# ECE 498AL

# Lecture 15: Reductions and Their Implementation

# Parallel Reductions

- Simple array reductions reduce all of the data in an array to a single value that contains some information from the entire array.
  - Sum, maximum element, minimum element, etc.

- Used in lots of applications, although not always in parallel form
  - Matrix Multiplication is essentially performing a sum reduction over the element-product of two vectors for each output element: but the sum is computed by a single thread

- Assumes that the operator used in the reduction is associative
  - Technically not true for things like addition on floating-point numbers, but it's common to pretend that it is

# 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 set

  [3 1 7 0 4 1 6 3]

  returns the set

  [0 3 4 11 11 15 16 22]

Each element is the array reduction of all previous elements

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

# Relevance 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:

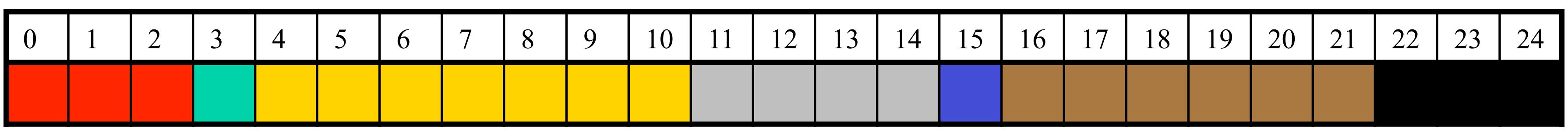
```
forall(j) { temp[j] = f(j) };
scan(out, temp);
```

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

# Example: Application of Scan

- Computing indexes into a global array where each thread needs space for a dynamic number of elements
  - Each thread computes the number of elements it will produce
  - The scan of the thread element counts will determine the beginning index for each thread.

| Tid | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Cnt | 3 | 1 | 7 | 0 | 4 | 1 | 6 | 3 |
| Scan | 0 | 3 | 4 | 11 | 11 | 15 | 16 | 22 |

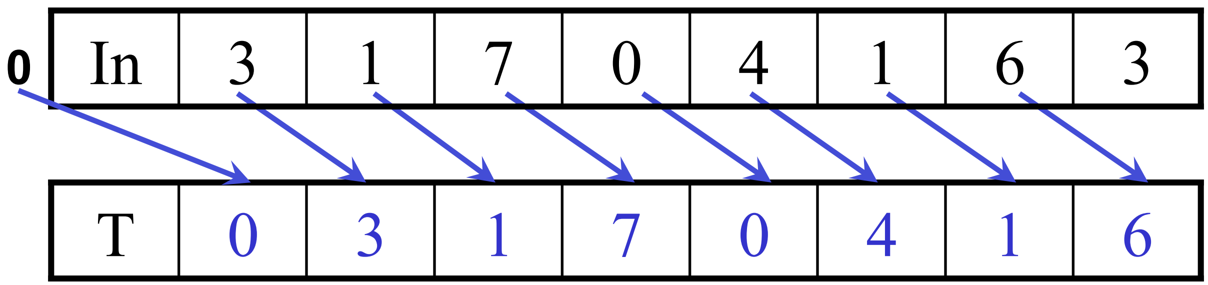


# Scan on the CPU

```
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: absolute minimum bound

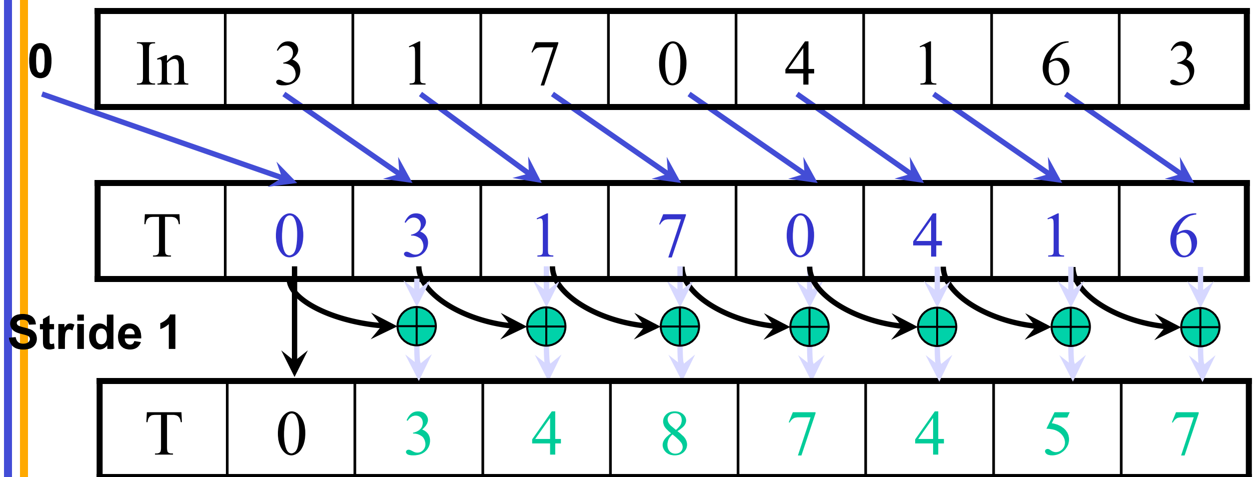
# A First-Attempt Parallel Scan Algorithm




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

1. Read from input into a temporary array we can work on in place. Set first element to zero and shift others right by one.

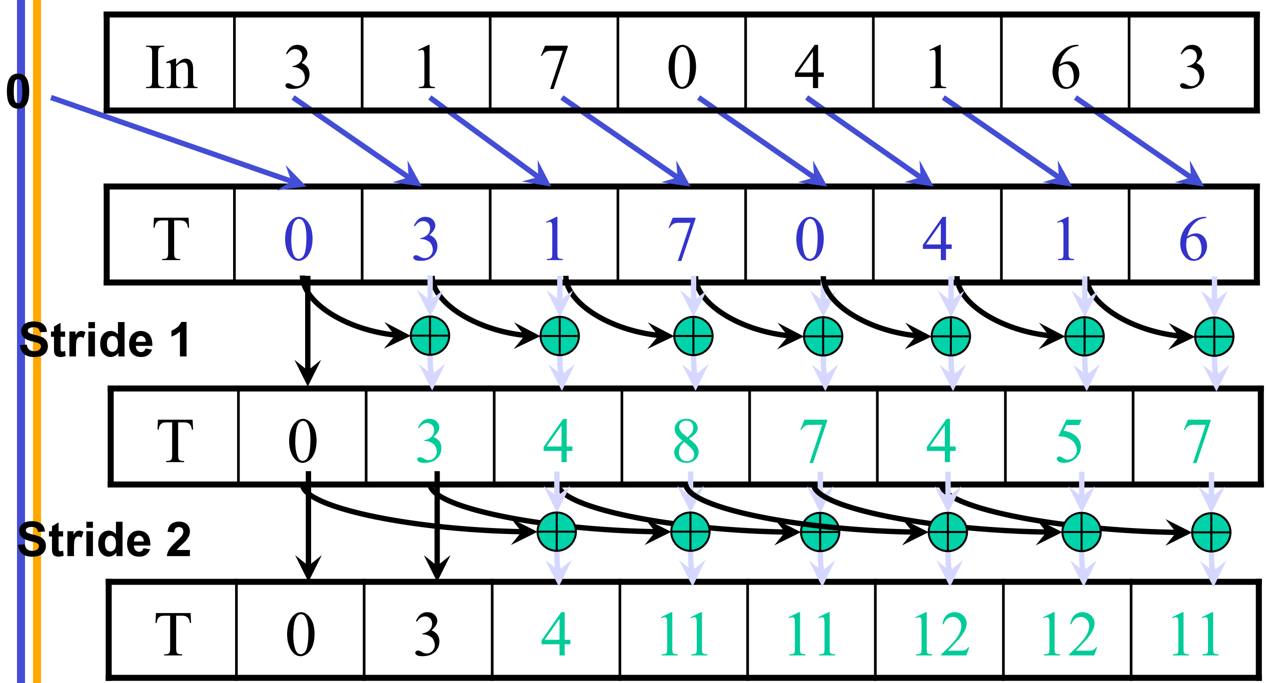
# A First-Attempt Parallel Scan Algorithm




1. Read input from into a temporary array (shared memory in CUDA). Set first element to zero and shift others right.
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration.

| Iteration #1<br>Stride = 1 | • Active threads: *stride* to *n*-1 (*n-stride* threads)<br>• Thread *j* adds elements *j* and *j-stride* from T |
|---|---|

# A First-Attempt Parallel Scan Algorithm

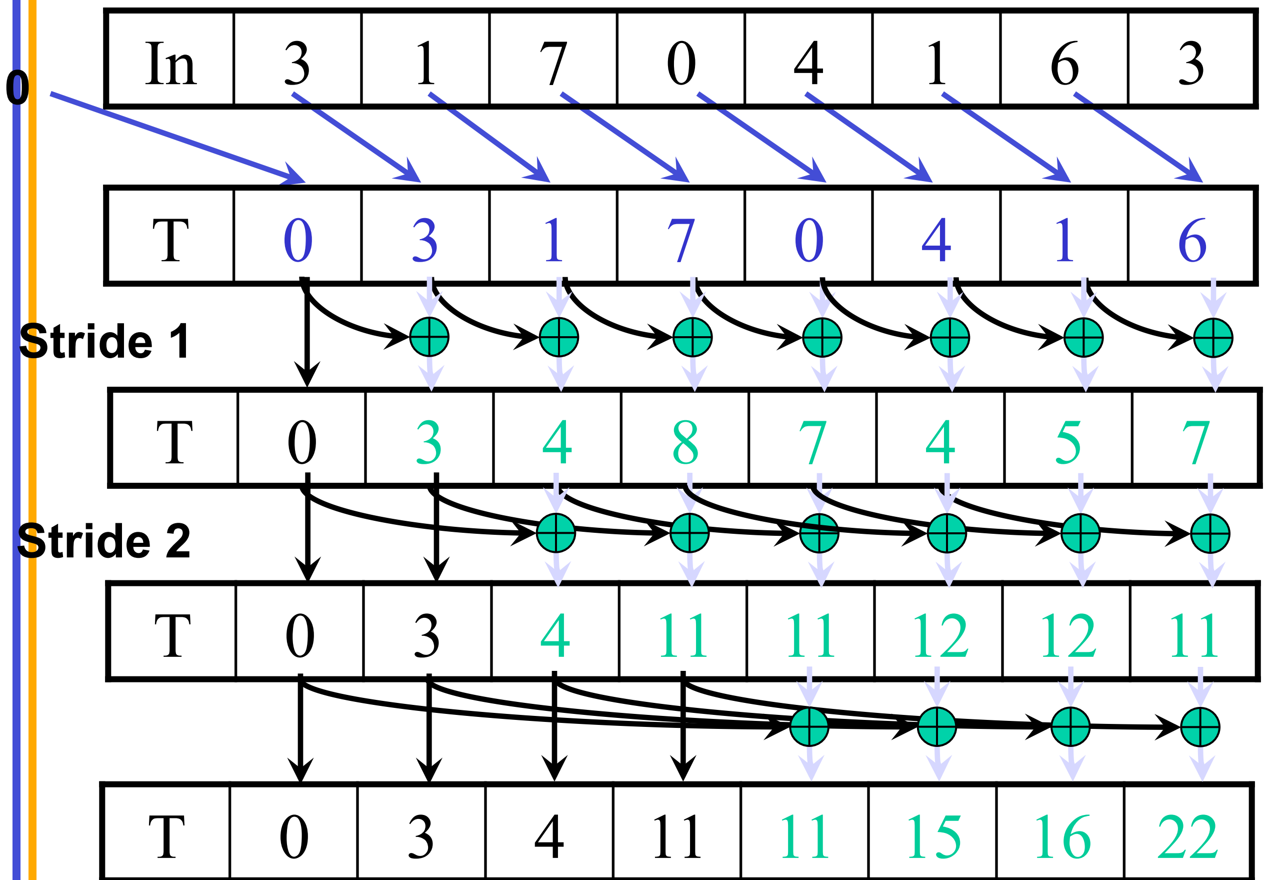



1. Read input from into a temporary array (shared memory in CUDA). Set first element to zero and shift others right.
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration.

Iteration #2
Stride = 2

Note that threads need to synchronize before they read and before they write to T. Double-buffering can allow them to only synchronize after writing.

# A First-Attempt Parallel Scan Algorithm



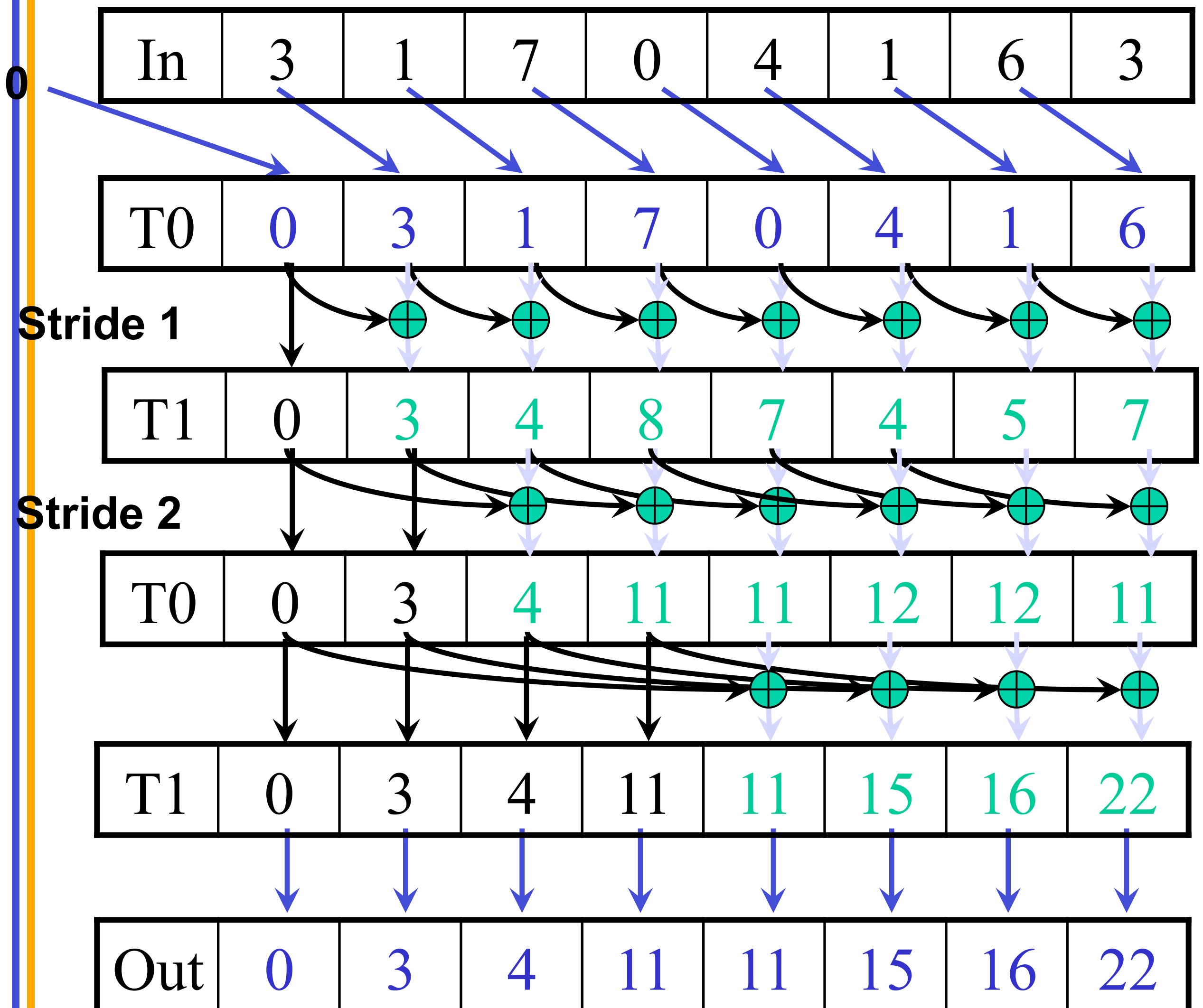

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration.

Iteration #3
Stride = 4

After each iteration, each array element contains the sum of the previous 2**stride* elements of the original array.

# 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: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration.

3. Write output.

# Work Efficiency Considerations

- The first-attempt Scan executes log(n) parallel iterations
  - The steps do at least n/2 operations every step for log(n) steps
  - Total adds → O(n*log(n)) work
- This scan algorithm is not very efficient on finite resources
  - Presumably, if you have N or more parallel processors, the number of steps matters more than the number of operations
  - For larger reductions, finite resources get their workload multiplied by factor of log(n) compared to a sequential implementation.
- Log(1024) = 10: this gets bad very quickly

# 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 thread does at each step

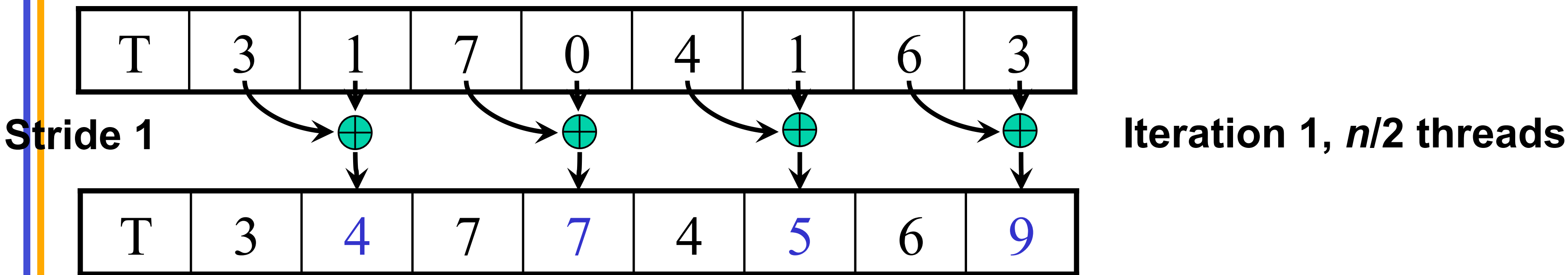
- For scan:
  – Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  – Traverse back up the tree building the scan from the partial sums

# Build the Sum Tree

| T | 3 | 1 | 7 | 0 | 4 | 1 | 6 | 3 |
|---|---|---|---|---|---|---|---|---|

Assume array is already in the temp array

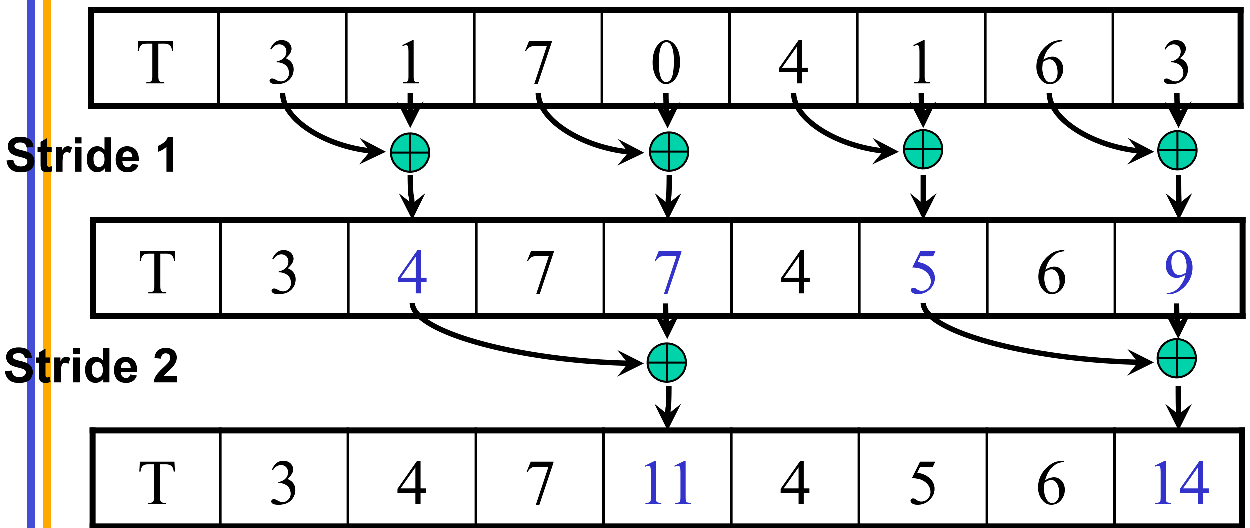
# Build the Sum Tree




Each ⊕ corresponds to a single thread.

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

# Build the Sum Tree

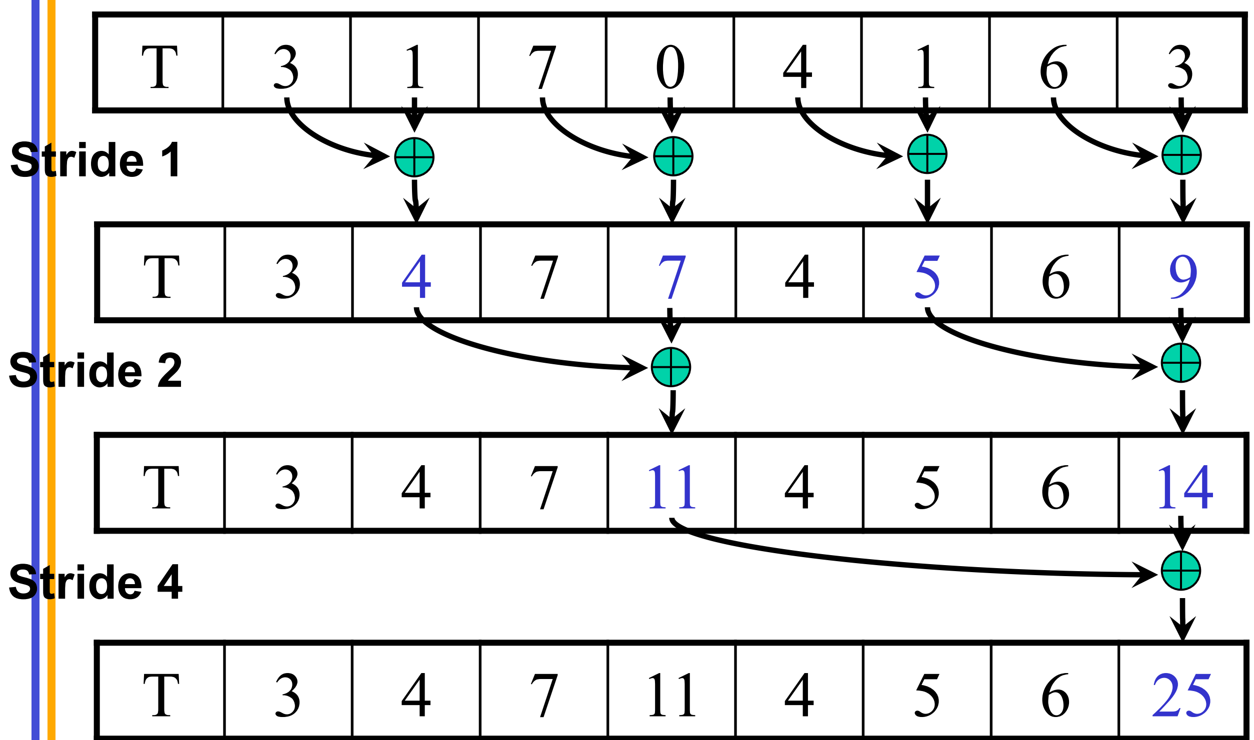



**Iteration 2, *n*/4 threads**

Each ⊕ corresponds to a single thread.

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

# Build the Sum Tree




**Iteration log(*n*), 1 thread**

Each ⊕ corresponds to a single thread.

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

After step with stride *k*, elements with indexes divisible by 2*k* contain the partial sum of itself and the preceding *2k-1* elements.

# Partial Sum Array

| Idx | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| T | 3 | 4 | 7 | 11 | 4 | 5 | 6 | 25 |
| Sum | 0 | 0-1 | 2 | 0-3 | 4 | 4-5 | 6 | 0-7 |

Index *k* holds the partial sum of the elements from *j* to *k* where *j* is the greatest index less than or equal to *k* that is divisible by the greatest power of 2 by which *k+1* is divisible, or 0 if there is none.

Trust me, it works

# Zero The Last Element

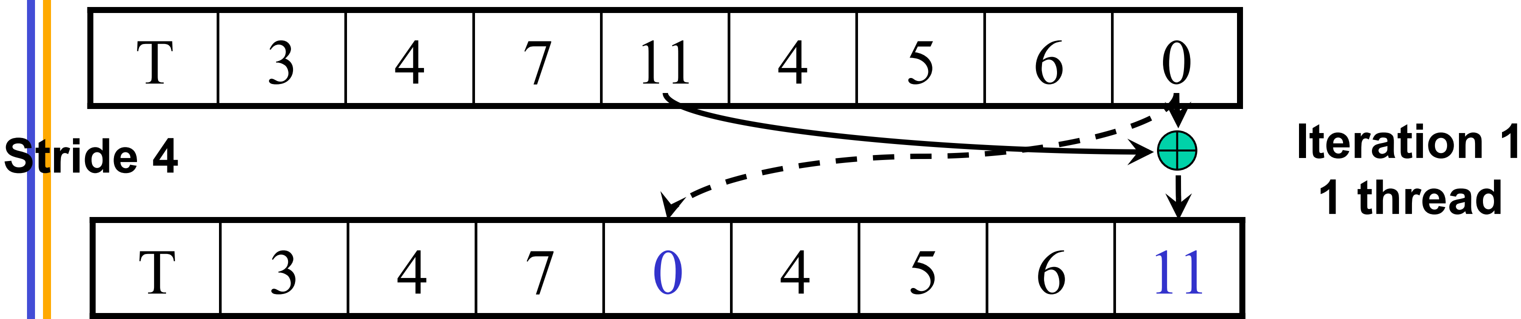
| Idx | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| T | 3 | 4 | 7 | 11 | 4 | 5 | 6 | 0 |
| Sum | 1 | 1-2 | 3 | 1-4 | 5 | 5-6 | 7 | 1-8 |

It's an exclusive scan, so we don't need it.
It'll propagate back to the beginning in the upcoming steps.

# Build Scan From Partial Sums

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| T | 3 | 4 | 7 | 11 | 4 | 5 | 6 | 0 |

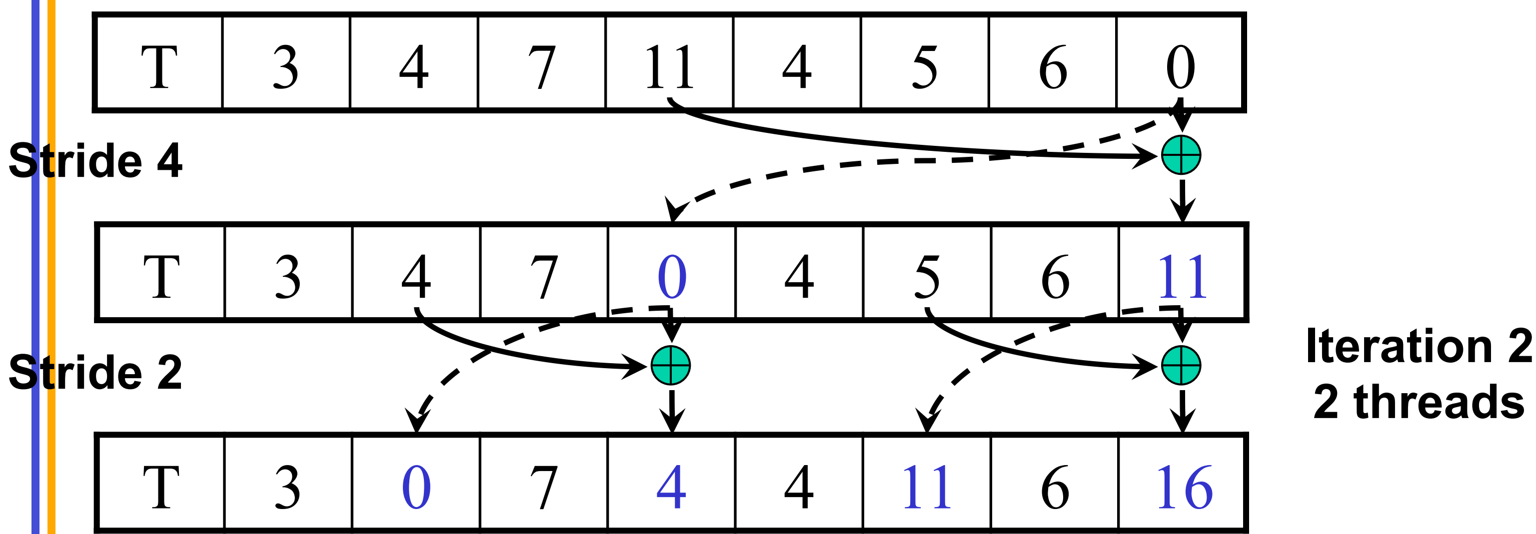
# Build Scan From Partial Sums




Each ⊕ corresponds to a single thread.

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

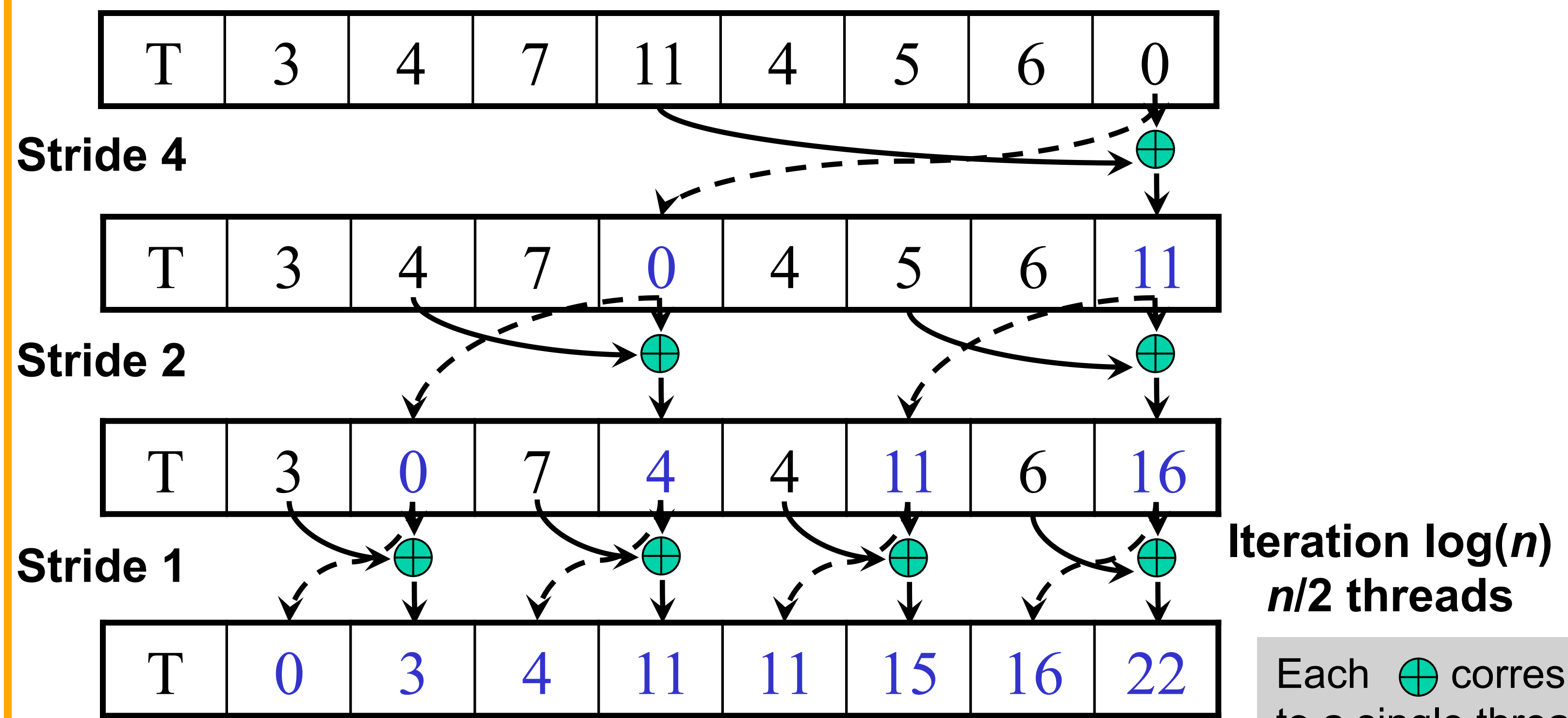
# Build Scan From Partial Sums




Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread 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 thread.

Done! We now have a completed scan that we can write to output.

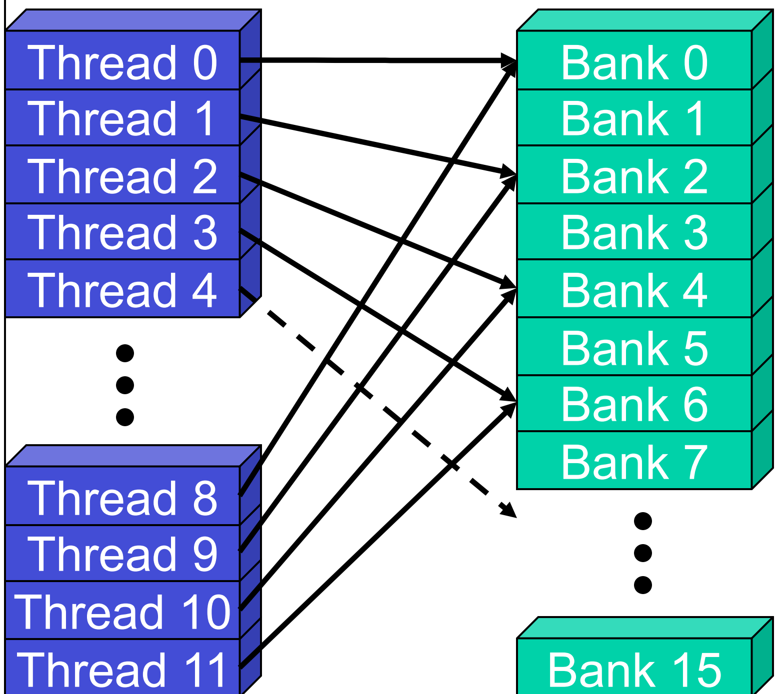
Total steps: 2 * log(*n*).
Total operations: 2 * (*n*-1) adds - **Work Efficient!**

# Shared memory bank conflicts

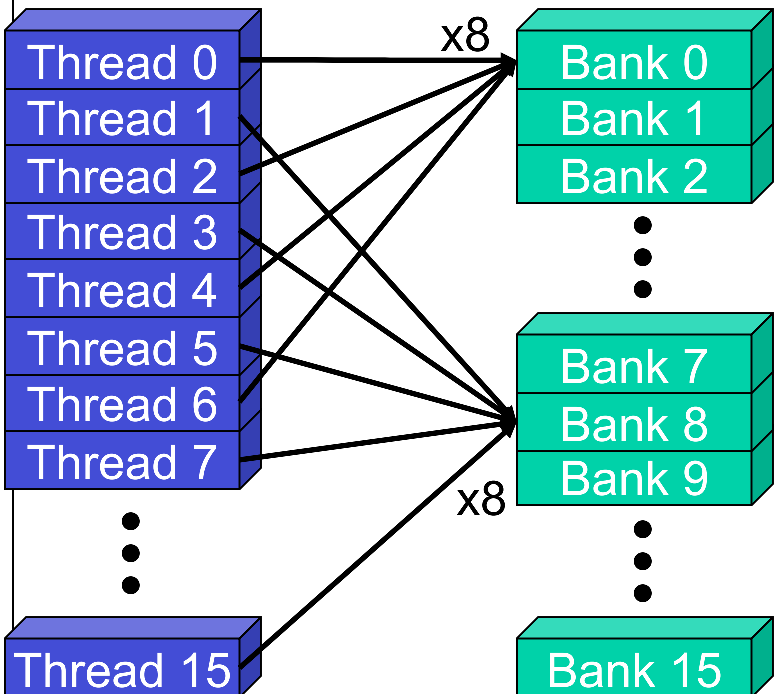
- Shared memory is as fast as registers if there are no bank conflicts
- The fast cases:
  - All 16 threads of a half-warp access different banks: no bank conflict
  - All 16 threads of a half-warp access the same address: broadcast
- The slow case:
  - Multiple threads in the same half-warp access different values in the same bank
  - Must serialize the accesses
  - Cost = max # of values requested from one of the 16 banks

# Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride == 2




- 8-way Bank Conflicts
  - Linear addressing stride == 8



# Use Padding to Reduce Conflicts

- This is a simple modification to the indexing

- After you compute a shared mem address like this:

```
Address = 2 * stride * thid;
```

- Add padding like this:

```
Address += (Address / 16); // divide by NUM_BANKS
```

- This removes most bank conflicts
  - Not all, in the case of deep trees, but good enough for us

# Fixing Scan Bank Conflicts

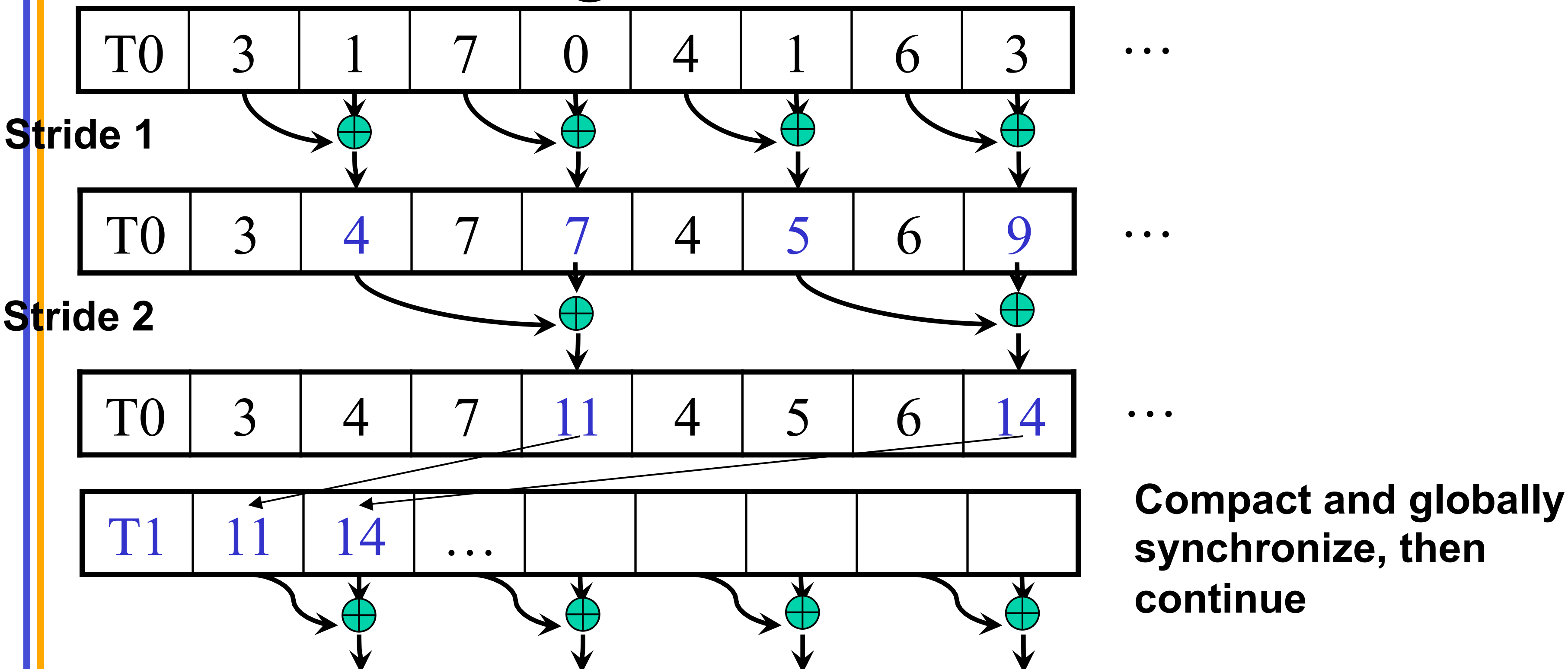
- Insert padding every NUM_BANKS elements

```
const int LOG_NUM_BANKS = 4; // 16 banks on G80
int tid = threadIdx.x;
int s = 1;
// Traversal from leaves up to root
for (d = n>>1; d > 0; d >>= 1)
{
   if (thid <= d)
   {
         int a = s*(2*tid); int b = s*(2*tid+1)
         a += (a >> LOG_NUM_BANKS); // insert pad word
         b += (b >> LOG_NUM_BANKS); // insert pad word
         shared[a] += shared[b];
   }
}
```

# What About Really Big Arrays?

- What if the array doesn't fit in shared memory?
  - After all, we care about parallelism because we have big problems, right?
- Tiled reduction with global synchronization
  1. Tile the input and perform reductions on tiles with individual thread blocks
  2. Store the intermediate results from each block back to global memory to be the input for the next kernel
  3. Repeat as necessary

# Building the Global Sum Tree




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

After step with stride *k*, elements with indexes divisible by 2*k* contain the partial sum of itself and the preceding *2k-1* elements.

# Global Synchronization in CUDA

- Remember, there is no barrier synchronization between CUDA thread blocks
  - You can have some limited communication through atomic integer operations on global memory on newer devices
  - Doesn’t conveniently address the global reduction problem, or lots of others
- To synchronize, you need to end the kernel (have all thread blocks complete)
  - Then launch a new one