EE382N (20): Computer Architecture - Parallelism and Locality
Spring 2015

Lecture 13 – Parallelism in Software I

Mattan Erez

The University of Texas at Austin

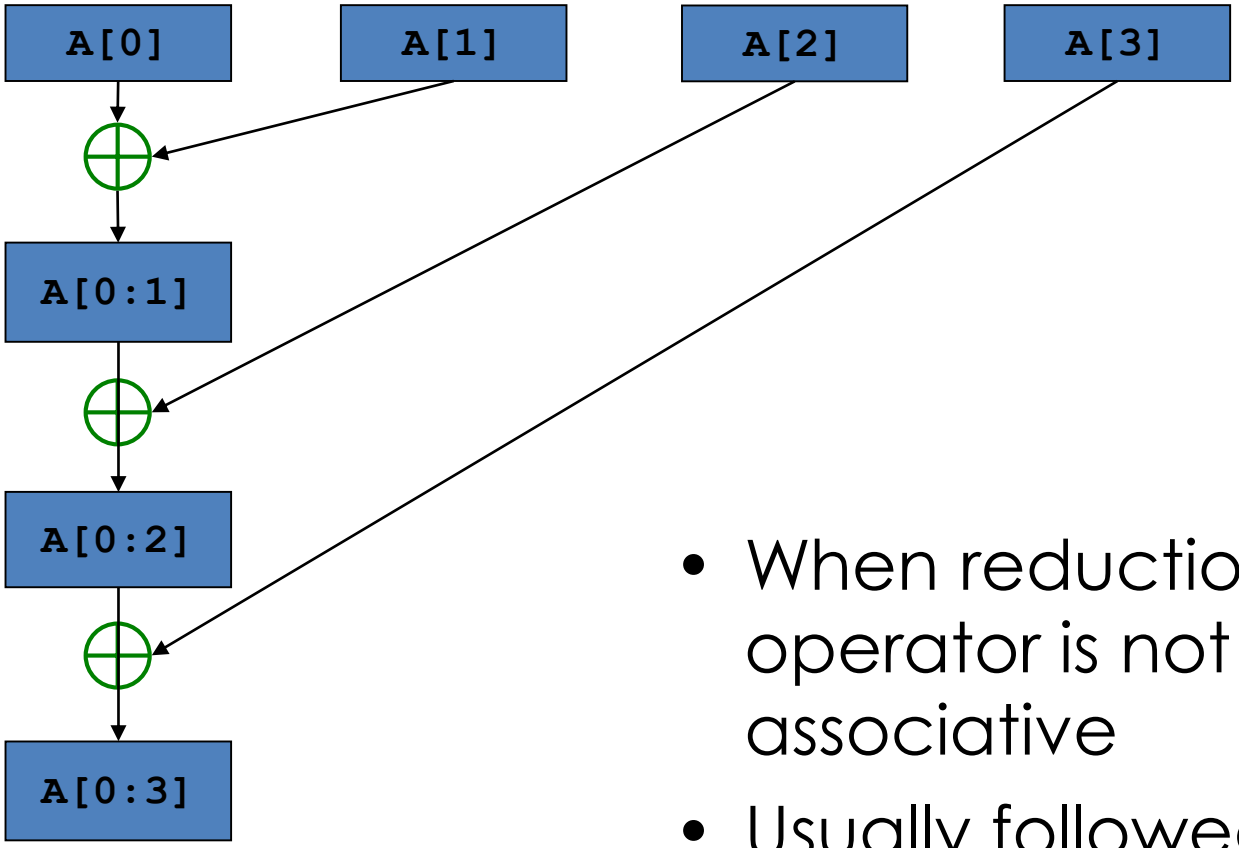
Credits

- Most of the slides courtesy Dr. Rodric Rabbah (IBM)
  - Taken from 6.189 IAP taught at MIT in 2007
- Parallel Scan slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  - Taken from EE493-AI taught at UIUC in Sprig 2007

Reductions

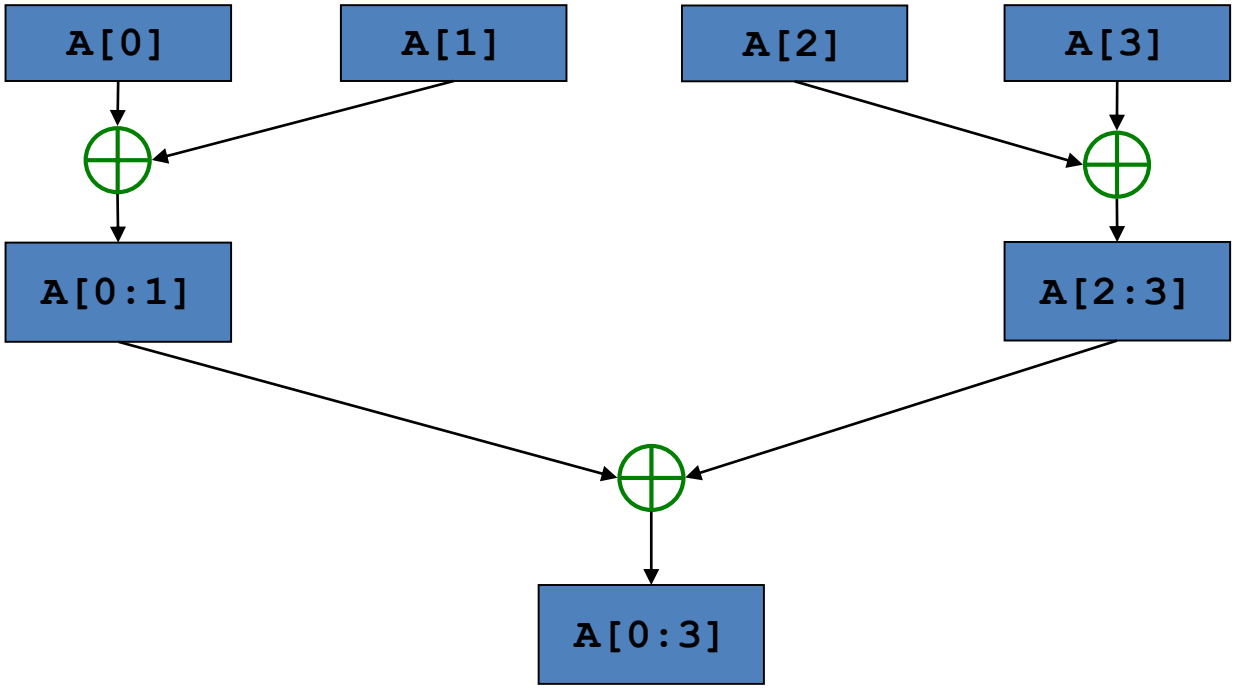
- Many to one
- Many to many
  - Simply multiple reductions
    - Also known as scatter-add and subset of parallel prefix sums
- Use
  - Histograms
  - Superposition
    - Physical properties

Serial Reduction



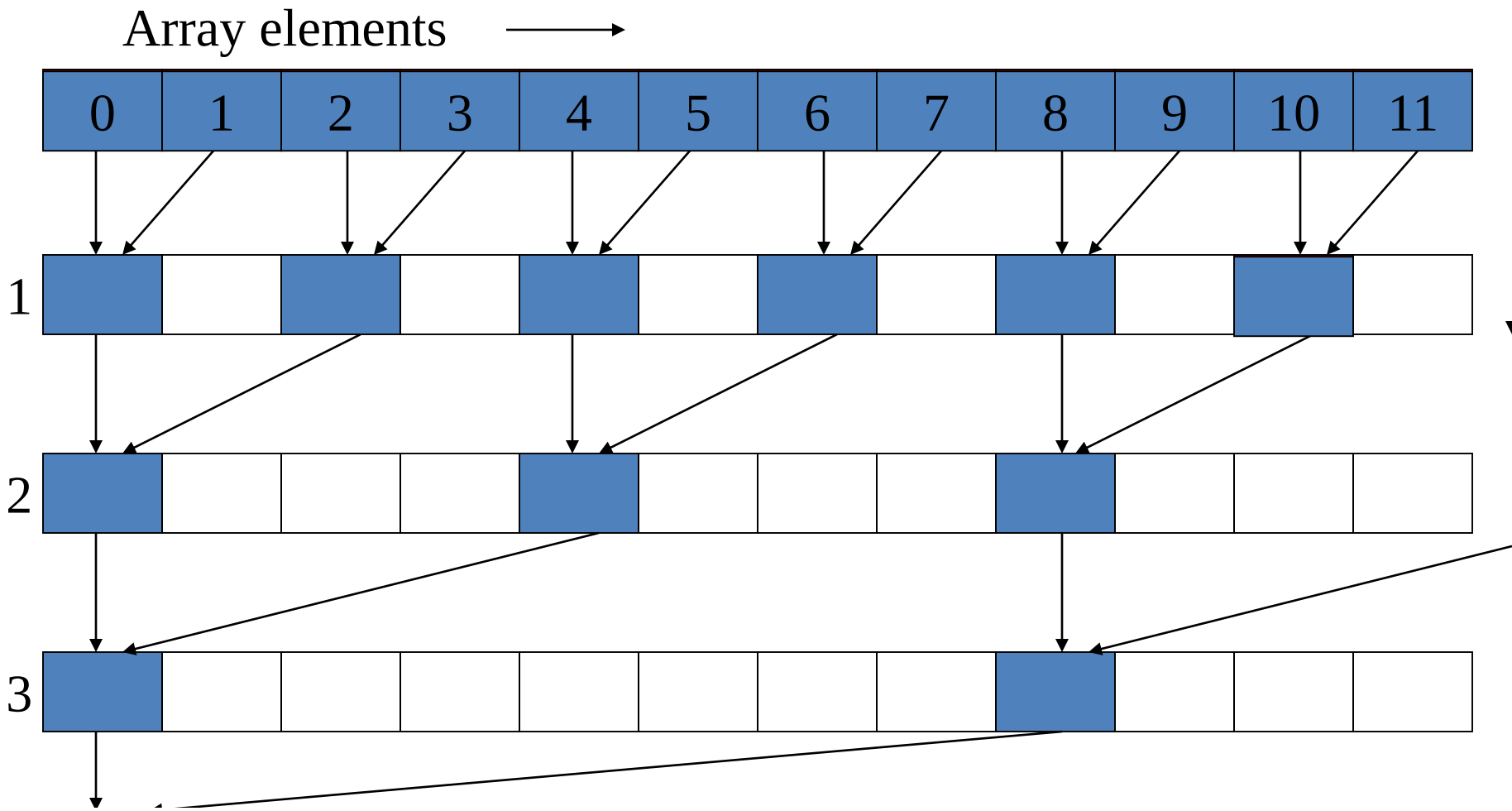
- When reduction operator is not associative
- Usually followed by a broadcast of result

Tree-based Reduction

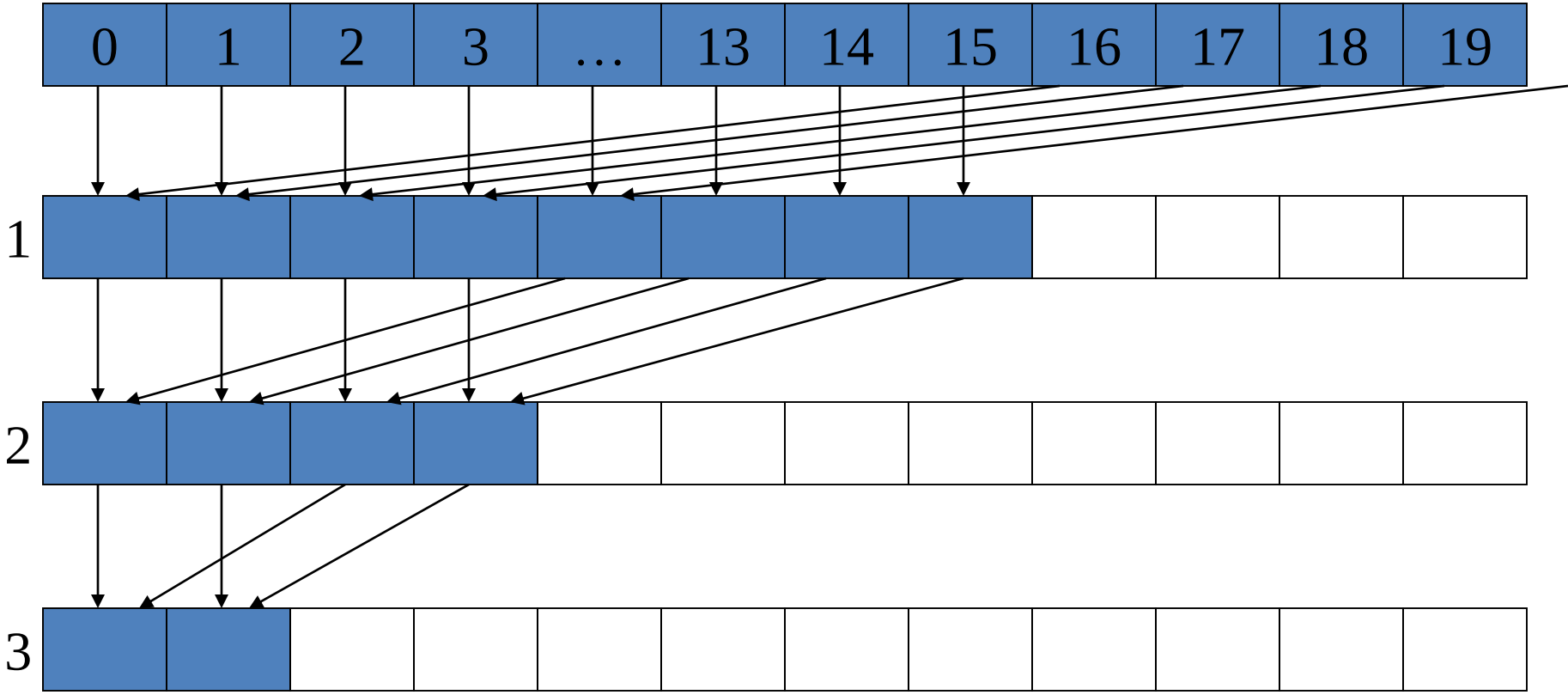


- n steps for 2^n units of execution
- When reduction operator is associative
- Especially attractive when only one task needs result

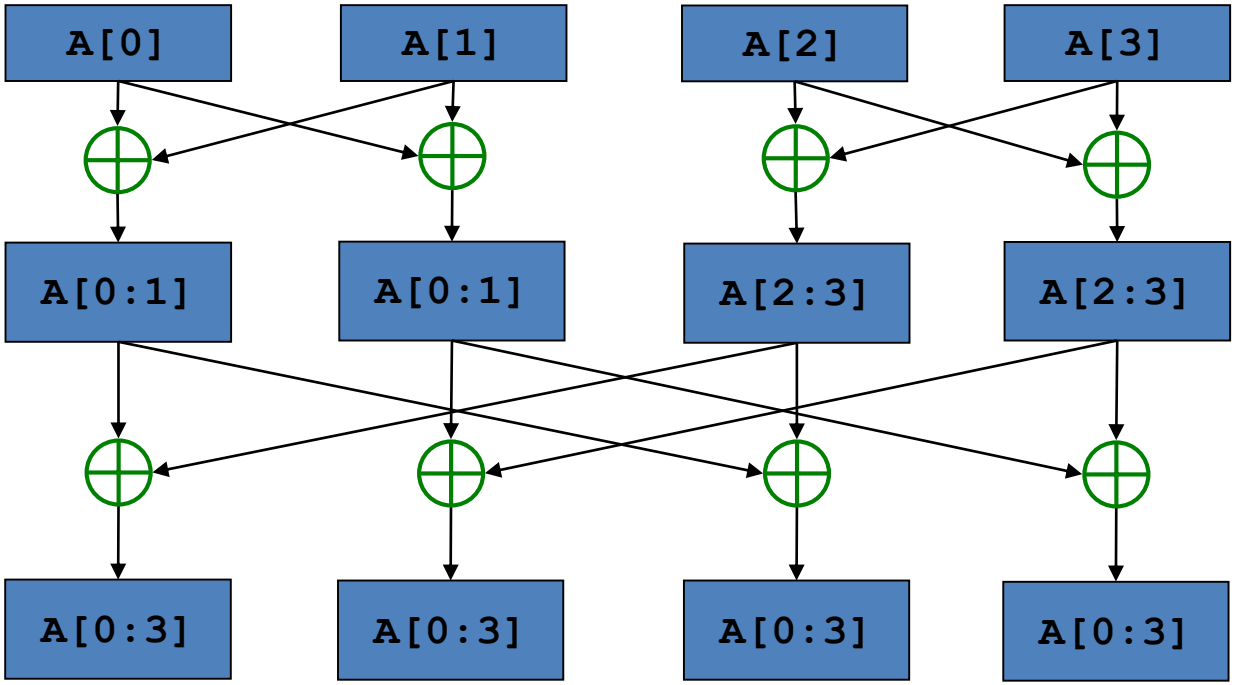
Vector Reduction with Bank Conflicts



No Bank Conflicts



Recursive-doubling Reduction



- n steps for 2^n units of execution
- If all units of execution need the result of the reduction

Recursive-doubling Reduction

- Better than tree-based approach with broadcast
  - Each units of execution has a copy of the reduced value at the end of n steps
  - In tree-based approach with broadcast
    - Reduction takes *n* steps
    - Broadcast cannot begin until reduction is complete
    - Broadcast can take *n* steps (architecture dependent)

Parallel Prefix Sum (Scan)

- Definition:

  The all-prefix-sums operation takes a binary associative operator $\oplus$ with identity I, and an array of n elements

  $$[a_0, a_1, \ldots, a_{n-1}]$$

  and returns the ordered set

  $$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})].$$

- Example:
  if $\oplus$ is addition, then scan on the se

  [3 1 7 0 4 1 6 3]

  returns the set

  [0 3 4 11 11 15 16 22]

Exclusive scan: last input element is not included in the result

(From Blelloch, 1990, "Prefix Sums and Their Applications)

Applications of Scan

- Scan is a simple and useful parallel building block
  - Convert recurrences from sequential :

```
for(j=1;j<n;j++)
   out[j] = out[j-1] + f(j);
```

  - into parallel:

```
forall(j) { temp[j] = f(j) };
scan(out, temp);
```

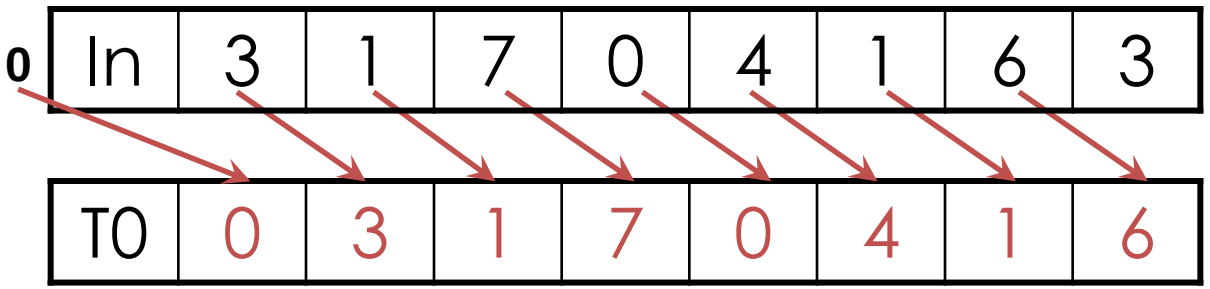
- Useful for many parallel algorithms:
  - radix sort
  - quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - **Building data structures**
  - Etc.

Scan on a serial CPU

```
void scan( float* scanned, float* input, int length)
{
  scanned[0] = 0;
  for(int i = 1; i < length; ++i)
  {
    scanned[i] = input[i-1] + scanned[i-1];
  }
}
```

- Just add each element to the sum of the elements before it
- Trivial, but sequential
- Exactly ***n*** adds: optimal

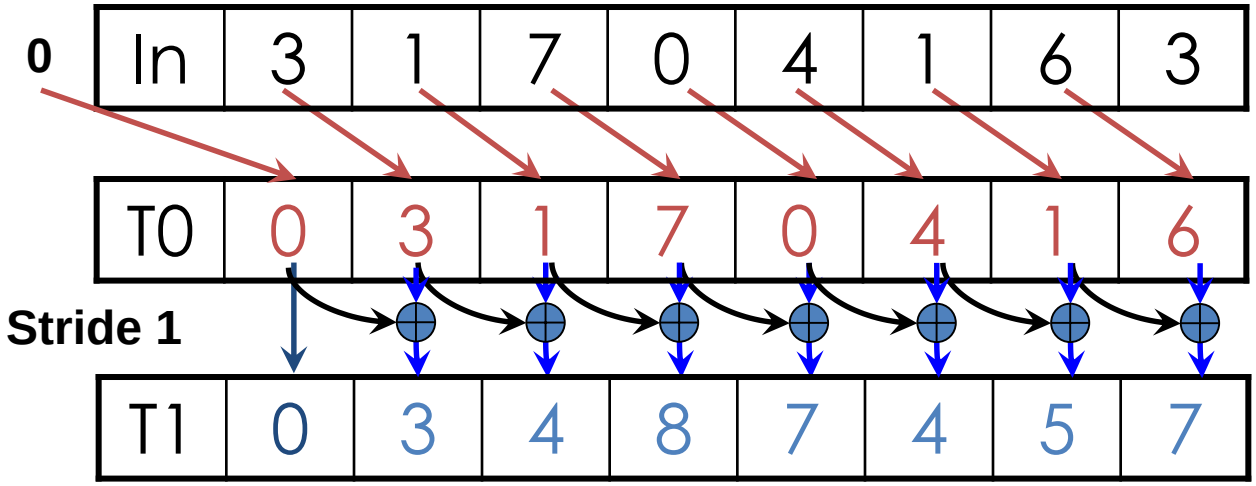
A First-Attempt Parallel Scan Algorithm



Each UE reads one value from the input array in device memory into shared memory array T0. UE 0 writes 0 into shared memory array.

1. Read input to shared memory. Set first element to zero and shift others right by one.

A First-Attempt Parallel Scan Algorithm

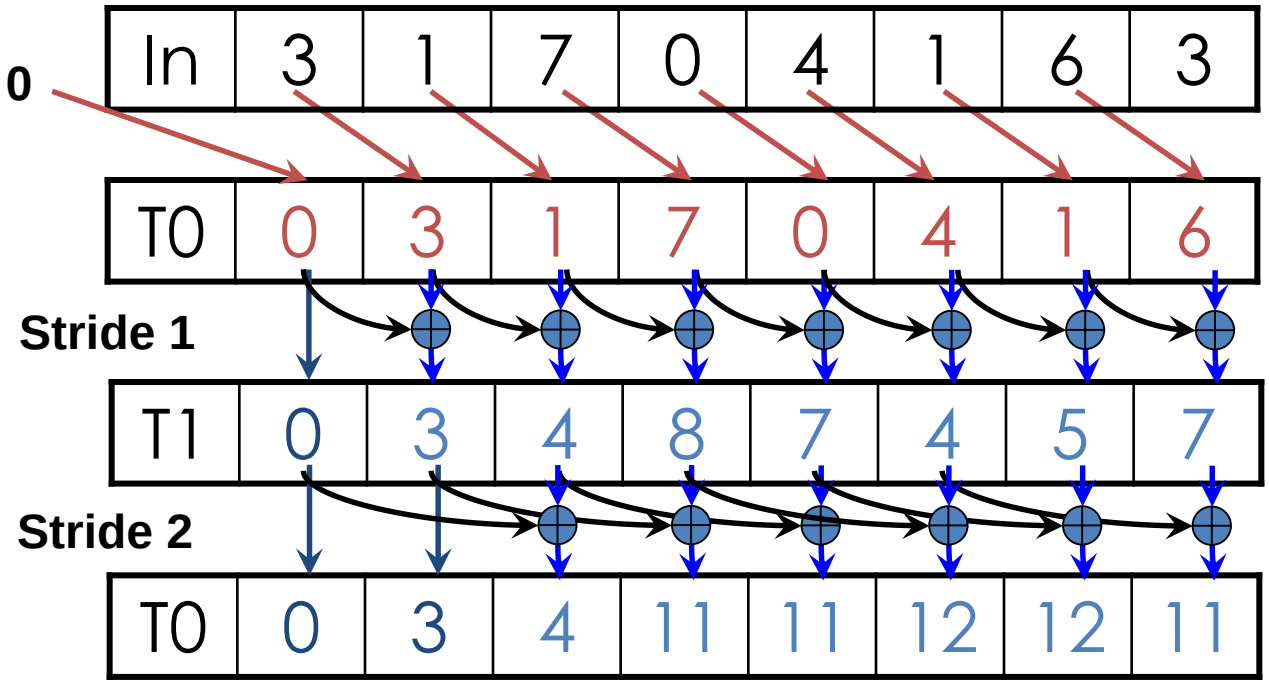


1. (previous slide)
2. Iterate log(n) times: UEs *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #1
Stride = 1

- Active UEs: *stride* to *n*-1 (*n-stride* UEs)
- UE *j* adds elements *j* and *j-stride* from T0 and writes result into shared memory buffer T1 (ping-pong)

A First-Attempt Parallel Scan Algorithm

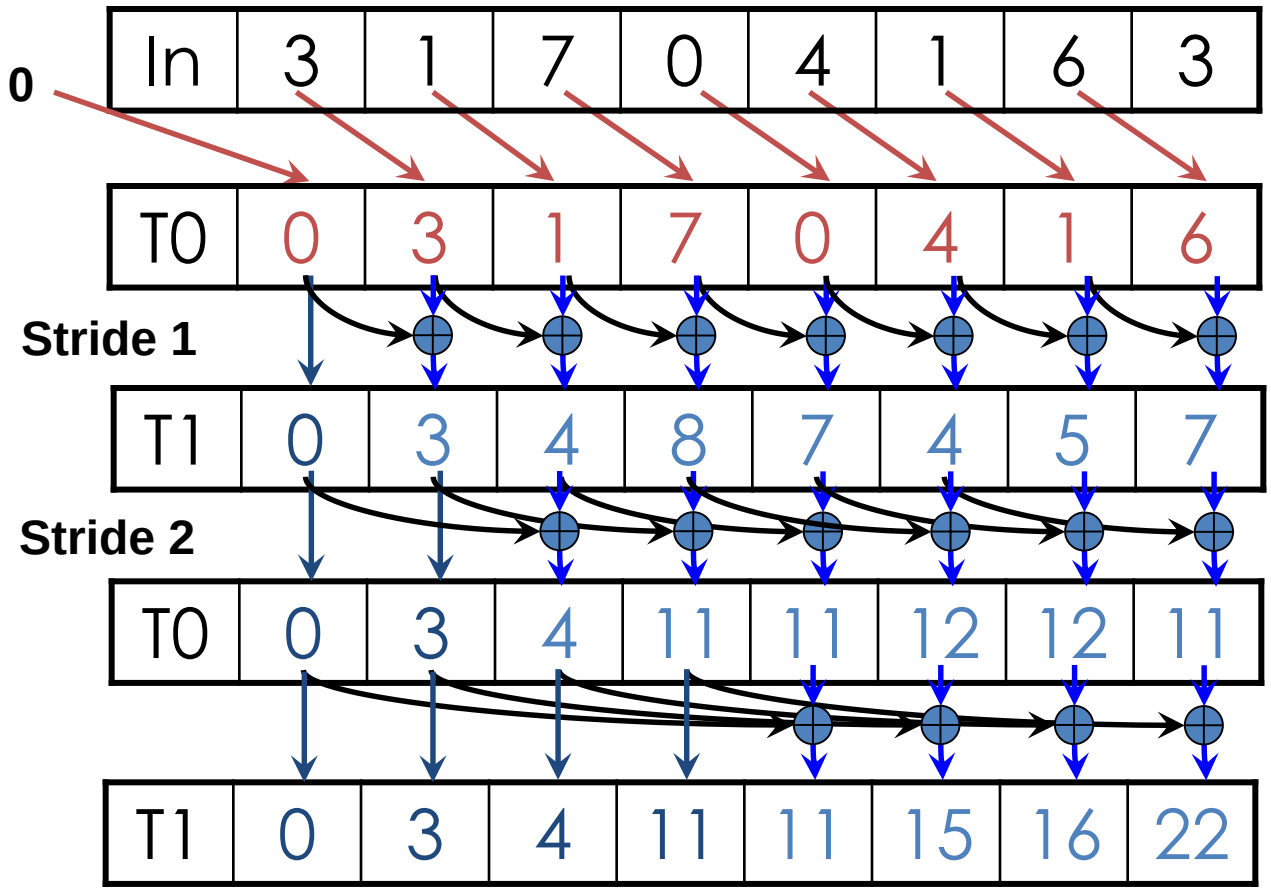


1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate log(n) times: UEs *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #2
Stride = 2

A First-Attempt Parallel Scan Algorithm

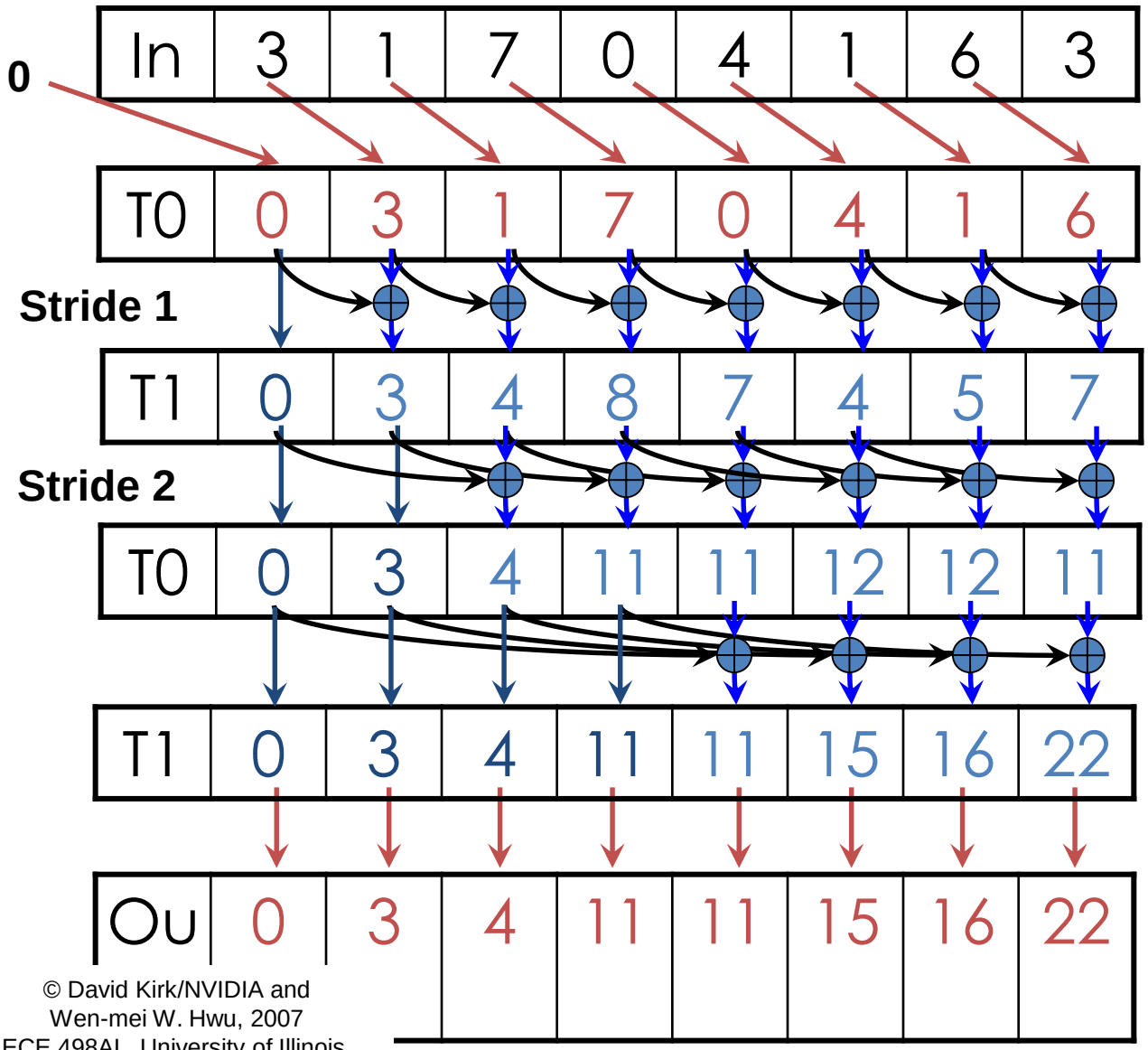


1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate log(n) times: UEs *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #3
Stride = 4

A First-Attempt Parallel Scan Algorithm



1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.
2. Iterate log(n) times: UEs *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)
3. Write output.

What is wrong with our first-attempt parallel scan?

- *Work Efficient:*
  - A parallel algorithm is work efficient if it does the same amount of work as an optimal sequential complexity

- Scan executes log(n) parallel iterations
  - The steps do n-1, n-2, n-4,... n/2 adds each
  - Total adds: n * (log(n) – 1) + 1 → O(n*log(n)) work

- This scan algorithm is NOT work efficient
  - Sequential scan algorithm does *n* adds
  - A factor of log(n) hurts: 20x for 10^6 elements!

Improving Efficiency

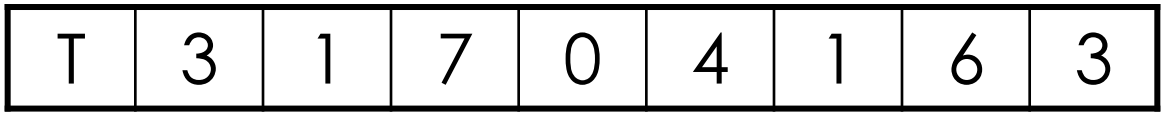
- A common parallel algorithm pattern:

  Balanced Trees

  - Build a balanced binary tree on the input data and sweep it to and from the root
  - Tree is not an actual data structure, but a concept to determine what each UE does at each step

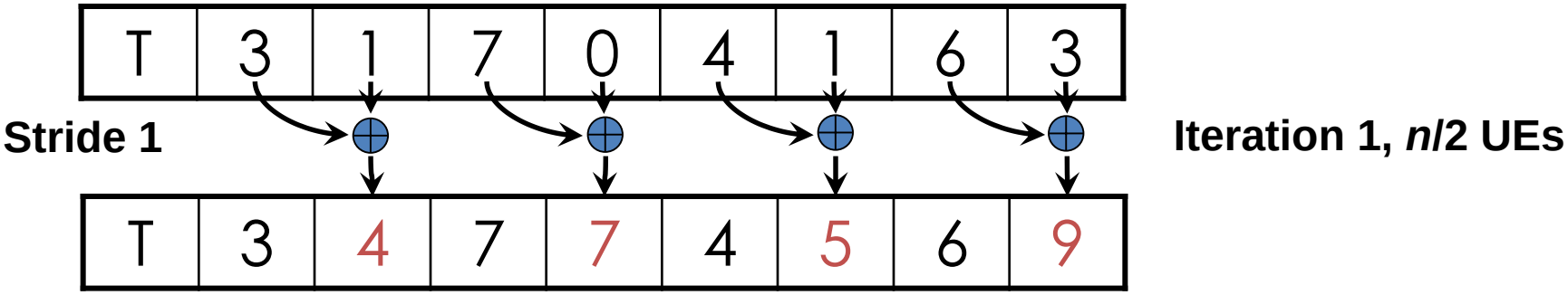
- For scan:
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  - Traverse back up the tree building the scan from the partial sums

Build the Sum Tree



Assume array is already in shared memory

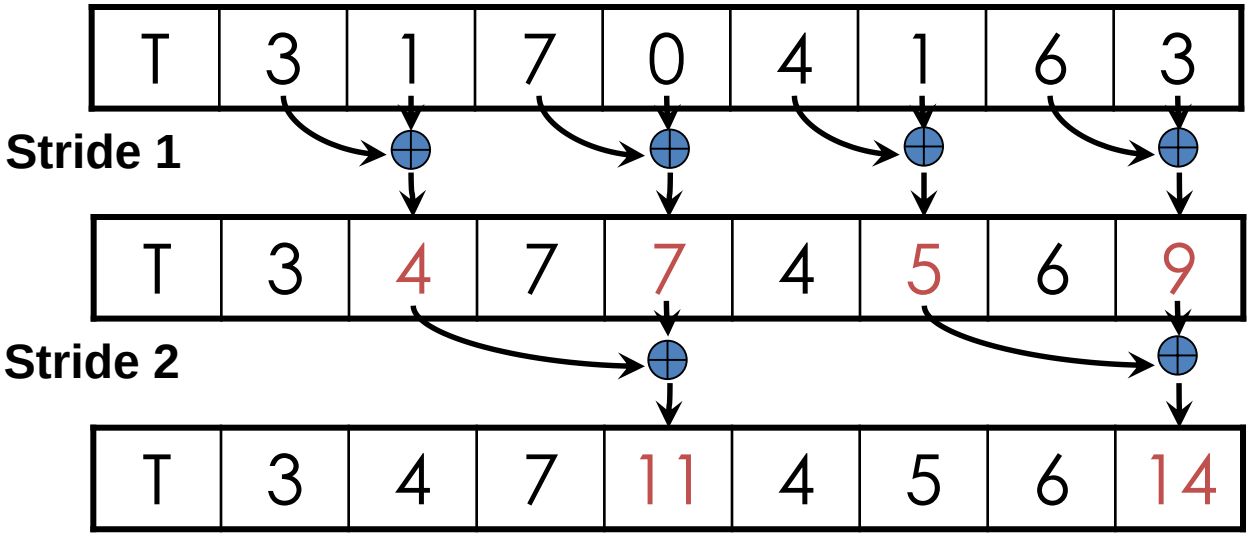
Build the Sum Tree



Each ⊕ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value

Build the Sum Tree

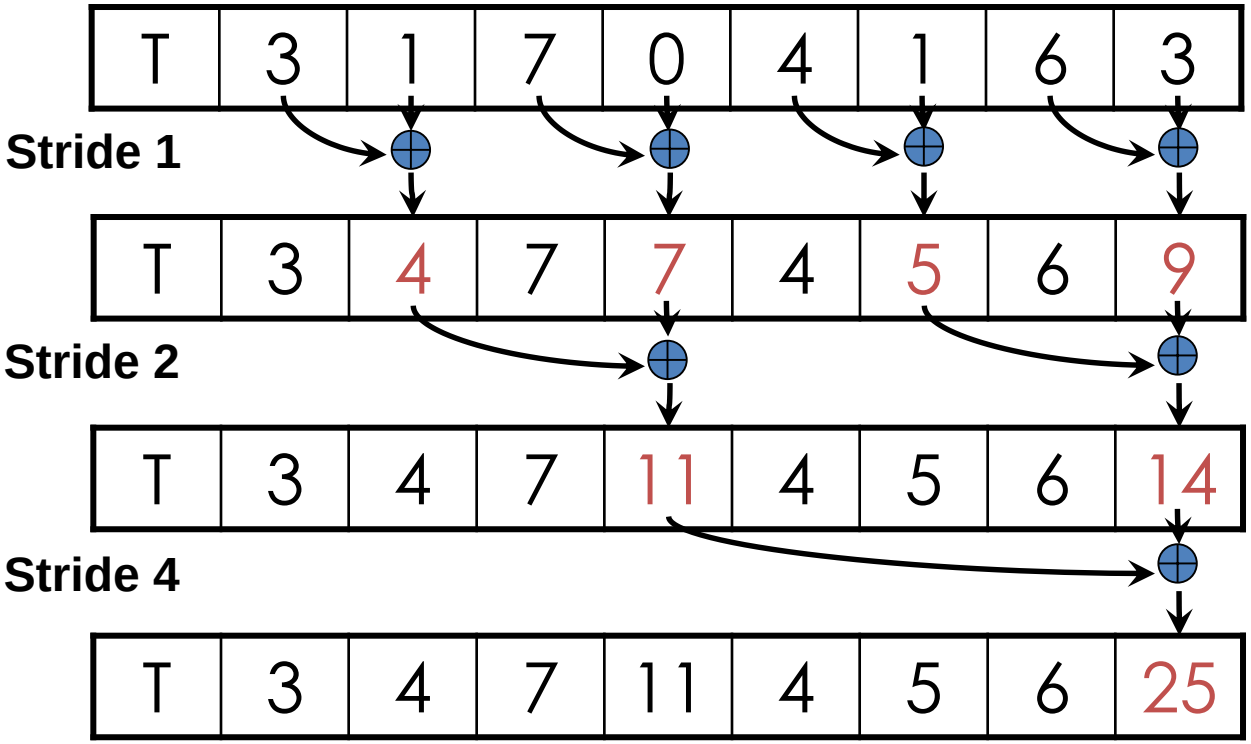


Iteration 2, *n*/4 UEs

Each ⊕ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value

Build the Sum Tree



Iteration log(*n*), 1 UE

Each ⊕ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value.

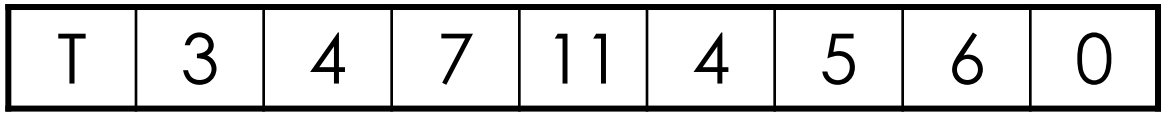
Note that this algorithm operates in-place: no need for double buffering

Zero the Last Element

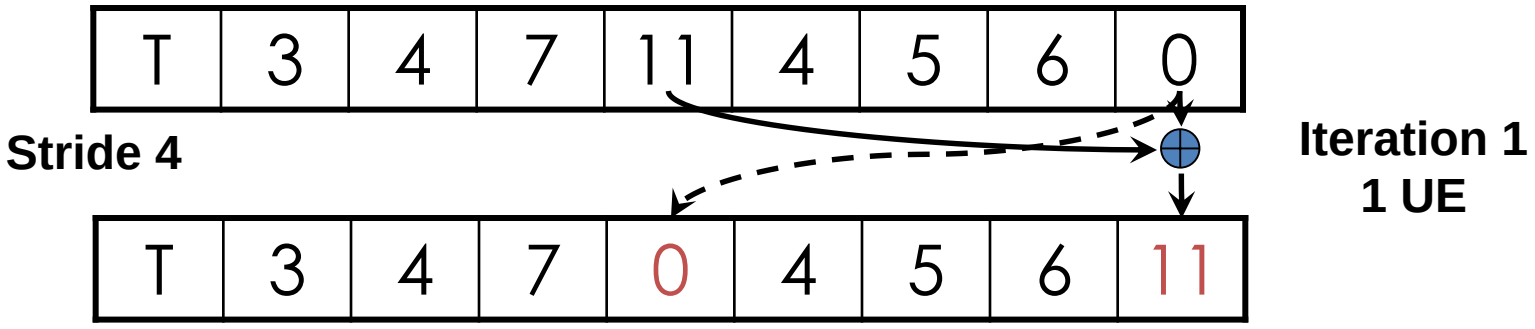
T	3	4	7	11	4	5	6	0

We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

Build Scan From Partial Sums



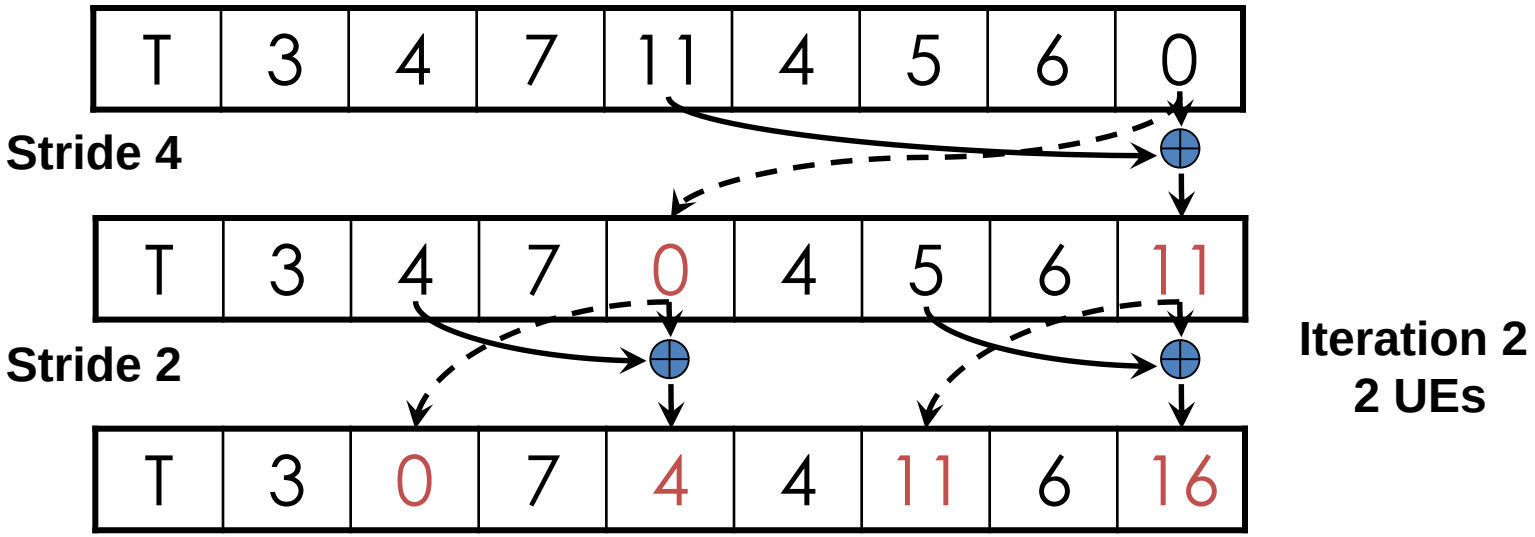
Build Scan From Partial Sums



Each ⊕ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

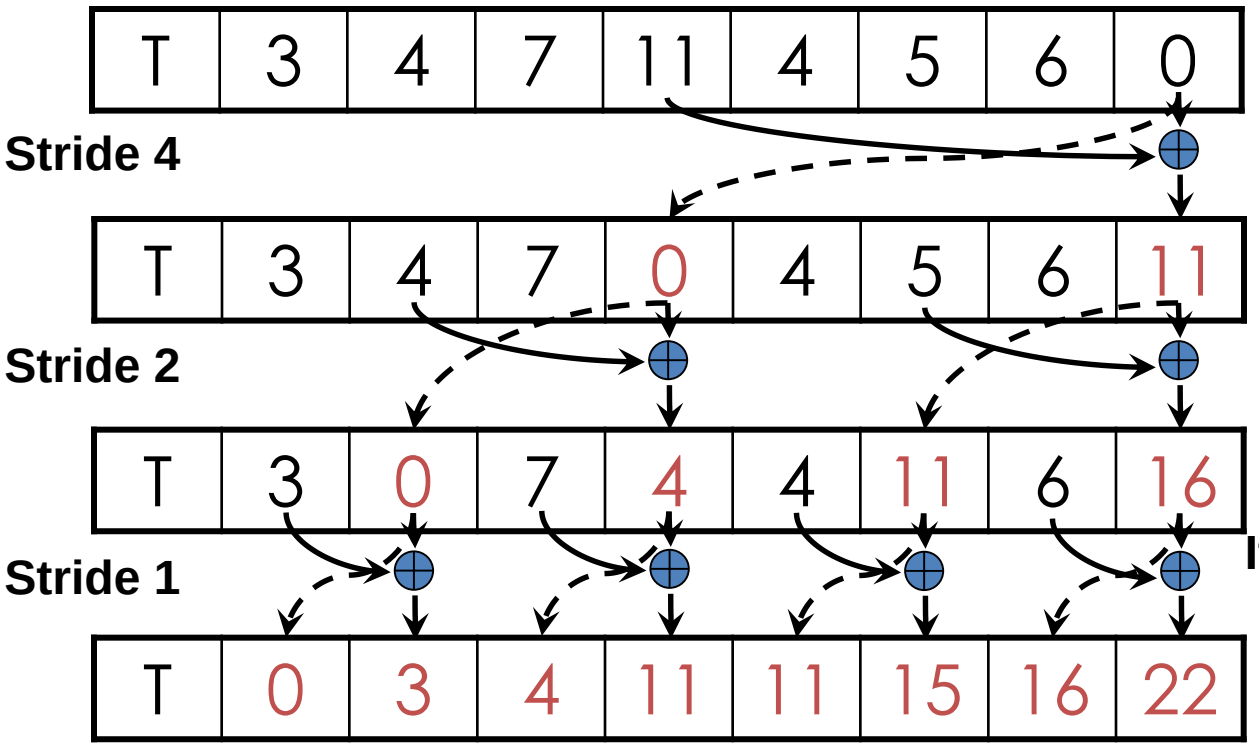
Build Scan From Partial Sums



Each ⊕ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

Build Scan From Partial Sums



Iteration log(*n*)
***n*/2 UEs**

Each ⊕ corresponds to a single UE.

Done! We now have a completed scan that we can write out to device memory.

Total steps: 2 * log(*n*).
Total work: 2 * (*n*-1) adds = *O*(*n*) **Work Efficient!**

Building Data Structures with Scans

- Fun on the board