A GPU Multiversion B-Tree

Muhammad A. Awad
mawad@ucdavis.edu
UC Davis
Davis, CA, USA

Serban D. Porumbescu
sdporumbescu@ucdavis.edu
UC Davis
Davis, CA, USA

John D. Owens
jowens@ece.ucdavis.edu
UC Davis
Davis, CA, USA

ABSTRACT

We introduce a GPU B-Tree that supports snapshots and offers updates, point queries, and linearizable multipoint queries. The supported operations can be performed in a phase-concurrent, asynchronous, or fully-concurrent fashion. Our B-Tree uses cache-line-sized nodes linked together to form a version list and a GPU epoch-based reclamation scheme to reclaim older nodes' versions safely. Our data structure supports snapshots with minimal overhead in point queries (1.04× slower) and insertions (1.11× slower) versus a B-Tree that does not support versioning. Our linearizable B-Tree performs similarly to the non-linearizable baseline for read-heavy workloads and 2.39× slower for write-heavy workloads when performing concurrent range queries and insertions. In addition, we introduce different GPU-aware snapshot scopes that allow the use of our data structure for phase-concurrent (synchronous), stream-concurrent (asynchronous), and on-device fully-concurrent operations.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; • **Theory of computation** → **Data structures design and analysis**.

KEYWORDS

GPU, B-Tree, versioning, snapshots, linearizable, multipoint queries

ACM Reference Format:
Muhammad A. Awad, Serban D. Porumbescu, and John D. Owens. 2022. A GPU Multiversion B-Tree. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 10–12, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3559009.3569681

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 [8].

Data scientists can today use multiple commercial and open-source GPU databases, with increasingly easier-to-use and high-level abstractions [6, 10, 18]. 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 [2] and B-Trees [3, 21], among others [4, 16], 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 *linearization* point—between the operation's invocation and response [11]. 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 [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 [4]. The composed data structure will benefit from the guarantees and the properties of the base data structure. A dynamic graph data structure with support for snapshots facilitates maintaining streams of graph updates (e.g., vertices and edges insertion and deletion) while maintaining time information for queries through timestamps.

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 [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 above-mentioned design requirements while adding minimal overhead. At

PACT '22, October 10–12, 2022, Chicago, IL, USA

ACM ISBN 978-1-4503-9868-8/22/10.
https://doi.org/10.1145/3559009.3569681

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 expected by traversals from parent nodes.

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 a GPU implementation of a B-Tree [3] and the CPU work of Wei et al. [20]. 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.[1]

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.

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 [2], B-Trees [3, 21], dynamic graphs [4], and skip lists [16]. Of this work, the GPU B-Tree of Awad et al. [3] 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 [3] 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.

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 [19] 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 [17], 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 [1], 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 [1] traverse and validate the range query results using dirty bits in the tree nodes. Basin et al. [5] 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. [20] 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

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

only requires a single-word read and single-word CAS operations. Wei et al. [20] 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 4.1) as the branching factor of the B-Tree necessitates locks and cannot be implemented using compare-and-swap objects, and efficient 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.

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) [15], 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 device (here, a 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 HP unsuitable for GPUs.

Epoch-based reclamation (EBR) [9] 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 [7] implements a distributed epoch-based reclamation scheme with a key contribution of eliminating the need for 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) and map a CPU process to a CUDA block. We believe EBR at a CUDA block granularity is more suitable for the GPU than other techniques for two reasons. First, *retired* pointers are not shared across processes (and hence can instead 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.

2.4 Graphics Processing Units

Graphics Processing Units (GPUs) contain a group of *streaming multiprocessors* (SMs), each of which can run one to many GPU *blocks* (also known as *CTA*s). All GPU blocks launched by a *kernel* execute on the same CUDA *stream* and have the same number of *threads*. A group of 32 threads (called a warp) are executed in SIMD-style; however, recent GPU architectures [13] allow independent thread scheduling (i.e., SIMT-style) where each thread can have its own program counter (PC). While the SIMT-style execution model allows porting concurrent CPU algorithms to the GPU, typical CPU-based solutions generally do not scale to the GPU's level of parallelism. Thus new designs targeting scalability are often necessary.

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. [14]. 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 a whole-system level. Scopes are unique to GPU architectures.

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 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 using a more restrictive one. 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).

3 DESIGN DECISIONS

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—Awad et al.'s B-Tree [3] also performed in-place updates. 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.

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 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 $t+1$ [20]. 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 timestamp, 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× 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 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 timestamp 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.

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

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

Listing 1: High-level APIs for different scopes.

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

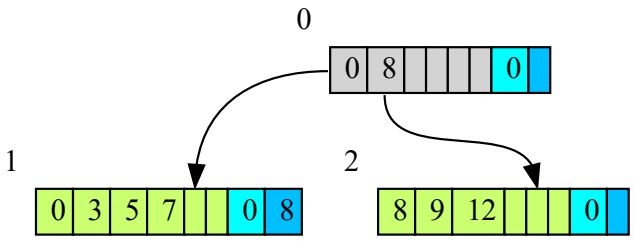
4 IMPLEMENTATION

Our Multiversion B-Tree is based on the implementation of Awad et al. [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 5.1. Our Multiversion B-Tree data structure supports the following operations:

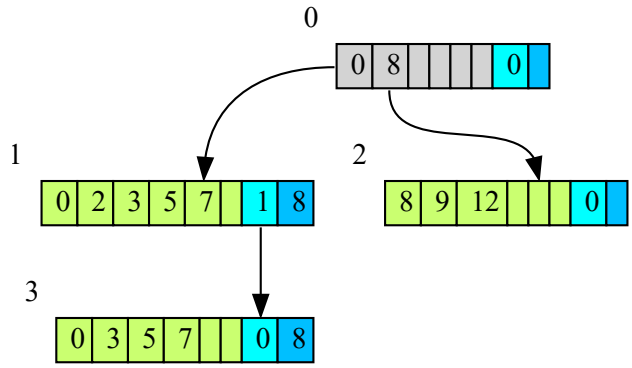
- **insert**(k, v)**:** inserts a key-value pair (k, v) into the Multiversion B-Tree into the latest version of the data structure.
- **delete**(k)**:** removes the key-value pair associated with the key k from the latest version of the Multiversion B-Tree.
- **takeSnapshot():** takes a snapshot of the 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 used by Awad et al. [3] 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[2]

[2]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 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.

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.

4.1 Insertion

We adapt the insertion algorithm of Awad et al. [3] 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 point 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 Awad et al. [3] showed, 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.

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 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 later 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. [20]; the main differences are how we perform copy-on-write and the B-Tree specific operations (e.g., splitting). Listing 2 shows how we perform concurrent insertions when the snapshot is taken concurrently with read-only operations. We refer the reader to Awad et al. [3] 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, or (2) the parent is not correct due to another insertion 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 we either update a parent node during a split or traverse 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) (lines 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 check that the parent 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,                  1
     Reclaimer& reclaimer){
  RootRestart: uint32_t node_index = root_index;                        2
  uint32_t parent_index = root_index;                                   3
  bool links_used = false;                                              4
  do{                                                                   5
    VersionedNode node(node_index);                                     6
    links_used |= traverse_links_init(node, key);                       7
    if(node.is_full() && node_index == parent_index &&                  8
         node_index != root_index){
      goto RootRestart;                                                 9
    }                                                                   10
    if(node.is_full() || node.is_leaf()){                               11
      if(!node.try_lock()){                                             12
        node_index = parent_index; continue;                            13
      }                                                                 14
      links_used |= traverse_locked_links_init(node, key);              15
      if(node.is_full() && links_used){                                 16
        node_index = parent_index; continue;                            17
      }                                                                 18
    }                                                                   19
    if(node.is_full() && node_index == root_index){                     20
      auto old_root_index = allocate(1);                                21
      auto [child0_index, child1_index] = allocate(2);                  22
      node.store_copy_at(old_root_index);                               23
      reclaimer.retire(old_node_index);                                 24
      auto two_children = node.split_as_root(child0_index,              25
           child1_index, old_node_index, cur_ts);
      two_children.right.store(); two_children.left.store();            26
      node.store(); node.unlock();                                      27
      node_index = node.find_next(key);                                 28
      if(node_index == child0_index){                                   29
        two_children.right.unlock();                                    30
        node = two_children.left;                                       31
      }else{                                                            32
        two_children.left.unlock();                                     33
        node = two_children.right;                                      34
      }                                                                 35
    }else if (node.is_full()){                                          36
      // split as intermediate                                          37
    }                                                                   38
    if(node.is_leaf()){                                                 39
      auto old_node_index = allocate(1);                                40
      node.store_copy_at(old_node_index);                               41
      reclaimer.retire(old_node_index);                                 42
      node.insert(key, value);                                          43
      node.set_older_version(old_node_index);                           44
      node.unlock();                                                    45
      node.initialize_timestamp(); return;                              46
    }else{                                                              47
      parent_index = node_index;                                        48
      node_index = node.find_next(key);                                 49
    }                                                                   50
  }while(true);                                                         51
}                                                                       52
```

Listing 2: Out-of-place Insertion.

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 having the ability to query 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 [20], multiple concurrent take_snapshot operations may return the same timestamp. Only one of the concurrent take_snapshots operations needs to succeed.

We show an example of a linearizable range query operation in Listing 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 4–6). 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 8). Otherwise, we start traversing side-links collecting all pairs that belong to the range (lines 10–15). 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 12).

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.

```
void VBTree::range_query(Key lower_bound, Key upper_bound,
     Timestamp timestamp, Pair result){
  uint32_t node_index = root_index;
  do{
    VersionedNode node(node_index);
    traverse_side_links_init(node);
    traverse_version_list(node, timestamp);
    if(node.is_intermediate()){
        node_index = node.find_next(lower_bound);
    }else{
      do{
        node.get_in_range(lower_bound, upper_bound, result);
        if(upper_bound < node.get_high_key()) return;
        node = node.get_sibling();
        node.initialize_timestamp();
        traverse_version_list(node, timestamp);
      }while(true);
    }
  }while(true);
}
```

Listing 3: Range query.

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 GPU Epoch-based Reclamation

Our GPU EBR implementation follows DEBRA [7]. 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 before 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 $\times$ 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. If the shared-memory limbo 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.

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 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 is complied with CUDA 11.5. Except for the memory reclamation evaluation (Section 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 Awad et al.'s B-Tree [3]. All the data structures in our benchmarks use our modified version of SlabAlloc [2] 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. 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.

5.1 Comparing to a B-Tree

In this benchmark, we analyze the performance of simple 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 2 shows the result of these benchmarks. In general, all tree operations' performance depends primarily on tree height. For instance, point queries find their result once they reach a leaf node. VB-Tree queries targeting the most recent timestamp will not traverse the version list (i.e., no additional overhead since the head of the version list is the most recent version). In addition to the tree traversal, update operations are lock-based and have the additional overhead of contention and memory fences.

Insertion. Our data structure achieves slightly lower performance than 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, the {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. The TLB effect does not appear when the memory allocated for the allocator pool, input, and output is less than 8 gigabytes (L2-cached TLB entries coverage is 8 gigabytes [12]). 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 throughputs in the {B-Tree, ViB-Tree, VoB-Tree} are {1512.964, 1453.697, 1549.977} million keys per second.

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 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 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.

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 3 and Table 1

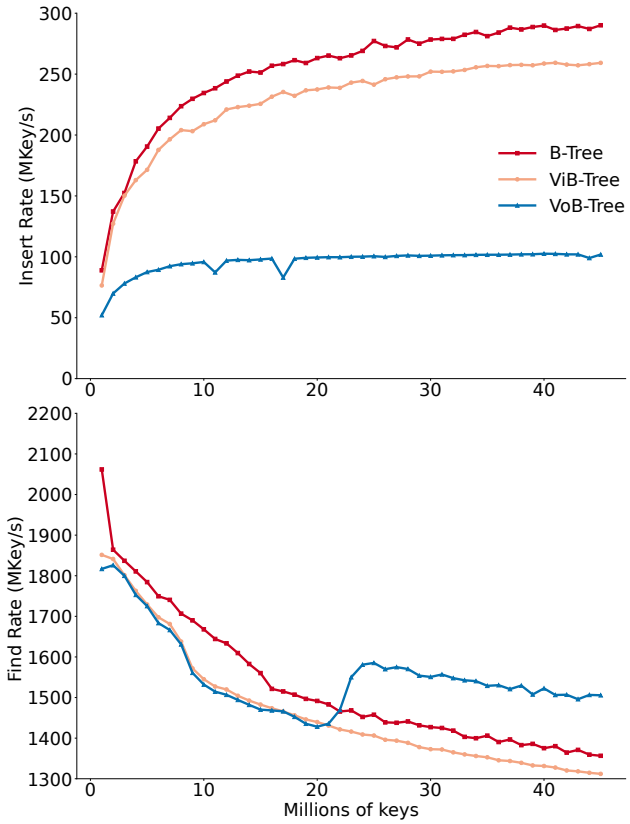


Figure 2: Insertion and find rates for trees containing different number of keys and the different B-Tree implementations. Find operations search for all keys in the data structure. Insertion performance is similar for the B-Tree and the ViB-Tree. However, the additional copies for out-of-place updates 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 5.1).

show the results and summary of this benchmark. Interestingly, our VoB-Tree outperforms the B-Tree for most workloads 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 overall throughput drops significantly in a VoB-Tree.

Figure 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,

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

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

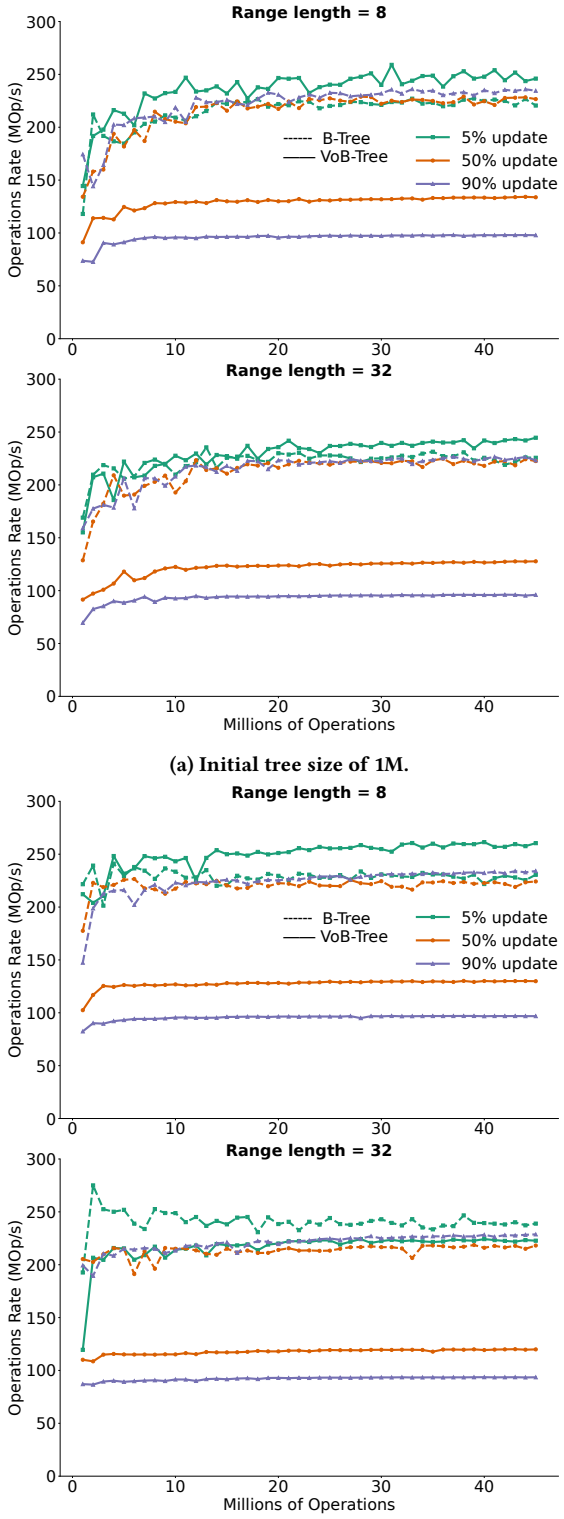
Table 2: Average concurrent delete and find rates for different update ratios (million operations per second).

the total operations rate drops from {197.253, 127.895} to {43.534, 27.923} for {B-Tree, VoB-Tree}.

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 5 shows the results of this benchmark. For all ratios, the B-Tree is 1.16× 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 2 summarizes the results of this benchmark.

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 6 shows the results of this benchmark. During the first 300 epochs, the allocator 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.

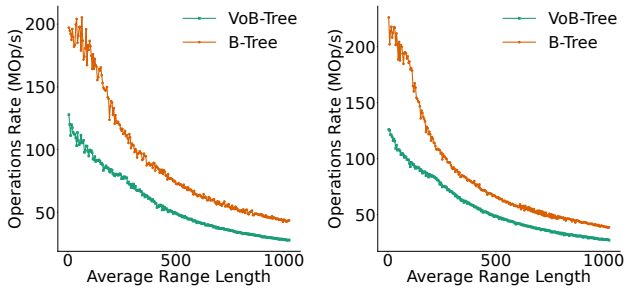
A ViB-Tree containing the same number of keys (67.5 million keys) uses 950 megabytes; using our EBR while concurrently building and querying the tree, the memory usage in the last epoch is 981 megabytes (thus, a 3.26% overhead for supporting versioning). The maximum memory overhead during the kernel's runtime (measured at each memory allocator call) is 3.5% compared to the



(a) Initial tree size of 1M.

(b) Initial tree size of 40M.

Figure 3: Concurrent insertion and range query for the B-Tree and our VoB-Tree with different update ratios and initial tree sizes.



(a) Initial tree size of 1M.

(b) Initial tree size of 40M.

Figure 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%.

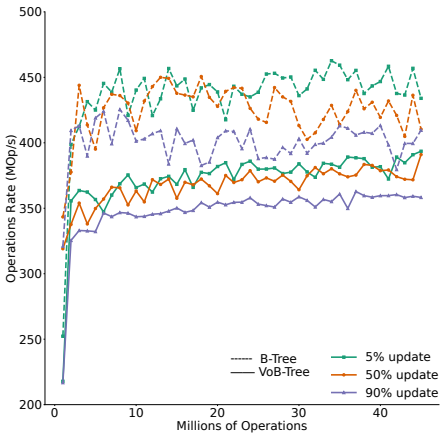
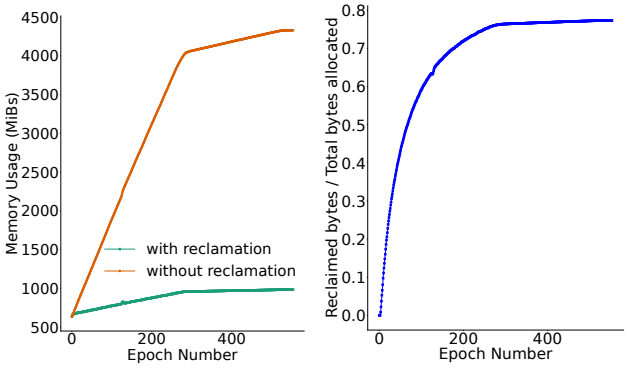


Figure 5: Performance of concurrent delete and point query in the B-Tree and VoB-Tree using different update ratios for a tree with an initial size of 45 million keys. Note that the graph has a false bottom.

final tree size. 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 finishing kernel execution, we can free these pointers or load them as shared limbo in future executions. In our benchmarks, we saw no noticeable runtime overhead for our SMR implementation.

6 CONCLUSION AND FUTURE WORK

In this paper 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 performance is the cost of supporting a more capable data structure. We believe that linearizable multipoint



(a) Memory usage in MiBs. **(b) Ratio between reclaimed and total-allocated bytes.**

Figure 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).

queries, first implemented in this work, are common and useful in real-world applications.

In the future, we will explore a multiversion dynamic graph data structure where we will represent each vertex adjacency list as a Multiversion B-Tree. Using the tools we developed, we want to explore wait-free techniques to build tree structures on the GPU. 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.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation (NSF)'s grant # CCF-1637442; by the Department of Defense Advanced Research Projects Agency (DARPA) under project HR0011-18-3-0007; and by an NVIDIA gift and hardware donations. We also thank the anonymous reviewers for their valuable feedback and suggestions for improving our paper.

REFERENCES

[1] 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 2018)*. 14–27. https://doi.org/10.1145/3178487.3178489

[2] 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

[3] 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

[4] 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

[5] 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

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

[7] 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 2015)*. 261–270. https://doi.org/10.1145/2767386.2767436

[8] 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

[9] 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

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

[11] 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

[12] 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

[13] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (March/April 2008), 39–55. https://doi.org/10.1109/MM.2008.31

[14] 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 2019)*. 257–270. https://doi.org/10.1145/3297858.3304043

[15] 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

[16] 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

[17] 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

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

[19] 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

[20] 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 2021)*. 31–46. https://doi.org/10.1145/3437801.3441602

[21] 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 2019)*. 133–144. https://doi.org/10.1145/3293883.3295704

A ARTIFACT DESCRIPTION

A.1 Abstract

The artifact contains our source code as a docker image bundled with all the required software dependencies. After loading the image you can compile the code then start running our different benchmarks to reproduce all figures and tables in Section 5. The docker image contains an Ubuntu 20.04 OS and CMake 3.8, Miniconda, Python and the CUDA 11.5 toolkit. We prepared the docker image on Ubuntu 20.4 with an NVIDIA driver version 510.85.02 and using docker version 20.10.6.

A.2 Description

A.2.1 Check-list (artifact meta information).

- **Algorithms:** Insertion, find, delete, and range query operations using our Multiversion B-Tree variants (ViB-Tree and VoB-Tree) and reference B-Tree implementation
- **Program:** `reproduce.sh` runs multiple executables
- **Data set:** Data sets are generated during benchmarking
- **Run-time environment:** NVIDIA driver version 450.80.02 or higher and linux OS
- **Hardware:** NVIDIA Volta GPU or later microarchitectures with at least 20 GiB DRAM
- **Execution:** Approximately 7 hours on our system[3]
- **Output:** Graphs and tables in Section 5
- **Experiment workflow:** Compilation of the code, running unit tests, running a script to collect benchmarking results, then running a script to plot the results
- **Publicly available?:** Yes
- **Code license:** Apache License 2.0

A.2.2 How delivered. The docker image is available on Docker Hub at https://hub.docker.com/repository/docker/maawad/mvgpubtree.

A.2.3 Hardware dependencies. Our implementation and benchmarks require an NVIDIA Volta GPU or a later microarchitectures with at least 20 GiB of DRAM. A modern CPU with at least 1 GiB of DRAM is required (we only use the CPU to generate the input datasets and launch GPU kernels). The docker image is 7.8 GiBs (3.6 GiBs when compressed). Our experiments used an NVIDIA Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM and a theoretical achievable DRAM bandwidth of 897 GiB/s. Other GPUs will achieve similar performance trends; however, performance will primarily depend on the achievable DRAM and atomic bandwidth of the GPU.

A.2.4 Software dependencies. Docker 19.03 or higher (ideally the same version we use) and an NVIDIA driver version 450.80.02 or higher.

A.3 Step By Step Instructions

(1) Install docker[4] and the NVIDIA GPU driver[5] (if not already installed)
(2) Clone the image from Docker Hub
    `$ docker pull maawad/mvgpubtree`
(3) Launch the docker image in interactive mode
    `$ docker run -it --name trees --gpus all maawad/mvgpubtree /bin/bash`
    Notice that on a system with multiple GPUs you can specify the GPU index after the `--gpus device` flag (e.g., `--gpus device=0` to use the GPU with index zero). Use a different image name (i.e., `--name other-name`) if you are already using the name for a different container
(4) Navigate to the directory containing the source code
    `$ cd MVGpuBTree`
(5) Build the source code
    ```
    mkdir build && cd build
    cmake .. && make -j
    ```
(6) Run the unit tests
    ```
    $ ./bin/unittest_btree
    $ ./bin/unittest_versioning
    ```
    Successful testing will output a log of the tests ending with a message similar to: `[ PASSED ] 10 tests.`
(7) Run the benchmarks
    `$ cd .. && source reproduce.sh`
(8) Generate the figures and tables
    `$ cd plots && source plot.sh`
(9) From the host machine and while the docker image is still running, copy the generated figures and text files back to the host
    `$ docker cp trees:MVGpuBTree/plots/figs .`
    Note that `.` means copy the `figs` directory to the current location on the host
(10) If you wish to view the raw data, copy the generated comma-separated values files back to the host machine
    `$ docker cp trees:MVGpuBTree/results .`

A.4 Evaluation and Expected Result

The `figs` directory will contain the output figures and tables that we used in our paper. The `figs` directory should include the following files:

```
$ ls figs/
Tesla-V100-PCIE-32GB
# The plotting script will store all figures and tables in a
      directory with the same name as the GPU used for
      benchmarking
$ ls figs/Tesla-V100-PCIE-32GB/
# Figure 2
## Rates for insertion and point query for B-Tree vs. VBTree
insertion_find_rates_slab.pdf
## Tabular summary of the  B-Tree vs. VBTree results
blink_vs_versioned.txt

# Figure 3 and Table 1
## Rates for operations (insert and RQ) on an initial tree size
       of 1 million keys
insertion_rq_rates_slab1.pdf
# Rates for operations (insert and RQ) on an initial tree size
      of 40 million keys
insertion_rq_rates_slab40.pdf
## Tabular summary of the concurrent insert RQ results
concurrent_insert_range.txt

# Figure 4
## Rates of operations using 1 million keys initial tree size
      and variable range length
insertion_vary_rq_rates_initial1M_update50_num_ops5_slab.pdf
## Rates of operations using 40 million keys initial tree size
      and variable range length
insertion_vary_rq_rates_initial40M_update50_num_ops5_slab.pdf

# Figure 5 and Table 2
## Rates for operations (find and erase) on an initial tree
      size of 45 million keys
erase_find_rates_slab45.pdf
## Tabular summary of the concurrent find erase results
concurrent_erase_find.txt

# Figure 6
## Memory usage in MiBs
insertion_find_memory_45m_45m_50_16_slab.pdf
## Ratio between reclaimed and total-allocated bytes
insertion_find_ratio_memory_45m_45m_50_16_slab.pdf
```

[3] using an an NVIDIA Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM and an Intel Xeon Gold 6146 CPU.
[4] https://docs.docker.com/engine/install/ubuntu/
[5] https://docs.nvidia.com/datacenter/tesla/tesla-installation-notes/index.html

If you copied the raw data, the `results` directory should contain the following subdirectories

```
$ ls results/
Tesla-V100-PCIE-32GB

$ ls results/Tesla-V100-PCIE-32GB/
## directory containing csv file for Figure 2
blink_vs_versioned
## directory containing csv files for Figure 3, Figure 6 and
     Table 1
versioned_insert_range
## directory containing csv files for Figure 4
versioned_insert_range_variable_range
## directory containing csv files for Figure 5
versioned_find_erase
```

The reproduced results should match our reported results if using the same GPU as the one we use in our benchmarks. If a different GPU is used, we expect the reproduced results to follow similar trends but the achieved throughput will differ. Specifically, we except the performance to improve as the achievable DRAM throughput or atomic throughput increase. The TLB effect observed in our Figure 2 may not be observed on other GPUs. For instance, we did not observe the TLB effect on A100 or Titan V NVIDIA GPUs.

A.5 Notes

Our source code is publicly available at https://github.com/owensgroup/MVGpuBTree. The GitHub repository also contains:

- The docker file used to generate the docker image
- Markdown files containing instructions to reproduce per-figure results and more extensive validation
- Benchmarking and plotting scripts
- Markdown file containing documentation of our Multiversion B-Tree APIs
- Archived results on different GPUs