Lecture 9:

Implementing GPU Reduce, Scan, and Sort

Modern Parallel Computing

John Owens

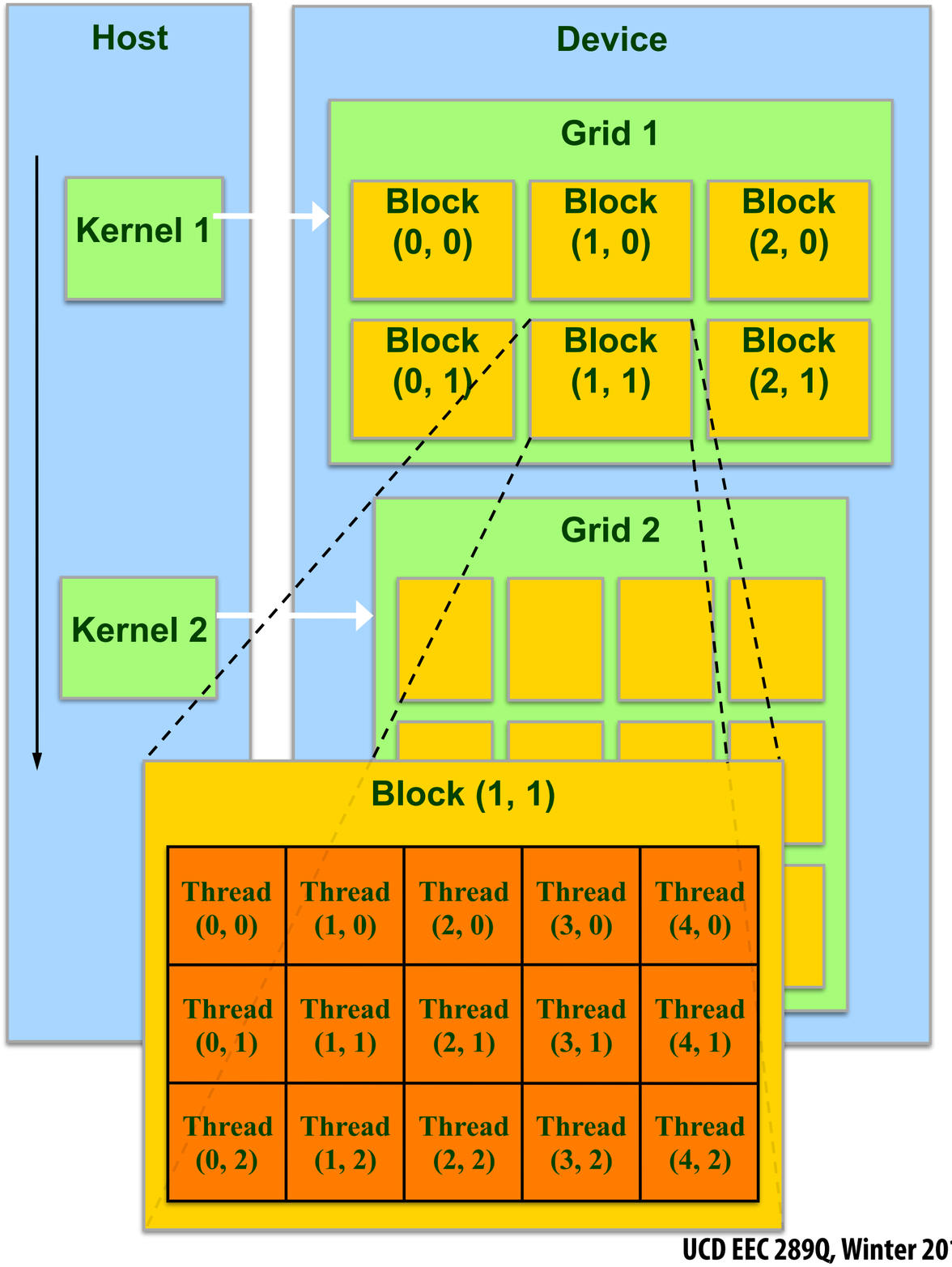
EEC 289Q, UC Davis, Winter 2018

Credits

- Thanks to Andrew Davidson (reduce), Michael Garland (Satish's sort), and especially Duane Merrill (Merrill's scan and sort) for slides and discussions.

Programming Model (SPMD + SIMD): Thread Batching

- A kernel is executed as a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Efficiently sharing data through shared memory
  - Synchronizing their execution
    - For hazard-free shared memory accesses
- Two threads from two different blocks cannot cooperate
  - Blocks are *independent*



Basic Efficiency Rules

- Develop algorithms with a data parallel mindset

- Minimize divergence of execution within blocks (actually warps)

- Maximize locality of global memory accesses

- Exploit per-block shared memory as scratchpad (registers > shared memory > global memory)

- Expose enough parallelism

Today's Big Picture

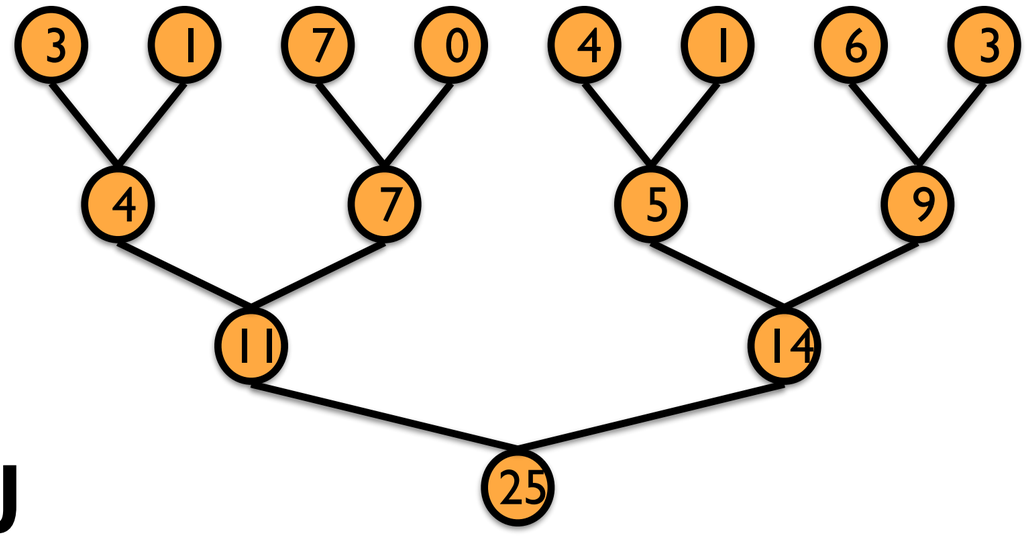
Last lecture

$$\text{Complexity} = k(O(f(n)))$$

This lecture

Reduction

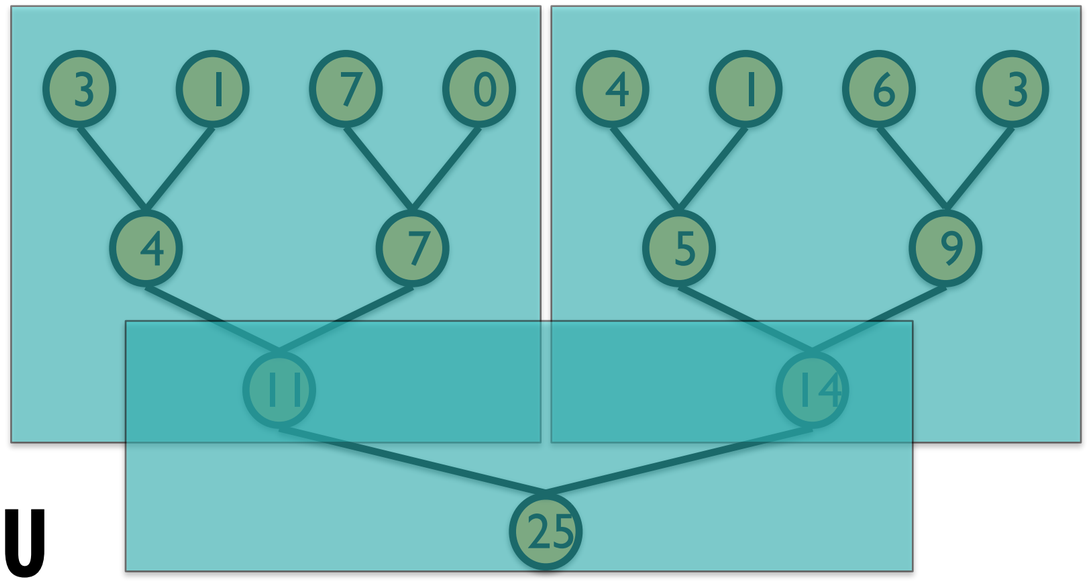
Tree-Based Parallel Reductions



- Commonly done in traditional GPGPU
  - Ping-pong between render targets, reduce by 1/2 at a time
  - Completely bandwidth bound using graphics API
  - Memory writes and reads are off-chip, no reuse of intermediate sums

- CUDA solves this by exposing on-chip shared memory
  - Reduce blocks of data in shared memory to save bandwidth

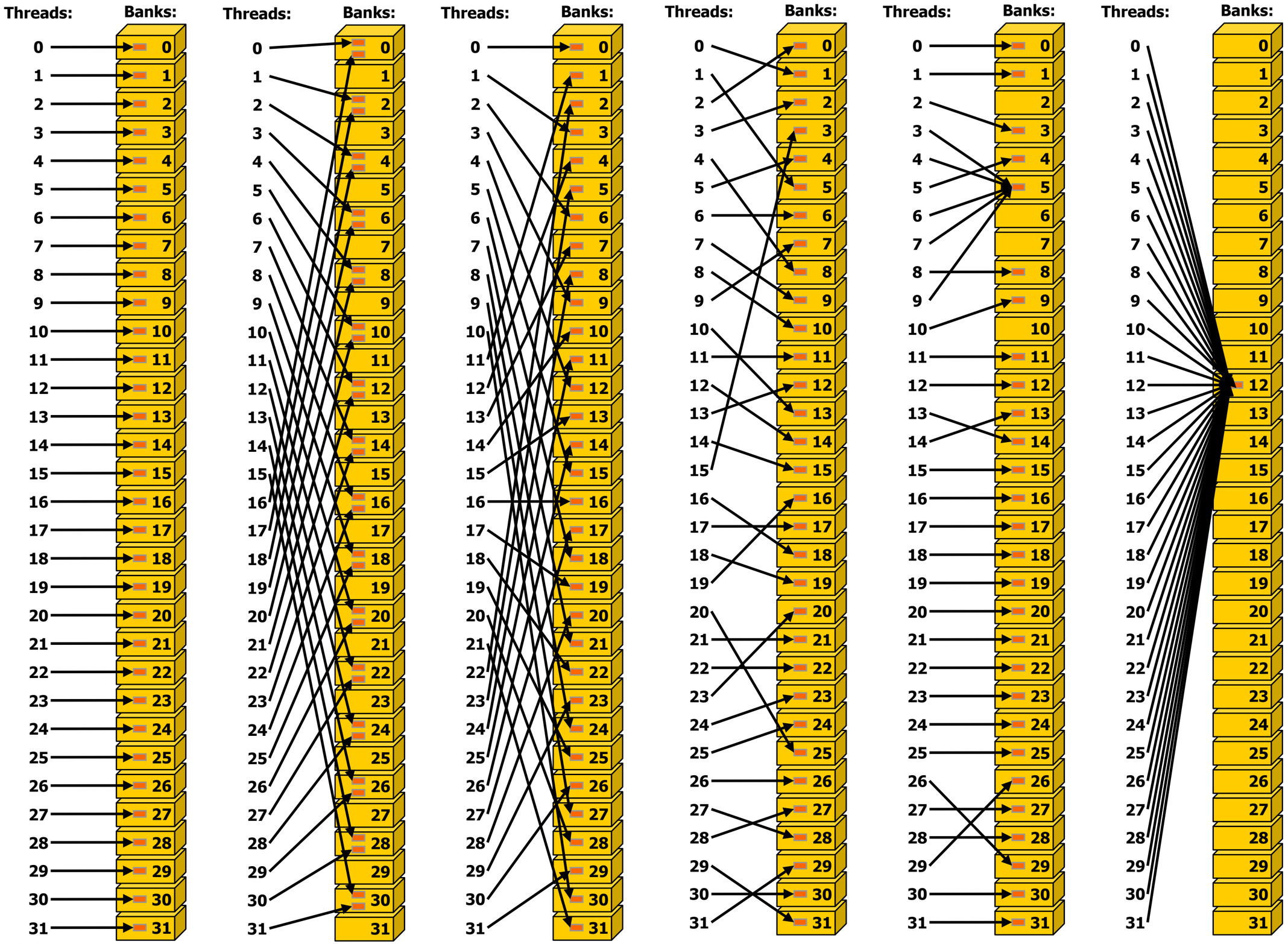
Tree-Based Parallel Reductions



- Commonly done in traditional GPGPU
  - Ping-pong between render targets, reduce by 1/2 at a time
  - Completely bandwidth bound using graphics API
  - Memory writes and reads are off-chip, no reuse of intermediate sums

- CUDA solves this by exposing on-chip shared memory
  - Reduce blocks of data in shared memory to save bandwidth

CUDA Bank Conflicts

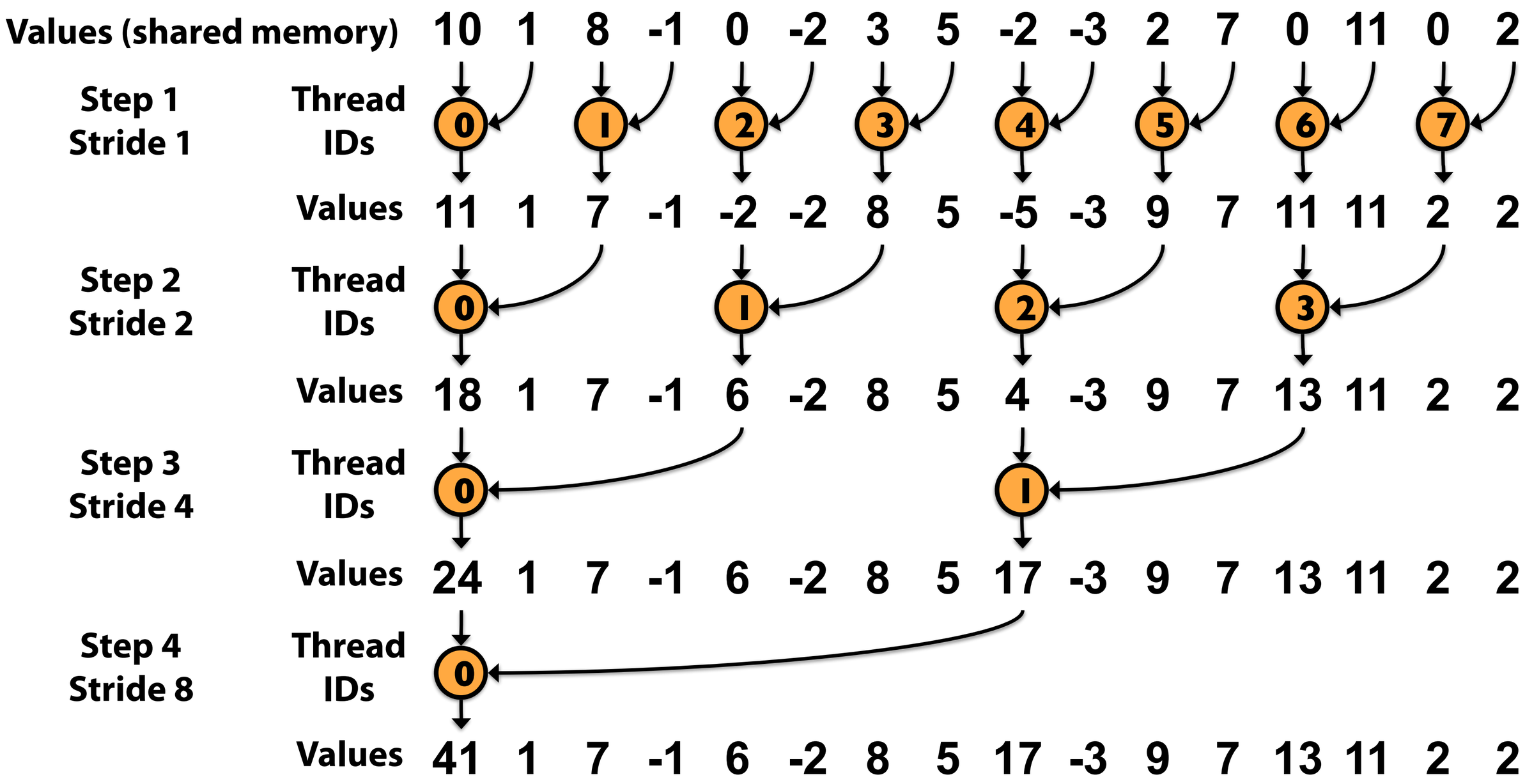


Left: Linear addressing with a stride of one 32-bit word (no bank conflict).
Middle: Linear addressing with a stride of two 32-bit words (2-way bank conflicts).
Right: Linear addressing with a stride of three 32-bit words (no bank conflict).

Left: Conflict-free access via random permutation.
Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.
Right: Conflict-free broadcast access (all threads access the same word).

Parallel Reduction: Interleaved Addressing

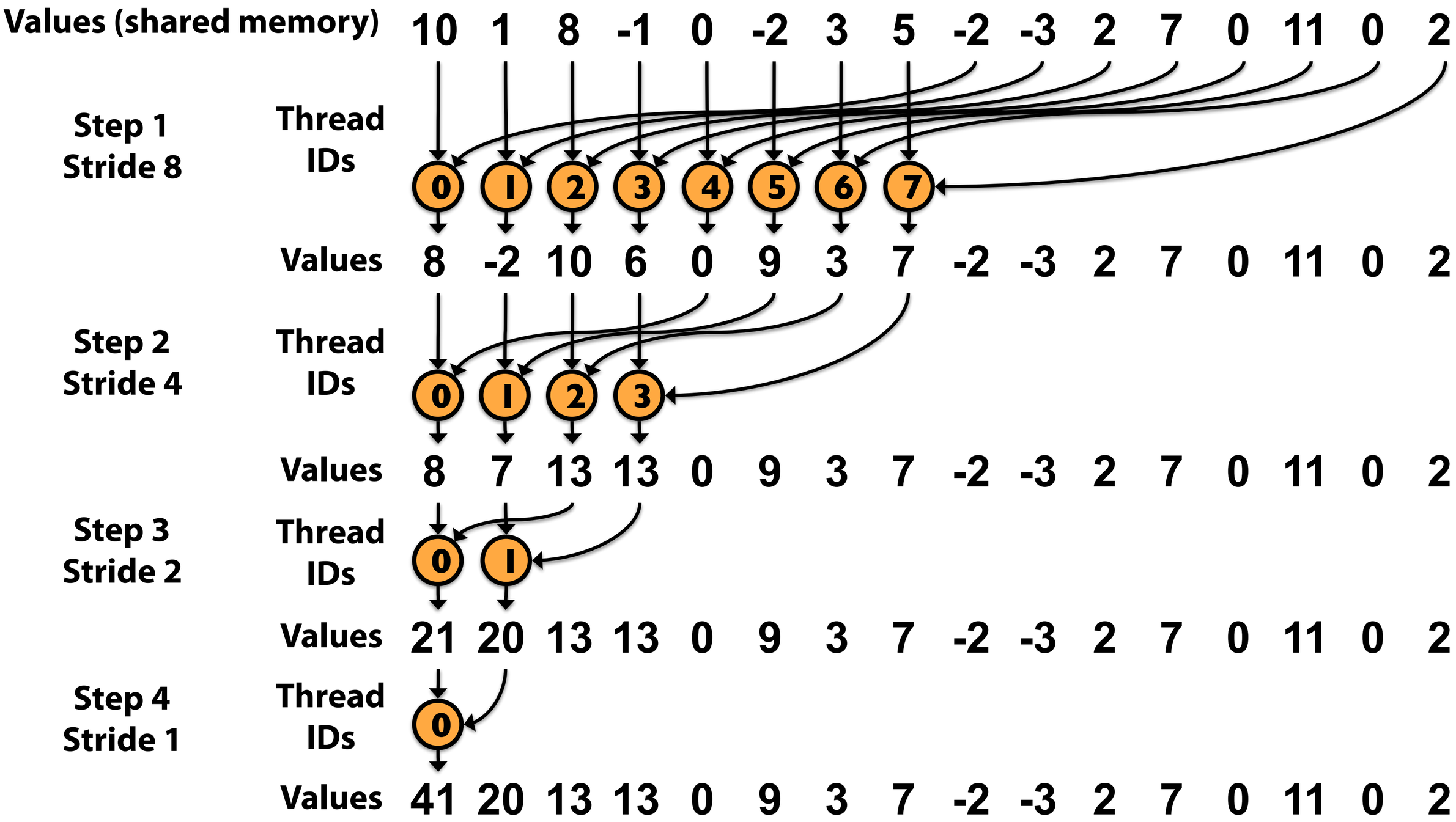
- Arbitrarily bad bank conflicts
- Requires barriers if N > warpsize
- Supports non-commutative operators



Interleaved addressing results in bank conflicts

Parallel Reduction: Sequential Addressing

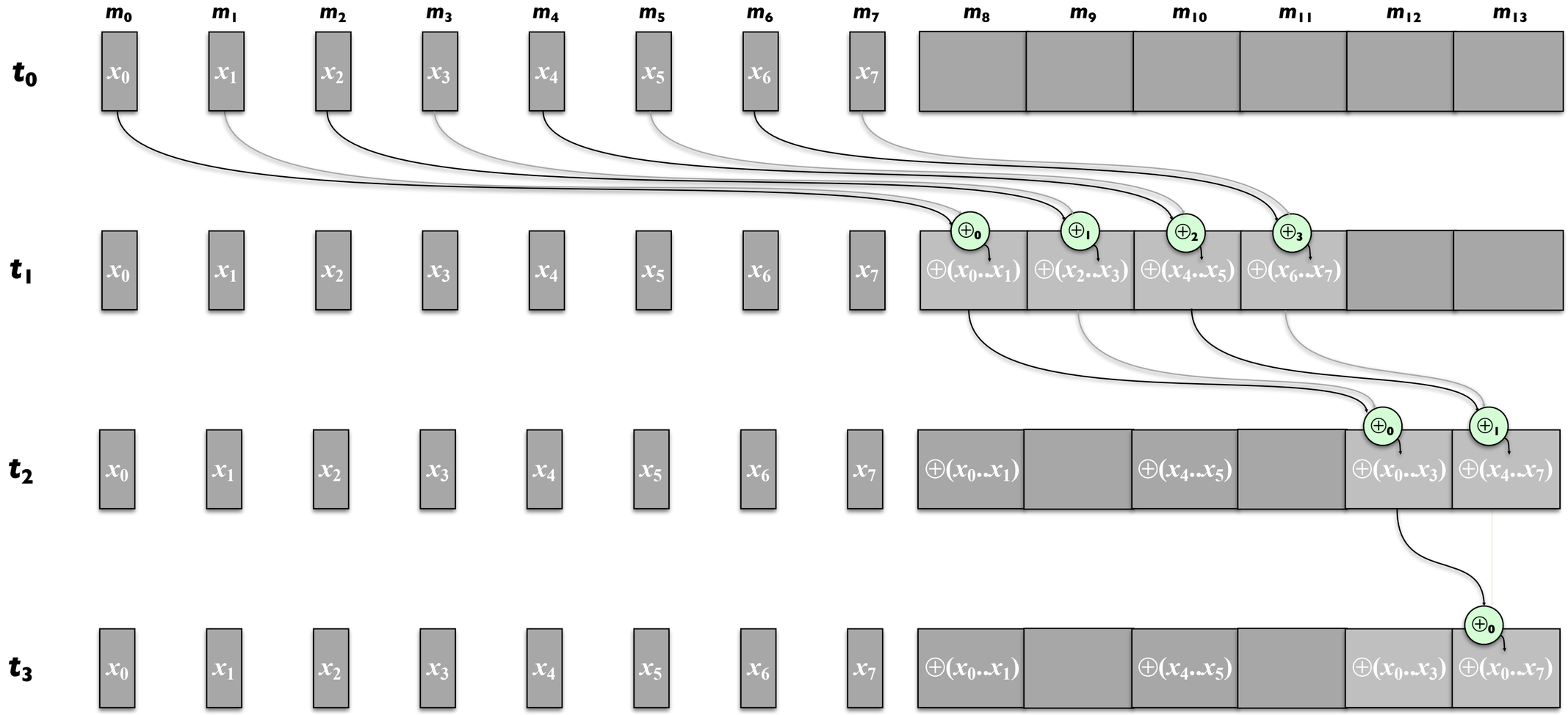
- No bank conflicts
- Requires barriers if N > warpsize
- Does not support non-commutative operators



Sequential addressing is conflict free

Reduction

- Only two-way bank conflicts
- Requires barriers if $N >$ warpsize
- Requires $O(2N\text{-}2)$ storage
- Supports non-commutative operators



Reduction memory traffic

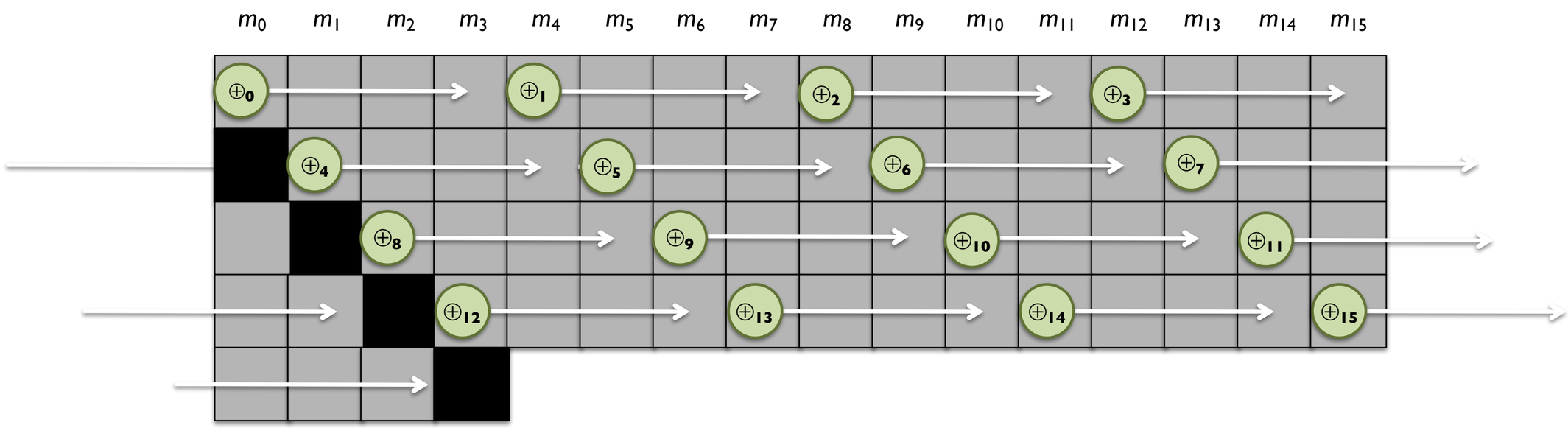
- Ideal: *n* reads, 1 write.
- Block size 256 threads. Thus:
  - Read *n* items, write back *n*/256 items.
  - Read *n*/256 items, write back 1 item.
  - Total: $n + n/128 + 1$. Not bad!

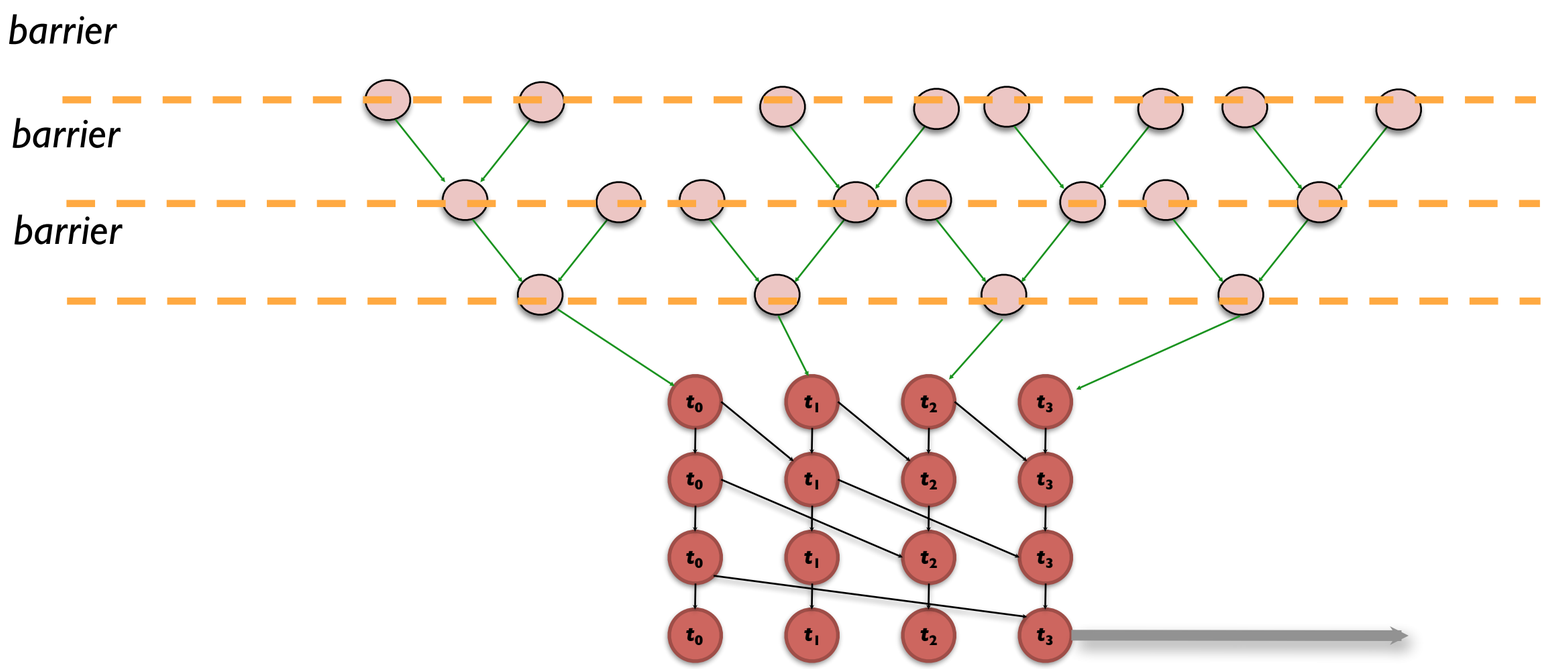
Reduction optimization

- Ideal: *n* reads, 1 write.
- Block size 256 threads. Thus:
  - Read *n* items, write back *n*/256 items.
  - Read *n*/256 items, write back 1 item.
  - Total: *n* + *n*/128 + 1. Not bad!
- What if we had more than one item (say, 4) per thread?
  - "Loop raking" is an optimization for all the algorithms I talk about today.
  - Tradeoff: Storage for efficiency

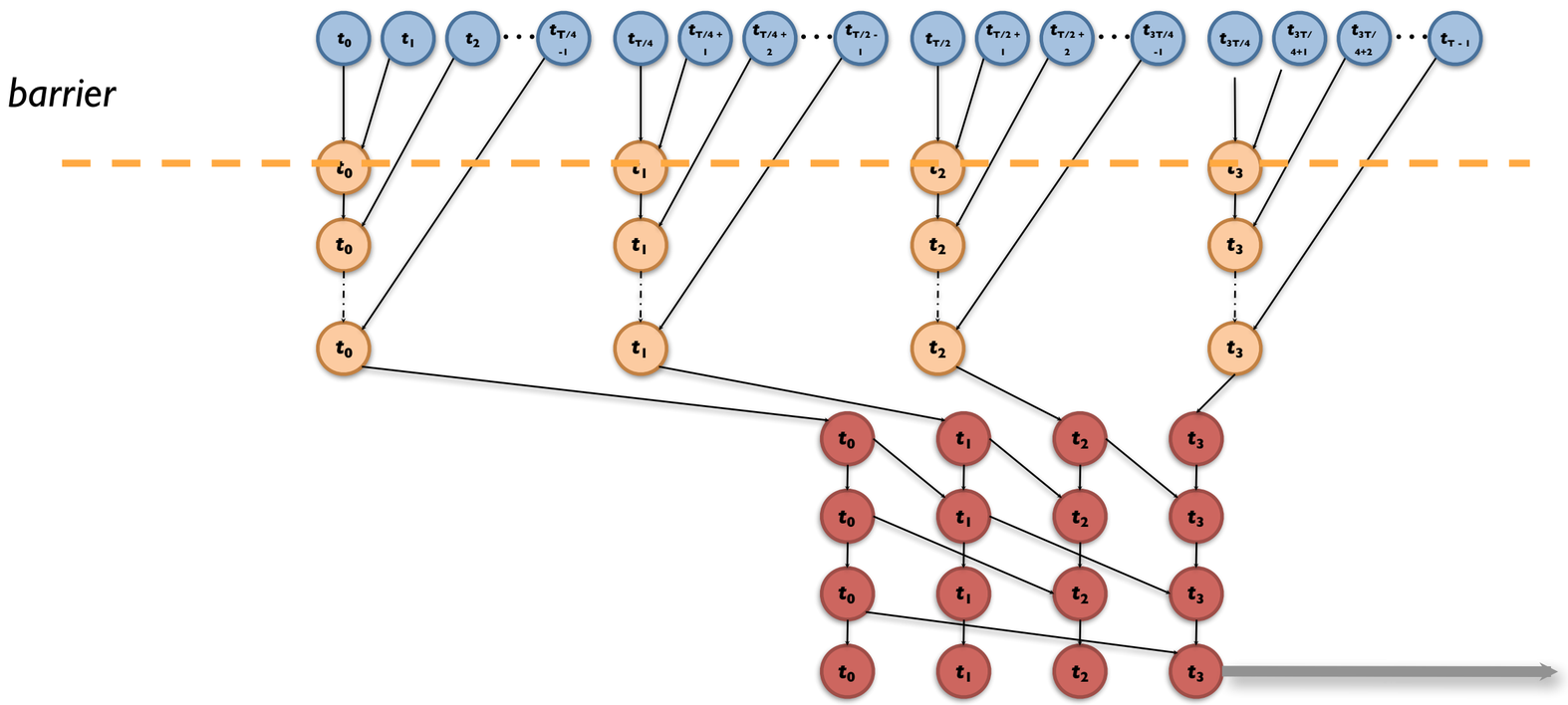
(Serial) Raking Reduction Phase

- No bank conflicts, only one barrier to after insertion into smem
- Supports non-commutative operators
- Requires subsequent warpscan to reduce accumulated partials
- Less memory bandwidth overall
- Exploits locality between items within a thread's registers
- When does this work? e.g., *O*(1) state for operation





VS.

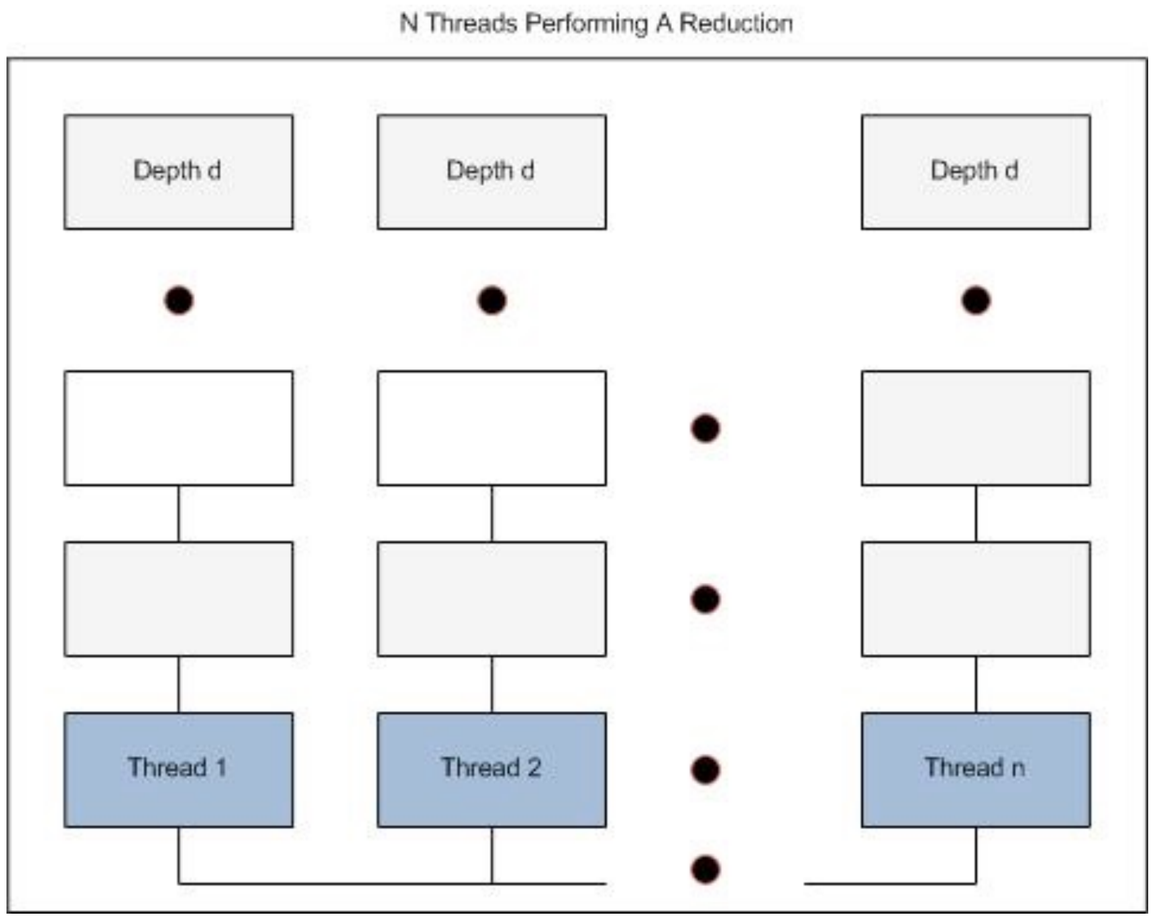


Persistent Threads

- GPU programming model suggests one thread per item
- What if you filled the machine with just enough threads to keep all processors busy, then asked each thread to stay alive until the input was complete?
- Minus: More overhead per thread (register pressure)
- Minus: Violent anger of vendors

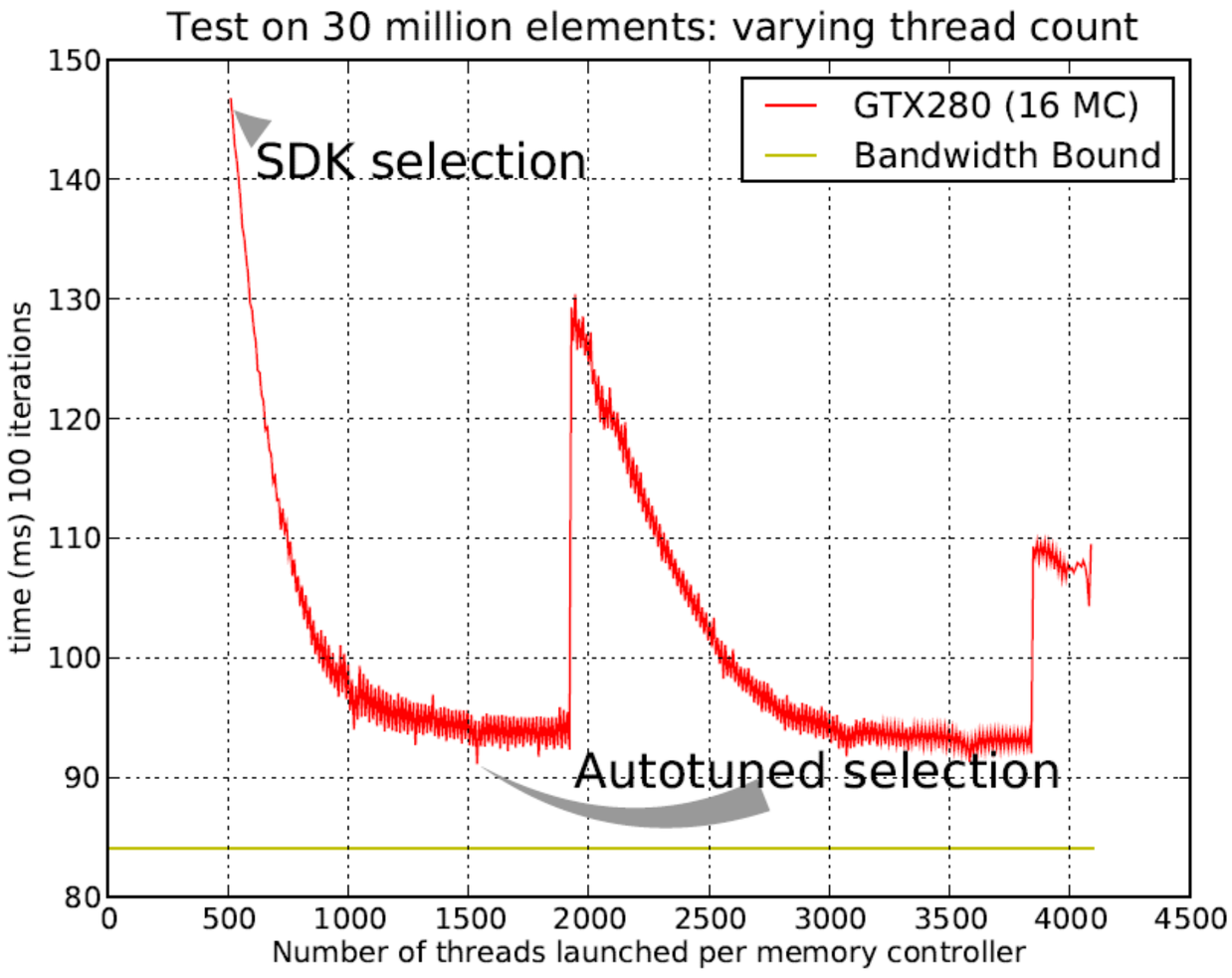
Reduction

- Many-To-One
  - Parameter to Tune => Thread Width (total number of threads)

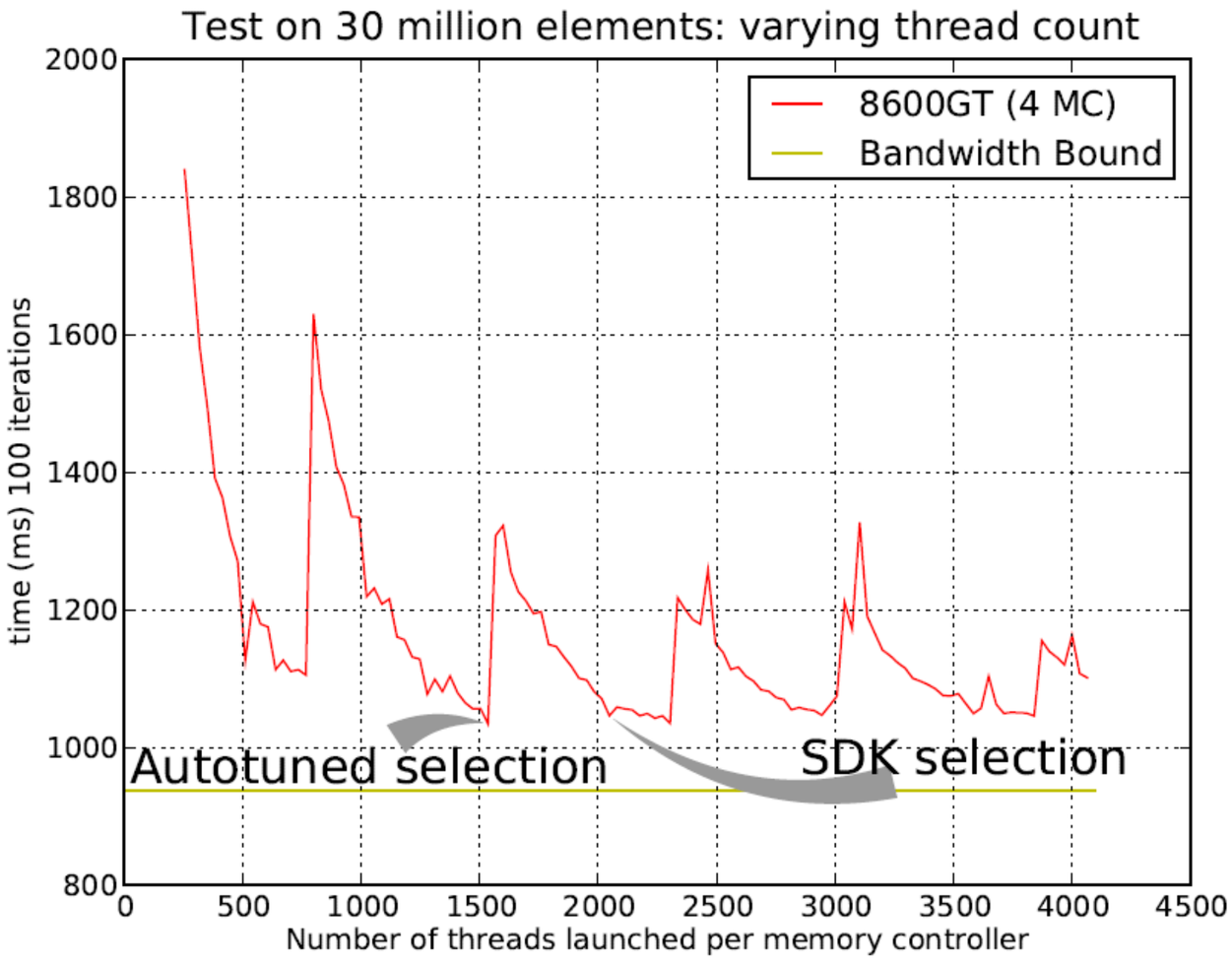


Parameter Selection Comparison

GTX280

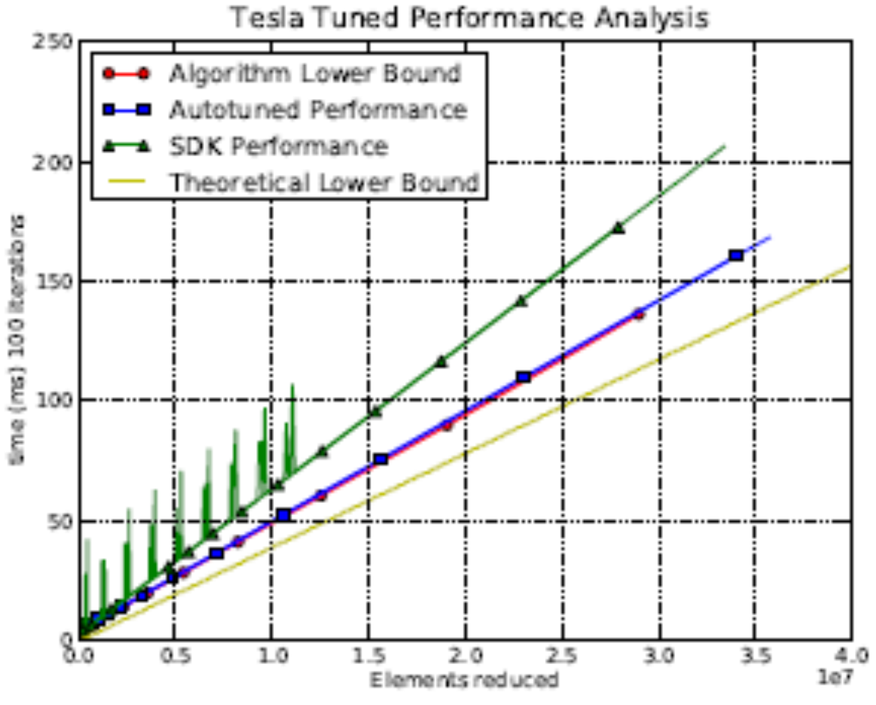


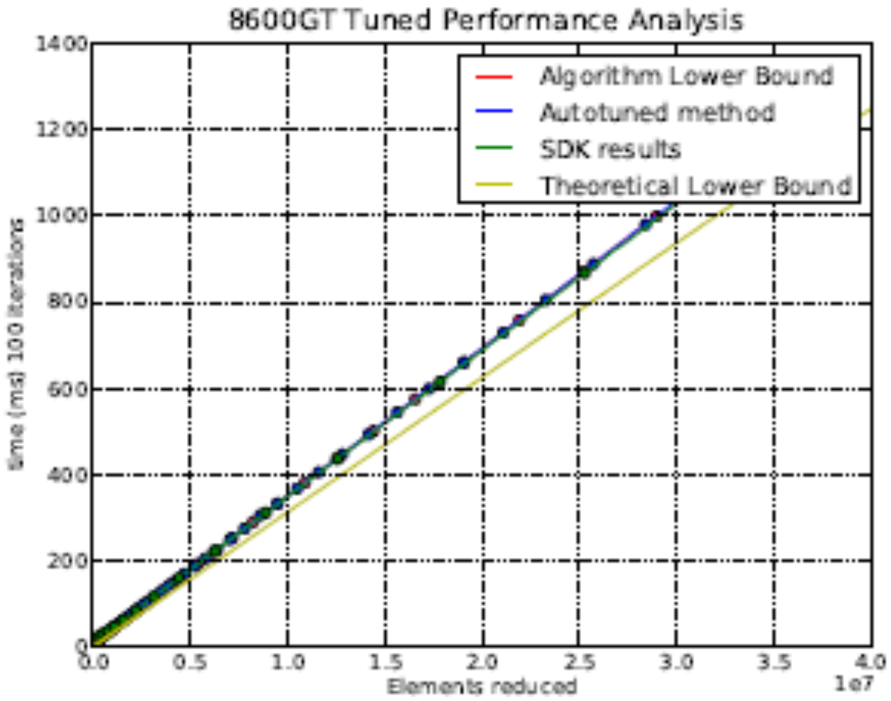
8600GT

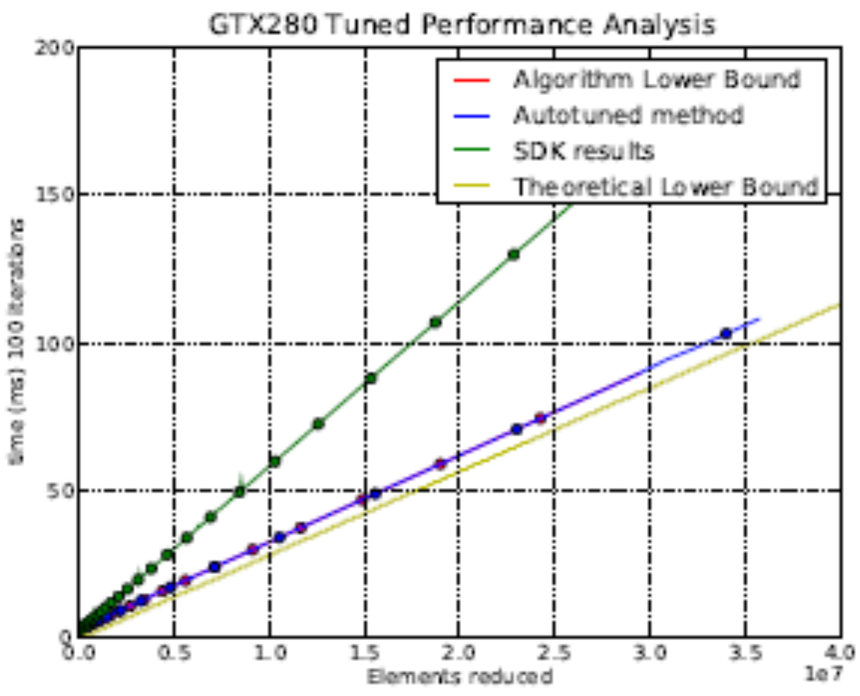


- Parameter selection comparison between the static SDK and our tuned (thread cap) algorithm
- We see some of the problems with having static thread parameters, for different machines.

Autotuning results







- **Auto-tuned performance always exceeded SDK performance**
  - **Up to a 70% performance gain for certain cards and workloads**

Reduction papers

- Mark Harris, Mapping Computational Concepts to GPUs, GPU Gems 2, Chapter 31, pp. 495–508, March 2005.
- Andrew Davidson and John Owens. Toward Techniques for Auto-tuning GPU Algorithms. In Kristján Jónasson, editor, Applied Parallel and Scientific Computing, volume 7134 of Lecture Notes in Computer Science, pages 110–119. Springer Berlin / Heidelberg, February 2012.
- NVIDIA SDK (reduction example)

Scan (within a block)

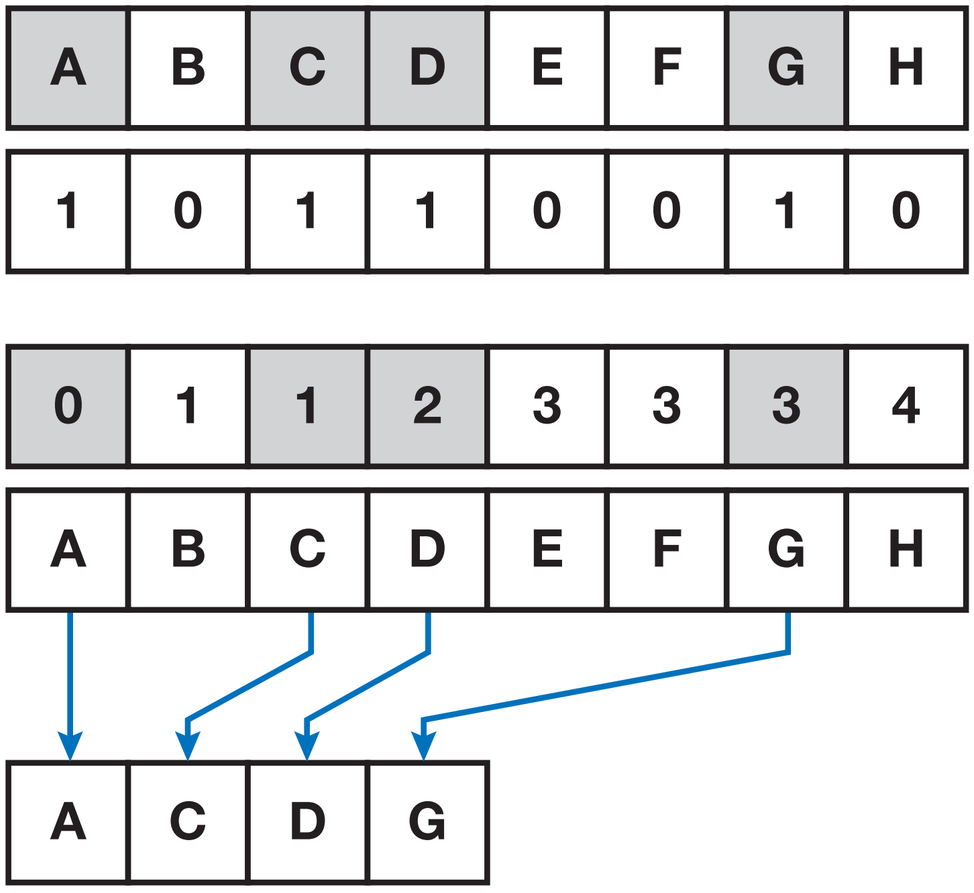
Parallel Prefix Sum (Scan)

- Given an array $A = [a_0, a_1, \ldots, a_{n-1}]$
  and a binary associative operator $\oplus$ with identity I,

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

- 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]

Application: Stream Compaction



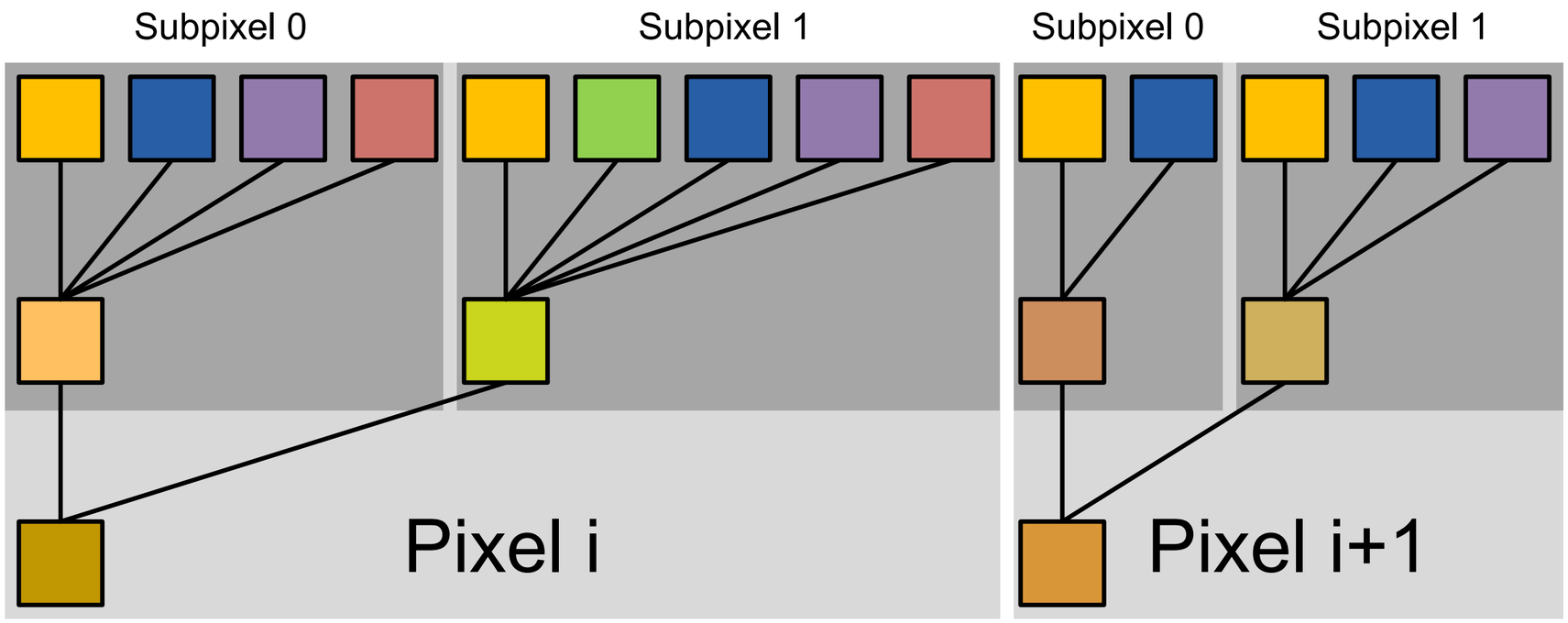
Input: we want to preserve the gray elements

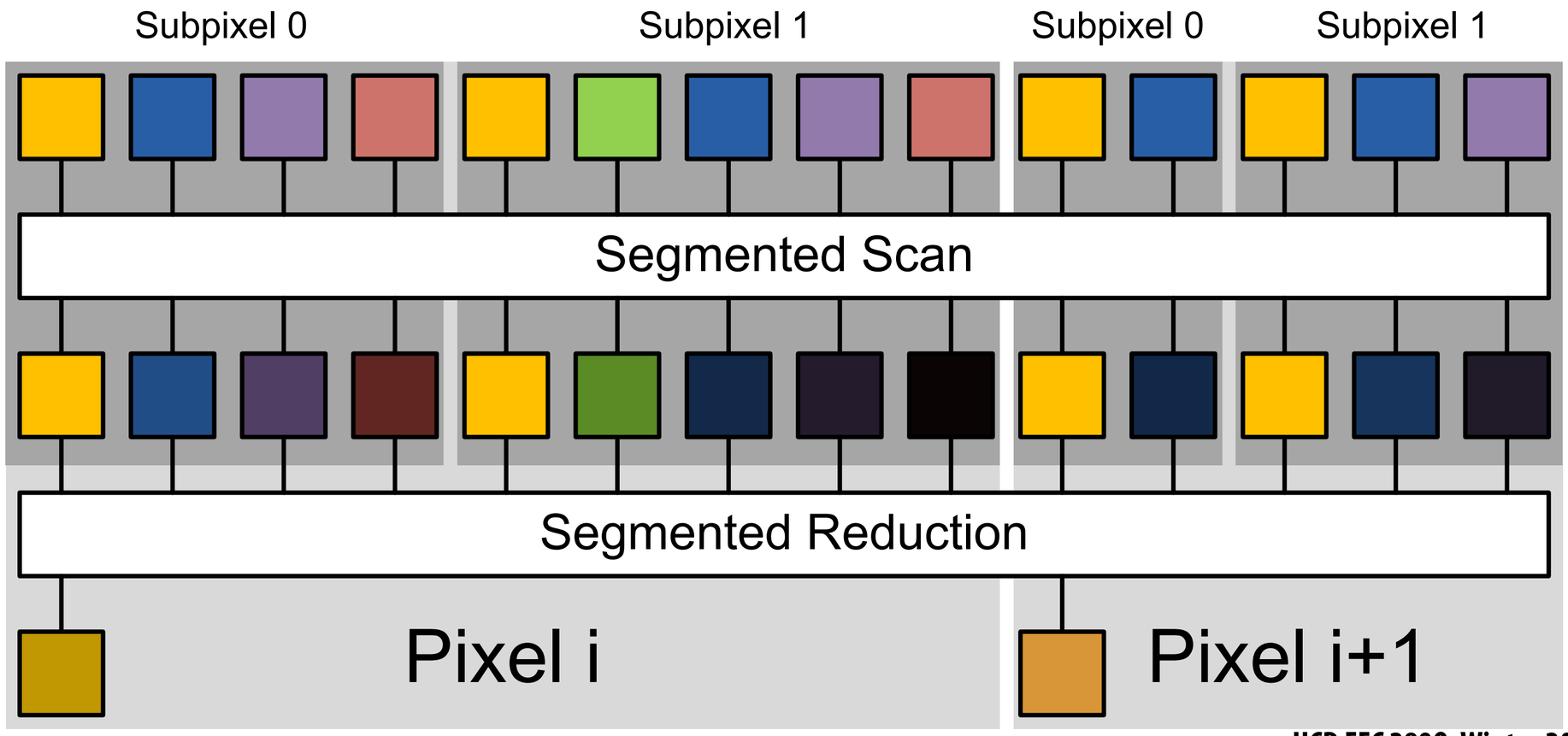
Set a “1” in each gray input

Scan

Scatter input to output, using scan result as scatter address

Pixel- vs. Sample-Parallel Composition

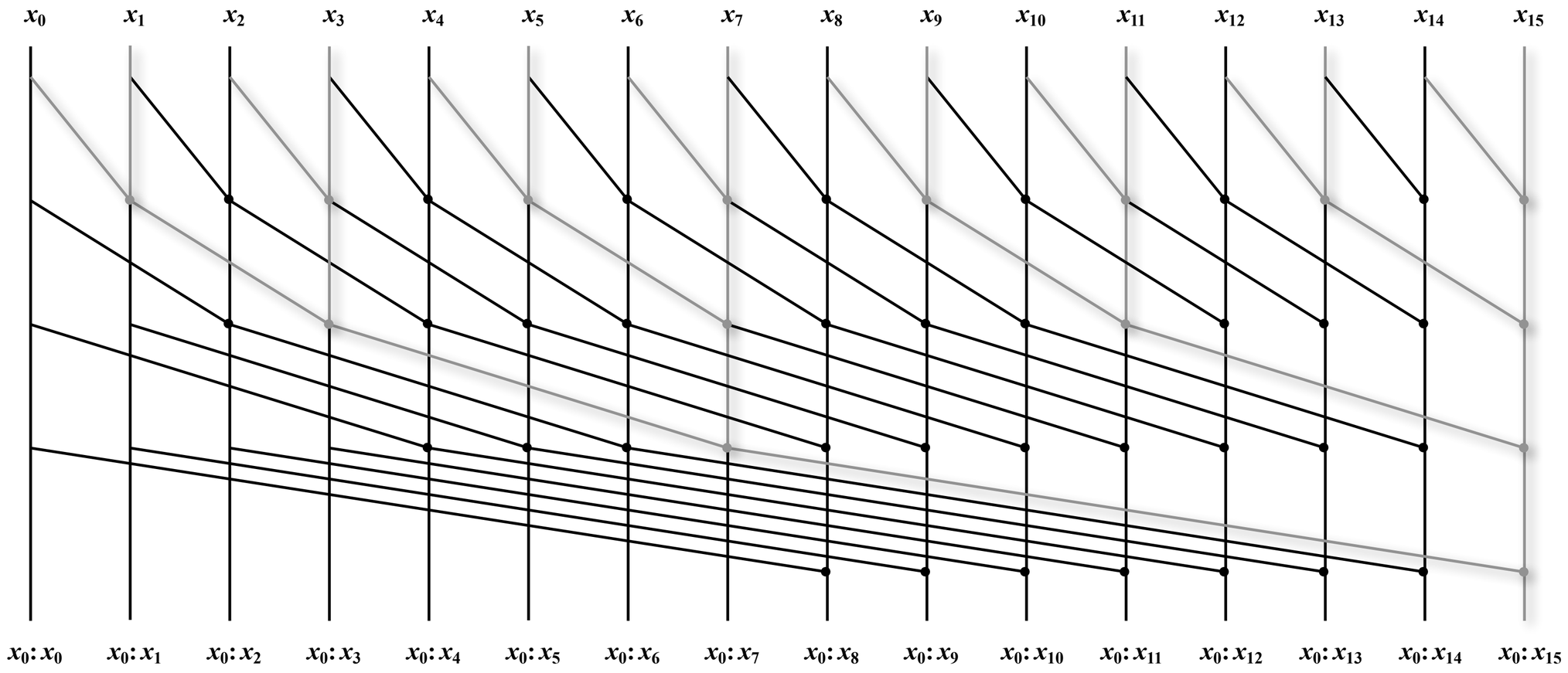




Move-to-front transform

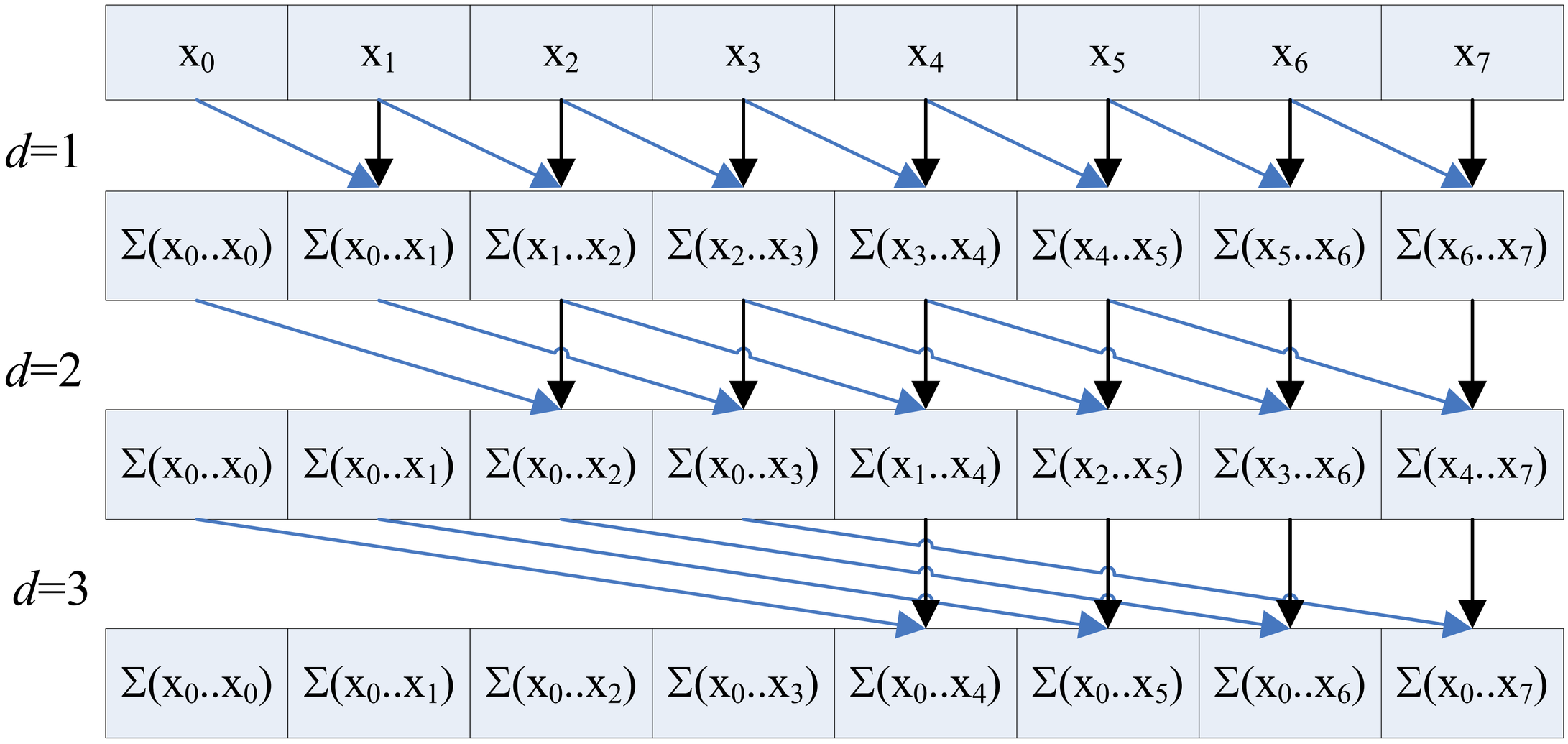
Iteration	Sequence	List
bananaaa	1	(abcdefghijklmnopqrstuvwxyz)
b**a**nanaaa	1,1	(bacdefghijklmnopqrstuvwxyz)
ba**n**anaaa	1,1,13	(abcdefghijklmnopqrstuvwxyz)
ban**a**naaa	1,1,13,1	(nabcdefghijklmopqrstuvwxyz)
bana**n**aaa	1,1,13,1,1	(anbcdefghijklmopqrstuvwxyz)
banan**a**aa	1,1,13,1,1,1	(nabcdefghijklmopqrstuvwxyz)
bananaaa	1,1,13,1,1,1,0	(anbcdefghijklmopqrstuvwxyz)
bananaa**a**	1,1,13,1,1,1,0,0	(anbcdefghijklmopqrstuvwxyz)
Final	1,1,13,1,1,1,0,0	(anbcdefghijklmopqrstuvwxyz)

Kogge-Stone Scan (circuit family)



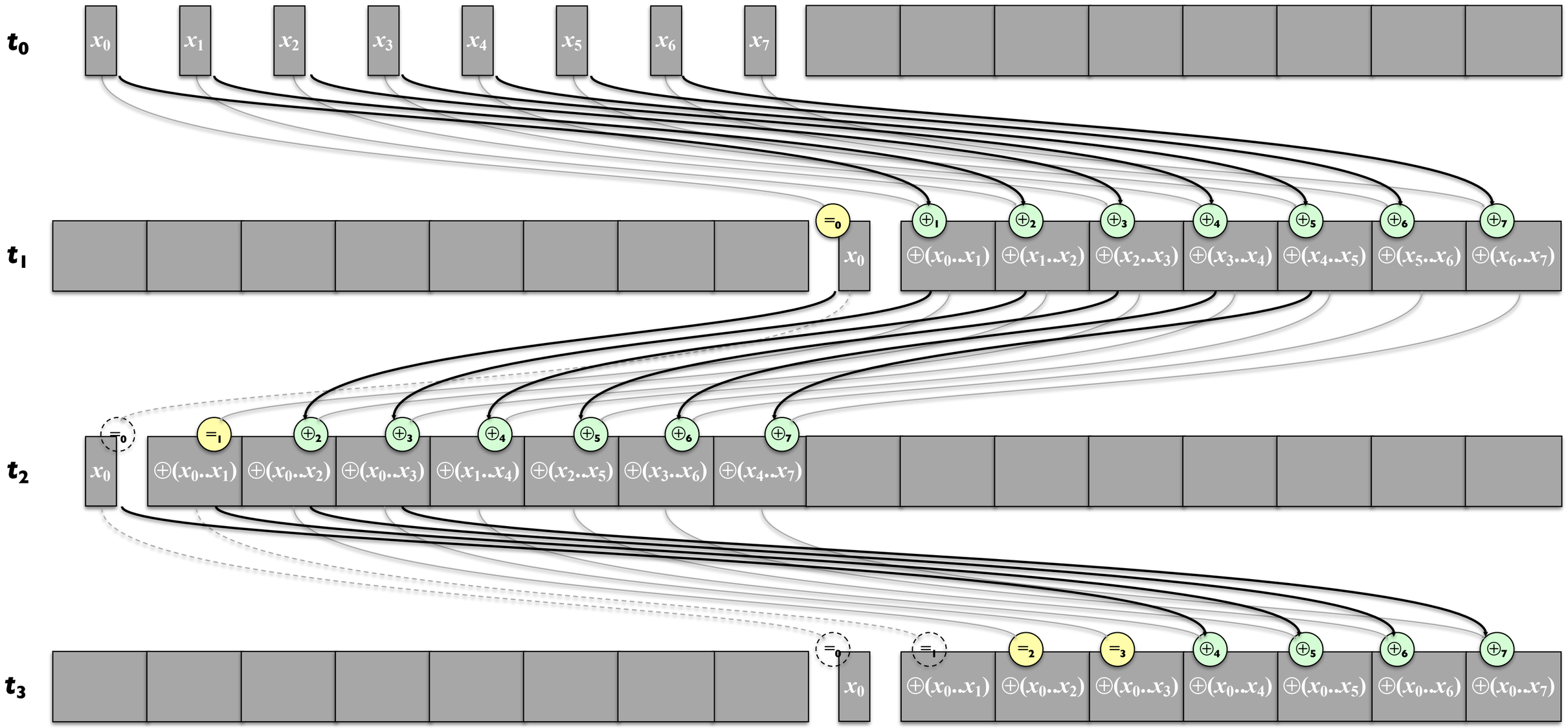
O(*n* log *n*) Scan

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



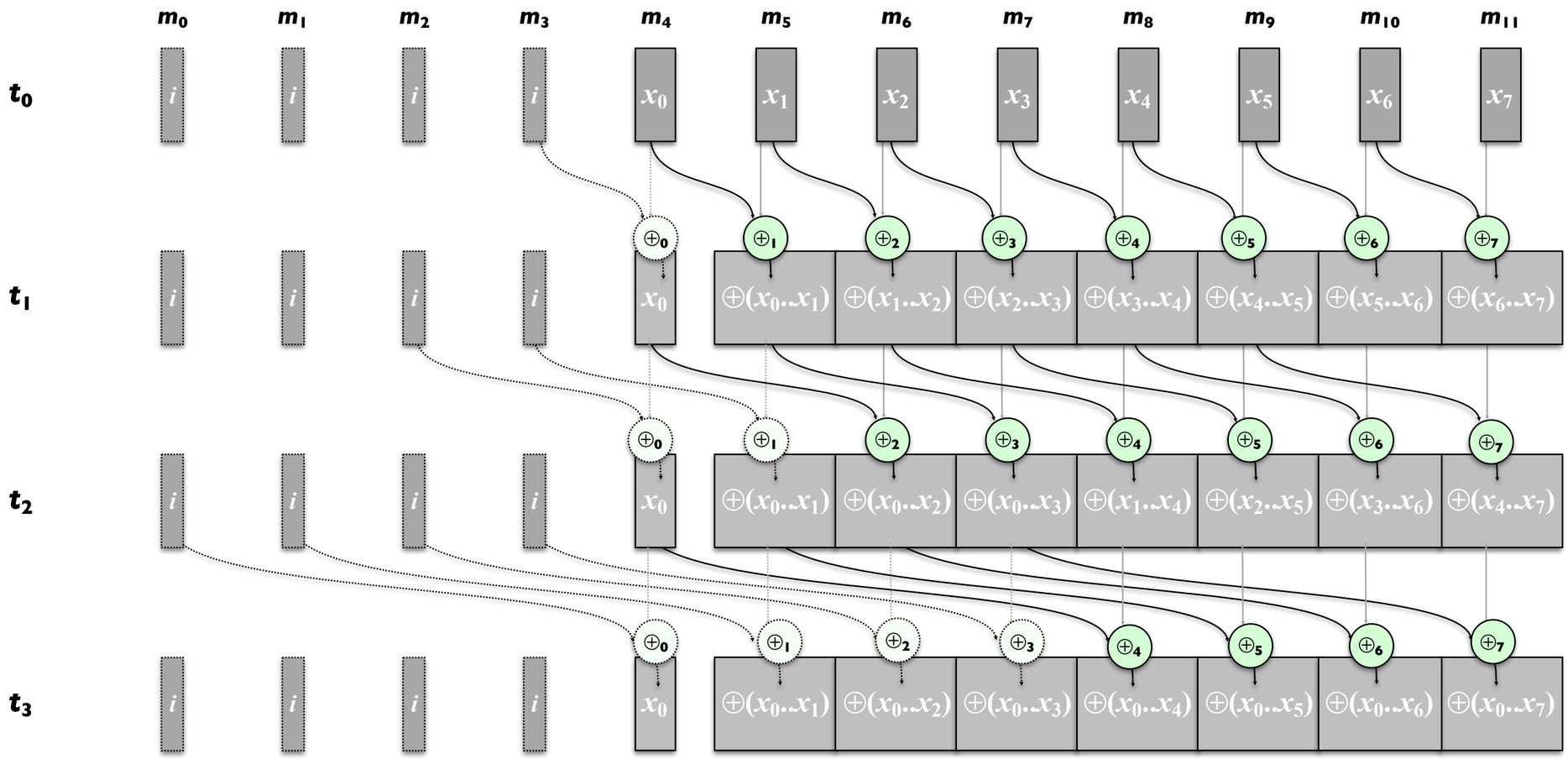
Alt. Hillis-Steele Scan Implementation

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

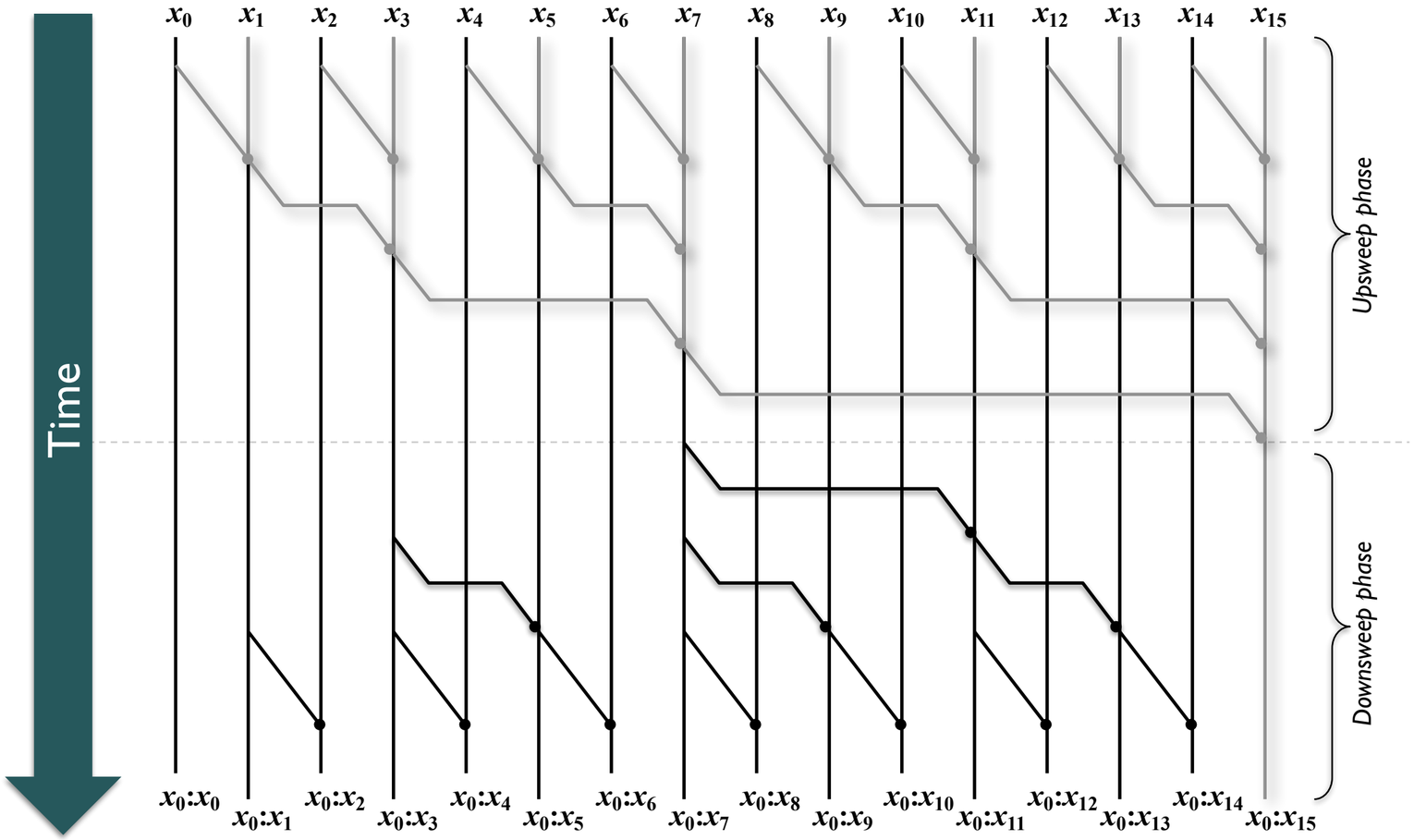


Alt. Hillis-Steele Scan

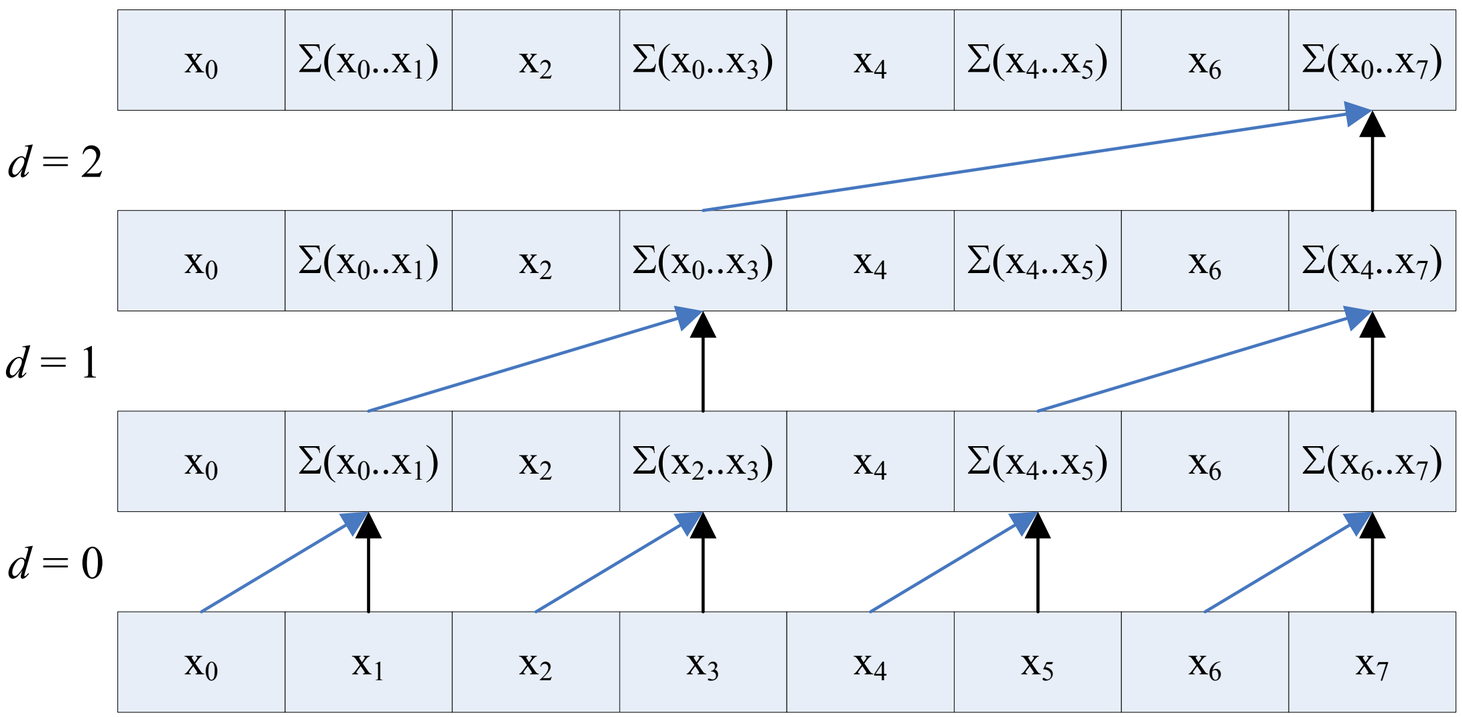
- What if we truly had a SIMD machine?
- Recall CUDA warps (32 threads) are strictly SIMD
- "Warp-synchronous": no divergence, no barriers



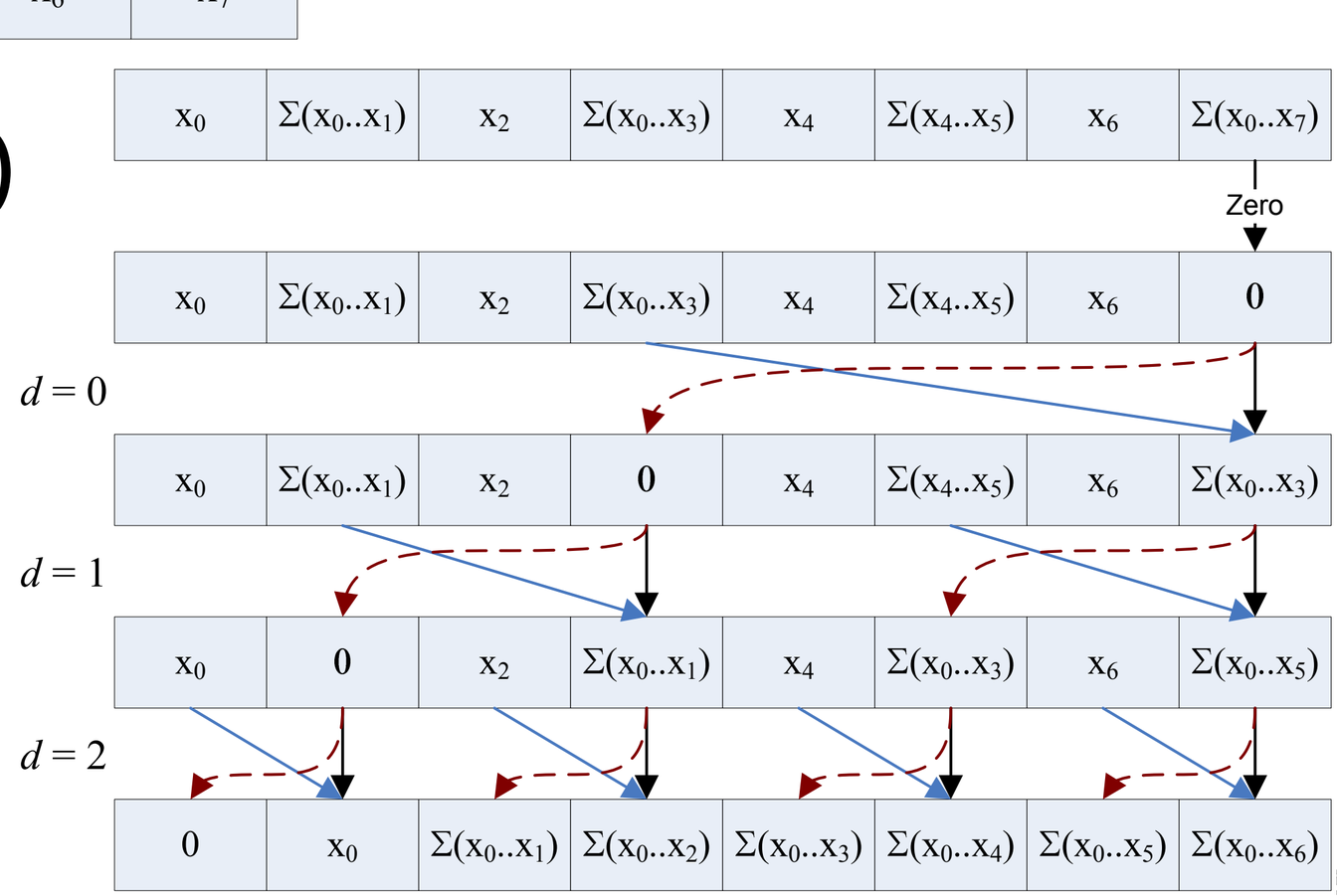
Brent Kung Scan (circuit family)



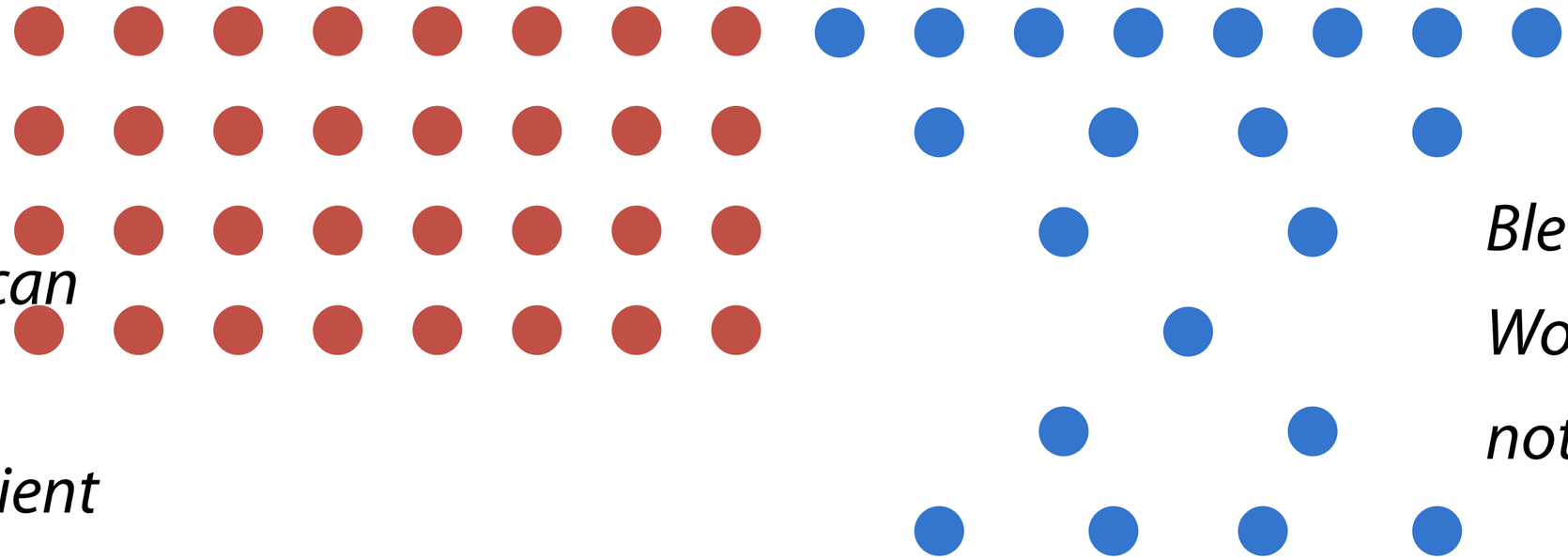
O(*n*) Scan [Blelloch]



- Not step efficient (2 log *n* steps)
- Work efficient (*O*(*n*) work)
- Bank conflicts, and lots of 'em



Hybrid methods



Hillis/Steele scan

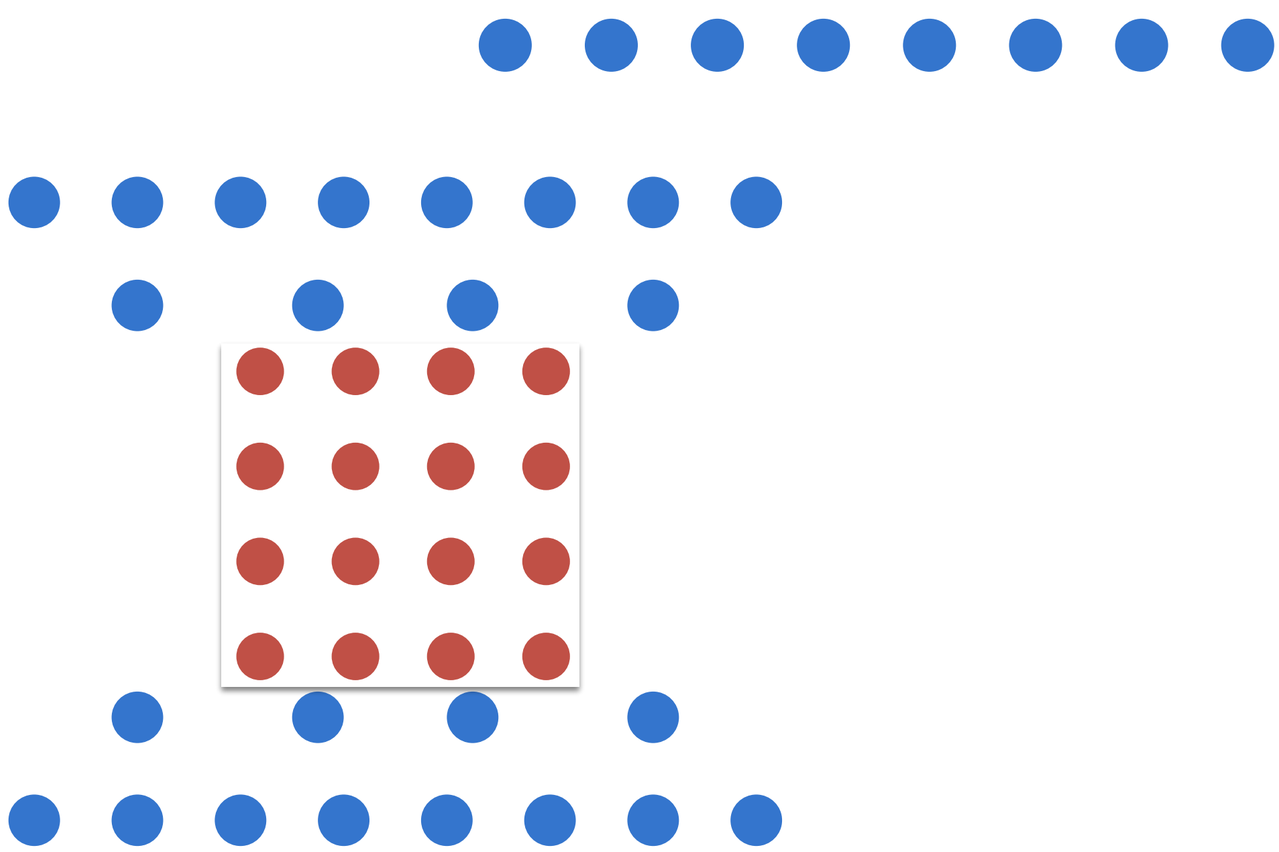
Step efficient,

not work efficient

Blelloch scan

Work efficient,

not step efficient

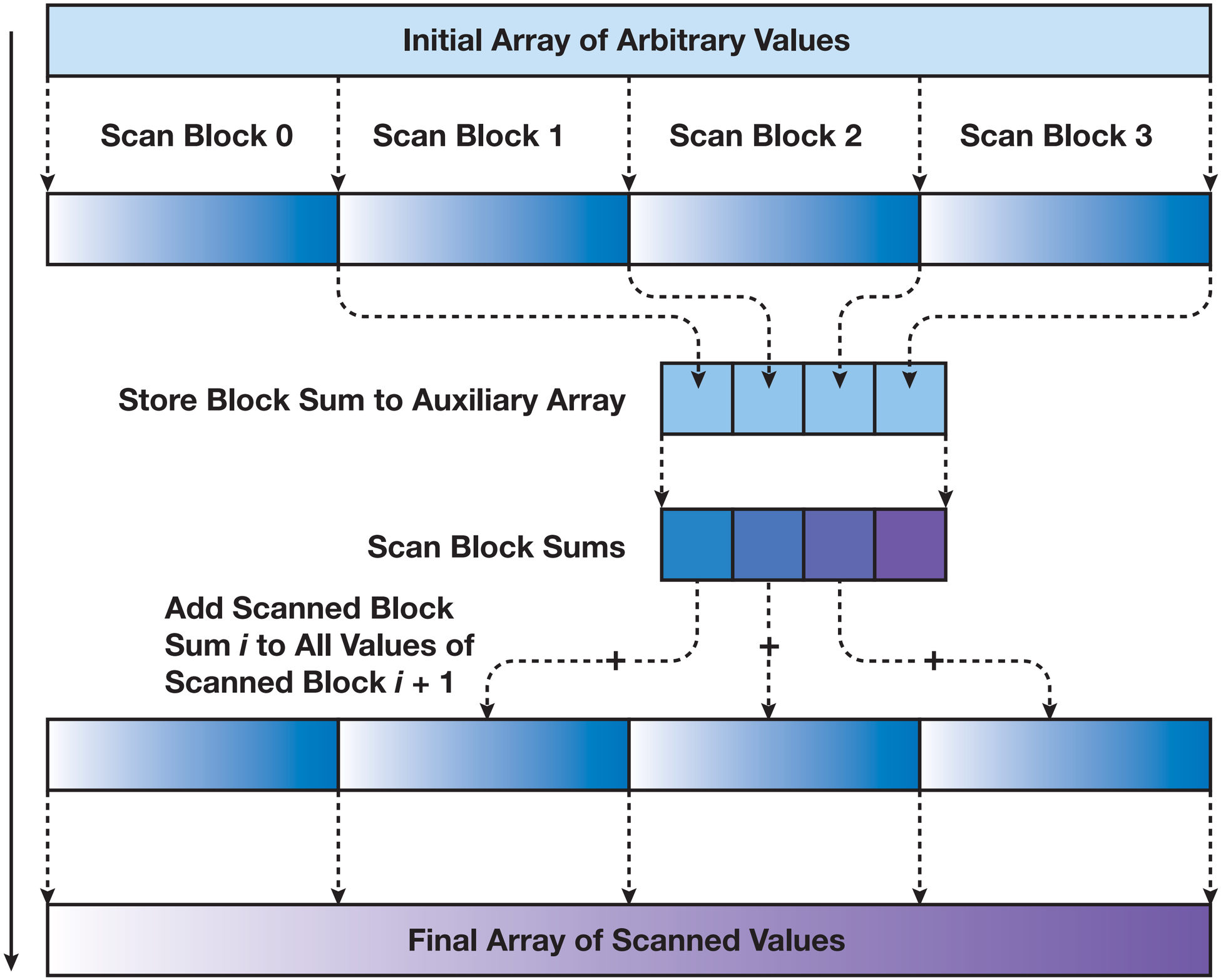


Scan papers

- 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.

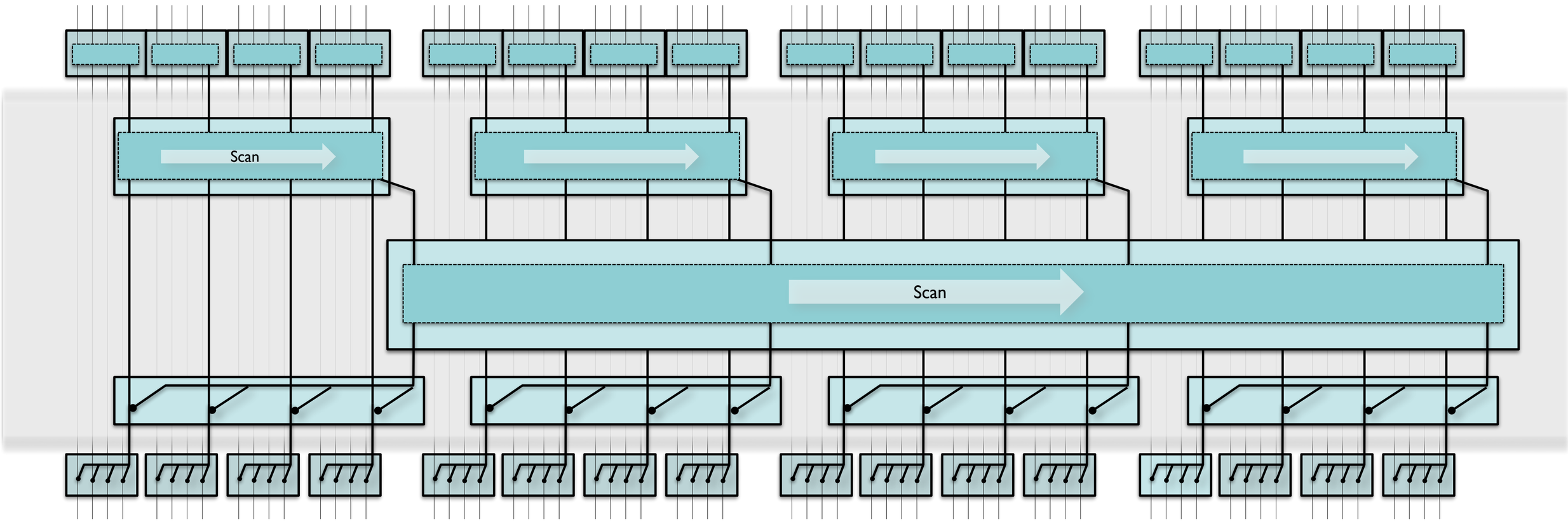
Scan (across blocks)

Scan-then-propagate (4x)



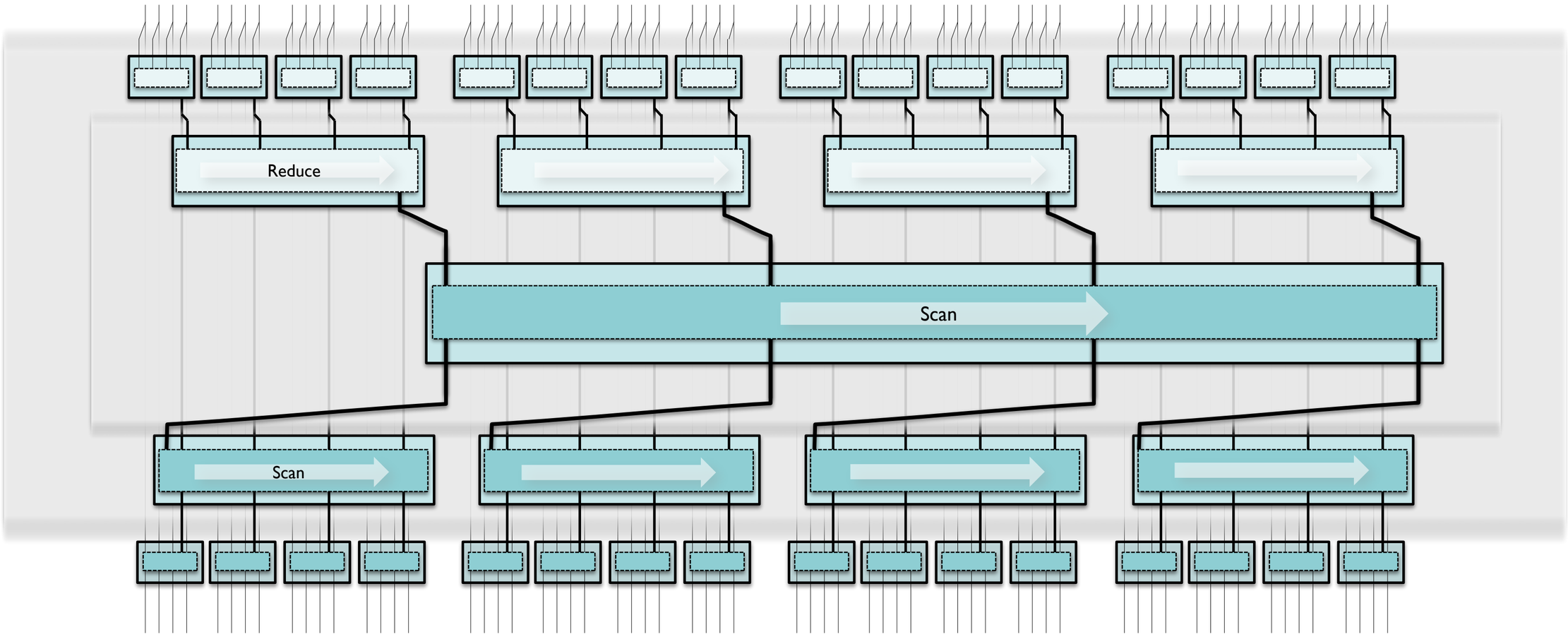
Scan-then-propagate (4x)

- $\log_b(N)$ –level upsweep/downsweep (scan-and-add strategy: CUDPP)



Reduce-then-scan (3x)

- $\log_b(N)$–level upsweep/downsweep
  (reduce-then-scan strategy: Matrix-scan)

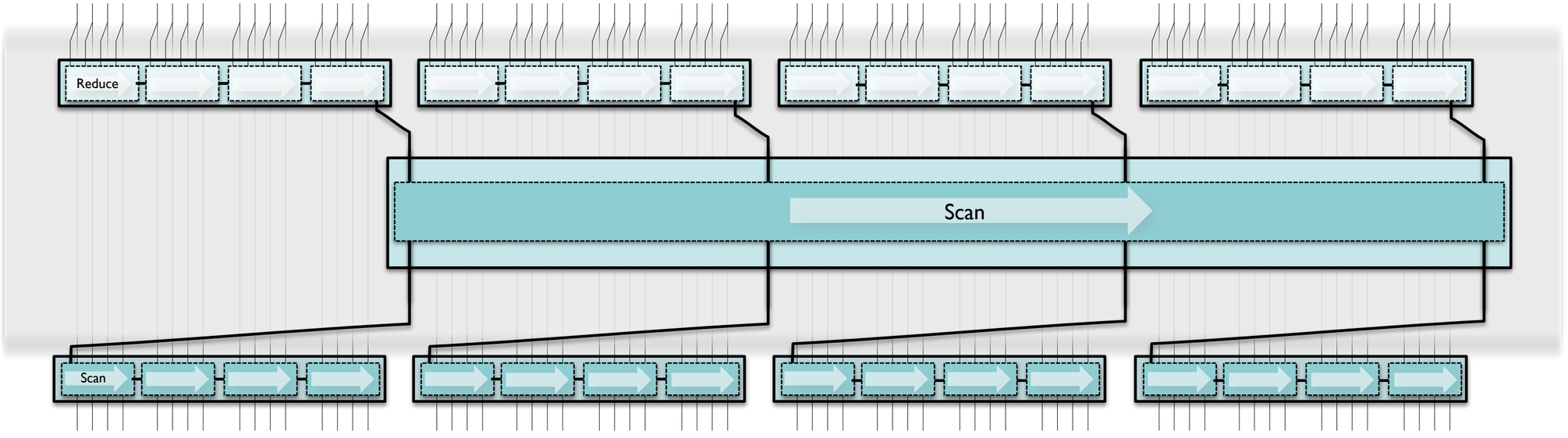


Persistent Threads (reminder)

- GPU programming model suggests one thread per item
- What if you filled the machine with just enough threads to keep all processors busy, then asked each thread to stay alive until the input was complete?
- Minus: More overhead per thread (register pressure)
- Minus: Violent anger of vendors

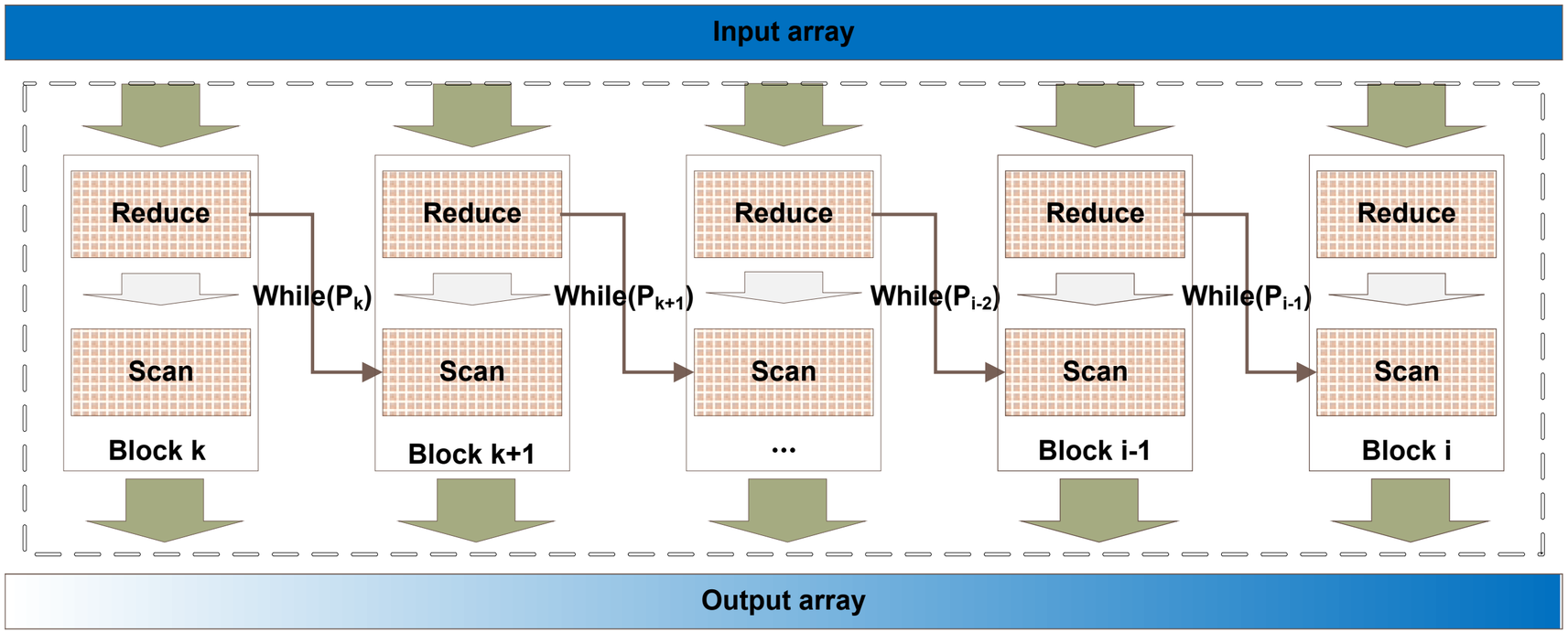
Merrill's 2–level upsweep/downsweep (reduce-then-scan)

- Persistent threads
- Requires only 3 kernel launches vs. log *n*
- Fewer global memory reads in intermediate step (constant vs. *O*(*n*))



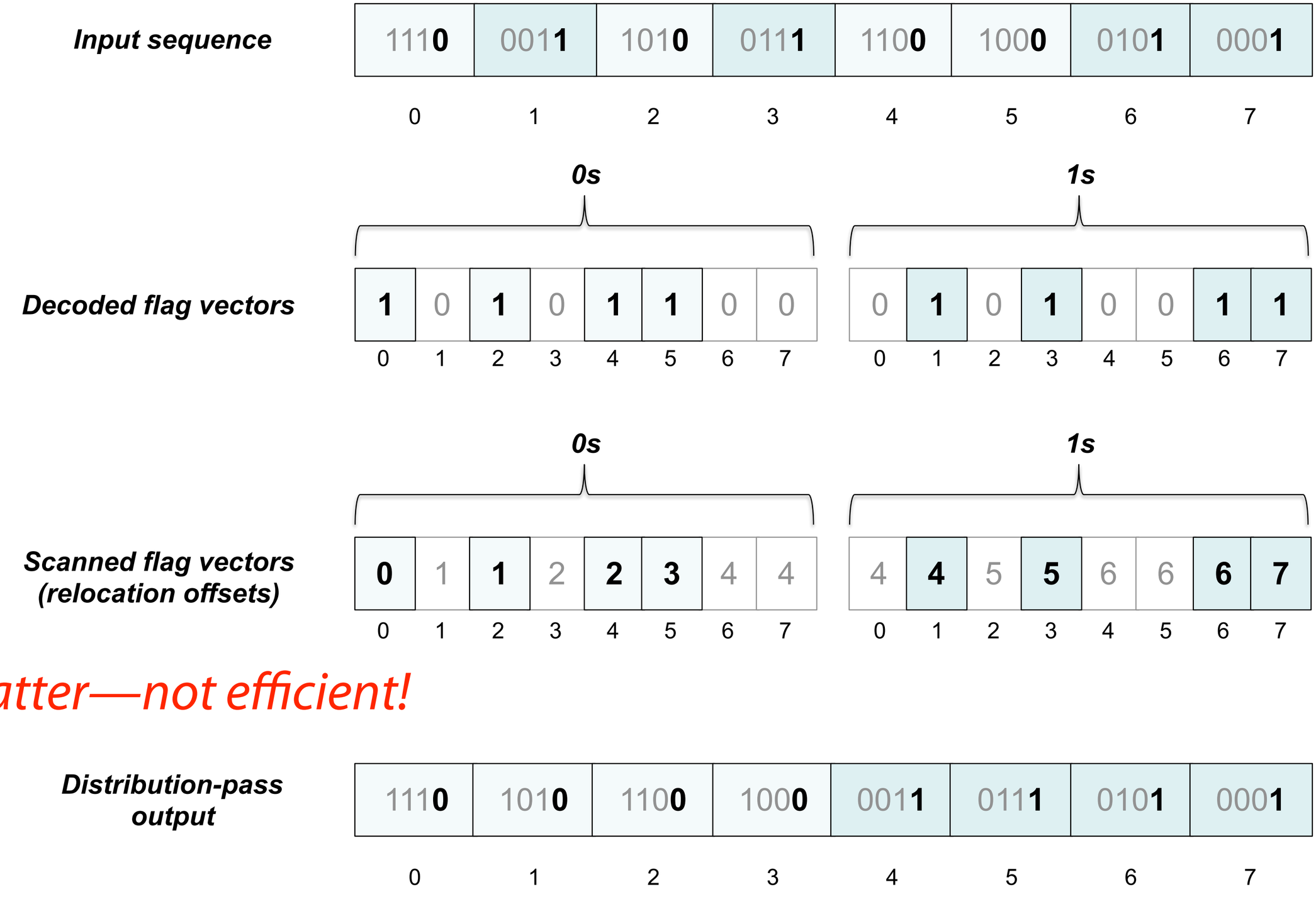
StreamScan (2x)

- Serialize reduce chain
- Atomic counter assigns blocks in proper order (no deadlocks)
- Global array, one element per block: "ready" flag + reduce value



Radix Sort

Radix Sort Fundamentals



scatter—not efficient!

Goals: (1) minimize number of scatters; (2) maximize coherence of scatters. But as radix goes up, coherence goes down.

Radix Sort Memory Cost

- ***d*-bit radix digits**
- **radix $r = 2^d$ (binary: $d = 1$, radix $r = 2$)**
- ***n*-element input sequence of *k*-bit keys**

Step	Kernel	Purpose	Read Workload	Write Workload
1	*binning*	Create flags	*n* keys	*nr* flags
2	*bottom-level reduce*	Compact flags (scan primitive)	*nr* flags	(*insignificant constant*)
3	*top-level scan*		(*insignificant constant*)	(*insignificant constant*)
4	*bottom-level scan*		*nr* flags + (*insignificant constant*)	*nr* offsets
5	*scatter*	Distribute keys	*n* offsets + *n* keys (+ *n* values)	*n* keys (+ *n* values)

Total Memory Workload: $(k/d)(n)(r + 4)$ keys only
$(k/d)(n)(r + 6)$ with values

Parallel Radix Sort

- **Assign tile of data to each block (1024 elements)**

Satish uses 256-thread blocks and 4 elements per thread

- **Build per-block histograms of current digit (4 bit)**

this is a reduction

- **Combine per-block histograms (P x 16)**

this is a scan

- **Scatter**

Per-Block Histograms

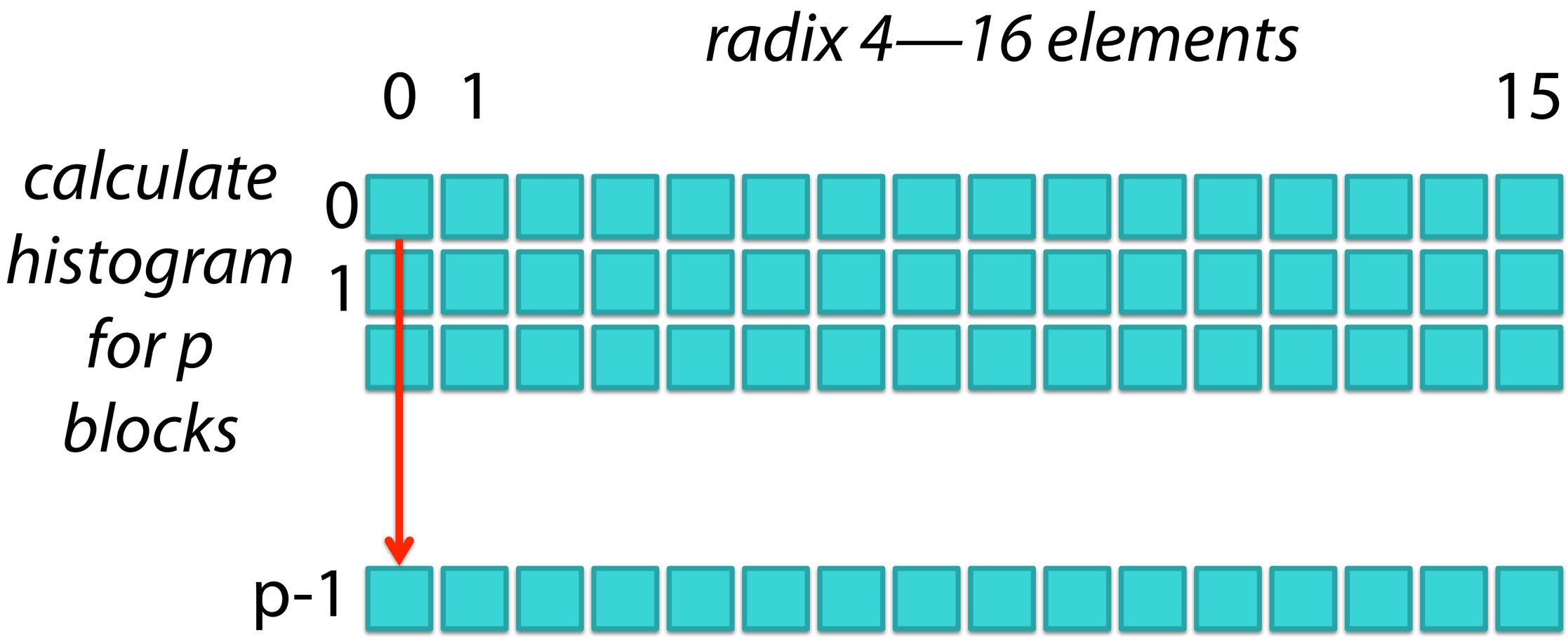
- Perform *b* parallel splits for *b*-bit digit
  - This problem is known in the literature as *multisplit*

- Each split is just a prefix sum of bits
  - each thread counts 1 bits to its left

- Write bucket counts & partially sorted tile
  - sorting tile improves scatter coherence later

Combining Histograms

- **Write per-block counts in column major order & scan**



cf. Zagha & Blelloch, *Radix sort for vector multiprocessors*, SC'91.

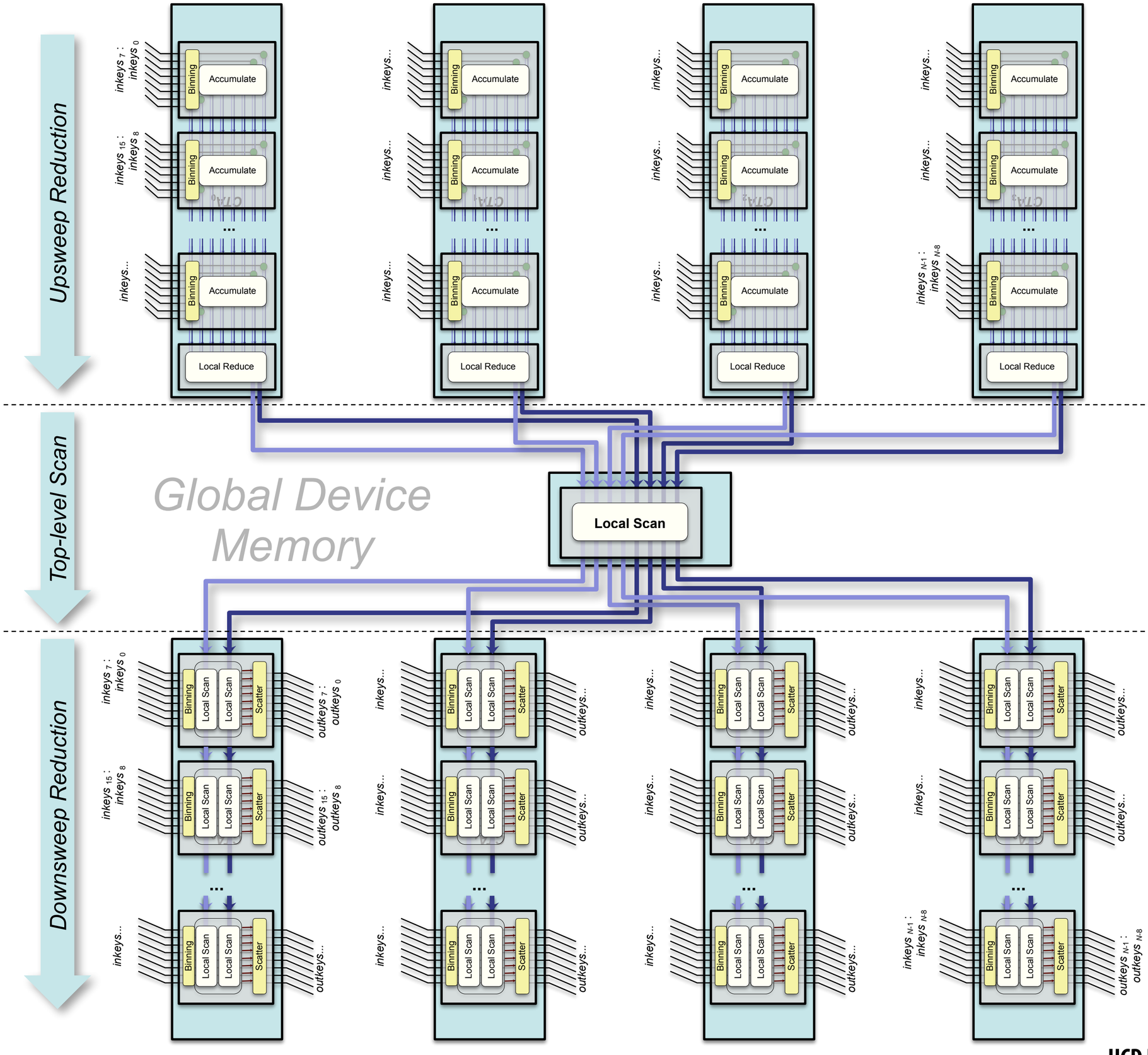
Satish's Radix Sort Memory Cost

- *d*-bit radix digits
- radix $r = 2^d$
- *n*-element input sequence of *k*-bit keys
- *b* bits per step

Step	Kernel	Purpose	Read Workload	Write Workload
1	*local digit-sort*	Maximize coherence	*n* keys (+ *n* values)	*n* keys (+ *n* values)
2	*histogram*	Create histograms	*n* keys	*nr/b* counts
3	*bottom-level reduce*	Scan histograms (scan primitive)	*nr/b* counts	(*insignificant constant*)
4	*top-level scan*		(*insignificant constant*)	(*insignificant constant*)
5	*bottom-level scan*		*nr/b* counts + (*insignificant constant*)	*nr/b* offsets
6	*scatter*	Distribute keys	*nr/b* offsets + *n* keys (+ *n* values)	*n* keys (+ *n* values)

Total Memory Workload: $(k/d)(n)(5r/b + 7)$ keys only
$(k/d)(n)(5r/b + 9)$ with values

Merrill's 3-step sort



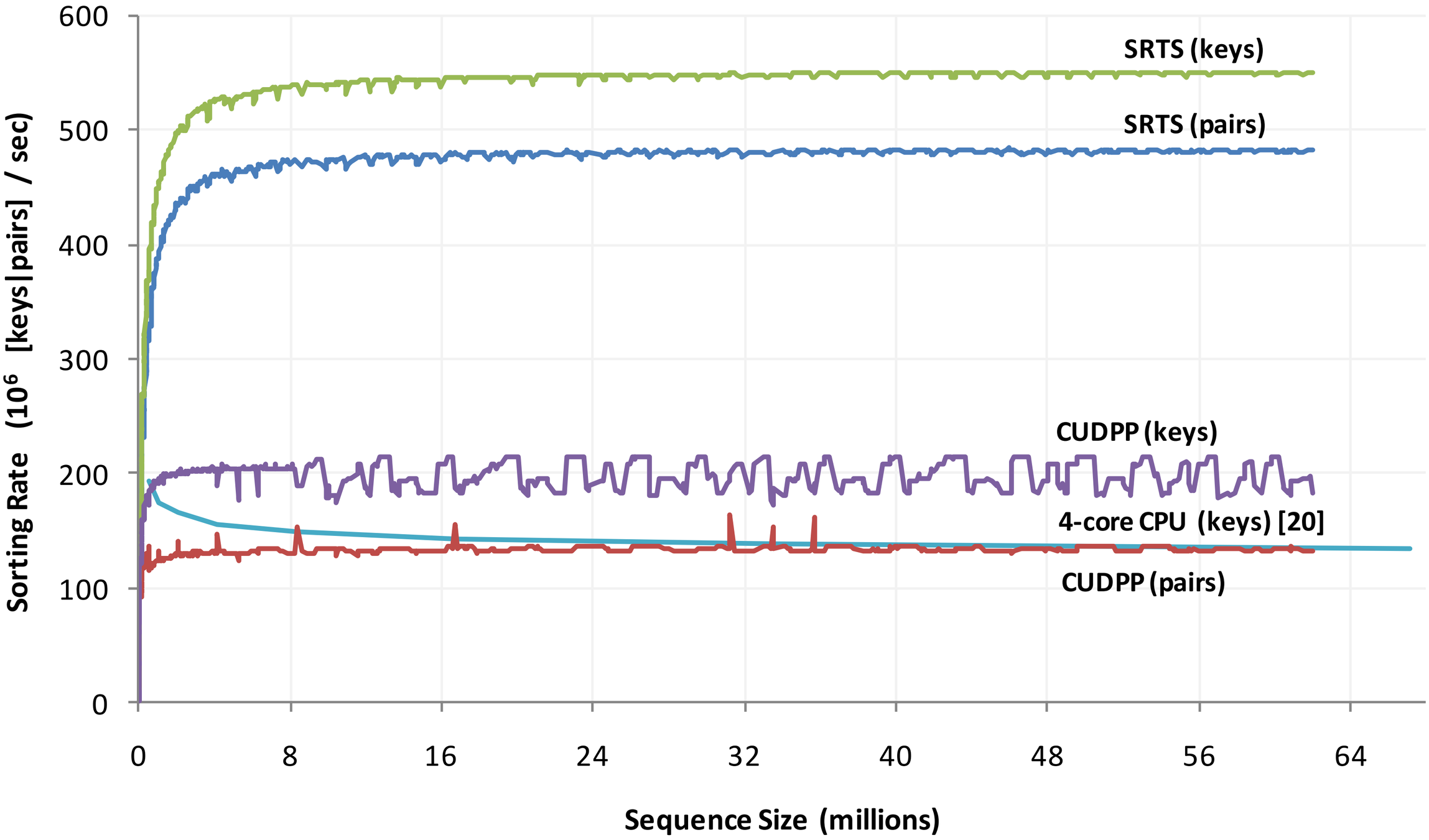
Merrill's sort, costs

- *d*-bit radix digits
- radix $r = 2^d$
- *n*-element input sequence of *k*-bit keys
- Current GPUs use *d*=4 (higher values exhaust local storage)

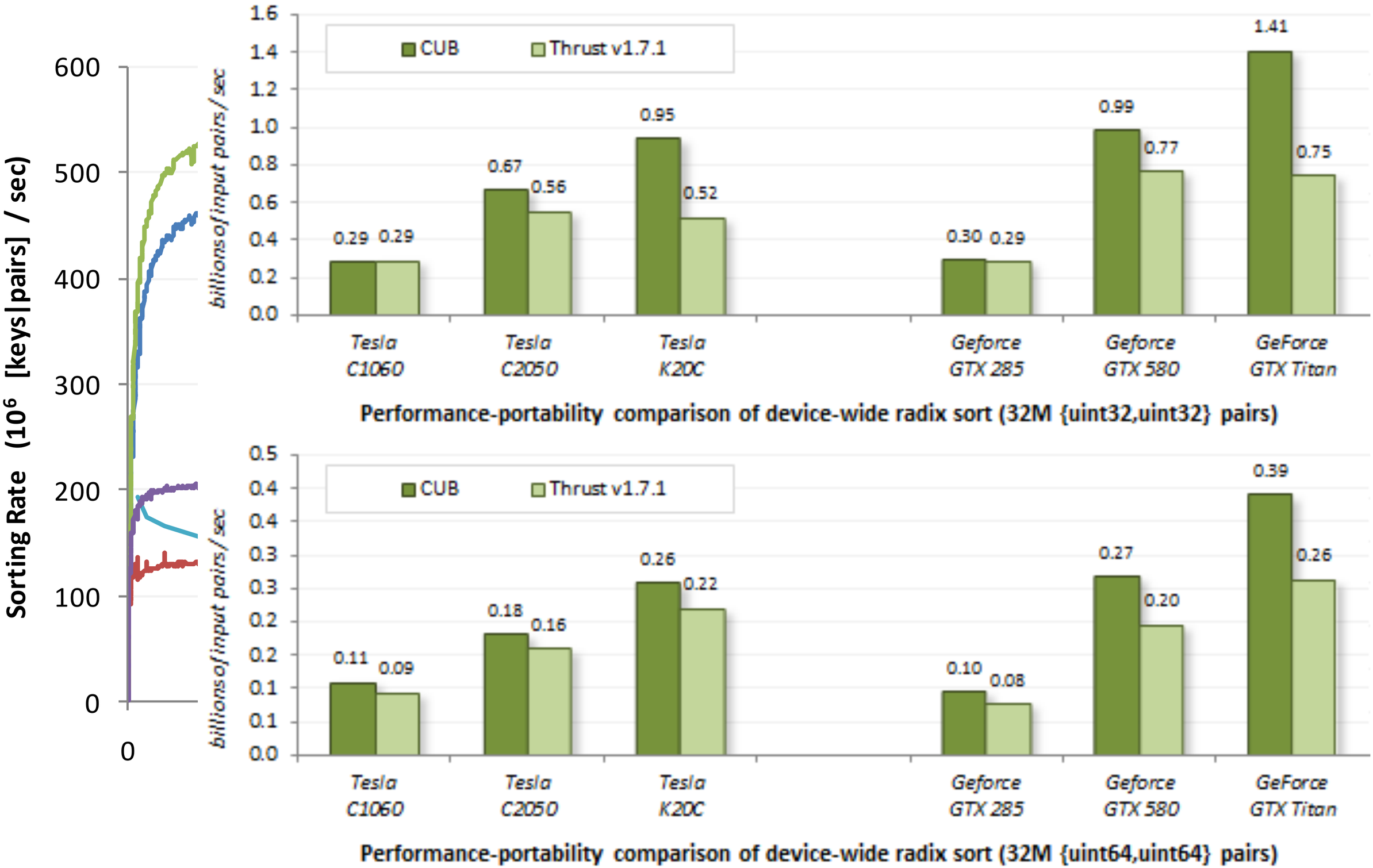
Step	Kernel	Purpose	Read Workload	Write Workload
1	*bottom-level reduce*	Create flags, compact flags, scatter keys	*n* keys	(*insignificant constant*)
2	*top-level scan*		(*insignificant constant*)	(*insignificant constant*)
3	*bottom-level scan*		*n* keys (+ *n* values) + (*insignificant constant*)	*n* keys (+ *n* values)

Total Memory Workload: (*k*/*d*)(3*n*) keys only
(*k*/*d*)(5*n*) with values

Results (NVIDIA GTX 285)



Results (NVIDIA GTX 285)



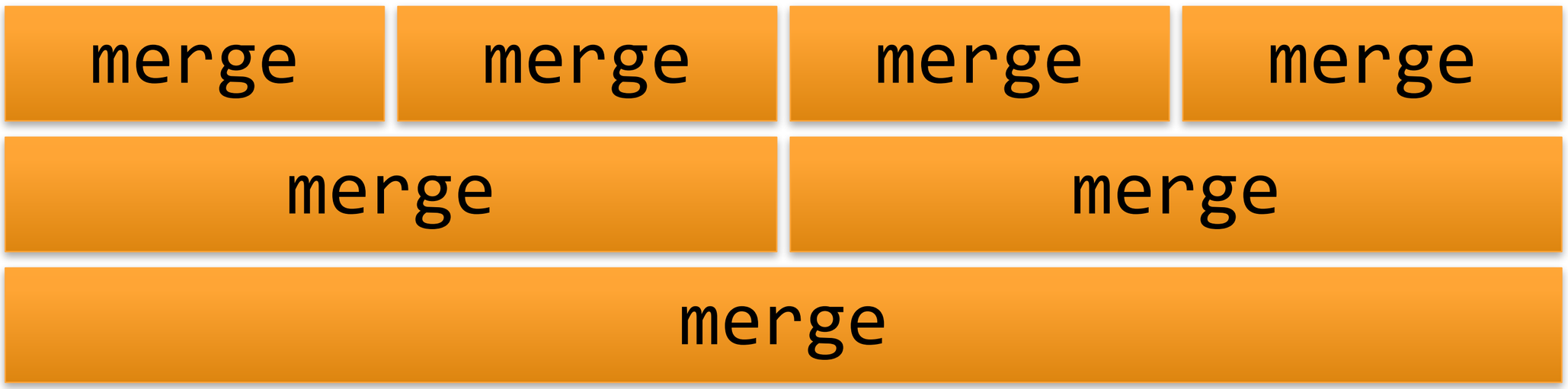
Merge Sort

Merge Sort

- Divide input array into 256-element tiles

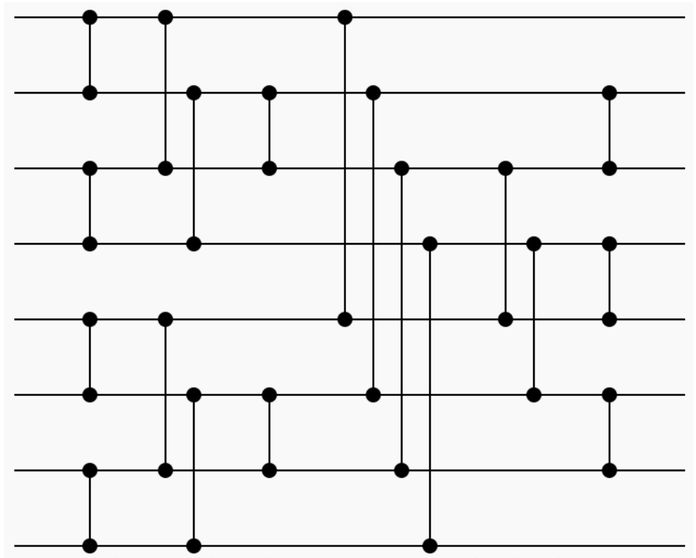
- Sort each tile independently

- Produce sorted output with tree of merges



Sorting a Tile

- Tiles are sized so that:
  - a single thread block can sort them efficiently
  - they fit comfortably in on-chip memory

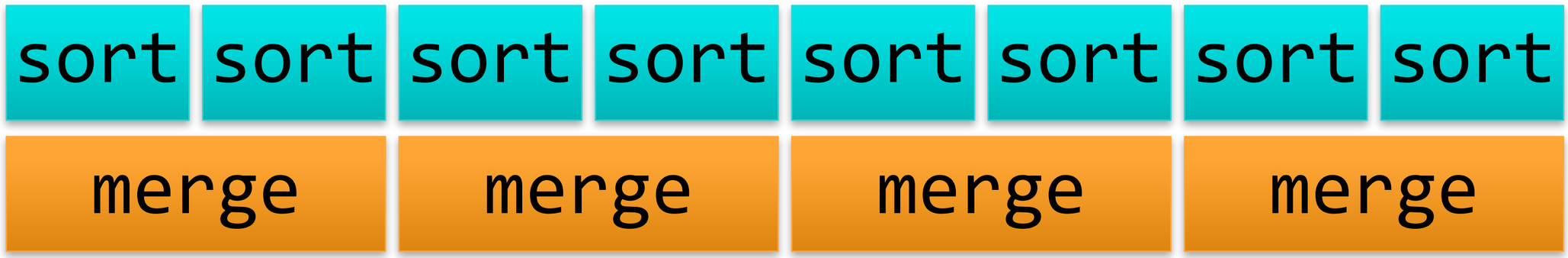


- Sorting networks are most efficient in this regime
  - odd-even merge sort
  - about 5–10% faster than comparable bitonic sort

- Caveat: sorting networks may reorder equal keys

Merging Pairs of Sorted Tiles

- Launch 1 thread block to process each pair of tiles

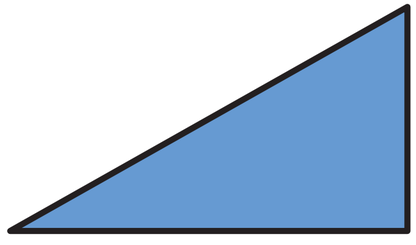


- Load tiles into on-chip memory

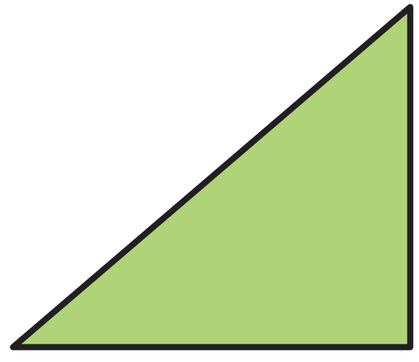
- Perform counting merge

- Stored merged result to global memory

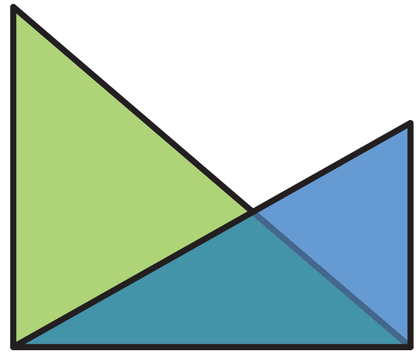
My grad-student-days merge



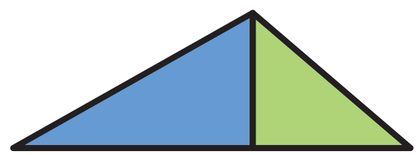
Sorted
Input A



Sorted
Input B

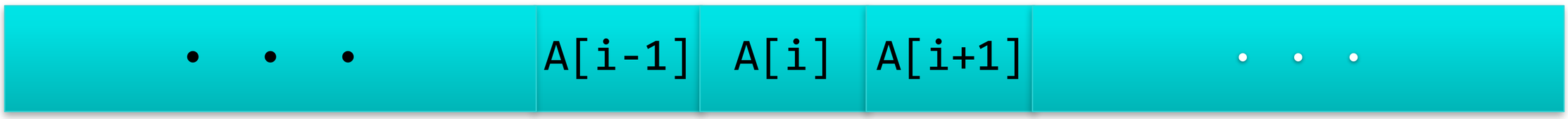


Flip B, pairwise compare to A



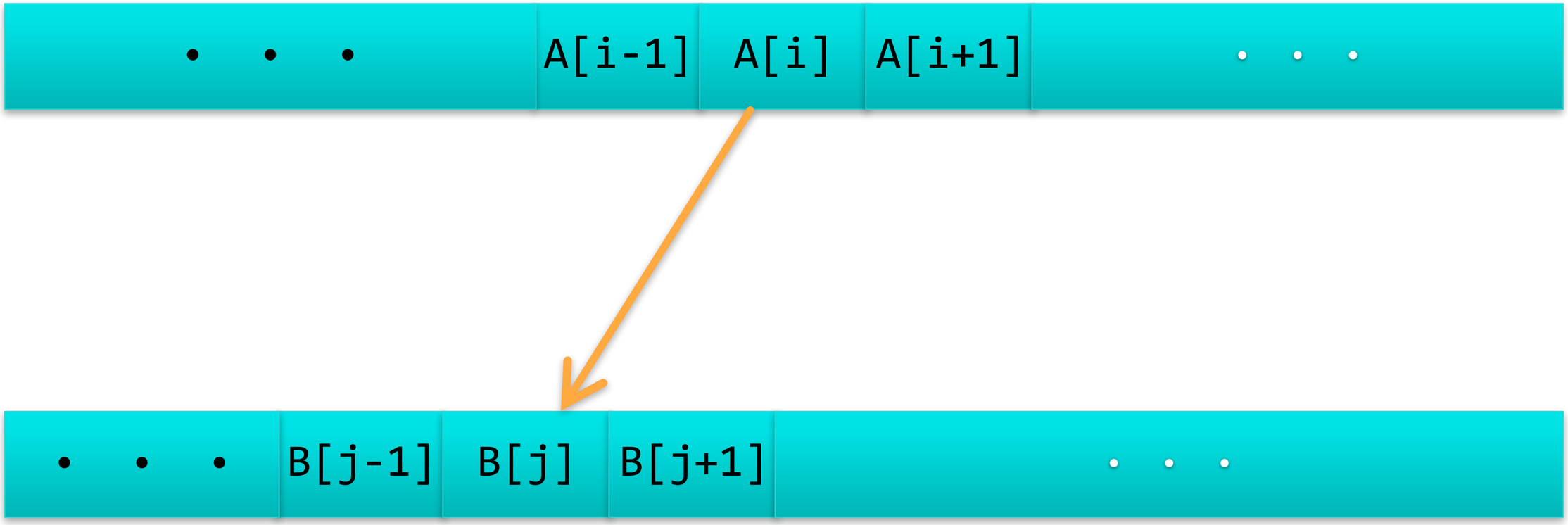
Smallest element in each comparison yields smallest p elements overall in a bitonic sequence

Counting Merge



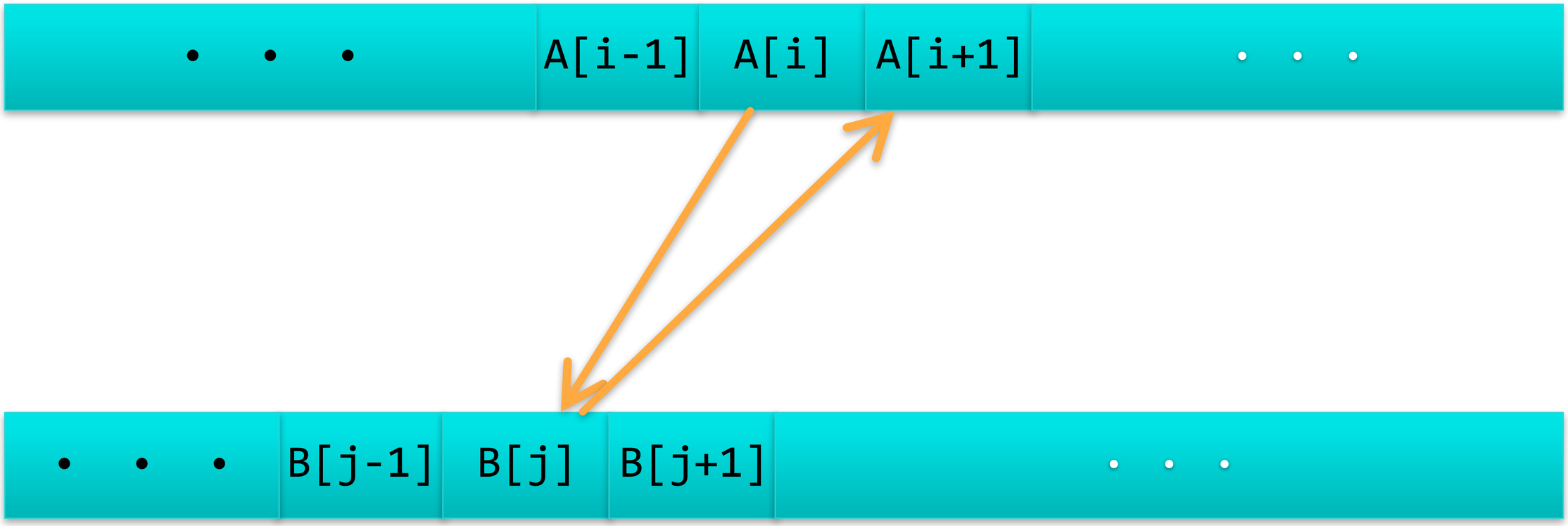
Counting Merge

upper_bound(A[i], B) = count(j where A[i] ≤ B[j])



Counting Merge

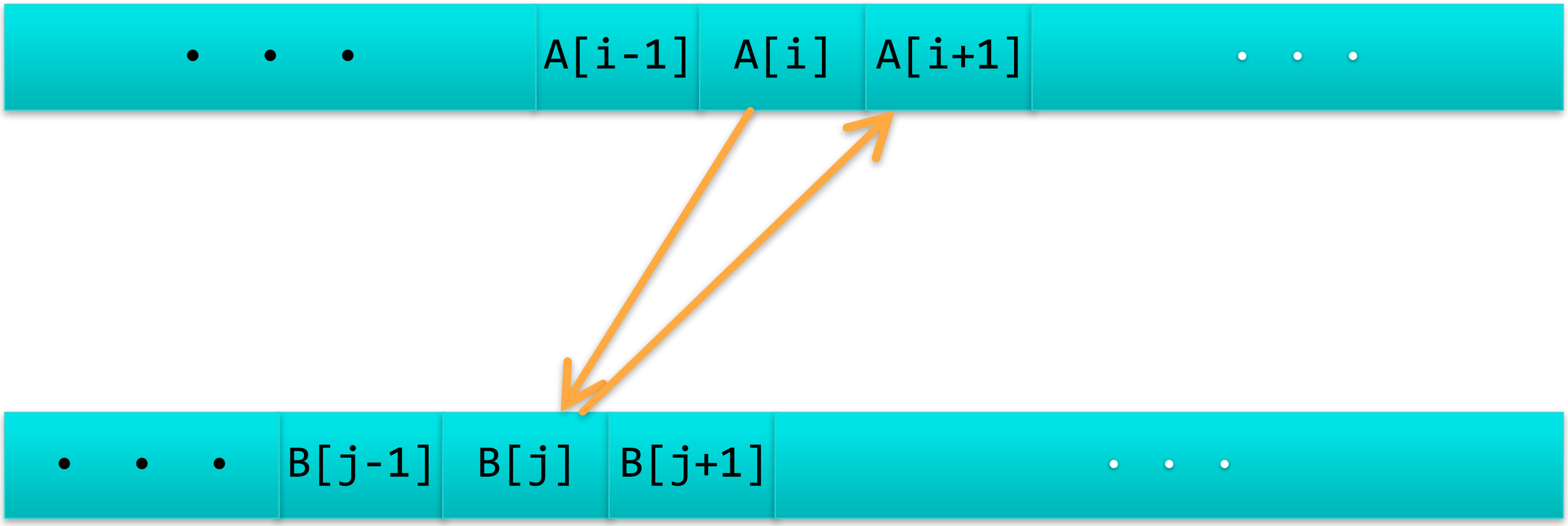
upper_bound(A[i], B) = count(j where A[i] ≤ B[j])



lower_bound(B[j], A) = count(i where B[j] < A[i])

Counting Merge

upper_bound(A[i], B) = count(j where A[i] ≤ B[j])

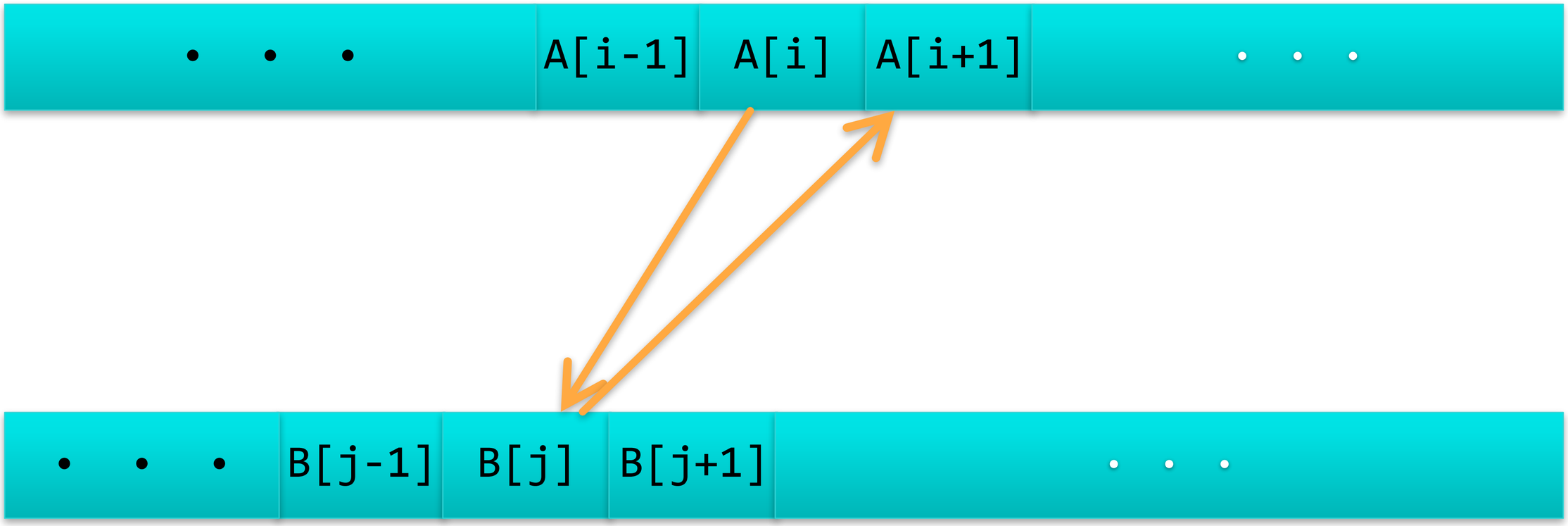


lower_bound(B[j], A) = count(i where B[j] < A[i])

Use binary search since A & B are sorted

Counting Merge

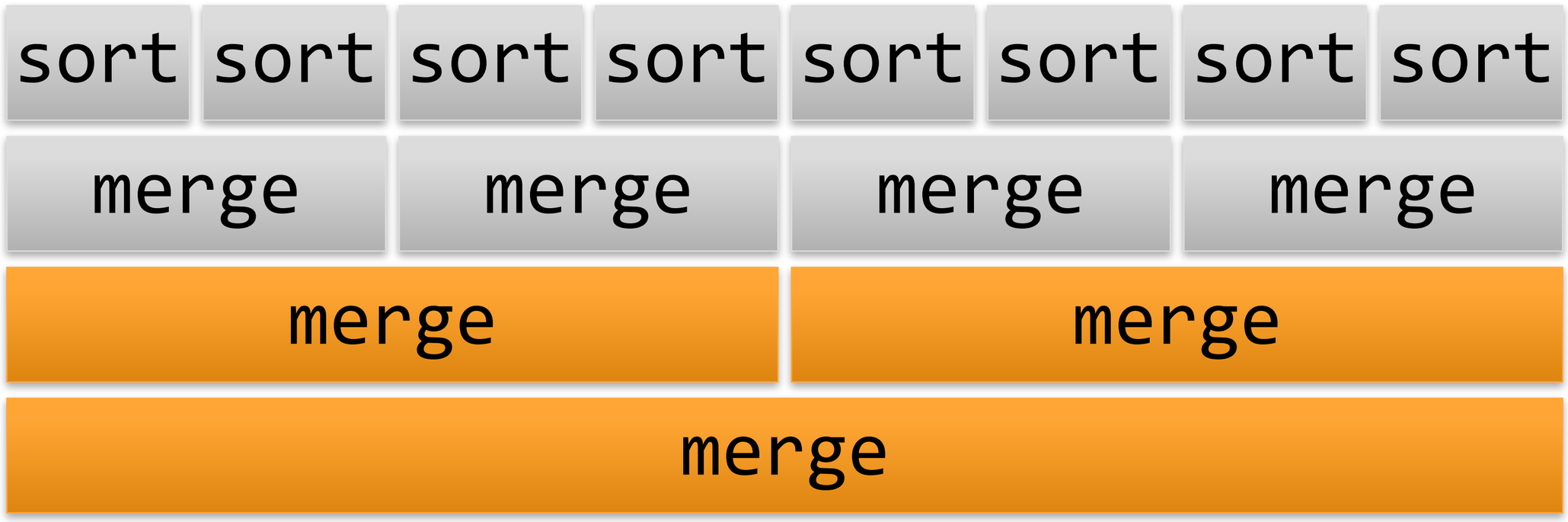
upper_bound(A[i], B) = count(j where A[i] ≤ B[j])



lower_bound(B[j], A) = count(i where B[j] < A[i])

```
scatter( A[i] -> C[i + upper_bound(A[i], B)] )
scatter( B[j] -> C[lower_bound(B[j], A) + j] )
```

Merging Larger Subsequences



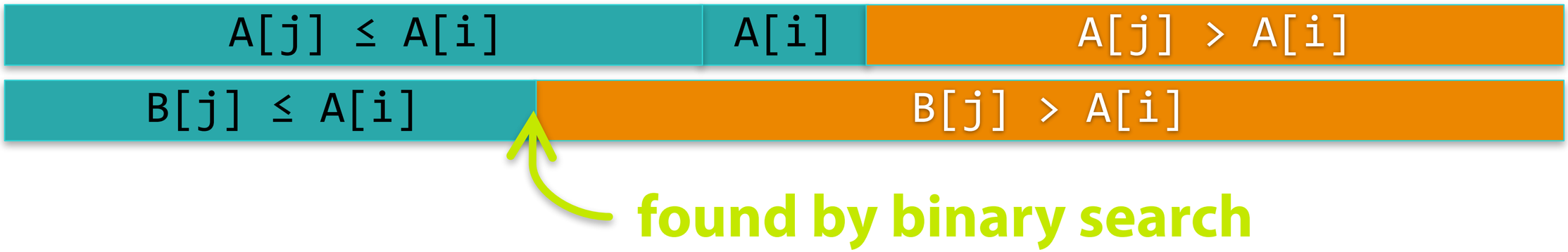
- Partition larger sequences into collections of tiles

- Apply counting merge to each pair of tiles

Two-way Partitioning Merge

- Pick a splitting element from either A or B

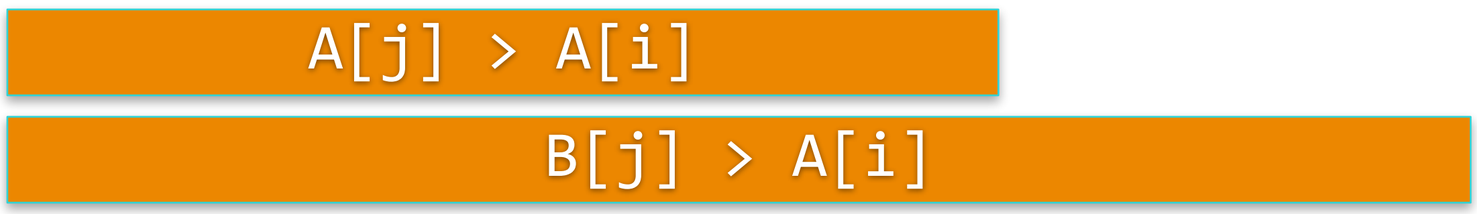
- Divide A and B into elements below/above splitter



- Recurse

merge :

merge :



Multi-way Partitioning Merge

- Pick every 256th element of A & B as splitter

- Apply merge recursively to merge splitter sets
  - recursively apply merge procedure

- Split A & B with merged splitters

- Merge resulting pairs of tiles (at most 256 elements)

Sort papers

- 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.
- N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," IPDPS 2009: IEEE International Symposium on Parallel & Distributed Processing, May 2009.
- D. Merrill and A. Grimshaw, Revisiting Sorting for GPGPU Stream Architectures. Technical Report CS2010-03, Department of Computer Science, University of Virginia, 2010, 17pp.