UCDAVIS

S05: High Performance Computing with CUDA

Data-parallel Algorithms & Data Structures

John Owens
UC Davis

One Slide Summary of Today

- GPUs are *great* at running many closely-coupled but independent threads in parallel
  - The programming model is SPMD—single program, multiple data
- GPU computing boils down to:
  - Define a *computation domain* that generates many parallel threads
    - This is the data structure
  - *Iterate* in parallel over that computation domain, running a program over all threads
    - This is the algorithm

Outline

- Data Structures
  - GPU Memory Model
  - Taxonomy
- Algorithmic Building Blocks
  - Sample Application
  - Map
  - Gather & Scatter
  - Reductions
  - Scan (parallel prefix)
  - Sort, search, …

GPU Memory Model

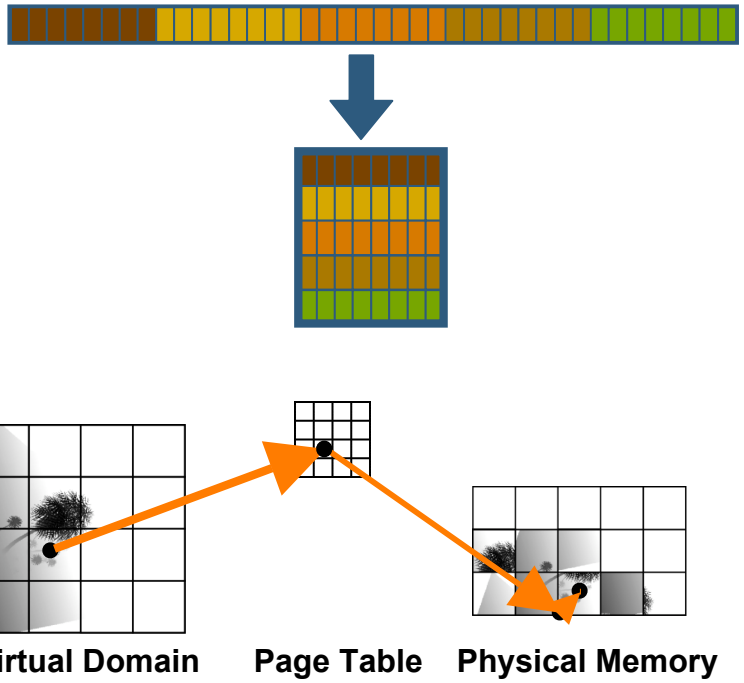
- More restricted memory access than CPU
  - Allocate/free memory only before computation
  - Transfers to and from CPU are explicit
  - GPU is controlled by CPU, can't initiate transfers, access disk, etc.
- To generalize …
  - GPUs are better at *accessing* data structures
  - CPUs are better at *building* data structures
  - As CPU-GPU bandwidth improves, consider doing data structure tasks on their "natural" processor

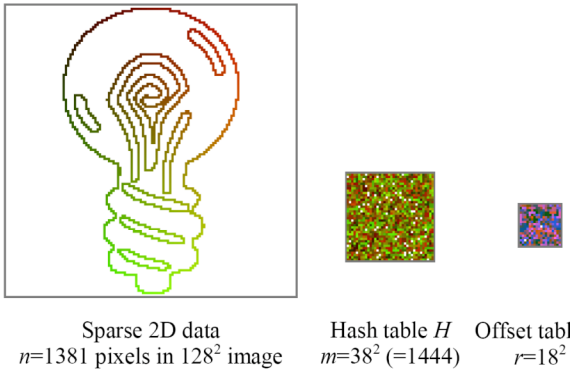
GPU Memory Model

- **Limited memory access during computation (kernel)**
  - **Registers (per fragment/thread)**
    - **Read/write**
  - **Shared memory (shared among threads)**
    - **Does not exist in general**
    - **CUDA allows access to shared memory btwn threads**
  - **Global memory (historical)**
    - **Read-only during computation**
    - **Write-only at end of computation (precomputed address)**
  - **Global memory (new)**
    - **Allows general scatter/gather (read/write)**
      - No collision rules!
      - Exposed in all modern GPUs

Properties of GPU Data Structures

- To be efficient, must support
  - Parallel read
  - Parallel write
  - Parallel iteration
- Generalized arrays fit these requirements
  - Dense (complete) arrays
  - Sparse (incomplete) arrays
  - Adaptive arrays





Think In Parallel

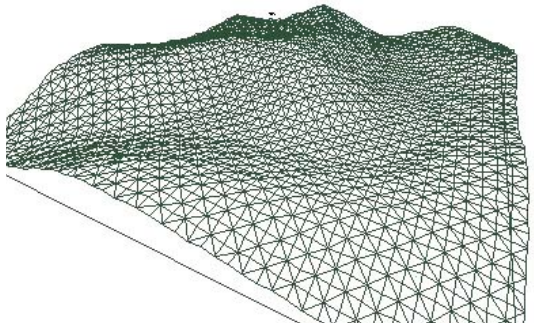
- The GPU is a data-parallel processor
  - Thousands of parallel threads
  - Thousands of data elements to process
  - All data processed by the same program
    - SPMD computation model
  - Contrast with task parallelism and ILP

- Best results when you "Think Data Parallel"
  - Design your algorithm for data-parallelism
  - Understand parallel algorithmic complexity and efficiency
  - Use data-parallel algorithmic primitives as building blocks

Data-Parallel Algorithms

- Efficient algorithms require efficient building blocks
- This talk: data-parallel building blocks
  - Map
  - Gather & Scatter
  - Reduce
  - Scan

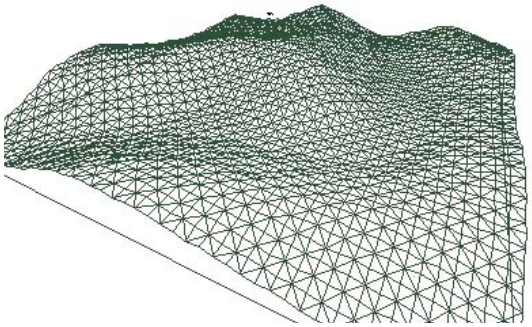
Sample Motivating Application



- How bumpy is a surface that we represent as a grid of samples?
- Algorithm:
  - Loop over all elements
  - At each element, compare the value of that element to the average of its neighbors ("difference"). Square that difference.
  - Now sum up all those differences.
    - But we don't want to sum all the diffs that are 0.
    - So only sum up the non-zero differences.
  - This is a fake application—don't take it too seriously.

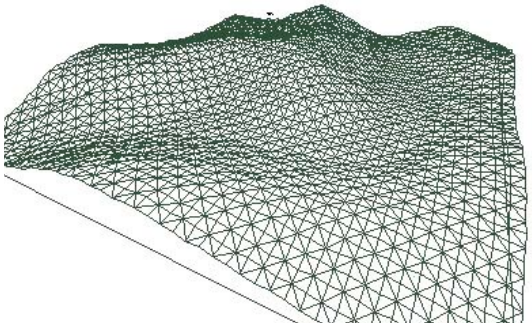
Picture courtesy http://www.artifice.com

Sample Motivating Application



```
for all samples:
    neighbors[x,y] =
            0.25 * (    value[x-1,y]+
                        value[x+1,y]+
                        value[x,y+1]+
                        value[x,y-1] ) )
    diff = (value[x,y] - neighbors[x,y])^2
result = 0
for all samples where diff != 0:
    result += diff
return result
```

Sample Motivating Application



```
for all samples:
    neighbors[x,y] =
            0.25 * (    value[x-1,y]+
                        value[x+1,y]+
                        value[x,y+1]+
                        value[x,y-1] ) )
    diff = (value[x,y] - neighbors[x,y])^2
result = 0
for all samples where diff != 0:
    result += diff
return result
```

The Map Operation

- Given:
  - Array or stream of data elements *A*
  - Function *f*(*x*)
- map(*A*, *f*) = applies *f*(*x*) to all $a_i \in A$
- CUDA implementation is straightforward
  - Statements in CUDA kernels are applied in parallel to all threads that execute them
- Map is as simple as:

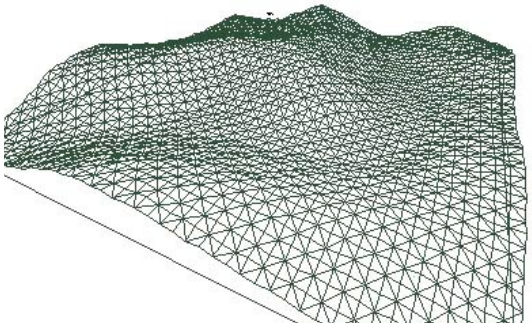
```
// for all samples – all threads execute this code
neighbors[x][y] =
    0.25f * (value[x-1][y]+
             value[x+1][y]+
             value[x][y+1]+
             value[x][y-1]);
diff = (value[x][y] - neighbors[x][y]);
diff *= diff; // squared difference
```

The Map Operation

- Given:
  - Array or stream of data elements *A*
  - Function *f*(*x*)
- map(*A*, *f*) = applies *f*(*x*) to all $a_i \in A$
- CUDA implementation is straightforward
  - Statements in CUDA kernels are applied in parallel to all threads that execute them
- Map is as simple as:

```
// for all samples - all threads execute this code
neighbors =
    0.25f * (value[thread_id.x-1][thread_id.y]+
             value[thread_id.x+1][thread_id.y]+
             value[thread_id.x][thread_id.y+1]+
             value[thread_id.x][thread_id.y-1]);
diff = (value- neighbors);
diff *= diff; // squared difference
```

Sample Motivating Application



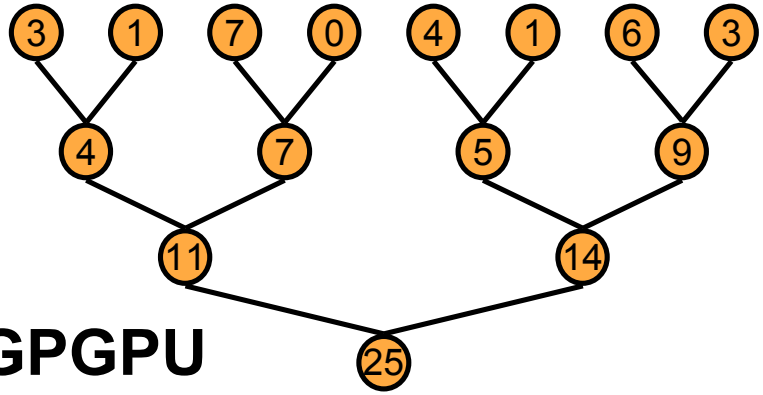
```
for all samples:
    neighbors[x,y] =
            0.25 * (    value[x-1,y]+
                        value[x+1,y]+
                        value[x,y+1]+
                        value[x,y-1] ) )
    diff = (value[x,y] - neighbors[x,y])^2
result = 0
for all samples where diff != 0:
    result += diff
return result
```

Parallel Reductions

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

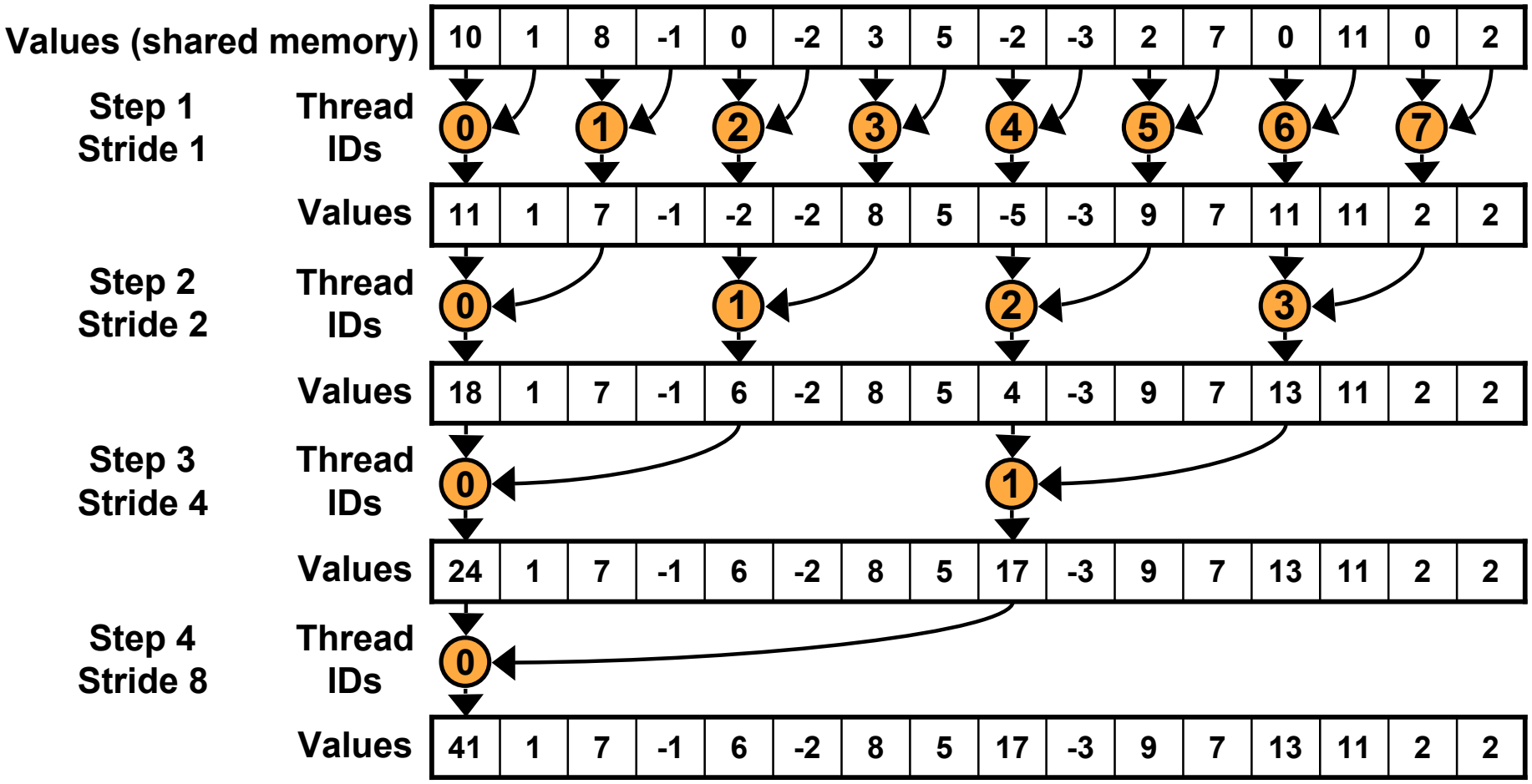
- **Reductions common in parallel algorithms**
  - **Common reduction operators are +, ×, min and max**
  - **Note floating point is only pseudo-associative**

Tree-Based Parallel Reductions



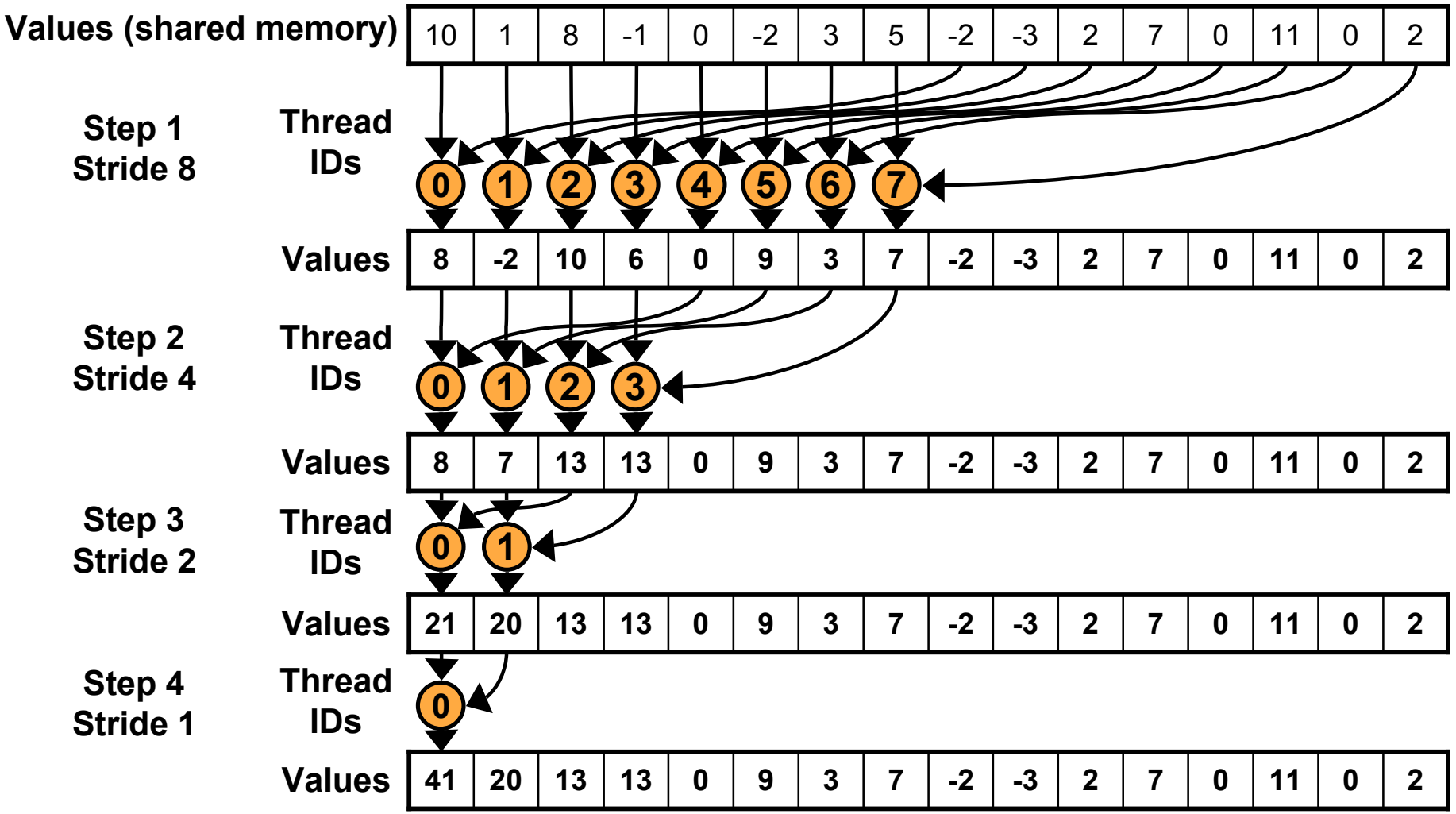
- Commonly done in traditional GPGPU
  - Ping-pong between render targets, reduce by 1/2 at a time
  - Completely bandwidth bound using graphics API
  - Memory writes and reads are off-chip, no reuse of intermediate sums
- CUDA solves this by exposing on-chip shared memory
  - Reduce blocks of data in shared memory to save bandwidth

Parallel Reduction: Interleaved Addressing



Interleaved addressing results in bank conflicts

Parallel Reduction: Sequential Addressing

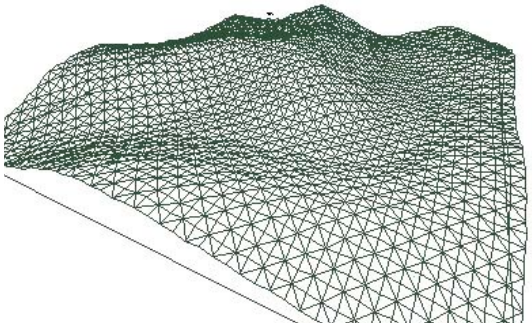


Sequential addressing is conflict free

Parallel Reduction Complexity

- log(*N*) parallel steps, each step *S* does $N/2^S$ independent ops
  - Step Complexity is O(log *N*)
- For $N=2^D$, performs $\sum_{S\in[1..D]}2^{D-S} = N-1$ operations
  - Work Complexity is O(*N*) – It is work-efficient
  - i.e. does not perform more operations than a sequential algorithm
- With *P* threads physically in parallel (*P* processors), time complexity is O(*N*/*P* + log *N*)
  - Compare to O(*N*) for sequential reduction

Sample Motivating Application



```
for all samples:
    neighbors[x,y] =
            0.25 * (    value[x-1,y]+
                        value[x+1,y]+
                        value[x,y+1]+
                        value[x,y-1] ) )
    diff = (value[x,y] - neighbors[x,y])^2
result = 0
for all samples where diff != 0:
    result += diff
return result
```

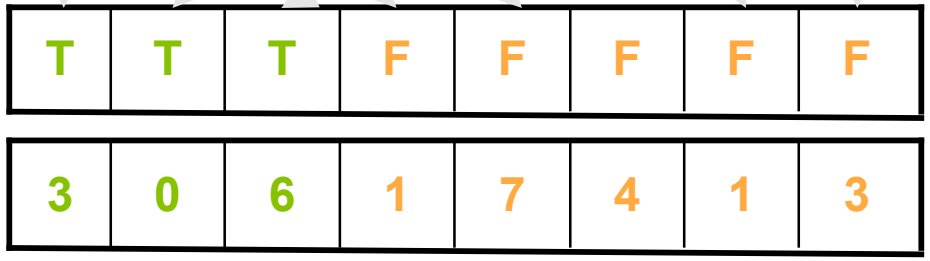
Common Situations in Parallel Computation

- Many parallel threads that need to partition data
  - Split
- Many parallel threads and variable output per thread
  - Compact / Expand / Allocate

Split Operation

- Given an array of true and false elements (and payloads)

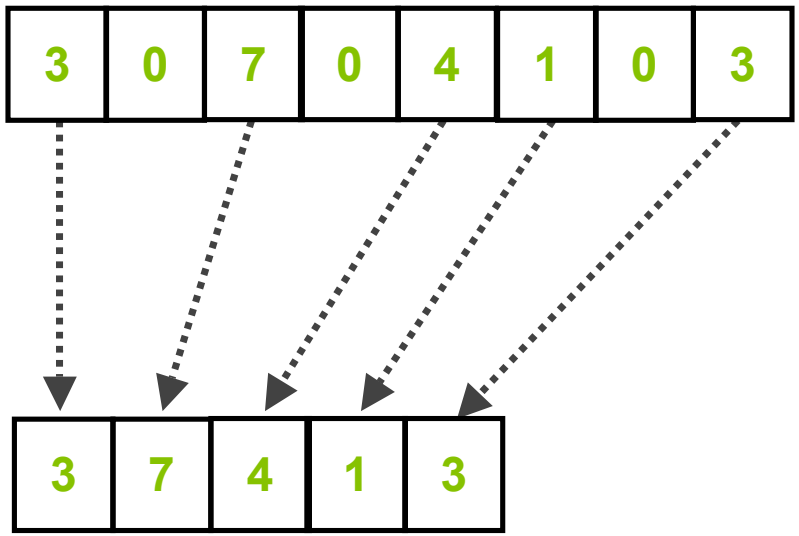
- Return an array with all true elements at the beginning



- Examples: sorting, building trees

Variable Output Per Thread: Compact

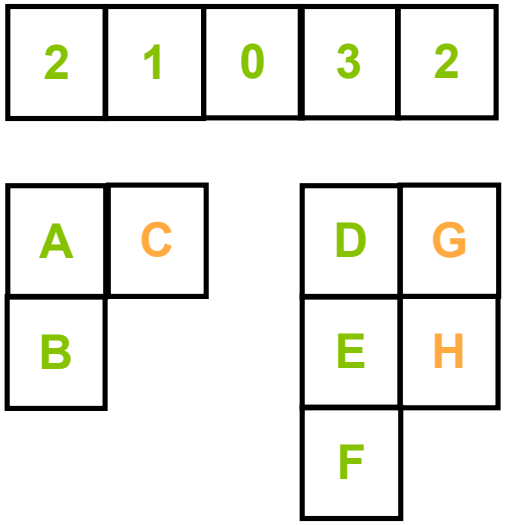
- Remove null elements



- Example: collision detection

Variable Output Per Thread

- Allocate Variable Storage Per Thread



- Examples: marching cubes, geometry generation

“Where do I write my output?”

- In all of these situations, each thread needs to answer that simple question
- The answer is:

  “That depends on how much the other threads need to write!”

  - In a serial processor, this is simple
- “Scan” is an efficient way to answer this question in parallel

Parallel Prefix Sum (Scan)

- Given an array A = [a_0, a_1, …, a_{n-1}]
  and a binary associative operator ⊕ with identity *I*,

  scan(A) = [*I*, a_0, ($a_0 \oplus a_1$), …, ($a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2}$)]

- Example: if ⊕ is addition, then scan on the set

  [3 1 7 0 4 1 6 3]

  returns the set

  [0 3 4 11 11 15 16 22]

Applications of Scan

- Scan is a simple and useful parallel building block for many parallel algorithms:
  - radix sort
  - quicksort (segmented scan)
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Run-length encoding
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms
  - Allocation
  - Etc.
- Fascinating, since scan is unnecessary in sequential computing!

Scan Literature

Pre-GPGPU

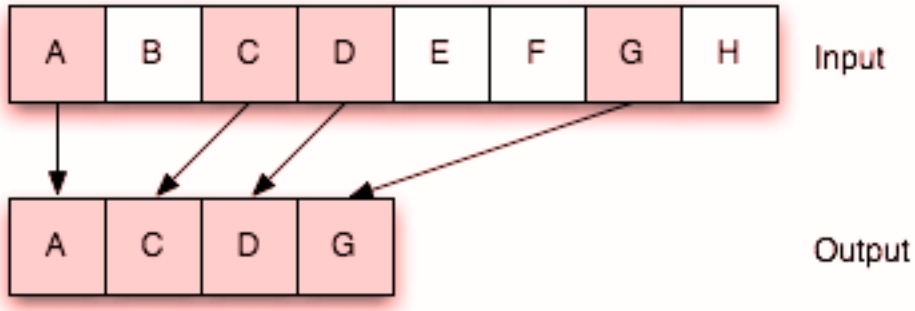
- First proposed in APL by Iverson
- Used as a data parallel primitive in the Connection Machine
  - Feature of C* and CM-Lisp
- Guy Blelloch used scan as a primitive for various parallel algorithms
  - *Blelloch, 1990, "Prefix Sums and Their Applications"*

GPGPU

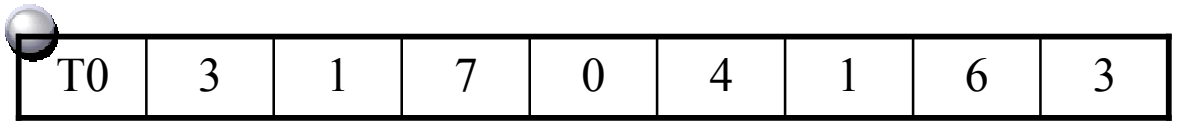
- *O*(*n* log *n*) GPU implementation by Daniel Horn (GPU Gems 2)
  - Applied to Summed Area Tables by Hensley et al. (EG05)
- *O*(*n*) work GPU scan by Sengupta et al. (EDGE06) and Greß et al. (EG06)
- *O*(*n*) work & space GPU implementation by Harris et al. (2007)
  - NVIDIA CUDA SDK and GPU Gems 3
  - Applied to radix sort, stream compaction, and summed area tables

Stream Compaction

- Input: stream of 1s and 0s
  [1 0 1 1 0 0 1 0]
- Operation:“sum up all elements before you”
- Output: scatter addresses for “1” elements
  [0 1 1 2 3 3 3 4]
- Note scatter addresses for red elements are packed!



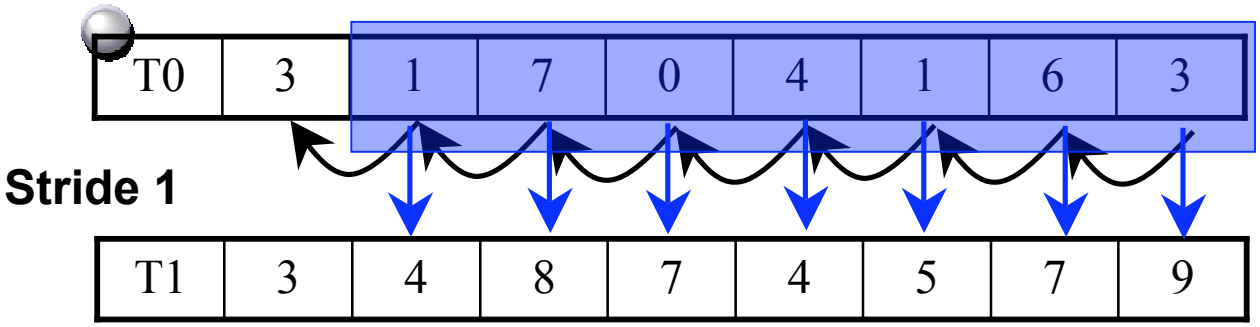
A Naive Parallel Scan Algorithm



log(*n*) iterations

Note: With graphics API, can't read and write the same texture, so must "ping-pong". Not necessary with newer APIs.

A Naive Parallel Scan Algorithm



log(*n*) iterations

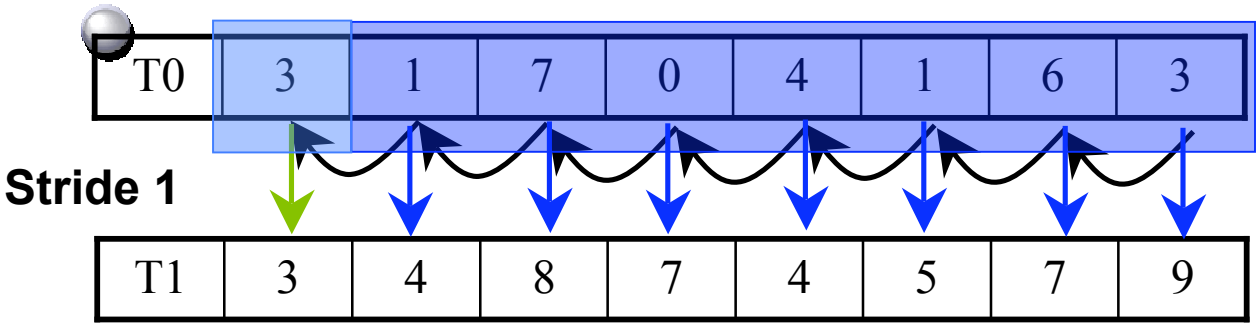
Note: With graphics API, can't read and write the same texture, so must "ping-pong"

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

- Specify domain from 2^i to n. Element *k* computes

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

A Naive Parallel Scan Algorithm



log(n) iterations

Note: With graphics API, can't read and write the same texture, so must "ping-pong"

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

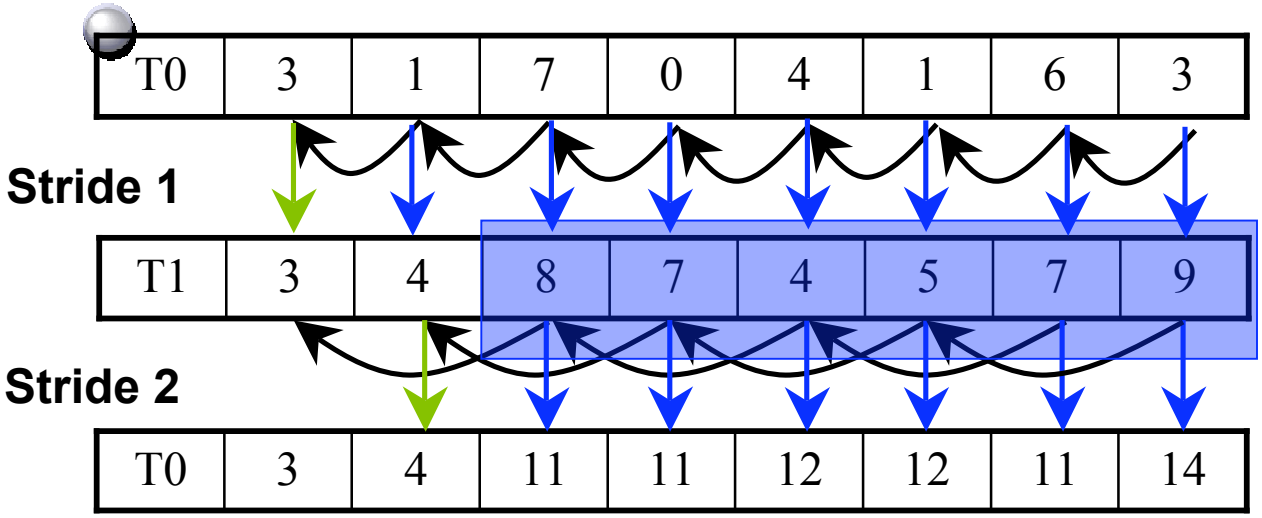
- Specify domain from 2^i to n. Element k computes

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

- Due to ping-pong, specify 2nd domain 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: With graphics API, can't read and write the same texture, so must "ping-pong"

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

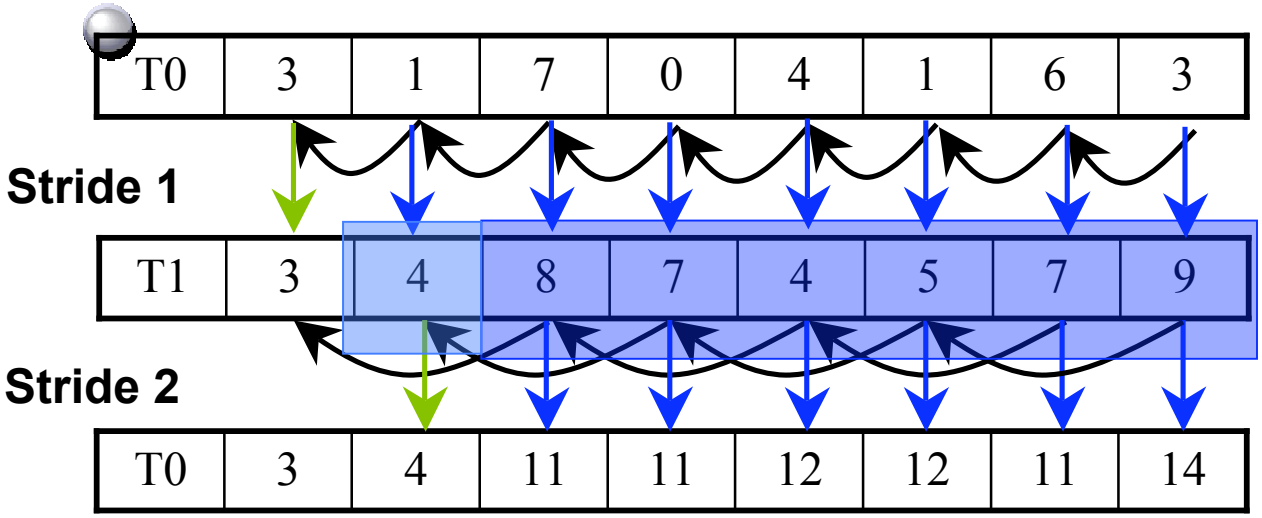
- Specify domain from 2^i to n. Element k computes

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

- Due to ping-pong, specify 2nd domain 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: With graphics API, can't read and write the same texture, so must "ping-pong"

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

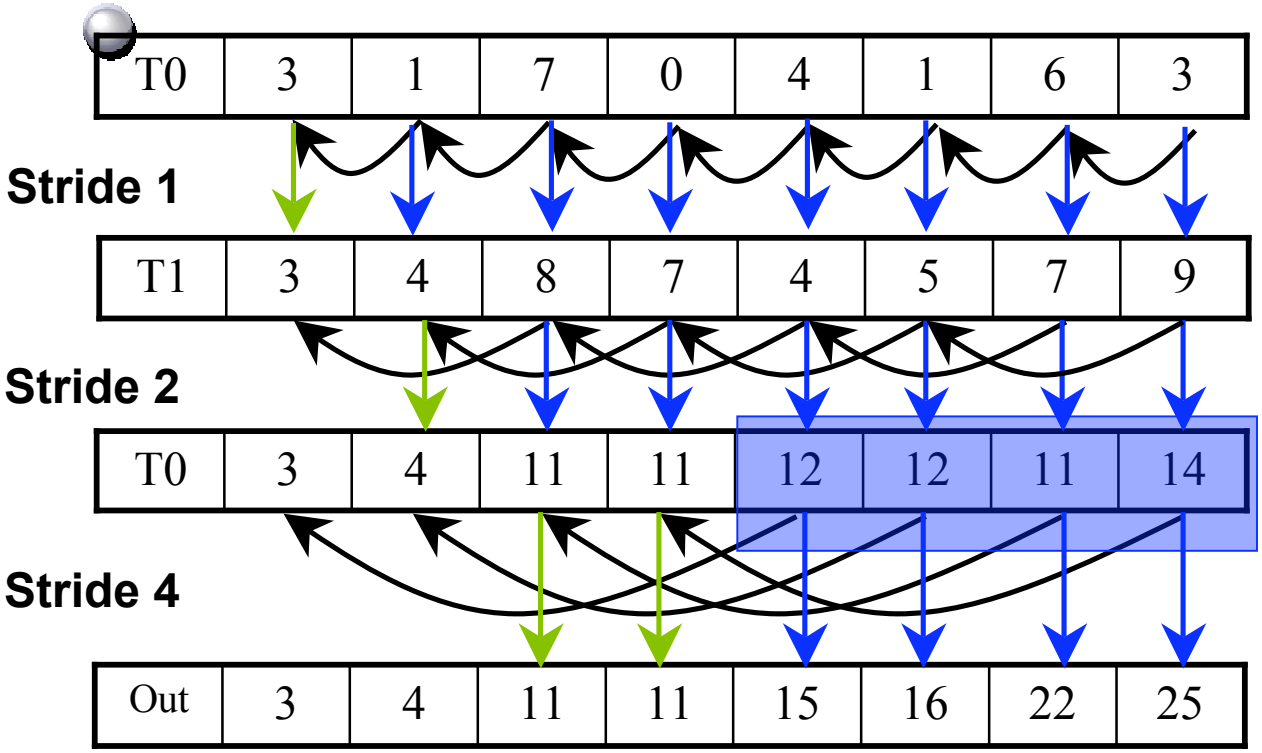
- Specify domain from 2^i to n. Element *k* computes

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

- Due to ping-pong, specify 2nd domain 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: With graphics API, can't read and write the same texture, so must "ping-pong"

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

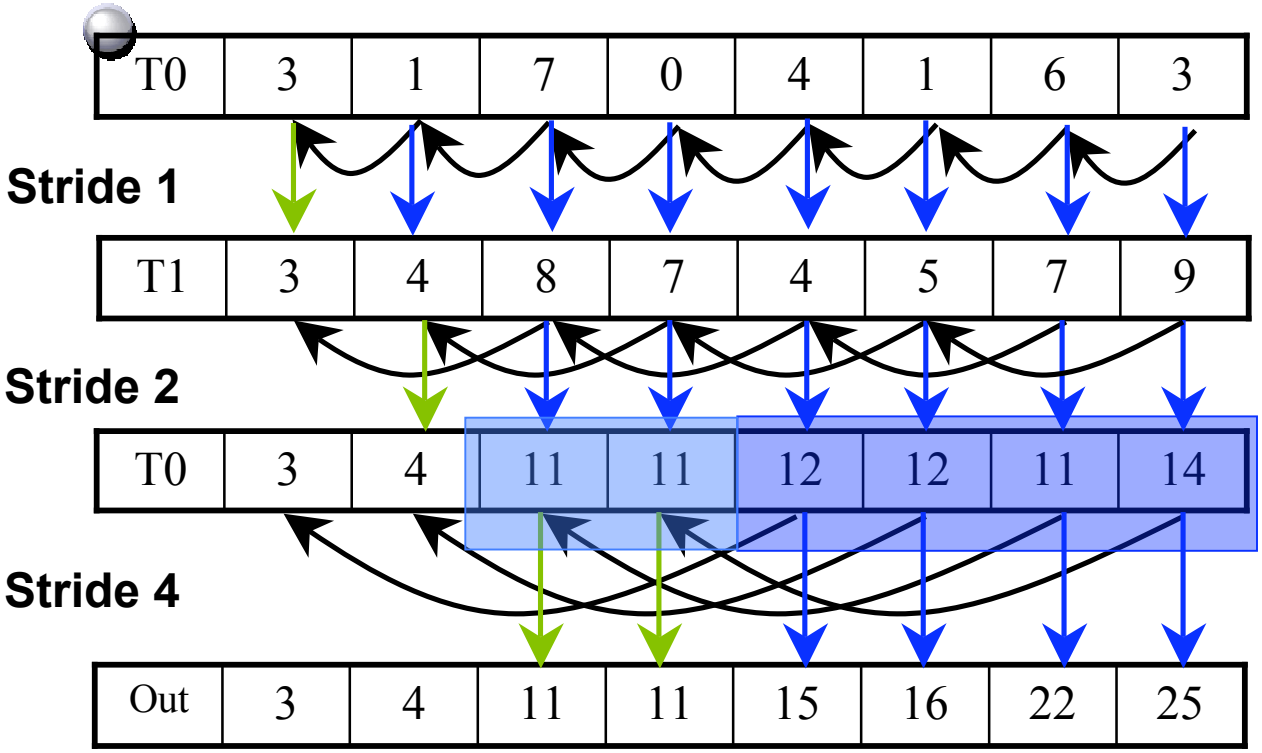
- Specify domain from 2^i to *n*. Element *k* computes

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

- Due to ping-pong, specify 2nd domain 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: With graphics API, can't read and write the same texture, so must "ping-pong"

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

- Specify domain from 2^i to n. Element k computes

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

- Due to ping-pong, specify 2nd domain 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 by 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”

Improving Efficiency

- A common parallel algorithms pattern:

  Balanced Trees

  - Build balanced binary tree on input data and sweep to and from the root
  - Tree is conceptual, not an actual data structure
- For scan:
  - Traverse from leaves to root building partial sums at internal nodes
    - Root holds sum of all leaves
  - Traverse from root to leaves building the scan from the partial sums

This algorithm originally described by Blelloch (1990)

Scan with Scatter

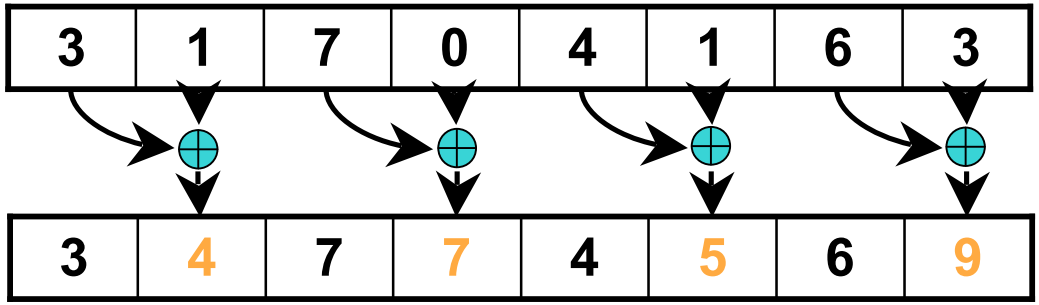
- Scatter in CUDA kernels makes algorithms that use scan easier to implement
  - NVIDIA CUDA SDK includes example implementation of scan primitive
  - http://www.gpgpu.org/developer/cudpp
- CUDA Parallel Data Cache improves efficiency
  - All steps executed in a single kernel
  - Threads communicate through shared memory
  - Drastically reduces bandwidth bottleneck!
  - Key for algorithmic efficiency: how to block computation and utilize parallel data cache in each block

Build the Sum Tree

3	1	7	0	4	1	6	3

Assume array is already in shared memory

Build the Sum Tree

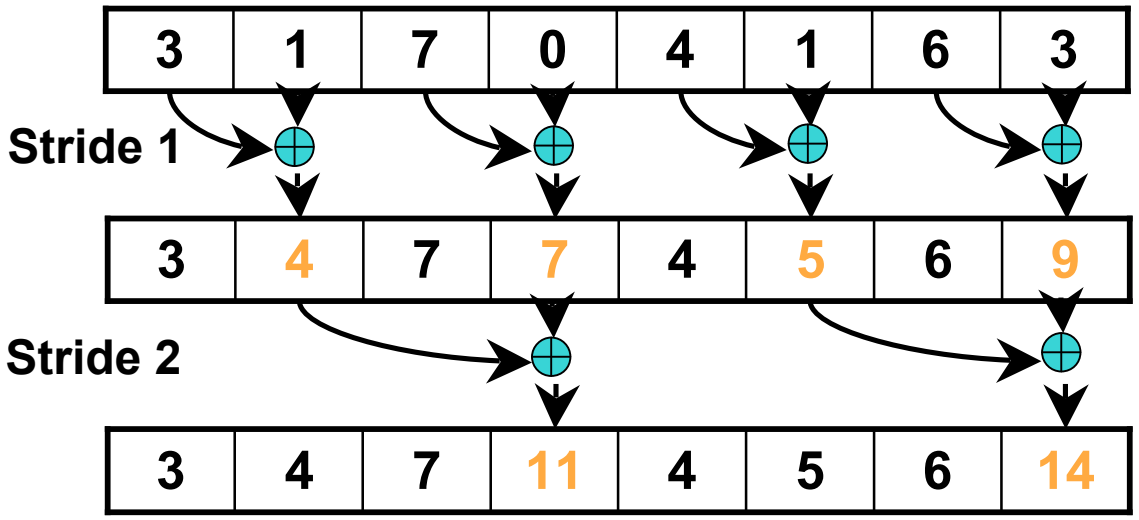


Iteration 1, *n*/2 threads

Each ⊕ corresponds to a single thread.

Iterate log(*n*) times. Each thread adds value *stride* elements away to its own value

Build the Sum Tree

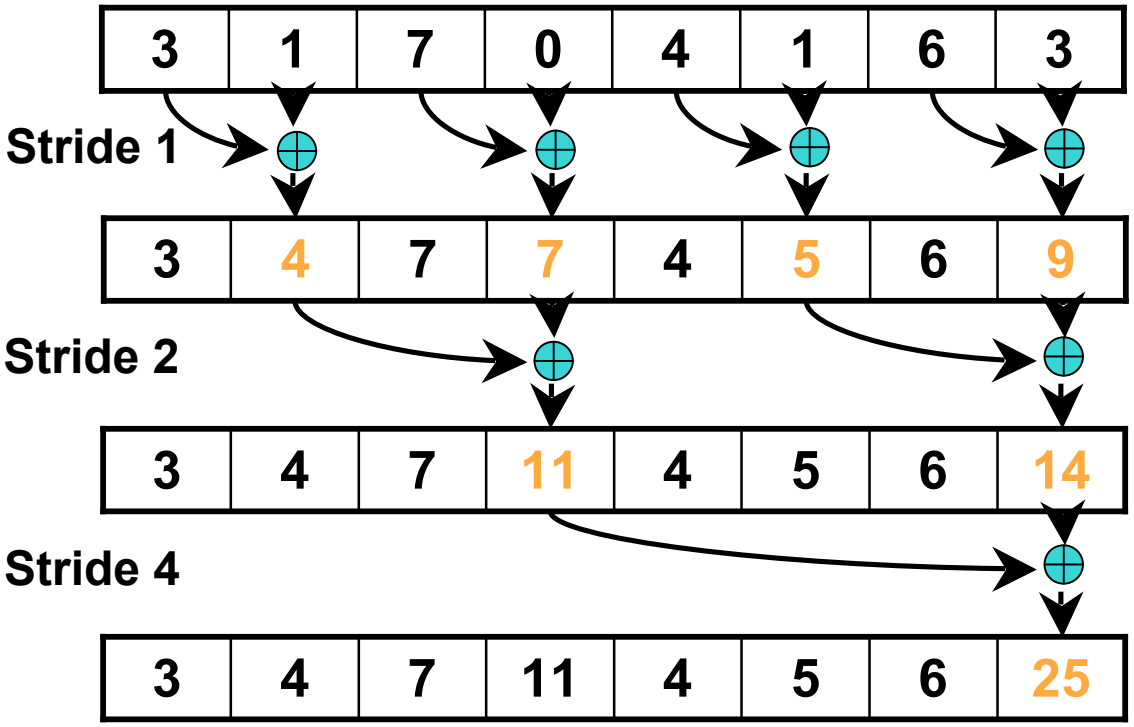


Iteration 2, *n*/4 threads

Each ⊕ corresponds to a single thread.

Iterate log(*n*) times. Each thread adds value *stride* elements away to its own value

Build the Sum Tree



Iteration log(*n*), 1 thread

Each ⊕ corresponds to a single thread.

Iterate log(*n*) times. Each thread adds value *stride* elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering

Zero the Last Element

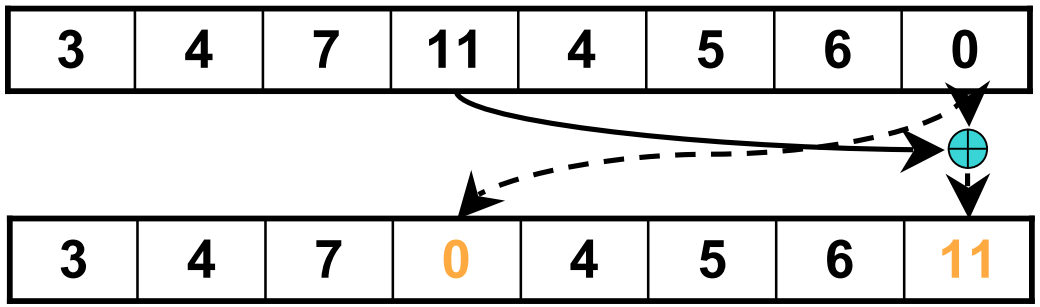
3	4	7	11	4	5	6	0

We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

Build Scan From Partial Sums

3	4	7	11	4	5	6	0

Build Scan From Partial Sums

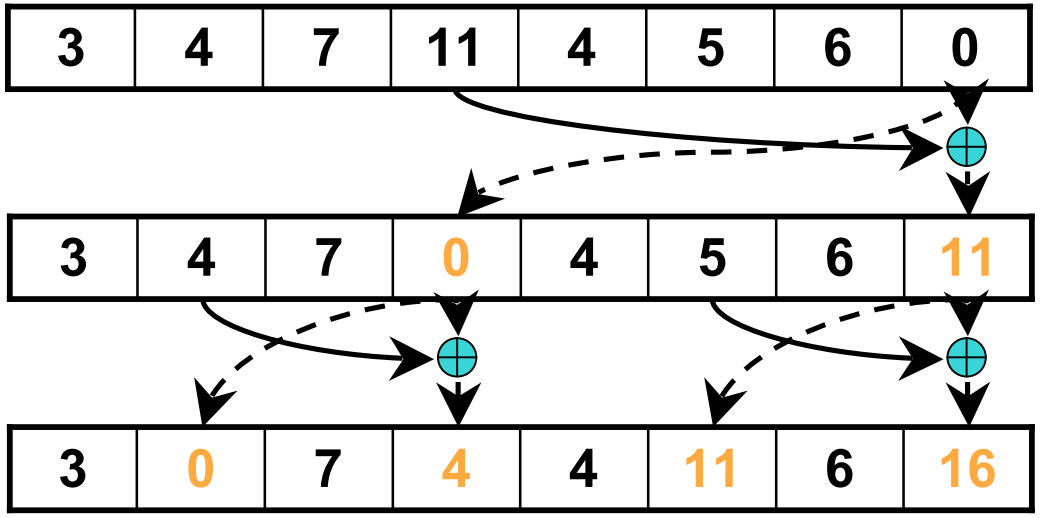


Iteration 1
1 thread

Each ⊕ corresponds to a single thread.

Iterate log(*n*) times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

Build Scan From Partial Sums

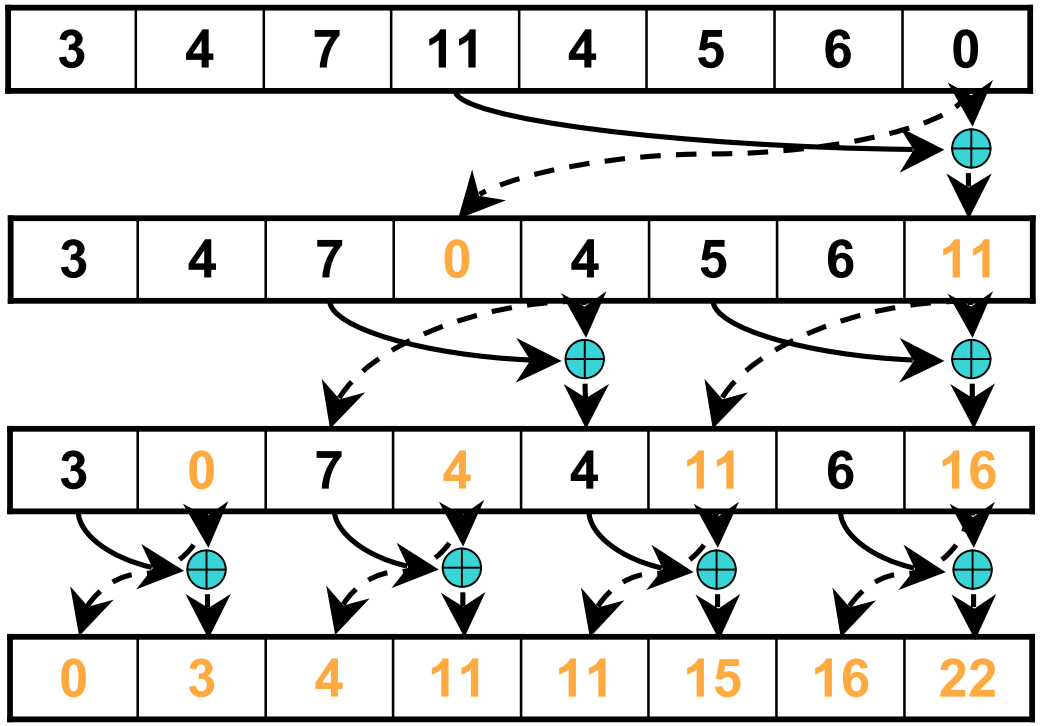


Iteration 2
2 threads

Each ⊕ corresponds to a single thread.

Iterate log(*n*) times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

Build Scan From Partial Sums

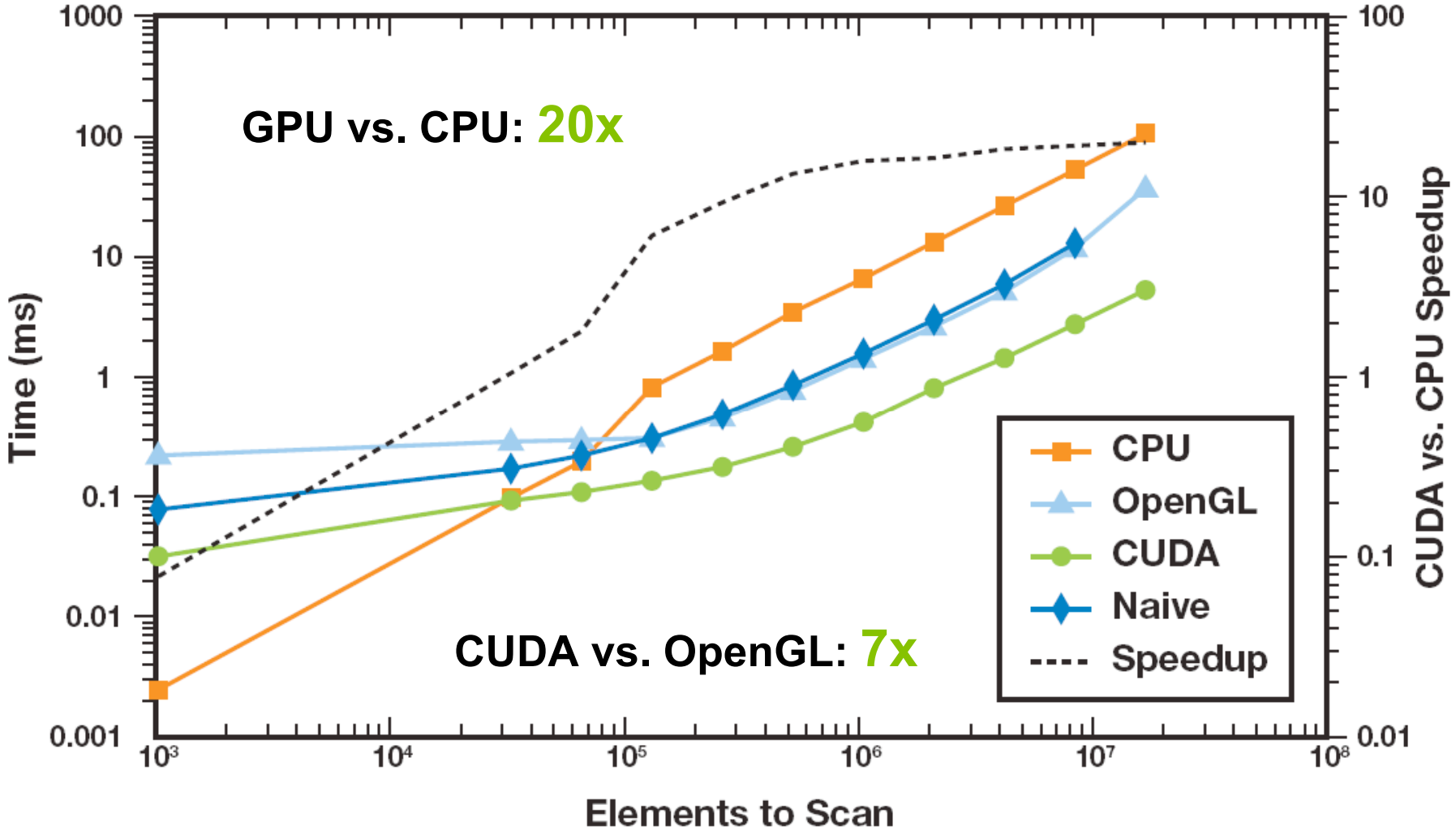


Iteration log(*n*)
***n*/2 threads**

Done! We now have a completed scan that we can write out to device memory.

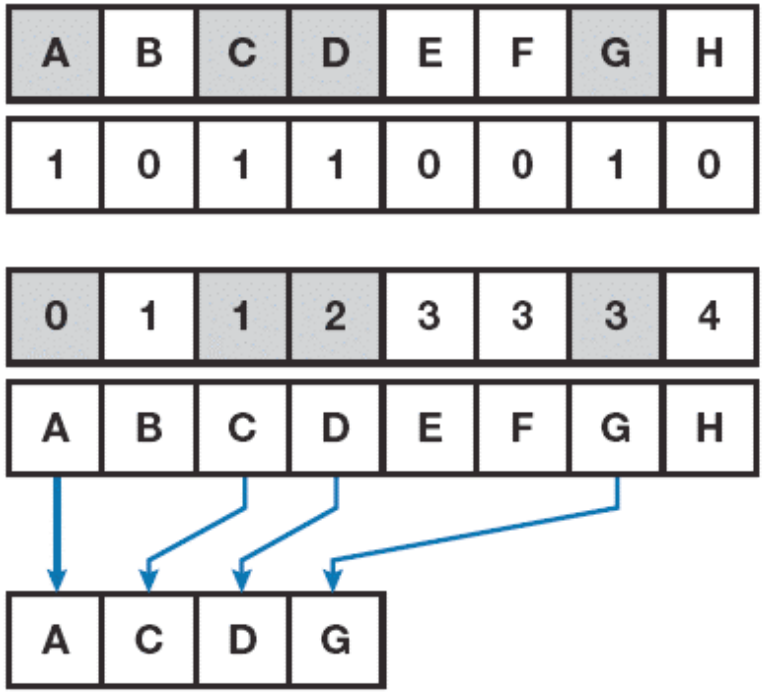
Total steps: 2 * log(*n*).
Total work: 2 * (*n*-1) adds = *O*(*n*) **Work Efficient!**

CUDA Scan Performance



GeForce 8800 GTX, Intel Core2 Duo Extreme 2.93 GHz

Application: Stream Compaction



Input: we want to preserve the gray elements

Set a "1" in each gray input

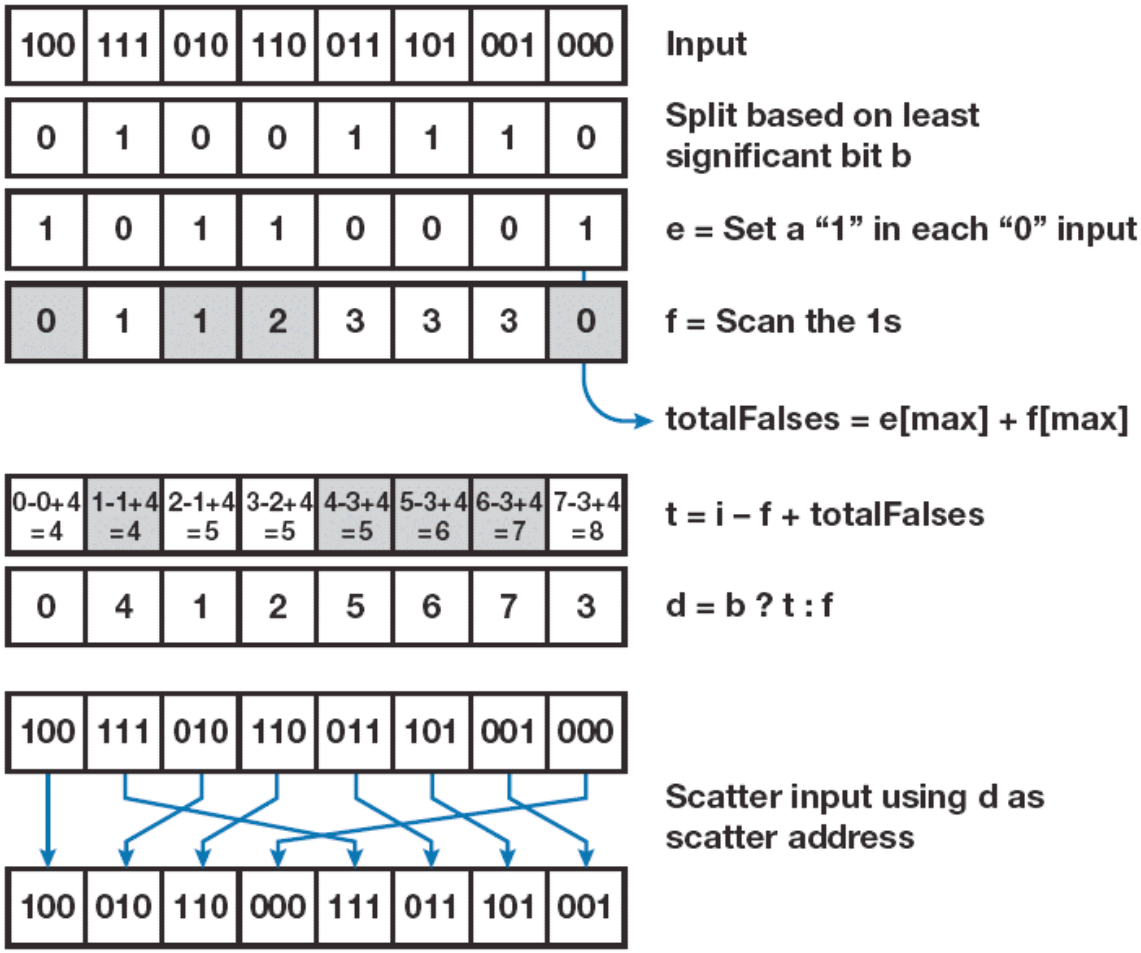
Scan

Scatter input to output, using scan result as scatter address

- 1M elements: ~0.6-1.3ms
- 16M elements: ~8-20ms
- Perf depends on # elements retained

Harris, M., S. Sengupta, and J.D. Owens. "Parallel Prefix Sum (Scan) in CUDA". *GPU Gems 3*

Application: Radix Sort



- Sort 4M 32-bit integers: 165ms
- Perform split operation on each bit using scan
- Can also sort each block and merge
  - Slower due to cost of merge

Application: Summed Area Tables

- Each pixel in SAT is the sum of all pixels below and to the left
- Can be used to perform box filter of arbitrary radius per pixel in constant time
- Easy to compute with scan
  - Scan all rows, then all columns
  - Transpose in between and scan only rows
  - GPU can scan all rows in parallel

- Scan all rows of 1024x1024 image in 0.85 ms
  - Build summed area table in 3.5 ms

Segmented Scan

- Segment Head Flags

0	0	1	0	0	1	0	0

Input Data Array

3	1	7	0	4	1	6	3

Segmented scan

0	3

0	7	7

0	1	7

- **Segmented scan enables another class of parallel algorithms**
  - **Parallel quicksort**
  - **Parallel sparse matrix-vector multiply in CSR format**

  - **Sengupta, S., M. Harris, Y. Zhang, and J.D. Owens. "Scan Primitives for GPU Computing". *Graphics Hardware 2007***

Parallel Sorting

- Given an unordered list of elements, produce list ordered by key value
  - Kernel: compare and swap
- GPU's constrained programming environment limits viable algorithms
  - Bitonic merge sort [Batcher 68]
  - Periodic balanced sorting networks [Dowd 89]
- Recent research results impressive
  - Govindaraju's GPUTeraSort (UNC/Microsoft work, published in ACM SIGMOD Dec. '05)
    - Hybrid radix-bitonic sort
    - 10x performance over CPU, PennySort champion

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 threads invoked determines number of searches executed in parallel
    - 1 thread == 1 search
- For details see:
  - "A Toolkit for Computation on GPUs". Ian Buck and Tim Purcell. In *GPU Gems*. Randy Fernando, ed. 2004

Conclusion

- Think parallel!
  - Effective programming on GPUs requires mapping computation & data structures into parallel formulations
- There is a huge scope for innovation in this space
  - Start contributing!
- Try out our CUDA Data Parallel Primitives library, CUDPP
  - Collaboration between NVIDIA and UC Davis
  - Information: http://www.gpgpu.org/developer/cudpp/

References

- "*A Toolkit for Computation on GPUs*". Ian Buck and Tim Purcell. In *GPU Gems*, Randy Fernando, ed. 2004.
- "*Parallel Prefix Sum (Scan) with CUDA*". Mark Harris, Shubhabrata Sengupta, John D. Owens. In *GPU Gems 3*, Herbert Nguyen, ed. Aug. 2007.
- "*Stream Reduction Operations for GPGPU Applications*". Daniel Horn. In *GPU Gems 2*, Matt Pharr, ed. 2005.
- "*Glift: Generic, Efficient, Random-Access GPU Data Structures*". Aaron Lefohn et al., ACM TOG, Jan. 2006.
- *"Scan Primitives for GPU Computing"*. Sengupta, S., M. Harris, Y. Zhang, and J.D. Owens. *Proceedings of Graphics Hardware 2007*