Data-Parallel Algorithms on GPUs

Mark Harris

NVIDIA Developer Technology

Outline

SC06

- **Introduction**
- **Algorithmic complexity on GPUs**
- **Algorithmic Building Blocks**
  - Gather & Scatter
  - Reductions
  - Scan (parallel prefix)
  - Sort
  - Search

GPGPU

Data-Parallel Algorithms

- **The GPU is a data-parallel processor**
  - Data-parallel kernels of applications can be accelerated on the GPU
- **Efficient algorithms require efficient building blocks**
- **This talk: data-parallel building blocks**
  - Gather & Scatter
  - Map
  - Reduce and Scan
  - Sort and Search

Algorithmic Complexity on GPUs

- **We will use standard "Big O" notation**
  - e.g., optimal sequential sort is O(*n* log *n*)
- **GPGPU element of parallelism is the pixel**
  - Each pixel generates one output element
  - O(*n*) typically means *n* pixels processed
- **In general, GPGPU O(*n*) usually means O(*n*/*p*) processing time**
  - *p* is the number of "pixel processors" on the GPU
    - NVIDIA G70 has 24 pixel shader pipelines
    - NVIDIA G80 has 128 unified shader processors

Step vs. Work Complexity

SC06

- **Important to distinguish between the two**

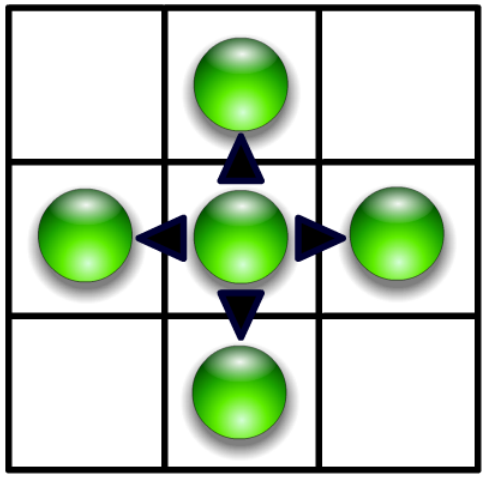
- **Work Complexity: O(# pixels processed)**
  - More correctly O(# pixels * work per pixel)
- **Step Complexity: O(# rendering passes)**

Data-Parallel Building Blocks

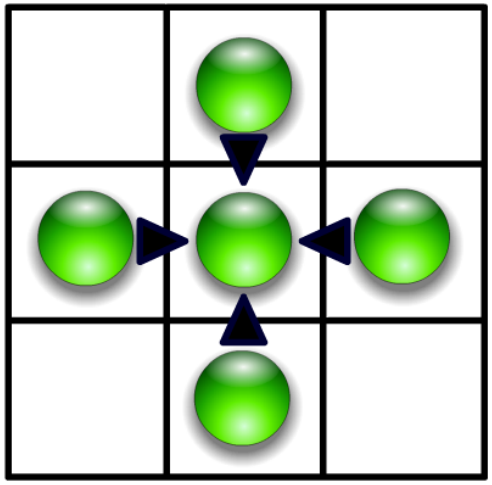
- Gather & Scatter
- Map
- Reduce
- Scan
- Sort
- Search

Scatter vs. Gather

- **Gather:** `p = a[i]`
  - Vertex or Fragment programs
- **Scatter:** `a[i] = p`
  - Vertex programs only



Scatter



Gather

GPGPU

Scatter Techniques

- **Scatter not available on most GPUs**
  - Recently made available to GPGPU applications
    - using ATI CTM and NVIDIA CUDA
    - see talks later by Mark Segal and Ian Buck
- **John Owens will discuss ways to simulate scatter**
- **Lack of scatter in fragment programs affects GPGPU algorithms**

The Map Operation

- **Given:**
  - Array or stream of data elements *A*
  - Function *f*(*x*)
- **map(*A*, *f*) = applies *f*(*x*) to all $a_i \in A$**
- **GPU implementation is straightforward**
  - *A* is a texture, a_i are texels
  - Pixel shader implements *f*(x), reads a_i as *x*
  - Draw a quad with as many pixels as texels in *A* with *f*(x) pixel shader active
  - Output stored in another texture

GPGPU

Parallel Reductions

SC06

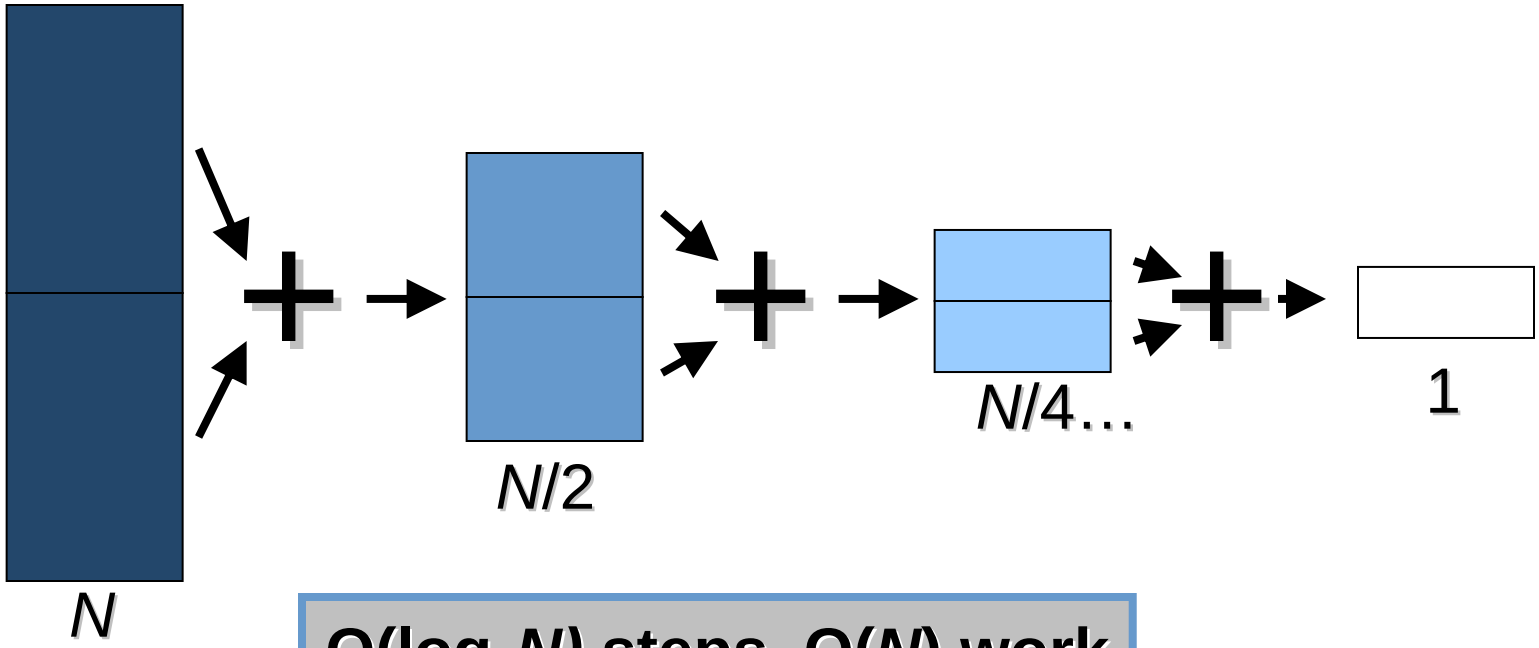
- **Given:**
  - Binary associative operator $\oplus$ *with identity* I
  - *Ordered set* $s = [a_0, a_1, \ldots, a_{n-1}]$ *of* n *elements*
- **reduce($\oplus$, s)** *returns* $a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}$
- **Example:**
  reduce(+, [3 1 7 0 4 1 6 3]) = 25

- **Reductions common in parallel algorithms**
  - Common reduction operators are +, ×, min and max
  - Note floating point is only pseudo-associative

GPGPU

Parallel Reductions on the GPU

- **1D parallel reduction:**
  - add two halves of texture together
  - repeatedly…
  - Until we're left with a single row of texels

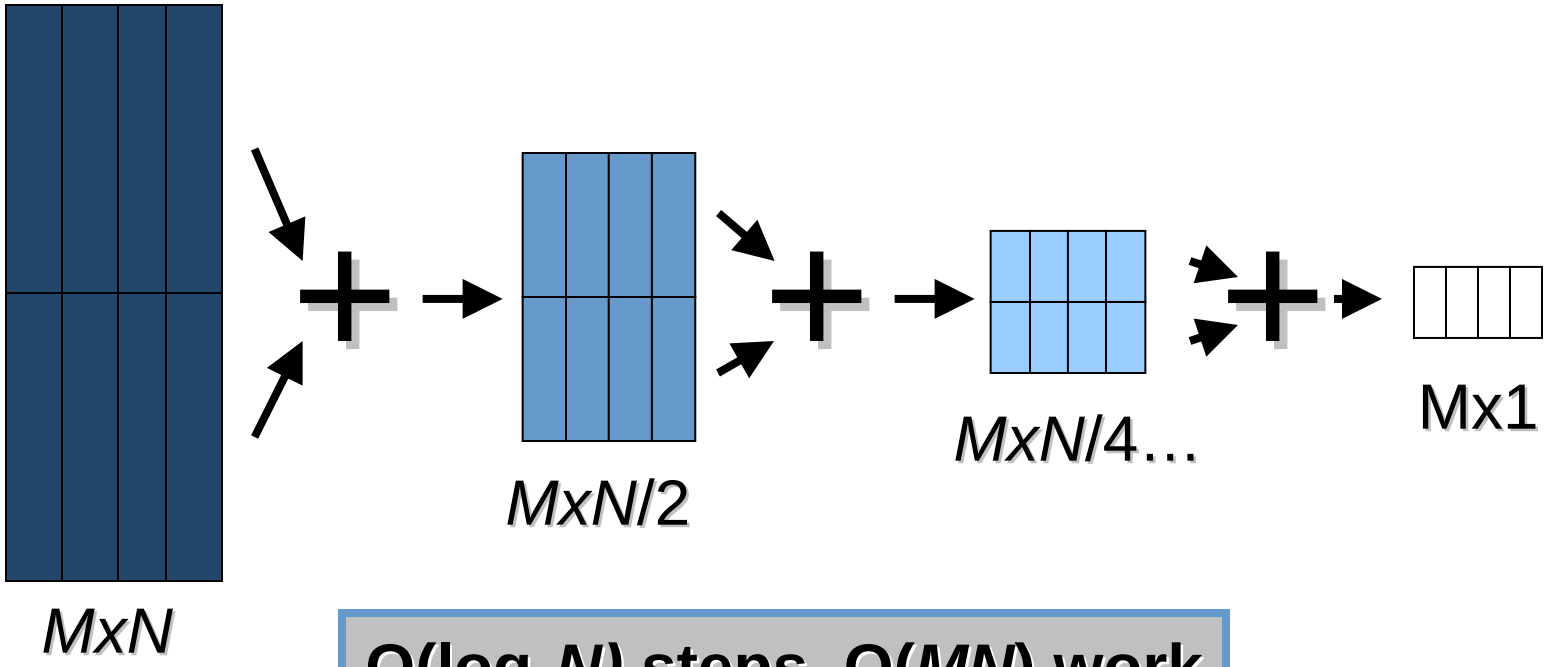


O(log$_2$*N*) steps, O(*N*) work

GPGPU

Multiple 1D Parallel Reductions

- **Can run many reductions in parallel**
  - Use 2D texture and reduce one dimension

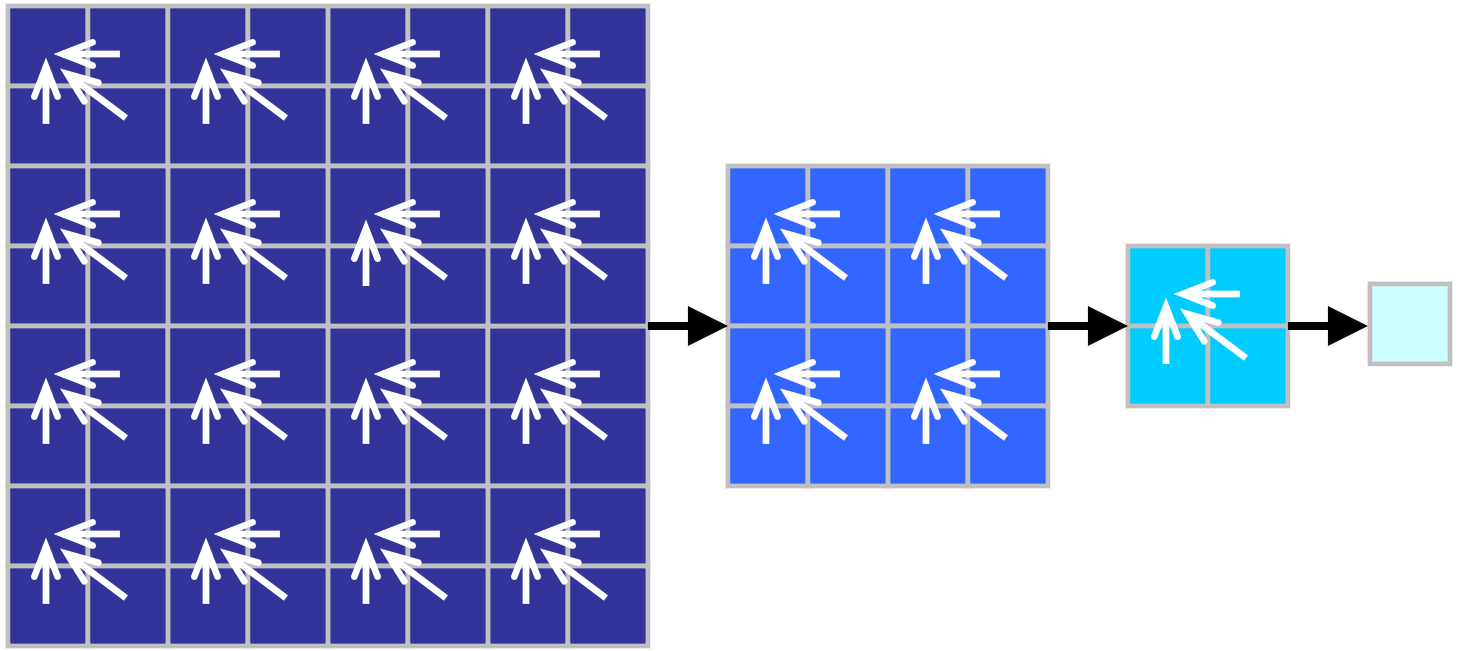


O(log$_2$*N*) steps, O(*MN*) work

GPGPU

2D reductions

- **Like 1D reduction, only reduce in both directions simultaneously**



  - Note: can add more than 2x2 elements per pixel
    - Trade per-pixel work for step complexity
    - Best perf depends on specific GPU (cache, etc.)

GPGPU

Parallel Scan (aka prefix sum)

SC06

- **Given:**
  - Binary associative operator $\oplus$ *with identity* I
  - *Ordered set* $s = [a_0, a_1, \ldots, a_{n-1}]$ *of* n *elements*
- **scan($\oplus$, s)** *returns*

  $$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

- **Example:**
  scan(+, [3 1 7 0 4 1 6 3]) =
  [3 4 11 11 14 16 22 25]

(From Blelloch, 1990, "Prefix Sums and Their Applications")

GPGPU

Applications of Scan

- Radix sort
- Quicksort
- String comparison
- Lexical analysis
- Stream compaction
- Stream compaction
- Polynomial evaluation
- Solving recurrences
- Tree operations
- Histograms

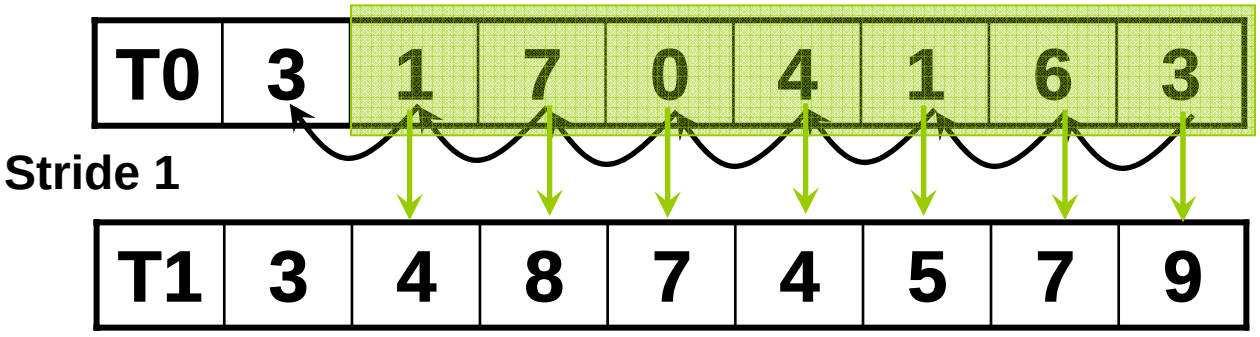
A Naive Parallel Scan Algorithm

T0	3	1	7	0	4	1	6	3

Log(n) iterations

Note: Can't read and write the same texture, so must "ping-pong"

A Naive Parallel Scan Algorithm



Log(n) iterations

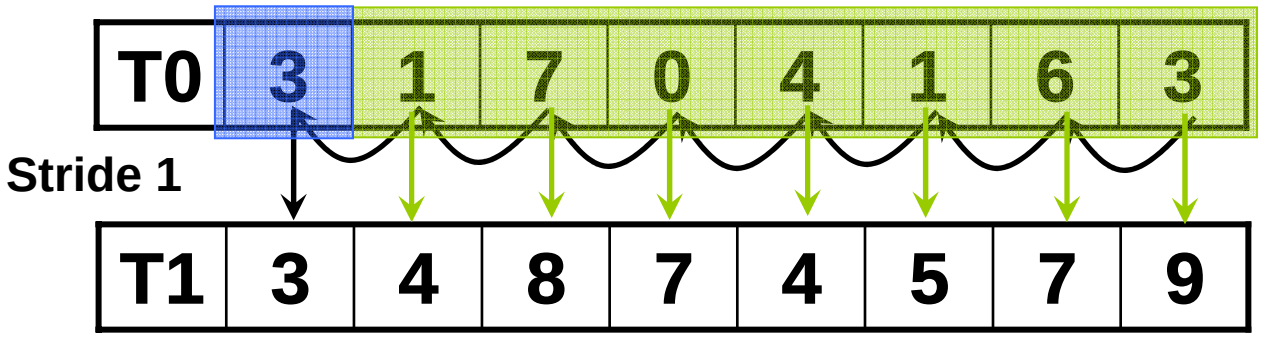
Note: Can't read and write the same texture, so must "ping-pong"

For i from 1 to log(n)-1:

- Render a quad from 2^i to n. Fragment k computes

  $v_{out} = v[k] + v[k-2^i]$.

A Naive Parallel Scan Algorithm



Log(n) iterations

Note: Can't read and write the same texture, so must "ping-pong"

For *i* from 1 to log(n)-1:

- Render a quad from 2^i to n. Fragment *k* computes

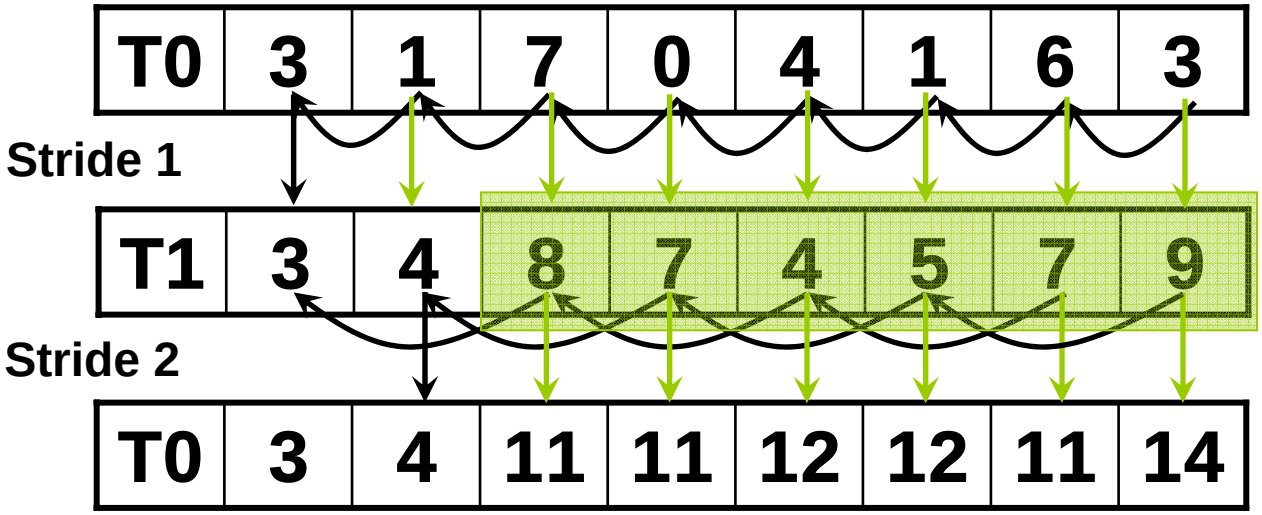
  $v_{out} = v[k] + v[k-2^i]$.

- Due to ping-pong, render a 2nd quad from $2^{(i-1)}$ to 2^i with a simple pass-through shader

  $v_{out} = v_{in}$.

GPGPU

A Naive Parallel Scan Algorithm



Log(n) iterations

Note: Can't read and write the same texture, so must "ping-pong"

For i from 1 to log(n)-1:

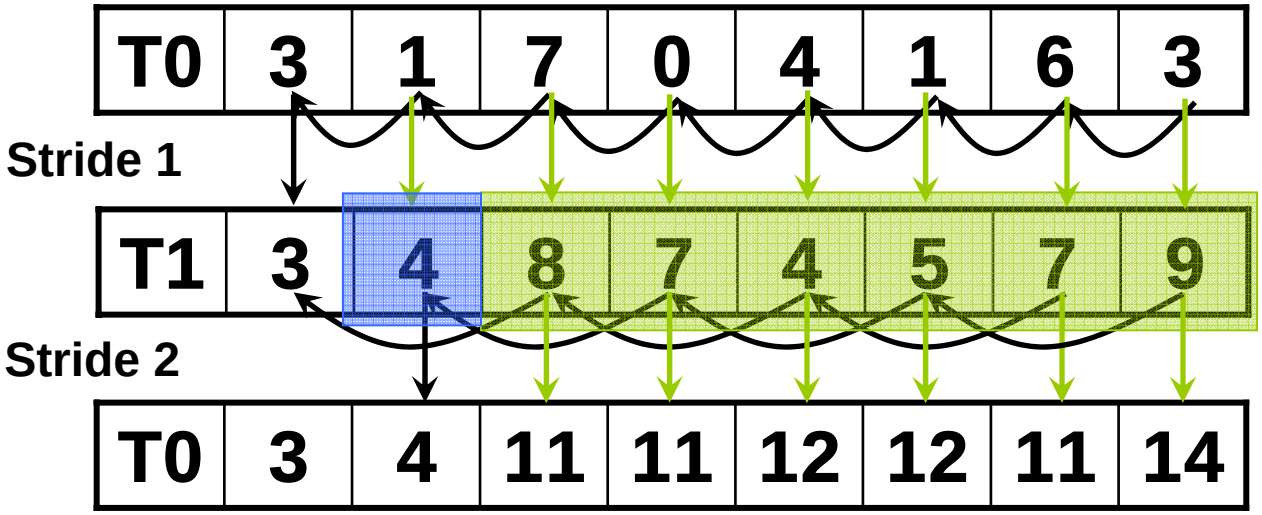
- Render a quad from 2^i to n. Fragment k computes

  $v_{out} = v[k] + v[k-2^i]$.

- Due to ping-pong, render a 2nd quad from $2^{(i-1)}$ to 2^i with a simple pass-through shader

  $v_{out} = v_{in}$.

A Naive Parallel Scan Algorithm



Log(n) iterations

Note: Can't read and write the same texture, so must "ping-pong"

For i from 1 to log(n)-1:

- Render a quad from 2^i to n. Fragment k computes

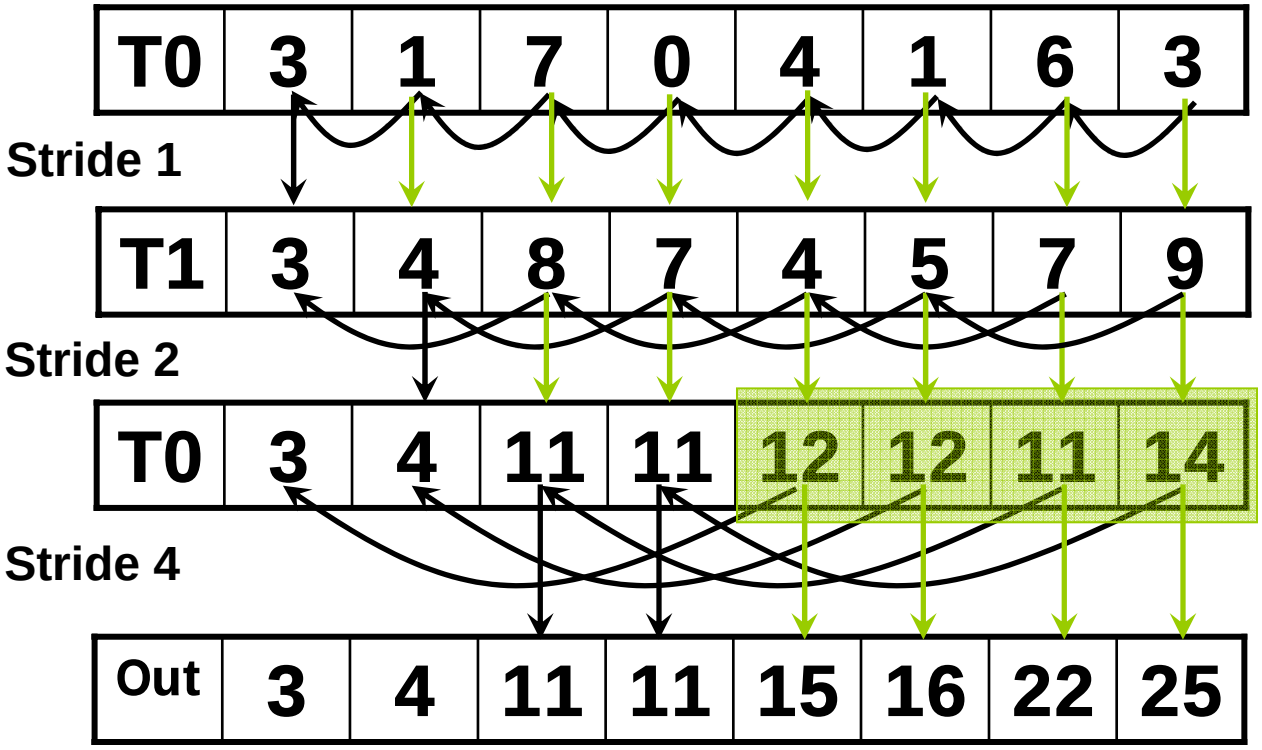
  $v_{out} = v[k] + v[k-2^i]$.

- Due to ping-pong, render a 2nd quad from $2^{(i-1)}$ to 2^i with a simple pass-through shader

  $v_{out} = v_{in}$.

GPGPU

A Naive Parallel Scan Algorithm



Log(n) iterations

Note: Can't read and write the same texture, so must "ping-pong"

For *i* from 1 to log(n)-1:

- Render a quad from 2^i to n. Fragment *k* computes

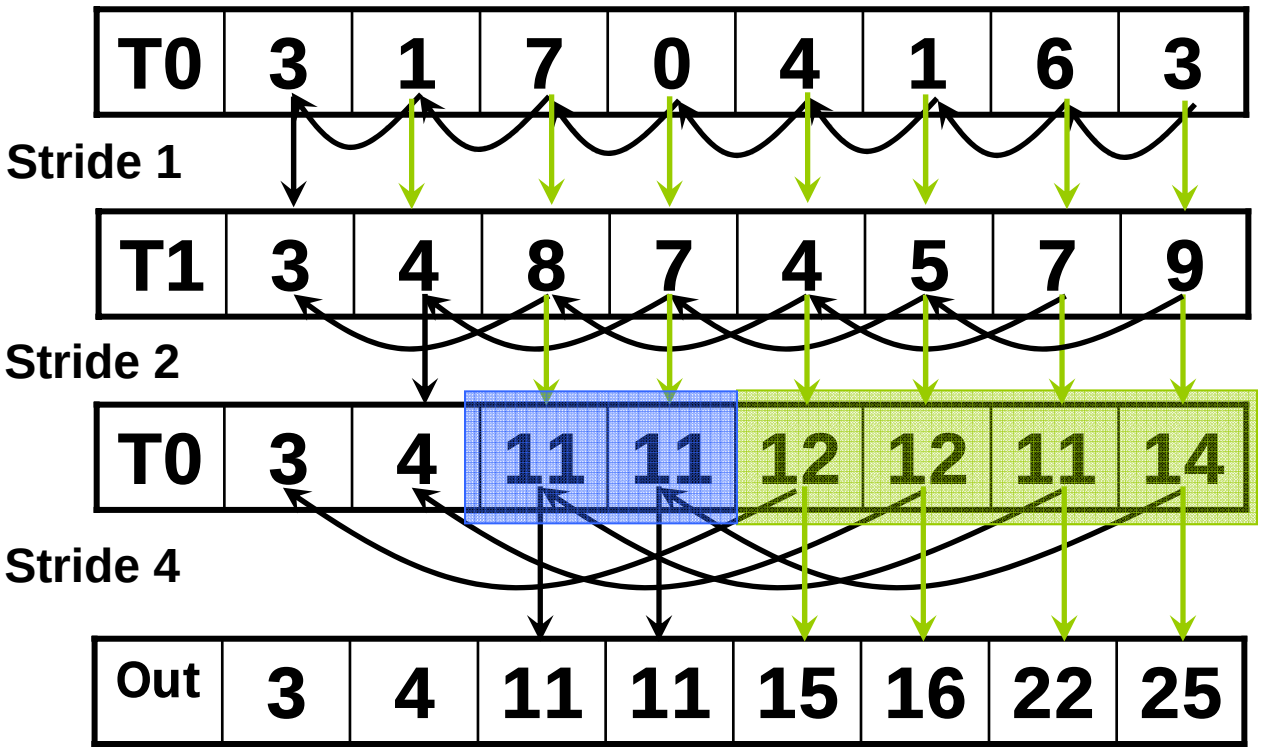
  $v_{out} = v[k] + v[k-2^i]$.

- Due to ping-pong, render a 2nd quad from $2^{(i-1)}$ to 2^i with a simple pass-through shader

  $v_{out} = v_{in}$.

GPGPU

A Naive Parallel Scan Algorithm



Log(n) iterations

Note: Can't read and write the same texture, so must "ping-pong"

For *i* from 1 to log(n)-1:

- Render a quad from 2^i to n. Fragment *k* computes

  $v_{out} = v[k] + v[k-2^i]$.

- Due to ping-pong, render a 2nd quad from $2^{(i-1)}$ to 2^i with a simple pass-through shader

  $v_{out} = v_{in}$.

A Naive Parallel Scan Algorithm

- **Algorithm given in more detail in [Horn '05]**
- **Step-efficient, but not work-efficient**
  - O(log n) steps, but O(n log n) adds
  - Sequential version is O(n)
  - A factor of log(n) hurts: 20x for 10^6 elements!
- **Dig into parallel algorithms literature for a better solution**
  - See Blelloch 1990, "Prefix Sums and Their Applications"

Balanced Trees

- **Common parallel algorithms pattern**
  - Build a balanced binary tree on the input data and sweep it to and from the root
  - Tree is conceptual, not an actual data structure
- **For scan:**
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  - Traverse back up the tree building the scan from the partial sums

Balanced Tree Scan

1. **First build sums in place up the tree**
2. **Traverse back down and use partial sums to generate scan**

- **Note: tricky to implement using graphics API**
  - Due to interleaving of new and old results
  - Can reformulate layout

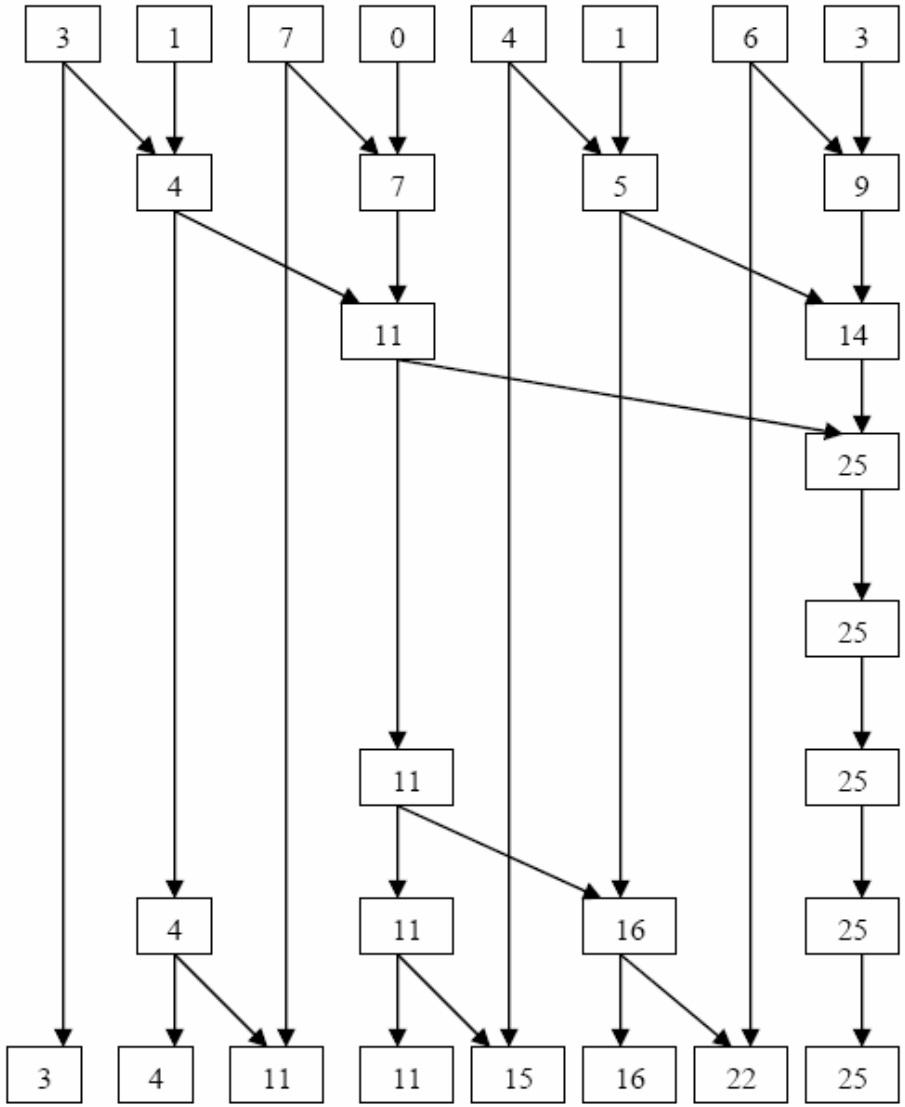
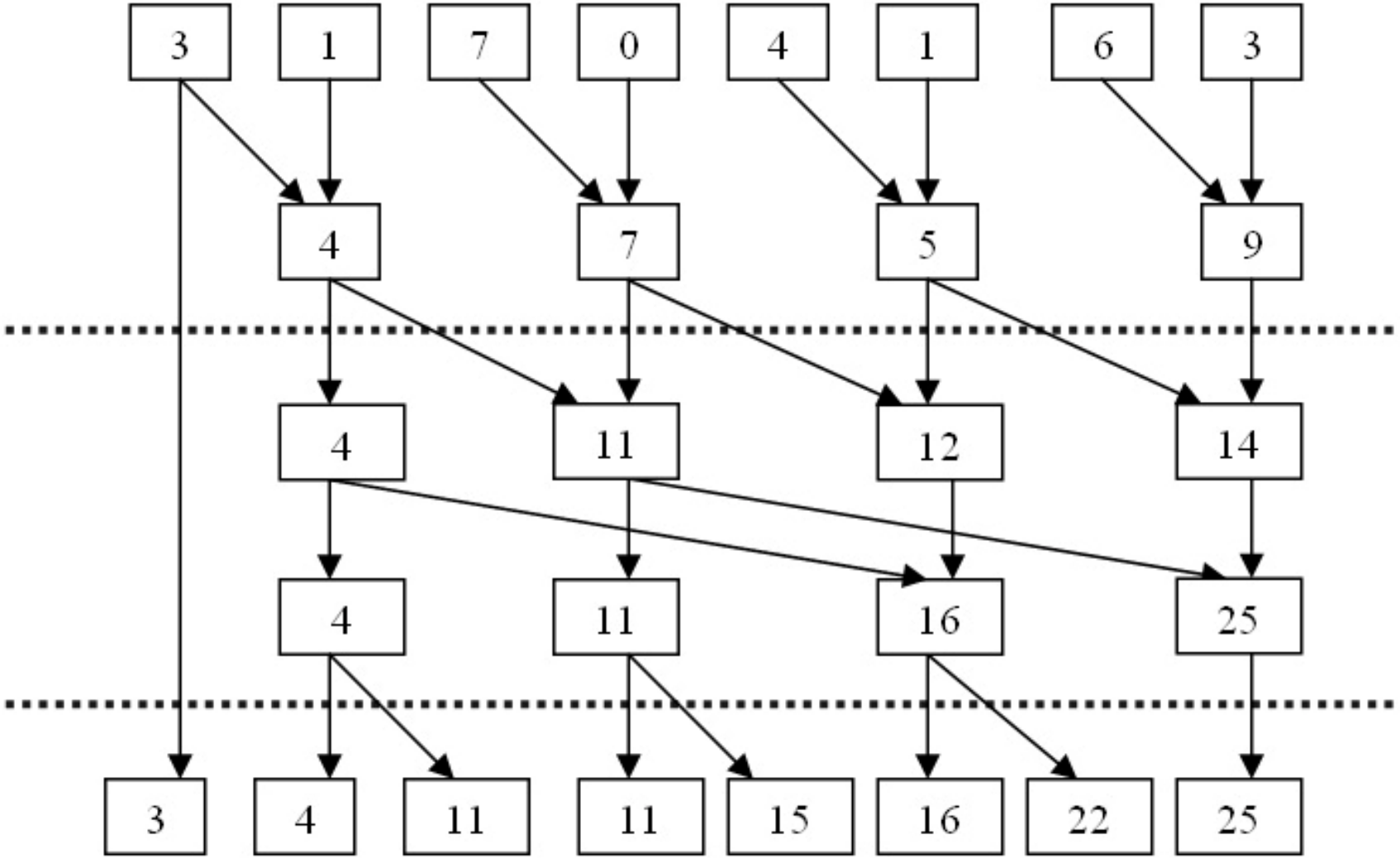


Figure courtesy Shubho Sengupta

Further Improvement

- **[Sengupta et al. ’06] observes that balanced tree algorithm is not step-efficient**
  - Loses efficiency on steps that contain fewer pixels than the GPU has pipelines
- **Hybrid work-efficient / step-efficient algorithm**
  - Simply switch from balanced tree to naïve algorithm for smaller steps

Hybrid work- and step-efficient algo



GPGPU

Figure courtesy Shubho Sengupta

Scan with Scatter

- **Scatter in pixel shaders makes scan easier to implement**
  - NVIDIA CUDA and ATI/AMD CTM enable this
  - NVIDIA CUDA SDK includes example implementation of Scan primitive
- **CUDA Parallel Data Cache improves efficiency**
  - All steps executed in a single kernel
  - Threads communicate through shared memory
  - Drastically reduces bandwidth bottleneck!

GPGPU

Parallel Sorting

- **Given an unordered list of elements, produce list ordered by key value**
  - Kernel: compare and swap
- **GPUs constrained programming environment limits viable algorithms**
  - Bitonic merge sort [Batcher 68]
  - Periodic balanced sorting networks [Dowd 89]
- **Recent research results impressive**
  - Naga Govindaraju will cover the algorithms in detail

Binary Search

- **Find a specific element in an ordered list**
- **Implement just like CPU algorithm**
  - Finds the first element of a given value *v*
    - If *v* does not exist, find next smallest element > *v*
- **Search is sequential, but many searches can be executed in parallel**
  - Number of pixels drawn determines number of searches executed in parallel
    - 1 pixel == 1 search
- **For details see:**
  - "A Toolkit for Computation on GPUs". Ian Buck and Tim Purcell. In *GPU Gems*. Randy Fernando, ed. 2004

References

- *"Prefix Sums and Their Applications"*. Guy E. Blelloch. Technical Report CMU-CS-90-190. November, 1990.
- *"A Toolkit for Computation on GPUs"*. Ian Buck and Tim Purcell. In *GPU Gems*. Randy Fernando, ed. 2004
- *"GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management"*. Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. In Proceedings of *ACM SIGMOD 2006*
- *"Stream Reduction Operations for GPGPU Applications"*. Daniel Horn. In *GPU Gems 2.* Matt Pharr, ed. 2005
- *"Improved GPU Sorting".* Peter Kipfer. In *GPU Gems 2*. Matt Pharr, ed. 2005
- *"A Work-Efficient Step-Efficient Prefix Sum Algorithm".* Shubhabrata Sengupta, Aaron E. Lefohn, John D. Owens. In *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*