer for $50,000

By JOHN MARKOFF
Published: May 26, 2003

May 26, 2003

E-MAIL
SEND TO PHONE
PRINT

As perhaps the clearest evidence yet of the computing power of sophisticated but inexpensive video-game consoles, the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign has assembled a supercomputer from an army of Sony PlayStation 2's.

The resulting system, with components purchased at retail prices, cost a little more than $50,000. The center's researchers believe the system may be capable of a half trillion operations a second, well within the definition of supercomputer, although it may not rank among the world's 500 fastest supercomputers.

Perhaps the most striking aspect of the project, which uses the open source Linux operating system, is that the only hardware engineering involved was placing 70 of the individual game machines in a rack and plugging them together with a high-speed Hewlett-Packard network switch. The center's scientists bought 100 machines, but are holding 30 in reserve, possibly for high-resolution display application.

Animusic Demo

Overview

- GPUs today have 10s of cores (soon, 100s !)
- Have huge data bandwidth (100s of GB/s)
- Force a SIMD-style mode of computation
- HPC on the cheap !

However,

- GPU models constantly changing
- Almost no algorithmic work on GPUs
- Disconnect between programming model and execution model

Overview

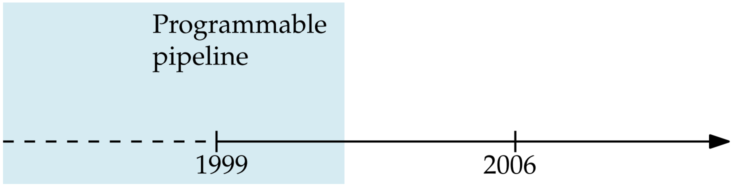
- GPUs today have 10s of cores (soon, 100s !)
- Have huge data bandwidth (100s of GB/s)
- Force a SIMD-style mode of computation
- HPC on the cheap !

However,

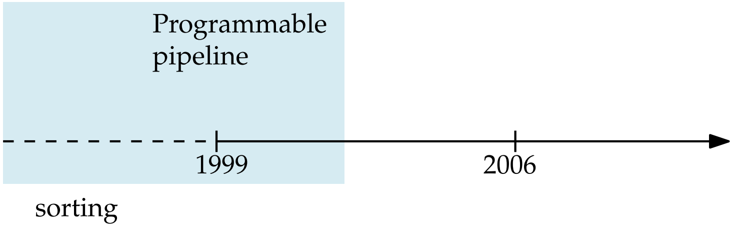
- GPU models constantly changing
- Almost no algorithmic work on GPUs
- Disconnect between programming model and execution model

No proofs!

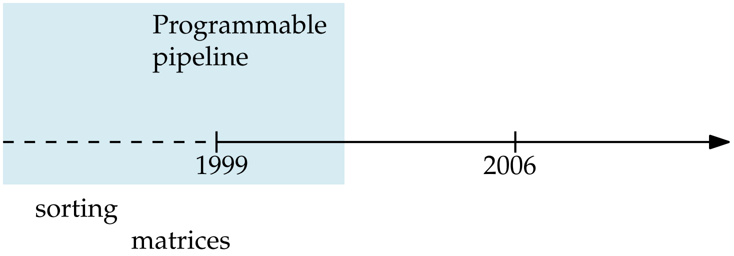
Outline



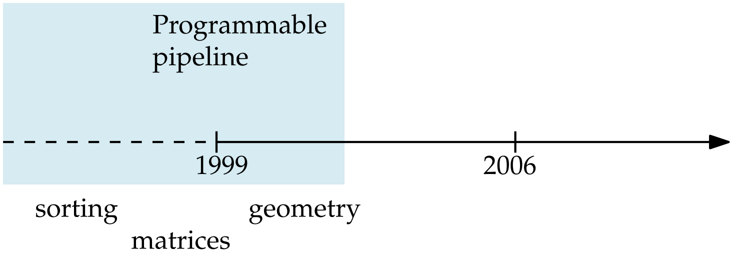
Outline



Outline

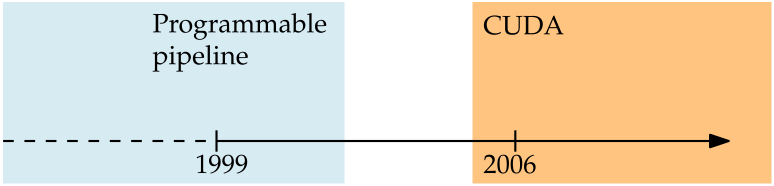


Outline



A streaming model

Outline



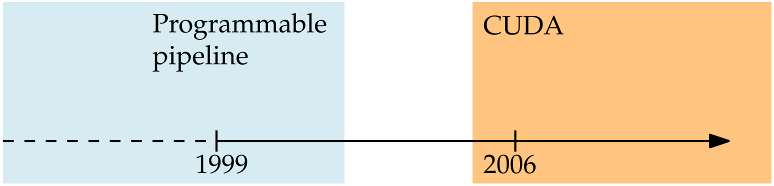
sorting

matrices

geometry

A streaming model

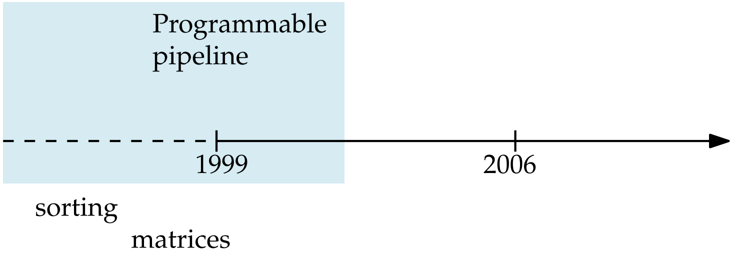
Outline



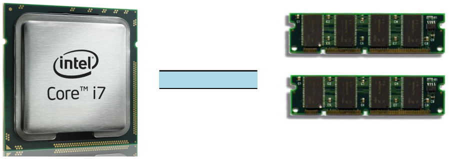
sorting matrices geometry sorting matrices graphs

A streaming model

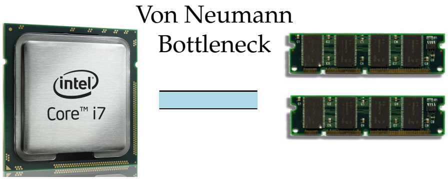
This Lecture

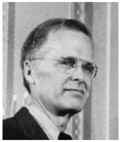


The von Neumann Bottleneck



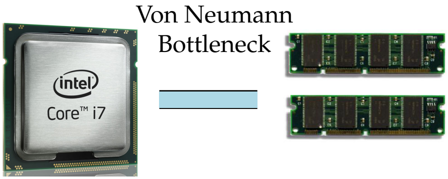
The von Neumann Bottleneck

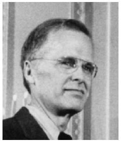




Coined by John Backus

The von Neumann Bottleneck

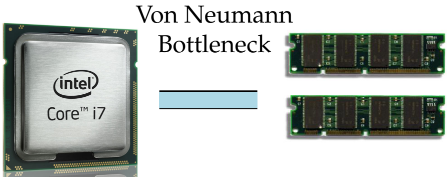


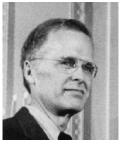


Coined by John Backus

Has always been a problem (regardless of technology)

The von Neumann Bottleneck



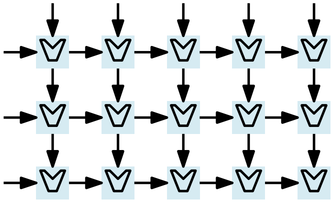


Coined by John Backus

Has always been a problem (regardless of technology)

GPU design is an attempt to get around it

Addressing the Bottleneck



- Systolic arrays move data between a grid of processing elements.

Addressing the Bottleneck

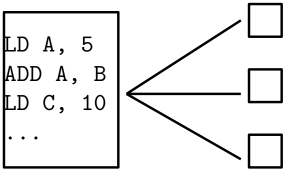
- Systolic arrays move data between a grid of processing elements.
- Vector processors operate on multiple *words* at a time

Addressing the Bottleneck

```
LD A, 5
ADD A, B
LD C, 10
...
```

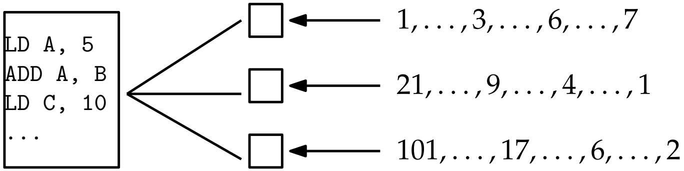
- Systolic arrays move data between a grid of processing elements.
- Vector processors operate on multiple *words* at a time
- SIMD processors have a fixed program that runs on different streams

Addressing the Bottleneck



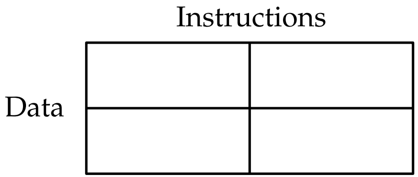
- Systolic arrays move data between a grid of processing elements.
- Vector processors operate on multiple *words* at a time
- SIMD processors have a fixed program that runs on different streams

Addressing the Bottleneck



- Systolic arrays move data between a grid of processing elements.
- Vector processors operate on multiple *words* at a time
- SIMD processors have a fixed program that runs on different streams

Addressing the Bottleneck



Flynn's taxonomy

- Systolic arrays move data between a grid of processing elements.
- Vector processors operate on multiple *words* at a time
- SIMD processors have a fixed program that runs on different streams

Addressing the Bottleneck

	Instructions	
Data	SISD	MISD
	SIMD	MIMD

Flynn’s taxonomy

- Systolic arrays move data between a grid of processing elements.
- Vector processors operate on multiple *words* at a time
- SIMD processors have a fixed program that runs on different streams

Addressing the Bottleneck

Instructions

Data	SISD	MISD
	SIMD	MIMD

Flynn’s taxonomy

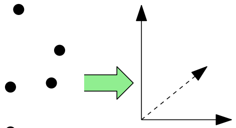
- Systolic arrays move data between a grid of processing elements.
- Vector processors operate on multiple *words* at a time
- SIMD processors have a fixed program that runs on different streams

GPU Basics

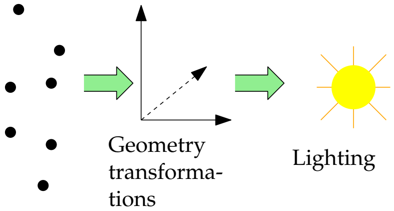
The GPU Pipeline

-
    -
-
    -
-
    -
    -

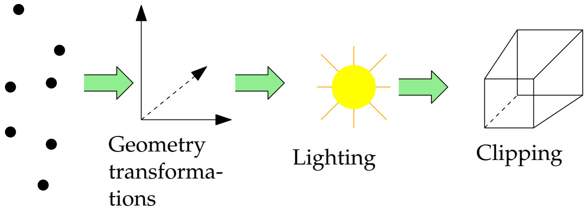
The GPU Pipeline



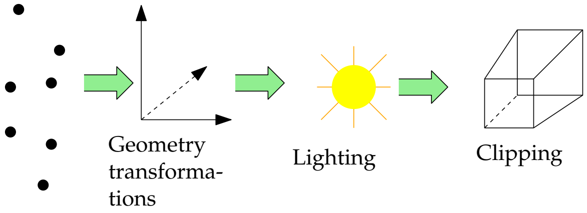
The GPU Pipeline



The GPU Pipeline

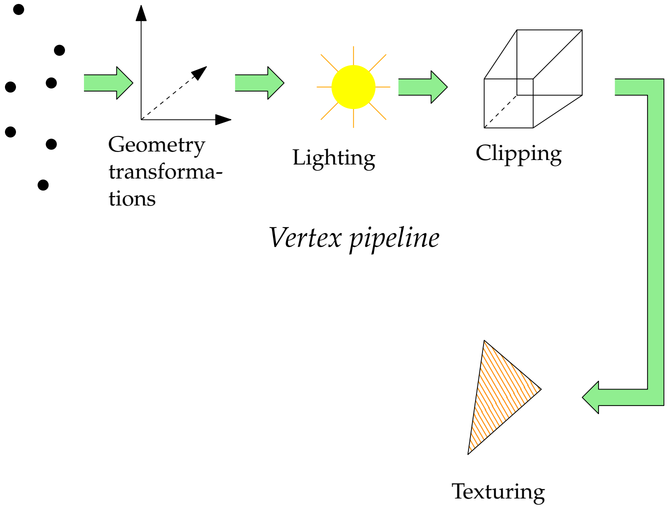


The GPU Pipeline

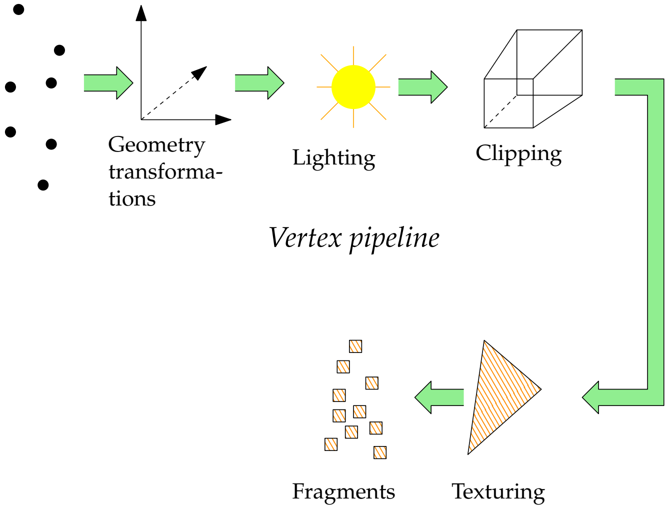


Vertex pipeline

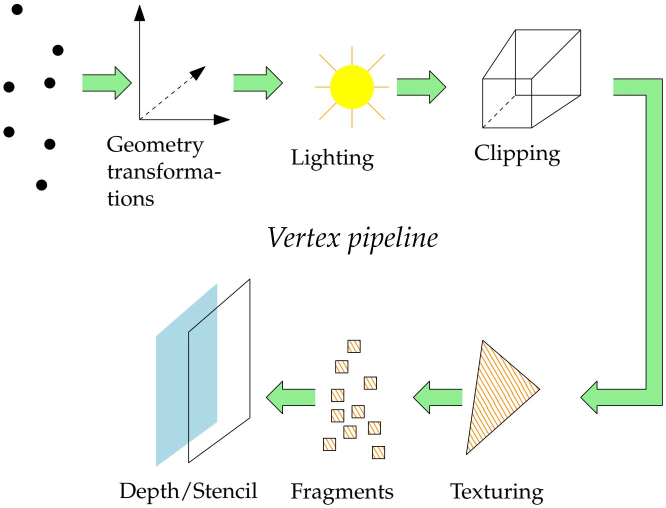
The GPU Pipeline



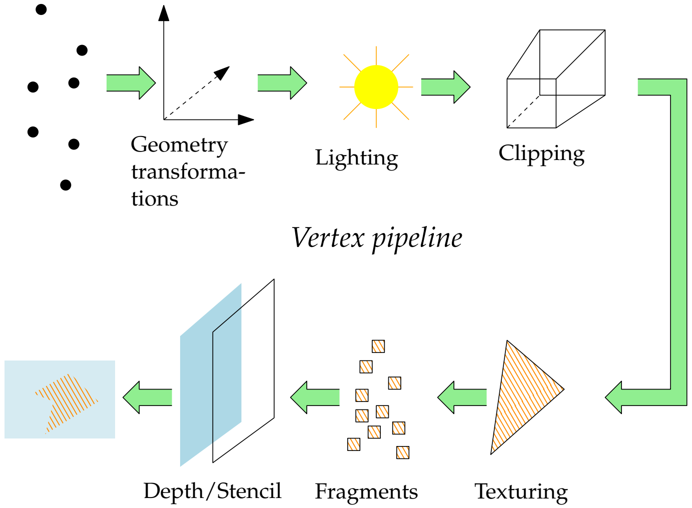
The GPU Pipeline



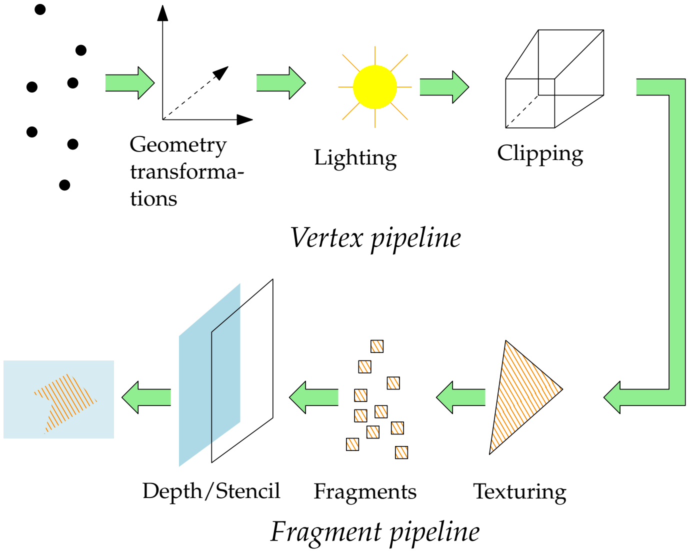
The GPU Pipeline



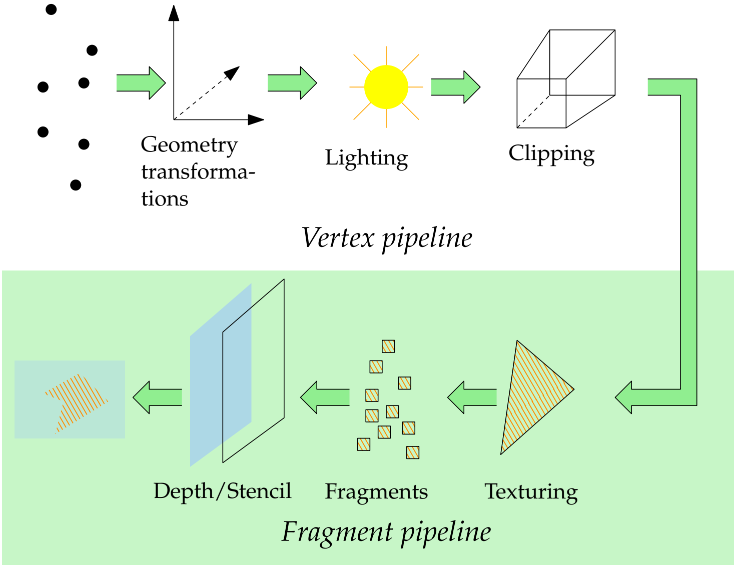
The GPU Pipeline



The GPU Pipeline



The GPU Pipeline



Fragment shader operations

- *Every pixel* acts like an SIMD processor

Fragment shader operations

- *Every pixel* acts like an SIMD processor
    - actually, some fixed number are processed in parallel

Fragment shader operations

- *Every pixel* acts like an SIMD processor
  - actually, some fixed number are processed in parallel
- Fragment processor could perform simple *straight-line operations* and conditionals (no looping)

Fragment shader operations

- *Every pixel* acts like an SIMD processor
  - actually, some fixed number are processed in parallel
- Fragment processor could perform simple *straight-line operations* and conditionals (no looping)
- (limited) texture memory for local storage

Fragment shader operations

- *Every pixel* acts like an SIMD processor
    - actually, some fixed number are processed in parallel
- Fragment processor could perform simple *straight-line operations* and conditionals (no looping)
- (limited) texture memory for local storage
- Each pixel processor could do a simple `reduce` (`add`, `blend`)

Fragment shader operations

- *Every pixel* acts like an SIMD processor
  - actually, some fixed number are processed in parallel
- Fragment processor could perform simple *straight-line operations* and conditionals (no looping)
- (limited) texture memory for local storage
- Each pixel processor could do a simple `reduce` (`add`, `blend`)
- Computation initiated by “rendering call” from host machine.

Fragment shader operations

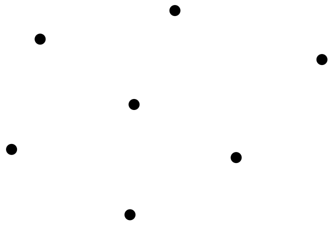
- *Every pixel* acts like an SIMD processor
  - actually, some fixed number are processed in parallel
- Fragment processor could perform simple *straight-line operations* and conditionals (no looping)
- (limited) texture memory for local storage
- Each pixel processor could do a simple `reduce` (`add`, `blend`)
- Computation initiated by "rendering call" from host machine.
- All computation resides on GPU from start of the vertex pipeline

Fragment shader operations

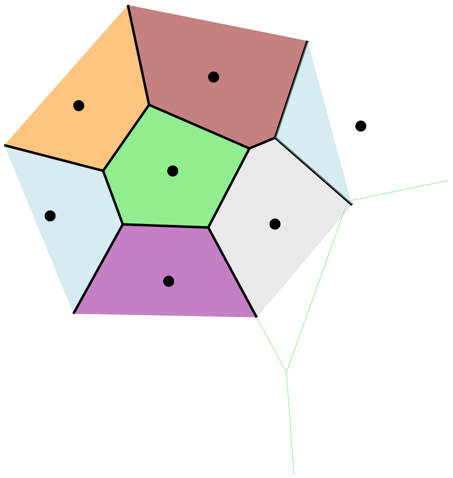
- *Every pixel* acts like an SIMD processor
    - actually, some fixed number are processed in parallel
- Fragment processor could perform simple *straight-line operations* and conditionals (no looping)
- (limited) texture memory for local storage
- Each pixel processor could do a simple `reduce` (`add`, `blend`)
- Computation initiated by "rendering call" from host machine.
- All computation resides on GPU from start of the vertex pipeline
- Computation proceeds in *passes*: output could be rendered or stored in memory for next pass.

Simple GPU Algorithms

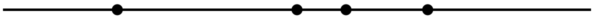
Voronoi Diagrams



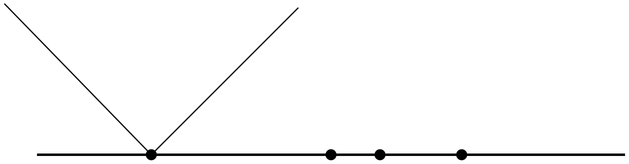
Voronoi Diagrams



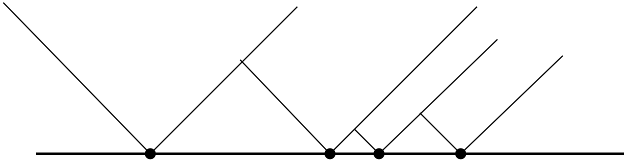
Voronoi Diagrams



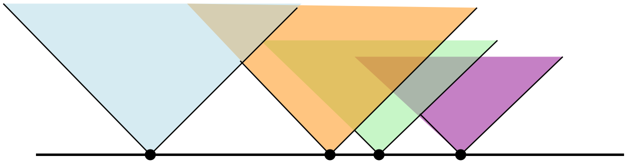
Voronoi Diagrams



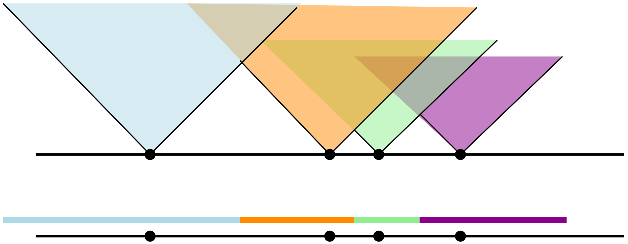
Voronoi Diagrams



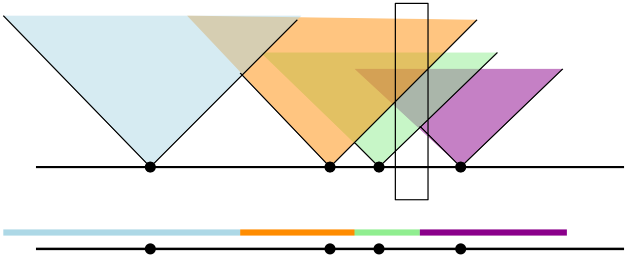
Voronoi Diagrams



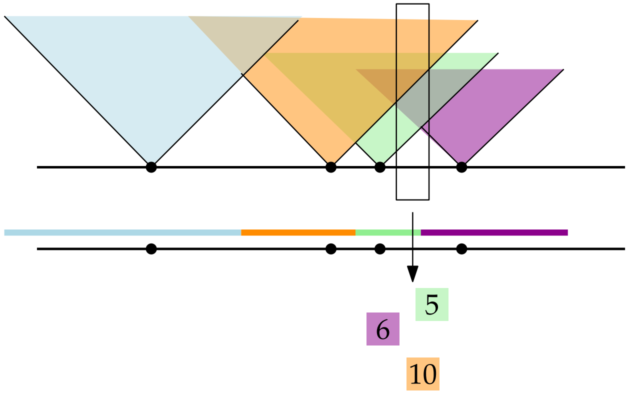
Voronoi Diagrams



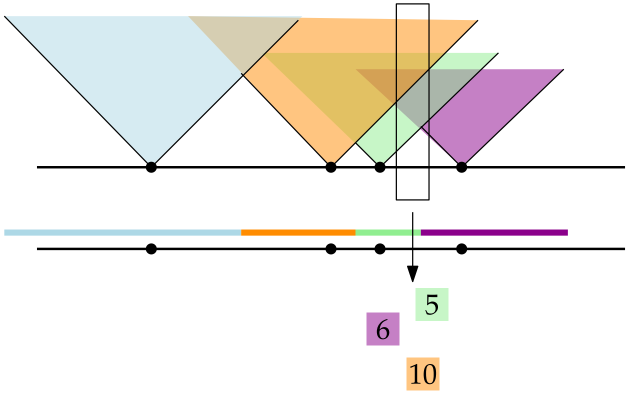
Voronoi Diagrams



Voronoi Diagrams



Voronoi Diagrams



Voronoi diagram is *lower envelope* of collection of distance functions

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles
    - Use many triangles to approximate smooth cone

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles
    - Use many triangles to approximate smooth cone
    - Use shading to encode distance as color value

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles
  - Use many triangles to approximate smooth cone
  - Use shading to encode distance as color value
- Fragment processor at (x, y) receives stream of pixels $p_1, p_2, \ldots$

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles
  - Use many triangles to approximate smooth cone
  - Use shading to encode distance as color value
- Fragment processor at (x, y) receives stream of pixels $p_1, p_2, \ldots$

```
$\min \leftarrow 0$
```

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles
    - Use many triangles to approximate smooth cone
    - Use shading to encode distance as color value
- Fragment processor at (x, y) receives stream of pixels $p_1, p_2, \ldots$

```
min $\leftarrow 0$
**if** depth($p_i$) $<$ min **then**
```

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles
  - Use many triangles to approximate smooth cone
  - Use shading to encode distance as color value
- Fragment processor at (x, y) receives stream of pixels $p_1, p_2, \ldots$

```
min $\leftarrow$ 0
**if** depth($p_i$) $<$ min **then**
   {GPU Z-test}
```

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles
  - Use many triangles to approximate smooth cone
  - Use shading to encode distance as color value
- Fragment processor at (x, y) receives stream of pixels $p_1, p_2, \ldots$

```
min $\leftarrow 0$
**if** depth($p_i$) $<$ min **then**
   {GPU Z-test}
   min $=$ depth($p_i$) {GPU blending operation}
```

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles
  - Use many triangles to approximate smooth cone
  - Use shading to encode distance as color value
- Fragment processor at (x, y) receives stream of pixels $p_1, p_2, \ldots$

```
min $\leftarrow 0$
**if** depth($p_i$) $<$ min **then**
    {GPU Z-test}
    min $=$ depth($p_i$) {GPU blending operation}
    color$(x, y) =$ color$(p_i)$
```

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles
  - Use many triangles to approximate smooth cone
  - Use shading to encode distance as color value
- Fragment processor at (x, y) receives stream of pixels $p_1, p_2, \ldots$

```
min ← 0
**if** depth($p_i$) $<$ min **then**
    {GPU Z-test}
    min $=$ depth($p_i$) {GPU blending operation}
    color$(x, y)$ $=$ color$(p_i)$
**end if**
```

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles
  - Use many triangles to approximate smooth cone
  - Use shading to encode distance as color value
- Fragment processor at (x, y) receives stream of pixels $p_1, p_2, \ldots$

```
min $\leftarrow$ 0
**if** depth($p_i$) $<$ min **then**
    {GPU Z-test}
    min $=$ depth($p_i$) {GPU blending operation}
    color$(x, y)$ $=$ color$(p_i)$
**end if**
```

- Rendering engine is the *mapper*

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles
  - Use many triangles to approximate smooth cone
  - Use shading to encode distance as color value
- Fragment processor at (x, y) receives stream of pixels $p_1, p_2, \ldots$

```
$\text{min} \leftarrow 0$
**if** $\text{depth}(p_i) < \text{min}$ **then**
    {GPU Z-test}
    $\text{min} = \text{depth}(p_i)$ {GPU blending operation}
    $\text{color}(x, y) = \text{color}(p_i)$
**end if**
```

- Rendering engine is the *mapper*
- *Gathering* happens automatically, with fixed key (x, y)

GPU Voronoi Diagrams

- For each point, render a *cone* of colored triangles
  - Use many triangles to approximate smooth cone
  - Use shading to encode distance as color value
- Fragment processor at (x, y) receives stream of pixels $p_1, p_2, \ldots$

```
min ← 0
if depth($p_i$) < min then
   {GPU Z-test}
   min = depth($p_i$) {GPU blending operation}
   color$(x, y)$ = color$(p_i)$
end if
```

- Rendering engine is the *mapper*
- *Gathering* happens automatically, with fixed key (x, y)
- Fragment processors implement *reduce*

Simple GPU Matrix Multiplication

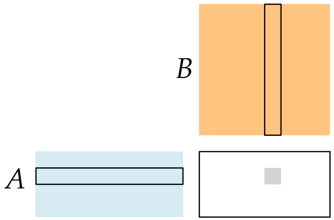
$$C = A \cdot B$$

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Simple GPU Matrix Multiplication

$$C = A \cdot B$$

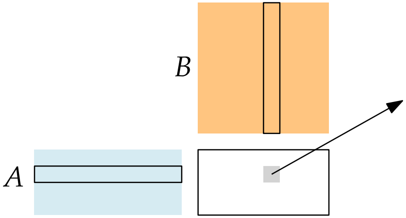
$$C_{ij} = \sum_k A_{ik} B_{kj}$$



Simple GPU Matrix Multiplication

$$C = A \cdot B$$

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

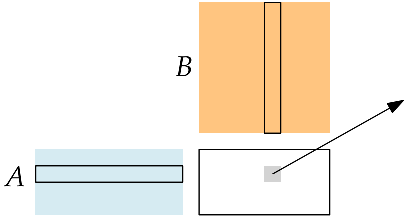


```
for $i = 1 \ldots k$ do
    $C[x,y] = C[x,y] +$
    $A[i,k] * B[k,j]$
end for
```

Simple GPU Matrix Multiplication

$$C = A \cdot B$$

$$C_{ij} = \sum_k A_{ik} B_{kj}$$



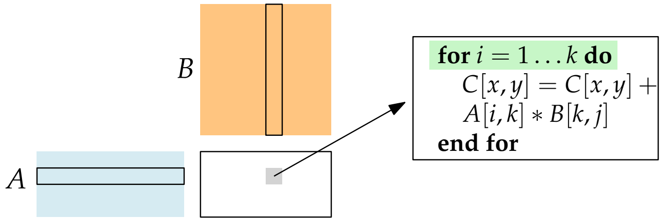
```
for $i = 1 \ldots k$ do
    $C[x,y] = C[x,y] + A[i,k] * B[k,j]$
end for
```

- GPU loops have to be unrolled

Simple GPU Matrix Multiplication

$$C = A \cdot B$$

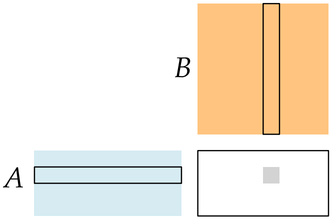
$$C_{ij} = \sum_k A_{ik} B_{kj}$$



```
for $i = 1 \dots k$ do
    $C[x,y] = C[x,y] + A[i,k] * B[k,j]$
end for
```

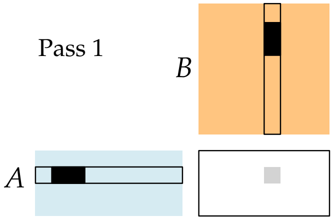
- GPU loops have to be unrolled
- This works only if k is small

GPU Matrix Multiplication II

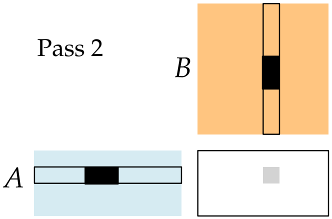


GPU Matrix Multiplication II

Pass 1

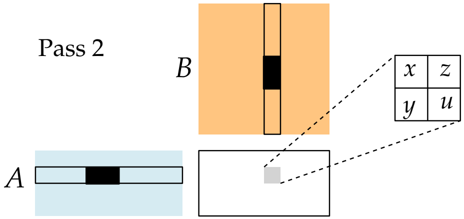


GPU Matrix Multiplication II



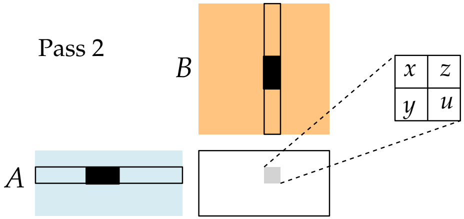
- Multiple passes increase number of memory access, but help with caching

GPU Matrix Multiplication II



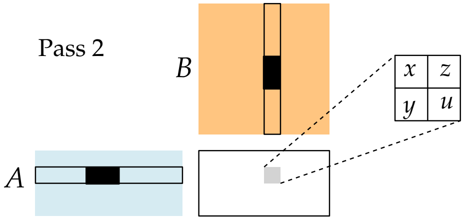
- Multiple passes increase number of memory access, but help with caching
- Each ”field” is actually four values

GPU Matrix Multiplication II



- Multiple passes increase number of memory access, but help with caching
- Each ”field” is actually four values
- Can get a factor 4 speedup with careful partitioning of matrix

GPU Matrix Multiplication II



- Multiple passes increase number of memory access, but help with caching
- Each ”field” is actually four values
- Can get a factor 4 speedup with careful partitioning of matrix
- More complicated methods needed for *sparse* multiplication

Sorting on the GPU

Can we sort on the GPU ?

- Compute parallelism is not enough

Sorting on the GPU

Can we sort on the GPU ?

- Compute parallelism is not enough
- Need SIMD structures (find repeated instruction patterns)

Sorting on the GPU

Can we sort on the GPU ?

- Compute parallelism is not enough
- Need SIMD structures (find repeated instruction patterns)
- External memory methods manage I/O bottlenecks but don’t exploit compute power.

Sorting on the GPU

Can we sort on the GPU ?

- Compute parallelism is not enough
- Need SIMD structures (find repeated instruction patterns)
- External memory methods manage I/O bottlenecks but don't exploit compute power.
- Plan: Use *sorting networks:*

Sorting on the GPU

Can we sort on the GPU ?

- Compute parallelism is not enough
- Need SIMD structures (find repeated instruction patterns)
- External memory methods manage I/O bottlenecks but don’t exploit compute power.
- Plan: Use *sorting networks:*
    - Many simple (and local) compute elements

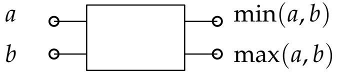
Sorting on the GPU

Can we sort on the GPU ?

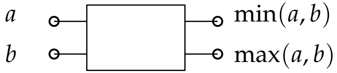
- Compute parallelism is not enough
- Need SIMD structures (find repeated instruction patterns)
- External memory methods manage I/O bottlenecks but don't exploit compute power.
- Plan: Use *sorting networks:*
  - Many simple (and local) compute elements
  - High-throughput and synchronous

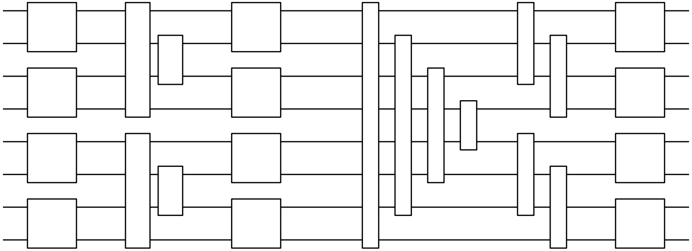
Bitonic Sorting

Bitonic Sorting

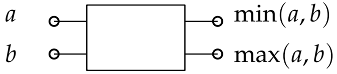


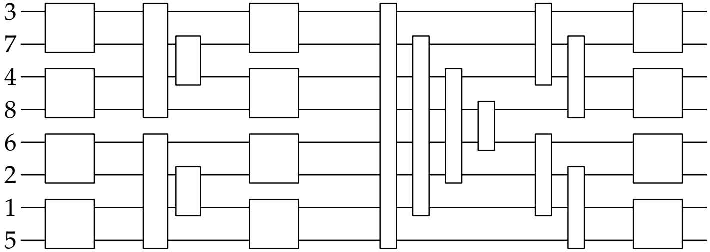
Bitonic Sorting



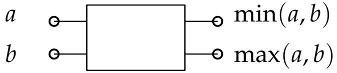


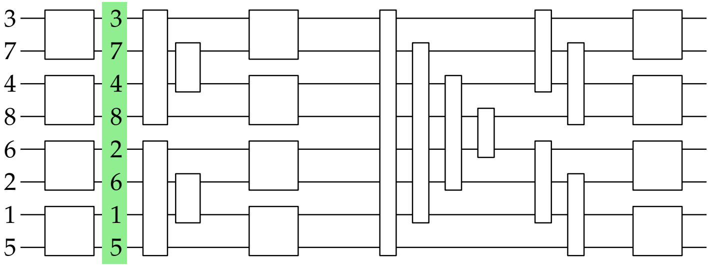
Bitonic Sorting



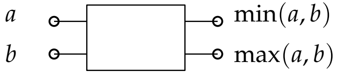


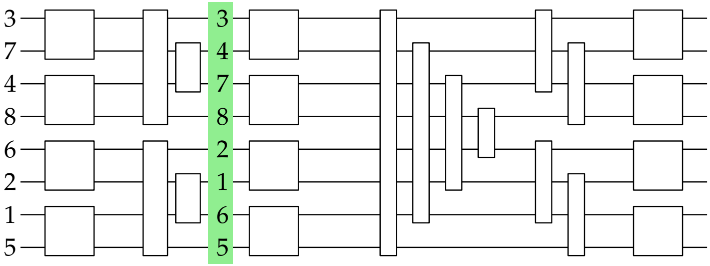
Bitonic Sorting



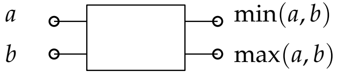


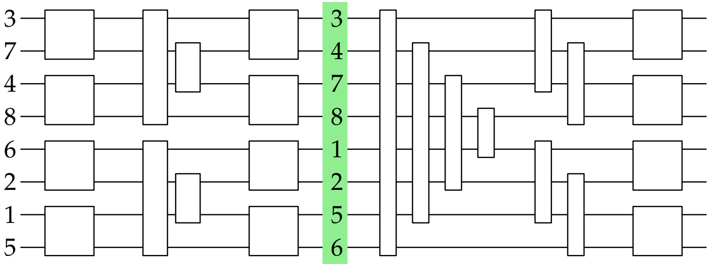
Bitonic Sorting



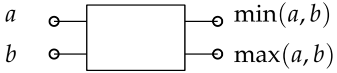


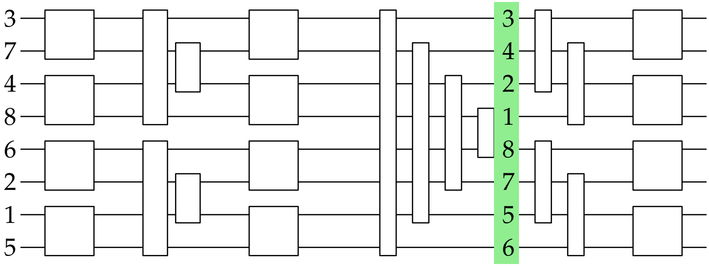
Bitonic Sorting



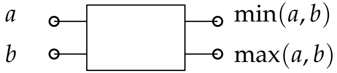


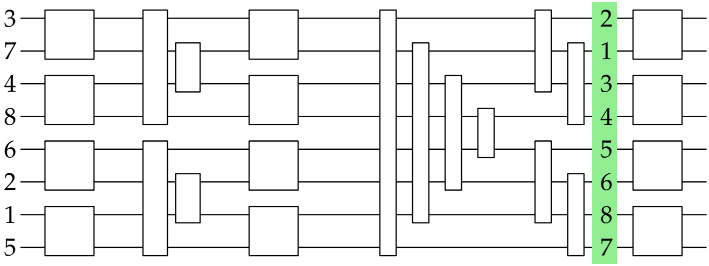
Bitonic Sorting



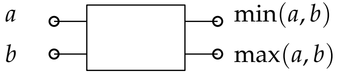


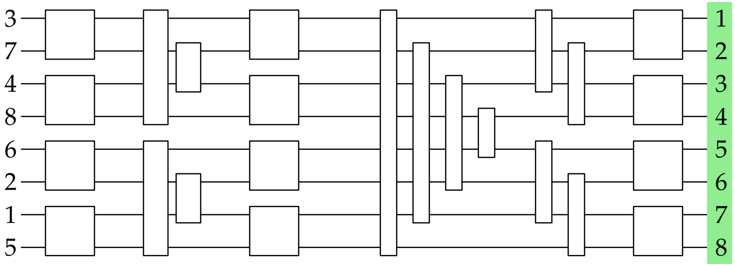
Bitonic Sorting





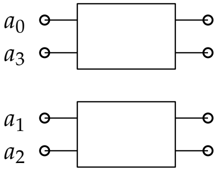
Bitonic Sorting



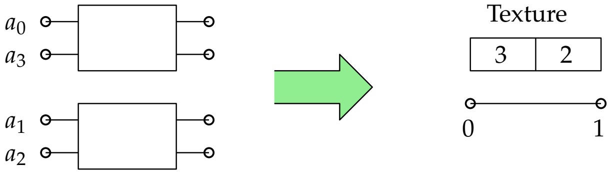


Bitonic sort requires $\log^2 n$ layers, $n/2$ comparators/layer

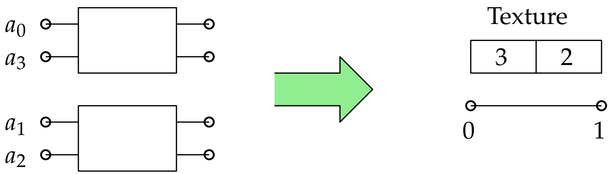
GPU Bitonic Sorting



GPU Bitonic Sorting

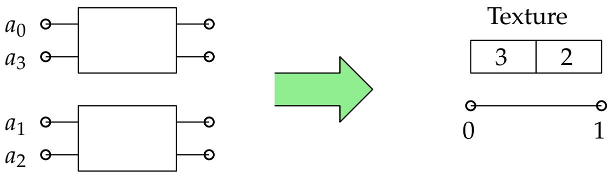


GPU Bitonic Sorting



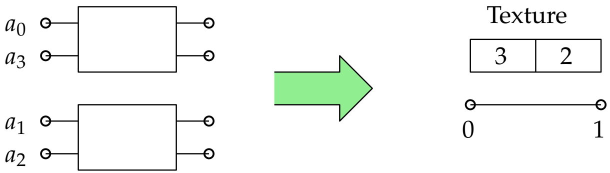
- Fill 2D array with values

GPU Bitonic Sorting



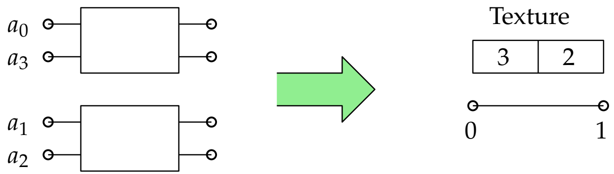
- Fill 2D array with values
- (for each pass) construct quadrilateral with lookup values

GPU Bitonic Sorting



- Fill 2D array with values
- (for each pass) construct quadrilateral with lookup values
- Texture hardware locates lookup values, and fragment program does comparisons

GPU Bitonic Sorting



- Fill 2D array with values
- (for each pass) construct quadrilateral with lookup values
- Texture hardware locates lookup values, and fragment program does comparisons
- $\log^2 n$ passes used to complete the computation

Review

This Lecture

- Brief history of GPU model
- Simple GPU SIMD model
- Examples: Voronoi diagrams, matrix multiplication and sorting

Next Lecture

- More simple GPU examples
- Toy example of algorithmic view: “GPU as streaming processor”
- The CUDA model for modern GPUs
- “Hello world” example: matrix multiplication in CUDA

Questions?