Scan Primitives for GPU Computing

Shubho Sengupta, Mark Harris[*], Yao Zhang, John Owens

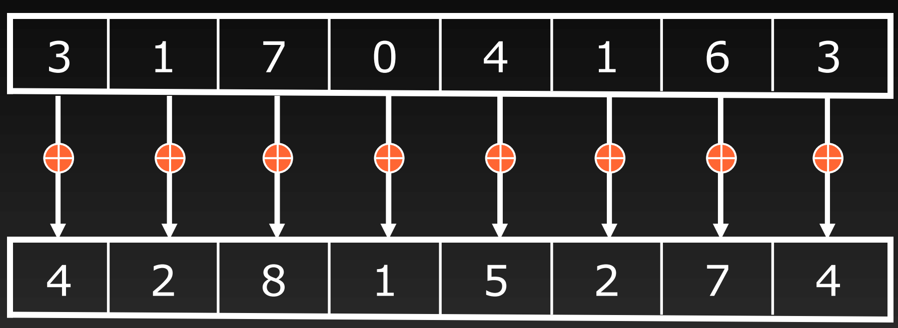
University of California Davis, *NVIDIA Corporation

Motivation

- Raw compute power and bandwidth of GPUs increasing rapidly
- Programmable unified shader cores
- Ability to program outside the graphics framework
- However lack of efficient data-parallel primitives and algorithms

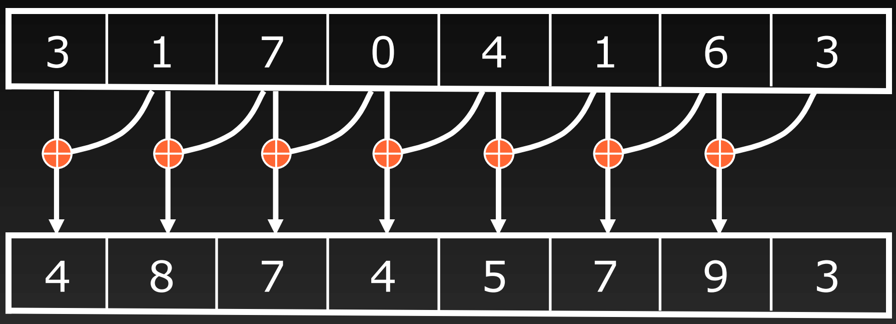
Motivation

- Current efficient algorithms either have streaming access
- 1:1 relationship between input and output element



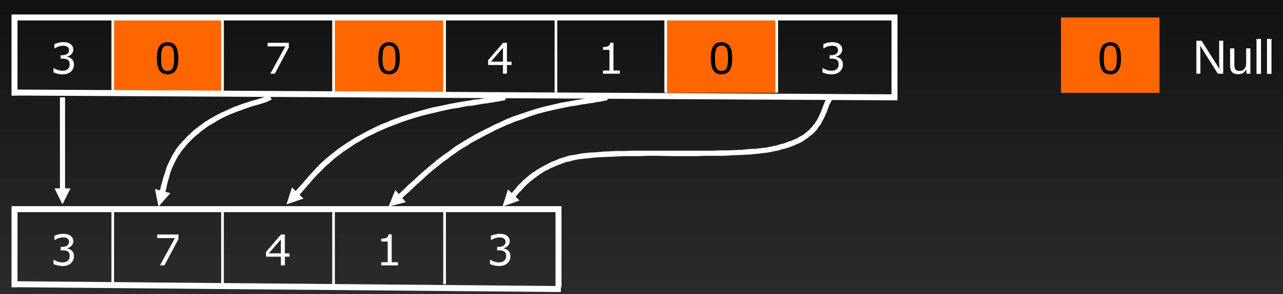
Motivation

- Or have small “neighborhood” access
- N:1 relationship between input and output element where N is a small constant



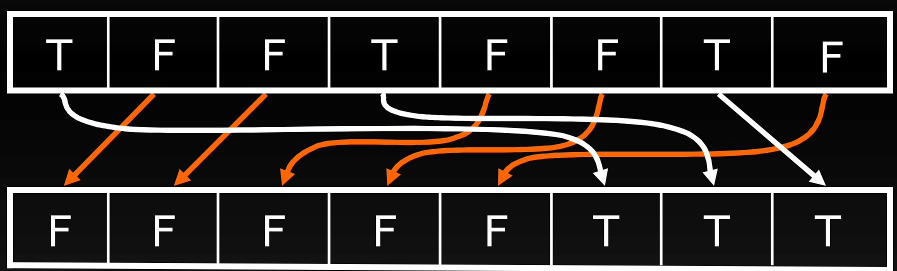
Motivation

- However interesting problems require more general access patterns
  - Changing one element affects everybody
- Stream Compaction



Motivation

- Split



- Needed for Sort

Motivation

- Common scenarios in parallel computing
  - Variable output per thread
  - Threads want to perform a split – radix sort, building trees
- “What came before/after me?”
- “Where do I start writing my data?”
- Scan answers this question

System Overview

Algorithms
Sort, Sparse matrix operations,…

Higher Level Primitives
Enumerate, Distribute,…

Low Level Primitives
Scan and variants

Libraries and Abstractions for data parallel programming

Scan

- Each element is a sum of all the elements to the left of it (Exclusive)
- Each element is a sum of all the elements to the left of it and itself (Inclusive)

3	1	7	0	4	1	6	3	Input
0	3	4	11	11	15	16	22	Exclusive
3	4	11	11	15	16	22	25	Inclusive

Scan – the past

- First proposed in APL (1962)
- Used as a data parallel primitive in the Connection Machine (1990)
- Guy Blelloch used scan as a primitive for various parallel algorithms (1990)

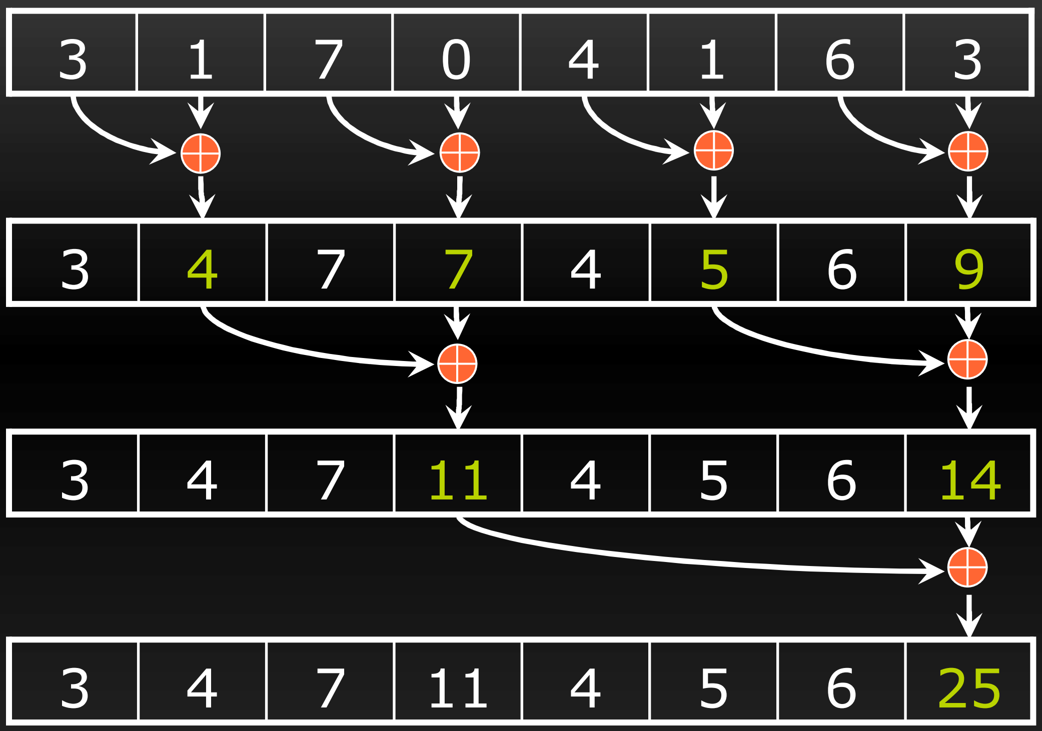
Scan – the present

- First GPU implementation by Daniel Horn (2004), $O(n \log n)$
- Subsequent GPU implementations by
  - Hensley (2005) $O(n \log n)$, Sengupta (2006) $O(n)$, Greß (2006) $O(n)$ 2D
- NVIDIA CUDA implementation by Mark Harris (2007), $O(n)$, space efficient

Scan – the implementation

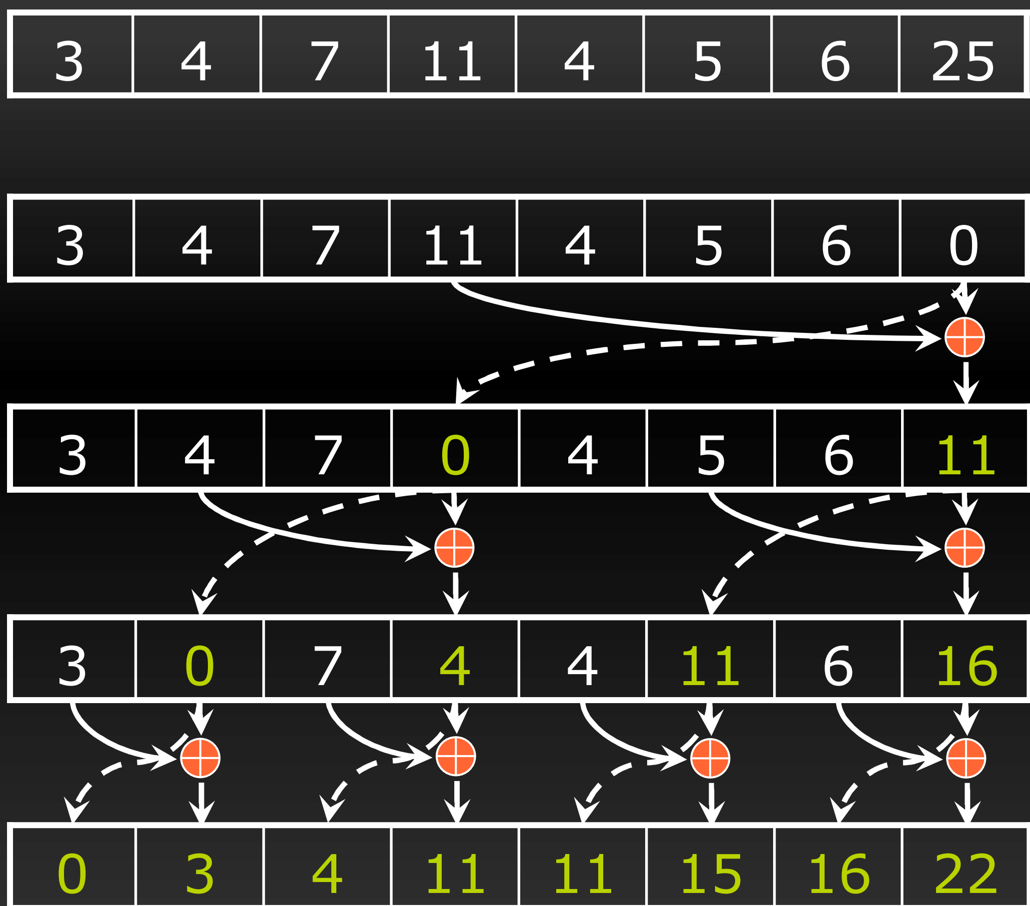
- O(n) algorithm – same work complexity as the serial version
- Space efficient – needs O(n) storage
- Has two stages – reduce and down-sweep

Scan - Reduce



- log n steps
- Work halves each step
- O(n) work
- In place, space efficient

Scan - Down Sweep



- log n steps
- Work doubles each step
- O(*n*) work
- In place, space efficient

Segmented Scan

- Input

3	1	7	0	4	1	6	3

- Scan within each segment in parallel
- Output

0	3	0	7	7	0	1	7

Segmented Scan

- Introduced by Schwartz (1980)
- Forms the basis for a wide variety of algorithms
  - Quicksort, Sparse Matrix-Vector Multiply, Convex Hull

Segmented Scan - Challenges

- Representing segments
- Efficiently storing and propagating information about segments
- Scans over all segments should happen in parallel
  - Overall work and space complexity should be $O(n)$ regardless of the number of segments

Representing Segments

- Possible Representations are
  - Vector of segment lengths
  - Vector of indices which are segment heads
  - Vector of flags: 1 for segment head, 0 if not
- First two approaches hard to parallelize as different size as input
- We use the third as it is the same size as input

Segmented Scan – Flag Storage

- Space-Inefficient to store one flag in an integer
- Store one flag in a byte striped across 32 words
- Reduces bank conflicts

Segmented Scan – implementation

- Similar to Scan
  - O(*n*) space and work complexity
  - Has two stages – reduce and down-sweep

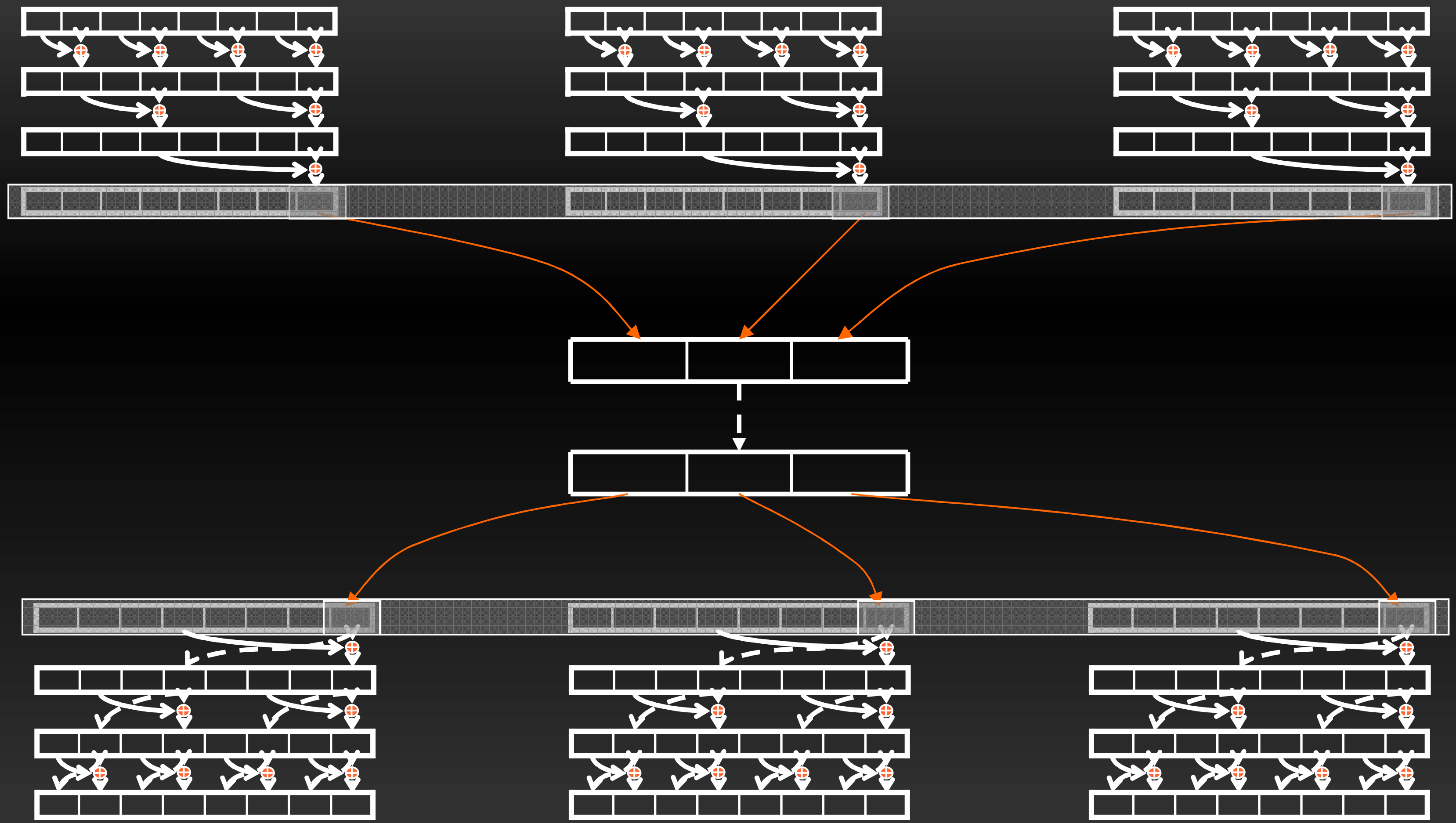
Segmented Scan – implementation

- Unique to segmented scan
  - Requires an additional flag per element for intermediate computation
    - Additional flags get set in reduce step
    - Additional book-keeping with flags in down-sweep
  - These flags prevent data movement between segments

Platform – NVIDIA CUDA and G80

- Threads grouped into blocks
- Threads in a block can cooperate through fast on-chip memory
- Hence programmer must partition problem into multiple blocks to use fast memory
- Adds complexity but usually much faster code

Segmented Scan – Large Input



Segmented Scan – Advantages

- Operations in parallel over all the segments
- Irregular workload since segments can be of any length
- Can simulate divide-and-conquer recursion since additional segments can be generated

Primitives - Enumerate

- Input: a true/false vector

F	F	T	F	T	T	T	F

- Output: count of true values to the left of each element

0	0	0	1	1	2	3	3

- Useful in stream compact
- Output for each true element is the address for that element in the compacted array

Primitives - Distribute

- Input: a vector with segments

3	1	7	4	0	1	6	3

- Output: the first element of a segment copied over all other elements

3	3	3	4	4	4	6	6

Primitives – Distribute

- Set all elements except the head elements to zero

3	0	0	4	0	0	6	0

- Do inclusive segmented scan

3	3	3	4	4	4	6	6

- Used in quicksort to distribute pivot

Primitives – Split and Segment

- Input: a vector with true/false elements. Possibly segmented

- Output: Stable split within each segment – falses on the left, trues on the right

Primitives – Split and Segment

- Can be implemented with Enumerate
  - One enumerate for the falses going left to right
  - One enumerate for the trues going right to left
- Used in quicksort

Applications – Quicksort

- Traditional algorithm GPU unfriendly
- Recursive
- Segments vary in length, unequal workload
- Primitives built on segmented scan solves both problems
  - Allows operations on all segments in parallel
  - Simulates recursion by generating new segments in each iteration

Applications – Quicksort

5	3	7	4	6	8	9	3	Input
5	5	5	5	5	5	5	5	Distribute pivot
F	F	T	F	T	T	T	F	Input > pivot
5	3	4	3	7	6	8	9	Split and Segment

Applications – Quicksort

[5 | 3 | 4 | 3] [7 | 6 | 8 | 9]

[5 | 5 | 5 | 5] [7 | 7 | 7 | 7] Distribute pivot

[T | F | F | F] [T | F | T | T] Input ≥ pivot

[3 | 4 | 3] [5] [6] [7 | 8 | 9] Split and segment

Applications – Quicksort

[3 4 3] [5] [6] [7 8 9]	
[3 3 3] [5] [6] [7 7 7]	Distribute pivot
[F T F] [F] [F] [F T T]	Input > pivot
[3] [3 4] [5] [6] [7] [8 9]	Split and segment

Applications – Sparse M-V multiply

- Dense matrix operations are much faster on GPU than CPU
- However Sparse matrix operations on GPU much slower
- Hard to implement on GPU
  - Non-zero entries in row vary in number

Applications – Sparse M-V multiply

- Three different approaches
  - Rows sorted by number of non-zero entries [Bolz]
  - Stored as diagonals and processed them in sequence [Krüger]
  - Rows computed in parallel but runtime determined by longest row [Brook]

Applications – Sparse M-V multiply

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

a	b	c	d	e	f	Non-zero elements
0	2	0	1	2	2	Column Index
0	2	5				Row begin Index

Applications – Sparse M-V multiply

Column Index

0	2	0	1	2	2

a	b	c	d	e	f

x

x_0	x_2	x_0	x_1	x_2	x_2

=

ax_0	bx_2	cx_0	dx_1	ex_2	fx_2

Backward inclusive segmented scan
Pick first element in segment

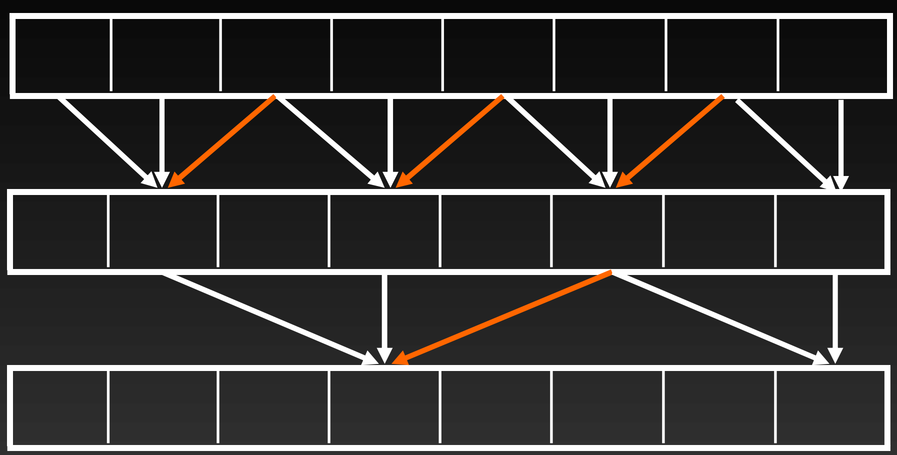
$ax_0 + bx_2$	$cx_0 + dx_1 + ex_2$	fx_2

Applications – Tridiagonal Solver

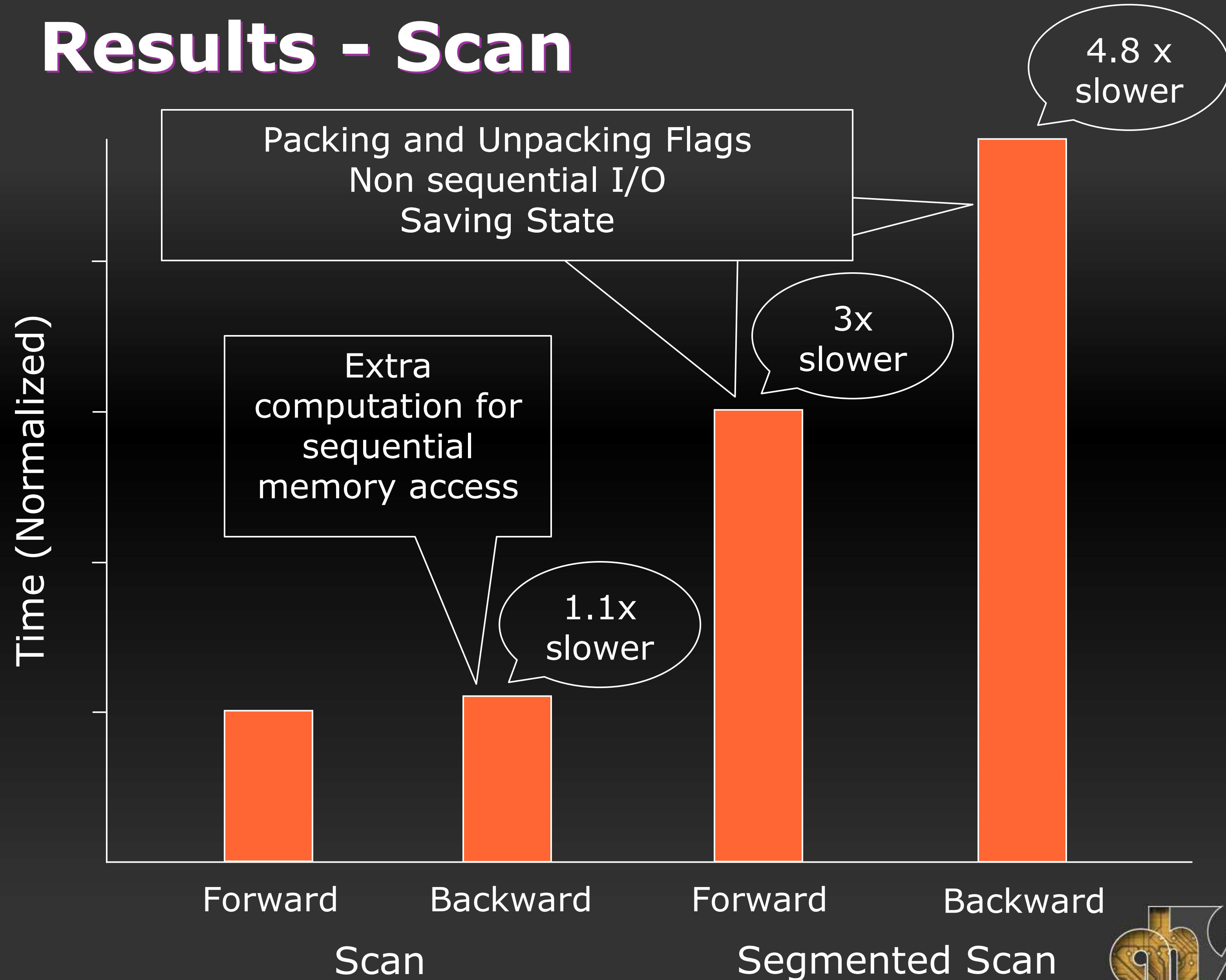
- Implemented Kass and Miller's shallow water solver
  - Water surface described as a 2D array of heights
- Global movement of data
  - From one end to the other and back
- Suits the Reduce/Down-sweep structure of scan

Applications – Tridiagonal Solver

- Tridiagonal system of *n* rows solved in parallel
- Then for each of the *m* columns in parallel
- Read pattern is similar to but more complex than scan



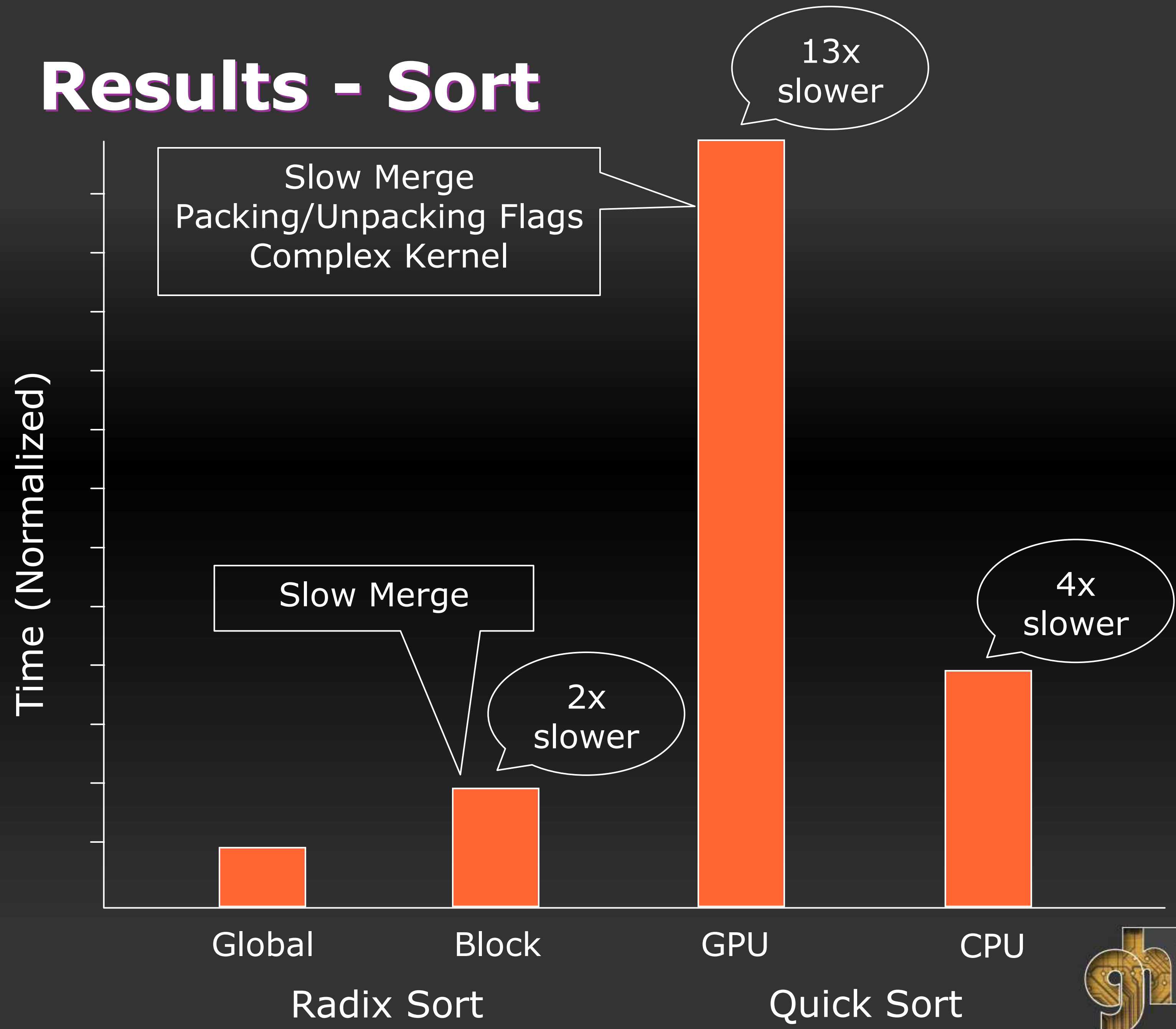
Results - Scan



Results – Sparse M-V Multiply

- Input: “raefsky” matrix, 3242 x 3242, 294276 elements
- GPU (215 MFLOPS) half as fast as CPU “oski” (522 MFLOPS)
  - Hard to do irregular computation
- Most time spent in backward segmented scan

Results - Sort



Results – Tridiagonal solver

- 256 x 256 grid: 367 simulation steps per second
- Dominated by the overhead of mapping and unmapping vertex buffers
- 3x faster than a CPU cyclic reduction solver
- 12x faster when using shared memory

Improved Results Since Publication

- Twice as fast for all variants of scan and sparse matrix-vector multiply
- Scan
  - More work per thread – 8 elements vs 2 before
- Segmented Scan
  - No packing of flags
  - Sequential memory access
- More optimizations possible

Contribution and Future Work

- Algorithm and implementation of segmented scan on GPU
- First implementation of quicksort on GPU
- Primitives appropriate for complex algorithms
  - Global data movement, unbalanced workload, recursive
  - Scan never occurs in serial computation
- Tiered approach, standard library and interfaces

Acknowledgments

- Jim Ahrens, Guy Blelloch, Jeff Inman, Pat McCormick
- David Luebke, Ian Buck
- Jeff Bolz
- Eric Lengyel
- Department of Energy
- National Science Foundation

Shallow Water Simulation