Irregular Algorithms & Data Structures

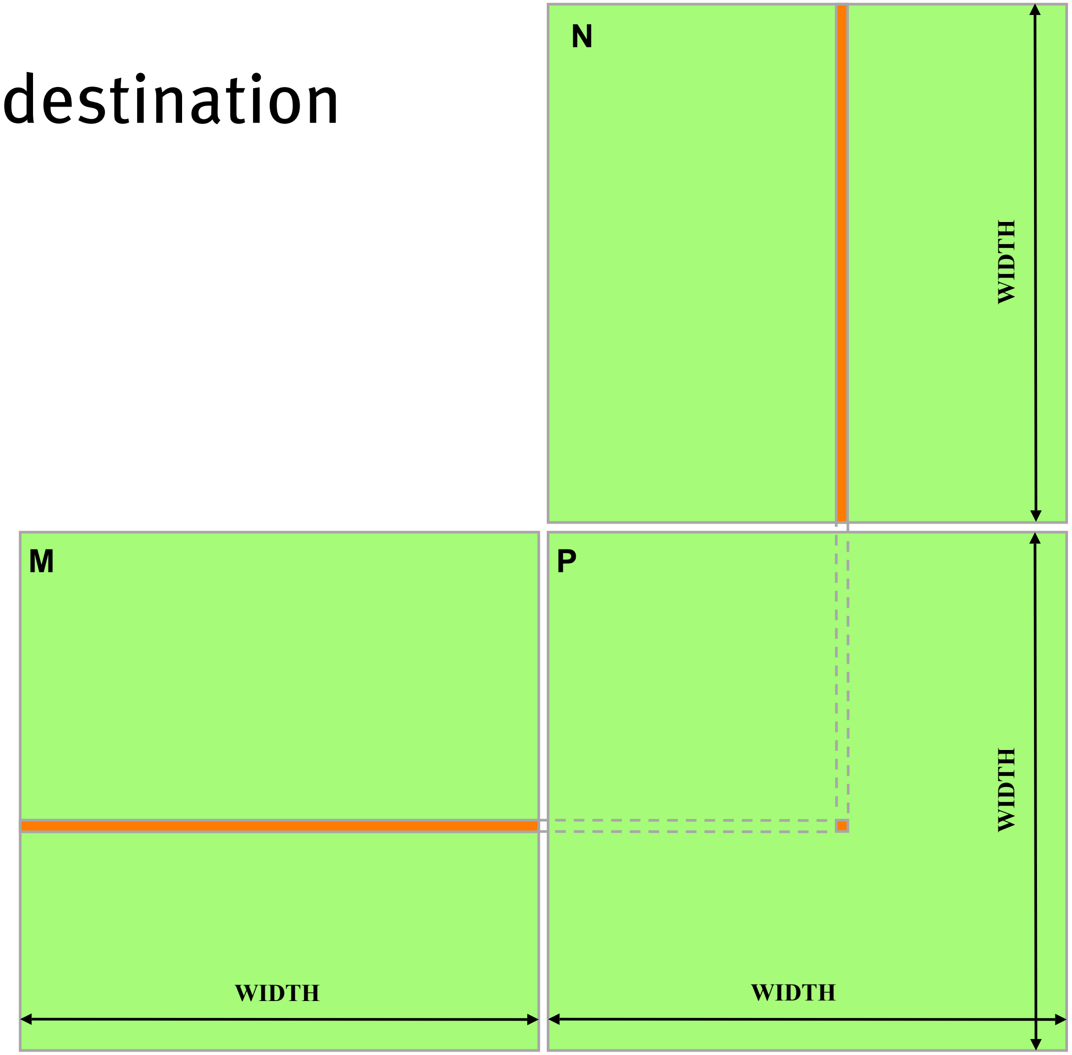
John Owens
Associate Professor, Electrical and Computer Engineering
SciDAC Institute for Ultrascale Visualization
University of California, Davis

Design Principles

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

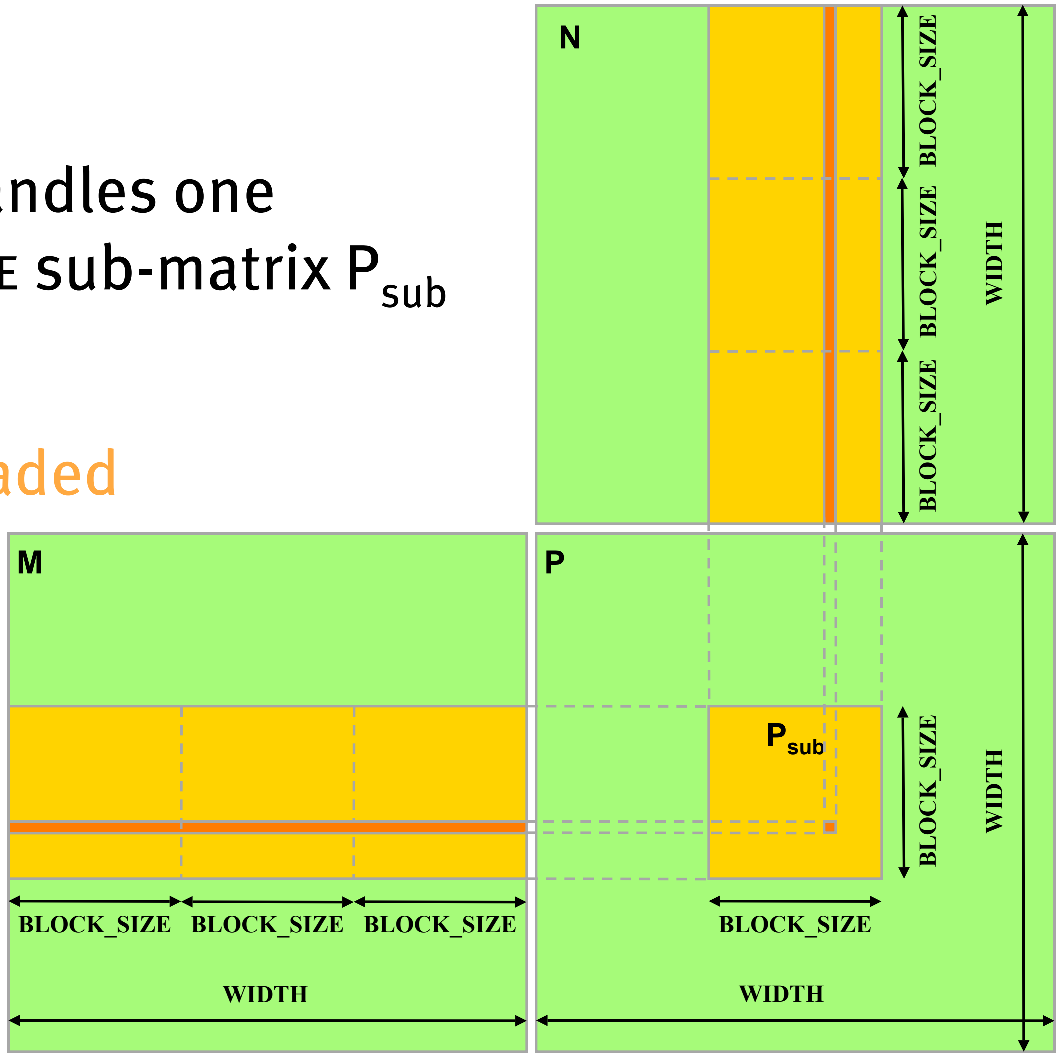
Dense Matrix Multiplication

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



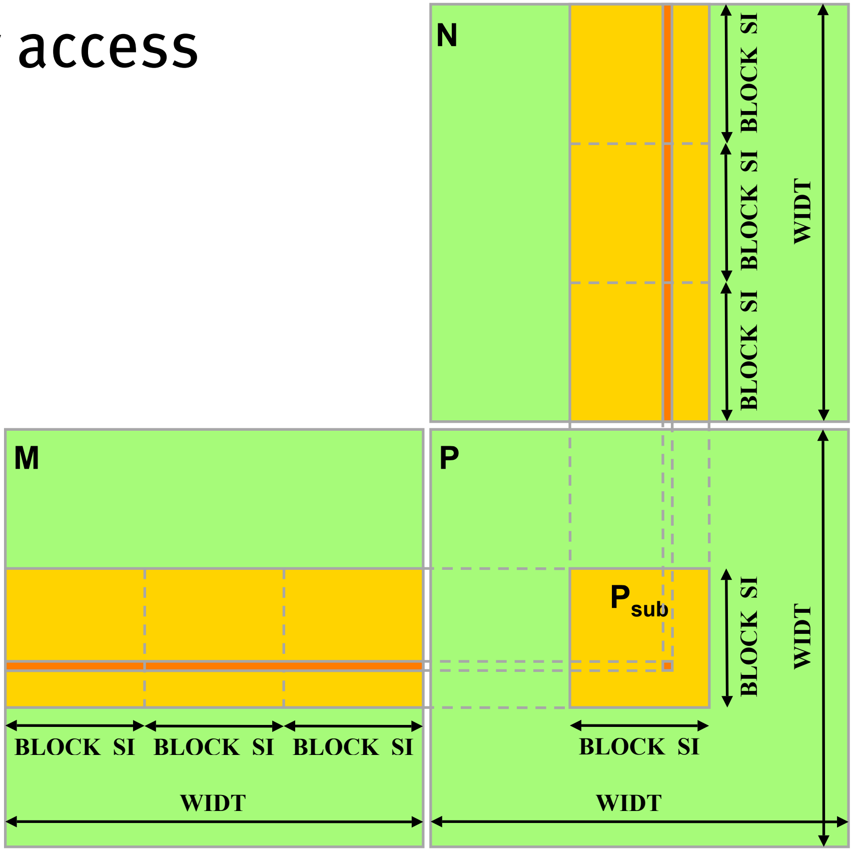
Dense Matrix Multiplication

- 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!



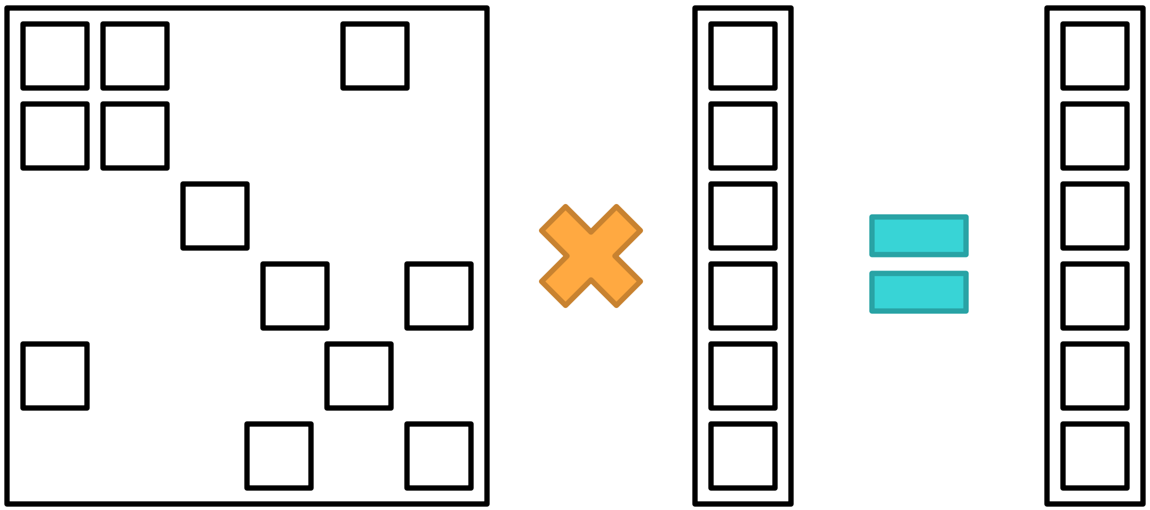
Dense Matrix Multiplication

- 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?

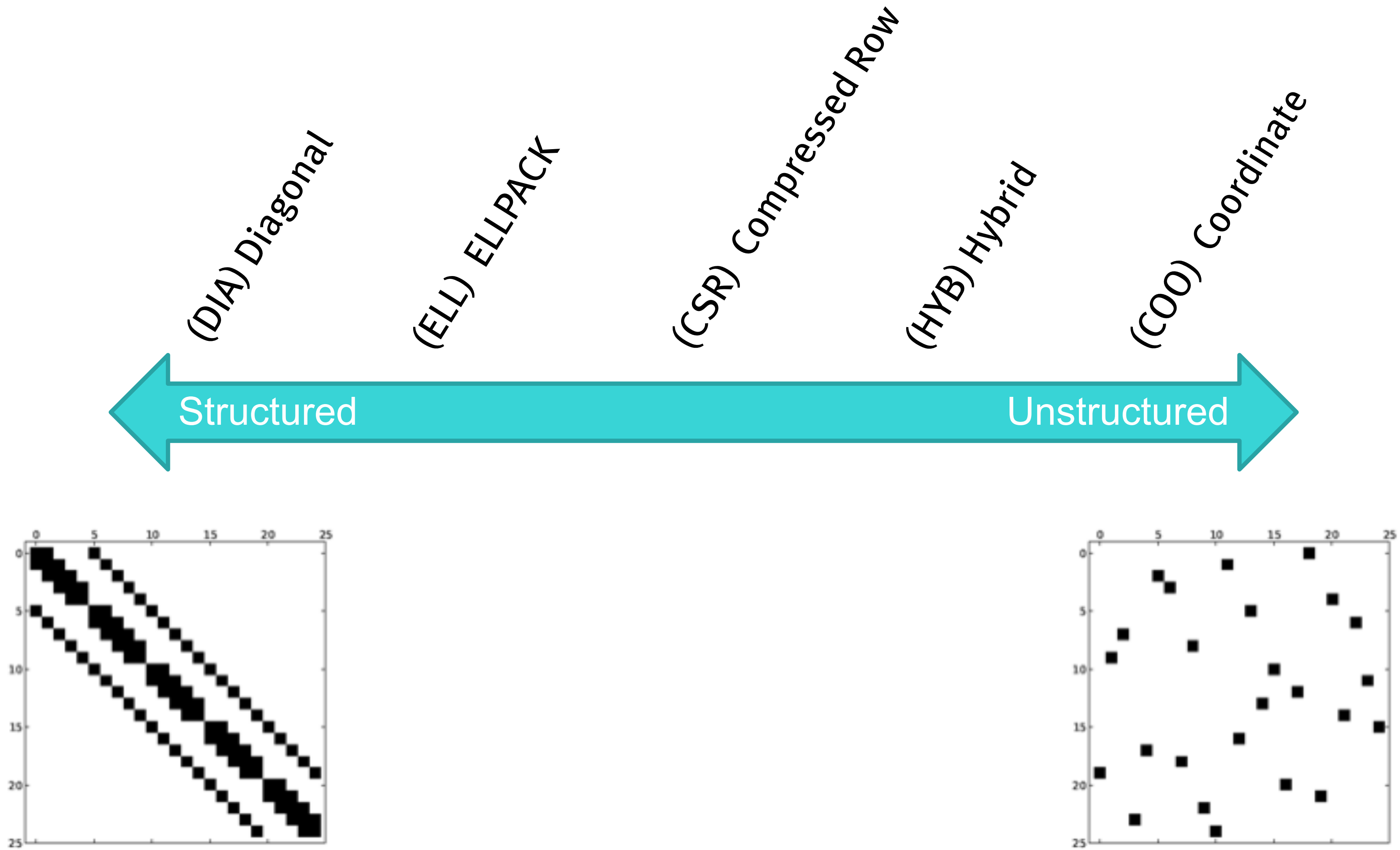
- Dense approach is wasteful
- Unclear how to map work to parallel processors
- Irregular data access



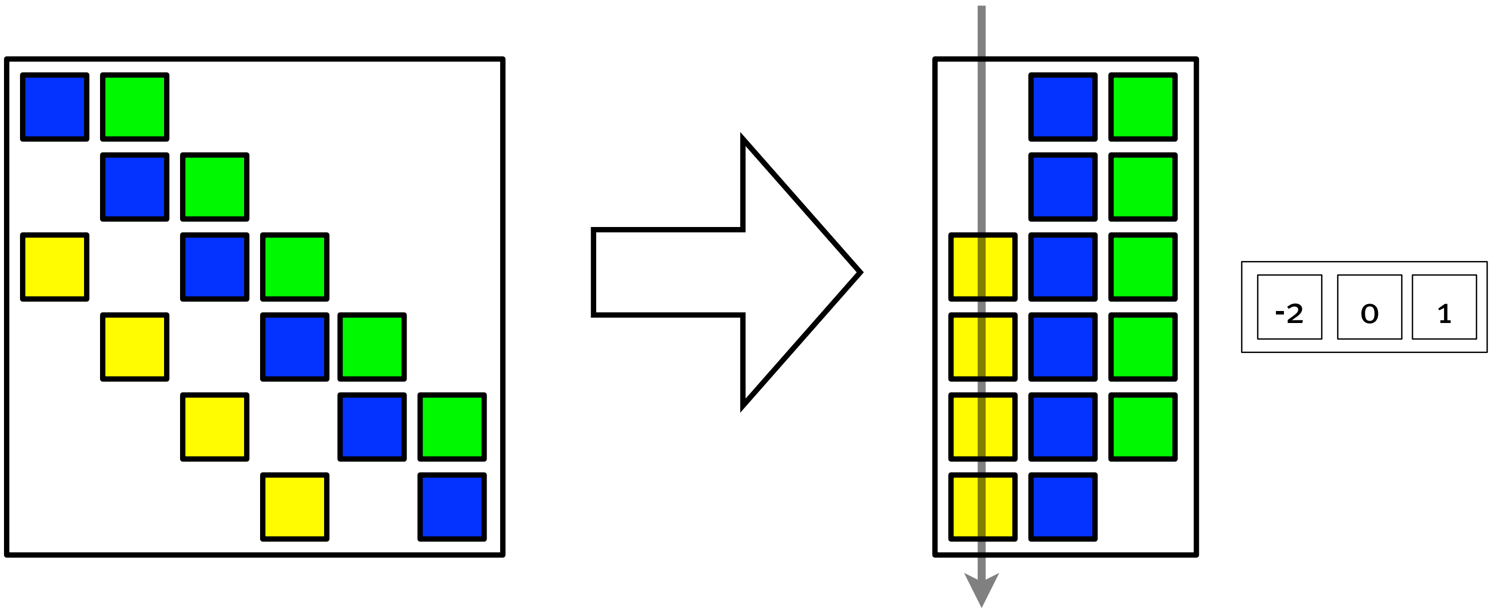
Go see the paper!

- “Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors” by Nathan Bell and Michael Garland, NVIDIA Research
    - Tuesday Nov 17, 2–2:30p, PB252

Sparse Matrix Formats

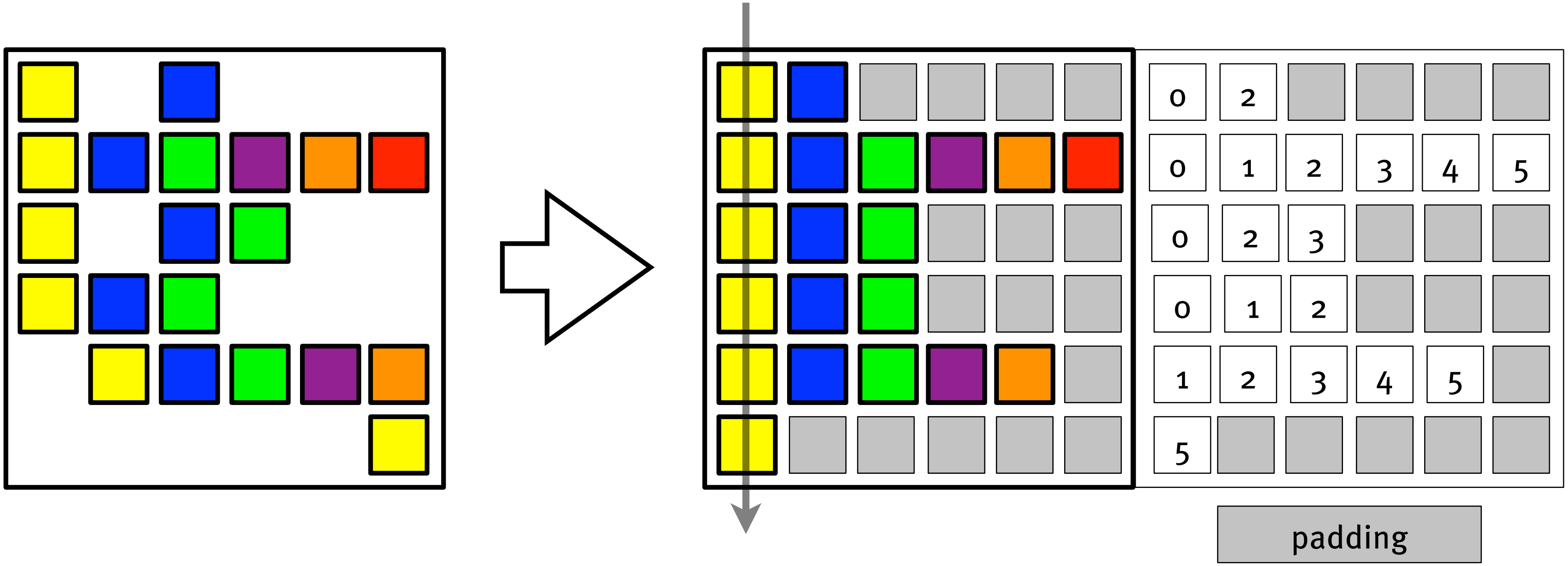


Diagonal Matrices



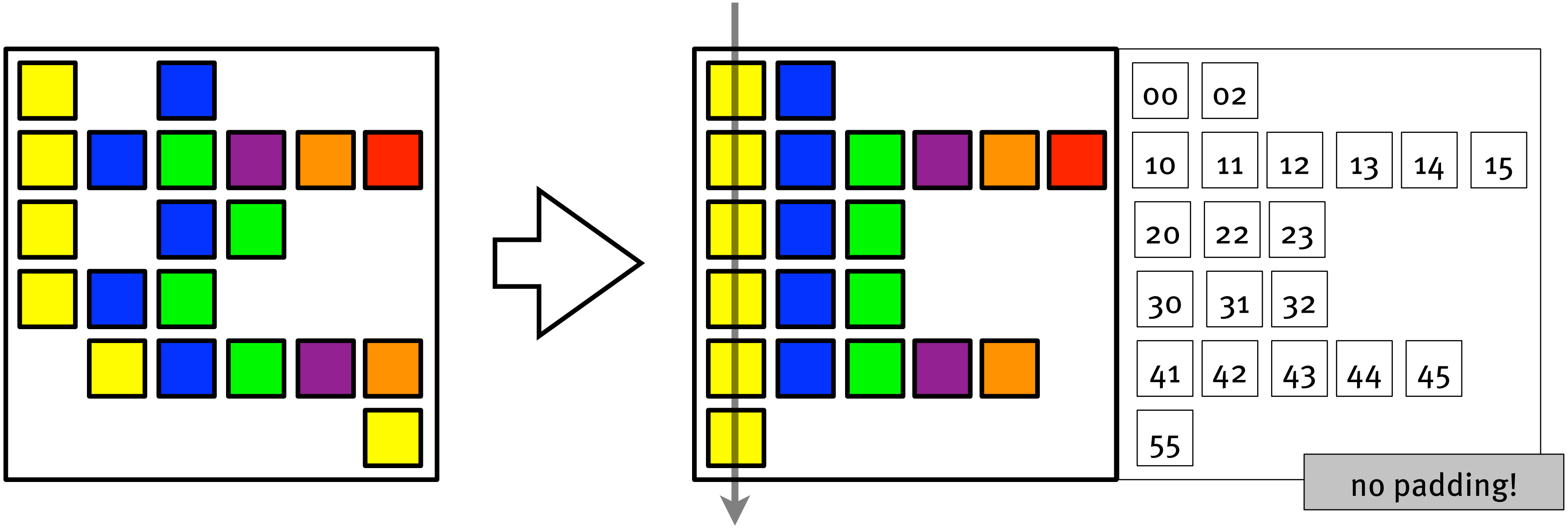
- Diagonals should be mostly populated
- Map one thread per row
  - Good parallel efficiency
  - Good memory behavior [column-major storage]

Irregular Matrices: ELL



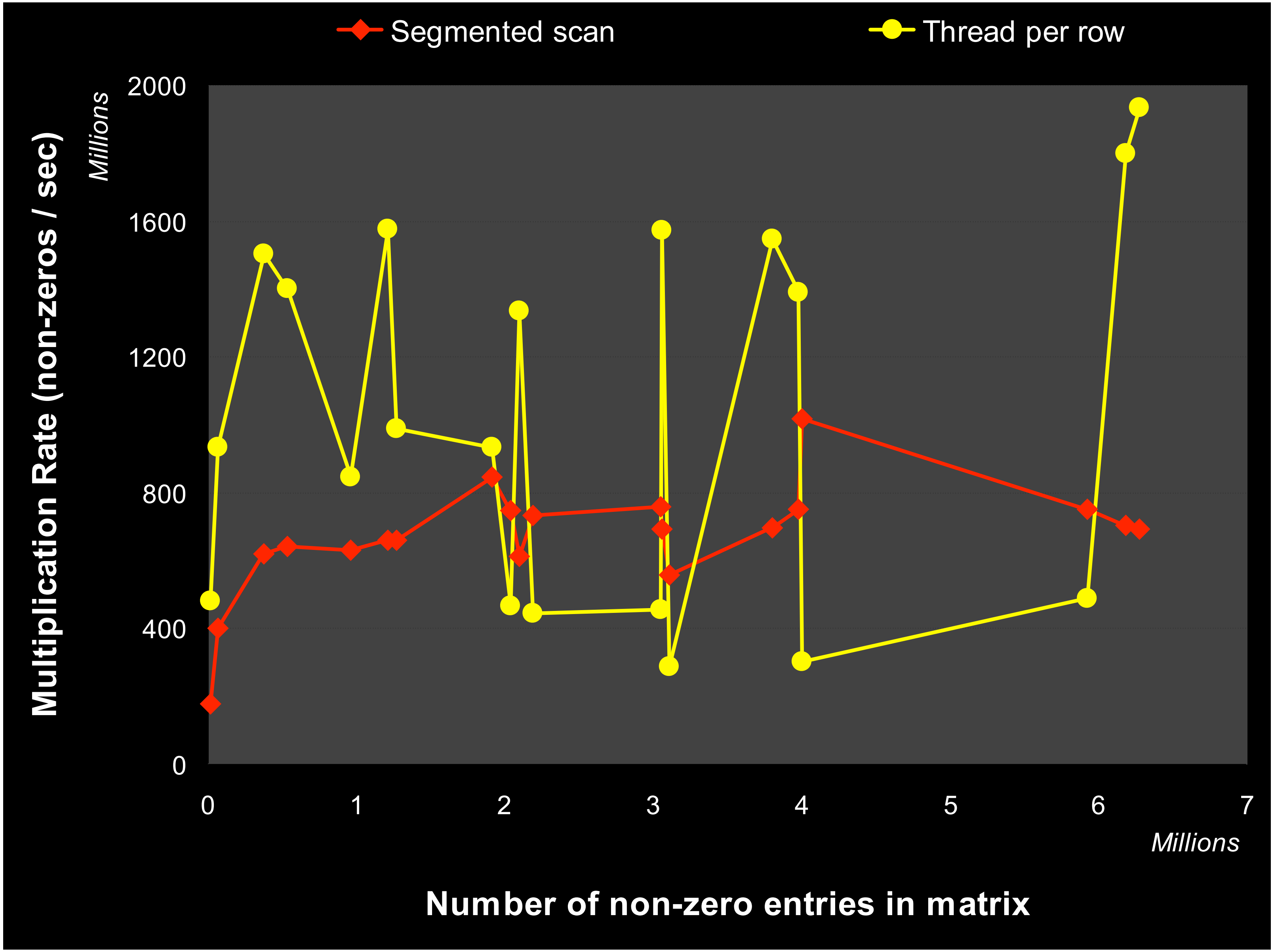
- Assign one thread per row again
- But now:
    - Load imbalance hurts parallel efficiency

Irregular Matrices: COO

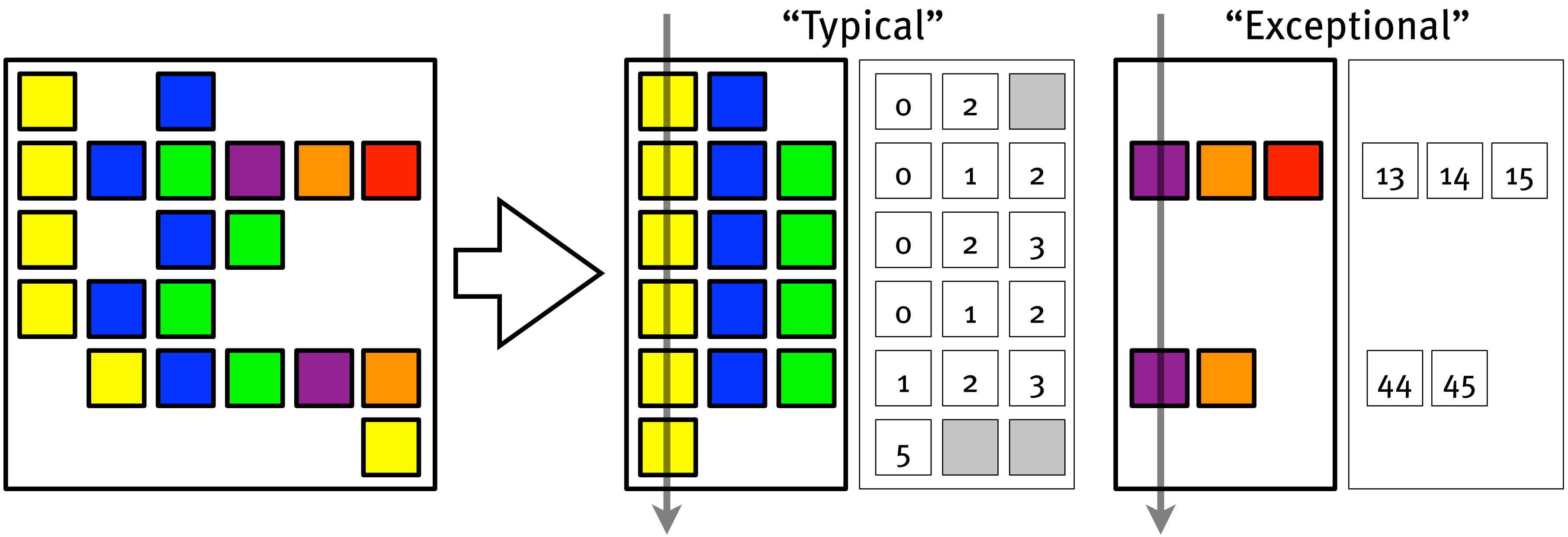


- General format; insensitive to sparsity pattern, but ~3x slower than ELL
- Assign one thread per element, combine results from all elements in a row to get output element
  - Req segmented reduction, communication btwn threads

Thread-per-{element,row}



Irregular Matrices: HYB

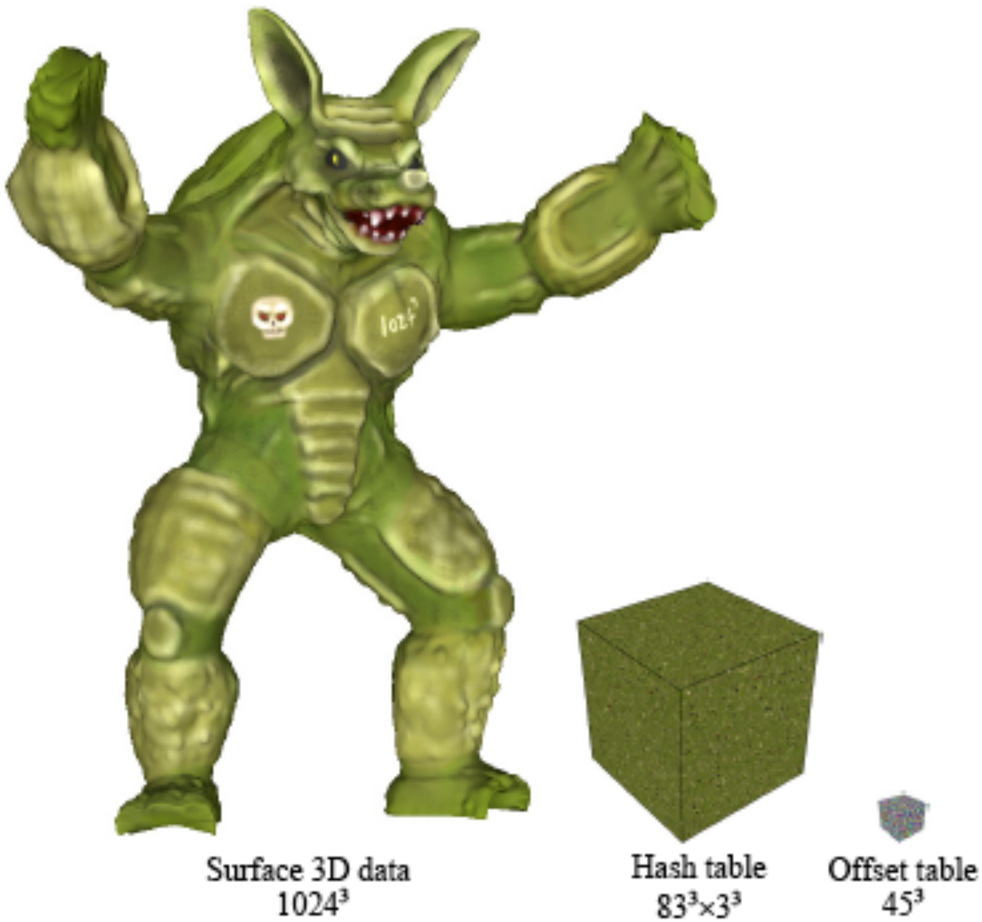


- Combine regularity of ELL + flexibility of COO

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

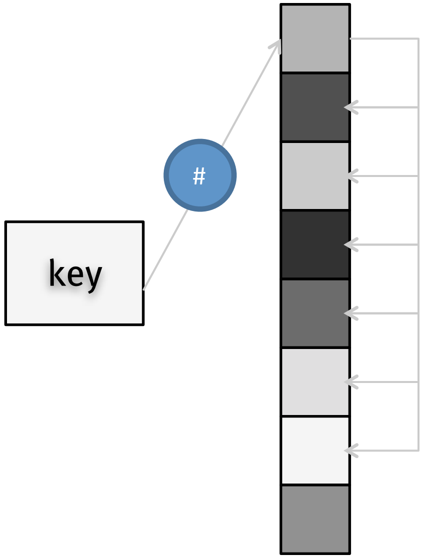
Hash Tables & Sparsity



- Lefebvre and Hoppe, Siggraph 2006

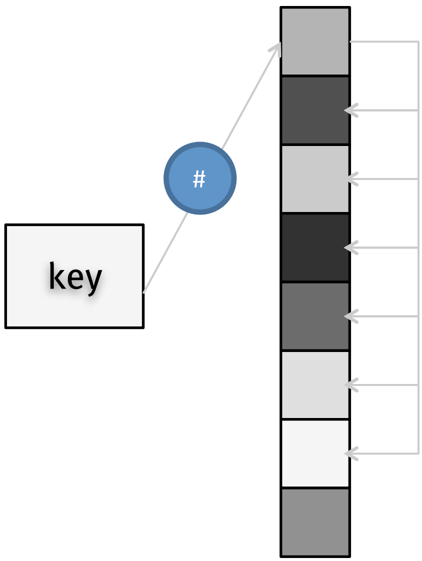
Scalar Hashing

Scalar Hashing

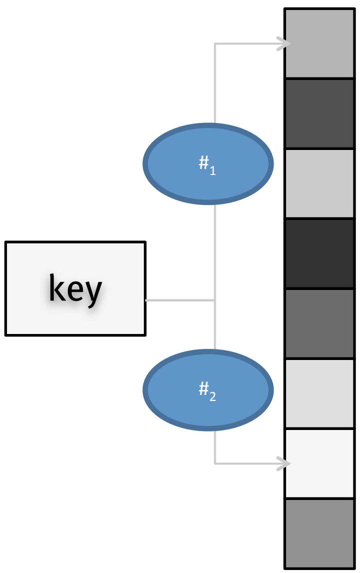


Linear Probing

Scalar Hashing

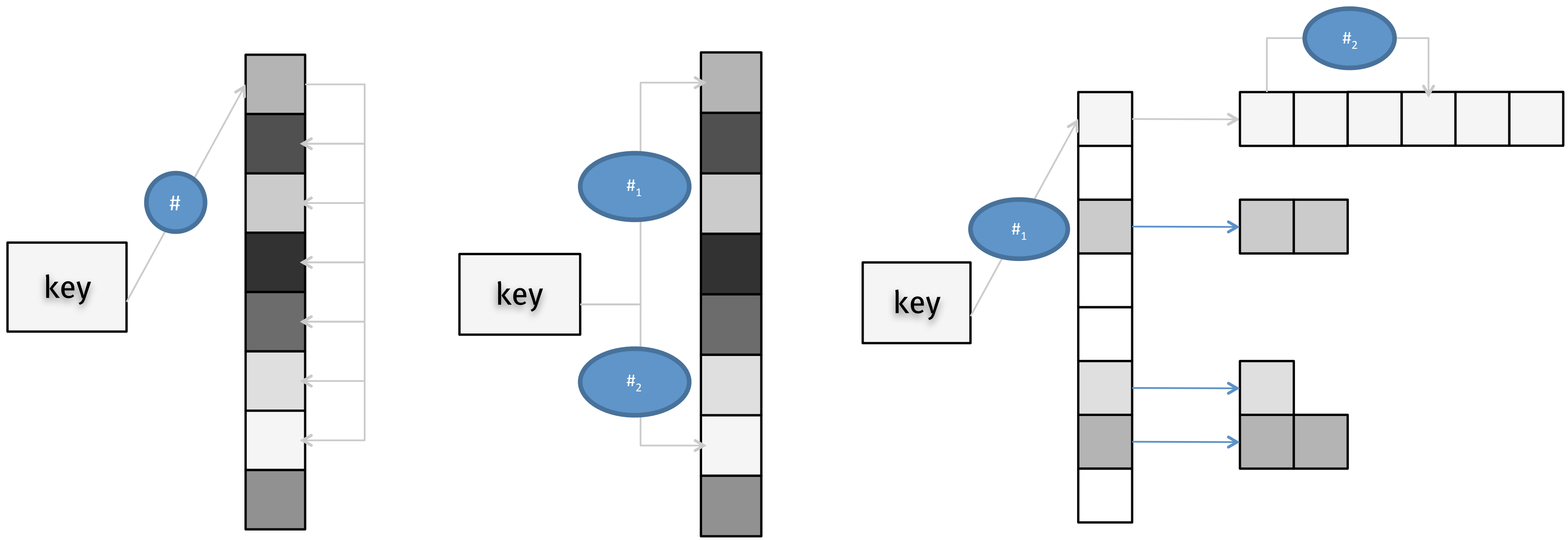


Linear Probing



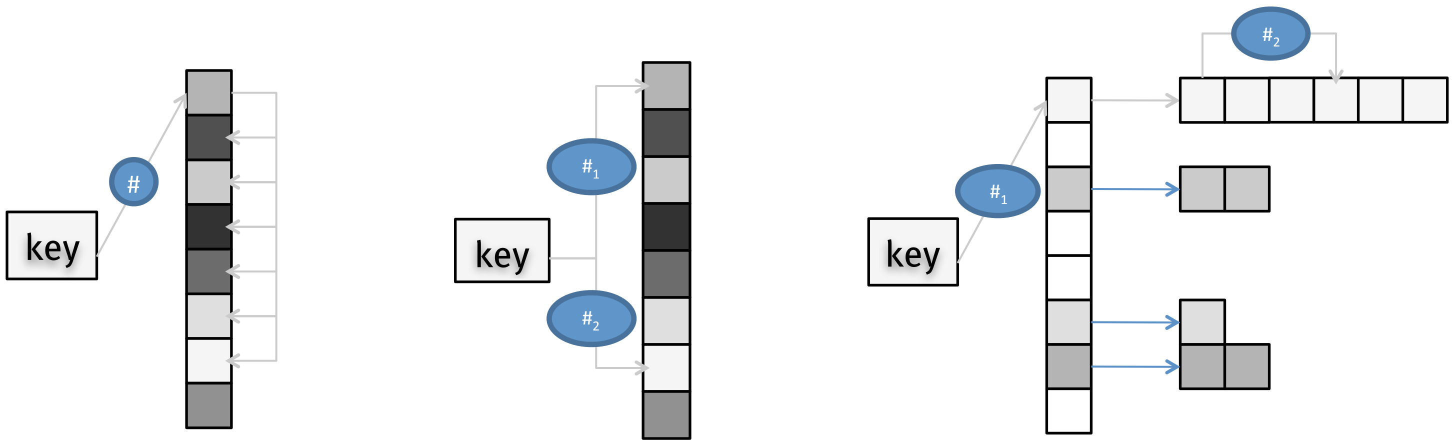
Double Probing

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

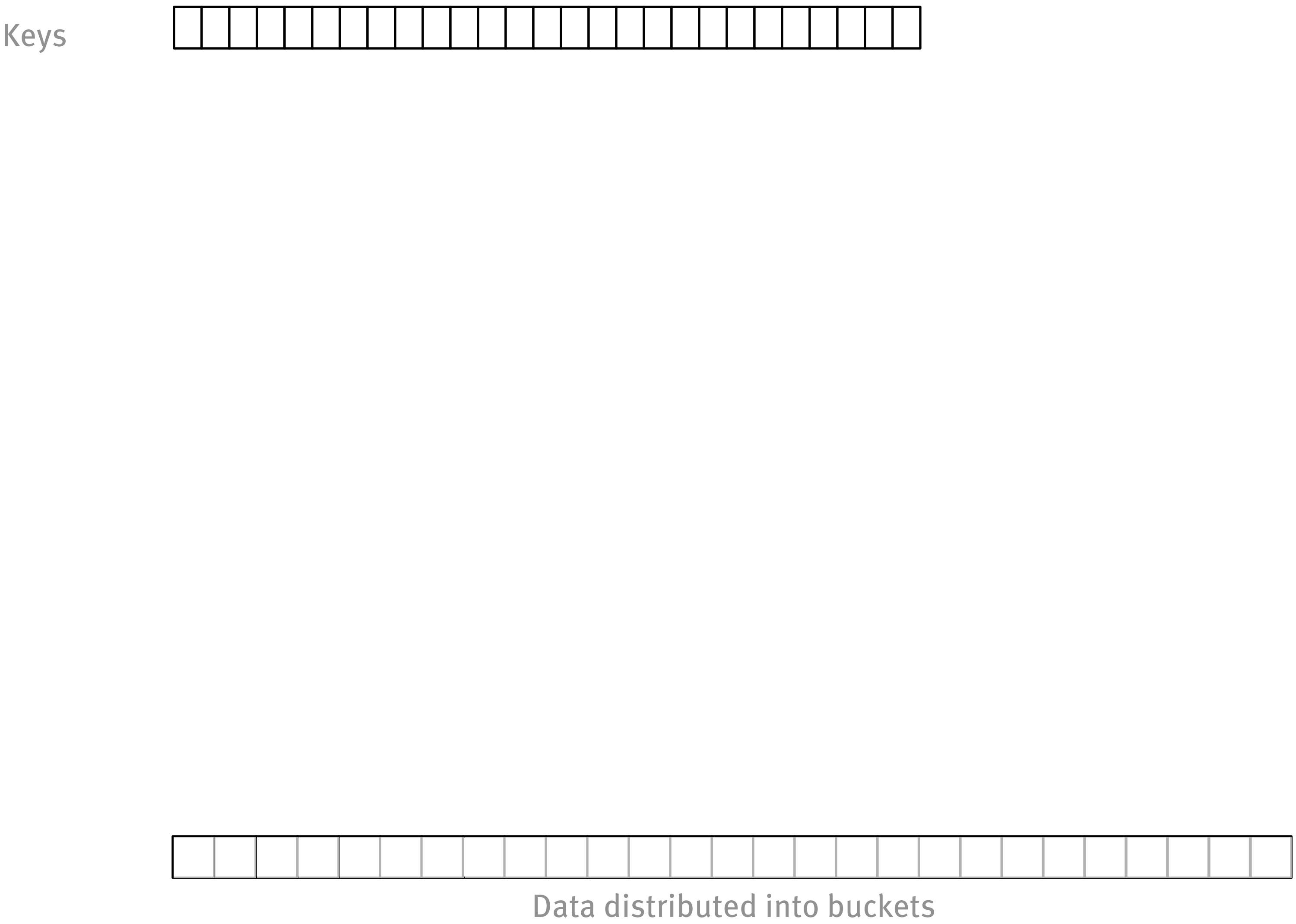
- Hash tables are good for sparse data.
- Input: Set of key-value pairs to place in the hash table
- Output: Data structure that allows:
  - Determining if key has been placed in hash table
  - Given the key, fetching its value
- Could also:
  - Sort key-value pairs by key (construction)
  - Binary-search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

Parallel Hashing: What We Want

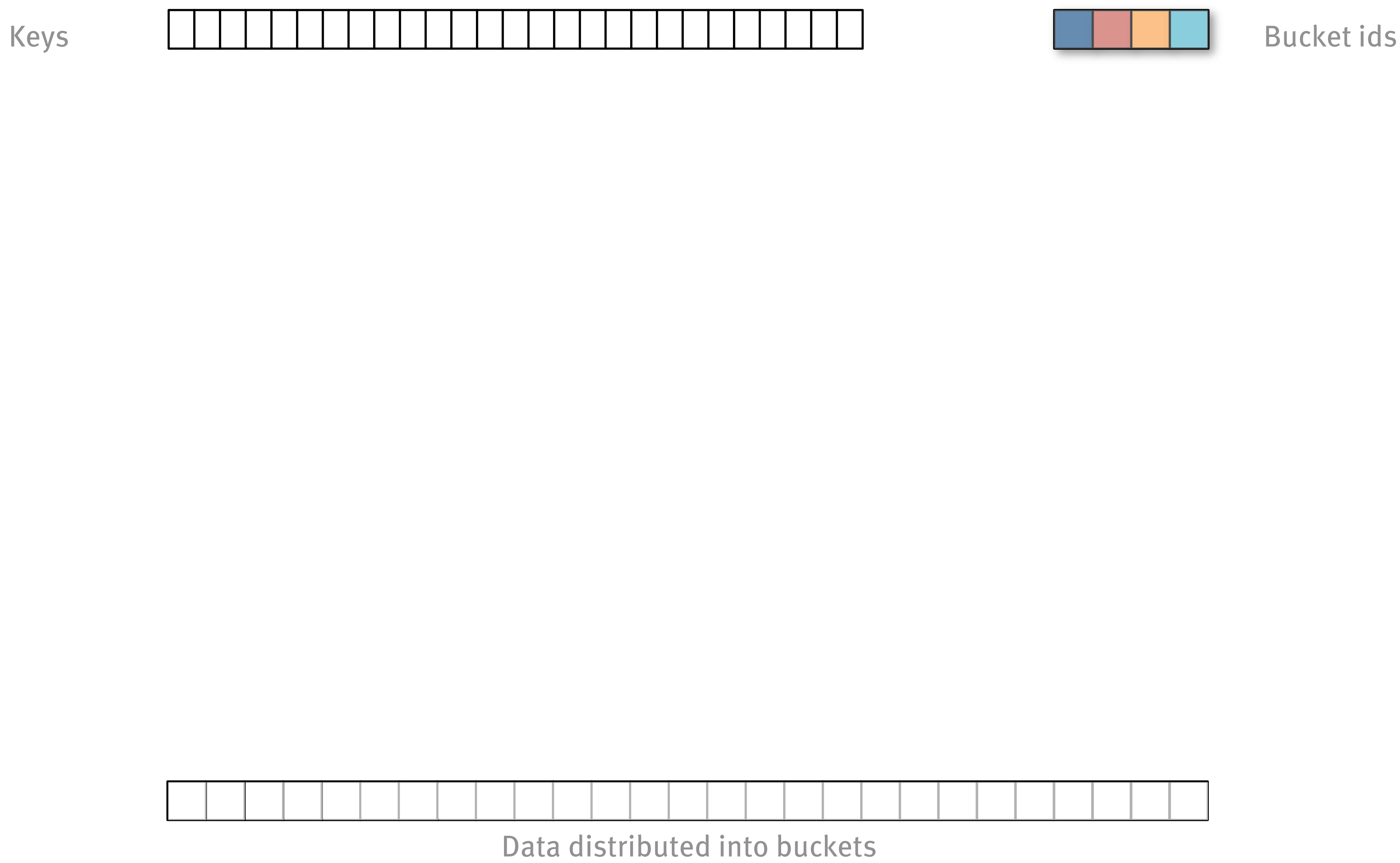
- Fast construction time
- Fast access time
  - $O(1)$ for any element, $O(n)$ for n elements in parallel
- Reasonable memory usage

- 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

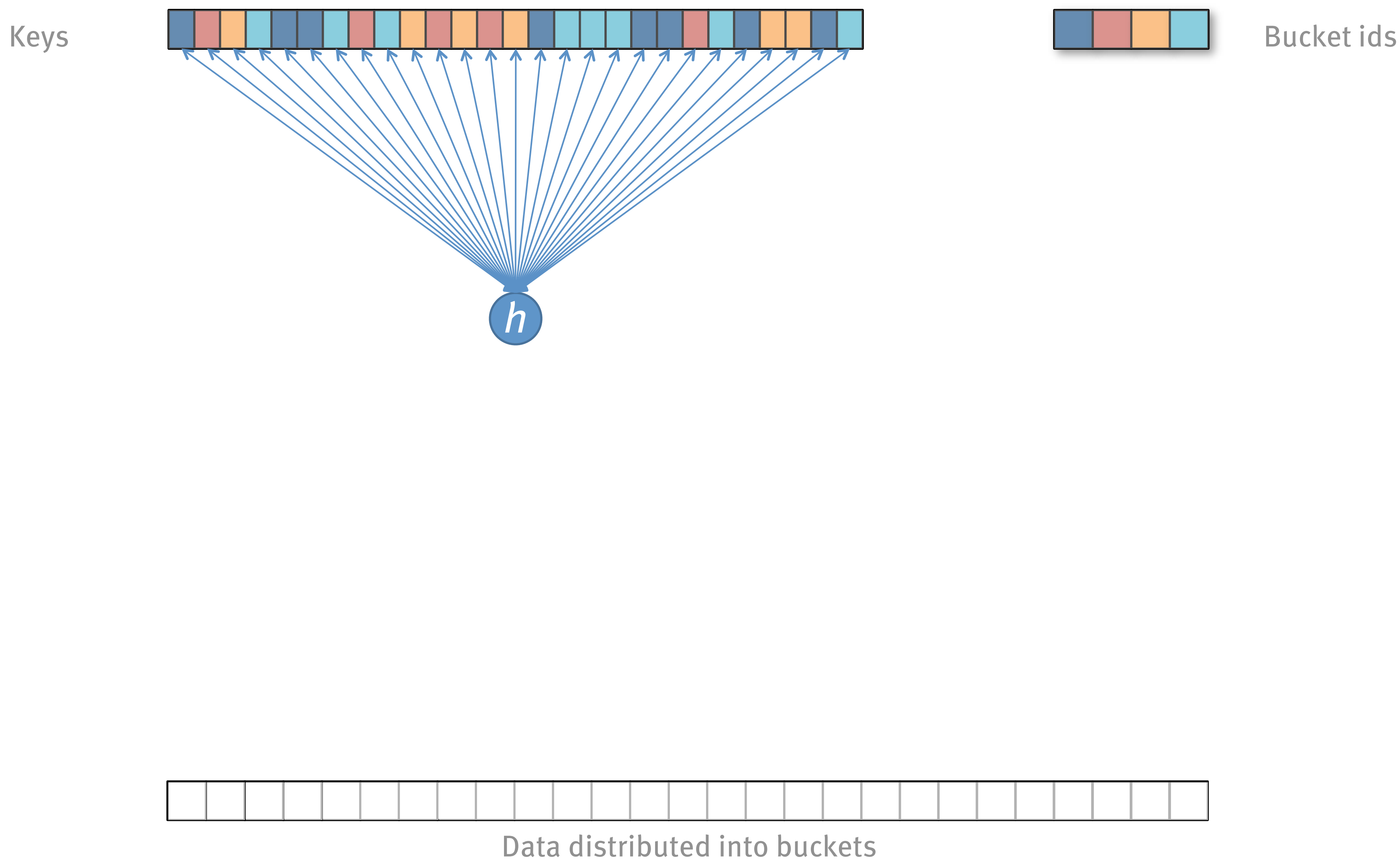
Level 1: Distribute into buckets



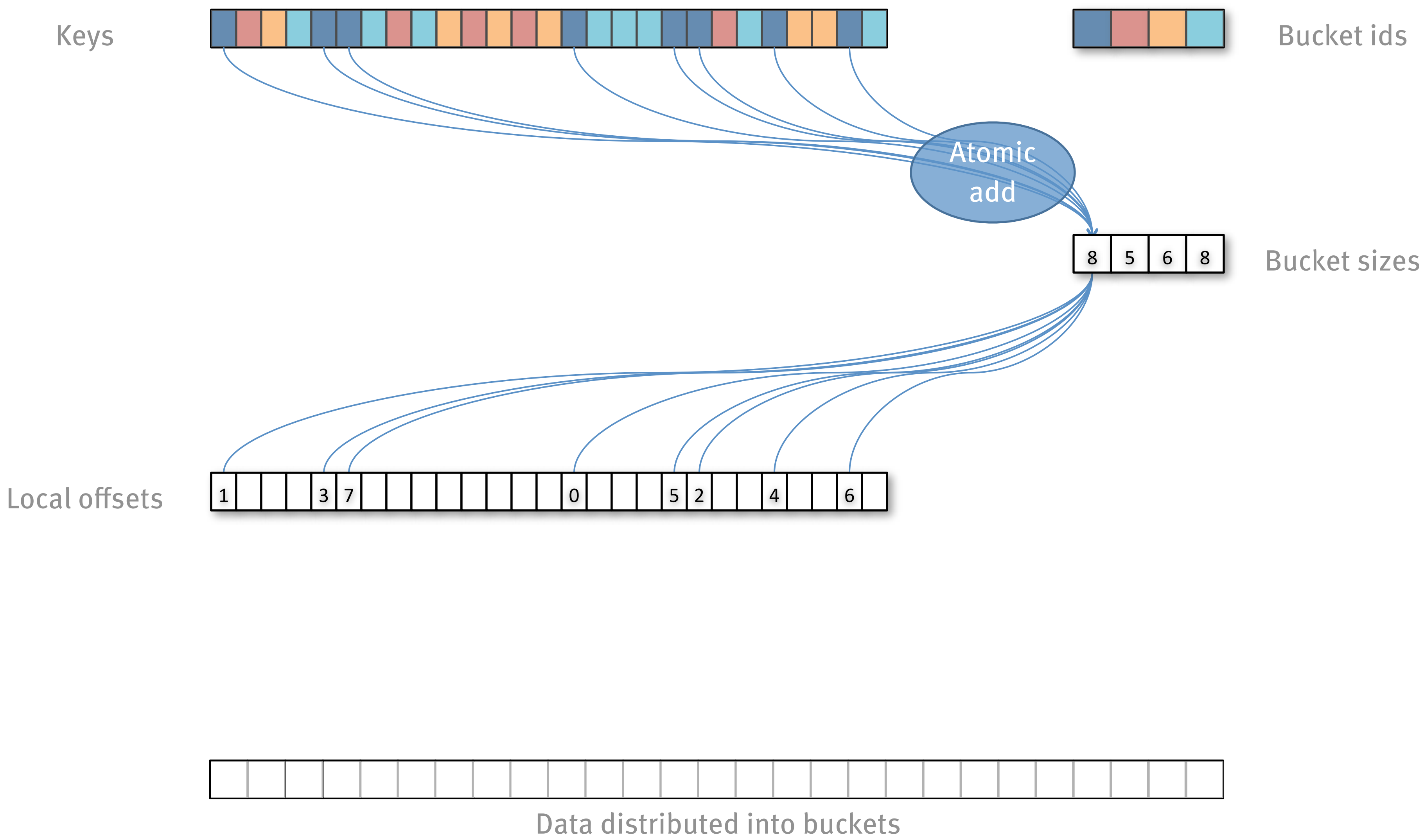
Level 1: Distribute into buckets



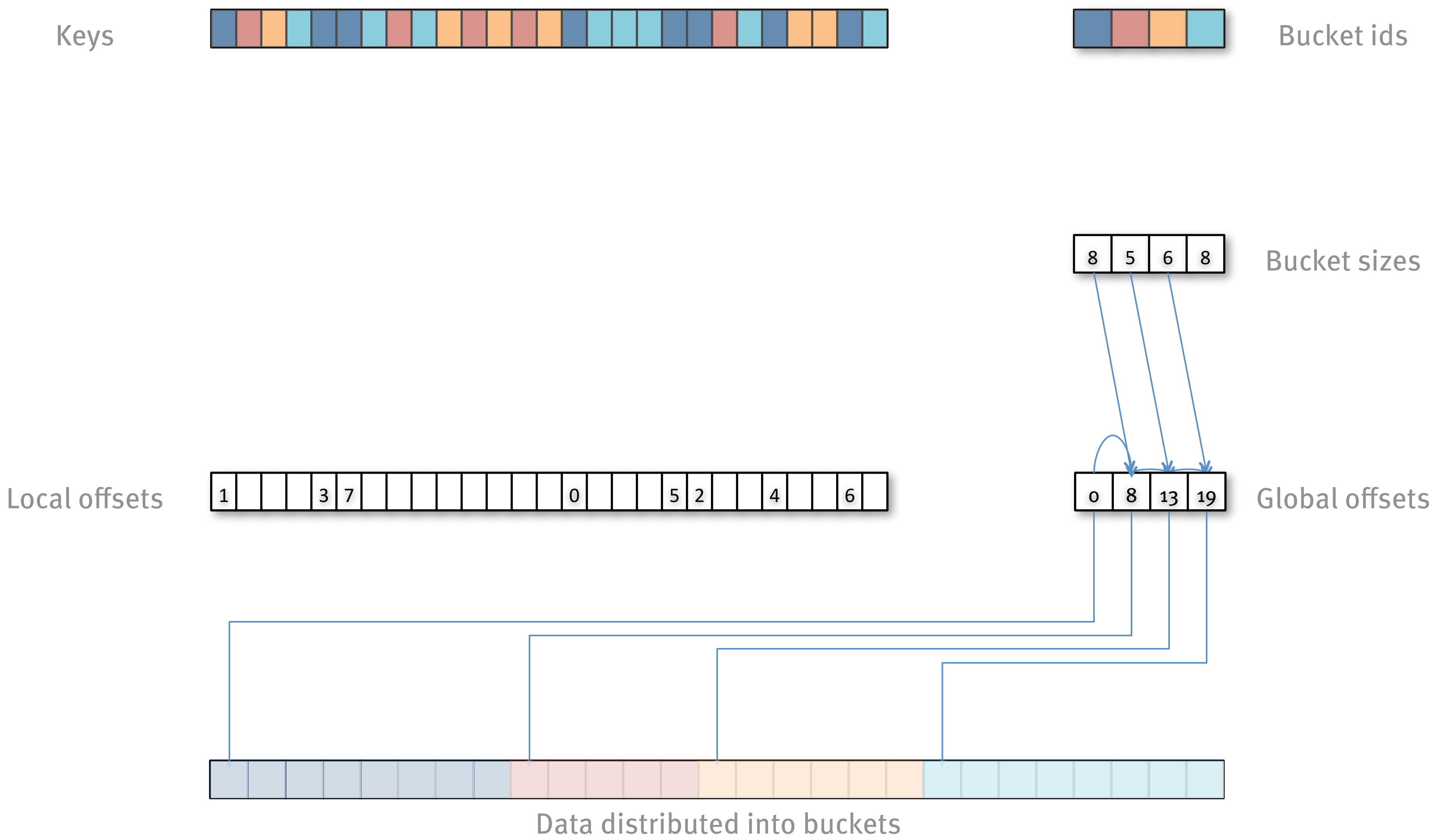
Level 1: Distribute into buckets



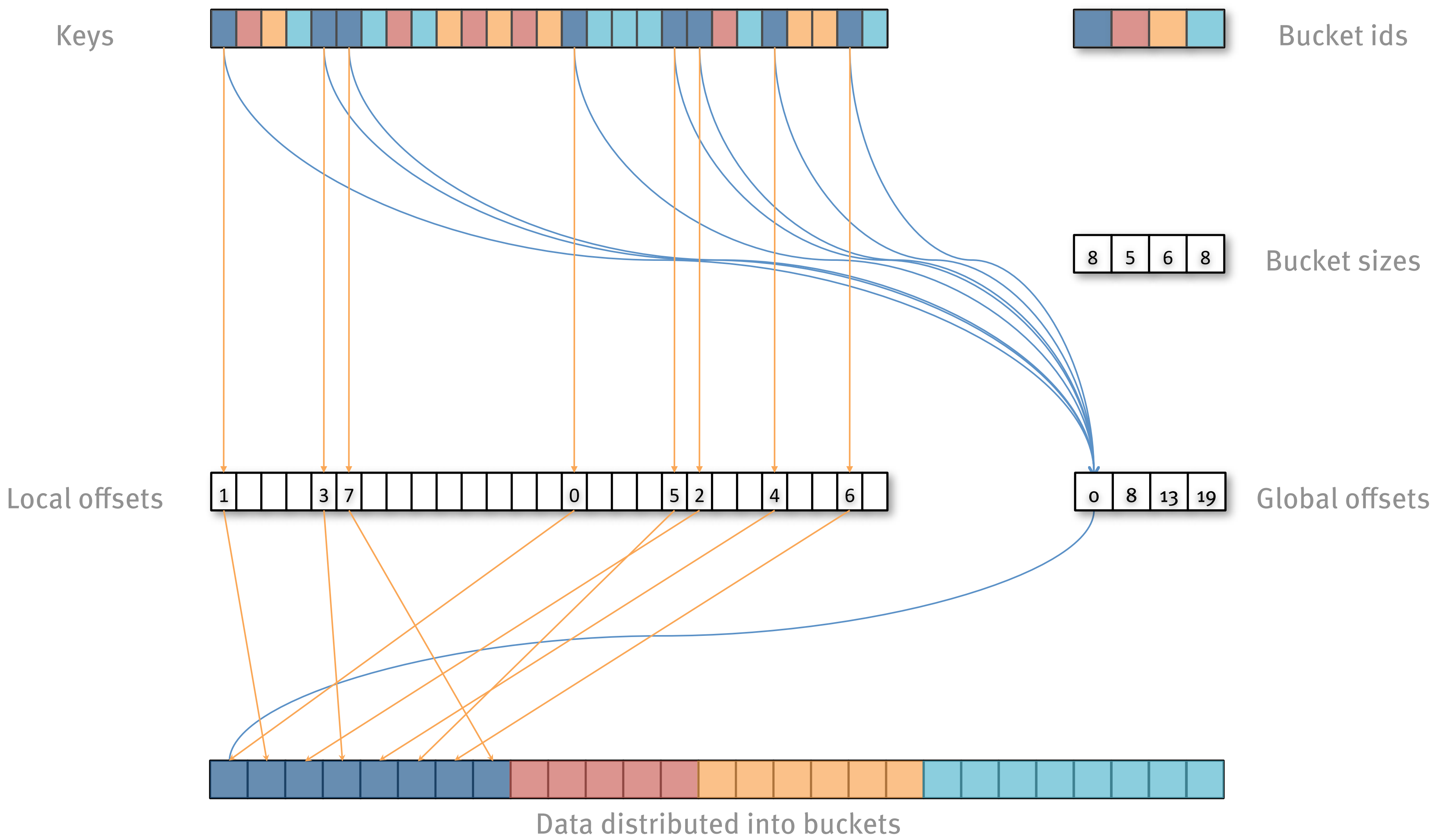
Level 1: Distribute into buckets



Level 1: Distribute into buckets



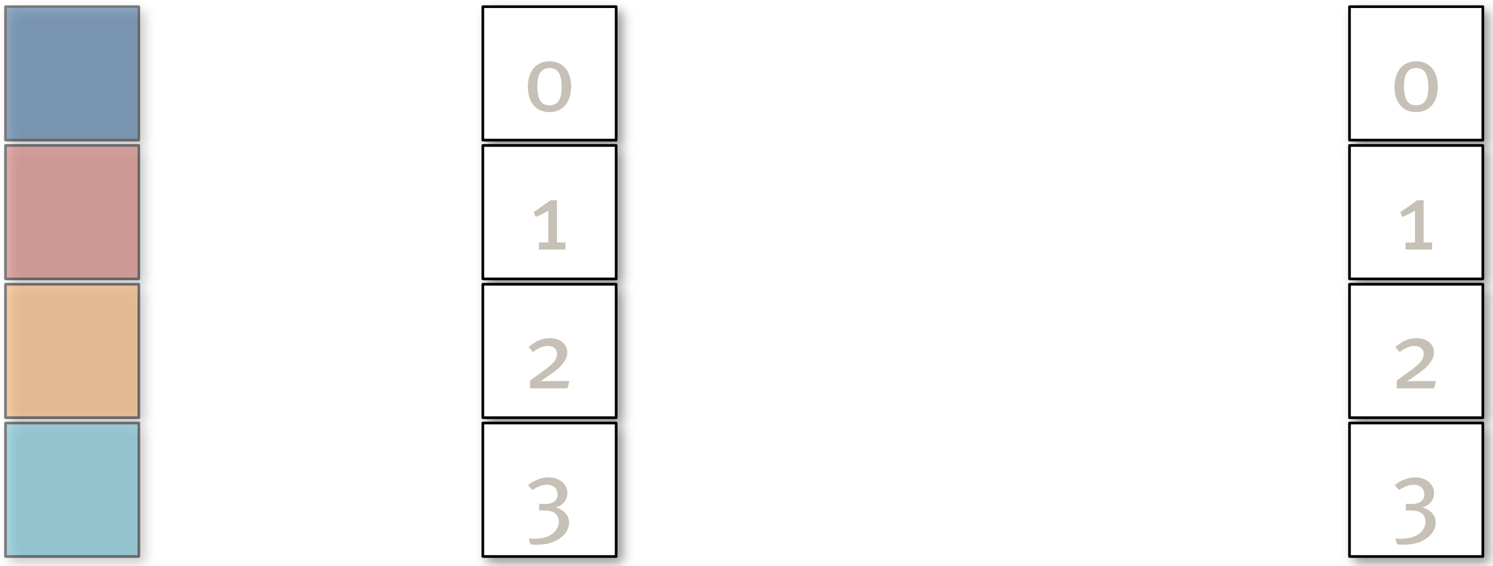
Level 1: Distribute into buckets



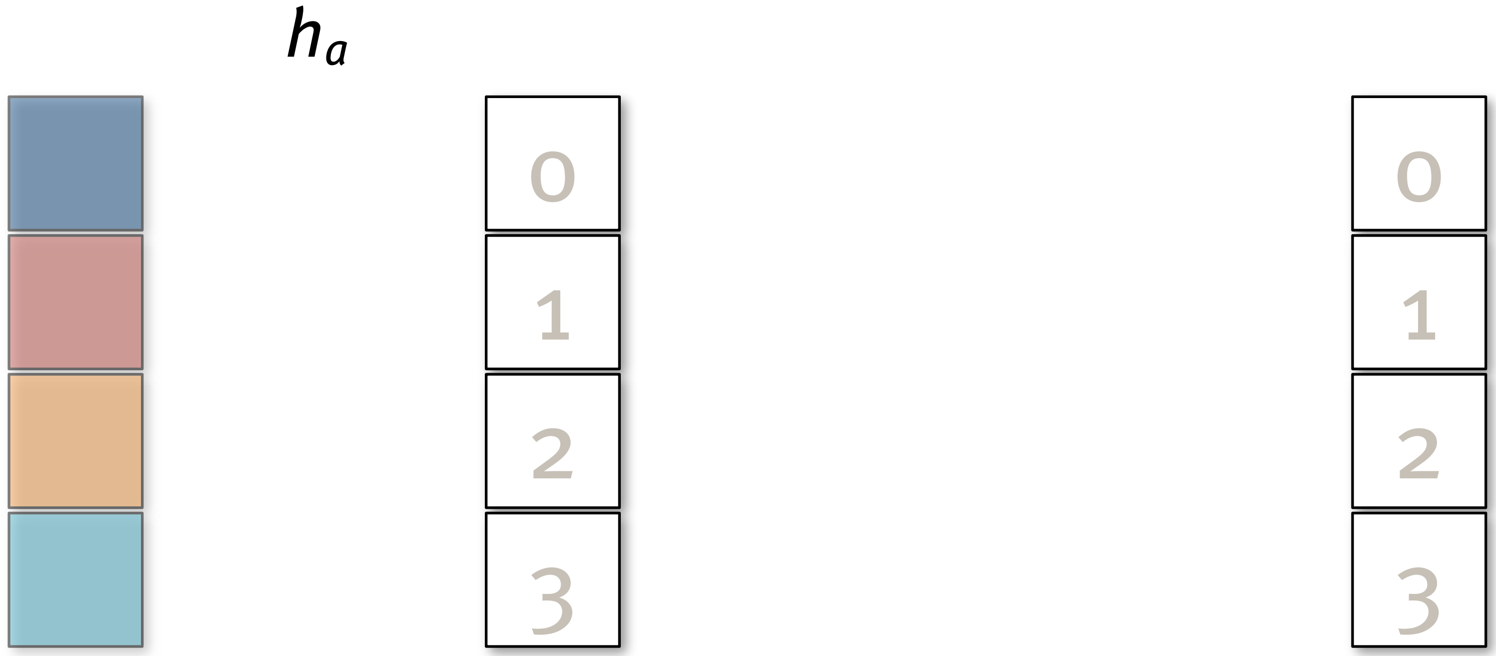
Parallel Hashing: Level 1

- 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 $O(n^2)$

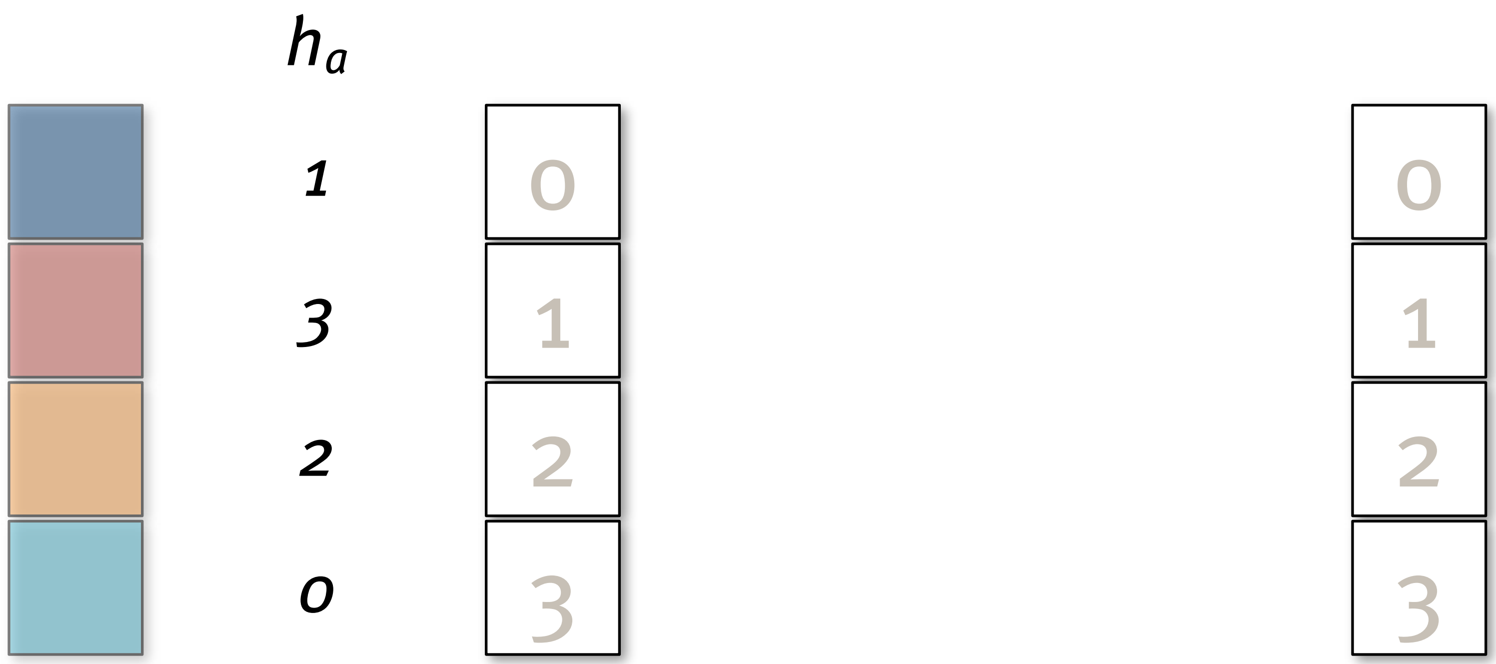
Hashing in Parallel



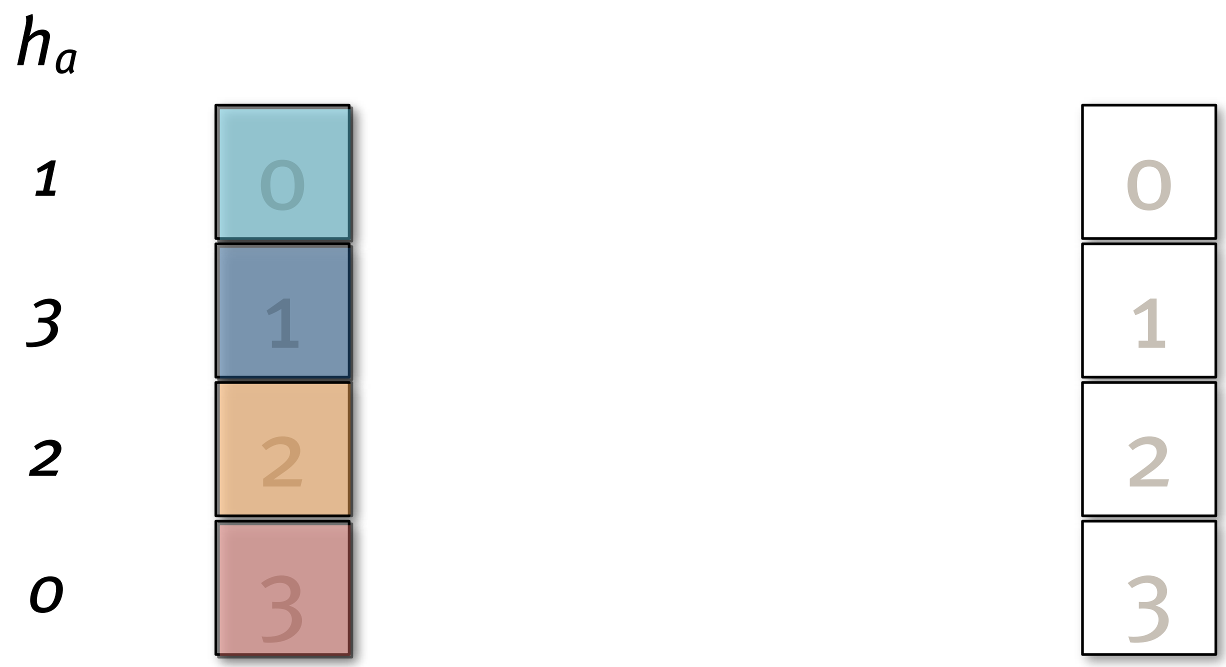
Hashing in Parallel



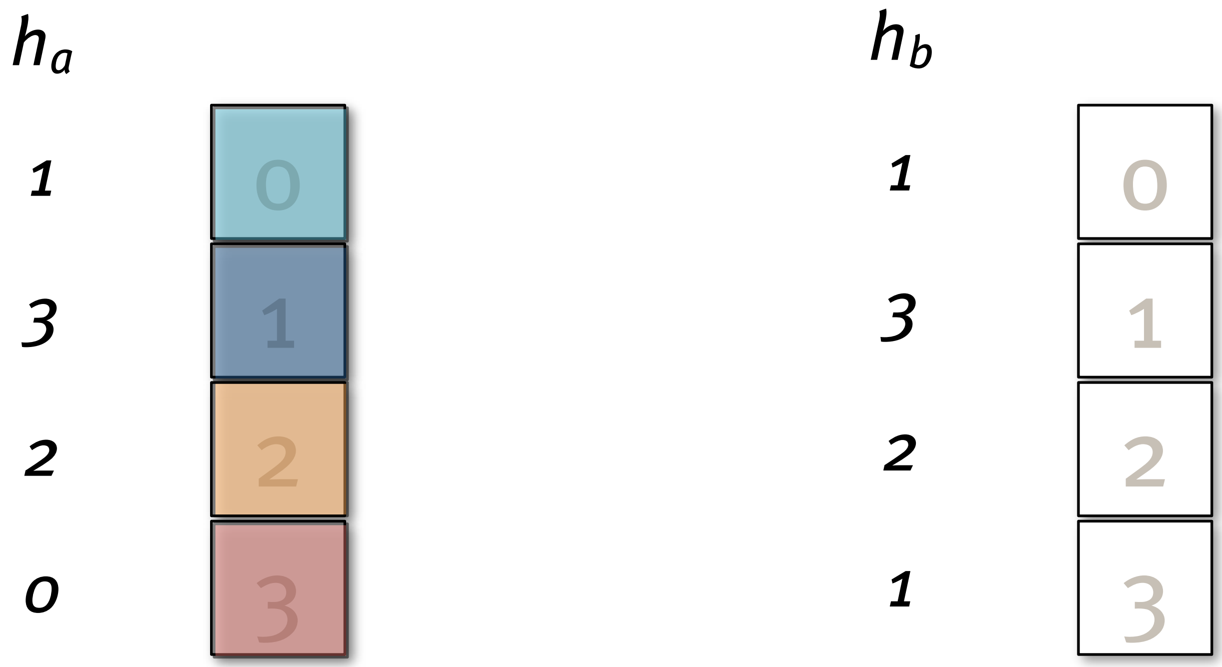
Hashing in Parallel



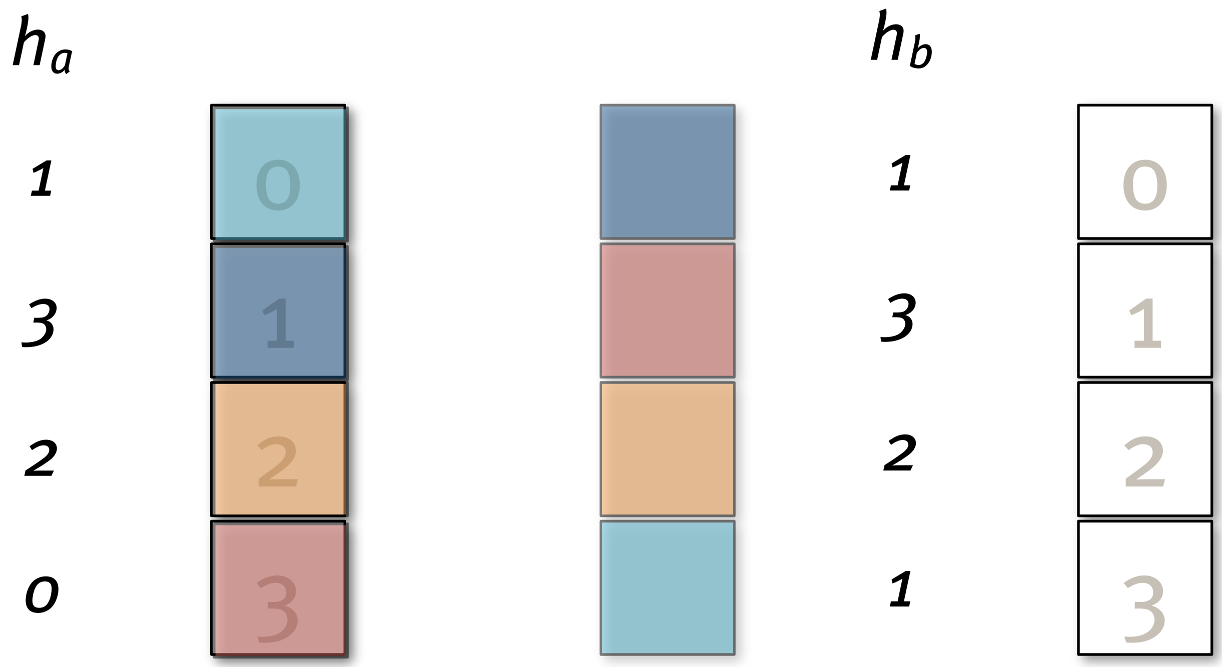
Hashing in Parallel



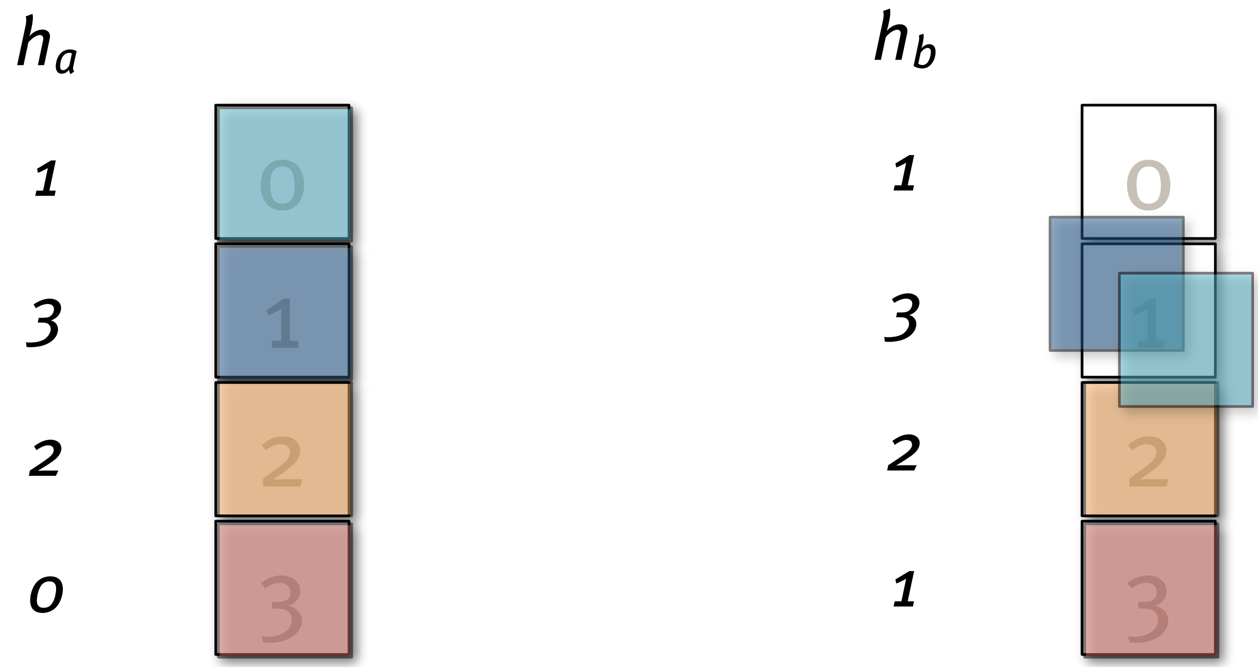
Hashing in Parallel



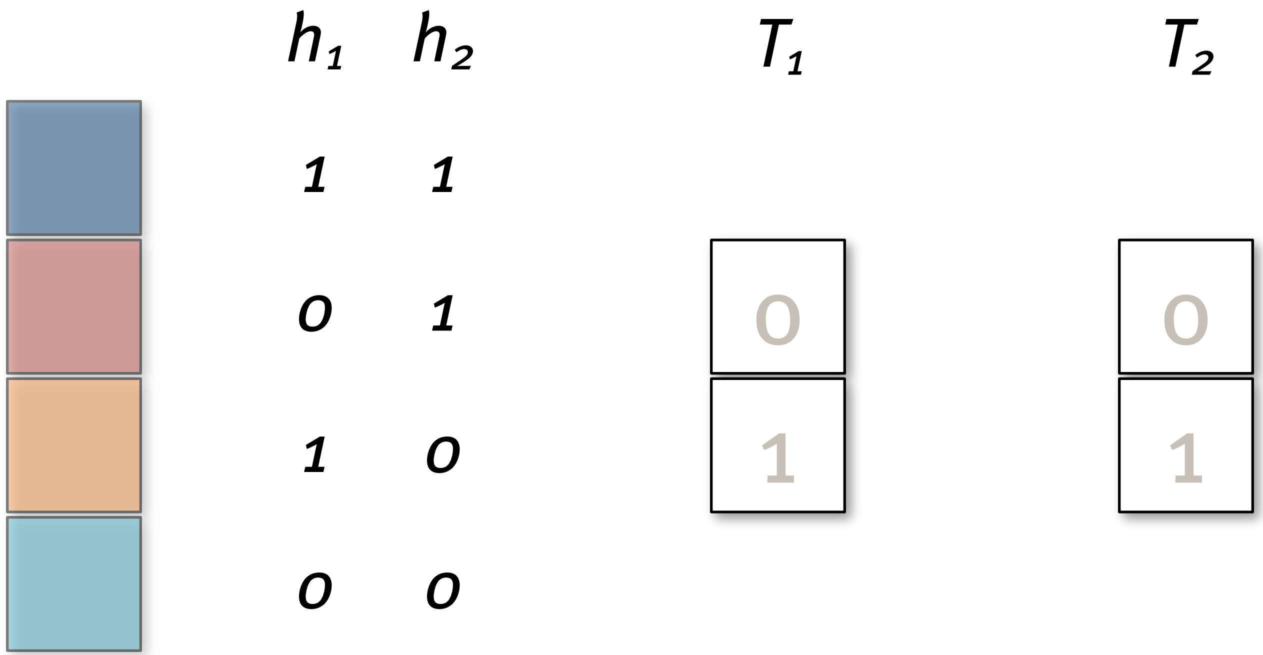
Hashing in Parallel



Hashing in Parallel

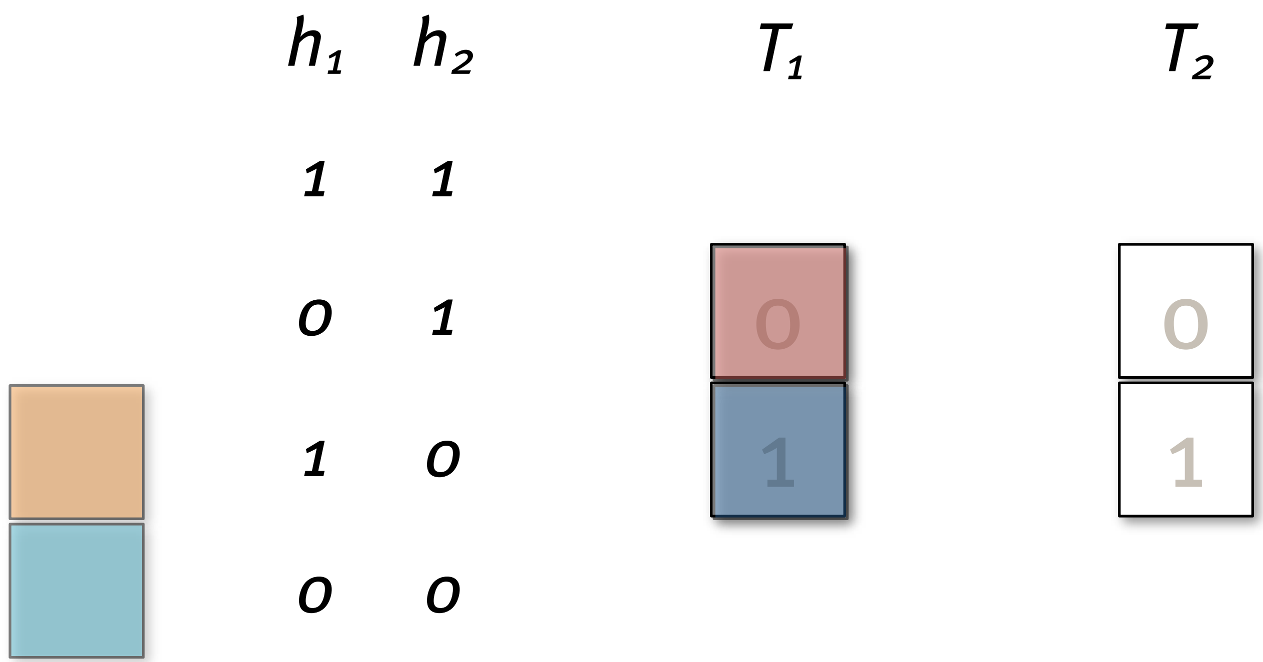


Cuckoo Hashing Construction



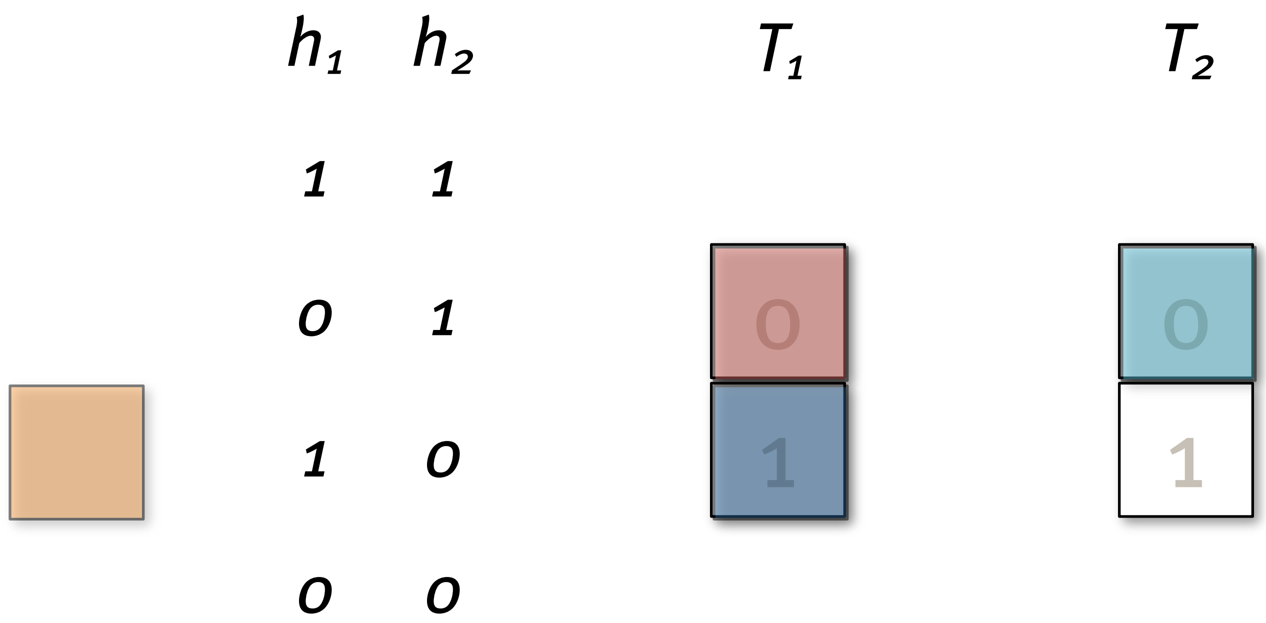
- Lookup procedure: in parallel, for each element:
  - Calculate h_1 & look in T_1;
  - Calculate h_2 & look in T_2; still $O(1)$ lookup

Cuckoo Hashing Construction



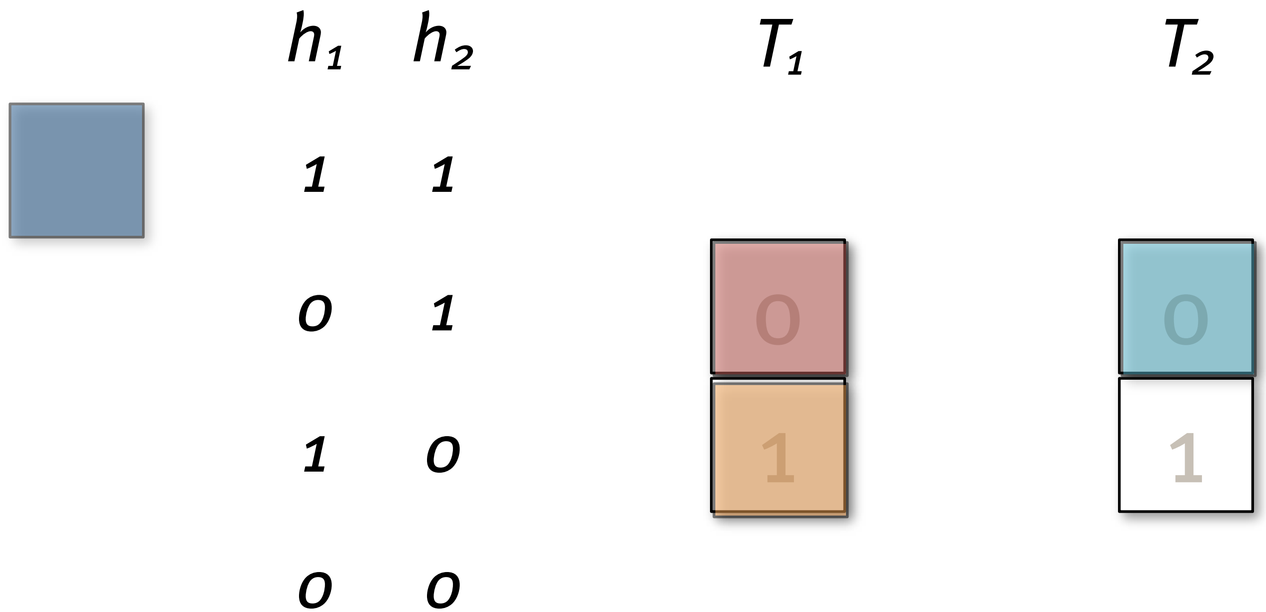
- Lookup procedure: in parallel, for each element:
    - Calculate h_1 & look in T_1;
    - Calculate h_2 & look in T_2; still $O(1)$ lookup

Cuckoo 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 Hashing Construction

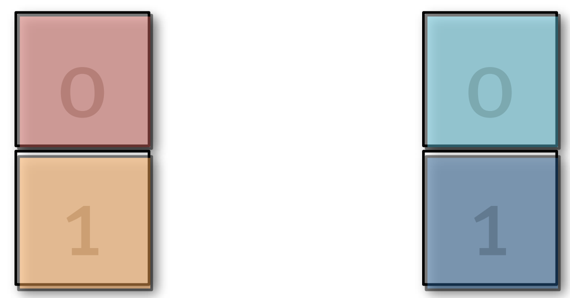


- Lookup procedure: in parallel, for each element:
  - Calculate h_1 & look in T_1;
  - Calculate h_2 & look in T_2; still $O(1)$ lookup

Cuckoo Hashing Construction

h_1	h_2
1	1
0	1
1	0
0	0

T_1 T_2

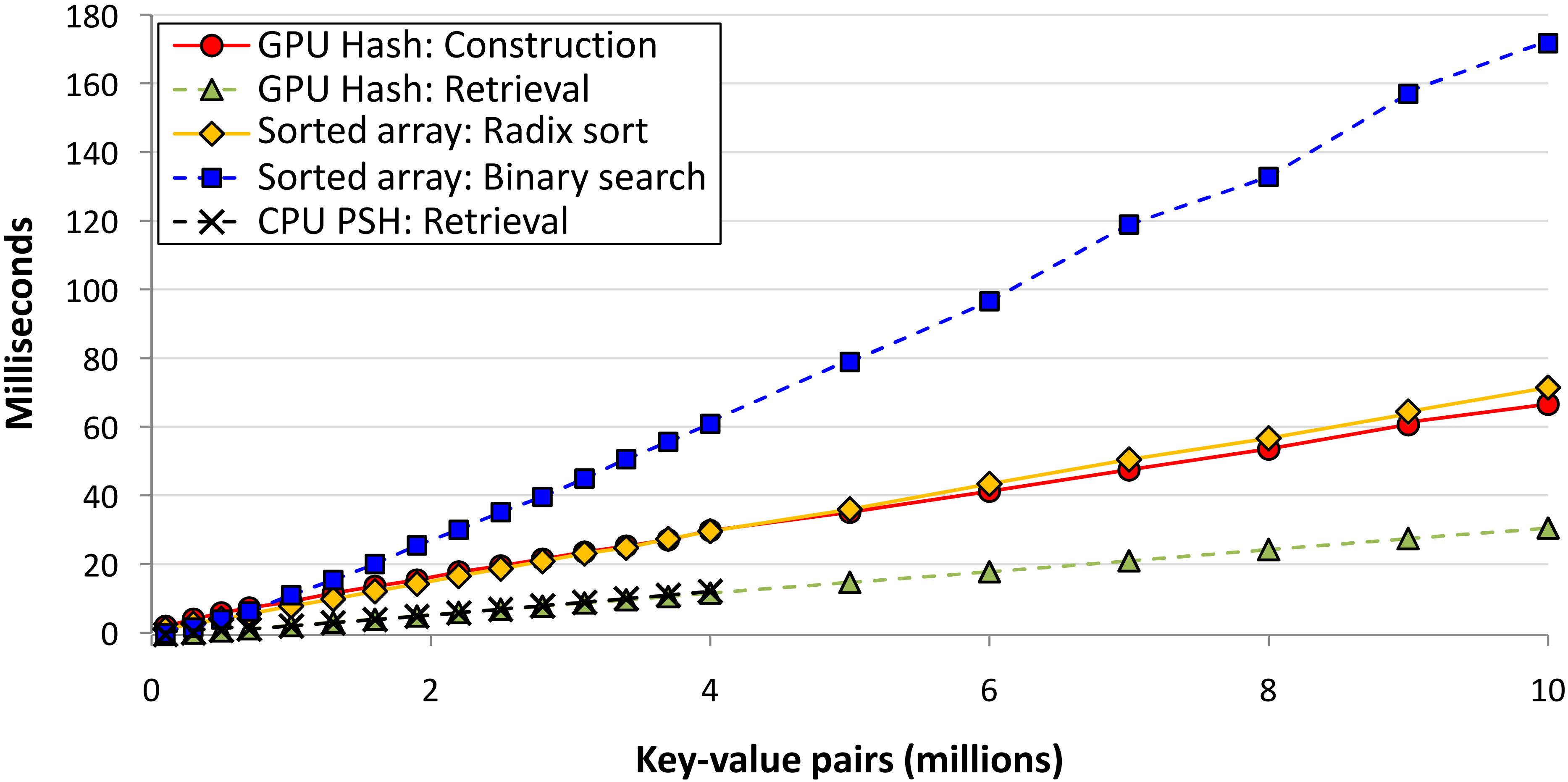


- Lookup procedure: in parallel, for each element:
  - Calculate h_1 & look in T_1;
  - Calculate h_2 & look in T_2; still $O(1)$ lookup

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

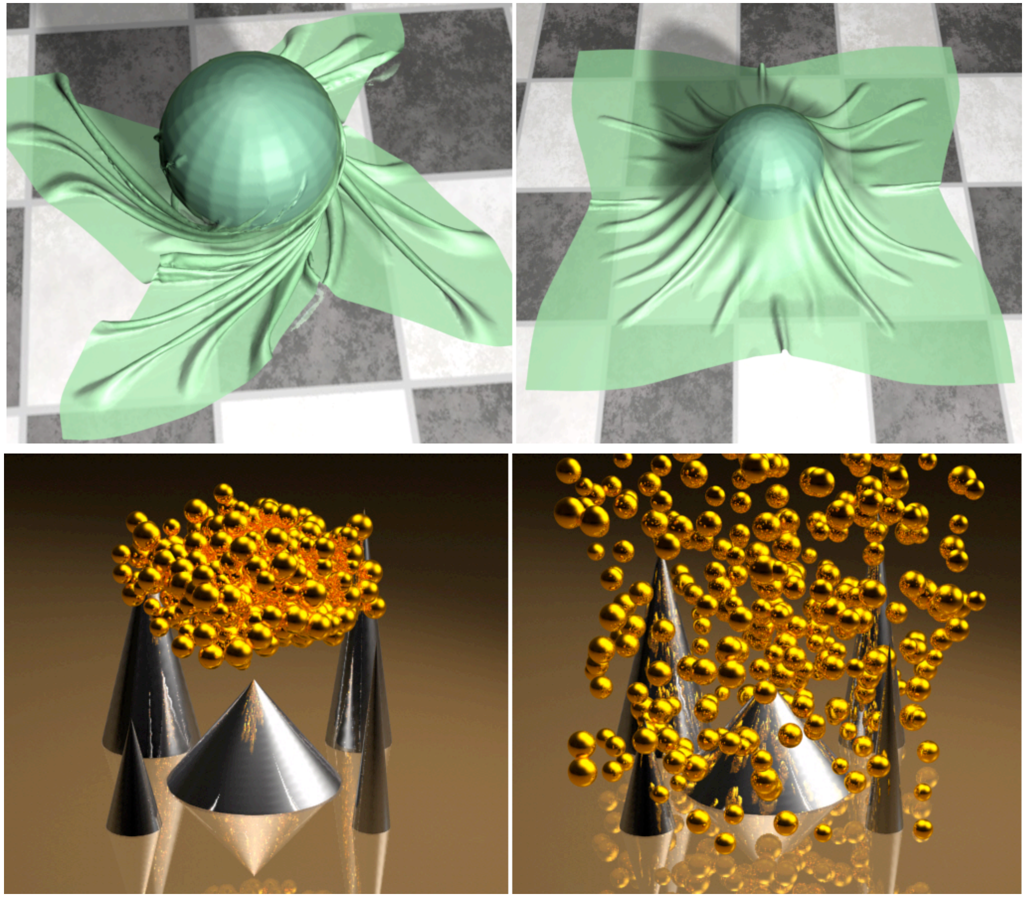


Hashing: Big Ideas

- 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

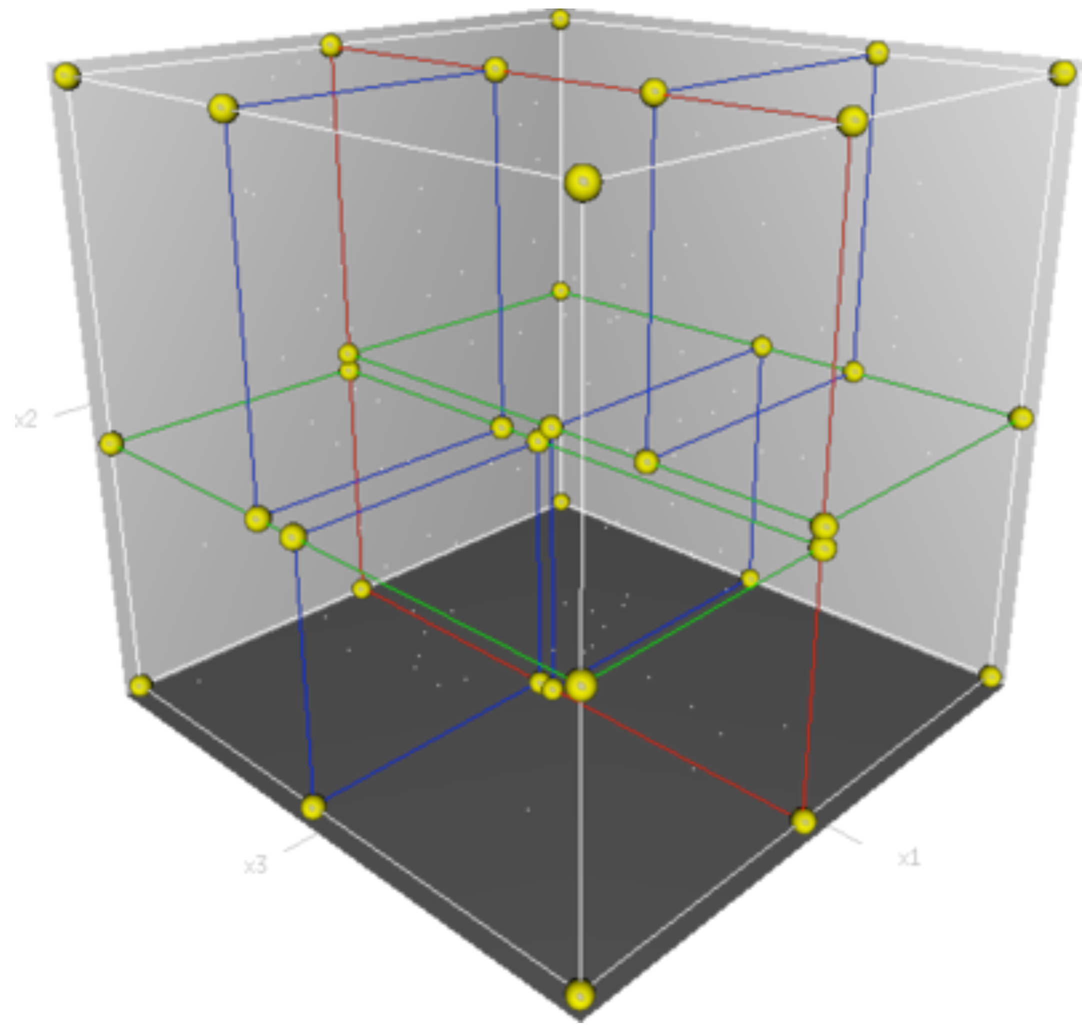
Trees: Motivation

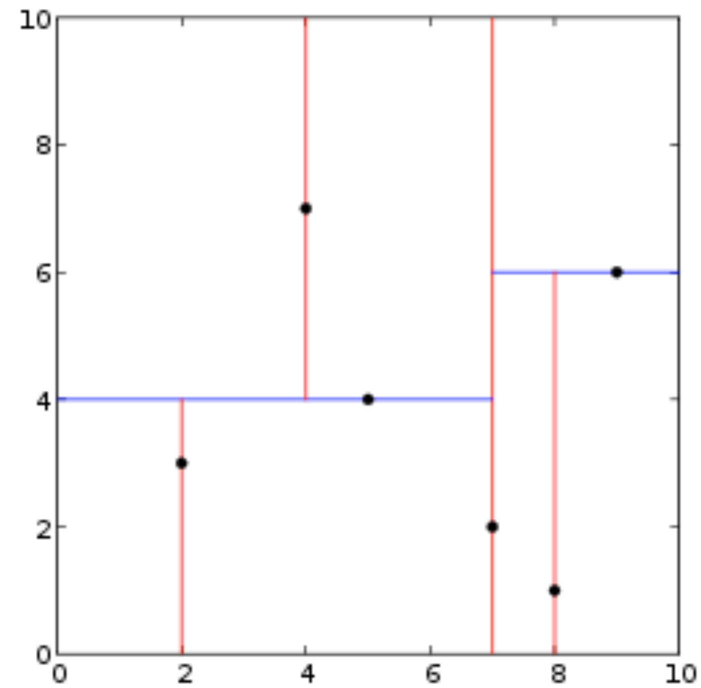
- Query: Does object X intersect with anything in the scene?
- Difficulty: X and the scene are dynamic
- Goal: Data structure that makes this query efficient (in parallel)



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

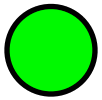
k-d trees





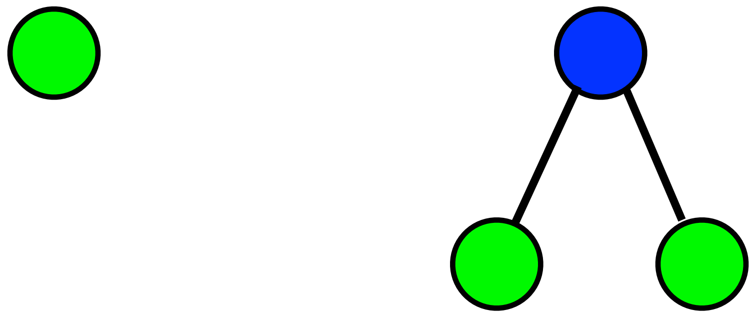
Images from Wikipedia, “Kd-tree”

Generating Trees



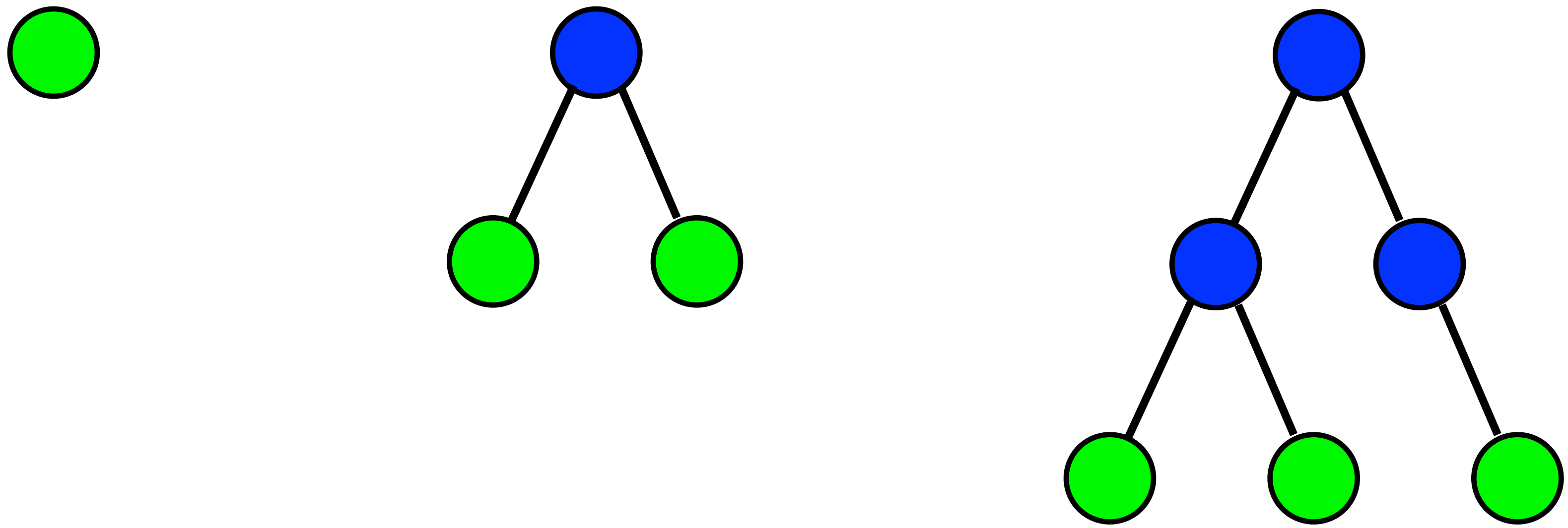
- Increased parallelism with depth
- Irregular work generation

Generating Trees



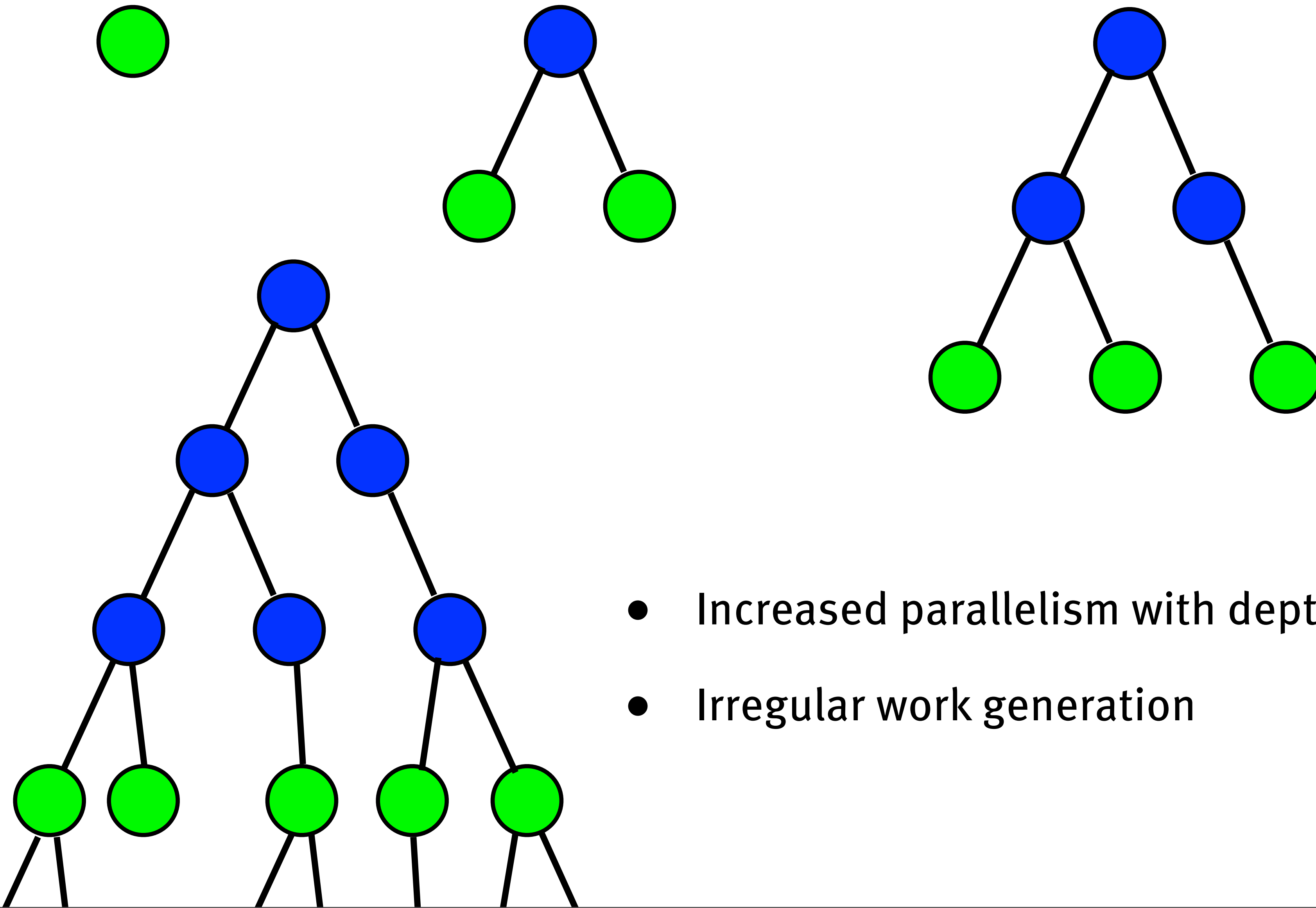
- Increased parallelism with depth
- Irregular work generation

Generating Trees



- Increased parallelism with depth
- Irregular work generation

Generating Trees

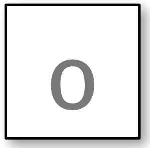


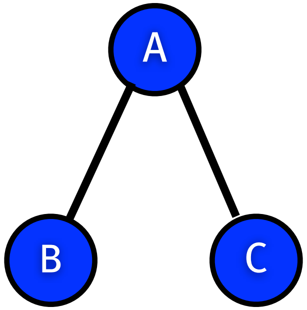
- Increased parallelism with depth
- Irregular work generation

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?

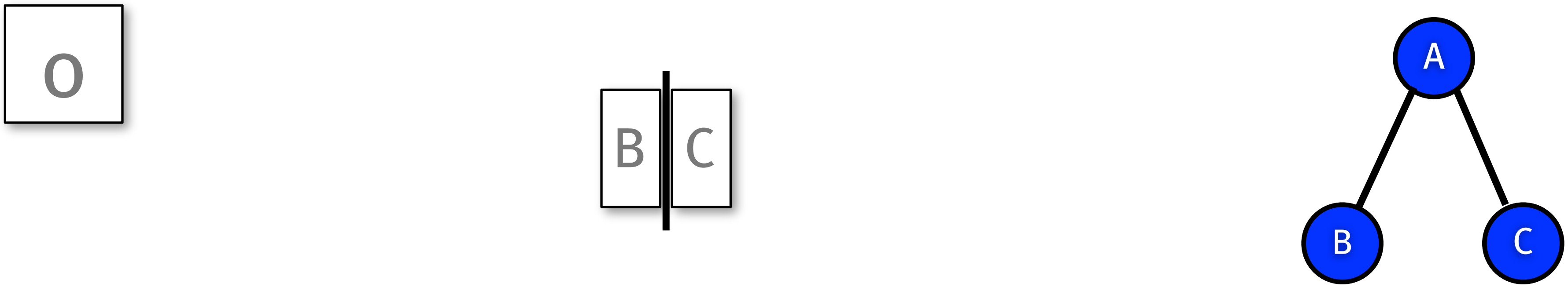
Tree Construction on a GPU





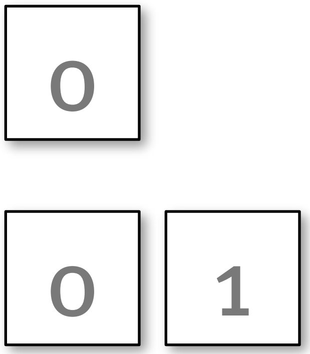
- At each stage, any node can generate 0, 1, or 2 new nodes
- Increased parallelism, but some threads wasted
- Compact after each step?

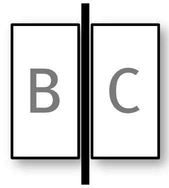
Tree Construction on a GPU

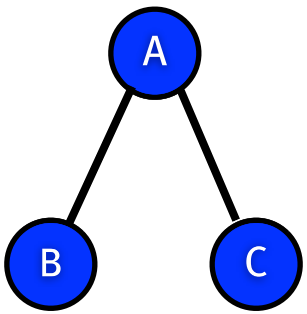


- At each stage, any node can generate 0, 1, or 2 new nodes
- Increased parallelism, but some threads wasted
- Compact after each step?

Tree Construction on a GPU

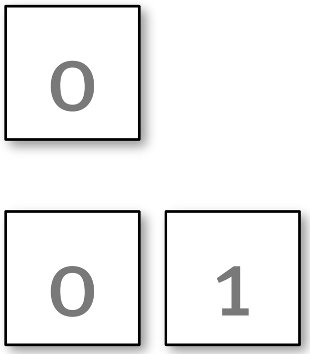


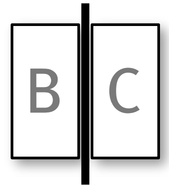


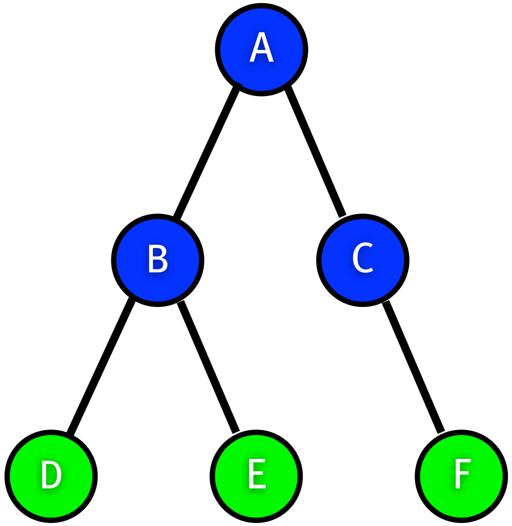


- At each stage, any node can generate 0, 1, or 2 new nodes
- Increased parallelism, but some threads wasted
- Compact after each step?

Tree Construction on a GPU

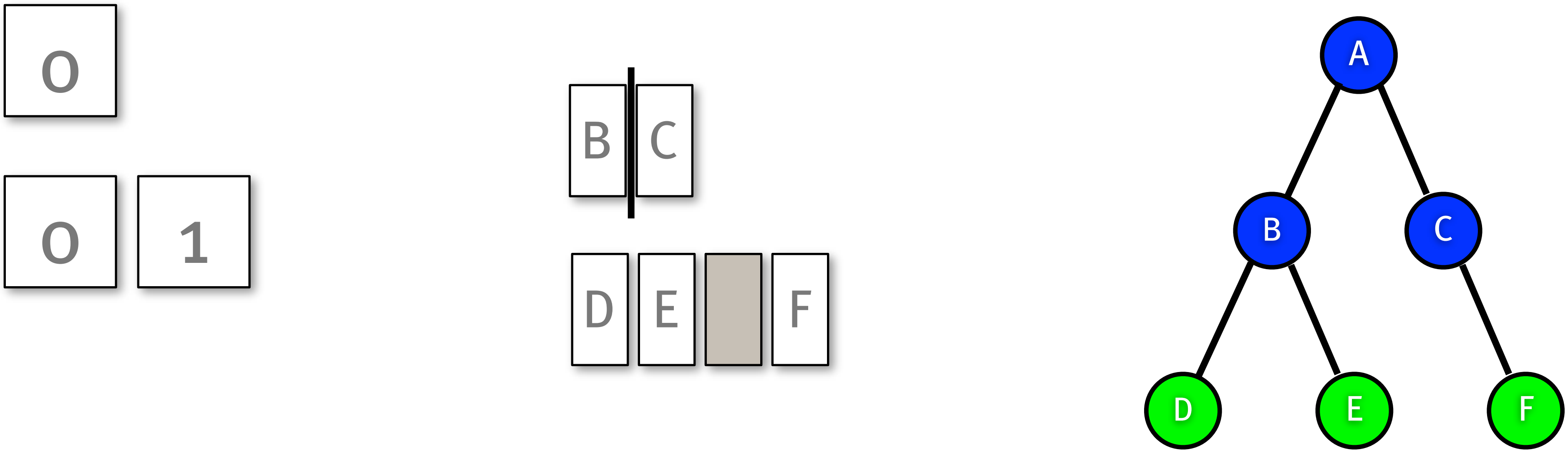






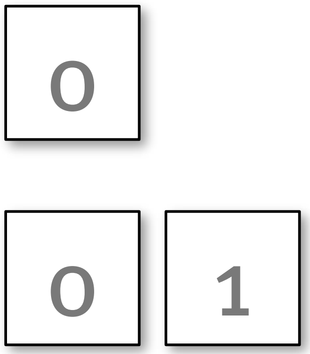
- At each stage, any node can generate 0, 1, or 2 new nodes
- Increased parallelism, but some threads wasted
- Compact after each step?

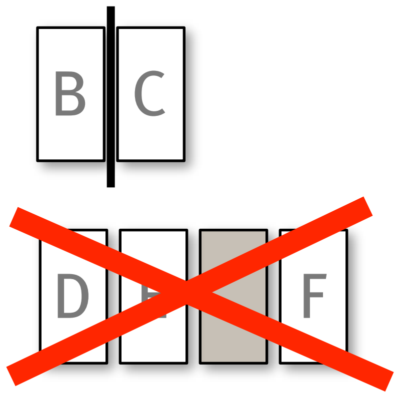
Tree Construction on a GPU

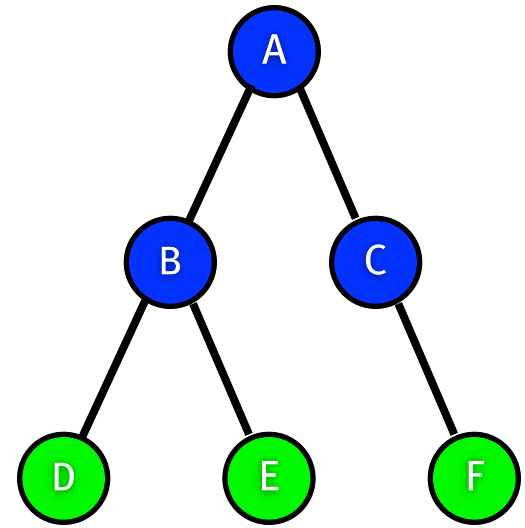


- At each stage, any node can generate 0, 1, or 2 new nodes
- Increased parallelism, but some threads wasted
- Compact after each step?

Tree Construction on a GPU

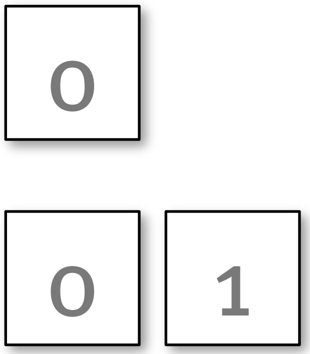


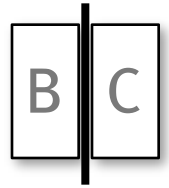


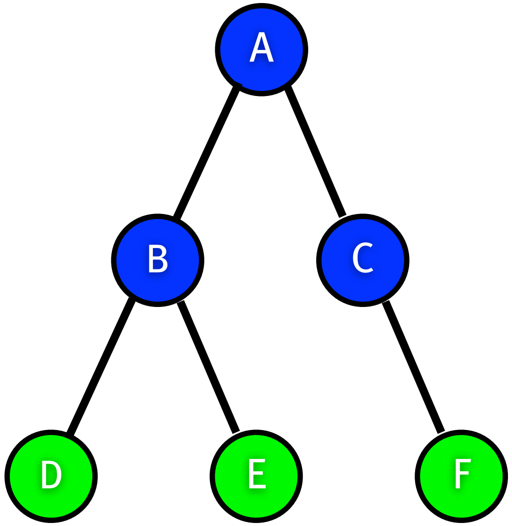


- At each stage, any node can generate 0, 1, or 2 new nodes
- Increased parallelism, but some threads wasted
- Compact after each step?

Tree Construction on a GPU

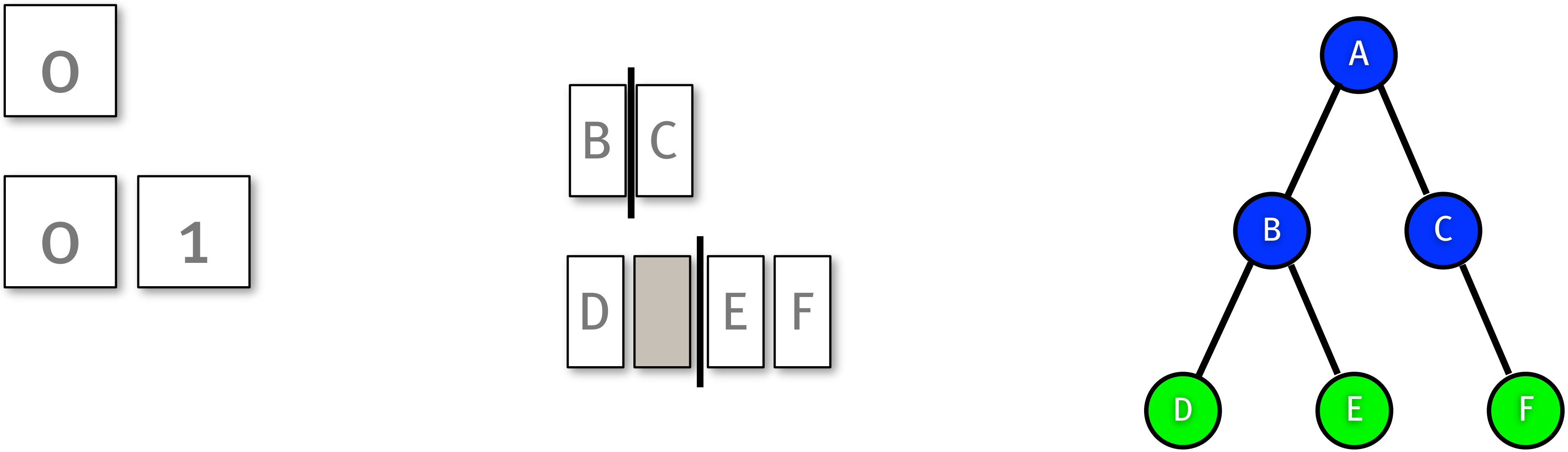






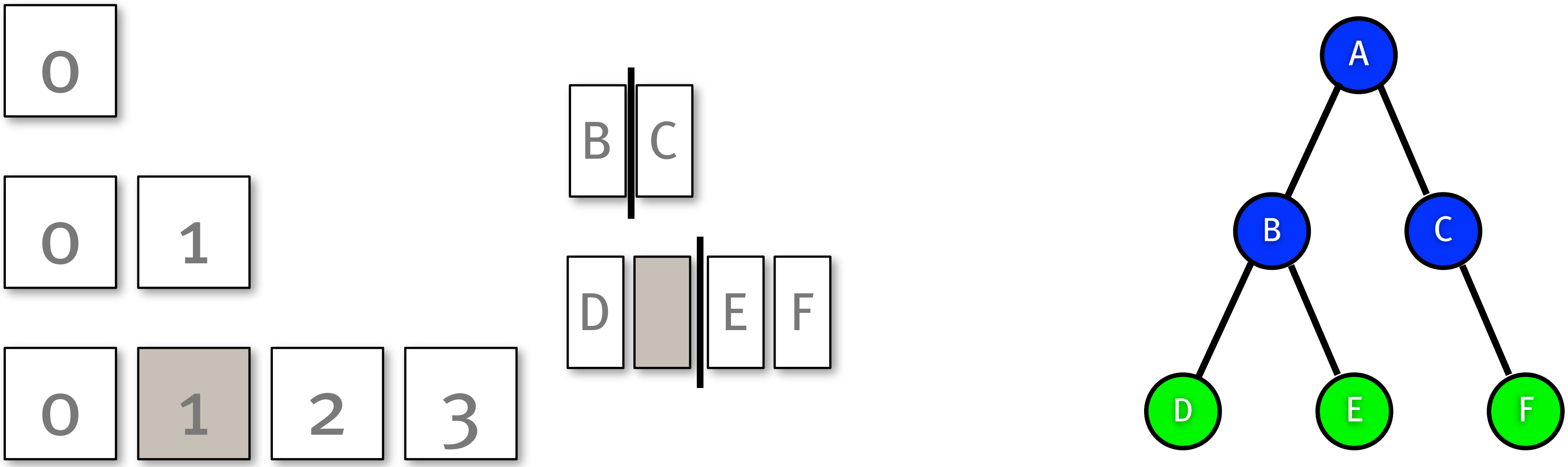
- At each stage, any node can generate 0, 1, or 2 new nodes
- Increased parallelism, but some threads wasted
- Compact after each step?

Tree Construction on a GPU



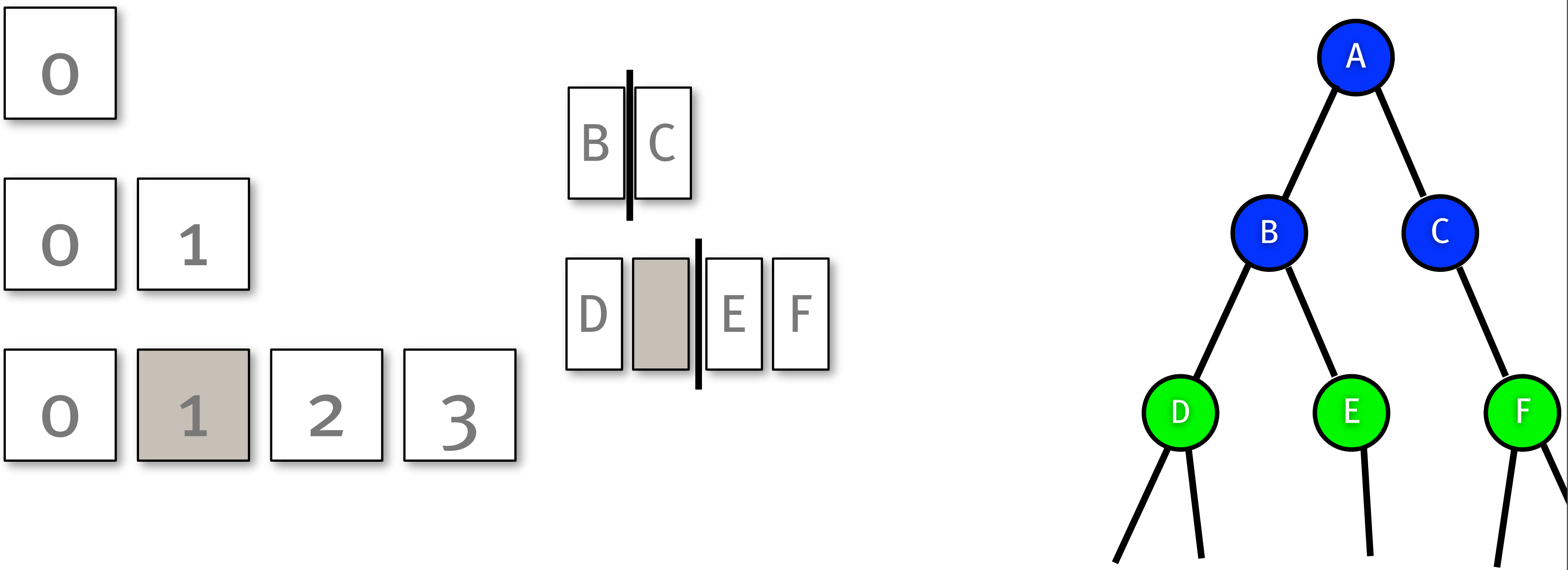
- At each stage, any node can generate 0, 1, or 2 new nodes
- Increased parallelism, but some threads wasted
- Compact after each step?

Tree Construction on a GPU



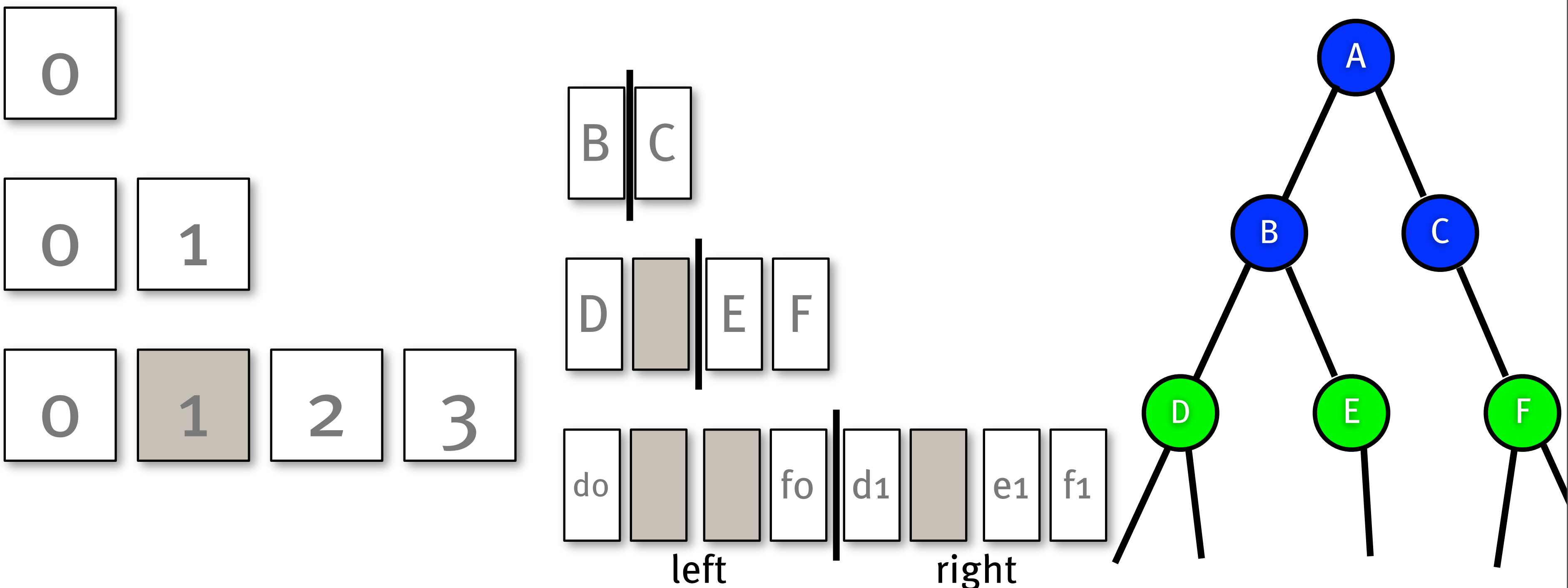
- At each stage, any node can generate 0, 1, or 2 new nodes
- Increased parallelism, but some threads wasted
- Compact after each step?

Tree Construction on a GPU



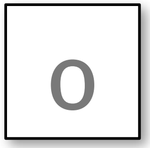
- At each stage, any node can generate 0, 1, or 2 new nodes
- Increased parallelism, but some threads wasted
- Compact after each step?

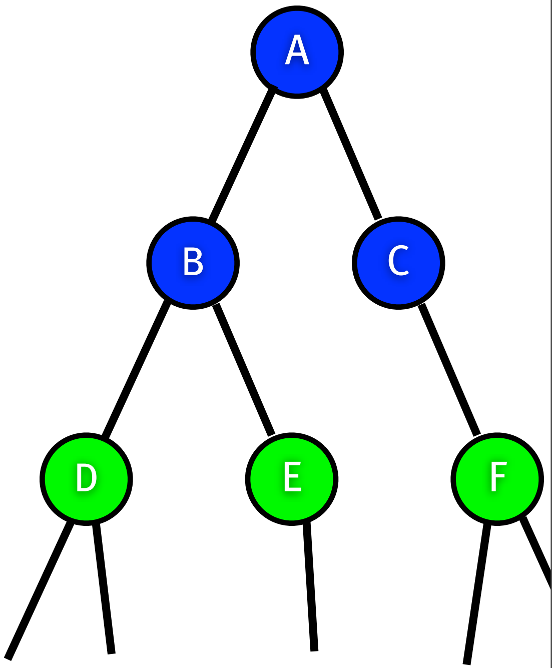
Tree Construction on a GPU



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

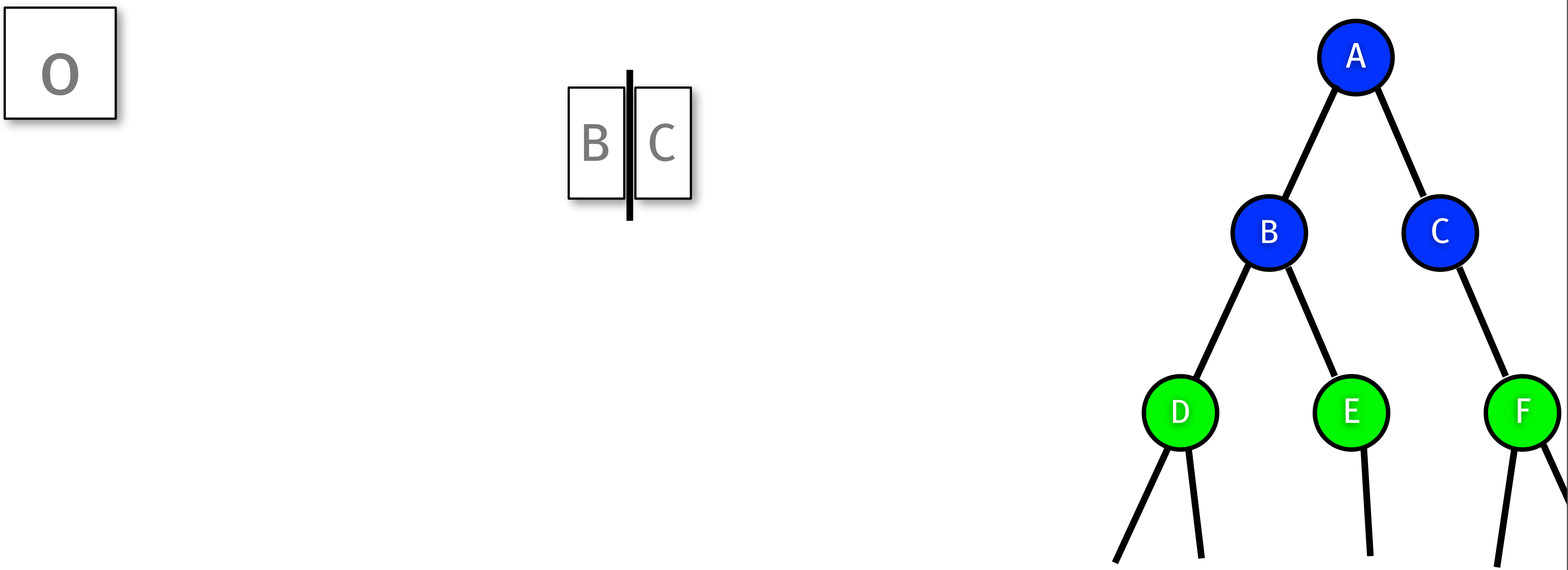
Tree Construction on a GPU





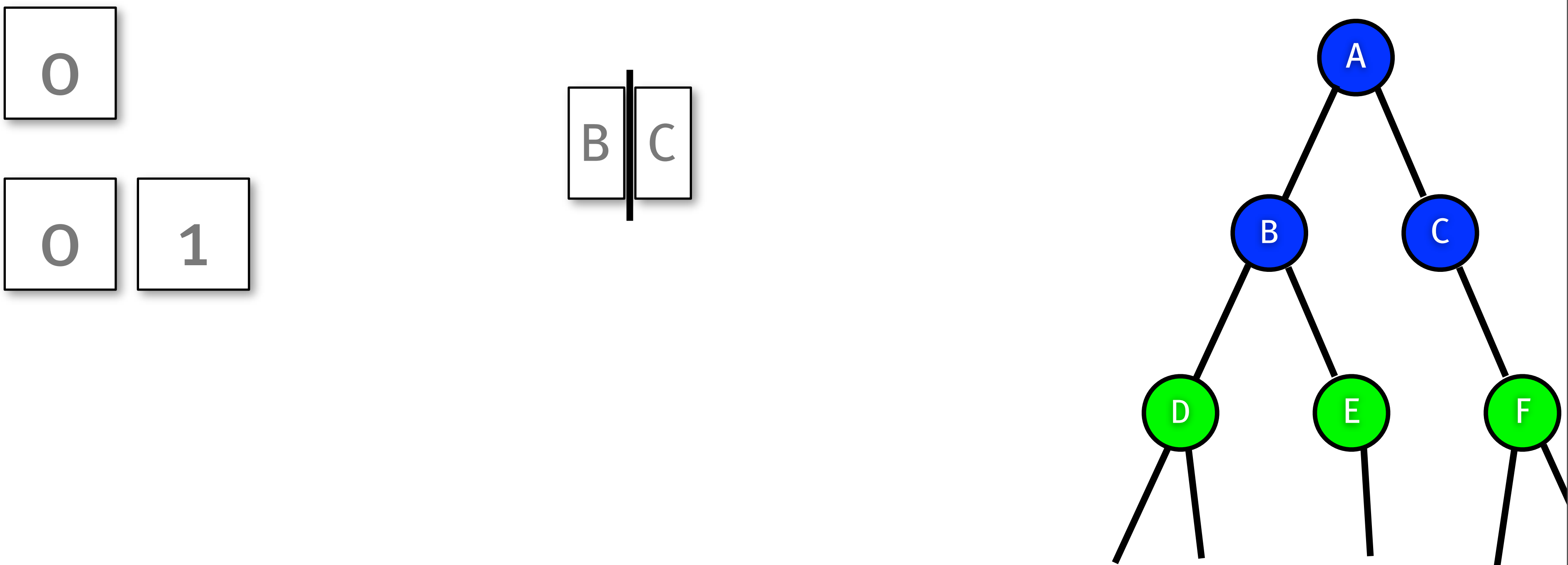
- Compact reduces overwork, but ...
- ... requires global compact operation per step
- Also requires worst-case storage allocation

Tree Construction on a GPU



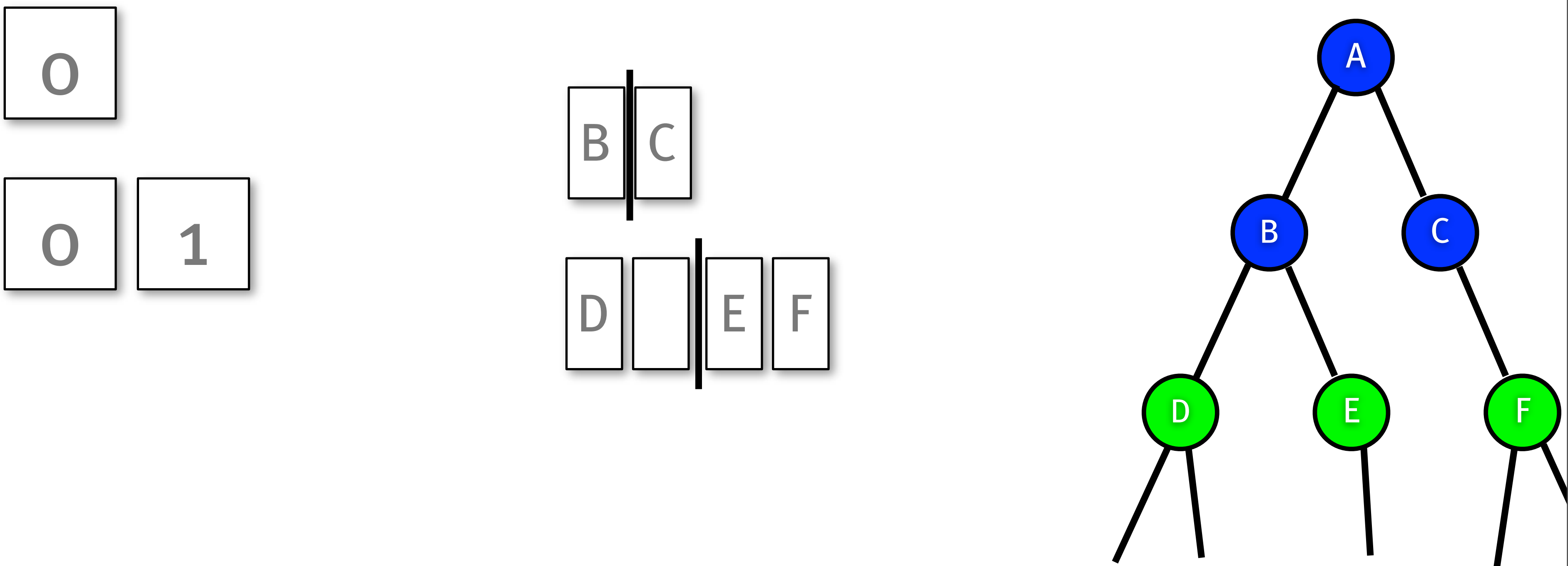
- Compact reduces overwork, but ...
- ... requires global compact operation per step
- Also requires worst-case storage allocation

Tree Construction on a GPU



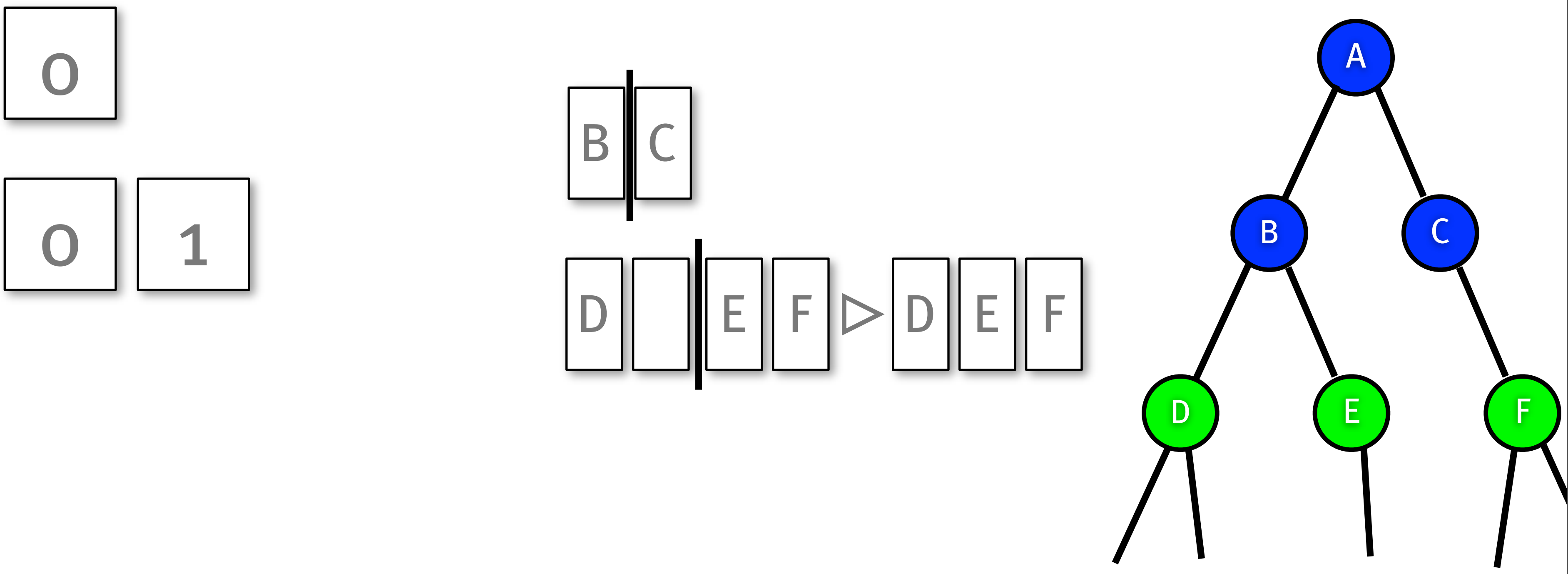
- Compact reduces overwork, but ...
- ... requires global compact operation per step
- Also requires worst-case storage allocation

Tree Construction on a GPU



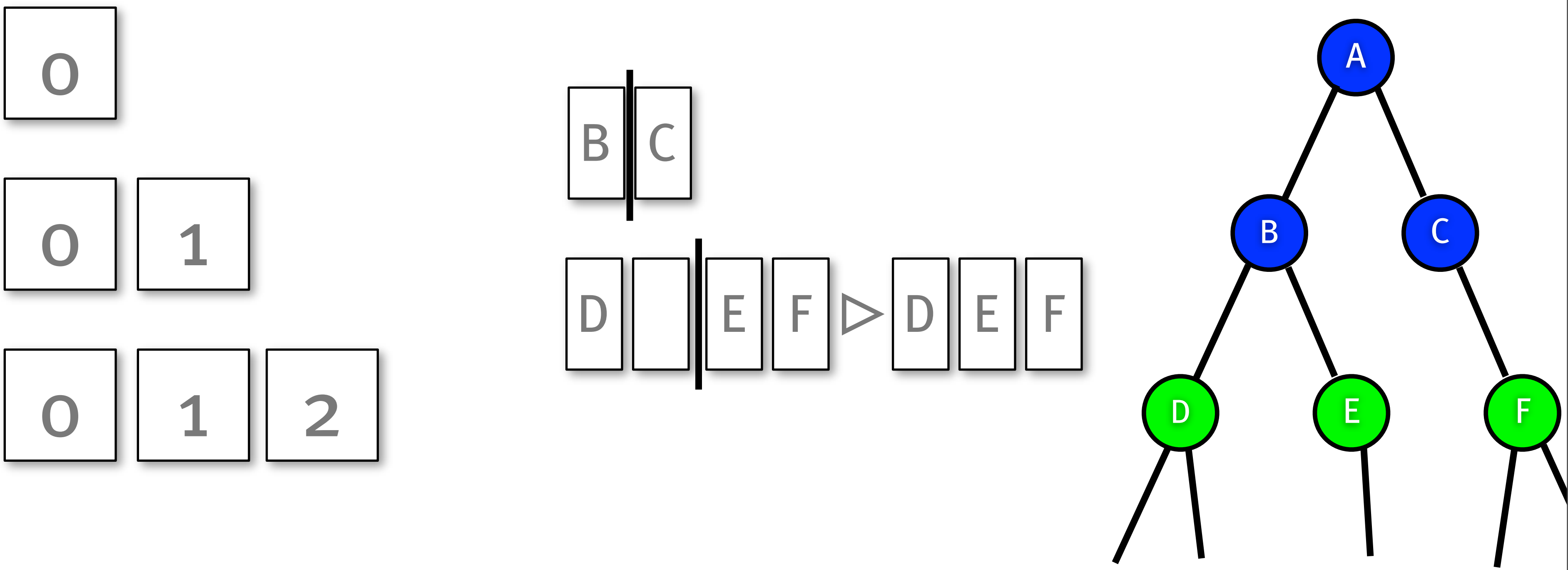
- Compact reduces overwork, but ...
- ... requires global compact operation per step
- Also requires worst-case storage allocation

Tree Construction on a GPU



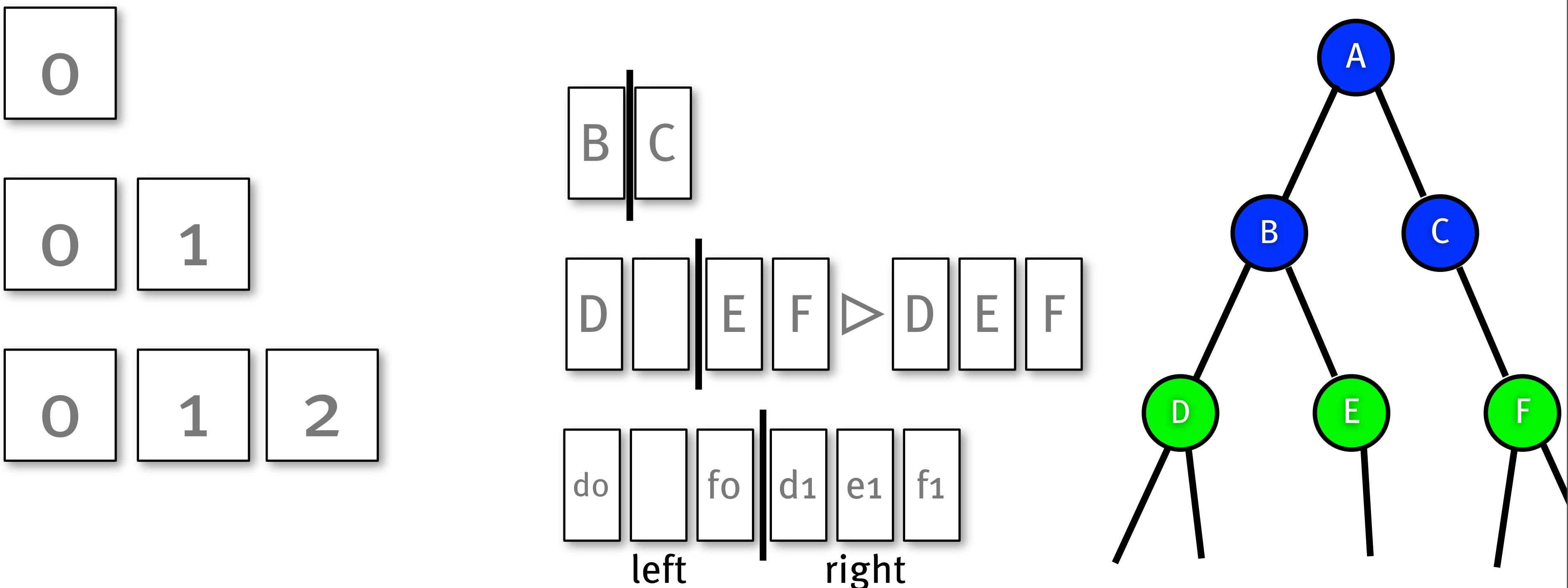
- Compact reduces overwork, but ...
- ... requires global compact operation per step
- Also requires worst-case storage allocation

Tree Construction on a GPU



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

Tree Construction on a GPU



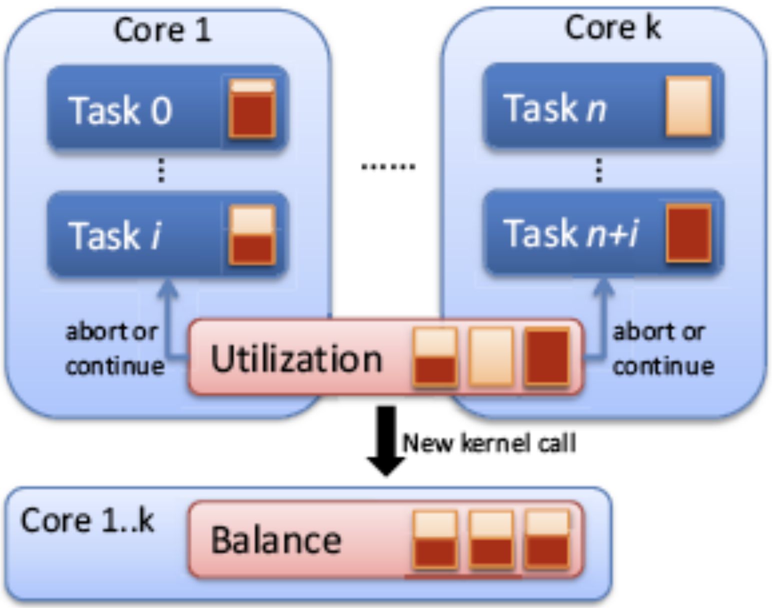
- Compact reduces overwork, but …
- … requires global compact operation per step
- Also requires worst-case storage allocation

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

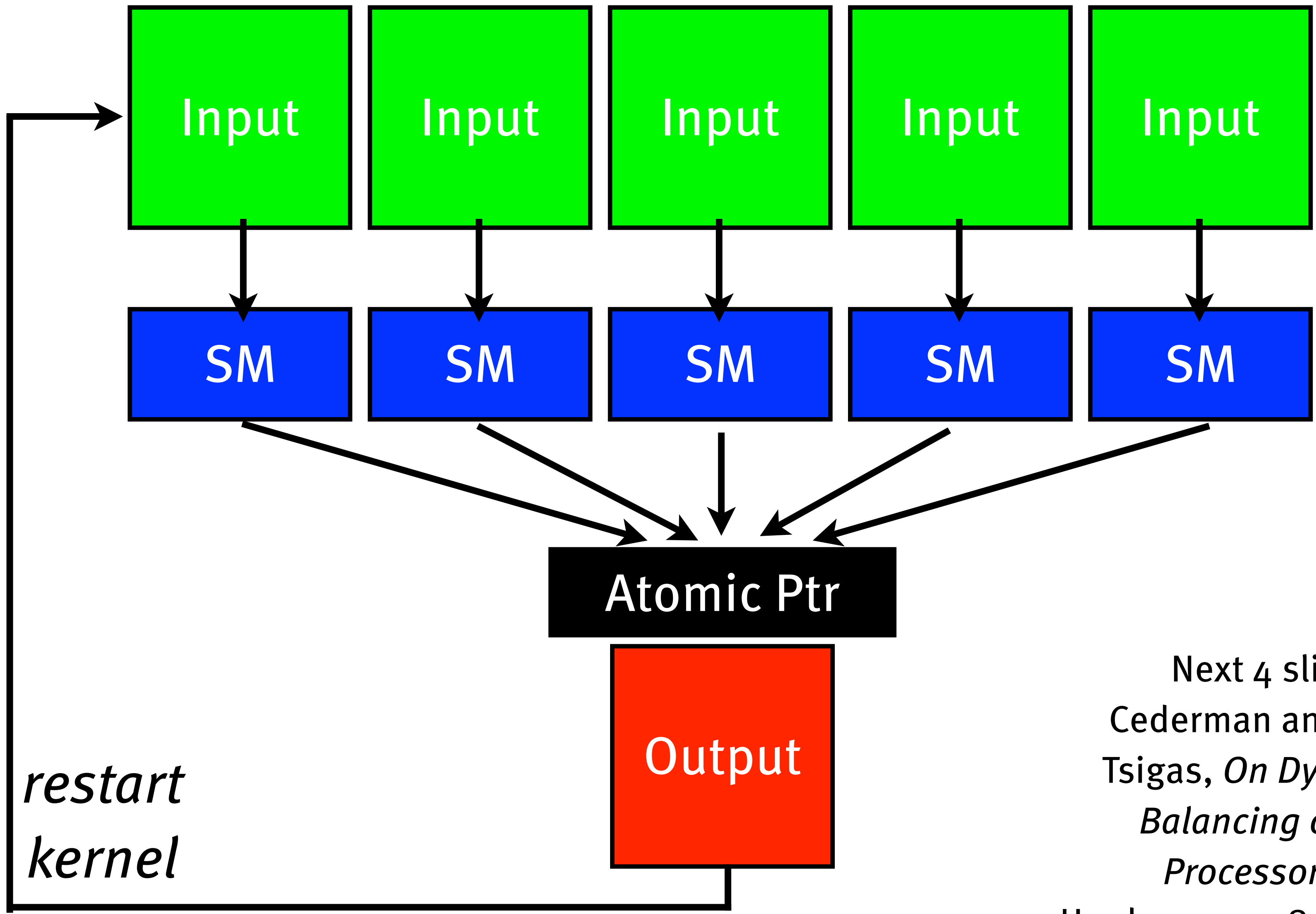
Work Queue Approach

- 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



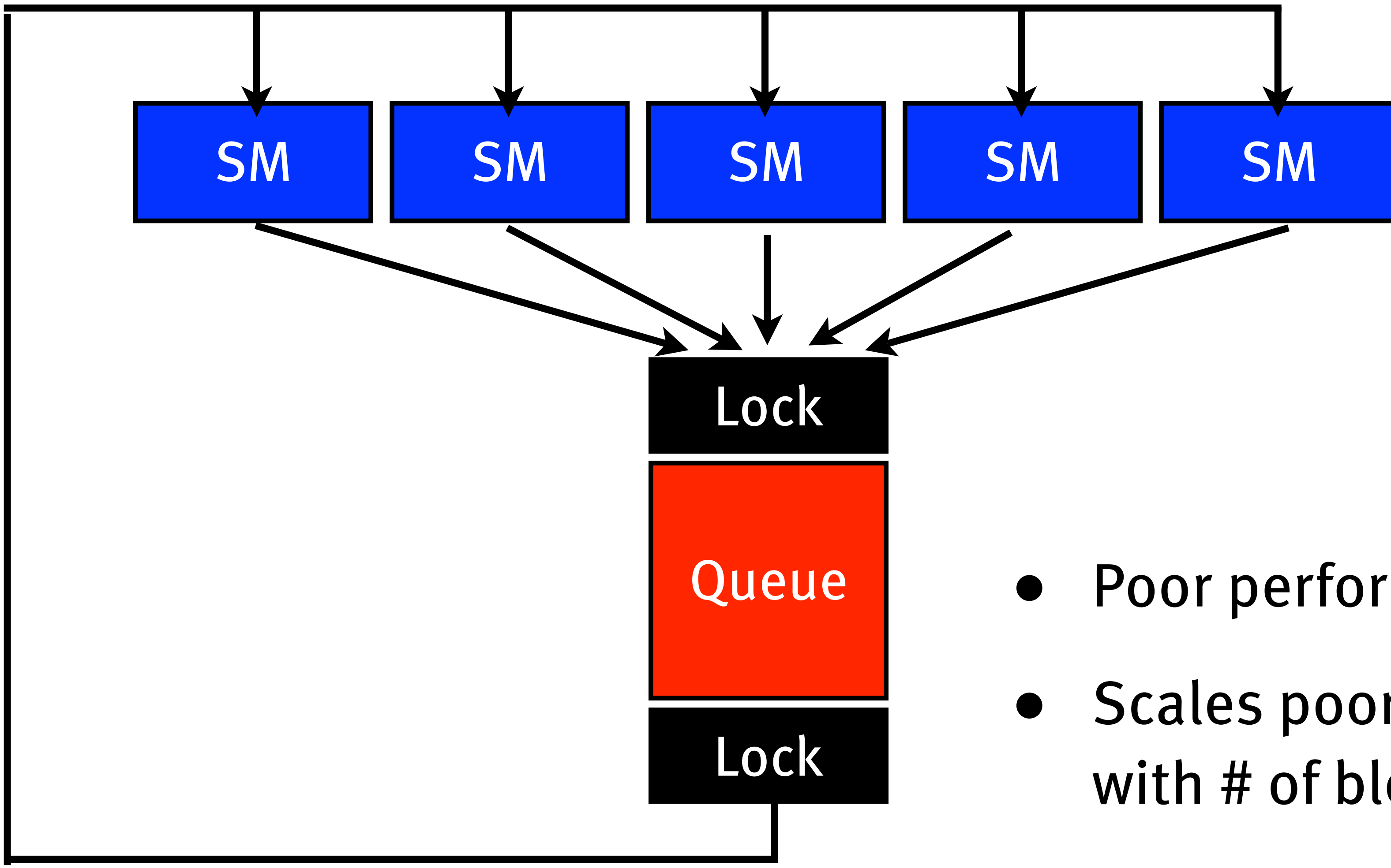
Fast Hierarchy Operations on GPU Architectures, Lauterbach et al.

Static Task List



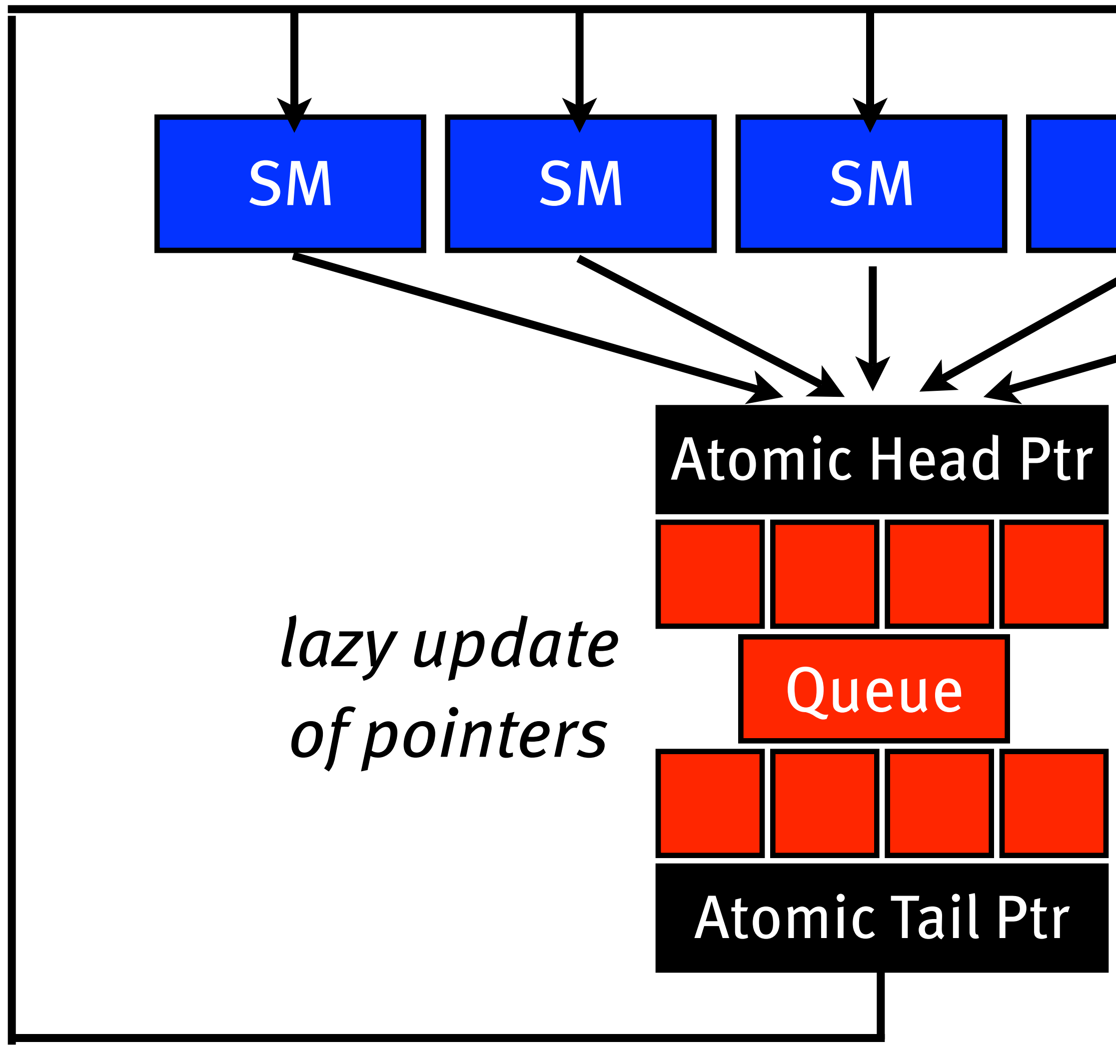
Next 4 slides: Daniel Cederman and Philippas Tsigas, *On Dynamic Load Balancing on Graphics Processors*. Graphics Hardware 2008, June 2008.

Blocking Dynamic Task Queue



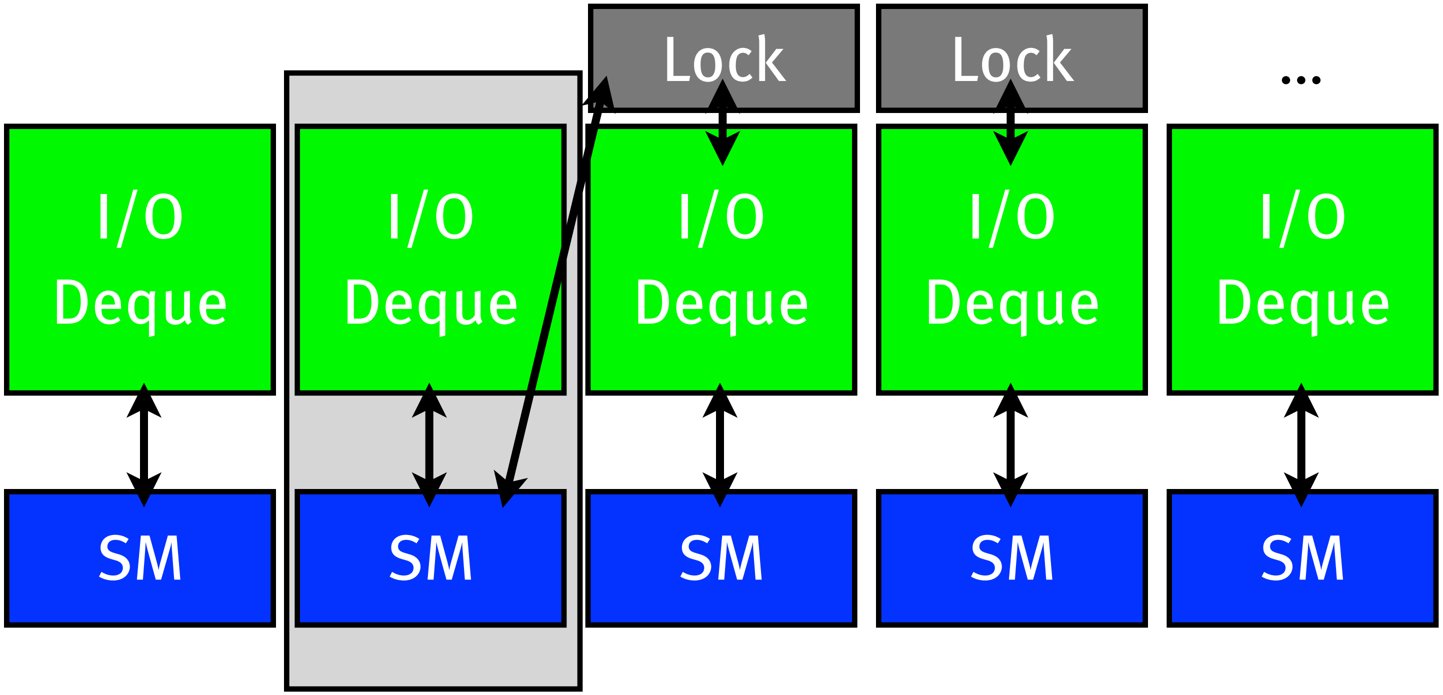
- Poor performance
- Scales poorly with # of blocks

Non-Blocking Dynamic Task Queue



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

Work Stealing



- Best performance and scalability

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

Thanks to …

- Nathan Bell, Michael Garland, David Luebke, 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.

- 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.