Scan Primitives for GPU Computing

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

Presented by Mary Fletcher
Slides adapted from authors' slides

Motivation

- Raw compute power and bandwidth of GPUs increasing rapidly
- Move to general-purpose applications on GPU
- Lack of efficient, general data-parallel primitives and algorithms

2

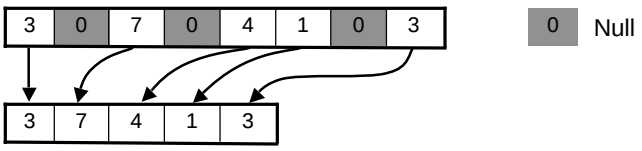
Motivation

- Current efficient algorithms either have streaming access
  - 1:1 relationship between input and output element
- Or have small “neighborhood” access
  - k:1 relationship between input and output element where k is a small constant
  - Example, image convolution

3

Motivation

- However interesting problems require more general access patterns
  - Output depends on arbitrary number of inputs
- Stream Compaction



4

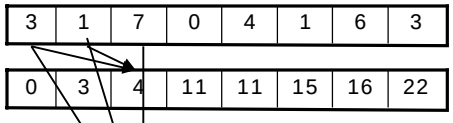
Motivation

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

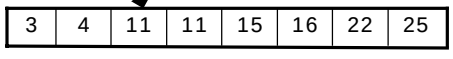
5

Scan (aka prefix sum)

- 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)



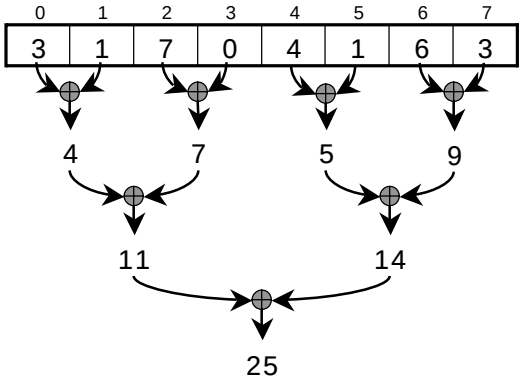
6

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

7

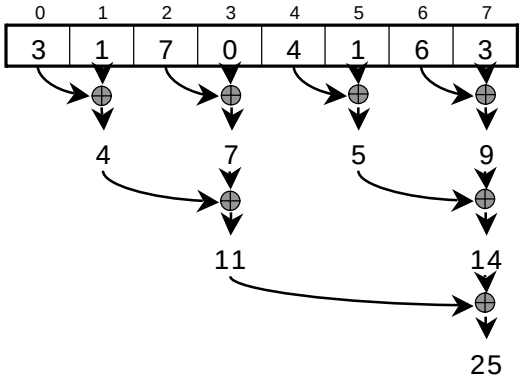
Scan - Reduce Stage



- log n steps
- Work halves each step
- O(*n*) total work

8

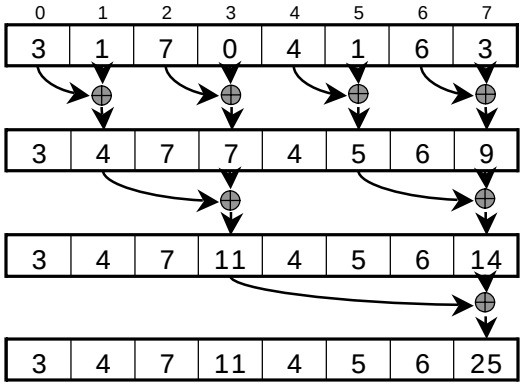
Scan - Reduce Stage



- log n steps
- Work halves each step
- O(*n*) total work

9

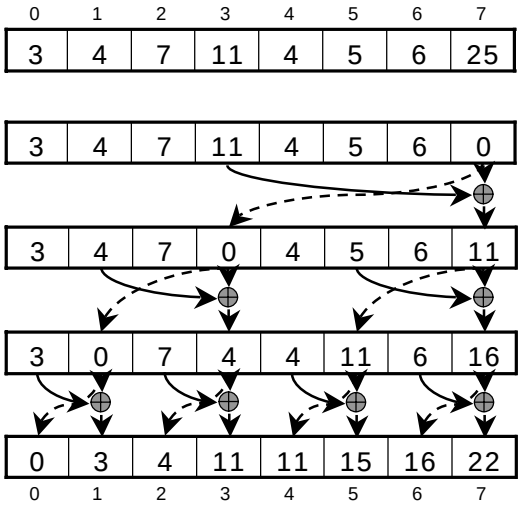
Scan - Reduce Stage



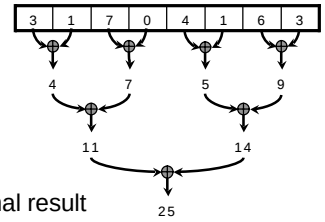
- log n steps
- Work halves each step
- O(*n*) total work
- In place, space efficient

10

Scan - Down Sweep Stage



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



Final result

11

Scan - Implementation

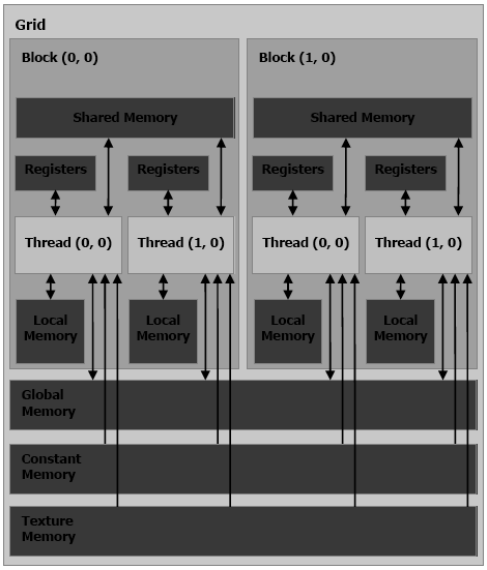
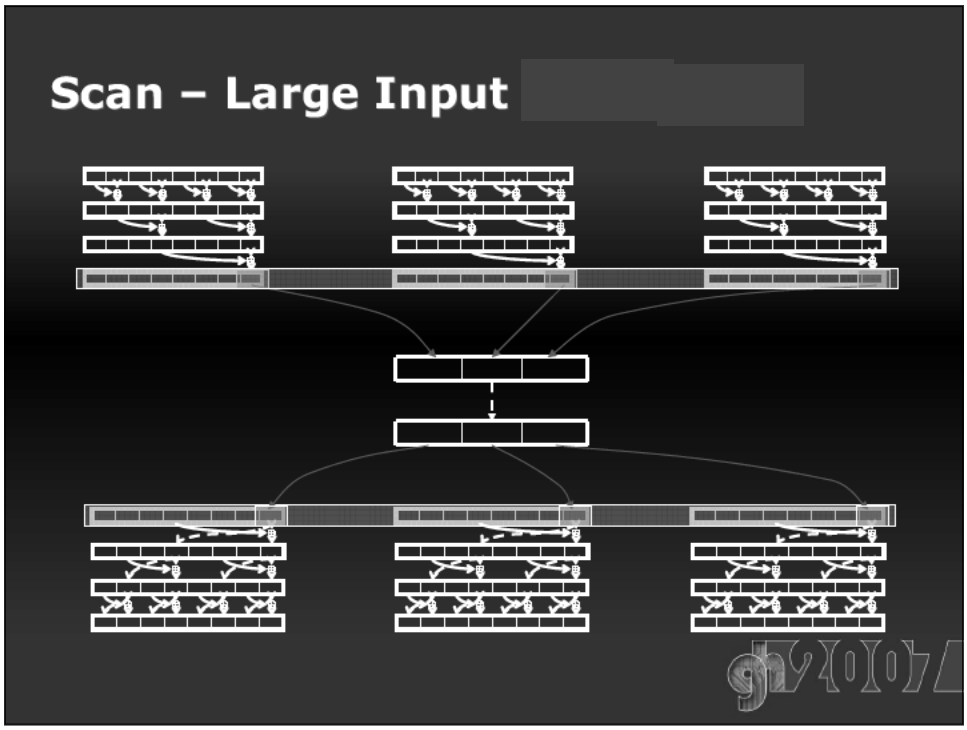


Image from the CUDA programming guide

12

Scan – Large Input



Segmented Scan

- Input - array broken into segments

3	1	7	2	4	1	6	3

- Scan within each segment in parallel
- Output

0	3	0	7	9	0	1	7

14

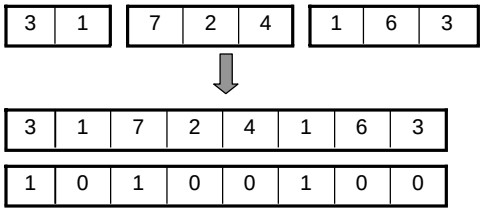
Segmented Scan - Challenges

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

15

Representing Segments

- Vector of flags: 1 if segment head, 0 if not



- Store one flag in a byte striped across 32 words
  - Reduces bank conflicts

16

Segmented Scan – Implementation

- Similar to Scan
  - O(*n*) space and work complexity
  - Has two stages – reduce and down-sweep
- Unique to segmented scan
  - Requires an additional flag per element for intermediate computation
  - These flags prevent data movement between segments

17

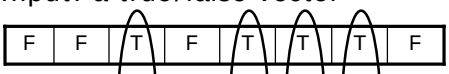
Segmented Scan – Advantages

- Operates in parallel over all the segments
- Good for irregular workload since segments can be of any length
- Can simulate divide-and-conquer recursion since additional segments can be generated

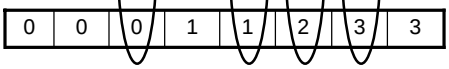
18

Primitives - Enumerate

- Input: a true/false vector



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

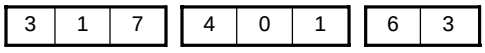


- Useful in stream compact
  - Output for each true element is the address for that element in the compacted array

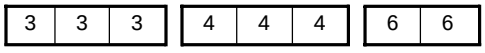
19

Primitives - Distribute

- Input: a vector with segments



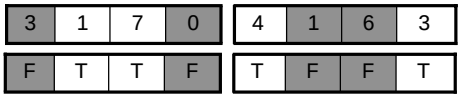
- Output: the first element of a segment copied over all other elements



20

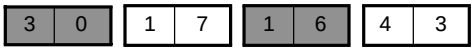
Primitives – Split and Segment

- Input: a vector with true/false elements, possibly segmented



False

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



21

Applications – Quicksort

- Traditional algorithm GPU unfriendly
- Recursive
- Subarrays vary in length, unequal workload
- Primitives built on segmented scan solve both problems
  - Allow operations on all segments in parallel
  - Simulate recursion by generating new segments in each iteration

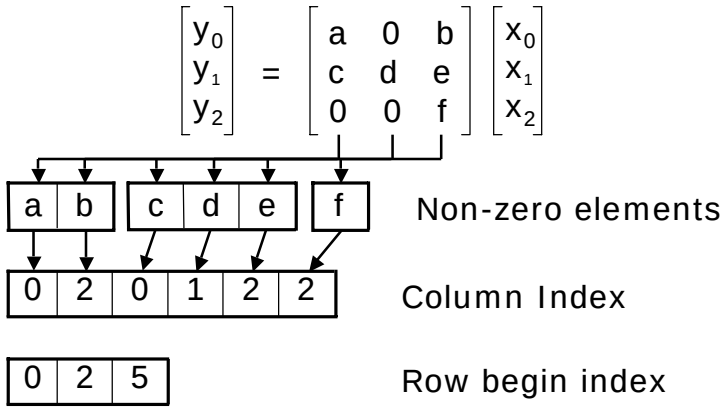
22

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

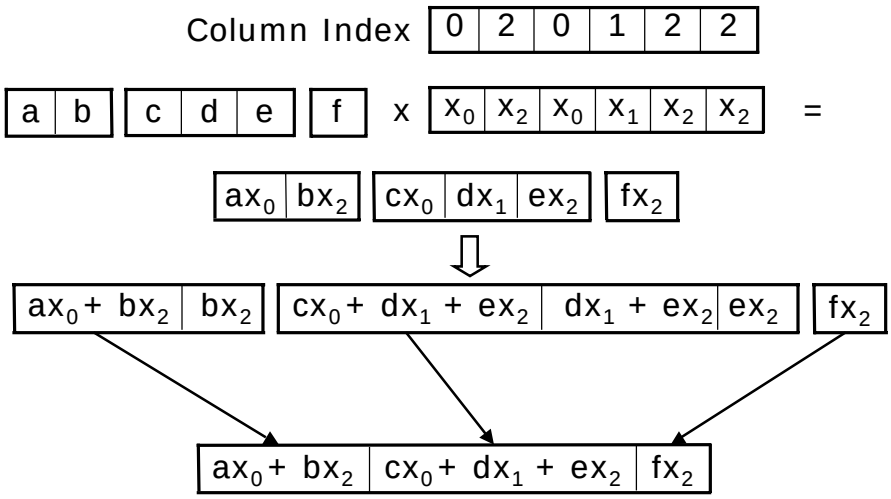
23

Applications – Sparse M-V multiply



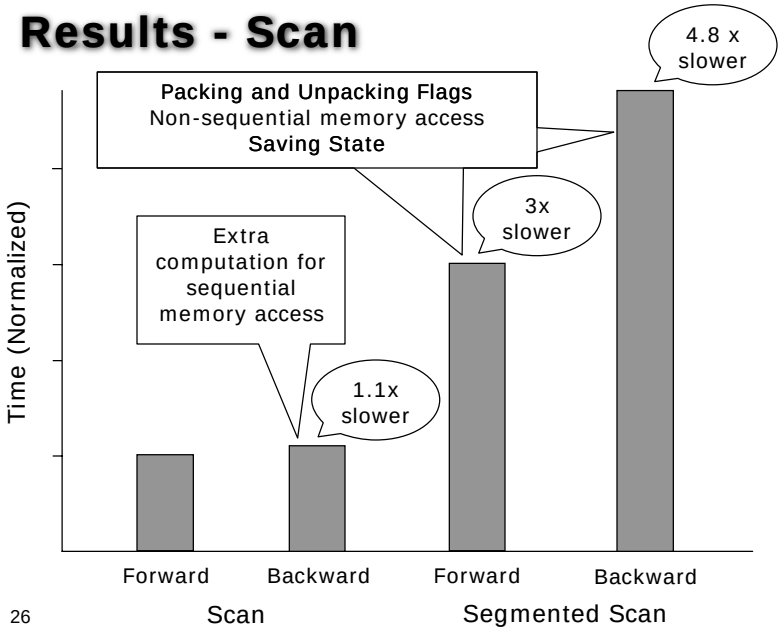
24

Applications – Sparse M-V multiply



25

Results - Scan



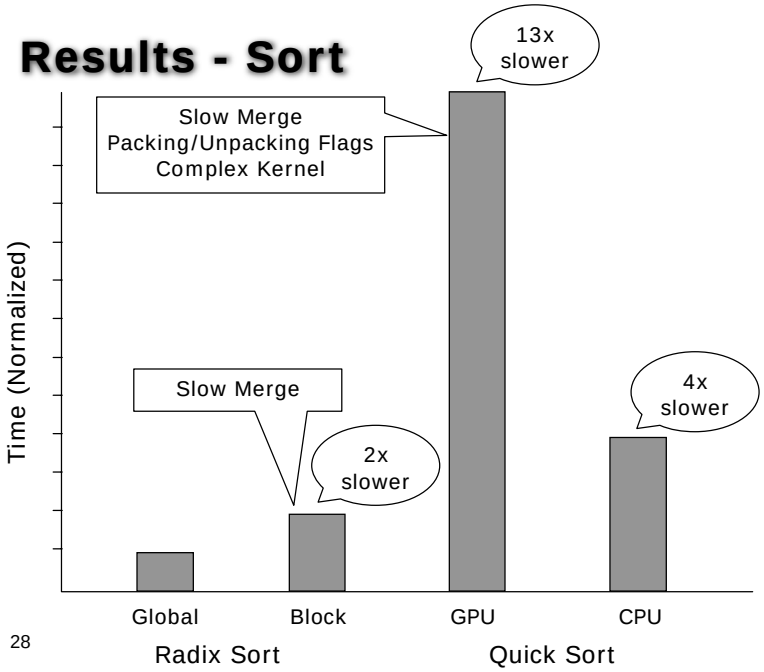
26

Results – Sparse M-V Multiply

- Input: “raefsky” matrix, 3242 x 3242, 294276 elements
- GPU - 215 MFLOPS
- OSKI on Pentium 4 - 522 MFLOPS
- Most time spent in backward segmented scan

27

Results - Sort



28

Improved Results Since Publication

- Twice as fast for all variants of scan and sparse matrix-vector multiply
- More optimizations possible

29

Conclusions

- 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
- CUDPP: CUDA Data Parallel Primitives Library
  - http://www.gpgpu.org/scan-gpugems3

30