Dynamic Data Structures on the GPU

John Owens

Child Family Professor of Engineering and Entrepreneurship

University of California, Davis

Student principal authors: Saman Ashkiani, Muhammad Awad, Afton Geil

Joint work with Martin Farach-Colton (Rutgers, co-PI), Nina Amenta (UC Davis), Rob Johnson (VMWare Research)

Why data structures?

- Data structures organize sparsity
- Part of any programmer's toolbox!

```
#include <map>
#include <list>
#include <set>
```

And they just work!

What do we want to do?

- Queries (do not modify data structure)
  - Membership (is this key in the data set?)
    - Probabilistic membership (is this key probably in the dataset?)
  - Search (given key, return value associated with key)
    - SearchAll (given key, return all values associated with key)
- Successor (given key, what's the next key?)
- Range (given key range, return all keys / all values within that range)

What do we want to do?

- Change (mutate) data structure
  - Insert (add new key or key-value pair)
  - Replace (insert, but replaces old value with new value)
  - Delete (remove key and associated value(s))

- We can always just rebuild the data structure from scratch, but:

- Goal: Cost is *O*(update size) instead of *O*(dataset size)

What do we want to do?

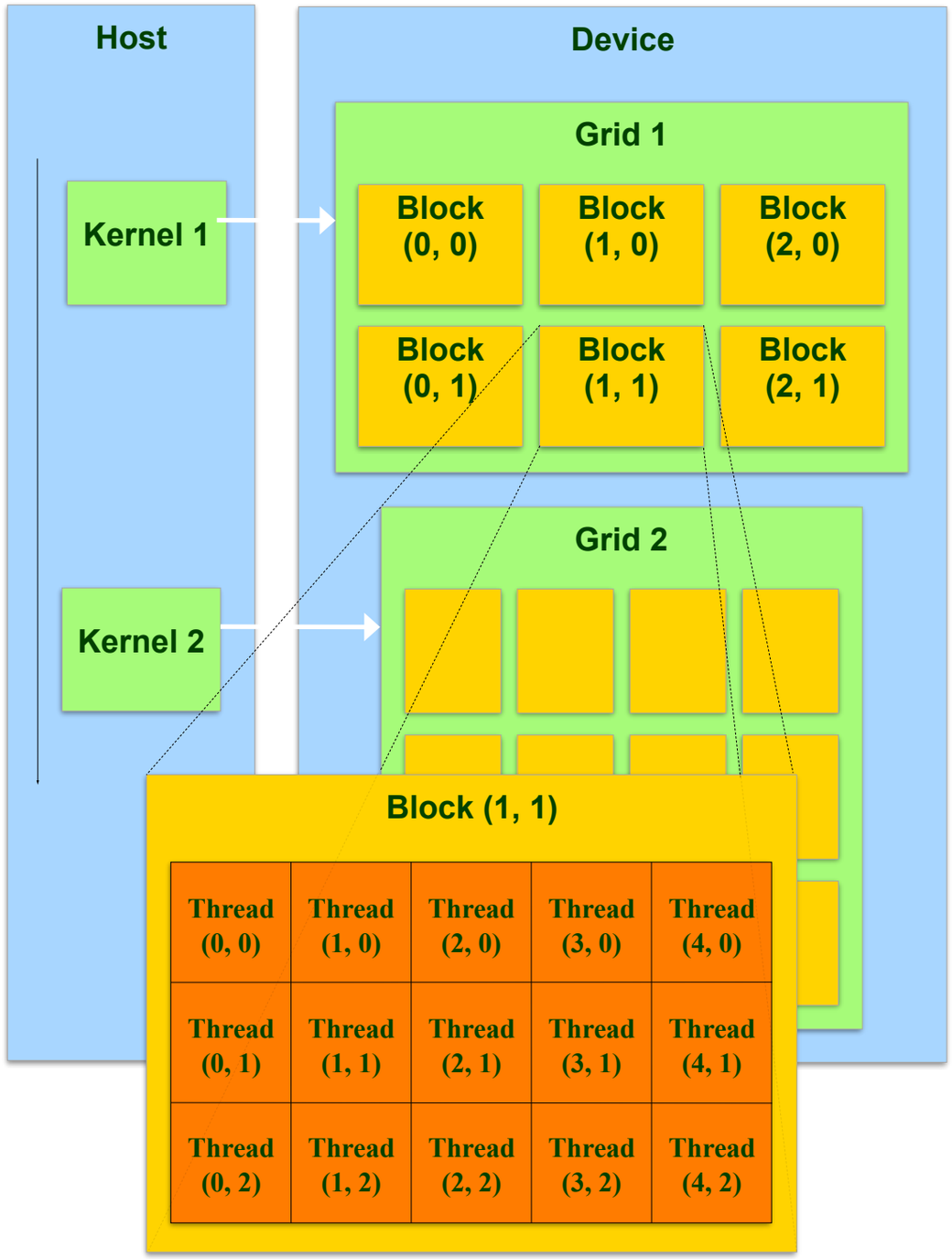
- **Build data structure**
  - "Bulk build" (build complete data structure from large amount of data)
  - Incremental build (e.g., data is streaming in and we want to build the data structure as we go)
    - What are the use cases for incremental build?
      - "Add and age": as data streams in, add new data & delete data older than a certain threshold [DARPA]
      - Use cases for this are not yet clear (more later)

Why is this hard?

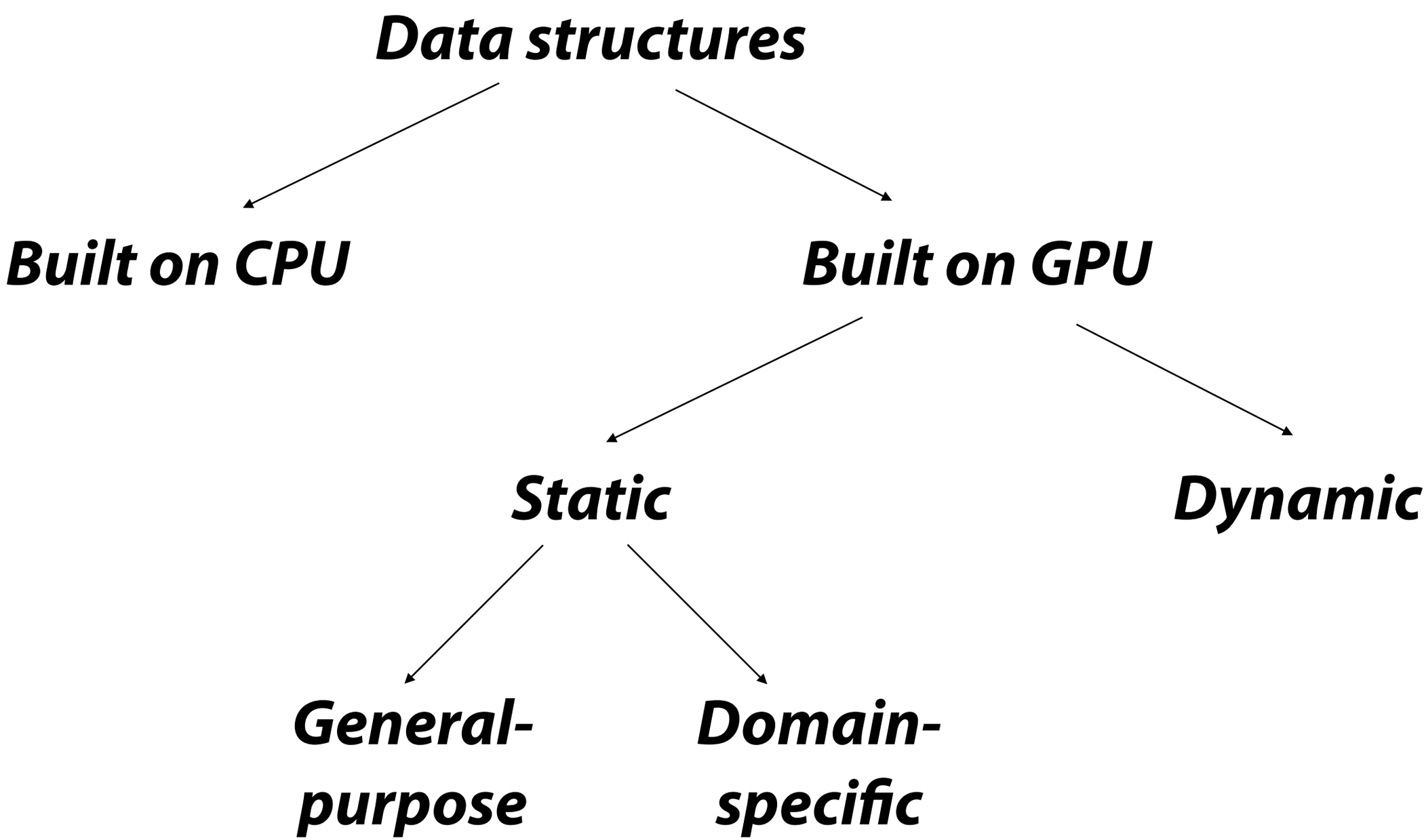
- Some data structures are simply a poor fit for the GPU
- Conventional data structures are typically defined and described serially
  - e.g., "insert item" == "find location for item in a tree, add it, rebalance the tree"
  - NOT described as bulk / parallel operations
- Even existing parallel data structures rarely see concurrency on a GPU scale

GPUs in one slide

- Kernels run over a grid of thread blocks; blocks are independent
- Thread blocks are mapped to SMs
- Threads running within a thread block can communicate through local "shared memory"
- SMs have 32-wide SIMD hardware that runs one 32-wide "warp" of threads
- Warps accessing neighboring memory locations are "coalesced"
- Thread or memory divergence: bad



Taxonomy


Data structures
Built on CPU
Built on GPU
Static
Dynamic
General-purpose
Domain-specific

Taxonomy

Data structures

Built on CPU

Built on GPU

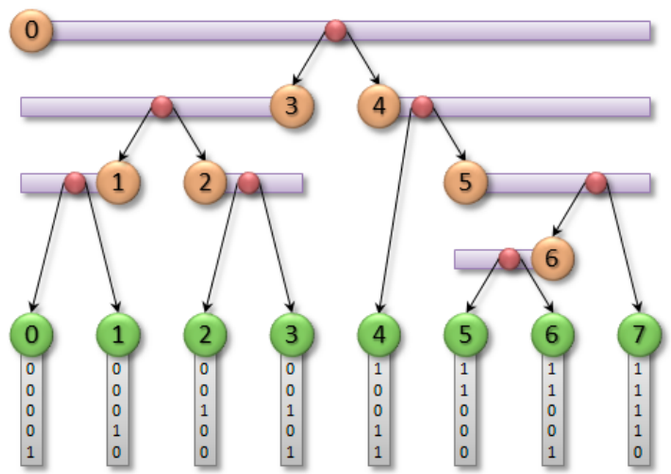
Static

Graphics data structures are typically rebuilt every frame

Dynamic

General-purpose

Domain-specific



e.g., BVH tree

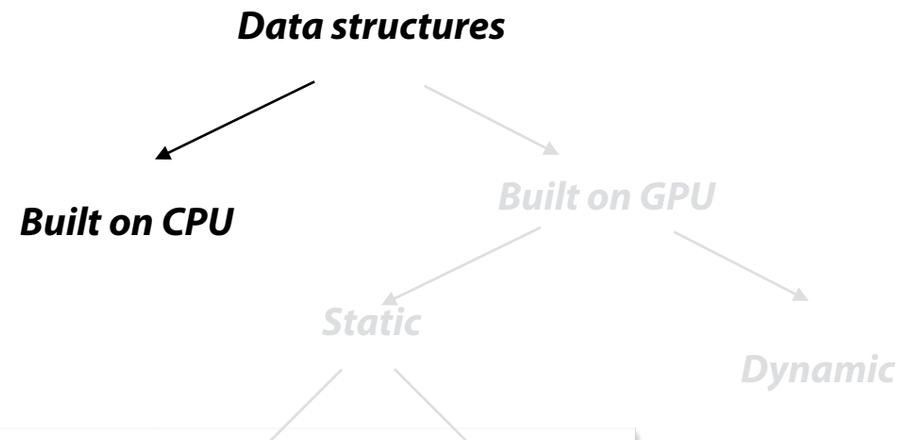
Tero Karras, Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees, Proc. of High Performance Graphics 2012

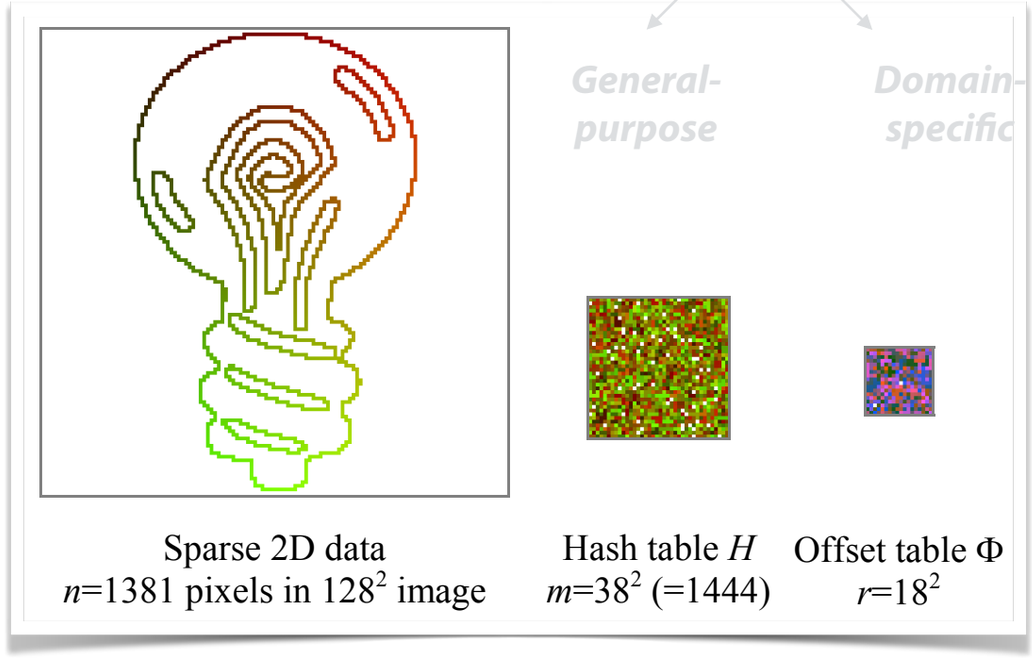
All data structures involve tradeoffs

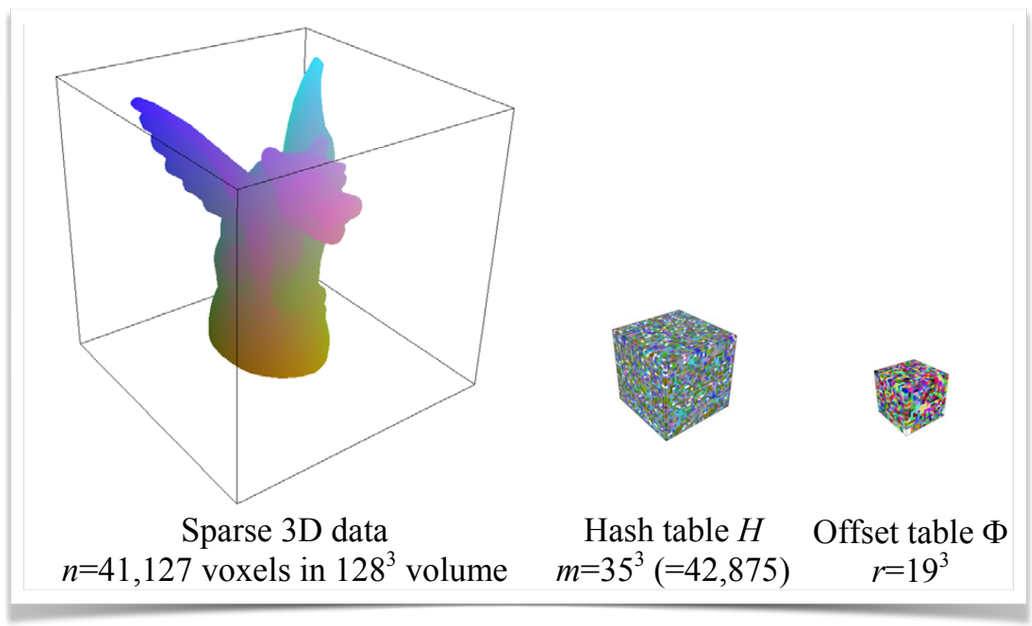
- GPU hash table implementations, for instance, typically balance:
  - Lookup time
  - Construction time
  - (Mutation time)
  - Memory usage (load factor)

Perfect Spatial Hashing

- Perfect hash function
- Built on CPU
  - Construction times: seconds to minutes
- Transferred to GPU
- Optimized for lookups

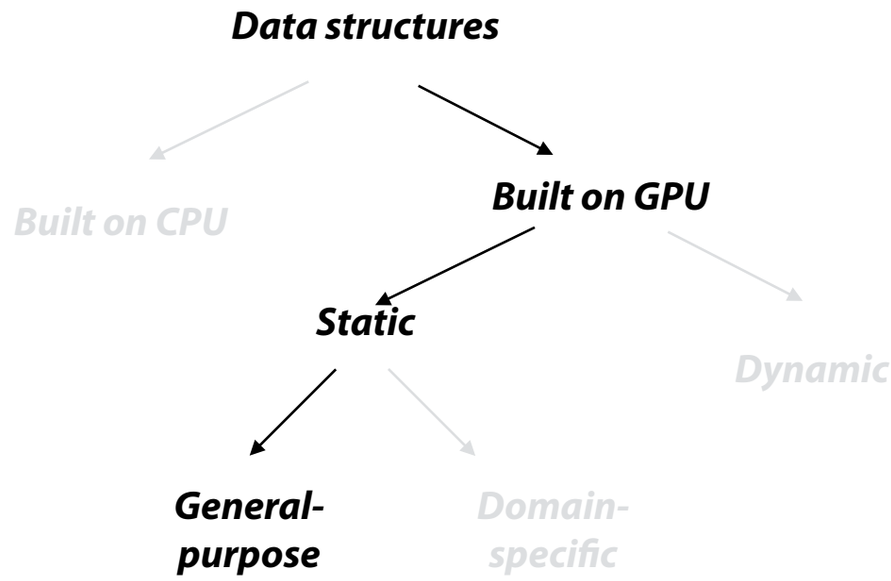




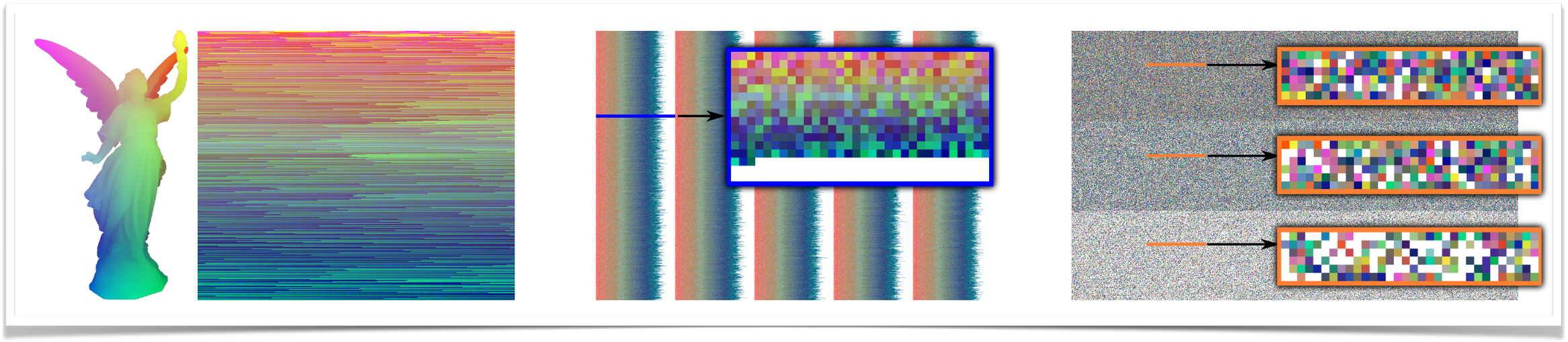


S. Lefebvre and H. Hoppe, Perfect Spatial Hashing, ACM Transactions on Graphics, 2006, vol. 25, no. 3, pp. 579–588.

GPU Cuckoo Hashing

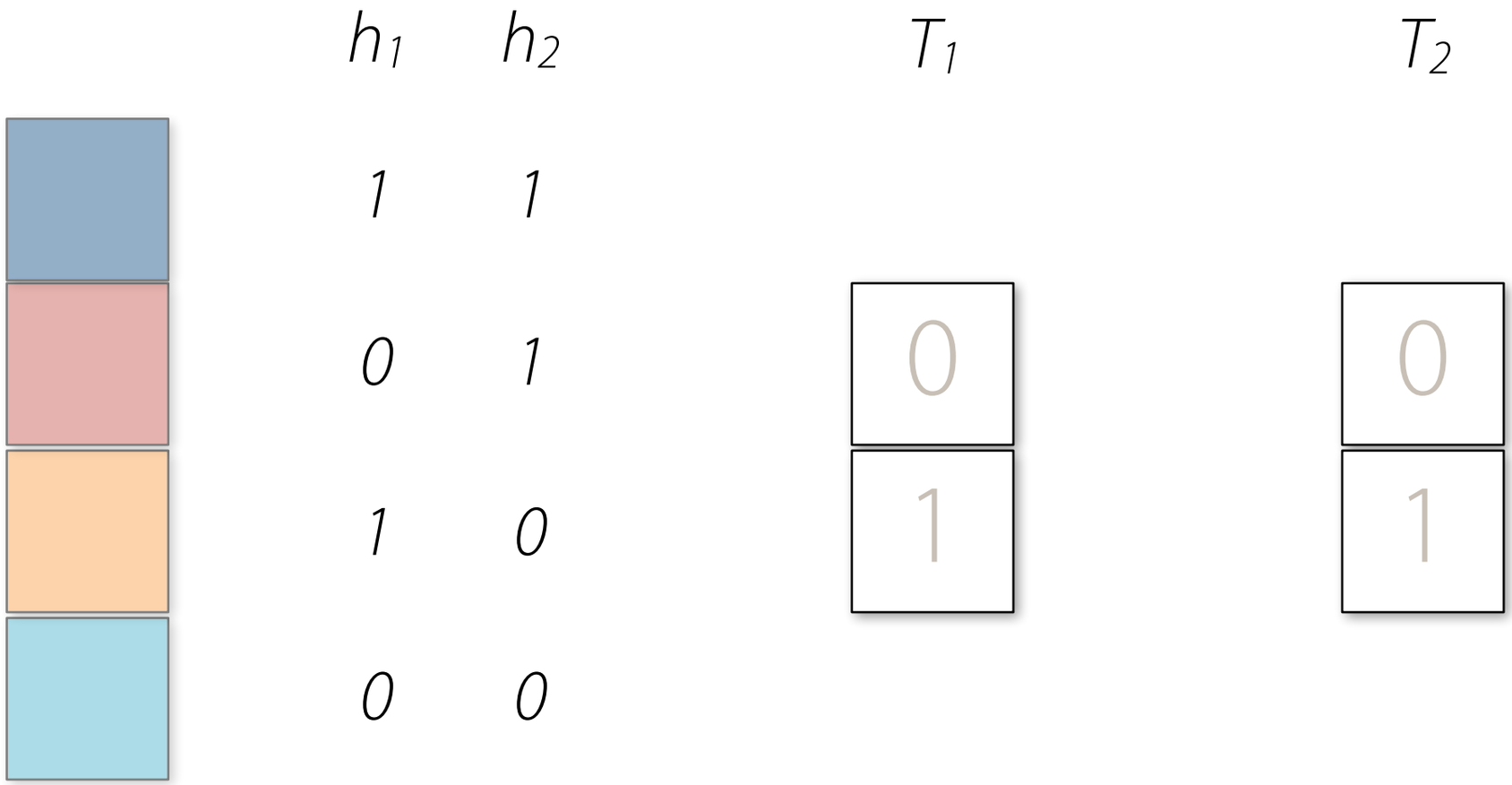


- First hash tables to be constructed on the GPU
- Cuckoo hashing: each element can be placed in one of *n* buckets; if full, ejects item already there
- Excellent best-case performance: one atomic exchange for insertion, one (non-atomic) read for query (note: stresses the TLB)
  - This is the best comparison for read & bulk build performance
- Incremental updates require a complete rebuild



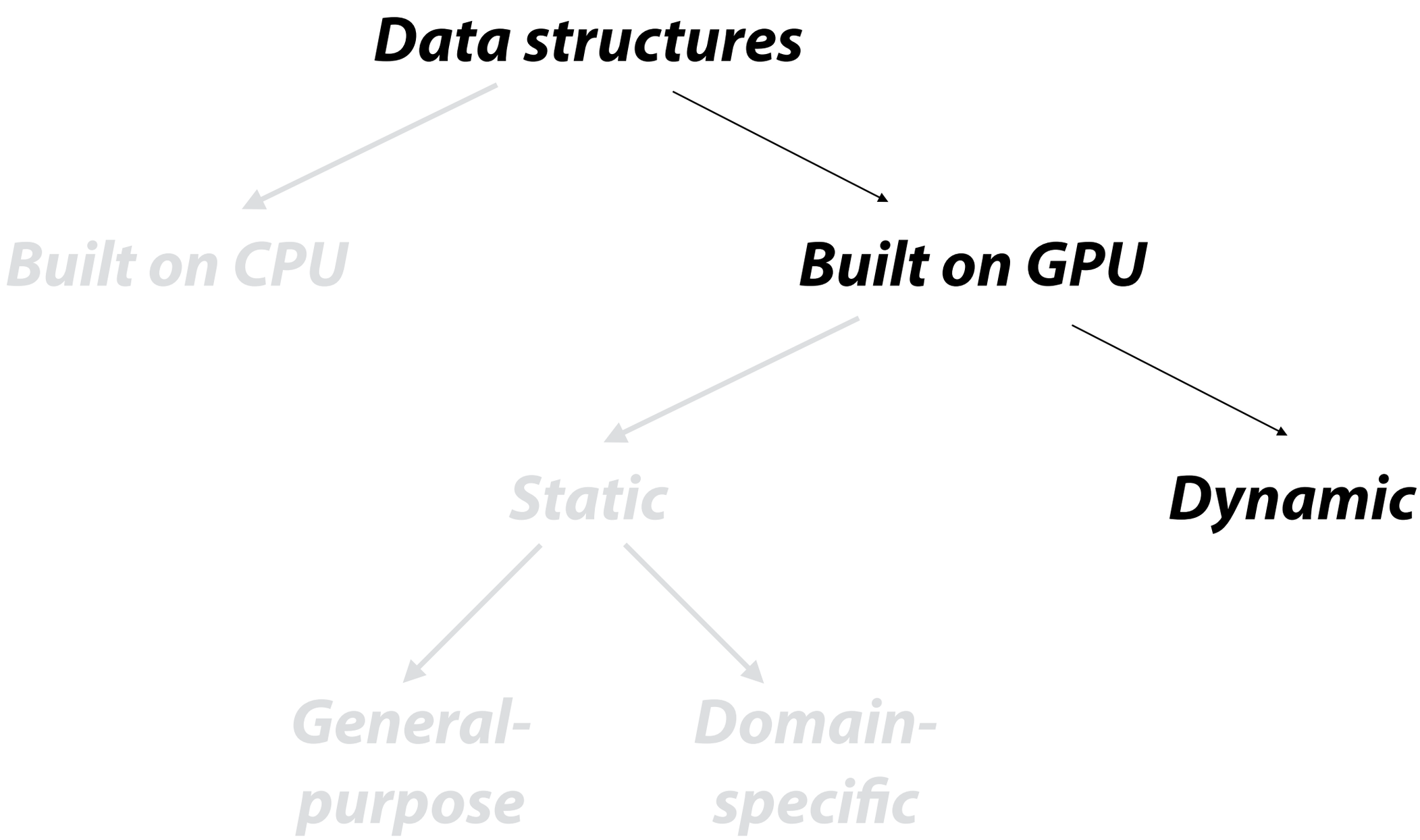
D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, Real-time Parallel Hashing on the GPU, ACM Transactions on Graphics, 2009, vol. 28, no. 5, p. 1.

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

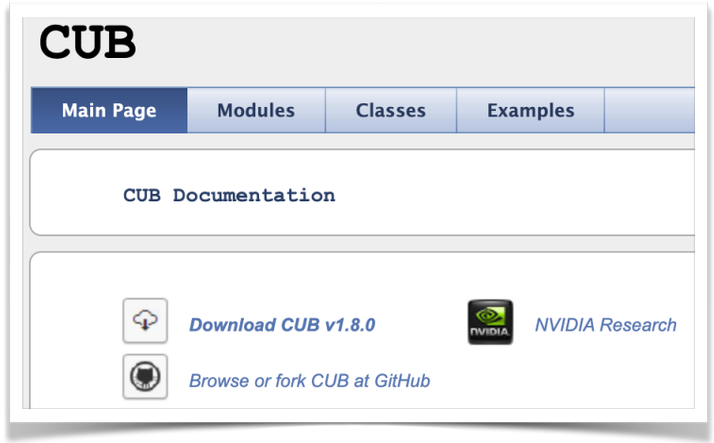
Our goal with this work



- **Goal: Cost is *O*(update size) instead of *O*(dataset size)**

Baseline data structure: sorted arrays

- Sorted arrays support point/range queries (binary search), insertions (merge), deletions (filter)
- Sort, merge, and filter are highly optimized and fast. Binary search is pretty fast.
- Sorted arrays are easy to write



- ∴ Sorted arrays are a pretty good comparison

What We Built

- Dictionary: membership, lookup, range queries, insertions/deletions
- Prioritize insert performance
- *Log-structured merge tree*

- Dictionary: membership, lookup, range queries, insertions/deletions, successor
- Prioritize query performance
- *B-tree*

- Approximate membership query: membership(k) may return "true" even if false with tunable probability 1/ϵ
- *Quotient filter*

- Dictionary: membership, lookup, insertions/deletions
- Prioritize competitiveness with static hash table
- *Hash table (& linked list & dynamic graph)*

There are no mutable GPU data structures

先有鸡还是先有蛋

There are no applications that benefit from mutable GPU data structures

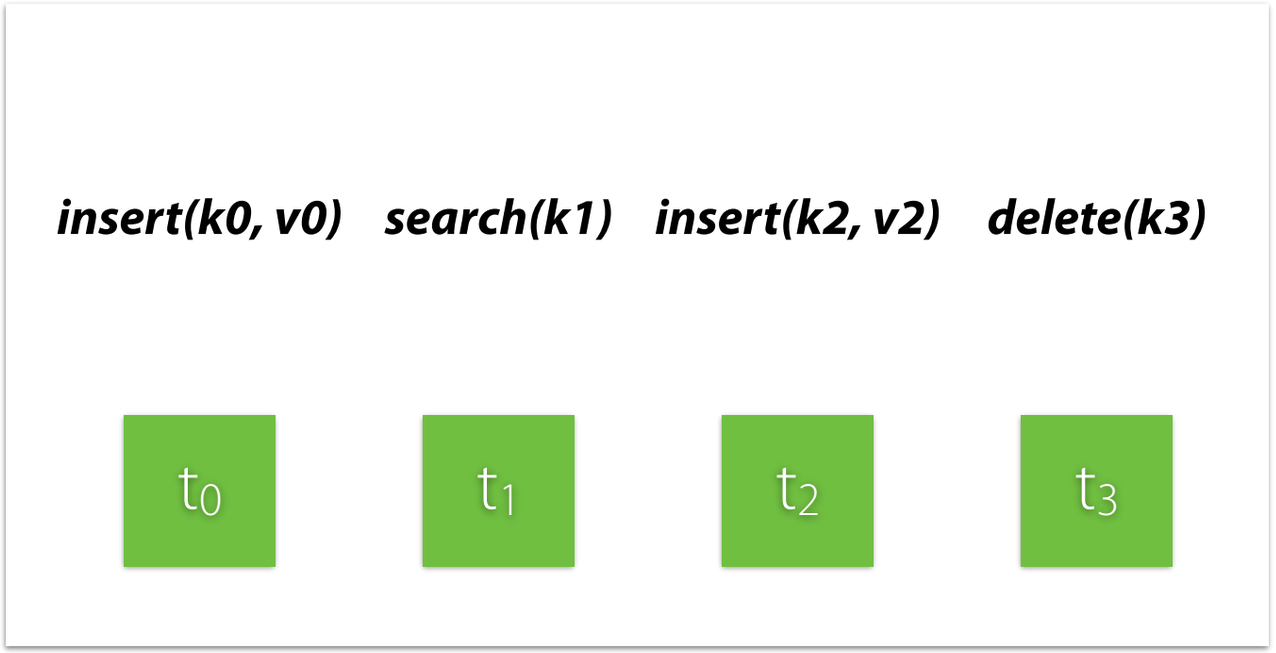
Scale of updates

- Update 1–few items
  - Fall back to serial case, slow, probably don't care
- Update very large number of items
  - Rebuild whole data structure from scratch
    - *O*(update size) ≅ *O*(dataset size)
      - Usually faster than incremental-update approach
      - May be an interesting parallel problem on its own
- Middle ground: our primary goal
  - Question: When do you do this in practice?

Operations: Coarse or Fine?

- GPU requires 100k+ threads to stay busy
- When we do lookups or mutations, we're probably going to do ~100k of them at a time, assigning one per thread
- For the underlying computation, are those threads:
  - All cooperating on a single task? (*coarse*)
  - Each performing an individual task? (*fine*)
- Another way to think about it: Do we treat the input to whatever operations we're doing as a bulk chunk of data or as a bunch of independent items?

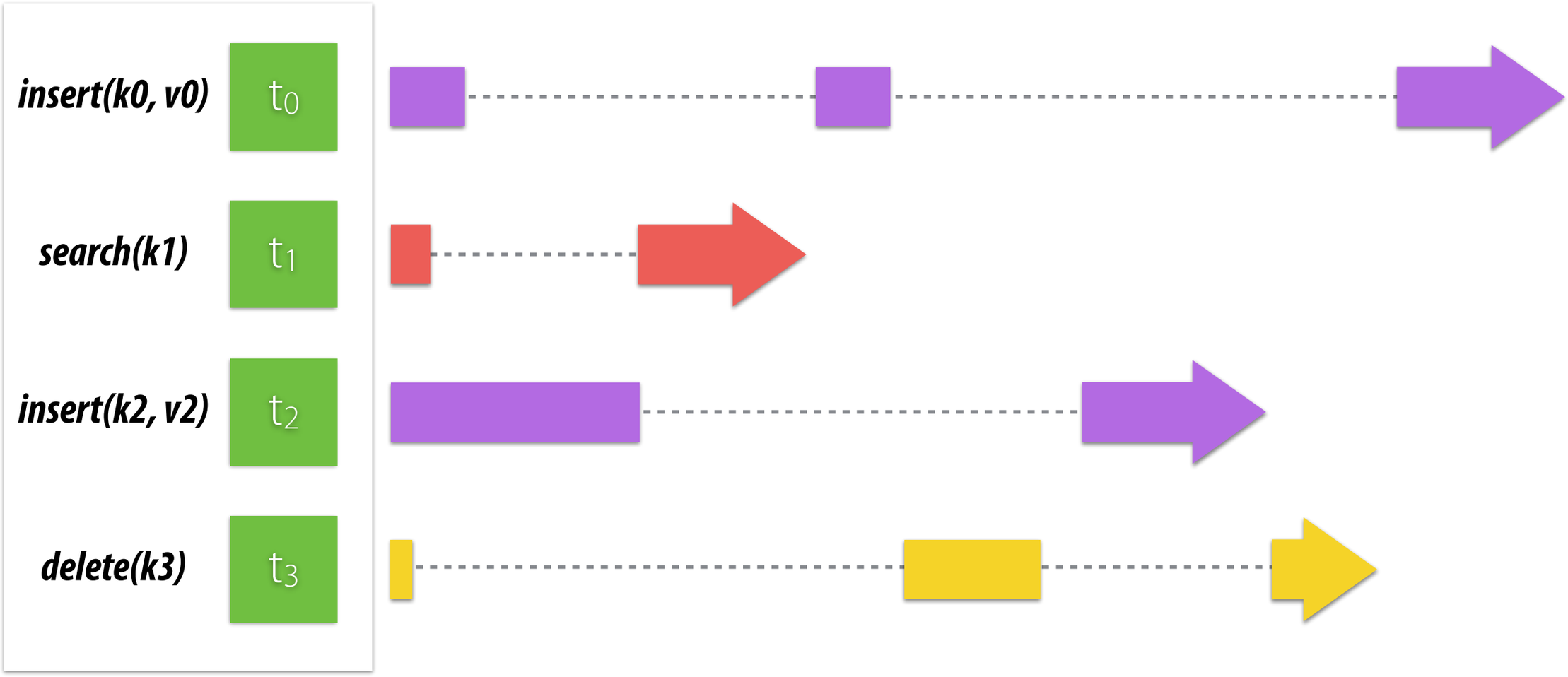
Traditionally: Each thread is independent



Input: A set of independent operations, one per thread

Traditional way: per-thread processing

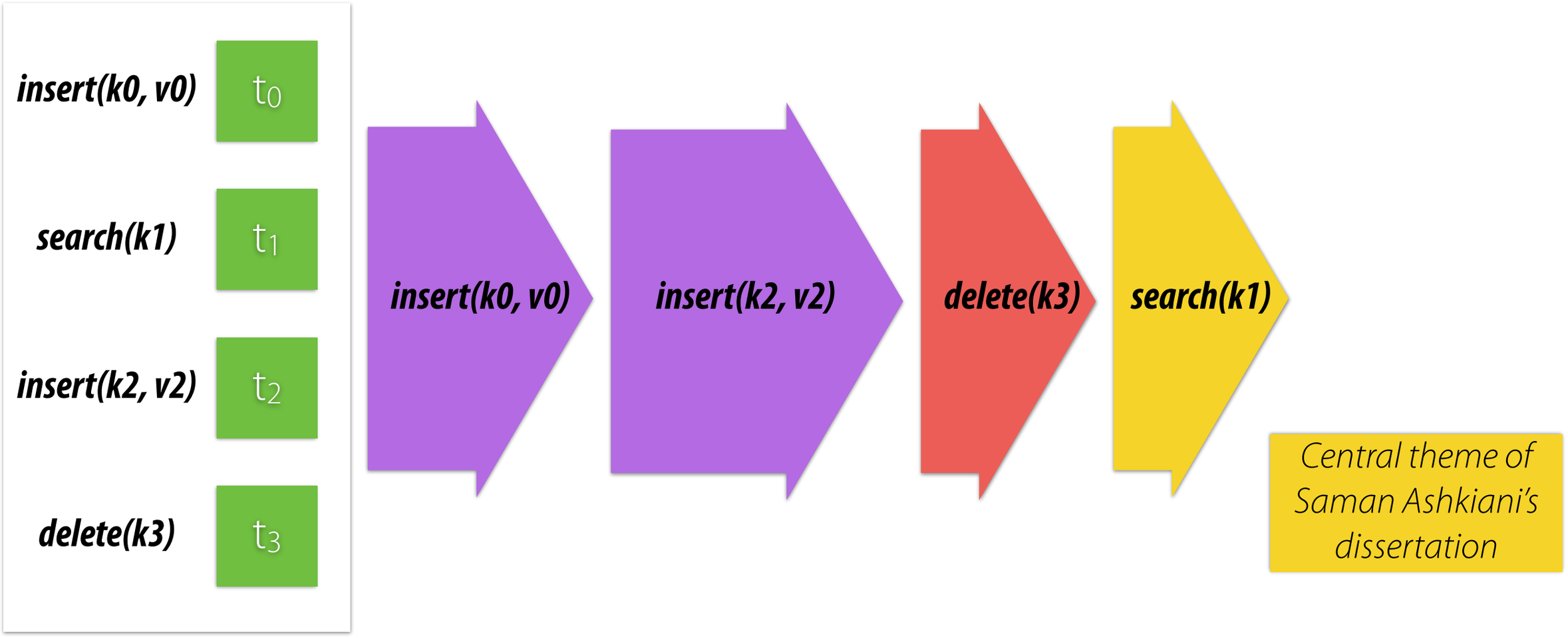
Downsides to independence



- A set of independent operations: per thread work assignment
- Traditional way: per-thread processing
- Poor performance on GPU because of uncoalesced memory accesses and branch divergence

Warp-cooperative work sharing strategy

Design Decision



Central theme of Saman Ashkiani's dissertation

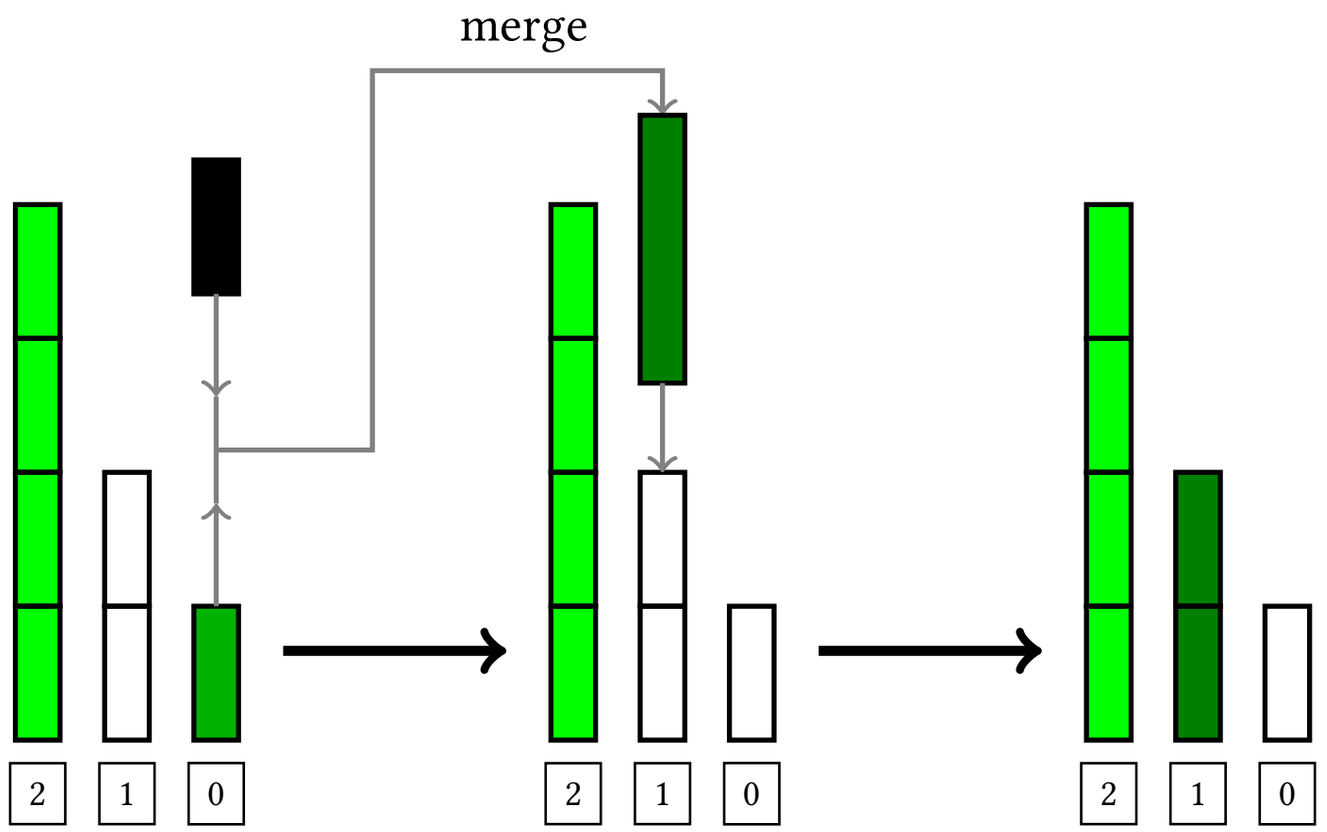
All threads within a warp cooperate to perform operations in parallel, one at a time

- **coalesced memory accesses**
- **minimal branch divergence**
- **parallel intra-warp implementation using ballots/shuffles**

Requires designing data structure to target warp-wide operations

Note: We can perform different operations concurrently!

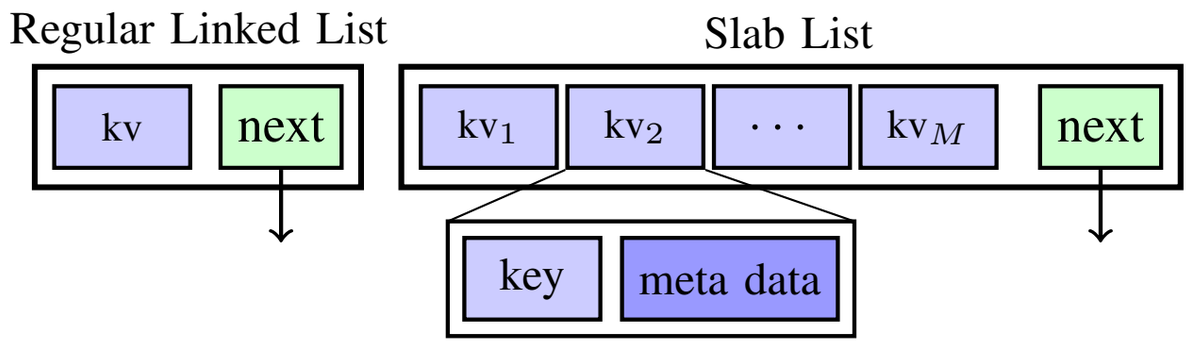
Log-structured merge tree



- Supports dictionary and range queries
- log *n* sorted levels, each level 2x the size of the last
- Insert into a filled level results in a merge, possibly cascaded. Inserts are fast. Operations are *coarse* (threads cooperate).

Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D. Owens. **GPU LSM: A Dynamic Dictionary Data Structure for the GPU**. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2018, pages 430–440, May 2018.

Slab Lists & Slab Hash

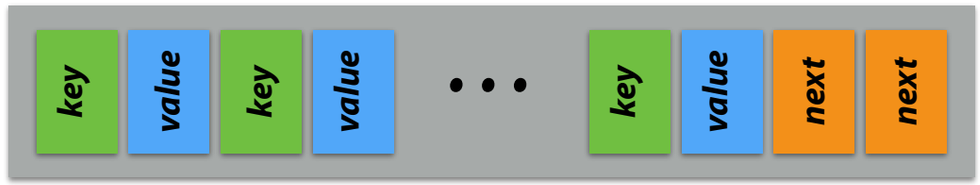


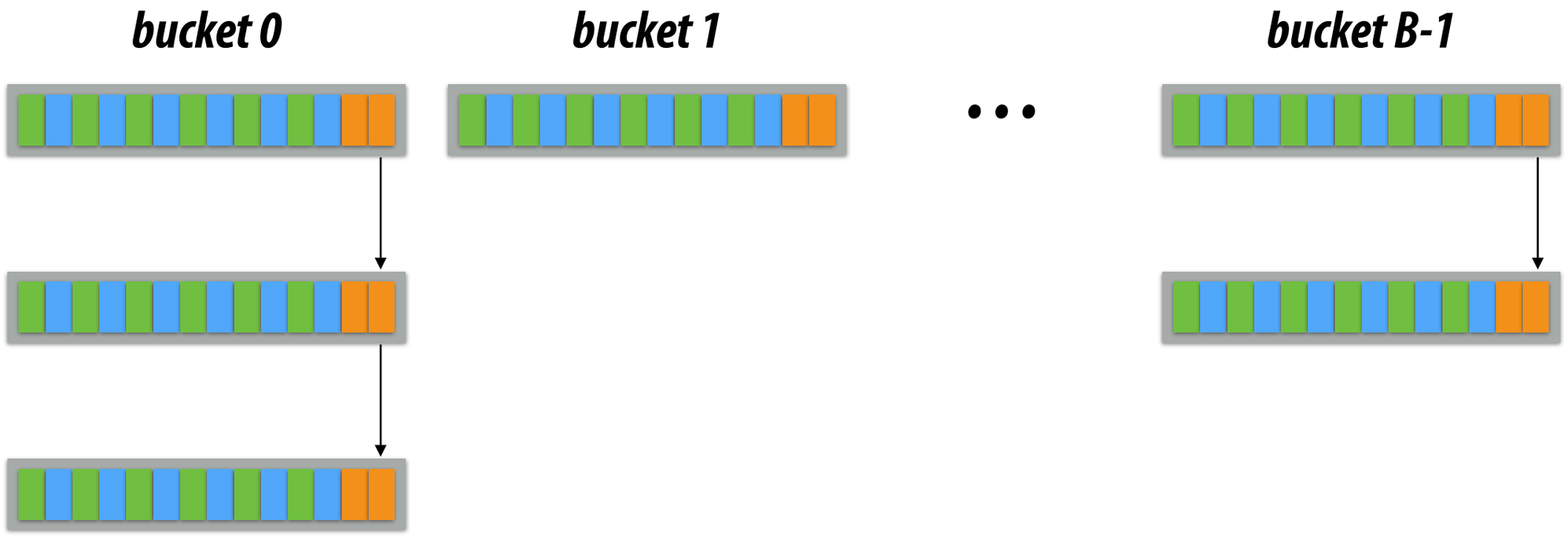
- **Linked lists are flexible, but are poorly suited for GPUs**
  - **Space inefficient. Poor locality (uncoalesced). Poor branch divergence.**
- **The "slab list" stores multiple items per "atom", allowing good parallel performance**
- **The "slab hash" (hash table) is a bucketing hash table with a slab list per bucket**

Saman Ashkiani, Martin Farach-Colton, and John D. Owens. **A Dynamic Hash Table for the GPU**. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2018, pages 419–429, May 2018.

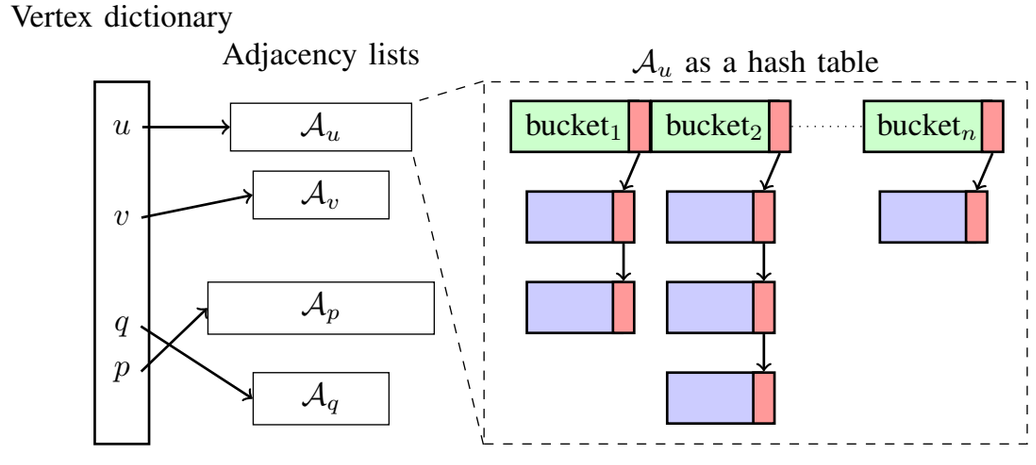
Slab Lists & Slab Hash

1. A hash table with *B* buckets, each represented by a slab list
2. Each slab is 128 bytes wide: 30 × 4B data elements + 2 × 4B next pointers
3. Green elements: keys, Blue elements: values, orange elements: next pointers and/or auxiliary





Dynamic Graphs

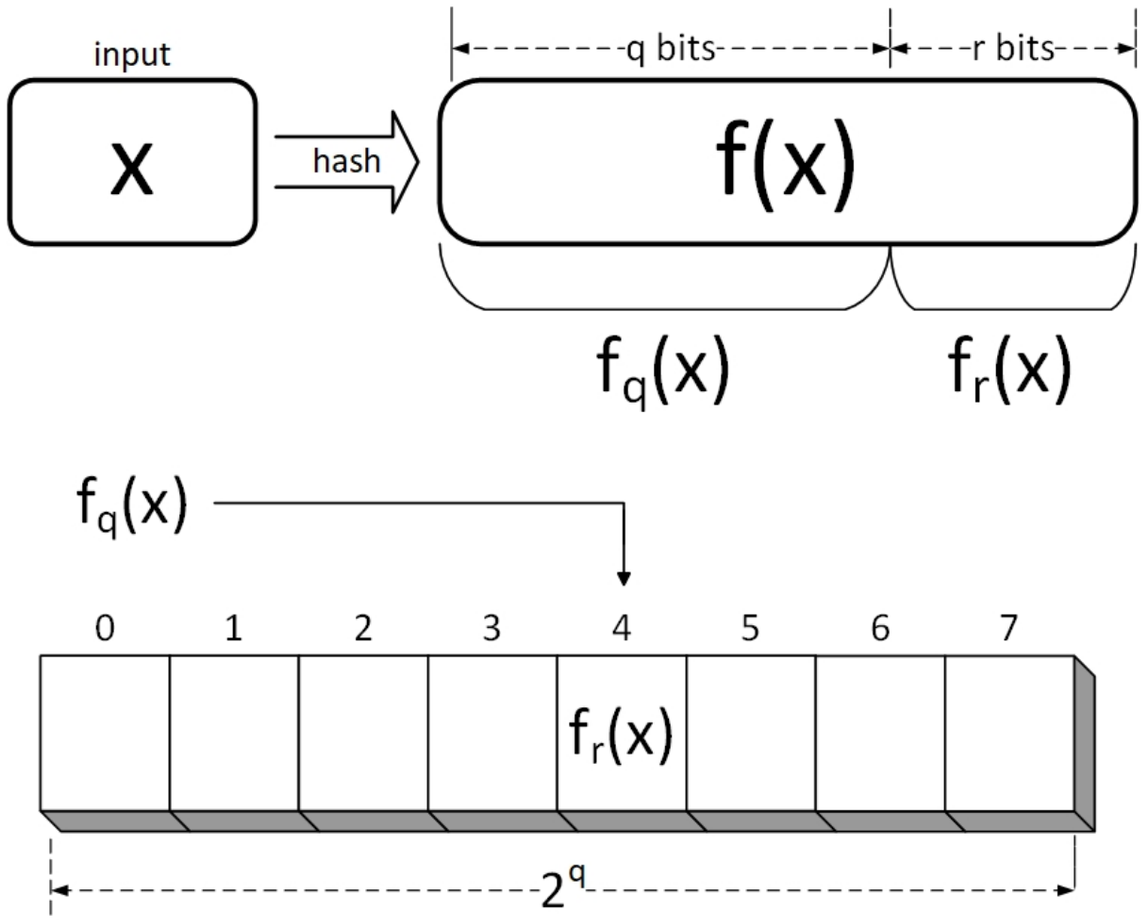


- Consider the query "does *u* have *v* as a neighbor?"
- Consider that vertex adjacencies are stored in a list
  - If that list is unsorted, the query requires visiting all elements
  - If that list is sorted, then the framework has to keep it sorted
- Conclusion: List data structures aren't a good fit for dynamic graphs
- Our work uses a hash table per vertex to store adjacencies

Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, and John D. Owens. **Dynamic Graphs on the GPU**.
In *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2020, May 2020. [bib | http]

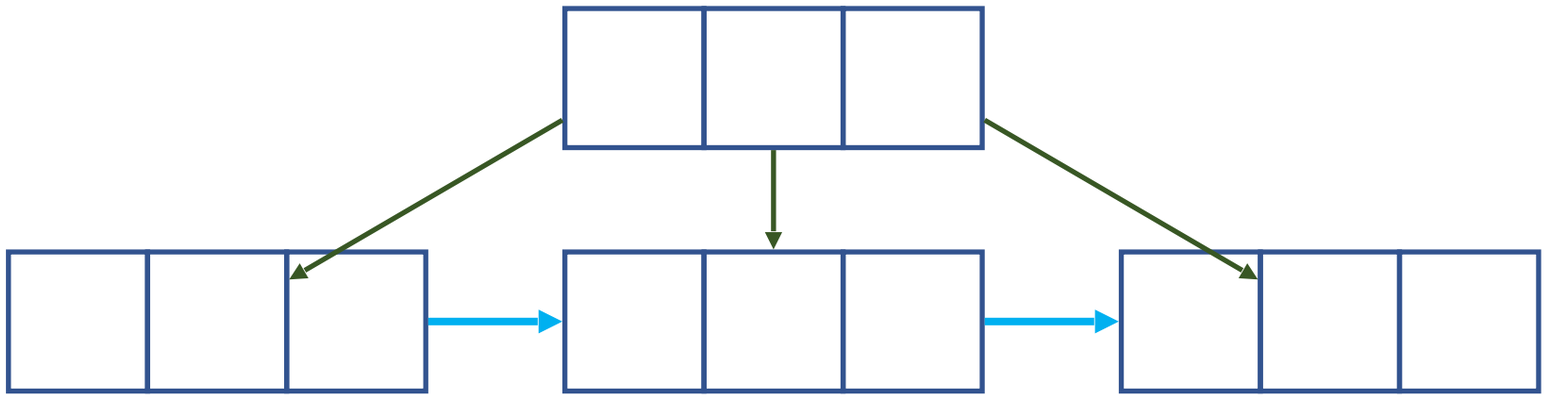
Quotient Filters

- Probabilistic membership query (cf.: Bloom filter)
- Hash all inputs into fingerprints
- Use quotienting to divide fingerprints into *quotient* and *remainder* bits
- Quotient bits determine the slot number
- Remainder bits are stored in slot



Afton Geil, Martin Farach-Colton, and John D. Owens. **Quotient Filters: Approximate Membership Queries on the GPU**.
In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2018, pages 451–462, May 2018.

GPU B-Trees



Intermediate Node	$\{Pivot, Ptr\}_0$	$\{Pivot, Ptr\}_1$	…	$\{Pivot, Ptr\}_{14}$	$Link\{Min, Ptr\}$
Leaf Node	$\{Key, Value\}_0$	$\{Key, Value\}_1$	…	$\{Key, Value\}_{14}$	$Link\{Min, Ptr\}$

- **Key design decisions:**
  - **All blocks are warp-sized and processed with warp-cooperative work sharing**
  - **Side links reduce need for locking**

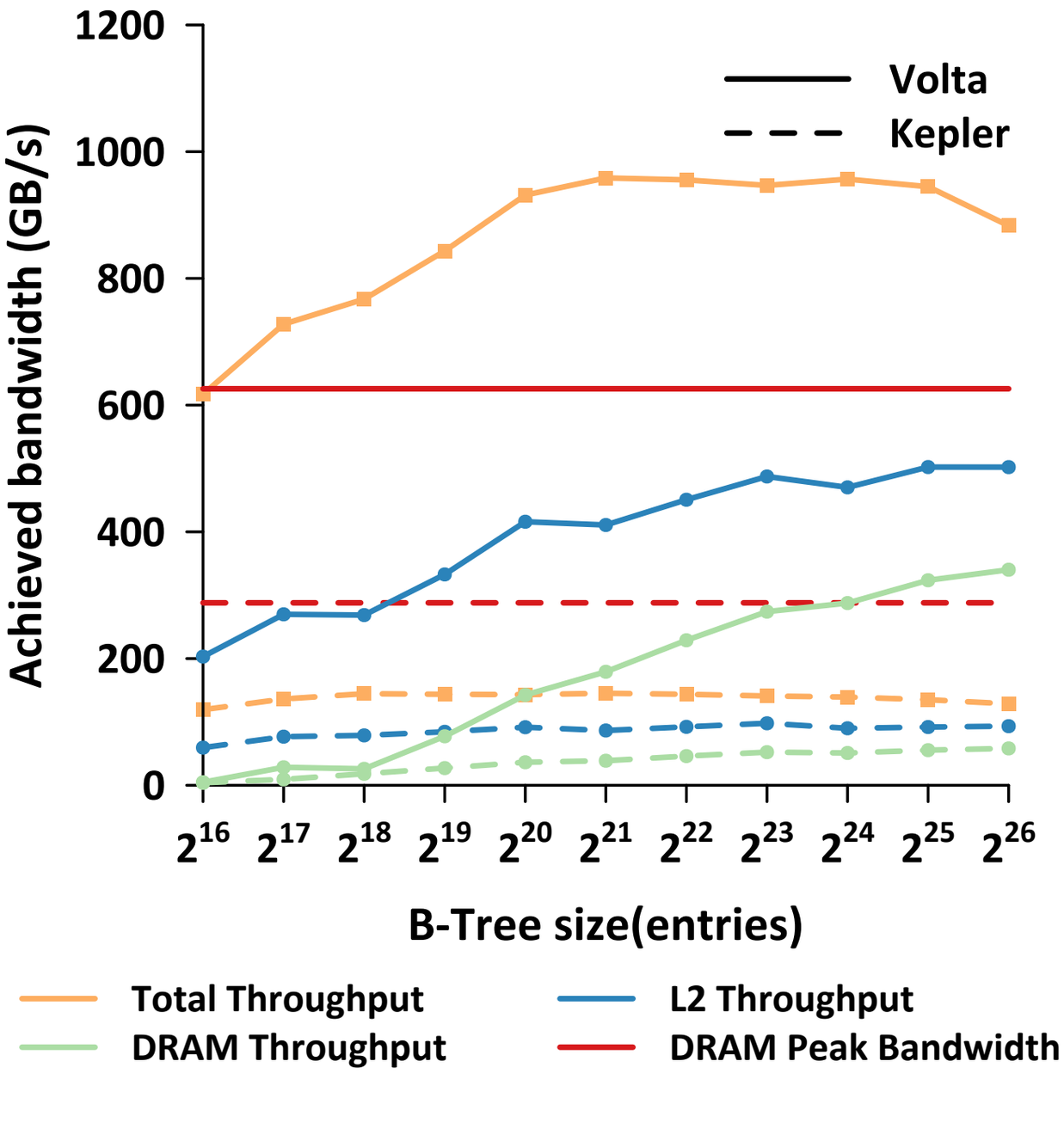
Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. **Engineering a High-Performance GPU B-Tree**. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2019, pages 145–157, February 2019. [bib | DOI | http]

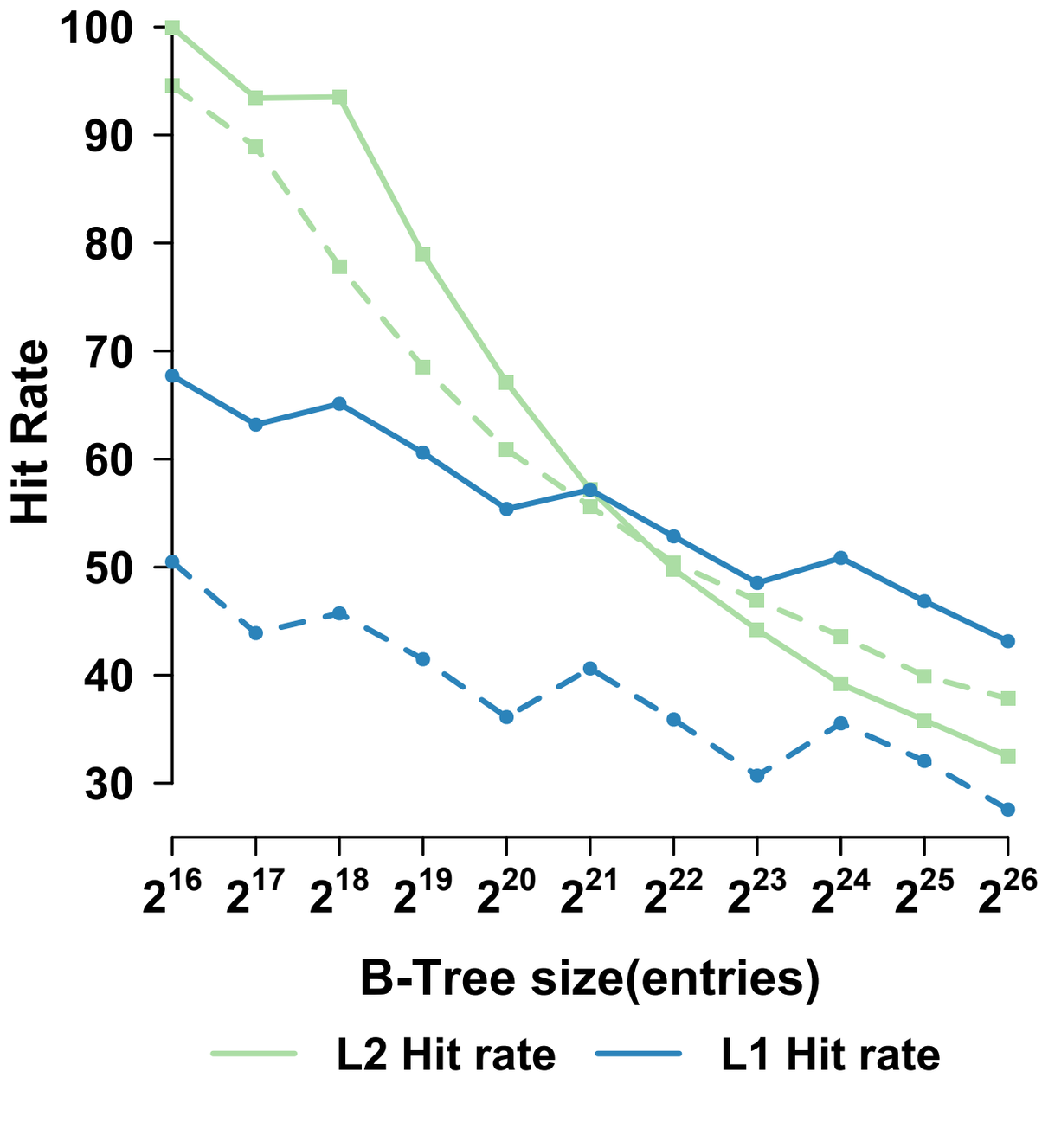
B-Tree: Contention is #1

- Side links & proactive splitting remove need for locking parents of node insertions
- Reads are latch-free; more work to switch to write but fewer locks are worth it
- Restarts instead of spinlocks

- Baseline (latch coupling, proactive splitting): 0.166 MKey/s
- Above design: 182.9 MKey/s

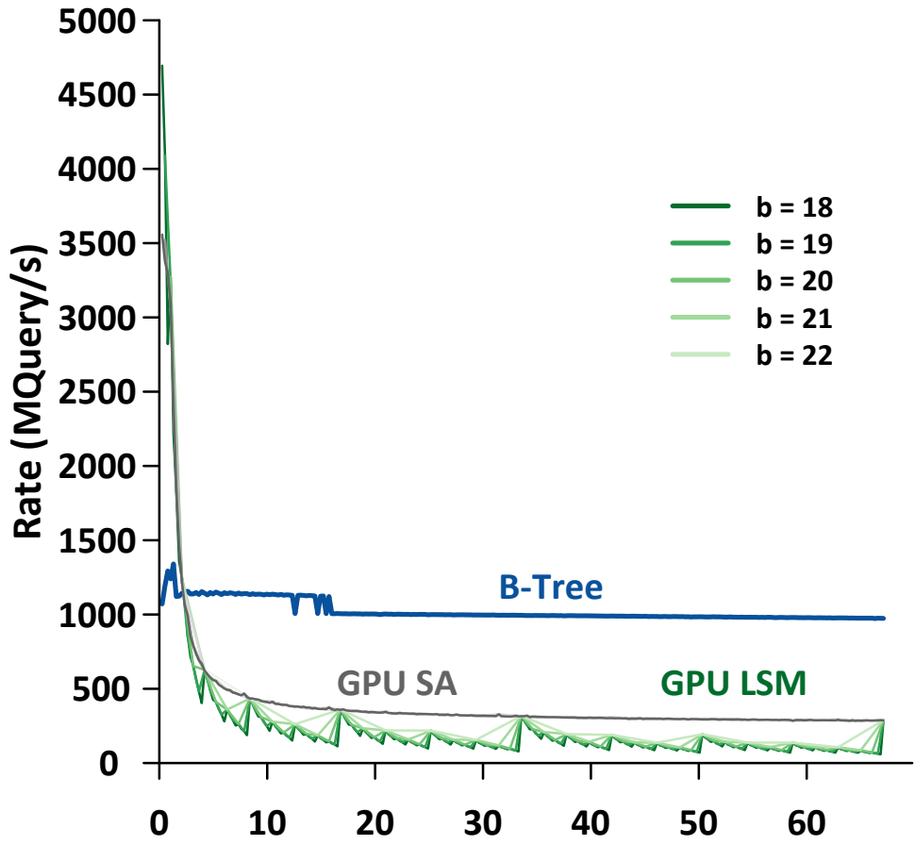
B-Tree: Caches Work





B-Tree vs. LSM

- No surprise that B-tree queries are faster than LSM …



- … but so are insertions of small batches.
- In general 10k–1M elements are necessary to approach peak performance

Millions of elements

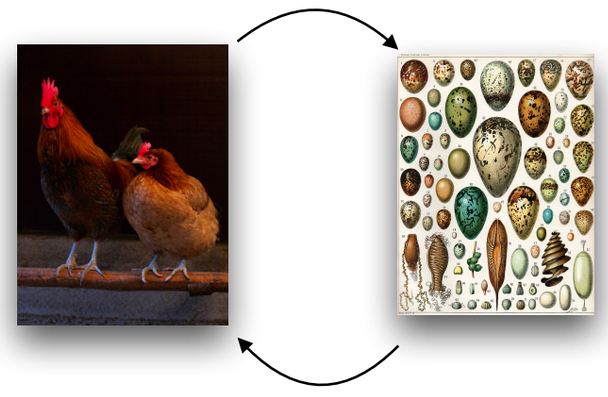
batch size	B-Tree	GPU LSM	GPU SA
2^{16}	168.0	61.5	44.9
2^{17}	139.7	121.3	87.6
2^{18}	171.7	218.6	160.6
2^{19}	190.3	402.5	292.6
2^{20}	205.1	685.9	543.0
2^{21}	211.9	1103.8	907.5
2^{22}	223.0	1603.1	1472.7
Mean	182.9	202.6	149.1

Issues with Hash Tables (that are actually general issues)

- Memory allocation was critical for performance
  - Built custom allocator just to allocate new slabs (cost: 2 registers)
  - Default allocators are terrible at small allocations
- Ideal item count per bucket approaches filling one slab
  - Performance suffers when that changes
  - In general: is stop-and-rebuild (garbage collection) a viable strategy?
- For mutable graphs:
  - We store outgoing edges with a hash table per vertex
  - Unbalanced inputs (e.g., neighbors in a scale-free graph) will be problematic for performance

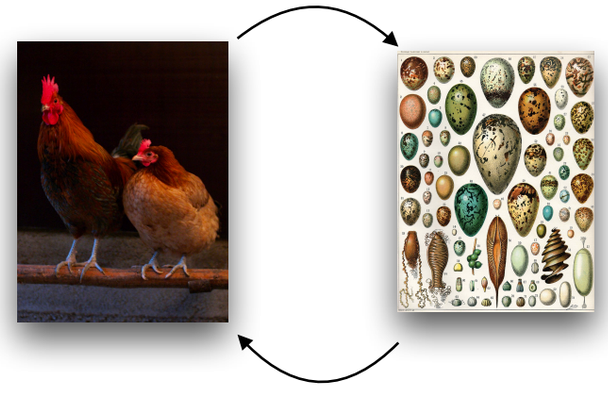
Do our APIs work?

- High priority: Build something programmers can use
  #include <data_structure.h>
- Generally, those look like “every thread has an item”
- Is that what programmers want?



Semantics

- GPUs (generally) operate with bulk semantics
  - What does this mean for ordering?
  - Within a batch of updates, how do we handle "collisions"?
- Do applications actually do updates and queries simultaneously or not?
  - Do updates and queries occur in different *phases*?
  - If not, what should happen when an insert, a delete, and a query all happen at the same time?
  - Need more applications to understand need for phases



The Future

- Dynamic graph data structure integration (cf. NVIDIA RAPIDS)
  - This work is already in Gunrock
- Revisit open-addressing hash tables
- Functional programming data structures
  - Persistent data structures
  - Data structures used in (e.g.) Haskell, Scala, Clojure
- Better understanding of interesting workloads
- Integrate into languages (~higher level than CUDA) that target the GPU
- Motivates investigating GPU runtimes

Thank you!

- NSF award CCF-1637442 (“Algorithms in the Field” program), Martin Farach-Colton, co-PI. Thanks to PM Tracy Kimbrel.
- NVIDIA AI Lab at UC Davis (est. 2019)
- DARPA support via HIVE and Software-Defined Hardware (SDH) programs
- Adobe Data Science Award
- Bill Dally, Joe Eaton, Michael Garland, Ujval Kapasi, Steve Keckler, Lars Nyland, Josh Patterson, Brad Rees, Nikolay Sakharnykh (NVIDIA)

References

Muhammad Awad, Saman Ashkiani, Serban D. Porumbescu, and John D. Owens. *Dynamic Graphs on the GPU*. In Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2020, pages 739–748, May 2020. [bib | http]

Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. *Engineering a High-Performance GPU B-Tree*. In Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, pages 145–157, February 2019. [bib | DOI | http]

Afton Geil, Martin Farach-Colton, and John D. Owens. *Quotient Filters: Approximate Membership Queries on the GPU*. In Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, pages 451–462, May 2018. [bib | DOI | http]

Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D. Owens. *GPU LSM: A Dynamic Dictionary Data Structure for the GPU*. In Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, pages 430–440, May 2018. [bib | DOI | http]

Saman Ashkiani, Martin Farach-Colton, and John D. Owens. *A Dynamic Hash Table for the GPU*. In Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, pages 419–429, May 2018. [bib | DOI | http]