Fully Concurrent GPU Data Structures

By

MUHAMMAD ABDELGHAFFAR AWAD

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

John D. Owens, Chair

Jason Lowe-Power

Soheil Ghiasi

Committee in Charge

2022

To my parents, Zaynab and Abdelghaffar.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Algorithms

LIST OF CODE LISTINGS

ABSTRACT

Fully Concurrent GPU Data Structures

Building efficient concurrent data structures that scale to the level of GPU parallelism is a challenging problem. Solutions designed for the CPU do not scale to the thousands of cores that modern GPUs offer. In this dissertation, we show how to efficiently build a lock-based B-Tree on the GPU; then, we show how to extend the B-Tree to support snapshots and linearizable multipoint queries. Finally, we show how to compose a special-purpose data structure from general-purpose ones (e.g., hash tables) and discuss the design decisions that make composing data structures easy.

Our B-Tree supports concurrent queries (point, range, and successor) and updates (insertions and deletions). The B-Tree design use fine-grain locks to synchronize between concurrent updates, yet with clever design designs that reduce contention, the tree provide high update throughput. We show how our cache-aware B-Tree design take advantage of the cache-hierarchy of the GPU, achieving lookup throughput that exceeds the DRAM bandwidth of the GPU.

We address the critical question of understanding and providing semantics for concurrent updates and multipoint queries (e.g., range query). Using linearizability, we offer an intuitive understanding of concurrent operations. We show how to build a linearizable GPU B-Tree that uses snapshots to ensure linearizable multipoint queries. To support our snapshot B-Tree, we design a GPU epoch-based-reclamation scheme.

Finally, we show how to compose a graph data structure from general-purpose hash tables resulting in a hash-based graph data structure that excels at updates and point queries. Our graph data structure outperforms sorted-array-based solutions. The design of the data structure is a general one such that we can replace the hash tables with a B-Tree to address graph problems that require sorted adjacency lists.

ACKNOWLEDGMENTS

I had the honor of learning and getting mentored by two of the most extraordinary personalities, Mohamed S. Ebeida and John D. Owens. I first virtually met Mohamed in 2014 when he offered to teach undergraduate students about computational fluid dynamics. Later, I had the chance to work on research projects with Mohamed and his collaborators. I had the honor of working with his team and collaborators, including Scott A. Mitchell, Li-Yi Wei, Ahmad A. Rushdi, and Laura P. Swiler. I listened to many of Mohamed's advice, and one of these pieces of advice was to learn parallel programming. I am very grateful to Mohamed and his collaborators for their guidance and mentoring.

Through Mohamed, I started my Ph.D. journey in John Owens' group. John is an outstanding person on the personal, academic, and managerial levels. He supported, trusted, and believed in me during my graduate-school journey. John believes in all of his students and supports them. He has a unique relationship with his current and previous students. We all appreciate you, John, and you are a role model everyone looks up to.

My dissertation work is built on a very successful collaboration between John and Martín Farach-Colton. Working with Martín was a unique experience. I learned from Martín how to explain my work to others with different backgrounds than mine. Martín supported me and advised me on multiple career decisions.

I was lucky to work with Saman Ashkiani on multiple publications. Over a very short period, I learned many things about writing efficient GPU code from Saman. Saman's approach to solving problems has influenced how I think about parallel problems.

I met Ahmed H. Mahmoud back when we were both undergraduate students in 2010. We worked together with Mohamed Ebeida, then joined John's group. Ahmed is a good friend and always supported and helped me during those many years.

I would also like to thank Serban D. Porumbescu for his support and guidance in navigating academic, professional, and other personal matters. Serban has been an outstanding postdoc in our group, and he had a significant positive influence on all the students who worked with him.

I would also like to thank my dissertation and qualifying exam committee members who advised me and provided feedback on my research direction: Jason Lowe-Power, Soheil Ghiasi,

Nina Amenta, and Venkatesh Akella. Their support and feedback on my dissertation helped me present my work in a way that I am proud of.

I enjoyed working with John's students and postdocs throughout my graduate school journey. Kerry A. Seitz has been a great help in answering various questions on the academic and professional levels. I had excellent interactions with all the students in our group, including Yangzihao Wang, Afton Geil, Yuechao Pan, Muhammad Osama, Yuxin Chen, Shalini Venkataraman, Nima Johari, Leyuan Wang, Carl Yang, Vehbi Eşref Bayraktar, Collin McCarthy, Chenshan Shari Yuan, Weitang Liu, Jason Mak, Zhongyi Lin, Matthew Yih, Jonathan Wapman, Agnieszka Łupińska, Radoyeh Shojaei, Chuck Rozhon, Toluwanimi Odemuyiwa, Teja Aluru, and Matthew Drescher. Thank you all. You inspire me with your dedication and hard work.

My research would not have been possible without the financial support from the National Science Foundation (awards CCF-1637442, CCF-1637458, CCF-1745331, CCF-1629657 and OAC-1740333), DARPA (AFRL awards FA8650-18-2-7835 and HR0011-18-3-0007), an Adobe Data Science Research Award, equipment donations from NVIDIA and their funding of an NVIDIA AI Lab at UC Davis, and a 2022 UC Davis Dissertation Fellowship.

Finally, I would like to thank my family. My parents, Zaynab and Abdelghaffar, have always been supportive and made me the person I am now. My two sisters, Noura and Mona, my brother Abdallah, and my brother-in-law Ahmed, thank you all for your support.

Chapter 1

Introduction

Graphical Processing Unit (GPU) computing has been a continuously evolving field in recent years. Using GPU computing, researchers solve problems that started with only graphics-related ones but now include many others such as data science, machine learning, and databases, among others. The continuous development of the GPU hardware and the introduction of General-Purpose Graphics Processing Unit (GPGPU) computing through platforms such as CUDA and OpenCL aided and motivated researchers to provide highly parallel, elegant, and efficient solutions to solve different problems.

One of the continuously evolving research fields is designing and building concurrent data structures on the GPU. Data structures are in the heart of most branches of computing. Driven by the advances in machine learning problems, data analytics is one of these branches that recently flourished. Platforms such as HEAVY.AI [28] and Rapids [52] provide their users with high-level interfaces and abstractions to analyze streams of terabytes of data. We would like to perform all operations on the GPU to avoid the high latency of memory transfers between the GPU and the CPU. Moreover, the dynamic nature of real-life workloads requires sophisticated data structures that can handle streams of update and read operations.

Dynamic data structures are ones that support updating the data structure without the need to rebuild the data structure from scratch on every update. Concurrent data structures support performing different types of operations concurrently—enabling algorithms and workloads that perform a mix of read and update operations. In this dissertation, we will focus on two main problems. For the first problem, we will explore the problem of building a dynamic concurrent

GPU lock-based data structure. For the second problem, we will show how we can compose a special-purpose data structure such as a graph from general-purpose ones.

Fully-concurrent GPU B-Tree. While lock-based solutions may seem easier to implement than wait-free ones, an efficient design of a lock-based data structure is a challenging one. Reducing contention is one of the key challenges when designing an efficient lock-based data structure. Minimizing and avoiding contention is even more challenging on GPUs which run thousands of threads. CPU-based solutions do not scale to that level of parallelism. Additionally, GPUs require solutions that are aware of the hardware specifics. For instance, a GPU data structure must achieve coalesced memory accesses (i.e., avoid memory divergence), and it needs to adapt to challenges such as cache-incoherence. To address these challenges, we will introduce a GPU B-Tree design that scales to hundreds of thousands of threads on a GPU. Two key insights allow us to build the data structure efficiently and avoid contention. First, an efficient highly-parallel design must accept temporary intermediate states of the data structure where its invariants (e.g., balance) are not maintained. Second, locking multiple nodes of the data structures must be minimized. Strategies such as proactive splitting allow us to minimize the number of locks.

While efficiently performing updates is necessary, it is also essential to perform tree traversals efficiently. Traversing the tree is the core operation in the data structure. Performing data structures operations in a cooperative fashion allows us to traverse the tree efficiently. Cooperative processing allows us to avoid branch divergence and achieve coalesced memory accesses. Another challenge with data structures that support multipoint queries is providing linearizable multipoint queries and updates. Linearizability is an essential property of a data structure to allow programmers to view the data structure as an atomic object, providing an intuitive understanding of concurrent operations. We add support for snapshots into our B-Tree design to address this challenge. Taking a snapshot of the data structure allows its users to capture the state of all the keys and values stored in the data structure. Any read-only operation can run on the snapshot while having the illusion that the operation has exclusive access to the data structure.

Composing GPU data structures. Composability is an elegant way to build systems. It allows us to reuse components to build more special-purpose complex systems. We will compose a dynamic graph data structure from a general-purpose one. Using a hash table, we will build a hash-based graph data structure. There are two main challenges we will discuss. The first challenge: how can we design a GPU data structure in a way such that it can be easily composable? Performing operations on the GPU in bulk and relying on device-wide primitives such as sort do not compose well when building a data structure that may store few keys. These device-wide primitives achieve high performance only when the GPU is fully utilized. Our fully-concurrent data structure design focuses on achieving optimal performance when operations are performed on the device and makes no assumptions on the workload of the updates or queries. Our design is suitable for composition.

The second challenge: how do we choose the right data structure when composing a graph data structure? The answer to this question depends on the workload and the operations that the graph algorithm will require. For instance, a hash-table-based design will excel when the graph constantly changes (i.e., focusing on the update performance). We will focus on achieving high update throughput and hence compose our graph data structure from hash tables to solve a dynamic triangle counting problem. The triangle counting problem is interesting because it depends on the set intersection operation. Although maintaining adjacency lists of graph vertices in sorted order—a property that hash tables do not support efficiently—allows performing the intersection operation more efficiently, as we shall see, the fast update throughput will allow us to amortize the slower hash-based interaction operations.

Nevertheless, one may compose the graph data structure from other data structures such as a B-Tree. In fact, we can compose our data structure from both data structures allowing vertices to have one of the two representations or even both. One can dynamically choose and switch between the base data structures during runtime based on graph and problem-specific properties (e.g., node degree).

To summarize, this dissertation has the following contributions:

1. Efficient design of a dynamic GPU B-Tree,

2. Our B-Tree support versioning (snapshots),

3. Our B-Tree support linearizable mutipoint queries,

4. A dynamic hash-based graph data structure.

We will provide readers building dynamic data structures in the future with insights for:

- Achieving composability easily,

- Building efficient data structures that scale to GPU parallelism.

Chapter 2

Background

2.1 Taxonomy of GPU Data Structures

GPU data structures can be divided into two main categories: static and dynamic. In all cases, at least queries are accelerated using the GPU. From a GPU perspective, a static data structure is one that, on each update, the data structure is either updated on the CPU then copied to the GPU or built in bulk on the GPU. The cost of updating a static data structure is related to the total size of the data structure and not the size of the update itself. On the other hand, dynamic data structures reside entirely on the GPU. They can support phase-concurrent and fully-concurrent updates. In a phase-concurrent data structure, the data structure only supports updates and queries in phases (i.e., queries and updates do not overlap). On the other hand, a fully concurrent data structure supports queries and updates within a single GPU kernel or on multiple GPU streams. Figure 2.1 shows a summary of the taxonomy of GPU data structures.

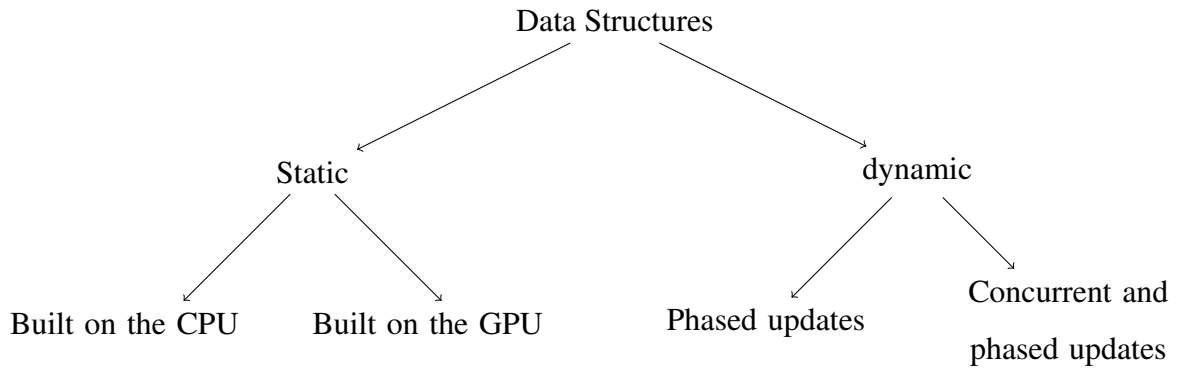


Figure 2.1: Taxonomy of GPU data structures.

Static GPU data structures. A static data structure can either be constructed on the CPU (then moved to the GPU) or on the GPU. In both cases, queries are accelerated by utilizing the GPU. For example, Breslow et al. [15] builds their hash tables on the CPU then accelerates queries on the GPU. Similarly, Shahvarani and Jacobsen [56] build and update their B-Tree on the CPU. On the other hand, many static data structures are constructed on the GPU. Karras [35] introduced an in-place binary radix trees construction algorithm which they use to build bounding volume hierarchies among other data structures. Alcantara et al. [2] accelerate building a static hash table using the GPU.

Dynamic GPU data structures. A dynamic GPU data structure, on the other hand, allows updating the data structure without rebuilding it from scratch on each update. In one case, updates are only supported in a *phased* approach, which means that the updates are applied to the data structure without any overlap with query operations. In a phased approach, updating the data structure is either performed entirely on the GPU or broken down into multiple serial stages performed on the GPU and the CPU (to avoid locking, splitting, or re-balancing the data structure). The GPU Log-Structured Merge tree [5] is an example of a write-optimized data structure where updates and queries are performed in phases. For concurrent data structures, Ashkiani et al. [4] introduced a fully-concurrent hash table. In this dissertation, we focus on fully concurrent data structures.

2.2 Graphics Processing Units (GPUs)

2.2.1 Execution Model

Graphics Processing Units (GPUs) feature several streaming multiprocessors (SMs). A group of threads is called a thread-block, and each is assigned to one of the SMs. All resident thread-blocks on an SM share the local resources available for that SM. The hardware assigns thread-blocks to SMs, and the programmer has no direct control over it. In reality, not all resident threads on an SM are executed in parallel. Each SM executes instructions for a group of 32 threads, a warp, in a single-instruction-multiple-data (SIMD) fashion. Recent architectures (e.g., Volta) support independent thread scheduling where each thread has its program counter (PC), enabling a true SIMT-model at the hardware level where per-thread forward-progress

is guaranteed [19]. The forward-progress guarantee facilitates porting concurrent algorithms to the GPU. As we shall see in this dissertation, designing concurrent data structures for the GPU is not a simple porting process. Designs targeting the GPU require careful attention to the details of how the GPU hardware work to achieve optimal memory access patterns, avoid branch divergence, and avoid contention.

Cooperative groups provide an abstraction to organize a group of threads into implicit or explicit groups. We will discuss a type of the explicit groups called *thread block tile*, which is of interest to this dissertation. A thread block tile can contain a number of threads between 1 and 512 threads. A group can communicate using instructions to broadcast a value across the tile using a shuffle instruction or vote using "all" or "any" instructions. When the group size is at most a warp, communication is performed through the fast registers, and additional functionality is available (e.g., ballot vote instruction). However, when the group size is larger than the warp size, communication within the tile is performed through an L1 user-managed cache shared between threads in a block.

2.2.2 Memory Hierarchy

GPUs have a DRAM global memory and an L2 cache that all SMs can access. Each SM has its own private L1 cache. Jia et al. [33] explored the details of the memory hierarchy through microbenchmarks. We summarize their finding here. On Volta architectures, the L1 caches are indexed using virtual addresses; however, L2 caches are addressed by physical addresses. TLB entries are cached at both the L1 and L2 caches. Memory accesses hitting the L1 cache enjoy a low latency of tens of cycles. Missing the L1 cache increases the latency up to 193 cycles—an order of magnitude higher. Accessing the DRAM while missing all caches and TLBs increases the latency by an additional order of magnitude, with latency as high as a thousand cycles. It is also worth noting that recent Ampere architectures allow programmers to control the caches in ways such as prefetching, eviction priorities, and invalidation. Memory transactions are typically divided into sectors (32 bytes) when transferred between the different cache hierarchy levels; however, accessing the memory on a cache-line granularity (i.e., 128 bytes) is the most efficient one as it reduces the cost of accessing the DRAM. Table 2.1 summarizes the GPU memory hierarchy latency and sizes.

		Volta V100	Kepler K80
L1 data	Size	32–128 KiB	16–48 KiB
	Line Size	32 B	128 B
	Hit latency	28 cycles	35 cycles
	Update policy	non-LRU	non-LRU
L2 data	Size	6,144 KiB	1,536 KiB
	Line size	64 B	32 B
	Hit latency	∼193 cycles	∼200 cycles

Table 2.1: Summary of memory hierarchy microbenchmarking results [33] on the Volta and Kepler architectures.

An efficient GPU data structure must minimize the DRAM's expensive accesses and take advantage of the cache hierarchy whenever possible.

2.2.3 Memory Model

NVIDIA's Parallel Thread Execution (PTX) ISA follows a weakly-ordered and scoped memory model and was formalized recently by Lustig et al. [46]. Similar to C++'s standard, memory instructions (e.g., load, store, and atomics) support different memory orders such as `relaxed`, `release`, or `acquire`. Memory instructions can be qualified as `weak`, indicating a memory instruction with no synchronization (i.e., can read stale data from the L1 cache). PTX also provides memory fences to establish synchronization between different memory accesses. Moreover, scopes define the synchronization boundaries for a memory operation. Scopes can synchronize memory operations on a CTA, GPU, or whole-system level. Scopes are not traditionally used on CPUs.

The efficient implementation and design of a GPU data structure requires understanding and using proper memory orderings to guarantee correctness but also avoiding unnecessary synchronizations that limit performance. For instance, data structure operations may still use stale data in the L1 cache while maintaining correctness. Similarly, limiting the scope to the smallest necessary level guarantees correctness while avoiding synchronization across the entire GPU or system.

It is also worth noting that although different memory orders are well-defined, the compiler may implement a memory order by adding more synchronizations. Inspecting the lower level

assembly (SASS) provides more insights into how the different memory orders are implemented (i.e., how they map to memory- and cache-flush operations).

2.2.4 Putting It All Together

In order to realize an efficient GPU implementation, programmers should consider the following two criteria for a warp's threads: 1) avoid discrepancy between neighboring threads' instructions, 2) minimize the number of memory transactions required to access each thread's data. The former is usually achieved by avoiding branch divergence and load imbalance across threads, while the latter is usually achieved when consecutive threads access consecutive memory addresses (a coalesced access). Unfortunately, it is not always possible to achieve such design criteria. Depending on the application, programmers have devised different strategies to avoid performance penalties caused by diverging from the mentioned preferences. In the context of concurrent data structures, each thread within a warp may have a different task to pursue (insertion, deletion, or search), while each thread may have to access an arbitrarily positioned part of the memory (uncoalesced access). To address these two problems, Ashkiani et al. proposed a Warp Cooperative Work Sharing (WCWS) strategy [4]: independent operations are still assigned to each thread (per-thread work assignment), but all threads within a warp cooperate to process in parallel (per-warp processing). By doing so, threads cooperate in both memory accesses and executed instructions, resulting in coalesced accesses and reduced branch divergence. The WCWS can be generalized to use a cooperative groups tile instead of a warp. However, we prefer tiles with a size less or equal to the maximum warp size since they can utilize the fast intra-warp communication instructions that use registers.

Chapter 3

Engineering a High-Performance GPU B-Tree

3.1 Introduction

The toolbox of general-purpose GPU data structures is sparse. Particularly challenging is the development of dynamic (mutable) data structures that can be built, queried, and updated on the GPU. Until recently, the prevailing approaches for dealing with mutability have been to update the data structure on the CPU or to rebuild the entire data structure from scratch. Neither is ideal.

Only recently have dynamic GPU versions of four basic data structures been developed: hash tables [4], sparse graphs with phased updates [27], quotient filters [25], and log-structured merge trees (LSMs) [5]. LSMs provide one of the most basic data-structural primitives, sometimes called a key-value store and sometimes called a dictionary. Specifically, an LSM is a data structure that supports key-value lookups, successor and range queries, and updates (deletions and insertions). This combination of operations, as implemented by red-black trees, B-trees, LSMs or B^ϵ-trees, is at the core of many applications, from SQL databases [31, 47] to NoSQL databases [18, 38] to the paging system of the Linux kernel [44].

In this work, we revisit the question of developing a mutable key-value store for the GPU. Specifically, we design, implement, and evaluate a GPU-based dynamic B-Tree. The B-Tree offers, in theory, a different update/query tradeoff than the LSM. LSMs are known for their

This chapter appeared as "Engineering a High-Performance GPU B-Tree" published at PPoPP 2019 [6].

	B-Tree	Sorted Array	LSM
Insert/Delete	$O(\log_B n)$	$O(n)$	$O((\log n)/B)$ amortized
Lookup	$O(\log_B n)$	$O(\log n)$	$O(\log^2 n)$
Count/Range	$O(\log_B n + L/B)$	$O(\log n + L/B)$	$O(\log^2 n + L/B)$

Table 3.1: Summary of the theoretical complexities for the B-Tree, sorted array (SA), and LSM. B is the cache-line size, n is the total number of items, and L is the number of items returned (or counted) in a range (or count) query.

insertion performance, but they have relatively worse query performance than a B-Tree [11, 42].

Table 3.1 summarizes the standard theoretical analysis of insert/delete, lookup, and count/range for n key-value pairs in our B-Tree, in a sorted array (SA), and in the LSM. Searches in GPU versions of these data structures are limited by GPU main-memory performance. Here, we use the external memory model [1], where any access within a 32-word block of memory counts as one access, for our analysis.[1]

We find that, not surprisingly, our B-Tree implementation outperforms the existing GPU LSM implementation by a speedup factor of 6.44x on query-only workloads. More surprisingly, despite the theoretical predictions, we find that for small- to medium-sized batch insertions (up to roughly 100k elements per insertion), our B-Tree outperforms the LSM. Why? The thread-centric design and use of bulk primitives in the LSM means in practice that it takes a large amount of work for the LSM to run at full efficiency; in contrast, our warp-centric B-Tree design hits its peak at much smaller insertion batch sizes. We believe that insertions up to this size are critical for the success of the underlying data structure: if the data structure only performs well on large batch sizes, it will be less useful as a general-purpose data structure.

Our implementation addresses three major challenges for an efficient GPU dynamic data structure: 1) achieving full utilization of global memory bandwidth, which requires reducing the required number of memory transactions, structuring accesses as coalesced, and using on-chip caches where possible; 2) full utilization of the thousands of available GPU cores, which requires eliminating or at least minimizing required communication between GPU threads and

[1] On the GPU, the external memory model corresponds to a model where a warp-wide coalesced access (to 32 contiguous words in memory) costs the same as a one-word access; this is a reasonable choice because, in practice, a GPU warp that accesses 32 random words in memory incurs 32 times as many transactions (and achieves 1/32 the bandwidth) as a warp that accesses 32 coalesced words, to first order.

branch divergence within a SIMD instruction; and 3) careful design of the data structure that both addresses the previous two challenges and simultaneously achieves both mutability and performance for queries and updates. Queries are the easier problem, since they can run independently with no need for synchronization or inter-thread communication. Updates are much more challenging because of the need for synchronization and communication.

To this list we add a fourth challenge, the most significant challenge in this work: contention. A "standard" B-Tree, implemented on a GPU, simply does not scale to thousands of concurrent threads. Our design directly targets this bottleneck with its primary focus of high concurrency. The result is a design and implementation that is a good fit for the GPU. Our contributions in this work include:

1. A GPU-friendly, cache-aware design of the B-Tree node data structure;
2. A warp-cooperative work-sharing (WCWS) strategy that achieves coalesced memory accesses, avoids branch divergence, and allows neighboring threads to run different operations (e.g., queries, insertions, and deletions); and
3. Analysis that shows that contention is a critical limiter to performance, which motivates three design decisions that allow both high performance and mutability:
   (a) A proactive splitting strategy that correctly handles node overflows while minimizing the number of latched nodes during the split operation;
   (b) Level-wise links that allow more concurrency during updates, specifically during split operations; and
   (c) Restarts on split failure to alleviate contention and avoid spinlocks.

3.2 Background and Previous Work

3.2.1 B-Tree

Key-value stores are fundamental to most branches of computing. Assuming all keys in the data structure are unique, a key-value store implements the following operations:

Insert(k, v)**:** Adds (k, v) to the set of key-value pairs (or replace the value if such key already existed).

Delete(k)**:** Removes any pair $(k, *)$ from the set.

Lookup(k)**:** Returns the pair $(k, *)$ in set, or $\perp$ if no such pair exists.

Range(k_1, k_2)**:** Returns all pairs $(k, *)$ in set, where $k_1 \leq k \leq k_2$.

Successor(k)**:** returns the pair $(k', *)$ where k' is the smallest key greater than k, or $\perp$ if no such k' exists.

When the set of key-value pairs is small, in-memory solutions such as balanced search trees are typically used. When data is too large to fit into memory—and for a GPU, when the main body of the data structure only fits into global DRAM—such data structures as B-Trees, LSMs, and B$^\epsilon$-trees are used. B-Trees are optimized for query performance. The ubiquitous B-Tree as described by Comer [20] was introduced by Bayer and McCreight [10] to handle scenarios where records exceed the size of the main memory and disk operations are required. Therefore, a B-Tree is structured in a way such that each node has a size of a disk block, intermediate nodes contain pointers and separators (*pivots*) that guide the tree traversal, and leaf nodes contain keys and records (values). For a tree of fanout B, each intermediate node in the tree can have at most B children and must have at least B/2 children, except for the root, which can have as few as two children.

During insertion into a B-Tree, a tree node is split whenever it overflows and nodes are merged to handle underflows. For a B-Tree that stores N keys, the tree will have a height of $O(\log_B N)$, which is shallower than a balanced binary tree, which has height $\Theta(\log_2 N)$. This difference in height is the basis for the difference between the I/O costs of searches in B-trees and in sorted arrays given in Table 3.1.

3.2.2 Previous Work

Splitting. A major challenge for concurrent updates on the B-Tree is splitting an overflowing tree node, where updates to the overflowing node, its new sibling (new child to the parent), and the parent are required to be done atomically. This requires locking two tree nodes on different

Work	Usage	Data structure notes
YHFL+ [65]	Grid files for multidimensional database queries.	Built on CPU.
KCSS+ [36]	Index search for databases using binary tree optimized for architecture.	Inefficient parallelism: only run one tree-building thread per half warp. Updates require complete rebuild.
FWS [23]	Processing B+ tree queries for databases.	Built on CPU.
LWL [45]	Construct R-trees by parallelizing sorting and packing stages. Tree traversal based on BFS.	GPU-built trees have poor range query performance. Updates require complete rebuild.
BGTM+ [8]	Single- and multi-GPU range queries for List of Clusters and Sparse Spatial Selection indexing approaches.	Built on CPU.
SKN [57]	Compute range queries by constructing Cartesian tree and finding least common ancestors.	Updates require complete rebuild.
KKN [37]	R-tree traversal for spatial data. Sequential search between nodes, parallel search within each node.	Built on CPU.
YZG [66]	R-tree construction and querying for geospatial data. Compares performance of trees constructed on GPU and CPU.	GPU-built trees have poor range query performance. Updates require complete rebuild.
LYWZ [43]	Range query processing for moving objects using query buffers, hashing, and matrices to calculate and track distances between objects.	Process stream of data instead of building data structure.
LSOJ [41]	Spatial range queries for moving objects using grid indexing, quad trees, and intermediate bitmap data structures.	Only works on databases with evenly distributed objects. Updates require complete rebuild.
ALFA+ [5]	First dynamic general-purpose dictionary data structure for the GPU based on the Log Structured Merge tree (LSM).	High insertion rates, but primarily for large insertions; competitive query performance.
SJ [56]	Large trees that don't fit on a GPU's memory, with emphasis on query performance. GPU is used to speed up query performance.	Built and updated on CPU.
YLPZ [64]	Phased queries and updates on the GPU.	State-of-the art query throughput. Less efficient update throughput.

Table 3.2: Chronological summary of previous work on dictionary data structures that support point and range query on GPUs.

levels (the new sibling doesn't need to be locked as no pointers to it exist yet), which bottlenecks the updating process, particularly at the root and upper tree nodes. Moreover, splitting could propagate up the tree (when the parent node is full), thus requiring locking more nodes on different levels.

Graefe [26] surveyed the different locking techniques that are typically used on CPUs. Latch coupling and B-link-trees are two different approaches to maintain consistency of the B-Tree during split operations without causing concurrency bottlenecks. In latch coupling, a thread releases a node's latch only after it acquires the next node's latch. For splitting with a latch coupling strategy, in addition to latching the next node, the parent node is unlatched only if the lower node is not full, guaranteeing that subsequent split operations will successfully complete.

Another approach to splitting is to proactively split nodes during a thread's root-to-leaf traversal. Proactive splitting avoids concurrency bottlenecks but may lead to unnecessary splits, and it may be challenging to extend it to variable-length records.

The B-link-tree [40] relaxes the constraints of a B-Tree and divides the split operation into two steps: splitting the node and updating the parent. In between these two steps the B-Tree is in an intermediate tree state where the parent doesn't have information about the new node but the split node and its new sibling are linked. Linking nodes requires the addition of a high key and a pointer in recently split nodes to their neighbor nodes, and during traversals, threads are required to check the high key at each node to determine if level-wise traversal is required. Good performance requires that updating the parent with a pointer to its new child should be done quickly to avoid traversing long linked lists and to improve traversal performance.

Early lock releasing techniques were used by Lehman and Yao [40] to provide more concurrency. The merging of nodes to reduce the tree height after deletions was presented by Lanin and Shasha [39] and Sagiv [54]. Latch coupling was used in B-link-trees by Jaluta et al. [32] along with recovery techniques.

GPU work. While many previous GPU projects have targeted B-Trees and similar data structures that support the same operations (Table 3.2), few support incremental updates, those that do typically have poor update rates, and many cannot even build the B-Tree on the GPU. No previous work has competitive performance on both queries and updates. Fix et al. [23] was

among the first to build a GPU B-Tree but only used the GPU to accelerate searches. The work most directly on point is from Kaczmarski [34], who specifically targets the bulk-update problem with a combined CPU-GPU approach that contains optimizations beyond rebuilding the entire data structures; Huang et al. [30], who extend Kaczmarski's work but with non-clustered indexes that would be poorly suited for range queries; and Shahvarani and Jacobsen [56], who focus their work on high query rates using large fanouts, but with poor insertion performance. Their work proposed a hybrid CPU-GPU B-Tree to handle scenarios where the tree size exceeds the GPU memory size. They focus on high search throughput using GPUs; insertions are done in parallel on the CPU. Our work does not target B-Trees larger than the GPU memory capacity. In concurrent work, Yan et al. [64] propose a novel B-Tree structure where the tree is divided into key and child regions. The key region contains keys of the regular B-Tree laid out in memory in a breadth-first order. The child region is a prefix-sum array of each node's first child (which is small enough to fit inside the cache). Moreover, they offer two optimizations: partial sorting of queries to achieve coalesced memory access, and grouping of queries while reducing the number of useless comparisons within a warp to minimize the warp execution time. With these design decisions they achieve state-of-the-art query performance at the expense of a higher cost to maintain the B-Tree structure when updating. Our work offers a different tradeoff between query and update performance.

The GPU LSM [5] takes a different approach to provide a dynamic GPU data structure that supports the same operations as the B-Tree. The GPU LSM is a hierarchy of dictionaries, each with a capacity of $b2^i$, where i represents the level and b represents the batch size. It derives from the Cache Oblivious Lookahead Array (COLA), where each dictionary is represented using a sorted array of elements, with updates modifying the small dictionary. Once a dictionary reaches its capacity, it is merged with the next larger one. Updates are done using two primitives, sort and merge, each of which can be done efficiently on GPUs. For queries, the search starts at the smallest dictionary and proceeds along the hierarchy of dictionaries. GPU LSM performance generally depends on the batch size, where larger batch sizes improve the performance.

We compare our performance to the GPU LSM and GPU sorted array performance in Sec-

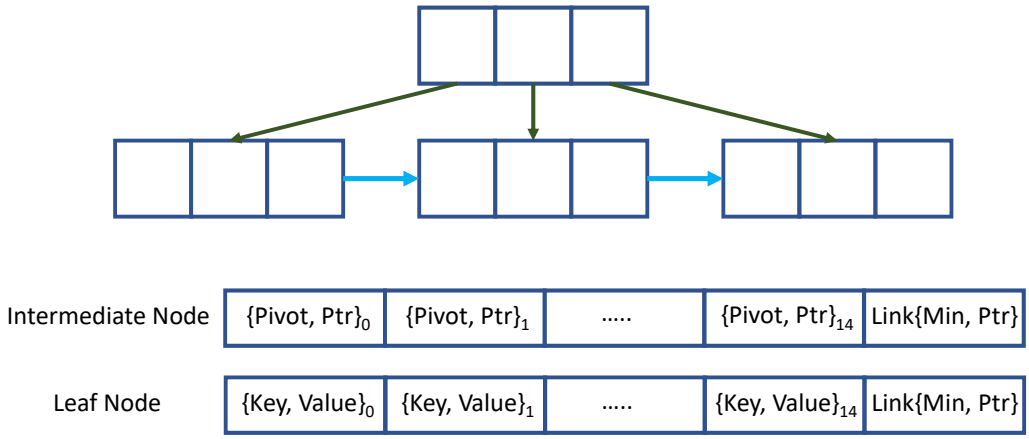


Figure 3.1: B-Link-Tree (with $B = 3$) schematic (top). Our B-Tree (with $B = 15$) node structure (bottom). A tree node contains 15 pivot-pointer (or key-value) pairs. A pointer to the node's child is represented by the child's offset. The last pair in a node represents the right sibling minimum value and its pointer. The minimum of the right sibling serves as a high key for the node.

tion 3.5.

3.3 Design Decisions

In our design we assume 32-bit keys, values, pivots (separators), and offsets (pointers). We use the most significant bit of each of the node's entries to distinguish leaves from intermediate nodes and to mark locked (*latched*) nodes. Figure 3.1 shows a schematic of our B-Tree's node structure. Offsets are used to identify the next tree node during traversal by simply multiplying the offset by the size of the tree node.

We use the same structure for internal B-Tree nodes and leaves. All key-value pairs are stored in leaf nodes; internal nodes store pivot-offset pairs, where the offset points to another node in the tree. Each node in our B-Tree stores 15 key-value or key-offset pairs (Section 3.3.1), and an additional pair containing a pointer to its right sibling and the minimum key of its right sibling (Section 3.3.2).

Reading a tree node is not blocked by any other operation (Section 3.3.3). When we insert into the tree and must split, we use proactive splitting (Sections 3.2.2 and 3.3.4) with restarts on failures (Section 3.3.5). We use a simple write latch per node to synchronize concurrent modifications to the same node. When an insertion into a node causes a split, we first move half of

the leaf (or intermediate) node's key-values (or pivot-offsets) to a new node, then insert a pivot-offset pair into a parent node. We use a warp-cooperative work-sharing strategy (Section 3.3.6) where work is generated per thread but performed per warp.

The remainder of this section discusses the details, motivations, and implications of these design decisions.

3.3.1 Choice of B

To maximize memory throughput, each of our B-Tree nodes is the size of a cache line, which is 128 bytes on NVIDIA GPUs. Thus a warp of 32 threads can read a tree node (cache line) in a coalesced manner. Each tree level is a linked list (we motivate this decision in Section 3.3.2) to allow for more concurrency, specifically during insertion. The overhead for making each tree level a linked list is 8 bytes divided equally between a pointer to the node's right sibling and the right sibling's minimum key. The remaining 120 bytes are used to store either pivot-pointer pairs for intermediate nodes or key-value pairs for leaf nodes; therefore our B-Tree has $B = 15$. Figure 3.1 illustrates the tree node structure.

3.3.2 B-Link-Tree

Adding new items to a B-Tree may require splitting a node, which in turn requires changing nodes on at least two levels of the tree. Traditional implementations exclusively lock a safe path during an insertion traversal. On a GPU, such locks rapidly bottleneck any tree traversal, particularly at the root and upper tree nodes. We eliminate the need for an exclusive lock, allowing other warps to concurrently read, by adopting the side-link strategy of the B-Link tree [40]. In a B-Link tree, each node stores a link to its right neighbor as well as storing the right neighbor's minimum key, i.e., each tree level is a linked list. With this additional information, we no longer must lock the upper node in the split. Why?

We traditionally exclusively lock (at least) both the upper and lower node to handle the case where a split operation and a read operation are concurrent. The split divides the lower node into two nodes then updates the parent node with the information about the new node. If a read occurs after the lower-node division but before the upper-node update, it may not find the right path down the tree. However, the side link solves this problem: if the read occurs after the

lower-node division and the item is not in the left lower node, the read operation traverses the side link to find the new right lower node.

Maintaining the level-wise links is simple. During a split operation, the right tree node gets the side-link data from the original node, and the left node's side link points to the right node and also stores the minimum key or separator of the right node.

In our GPU implementation, the addition of the side link itself does not solve the concurrency problem, but together with a proactive splitting strategy (Section 3.3.4), it improves concurrency.

3.3.3 Decoupled Read and Write Modes

A complementary decision to the previous one is to decouple reads and writes. In other words, our design has only one latch type: an exclusive write latch, only required when modifying a node's content, during inserting or deleting a key-value or separator-offset pair. Any warp starts the tree traversal for any update operation in read mode; reads require no latches. But once a warp decides to switch from read mode to write mode, an additional read is required after latching the node. The additional read is required to ensure we have the most recent node content as other warps might have subsequently modified the contents of the node.

3.3.4 Proactive Splitting

Splitting a tree node is required whenever the node becomes full, and in the most extreme case the splitting process will propagate up the tree all the way to its root. The traditional approach to splitting is latch coupling, which involves exclusively locking a subtree starting at a "safe" node that guarantees that any future splits will not propagate further up the tree. Latch coupling disallows both reads and writes in this subtree. This strategy significantly bottlenecks GPU performance by limiting concurrency; exclusively locking (or even write-only locking) an entire subtree idles any thread that accesses (or modifies) that subtree. This loss of concurrency results in unacceptably low performance.

Instead we use a proactive splitting strategy. Proactive splitting, together with the side links of a B-Link-Tree (Section 3.3.2), maximizes concurrency: with them, we both limit node modifications to only two tree levels and also allow concurrent reads of these nodes.

During insertions, a node is split whenever it is full. We begin by reading a node; if that node is full, we begin the splitting process. To further reduce the time we need to latch the upper tree node, we process the first splitting stage without latching the upper node. But, before committing the changes to the split node and its new sibling, we must latch the parent. Then we check if the parent, which will now gain an additional child, has subsequently become full (the high level of concurrency makes this a distinct possibility). We handle this case with a restart, as we describe in the next subsection. It is the combination of side links, proactive splitting, and restarts that together allow our implementation to achieve high levels of concurrency.

3.3.5 Restarts Instead of Spinlocks

Traditionally, threads in a B-Tree that encounter a locked latch spin until that latch is available (a spinlock). GPU software transactional memory techniques [62, 63] provide the same functionality of fine-grained synchronization, but we opt for lightweight latches embedded in our B-Tree's nodes. We further tune our synchronization technique using the fact that only writes requires latches (Sections 3.3.2 and 3.3.3). Moreover, in our design, we generally replace spinlocks with restarting the operation from the node's last-known parent or the root. The restart has a similar effect to backoff locking [58], where a spinlocking thread does meaningless work to temporarily relieve contention over the atomic unit; this is useful when DRAM operations are not slow and atomic operations are fast so that the backoff window is small. ElTantawy and Aamodt [22] showed that an adaptive backoff improves the performance even further, since small backoff delay may increase spinning overheads while a large backoff delay may throttle warps more than necessary. From our experiments we find that spinlocks on high-contention nodes—specifically, full and leaf nodes during insertions—reduce the amount of resident warps that can make progress. Moreover, restarts improve memory throughput and insertion rates. For a B-Tree of size 2^{16}, we find that restarts improve the throughput by a factor of 6.39x over spinlocks, while backoff improves the performance by a factor of only 1.47x.

We use spinlocks in three cases: (1) during the second stage of splitting a node that modifies the node's parent, (2) during traversal of side links (after latching a leaf node), and (3) during the deletion of key-value pair from a leaf node. More commonly, we restart traversal. We restart from the node's last-known parent if we fail to latch a leaf node or a full leaf (or intermediate)

node. Another scenario for restarting from the last-known parent node is when we detect that the last-known parent is not the true parent, as the true parent might be the new sibling of the last-known parent after splitting. After restarting with the last-known parent as the current node, we find the true parent using side-link traversal. We restart from the root if the split operation requires information that is unknown. Since we do not keep track of the grandparent node, the unknown information is either 1) the grandparent node when the parent node is full or 2) the parent node when the current node became full after a restart to detect the true parent. We find that restart overhead becomes less significant as the tree size grows and that restarts increase our insertion throughput. We note that using a spinlock, specifically when latching a parent node during splitting of its child, guarantees that at least one warp will make progress.

3.3.6 Warp Cooperative Work Sharing Strategy

We expect that the predominant use of our B-Tree will be in scenarios where the GPU is running many threads and each thread potentially generates a single access (a query, an insert, or a delete) into the B-Tree. Consequently, our abstraction supports inputting work from threads. However, we *process* work with entire warps in an approach first proposed for dynamic GPU hash tables [4]. In the common case, 32 threads in a warp each have an individual piece of work, but the entire warp serializes those 32 pieces of work in a queue, working on one at a time. This strategy has two clear benefits: avoiding thread divergence within a warp and achieving coalesced memory accesses while reading or writing a tree node. A third benefit is alleviating the need for load balancing. Although the path from the root of the tree to the leaves in a B-Tree is a uniform one, the insertion process will be an irregular task based on the thread's path. In particular, the irregularity comes from the additional process of node splitting. Because WCWS leverages the entire warp to do these irregular tasks, it avoids any need to load-balance work across threads.

3.4 Implementation

With the exception of a bulk-build scenario, all of our implementations follow the warp cooperative work sharing strategy (WCWS). In WCWS each thread has its own assignment, either an update (insertion or deletion) or a query (lookup, range, or successor). A warp cooperates on

performing each of its 32 threads' tasks using warp-wide instructions. With our design decision for B, each thread in the warp reads one item in the tree node. Even-lane threads read keys (or pivots), and odd-lane threads read values (or offsets); the last two threads read the node's high key and its right-sibling offset.

In all of our operations, we leverage CUDA's intrawarp communication instructions in two ways. (1) `ballot` performs a reduction-and-broadcast operation over a predicate. The predicate is usually a comparison between a key (or a pivot) and each thread's key. `ballot` is always followed by a `ffs` instruction (i.e., find first set bit) to determine the first lane that satisfies the `ballot` predicate. (2) `shfl` ("shuffle") broadcasts a variable to all threads in a warp.

Algorithm 3.1 shows the general pattern in a warp cooperative work sharing algorithm, which we use as the entry point in our simultaneous query and update algorithm. We now discuss the implementations of the various operations that we support, omitting intrawarp communication details.

Algorithm 3.1 Warp cooperative work sharing strategy.

```
1:  procedure WCWS(Tree btree, Pair pairs, Task tasks)
2:      is_active ← true
3:      thread_pair ← pairs[threadIdx]
4:      thread_task ← tasks[threadIdx]
5:      while work_queue ← ballot(is_active) do
6:          current_lane ← ffs(work_queue)
7:          current_pair ← shfl(thread_pair, current_lane)
8:          current_task ← shfl(thread_task, current_lane)
9:          performTask(current_task, current_pair, btree)
10:         if laneId = current_lane then
11:             is_active ← false
12:         end if
13:     end while
14: end procedure
```

3.4.1 Bulk-Build

The bulk build operation constructs a B-Tree directly from a bulk input of key-value pairs. We start by sorting the input pairs with CUB's [48] sort-by-key primitive. Then we start building the tree bottom-up. To avoid splitting after a bulk-build process, we fill each of the tree nodes

with only 8 pairs of either key-values or pivot-offset. We reserve the zeroth node as the root. The remainder of the tree nodes are organized in a left-to-right level-wise order starting from the leaf nodes. We assign each tree node to a warp. Each warp is only responsible for loading the required 8 key-value pairs if the node is a leaf. Since we already know the structure of the tree, we can easily determine the current node height and the indices of its children for intermediate nodes. We also avoid the complexity of merging nodes that are underfull and allow underfull nodes to exist in the constructed tree.

3.4.2 Incremental Insertion

In incremental insertion, a thread has a new key-value pair that must be added to the appropriate leaf node. This operation requires tree traversal and split operations when needed. Algorithm 3.2 summarizes the incremental insertion algorithm. A warp traverses the tree starting from the root (line 2). The most significant bit in any node's first entry identifies whether it is a leaf or an intermediate node. If we reach a leaf or a full node, then the current node must be modified; we attempt to latch it (line 13). As we detailed in Section 3.3.5, if we cannot acquire the lock, we restart the insertion process from the node's parent instead of spinning (line 15).

Latches. Each tree node has a one-bit lock (the most significant bit in the second node entry), which we try to change using an `atomicOr`. Out of a warp's 32 threads, only the second thread acquires the latch for the warp. If the `atomicOr` function returns a value where the most significant bit is one, then the latch failed. A zero indicates that we successfully latched the node. Due to the weak memory behavior on a GPU, latching a node using only an atomic call guarantees serialization over the latch, but not the tree nodes themselves. Load and store instructions could be reordered around the atomic call. Therefore, we must add a global memory fence both after acquiring a latch and before releasing a latch. This fence guarantees that all writes to global memory before the fence are observed by all other threads before the fence. We also must use the `volatile` keyword to bypass the L1 cache to avoid reading stale tree nodes from the L1 cache. The memory fences and the L1 cache bypass degrade performance, but are necessary to ensure correctness. For example, building a B-Tree that contains 2^{16} keys is on average 1.77x faster, averaged over successful runs, if memory fences and the L1 cache bypass are not used. All reported results for insertions in Section 3.5 use both memory fences and a L1

Algorithm 3.2 Incremental insertion.

```
 1: procedure INSERT(Tree btree, Pair pair)
 2:     current ← parent ← btree.root
 3:     repeat
 4:         while pair.key ≥ current.link_min do
 5:             current ← current.link_ptr
 6:         end while
 7:         if current is full then
 8:             if current = parent and current is not root then
 9:                 current ← parent ← btree.root
10:             end if
11:         end if
12:         if current is full or current is leaf then
13:             if tryLatch(current) = failed then
14:                 current ← parent
15:                 continue
16:             end if
17:             link_used ← false
18:             while pair.key ≥ current.link_min do
19:                 if current is full then
20:                     releaseLatch(current)
21:                     link_used ← true
22:                     current ← parent
23:                     break
24:                 end if
25:                 releaseLatch(current)
26:                 current ← current.link_ptr
27:                 acquireLatch(current)
28:             end while
29:             if link_used then
30:                 continue
31:             end if
32:         end if
33:         if current is full then
34:             result ← trySplitAndUpdateParent(current, parent)
35:             if result = success and current is not leaf then
36:                 releaseLatch(current)
37:             else if result = parent full or unknown then
38:                 releaseLatch(current)
39:                 current ← parent
40:                 continue
41:             end if
42:         end if
43:         if current is leaf then
44:             insertPair(pair, current)
45:             releaseLatch(current)
46:         else if current is intermediate then
47:             current ← getNext(pair.key, current)
48:         end if
49:     until current is leaf
50: end procedure
```

cache bypass.

Using side links. After we read a node (line 4), and after we latch it (if it is a leaf or a full node) (line 18), we check if the key is less than the node's high key; this is the usual case. However, the key may now be larger than the high key; for instance, another insert may split the current node after the read but before the latch. In these cases, we traverse to the next node on this level using the side link. Using a `shfl` instruction, we broadcast the right sibling node offset to all threads in the warp and continue the insertion process from this node. In case when the node is full and the side link is used, we restart the process from the last-known parent (line 30).

Splitting. If the latched node is full, and we never traversed side links (i.e., we know the parent node), we begin the splitting process (line 34). We perform the first stage of the split without latching the parent or creating the new node. We prepare new pairs for the now-half-full node and its new sibling. Then we latch the parent and check if the parent is the current true parent of the node. It may not be if another warp has subsequently split the last-known parent and the new true parent is the new sibling; if that is the case, we restart the process from the last-known parent (line 40). If we detect that the parent is full, we restart the process from the root of the tree (line 9). If the splitting succeeds, we detect which of the new nodes is our next node and move to that node.

Inserting the new pair. If the node is a leaf node, we move pairs in the node to create space for the new pair, then write the node changes back to memory (line 44).

3.4.3 Search

Searching the tree for a value (Algorithm 3.3) is much simpler than insertion. A warp simply traverses the tree by comparing the lookup key and the intermediate-node pivots using a warp-wide comparison. The warp then determines the lane that contains the next pivot and hops to the next node. Once the warp reaches the leaf node, a second warp-wide comparison of the key and the leaf node keys determines if the key exists in the tree (in which case the associated value is returned), or if the key doesn't exist in the tree.

Algorithm 3.3 Lookup, range, successor, and delete.

```
1: procedure QUERYORDELETE(Tree btree, Key key, Key key_upper_bound, Result result,
   Operation operation)
2:     current ← parent ← btree.root
3:     result ← NOT_FOUND
4:     repeat
5:         if current is intermediate then
6:             current ← getNext(key, current)
7:         else if current is leaf then
8:             switch operation do
9:                 case lookup:
10:                    result ← getValue(key, value)
11:                    break
12:                case delete:
13:                    latchNode(current)
14:                    volatileReadNode(current)
15:                    current ← deleteKey(key, current)
16:                    volatileWriteNode(current)
17:                    break
18:                case range:
19:                    while true do
20:                        result += inRange(key, key_upper_bound, current)
21:                        if key_upper_bound < current.link_min then
22:                            break
23:                        end if
24:                        current ← current.link_ptr
25:                    end while
26:                    break
27:                case successor:
28:                    while result = NOT_FOUND do
29:                        result ← getNextValidPair(key, current)
30:                        current ← current.link_ptr
31:                    end while
32:                    break
33:            end switch
34:        end if
35:    until current is leaf
36: end procedure
```

3.4.4 Deletion

In deletion, a warp first traverses the tree to find the deleted key. Once it reaches the leaf, it latches the leaf node and reads it again, since between the time of traversal and latching, other warps might have deleted keys from the node. Once a warp latches the leaf node, a warp-wide comparison locates the key. The deleting warp shuffles down higher keys and their associated values, if any, two spots to overwrite the deleted key-value pair. Similar to insertion, memory fences are required for latching, but since in our deletion we do not modify intermediate nodes, we can avoid using the keyword `volatile` and take advantage of the L1 cache when reading intermediate nodes. But for reads and writes to leaf nodes, we use custom PTX read (`ld.global.relaxed.sys.u32`) and write (`st.global.relaxed.sys.u32`) functions to bypass the L1 cache. We avoid merging underfull tree nodes, as it slows down the deletion process without a corresponding gain in search performance. A high-level description of the algorithm is shown in Algorithm 3.3.

3.4.5 Range Query

Given a pair of upper/lower bounds, a warp first traverses the tree searching for the location of the lower bound. Once the location is determined, the warp uses the side links to perform level-wise traversals until it locates the upper-bound key. During this side traversal, all key-value pairs belonging to the range are written back to global memory. The counter that keeps track of the pairs within the range could be used to provide a count query, which is faster since no global memory writes are required. The range query (or count) algorithm is similar to the point query algorithm with the lookup key as the lower bound, with the addition of both link traversal and writing back the in-range pairs (or the count). The amount of work required to perform a Range(k_1, k_2) is directly dependent on the range length (i.e., $k_2 - k_1$). A high-level description of the algorithm is shown in Algorithm 3.3.

3.4.6 Successor Query

Given a key, to find its successor we first perform a point query to locate the key. Then we check if any larger key exists in the current leaf. If the key was the last valid key in the node, we perform level-wise traversals using side links to find the first valid key. Since in deletion we

do not merge tree nodes, the warp might need to perform more than one traversal. A high-level description of the algorithm is shown in Algorithm 3.3.

3.5 Results

In this section we compare our B-Tree implementation[2] to a GPU sorted array (GPU SA) and a GPU LSM. GPU LSM and GPU SA implementations are from Ashkiani et al. [5]. The GPU LSM implementation uses CUB [48] in its sort primitive and moderngpu[3] in its merge primitive. We run all of our experiments on an NVIDIA TITAN V (Volta) GPU with 12 GB DRAM and an Intel Xeon CPU E5-2637.

For all of our experiments we used 32-bit keys and values. We reserved the most significant bit of keys for locking and identifying leaves and intermediate nodes.

At a high level, all B-Tree operations have throughput proportional to the height of the tree. Because of the large fanout of a B-Tree, this means that for most B-Tree sizes of interest (large enough to make a B-Tree worthwhile at all, small enough to fit into GPU memory), the B-Tree's height is constant and we thus essentially have constant throughput. This makes the B-Tree's performance much more predictable than the LSM (e.g., Figure 3.2).

For rates or throughputs, all "mean" or "average" results in this section are harmonic means.

3.5.1 Insertion

Baseline B-Tree. Our baseline B-Tree implementation is most similar to the B-Tree design of Rodeh [53]. In the baseline implementation we used latch coupling and a proactive splitting strategy. The baseline B-Tree branching factor was 16. As discussed in Section 3.3.2, with the GPU's high level of concurrency, latch coupling will severely bottleneck any tree traversal. We see the effect of using latch coupling and its exclusive latches in the resulting insertion throughput of 0.166 MKey/s. Our design decisions allow us to make much better use of the thousands of active warps on the GPU, achieving an average insertion throughput of 182.9 MKey/s, more than three orders of magnitude greater than the baseline.

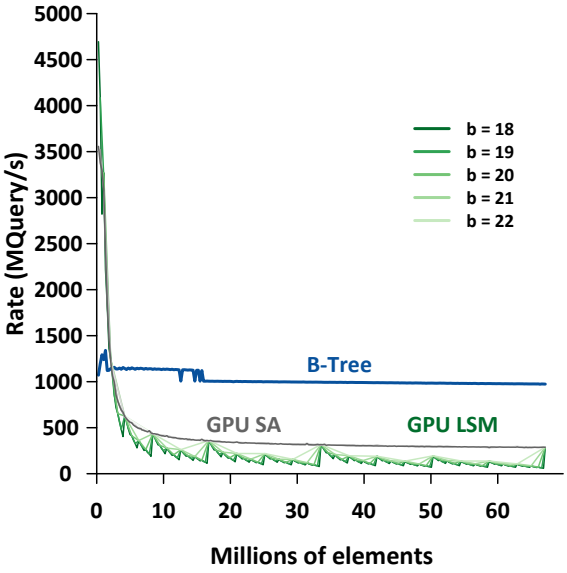
[2]Our implementation is available at `https://github.com/owensgroup/GpuBTree`.
[3]Moderngpu is available at `https://github.com/moderngpu/moderngpu`.

Bulk-build vs. incremental update. We investigate the advantage of incremental update over complete rebuild of the B-Tree. Figure 3.3 compares the time required to bulk-build a B-Tree of size m from scratch vs. inserting a batch of size 2^i into a B-Tree of size $m - 2^i$. As the batch size decreases, we see the advantage of incremental insertion over bulk-rebuild. For example, once the tree size reaches 3.15 million keys, inserting a batch of 2^{18} (262k) elements into the tree has a clear advantage over rebuilding the tree. As the batch size gets larger, the tree size at which updating the tree is more efficient than rebuilding the tree from scratch grows, which is expected since a bulk-build only requires a sort (which is done efficiently on the GPU) and writing the tree nodes. We note that the throughput of bulk-build is on average 3124.32 MKey/s.

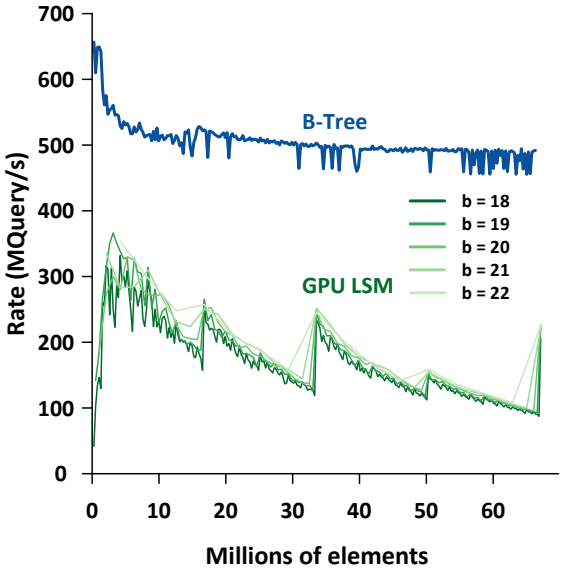
Incremental updates. To evaluate batched incremental updates for B-Tree, GPU LSM, or GPU SA we build all possible data structure sizes incrementally using batches of size b. The mean of all insertion rates for a given b is reported in Table 3.3. For smaller batch sizes $b \leq 2^{17}$ we find that although a GPU LSM is optimized for insertions and should be theoretically faster than a B-Tree, our B-Tree is faster with a speedup factor of 2.73x and 1.15x for $b = 2^{16}$ and $b = 2^{17}$ respectively. Why? The GPU LSM uses sort and merge primitives that perform better for large bulk inputs. On the other hand, our B-Tree uses a warp-centric approach that allows us to reach higher performance for smaller batches. Similarly, GPU SA reaches almost the same throughput as our B-Tree when using a batch size of $b = 2^{18}$. Our B-Tree is $\{$3.74x, 1.59x$\}$ faster than the GPU SA for batch sizes of $\{2^{16}, 2^{17}\}$ respectively. As theory predicts, as the batch size increases, GPU LSM and GPU SA start to outperform our B-Tree, reaching speedup factors of 2.12x and 1.54x for a batch size of $b = 2^{19}$ and speedup factors of 7.19x and 6.6x for a batch size of 2^{22}. We note that for batch sizes of $b = 2^{19}$ and $b = 2^{22}$, if the B-Tree size exceeds 6.82 and 57.67 million keys respectively, an entire rebuild for the B-Tree will be the right choice to handle the update. A bulk rebuild of $\{$6.82, 57.67$\}$ MKeys trees takes $\{$2.25, 17.11$\}$ ms, yielding an effective insertion throughput of $\{$116.16, 245.17$\}$ MKey/s for batch sizes of $\{2^{19}, 2^{22}\}$.

3.5.2 Search

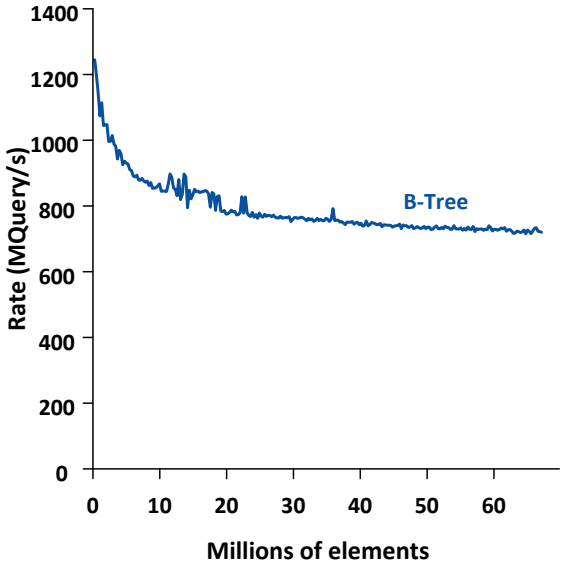
Search is where our B-Tree shows large improvements over GPU LSM and GPU SA. Our B-Tree throughput is almost constant over a wide range of tree sizes. Figure 3.2a shows the



(a) Point query rate.



(b) Range query rate.



(c) Successor query rate.

Figure 3.2: Search, range, and successor query rates for different batch size operations applied to the GPU LSM, the GPU SA, and our B-Tree. In each query we search for all keys existing in the tree. Point query throughput for the B-Tree is a function of its height, which makes its throughput constant over a large range of tree sizes. A tree of height $= 8$ starts when the number of keys is $\approx$ 18M all the way up to $\approx$ 200M. For the range query, the expected range length is 8. On average, our B-Tree is 6.44x and 3x faster than GPU LSM, and GPU SA, respectively in search queries, and 3x faster than GPU LSM in range query.

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

Table 3.3: Mean rates (in MKey/s) for different batch-sized insertions into the B-Tree, GPU LSM and GPU SA.

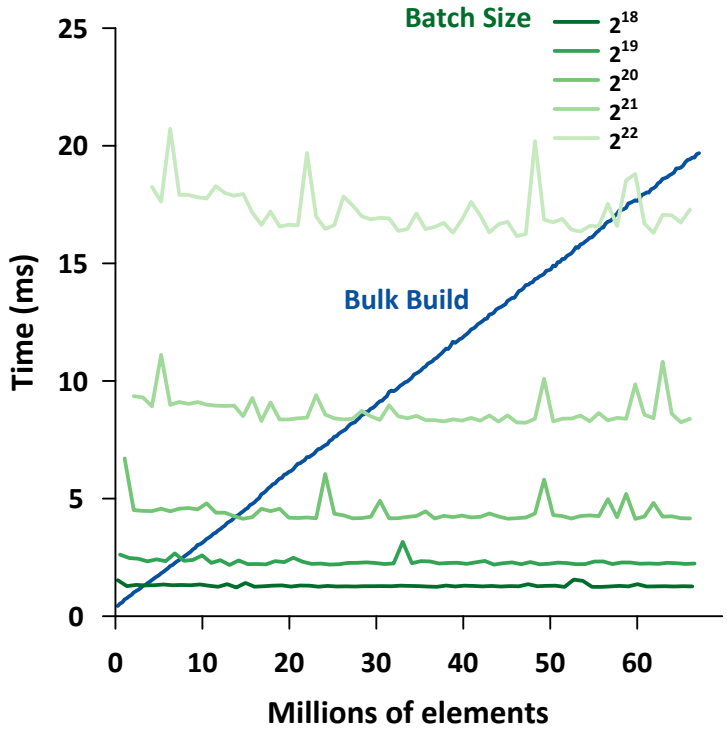


Figure 3.3: Crossover points between bulk-rebuild of the B-Tree (including sort time) and inserting a batch of size 2^i that result in a tree with the same number of elements.

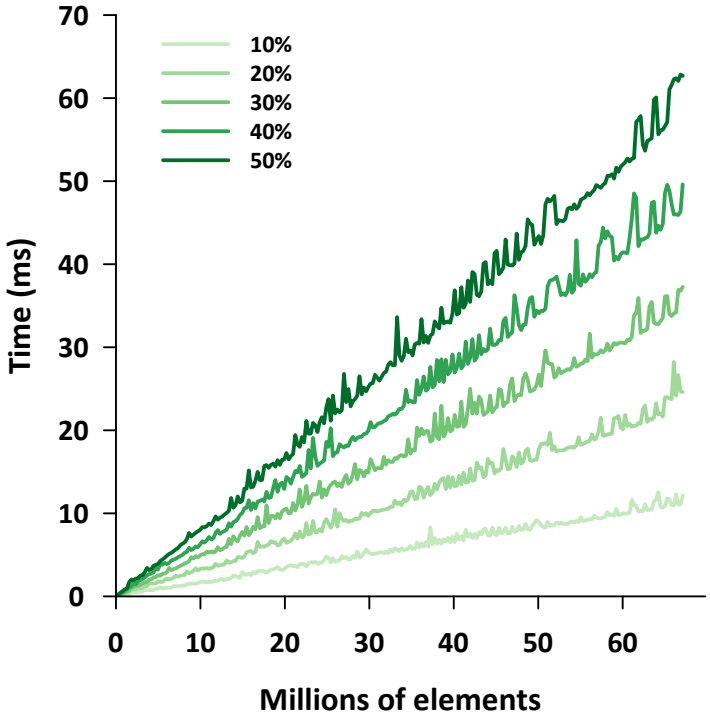


Figure 3.4: Deletion time for different percentages of the number of key-value pairs in the tree.

throughput of search queries for trees with different sizes. For GPU LSM and GPU SA, we run the same experiments as we did for the updates, where we construct the data structure using different batch sizes for different sizes. In all of the experiments we search for all elements in the data structure. The average search throughputs for the {B-Tree, GPU LSM, GPU SA} are {1020.27, 158.44, 335.17} MQuery/s respectively.

3.5.3 Deletion

Given a B-Tree of size m, we measure the time required to delete $x\%$ of the key-value pairs in the data structure. In practice, deletion is essentially a tree traversal with an additional write-back. We present the results in Figure 3.4 for deletion percentages between 10% and 50%. Throughput for a deletion percentage of 10% is 570.64 MDeletion/s. For the remaining deletion percentages, throughput is between 581.78 and 583.35 MDeletion/s. Taking advantage of the L1 cache for intermediate nodes (Section 3.4.4) speeds up deletion rates by a factor of 2.4x.

3.5.4 Range Query

Figure 3.2b shows the throughput of a B-Tree range query and a GPU LSM one; we see similar trends as in other search queries. We performed a range query with an expected range length of 8. GPU LSM results are generated for different batch sizes. Our B-Tree's average range query has roughly three times the throughput as the GPU LSM's (502.28 MQuery/s vs. 166.02 MQuery/s).

3.5.5 Successor Query

For successor queries, we benchmarked different sized B-Trees. In each tree, we searched for the successor of each key in the tree. The average throughput for a successor query is 783.13 MQuery/s. The GPU LSM does not currently implement this operation, although the LSM data structure is well-suited to support it.

3.5.6 Concurrent Benchmark

Benchmark setup. To evaluate concurrent updates and queries we define an update ratio α, where $0 \leq \alpha \leq 1$, such that we perform α updates and $1 - \alpha$ queries. For any given α we divide the update and query ratios equally between the different supported update and query operations. We start our benchmark on a tree of size n, and when deletion and insertion ratios are equal, the tree size remains the same for each experiment. For simplicity, we perform the same number of operations as the tree size. We randomly assign each thread an operation to perform.

Semantics. We support concurrent operations, which guarantees that all pre-existing keys in the tree will be included in the results of the batch of operations, as long as they are not updated within the batch. However, results of operations on keys that are updated within the batch will be dependent on the hardware scheduling of blocks and switching between warps. For instance, a batch may contain an insert, a delete, and a query of a key that is already stored in the data structure. All three of these operations will complete but the order in which they will complete is undefined. Many applications may choose to address this with phased operations, where changes to the data structure (insertions, deletions) are in different batches than queries into it. Strictly serial semantics, however, are incompatible with our implementation of the B-Tree.

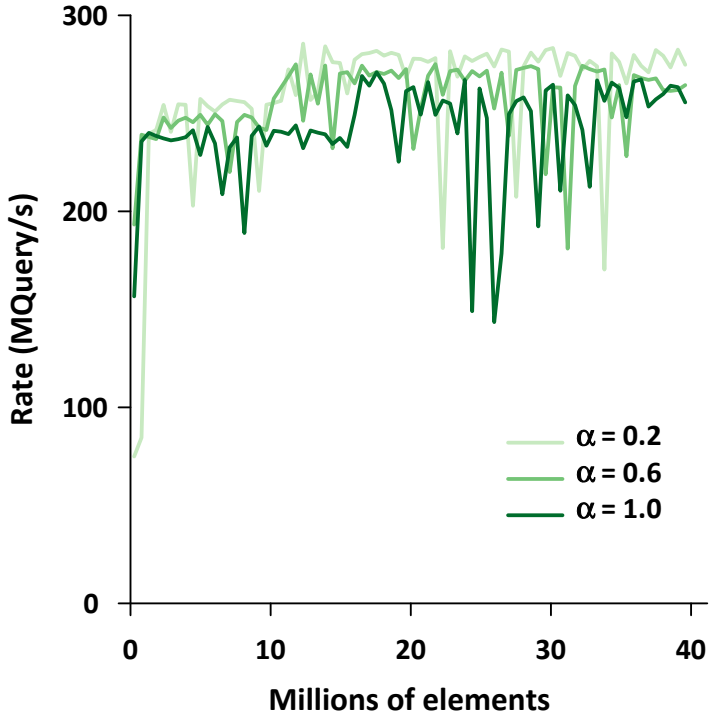


Figure 3.5: Concurrent benchmark operations rates for different B-Tree sizes.

Results. Figure 3.5 shows the results for this benchmark. We note that for correctness, bypassing the L1 cache is required for all of the operations for this benchmark, which reduces the achieved throughput compared to the phased-query operations of Figure 3.2. Moreover, additional costs for concurrent operations are: 1) intrawarp communications to determine the inputs for each of the different operations, and 2) maintenance of a work-queue (using an extra intrawarp communication) to track the progress of each of the different operations. Since all of the B-Tree operations are a function of only the tree height, performance is similar for different α ratios $\{0.2, 0.6, 1.0\}$, which achieve an average throughput of $\{247.67, 257.25, 237.79\}$ MOp/s respectively.

3.5.7 Cache Utilization

Because of the importance of caching in our results, we contrast the memory systems in the Volta and Kepler GPU architectures, whose characteristics are summarized in Table 2.1. We profiled our point query kernel on a TITAN V GPU and a TESLA K40c GPU. Figure 3.6 plots different memory hierarchy levels' throughput and hit rates. A 2.6x-larger L1 data cache

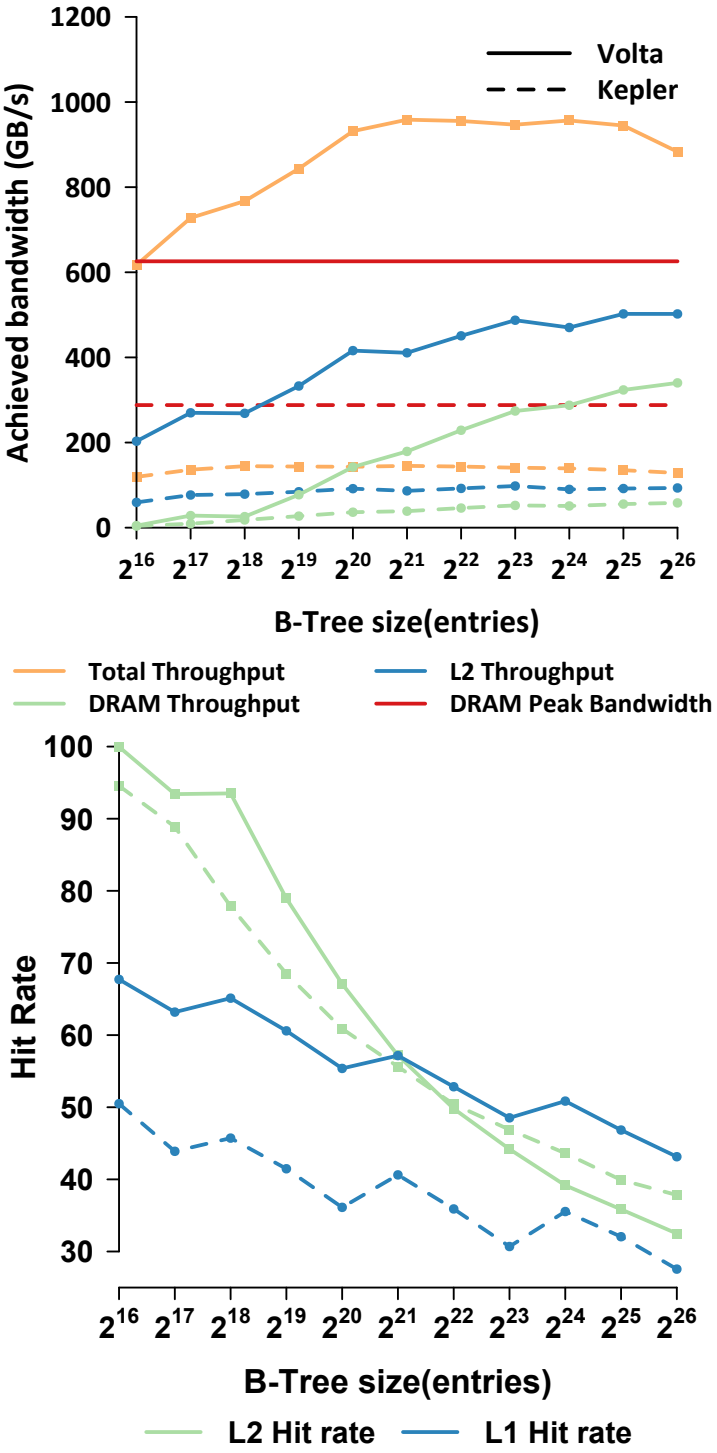


Figure 3.6: Throughput (top) and hit rates (bottom) for the different memory hierarchy levels during search queries. Upper-level tree nodes of the B-Tree are cached throughput the memory hierarchy, thus achieving high hit rates in L1 and L2 caches, and allowing the total throughput of our B-Tree to exceed the peak DRAM bandwidth on Volta.

on Volta improves the hit rate by an average factor of 1.47, which in turn improves the total memory throughput and allows it to even exceed the DRAM peak bandwidth. On average, for search queries, Volta's L2 cache throughput is 4.5x faster than the K40c, achieved DRAM throughput is 4x faster, and total throughput is 6.2x faster, for a memory system whose DRAM has only 2.27x the peak throughput.

3.6 Conclusion and Future Work

The focus of this work is not the design of a novel data structure for GPUs. Instead, we show how careful design decisions with respect to a classic B-Tree data structure allow the B-Tree to support high-performance queries, insertions, and deletions on the GPU. While memory and computational efficiency are important aspects of our implementation, the principle reason for our high performance is a design that is focused on achieving maximum concurrency by reducing or eliminating contention.

Since all nodes have size of at least 128 bytes, by using 31-bit offsets we can theoretically support up to 2^{38} bytes of storage (much larger than current GPU memories). However, limiting keys to 31 bits can be a restricting factor for some (where larger keys are required). In the future, we will focus on allowing wider key spans, either by separating the lock-bit from the rest of the key (sacrifices performance), or through a hierarchical structure and grouping a set of elements together so that they share the same key (e.g., like in quotient filters [12] or lifted B-trees [67]).

Relaxing the B-Tree invariants and introducing side-links to allow decoupled read and write modes enabled us to offer a design that scales with thousands of threads on the GPU. However, one drawback of such a design is that our B-Tree does not support linearizable multipoint queries (e.g., range query) while concurrently updating the tree. Informally speaking, a linearizable data structure guarantees that the results of concurrent operations match one sequential execution of these operations. Linearizability enables the data structure users to reason about the results of concurrent operations by providing intuitive semantics.

Suppose we can modify the data structure to guarantee that concurrent read-only operations read the data structure as if they have exclusive access to the data structure. In that case, we can guarantee linearizable multipoint queries. Indeed, we can provide concurrent data structure

operations with the illusion of having exclusive access to the data structure using snapshots.

The next chapter introduces a Multiversion B-Tree that builds on our B-Tree data structure design and offers linearizable multipoint queries.

Chapter 4

A GPU Multiversion B-Tree

4.1 Introduction

GPU databases are becoming the norm in data science pipelines to solve data analytics and machine learning inference and training problems. Using GPUs in these pipelines has advantages that include leveraging their large compute capabilities and avoiding the high latency required to move data between CPUs and GPUs. The rate of growth of GPU performance motivates continued investigation of the use of GPUs for database tasks; moreover, it appears that beyond general-purpose compute, GPUs will have additional on-chip special-purpose hardware for applications that include databases [21].

Data scientists can today use multiple commercial and open-source GPU databases, with increasingly easier-to-use and high-level abstractions [14, 28, 52]. At the core of these databases lies the set of underlying data structures that store data and provide ways to update and query it. While GPU data structures have not historically supported dynamic updates, recent work has successfully shown that hash tables [4] and B-Trees (Chapter 3) [64] can support both queries and updates at rates up to billions of operations per second. However, supporting *multipoint* queries, such as range queries on B-Trees, in the presence of updates is a more challenging task than the point queries currently supported by GPU data structures.

The specific challenge when supporting efficient concurrent multipoint queries (e.g., range queries) and updates is to provide *linearizable* query results. Linearizability ensures that the effect of a data structure operation must appear to take effect atomically at a point—a *lineariza-*

tion point—between the operation's invocation and response [29]. A sequence of concurrent operations is linearizable if their result matches the result of one sequential execution of these operations; this provides an intuitive understanding of the result of concurrent operations. While the state of the art in GPU B-Trees (Chapter 3) supports concurrent updates and range queries, it does not support linearizable range queries.

In our implementation, *snapshots* facilitate linearizability. A snapshot records the state of a data structure at a particular point in time. We achieve linearizable multipoint queries by taking a snapshot of the data structure and then performing queries on that snapshot. Updates are performed on the most recent version of the data structure. Moreover, snapshots allow maintaining multiple versions of the same data structure. Linearizability is composable, meaning that a system consisting of linearizable components is linearizable. For instance, using a data structure that supports snapshots, we can *compose* a special-purpose data structure such as a graph data structure (Chapter 5). The composed data structure will benefit from the guarantees and the properties of the base data structure.

Implementing snapshots on GPUs is a significant challenge. In general, approaches that implement concurrent CPU data structures do not scale to the level of parallelism on the GPU. GPUs require designs that achieve coalesced memory accesses, eliminate branch divergence, and minimize contention between the hundreds of thousands of threads (Chapter 3). Ensuring that the result of concurrent multipoint queries and updates are linearizable adds to the data structure's complexity and requires solutions that follow the abovementioned design requirements while adding minimal overhead. At a high level, we achieve efficient linearizable multipoint queries by making each tree node a versioned node, treating each versioned node as a single node (by only pointing to the head of the version list), and maintaining each version list's head at the exact location that traversals from parent nodes expect.

In this work, we explore and provide a solution to taking snapshots of a GPU B-Tree data structure. Our solution builds in part on our GPU implementation of a B-Tree (Chapter 3) and the CPU work of Wei et al. [60]. The following goals drive our design decisions:

- Maintain high performance when performing operations on the data structure compared to the base data structure with no support for versioning

- Snapshots should add minimal overhead
- Optimize for accessing new versions of the data structure compared to older versions
- GPU-friendly solutions should introduce as few memory accesses as possible and avoid contention.

Our contributions are:

1. An efficient GPU B-Tree that supports snapshots
2. Our data structure supports linearizable multipoint queries in the presence of updates
3. Our data structure supports phase-concurrent (synchronous), stream-concurrent (asynchronous), and on-device fully-concurrent operations
4. Efficient handling of per-node versioning using fine-grained locks and minimal locking
5. Introducing GPU-aware *scoped* snapshots
6. A GPU implementation of safe memory reclamation using epoch-based reclamation.

One of our primary goals is to develop tools and implementation components that enable designing future concurrent GPU data structures. Currently, our community lacks robust and fundamental data-structure building blocks such as flexible and efficient memory allocators and safe memory reclamation schemes. To this end, we make our implementations of these available.

4.2 Background and Previous Work

Our data structure supports concurrent queries and updates and uses snapshots to achieve linearizable multipoint queries. Having multiple snapshots requires safe memory reclamation techniques to reclaim older tree versions once they are no longer used and are not currently accessible by concurrent operations. In this section, we will summarize the efforts of building concurrent GPU and CPU data structures, snapshots, and safe memory reclamation.

4.2.1 Concurrent GPU Data Structures

Driven by a need for flexible and powerful data structures for database and data-science applications, researchers have recently produced numerous *dynamic* GPU data structures. Only a few of these data structures support concurrent updates and queries; these include hash tables [4], B-Trees (Chapter 3) [64], dynamic graphs (Chapter 5), and skip lists [50]. Of this work, our GPU B-Tree is the only one that supports concurrent range queries and updates; however, it provides no guarantees on the linearizability of these concurrent operations.

Our work builds on this GPU B-Tree which uses cache-line-sized nodes, where each node has a branching factor of 15. In this design, tree nodes on each level are chained, forming a linked list. Additionally, each node stores its right sibling's minimum key (i.e., high key)—the side-links alongside the node's high key help allow updates and traversals to run concurrently. A traversal operation can traverse the side-links and reach a sibling when a concurrent insertion performs a split on a tree node, and the traversal operation reads an older instance of the node. Allowing concurrent updates and traversals that do not require locking is essential to ensure high performance. However, one consequence of this B-Tree design is that range query operations are not linearizable.

For instance, two range queries and two insertions all concurrently running may result in non-linearizable results. Consider the case when the two range queries read the nodes a and b and the two concurrent insertions update the same nodes a and b, creating a' and b'. One possible overlapping of the operations that is not linearizable will occur if each of the two query operations reads the modified tree nodes in an order such that each query sees only one of the newly inserted keys. In other words, the first query will read a and b', while the second query reads a' and b. The results of the two range queries are not linearizable and do not match the result of any sequential execution. Our work focuses on achieving linearizable multipoint queries in a B-Tree.

4.2.2 Snapshots and Linearizable Data Structures

A snapshot of a data structure is a read-only version that contains all the key-value pairs stored inside the data structure when a take_snapshot operation is performed. Taking snapshots of a data structure has been explored on the CPU for different contexts such as recovery and

backup. Rodeh [53] showed how to support snapshots using a shadowing technique where the entire path from the root to the leaf is shadowed. A different use case of concurrent-data-structure snapshots is to enable a consistent view of the data structure for query operations that require reading multiple parts of the data structure and produce linearizable results. Other solutions for linearizable concurrent multipoint queries include persistent hash tries, epoch-based reclamation schemes, or other versioning-based schemes. In Ctrie [51], a persistent hash trie uses a lazy copying technique after an update—lazy copying requires a variant of double-word compare and swap. In the EBR-based scheme [3], range queries traverse the data structure and the reclaimed nodes to determine all keys that belong to the range query. K-ary search trees by Arbel-Raviv and Brown [3] traverse and validate the range query results using dirty bits in the tree nodes. Basin et al. [9] proposed a chunk-based data structure (similar to a B-Tree) where each key has a version and old versions are overwritten with no ongoing scans.

Recently, Wei et al. [60] introduced a general approach, versioned CAS objects (vCAS), to convert a concurrent data structure that uses compare-and-swap objects to one that supports snapshots. Notably, vCAS was the first algorithm that allows taking a snapshot of a data structure in a constant number of steps and preserves the data structure's asymptotic time bounds. More importantly, vCAS only requires a single-word read and CAS operations. Wei et al. [60] applied their algorithm to existing concurrent data structures, including a queue, linked list, and binary trees.

The challenge of achieving linearizability using snapshots is making the traversal operations (in update or query), and the timestamp increment (take a snapshot) appear to happen atomically. In Wei's work, the atomicity is realized by using an invalid timestamp to *mark* new nodes. Any data structure operation tries to initialize the invalid timestamp when encountering a marked node. This solution is suitable for a GPU data structure. Other solutions that use locks would limit the performance of any operations on the GPU.

Prior to our current work, snapshots have not been explored on the GPU. Our work builds on vCAS and implements its algorithm on top of a GPU B-Tree. We make additional modifications to build our B-Tree (described in Section 4.4.1) as the branching factor of the B-Tree necessitates locks and cannot be easily implemented using compare-and-swap objects, and effi-

cient implementations on GPUs require making design decisions that minimize any additional memory accesses. Our solution uses fine-grained locks alongside always maintaining pointers (including side-links) to the most recent node version, allowing us to perform concurrent reads and synchronize with other updates efficiently.

4.2.3 Safe Memory Reclamation

Safe memory reclamation (SMR) for concurrent CPU data structures also has a rich history. Solutions to the SMR problem include using reference counting, hazard pointers, epoch-based techniques along with other variants, and improvements of these techniques. Prior to this work, SMR has not been explored on the GPU. Next, we discuss the basics of these techniques and their appropriateness for GPUs.

Reference counting (RC) is a simple technique where a reference count is attached to each data structure node. Once the reference count is zero, the node can be reclaimed. A significant issue with using RC on the GPU is the additional overhead of memory operations on each access to a data structure node. Although DRAM bandwidth on the GPU is high compared to the CPU, data structure operations on the GPU are generally memory-bound, so this approach is undesirable.

In hazard pointers (HP) [49], each data structure operation first tries to *protect* all pointers that it may access, followed by ensuring that the protected pointers are still reachable from the data structure. Protecting a pointer means that the operation must share the pointer with all other threads on the GPU. Similar to RC, this additional memory operation and the fact that we need to perform additional reads to ensure that the pointer is reachable makes the overhead of RC unsuitable for GPUs.

Epoch-based reclamation (EBR) [24] reclaims memory by maintaining a global epoch count, a global array called the announce array storing states of all operations (e.g., their observed epoch number, and whether they are using the data structure or are instead *quiescent*), and a per-process local *limbo list* where *retired* pointers are stored then freed when it is safe to do so. Limbo lists are maintained for three epochs $\{e-1, e, e+1\}$. Only when we reach epoch $e+1$ can we reclaim pointers that are retired in epoch $e-1$, since a process at an epoch e might still be accessing pointers in the previous epoch. Processes performing operations on

the data structure (e.g., insertions or queries) first announce their observed epoch number, then inspect the state of all other processes to check if they observed the current epoch. Only then do the processes advance the epoch and move to the next limbo list reclaiming pointers from the oldest list. DEBRA [16] implements a distributed epoch-based reclamation scheme with a key contribution of eliminating the need of inspecting states of all other processes at the beginning of leaving a quiescent state. Instead, DEBRA reads the state of other processes incrementally over multiple operations.

In our implementation, we choose EBR (and follow DEBRA's approach), which we believe is more suitable for the GPU than other techniques, for two reasons. First, *retired* pointers are not shared across processes (i.e., can be stored inside a fast local shared memory). Second, since the scan of other processes' operations is performed in bulk (i.e., per process), *coalescing* the scan (thus optimizing memory access) is straightforward. We discuss our GPU EBR implementation in Section 4.4.4.

4.3 Design Decisions

4.3.1 In-place and Out-of-place Updates

An efficient implementation minimizes the cost of node updates. Our implementation supports two different strategies for updating a node: in-place and out-of-place. In an in-place update, tree nodes are mutated directly, without replacing the node (in Chapter 3 we also performed updates in-place). In contrast, an out-of-place update creates a copy of a node each time we update it. We prefer in-place updates because they are faster: in-place updates require only one write, out-of-place two. However, we can only perform an in-place update when we can ensure that a take_snapshot is not running concurrently with the update, and the current global timestamp matches the modified node timestamp. If either condition is false, we instead update out-of-place.

4.3.2 Scoped Snapshots

We have designed our data structure to be used in three scenarios for simultaneous updates and queries: (1) phase-concurrent (synchronous host-side calls), (2) stream-concurrent (asynchronous host-side calls), and (3) fully-concurrent (on-device calls). These three scenarios give

our users maximum flexibility to use our data structure; each offers a different tradeoff between control, functionality, and performance. Listing 4.1 summarizes the different APIs for these three scenarios.

Recall that the linearization point of a take snapshot operation is when the timestamp changes from t to $t+1$ [60]. This linearization point is essential to our operations and their corresponding optimizations. Our three use cases correspond to different synchronization scopes around the take_snapshot operation:

Host-side snapshot. A take_snapshot operation is performed from the CPU and it acts as a device-wide barrier. Using the snapshot handle, future query kernels performing read-only queries can execute safely alongside concurrent update operations. In this scope, read-only kernels can fully utilize the incoherent L1 cache—this can result in a $2\times$ performance gain. Moreover, since the timestamp will not change once we launch a GPU kernel, the same nodes updated in concurrent update kernels are performed using in-place updates. Performing in-place updates saves memory and improves the modified operations' performance (Section 4.3.1).

On-stream snapshot. A take_snapshot operation is performed from the CPU on a specific CUDA stream. The difference between this scope and the previous one is that we may take a snapshot while an updating kernel runs (in a different stream). Since the take_snapshot operation may execute concurrently with other query and update operations, this scope (and the following one) preclude using the L1 cache and instead perform out-of-place updates in all scenarios.

Tile-wide snapshot. A CUDA tile is a group of threads whose size does not exceed a CUDA block size. Here, a CUDA tile takes a snapshot of the data structure (i.e., the operation happens on the device) and broadcasts the version handle to the threads inside the tile. All queries inside the tile use the snapshot handle to traverse the tree alongside all the device threads performing any operations. When the tile size is one, we take one snapshot per query operation.

4.3.3 Older Version Access in Versioned Nodes

Pointers in our tree data structure only point to head nodes of version lists. That way, we can ensure that each version list has a single entry point through its head (i.e., older versions are only accessible through the version list and not side links). Section 4.4.1 discusses how we

```
struct gpu_data_structure{
 // Host-side APIs
 void insert(Pair* pairs, Stream stream);
 Timestamp take_snapshot(Stream stream);
 Result* query(/*..query arguments..*/, Stream stream);

 // Device-side APIs
 void insert(Pair pair, Tile& tile, Reclaimer& reclaimer);
 Timestamp take_snapshot(Tile& tile);
 Result query(/*..query arguments..*/, Tile& tile);
};
```

Listing 4.1: High-level APIs for different scopes.

take advantage of this design decision to easily perform updates concurrently with other tree traversal operations.

4.4 Implementation

Our Multiversion B-Tree is based on our B-Tree implementation that we discussed in Chapter 3. In both implementations, each tree node occupies a cache line (i.e., 128 bytes). Using an entire cache line to represent a tree node and operating on a tree node in a tile-wide fashion (i.e., SIMD-style group of threads) allow achieving coalesced memory access and avoiding branch divergence. Nodes in both implementations hold a side-link pointer and the minimum sibling node key. The Multiversion B-Tree node contains a timestamp field and holds an additional pointer, to the next version of the tree node, thus reducing the branching factor by one (assuming 8-byte key-value pairs, the branching factor is 14). Additionally, our tree maintains a global timestamp that we increment each time we take a snapshot. We discuss the performance differences that result from this reduction in Section 4.5.1.

Our Multiversion B-Tree data structure supports the following operations:

insert(k, v)**:** inserts a key-value pair (k, v) into the Multiversion B-Tree into the latest version of the data structure.

delete(k)**:** removes the key-value pair associated with the key k from the latest version of the Multiversion B-Tree.

takeSnapshot(): takes a snapshot of the data structure and returns a handle to the snapshot.

find(k, ts)**:** finds the value associated with the key k from the Multiversion B-Tree at a timestamp ts.

rangeQuery(k_1, k_2, ts)**:** Returns all key-value pairs in the range defined by $k_1 \leq k \leq k_2$ at timestamp ts.

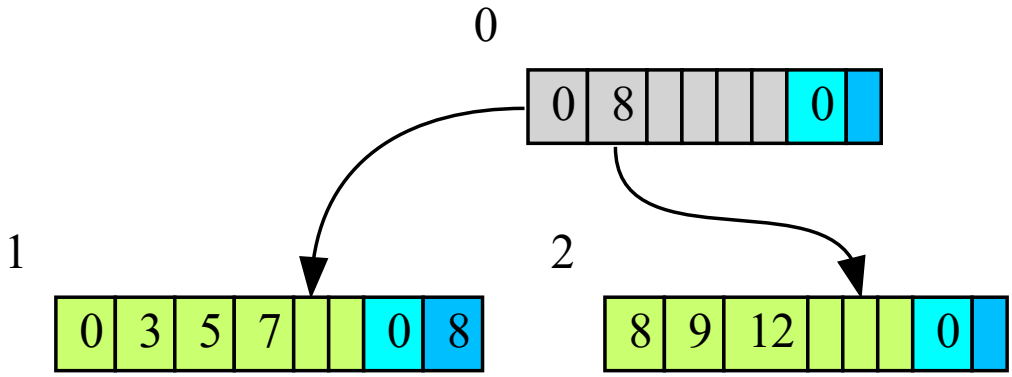
Next, we will discuss each of the operations our data structure supports. We extend the warp-cooperative work-sharing strategy described in Section 3.3.6 and perform each operation in a *tile-wide* fashion, where the tile width matches the tree node width (i.e., tile width is 16). We use CUDA's cooperative-groups abstraction[4] to represent tiles and perform intra-tile communication. These communications include threads voting and broadcasting keys (or key-value pairs) within a tile. We omit the description and usage of tiles for brevity.

4.4.1 Insertion

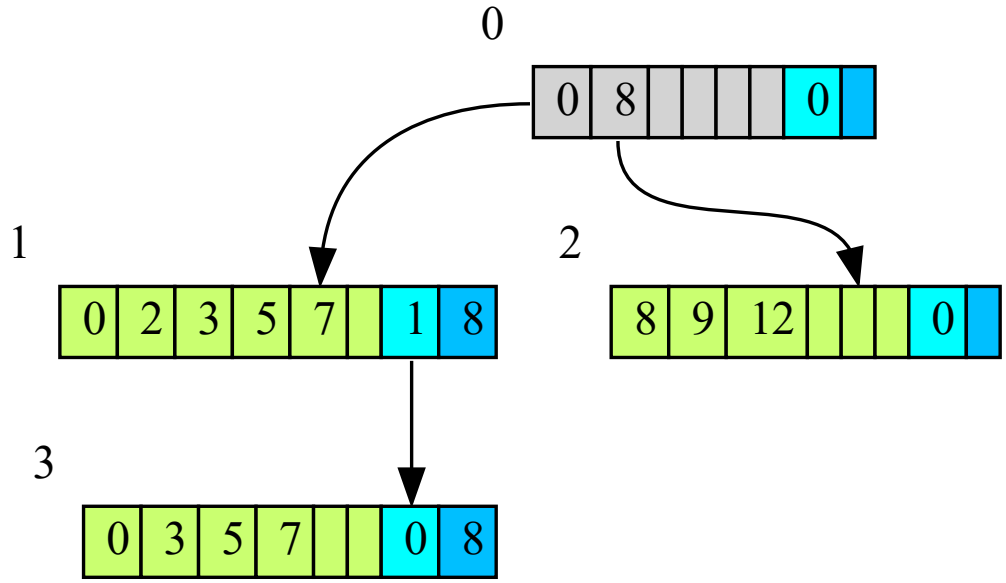
We adapt the insertion algorithm described in Section 3.4.2 and extend it to support snapshots and linearizable multipoint queries.

To realize a versioned B-Tree, we represent tree nodes as a version-list where the head of the list is the most recent version of the node. Any pointer in the tree structure will only point to a head node of a version list. Only pointing to the head node allows us to treat an entire version list as a single tree node, which simplifies the traversal operations because only one entry points to a version list exists. Whenever we need to create a new version of the tree node, we must copy it using a copy-on-write (COW) scheme. Traditionally, COW is performed by copying the tree node, modifying the copy, and finally swinging the pointer that points to the node (from its parent) to the new tree node. COW is efficient since it does not block concurrent reads. However, modifying a pointer in the parent node requires locking. Since all the tree nodes will have multiple versions, COW would introduce contention and scale poorly on the GPU. As we showed in Chapter 3, locking tree nodes on multiple levels and (particularly) close to the root is unsuitable for the GPU. Thus we present an alternate strategy to efficiently copy a tree node on the GPU.

[4] `https://developer.nvidia.com/blog/cooperative-groups/`



(a) Initial Multiversion B-Tree at the initial timestamp, $t = 0$. Side-links are hidden.



(b) At the new timestamp, $t = 1$, insertion is performed over two steps. After locking the tree node, we copy the node to a new memory location (index $= 3$). Then the initial node (with index $= 1$) is modified in place. Notice that the pointer from the root node always points to the most recent version of the node. Also, notice that the two version-list nodes (nodes 1 and 3) are linked to the same sibling (node 2).

Figure 4.1: An example of inserting a key in a Multiversion B-Tree with a branching factor of 6. Intermediate and leaf nodes are colored gray and olive green, respectively. Version-list and side-links locations are colored in cyan and blue, showing the node's timestamp and the high-key, respectively.

Copying a tree node. To avoid modifying the parent node (when copying one of its children), we maintain the invariant that the most recent version of the child node occupies the same memory location pointed by the pointer in its parent node. That way, any active traversal through the parent node will always reach the most recent version of the copied node. Recall that one of our goals is to optimize accessing the most recent version of the tree. For example, if we need to copy a locked child c with a parent p to a new location c', we first copy the node c to c', then update c in place. The modified copy of c contains both the required mutation and a pointer to c'. Notice that both c and c' will have the same right sibling even if the right sibling has multiple versions. Traversals heading to the old node's version that read the node c before updating it reach the required tree node without additional steps. However, traversals that need to reach c' that read the modified c will inspect its timestamp and then traverse the version-list to c'. The traversal will reach the required node with the old timestamp in both cases. Other traversals that need to mutate the node will be blocked by the thread performing the copy (i.e., holding the node's lock). These traversals will either spin or restart their traversal. Figure 4.1 shows an example of copying a tree node.

In-place and out-of-place insertions. We distinguish between in-place and out-of-place insertion based on the scope of the snapshot (i.e., how synchronization occurs around a take_snapshot operation).

For a host-side snapshot, incrementing the snapshot counter may follow the following sequence: (1) kernel that performs updates on the data structure; (2) kernel that takes a snapshot (i.e., increments the snapshot counter); (3) query kernel. The take_snapshot kernel introduces an implicit device-wide barrier. Once we increment the snapshot counter, any query operation using the snapshot identifier can run concurrently with any future insertions. The barrier between the take_snapshot kernel and others makes it possible for following update operations to modify tree nodes in place whenever possible.

Our rule for copying a tree node is to create a copy of a tree node if its timestamp differs from the global timestamp; otherwise, we perform the update in place. Insertion into a leaf node requires modifying only that node. During insertion traversals, any tile that reads a full node proactively attempts to split the node. Splitting a full tree node can be broken down into

multiple steps that follow the same rule:

Splitting a full root node. We only need to check the root timestamp. If we need to copy the root, the new root will be at the same location as the previous one. The two new children will have the same timestamp as the modified root.

Splitting a full intermediate (or leaf) node. Splitting will update both the full node and its parent node. We check both nodes' timestamps and create a copy following our rule for node copying.

For other scopes (e.g., tile-wide snapshot or on-stream snapshot), taking a snapshot is performed concurrently with a read-only operation. Our approach is similar to Wei et al. [60]; the main differences being how we perform copy-on-write and the B-Tree specific operations (e.g., splitting). Listing 4.2 shows how we perform insertion when the snapshot is taken concurrently with read-only operations. We refer the reader to Section 3.4.2 for the full description of the base insertion algorithm and the reasoning behind the restart logic, but in summary: restarting the traversal from a parent node happens in the following cases: (1) failure when trying to acquire a lock, (2) parent is not correct due to another operation splitting that parent node. Restarting the traversal from the root happens when the operation cannot proceed as the parent is unknown, for instance, after a restart and finding that we restarted from a full node or if the parent node is full during a split. Note that we generally try to acquire locks and restart if acquiring the lock fails. We only spin on a lock when updating a parent node during a split or the traversal of side-links after locking a tree node.

Now we discuss the modifications to the insertion algorithm to perform out-of-place updates. We omit the description of the in-place algorithm as it is similar to the base algorithm with the addition of copying nodes when necessary.

Timestamp initialization. We always attempt to initialize the node's timestamp when traversing side-links (either with or without holding locks) (line 7 and 15). Note that we only need to initialize the timestamp for leaf nodes; a call to `initialize_timestamp` can quickly detect if the node's timestamp is uninitialized using intra-tile communication after loading the node. We initialize a timestamp by atomically performing a compare-and-swap on the node's timestamp field to attempt swapping an uninitialized timestamp with the current value of the

global timestamp.

Splitting a root node. We first start by allocating a node that will hold the old root contents and two nodes that will hold the two children (lines 21–22). We store a copy of the root into the newly allocated node and retire it; then, we split the root setting all the node timestamps to the current one (lines 23–25). Notice that splitting does not change the contents of the key-value pairs stored in the tree but only the layout of some nodes. It is a requirement that the three nodes resulting from a split have the same timestamp; this way, we ensure that all key-value pairs are accessible at any timestamp. Once we finish splitting the root, we store the three nodes and traverse to either of the children unlocking the other child and the root (lines 26–34).

Splitting an intermediate node. The difference between splitting an intermediate and a root node is that we need to acquire a lock over the parent node, ensure that the parent is not full, and that is the expected one (i.e., other tiles did not split the parent). After successfully acquiring the lock, splitting occurs similarly to splitting a root. We create copies of both the parent and the full node before splitting.

Inserting into a leaf node. Once the traversal reaches the tree node (line 39), we allocate a node to hold the tree node's old contents, store a copy at the newly allocated node, then retire it (lines 40–42). We modify the leaf node in-place adding a link to the older tree node version, then store the leaf node with an uninitialized timestamp and unlock it (lines 43–45). Finally, we try to initialize the node's timestamp (line 46). The tile that performs the modification or the tile that reads the uninitialized timestamp will successfully set the timestamp. The addition of the version-list node is linearized once we read the timestamp and successfully set the timestamp.

```
void VBTree::insert_out_of_place(Key key, Value value, Reclaimer& reclaimer){
  RootRestart: uint32_t node_index = root_index;
  uint32_t parent_index = root_index;
  bool links_used = false;
  do{
    VersionedNode node(node_index);
    links_used |= traverse_links_init(node, key);
    if(node.is_full() && node_index == parent_index && node_index != root_index){
      goto RootRestart;
    }
    if(node.is_full() || node.is_leaf()){
      if(!node.try_lock()){
        node_index = parent_index; continue;
      }
      links_used |= traverse_locked_links_init(node, key);
      if(node.is_full() && links_used){
        node_index = parent_index; continue;
      }
    }
    if(node.is_full() && node_index == root_index){
      auto old_root_index = allocate(1);
      auto [child0_index, child1_index] = allocate(2);
      node.store_copy_at(old_root_index);
      reclaimer.retire(old_node_index);
      auto two_children = node.split_as_root(child0_index, child1_index,
          old_node_index, cur_ts);
      two_children.right.store(); two_children.left.store();
      node.store(); node.unlock();
      node_index = node.find_next(key);
      if(node_index == child0_index){
        two_children.right.unlock();
        node = two_children.left;
      }else{
        two_children.left.unlock();
        node = two_children.right;
      }
    }else if (node.is_full()){
      // split as intermediate
    }
    if(node.is_leaf()){
      auto old_node_index = allocate(1);
      node.store_copy_at(old_node_index);
      reclaimer.retire(old_node_index);
      node.insert(key, value);
      node.set_older_version(old_node_index);
      node.unlock();
      node.initialize_timestamp(); return;
    }else{
      parent_index = node_index;
      node_index = node.find_next(key);
    }
  }while(true);
}
```

Listing 4.2: Out-of-place insertion.

4.4.2 Query Operations

A query operation requires read-only access to the data structure. A query operation can be as simple as a point query or as complex as scanning the entire data structure. Moreover, in our Multiversion B-Tree, a query operation can have an additional argument specifying the timestamp (i.e., a number mapping to a point in the history of the data structure). In addition to the ability of querying an older version of the data structure, snapshots (i.e., timestamped queries) provide a linearizable view of the data structure.

Taking a snapshot of our Multiversion B-Tree is a simple operation. A take_snapshot tries to increment the global timestamp using a compare-and-swap operation atomically. Only one thread in a tile reads the timestamp t then tries to set it to $t + 1$. Similar to Wei's work [60], multiple concurrent take_snapshot operations may return the same snapshot handle. Only one of the concurrent take_snapshots operations needs to succeed.

We show an example of a linearizable range query operation in Listing 4.3. We first start the traversal from the root of the tree. Each time we load a tree node, we attempt to initialize its timestamp and traverse the side-links (while initializing each sibling timestamp), and then we traverse the version list (lines 5–7). During the version-list traversals, we traverse the list until we find a node with a timestamp that is at most the query's timestamp. Initializing leaf node timestamps ensures the linearizability property. Concurrent insertion performing splits may move our target key (or pivot) into a sibling node, hence side-link traversal is necessary to improve performance and reach the correct tree node. Performing side-link traversal improves the performance of the top-down search as it allows us to skip over parts of the tree. Notice that insertion guarantees that any older version of a tree node will have an initialized timestamp; therefore, traversing the version-list does not require initializing the version-list nodes.

If we reach an intermediate node, we find the next node down the tree (line 9). Otherwise, we start traversing side-links collecting all pairs that belong to the range (lines 11–16). Similar to the traversal part, we initialize each sibling node timestamp and traverse the sibling's version-list. We terminate the search once the high-key of the node is less than the sibling's high-key (line 13).

Point queries are more straightforward and follow the same logic; however, they don't need

to collect a range once their traversal reaches the correct leaf node.

```
1  void VBTree::range_query(Key lower_bound, Key upper_bound, Timestamp timestamp, Pair
       result)
2  {
3    uint32_t node_index = root_index;
4    do{
5      VersionedNode node(node_index);
6      traverse_side_links_init(node);
7      traverse_version_list(node, timestamp);
8      if(node.is_intermediate()){
9          node_index = node.find_next(lower_bound);
10     }else{
11       do{
12         node.get_in_range(lower_bound, upper_bound, result);
13         if(upper_bound < node.get_high_key()) return;
14         node = node.get_sibling();
15         node.initialize_timestamp();
16         traverse_version_list(node, timestamp);
17       }while(true);
18     }
19   }while(true);
20 }
```

Listing 4.3: Range query.

4.4.3 Deletion

In deletion, we traverse the tree until we find the leaf node that contains the key. Deletion is similar to an insertion that does not perform any splits. Similar to insertion, we perform side-link traversal and initialization of nodes with invalid timestamps. Once we reach the target leaf node, we either perform the deletion in-place or out-of-place. In an in-place deletion, we only create a copy of the old tree node if the global timestamp differs from the node's timestamp. In an out-of-place deletion, we copy the node contents, perform the modification, then retire the old copy. Whenever we create a copy of the node, we link the new version of the node with the copy to form the version list. We perform deletion either in-place or out-of-place (similar to insertion) based on the different snapshot scope.

4.4.4 Safe Memory Reclamation

Our EBR implementation follows DEBRA [16]. The main differences between our implementation and DEBRA are GPU-specific implementation details.In general, we see three different possible granularities for implementing EBR on a GPU: device-wide, block-wide, or tile-wide. A device-wide reclamation scheme would wait for all concurrent kernel launches to finish be-

fore freeing its limbo bags. Such granularity is suitable if operations are performed only from the CPU (i.e., host-side snapshot). We will focus on concurrent operations performed on the device, requiring either a block-wide or a tile-wide reclamation granularity.

Since EBR requires scanning the state (i.e., announce array) of all the other processes (i.e., block or a tile), we must store the reclamation scheme state in a memory accessible to the entire device (i.e., device DRAM). Since memory accesses are expensive, we see block granularity as the one that delivers the highest performance. We note that tile-wide reclamation would give the data structure user more flexibility since we need to synchronize only a tile (not the entire block).

A critical optimization in our implementation is limiting the number of thread blocks used by any kernel that uses our EBR implementation. In our implementation, kernels perform data structure operations in a persistent-kernel style. This optimization allows us to minimize the number of memory accesses we need to perform when scanning the entire state of the GPU blocks. For instance, if we use blocks of size 128 threads on a GPU with 80 streaming multiprocessors with 16 resident blocks, we only need to scan 16×80 announce entries (i.e., 40 cache lines). We examine the entire announce array entries cooperatively using the block then communicate through shared memory to detect if we should advance the epoch number. In practice, the maximum number of concurrent blocks is limited by the kernel usage of shared memory and register usage, among other limiters. We detect and configure reclamation maximum blocks dynamically during runtime.

Our block-wide EBR utilizes per-block (fast) shared memory to avoid directly storing its local state (i.e., limbo bags) into (slower) GPU DRAM. Once a data structure operation retires a pointer, the EBR stores the pointer into the fast shared memory and atomically increments the retired-pointers count. Note that except for reading or modifying the announce array, all block-wide EBR operations use a CTA scope.

Since shared memory is a limited resource, our EBR is configurable with a maximum number of pointers stored into shared memory. In case of bags overflow, we store the pointers into a bag stored in the DRAM. Similar to scanning the announce array, when we free pointers, the block cooperatively loads the limbo bag (either from shared or global memory) then deallocates

the pointers.

4.5 Results

In this section, we will evaluate the performance of our Multiversion B-Tree and compare it to one that does not support versioning or linearizable multipoint queries. We recognize and expect that supporting snapshots and achieving linearizable multipoint queries will come at a cost, and we would like to quantify that cost (recall our goals in Section 4.1).

Methodology. We evaluate our implementations on an NVIDIA Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM and an Intel Xeon Gold 6146 CPU. The GPU has a theoretical achievable DRAM bandwidth of 900 GiB/s. Our code[5] is complied with CUDA 11.5. Except for the memory reclamation evaluation Section 4.5.2.3, all results are averaged over 20 experiments. All keys (and values) are unsigned 32-bit randomly generated unique keys and uniformly distributed between 1 up to the maximum unsigned integer. We refer to a Multiversion B-Tree with in-place and out-of-place updates as ViB-Tree and VoB-Tree, respectively. We compare our results to our reference B-Tree (Chapter 3). All the data structures in our benchmarks use our modified version of SlabAlloc [4] where the allocator is configured with a memory pool of 8 gigabytes. Our EBR is configured with bags that can hold up to 128 pointers per bag and we store any additional pointers in a private per-processor memory stored in global memory.

Summary of results. Using snapshots and EBR, our data structure supports linearizable multipoint queries with minimal additional memory overhead (3.26% overhead). We achieve similar performance (1.11× and 1.04× slower for insertion and queries, respectively) to a B-Tree when using our data structure to perform in-place updates (ViB-Tree). As the update ratio increases for concurrent operations, the cost of insertions and copying tree nodes starts to reduce our throughput. Our VoB-Tree performs similarly to a non-linearizable baseline at low update ratios and 2.39× slower at high update ratios.

[5]Our implementation is available at `https://github.com/owensgroup/MVGpuBTree`.

4.5.1 Comparing to a B-Tree

In this benchmark, we compare the performance of non-concurrent (i.e., phase-concurrent) operations that do not require versioning. The goal is to quantify the cost of using our data structure over a regular B-Tree. We build the data structure from scratch using a different number of keys then we query all keys in the data structure. Figure 4.2 shows the result of these benchmarks.

Insertion. Our data structure achieves slightly lower performance as a B-Tree when performing in-place builds (our ViB-Tree is on average 1.11× slower). Reading the global timestamp and broadcasting a node's timestamp across the tile adds a very low overhead. Out-of-place insertion in our data structure achieves lower performance (2.5× slower). We expect out-of-place insertion to be slower as it performs at least two writes instead of one for the typical case of inserting into a leaf node. Moreover, the critical section length increases as the out-of-place copy is performed within the critical section. On average, {B-Tree, ViB-Tree, VoB-Tree} have build rates of {240.065, 216.512, 94.605} million keys per second.

Point query. Searching the latest version of the tree does not have any additional overhead as no version-list traversal is required. The only factor that affects the performance of a point query is the tree height. Recall that the branching factor of our versioned tree is 14 and the B-Tree is 15. This decrease in the branching factor shifts the number of keys that increase the tree height from seven to eight from 14 million to 8 million. Interestingly, VoB-Tree outperforms the others when the number of keys exceeds ≈23 million. We believe that TLB misses (an artifact from the memory allocator) decrease query throughput as the tree size increases. However, once insertions allocate enough tree nodes (i.e., enough collisions happen in the allocator), the allocator starts allocating memory from neighbor blocks, reducing the number of TLB misses and improving the query performance. All of the B-Trees have a similar performance trend; however, the higher number of allocations in the VoB-Tree makes this effect appear earlier than the other two. On average, the query throughput in {B-Tree, ViB-Tree, VoB-Tree} are {1512.964, 1453.697, 1549.977} million keys per second.

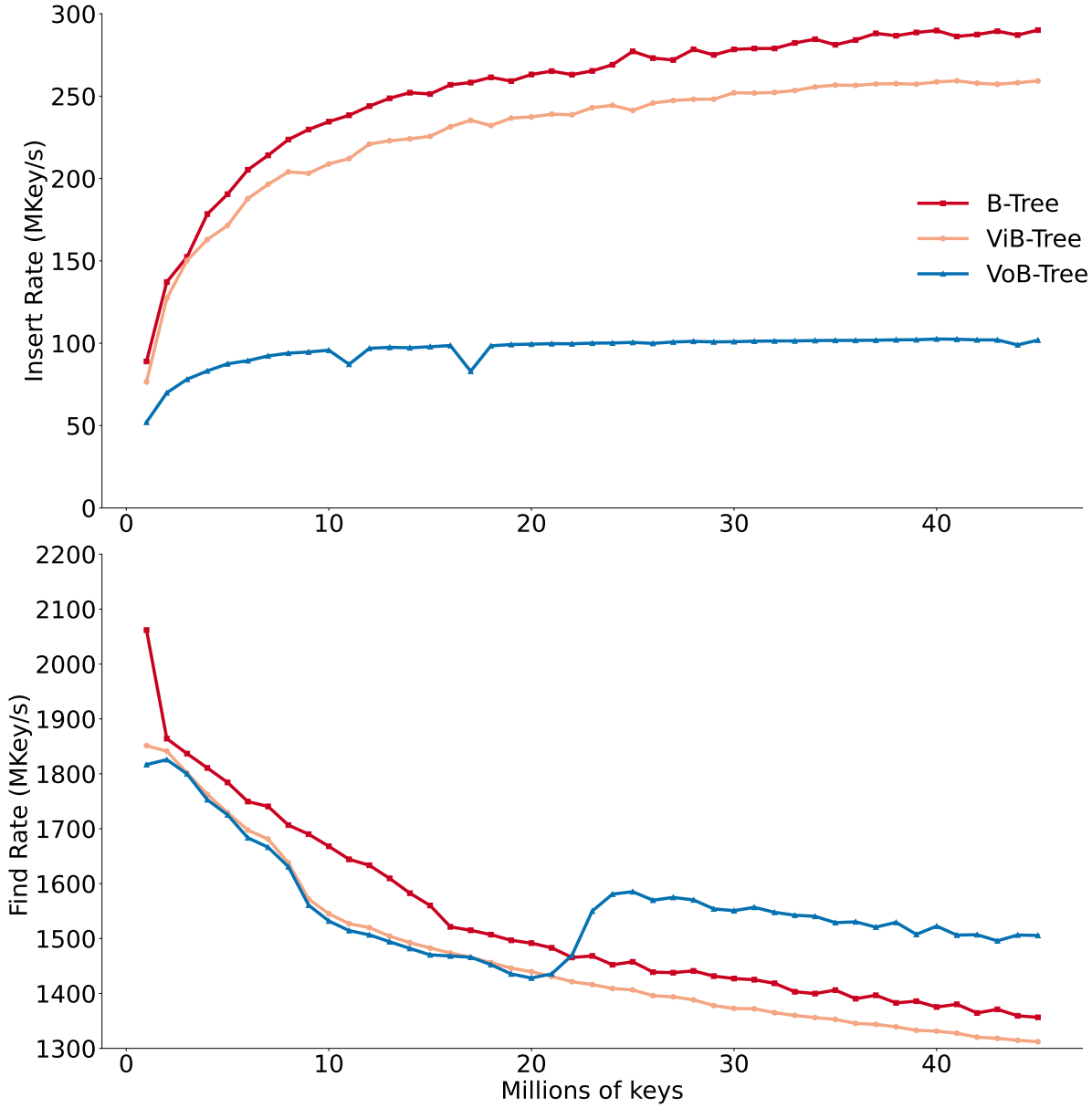


Figure 4.2: Insertion and find rates for trees containing different number of keys and the different B-Tree implementations. Find operations search for all keys in the data structure. Insertion performance is similar for the B-Tree and the ViB-Tree. However, the additional copies for out-of-place updates lower the insertion performance in a VoB-Tree. For queries, VB-Trees split the root earlier than the B-Tree, increasing the height and adding an additional read. Interestingly, VoB-Tree query throughput improves after 20M keys due to better TLB performance (Section 4.5.1).

4.5.2 Multiversion B-Tree Performance

The major use case of our implementation is to support concurrent queries and updates to the data structure, guaranteeing linearizable multipoint queries. To evaluate the performance of the concurrent operations in our VoB-Tree, we perform only two types of operations at a time where one of the operations is an update, and the other is a read-only query. Limiting the number of concurrent operation types to only two allows us to better understand the performance of each operation when it dominates the runtime of the kernel.

Benchmark setup For any concurrent operations benchmark, we first build an initial tree using in-place updates (ViB-Tree); then, we launch a persistent kernel that divides the GPU into two partitions where each partition performs a single operation. Both partitions will execute in parallel, performing the operations over multiple iterations. Each iteration performs a number of operations equal to the block size. We instantiate a memory reclaimer within each block and then perform the operation in a tile-wide fashion (i.e., we use tile-wide snapshots and a VoB-Tree). The tile width matches the tree node width. Each time a tile starts (or finishes) performing its operations, it leaves (or enters) the quiescent state. Note that our Multiversion B-Tree results are linearizable, but the B-Tree results are not linearizable.

4.5.2.1 Concurrent Insertion and Range Query

In our first benchmark, we perform concurrent insertion and range query operations. We build the initial data structure with either 1 or 40 million keys; then, we perform a number of operations divided between update and query using the update ratio α. We use two average range lengths in our experiments (8 and 32) to evaluate the difference between performing short and long level-wise and version-list traversals (on average, each node is $2/3$ full). Figure 4.3 and Table 4.1 show the results and summary of this benchmark. Interestingly, our VoB-Tree outperforms the B-Tree when the update ratio is 5%. Compared to the VoB-Tree range query results, the B-Tree ones include more pairs from the concurrent insertions. The difference in the range query result size is because traversing the snapshot stops the range traversal earlier than the ones in the B-Tree (i.e., range queries in the B-Tree read more nodes and write more results). As the update ratio increases, the high insertion cost dominates the overall operations rate. Since updates are more costly in a VoB-Tree than in a B-Tree, for high α scenarios, the

		1M pairs initial tree		40M pairs initial tree	
RQ length	α	B-Tree	VoB-Tree	B-Tree	VoB-Tree
8	5%	212.932	233.537	228.694	248.86
	50%	211.936	128.211	220.186	127.256
	90%	219.688	95.073	222.729	95.391
32	5%	221.691	227.793	240.13	214.679
	50%	210.405	120.956	213.199	117.642
	90%	213.406	93.199	220.329	92.154

Table 4.1: Average concurrent insertion and range query rates (million operations per second) for different update ratios, initial tree sizes, and range query lengths.

overall throughput drops significantly in a VoB-Tree.

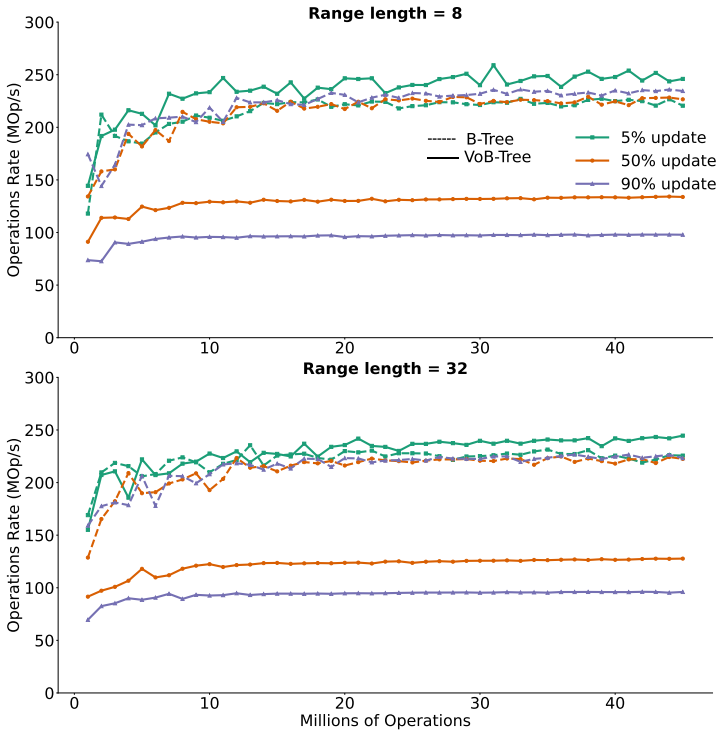
Figure 4.4 shows the result of varying the range query length while performing the concurrent range query and insertion benchmark. As the range query length increases, the total operations rate drops since the range query operations serially traverse more leaf nodes and version lists. For a tree with an initial size of 1 million keys, the total operations rate drops from $\{197.253, 127.895\}$ to $\{43.534, 27.923\}$ for {B-Tree, VoB-Tree}.

4.5.2.2 Concurrent Delete and Point Query

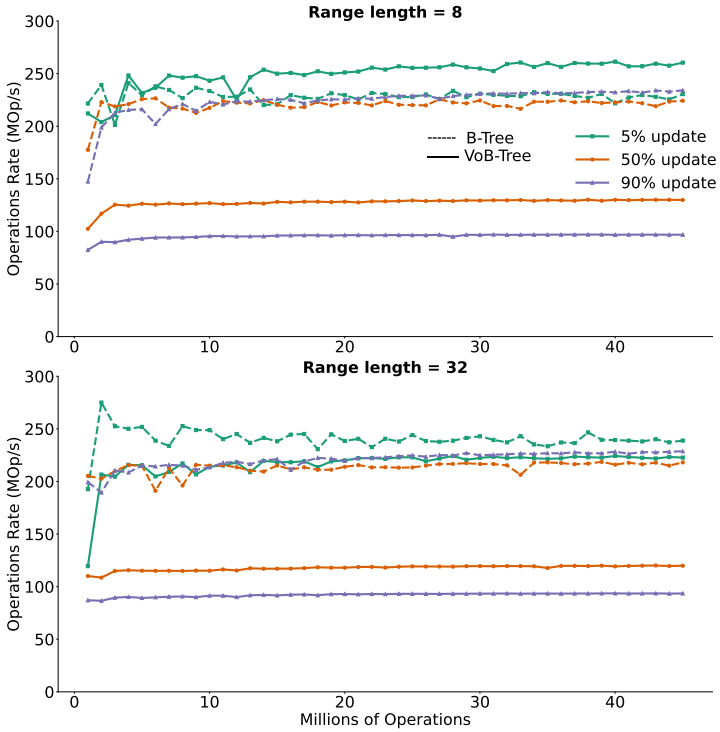
For our second benchmark, we perform concurrent delete and point query operations. This benchmark uses an initial tree size of 45 million keys. We perform α deletes and $1 - \alpha$ queries for different numbers of operations. Figure 4.5 shows the results of this benchmark. For all ratios, the B-Tree is $1.16\times$ faster than the VoB-Tree averaged over all experiments. Deletion in a VoB-Tree always performs two writes compared to a single write in a B-Tree. Since deletes have a higher cost than queries, the total rates start to drop when the update ratio increases. Table 4.2 summarizes the results of this benchmark.

4.5.2.3 Memory Usage and Reclamation

One of the critical components in our system is memory reclamation. To measure its performance, we instrument one of the concurrent-insertion-and-range-query benchmarks to query the allocator's number of allocated and freed bytes each time a block successfully advances an epoch. Figure 4.6 shows the results of this benchmark. During the first 300 epochs, the allo-

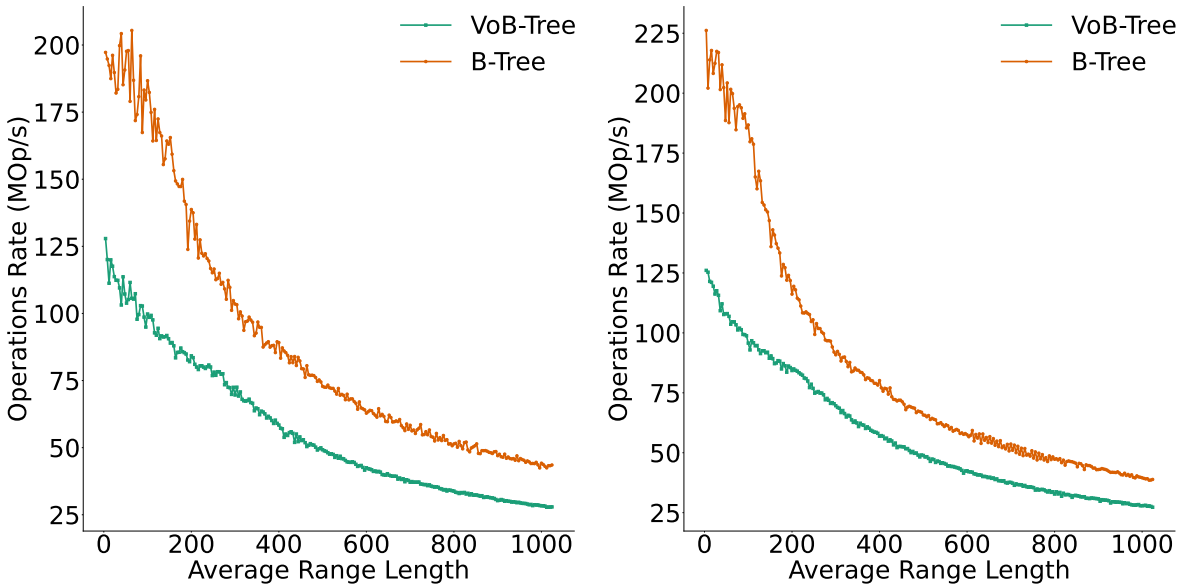


(a) Initial tree size of 1M.



(b) Initial tree size of 40M.

Figure 4.3: Concurrent insertion and range query using different update ratios and initial tree size.



(a) Initial tree size of 1M.

(b) Initial tree size of 40M.

Figure 4.4: Effect of varying the range query length on the concurrent insertion and range query rates when performing 5 million operations with an update ratio of 50%.

α	B-Tree	VoB-Tree
5%	433.462	369.835
50%	424.131	367.432
90%	399.863	346.188

Table 4.2: Average concurrent delete and find rates for different update ratios (million operations per second).

cator allocates around 11 megabytes per epoch (i.e., 91 thousand nodes allocated per epoch) and reclaims 10 megabytes per epoch. After epoch 300, blocks performing insertion start to exit, thus lowering the allocation rate to 1 megabyte per epoch, ≈77% of which are reclaimed. Notice that the first 300 epochs correspond to ≈72% of the kernel runtime in this experiment.

A ViB-Tree containing the same number of keys (67.5 million keys) uses 950 megabytes; using our EBR while concurrently building and querying the tree, the memory usage in the last epoch is 981 megabytes (3.26% overhead for supporting versioning). Notice that the shared memory does not persist between kernels. Therefore, we must flush all the block's reclaimer limbo bags stored in shared memory to the private block storage in global memory. After

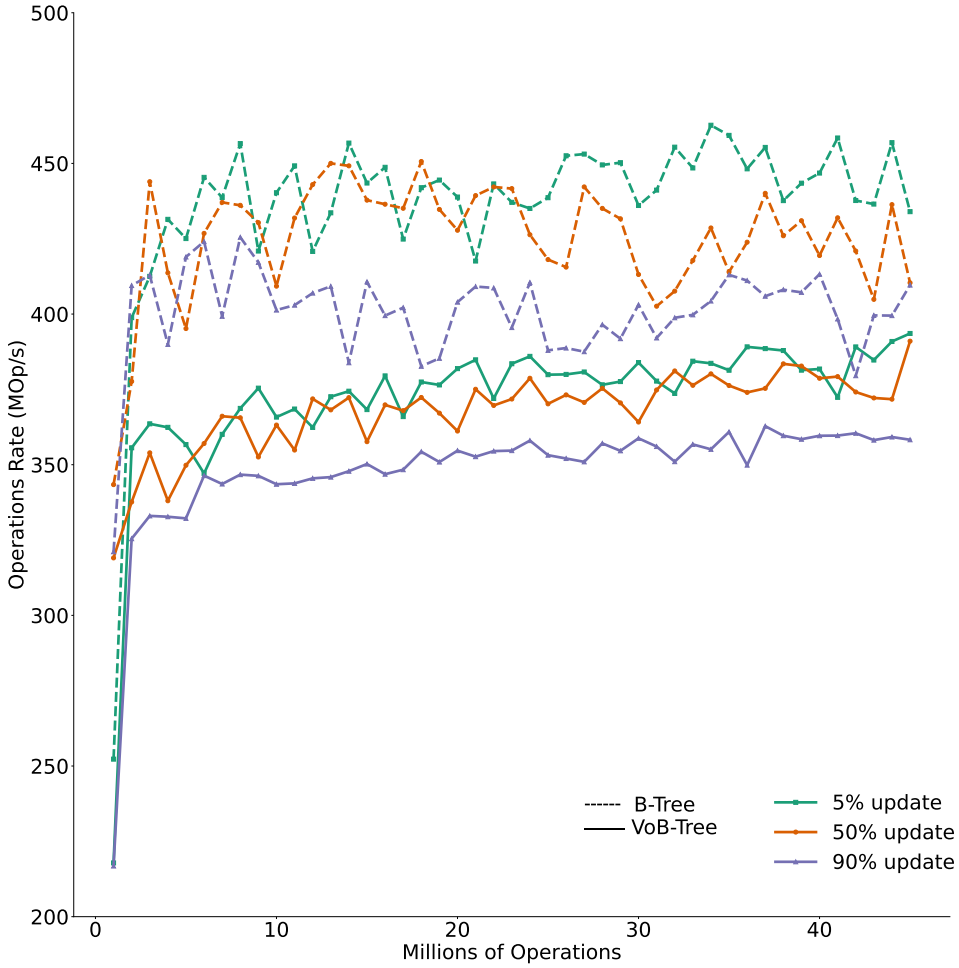
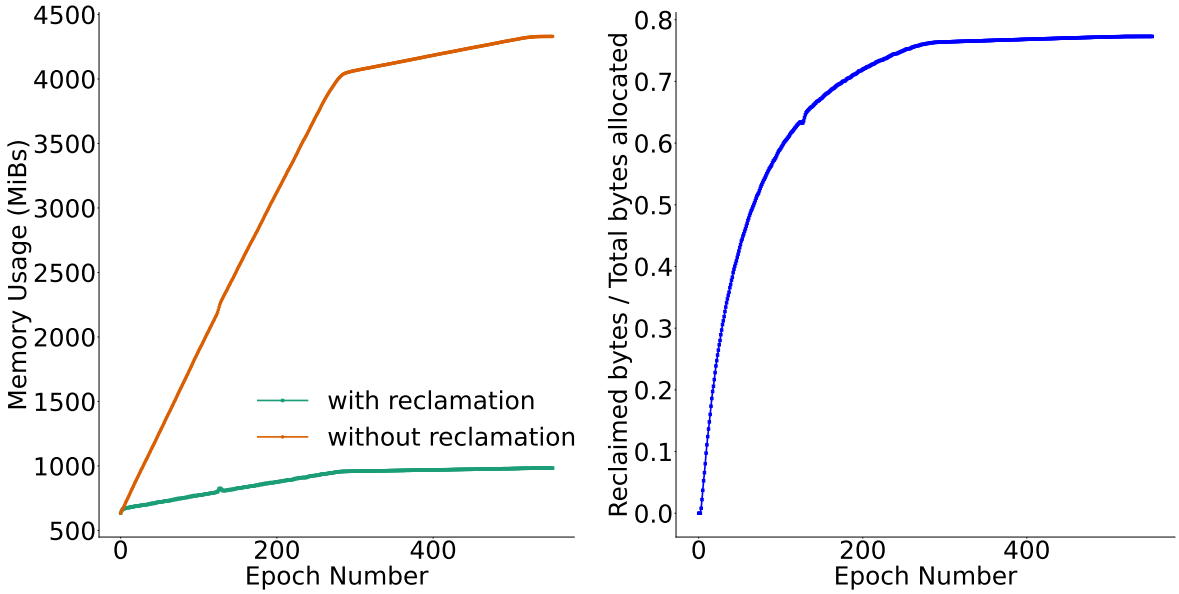


Figure 4.5: Concurrent delete and point query using different update ratios for a tree with an initial size of 45 million keys.

finishing kernel execution, we can free these pointers or load them as shared limbo in future executions.

4.6 Conclusion and Future Work

In this work we describe the design and implementation of a GPU B-Tree with snapshots and linearizable multipoint queries. Our design encompasses different GPU data structure common use cases and can perform in-place updates and take advantage of L1 cache whenever possible. Although fine-grained locks and restarts of update operations reduce contention, supporting snapshots requires performing additional operations inside the critical section (e.g., copying nodes), which reduces the overall update performance by a factor of 2.4x. This reduced per-



(a) Memory usage in MiBs.

(b) Ratio between the reclaimed to the total allocated bytes.

Figure 4.6: Memory usage for a Multiversion B-Tree performing 45 million concurrent insertion and range queries (50% update ratio and average range length of 16). The initial tree size is 45 million keys. Blocks that perform insertion start to exit when the epoch number reaches ≈ 300, reducing the allocation rate (left). Once three epochs pass, the ratio between the reclaimed to total bytes allocated starts to exceed zero (right).

formance is the cost of supporting a more capable data structure. We believe that linearizable multipoint queries, first implemented in this work, are common and useful in real-world applications.

Using the tools we developed, we want to explore wait-free techniques to build tree structures on the GPU in future work. We believe that wait-free data structures will potentially reduce the overhead of supporting snapshots, and more broadly, a broader toolbox of techniques for building data structures will help advance the GPU as a vibrant target for database and data science applications.

We discussed how to build two general-purpose data structures in the previous chapters. The next chapter will discuss composing a special-purpose data structure (dynamic graph) from a general-purpose one.

Chapter 5

Dynamic Graphs on the GPU

5.1 Introduction

While interest in graph analytics on GPUs has exploded in recent years, the vast majority of this work focuses on static graphs that never change during the graph computation. Today's GPU graph analytic frameworks generally lack a GPU-managed dynamic graph data structure that supports changes (insertions and deletions) to the graph as well as queries into this data structure. The key challenge is representing the neighbors of each vertex (the "adjacency list"), where that data structure must be flexible enough to support a wide range of sizes and efficiently allow both queries into and changes to this data structure.

Previous efforts to support GPU-based dynamic graph data structures represent an adjacency list with a list-based data structure [17, 61] or an array-based data structure that maintains sort order [55]. Its implementation as a list/array presents a dilemma for its designer:

- Adjacencies can be stored as an unsorted list, which is easy to maintain. However, unsorted lists are unacceptably slow for important operations on the data structure, e.g., edge-existence queries ("does v have u as a neighbor") or insertions that do not result in duplicates, both of which require traversing the entire list. Consequently, with an unsorted list, many operations are $O(n)$ in the size of the list. This cost is prohibitive for vertices with many neighbors, as is common in scale-free graphs.

This chapter appeared as "Dynamic Graphs on the GPU" published at IPDPS 2020 [7]

- To eliminate this $O(n)$ cost, the adjacencies can instead be stored as a sorted list. These expensive operations become $O(\log n)$ in the size of the list, but now the data structure must maintain the list in sorted order, which incurs a significant cost.

From a performance standpoint, neither alternative is acceptable. Now, some graph operations can be implemented with high performance on an unsorted list, and thus for a subset of graph workloads, a list-based data structure may deliver acceptable performance. But many graph operations cannot be implemented without paying the high cost of a full search of an unordered list or the maintenance cost of preserving sorted order.

We believe that existing dynamic GPU graph data structures like faimGraph [61] and Hornet [17] are suboptimal when considering real-world dynamic graph scenarios. Truly dynamic data structures need to support continuous modifications not only from running the algorithm (e.g., edge deletion in k-truss), but also from a flowing stream of edge and vertex insertions and deletions. While a reasonable first step, the experiments presented in the faimGraph and Hornet works lack the true dynamism we expect in real world scenarios. Their chosen approaches both rely heavily on potentially expensive sorting operations necessary for vertex and edge deduplication. In general, duplicate entries lead to incorrect graph analytic results for the many graph primitives where idempotence does not apply (e.g., triangle counting or betweenness centrality).

We show that using a more sophisticated data structure (e.g., a hash table), we achieve better performance compared to list-based techniques provided by alternative data structures (e.g., faimGraph or Hornet). A major reason for our superior performance is the fast query rates that hash tables offer and their ability to ensure uniqueness while performing updates. In contrast, list-based data structures require explicit sorting for deduplication to maintain uniqueness. To evaluate our data structure, we integrate it into the Gunrock GPU graph analytics framework [59] and compare it against other dynamic graph data structures. The resulting overall performance of our data structure on a range of workloads and applications, particularly on insertions, is superior to existing alternatives and also allows reasonable tradeoffs dependent on the selected workload.

Our contributions in this work are:

1. A high-performance hash table based dynamic graph data structure that supports ex-

tremely high rates of insertions and deletions (Sections 5.3 and 5.4);

2. An evaluation strategy that defines a set of workloads to benchmark a dynamic graph data structure (Section 5.5); and

3. Exploring the use of a dynamic graph data structure in applications while maintaining an updated graph (Section 5.6).

5.2 Background and Previous Work

5.2.1 Background

Consider a directed weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{W})$[6], where $\mathcal{V}$, $\mathcal{E}$, and $\mathcal{W}$ represent the vertex set, edge set, and edge weight set respectively. For each arbitrary vertex $u \in \mathcal{V}$, we represent its outgoing neighbors (adjacency list) with $\mathcal{A}_u$. $e = \langle u, v, w \rangle \in \mathcal{E}$ represents an edge from vertex $u \in \mathcal{V}$ to $v \in \mathcal{V}$ with weight w.

It is common to use an adjacency matrix with $|\mathcal{V}|^2$ elements to represent dense graphs. Updating an adjacency matrix is trivial. However, if $\mathcal{G}$ is a sparse graph (i.e., $|\mathcal{E}| \ll |\mathcal{V}|^2$), then the adjacency matrix is largely empty and, in practice, requires far too much memory for large graphs. Several alternative sparse graph representations exist. For example, *Compressed Sparse Row* (CSR), *Compressed Sparse Column* (CSC), and *Coordinate list* (COO) all offer better memory efficiency than the equivalent sparse adjacency matrix.

In CSR, for instance, $\mathcal{A}_u$ is stored as an array of values (weights) and an array of destination vertex IDs. Then all adjacency lists (arrays of size equal to the out-degree of each vertex d_u, where $\sum_{u \in \mathcal{V}} d_u = |\mathcal{E}|$) are concatenated to form two large arrays of size $|\mathcal{E}|$. A ternary array (i.e., row pointers) marks the start and end index of each adjacency list (i.e., each vertex's neighbors). CSR's representation requires $O(|\mathcal{E}| + |\mathcal{V}|)$ elements, which is considerably more memory efficient than the $O(\mathcal{V}^2)$ required for adjacency matrix. However, since it is a packed data structure, it is not possible to update it (i.e., delete or insert a new edge or vertex) without rebuilding the entire data structure.

[6]In general, we would like to support arbitrary meta-data assigned to each vertex or edge ($\mathcal{D}_\mathcal{V}$ and $\mathcal{D}_\mathcal{E}$). Here, for the sake of clarity, we assume $\mathcal{W}$ represents any sort of meta-data associated with vertices or edges.

In this work, we design a graph data structure that is both memory-efficient (like CSR) and also update-friendly (like the adjacency matrix). Suppose $\mathcal{A}_u$ represents the adjacency list of a particular vertex $u \in \mathcal{V}$, where $\mathcal{A}_u$ contains destinations of all outbound edges connected to u. Then a dynamic graph data structure should support the following operations:

1. Retrieving the adjacency list of vertex u: returns $\mathcal{A}_u$ if it exists, $\perp$ otherwise.
2. Inserting a new vertex u: Insert(u), where $\mathcal{A}_u$ is initialized with all connected vertices to u.
3. Deleting a vertex u: Delete(u), where $\mathcal{A}_u$ can no longer be located. $\mathcal{A}_u$ is deleted. All edges that have u as their destination should be removed either immediately or in a lazy fashion. After a deletion, no edge query involving u may have a false positive result. Querying $\mathcal{A}_u$ returns no edges.
4. Inserting a new edge: Insert$(\langle u, v, w\rangle)$, where $\mathcal{A}_u$ is first located, then a new pair $\langle v, w\rangle$ is inserted into $\mathcal{A}_u$.
5. Deleting an edge: Delete$(\langle u, v, w\rangle)$, where $\mathcal{A}_u$ is first located and then the pair $\langle v, w\rangle$ is deleted.[7]

We assume that for each vertex $u \in \mathcal{V}$, the adjacency list $\mathcal{A}_u$ is stored in a data structure that supports the following three operations:

1. Search$_{\mathcal{A}_u}(v)$: search through adjacency list $\mathcal{A}_u$ and returns $\langle v, w\rangle$ if present, $\perp$ otherwise.
2. Insert$_{\mathcal{A}_u}\langle v, w\rangle$: inserts a new entry $\langle v, w\rangle$ into the adjacency list $\mathcal{A}_u$.[8]
3. Delete$_{\mathcal{A}_u}(v)$: delete any entries $\langle v, w\rangle$ from the adjacency list $\mathcal{A}_u$.

In general we assume that all of the operations are batched and performed in a *phase-concurrent* fashion (i.e., updating the data structure does not happen concurrently with any kind of read-only search query from the data structure).

[7] One can define more general edge deletion operations such that all instances of $\langle u, v, \cdot\rangle$ are deleted regardless of their weights. This version might be useful if we allow multiple edges from a source to a destination, each with a different weight or meta-data.

[8] Duplicates are not allowed: first search for v (i.e., search$_{\mathcal{A}_u}(v)$), and replace a previously inserted element if it exists. Otherwise, insert a new pair.

5.2.2 Previous Work

The major challenge for an efficient dynamic graph data structure on the GPU is the design of the adjacency list data structure to best accommodate potential updates. Memory management of this data structure is inherent in this challenge. Sha et al. proposed GPMA as a GPU-friendly data structure for dynamic graphs [55] based on the Packed Memory Array data structure (PMA) [13]. PMA is a kind of balanced binary search tree, where nodes are sorted arrays with some anticipated empty gaps to support potential updates. PMA uses lower and upper bound density thresholds for each node, and these thresholds are used to make decisions to either copy a node into a larger newly allocated node (high density), or properly merge multiple nodes into a single node (low density). In GPMA, a batch of updates is first sorted. The sorted batch is further partitioned into several continuous parts, where each part will only belong to a single node in the tree. Then each node is properly updated based on its partition's size in three granularities of warp/block/device. Sha et al. proposed a method to store a CSR format for a graph in a GPMA data structure. There is little discussion of memory management. Most of the experiments on updating the data structure are around edge insertions, but lazy edge deletions are also briefly discussed.

Hornet [17] divides the allocated available memory into blocks that can store a number of edges up to a specific power of two. Initially an adjacency list is stored inside the smallest power-of-two memory block that can contain it. During edge insertion, if the newly inserted edges exceed the capacity of a memory block, the vertex adjacency list is copied to the next smallest power-of-two memory block. For each array of blocks, a B-Tree tracks the free and used ones. Memory management is done on the CPU. Hornet achieves a compact representation for an adjacency list at the expense of memory fragmentation. Moreover, it supports vertex insertion (or deletion) through a series of corresponding edge insertions (or deletions).

faimGraph [61] uses a single memory pool on the GPU for both the data structure and the algorithm that solves a graph problem. In contrast to Hornet, faimGraph's memory management is entirely on the GPU. Queues are used for memory reclamations of pages and deleted vertex IDs. faimGraph maintains a mapping between vertex IDs on the GPU and CPU. It also offers both structure-of-arrays (SoA) and array-of-structures (AoS) representations to store edge data,

where the former is used for edges with a single property and the latter is used for edges with multiple properties. Memory pages configurable in size contain pointers to next pages when the adjacency list size exceeds a single page size. Using different GPU and CPU vertex IDs allows for flexible memory reclamation.

Edge duplication is not allowed in either Hornet or faimGraph. Both take preventive measures during updates to ensure edge uniqueness in the data structure.

We discuss and compare our results to faimGraph and Hornet. faimGraph is the state-of-the-art dynamic graph data structure and Hornet is a maintained graph processing library. In terms of similarities, our work is similar to faimGraph as our hash table is represented using fixed-size memory pages. In other words, if the hash table consists of a single bucket, which is true in road-network-like graphs (but not scale-free graphs), our work and faimGraph are similar. Similar to Hornet, addition of new vertices in our system requires overallocation of the graph data structure capacity to avoid reallocation, but we keep in mind that the cost of reallocation only requires copying of adjacency-list pointers and not the entire data structure.

5.3 Our GPU Dynamic Graph

The key challenge in designing a dynamic graph data structure is storing per-vertex adjacency lists. Our graph representation uses a separate data structure for each vertex adjacency list together with associated handles to reach those adjacency lists as necessary to perform various operations. The choice of the data structure used to store adjacency lists must be based on tradeoffs between what operations the data structure supports, the performance of individual operations, and the requirements of the graph library for solving specific problems. For instance, the simplest data structure choice is to use a variable-sized list data structure per adjacency list, with the designer either choosing to keep that list unsorted or maintain it as a sorted list. Hornet [17] embodies this approach. This representation is the most compact, but incurs a large maintenance overhead when compared to other options. Another possibility is the approach taken by faimGraph [61], which relaxes the variable-sized list constraint and instead breaks lists into fixed-size pages to simplify the maintenance of the data structure. Our position is that these primitive GPU data structures can be replaced by more sophisticated ones such as hash

tables [4] or B-Trees (Chapter 3), depending on the requirements of the problem in terms of the performance and availability of data structure operations. In this work our goal is to provide a high throughput of both updates and lookups, thus we pick hash tables.

Advantages of a hash table representation The primary advantage of hash tables is their efficient operations (both mutations and queries). Supporting efficient queries is an essential requirement in a graph data structure. Not only do graph applications perform read-only queries into graphs, but even mutation operations typically incorporate queries. For instance, an insertion while maintaining unique edges first requires a query determining whether the edge exists or not, followed by the insertion process itself (this is simply writing the new pair into an empty location). For a list-based data structure, performing a query operation is either $O(n)$ (for an unsorted list) or $O(\log n)$ (for a sorted list) in the size of the list. For a hash table with a suitable load factor, queries are instead $O(1)$. In a dynamic setting, hash table performance can decrease as the chain-length increases (i.e., load factor increases). In practice we can maintain low-cost metrics per vertex to determine the chain-length and periodically perform rehashing if it exceeds a given threshold.

Figure 5.1 shows a high-level representation of our graph data structure, which is divided into two parts:

Vertex dictionary. We store vertices, $\mathcal{V}$, in a simple fixed-size array, indexed by vertex ID. The array size can be increased if needed, but frequent reallocation should be avoided to minimize the performance costs commonly associated with memory allocation. Selecting a large-enough initial capacity based on graph problem requirements ensures good performance during vertices insertion.

Adjacency lists. We use one hash table per vertex to store its associated adjacency list $\mathcal{A}_u$. Given a load factor and number of edges in an adjacency list, we calculate the number of buckets in a hash table. Note that the load factor, directly related to the number of buckets, provides a tradeoff between two main operations: 1) reading a complete adjacency list associated with a vertex and 2) performing an edge-exists query in a vertex's adjacency list. In practice we select a single load factor for all hash tables (in this work, we use a load factor of 0.7). This is not strictly necessary, but determining an ideal load factor per-vertex (per-hash-table) a priori

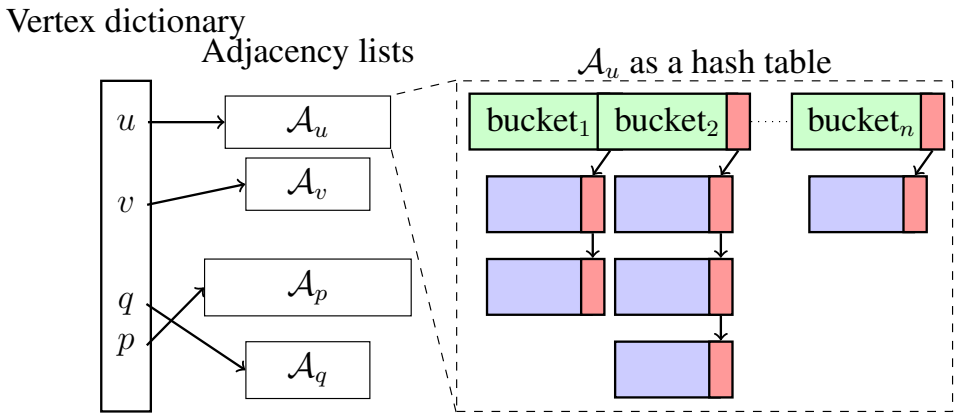


Figure 5.1: High level schematic of our graph data structure. Each adjacency list is represented using a slab hash. The number of base slabs per adjacency list depends on the load factor used per adjacency list. Base slabs are statically allocated in consecutive memory locations, while the slabs used to resolve collisions are allocated dynamically and reached through pointers.

is difficult. Our dynamic graph data structure can make use of given connectivity information along with the choice of the load factor to determine the number of necessary buckets to allocate. This decision results in significant performance gains by reducing memory allocation overhead. Using a dynamic memory allocator, any hash table can dynamically allocate additional slabs as needed (Figure 5.1). If the connectivity information for a vertex is not available, we construct a hash table with a single bucket for this vertex.

5.4 Implementation

Our dynamic graph data structure's adjacency lists are stored as hash tables. In this work, we use *Slab Hash*, a dynamic hash table data structure for the GPU [4], as the basis of our underlying hash tables.[9] We have significantly improved the functionality of the original slab hash in order to meet our requirements.[10] We offer two variants of our dynamic graph data structure. One uses Slab Hash's *concurrent map*, which should be used if storing a value per edge is required. The second variant uses Slab Hash's new *concurrent set*, which should be used if edge values are not required. Any hash table design can be used for this underlying data structure, as long as it is efficient in both searching (for queries) and updating the data structure itself (for insertions

[9] `https://github.com/owensgroup/SlabHash`

[10] The original slab hash only provided a *concurrent map* data structure, without any restrictions on duplicate keys. To name a few of our recent additions: maintaining key-uniqueness, proper iterator access, and design and implementation of a new concurrent set (keys only, and no values).

and deletions). In the end, the performance of our graph data structure directly depends on the performance of its underlying hash tables.

We integrate our dynamic graph data structure into the Gunrock GPU graph analytics framework [59]. In order to take advantage of the high-performance operations Slab Hash offers, all our operations are implemented based on the Warp Cooperative Work Sharing (WCWS) strategy [4]. In WCWS, each thread has an independent task assigned to it, but all threads within a warp cooperate with each other to collectively perform one independent task at a time. This is the right design decision because it better matches the memory access pattern desired by the GPU hardware (coalesced memory accesses), and hence it provides better performance for updates. On the downside, it requires all threads within a warp to be active. In other words, an operation on the data structure cannot be performed within a branch where threads (in a warp) diverge when executing it.

5.4.1 Memory Management

Our memory management is divided into two parts: 1) vertex dictionary memory, and 2) hash table memory.

Vertex dictionary memory. Defining a graph requires defining the graph's vertex capacity. The vertex dictionary stores pointers to the hash table associated with each vertex's adjacency list. When inserting more vertices than the vertex dictionary's capacity, we copy the vertex dictionary to a new memory location after increasing its capacity. This process only requires shallow copying of the pointers to each of the hash tables (including pointers to the hash tables associated with the new vertices).

Adjacency list hash table memory management. Constructing a hash table requires choosing and allocating a number of buckets (base slabs) that are required for insertion processes. The initial number of buckets for a vertex u is $\lceil |\mathcal{A}_u|/(lf \times B_c) \rceil$, where *lf* is the load factor and B_c, the bucket capacity per slab, is either 15 or 30 for Slab Hash map or set, respectively. During insertion, if a bucket's slab becomes full (capacity achieved), Slab Hash dynamically allocates a new slab for that bucket that is singly linked to the tail of the list; a dynamic memory allocator handles these dynamic allocations [4]. Only when we perform vertex deletion (essentially this deletes an entire hash table) do we free this dynamically allocated memory (Section 5.4.4).

Our graph data structure handles the memory allocation required for the initial buckets by statically allocating all the memory required for the initial buckets in bulk. This is more desirable than requiring each hash table to independently allocate a small number of buckets with different `cudaMalloc` calls. We initialize a vertex's hash table with its initial number of buckets, the memory address for its first bucket, and the number of neighbors (to zero). In cases where the number of neighbors is not defined, we allocate a single bucket.

5.4.2 Query Operations

To iterate over a vertex's adjacency list, we provide a vertex adjacency list iterator. For a given vertex, the iterator loops over all of the hash table buckets associated with the vertex as well as additional slabs used to resolve hash collisions. The iterator loads one slab at a time and moves from one slab to the next using a `next` operator.

We also provide an `edgeExist` query that checks if the destination v of a given pair $\langle u, v \rangle$ exists in the hash table associated with u. `edgeExist` simply performs a search query [4] in u's hash table.

5.4.3 Edge Operations

We interpret edge operations (insertion and deletion) as modifications to the source vertex's adjacency list. As discussed in Section 5.3, we use a hash table to represent each vertex's adjacency list. We discuss this in more depth below in the context of a directed graph. In an undirected graph, inserting (or deleting) an edge between a source and destination is similar but also requires an operation on the edge in the other direction. Our semantics for edge operations follows the semantics for hash table operations, which we discuss below.

Edge insertion. Algorithm 5.1 shows high-level pseudocode for an edge insertion. We assume that each thread has a single edge to insert. Initially, in line 3, we ensure no self-edges are allowed. A work queue is constructed within a warp (line 4) through a ballot instruction on all threads' remaining tasks. All threads locate the next task to perform (through finding the first set bit in the work queue as in line 5) and then get the corresponding source vertex of the chosen task (through a shuffle instruction as in line 6). Since multiple edges within a warp might share the same source vertex (lines 7 and 8), all these insertions are grouped together to

be performed in one single coalesced call to the hash table associated with the source vertex. We implemented and used a new slab-hash *replace* operation to ensure key-uniqueness in the hash table (i.e., unique destination vertices). In this operation, if a key (a destination vertex) already exists in the hash table, it will be replaced with the most recent value. Otherwise, a new key-value pair is added to the hash table. If the batch of edges contains the same unique edge but with different weights, only the most recent edge and its weight will be stored in the graph. The replace operation returns a boolean value indicating whether a new key (i.e., an edge) was added to the hash table or the key previously existed and was hence just replaced. We use this returned boolean variable to maintain an exact number of edges per vertex (population count on all successful additions within a warp in line 10). The thread whose task was just completed, as well as all threads that shared the same source vertex (i.e., the coalesced insertion group), mark themselves as completed (line 11). We repeat this procedure until the work queue is completely empty, i.e., no more edges remain to be inserted.

Edge deletion. Edge deletion is similar to edge insertion with two major differences: 1) instead of using the replace operation in Algorithm 5.1, we use the *delete* operation; 2) the delete operation also returns a boolean variable as to whether the key already existed. This boolean variable is used to decrement the number of edges that belongs to the adjacency list of the vertex. Note that in order to maintain uniqueness within the slab hash (due to how insertion/replace operations are designed and implemented), deleted edges (i.e., keys) are only marked as deleted (i.e., by replacing a key with a tombstone) and not explicitly removed. Tombstones are disregarded in edge insertion (as if that particular location is not empty). Tombstones can later be completely flushed out of the data structure, if required. Together, these design decisions ensure that empty locations can only exist at the end of each bucket's list in the slab hash. Moreover, not overwriting tombstones results in faster insertion rates (since new edges are only added to the end of the bucket's linked list). This comes at the expense of having unused memory locations. A different approach would be to break down the insertion process into two stages: 1) traversing the bucket's linked list to ensure uniqueness, then 2) for unique keys, a follow-up insertion that overwrites tombstones. We use the former approach during insertions, but the latter could be used to optimize for memory usage on the expense of decreased insertion throughput.

5.4.4 Vertex Operations

Vertex insertion. We define a vertex insertion operation as the operation of inserting edges connected to a vertex that has an empty adjacency list. As discussed earlier, if the new vertex count exceeds the capacity of the vertex dictionary, we first extend the vertex dictionary. Once the vertex is entered into the dictionary, we then insert all attached edges using Algorithm 5.1.

Vertex deletion. Algorithm 5.2 summarizes this process for an undirected graph. Each warp deletes one vertex at a time. Because each vertex in a multi-vertex deletion operation may have a different number of edges, a straightforward implementation would suffer from load imbalance. We address this imbalance with a simple technique. We maintain a queue of deleted vertices with an atomic counter (line 4). A single thread inside the warp acquires a new vertex from the queue (line 3). The new vertex queue location is broadcast for all threads in the warp (line 6). The vertex-deletion kernel only exits after deleting all the required vertices (line 8). A warp reads the vertex index (line 10) and requests an edge iterator over the all the slabs associated with the vertex (line 11). Using the iterator, we loop over all of the vertex destinations and delete the vertex from their adjacency lists (line 16). Additionally, all dynamically allocated memory (i.e., memory used to resolve collisions) is freed and reclaimed by the memory allocator (line 19). Finally, the count of edges connected to the vertex is set to zero (line 22). Statically allocated memory is not reclaimed. For a directed graph, the only requirement is to free the memory. To clean up, we end with a follow-up lookup and delete all of the deleted vertices in all of the hash tables.

5.5 Evaluation Strategy

Our community has not yet defined consistent standards for evaluating a dynamic graph data structure. In part this is because of the very recent development of these data structures as a topic for study. As a result, we lack a broad set of applications or workloads that require dynamic data structures. We believe the evaluation we present below improves on previous work by identifying and characterizing a set of operations, workloads, and applications that together encompass the wide range of use cases that will be addressed with dynamic graph data structures.

Algorithm 5.1 Graph edge insertion.

```
 1: procedure INSERTEDGES(GpuGraph graph, Edges edges)
 2:     thread_edge ← edges[threadIdx]
 3:     to_insert ← thread_edge.src != thread_edge.dst
 4:     while work_queue ← ballot(to_insert) do
 5:         current_lane ← find_first_set_bit(work_queue)
 6:         current_src ← shuffle(thread_edge.src, current_lane)
 7:         same_src ← thread_edge.src == current_src
 8:         success ← graph[current_src].replace(thread_edge, same_src & to_insert)
 9:         added_count ← popc(ballot(success))
10:         graph[current_src].incrementEdgesCount(added_count)
11:         if same_src & to_insert then
12:             to_insert ← false
13:         end if
14:     end while
15: end procedure
```

We believe a comprehensive evaluation requires three components:

Operations Dynamic data structures support particular operations, e.g., edge and vertex deletion and insertion. We enumerate these operations and measure their throughput.

Workloads Because dynamic graph data structures on the GPU are not yet in significant use, applications that use them are few. However, we present a set of workloads—common patterns of how we will use the data structure—that we believe will underlie future applications in this area.

Applications Finally, prior work has identified particular applications on which they evaluate their data structure. We evaluate our work on these specific applications as well.

5.5.1 Low-Level Operations on a Dynamic Graph Data Structure

We begin with measuring throughput for the important low-level operations on our data structure:

Edge Insertion and Deletion. Starting from a static graph stored in a dynamic data structure, we measure the throughput of edge insertion and deletion operations for different batch sizes. Edges are inserted or deleted between existing vertices in the graph. Duplicate edges

Algorithm 5.2 Graph vertex deletion.

```
1: procedure DELETEVERTICES(GpuGraph graph, Vertices vertices, Count count, Queue
   queue)
2:     while true do
3:         if laneId == 0 then
4:             queueId ← atomicAdd(queue, 1)
5:         end if
6:         queueId ← shuffle(queueId, 0)
7:         if queueId ≥ count then
8:             return
9:         end if
10:        warp_vertex ← vertices[queueId]
11:        vertex_edges_it ← GpuGraph::EdgeIterator(warp_vertex)
12:        while vertex_edges_it.next() do
13:            lane_dst ← vertex_edges_it.getDst(laneId)
14:            for lane in lanes do
15:                current_dst ← shuffle(lane_dst, lane)
16:                graph[current_dst].delete(warp_vertex)
17:            end for
18:            if vertex_edges_it.current() is not base slab then
19:                free(vertex_edges_it.getAddress())
20:            end if
21:        end while
22:        graph[warp_vertex].setEdgesCount(0)
23:    end while
24: end procedure
```

are allowed within a batch and across the batch and the graph. The graph data structure only maintains unique edges.

Vertex Insertion and Deletion. Similar to edge insertion and deletion, we start from a static graph and measure the throughput of inserting and deleting vertices in different batch sizes.

5.5.2 Workloads on a Dynamic Graph Data Structure

To evaluate a dynamic graph data structure we propose the following different set of workloads. Each one of these workloads targets a different scenario, not specific to any particular application, that we expect to be a pattern that can be used in real-world applications.

Static Workloads / Bulk-build. We start by comparing our dynamic graph data structure to other alternatives in a static setting. Specifically, we evaluate the performance of building a static graph. We assume that the number of edges per vertex and the number of vertices is known a priori.

Given a static graph, we measure the time required for building the graph in bulk and compare this with previous work. We assume that the input is given in a COO format (i.e., a list of edges each defined by source vertex, destination vertex, and edge value).

Dynamic Workloads / Incremental Build. Starting with an empty graph, we incrementally build a graph using different batch sizes. In general, to avoid memory reallocation we assume that a suitable vertex capacity (i.e., maximum number of vertices) is known.

5.5.3 Applications with a Dynamic Graph Data Structure

We again emphasize that the existing set of graph applications that use dynamic data structures is small. We thus primarily evaluate work on the applications developed in and described by previous work where possible. For the *static application* case, we use static triangle counting to evaluate and compare our dynamic graph data structure performance to static CSR [59] and the two dynamic graph representations [17, 61]. In the *dynamic application* case, we evaluate based on a dynamic triangle counting application that performs triangle counting after each batch insertion. Note that in this work we only focus on the performance of the dynamic graph data structure. Optimizing a static or dynamic graph algorithm (e.g., triangle counting) is beyond the scope of this work.

5.6 Results

We evaluate and compare our dynamic graph data structure to Hornet[11] and faimGraph.[12] faimGraph is the state of the art in dynamic GPU graph data structures. Hornet is an actively maintained GPU data structure for sparse graphs and matrices. In our tests, faimGraph's page size is configured to be 128 bytes to match our slab page size. Our tests do not require that either faimGraph or Hornet maintain a sorted adjacency list. All of our measured performance tim-

[11] https://github.com/hornet-gt/hornet/tree/a5c754d9616f54404a7b2a15c11143d52a346ab9
[12] https://bitbucket.org/mwinter92/faimgraph

Table 5.1: Graph datasets.

Dataset	Vertices	Edges	Degree			
			Min.	Max.	Avg.	σ
luxembourg_osm	114K	239K	1	6	2.1	0.41
germany_osm	11.5M	24.7M	1	13	2.1	0.51
road_usa	23.9M	57.71M	1	9	2.4	0.85
delaunay_n23	8.4M	50.3M	3	28	6.0	1.33
delaunay_n20	1M	6.3M	3	23	6.0	1.33
rgg_n_2_20_s0	1M	13.8M	0	36	13.1	3.62
rgg_n_2_24_s0	16.8M	265.1M	0	40	16.0	3.99
coAuthorsDBLP	299K	1.9M	1	336	6.4	9.80
ldoor	952K	45.5M	27	76	47.7	11.97
soc-LiveJournal1	4.8M	85.7M	0	20K	17.2	50.65
soc-orkut	3M	212.7M	1	27K	70.9	139.72
hollywood-2009	1.1M	112.8M	0	11K	98.9	271.70

ings for all libraries only include the time to perform the operation and do not include the time required to transfer memory between CPU and GPU. We perform our benchmarking using the datasets shown in Table 5.1 on an NVIDIA TITAN V (Volta) GPU with 12 GB DRAM and an Intel Xeon CPU E5-2637.

5.6.1 Operations

Batched Edge Insertion. We perform a batched edge insertion (Section 5.5.1) for different batch sizes and measure the average throughput for all the given datasets. faimGraph only supports batch updates of sizes less than 1M. Table 5.2 compares our edge insertion throughput to Hornet and faimGraph. Our speedup ranges between 5.8–14.8x compared to Hornet and 3.4–5.4x compared to faimGraph.

Batched Edge Deletion. Similar to batched edge insertion, we run a batched edge deletion. Table 5.3 shows the result of this benchmark. This is where Hornet performance becomes competitive with ours. Deletion is a simple process and does not require cross-duplicate checking between the graph and the input batch. Note that for small datasets, the true number of deleted edges (i.e., unique edges within the batch) is much lower than the number of randomly gener-

Table 5.2: Mean edge insertion rates (in MEdge/s) for different batch sizes.

Batch size	Hornet	faimGraph	Ours
2^{16}	33.67	92.47	501.33
2^{17}	44.71	133.97	513.56
2^{18}	51.43	157.15	591.06
2^{19}	70.81	188.98	641.25
2^{20}	83.54	—	664.52
2^{21}	97.41	—	658.09
2^{22}	110.89	—	646.01

Table 5.3: Mean edge deletion rates (in MEdge/s) for different batch sizes.

Batch size	Hornet	faimGraph	Ours
2^{16}	91.73	111.71	640.63
2^{17}	159.69	112.96	886.92
2^{18}	259.31	171.75	947.60
2^{19}	377.79	257.66	939.98
2^{20}	537.73	—	988.85
2^{21}	739.82	—	1,007.16
2^{22}	1,024.87	—	1,015.47

ated edges, hence resulting in less work in general. Our performance is as fast as Hornet for a large batch size and almost 7x faster for a smaller batch size of 2^{16}. Our deletion rates are between 3.6–7.8x faster than faimGraph's.

Vertex Deletion. Beginning with an undirected graph, we delete a batch of vertices and measure the throughput of vertex deletion. Table 5.4 shows the throughput for different batch sizes averaged over four datasets: soc-orkut, soc-LiveJournal1, delaunay_n23, and germany_osm. Our vertex deletion throughput is between 8.9–12.2x faster than faimGraph (Hornet does not implement vertex deletion). Both we and faimGraph delete vertices from neighbor adjacency lists and free the memory used to store the vertex adjacency list, but faimGraph implements one operation that we do not: it places the deleted vertex into a vertex queue and can thus reuse

Table 5.4: Mean vertex deletion throughput (in MVertex/s) for different batch sizes.

Batch size	faimGraph	Ours
2^{16}	0.44	5.35
2^{17}	0.71	9.23
2^{18}	1.12	12.66
2^{19}	1.62	19.21
2^{20}	2.96	26.49

identifiers of deleted vertices during subsequent vertex insertions. This allows faimGraph to be more memory efficient compared to our approach. It would be straightforward to implement the same strategy with our data structure but we have not yet done so. If we compare only delete and free operations common to both data structures, our speedup is 8.5–11.56x over faimGraph. As with query operations, the dominant factor in vertex-deletion performance is looking up a deleted vertex in its neighbors' adjacency lists; this is faster in a hash table than in a list.

5.6.2 Workloads

Bulk Build. We perform the bulk build benchmark from Section 5.5.2. Bulk build is simply inserting all edges from a graph into the graph data structure in one single batch. We implemented the bulk build functionality in Hornet only. Table 5.5 shows the time required to bulk build the datasets. For two datasets—rgg_n_2_24_s0 and soc-orkut—Hornet runs out of memory. We believe that this is due to the memory overhead of sorting and duplicate checking. Our dynamic graph data structure is 2–30x faster. Note that for a large dataset, hollywood-2009, 45% of Hornet's insertion time is spent in duplication checking alone, which is the same time as our entire build.

Incremental Build. In this benchmark we begin with an empty graph and incrementally insert edges (Section 5.5.2). The goal is to test and measure the edge throughput when building a graph data structure given a known bound on the number of vertices a priori, but an unknown number of edges. For our dynamic graph data structure, this means that each hash table is given only one bucket (i.e., a single linked list). Note that in this experiment our data structure is similar

Table 5.5: Bulk-build elapsed time (ms).

Dataset	Hornet	Ours
luxembourg_osm	5.562	0.184
germany_osm	330.311	12.407
road_usa	644.308	27.910
delaunay_n23	273.532	19.590
delaunay_n20	37.68	2.494
rgg_n_2_20_s0	37.084	5.053
rgg_n_2_24_s0	—	97.886
coAuthorsDBLP	11.672	0.835
ldoor	46.486	15.936
soc-LiveJournal1	179.879	26.176
soc-orkut	—	39.907
hollywood-2009	90.705	42.387

to faimGraph, but differs from Hornet in our use of a linked list of pages versus a single block containing the adjacency list. For our hash table based graph data structure, this represents the worst-case scenario.

Table 5.6 shows the average throughput for building graphs with a similar number of edges (ldoor, delaunay_n23, road_usa, soc-LiveJournal1) using different batch sizes. We implemented incremental build in Hornet only. On average our data structure is 5x faster than Hornet. For the two low-variance graph datasets (delaunay_n23 and road_usa), our speedups are between 15–25x. We believe that the main reason for our performance advantage is that Hornet maintains its adjacency list in a single fixed-size block. When an added edge exceeds the size of the block, the entire adjacency list must be copied to an existing or newly allocated empty block of the appropriate size. In contrast, our linked lists avoid copying and simply allocate new pages to accommodate new edges as needed. With Hornet, copying into larger-sized blocks is expected to happen more often for low-variance datasets. For high-variance datasets, Hornet's doubling adjacency list strategy becomes more efficient because the need for copying to new blocks decreases. We see speedups of 1.6–2.5x for the ldoor dataset, but for the soc-LiveJournal1 dataset our throughput is 0.92x slower.

Table 5.6: Incremental build mean edge insertion rates (in MEdge/s) for different batch sizes.

Batch size	Hornet	Ours
2^{20}	164.44	841.31
2^{21}	176.96	945.64
2^{22}	184.75	993.82

5.6.3 Applications

We pick triangle counting as a simple application to explore the interaction between a dynamic data structure and solving a graph problem. Our goal here is not to provide an optimal solution to the dynamic triangle counting problem; rather, we would like to explore the performance of triangle counting's main query operation, intersect. The intersect operation inputs two adjacency lists and counts the number of edges in common. If the adjacency list is stored as a list, to perform an intersect operation efficiently, the list must be sorted. The hash-based data structure we present here does not have this constraint. We thus compare the cost of maintaining the sorted list-based data structures used by Hornet and faimGraph with our approach.

Static. In this experiment, we compare dynamic graph data structures to solve the static graph problem of triangle counting. Since triangle counting only requires maintaining the destinations of edges and not their values, we use the set variant of the dynamic graph data structure. The Hornet and faimGraph data structures require sorted adjacency lists to efficiently compute set intersections. Table 5.7 shows the time required to perform triangle counting on different datasets. On most datasets, our dynamic data structure performs worse than either Hornet or faimGraph, because their intersection operation between two sorted lists is efficient. They find the starting location of one list in the other and then (serially) walk to the end of the lists, accumulating the number of matches. While this exhibits little parallelism, it is cheaper and faster than a hash-table-based solution. In our hash table representation, we perform an `edgeExist` query for all edges.

Note, the sort in the list-based data structures is not free, and is not counted in the results above. Table 5.8 summarizes sort cost on these datasets. Hornet does not provide a GPU

Table 5.7: Static triangle counting time in ms.

Dataset	Hornet	faimGraph	Ours
luxembourg_osm	0.57	1.01	0.31
germany_osm	26.31	16.61	29.69
road_usa	51.57	39.19	66.20
delaunay_n23	25.74	21.14	56.38
delaunay_n20	3.35	2.92	6.43
rgg_n_2_20_s0	6.42	7.35	23.24
rgg_n_2_24_s0	154.75	165.86	493.6
coAuthorsDBLP	1.20	4.75	6.98
ldoor	22.09	48.04	222.07
soc-LiveJournal1	482.98	705.71	1526
soc-orkut	3832	8986	6758
hollywood-2009	9784	57311	11060

sort for their data structure, so we substitute CUB's segmented sort by key [48]. Interestingly, faimGraph's sorting is faster than CUB's when the maximum vertex degree of the graph is small, but for a large maximum vertex degree, faimGraph's sort is much slower than CUB's. These results raise the question of the overhead of maintaining a sorted Hornet or faimGraph data structure in order to perform a dynamic application that requires a sorted list, such as triangle counting. We further investigate this in the following section.

Dynamic. For this experiment we pick two datasets, one with a small largest-vertex degree (road_usa) and one with a large largest-vertex degree (hollywood-2009). We perform triangle counting after incrementally inserting edges five times. This scenario was not previously implemented in either Hornet or faimGraph; we implemented it for Hornet only. Table 5.9 shows the result of this experiment. For road_usa, our implementation offers a 1.8x speedup over Hornet's, largely due to our faster insertion. For hollywood-2009, although our insertion performance is around 6x faster than Hornet, Hornet's faster triangle counting is still fast enough to cover the cost of maintaining sorted adjacency lists. We are 0.9x slower than Hornet on this dataset.

Table 5.8: CSR-sort (with CUB) and faimGraph-sort time in ms. Sort time is in general comparable to (and often considerably larger than) the time for triangle counting (Table 5.7).

Dataset	Sort CSR	Sort faimGraph
luxembourg_osm	58.13	0.07
germany_osm	5260	4.84
road_usa	10875	12.65
delaunay_n23	3854	18.98
delaunay_n20	503.29	2.48
rgg_n_2_20_s0	496.85	8.62
rgg_n_2_24_s0	7753	178.37
coAuthorsDBLP	136.89	7.36
ldoor	442.15	175.12
soc-LiveJournal1	2226	20428
soc-orkut	1404	41833
hollywood-2009	540.30	8504

Table 5.9: Cumulative time (ms) required to perform triangle counting and inserting a batch of size 2^{22} into the graph.

Iter.	Ours			Hornet			Speedup
	Insert	TC	Total	Insert	TC	Total	
road_usa							
1		65.5	64.1		51.6	116.0	1.81
2	14.8	135.8	129.5	214.1	110.2	235.1	1.82
3	29.7	201.7	195.0	438.4	174.6	356.4	1.83
4	44.5	267.5	260.5	652.6	243.8	476.2	1.83
5	59.4	333.2	325.8	855.7	319.7	597.8	1.83
hollywood-2009							
1		11151	11151		9893	9893	0.89
2	12.4	22539	22551	73.4	19982	20056	0.89
3	24.9	33921	33946	149.0	30174	30323	0.89
4	37.3	45297	45335	229.3	40552	40781	0.90
5	49.8	56724	56774	313.0	51090	51403	0.91

5.6.4 Effect of the Load Factor on Our Graph Data Structure

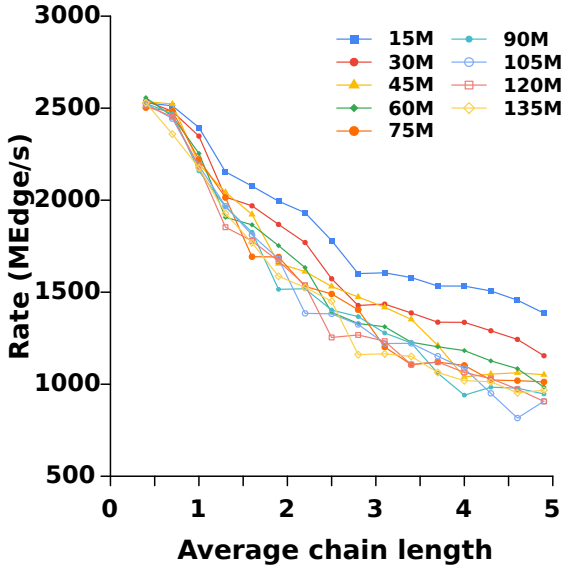
To measure the effect of load factor on our hash table, we perform two experiments. Figure 5.2 shows our first experiment where we manipulate the chain length of our hash tables by building a graph with the suitable average degree and load factor. As we expect, the insertion throughput of our data structure drops as the chain length increases. On the other hand, the memory utilization increases. This is due to the fact that buckets are now more full. Moreover, the amount of memory used decreases as the average chain length increases. This is due to the fact that fewer buckets are now needed. Figure 5.3 shows our second experiment. Similar to the first one, we explore the query performance as the average chain length increases (we use static triangle counting to provide the query workload). The results show the optimal average chain length for our hash table, which is around 0.7.

5.7 Conclusion and Future Work

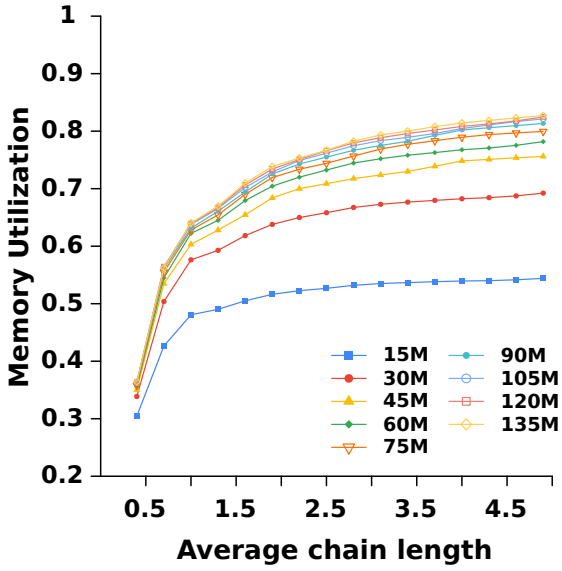
Our dynamic GPU graph data structure uses hash tables to represent per-vertex adjacency lists. This representation suits operations that require fast insertion, deletion, and edge lookups, and is superior in performance to previous work that focuses on list data structures for adjacency lists.

In regards to future work, we note that other data structures can be used to represent adjacency lists. For instance, a B-Tree (Chapter 3) provides a different set of operations as well as maintaining a sorted adjacency list, an optimization that is useful in certain graph algorithms. We also note that supporting hash tables with varying slab sizes may better suit load balancing in scale-free graphs where vertex degrees vary over several orders of magnitude. This would complicate the update procedure at the expense of providing better scalability in graph processing. Moreover, it would require a more complicated dynamic memory allocator design (compared to SlabAlloc used in slab hash [4]) that can support variable-sized memory allocations efficiently.

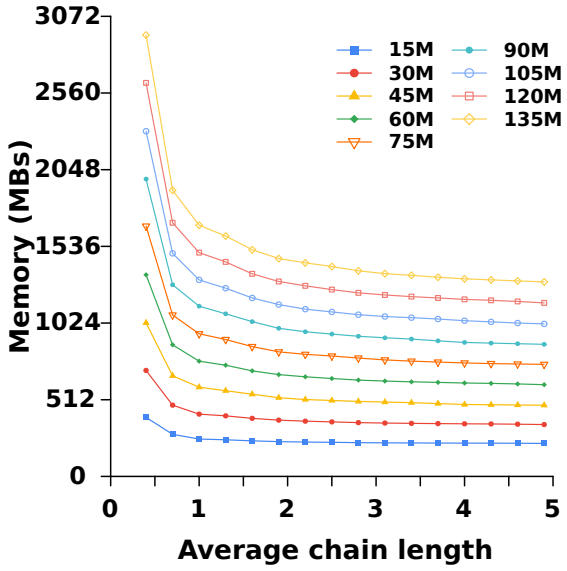
Although we only discussed phase-concurrent updates and queries in this work, both the slab hash and the B-Tree provide concurrent queries and updates. These concurrent operations can be exposed to a dynamic graph algorithm. The key challenge here is to carefully consider the semantics of these operations. Using data structures that support versioning (e.g., our



(a) Insertion Rate.



(b) Memory Utilization.



(c) Memory Usage.

Figure 5.2: For different directed RMAT graphs with 2^{20} vertices but different average degree (different number of edges), we build the graph using different load factors. () The insertion throughput drops by a factor of 2.5 if the hash tables have, on average, chains of length 5. On the other hand, the memory utilization increases (a) and the memory usage decreases (b) as the average chain-length increases.

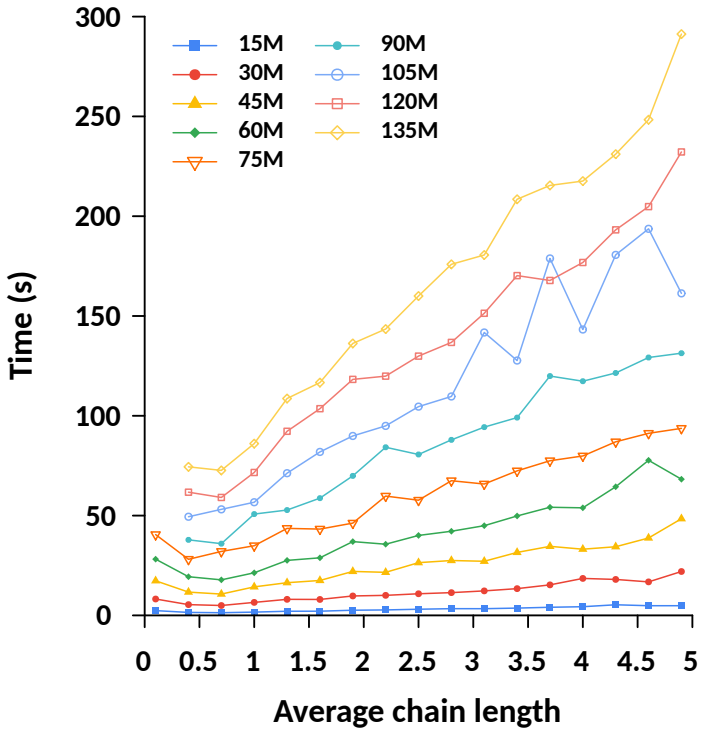


Figure 5.3: Static triangle counting performance for different undirected RMAT graphs with 2^{20} vertices, but different average degree using hash tables with different load factors. Our data structure achieves its optimal performance when the load factor is around 0.7.

Multiversion B-Tree) will allow us to provide the graph data structure users with meaningful semantics and a consistent view of the adjacency lists.

Chapter 6

Conclusion and Future Research Directions

6.1 Conclusion

When designing a fully-concurrent GPU lock-based data structure, the two main challenges are minimizing contention and efficiently performing the data structure traversals. To reduce contention, the two primary considerations a data structure design must consider are: 1) how to temporarily relax the data structure requirements to avoid excessive use of locks and allow concurrent operations, 2) how to minimize the number of locks an update operation needs. These two considerations are closely related, and they complement each other. The design must closely match the hardware memory system to achieve efficient traversal. The designer of a massively-parallel data structure must have deep knowledge about the hardware.

Defining abstraction to data structure operations using cooperative processing techniques allows the data structure to be easily composable and integrated into a more extensive system and special-purpose data structures. Making no assumptions on the batched operations size and avoiding using device-wide bulk primitives is essential to build a flexible, composable data structure.

Using these strategies, we showed how to build a fully-concurrent GPU B-Tree and then added snapshot support to it. Our design minimizes contention and is cache-aware, achieving memory throughputs higher than the peak DRAM bandwidth. Using snapshots, we solved the problem of linearizable multipoint queries and added the ability to maintain multiple versions

of the data structure. Moreover, we designed and built a safe memory reclamation scheme for the GPU to reclaim older snapshots. SMR is one of the most critical components that support concurrent data structures. Finally, we showed how this successful design enables composing a special-purpose data structure such as a graph.

Although there is no one formula to solving all concurrency-related problems, a critical design that appears in multiple successful concurrent data structure designs is relaxing the requirement of the data structure and being able to tolerate faults and correct them. The challenge is how to detect a fault to correct it. In this dissertation, we detected these faults and performed corrections in different ways. We used a node's high-key and side-link to detect and correct a B-Tree traversal. To improve deletion performance in a B-Tree, we allowed reading nodes from the incoherent L1 cache then forced a coherent read through the memory system. Although finding and correcting an operation depends on the specifics of the problem, the idea of tolerating faults and correcting them should be explored and considered in all massively parallel systems. This idea is similar to optimistic concurrency control but on a very fine granularity.

6.2 Future Research Directions

6.2.1 Near-Future Research Directions

We only scratched the surface of designing concurrent GPU data structures. We, alongside other researchers, built tools such as memory allocators and reclaimers that will aid us in building interesting data structures. We want to explore wait-free data structures, and we believe that these data structures can be efficiently implemented and designed with GPUs in mind. As we showed, CPU solutions do not scale to the GPU level of parallelism; therefore, it is essential to think about the high-level properties and guarantees that a data structure offers when building and designing a GPU solution.

Binary trees. A binary search tree can be easily ported to the GPU using current GPU libraries; however, one-to-one porting will perform poorly as the traditional CPU design of a binary tree suffers from branch and memory divergence. An efficient GPU leaf-oriented binary search tree design would use immutable cache-line-sized leaf nodes while using intermediate nodes with the binary branching factor. This design is optimizing for accessing the leaf nodes—

where most of the tree data is stored. To further optimize the tree traversal, this design can use two different memory address spaces when allocating intermediate and leaf tree nodes (e.g., using two different allocators). By ensuring that intermediate nodes are always cached at the L2 level using CUDA's new memory access properties, one can coalesce intermediate node accesses through the L2 cache. Our insights throughout this dissertation inspire this binary search tree design.

Beyond 32-bit keys and values. Generally, we would like the node size to match the cache line, which means that the number of available key-value pairs per node will decrease for pairs with larger sizes (e.g., cache-line-sized nodes will only contain eight entries when the key-value pair size is 16 bytes). We note that reducing the number of elements per node reduces the branching factor of the tree structure and increases the tree height. Reducing the branching factor will increase the contention levels, especially at nodes closer to the tree's root.

Immutable data structures. An immutable (persistent) data structure is a different type of data structure that we would like to investigate on the GPU. These data structures are successful in functional programming languages and are friendly towards parallel computing. However, we believe that to have persistent data structures targeting the GPU, we must relax the requirements of a fully-persistent data structure and provide partial immutability (e.g., in a tree structure, only a subtree is immutable).

Memory management. We want to investigate massively-parallel memory allocators and safe memory reclaimers. We believe that composing memory allocators of simple ones is a direction worth pursuing. Improving cache locality and reducing the costly TLB misses on the GPU is the motivation behind this research direction.

Graph data structures. We believe that dynamically switching the adjacency list representation from one representation to another is an exciting research direction. For instance, when the vertex adjacency list size is small, we can represent the adjacency list using a sorted fixed-size array. As long as the adjacency size is small enough to fit within a cache line, maintaining the sorted order will not be expensive. When the adjacency list size exceeds what we can store in a cache line, we switch the representation to a more sophisticated data structure like a B-Tree or a hash table. Suppose we switch the adjacency list representation to a B-Tree with a branching

factor of 15 (i.e., cache-line sized node). In that case, splitting the fixed-size array neighbors and adding the tree root will allow us to store up to 225 neighbor vertices in the leaf nodes.

In our dynamic graph data structure representation, we assumed that a graph would have a fixed-size capacity of vertices. Our assumption allowed us to store pointers to vertices' adjacency lists in a fixed-size array. Extending the fixed-size array is not expensive as it does not require a deep copy of the entire graph adjacency lists. However, we believe that there are better ways to provide even more control over the number of vertices and maintain vertices with indices that do not form a sequence. One approach would be to add another level of indirection using a hash table. The hash table will store key-value pairs that map to vertices indices and adjacency lists pointers.

Future CUDA. We look forward to new GPU hardware generations and CUDA compiler and programming model improvements to aid in designing efficient abstractions of data structures. One of the challenges when designing efficient abstractions for a C++ object (e.g., data structure class) is providing the user with a single C++ heterogeneous object with APIs that can be used on the device or the host. Unfortunately, the CUDA programming model and compiler do not offer features that easily facilitate the construction and destruction of such heterogeneous objects. To address these challenges, we propose the two following features:

- **Device-side object constructors.** In many cases, we would like to reconstruct a C++ object from a parent object constructed on the CPU. For instance, device-side memory allocators typically allocate a large memory pool when first constructed from the host side. The allocators then manage the memory pool on a per-block (or per-thread) basis. In the current programming model, we would have to implement two classes (one for the host and another for the device), adding redundant, unnecessary code and not providing a heterogeneous view of the CPU-GPU system. Listing 6.1 shows an example of the proposed feature that would allow programmers to design heterogeneous objects effectively.

- **Shared memory allocation.** The CUDA programming model prevents programmers from allocating shared memory outside global functions (i.e., GPU kernel entry point). The lifetime of a shared memory is currently only managed by a thread block, therefore

preventing a class instance or device functions from encapsulating shared memory objects. The typical workaround to this issue is to ask object callers to preallocate shared memory that the object will use and pass the allocated memory pointer to the object's constructor. The shared memory allocation constraint prevents programmers from realizing the benefits of encapsulation and exposes implementation details to the object user. Listing 6.1 shows an example where an object provides a proper encapsulation of both register and shared memory variables.

```
1  struct foo{
2   __host__
3   foo()  {
4    /* host-side constructor */
5   }
6   __device_constructor__
7   foo(const foo& parent)  {
8       /* device-parameter copy constructor*/
9       /* Access the parent foo and setup variables and
10         state in shared memory and registers */
11  }
12  __shared__ int x; // per-block shared variable
13  int y;            // per-thread variable register
14 };
15
16 __global__
17 void kernel(foo f){
18  // implicitly call foo device-wide copy constructor
19 }
20
21 int main(){
22  foo f;
23  kernel<<<1,1>>>(f);
24 }
```

Listing 6.1: Heterogeneous CPU-GPU C++ object.

6.2.2 Distant-Future Research Directions

Our insights from building and designing efficient composable GPU data structures motivate us to investigate heterogeneous data structures further. Given our composable data structure design methodology, coupled with current and future systems that offer hardware-based unified memory (i.e., shared between CPU and GPU), we believe the time is ripe to explore and develop *truly* heterogenous data structures. CPUs alongside GPUs and other coprocessors cooperating on solving the same problem without rigid barriers between them opens the door to exciting directions for heterogeneous systems.

REFERENCES

[1] Alok Aggarwal and S. Vitter, Jeffrey. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (sep 1988), 1116–1127. https://doi.org/10.1145/48529.48535

[2] Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. 2011. Building an Efficient Hash Table on the GPU. In *GPU Computing Gems*, Wen-mei W. Hwu (Ed.). Vol. 2. Morgan Kaufmann, Chapter 4, 39–53. https://doi.org/10.1016/B978-0-12-385963-1.00004-6

[3] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing Epoch-Based Reclamation for Efficient Range Queries. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. 14–27. https://doi.org/10.1145/3178487.3178489

[4] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. 2018. A Dynamic Hash Table for the GPU. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018)*. 419–429. https://doi.org/10.1109/IPDPS.2018.00052

[5] Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D. Owens. 2018. GPU LSM: A Dynamic Dictionary Data Structure for the GPU. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018)*. 430–440. https://doi.org/10.1109/IPDPS.2018.00053

[6] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. 2019. Engineering a High-Performance GPU B-Tree. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2019)*. 145–157. https://doi.org/10.1145/3293883.3295706

[7] Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, and John D. Owens. 2020. Dynamic Graphs on the GPU. In *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2020)*. 739–748. https://doi.org/10.1109/IPDPS47924.2020.00081

[8] Ricardo J. Barrientos, José I. Gómez, Christian Tenllado, Manuel Prieto Matias, and Mauricio Marin. 2012. Range Query Processing in a Multi-GPU Environment. In *IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA-12)*. 419–426. https://doi.org/10.1109/ISPA.2012.61

[9] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2020. KiWi: A Key-Value Map for Scalable Real-Time Analytics. *ACM Transactions on Parallel Computing* 7, 3, Article 16 (June 2020), 28 pages. https://doi.org/10.1145/3399718

[10] R. Bayer and E. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (Houston, Texas) *(SIGFIDET '70)*. 107–141. https://doi.org/10.1145/1734663.1734671

[11] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. 2007. Cache-oblivious Streaming B-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (San Diego, California, USA) *(SPAA '07)*. 81–92. https://doi.org/10.1145/1248377.1248393

[12] Michael A. Bender, Martín Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't Thrash: How to Cache Your Hash on Flash. *Proc. VLDB Endow.* 5, 11 (July 2012), 1627–1637. https://doi.org/10.14778/2350229.2350275

[13] Michael A. Bender and Haodong Hu. 2006. An Adaptive Packed-memory Array. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '06)*. 20–29. https://doi.org/10.1145/1142351.1142355

[14] BlazingSQL. 2022. BlazingSQL. https://blazingsql.com/ [Online; accessed 2-February-2022].

[15] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. 2016. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 281–294.

[16] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC '15)*. 261–270. https://doi.org/10.1145/2767386.2767436

[17] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC '18)*. https://doi.org/10.1109/HPEC.2018.8547541

[18] Kristina Chodorow. 2013. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media, Inc.

[19] Jack Choquette, Oliver Giroux, and Denis Foley. 2018. Volta: Performance and Programmability. *IEEE Micro* 38, 2 (April 2018), 42–52. https://doi.org/10.1109/MM.2018.022071134

[20] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (June 1979), 121–137. https://doi.org/10.1145/356770.356776

[21] William J. Dally, Stephen W. Keckler, and David B. Kirk. 2021. Evolution of the Graphics Processing Unit (GPU). *IEEE Micro* 41, 6 (2021), 42–51. https://doi.org/10.1109/MM.2021.3113475

[22] A. ElTantawy and T. M. Aamodt. 2018. Warp Scheduling for Fine-Grained Synchronization. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 375–388. https://doi.org/10.1109/HPCA.2018.00040

[23] Jordan Fix, Andrew Wilkes, and Kevin Skadron. 2011. Accelerating Braided B+ Tree Searches on a GPU with CUDA. In *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC 2011)*.

[24] Keir Fraser. 2004. *Practical Lock-freedom*. Technical Report UCAM-CL-TR-579. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

[25] Afton Geil, Martin Farach-Colton, and John D. Owens. 2018. Quotient Filters: Approximate Membership Queries on the GPU. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018)*. 451–462. https://doi.org/10.1109/IPDPS.2018.00055

[26] Goetz Graefe. 2010. A Survey of B-tree Locking Techniques. *ACM Transactions on Database Systems* 35, 3, Article 16 (July 2010), 26 pages. https://doi.org/10.1145/1806907.1806908

[27] Oded Green and David A. Bader. 2016. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *InProceedings of 2016 IEEE High Performance Extreme Computing Conference (HPEC 2016)*. 1–6. https://doi.org/10.1109/HPEC.2016.7761622

[28] HEAVY.AI. 2022. HEAVY.AI. https://www.heavy.ai/ [Online; accessed 17-April-2022].

[29] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[30] Yulong Huang, Benyue Su, and Jianqing Xi. 2014. CUBPT: Lock-free bulk insertions to B+ tree on GPU architecture. *Computer Modelling & New Technologies* 18, 10 (2014), 224–231.

[31] Oracle Inc. 2022. Oracle. http://www.oracle.com/ [Online; accessed 18-April-2022].

[32] Ibrahim Jaluta, Seppo Sippu, and Eljas Soisalon-Soininen. 2005. Concurrency Control and Recovery for Balanced B-link Trees. *The VLDB Journal* 14, 2 (April 2005), 257–277. https://doi.org/10.1007/s00778-004-0140-6

[33] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR* (April 2018). https://doi.org/10.48550/ARXIV.1804.06826 arXiv:1804.06826

[34] Krzysztof Kaczmarski. 2012. B+-Tree Optimized for GPGPU. In *On the Move to Meaningful Internet Systems: OTM 2012*, Robert Meersman, Hervé Panetto, Tharam Dillon, Stefanie Rinderle-Ma, Peter Dadam, Xiaofang Zhou, Siani Pearson, Alois Ferscha, Sonia Bergamaschi, and Isabel F. Cruz (Eds.). Lecture Notes in Computer Science, Vol. 7566. Springer Berlin Heidelberg, 843–854. https://doi.org/10.1007/978-3-642-33615-7_27

[35] Tero Karras. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In *High-Performance Graphics* (Paris, France) *(HPG '12)*. 33–37. https://doi.org/10.2312/EGGH/HPG12/033-037

[36] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. 339–350. https://doi.org/10.1145/1807167.1807206

[37] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. 2013. Parallel multi-dimensional range query processing with R-trees on GPU. *J. Parallel and Distrib. Comput.* 73, 8 (Aug. 2013), 1195–1207. https://doi.org/10.1016/j.jpdc.2013.03.015

[38] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[39] Vladimir Lanin and Dennis Shasha. 1986. A Symmetric Concurrent B-tree Algorithm. In *Proceedings of the 1986 ACM Fall Joint Computer Conference* (Dallas, Texas, USA) *(ACM '86)*. 380–389. https://doi.org/10.5555/324493.324589

[40] Philip L. Lehman and S. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems* 6, 4 (Dec. 1981), 650–670. https://doi.org/10.1145/319628.319663

[41] Francesco Lettich, Claudio Silvestri, Salvatore Orlando, and Christian S. Jensen. 2014. GPU-Based Computing of Repeated Range Queries over Moving Objects. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 640–647. https://doi.org/10.1109/PDP.2014.27

[42] Yinan Li, Bingsheng He, Qiong Luo, and Ke Yi. 2009. Tree Indexing on Flash Disks. In *IEEE 25th International Conference on Data Engineering (ICDE '09)*. IEEE, 1303–1306. https://doi.org/10.1109/ICDE.2009.226

[43] Wei Liao, Zhimin Yuan, Jiasheng Wang, and Zhiming Zhang. 2014. Accelerating Continuous Range Queries Processing In Location Based Networks On GPUs. In *Management Innovation and Information Technology*. 581–589. https://doi.org/10.2495/MIIT130751

[44] Robert Love. 2010. *Linux Kernel Development*. Pearson Education.

[45] Lijuan Luo, Martin D. F. Wong, and Lance Leong. 2012. Parallel implementation of R-trees on the GPU. In *2012 17th Asia and South Pacific Design Automation Conference (ASP-DAC 2012)*. 353–358. https://doi.org/10.1109/ASPDAC.2012.6164973

[46] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 257–270. https://doi.org/10.1145/3297858.3304043

[47] MySQL 5.7 Reference Manual. 2022. The InnoDB Storage Engine. https://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html [Online; accessed 18-April-2022].

[48] Duane Merrill. 2015–2022. CUDA UnBound (CUB) Library. https://nvlabs.github.io/cub/.

[49] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-free Objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

[50] Nurit Moscovici, Nachshon Cohen, and Erez Petrank. 2017. A GPU-Friendly Skiplist Algorithm. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 246–259. https://doi.org/10.1109/PACT.2017.13

[51] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent Tries with Efficient Non-blocking Snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. 151–160. https://doi.org/10.1145/2145816.2145836

[52] RAPIDS. 2022. RAPIDS. https://rapids.ai/ [Online; accessed 2-February-2022].

[53] Ohad Rodeh. 2008. B-trees, Shadowing, and Clones. *ACM Transactions on Storage* 3, 4, Article 2 (Feb. 2008), 27 pages. https://doi.org/10.1145/1326542.1326544

[54] Yehoshua Sagiv. 1986. Concurrent Operations on B*-trees with Overtaking. *J. Comput. Syst. Sci.* 33, 2 (Oct. 1986), 275–296. https://doi.org/10.1016/0022-0000(86)90021-8

[55] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proceedings of the VLDB Endowment* 11, 1 (Sept. 2017), 107–120. https://doi.org/10.14778/3151113.3151122

[56] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A Hybrid B+-tree As Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 1523–1538. https://doi.org/10.1145/2882903.2882918

[57] Jyothish Soman, Kishore Kothapalli, and P. J. Narayanan. 2012. Discrete Range Searching Primitive for the GPU and Its Applications. *J. Exp. Algorithmics* 17, Article 4.5 (Oct. 2012), 1.07 pages. https://doi.org/10.1145/2133803.2345679

[58] Jeff A. Stuart and John D. Owens. 2011. Efficient Synchronization Primitives for GPUs. *CoRR* abs/1110.4623, 1110.4623v1 (Oct. 2011). arXiv:1110.4623v1 [cs.OS]

[59] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing* 4, 1 (Aug. 2017), 3:1–3:49. https://doi.org/10.1145/3108140

[60] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-Time Snapshots with Applications to Concurrent Data Structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. 31–46. https://doi.org/10.1145/3437801.3441602

[61] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. faimGraph: High Performance Management of Fully-Dynamic Graphs Under Tight Memory Constraints on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. Article 60, 13 pages. https://doi.org/10.1109/SC.2018.00063

[62] Yunlong Xu, Lan Gao, Rui Wang, Zhongzhi Luan, Weiguo Wu, and Depei Qian. 2016. Lock-based Synchronization for GPU Architectures. In *Proceedings of the ACM International Conference on Computing Frontiers* (Como, Italy) *(CF '16)*. 205–213. https://doi.org/10.1145/2903150.2903155

[63] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian. 2014. Software Transactional Memory for GPU Architectures. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA)

(CGO '14). Article 1, 1:1–1:10 pages. https://doi.org/10.1145/2581122.2544139

[64] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. 2019. Harmonia: A High Throughput B+tree for GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. 133–144. https://doi.org/10.1145/3293883.3295704

[65] Ke Yang, Bingsheng He, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, Pedro Sander, and Jiaoying Shi. 2007. In-memory Grid Files on Graphics Processors. In *Proceedings of the 3rd International Workshop on Data Management on New Hardware* (Beijing, China) *(DaMoN '07)*. Article 5, 7 pages. https://doi.org/10.1145/1363189.1363196

[66] Simin You, Jianting Zhang, and Le Gruenwald. 2013. Parallel Spatial Query Processing on GPUs Using R-trees. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data* (Orlando, Florida) *(BigSpatial '13)*. 23–31. https://doi.org/10.1145/2534921.2534949

[67] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. 2018. The Full Path to Full-Path Indexing. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (Oakland, CA, USA) *(FAST'18)*. USENIX Association, USA, 123–138. https://doi.org/10.5555/3189759.3189771