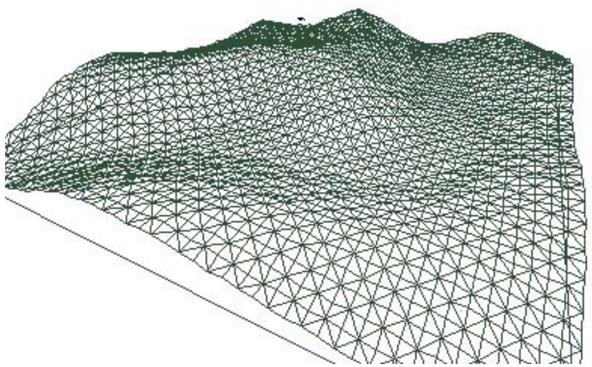
Parallel Algorithms and Data Structures
CS 448s, Stanford University
20 April 2010

John Owens
Associate Professor, Electrical and Computer Engineering
UC Davis

Data-Parallel Algorithms

- Efficient algorithms require efficient building blocks
- Five data-parallel building blocks
  - Map
  - Gather & Scatter
  - Reduce
  - Scan
  - Sort
- Advanced data structures:
  - Sparse matrices
  - Hash tables
  - Task queues

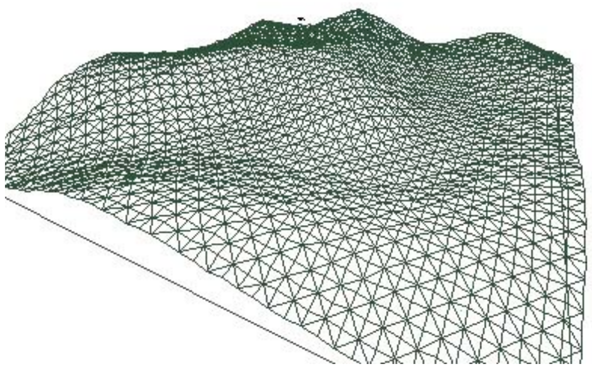
Sample Motivating Application



- How bumpy is a surface that we represent as a grid of samples?
- Algorithm:
  - Loop over all elements
  - At each element, compare the value of that element to the average of its neighbors (“difference”). Square that difference.
  - Now sum up all those differences.
    - But we don’t want to sum all the diffs that are 0.
    - So only sum up the non-zero differences.
  - This is a fake application—don’t take it too seriously.

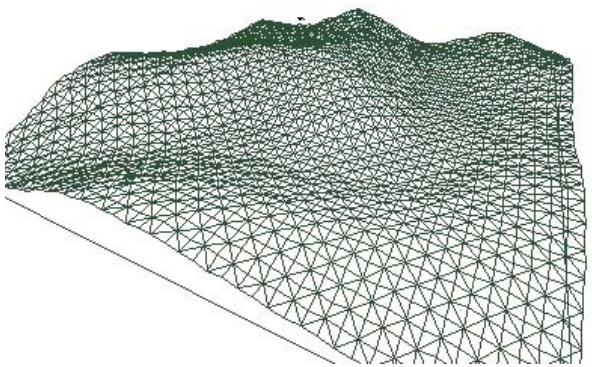
Picture courtesy http://www.artifice.com

Sample Motivating Application



```
for all samples:

    neighbors[x,y] =
          0.25 * (  value[x-1,y]+
                    value[x+1,y]+
                    value[x,y+1]+
                    value[x,y-1] ) )

    diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

    result += diff

return result
```

Sample Motivating Application

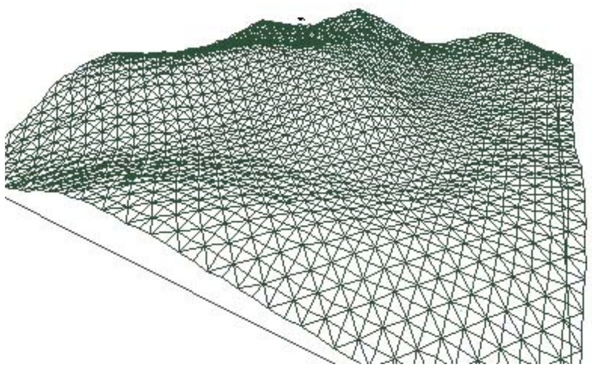


```
for all samples:

    neighbors[x,y] =
          0.25 * (  value[x-1,y]+
                    value[x+1,y]+
                    value[x,y+1]+
                    value[x,y-1] ) )

    diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

    result += diff

return result
```

The Map Operation

- Given:
  - Array or stream of data elements *A*
  - Function $f(x)$
- $\text{map}(A, f)$ = applies $f(x)$ to all $a_i \in A$
- How does this map to a data-parallel processor?

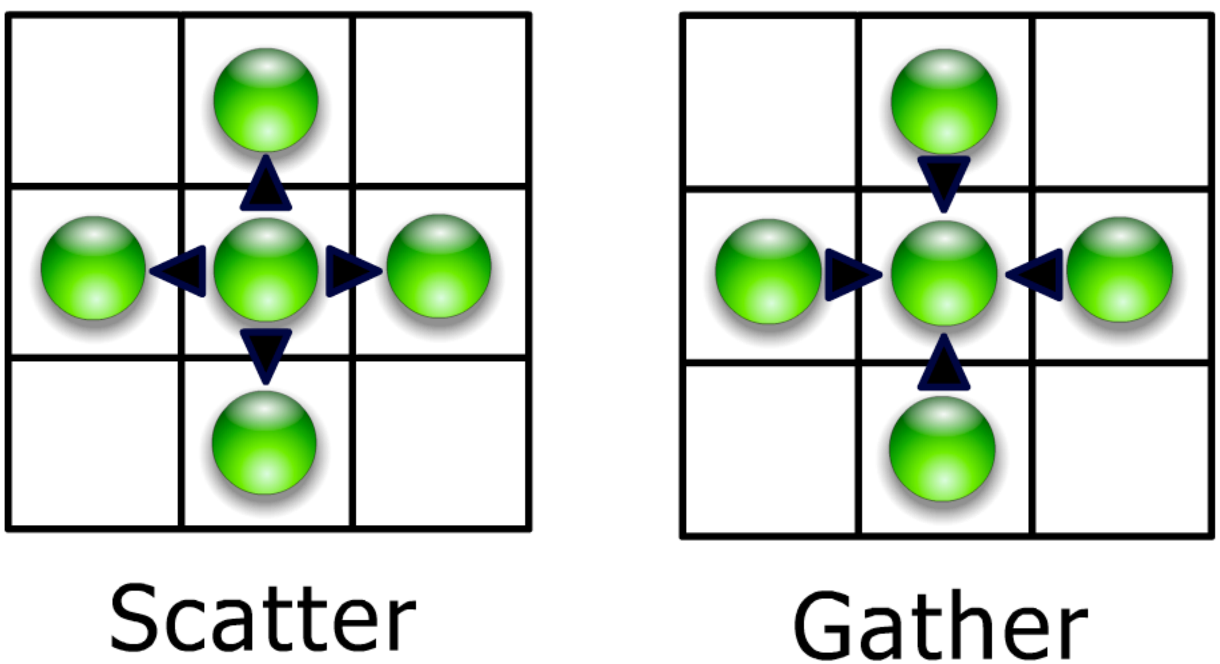
Sample Motivating Application



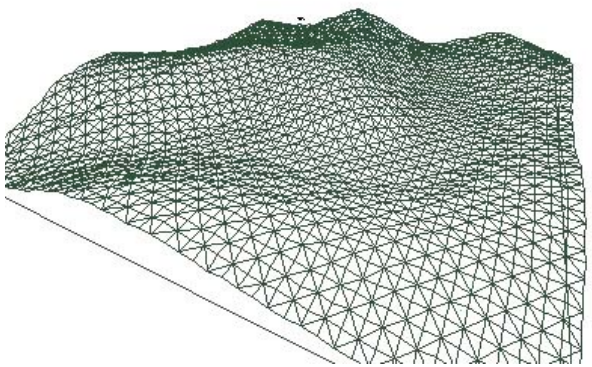
```
for all samples:

    neighbors[x,y] =
         0.25 * (  value[x-1,y]+
                   value[x+1,y]+
                   value[x,y+1]+
                   value[x,y-1] ) )

    diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

    result += diff

return result
```

Scatter vs. Gather

- Gather: `p = a[i]`
- Scatter: `a[i] = p`
- How does this map to a data-parallel processor?



Sample Motivating Application



```
for all samples:

    neighbors[x,y] =
           0.25 * (  value[x-1,y]+
                     value[x+1,y]+
                     value[x,y+1]+
                     value[x,y-1] ) )

    diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

    result += diff

return result
```

Parallel Reductions

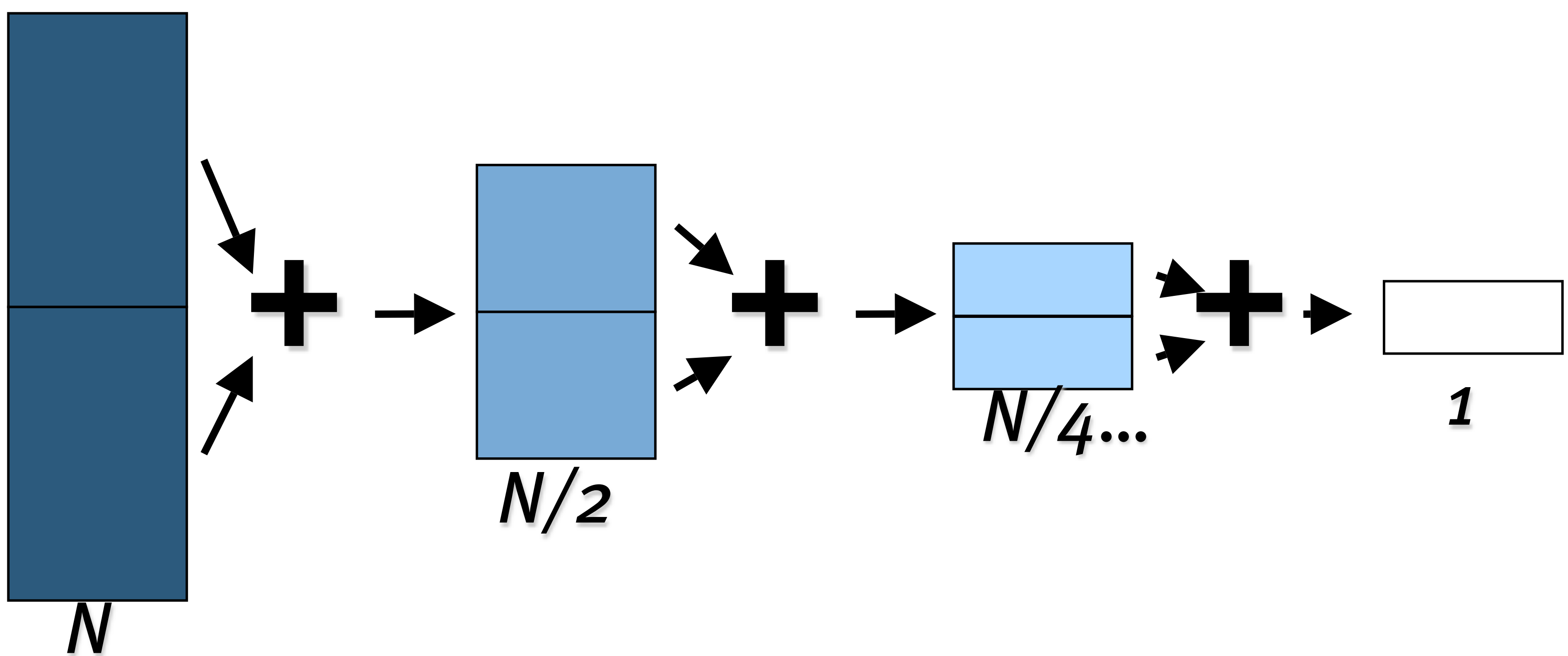
- Given:
  - Binary associative operator $\oplus$ with identity I
  - Ordered set $s = [a_0, a_1, \ldots, a_{n-1}]$ of n elements
- reduce($\oplus$, s) returns $a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}$
- Example:
  reduce(+, [3 1 7 0 4 1 6 3]) = 25
- Reductions common in parallel algorithms
  - Common reduction operators are +, ×, min and max
  - Note floating point is only pseudo-associative

Efficiency

- Work efficiency:
  - Total amount of work done over all processors
- Step efficiency:
  - Number of steps it takes to do that work
- With parallel processors, sometimes you're willing to do more work to reduce the number of steps
- Even better if you can reduce the amount of steps and still do the same amount of work

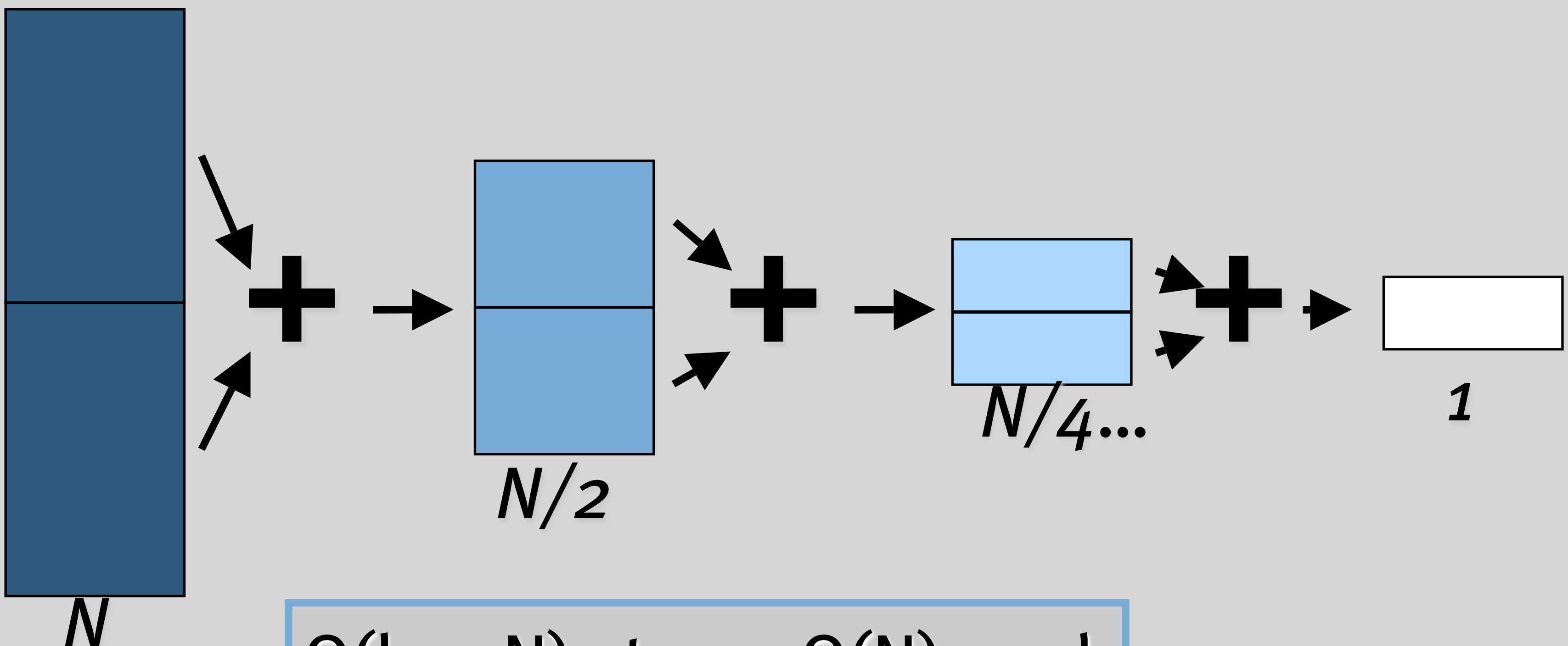
Parallel Reductions

- 1D parallel reduction:
  - add two halves of domain together repeatedly...
  - ... until we're left with a single row



Parallel Reductions

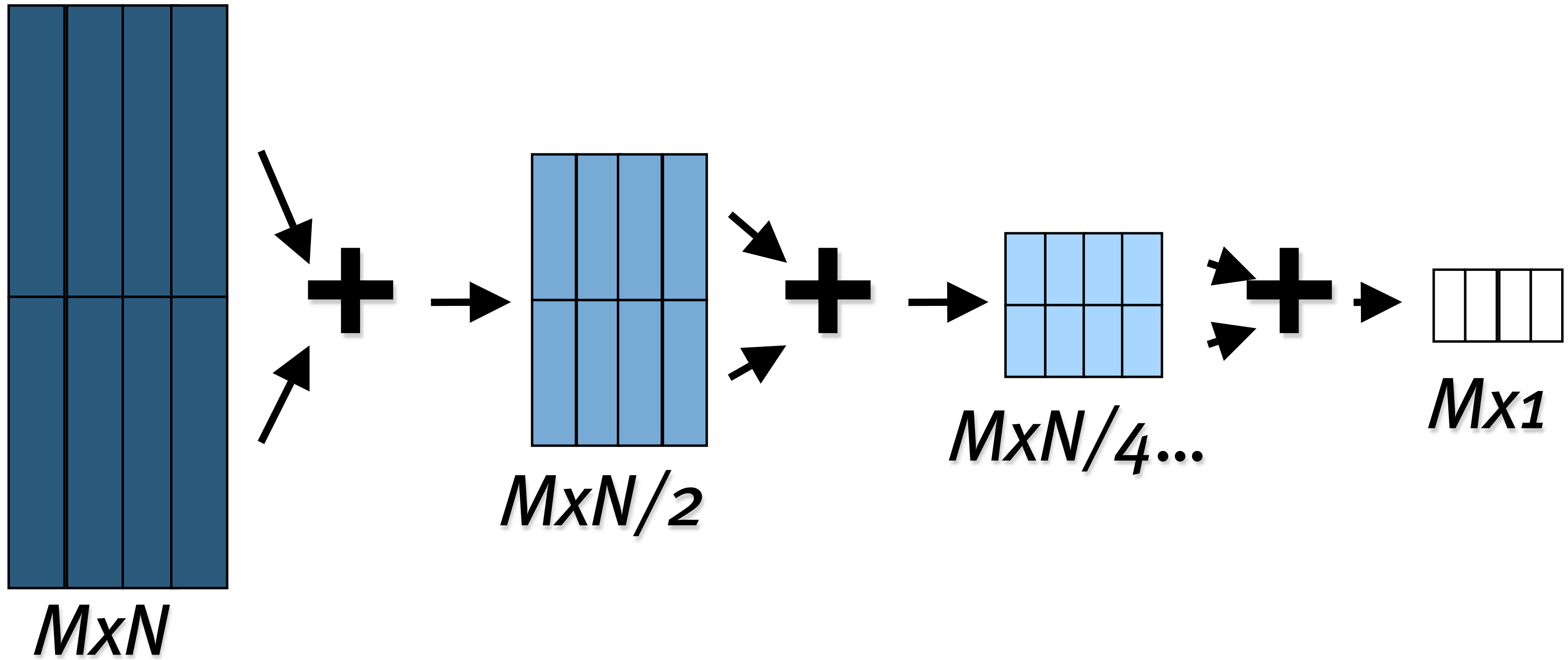
- 1D parallel reduction:
  - add two halves of domain together repeatedly...
  - ... until we're left with a single row



O($\log_2$N) steps, O(N) work

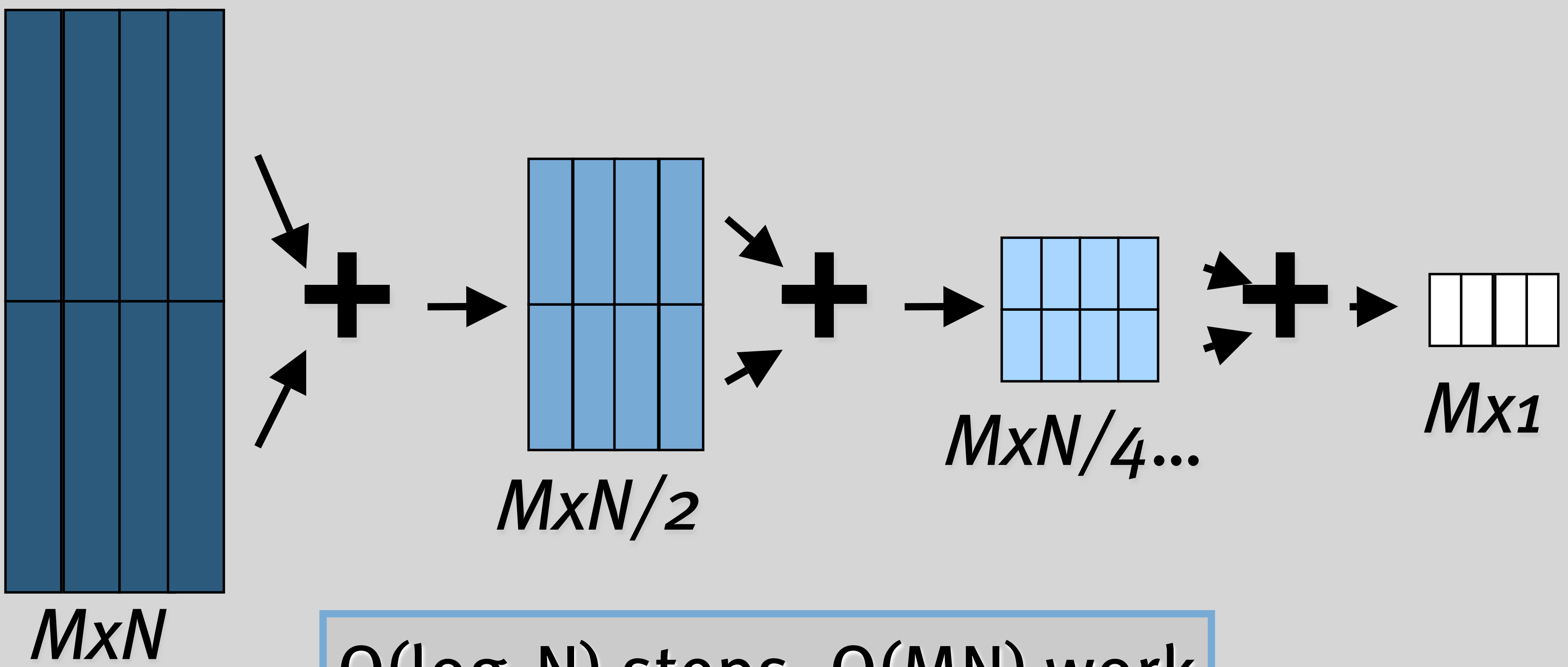
Multiple 1D Parallel Reductions

- Can run many reductions in parallel
- Use 2D grid and reduce one dimension



Multiple 1D Parallel Reductions

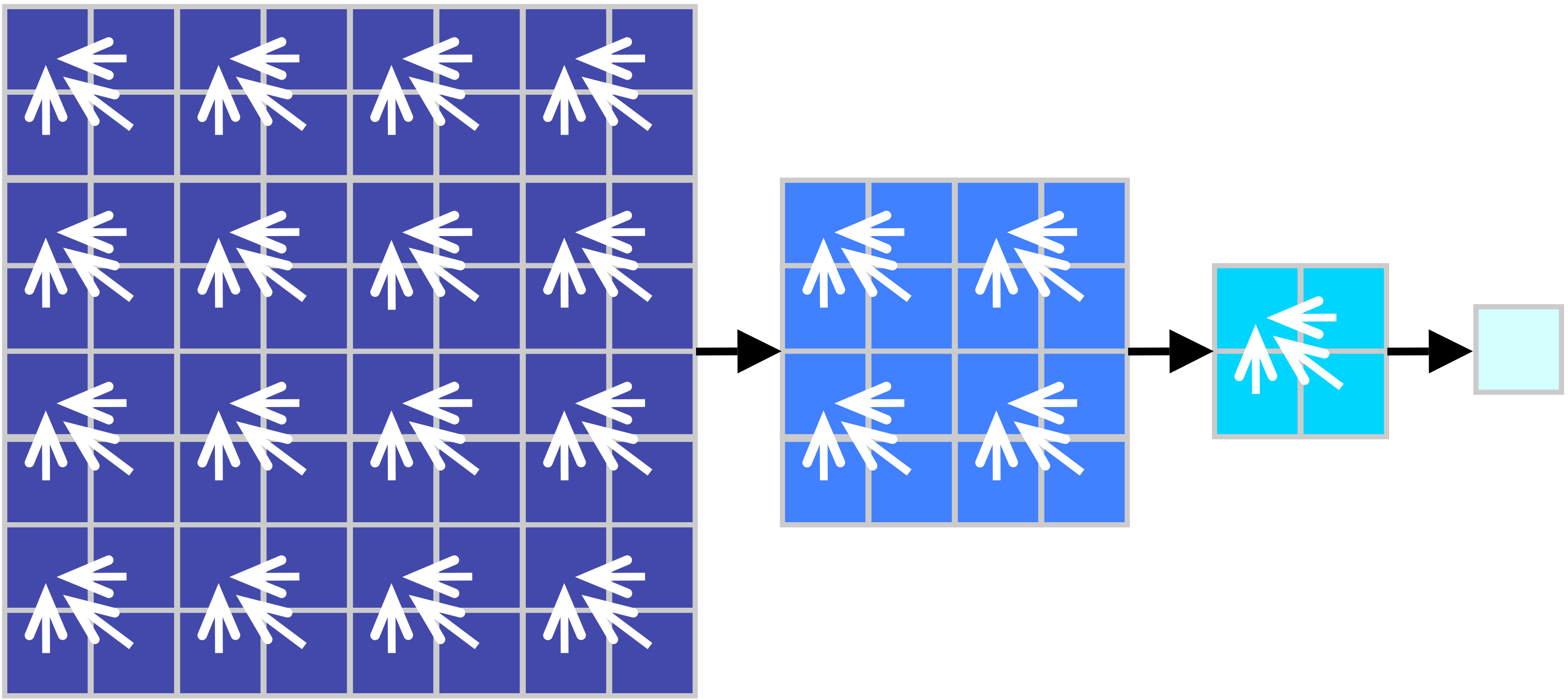
- Can run many reductions in parallel
- Use 2D grid and reduce one dimension



O($\log_2$N) steps, O(MN) work

2D reductions

- Like 1D reduction, only reduce in both directions simultaneously

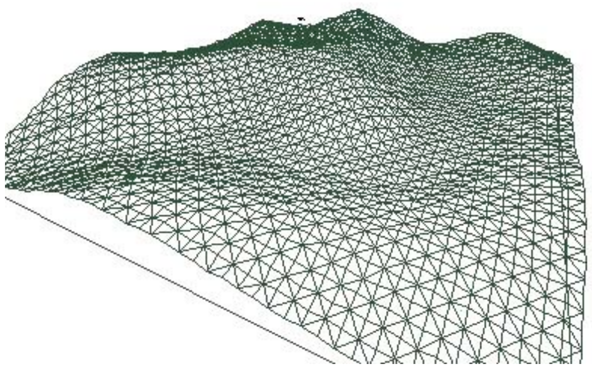


- Note: can add more than 2x2 elements per step
  - Trade per-pixel work for step complexity
  - Best perf depends on specific hardware (cache, etc.)

Parallel Reduction Complexity

- log(n) parallel steps, each step S does $n/2s$ independent ops
  - Step Complexity is $O(\log n)$
- Performs $n/2 + n/4 + \ldots + 1 = n-1$ operations
  - Work Complexity is $O(n)$—it is work-efficient
    - i.e. does not perform more operations than a sequential algorithm
- With p threads physically in parallel (p processors), time complexity is $O(n/p + \log n)$
  - Compare to $O(n)$ for sequential reduction

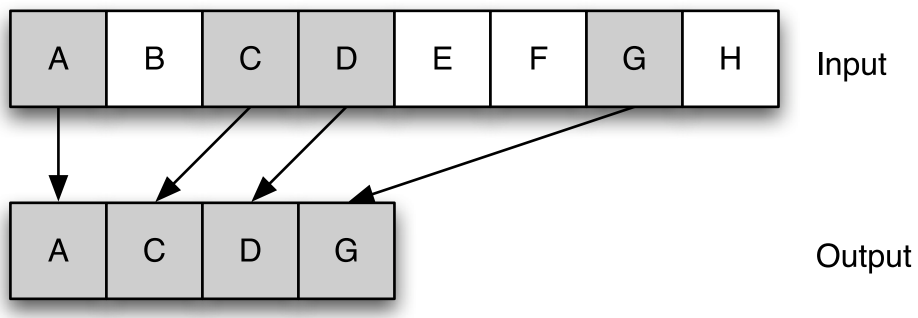
Sample Motivating Application



```
for all samples:

    neighbors[x,y] =
          0.25 * (  value[x-1,y]+
                    value[x+1,y]+
                    value[x,y+1]+
                    value[x,y-1] ) )

    diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

    result += diff

return result
```

Stream Compaction

- Input: stream of 1s and 0s
  [1 0 1 1 0 0 1 0]
- Operation: "sum up all elements before you"
- Output: scatter addresses for "1" elements
  [0 1 1 2 3 3 3 4]
- Note scatter addresses for red elements are packed!

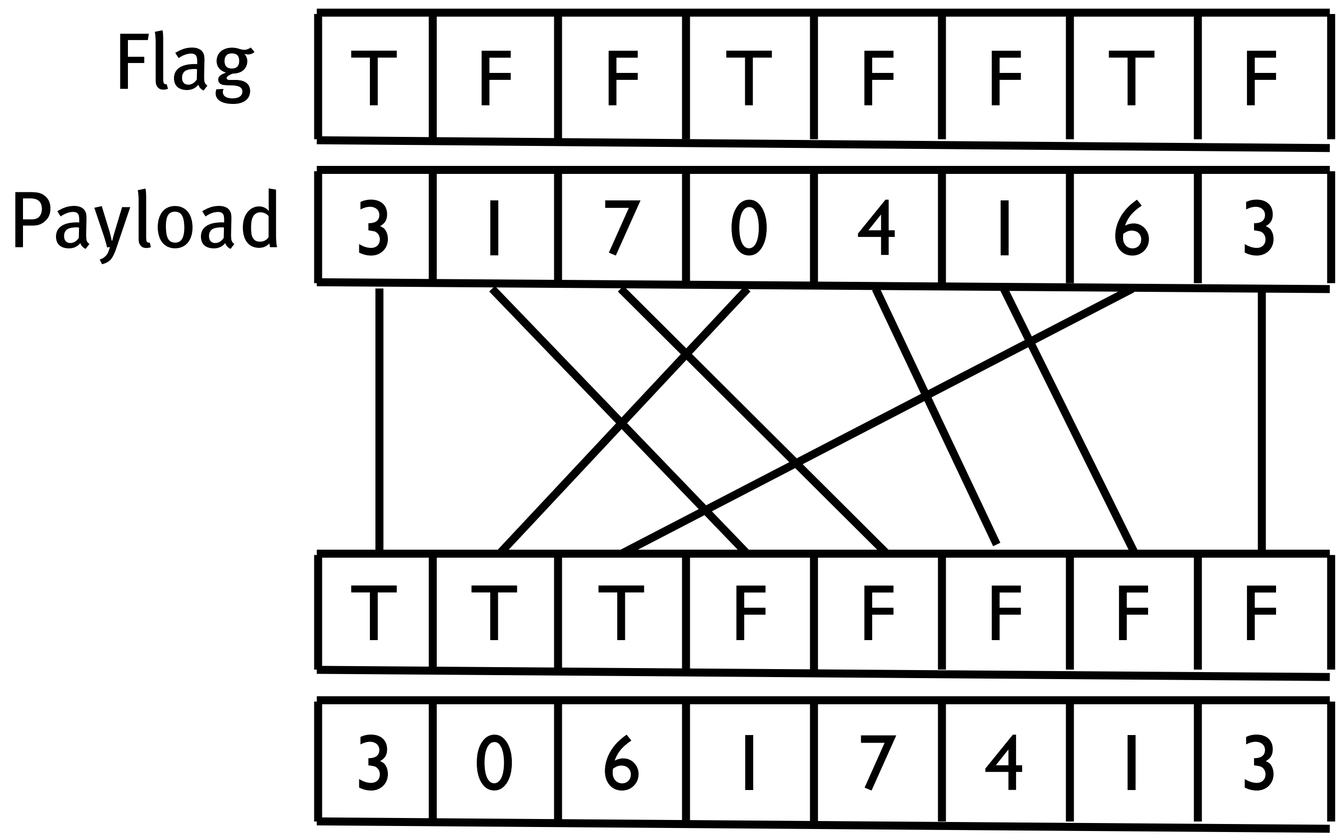


Common Situations in Parallel Computation

- Many parallel threads that need to partition data
  - Split
- Many parallel threads and variable output per thread
  - Compact / Expand / Allocate
- More complicated patterns than one-to-one or all-to-one
  - Instead all-to-all

Split Operation

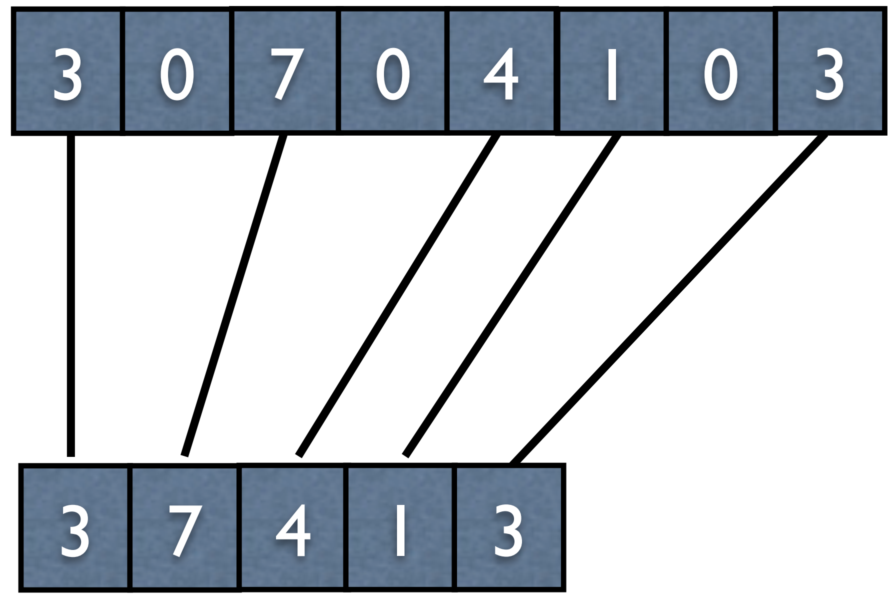
- Given an array of true and false elements (and payloads)



- Return an array with all true elements at the beginning
- Examples: sorting, building trees

Variable Output Per Thread: Compact

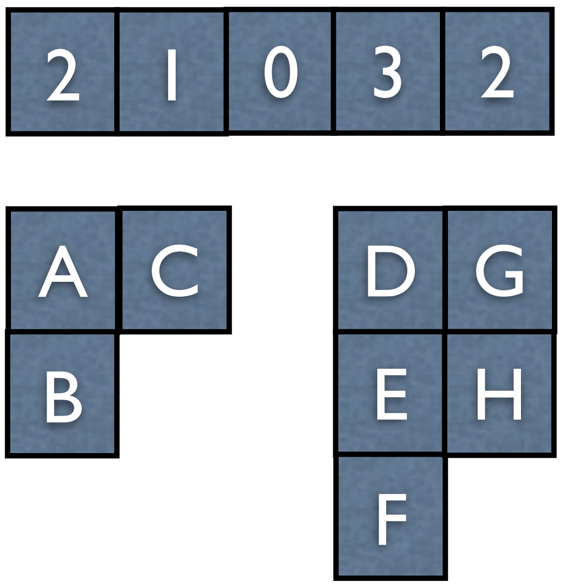
- Remove null elements



- Example: collision detection

Variable Output Per Thread

- Allocate Variable Storage Per Thread



- Examples: marching cubes, geometry generation

“Where do I write my output?”

- In all of these situations, each thread needs to answer that simple question
- The answer is:
- “That depends on how much the other threads need to write!”
  - In a serial processor, this is simple
- “Scan” is an efficient way to answer this question in parallel

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,

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

- 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]

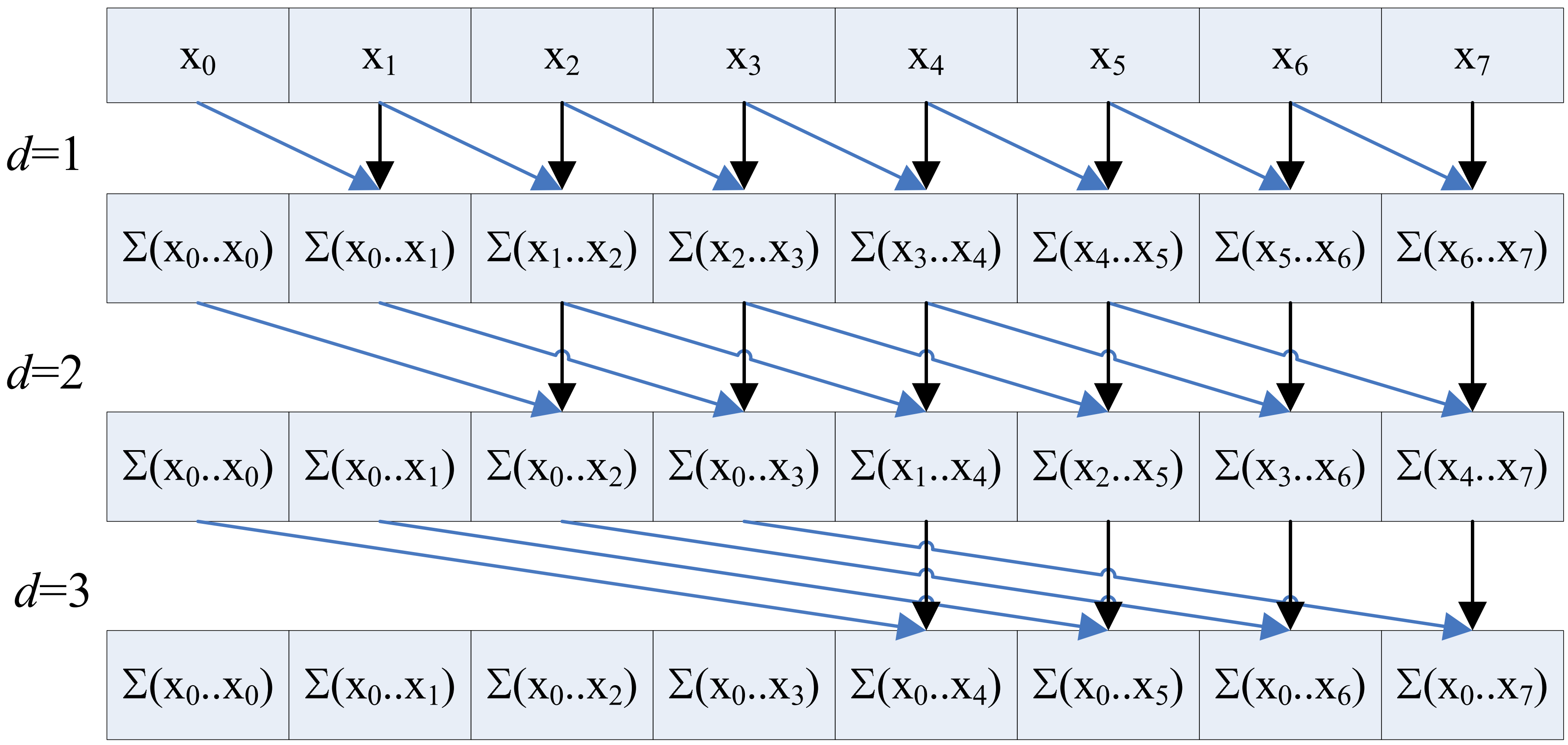
Segmented Scan

- Example: if $\oplus$ is addition, then scan on the set
  - [3 1 7 | 0 4 1 | 6 3]
- returns the set
  - [0 3 4 | 0 0 4 | 0 6]
- Same computational complexity as scan, but additionally have to keep track of segments (we use head flags to mark which elements are segment heads)
- Useful for *nested data parallelism* (quicksort)

Quicksort

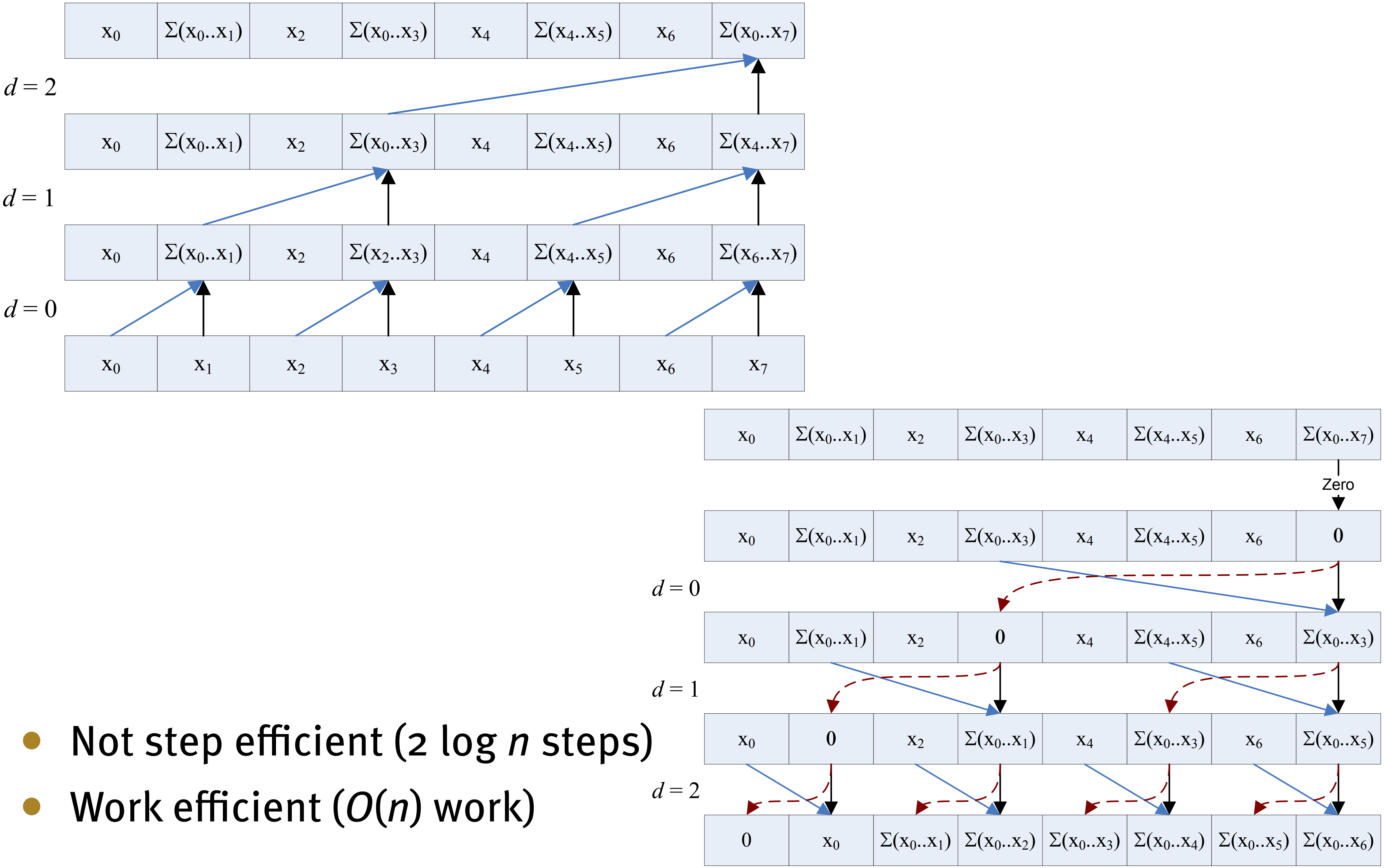
```
[5 3 7 4 6]   # initial input
[5 5 5 5 5]   # distribute pivot across segment
[f f t f t]   # input > pivot?
[5 3 4][7 6]  # split-and-segment
[5 5 5][7 7]  # distribute pivot across segment
[t f f][t f]  # input >= pivot?
[3 4 5][6 7]  # split-and-segment, done!
```

$O(n \log n)$ Scan



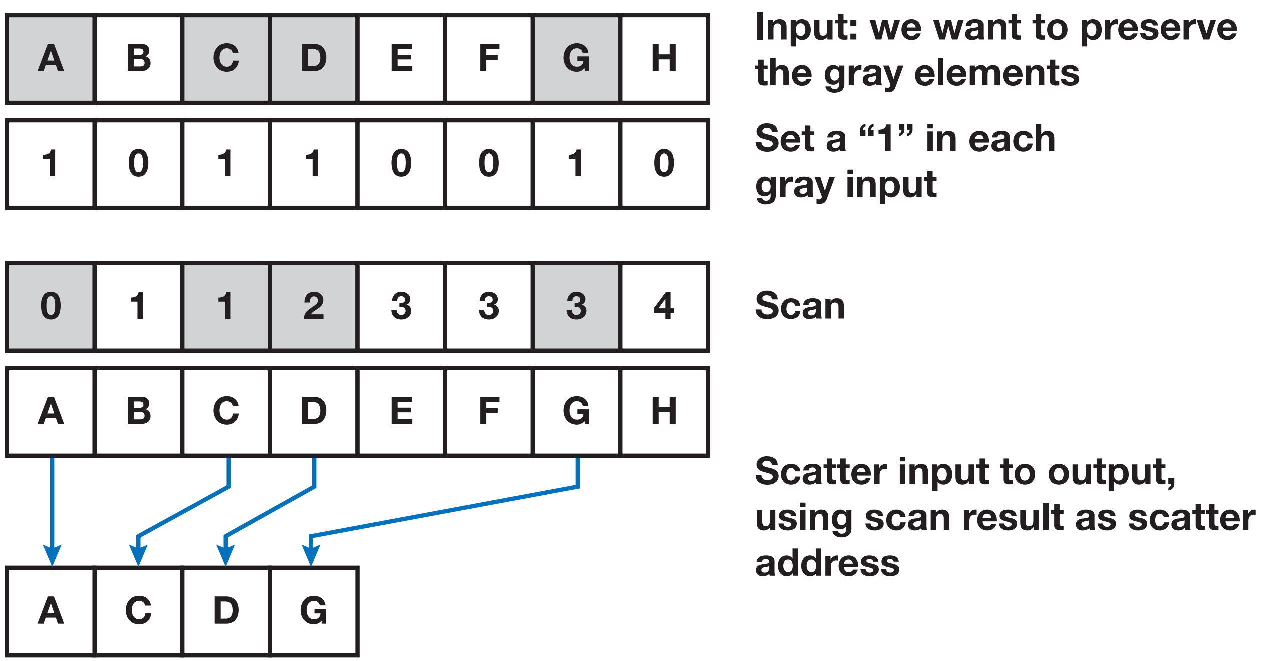
- Step efficient (log *n* steps)
- Not work efficient (*n* log *n* work)

$O(n)$ Scan



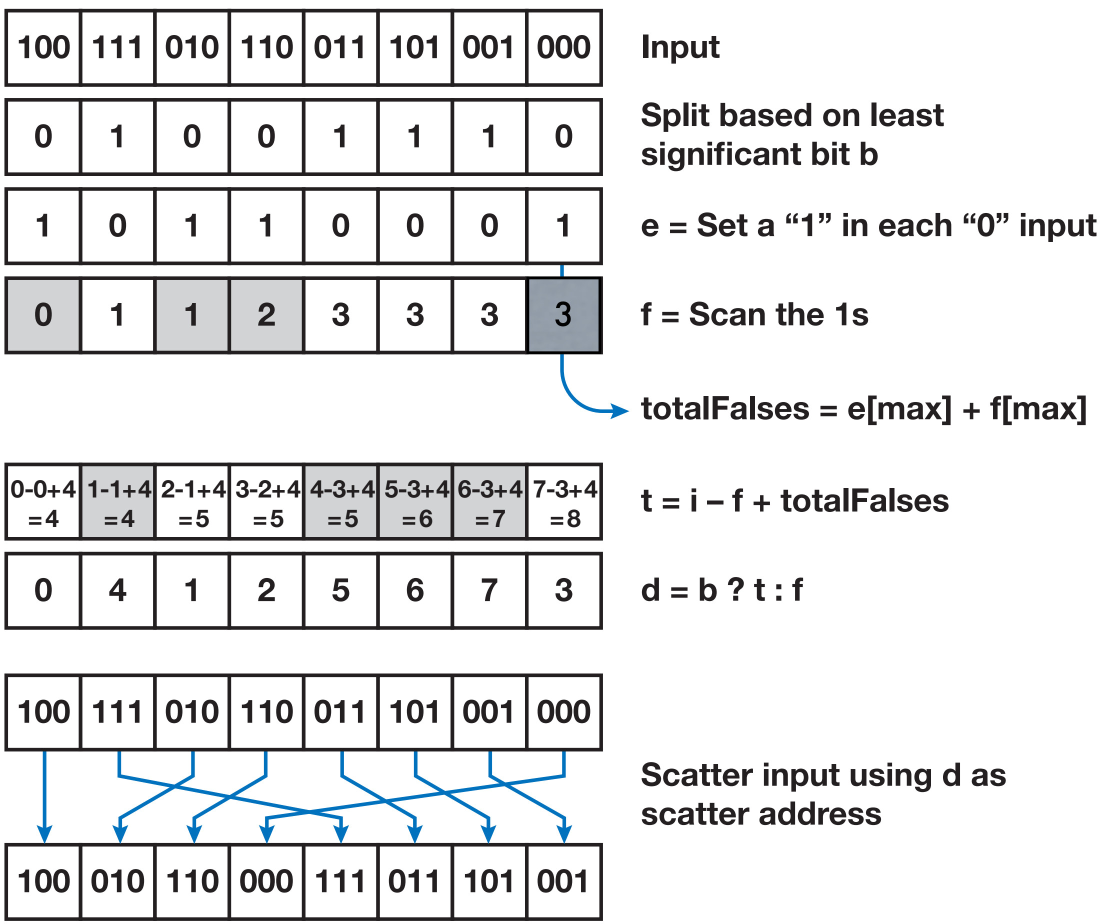
- Not step efficient (2 log n steps)
- Work efficient ($O(n)$ work)

Application: Stream Compaction



- 1M elements: ~0.6-1.3 ms
- 16M elements: ~8-20 ms
- Perf depends on # elements retained

Application: Radix Sort



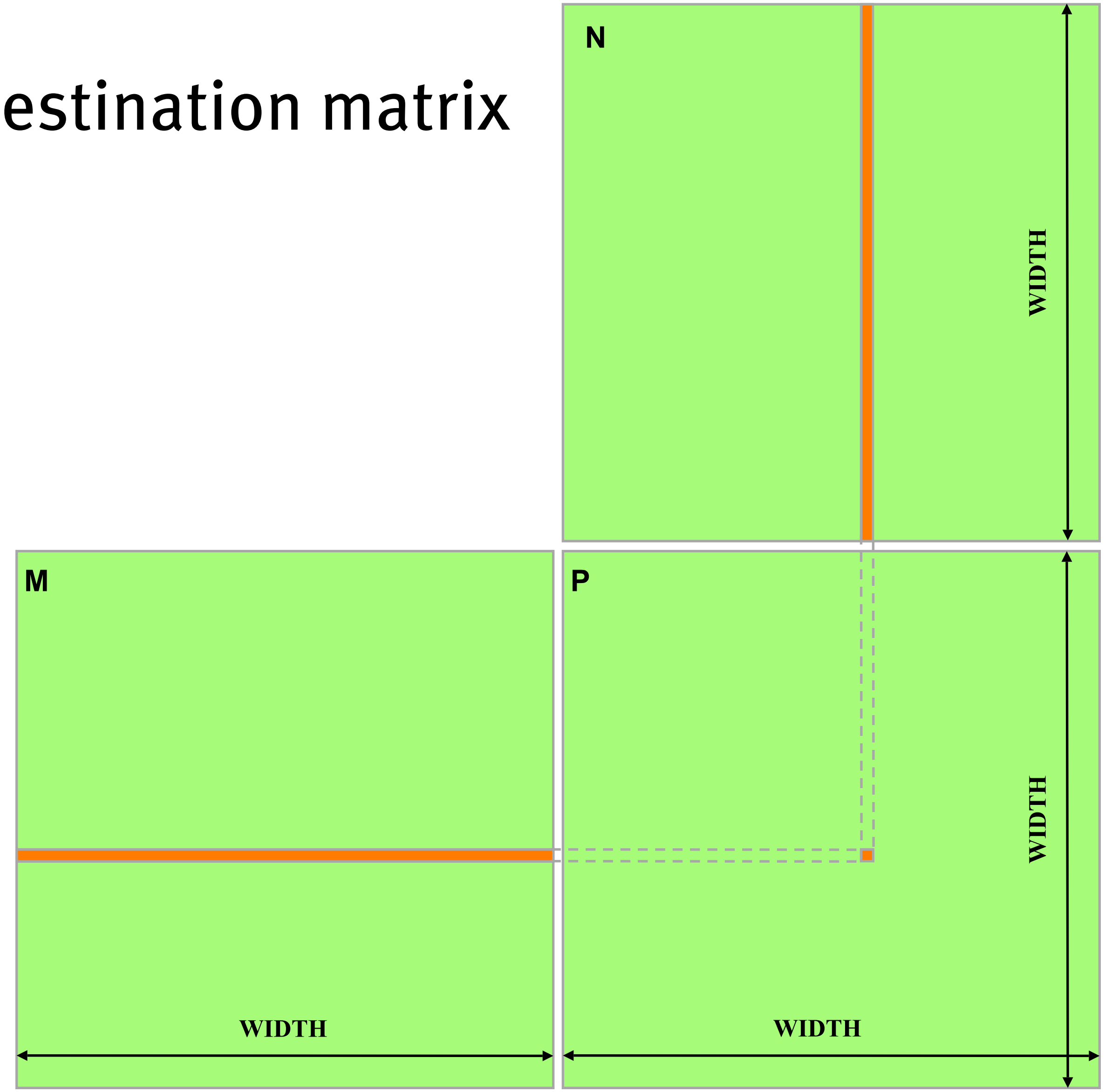
- Sort 16M 32-bit key-value pairs: ~120 ms
- Perform split operation on each bit using scan
- Can also sort each block and merge
  - Efficient merge on GPU an active area of research

GPU Design Principles

- Data layouts that:
  - Minimize memory traffic
  - Maximize coalesced memory access
- Algorithms that:
  - Exhibit data parallelism
  - Keep the hardware busy
  - Minimize divergence

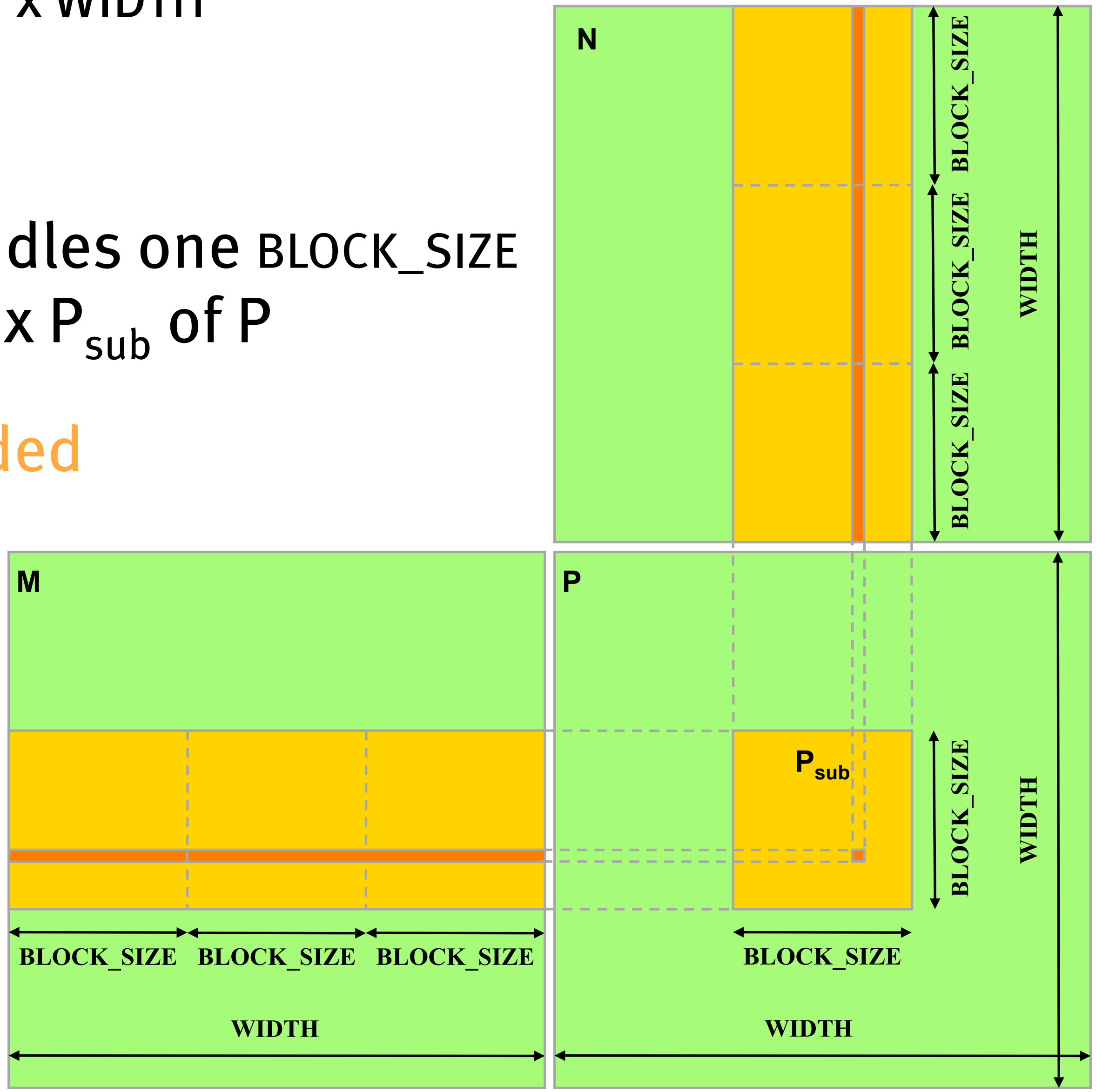
Dense Matrix Multiplication

- for all elements E in destination matrix P
  - $P_{r,c} = M_r \bullet N_c$



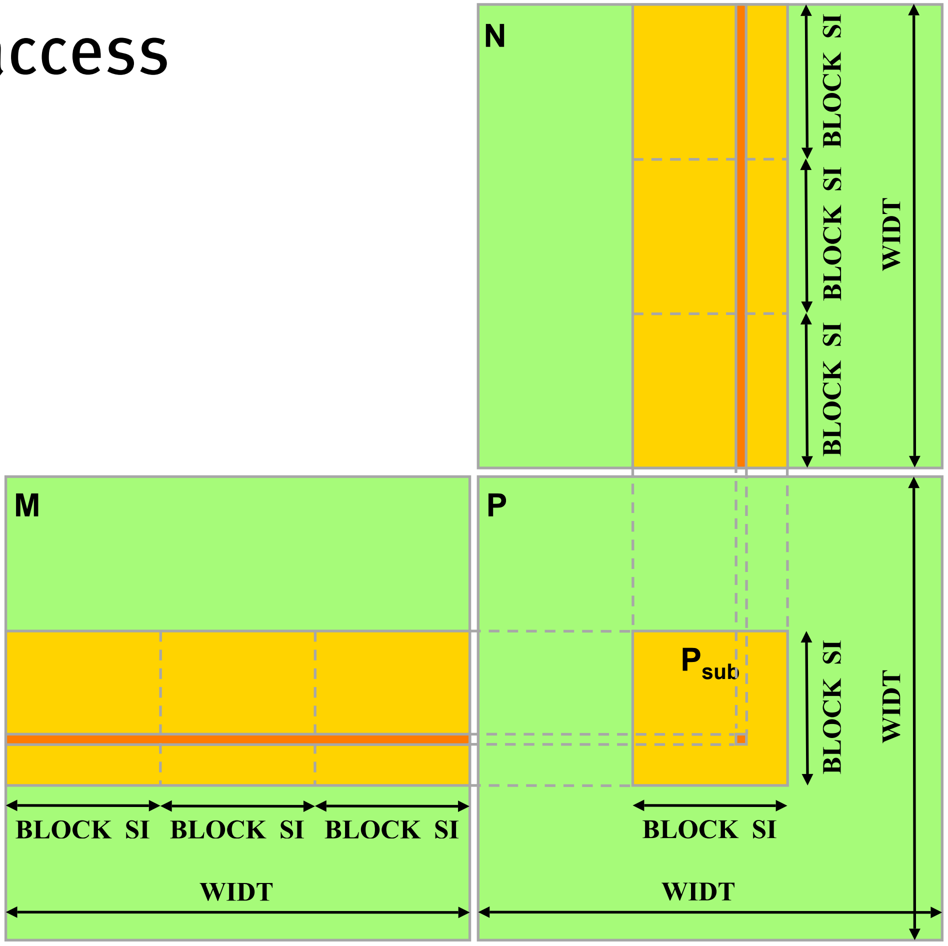
Dense Matrix Multiplication

- P = M * N of size WIDTH x WIDTH
- With blocking:
  - One thread block handles one BLOCK_SIZE x BLOCK_SIZE sub-matrix P_{sub} of P
  - M and N are only loaded WIDTH / BLOCK_SIZE times from global memory
- Great saving of memory bandwidth!



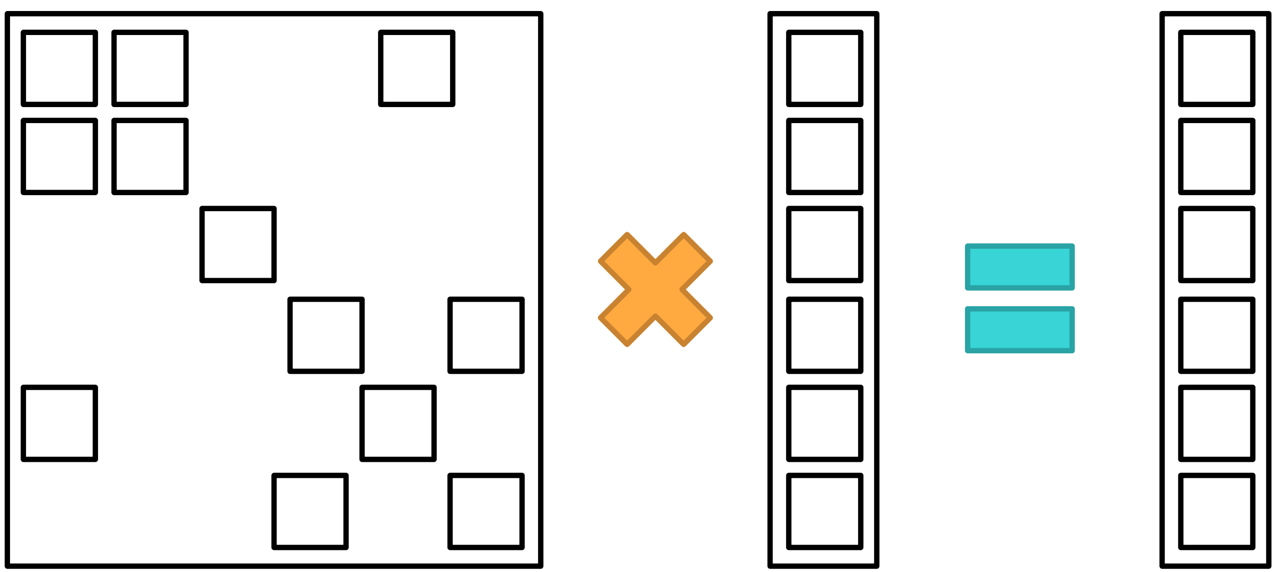
Dense Matrix Multiplication

- Data layouts that:
  - Minimize memory traffic
  - Maximize coalesced memory access
- Algorithms that:
  - Exhibit data parallelism
  - Keep the hardware busy
  - Minimize divergence

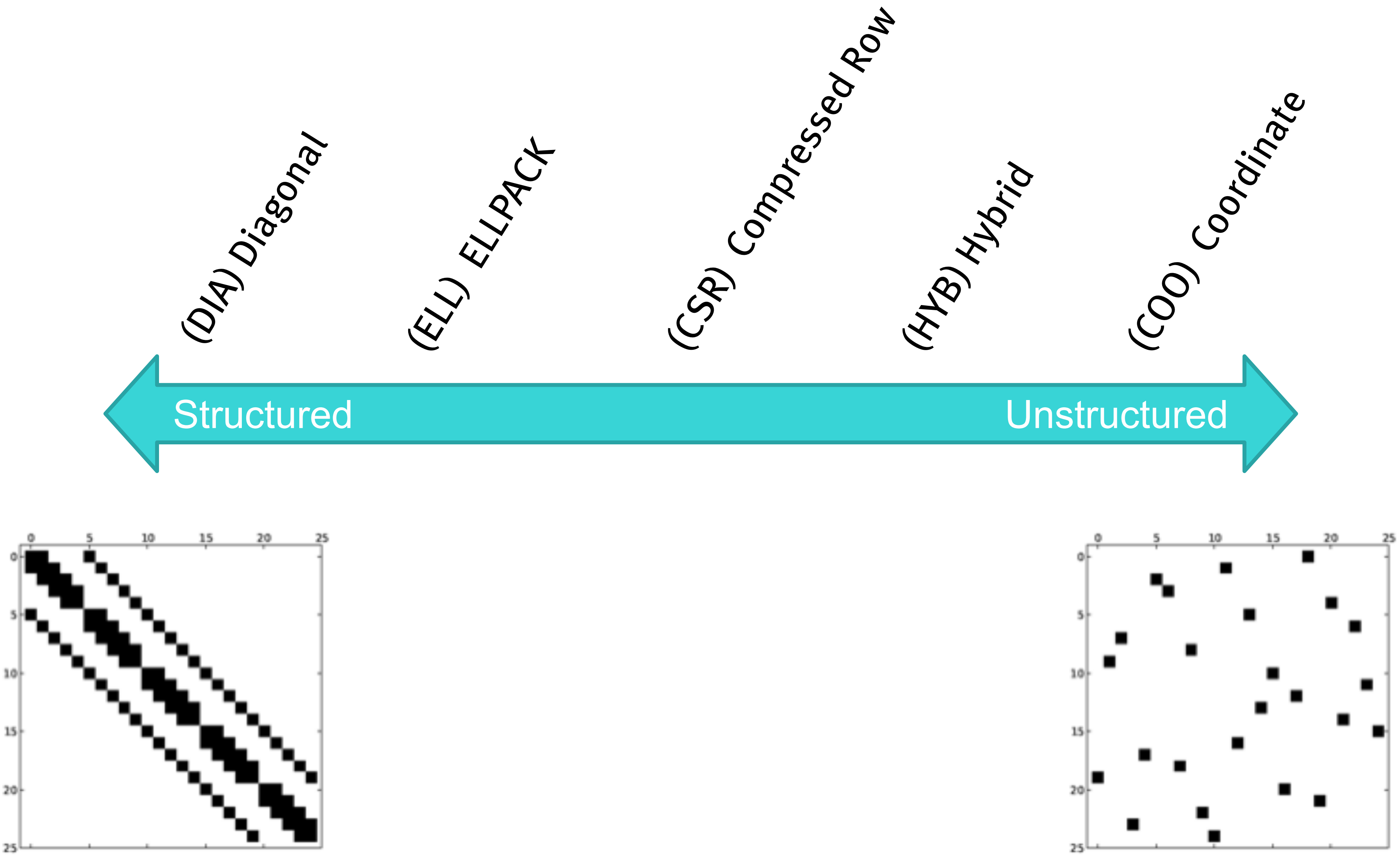


Sparse Matrix-Vector Multiply: What's Hard?

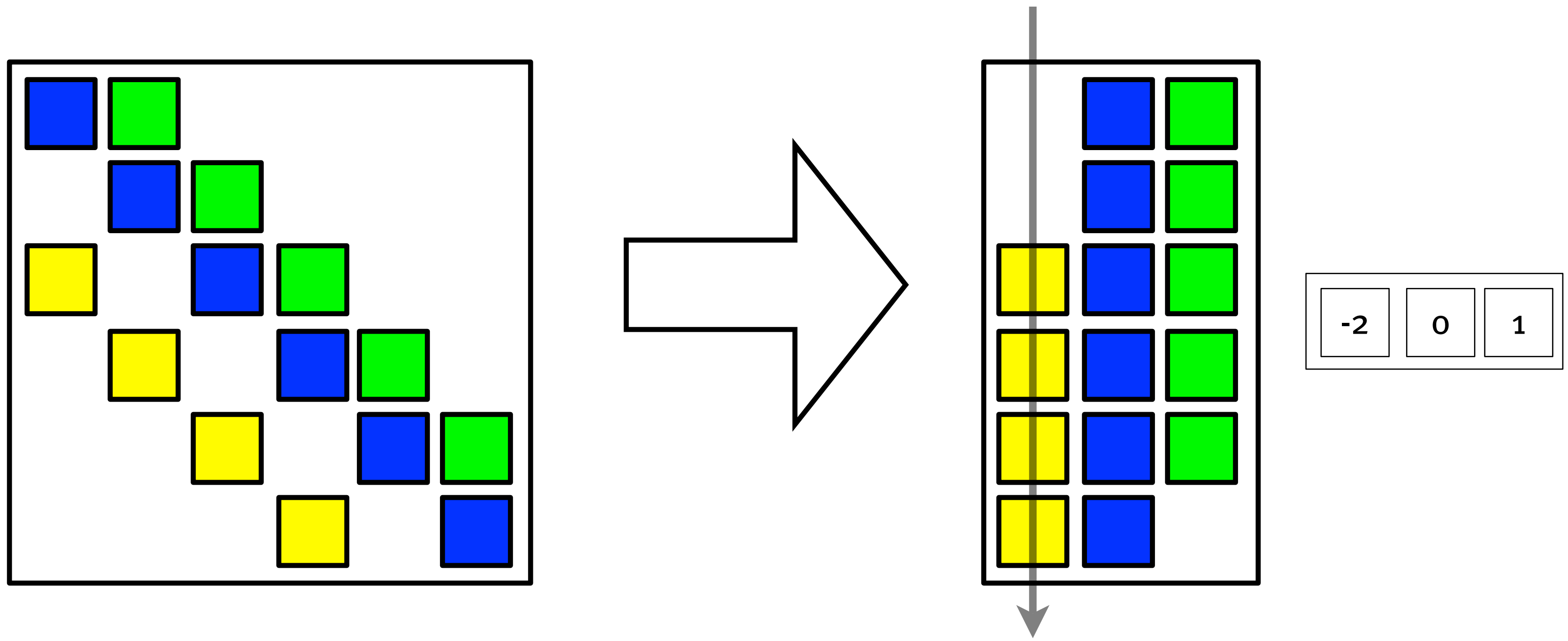
- Dense approach is wasteful
- Unclear how to map work to parallel processors
- Irregular data access



Sparse Matrix Formats

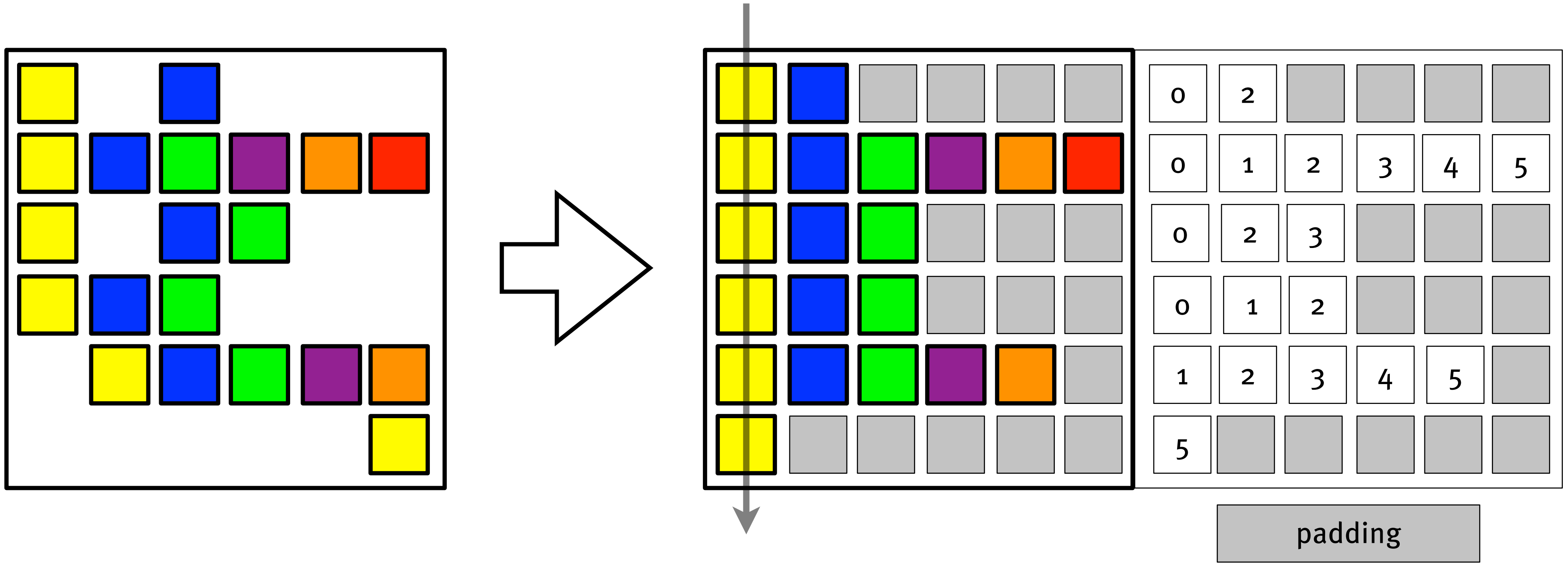


Diagonal Matrices



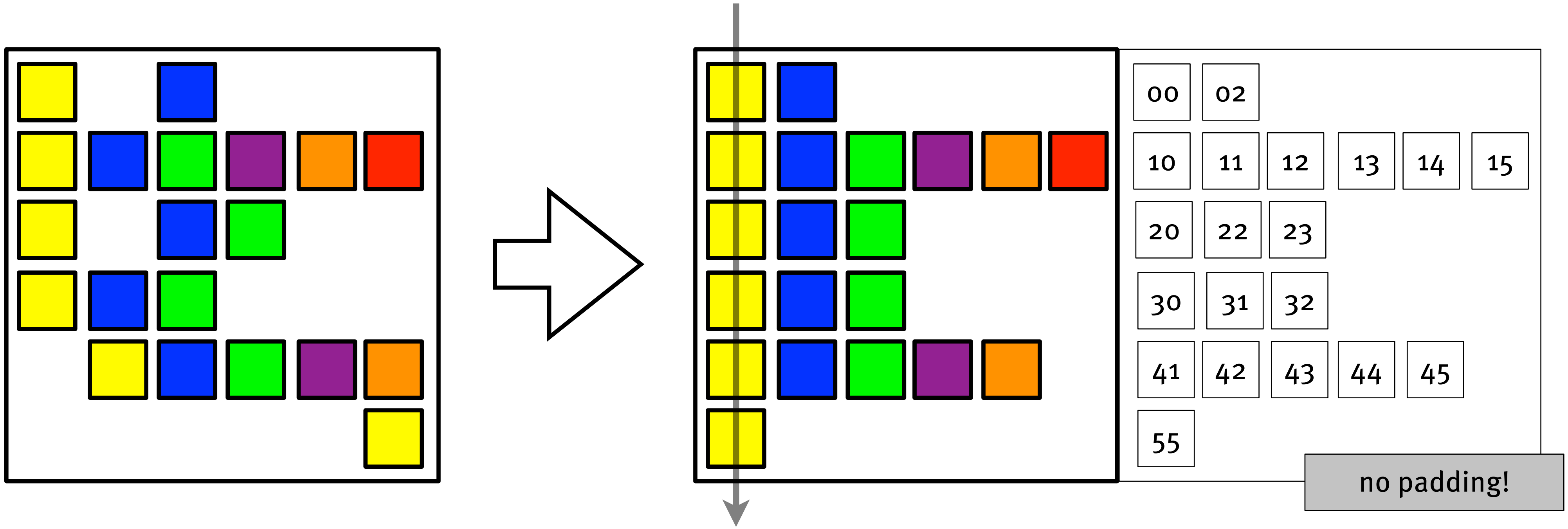
- Diagonals should be mostly populated
- Map one thread per row
  - Good parallel efficiency
  - Good memory behavior [column-major storage]

Irregular Matrices: ELL



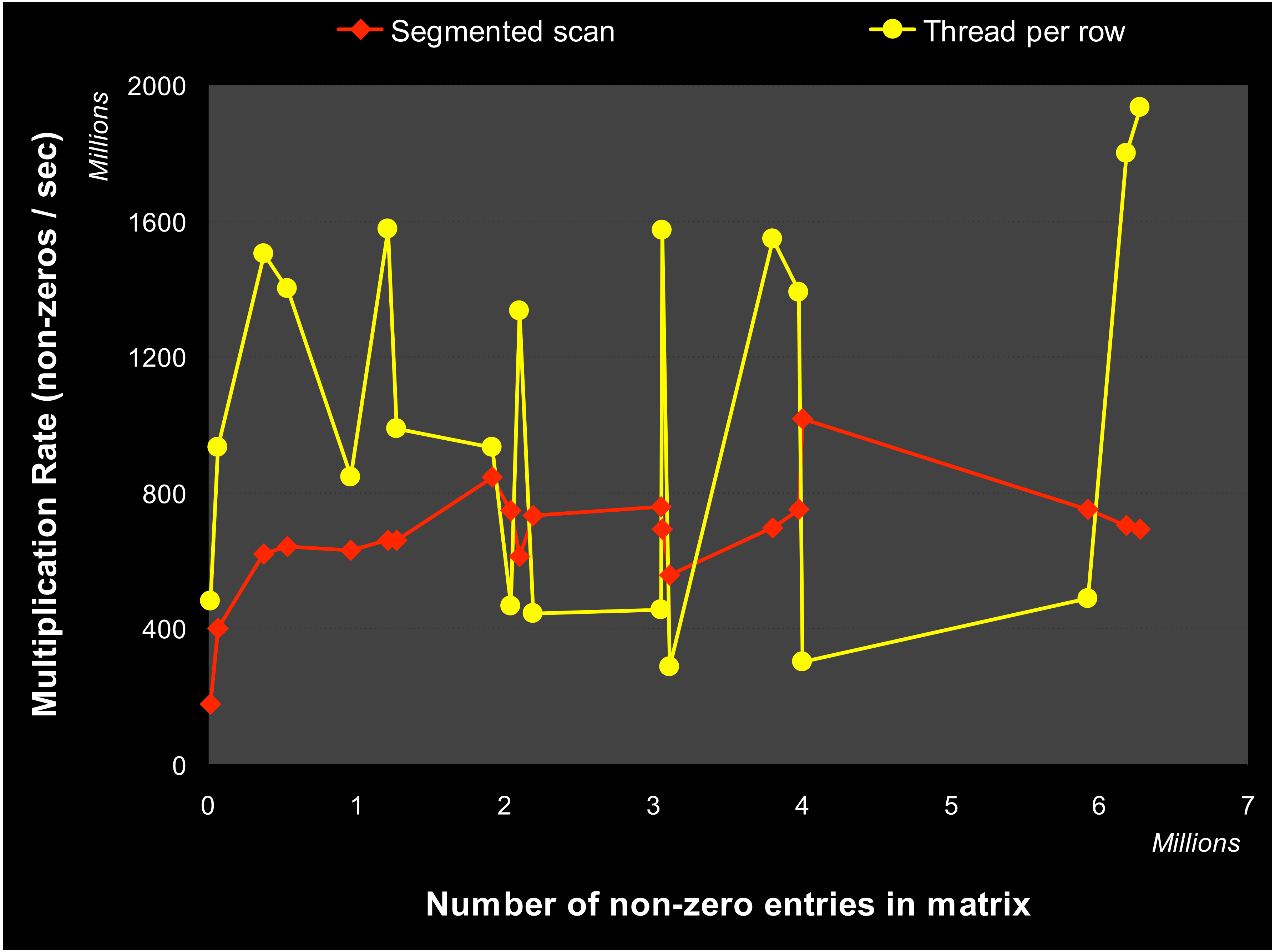
- Assign one thread per row again
- But now:
  - Load imbalance hurts parallel efficiency

Irregular Matrices: COO

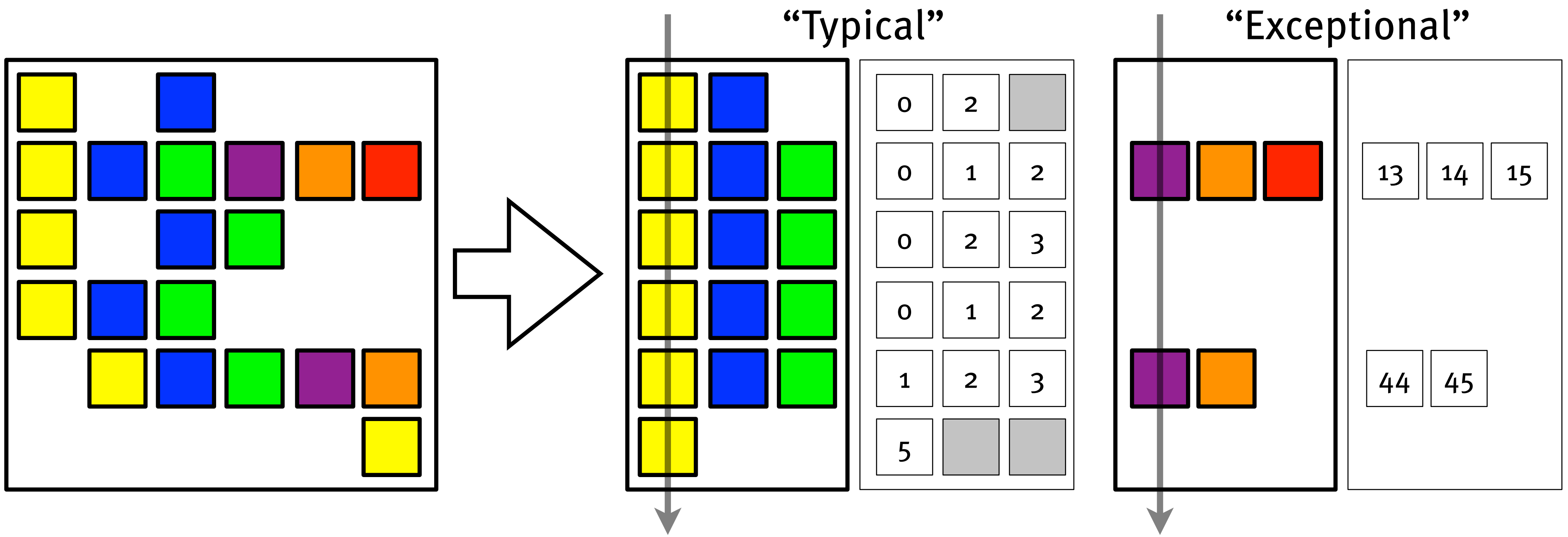


- General format; insensitive to sparsity pattern, but ~3x slower than ELL
- Assign one thread per element, combine results from all elements in a row to get output element
  - Req segmented reduction, communication btwn threads

Thread-per-{element,row}



Irregular Matrices: HYB

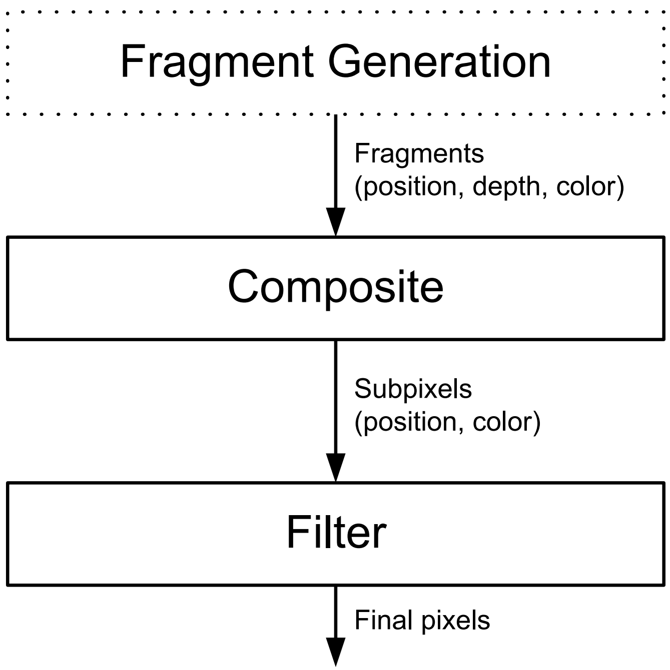


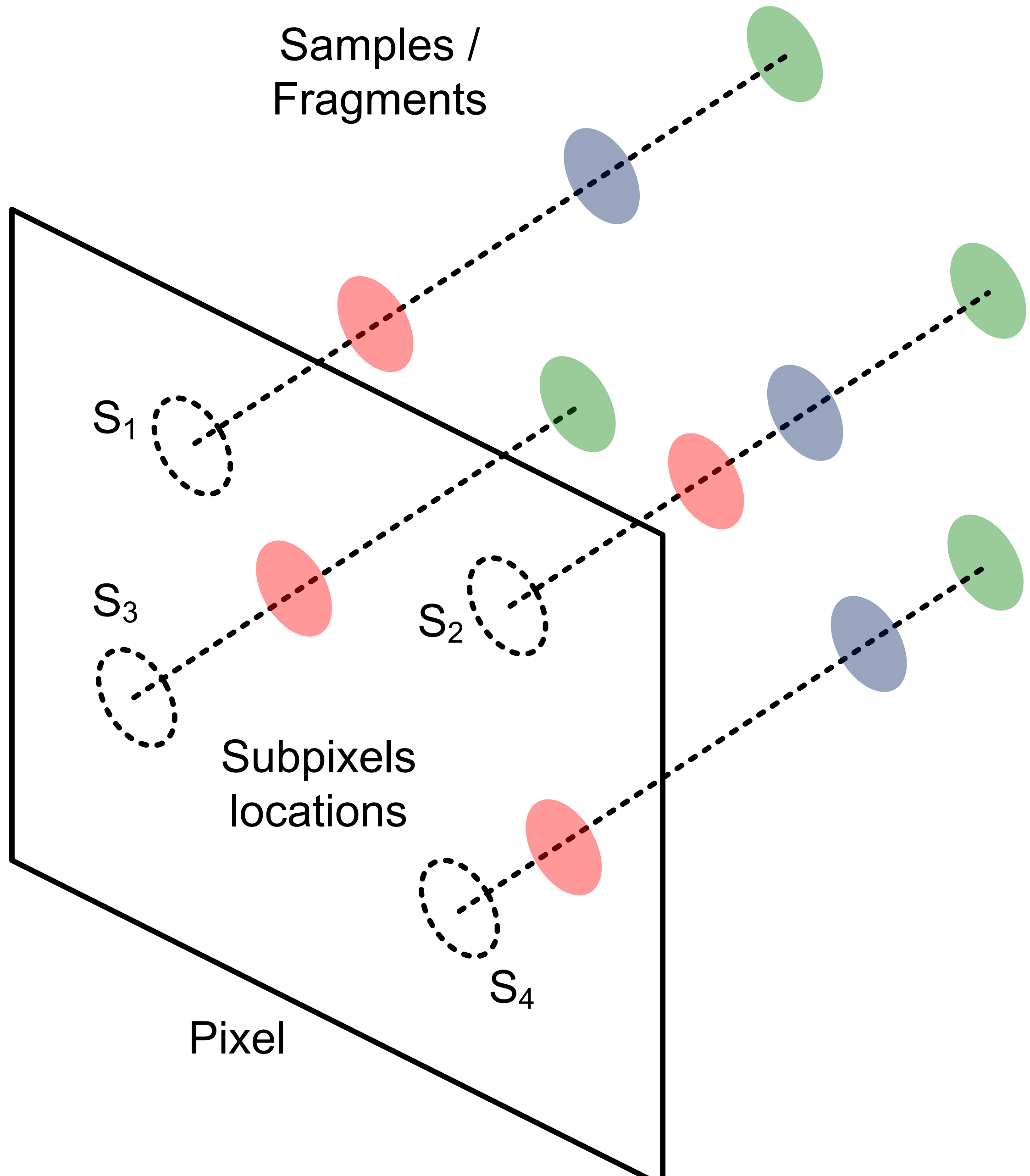
- Combine regularity of ELL + flexibility of COO

SpMV: Summary

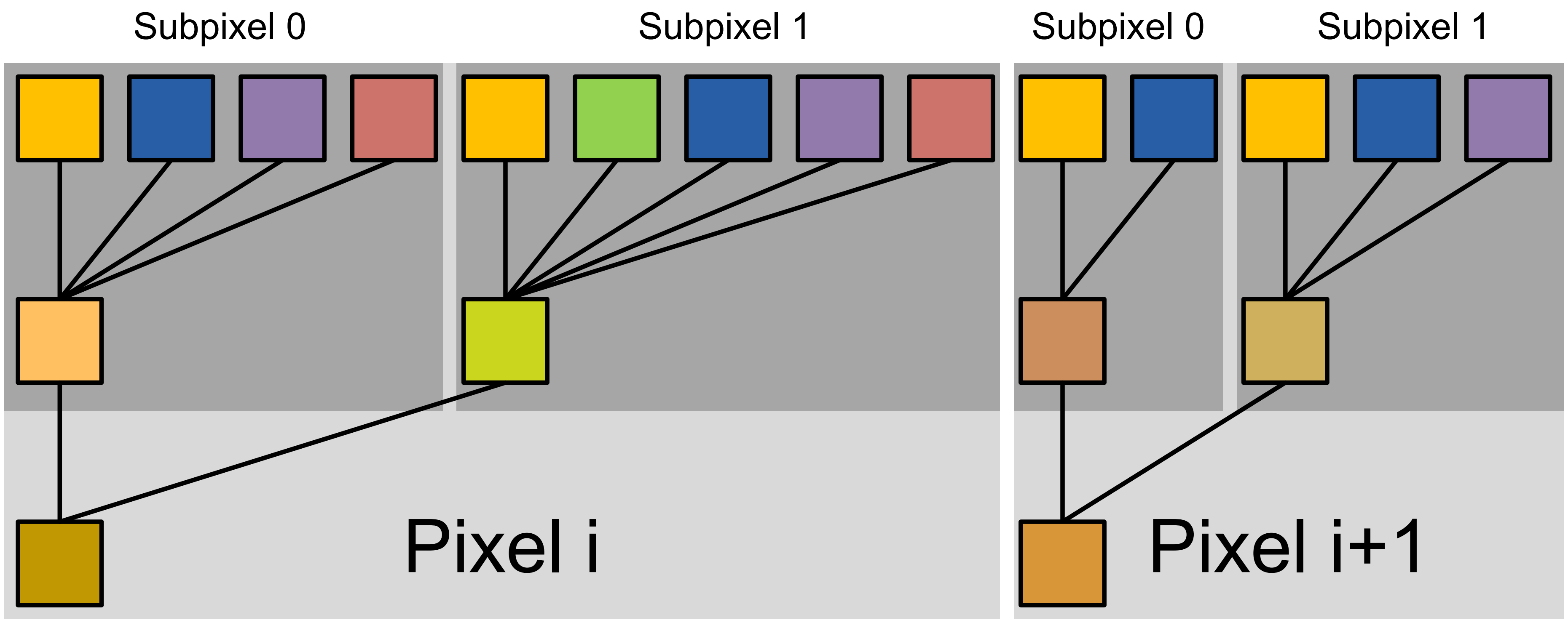
- Ample parallelism for large matrices
  - Structured matrices (dense, diagonal): straightforward
- Take-home message: Use data structure appropriate to your matrix
- Sparse matrices: Issue: Parallel efficiency
  - ELL format / one thread per row is efficient
- Sparse matrices: Issue: Load imbalance
  - COO format / one thread per element is insensitive to matrix structure
- Conclusion: Hybrid structure gives best of both worlds
  - Insight: Irregularity is manageable if you regularize the common case

Composition

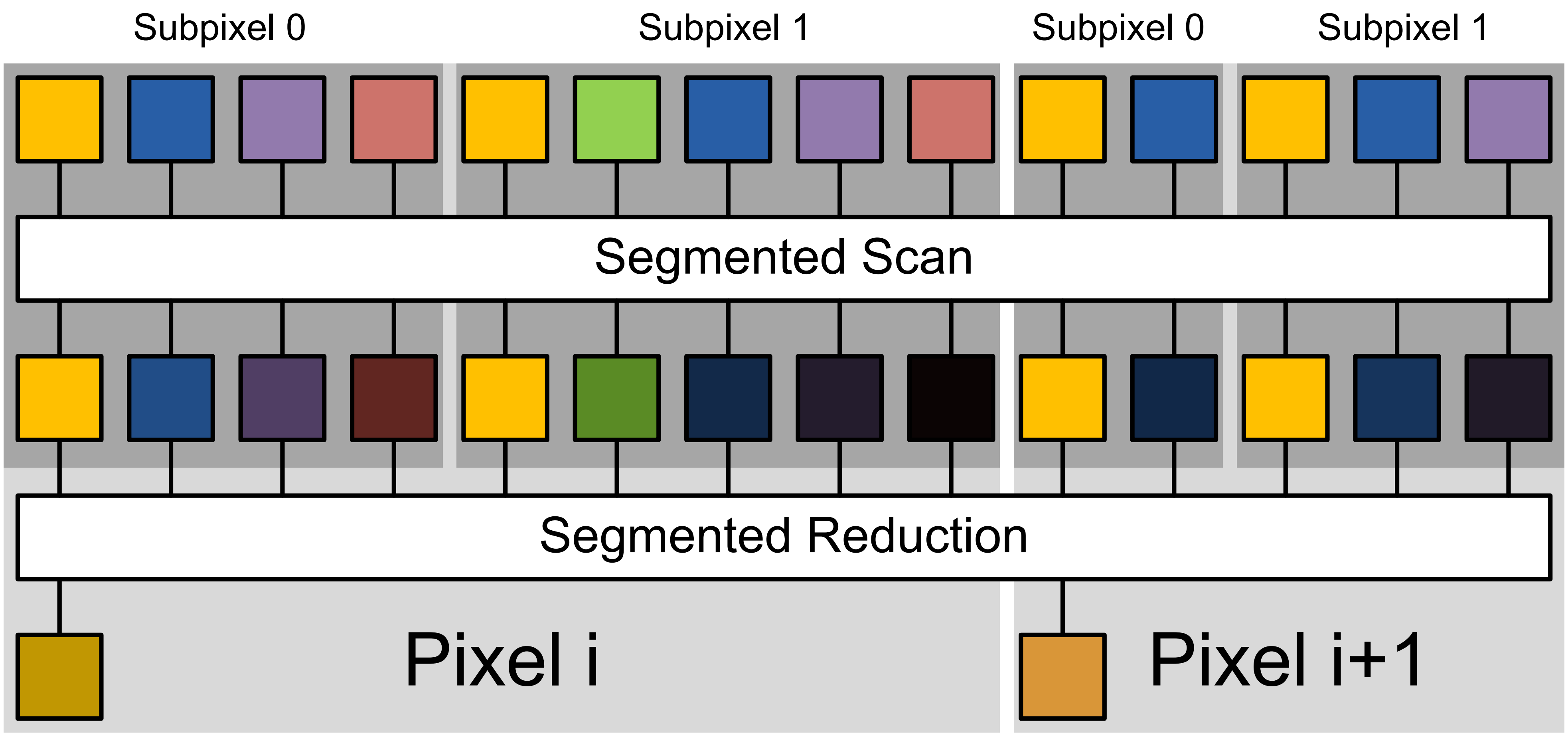


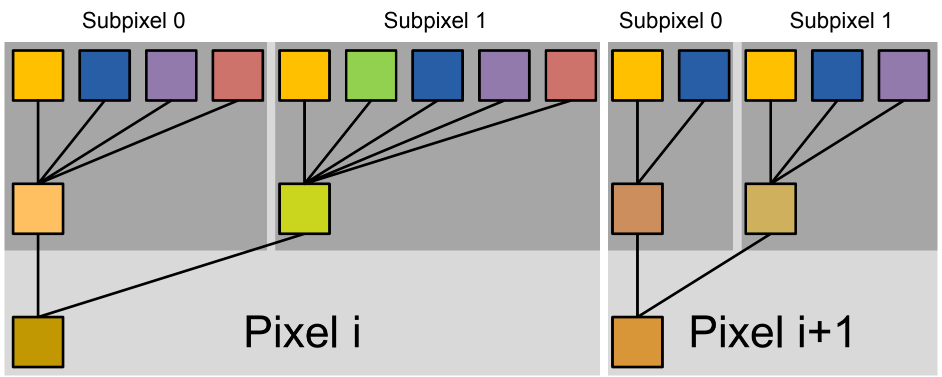


Pixel-Parallel Composition

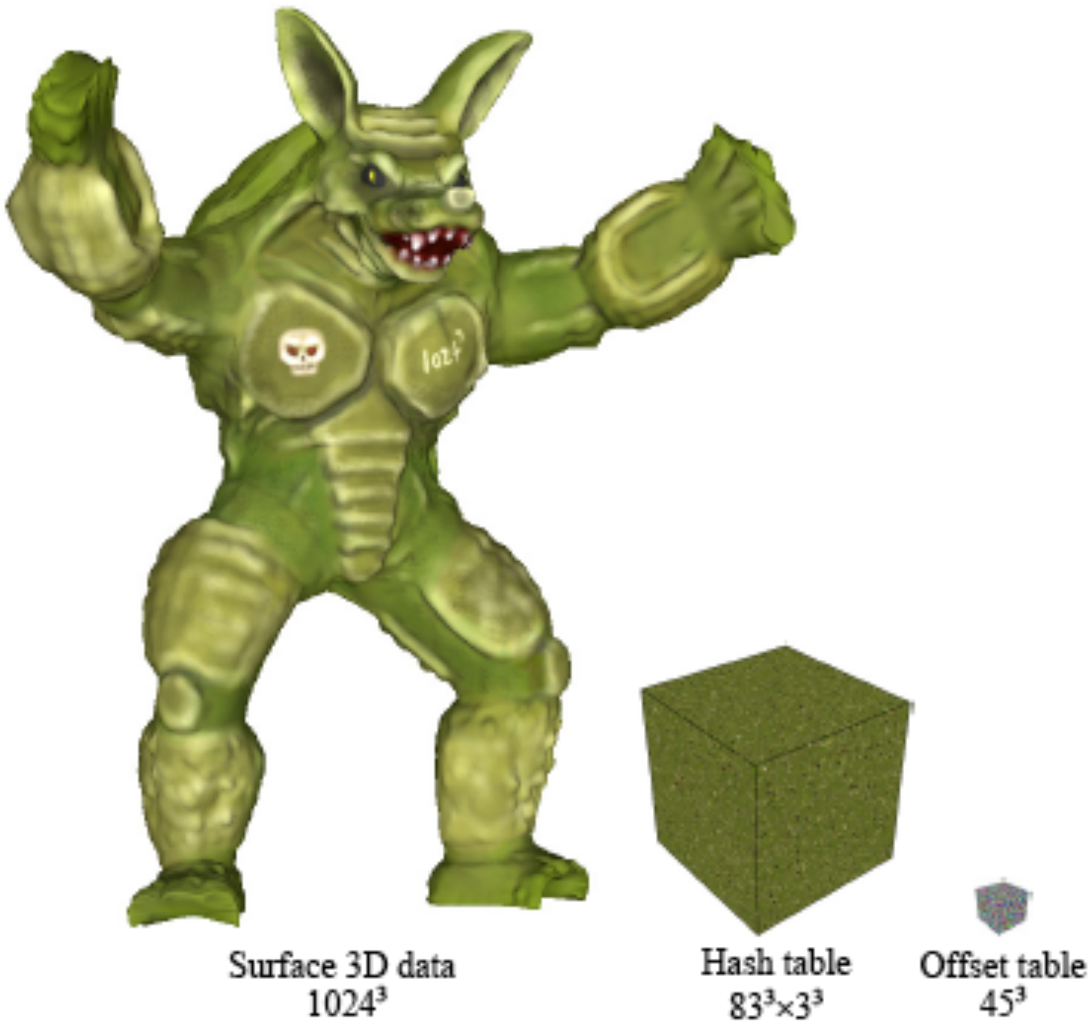


Sample-Parallel Composition



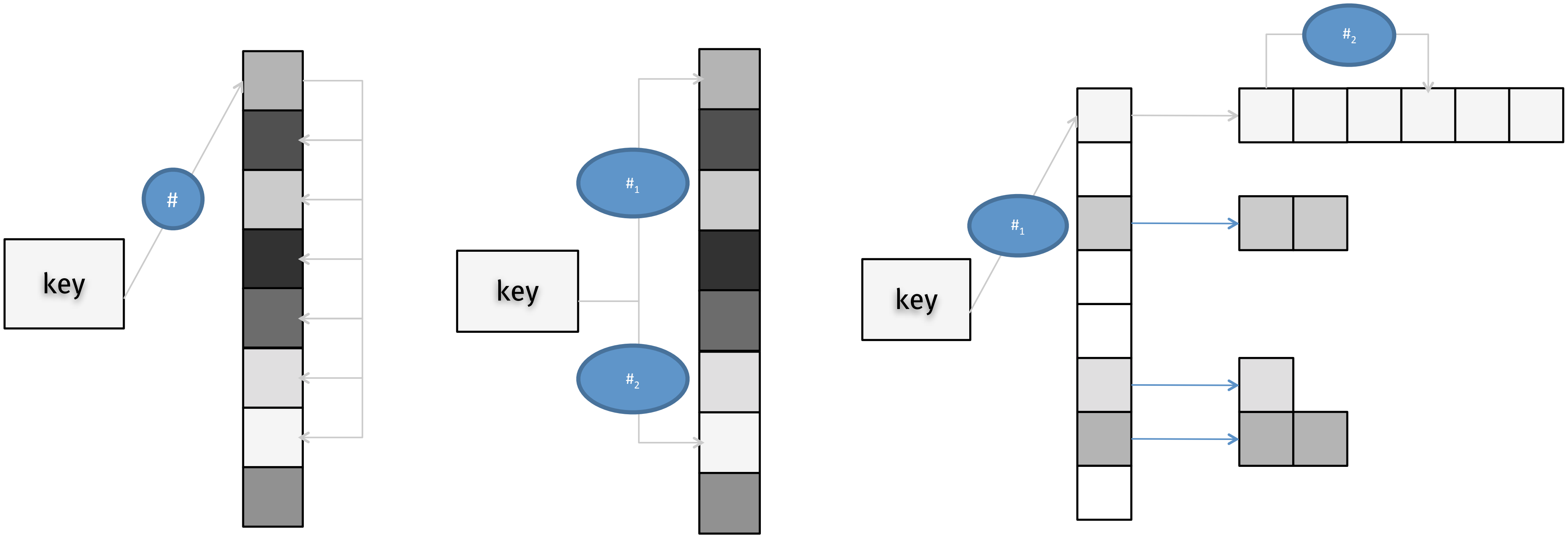


Hash Tables & Sparsity



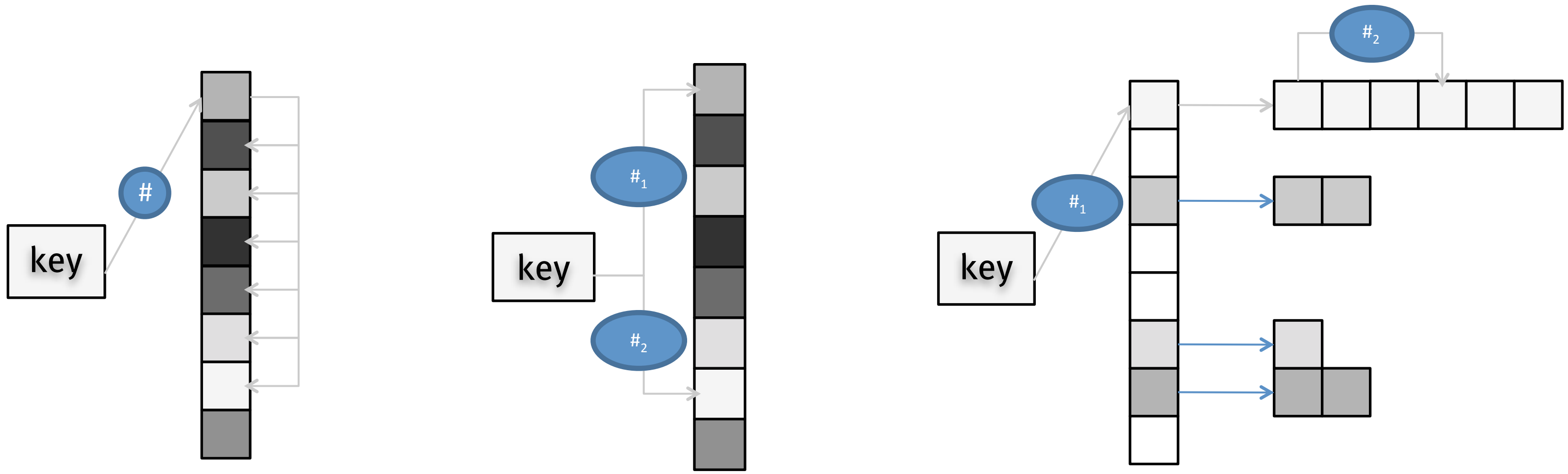
- Lefebvre and Hoppe, Siggraph 2006

Scalar Hashing



Linear Probing Double Probing Chaining

Scalar Hashing: Parallel Problems



- Construction and Lookup
  - Variable time/work per entry
- Construction
  - Synchronization / shared access to data structure

Parallel Hashing: The Problem

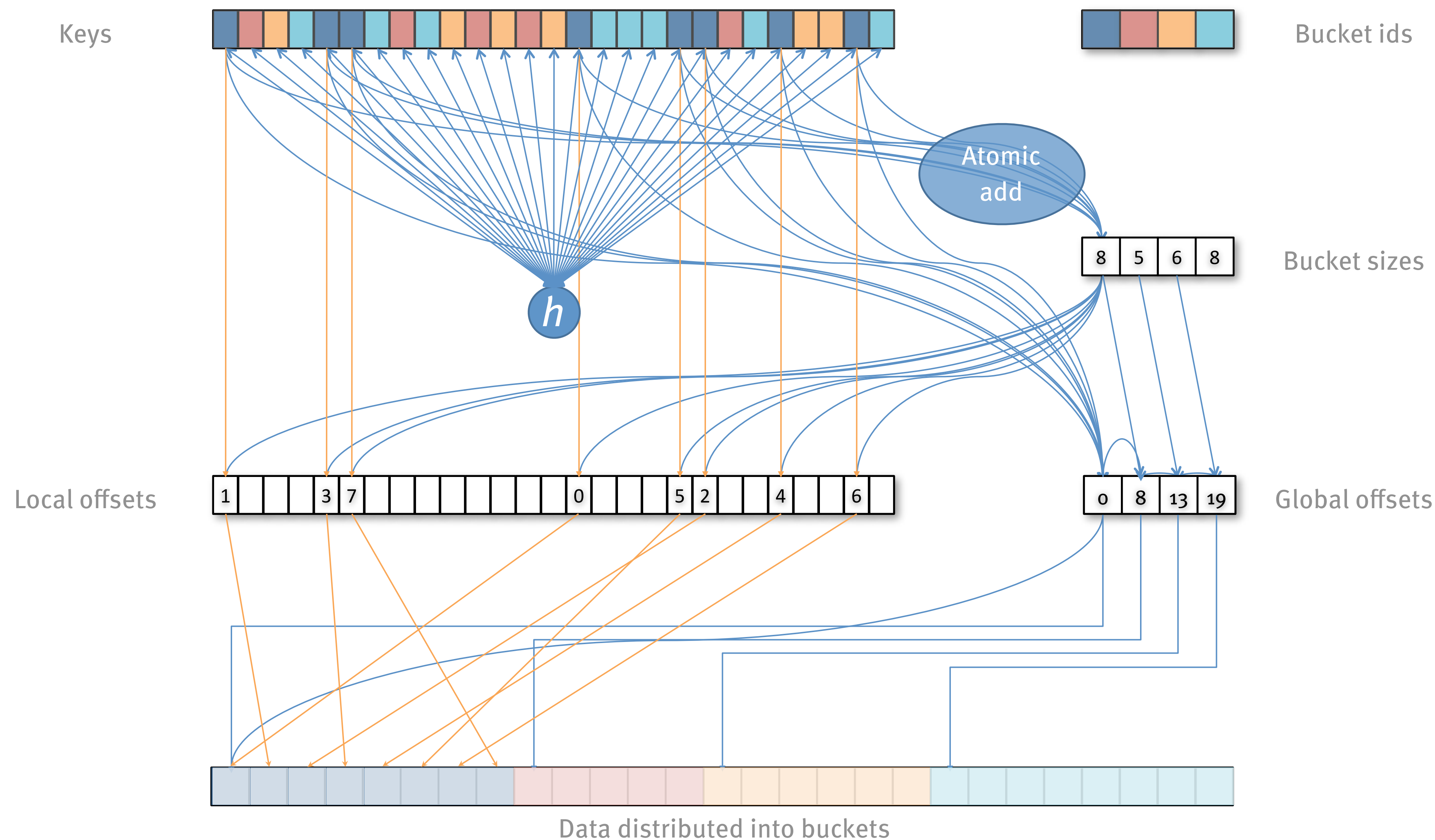
- Hash tables are good for sparse data.
- Input: Set of key-value pairs to place in the hash table
- Output: Data structure that allows:
  - Determining if key has been placed in hash table
  - Given the key, fetching its value
- Could also:
  - Sort key-value pairs by key (construction)
  - Binary-search sorted list (lookup)
- Recalculate at every change

Parallel Hashing: What We Want

- Fast construction time
- Fast access time
  - $O(1)$ for any element, $O(n)$ for n elements in parallel
- Reasonable memory usage

- Algorithms and data structures may sit at different places in this space
  - Perfect spatial hashing has good lookup times and reasonable memory usage but is very slow to construct

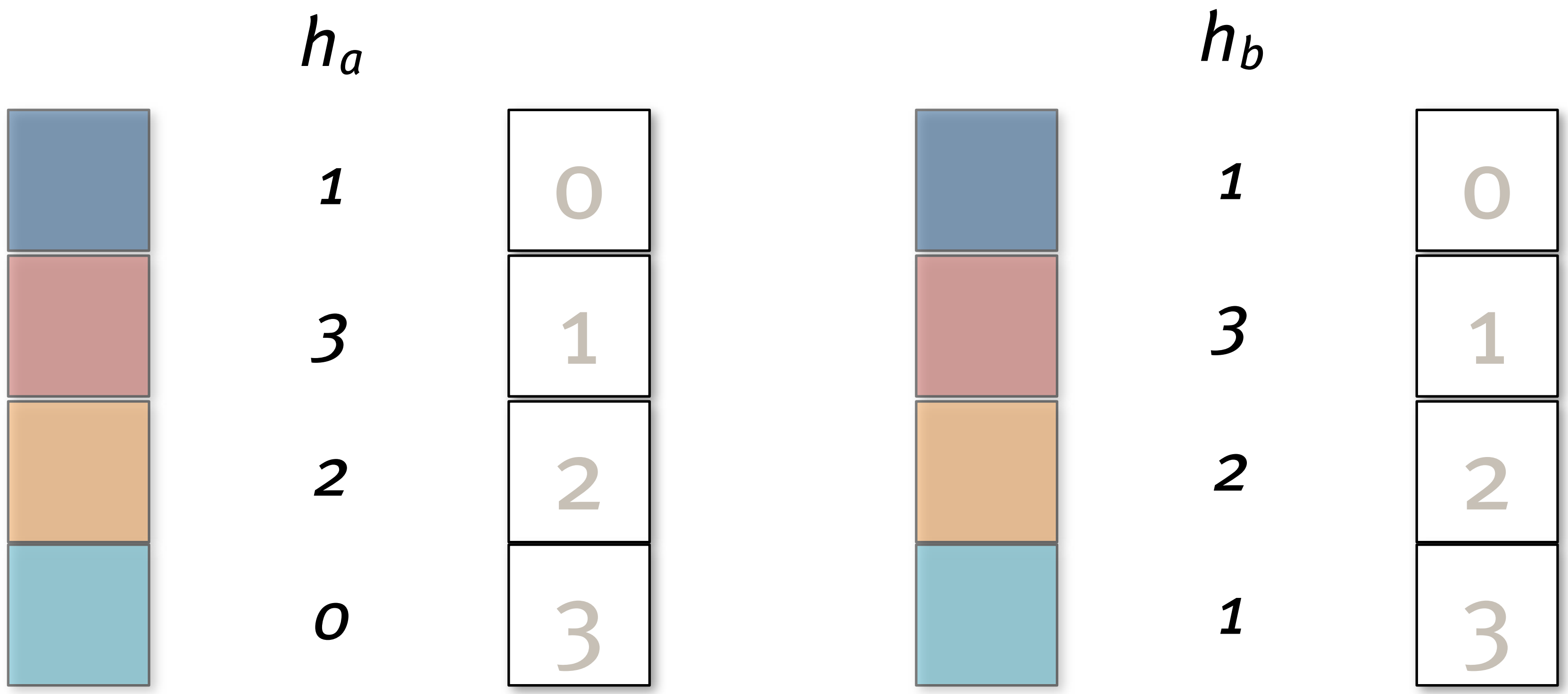
Level 1: Distribute into buckets



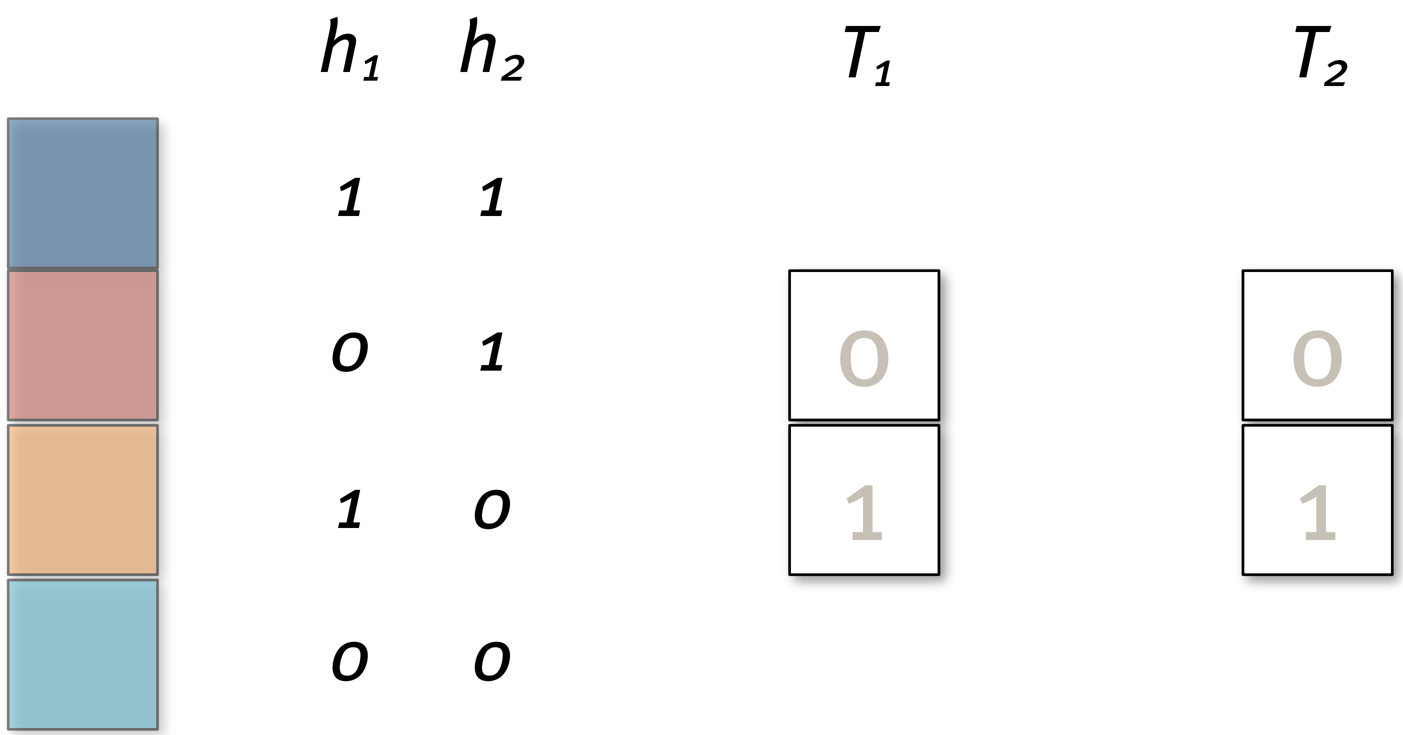
Parallel Hashing: Level 1

- Good for a coarse categorization
  - Possible performance issue: atomics
- Bad for a fine categorization
  - Space requirements for n elements to (probabilistically) guarantee no collisions are $O(n^2)$

Hashing in Parallel



Cuckoo Hashing Construction

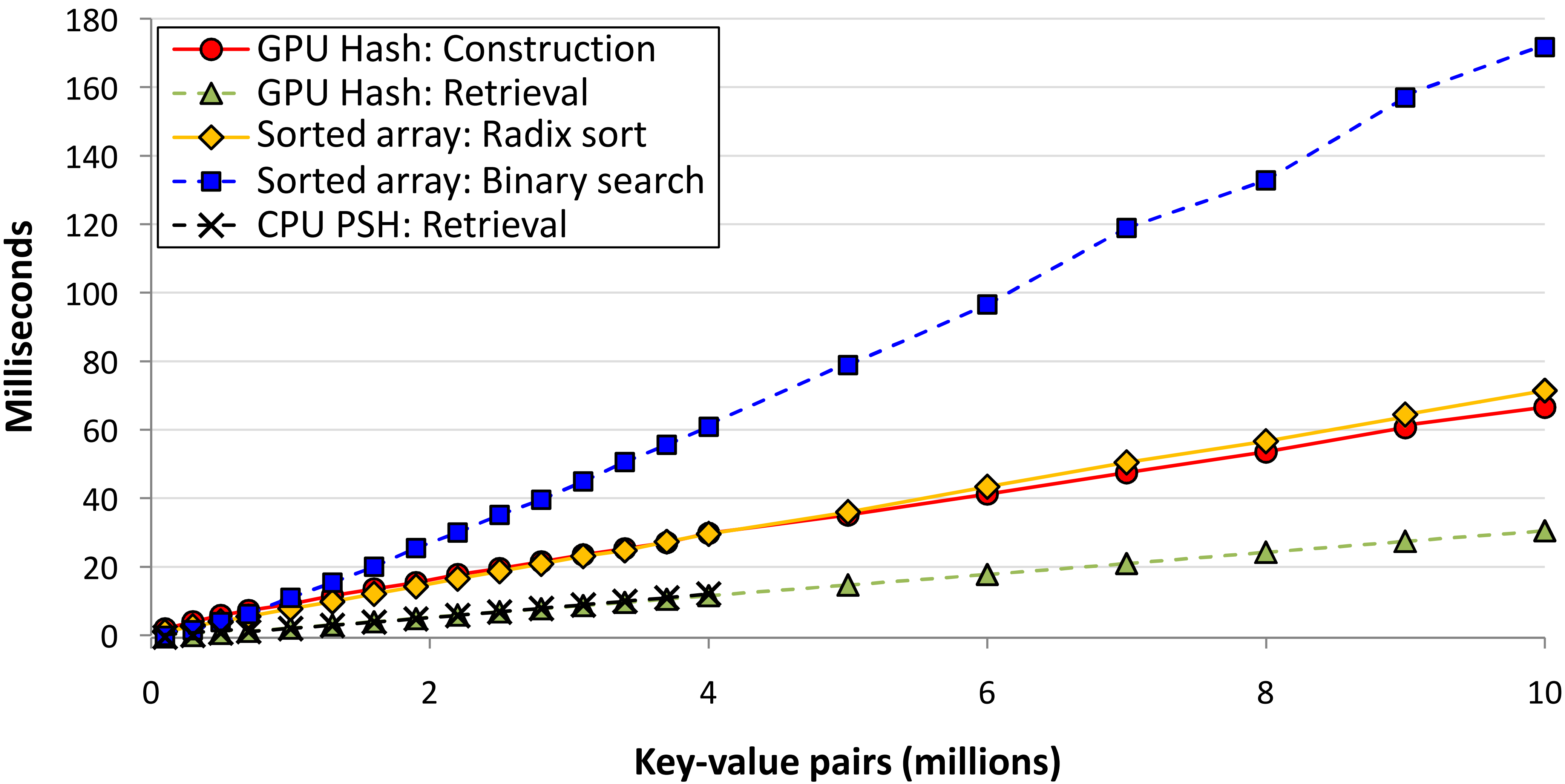


- Lookup procedure: in parallel, for each element:
  - Calculate h_1 & look in T_1;
  - Calculate h_2 & look in T_2; still $O(1)$ lookup

Cuckoo Construction Mechanics

- Level 1 created buckets of no more than 512 items
  - Average: 409; probability of overflow: $< 10^{-6}$
- Level 2: Assign each bucket to a thread block, construct cuckoo hash per bucket entirely within shared memory
  - Semantic: Multiple writes to same location must have one and only one winner
- Our implementation uses 3 tables of 192 elements each (load factor: 71%)
- What if it fails? New hash functions & start over.

Timings on random voxel data

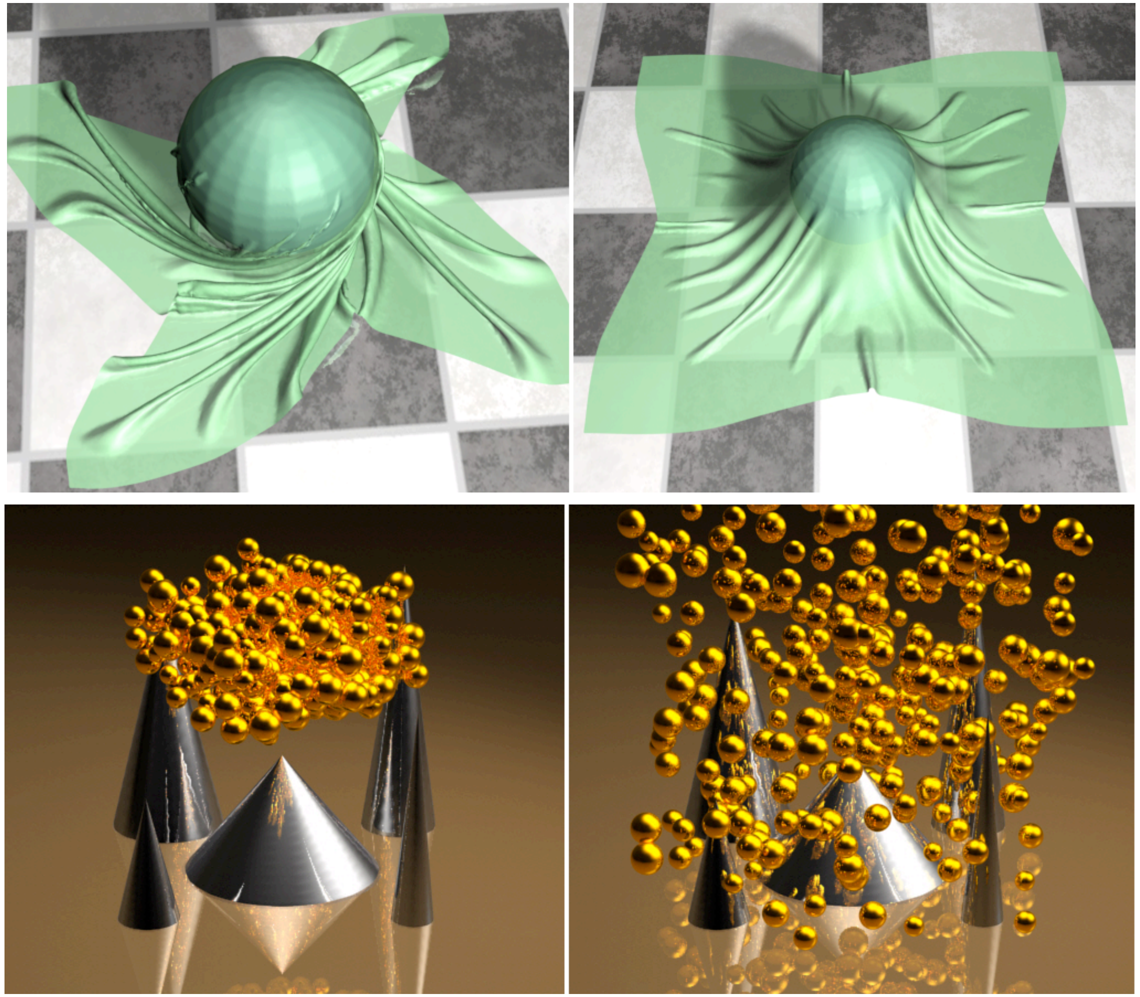


Hashing: Big Ideas

- Classic serial hashing techniques are a poor fit for a GPU.
  - Serialization, load balance
- Solving this problem required a different algorithm
  - Both hashing algorithms were new to the parallel literature
  - Hybrid algorithm was entirely new

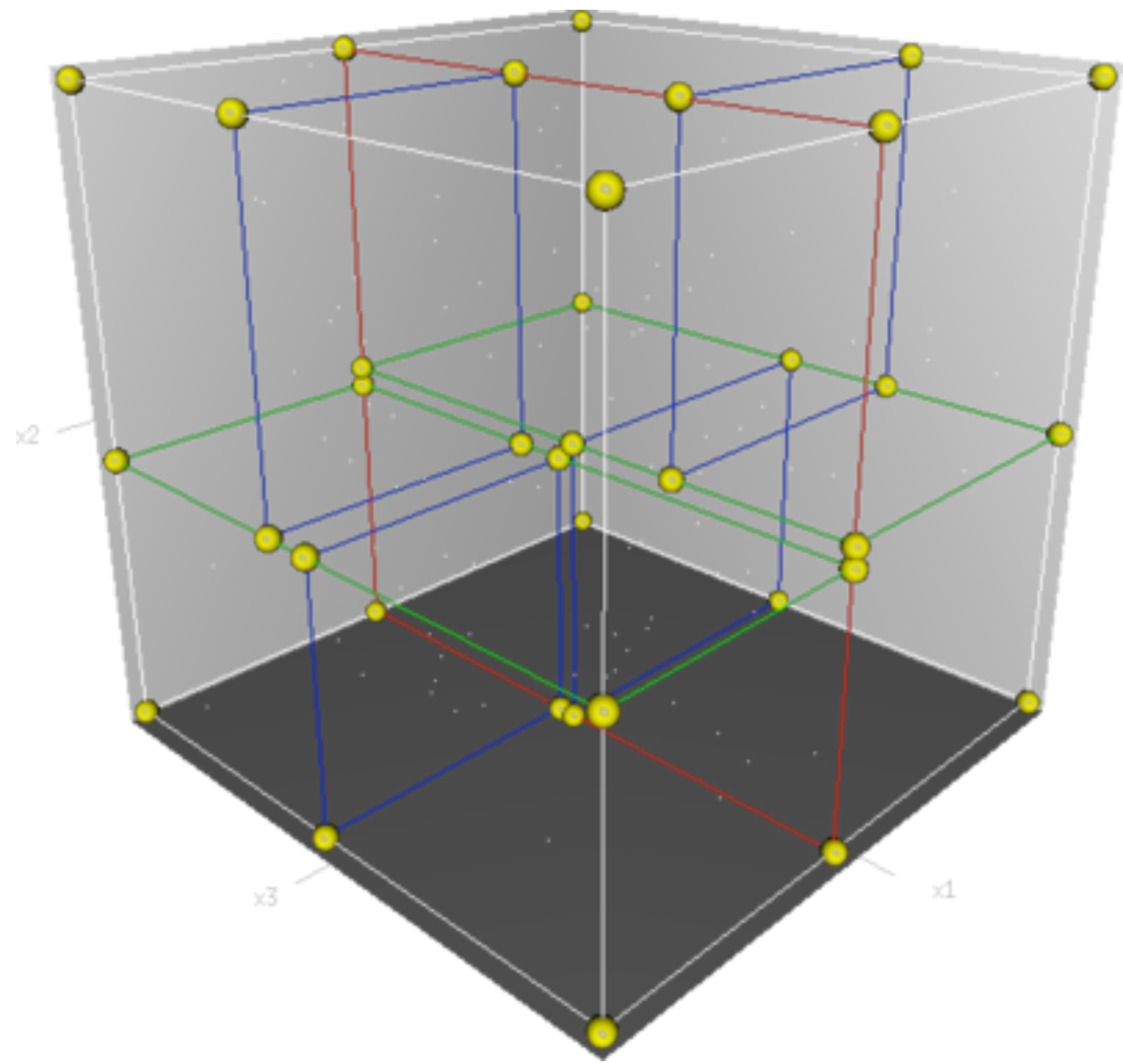
Trees: Motivation

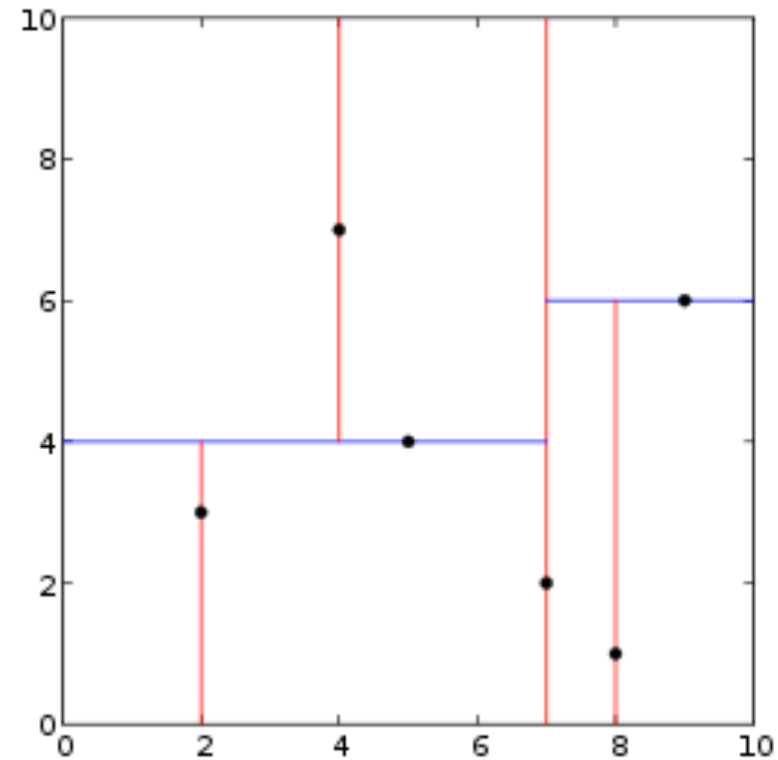
- Query: Does object X intersect with anything in the scene?
- Difficulty: X and the scene are dynamic
- Goal: Data structure that makes this query efficient (in parallel)



Images from *HPCCD: Hybrid Parallel Continuous Collision Detection*, Kim et al., Pacific Graphics 2009

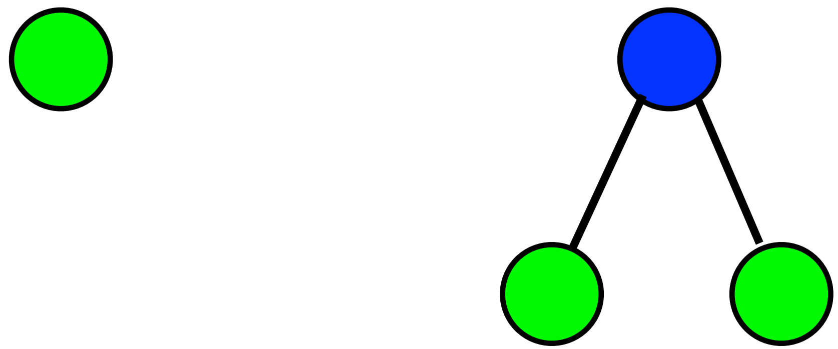
k-d trees

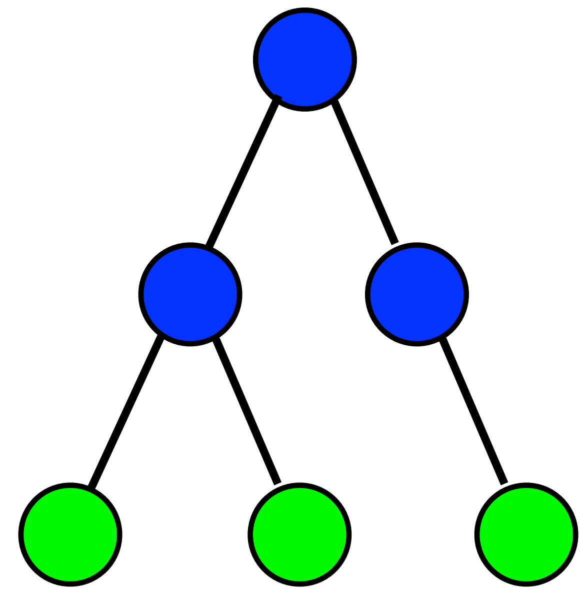


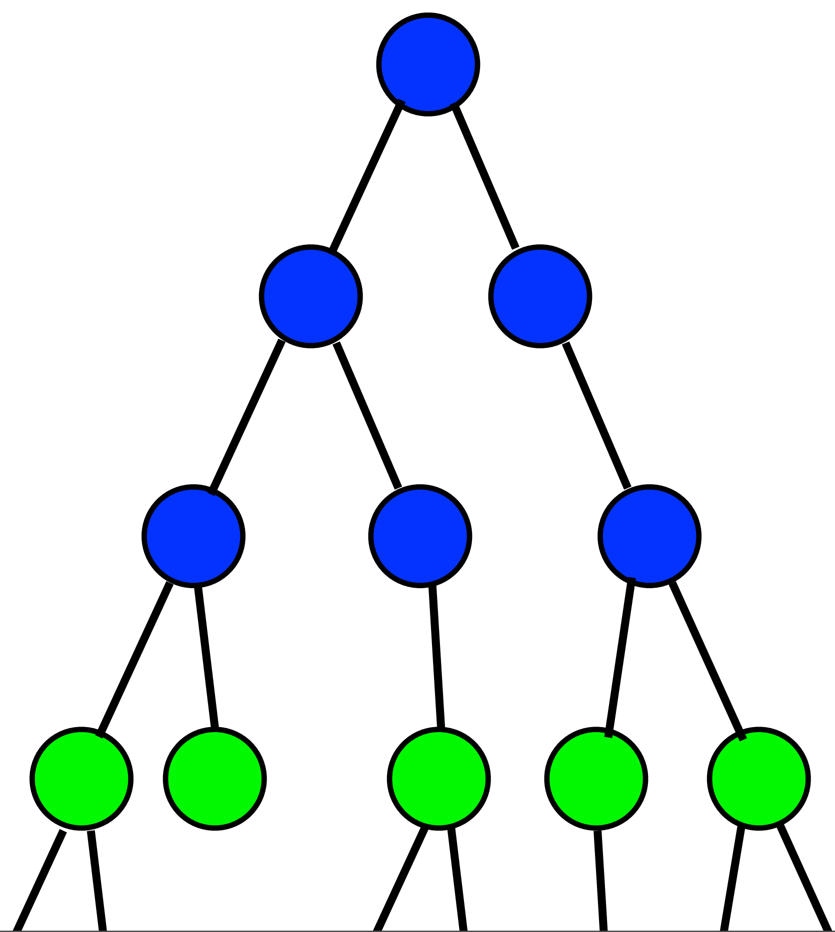


Images from Wikipedia, “Kd-tree”

Generating Trees

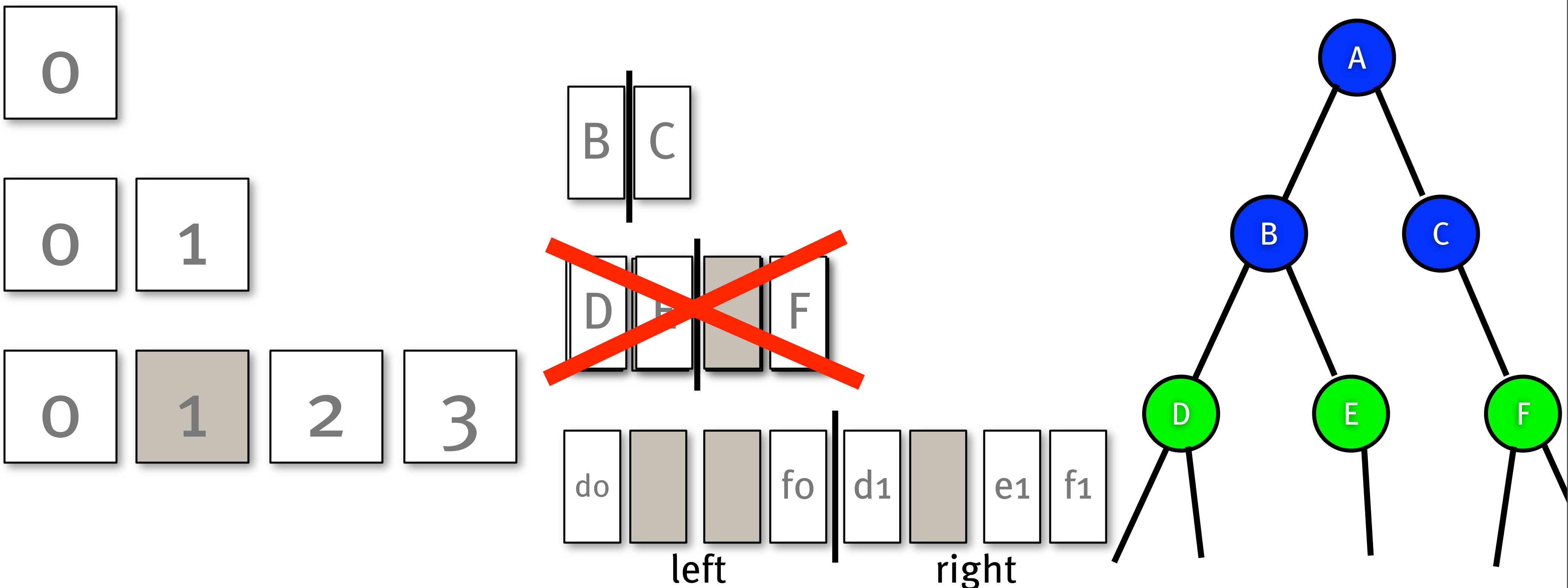






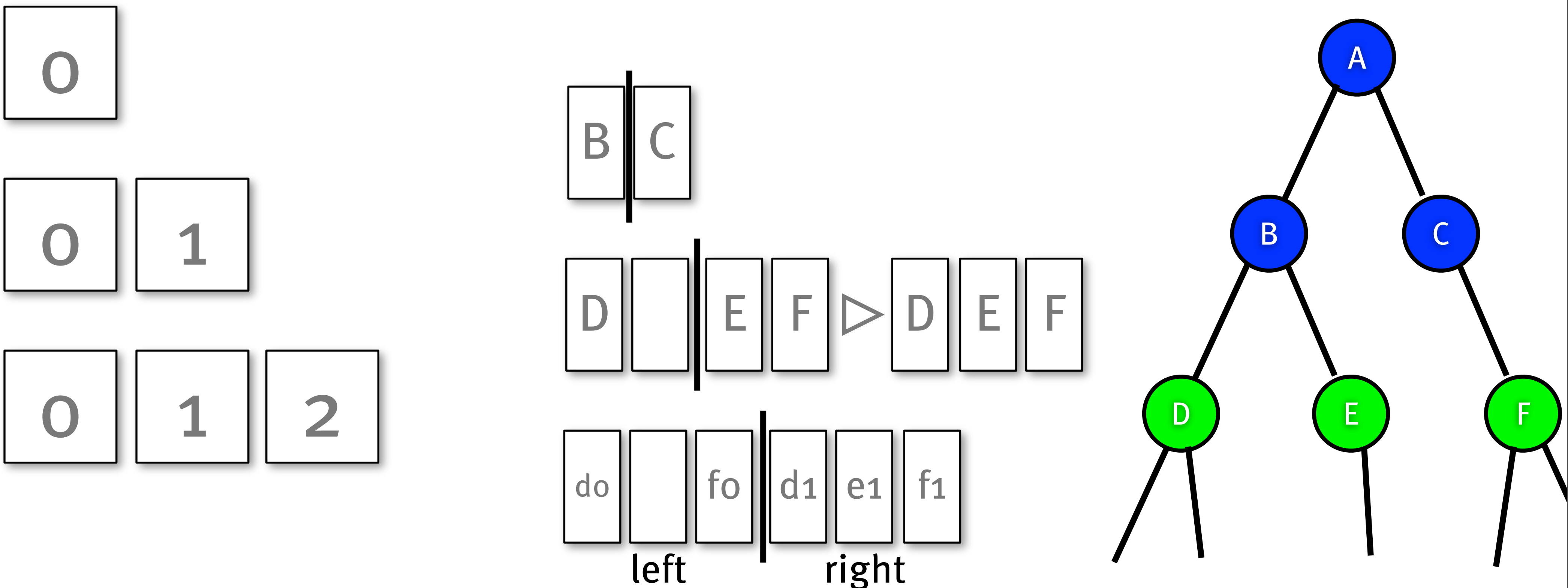
- Increased parallelism with depth
- Irregular work generation

Tree Construction on a GPU



- At each stage, any node can generate 0, 1, or 2 new nodes
- Increased parallelism, but some threads wasted
- Compact after each step?

Tree Construction on a GPU



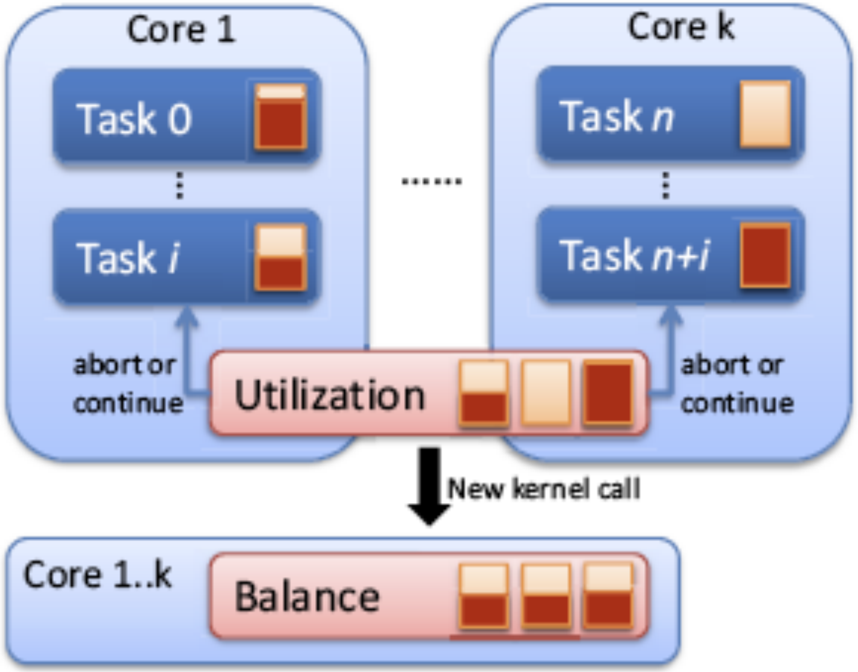
- Compact reduces overwork, but …
- … requires global compact operation per step
- Also requires worst-case storage allocation

Assumptions of Approach

- Fairly high computation cost per step
  - Smaller cost -> runtime dominated by overhead
- Small branching factor
  - Makes pre-allocation tractable
- Fairly uniform computation per step
  - Otherwise, load imbalance
- No communication between threads at all

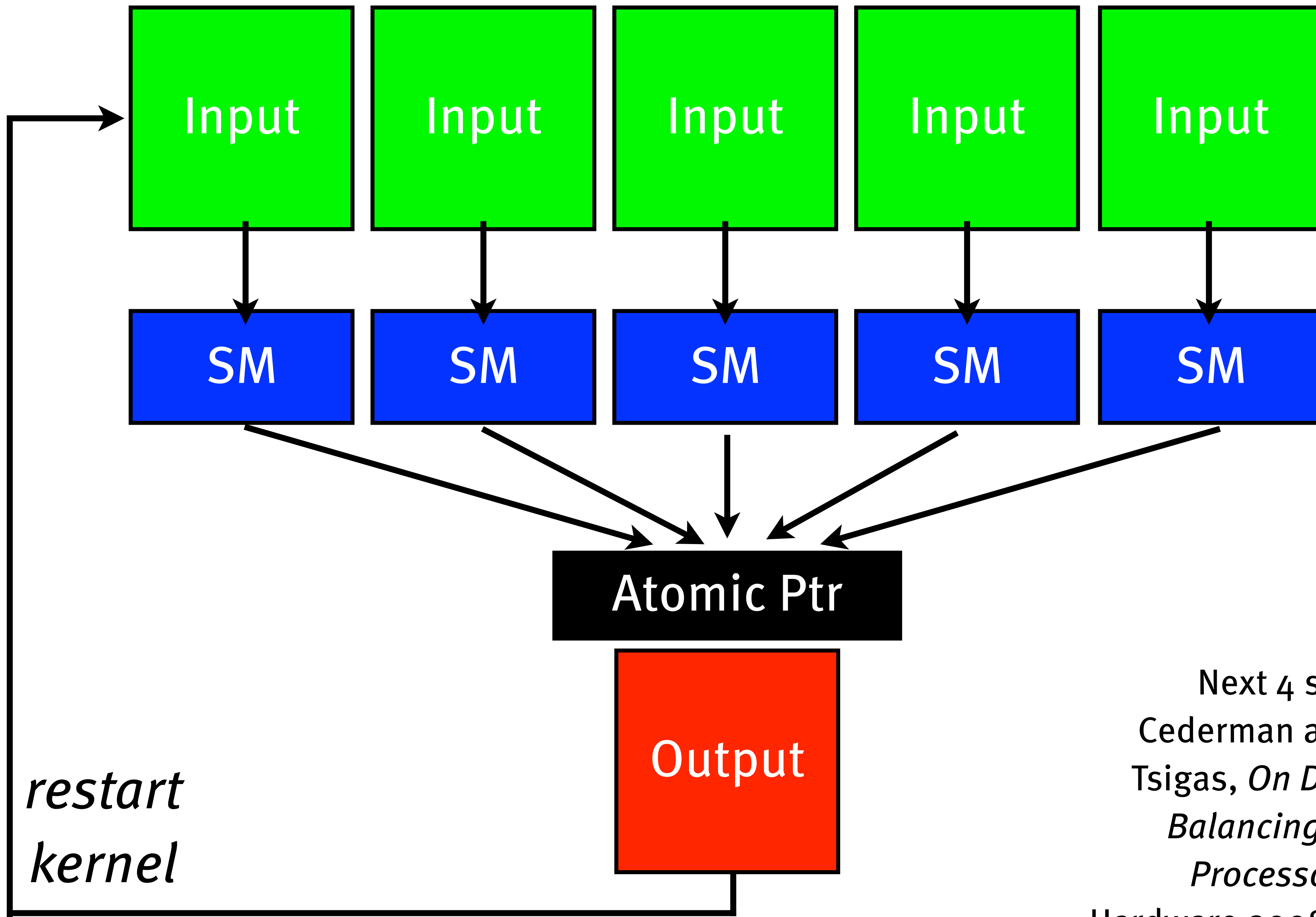
Work Queue Approach

- Allocate private work queue of tasks per core
  - Each core can add to or remove work from its local queue
- Cores mark self as idle if {queue exhausts storage, queue is empty}
- Cores periodically check global idle counter
- If global idle counter reaches threshold, rebalance work



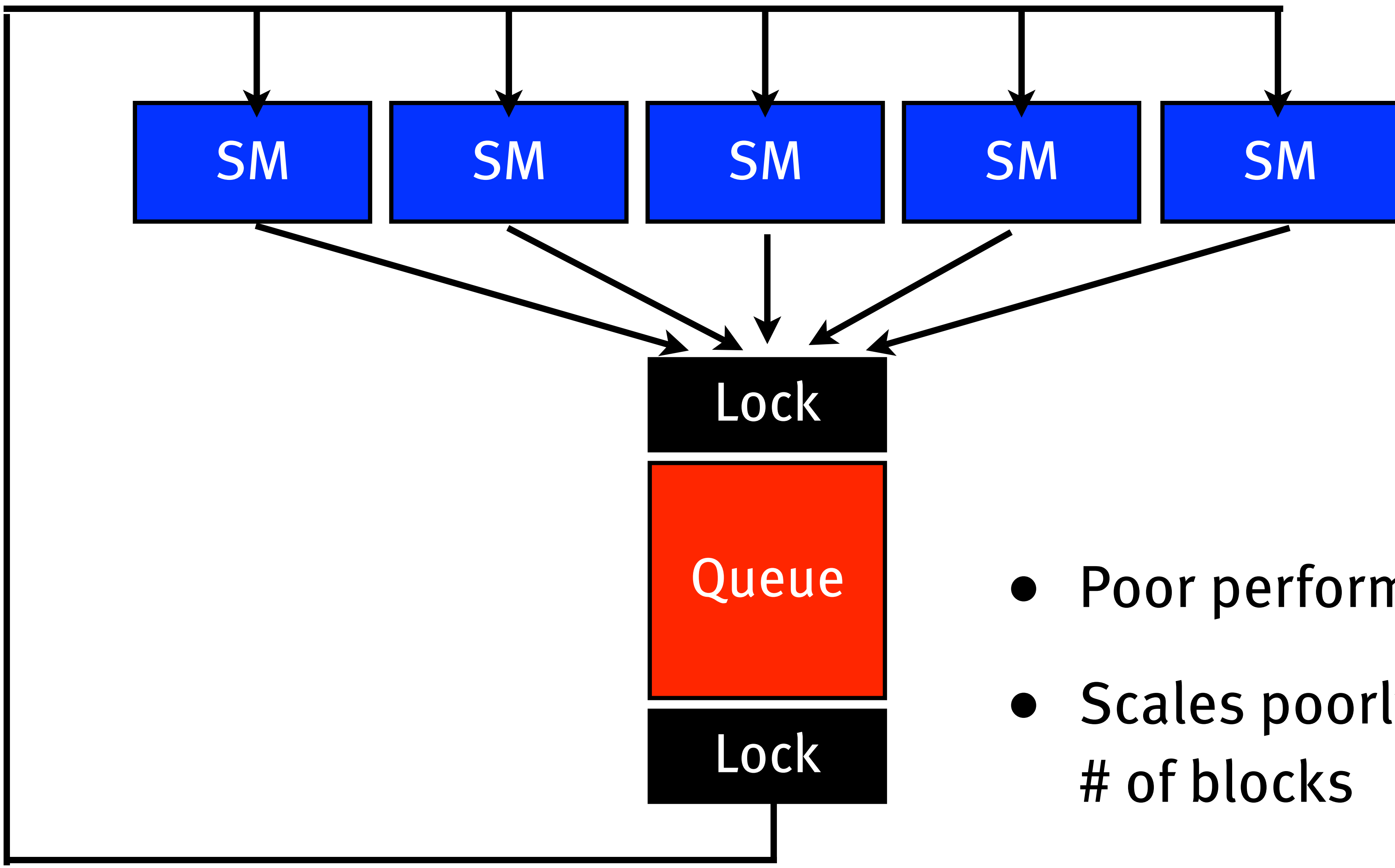
Fast Hierarchy Operations on GPU Architectures, Lauterbach et al.

Static Task List



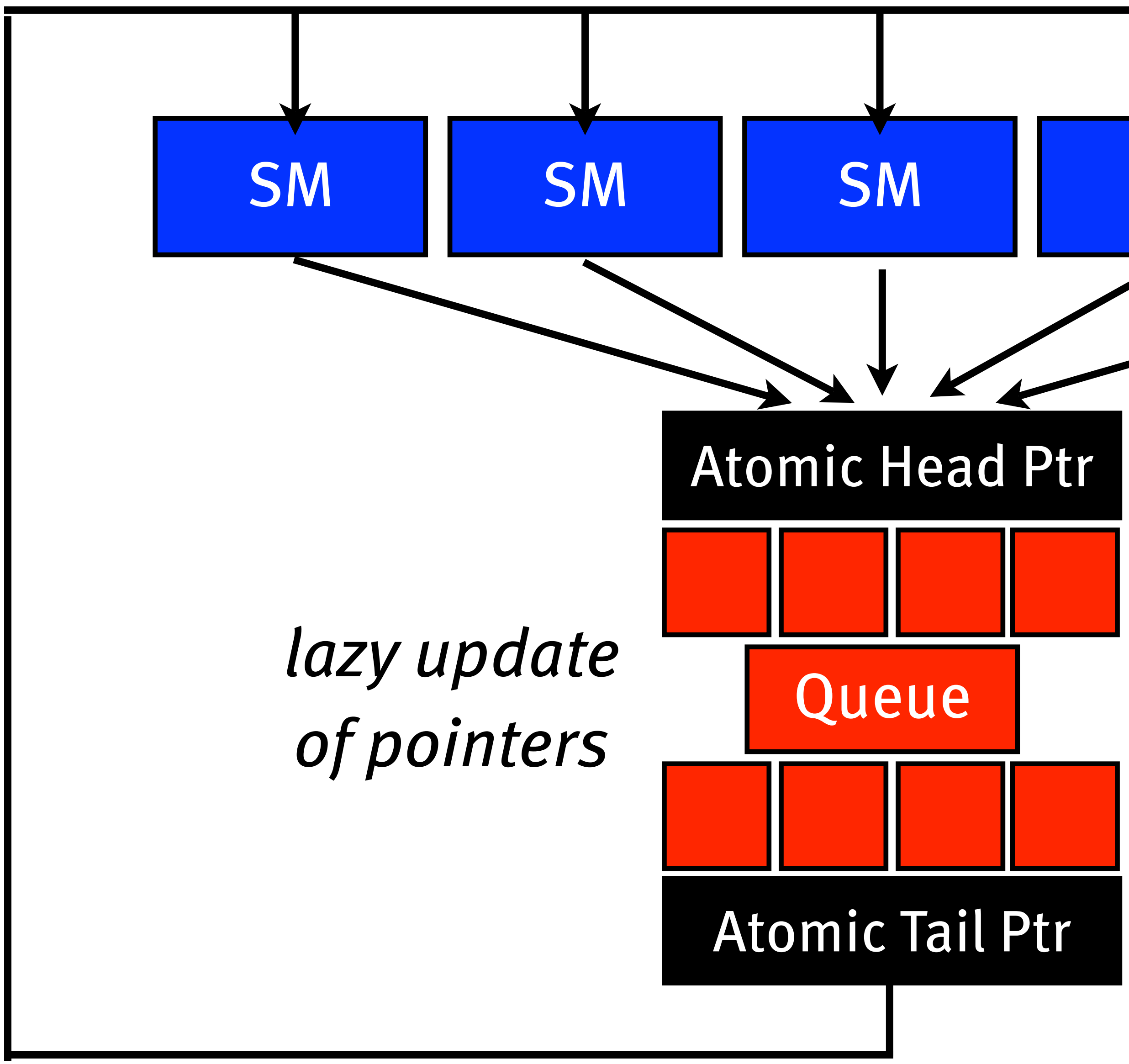
Next 4 slides: Daniel Cederman and Philippas Tsigas, *On Dynamic Load Balancing on Graphics Processors*. Graphics Hardware 2008, June 2008.

Blocking Dynamic Task Queue



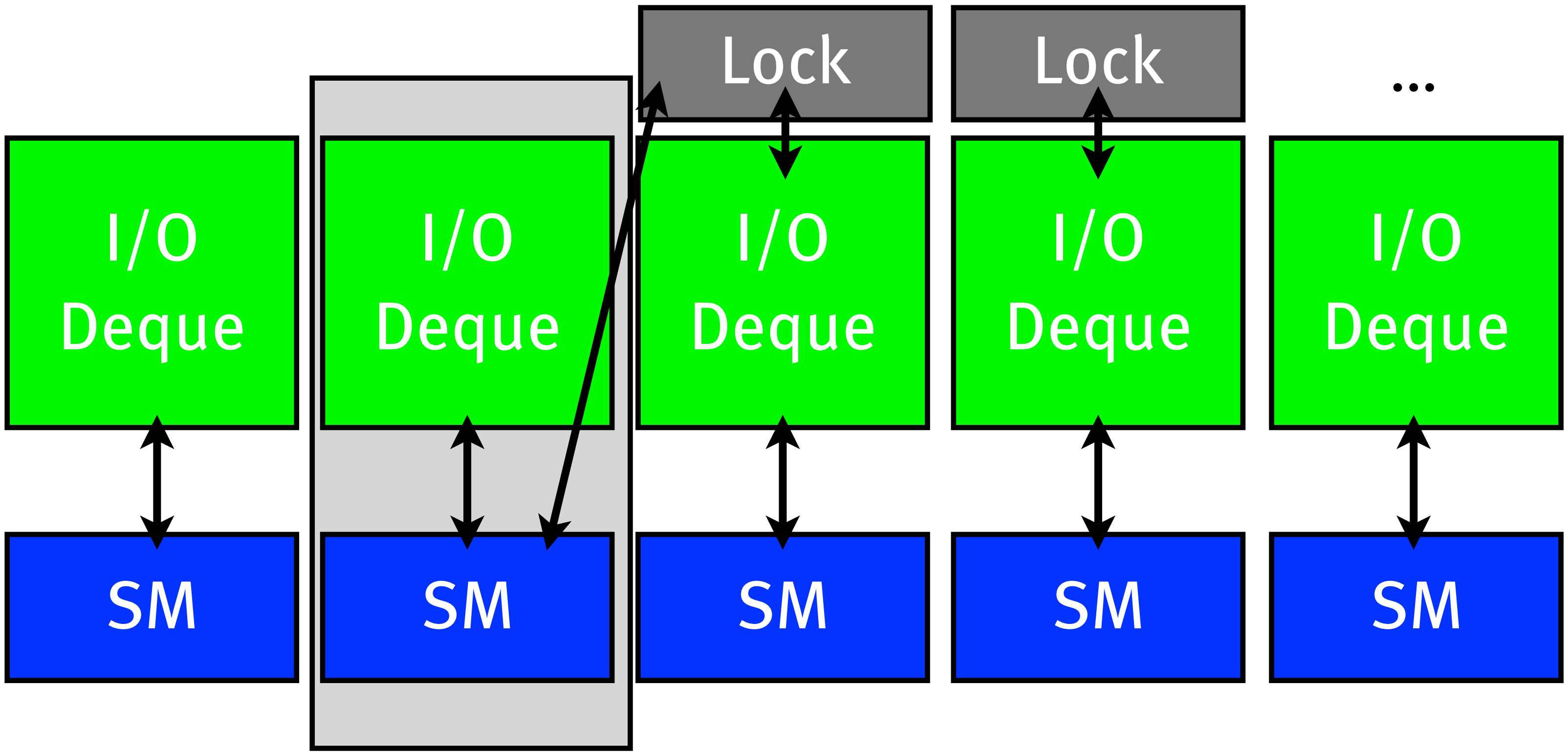
- Poor performance
- Scales poorly with # of blocks

Non-Blocking Dynamic Task Queue



- Better performance
- Scales well with small # of blocks, but poorer with large

Work Stealing



- Best performance and scalability
- Recent work by our group explored *task donating*
    - Win for memory consumption overall

Big-Picture Questions

- Relative cost of computation vs. overhead
- Frequency of global communication
- Cost of global communication
- Need for communication between GPU cores?
  - Would permit efficient in-kernel work stealing

DS Research Challenges

- String-based algorithms
- Building suffix trees (DNA sequence alignment)
- Graphs (vs. sparse matrix) and trees
- Dynamic programming
- Neighbor queries (kNN)
- Tuning
- True “parallel” data structures (not parallel versions of serial ones)?
- ***Incremental data structures***

Thanks to ...

- Nathan Bell, Michael Garland, David Luebke, Shubho Sengupta, and Dan Alcantara for helpful comments and slide material.
- Funding agencies: Department of Energy (SciDAC Institute for Ultrascale Visualization, Early Career Principal Investigator Award), NSF, BMW, NVIDIA, HP, Intel, UC MICRO, Rambus

Bibliography

- Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. *Real-Time Parallel Hashing on the GPU*. ACM Transactions on Graphics, 28(5), December 2009.

- Nathan Bell and Michael Garland. *Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors*. Proceedings of IEEE/ACM Supercomputing, November 2009.

- Daniel Cederman and Philippas Tsigas, *On Dynamic Load Balancing on Graphics Processors*. Graphics Hardware 2008, June 2008.

- C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. *Fast BVH Construction on GPUs*. Computer Graphics Forum (Proceedings of Eurographics 2009), 28(2), April 2009.

- Christian Lauterbach, Qi Mo, and Dinesh Manocha. *Fast Hierarchy Operations on GPU Architectures*. Tech report, April 2009, UNC Chapel Hill.

- Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. *Scan Primitives for GPU Computing*. In Graphics Hardware 2007, August 2007.

- Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. *Real-Time KD-Tree Construction on Graphics Hardware*. ACM Transactions on Graphics, 27(5), December 2008.