CS 380 - GPU and GPGPU Programming
Lecture 24: GPU Parallel Prefix Sum / Scan

Markus Hadwiger, KAUST

Reading Assignment #13 (until Nov 30)

Read (required):

- Programming Massively Parallel Processors book, 3rd edition
  Chapter 9 (Parallel patterns – parallel histogram computation)
- Programming Massively Parallel Processors book, 3rd edition
  Chapter 13 (CUDA dynamic parallelism)

Read (optional):

- Prefix Sums and Their Applications, Guy Blelloch

https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf

GPU Parallel Prefix Sum

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

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

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

- **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]

(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
  - Range Histograms
  - Etc.

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: optimal in terms of work efficiency**

Prefix Sum Application
- Compaction -

Parallel Data Compaction

- **Given an array of marked values**

3	1	7	4	2	1	5	6	3	1
1	0	1	0	0	0	0	1	0	0

- **Output the compacted list of marked values**

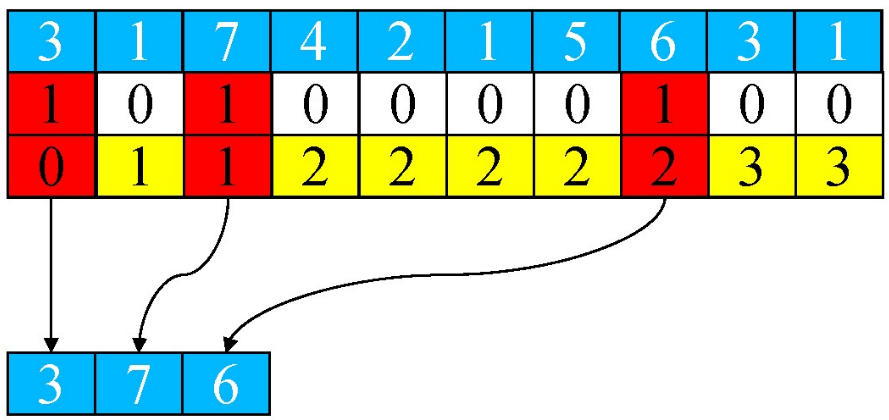
3	7	6

Using Prefix Sum

- **Calculate prefix sum on index array**

3	1	7	4	2	1	5	6	3	1
1	0	1	0	0	0	0	1	0	0
0	1	1	2	2	2	2	2	3	3

- **For each marked value lookup the destination index in the prefix sum**



- **Parallel write to separate destination elements**

Prefix Sum Application
- Range Histogram -

Range Histogram

- **A histogram calculate the occurance of each value in an array.**

  $$h[i] = |J| \quad J=\{j|\ v[j] = i\}$$

- **Range query: number over elements in interval [a,b].**

- **Slow answer:**

```
hrange = 0;
for (i = a; i<=b; ++i)
    hrange += h[i];
```

Fast Range Histogram

- **Compute prefix sum of histogram**
- **Fast answer:**

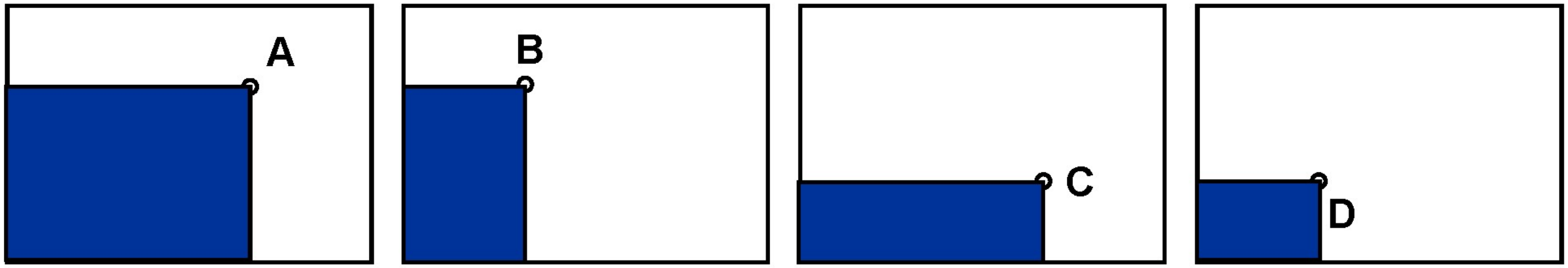
```
hrange = pref[B] - pref[A];
```

$$= \sum_{0}^{B} h[i] - \sum_{0}^{A} h[i] = \sum_{A}^{B} h[i]$$

Prefix Sum Application
- Summed Area Tables -

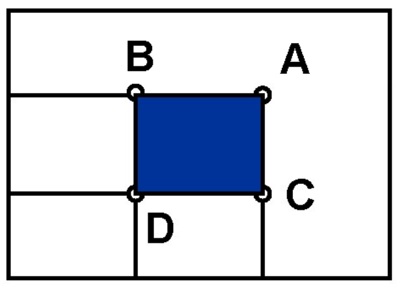
Summed Area Tables

- Per texel, store sum from (0, 0) to (u, v)



- Many bits per texel (sum !)
- Evaluation of 2D integrals in constant time!

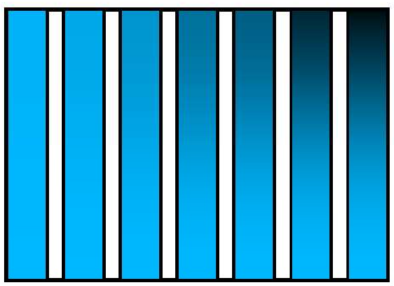
$$\int_{Bx}^{Ax}\int_{Cy}^{Ay} I(x,y)\,dx\,dy = A - B - C + D$$



Summed Area Table with Prefix Sums

- One possible way:
- Compute prefix sum horizontally

- … then vertically on the result



Work Efficiency

Guy E. Blelloch and Bruce M. Maggs:
Parallel Algorithms, 2004 (https://www.cs.cmu.edu/~guyb/papers/BM04.pdf)

In designing a parallel algorithm, it is more important to make it efficient than to make it asymptotically fast. The efficiency of an algorithm is determined by the total number of operations, or work that it performs. On a sequential machine, an algorithm's work is the same as its time. On a parallel machine, the work is simply the processor-time product. Hence, an algorithm that takes time t on a P-processor machine performs work W = Pt. In either case, the work roughly captures the actual cost to perform the computation, assuming that the cost of a parallel machine is proportional to the number of processors in the machine.

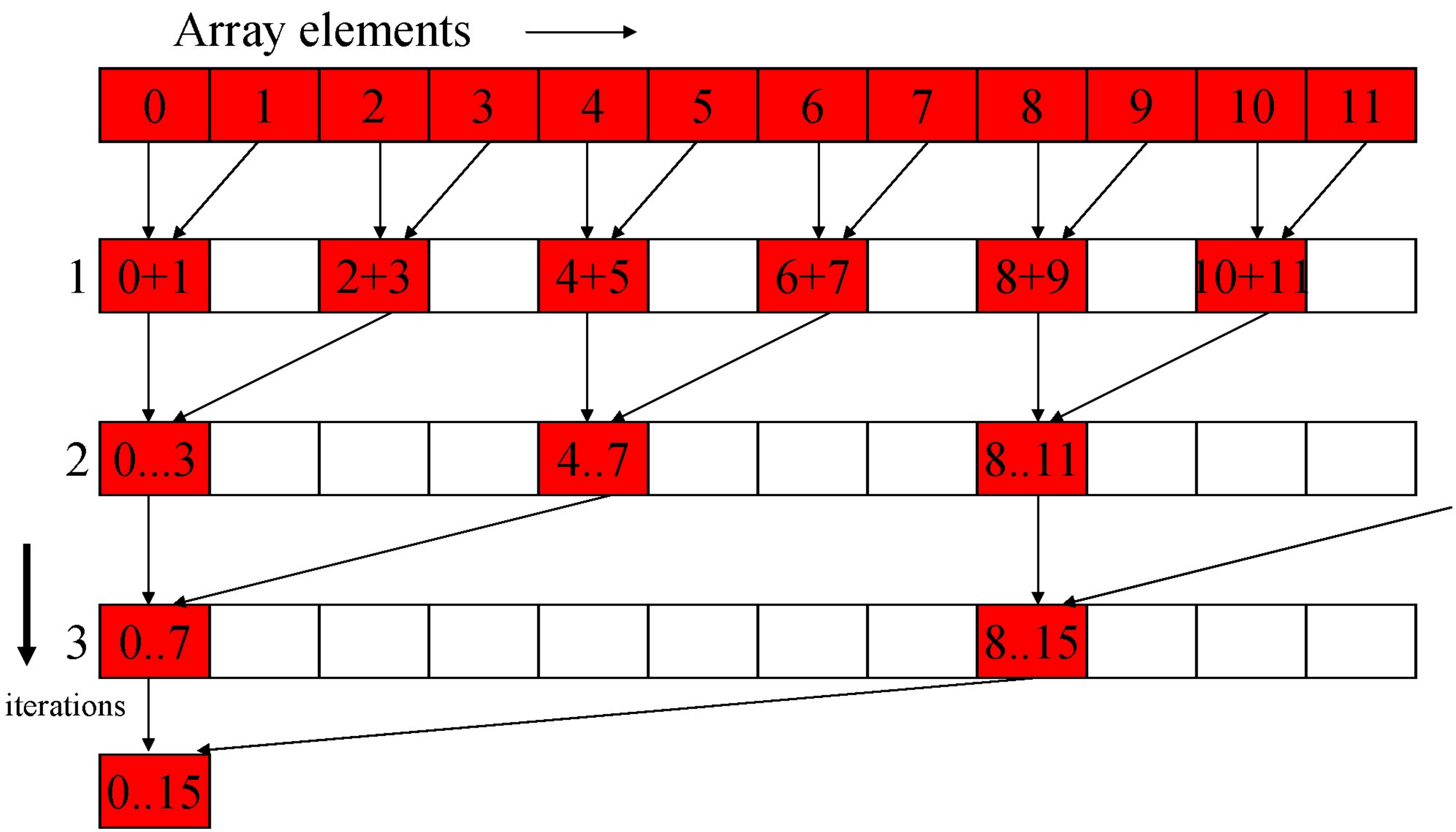
We call an algorithm work-efficient (or just efficient) if it performs the same amount of work, to within a constant factor, as the fastest known sequential algorithm.

For example, a parallel algorithm that sorts n keys in O(sqrt(n) log(n)) time using sqrt(n) processors is efficient since the work, O(n log(n)), is as good as any (comparison-based) sequential algorithm.

However, a sorting algorithm that runs in O(log(n)) time using n^2 processors is not efficient.

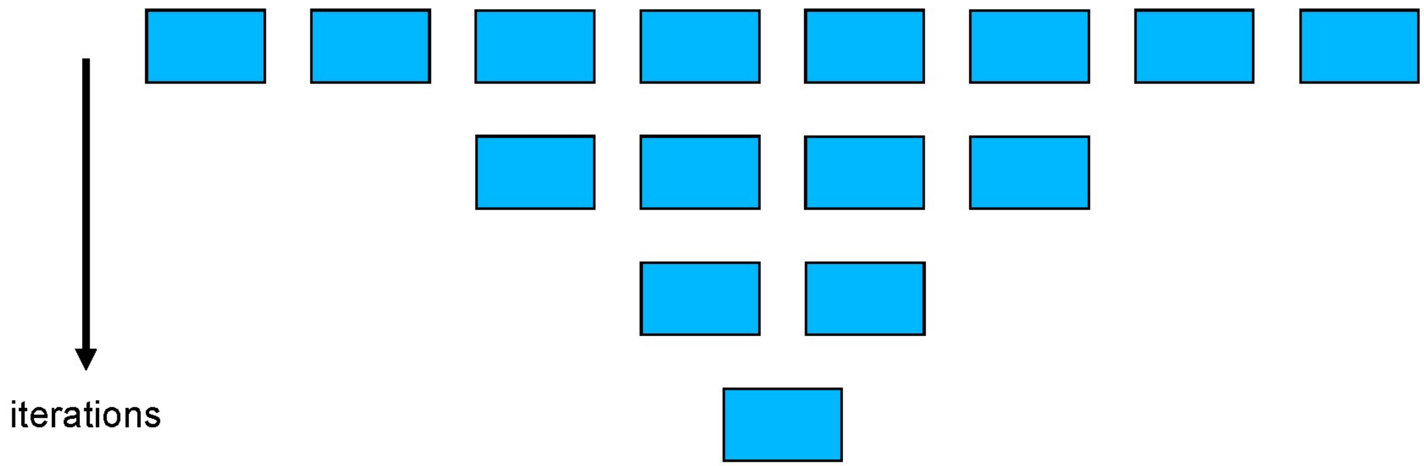
The first algorithm is better than the second - even though it is slower - because its work, or cost, is smaller. Of course, given two parallel algorithms that perform the same amount of work, the faster one is generally better.

Vector Reduction



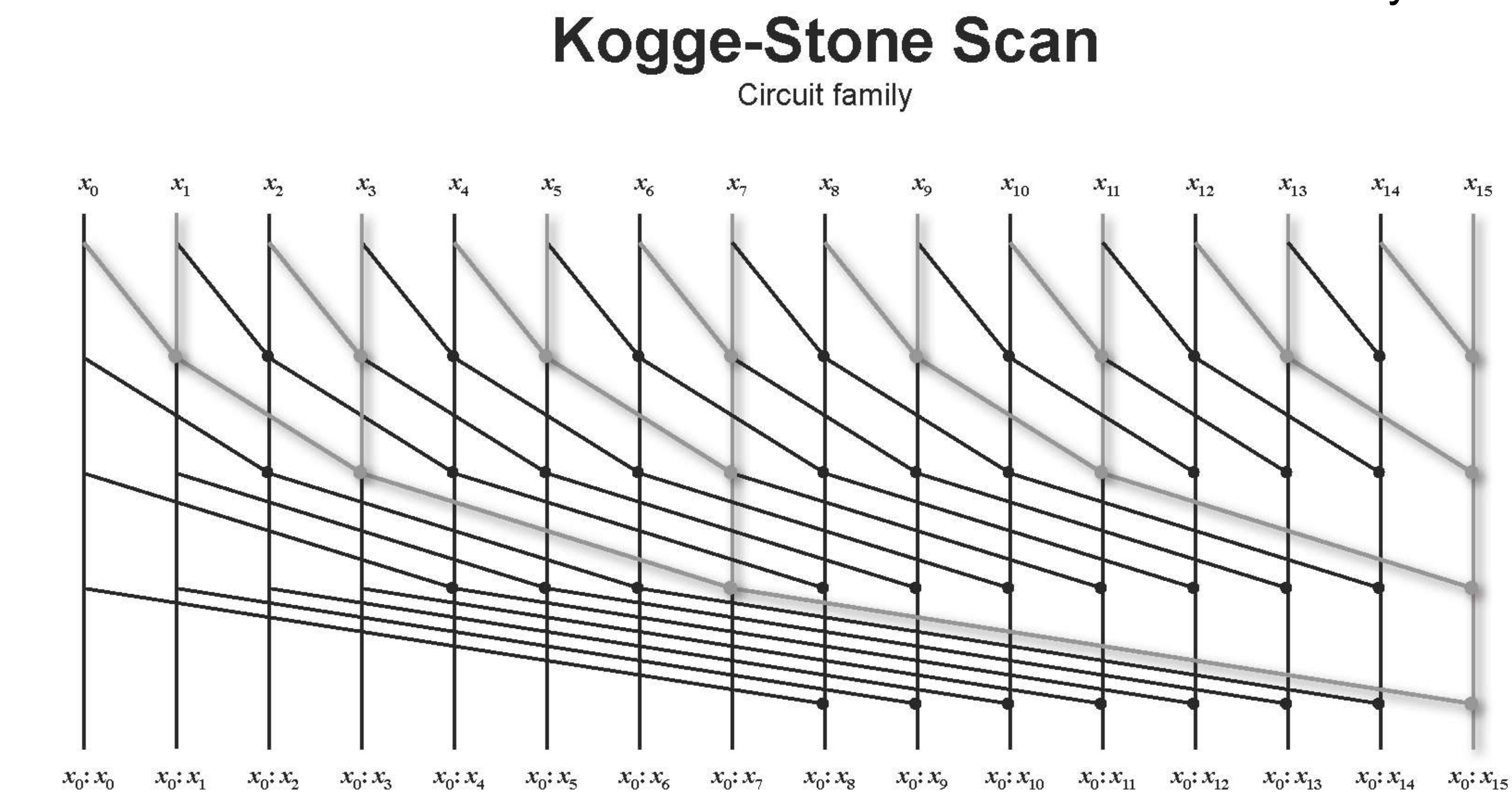
Typical Parallel Programming Pattern

- **log(n) steps**



Helpful fact for counting nodes of full binary trees:
If there are N leaf nodes, there will be N-1 non-leaf nodes

Courtesy John Owens

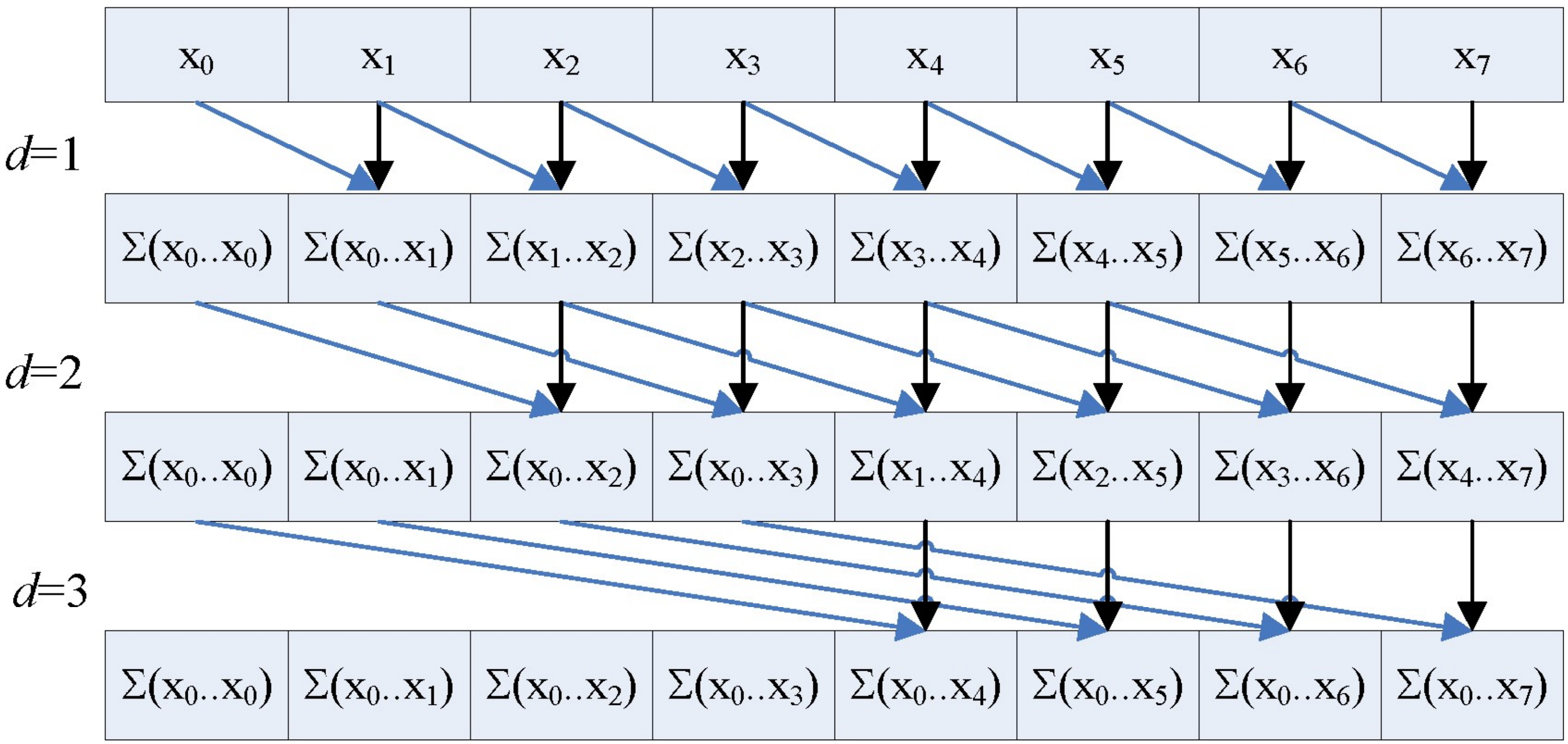


A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations, Kogge and Stone, 1973

See “carry lookahead” adders vs. “ripple carry” adders

$O(n \log n)$ Scan

Courtesy John Owens

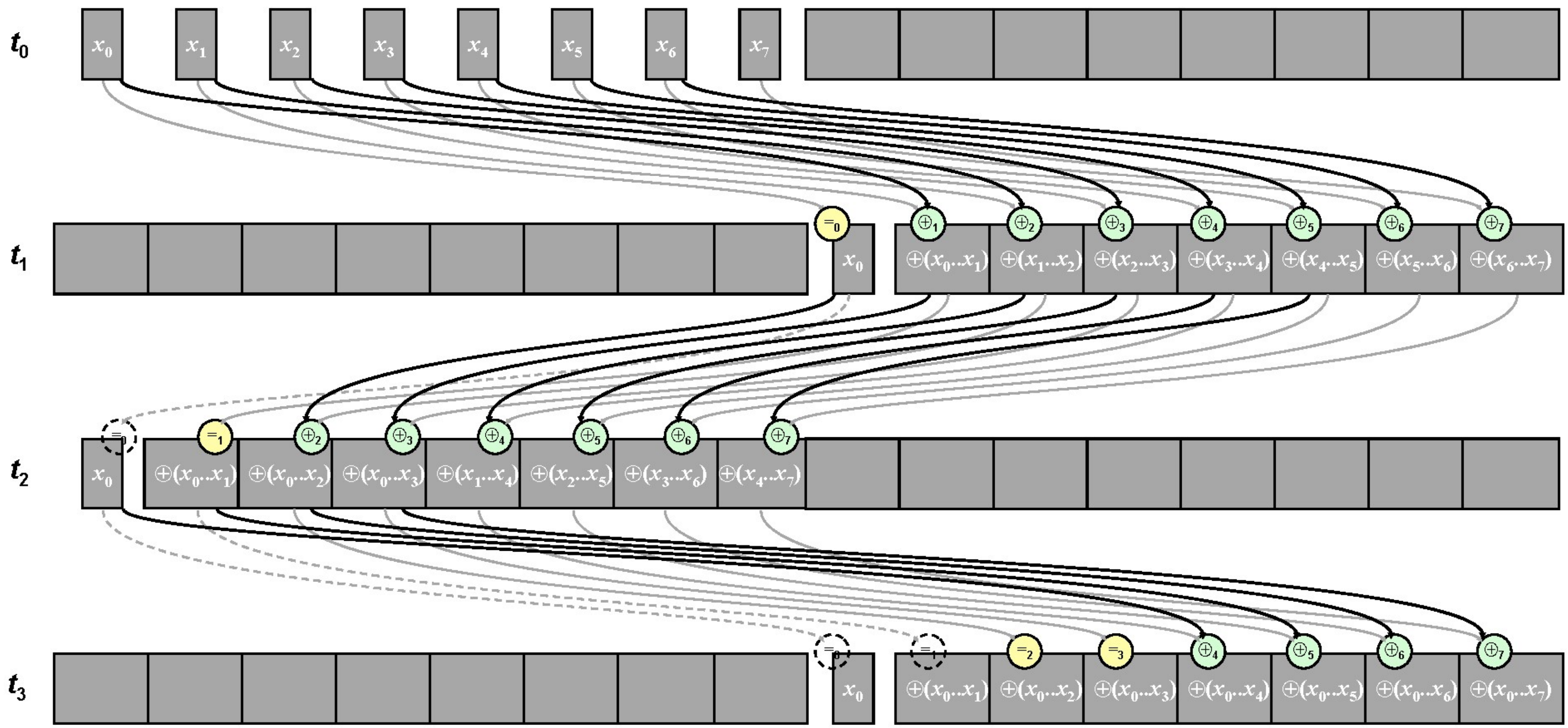


- Step efficient ($\log n$ steps)
- Not work efficient ($n \log n$ work)
- Requires barriers at each step (WAR dependencies)

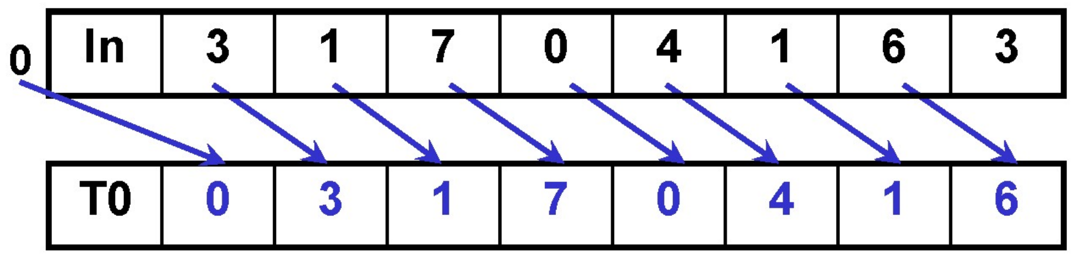
Courtesy John Owens

Hillis-Steele Scan Implementation

No WAR conflicts, $O(2N)$ storage



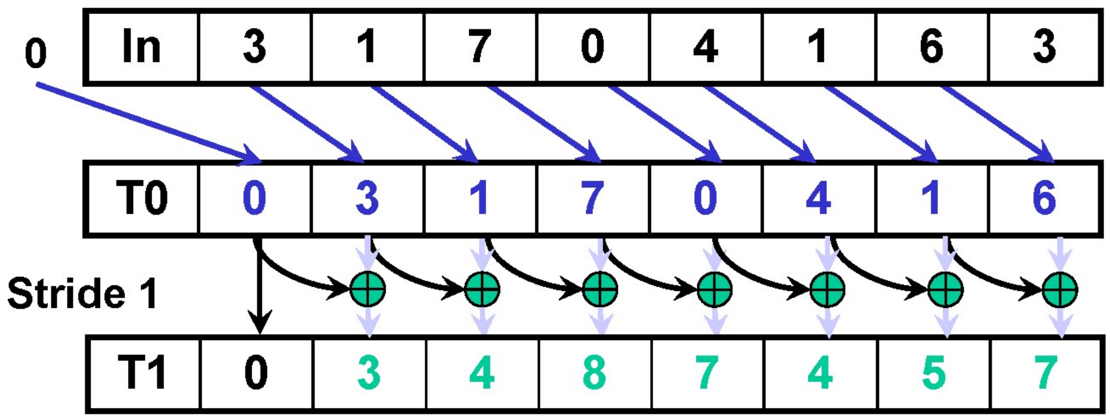
A First-Attempt Parallel Scan Algorithm



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

A First-Attempt Parallel Scan Algorithm

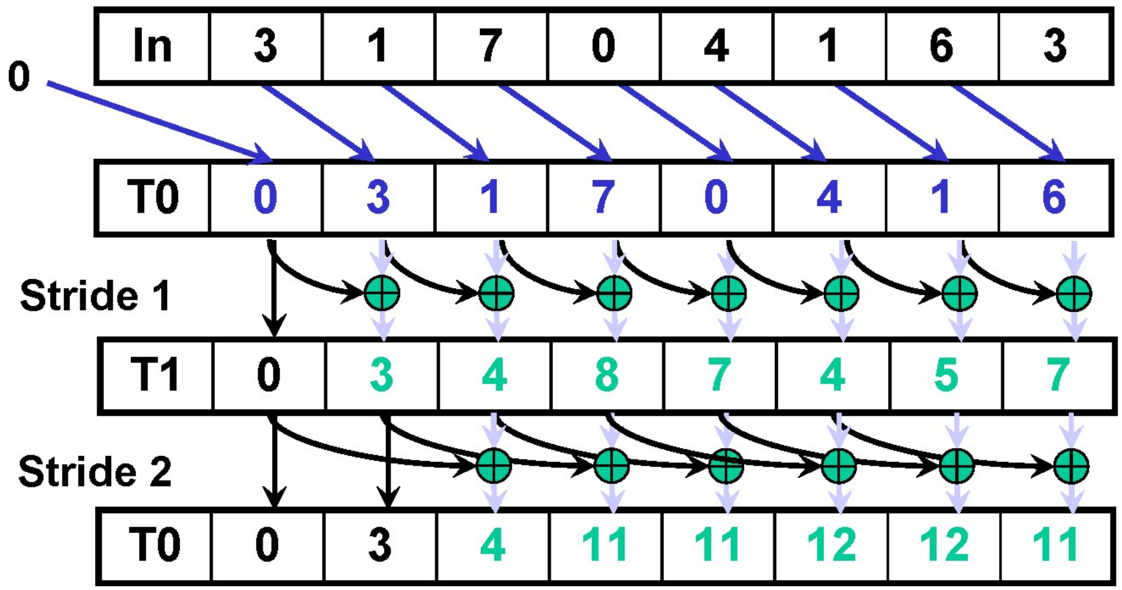


1. (previous slide)
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #1
Stride = 1

• Active threads: *stride* to *n*-1 (*n-stride* threads)
• Thread *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

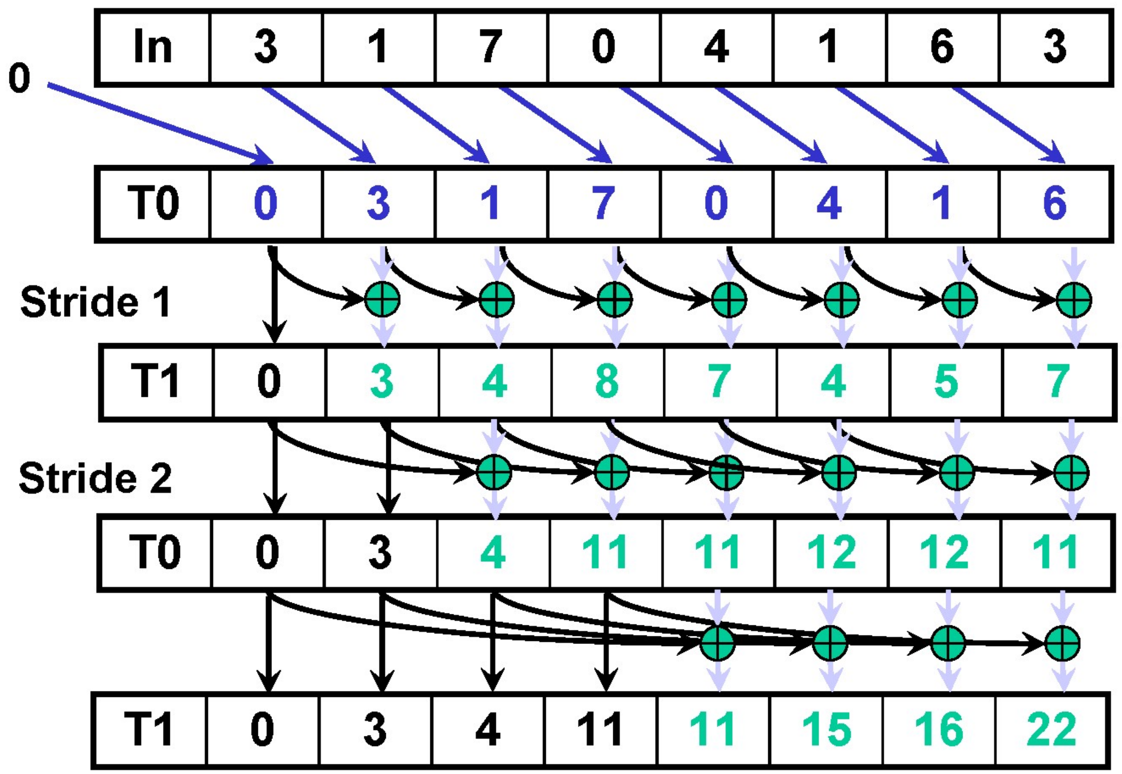


Iteration #2
Stride = 2

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. (note must double buffer shared mem arrays)

A First-Attempt Parallel Scan Algorithm

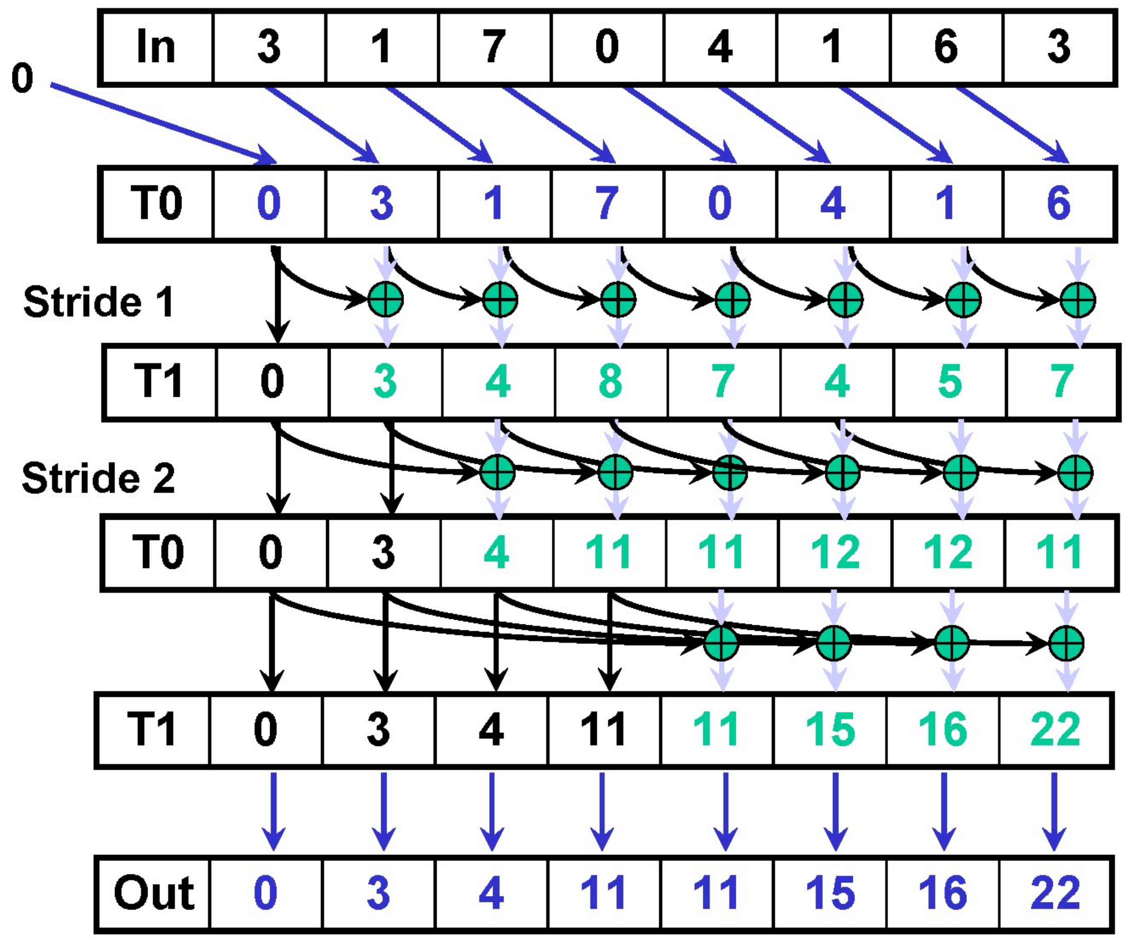


Iteration #3
Stride = 4

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. (note must double buffer shared mem arrays)

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. (note must double buffer shared mem arrays)
3. Write output to device memory.

Work Efficiency Considerations

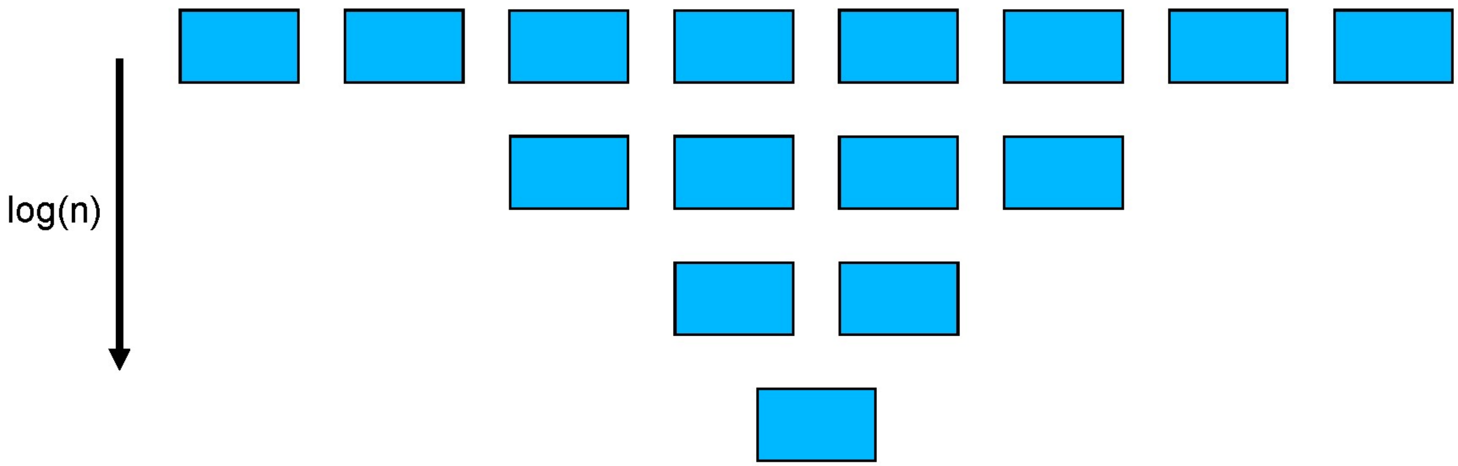
- **The first-attempt Scan executes log(n) parallel iterations**
  - Total adds: n * (log(n) – 1) + 1 → O(n*log(n)) work
- **This scan algorithm is not very work efficient**
  - Sequential scan algorithm does *n* adds
  - A factor of log(n) hurts: 20x for 10^6 elements!
- **A parallel algorithm can be slow when execution resources are saturated due to low work efficiency**

Balanced Trees

- **For improving efficiency**
- **A common parallel algorithm pattern:**
  - 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

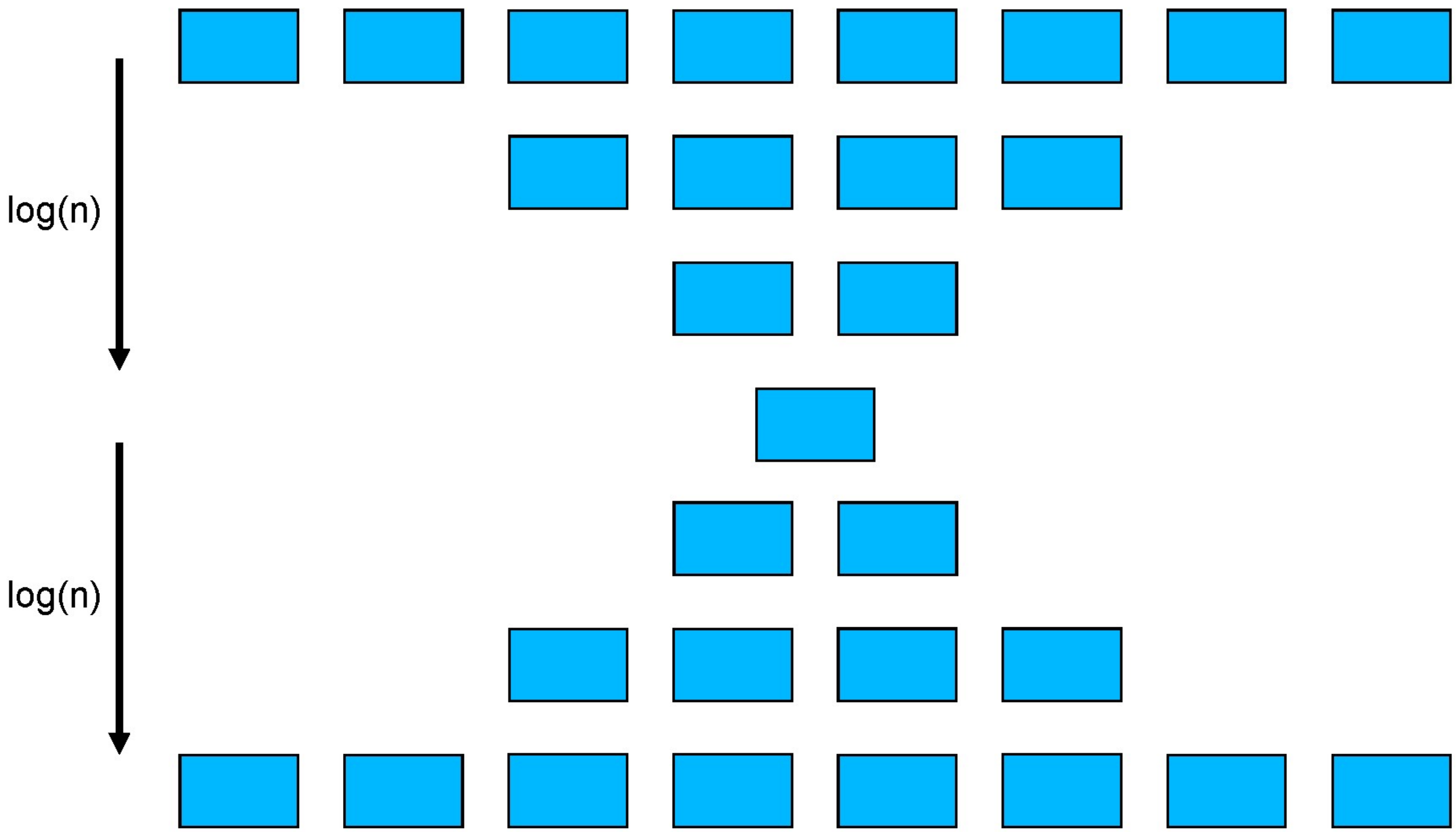
Typical Parallel Programming Pattern

- **2 log(n) steps**



Typical Parallel Programming Pattern

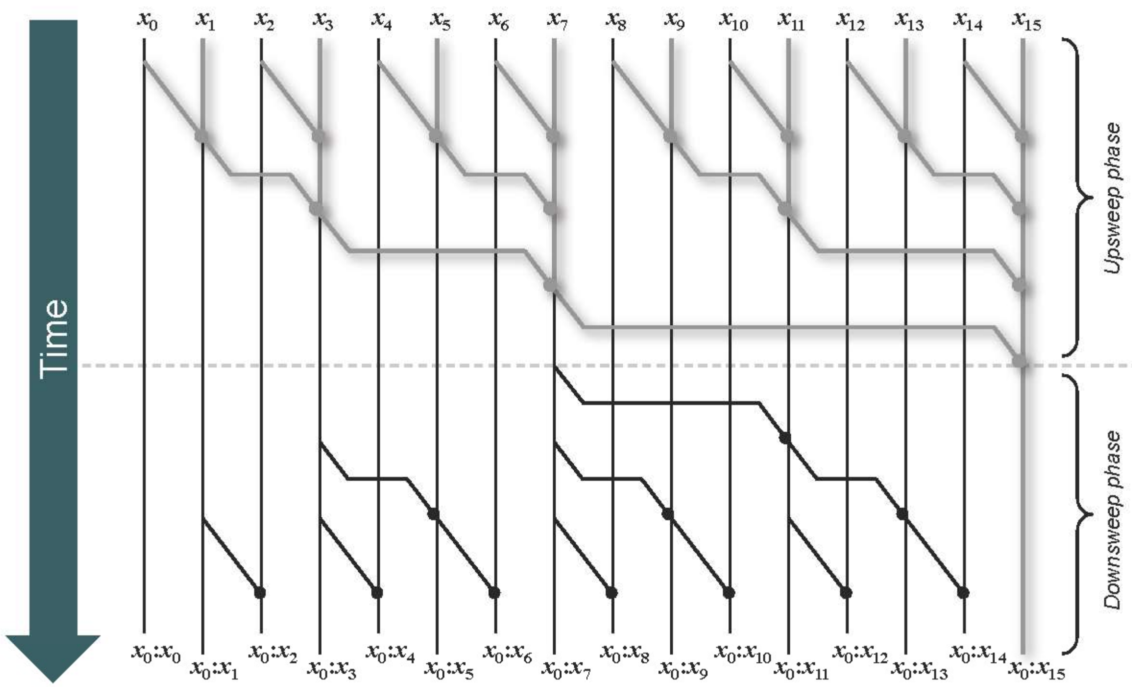
- **2 log(n) steps**



Courtesy John Owens

Brent Kung Scan

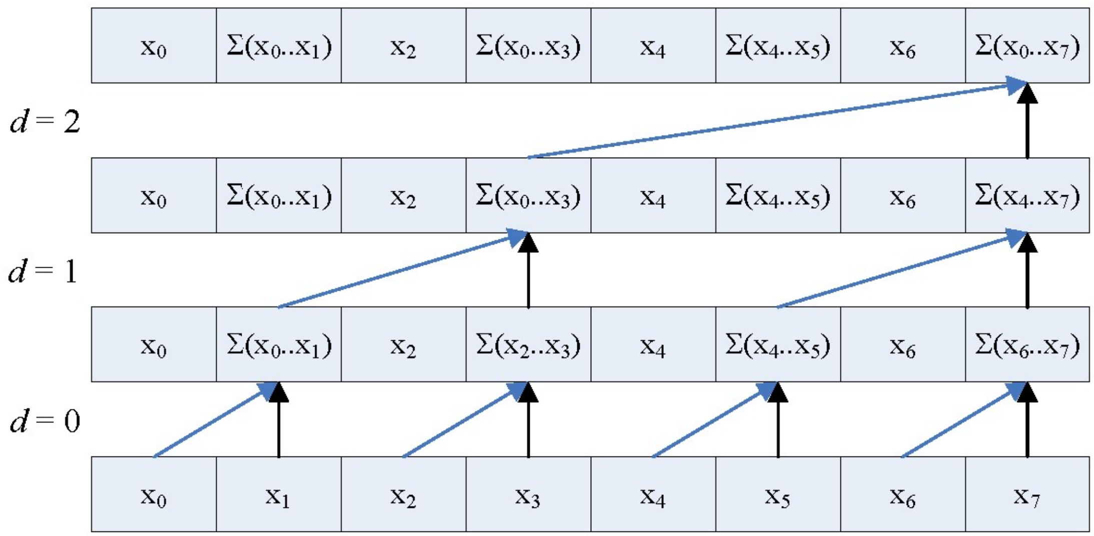
Circuit family

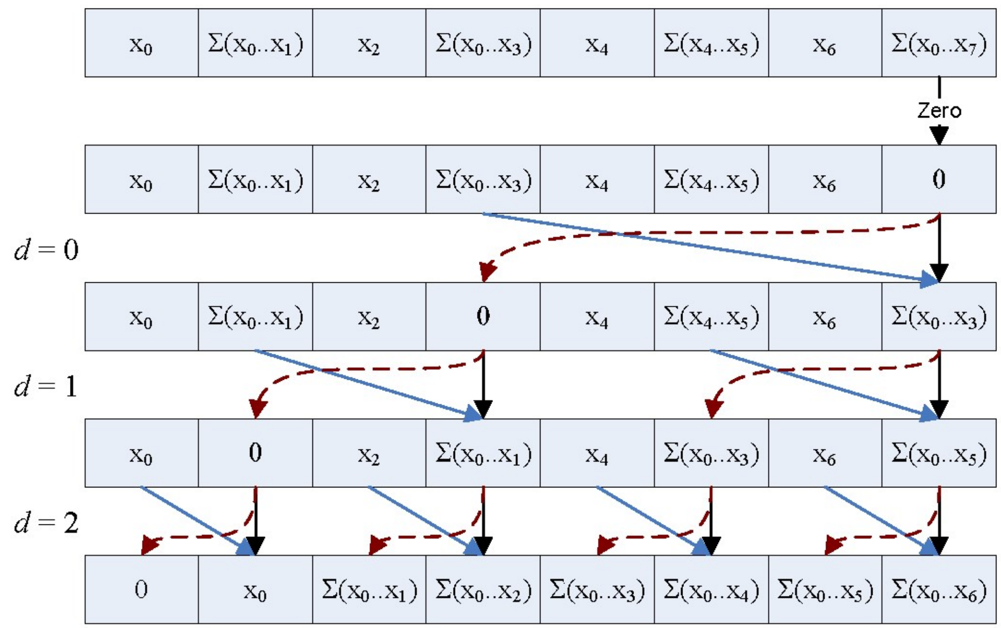


A Regular Layout for Parallel Adders, Brent and Kung, 1982

$O(n)$ Scan [Blelloch]

Courtesy John Owens





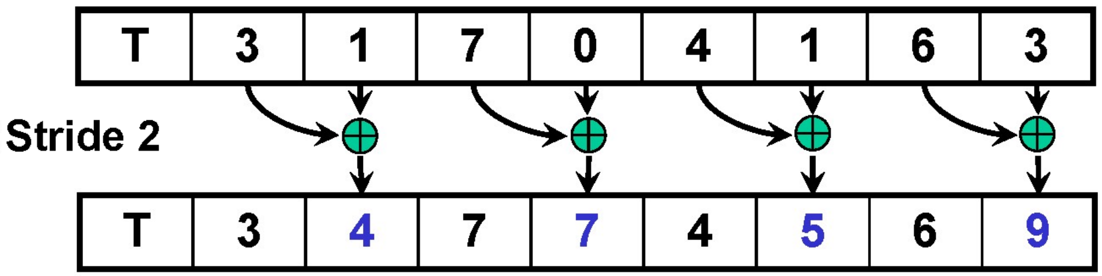
- Work efficient ($O(n)$ work)
- Bank conflicts, and lots of 'em

Build the Sum Tree

T	3	1	7	0	4	1	6	3

Assume array is already in shared memory

Build the Sum Tree

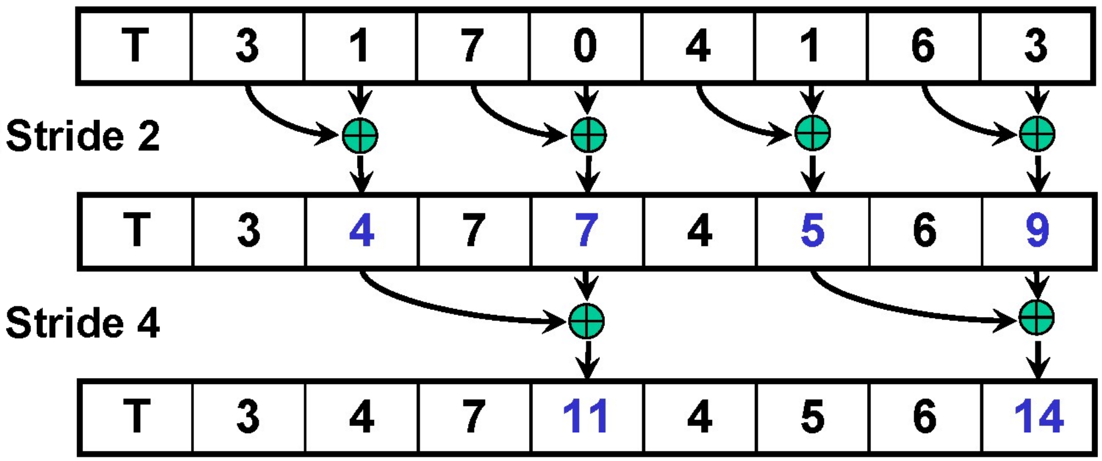


Iteration 1, *n*/2 threads

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* / 2 elements away to its own value.

Build the Sum Tree

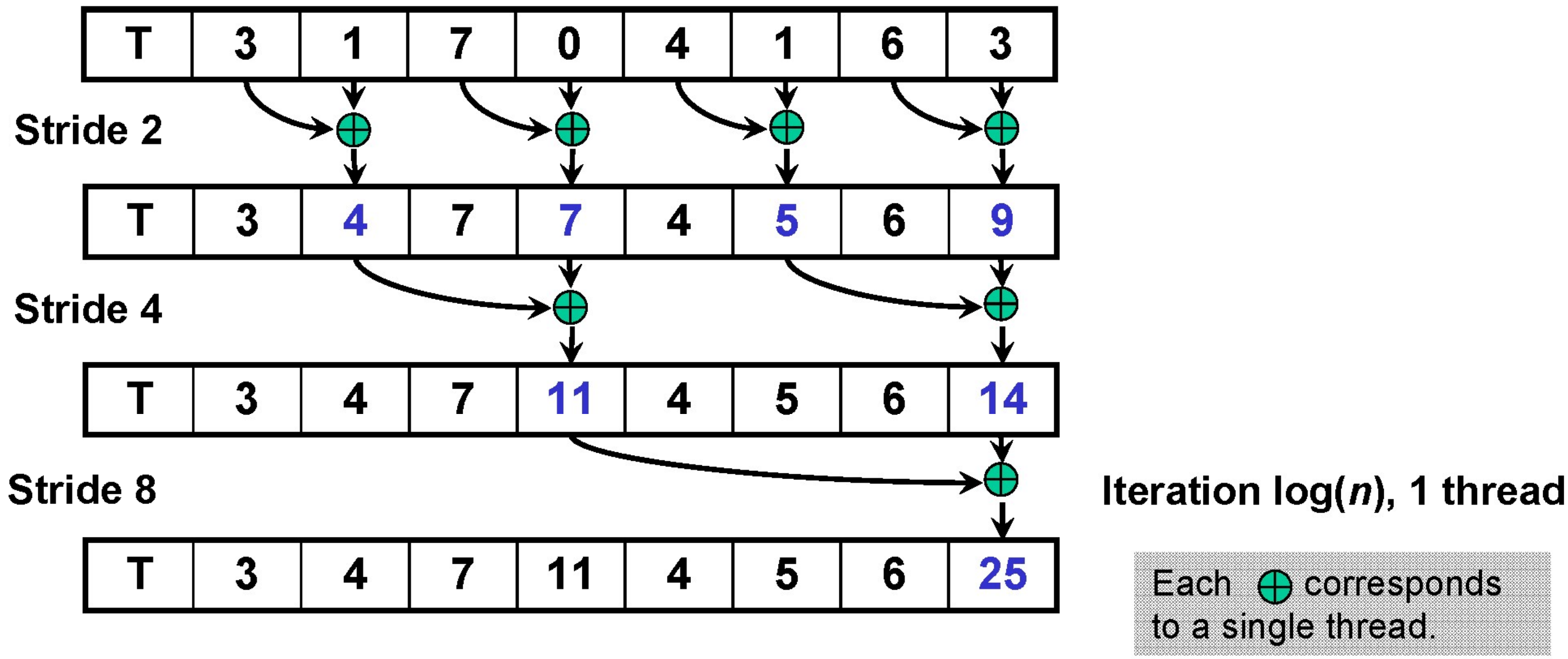


Iteration 2, *n*/4 threads

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* / 2 elements away to its own value.

Build the Sum Tree



Iterate log(n) times. Each thread adds value *stride* / 2 elements away to its own value.

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

Down-Sweep Variant 1: Exclusive Scan

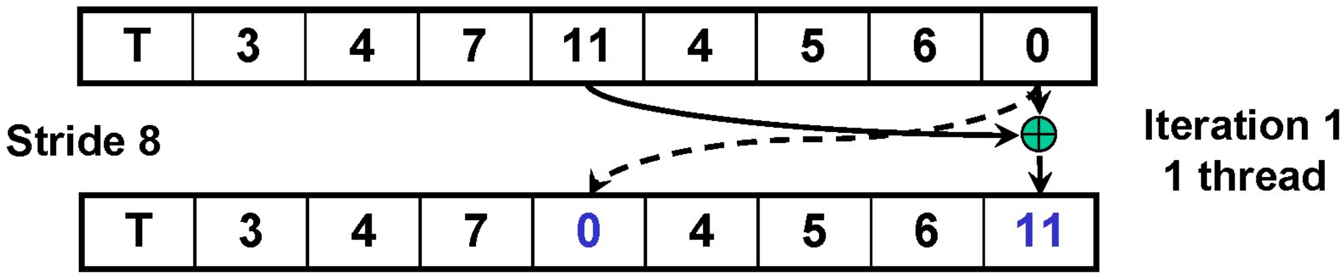
T	3	4	7	11	4	5	6	0

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

T	3	4	7	11	4	5	6	0

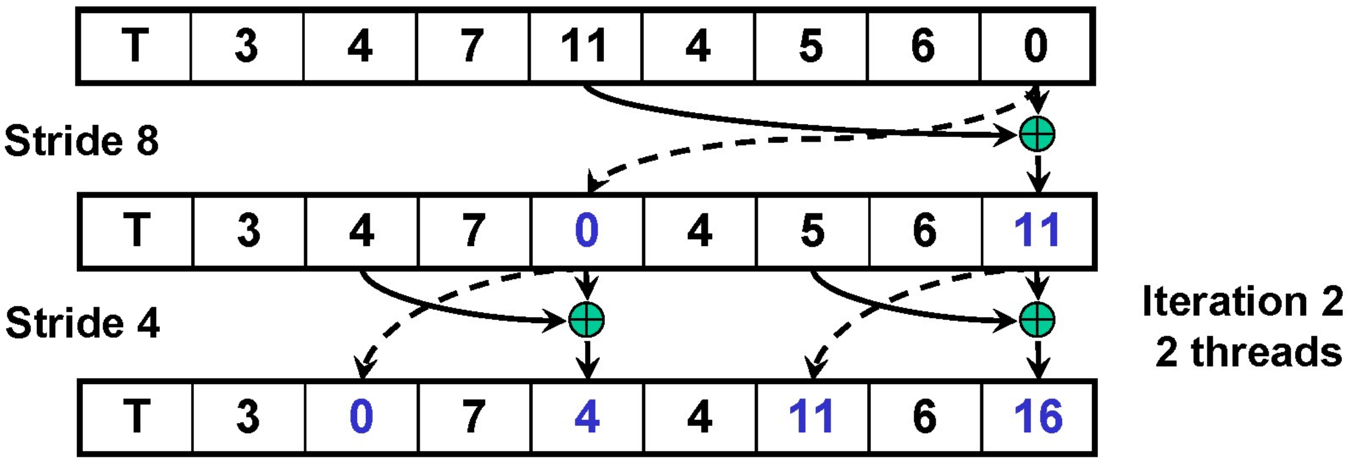
Build Scan From Partial Sums



Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* / 2 elements away to its own value. and sets the value *stride* elements away to its own *previous* value.

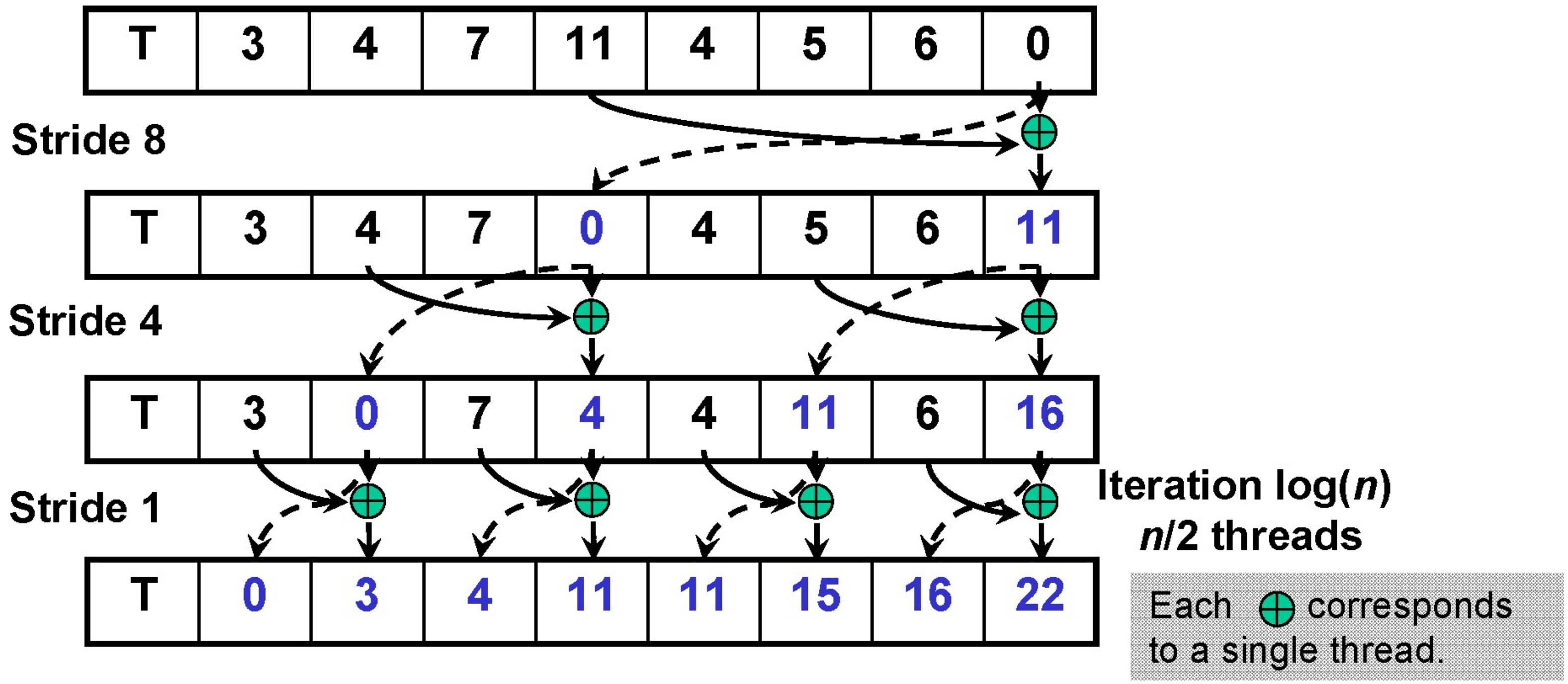
Build Scan From Partial Sums



Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* / 2 elements away to its own value. and sets the value *stride* / 2 elements away to its own *previous* value.

Build Scan From Partial Sums



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

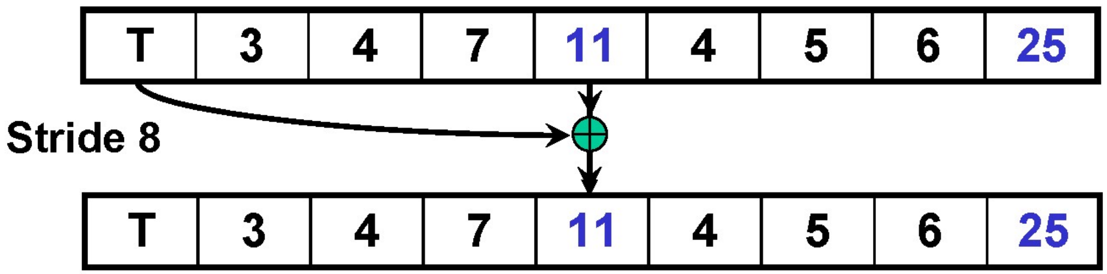
Total steps: 2 * log(*n*).
Total work: 2 * (*n*-1) adds = *O*(*n*) **Work Efficient!**

Down-Sweep Variant 2: Inlusive Scan

T	3	4	7	11	4	5	6	25

We now have an array of partial sums. Let’s propagate the sums back.

Build Scan From Partial Sums

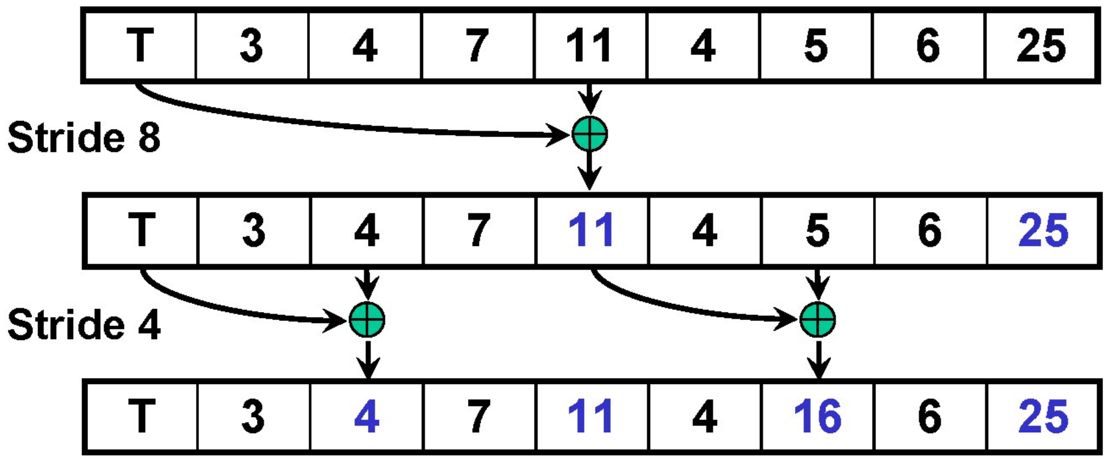


no operation

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* / 2 elements away to its own value. First element adds zero.

Build Scan From Partial Sums

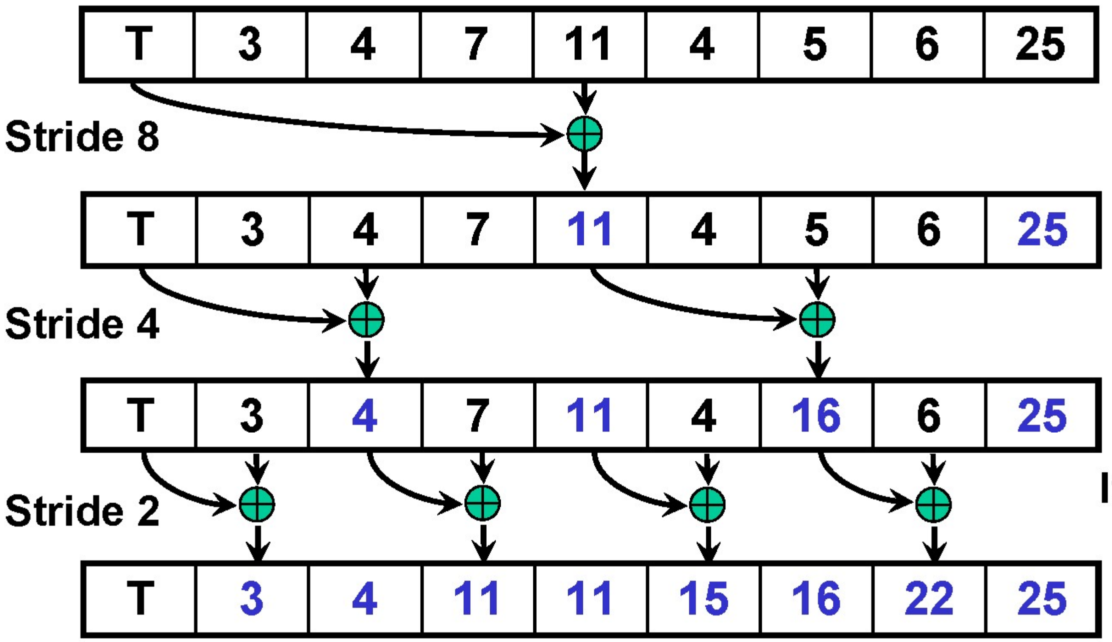


Iteration 2
2 threads

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* / 2 elements away to its own value. First element adds zero.

Build Scan From Partial Sums



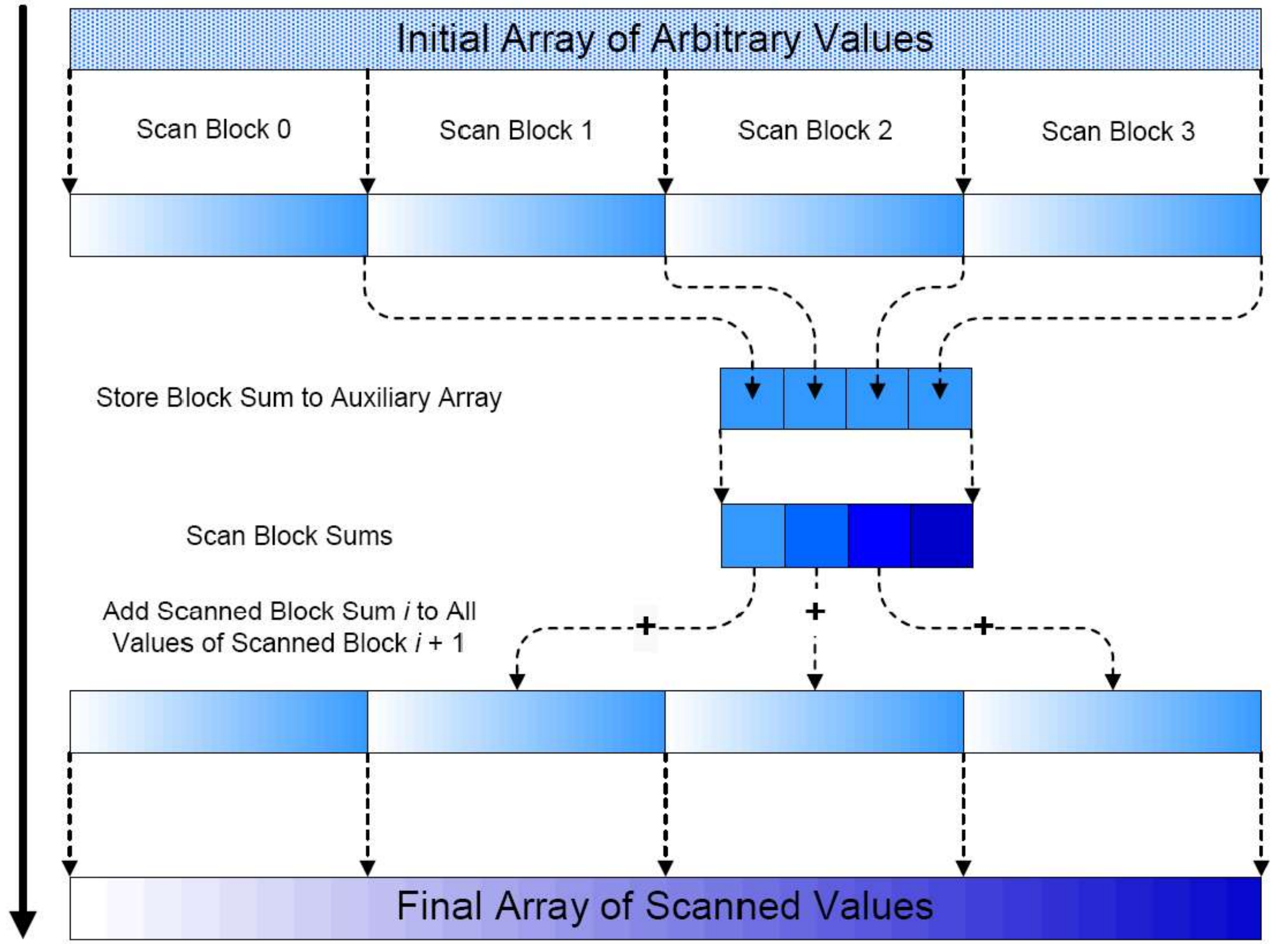
Iteration log(n)
n/2 threads

Each ⊕ corresponds to a single thread.

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!**

Application to Large Arrays



[Mark Harris]

Scan papers

Courtesy John Owens

- Daniel Horn, Stream Reduction Operations for GPGPU Applications, GPU Gems 2, Chapter 36, pp. 573–589, March 2005.
- Shubhabrata Sengupta, Aaron E. Lefohn, and John D. Owens. A Work-Efficient Step-Efficient Prefix Sum Algorithm. In Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures, pages D–26–27, May 2006
- Mark Harris, Shubhabrata Sengupta, and John D. Owens.Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, GPU Gems 3, chapter 39, pages 851–876. Addison Wesley, August 2007.
- Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In Graphics Hardware 2007, pages 97–106, August 2007.
- Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," in ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing, 2008, pp. 205–213.
- Shubhabrata Sengupta, Mark Harris, Michael Garland, and John D. Owens. Efficient Parallel Scan Algorithms for many-core GPUs. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, Scientific Computing with Multicore and Accelerators, Chapman & Hall/CRC Computational Science, chapter 19, pages 413–442. Taylor & Francis, January 2011.
- D. Merrill and A. Grimshaw, Parallel Scan for Stream Architectures. Technical Report CS2009-14, Department of Computer Science, University of Virginia, 2009, 54pp.
- Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: fast scan algorithms for GPUs without global barrier synchronization. In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13). ACM, New York, NY, USA, 229-238.

Bank Conflicts in Scan
- Non-power-of-two -

Initial Bank Conflicts on Load

- **Each thread loads two shared mem data elements**
- **Tempting to interleave the loads**

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

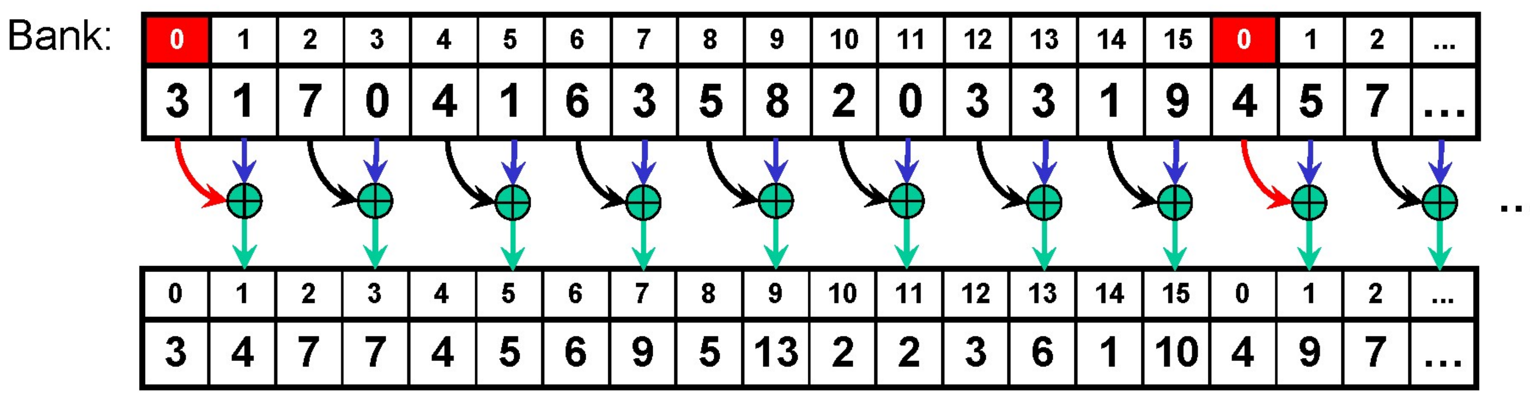
- **Threads:(0,1,2,…,8,9,10,…)→banks:(0,2,4,…,0,2,4,…)**

- **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 the tree algorithm

- When we build the sums, each thread reads two shared memory locations and writes one:
- Th(0,8) access bank 0



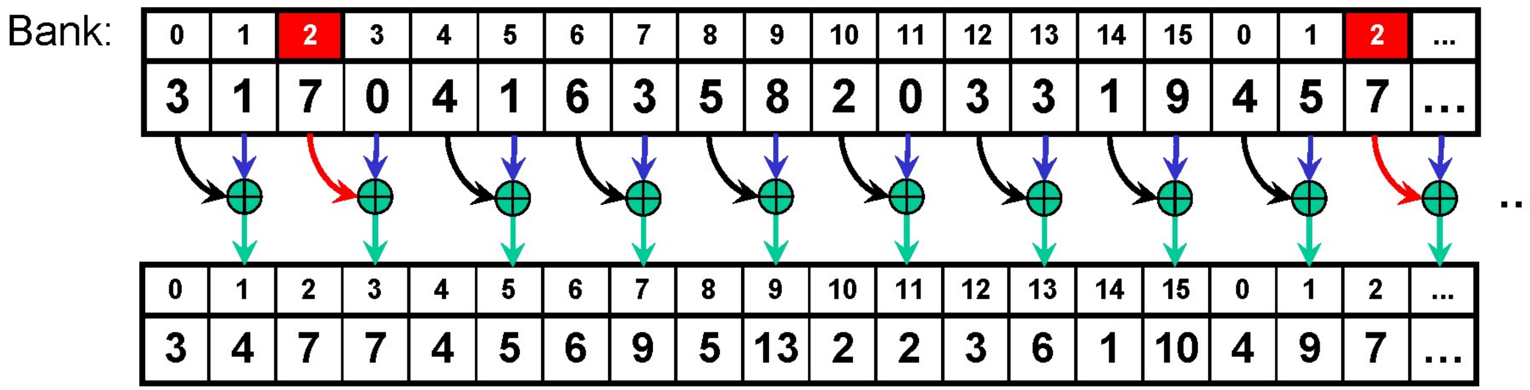
First iteration: 2 threads access each of 8 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Bank Conflicts in the tree algorithm

- When we build the sums, each thread reads two shared memory locations and writes one:
- Th(1,9) access bank 2, etc.



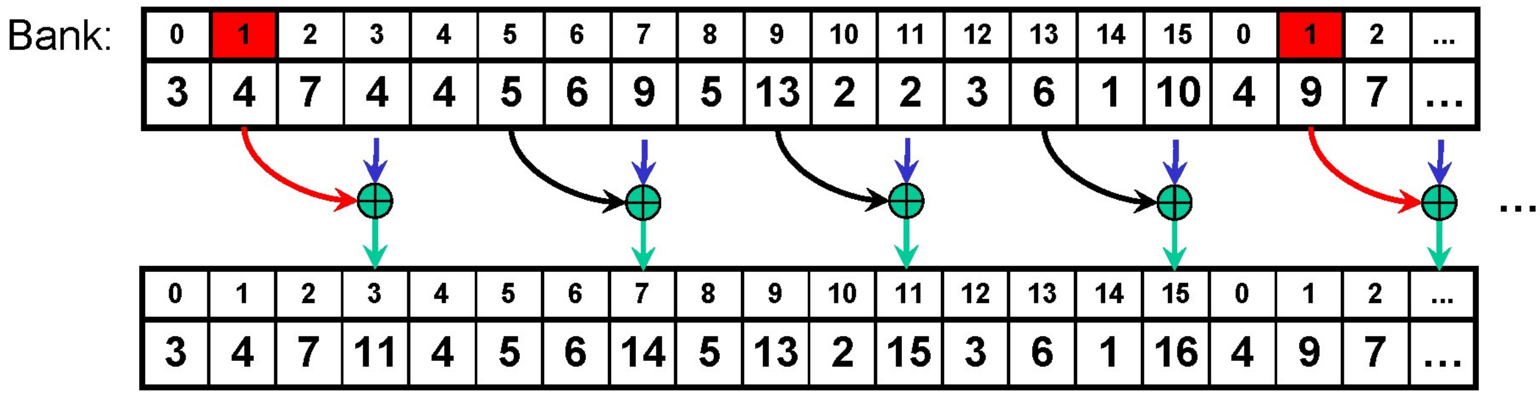
First iteration: 2 threads access each of 8 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Bank Conflicts in the tree algorithm

- **2nd iteration: even worse!**
  - 4-way bank conflicts; for example:
    Th(0,4,8,12) access bank 1, Th(1,5,9,13) access Bank 5, etc.



2nd iteration: 4 threads access each of 4 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Scan Bank Conflicts (1)

- **A full binary tree with 64 leaf nodes:**

Scale (s)	Thread addresses																															
1	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62
2	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60																
4	0	8	16	24	32	40	48	56																								
8	0	16	32	48																												
16	0	32																														
32	0																															
Conflicts	Banks																															
2-way	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14
4-way	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12																
4-way	0	8	0	8	0	8	0	8																								
4-way	0	0	0	0																												
2-way	0	0																														
None	0																															

- **Multiple 2-and 4-way bank conflicts**
- **Shared memory cost for whole tree**
  - 1 32-thread warp = 6 cycles per thread w/o conflicts
    - Counting 2 shared mem reads and one write (s[a] += s[b])
  - 6 * (2+4+4+4+2+1) = 102 cycles
  - 36 cycles if there were no bank conflicts (6 * 6)

Scan Bank Conflicts (2)

- **It's much worse with bigger trees!**
- **A full binary tree with 128 leaf nodes**
  - Only the last 6 iterations shown (root and 5 levels below)

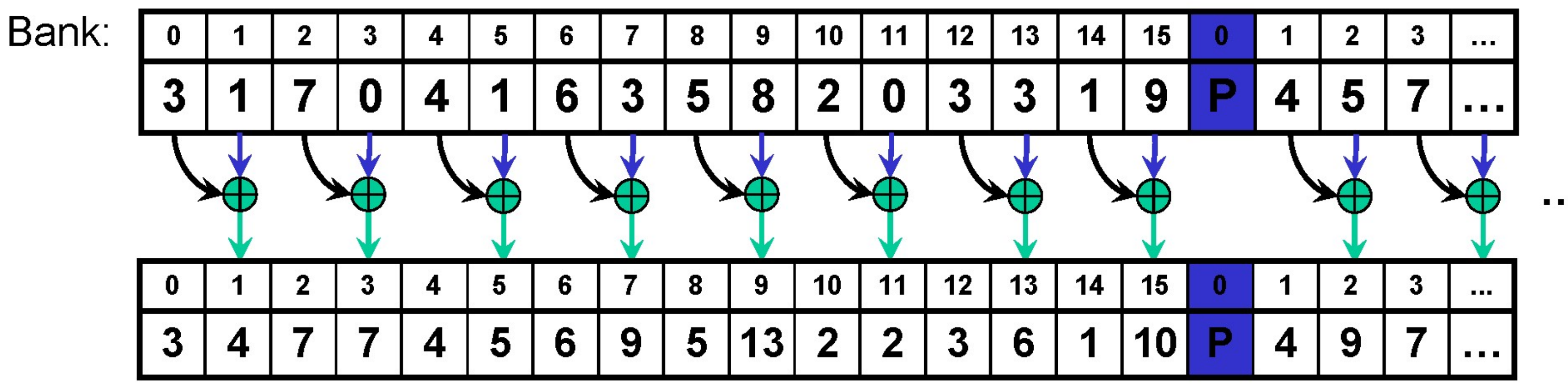
Scale (s)	Thread addresses																															
2	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	122
4	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120																
8	0	16	32	48	64	80	96	112																								
16	0	32	64	96																												
32	0	64																														
64	0																															
Conflicts	Banks																															
4-way	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	10
8-way	0	8	0	8	0	8	0	8	0	8	0	8	0	8	0	8																
8-way	0	0	0	0	0	0	0	0																								
4-way	0	0	0	0																												
2-way	0	0																														
None	0																															

- **Cost for whole tree:**
  - 12*2 + 6*(4+8+8+4+2+1) = 186 cycles
  - 48 cycles if there were no bank conflicts! 12*1 + (6*6)

Bank Conflicts in the tree algorithm

- **We can use padding to prevent bank conflicts**
  - Just add a word of padding every 16 words:
- **No more conflicts!**

32 for full warps!



Now, within a 16-thread half-warp, all threads access different banks.

32-thread full warp!

(Note that only arrows with the same color happen simultaneously.)

Use Padding to Reduce Conflicts

- **This is a simple modification to the last exercise**
- **After you compute a shared mem address like this:**

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

- **Add padding like this:**

```
Address += (Address >> 4); // divide by NUM_BANKS
```

- **This removes most bank conflicts**
  - Not all, in the case of deep trees

Fixing Scan Bank Conflicts

- **Insert padding every NUM_BANKS elements**

```
const int LOG_NUM_BANKS = 4; // 16 banks
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];
    }
}
```

Fixing Scan Bank Conflicts

- **A full binary tree with 64 leaf nodes**

Leaf Nodes	Scale (s)	Thread addresses																														
64	1	0	2	4	6	8	10	12	14	17	19	21	23	25	27	29	31	34	36	38	40	42	44	46	48	51	53	55	57	59	61	63
	2	0	4	8	12	17	21	25	29	34	38	42	46	51	55	59	63															
	4	0	8	17	25	34	42	51	59																							
	8	0	17	34	51																											
	16	0	34											= Padding inserted																		
	32	0																														
	Conflicts	Banks																														
	None	0	2	4	6	8	10	12	14	1	3	5	7	9	11	13	15	2	4	6	8	10	12	14	0	3	5	7	9	11	13	15
	None	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15															
	None	0	8	1	9	2	10	3	11																							
	None	0	1	2	3																											
	None	0	2																													
	None	0																														

- **No more bank conflicts!**
  - However, there are ~8 cycles overhead for addressing
    - For each s[a] += s[b] (8 cycles/iter. * 6 iter. = 48 extra cycles)
  - So just barely worth the overhead on a small tree
    - 84 cycles vs. 102 with conflicts vs. 36 optimal

Fixing Scan Bank Conflicts

- **A full binary tree with 128 leaf nodes**
  - Only the last 6 iterations shown (root and 5 levels below)

Scale (s)	Thread addresses																															
2	0	4	8	12	17	21	25	29	34	38	42	46	51	55	59	63	68	72	76	80	85	89	93	97	102	106	110	114	119	123	127	131
4	0	8	17	25	34	42	51	59	68	76	85	93	102	110	119	127																
8	0	17	34	51	68	85	102	119																								
16	0	34	68	102																												
32	0	68																														
64	0																															
Conflicts	Banks																															
None	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15	4	8	12	0	5	9	13	1	6	10	14	2	7	11	15	3
None	0	8	1	9	2	10	3	11	4	12	5	13	6	14	7	15																
None	0	1	2	3	4	5	6	7																								
None	0	2	4	6																												
None	0	4																														
None	0																															

= Padding inserted

- **No more bank conflicts!**
  - Significant performance win:
    - 106 cycles vs. 186 with bank conflicts vs. 48 optimal

Fixing Scan Bank Conflicts

- **A full binary tree with 512 leaf nodes**
  - Only the last 6 iterations shown (root and 5 levels below)

Scale (s)	Thread addresses																															
8	0	17	34	51	68	85	102	119	136	153	170	187	204	221	238	255	272	289	306	323	340	357	374	391	408	425	442	459	476	493	510	527
16	0	34	68	102	136	170	204	238	272	306	340	374	408	442	476	510																
32	0	68	136	204	272	340	408	476																								
64	0	136	272	408																												
128	0	272												= Padding inserted																		
256	0																															
Conflicts	Banks																															
None	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2-way	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14																
2-way	0	4	8	12	0	4	8	12																								
2-way	0	8	0	8																												
2-way	0	0																														
None	0																															

- **Wait, we still have bank conflicts**
  - Method is not foolproof, but still much improved
  - 304 cycles vs. 570 with bank conflicts vs. 120 optimal
- **But it does not pay of to optimize for the rest. Address calculations are getting too expensive**

Summary

- **Parallel Programming requires careful planning**
  - of the branching behavior
  - of the memory access patterns
  - of the work efficiency

- **Vector Reduction**
  - branch efficient
  - bank efficient

- **Scan Algorithm**
  - based in Balanced Tree principle:
    bottom up, top down traversal

Thank you.

- Hendrik Lensch, Robert Strzodka
- NVIDIA