Boosting Performance and QoS for Concurrent GPU B+trees by Combining-based Synchronization

Weihua Zhang*†‡¶, Chuanlei Zhao*, Lu Peng§, Yuzhe Lin*, Fengzhe Zhang*, Yunping Lu†¶
* School of Computer Science, Fudan University
† Institute of Big Data, Fudan University
‡ State Key Laboratory of Mathematical Engineering and Advanced Computing
¶ Parallel Processing Institute, Fudan University
§ Department of Computer Science, Tulane University
{zhangweihua,clzhao20,yzlin14,fzzhang}@fudan.edu.cn,lpeng3@tulane.edu,luyping@sina.com

Abstract

Concurrent B+trees have been widely used in many systems. With the scale of data requests increasing exponentially, the systems are facing tremendous performance pressure. GPU has shown its potential to accelerate concurrent B+trees performance. When many concurrent requests are processed, the conflicts should be detected and resolved. Prior methods guarantee the correctness of concurrent GPU B+trees through lock-based or software transactional memory (STM)-based approaches. However, these methods complicate the request processing logic, increase the number of memory accesses and bring execution path divergence. They lead to performance degradation and variance in response time increasing. Moreover, previous methods do not guarantee linearizability among concurrent requests.

In this paper, we design a combined-based concurrency control framework, called Eirene, for GPU B+tree to reduce the overhead of conflict detection and resolution. First, a combining-based synchronization method is designed to combine and issue requests. It combines the requests with the same key, constructs their dependence, decides the issued request, and determines their return values. Since only one request for each key is issued, key conflicts are eliminated. Then, an optimistic STM method is used to reduce structure conflicts. The *query* and the *update* requests are partitioned into different kernels. For the *update* kernels, STM is involved only when the number of the retry reaches a threshold. Finally, a locality-aware warp reorganization optimization is proposed to improve memory behavior and reduce conflicts by exploiting the locality among requests. Evaluations on an NVIDIA A100 GPU show that Eirene is efficient (a throughput of 2.4 billion per second) and can guarantee linearizability. Compared to the state-of-the-art GPU B+tree, it can achieve a speedup of 7.43X and reduce the response time variance from 36% to 5%.

CCS Concepts: • **Theory of computation → Data structures design and analysis**; • **Software and its engineering → Concurrency control**.

Keywords: GPU B+tree, Concurrency

1 Introduction

As one of the most popular index structures, concurrent B+trees have been widely adopted in different fields, such as file systems [36], relational databases [20, 27, 33], and key-value store systems [22, 24]. Nowadays, the volume of concurrent requests increases exponentially in domains like big data. For example, more than 80 million requests per second were processed by Amazon's database on Prime Day 2020 [6]. The underlying B+trees are facing significant pressure for serving such a tremendous amount of concurrent requests.

GPUs have become one of the most popular accelerators due to their high computing power and large memory bandwidth [18, 25, 42]. They have already been adopted in datacenter environments to boost the performance of core algorithms of many applications [3, 15]. GPUs also show the potential to improve the performance of concurrent B+trees. To fully utilize parallel computing resources on GPUs, concurrent requests are first buffered in host memory and then transferred to GPUs to be processed [43, 44]. To support concurrent request processing on GPUs, some concurrency control approaches like locks or software transactional memory (STM) are used to detect and resolve conflicts[1] among requests. However, by analyzing the existing designs for concurrent GPU B+trees [4, 19], we realize that the detection and resolution of concurrent conflicts complicate the

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0015-6/23/02…$15.00
https://doi.org/10.1145/3572848.3577474

[1]The conflicts can be classified as key conflicts (caused by the requests on the same key) and structure conflicts (caused by tree rebalancing operations triggered by updates).

processing logic and hence increase the memory accesses and branch control instructions. It dramatically impacts GPU B+tree performance and quality of service (QoS) and causes a significant variance in response time. And none of these concurrent GPU B+trees guarantees linearizability [13, 16, 17], which is essential for the fairness of concurrent requests.

Enlightened by the above analysis, we design and implement a combining-based concurrency control framework called Eirene, the Greek goddess of peace, for GPU B+tree to minimize the overhead of conflict detection and resolution. First, a combining-based synchronization approach is designed to eliminate key conflicts among requests. It combines the requests with the same key and constructs the dependence among them by their logical timestamps. Timestamp refers to the arrival order of requests, which in the linearizability semantics determines the outcome of the requests. For the requests with the same key, only one request is issued, and the return values of the other requests are determined by their dependency relationship. To reduce structure conflicts, an optimistic STM method is used. The requests are partitioned into different kernels (*query* kernel and *update* kernel) according to their types. For the *query* kernel, the requests are processed without STM protection. For the *update* kernel, STM is involved only when retries reach a threshold. Finally, a locality-aware warp reorganization optimization is proposed to improve memory performance and reduce conflicts by exploiting the locality among the combined requests.

These designs of Eirene can achieve better performance and QoS, and keep linearizability by eliminating key conflicts, reducing the possibility of structure conflicts, and exploiting locality. We use the YCSB benchmark [11] to evaluate key-value store performance. Evaluation results show Eirene has a substantial improvement in both performance and response time relative to existing concurrent GPU B+trees (STM GB-tree [19] and Lock GB-tree [4]). It achieves a throughput of 2.4 billion per second for a 95%*query*/5%*update* concurrent input. Compared to the state-of-the-art Lock GB-tree [4], it achieves about a 7.43X performance speedup, and the variance of response time is improved from 36% to 5%.

In summary, the contributions of this work include the following:

- We make a comprehensive analysis of previous works of concurrent GPU B+tree and find several performance bottlenecks.
- We propose a combining-based concurrency control framework called Eirene, which is the first combining-based synchronization approach for concurrent GPU B+trees. Eirene can reduce conflicts and overhead of conflict detection and resolution to achieve better performance and QoS with linearizability.
- Results show Eirene can achieve a throughput of 2.4 billion per second with only a 5% response time variance.

The remainder of this paper is organized as follows. Section 2 introduces the background and discusses our motivation. Section 3 surveys the related works. Section 4 introduces the combining-based concurrent control approach. Section 5 gives out the locality-aware warp reorganization optimization. Section 7 introduces the implementation of Eirene. Section 8 shows the experimental results and Section 9 concludes this work.

2 Background and Motivation

This section first introduces concurrent GPU B+trees. Then, we discuss the problems of existing concurrent GPU B+trees.

2.1 Concurrent GPU B+trees

B+tree is a popular index structure for large-scale data storage and has been widely used in different fields, such as databases [20, 27, 33] and file systems [36]. It is one of the B-tree varieties where inner nodes only store keys and child references. The leaf node contains keys and values. In the real-world, concurrent requests (*query* requests mixed with *update* requests, which include update, insertion, and deletion) for B+trees are processed simultaneously. With the requests increasing exponentially, GPUs show the potential to improve the performance of concurrent B+trees [4, 41].

A GPU contains several streaming multiprocessors (SMs). Multiple contiguous threads (e.g., 32 threads) are organized into a warp and execute the same instructions in a lockstep fashion. The thread blocks composed of multiple warps constitute the GPU kernel, a parallel function that executes on all SMs. To fully utilize parallel computing resources on GPUs, concurrent key-value requests are first buffered in host memory and then transferred to GPUs to be processed [43, 44] in GPU kernels. When concurrent requests are processed in parallel, the conflicts among them should be detected and resolved to guarantee correctness. Conflicts could be detected and resolved using software transactional memory (STM)-based [8, 19, 40] or lock-based approaches [4, 43, 44].

2.2 Motivation

For concurrent GPU B+trees, no matter lock-based or STM-based approaches, conflict detection, and resolution decrease the efficiency of request processing. Conflict detection and resolution are conducted during tree traversal and node operations. To detect and resolve conflicts, a large number of conditional statements and branches are introduced, which is not friendly to GPU SIMT architecture and significantly increases the number of memory access. Moreover, the requests have to be re-executed or stalled when conflicts happen, which increases the processing delay. As a result, conflict detection and resolution bring a severe impact on request processing throughput and response time [23, 37, 45].

To illustrate the impact, we collect the memory accesses (*memory_inst*) and the control-flow instructions (*control_inst*)

of STM-based GPU B+tree [19] (STM GB-tree for short) and lock-based GPU B-tree [4] (Lock GB-tree for short) using Nsight Compute [31]. The profiling environment is the same as that described in Section 8.

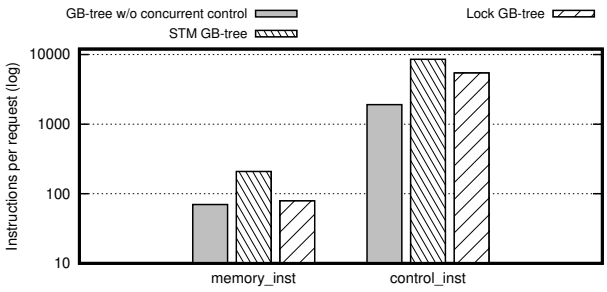


Figure 1. Profiling of STM GB-tree and Lock GB-tree. Note that the y-axis is in the log scale.

As the data in Figure 1 show, there are 209 and 79 memory accesses per request for STM GB-tree and Lock GB-tree respectively. And there are 8562 and 5445 control-flow instructions per request for STM GB-tree and Lock GB-tree respectively. Compared to a GPU B+tree without concurrent control (the first bar in each group of Figure 1, which can be considered as the ideal condition.), the memory accesses of STM GB-tree and Lock GB-tree are 2.98X and 1.12X, respectively, and the control-flow instructions are 4.49X and 2.85X, respectively. Moreover, the memory accesses of STM GB-tree and Lock GB-tree account for 18% and 10% of the total instructions, respectively. The control-flow instructions of STM GB-tree and Lock GB-tree account for 30% and 25% of their total instructions, respectively.

To illustrate the situation in response time, we also collect maximal response time, minimal response time, and average response time of STM GB-tree and Lock GB-tree. To compare the performance, all average results are normalized to the average response time of the STM GB-tree. The maximal time and the minimal time of different methods are normalized to their average response time respectively for computing the variance. As the left two bars of the data in Figure 2 show, the variance of the response time of the methods mentioned above is large (40% for STM GB-tree and 36% for Lock GB-tree).

Linearizability [17] is critical for the correctness and fairness of concurrent processing. Concurrent processing is linearizable if the results are the same as that obtained from the sequential execution in their precedence order (usually in their real-time order). Linearizability can provide a vital correctness condition and fairness for concurrent requests, which is required in most concurrent systems [13]. In the design of STM GB-tree or Lock GB-tree, efforts mainly focus on exploiting the parallelism in GPUs. Neither of them guarantees linearizability.

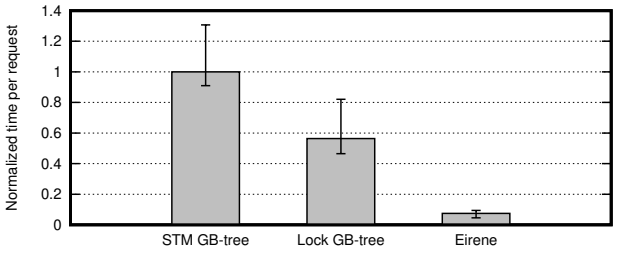


Figure 2. Normalized time per request.

3 Related Works

To handle concurrent requests more efficiently, many works aim to improve the index structure performance with software or hardware acceleration. This section discusses related works on index structure acceleration, concurrent GPU B+trees, and the *combining* synchronization technique.

Index Structure Acceleration. With the volume of concurrent requests increasing exponentially, efforts are made to accelerate concurrent index structure with parallel hardware, such as FPGAs for hash table [7, 26, 39], GPUs for hash table [43, 44] and skip list [29, 30], Hardware Transactional Memory (HTM) for B+tree [38], and GPU HTM [9, 10]. And also, some works are presented on improving *query* performance of index structure on GPUs [35, 41]. Compared to previous works, Eirene focuses on accelerating B+tree with GPU and addresses not only the *query* performance but also concurrent (mixed *query* and *update*) performance.

Concurrent GPU B+trees. Holey et al. [19] present a lightweight STM design with a simple STM interface that can scale well for many GPU threads. Awad et al. [4] present a concurrent GPU B-tree design. It uses fine-grained locks for concurrency control and is optimized for GPU SIMT to achieve coalesced memory accesses and avoid branch divergence. However, the two works cannot achieve satisfying performance and QoS and don't support linearizability semantics. Yan et al. [41] improve the query throughput of B+tree on GPUs using several optimizations. However, it doesn't support concurrent processing. In our work, we use the STM interfaces provided by [19] to guarantee the correctness during tree traversal for *update* requests, which provides lightweight STM primitives with little overhead. Based on it, we propose an optimistic strategy for STM on B+tree to minimize the possible fallback overhead caused by false transaction conflicts.

Combining Synchronization Technique. *Combining* synchronization is a technique used to reduce the synchronization conflicts for many-core environments. One or more threads become combiners and combine requests with the same key from other threads into fat requests, and the combined requests are processed by the combiners. It is experimentally shown that the combining-based implementation

outperforms the fine-grained locking implementation. Some works [5, 14, 28, 34] use the *combining* synchronization technique to improve the concurrency performance of various data structures such as heap, queue, and red-black tree. Aksenov et al. [2] propose a parallel-combining method that applies flat-combining to batch processing and demonstrates the performance benefits of explicit synchronization in concurrency scenarios. However, these works are designed for concurrent processing on CPUs but not GPUs. Compared to these works, Eirene is the first system to apply the *combining* synchronization technique to concurrent GPU B+trees.

4 Combining-based Concurrent Control

In this section, we introduce the designs of the combining-based concurrent control approach, which reduces the impact of conflicts and achieves linearizability.

4.1 Combining-based Synchronization

Combining synchronization is a low-overhead concurrent synchronization technique whereby the combiner threads combine concurrent requests with the same key and process them one by one to reduce the conflict overhead [2, 14, 16]. However, prior combining designs mainly focused on multi-core CPUs which cannot be applied to GPUs.

4.1.1 Combining-based synchronization.

When combining synchronization is applied to reduce the conflict overhead, a combiner needs to be able to detect the requests with the same key (find the key conflicts). Naively, it can be achieved by comparing the keys between any two requests. Since the incoming keys of different requests are randomly distributed, such detection is time-consuming. However, if the requests are sorted by their keys, the detection overhead will be much smaller because each request only needs to be compared with its adjacent requests. In Eirene, the requests are first sorted by their target keys, and the requests with the same keys are sorted by their logical timestamps. Note since these requests are buffered in time order when they reach the GPU, here we take the order in the buffer as their logical timestamps.

To combine the requests with the same key, the sorted requests are first scanned. If there exist multiple requests with the same key, and all of them are *query* requests or *update* requests, these requests are combined. Only the request with the largest timestamp will be issued to traverse the B+tree. If all requests for a specific key are *query*, the unissued *query* requests could share the result with the issued *query* request. If all requests for a specific key are *update*, the unissued *update* requests are ignored because their values will be overwritten by the last *update* request.

If there are mixed kinds of requests for the same key, i.e., *query* and *update*, the *update* request with the largest timestamp is issued, and the old (original) value in the leaf node is retrieved by the issued request. For a *query* request in the mixed requests, if there is a *delete* request before it without any other *update* request between them, its result should be null. If there is an *update* request before it without any *delete* request between them, it uses the value of this *update* request as its result. Otherwise, it uses the old (original) value retrieved by the issued request as its result. Thereby, all the unissued requests can calculate their results without traversing the B+tree.

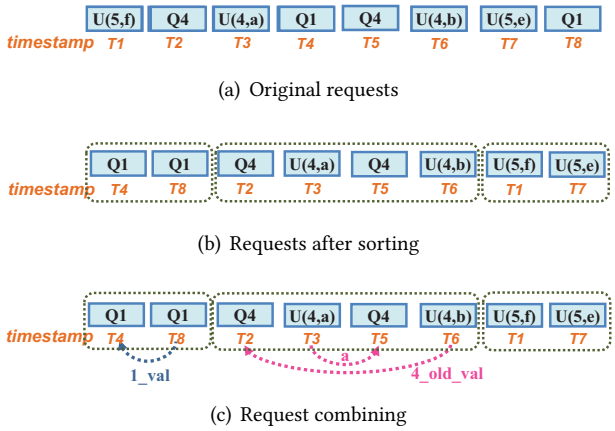


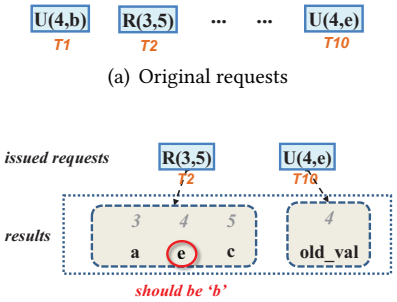
Figure 3. An example of how the combining approach works.

Here, an example is given to illustrate how the combining-based synchronization approach works. As shown in Figure 3, Qk denotes a *query* request with target key k. $U(k, v)$ denotes an *update* request to update the key k with value v. Tt represents a timestamp where smaller t corresponds to a smaller timestamp, which happens earlier. The requests in Figure 3(a) are original requests with timestamps. As Figure 3(b) shows, the sorted requests with the same key are adjacent in the timestamp order. In this example, the requests of target key 1 are all *query*, so $Q1$ with $T8$ is issued, and $Q1$ with $T4$ shares the result with $Q1$ with $T8$ (1_val). The requests of target key 5 are all *update*. Thus, only its last request ($U(5, e)$) is issued. The requests with target key 4 are of mixed kinds. After combining, the last *update* operation ($U(4, b)$) is issued, and the old (original) value (4_old_val) instead of value a or b is returned. There is no *update* whose timestamp is smaller than that of $Q4$ with $T2$, so its result is 4_old_val. $Q4$ with $T5$ obtains the value (a) of *update* $U(4, a)$, which is the nearest *update* operation before it, as its result.

After sorting and combining, the key conflicts among requests are eliminated because only one request for each key is issued. Meanwhile, all the other unissued requests with the same key can get their results according to their timestamps, which achieves linearizability [17].

4.1.2 Range Query.

Range query is a basic operation for concurrent B+tree that queries data within a range. Since a *range query* gets a range of keys instead of one key, a *range query* cannot be processed by the combining synchronization approach directly. Moreover, if *range queries* are still processed in their original way, they may get the wrong results because some *update* requests in their range may be combined with other requests.



(a) Original requests

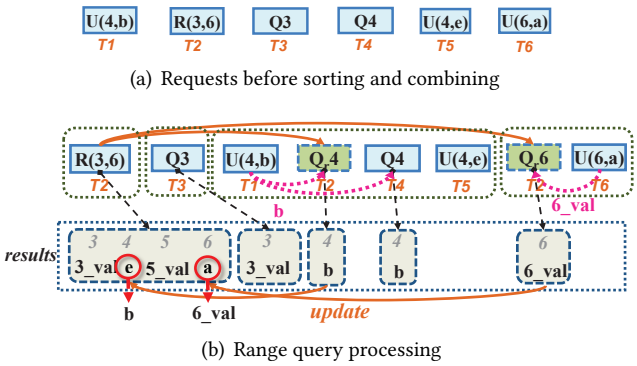
(b) Range error

Figure 4. An example of why the original *range query* would be wrong with the combining approach.

Figure 4 shows an example. $R(l, u)$ represents a *range query* with lower bound l and upper bound u. Figure 4(a) shows the original request sequence. All the requests for key 4 are shown in the figure. As Figure 4(b) shows, $R(3, 5)$ with a timestamp $T2$ should get the value b for key 4. However, after the requests for key 4 are combined, only $U(4, e)$ is issued. Therefore, $R(3, 5)$ can only get the old value of key 4 or get value e written by $U(4, e)$, which is the wrong value.

We introduce a mechanism to guarantee *range queries* can work correctly with our combining-based synchronization approach. When the buffered requests are processed, all *range queries* are also sorted with the other requests based on their lower bound values. After that, the sorted requests are scanned. If a key is in the range of a *range query* and there are some *update* requests for this key, an artificial *query* for this key with the *range query*'s timestamp is generated and inserted into the request sequence of that key based on its timestamp. A *range query* first gets its result in the original manner. For each key with an artificial *query*, its value is updated with the result of the corresponding artificial *query*.

Figure 5 gives an example of *range queries*. Figure 5(a) shows the original requests. $R(3, 6)$ is a *range query*, and it queries the keys from key 3 to key 6. Figure 5(b) shows how $R(3, 6)$ is processed. The requests are first sorted according to their keys and timestamps. Key 4 and key 6 are in the range of $R(3, 6)$. Moreover, there are *update* requests for them. Therefore, two artificial queries (Q_r4 and Q_r6) for $R(3, 6)$ are generated with the timestamp $T2$ and inserted into the corresponding positions. After these requests are



(a) Requests before sorting and combining

(b) Range query processing

Figure 5. An example for *range query* processing.

processed, Q_r4 and Q_r6 get their results (b and 6_val). After that, the *range query* result of key 4 is updated with that of Q_r4 (b), and the *range query* result of key 6 is updated with that of Q_r6 (6_val).

4.2 Combining-based Concurrent Control

Among concurrent requests, there are key conflicts and structure conflicts. In our concurrency control design, we first use our combining-based synchronization approach to eliminate key conflicts. Then, we use an STM-based method to process structure conflicts to guarantee correctness. The whole workflow of the combining-based concurrent control approach is shown in Algorithm 1.

When concurrent requests are transferred to GPUs, they are first processed by the combining-based synchronization approach (line 2) to eliminate key conflicts. After the requests with the same key are combined, there are only structure conflicts among issued requests. Then, we partition requests to be issued onto different GPU kernels according to their types (line 3): *query* kernel (for *query* requests and *range query* requests) and *update* kernel (for *update* requests including update, insertion, and deletion) (lines 3-5) [4, 43]. We perform such a kernel partition based on the following observations: for *query* requests, they only need to get the value without any tree structure modification. Even if they are not protected, they can be correctly executed. And not applying for synchronization protection on *query* requests can reduce the possibility of structure conflicts. The *query* kernel is launched first. After that, the *update* kernel is launched. These two kernels are executed separately. Because there are no key conflicts among issued requests, the execution order of the issued requests will not affect the returned results. There is no synchronization protection on *query* kernel, and some optimizations for *query* requests in previous work [41] can be applied to improve further performance for tree traversal (line19). For *update* kernel (lines 23-46), we use eager STM [19] to protect *update* requests, and the multiple-tree-regions splitting approach [12, 38] is used to

Algorithm 1 Combining-based Concurrent Control

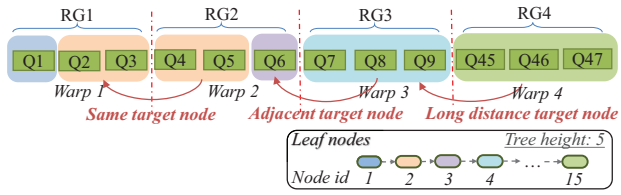
Declaration:
Reqs, ***QueryReqs***, ***UpdateReqs***: *pointers to the arrays of incoming requests, query requests and update requests.*
CombReqs: *pointer to the array of requests after combining.*

```
1:  procedure Combining-based Concurrent Control
2:      CombReqs = COMBINING(Reqs)
3:      QueryReqs, UpdateReqs = PARTITION(CombReqs)
4:      QUERY_KERNEL(QueryReqs)
5:      UPDATE_KERNEL(UpdateReqs)
6:      RESULT_CAL(Reqs,QueryReqs,UpdateReqs)
7:  end procedure
8:
9:  procedure COMBINING(Reqs)
10:     Sort(Reqs)
11:     CombReqs = Combine(Reqs)
12:     return CombReqs
13: end procedure
14:
15: procedure QUERY_KERNEL(QueryReqs)
16:     id = blockIdx.x * blockDim.x + threadIdx.x
17:     key = QueryReqs[id].key
18:     val = &QueryReqs[id].val
19:     leaf = findLeaf (root, key)
20:     val = findValInLeaf(leaf, key)
21: end procedure
22:
23: procedure UPDATE_KERNEL(UpdateReqs)
24:     id = blockIdx.x * blockDim.x + threadIdx.x
25:     key = UpdateReqs[id].key
26:     val = UpdateReqs[id].val
27: RETRY:                                   ▷ inner nodes region
28:     if retry_count < THRESHOLD then
29:         leaf = findLeaf (root, key)
30:     else
31:         TX_BEGIN()
32:         leaf = findLeaf(root,key)
33:         TX_END()
34:     end if
35:     leafvers = leaf.version
36:
37:     TX_BEGIN()                           ▷ leaf nodes region
38:     if leafvers == leaf.version & key in range(leaf)  then
39:         update(key, val, leaf)
40:     else
41:         retry_count++
42:         TX_END()
43:         goto RETRY
44:     end if
45:     TX_END()
46: end procedure
```

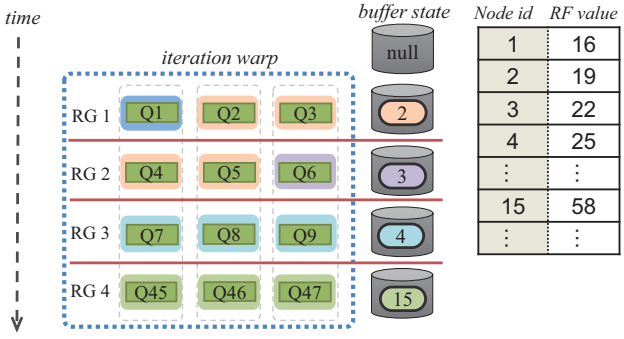
reduce structure conflicts. The tree traversal is partitioned into two parts: inner node traversal (line 29) and leaf node operations (line 39). Only the leaf node operations are always protected by STMs (lines 37-45), and the inner node traversal is firstly unprotected (line 29). When a conflict occurs during the request processing, it will retry. If the number of retries reaches the threshold, the inner node traversal for the corresponding request is protected by an STM (lines 30-34). After the inner node traversal, the request gets the reference to the target leaf node. There may be a consistency problem in that the target leaf node is split after the request gets its reference. To solve this problem, we use a validation mechanism between inner node traversal and leaf node operations [38]. Each leaf node has a version number, which is also stored by its parent node. Once the leaf node is split, its version number is increased atomically by one, and it will update the version number stored in its parent node. When the reference to a leaf node is obtained, the corresponding version number is also obtained (line 35). Before the leaf node is operated, the recorded version number is compared to that of the leaf node. If two version numbers are not equal, a conflict has occurred, and the request processing is aborted and retried (lines 38-44).

After the issued requests finish, they will retrieve the old values of the node. As discussed in subsection 4.1, the unissued requests calculate their results according to their dependency relationship with the issued requests by invoking the *RESULT_CAL* method (line 6). This correction procedure executes in parallel on GPU.

5 Locality-Aware Warp Reorganization



(a) Original warps



(b) Iteration warps

Figure 6. An example of the locality-aware warp reorganization approach.

After the requests are sorted and combined, the adjacent requests that are issued will likely access the same or adjacent leaf nodes. When these requests are partitioned into request groups (RGs) and mapped to different warps, these warps may also access the same or adjacent leaf nodes. Figure 6(a) shows such an example where Qk represents a request with key k. Multiple requests with the same color mean their target keys are in the same leaf node. For example, $Q3$ in RG1 and $Q4$ in RG2 have the same color (pink), indicating that their target keys are both in the leaf node 2. The tree height is 5 in this example. After these requests are partitioned into different RGs, the adjacent RGs will access the same or adjacent leaf nodes. For example, RG2 and RG1 will access the same leaf node 2, RG3 will access the leaf node 4 that is adjacent to the leaf node 3 accessed by RG2. Therefore, if an RG can get its target leaf node along with the linked leaf nodes from the target leaf node of its adjacent RG instead of traversing from the root, traversal steps (the number of nodes that need to be traversed) could be reduced compared to the traversal from the root. However, in reality, after these RGs are mapped to different warps, it is difficult to reuse such a locality because the warps are dynamically scheduled in the same SM, and the scheduling order cannot be predefined.

To utilize the locality among adjacent RGs, an approach called locality-aware warp reorganization is proposed here. In this approach, several adjacent RGs are organized into a warp and executed in a loop manner (called iteration warp). The RGs in an iteration warp are mapped to a warp, and executed one by one. There is a shared buffer for each iteration warp to store the target leaf node. When an iteration warp is executed, its first RG traverses the tree in the original way (from root to leaf, also called vertical traversal). After an RG finishes its execution, the last accessed leaf node (maybe the nearest to the next group) is stored in the buffer. The following RG will get its target leaf node by traversing the linked leaf nodes from the buffered node (called horizontal traversal). For *update* requests, the horizontal traversal is protected with locks or STM. If conflicts happen, they will retry and traverse the tree through vertical traversal. Because there may be structure conflicts among the original adjacent RGs, such a design can avoid some structure conflicts after they are combined into an iteration warp.

If the steps or the traversed node number between the buffered left node and the target leaf node is smaller than the tree height, such a design can perform better. However, if the step is larger than the tree height, it will degrade performance. To avoid such a condition, a mechanism is designed to help Eirene decide which way to traverse, vertically or horizontally. A range field (RF) is extended in each leaf node, and it is used to record the minimal key of a node whose traversal steps from this node are tree height plus one. For example, suppose the tree height is 5 and the minimal key of leaf node 6 is 16; the RF value of leaf node 1 will be 16. These RF values are initialized when the tree is constructed. The RF value of a leaf node is updated only when this node is the starting point of a horizontal traversal, and the traversal step is larger than the tree height. When the leaf node information is stored in the iteration warp buffer, its RF value is also stored in the buffer. For the next iteration, it will first check whether its key is larger than the buffered RF value. If so, the next iteration will vertically traverse the tree. Otherwise, the tree is traversed horizontally. Since the requests in one iteration are executed in a SIMT manner, the executed steps of different threads follow the maximal steps. Therefore, the maximal target key in an iteration instead of the minimal one is used to compare with the stored RF value.

Another design consideration is the iteration number in a warp. A large iteration number would increase the locality among the iterations. However, it would sacrifice the parallelism available among warps. To fully use the computing resources, the RGs are evenly distributed to different SMs on a GPU. Then, they are organized into iteration warps executed on each SM.

Figure 6(b) shows how this approach works. Suppose the tree height is 5, and the RF values of different leaf nodes are as shown in the table in Figure 6(b). Instead of being mapped to different warps, four RGs are organized as an iteration warp. Before the first iteration is executed, the buffer is empty. The requests in the first iteration (RG1) traverse the tree from the root to the leaf nodes. After the requests in RG1 finish, the buffer is filled with its maximal accessed leaf node (leaf node 2). When RG2 is executed, it gets the leaf node 2 from the buffer and checks whether its maximal key (6) is smaller than the buffered RF value (19). Since the maximal key is smaller than the RF value, it horizontally traverses the tree. After RG3 is finished, leaf node 4 is stored in the buffer. Before RG4 is executed, it first checks whether its maximal key (47) is larger than the buffered RF value (25). Since its maximal key is larger than the buffered RF value, it vertically traverses the tree.

6 Proof Sketch

This section outlines the arguments about correctness and linearizability for combining-based concurrent control. The main proof obligation is to show that the design can work correctly and the results of concurrent requests processed by it are the same as those of processing these requests in their logical timestamp order.

Proof outline. Linearizability [17] is a correctness condition for concurrency. It can be defined as the execution results of concurrent requests being consistent with that of the sequential execution in real-time or timestamp order. Thus, whether a design satisfies linearizability depends on how the requests with the same key are processed. If the execution results of the requests with the same key are equivalent to those of sequential execution in timestamp order, the design is linearizable. In our design, each request

has a logical timestamp according to its arrival order in the buffer. The logical timestamp has been determined when the request reaches the GPU. If the requests with the same key are all *query* or *range query* requests, no matter what order these requests are executed, they will always get the same results. Thus, in combining-based concurrent control, requests can get the same results as that in sequential execution. For some keys with mixed request types, they are first sorted by their timestamps, and their dependence relationship is constructed in our combining-based synchronization approach. Then only one request is issued, and the other unissued ones will get their results based on the dependence relationship. Therefore, such a design can achieve linearizability because the requests with the same key get consistent results as they are sequentially executed in their timestamp order.

After the combining-based synchronization, there exist only structure conflicts. Hence, it's safe to partition these requests into different kernels. For an *update* request, the tree traversal is partitioned into inner node traversal and leaf node operations. The inner nodes are first traversed without any STM protection. When it fails, it will retry until the threshold is reached. Then STMs protection is turned on. Moreover, the leaf node operations are always protected with STMs. The correctness of such an approach has been proved in [38]. Thus, the combining-based concurrent control design can work correctly.

In the locality-aware warp reorganization approach, different traversal manners are chosen to search the target leaf node through exploiting locality. The locality-aware warp reorganization does not impact the results of the concurrent requests, so it is harmless for the linearizability. Since horizontal traversals execute on leaf nodes and are always protected with STMs for *update* requests, the correctness of locality-aware warp reorganization can be guaranteed.

7 Implementation of Eirene

We have implemented a GPU concurrent B+tree of Eirene using CUDA C++ with 2200 lines of code. Here, we describe the details of the Eirene implementation.

The tree structure is a regular B+tree where the inner node contains keys and child pointers. A child pointer refers to the location of a child. The whole tree structure is stored in GPU global memory. We buffer the concurrent requests in the CPU end and transfer them to GPU after the number reaches a configurable threshold (1 million in the current implementation). In the combining-based synchronization approach, the radix sorting algorithm in the CUB library [32] is used to sort the bulk of requests based on their keys and logical timestamps. After sorting, we combine these requests by scanning each request with their neighbors in parallel, where each GPU thread scans one request. Then we implement two CUDA kernels (*query* and *update*) to issue different type requests where the *query* kernel is launched before the *update* kernel. Since the *query* kernel only processes *query* or *range query* requests which do not modify B+tree, we process these requests without any STM protection, and the narrowed-threads-group optimization in [41] is used to speed up *query* performance. The *update* kernel process executes tree traversal and *update* operations with an optimistic STM approach. We implement the STM with eager conflict detection [19]. Theoretically, synchronization schemes other than STM can be used in the implementation, such as fine-grained locks. To support the locality-aware warp reorganization approach, we reorganize multiple continuous warps together to form an iteration warp. In the original warp, each thread only processes one request. In the iteration warp, each thread will process requests from the same position of multiple original warps one by one.

8 Evaluation

In this section, we evaluate the performance of Eirene and try to answer the following questions:

- Can Eirene achieve better performance than state-of-the-art systems like Lock GB-tree?
- Can Eirene achieve better QoS?
- Does Eirene solve the challenges in Section 2?
- How does each design choice affect performance?
- How about *range query* performance for Eirene?

8.1 Experimental Methodology

Our experiments are conducted on a 64-core server (2-sockets 32-core AMD EPYC 7532s) with an Ampere GPU (NVIDIA A100). We compile all codes using GCC 7.5.0 and CUDA 11 on Ubuntu 18.04 (kernel 5.4.0) with the O3 optimization option. We implement a prototype of Eirene as described in Section 7. The STM GB-tree [19] and the Lock GB-tree [4] are used as the performance baseline. The Lock GB-tree is open source, and we implement the STM GB-tree according to the paper [19].

To demonstrate the performance effectiveness of the design, we evaluate Eirene with Yahoo! Cloud Serving Benchmark (YCSB) [11], which is a widely used benchmark for key-value storage. Each record has a 32-bit key and a 32-bit value. We use the request data set with different *query*/*update* ratios. The default ratio of *query*/*update* is 95%/5%. The default distribution is *Uniform*. These requests are processed on different tree sizes, including 2^{23}, 2^{24}, 2^{25}, and 2^{26}, and the default tree size is 2^{23}. All results are averaged by 5-time executions. All the evaluations of Eirene include the sorting time in the combining-based concurrent control approach. In this paper, we focus on the B+tree performance on GPU for concurrent requests. Thus the results do not include the transfer time between CPU and GPU. Although the transfer time between CPU to GPU is nonnegligible, we do not consider it in our current design and evaluation to compare it with the existing works. The individual response time of

each request is averaged over a batch of arriving requests. The average response time per request is obtained by averaging the individual response time from fifty tests. We also use Nsight Compute [31] to collect runtime information for analysis.

8.2 Overall Evaluation

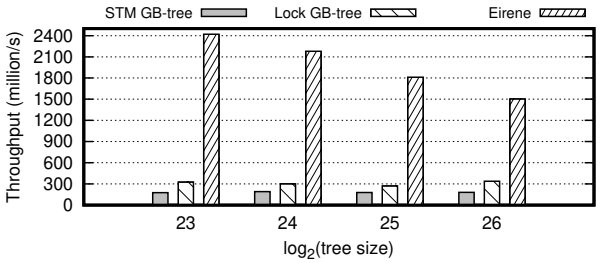


Figure 7. Overall performance.

We first compare the performance of Eirene with those of the STM GB-tree [19] and the Lock GB-tree [4]. As the data in Figure 7 show, Eirene can achieve a throughput of 2.4 billion per second on average under the default configuration, which is about 13.68X speedup to that of the STM GB-tree and about 7.43X speedup to that of the Lock GB-tree. With the tree size increasing, the performance of Eirene decreases. The reason is that the traversal steps become larger, with tree size increasing. Moreover, threads performing uniform accesses will be more spread out from one another on average, which leads to divergence for the same input.

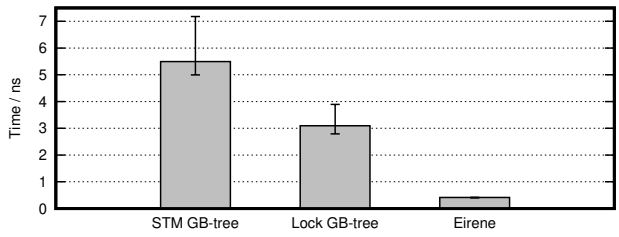


Figure 8. Time per request.

The variance of response time is a critical metric for QoS [1, 21, 46]. To illustrate the efficiency of Eirene, we collect the maximal response time, the minimal response time, and the average response time of Eirene. As the data in Figure 8 show, the average response time of Eirene is about 0.41 ns per request under the default configuration, which has about 92.6% average response time reduction compared to STM GB-tree (5.5 ns) and about 86.7% average response time reduction compared to Lock GB-tree (3.1 ns). For Eirene, the maximal response time is 0.42 ns, and the minimal response time is 0.40 ns. The variance of the response time of Eirene is only 5%, which means Eirene can achieve a better QoS (more stable service) compared to those of prior methods (40% for STM GB-tree [19] and 36% for Lock GB-tree [4]). For the length of the response time of a request, the uncertain part is the time for conflict detection and resolution. It is unpredictable where the conflict occurs and how many retries are required to get a lock or a successful STM commit. With our optimizations, no conflict detection is needed for *query* requests. For *update* requests, conflict detection and resolution are only needed when leaf nodes are operated for structure conflicts, and the conflict number is only about 4.8% of that of the STM GB-tree. Therefore, the uncertainty for response time is significantly reduced (the detailed conflict reduction data for the three optimizations is shown in Section 8.3), which leads to a much smaller variance of response time.

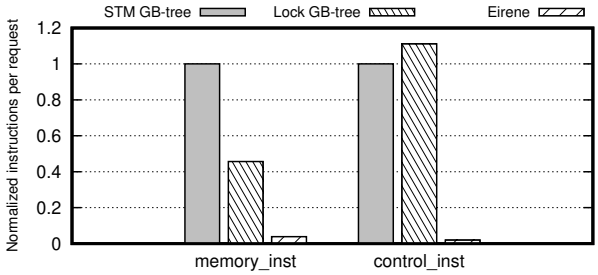


Figure 9. Metrics profiling of Eirene.

Furthermore, we profile several metrics discussed in Section 2, including the average number of global memory accesses and control-flow instructions for each request. The normalized results in Figure 9 show that all the metrics are reduced greatly. The number of global memory accesses and the control-flow instructions of Eirene is about 3.9% and 2.0% of those of the STM GB-tree respectively, about 8.5% and 1.8% of those of the Lock GB-tree respectively. We also collect the number of conflicts per request. The results show the average number of conflicts for each request for Eirene is about 4.8% of those of the STM GB-tree.

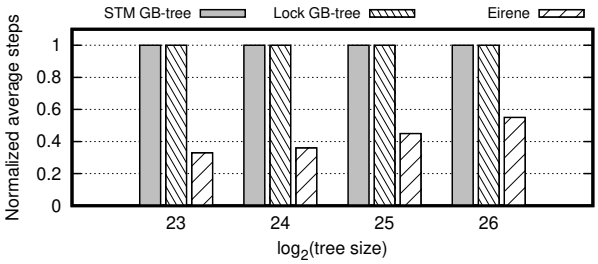


Figure 10. Traversal steps.

In addition, we compare the traversal steps of the STM GB-tree, the Lock GB-tree, and Eirene. As the data in Figure 10

show, the traversal steps of the STM GB-tree and the Lock GB-tree are approximately equal because it only relates to the tree height. The number of traversal steps of Eirene is much smaller than those of the STM GB-tree and the Lock GB-tree. For the tree size of 2^{23}, the traversal steps of Eirene are about 67% less than those of the STM GB-tree and the Lock GB-tree. When the tree size increases, the average steps of Eirene grow. The reason is that the larger the tree is, the higher the tree height is, and the key distribution in the leaf node is more dispersive. Therefore, more horizontal traversal steps are needed for large trees. For example, the average horizontal traversal steps for a 2^{23} tree are about 1.5, while those for a 2^{26} tree are about 3.4.

8.3 Different Design Choices

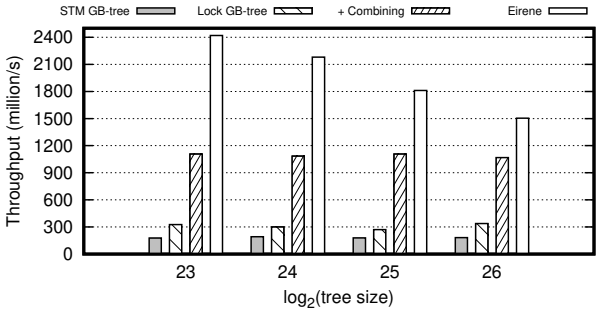


Figure 11. Different design choices.

To understand the performance influence of each optimization, we evaluate the optimizations separately. The results are shown in Figure 11. Since the Eirene implementation is STM-based, we use the result of the STM GB-tree as the baseline. '*+ Combining*' means that only the combining-based concurrent control optimization is applied based on the STM GB-tree. Eirene enables locality-aware warp reorganization based on '*+ Combining*', i.e., Eirene. Compared to the STM GB-tree, the combining-based concurrent control gets a 6.26X speedup. The performance gains of the combining-based concurrent control approach mainly come from the simplification of concurrency control instead of key reduction. Because key conflicts are eliminated, the subsequent conflict detection and resolution of the tree traversal can be greatly simplified, resulting in a significant reduction in memory accesses and control flow instructions. After applying the locality-aware warp reorganization optimization, Eirene achieves about a 13.68X speedup as it reduces the traversal steps by exploiting the locality among adjacent RGs. Moreover, the structure conflicts are also reduced.

Figure 12 shows the contribution of combining-based concurrent control and locality-aware warp reorganization on reducing conflicts, memory accesses, and control instructions, respectively. *combining* refers to the effect of the combining-based concurrent control. *locality* refers to the locality-aware

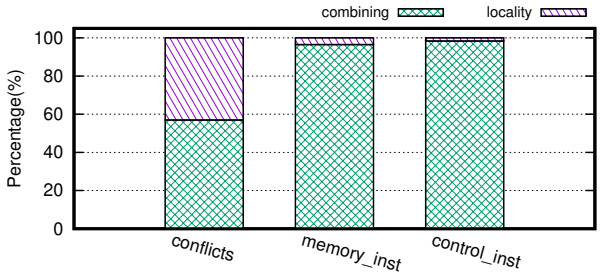


Figure 12. The contribution of different optimizations on reducing conflicts, memory accesses, and control instructions, respectively.

warp reorganization optimization. For *combining*, the requests accessing the same key are combined, and only one request for each key is issued. Then the *query* and *update* requests are distributed to different kernels as described in Section 7. Although the number of key conflicts in our default input distribution is small and the reduction percentage is not high for these metrics, benefit from different kernels for different requests, there is no concurrency control in *query* kernel, and the number of structure conflicts is significantly reduced. Thus, it eliminates about 57% of conflicts (key conflicts). Moreover, it reduces about 96.5% of memory access and 98.4% of control-flow instructions. For *locality*, it organizes multiple originally adjacent warps into an iteration warp, and the structure conflicts among them can be reduced. Therefore, it can reduce about 43% of structure conflicts. Due to the reduced number of traversal steps, the number of memory accesses for reading nodes and the number of control-flow instructions for key comparisons are reduced. It brings about a 3.5% reduction in the number of memory accesses and about a 1.6% reduction in the number of control-flow instructions.

8.4 Range Queries

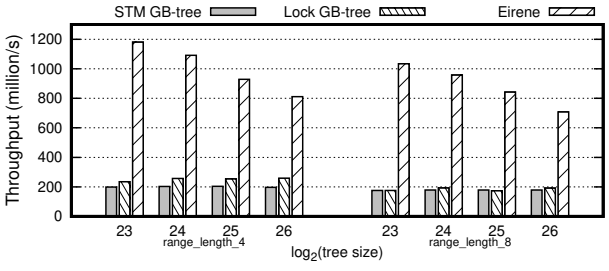


Figure 13. Range query.

Here, we evaluate pure *range query* performance under different configurations. As the data in Figure 13 show, Eirene achieves a throughput of 1181 million per second for range query length 4 and a throughput of 1034 million per second

for range query length 8. In comparison, the Lock GB-tree achieves a throughput of 235 million per second for range query length 4 and a throughput of 175 million per second for range query length 8. These results show that Eirene is effective for different configurations. Eirene achieves 5.94X overall performance speedup compared to the Lock GB-tree.

9 Conclusion

In this paper, we design a combining-based concurrency control system, called Eirene, for concurrent GPU B+trees to improve performance and QoS. It keeps linearizability by eliminating key conflicts and optimizing structure conflicts. Experimental results show Eirene is efficient. It can achieve a throughput of 2.4 billion per second with only a 5% variance of response time.

Acknowledgments

We appreciate our shepherd, Patrick Eugster, and the anonymous reviewers for their constructive comments. We are grateful to support from the National Natural Science Foundation of China (No. 62141211) and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing.

References

[1] Chaima Abid, Marouane Kessentini, and Hanzhang Wang. 2020. Early prediction of quality of service using interface-level metrics, code-level metrics, and antipatterns. Information and Software Technology 126 (2020), 106313. https://doi.org/10.1016/j.infsof.2020.106313

[2] Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto. 2018. Parallel Combining: Benefits of Explicit Synchronization. In 22nd International Conference on Principles of Distributed Systems (OPODIS 2018) (OPODIS '18, Vol. 125), Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 11:1–11:16. https://doi.org/10.4230/LIPIcs.OPODIS.2018.11

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

[4] 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 Symposium on Principles and Practice of Parallel Programming (Washington, District of Columbia) (PPoPP '19). ACM, New York, NY, USA, 145–157. https://doi.org/10.1145/3293883.3295706

[5] Gal Bar-Nissan, Danny Hendler, and Adi Suissa. 2011. A Dynamic Elimination-Combining Stack Algorithm. In Principles of Distributed Systems (Lecture Notes in Computer Science), Antonio Fernàndez Anta, Giuseppe Lipari, and Matthieu Roy (Eds.). Springer, Berlin, Heidelberg, 544–561. https://doi.org/10.1007/978-3-642-25873-2_37

[6] Jeff Barr. 2020. Amazon Prime Day 2020 – Powered by AWS. Retrieved March 16, 2021 from https://aws.amazon.com/blogs/aws/amazon-prime-day-2020-powered-by-aws/

[7] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. 2013. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 13). USENIX Association, San Jose, CA. https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/blott

[8] Daniel Cederman, Philippas Tsigas, and Muhammad Tayyab Chaudhry. 2010. Towards a Software Transactional Memory for Graphics Processors. In Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization (EG PGV'10). Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 121–129. https://doi.org/10.2312/EGPGV/EGPGV10/121-129

[9] S. Chen and L. Peng. 2016. Efficient GPU hardware transactional memory through early conflict resolution. In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). 274–284. https://doi.org/10.1109/HPCA.2016.7446071

[10] Sui Chen, Lu Peng, and Samuel Irving. 2017. Accelerating GPU Hardware Transactional Memory with Snapshot Isolation. In Proceedings of the 44th Annual International Symposium on Computer Architecture (Toronto, ON, Canada) (ISCA '17). ACM, New York, NY, USA, 282–294. https://doi.org/10.1145/3079856.3080204

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing (Indianapolis, Indiana, USA) (SoCC '10). ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[12] Nuno Diegues and Paolo Romano. 2016. STI-BT: A Scalable Transactional Index. IEEE Transactions on Parallel and Distributed Systems 27, 8 (2016), 2408–2421. https://doi.org/10.1109/TPDS.2015.2485267

[13] Simon Doherty and John Derrick. 2016. Linearizability and Causality. In Software Engineering and Formal Methods, Rocco De Nicola and Eva Kühn (Eds.), Vol. 9763. Springer International Publishing, Cham, 45–60. https://doi.org/10.1007/978-3-319-41591-8_4

[14] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the Combining Synchronization Technique. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12). Association for Computing Machinery, New York, NY, USA, 257–266. https://doi.org/10.1145/2145816.2145849

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

[16] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10). Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[17] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[18] Chandima HewaNadungodage, Yuni Xia, and John Jaehwan Lee. 2016. GStreamMiner: A GPU-accelerated Data Stream Mining Framework. In Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (Indianapolis, Indiana, USA) (CIKM '16). ACM, New York, NY, USA, 2489–2492. https://doi.org/10.1145/2983323.2983341

[19] Anup Holey and Antonia Zhai. 2014. Lightweight Software Transactions on GPUs. In 2014 43rd International Conference on Parallel Processing. IEEE, MN, USA, 461–470. https://doi.org/10.1109/ICPP.2014.55

[20] IBM Corp. 2021. DB2 Database. https://www.ibm.com/products/db2-database

[21] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient Cache Sharing with Strict Qos for Latency-Critical Workloads. In Proceedings of the 19th International Conference on Architectural Support for

Programming Languages and Operating Systems (Salt Lake City, Utah, USA) (ASPLOS '14). Association for Computing Machinery, New York, NY, USA, 729–742. https://doi.org/10.1145/2541940.2541944

[22] Peter Kieseberg, Sebastian Schrittwieser, Lorcan Morgan, Martin Mulazzani, Markus Huber, and Edgar Weippl. 2011. Using the Structure of B+-trees for Enhancing Logging Mechanisms of Databases. In Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services (Ho Chi Minh City, Vietnam). ACM, New York, NY, USA, 301–304. https://doi.org/10.1145/2095536.2095588

[23] Marios Kogias, Stephen Mallon, and Edouard Bugnion. 2019. Lancet: A Self-Correcting Latency Measuring Tool. In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (Renton, WA, USA) (USENIX ATC '19). USENIX Association, USA, 881–895. https://www.usenix.org/conference/atc19/presentation/kogias-lancet

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

[25] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, and Henk Corporaal. 2016. SFU-Driven Transparent Approximation Acceleration on GPUs. In Proceedings of the 2016 International Conference on Supercomputing (Istanbul, Turkey) (ICS '16). Association for Computing Machinery, New York, NY, USA, Article 15, 14 pages. https://doi.org/10.1145/2925426.2926255

[26] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In Proceedings of the 26th Symposium on Operating Systems Principles (Shanghai, China) (SOSP '17). Association for Computing Machinery, New York, NY, USA, 137–152. https://doi.org/10.1145/3132747.3132756

[27] Microsoft Corp. 2021. MS SQL Server. https://www.microsoft.com/en-us/sql-server/

[28] Changwoo Min and Young Ik Eom. 2015. Integrating Lock-Free and Combining Techniques for a Practical and Scalable FIFO Queue. IEEE Transactions on Parallel and Distributed Systems 26, 7 (July 2015), 1910–1922. https://doi.org/10.1109/TPDS.2014.2333007

[29] Prabhakar Misra and Mainak Chaudhuri. 2012. Performance Evaluation of Concurrent Lock-free Data Structures on GPUs. In 2012 IEEE 18th International Conference on Parallel and Distributed Systems. IEEE, Singapore, Singapore, 53–60. https://doi.org/10.1109/ICPADS.2012.18

[30] 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). IEEE, Portland, OR, 246–259. https://doi.org/10.1109/PACT.2017.13

[31] NVIDIA. 2022. Nsight Compute. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html.

[32] NVIDIA. 2022. NVIDIA/Cub. https://github.com/NVIDIA/cub.

[33] Oracle Inc. 2021. Oracle Database. https://www.oracle.com/database/technologies/

[34] Sergio Sainz-Palacios. 2019. Flat Combined Red Black Trees. arXiv:1912.11417 [cs] (Dec. 2019). arXiv:1912.11417 [cs] http://arxiv.org/abs/1912.11417.

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

[36] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System. In Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (ATEC '96). USENIX Association, USA, 1.

[37] Xiongchao Tang, Jidong Zhai, Xuehai Qian, and Wenguang Chen. 2019. PLock: A Fast Lock for Architectures with Explicit Inter-Core Message Passing. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (Providence, RI, USA) (ASPLOS '19). Association for Computing Machinery, New York, NY, USA, 765–778. https://doi.org/10.1145/3297858.3304030

[38] Xin Wang, Weihua Zhang, Zhaoguo Wang, Ziyun Wei, Haibo Chen, and Wenyun Zhao. 2017. Eunomia: Scaling Concurrent Search Trees Under Contention Using HTM. SIGPLAN Not. 52, 8 (Jan. 2017), 385–399. https://doi.org/10.1145/3155284.3018752

[39] Wei Liang, Wenbo Yin, Ping Kang, and Lingli Wang. 2016. Memory efficient and high performance key-value store on FPGA using Cuckoo hashing. In 2016 26th International Conference on Field Programmable Logic and Applications (FPL). IEEE, Lausanne, Switzerland, 1–4. https://doi.org/10.1109/FPL.2016.7577355

[40] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian. 2014. Software Transactional Memory for GPU Architectures. In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (Orlando, FL, USA) (CGO '14). ACM, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/2581122.2544139

[41] 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 (Washington, District of Columbia) (PPoPP '19). ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/3293883.3295704

[42] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. 2017. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In 2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA, 181–193. https://www.usenix.org/conference/atc17/technical-sessions/presentation/zhang

[43] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: A Case for GPUs to Maximize the Throughput of in-Memory Key-Value Stores. Proc. VLDB Endow. 8, 11 (July 2015), 1226–1237. https://doi.org/10.14778/2809974.2809984

[44] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Li, Xiaodong Zhang, Bingsheng He, Jiayu Hu, and Bei Hua. 2017. A Distributed In-Memory Key-Value Store System on Heterogeneous CPU—GPU Cluster. The VLDB Journal 26, 5 (Oct. 2017), 729–750. https://doi.org/10.1007/s00778-017-0479-0

[45] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: A Key-Value Store for Optane Persistent Memory. Association for Computing Machinery, New York, NY, USA, 194–209. https://doi.org/10.1145/3447786.3456237

[46] Yiwen Zhang, Kaibin Wang, Qiang He, Feifei Chen, Shuiguang Deng, Zibin Zheng, and Yun Yang. 2021. Covering-Based Web Service Quality Prediction via Neighborhood-Aware Matrix Factorization. IEEE Trans. Serv. Comput. 14, 5 (Sept. 2021), 1333–1344. https://doi.org/10.1109/TSC.2019.2891517

A Artifact Appendix

A.1 Abstract

This artifact includes the source codes and testing scripts to reproduce all results in Section 8 for this paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** CUB library

- **Program:** CUDA and C/C++ codes
- **Compilation:** gcc 7.5.0, NVCC 11.4 and CMake 3.10
- **Binary:** CUDA executables
- **Run-time environment:** Ubuntu 18.04 LTS with CUDA 11.4 and GPU driver 515; Python 3.6.9; Nsight compute 2022
- **Hardware:** NVIDIA Ampere GPU with at least 20 GiBs of DRAM and at least 20 GiBs of CPU DRAM
- **Metrics:** Throughput, QoS, Memory instructions and Control-flow instructions
- **Output:** Throughput, QoS, Memory instructions and Control-flow instructions
- **How much disk space required (approximately)?:** More than 20GB
- **How much time is needed to prepare workflow (approximately)?:** About 30 minutes
- **How much time is needed to complete experiments (approximately)?:** About 20 hours
- **Publicly available?:** No

A.3 Hardware Requirements

To better reproduce experiment results, we suggest an NVIDIA A100 GPU with 40 GiBs memory.

A.3.1 Software Requirements. Our evaluation requires the CUDA GPU driver 515, NVCC 11.4 and gcc 7.5.0 (or later) compiler. To profiling GPU metrics, Nsight compute 2022 are needed to install. To run the test scripts, Python 3.6.9 with numpy is required. You run use the docker 'nvidia/cuda:11.4.1-devel-ubuntu18.04'. The artifacts have been tested on Ubuntu 18.04, and are expected to run correctly under other Linux distributions.

A.4 Installation.

Source codes can be complied using GNU make to build the executables of Eirene, STM GB-tree and Lock GB-tree.

1. Decompress the artifact archive
   $ tar xzvf artifact-eirene-ppopp23-submit.tgz
   $ cd artifact-gbtree-ppopp23-submit
2. The building commands of Eirene, STM GB-tree and Lock GB-tree are wrapped in single script, you can run it to build all three executables.
   $ ./scripts/click-to-compile.sh
3. After this, three executables (*eirene*, *stm-gbtree*, *lock-gbtree*) are located in three build directories (*eirene/build/*, *stm-gbtree/build/*, and *lock-gbtree/build/*).

A.5 Data Set.

In the evaluation, we generate building tree data of different tree sizes, including 2^{23}, 2^{24}, 2^{26} and 2^{26}. And we generate test data with 95%/5% *query*/*update* ratio.

Generate test dataset by running the script:
$ ./scripts/click-to-gen-testdata.sh
All test data will be generated into directory *./dataset/*.

A.6 Experiment workflow

After compiling and generating dataset, use the *test-all.py* to run all the experiments to compare Eirene, STM GB-tree and Lock GB-tree. Use '– – *case*' parameter to identify the test metrics. The options for '– – *case*' can be tp, qos, range , profile. The test script will run executables of Eirene, STM GB-tree and Lock GB-tree respectively for 5 times, then output the average results.

A.6.1 Throughput.

Test throughput of Eirene, STM GB-tree and Lock GB-tree.
$ python3 ./scripts/test-all.py –case tp
It will output the throughput of Eirene, STM GB-tree and Lock GB-tree under 95% *query* and 5% *update* workload for four tree sizes (2^{23}, 2^{24}, 2^{25}, 2^{26}).

A.6.2 QoS.

Test QoS of Eirene, STM GB-tree and Lock GB-tree.
$ python3 ./scripts/test-all.py –case qos
It will output the max response time, min response time, average response time and time variance f Eirene, STM GB-tree and Lock GB-tree under 95% *query* and 5% *update* workload for 2^{23} tree sizes.

A.6.3 Range Query.

Test throughput of range query with length 4 and 8 of Eirene, STM GB-tree and Lock GB-tree..
$ python3 ./scripts/test-all.py –case range
It will output the *range query* throughput of Eirene, STM GB-tree and Lock GB-tree for four tree sizes (2^{23}, 2^{24}, 2^{25}, 2^{26}).

A.6.4 Profile.

Profile memory access and control-flow instructions under default workload.
$ python3 ./scripts/test-all.py –case profile
It will output the memory accesses and control-flow instructions of Eirene, STM GB-tree and Lock GB-tree under 95% *query* and 5% *update* workload for 2^{23} tree size.

A.7 Evaluation and expected results

All results in Section 8 are expected to be reproduced from the data in the artifact.

A.8 Notes

None.