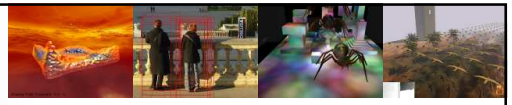


Parallel Algorithms Continued

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2012

Announcements

- Presentation topics due tomorrow, 02/07
- Homework 2 due next Monday, 02/13

Agenda

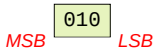
- Parallel Algorithms
  - Review Stream Compression
  - Radix Sort
- CUDA Performance

Radix Sort

- Efficient for small sort keys
  - k-bit keys require k passes

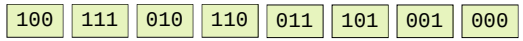
Radix Sort

- Each radix sort pass partitions its input based on one bit
- First pass starts with the *l*east *s*ignificant *b*it (*LSB*). Subsequent passes move towards the *m*ost *s*ignificant *b*it (*MSB*)



Radix Sort

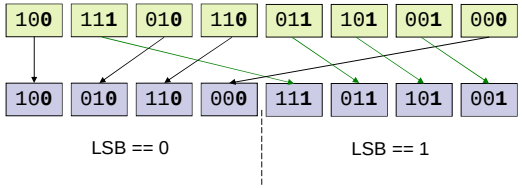
- Example input:



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

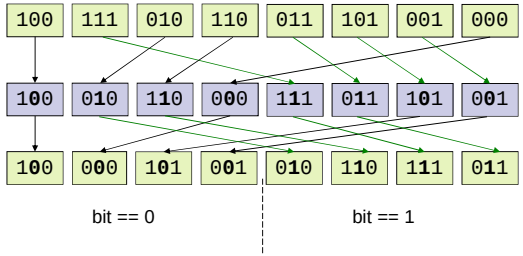
Radix Sort

- First pass: partition based on LSB



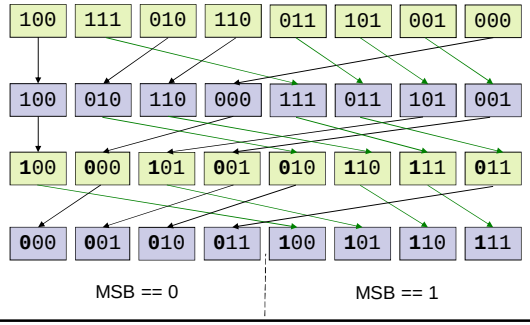
Radix Sort

- Second pass: partition based on middle bit



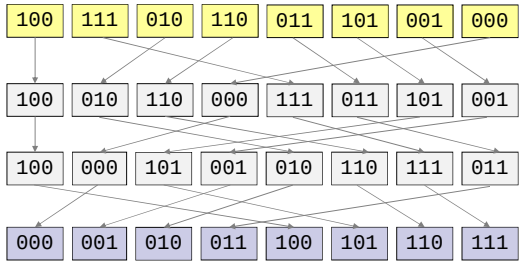
Radix Sort

- Final pass: partition based on MSB



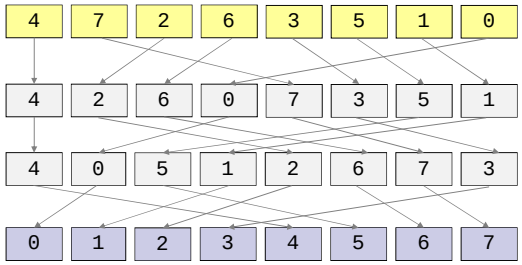
Radix Sort

- Completed:



Radix Sort

- Completed:



Parallel Radix Sort

- Where is the parallelism?

Parallel Radix Sort

1. Break input arrays into tiles
   - Each tile fits into shared memory for an SM
2. Sort tiles in *parallel* with *radix sort*
3. Merge pairs of tiles using a *parallel bitonic merge* until all tiles are merged.

Our focus is on Step 2

Parallel Radix Sort

- Where is the parallelism?
  - Each tile is sorted in parallel
  - Where is the parallelism within a tile?

Parallel Radix Sort

- Where is the parallelism?
  - Each tile is sorted in parallel
  - Where is the parallelism within a tile?
    - Each pass is done in sequence after the previous pass. No parallelism
    - Can we parallelize an individual pass? How?
  - Merge also has parallelism

Parallel Radix Sort

- Implement *spilt*. Given:
  - Array, *i*, at pass *b*:

100	111	010	110	011	101	001	000

  - Array, *b*, which is true/false for bit *b*:

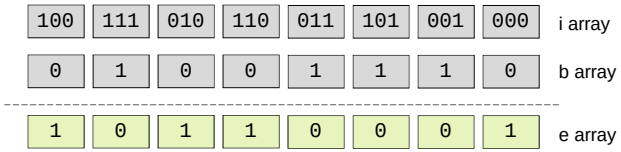
0	1	0	0	1	1	1	0

- Output array with false keys before true keys:

100	010	110	000	111	011	101	001

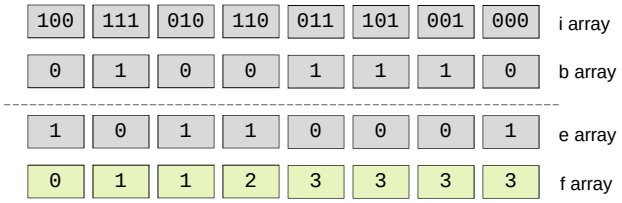
Parallel Radix Sort

- Step 1: Compute *e* array



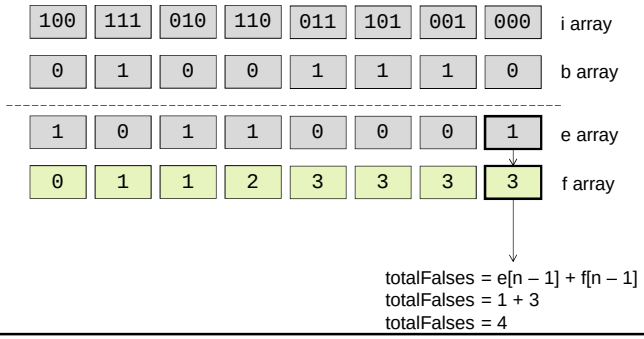
Parallel Radix Sort

- Step 2: Exclusive Scan *e*



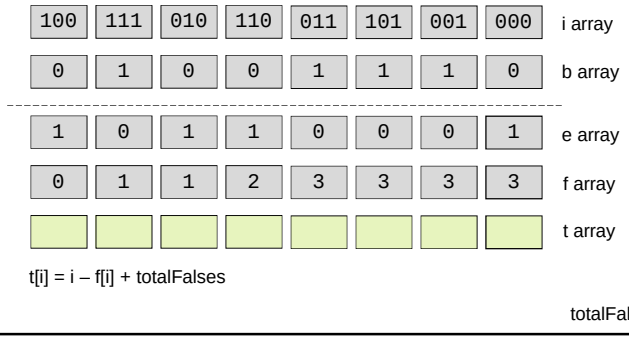
Parallel Radix Sort

- Step 3: Compute *totalFalses*



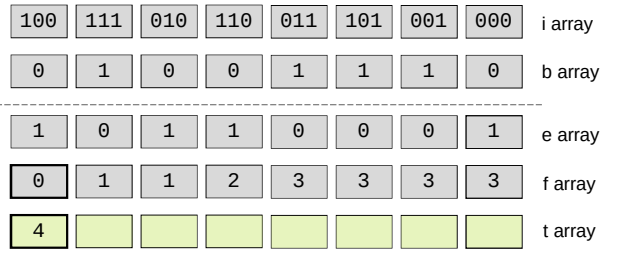
Parallel Radix Sort

- Step 4: Compute *t* array



Parallel Radix Sort

- Step 4: Compute *t* array

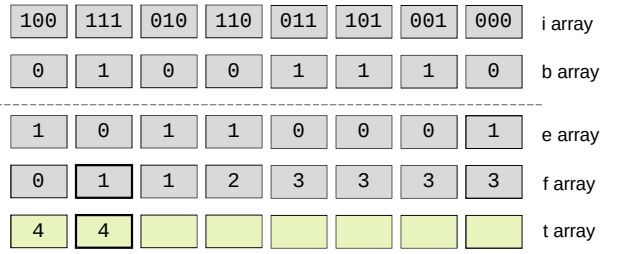


t[0] = 0 – f[0] + totalFalses
t[0] = 0 – 0 + 4
t[0] = 4

totalFalses = 4

Parallel Radix Sort

- Step 4: Compute *t* array

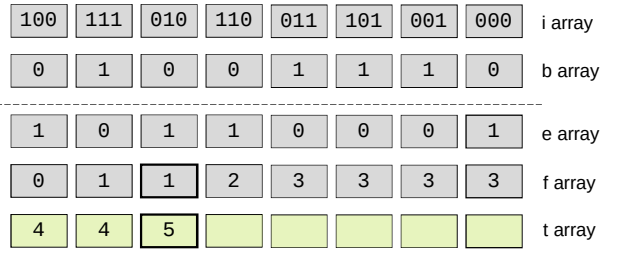


t[1] = 1 – f[1] + totalFalses
t[1] = 1 – 1 + 4
t[1] = 4

totalFalses = 4

Parallel Radix Sort

- Step 4: Compute *t* array

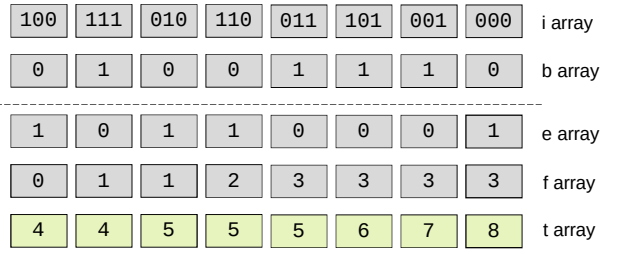


t[2] = 2 – f[2] + totalFalses
t[2] = 2 – 1 + 4
t[2] = 5

totalFalses = 4

Parallel Radix Sort

- Step 4: Compute *t* array

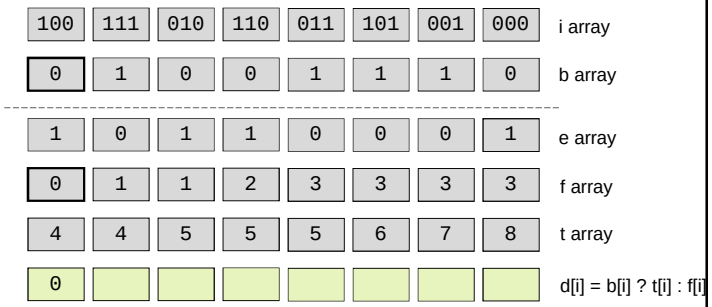


t[i] = i – f[i] + totalFalses

totalFalses = 4

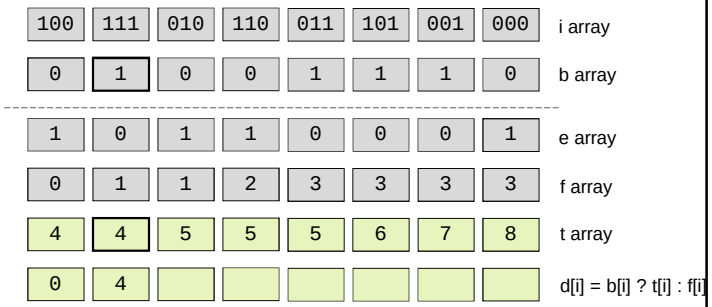
Parallel Radix Sort

- Step 5: Scatter based on address *d*



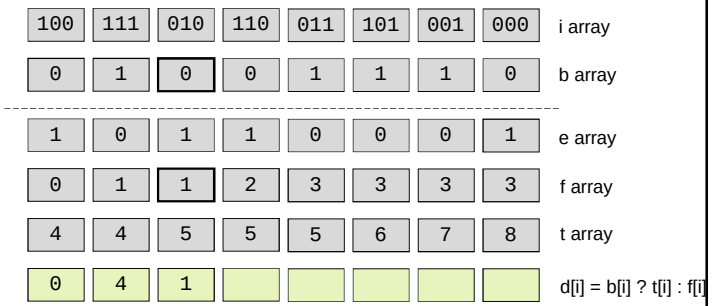
Parallel Radix Sort

- Step 5: Scatter based on address *d*



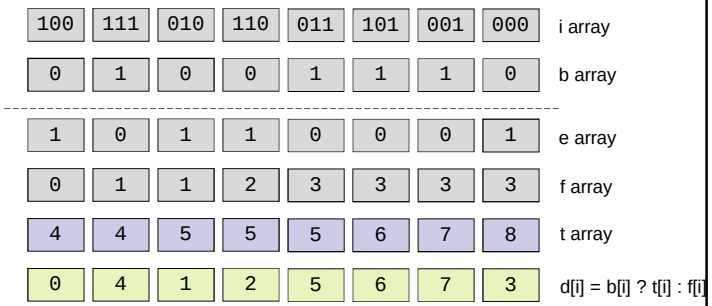
Parallel Radix Sort

- Step 5: Scatter based on address *d*



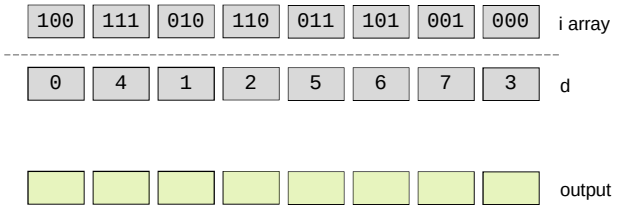
Parallel Radix Sort

- Step 5: Scatter based on address *d*



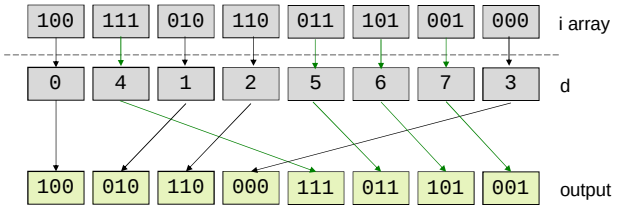
Parallel Radix Sort

- Step 5: Scatter based on address *d*



Parallel Radix Sort

- Step 5: Scatter based on address *d*



Parallel Radix Sort

- Given k-bit keys, how do we sort using our new *split* function?
- Once each tile is sorted, how do we merge tiles to provide the final sorted array?