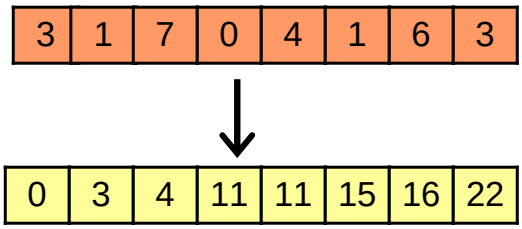
Scan

General-purpose Programming of Massively Parallel Graphics Processors
Shiraz University, Spring 2010
Instructor: Reza Azimi

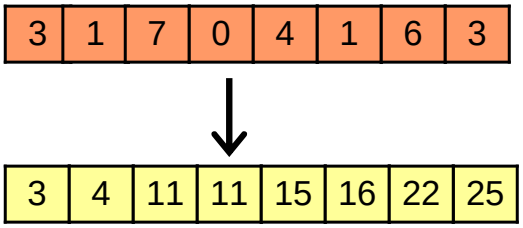
The slides are primarily adapted from:
Andreas Moshovos' Course at UofT

Scan / Parallel Prefix Sum



- Given
  - an array $A = [a_0, a_1, \ldots, a_{n-1}]$
  - a binary associative operator @ with identity **I**
- $scan(A) = [I, a_0, (a_0 \mathbin{@} a_1), \ldots, (a_0 \mathbin{@} a_1 \mathbin{@} \ldots \mathbin{@} a_{n-2})]$
  - This is called **exclusive scan**

Scan / Parallel Prefix Sum

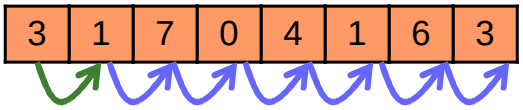


- Given
  - an array $A = [a_0, a_1, \ldots, a_{n-1}]$
  - a binary associative operator @ with identity **I**
- $scan(A) = [a_0, (a_0 \mathbin{@} a_1), \ldots, (a_0 \mathbin{@} a_1 \mathbin{@} \ldots \mathbin{@} a_{n-1})]$
  - This is called **inclusive scan**

Applications of Scan

- Scan is used as a building block for many parallel algorithms
  - Radix sort
  - Quicksort
  - String comparison
  - Lexical analysis
  - Run-length encoding
  - Histograms
  - Etc.
- See:
  - Guy E. Blelloch. *"Prefix Sums and Their Applications".* http://www.cs.cmu.edu/~blelloch/papers/Ble93.pdf

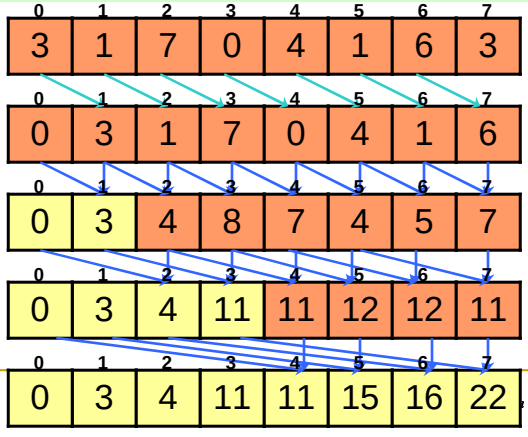
Sequential Algorithm




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


Naïve Parallel Algorithm

```
for d := 1 to log2n do
  forall k in parallel do
     if k  >= 2^d then x[k] := x[k − 2^(d-1)] + x[k]
```



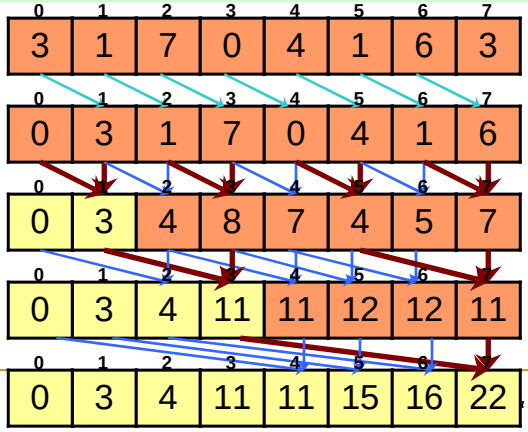
d = 1, 2^{d-1} = 1

d = 2, 2^{d-1} = 2

d = 3, 2^{d-1} = 4

Naïve Parallel Algorithm

```
for d := 1 to log2n do
  forall k in parallel do
     if k  >= 2^d then x[k] := x[k – 2^(d-1)] + x[k]
```



d = 1, $2^{d-1} = 1$

d = 2, $2^{d-1} = 2$

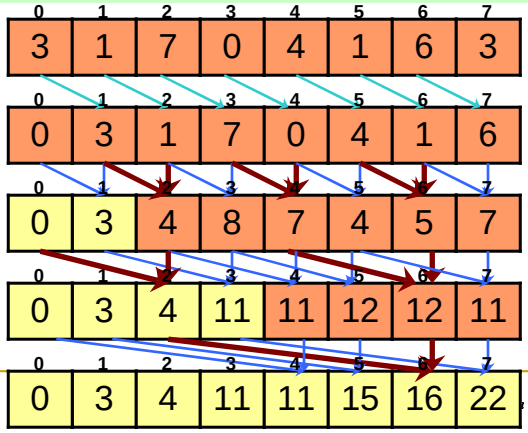
d = 3, $2^{d-1} = 4$

Azimi

7

Naïve Parallel Algorithm

```
for d := 1 to log2n do
  forall k in parallel do
     if k  >= 2^d then x[k] := x[k – 2^(d-1)] + x[k]
```



d = 1, $2^{d-1} = 1$

d = 2, $2^{d-1} = 2$

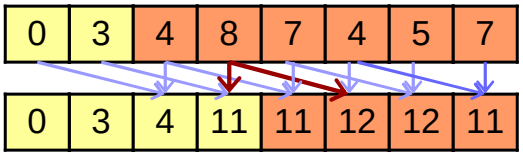
d = 3, $2^{d-1} = 4$

Azimi

8

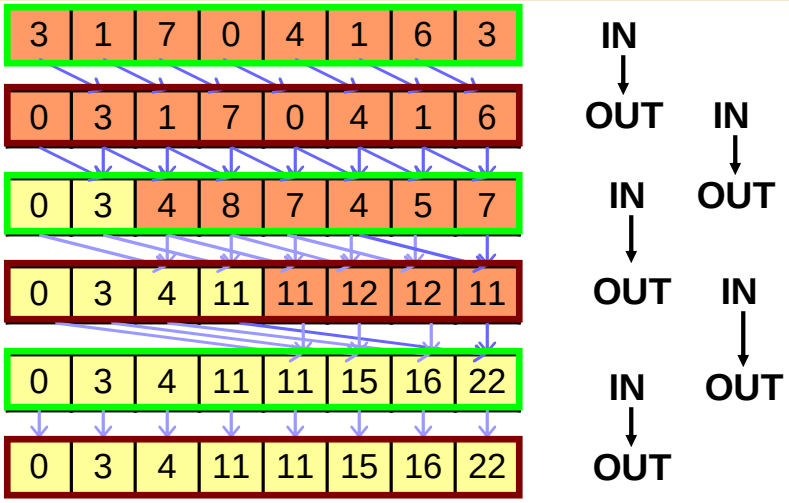
Need Double-Buffering

- First all read
- Then all write



- Solution
  - Use two arrays:
    - Input & Output
  - Alternate at each step

Double Buffering



Naïve Kernel in CUDA

```
__global__ void
scan_naive(float *g_odata, float *g_idata, int n) {
   extern  __shared__  float temp[];
    int thid = threadIdx.x, pout = 0, pin = 1;

   temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;

    for (int dd = 1; dd < n; dd *= 2) {
        pout = 1 - pout; pin = 1 - pin;
        int basein = pin * n, baseout = pout * n;
        syncthreads();
        temp[baseout +thid] = temp[basein +thid];

        if (thid >= dd)
             temp[baseout +thid] += temp[basein + thid - dd];
    }
    syncthreads();
    g_odata[thid] = temp[baseout +thid];
}
```

loading into shared mem

Switching In & Out Buffers

11

Analysis of naïve kernel

- This scan algorithm executes *log(n)* parallel iterations
  - The steps do n-1, n-2, n-4,... n/2 adds each
  - Total adds: **O(n*log(n))**
- This scan algorithm is NOT work efficient
  - Sequential scan algorithm does ***n*** adds

Improving Work Efficiency

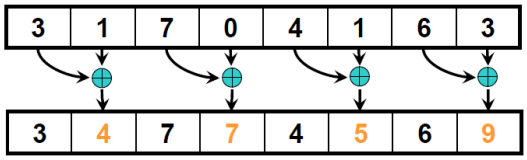
- A parallel algorithms based on *Balanced Trees*
  - Build balanced binary tree on input data and sweep to and from the root
  - Tree is conceptual, not an actual data structure
- For scan:
  - Traverse from leaves to root building partial sums at internal nodes
    - Root holds sum of all leaves
  - Traverse from root to leaves building the scan from the partial sums
- Algorithm originally described by Blelloch (1990)

Balanced Tree-Based Scan: Up-Sweep

3	1	7	0	4	1	6	3

Assume array is already in shared memory

Balanced Tree-Based Scan: Up-Sweep

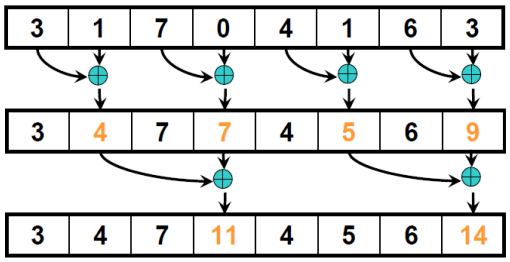


Iteration 1, *n*/2 threads

Each ⊕ corresponds to a single thread.

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

Balanced Tree-Based Scan: Up-Sweep

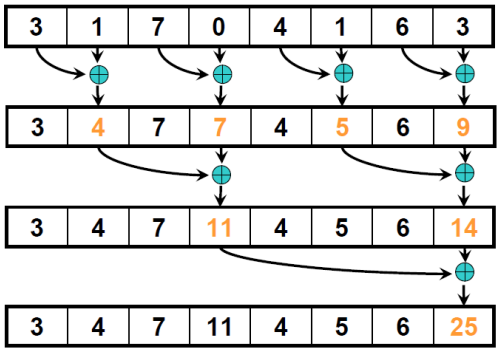


Iteration 2, *n*/4 threads

Each ⊕ corresponds to a single thread.

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

Balanced Tree-Based Scan: Up-Sweep



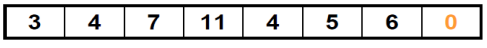
Iteration log(*n*), 1 thread

Each ⊕ corresponds to a single thread.

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

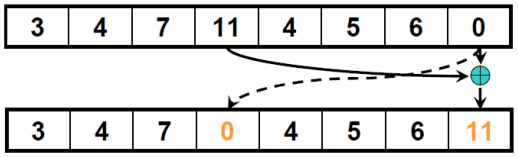
Note that this algorithm operates in-place: no need for double buffering

Balanced Tree-Based Scan: Up-Sweep



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.

Balanced Tree-Based Scan: Down-Sweep

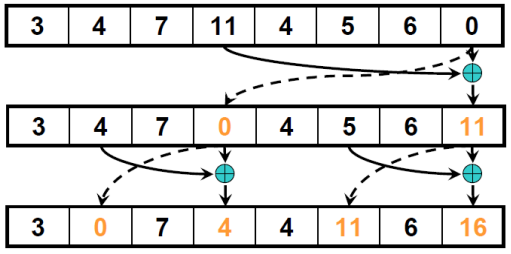


Iteration 1
1 thread

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.

Balanced Tree-Based Scan: Down-Sweep

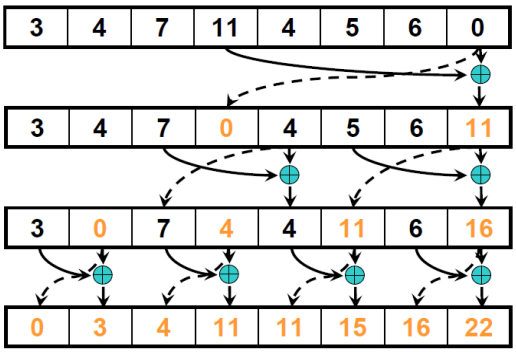


Iteration 2
2 threads

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.

Balanced Tree-Based Scan: Down-Sweep

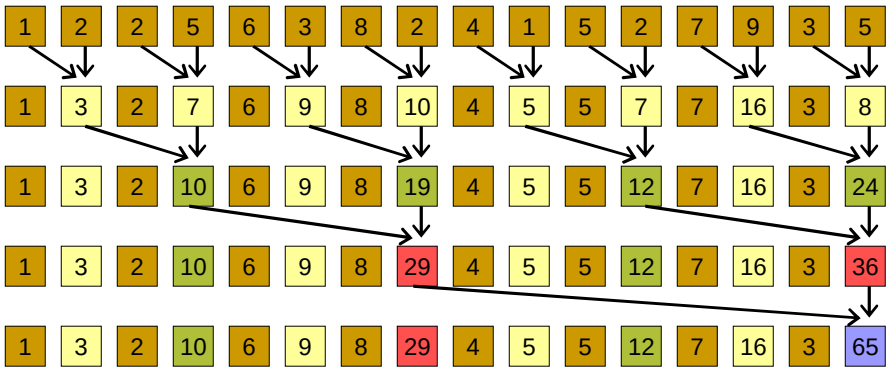


Iteration log(*n*)
***n*/2 threads**

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

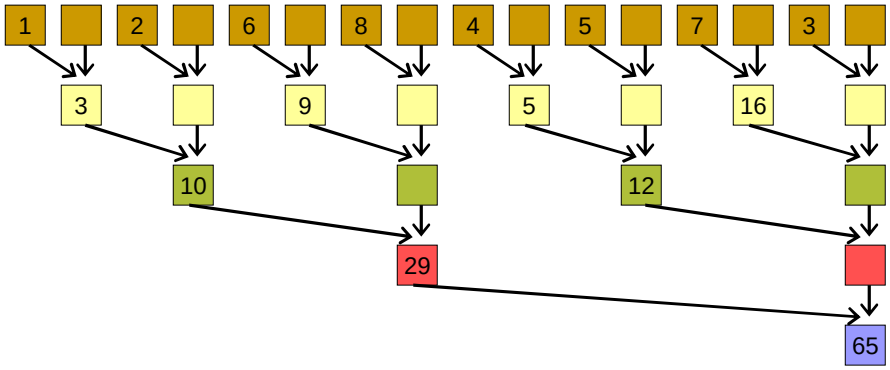
Total steps: 2 * log(*n*).
Total work: 2 * (*n*-1) adds = *O*(*n*) **Work Efficient!**

Up-Sweep



Additional explanatory slides for scan by A. Moshovos

Up-Sweep

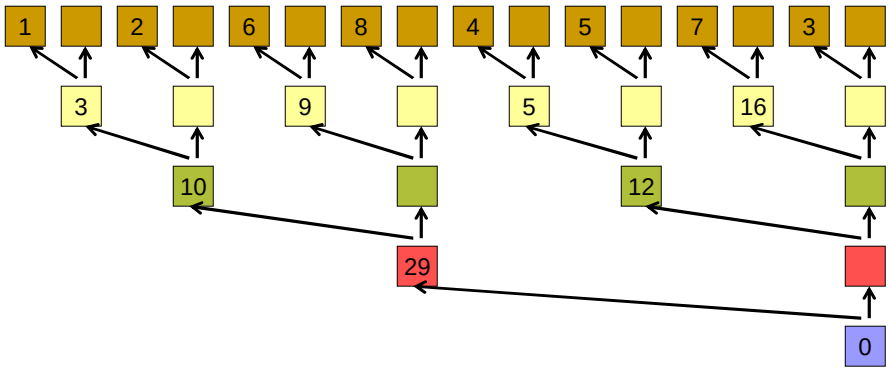


Nodes without value are places where intermediate results are written to (possibly many times)

Additional explanatory slides for scan by A. Moshovos

23

Down-Sweep

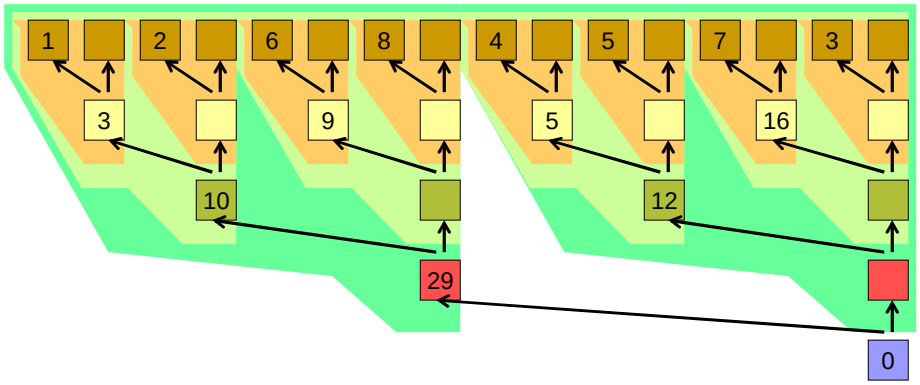


Edge directions are reversed.

GPU Programming, Shiraz University, Winter 88/Spring 89, Reza Azimi

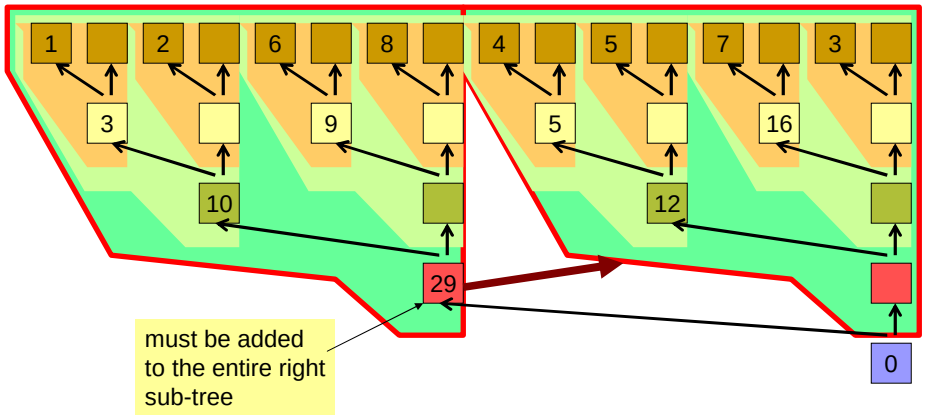
24

Down-Sweep

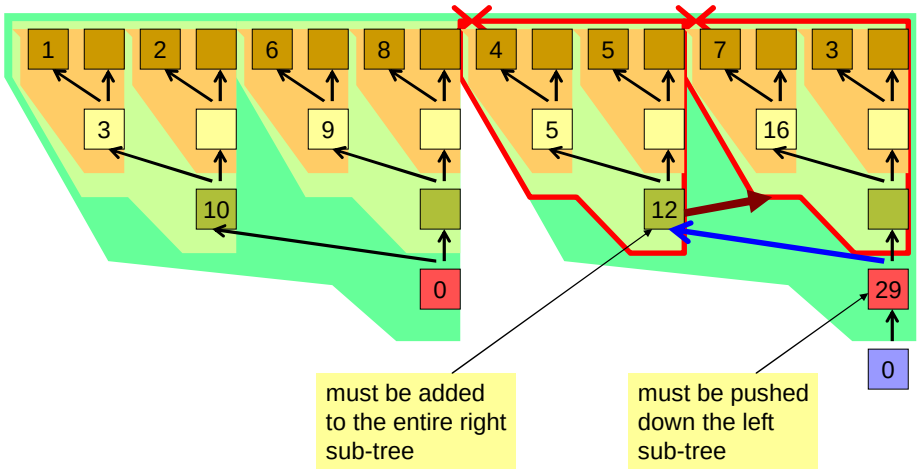


recursive subtree structure

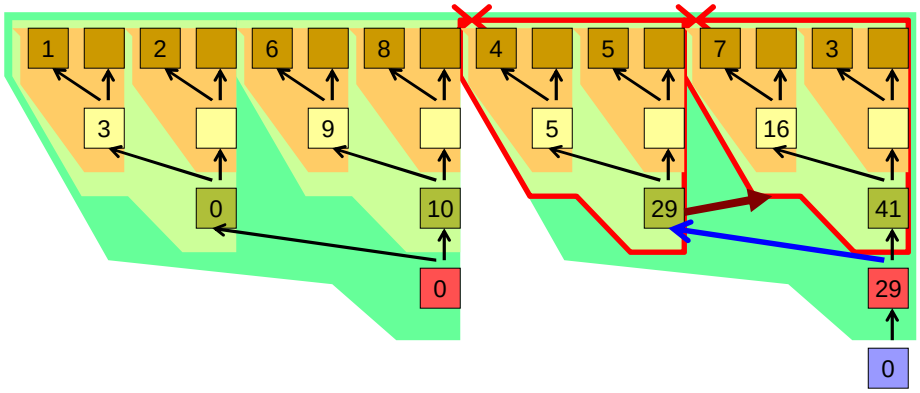
Down-Sweep



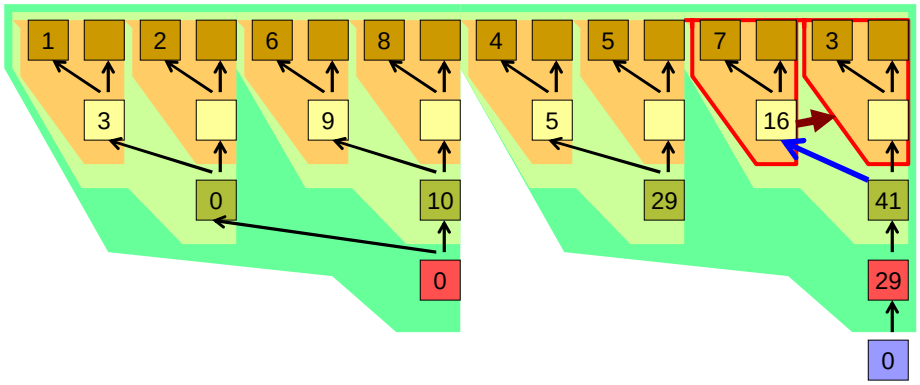
Down-Sweep



Down-Sweep

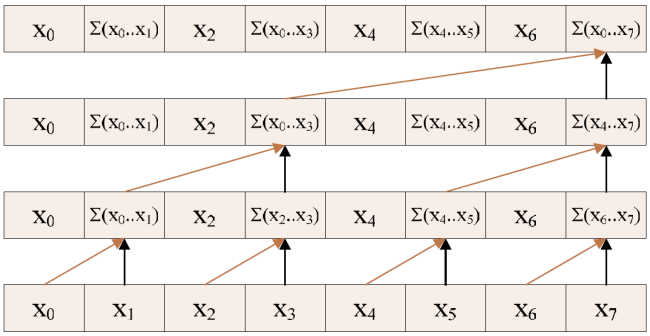


Down-Sweep



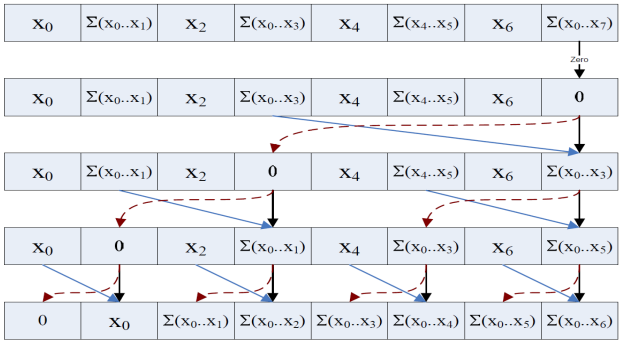
Up-Sweep Pseudo-Code

```
for d := 0 to log2n - 1 do
    for k from 0 to n – 1 by 2^(d+1) in parallel do
        x[k + 2^(d+1) - 1] := x[k + 2^d - 1] + x [k + 2^(d+1) - 1]
```



Down-Sweep Pseudo-Code

```
x[n - 1] := 0
for d := log2n down to 0 do
    for k from 0 to n – 1 by 2^(d+1) in parallel do
        t := x[k + 2^d - 1]
        x[k + 2^d - 1] := x [k + 2^(d+1) - 1]
        x[k + 2^(d+1) - 1] := t + x [k + 2^(d+1) - 1]
```



CUDA Implementation

```
__global__ void
scan(float *g_odata, float *g_idata, int n)
{
  extern __shared__ float temp[];

  int thid = threadIdx.x;
  int offset = 1;

  // load input into shared memory
  temp[2*thid] = g_idata[2*thid];
  temp[2*thid+1] = g_idata[2*thid+1];

...
```

CUDA Implementation: Up Sweep

```
// up sweep: essentially a reduction
for (int d = 1; d < (n >> 1); d <<=1) {
  int index  = 2 * d * thid;

  if (index < (n >> 1)) {
      // note that the result is supposed
      // to be stored in the last element
      // of the array
      temp[index + d] += temp[index];
  }
  __syncthreads();
}

…
```


CUDA Implementation: Down-Sweep

```
// clear the last element
if (thid == 0) { temp[n - 1] = 0; }

// traverse down tree & build scan
for (int d = (n >> 1) ; d > 0; d >>= 1) {
  int index = 2 * d * thid;

  if (index < (n >> 1)) {
      float t = temp[index + d];
      temp[index + d] += temp[index];
      temp[index] = t;
  }
  __syncthreads();
}

…
```

CUDA Implementation: Copy Back

```
  __syncthreads();

  // all threads write results to global memory
  g_odata[2*thid]   = temp[2*thid];
  g_odata[2*thid+1] = temp[2*thid+1];

} // end of the scan kernel
```


Bank Conflicts

- Current scan implementation has many shared memory bank conflicts
  - These really hurt performance on hardware
- Occur when multiple threads access the same shared memory bank with different addresses
- No penalty if all threads access different banks
  - Or if all threads access exact same address
- Access costs 2*M cycles if there is a conflict
  - Where M is max number of threads accessing single bank

Loading from Global Memory to Shared

- Original code interleaves loads:

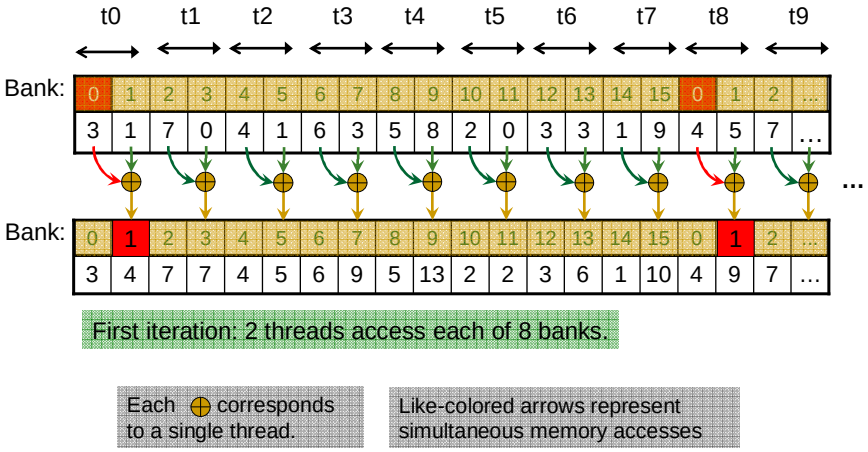
```
temp[2*thid]   = g_idata[2*thid];
temp[2*thid+1] = g_idata[2*thid+1];
```

- Threads:(0,1,2,…,8,9,10,…)→
  - banks:(0,2,4,…,0,2,4,…)
  - banks:(1,3,5,…,1,3,5,…)

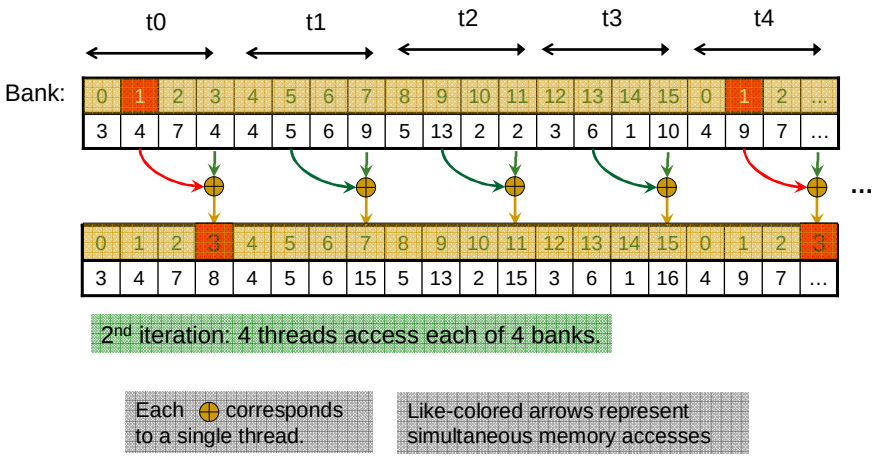
- Better to load one element from each half of the array

```
temp[thid]          = g_idata[thid];
temp[thid + (n/2)] = g_idata[thid + (n/2)];
```


Bank Conflicts in Up-Sweep

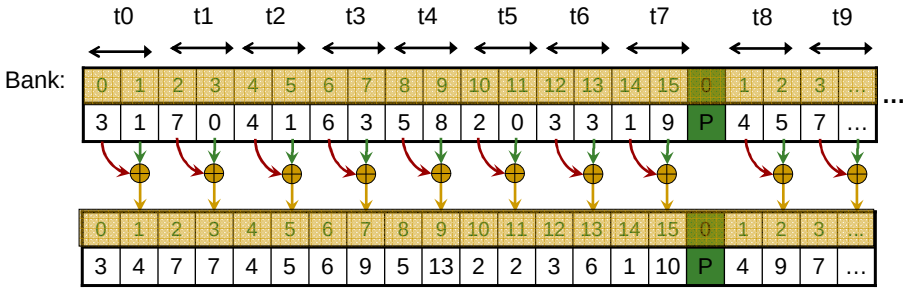


Bank Conflicts in Up-Sweep



Using Padding to Prevent Conflicts

- Just add a word of padding every 16 words:



Using Padding to Remove Conflicts

- Add padding

```
const int LOG_NUM_BANKS = 4;
...
address += (address >> LOG_NUM_BANKS);
```

- This removes most bank conflicts (why?)
  - Not all, in the case of deep trees

Padding in Up-Sweep

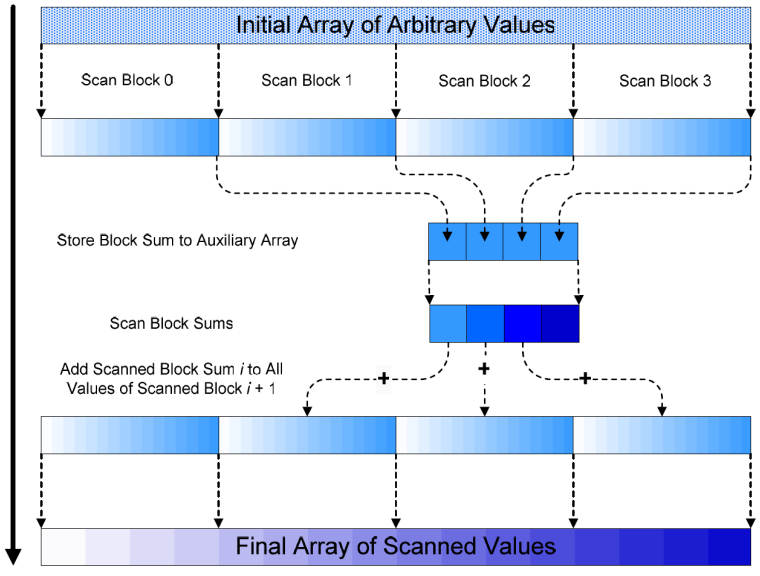
```
// up sweep: essentially a reduction
for (int d = 1; d < (n >> 1); d <<=1) {
   int index  = 2 * d * thid;

   if (index < (n >> 1)) {
       // note that the result is supposed
       // to be stored in the last element
       // of the array
       int addr1 = index;
       int addr2 = index + d;
       addr1 += addr1 >> LOG_NUM_BANKS;
       addr2 += addr2 >> LOG_NUM_BANKS;
       temp[addr2] += temp[addr1];
   }
   __syncthreads();
}
...
```

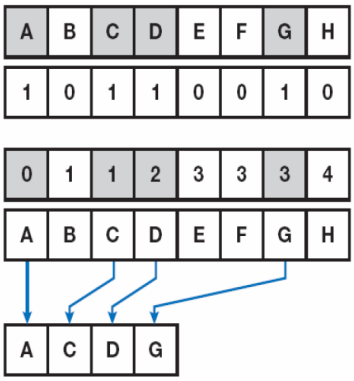
Large Arrays

- So far:
  - Array can be processed by a block
    - 1024 elements
- Larger arrays?
  - Divide into blocks
  - Scan each with a block of threads
  - Produce partial scans
  - Scan the partial scans
  - Add the corresponding scan result back to all elements of each block
- See Scan Large Array in the NVIDIA_CUDA_SDK

Large Arrays



Application: Stream Compaction



Input: we want to preserve the gray elements

Set a "1" in each gray input

Scan

Scatter input to output, using scan result as scatter address

1M elements: ~0.6-1.3ms

16M elements: ~8-20ms

Perf depends on # elements retained