CUDA 4
Prefix Sums

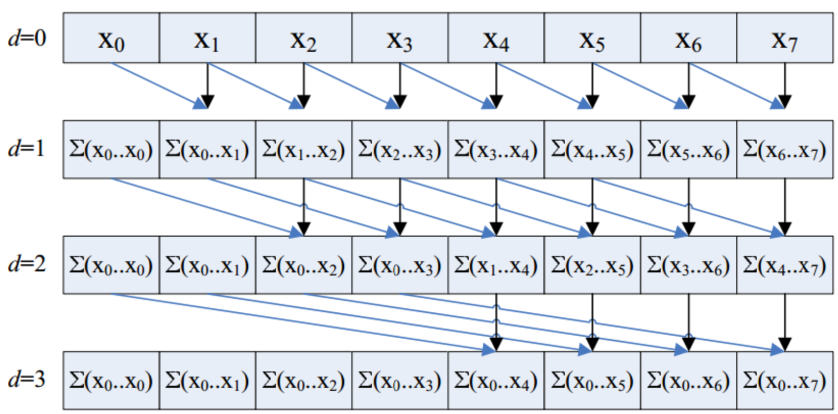
CS121 Parallel Computing

Spring 2019

Prefix sum

- Given an array $[x_0, x_1, \ldots, x_{n-1}]$, output sums of prefixes of the array, $[x_0, x_0+x_1, \ldots, x_0+\ldots+x_{n-1}]$.
- Also called inclusive "scan".
- Has a large number of applications in parallel algorithms.
  - Histograms, counting sort, radix sort, stream compaction, string comparison, tree algorithms, polynomial interpolation, recurrences, etc.
- Trivial sequential algorithm.
  - Does O(n) operations in O(n) time.
- Can replace sum with any associative operator.
  - $\oplus$ is associative if $a\oplus(b\oplus c) = (a\oplus b)\oplus c$.

Parallel prefix sum (naive)

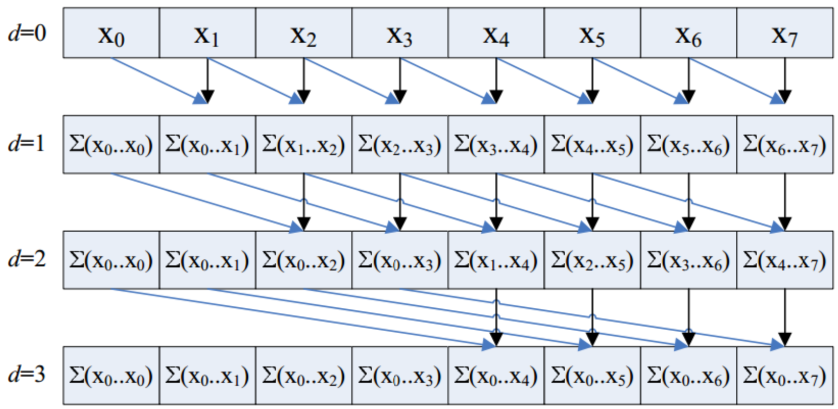



```
for (i = 1; i < log(n); i++)
   for all tid in parallel
      if (tid >= 2^i)
         sum[out][tid] = sum[in][tid-2^(i-1)]
         + x[in][tid]
      else
         sum[out][tid] = sum[in][tid]
      swap in, out
```

Parallel Prefix Sum (Scan) with CUDA, Mark Harris

- Map one thread to each element.
- $\log_2$ n iterations (assume n is power of 2).
  - Set stride to 1, 2, 4, ..., n.
  - Threads > stride add value from stride below to itself.
- Two output buffers sum[in], sum[out]. Initially in=0, out =1. Swap after each iteration.
  - Single buffer would have race condition (how?).

Work analysis



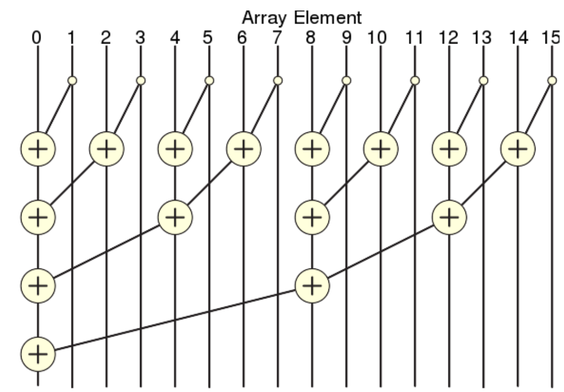

```
for (i = 1; i < log(n); i++)
  for all tid in parallel
    if (tid >= 2^i)
      sum[out][tid] = sum[in][tid-2^(i-1)]
      + x[in][tid]
    else
      sum[out][tid] = sum[in][tid]
    swap in, out
```

Parallel Prefix Sum (Scan) with CUDA, Mark Harris

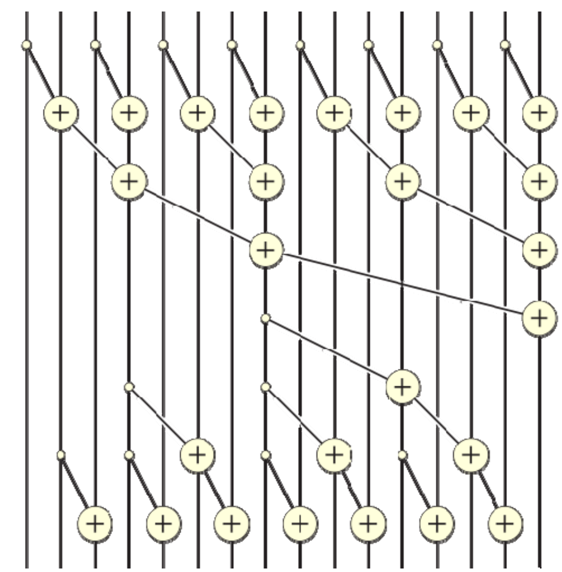
- Number of operations in iteration i is n – stride(i).
- Total number of operations is (n-1) + (n–2) + (n–4) + ... + (n–n/2) = O(n log n).
- Sequential (and optimal) complexity is O(n).
- Extra O(log n) factor complexity really matters in practice.
  - 20 times slower for n = 1M!

Efficient parallel prefix sum

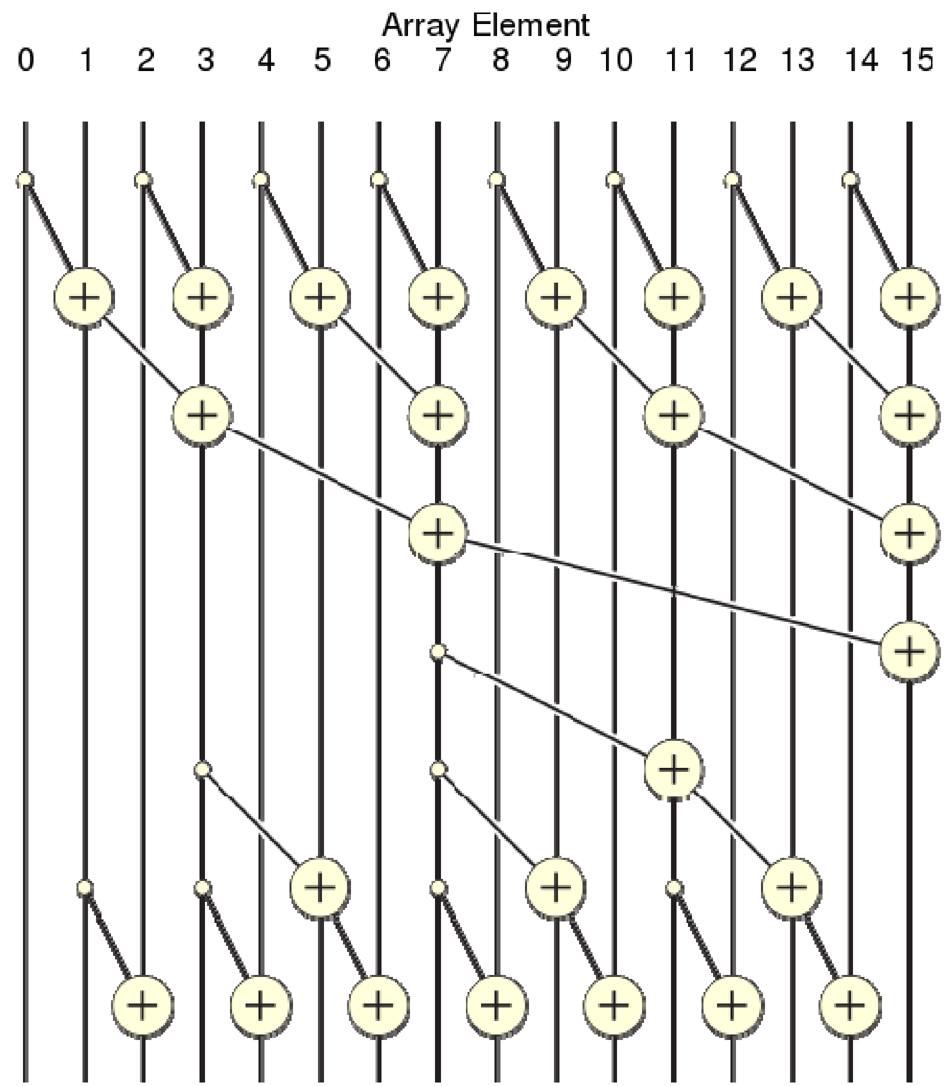
- Want algorithm to do O(n) work.
- Recall the parallel reduction algorithm, which does O(n) work.
- Efficient algorithm does a reduction, followed by the reduction "in reverse".
  - Call these the up-sweep and down-sweep phases, resp.



Prefix sum (Brent-Kung)



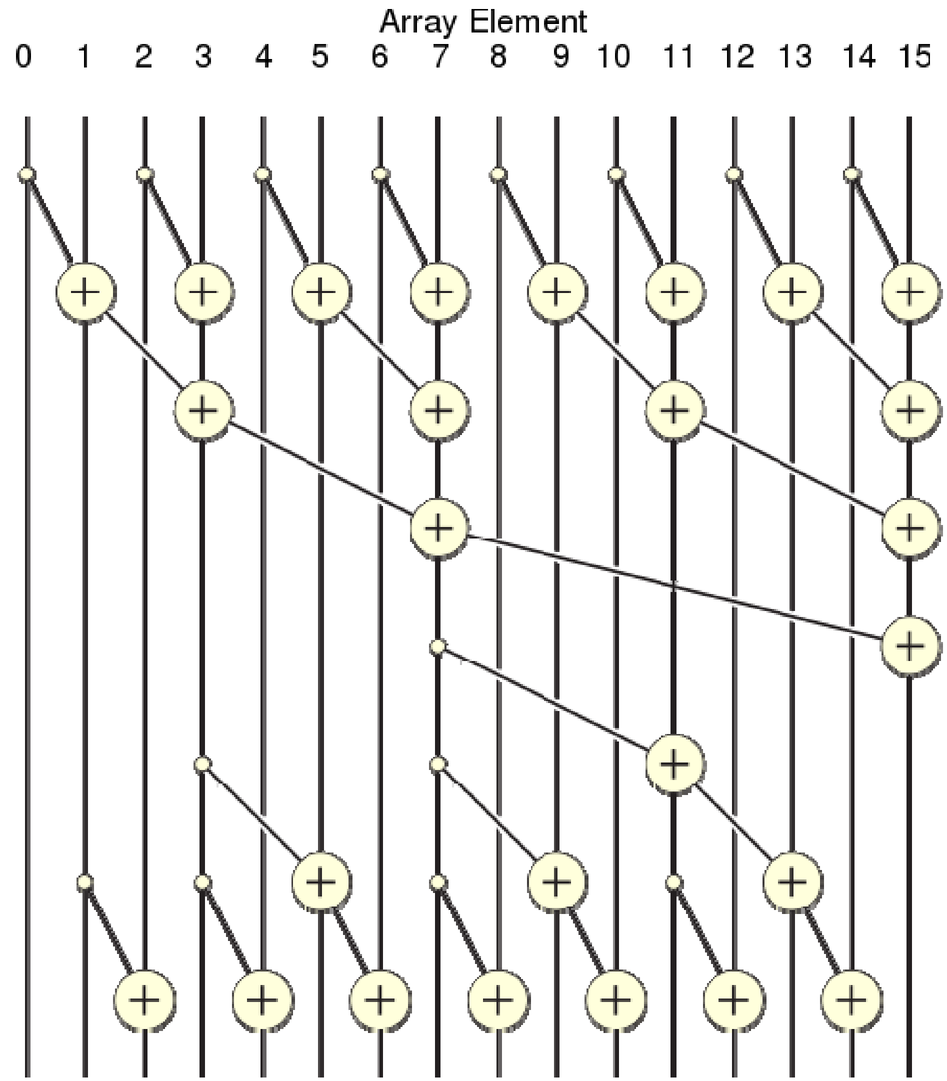
Efficient parallel prefix sum



Efficient parallel prefix sum

```
int stride = 1;
while (stride <= blockDim.x) {
   int i = 2*stride*(threadIdx.x+1)-1;
   if (i < 2*blockDim.x)
      sum[i] += sum[i-stride];
   stride *= 2;
   __syncthreads();
}

int stride = blockDim.x/2;
while (stride > 0) {
   int i = 2*stride*(threadIdx.x+1)-1;
   if (i+stride < 2*dimBlock.x)
      sum[i+stride] += sum[i];
   stride /= 2;
   __syncthreads();
}
```

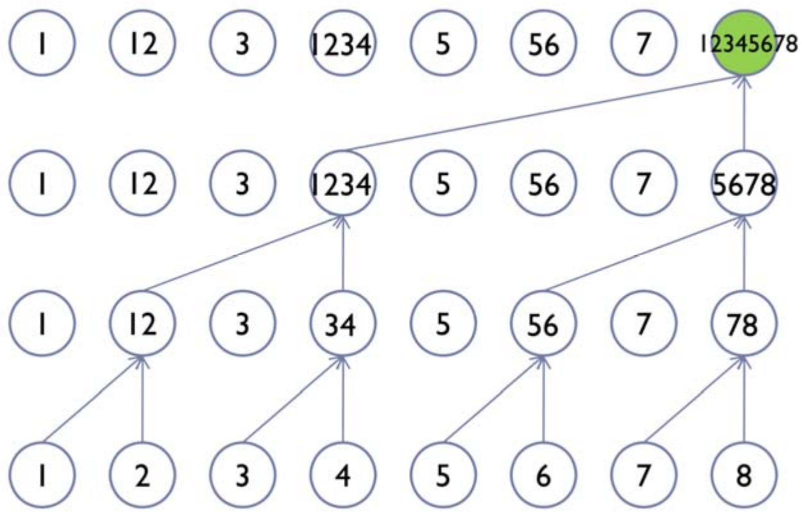


- A thread block computes prefix sum of array `sum` in shared memory.
  - Size of `sum` is 2*(block size).
  - In example, block size = 8.
- In down sweep, threads 0 to (block size) / stride – 1 work in iteration `stride`.
- In up sweep, threads 0 to (block size) / (2*stride) – 1 work in iteration `stride` .

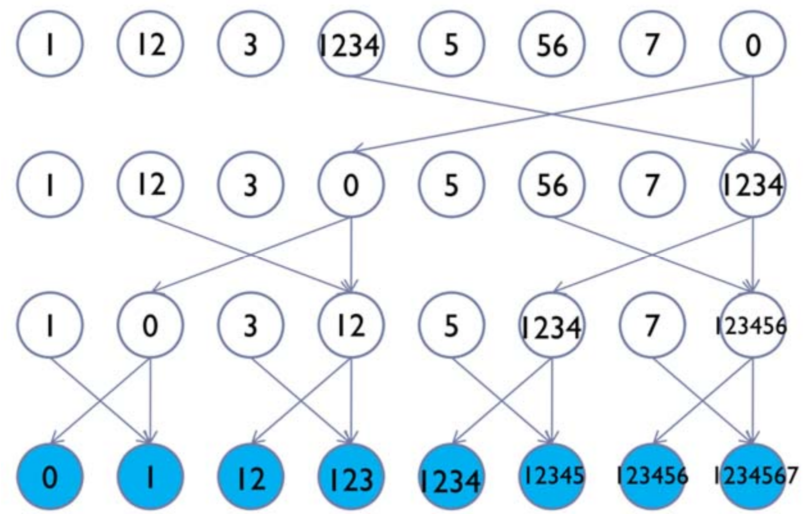
Exclusive scans

- Just like a normal scan, except each input value shouldn’t include itself in its output.
  - Ex [1,2,3,4] $\Rightarrow$ [0,1,3,6].
- Up-sweep is the same as in inclusive scan.
- But during down-sweep, first zero out the final output value.
- Then follow a half butterfly pattern downwards.
  - Each right child sums its parents’ values.
  - Each left child takes its parent’s value.

Exclusive scans



Up-sweep



Down-sweep

Up-sweep (reduce):

for $d = 0$ to $\log_2 n - 1$ **do**
  for all $k = 0$ to $n-1$ by 2^{d+1} in parallel **do**
    $x[k+2^{d+1}-1] \leftarrow x[k+2^d-1] + x[k+2^{d+1}-1]$

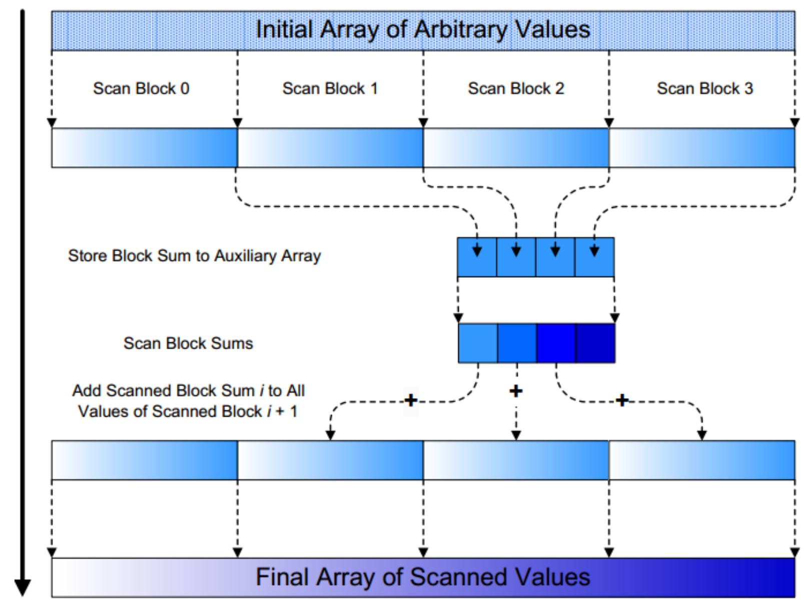
Down-sweep:

$x[n-1] \leftarrow 0$
for $d = \log_2 n - 1$ down to 0 **do**
  for all $k = 0$ to $n-1$ by 2^{d+1} in parallel **do**
    $t \leftarrow x[k+2^d-1]$
    $x[k+2^d-1] \leftarrow x[k+2^{d+1}-1]$
    $x[k+2^{d+1}-1] \leftarrow t + x[k+2^{d+1}-1]$

Source: http://courses.me.berkeley.edu/ME290R/S2009/lectures/lec15.PDF

Arbitrary input size

- The inclusive scan algorithm only works for array size ≤ 2*(block size).
- For bigger inputs, break it into segments of size 2*(block size).
- Compute prefix sum on each segment using block algorithm.
- Copy sum of whole segment (stored in `sum[blockDim.x-1]`) to `segment_sum` array.
- Do this for all blocks until they all finish.
  - Ensure blocks finished by ending kernel.
- Compute prefix sum of `segment_sum` array in a second kernel.
- In a third kernel, distribute prefix sums to each segment.
  - Segment increases all values by prefix sum received.



Bank conflicts

- Recall memory address x stored at x % n if shared memory has n banks.
  - Current GPUs have 32 banks.
- Current algorithm has many bank conflicts, causing serialized accesses.

bank 0	0	4	8	12	16
bank 1	1	5	9	13	17
bank 2	2	6	10	14	18
bank 3	3	7	11	15	19

16 banks, stride = 1. 2 way bank conflicts

tid	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
bank	1	3	5	7	9	11	13	15	1	3	5	7	9	11	13	15

16 banks, stride = 2. 4 way bank conflicts

tid	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i	3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63
bank	3	7	11	15	3	7	11	15	3	7	11	15	3	7	11	15

```
...
int i = 2*stride*
   (threadIdx.x+1)-1;
if (i < 2*blockDim.x)
   sum[i] += sum[i-
   stride];
...
```

Removing bank conflicts

- Remove bank conflicts by padding the sum array.
- Store i'th item at address i + floor(i / (# banks)) instead of address i.
  - Do this for reads and writes.
  - Waste some space (~3% with 32 banks), but get faster performance.
- Ex 4 banks.

array	0	1	2	3	4	5	6	7	8	9	10	11		
padded array	0	1	2	3	P	4	5	6	7	P	8	9	10	11

- Padding is a general strategy for removing bank conflicts, though exact scheme depends on problem.

Removing bank conflicts

16 banks, stride = 2. 4 way bank conflicts

tid	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i	3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63
bank	3	7	11	15	3	7	11	15	3	7	11	15	3	7	11	15

16 banks, stride = 2, i' = i + floor(i / # banks). No bank conflicts

tid	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i'	3	7	11	15	20	24	28	32	37	41	45	49	54	58	62	66
bank	3	7	11	15	4	8	12	0	5	9	13	1	6	10	14	2

Segmented scan

- Sometimes need to scan several segments at once.
  - Many applications, e.g. sparse matrix vector multiplication, processor allocation, etc.
  - We consider exclusive segmented scan.
- Ex [1 2 3 4] [6 5] [1 3 5] $\Rightarrow$ [0 1 3 6] [0 6] [0 1 4].
- If there are m segments and we do m scans, each of size n, then total time O(n log m).
- Segmented scan does all the scans in $O(\log mn)$ time.
- Use flags array to mark the start of segments.
  - Ex Array for example above is [1 0 0 0 1 0 1 0 0].
- Define new array of pairs, $c_i = [f_i, x_i]$.
- Define new associative operator $\odot$ on c_i

$$c_1 \odot c_2 = [f_1, x_1] \odot [f_2, x_2] = \begin{cases} [f_1 \mid f_2, x_1 + x_2], & f = 0 \\ [f_1 \mid f_2, x_2], & f = 1 \end{cases}$$

- Do a scan as before over array c_i with operator $\odot$.

Segmented scan

- `flag_original` is the array of flags marking the segments.
- `flags` is a new array defined based on `flags_original`.
- Follows the basic up / down sweep structure of normal scan.

Work-efficient segmented scan

Up-sweep:

```
for d=0 to (log₂n - 1) do
   forall k=0 to n-1 by 2^(d+1) do
     if flag[k + 2^(d+1) - 1] == 0:
        data[k + 2^(d+1) - 1] ← data[k + 2^d - 1] + data[k + 2^(d+1) - 1]
     flag[k + 2^(d+1) - 1] ← flag[k + 2^d - 1] || flag[k + 2^(d+1) - 1]
```

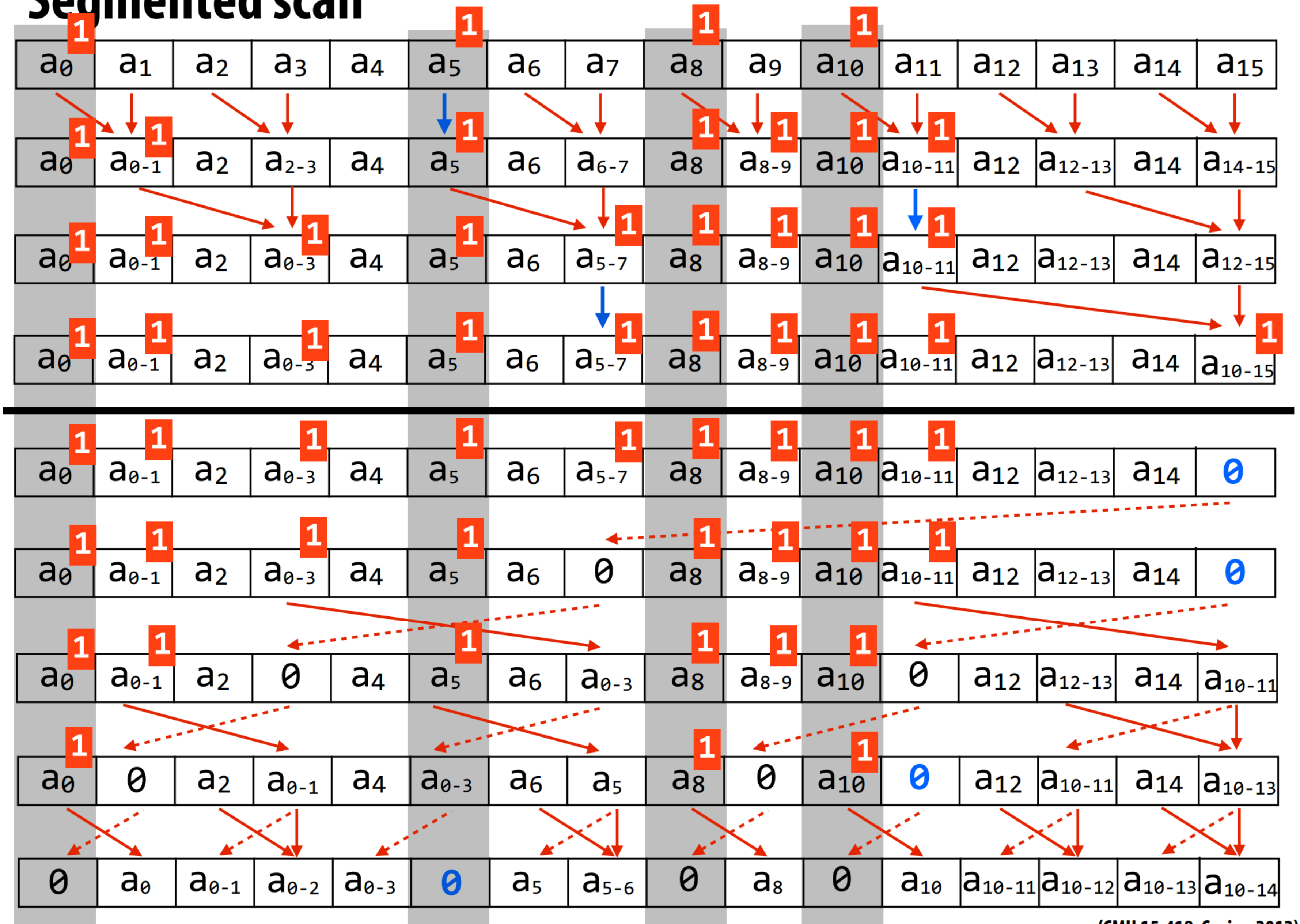
Down-sweep:

```
data[n-1] ← 0
for d=(log₂n - 1) down to 0 do
   forall k=0 to n-1 by 2^(d+1) do
     tmp ← data[k + 2^d - 1]
     data[k + 2^d - 1] ← data[k + 2^(d+1) - 1]
     if flag_original[k + 2^d] == 1:          // maintain copy of original flags
        data[k + 2^(d+1) - 1] ← 0
     else if flag[k + 2^d - 1] == 1:
        data[k + 2^(d+1) - 1] ← tmp
     else:
        data[k + 2^(d+1) - 1] ← tmp + data[k + 2^(d+1) - 1]
     flag[k + 2^d - 1 ] ← 0
```

Source: http://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/

(CMU 15-418, Spring 2012)

Segmented scan



(CMU 15-418, Spring 2012)

Application: compaction

- Create array containing elements of input array satisfying a condition.
- Ex Move all odd numbers in A to front of *output*.
  - Create filter array that's 1 if element satisfies condition.
  - Prefix sum the filter array.
  - For each element, if it satisfies condition, move it to index given by prefix sum.

A = [1 3 2 4 8 6 5 4 9 7 3]

filter = [1 1 0 0 0 0 1 0 1 1 1]

sums = [1 2 2 2 2 2 3 3 4 5 6]

output = [1 3 5 9 7 3]

Application: string comparison

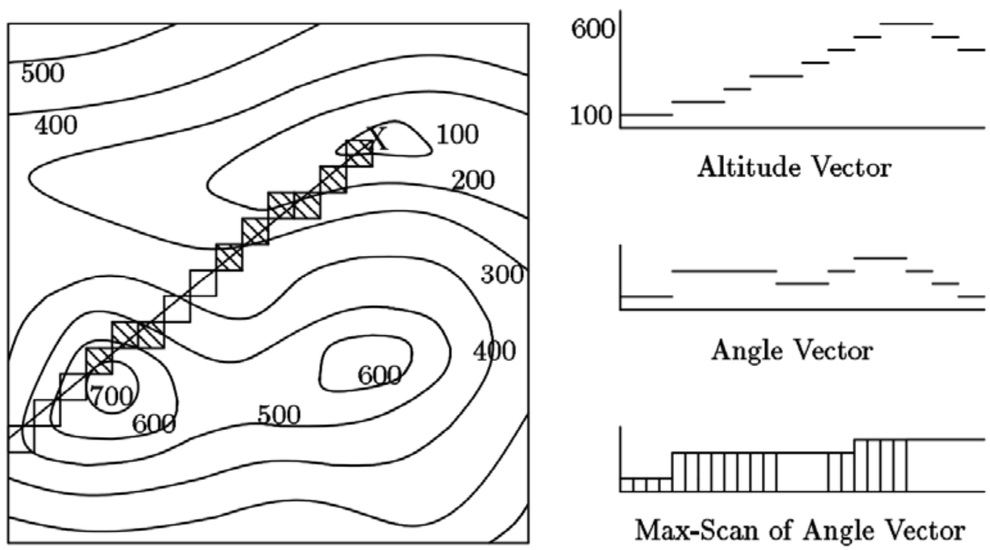
- Compare two strings alphabetically.
- Ex parallax < parallel.
- Let strings be S, T. Let S[i],T[i] denote i'th letter of S,T.

- In parallel, i'th processor compares S[i] to T[i].
  - If S[i]>T[i], set A[i]=1.
  - If S[i]=T[i], set A[i]=0.
  - If S[i]<T[i], set A[i]=-1.
  - If S[i] or T[i] doesn't exist, set A[i]=0.
- Compact A to remove all 0's.
- If output[1]=1, then S>T.
- If output[1]=-1, then T>S.
- If output is empty, then S=T.

- Ex S=parallax, T=parallel, A=[0,0,0,0,0,0,-1,1], output=[-1,1], so T>S.

Application: line of sight

```
procedure line-of-sight(altitude)
  in parallel for each index i
    angle[i] ← arctan(scale × (altitude[i] - altitude[0])/ i)
  max-previous-angle ← max-prescan(angle)
  in parallel for each index i
    if (angle[i] > max-previous-angle[i])
       result[i] ← "visible"
    else
       result[i] ← not "visible"
```



Altitude Map

Ray Vectors

- Given a contour map, an observation point X and a direction, want to know which points are visible.
- First, draw a line from X in the observing direction and record the altitudes along the line in an altitude vector.
- Then for each point calculate its angle, based on its altitude and distance from X.
- Then do a max-scan over the angle vectors.
- A point is visible iff its angle is larger than all the preceding angles.