Engineering a High-Performance GPU B-Tree

Muhammad A. Awad
University of California, Davis
mawad@ucdavis.edu

Saman Ashkiani*
University of California, Davis
sashkiani@ucdavis.edu

Rob Johnson
VMWare Research
robj@vmware.com

Martín Farach-Colton
Rutgers University
farach@cs.rutgers.edu

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

Abstract

We engineer a GPU implementation of a B-Tree that supports concurrent queries (point, range, and successor) and updates (insertions and deletions). Our B-tree outperforms the state of the art, a GPU log-structured merge tree (LSM) and a GPU sorted array. In particular, point and range queries are significantly faster than in a GPU LSM (the GPU LSM does not implement successor queries). Furthermore, B-Tree insertions are also faster than LSM and sorted array insertions unless insertions come in batches of more than roughly 100k. Because we cache the upper levels of the tree, we achieve lookup throughput that exceeds the DRAM bandwidth of the GPU. We demonstrate that the key limiter of performance on a GPU is contention and describe the design choices that allow us to achieve this high performance.

CCS Concepts • **Computing methodologies → Parallel algorithms**; • **Computer systems organization → Single instruction, multiple data**.

Keywords b-tree, dynamic, mutable, data structures, GPU

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.

	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 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.

Only recently have dynamic GPU versions of four basic data structures been developed: hash tables [2], sparse graphs with phased updates [14], quotient filters [12], and log-structured merge trees (LSMs) [3]. 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$^\epsilon$-trees, is at the core of many applications, from SQL databases [16, 31] to NoSQL databases [8, 22] to the paging system of the Linux kernel [28].

In this paper, 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 insertion performance, but they have relatively worse query performance than a B-Tree [6, 26].

Table 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

*Currently an employee at OmniSci, Inc.

PPoPP '19, February 16–20, 2019, Washington, DC, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295706

[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.

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 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 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.

2 Background and Previous Work

2.1 Graphics Processing Units

Graphics Processing Units (GPUs) feature several streaming multiprocessors (SMs), each with its own dedicated local resources (such as L1 cache, a manually managed cache called *shared memory*, and registers). 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 assignment of thread-blocks to SMs is done by the hardware and the programmer has no explicit control over it. All SMs, and hence all available threads on the GPU, have access to some globally shared resources such as the L2 cache and the DRAM *global memory*.

In reality, not all resident threads on an SM are actually executed in parallel. Each SM executes instructions for a group of 32 threads, a *warp*, in a single-instruction-multiple-data (SIMD) fashion. All memory transactions are performed in units of 128 bytes where each thread within a warp fetches 4 consecutive bytes. As a result, in order to achieve an efficient GPU program, 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 and 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 [2]: independent operations are still assigned to each thread (per-thread work assignment), but all threads within a warp cooperate with each each other to process in parallel (per-warp processing). By doing so, threads cooperate with each other in both memory accesses and executed instructions, resulting in more coalesced accesses and reduced branch divergence. While traditionally, communications between threads are done either through the shared memory (if within the same thread-block), or the global memory (among all threads), additionally utilizing high-bandwidth, low-latency warp-wide communication between threads may enable higher performance overall; threads within a warp can communicate with each other through voting (e.g., ballots) or reading the content of another thread's registers (e.g., shuffles).

The NVIDIA TITAN V GPU (an instance of NVIDIA's "Volta" microarchitecture) has 80 SMs and 64 thread processors per SM for a maximum of 5120 resident warps. It contains a 6 MB L2 cache, a 10 MB L1 cache distributed across SMs, and a global memory (DRAM) throughput of 625.8 GB/s.

2.2 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 [9] was introduced by Bayer and McCreight [5] 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 1.

2.3 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 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 [13] 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 [24] 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 [24] to provide more concurrency. The merging of nodes to reduce the tree height after deletions was presented by Lanin and Shasha [23] and Sagiv [33]. Latch coupling was used in B-link-trees by Jaluta et al. [17] 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 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. [11] 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 [19], 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. [15], who extend Kaczmarski's work but with non-clustered indexes that would be poorly suited for range queries; and Shahvarani and Jacobsen [34], 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

Work	Usage	Data structure notes
YHFL+ [40]	Grid files for multidimensional database queries.	Built on CPU.
KCSS+ [20]	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 [11]	Processing B+ tree queries for databases.	Built on CPU.
LWL [29]	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+ [4]	Single- and multi-GPU range queries for List of Clusters and Sparse Spatial Selection indexing approaches.	Built on CPU.
SKN [35]	Compute range queries by constructing Cartesian tree and finding least common ancestors.	Updates require complete rebuild.
KKN [21]	R-tree traversal for spatial data. Sequential search between nodes, parallel search within each node.	Built on CPU.
YZG [41]	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 [27]	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 [25]	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+ [3]	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 [34]	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 [39]	Phased queries and updates on the GPU.	State-of-the art query throughput. Less efficient update throughput.

Table 2. Chronological summary of previous work on dictionary data structures that support point and range query on GPUs.

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. [39] 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 [3] 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

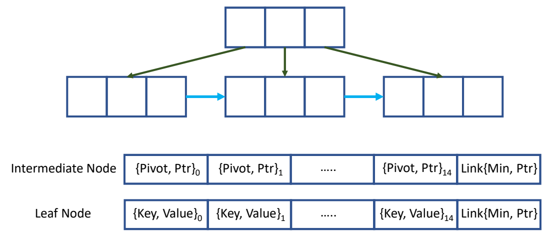


Figure 1. B-Link-Tree (with $B = 3$) schematic (top). Our B-Tree (with $B = 15$) node structure (bottom). A tree node contains 15 pivot-pointer (or key-value) pairs. A pointer to the node's child is represented by the child's offset. The last pair in a node represents the right sibling minimum value and its pointer. The minimum of the right sibling serves as a high key for the node.

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 Section 5.

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 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 3.1), and an additional pair containing a pointer to its right sibling and the minimum key of its right sibling (Section 3.2).

Reading a tree node is not blocked by any other operation (Section 3.3). When we insert into the tree and must split, we use proactive splitting (Sections 2.3 and 3.4) with restarts on failures (Section 3.5). We use a simple write latch per node to prevent 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 3.6) 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.

3.1 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 3.2) 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 1 illustrates the tree node structure.

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 [24]. 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 3.4), it improves concurrency.

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.

3.4 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 3.2), 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.

3.5 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 [37, 38] 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 3.2 and 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 [36], 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 [10] 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 don't 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.

3.6 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 [2]. 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.

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 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 1 Warp Cooperative Work Sharing Algorithm.

```
procedure WCWS(Tree btree, Pair pairs, Task tasks)
    is_active ← true
    thread_pair ← pairs[threadIdx]
    thread_task ← tasks[threadIdx]
    while work_queue ← ballot(is_active) do
        current_lane ← ffs(work_queue)
        current_pair ← shfl(thread_pair, current_lane)
        current_task ← shfl(thread_task, current_lane)
        performTask(current_task, current_pair, btree)
        if laneId = current_lane then
            is_active ← false
        end if
    end while
end procedure
```

4.1 Insertion

4.1.1 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 [30] 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.

4.1.2 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 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 3.5, 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.

Algorithm 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
```

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 5 use both memory fences and a L1 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).

4.2 Search

Searching the tree for a value (Algorithm 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.

4.3 Deletion

In deletion, a warp first traverses the tree to find the deleted key. Once it reaches the leaf, it latches the tree node and reads the leaf 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 don't 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.

Algorithm 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
```

4.4 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.

4.5 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.

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. [3]. The GPU LSM implementation uses CUB [30] 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 2).

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

5.1 Insertion

Baseline B-Tree. Our baseline B-Tree implementation is most similar to the B-Tree of Rodeh [32]. 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 3.2, 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.

Bulk-build vs. incremental update. We investigate the advantage of incremental update over complete rebuild of the B-Tree. Figure 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.

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

[3]Moderngpu is available at https://github.com/moderngpu/moderngpu.

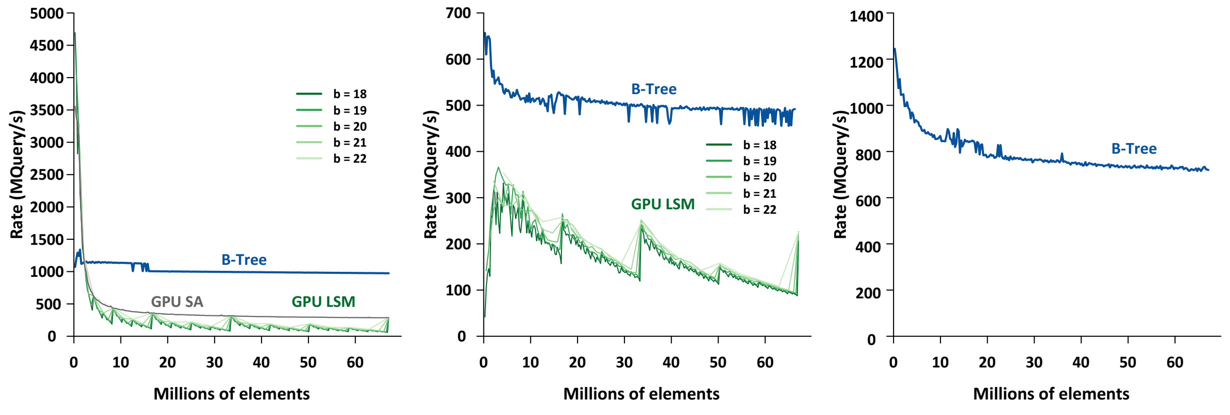
batch size	B-Tree	GPU LSM	GPU SA
2^{16}	168.0	61.5	44.9
2^{17}	139.7	121.3	87.6
2^{18}	171.7	218.6	160.6
2^{19}	190.3	402.5	292.6
2^{20}	205.1	685.9	543.0
2^{21}	211.9	1103.8	907.5
2^{22}	223.0	1603.1	1472.7
Mean	182.9	202.6	149.1

Table 3. Mean rates (in MKey/s) for different batch-sized insertions into the B-Tree, GPU LSM and GPU SA.

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. 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 $\{2^{19}, 2^{22}\}$.

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 2a shows the 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



(a) Point query rate **(b)** Range query rate **(c)** Successor query rate

Figure 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 ≈ 18M all the way up to a theoretical $15^8 \times 15 \approx 38$B. 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.

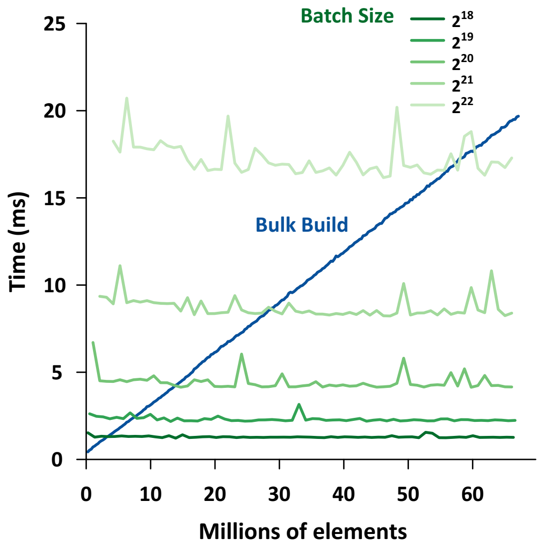


Figure 3. Crossover points between bulk-rebuild of the B-Tree (including sort time) and inserting a batch of size 2^i that result in a tree with the same number of elements.

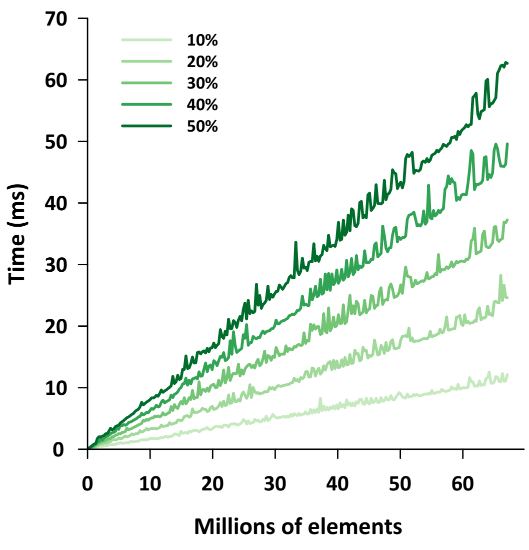


Figure 4. Deletion time for different percentages of the number of key-value pairs in the tree.

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.

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 writeback. We present the results in Figure 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 4.3) speeds up deletion rates by a factor of 2.4x.

5.4 Range Query

Figure 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).

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.

5.6 Concurrent Benchmark

Benchmark setup. To evaluate concurrent updates and queries we define an update ratio α, where $0 \le \alpha \le 1$, 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.

Results. Figure 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 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.

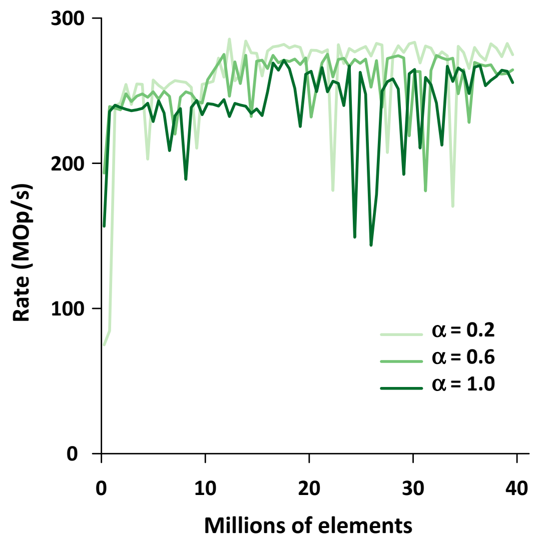


Figure 5. Concurrent benchmark operations througput for diffreent B-Tree sizes.

		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 4. Summary of memory hierarchy microbenchmarking results [18] on the Volta and Kepler architectures.

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 4. We profiled our point query kernel on a TITAN V GPU and a TESLA K40c GPU. Figure 6 plots different memory hierarchy levels' throughput and hit rates. A 2.6x-larger L1 data cache 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.

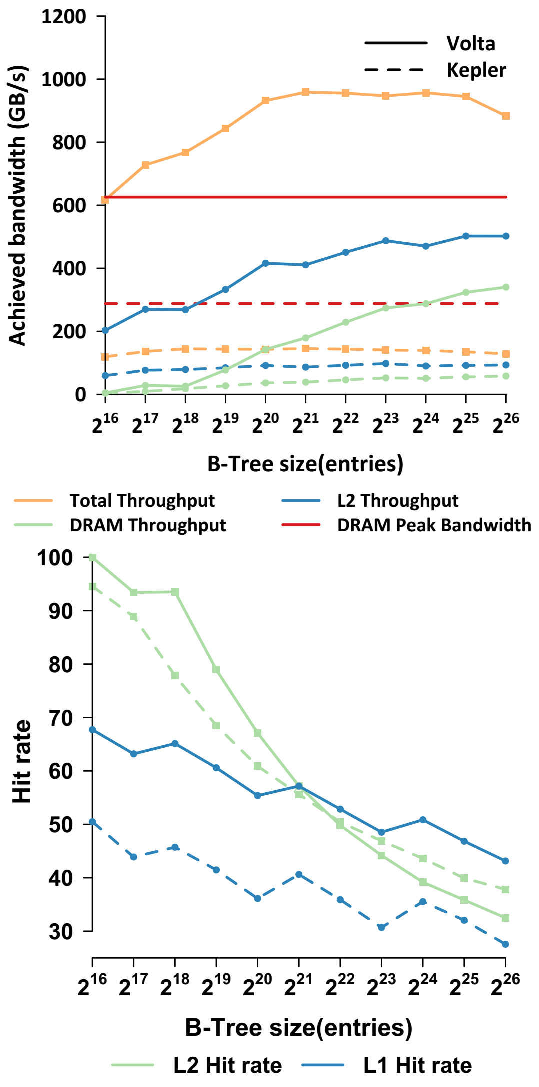


Figure 6. Throughput (top) and hit rates (bottom) for the different memory hierarchy levels during search queries. Upper-level tree nodes of the B-Tree are cached throughput the memory hierarchy, thus achieving high hit rates in L1 and L2 caches, and allowing the total throughput of our B-Tree to exceed the peak DRAM bandwidth on Volta.

6 Conclusion

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 [7] or lifted B-trees [42]).

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1637442 and Grant No. CCF-1637458, with this project initiated at the December 2017 AlgoPARC Workshop on Parallel Algorithms and Data Structures at the University of Hawaii at Manoa (Grant No. CCF-1745331). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We appreciate technical help from Lars Nyland and Tyler Sorensen along the way.

References

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

[2] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. 2018. A Dynamic Hash Table for the GPU. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018)*. https://doi.org/10.1109/IPDPS.2018.00052

[3] 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 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018)*. 430–440. https://doi.org/10.1109/IPDPS.2018.00053

[4] 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

[5] 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 (SIGFIDET '70)*. 107–141. https://doi.org/10.1145/1734663.1734671

[6] 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 (SPAA '07)*. 81–92. https://doi.org/10.1145/1248377.1248393

[7] Michael A. Bender, Martin 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. *PVLDB* 5, 11 (2012), 1627–1637.

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

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

[10] 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

[11] 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)*.

[12] Afton Geil, Martin Farach-Colton, and John D. Owens. 2018. Quotient Filters: Approximate Membership Queries on the GPU. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018)*. 451–462. https://doi.org/10.1109/IPDPS.2018.00055

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

[14] 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

[15] 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.

[16] Oracle Inc. 2011. Oracle. http://www.oracle.com/.

[17] 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

[18] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR* (April 2018). arXiv:1804.06826 http://arxiv.org/abs/1804.06826

[19] 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

[20] 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 (SIGMOD '10)*. 339–350. https://doi.org/10.1145/1807167.1807206

[21] 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

[22] 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

[23] Vladimir Lanin and Dennis Shasha. 1986. A Symmetric Concurrent B-tree Algorithm. In *Proceedings of the 1986 ACM Fall Joint Computer Conference (ACM '86)*. 380–389. http://dl.acm.org/citation.cfm?id=324493.324589

[24] 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

[25] 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 (PDP 2014)*. 640–647. https://doi.org/10.1109/PDP.2014.27

[26] 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

[27] 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

[28] Robert Love. 2010. *Linux kernel development*. Pearson Education.

[29] 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

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

[31] MySQL 5.7 Reference Manual. [n. d.]. Chapter 15 The InnoDB Storage Engine. http://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html.

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

[33] 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

[34] 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 (SIGMOD '16)*. ACM, New York, NY, USA, 1523–1538. https://doi.org/10.1145/2882903.2882918

[35] 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), 4.5:4.1–4.5:4.17 pages. https://doi.org/10.1145/2133803.2345679

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

[37] 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 (CF '16)*. 205–213. https://doi.org/10.1145/2903150.2903155

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

[39] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. 2019. Harmonia: A High Throughput B+tree for GPUs. In *Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*.

[40] 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 (DaMoN '07)*. Article 5, 7 pages. https://doi.org/10.1145/1363189.1363196

[41] 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 (BigSpatial '13)*. 23–31. https://doi.org/10.1145/2534921.2534949

[42] Yang Zhan, Alexander 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 *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, 123–138.