Parallel Algorithms

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2012

Announcements

- Presentation topics due 02/07
- Homework 2 due 02/13

Agenda

- Finish atomic functions from Monday
- Parallel Algorithms
  - Parallel Reduction
  - Scan
  - Stream Compression
  - *S*ummed *A*rea *T*ables

Parallel Reduction

- Given an array of numbers, design a parallel algorithm to find the sum.
- Consider:
  - *Arithmetic intensity*: compute to memory access ratio

Parallel Reduction

- Given an array of numbers, design a parallel algorithm to find:
  - The sum
  - The maximum value
  - The product of values
  - The average value
- How different are these algorithms?

Parallel Reduction

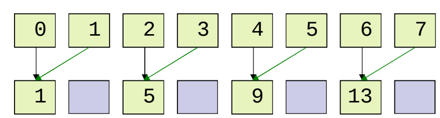
- *Reduction*: An operation that computes a single result from a set of data
- Examples:
  - Minimum/maximum value
  - Average, sum, product, etc.
- *Parallel Reduction*: Do it in parallel. Obviously

Parallel Reduction

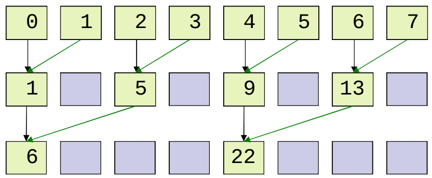
- Example. Find the sum:

0	1	2	3	4	5	6	7

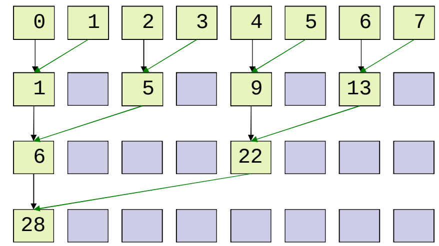
Parallel Reduction



Parallel Reduction

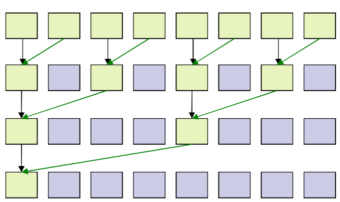


Parallel Reduction



Parallel Reduction

- Similar to brackets for a basketball tournament
- log(n) passes for n elements



All-Prefix-Sums

- *All-Prefix-Sums*
  - Input
    - Array of *n* elements: $[a_0, a_1, \ldots, a_{n-1}]$
    - Binary associate operator: $\oplus$
    - Identity: *I*
  - Outputs the array: $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$

Images from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

All-Prefix-Sums

- Example
  - If ⊕ is addition, the array
    - `[3 1 7 0 4 1 6 3]`
  - is transformed to
    - `[0 3 4 11 11 15 16 22]`
- Seems sequential, but there is an efficient parallel solution

Scan

- *Scan*: all-prefix-sums operation on an array of data
- *Exclusive Scan*: Element *j* of the result does not include element *j* of the input:
  - `In:  [3 1 7 0 4 1 6 3]`
  - `Out: [0 3 4 11 11 15 16 22]`
- *Inclusive Scan* (*Prescan*): All elements including *j* are summed
  - `In:  [3 1 7 0 4 1 6 3]`
  - `Out: [3 4 11 11 15 16 22 25]`

Scan

- How do you generate an *exclusive scan* from an *inclusive scan*?
  - `Input:     [3 1 7 0 4 1 6 3]`
  - `Inclusive: [3 4 11 11 15 16 22 25]`
  - `Exclusive: [0 3 4 11 11 15 16 22]`
    - `// Shift right, insert identity`
- How do you go in the opposite direction?

Scan

- Use cases
  - *Stream compaction*
  - *Summed-area tables* for variable width image processing
  - *Radix sort*
  - …

Scan

- Used to convert certain sequential computation into equivalent parallel computation

Sequential	Parallel

`out[0] = 0;`
`for j from 1 to n do`
`  out[j] = out[j-1] + f(in[j-1]);`
	`forall j in parallel do`
	`  temp[j] = f(in[j]);`
	`all_prefix_sums(out, temp);`

Image from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Scan

- Design a parallel algorithm for exclusive scan
  - In: [3 1 7 0 4 1 6 3]
  - Out: [0 3 4 11 11 15 16 22]
- Consider:
  - Total number of additions

Scan

- *Sequential Scan*: single thread, trivial

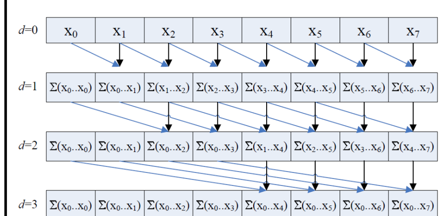
```
out[0] := 0
for k := 1 to n do
  out[k] := in[k-1] + out[k-1]
```

- *n* adds for an array of length *n*
- *Work complexity*: O(n)
- How many adds will our parallel version have?

Image from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Scan

- *Naive Parallel Scan*




```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

- Is this exclusive or inclusive?
- Each thread
  - Writes one sum
  - Reads two values

Image from http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf

Scan

- *Naive Parallel Scan*: Input

0	1	2	3	4	5	6	7

Scan

- *Naive Parallel Scan*: d = 1, 2^{d-1} = 1

0	1	2	3	4	5	6	7
0							

```
for d = 1 to log_2 n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

- *Naive Parallel Scan*: d = 1, 2^{d-1} = 1

0	1	2	3	4	5	6	7
0	1						

```
for d = 1 to log_2 n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

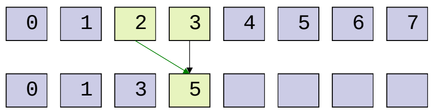
- *Naive Parallel Scan*: d = 1, 2^{d-1} = 1

0	1	2	3	4	5	6	7
0	1	3					

```
for d = 1 to log_2 n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

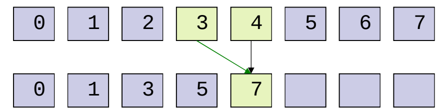
- *Naive Parallel Scan*: `d = 1, 2^(d-1) = 1`




```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

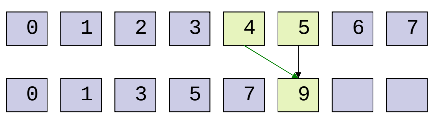
- *Naive Parallel Scan*: `d = 1, 2^(d-1) = 1`




```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

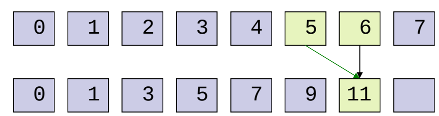
- *Naive Parallel Scan*: `d = 1, 2^(d-1) = 1`




```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

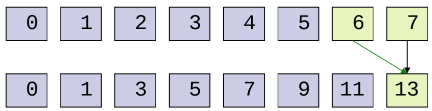
- *Naive Parallel Scan*: `d = 1, 2^(d-1) = 1`




```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

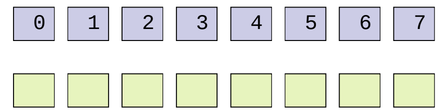
- *Naive Parallel Scan*: d = 1, 2^{d-1} = 1




```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

- *Naive Parallel Scan*: d = 1, 2^{d-1} = 1

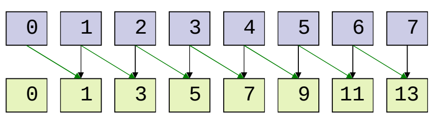


- Recall, it runs in parallel!

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

- *Naive Parallel Scan*: d = 1, 2^{d-1} = 1

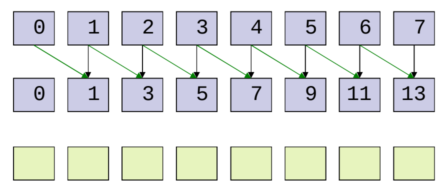


- Recall, it runs in parallel!

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

- *Naive Parallel Scan*: d = 2, 2^{d-1} = 2

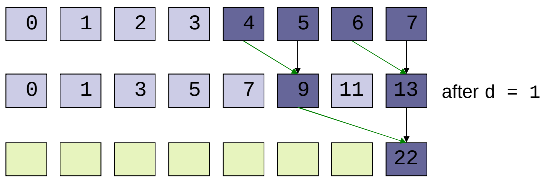


after d = 1

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

- *Naive Parallel Scan*: d = 2, $2^{d-1} = 2$

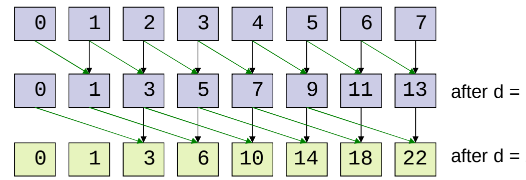


- Consider only k = 7

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

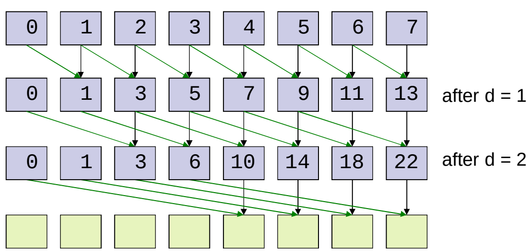
- *Naive Parallel Scan*: d = 2, $2^{d-1} = 2$




```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

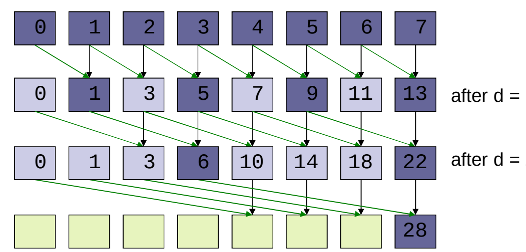
- *Naive Parallel Scan*: d = 3, $2^{d-1} = 4$




```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

- *Naive Parallel Scan*: d = 3, $2^{d-1} = 4$

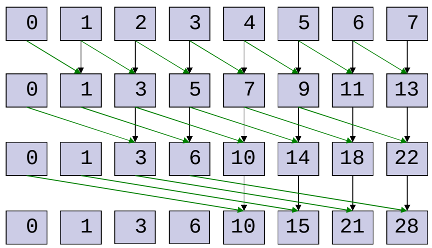


- Consider only k = 7

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

Scan

- *Naive Parallel Scan*: Final

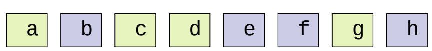


Scan

- *Naive Parallel Scan*
  - What is naive about this algorithm?
    - What was the work complexity for sequential scan?
    - What is the work complexity for this?

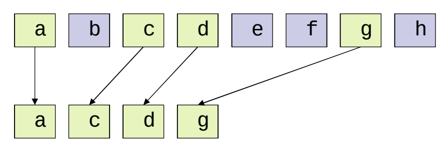
Stream Compaction

- *Stream Compaction*
  - Given an array of elements
    - Create a new array with elements that meet a certain criteria, e.g. non null
    - Preserve order



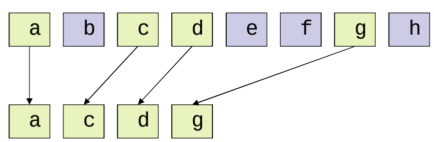
Stream Compaction

- Stream Compaction
  - Given an array of elements
    - Create a new array with elements that meet a certain criteria, e.g. non null
    - Preserve order



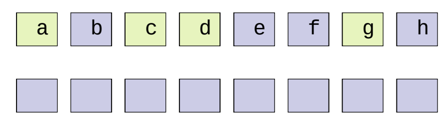
Stream Compaction

- Stream Compaction
  - Used in collision detection, sparse matrix compression, etc.
  - Can reduce bandwidth from GPU to CPU



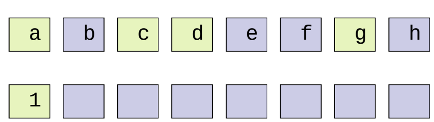
Stream Compaction

- Stream Compaction
  - *Step 1*: Compute temporary array containing
    - 1 if corresponding element meets criteria
    - 0 if element does not meet criteria



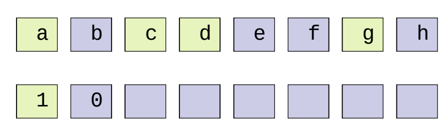
Stream Compaction

- Stream Compaction
  - *Step 1*: Compute temporary array



Stream Compaction

- Stream Compaction
  - *Step 1*: Compute temporary array



Stream Compaction

- Stream Compaction
  - *Step 1*: Compute temporary array

a	b	c	d	e	f	g	h
1	0	1					

Stream Compaction

- Stream Compaction
  - *Step 1*: Compute temporary array

a	b	c	d	e	f	g	h
1	0	1	1				

Stream Compaction

- Stream Compaction
  - *Step 1*: Compute temporary array

a	b	c	d	e	f	g	h
1	0	1	1	0			

Stream Compaction

- Stream Compaction
  - *Step 1*: Compute temporary array

a	b	c	d	e	f	g	h
1	0	1	1	0	0		

Stream Compaction

- Stream Compaction
  - *Step 1*: Compute temporary array

a	b	c	d	e	f	g	h
1	0	1	1	0	0	1	

Stream Compaction

- Stream Compaction
  - *Step 1*: Compute temporary array

a	b	c	d	e	f	g	h
1	0	1	1	0	0	1	0

Stream Compaction

- Stream Compaction
  - *Step 1*: Compute temporary array

a	b	c	d	e	f	g	h

- It runs in parallel!

Stream Compaction

- Stream Compaction
  - *Step 1*: Compute temporary array

a	b	c	d	e	f	g	h
1	0	1	1	0	0	1	0

- It runs in parallel!

Stream Compaction

- Stream Compaction
  - *Step 2*: Run exclusive scan on temporary array

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan result:								

Stream Compaction

- Stream Compaction
  - *Step 2*: Run exclusive scan on temporary array

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan result:	0	1	1	2	3	3	3	4

  - Scan runs in parallel
  - What can we do with the results?

Stream Compaction

- Stream Compaction
  - *Step 3*: Scatter
    - Result of scan is index into final array
    - Only write an element if temporary array has a 1

Stream Compaction

- Stream Compaction
  - *Step 3*: Scatter

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan result:	0	1	1	2	3	3	3	4

Final array:				
	0	1	2	3

Stream Compaction

- Stream Compaction
  - *Step 3*: Scatter

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan result:	0	1	1	2	3	3	3	4

Final array:	a			
	0	1	2	3

Stream Compaction

- Stream Compaction
  - *Step 3*: Scatter

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan result:	0	1	1	2	3	3	3	4

Final array:	a	c		
	0	1	2	3

Stream Compaction

- Stream Compaction
  - *Step 3*: Scatter

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan result:	0	1	1	2	3	3	3	4

Final array:	a	c	d	
	0	1	2	3

Stream Compaction

- Stream Compaction
  - *Step 3*: Scatter

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan result:	0	1	1	2	3	3	3	4

Final array:	a	c	d	g
	0	1	2	3

Stream Compaction

- Stream Compaction
  - *Step 3*: Scatter

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan result:	0	1	1	2	3	3	3	4

Final array:				
	0	1	2	3

- Scatter runs in parallel!

Stream Compaction

- Stream Compaction
  - *Step 3*: Scatter

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan result:	0	1	1	2	3	3	3	4

Final array:	a	c	d	g
	0	1	2	3

- Scatter runs in parallel!

Summed Area Table

- *S*ummed *A*rea *T*able (*SAT*): 2D table where each element stores the sum of all elements in an input image between the lower left corner and the entry location.

Summed Area Table

- Example:

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

4	9	12	14
2	6	9	11
2	5	6	8
1	2	2	4

$(1 + 1 + 0) + (1 + 2 + 1) + (0 + 1 + 2) = 9$

Summed Area Table

- Benefit
  - Used to perform different width filters at every pixel in the image in constant time per pixel
  - Just sample four pixels in SAT:

$$s_{filter} = \frac{s_{ur} - s_{ul} - s_{lr} + s_{ll}}{w \times h},$$

Image from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Summed Area Table

- Uses
  - Glossy environment reflections and refractions
  - Approximate depth of field

Image from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

1			

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

1	2		

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

1	2	2	

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

2			
1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

2	5		
1	2	2	4

Summed Area Table

. . .

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

4	9		
2	6	9	11
2	5	6	8
1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

4	9	12	
2	6	9	11
2	5	6	8
1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

4	9	12	14
2	6	9	11
2	5	6	8
1	2	2	4

Summed Area Table

How would implement this on the GPU?

Summed Area Table

How would compute a SAT on the GPU using inclusive scan?

Summed Area Table

- Step 1 of 2:

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

Partial SAT

2	3	3	3
0	1	3	3
1	3	4	4
1	2	2	4

One inclusive scan for each row

Summed Area Table

- Step 2 of 2:

Partial SAT

2	3	3	3
0	1	3	3
1	3	4	4
1	2	2	4

Final SAT

4	9	12	14
2	6	9	11
2	5	6	8
1	2	2	4

One inclusive scan for each column, bottom to top

Summary

- Parallel reductions and scan are building blocks for many algorithms
- An understanding of parallel programming and GPU architecture yields efficient GPU implementations