Accelerating Merkle Patricia Trie with GPU

Yangshen Deng*
Southern University of Science and Technology
AlayaDB AI
dengys2022@mail.sustech.edu.cn

Muxi Yan*
Southern University of Science and Technology
AlayaDB AI
yanmx2022@mail.sustech.edu.cn

Bo Tang†
Southern University of Science and Technology
AlayaDB AI
tangb3@.sustech.edu.cn

ABSTRACT

Merkle Patricia Trie (MPT) is a type of trie structure that offers efficient lookup and insert operators for immutable data systems that require multi-version access and tamper-evident controls, such as blockchains and verifiable databases. The performance of these systems is critically dependent on the throughput of the underlying index structure MPT. In this paper, we present a novel approach to accelerate MPT by leveraging the massive parallelism of GPU. However, achieving it is challenging as (i) lock-free data structures are difficult to implement and (ii) traditional fine-grained locking does not scale on GPU.

To address them, we first analyze the technical challenges of accelerating MPT via GPU, including node splitting conflicts and hash computing conflicts caused by parallel insert operations. We then propose a lock-free algorithm PhaseNU and a lock-based algorithm LockNU on GPU to resolve the node splitting conflict. We also devise a decision model for users to choose the proper one for different workloads. We next propose a GPU-based hash-compute algorithm PhaseHC to avoid hash computing conflicts. Last, we demonstrate the effectiveness of our proposed techniques by: (i) integrating them into both the real-world blockchain system Geth and verifiable database LedgerDB, and demonstrating its superiority with corresponding workloads; and (ii) conducting extensive experimental studies on two real-world datasets and one synthetic dataset. Our proposed solutions significantly outperform the deployed MPT solution in Geth in all datasets.

PVLDB Reference Format:
Yangshen Deng, Muxi Yan, Bo Tang. Accelerating Merkle Patricia Trie with GPU. PVLDB, 17(8): 1856 - 1869, 2024.
doi:10.14778/3659437.3659443

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DBGroup-SUSTech/GPU-Merkle-Patricia-Trie.

1 INTRODUCTION

Merkle Patricia Trie (MPT) is a cryptographically authenticated data index structure [78] for immutable data management, which provides efficient lookup and update operations on immutable data. Specifically, MPT stores and manages KV bindings, and it also maintains a cryptographic hash value for each node in it, which is computed recursively after the insertion, to ensure tamper evidence or verifiability. In recent years, MPT has been widely used in many immutable data systems, e.g., *blockchain systems* [5, 7, 14, 15, 17], and *verifiable databases* [1, 80, 84, 87]. The role of MPT for these immutable data systems is an analogue of B-tree for traditional database systems. For example, many blockchain systems (e.g., Ethereum [5], Near [14], Zilliqa [17], and Quorum [15]) utilize MPT to index immutable data, provide multi-version accesses and tamper-evident controls. Moreover, Alibaba offers LedgerDB [80] as a service on its cloud, in which MPT is employed to verify the integrity of data, history, and query results [1, 84].

Both blockchain systems and verifiable databases, e.g., Conflux [53] and LedgerDB [80], are targeted for high throughput transaction processing. The efficiency of the underlying data index structure MPT of these immutable data systems plays a key role in achieving that. In the literature, many recent studies [27, 83] have been proposed to analyze, evaluate, and improve the performance of the lookup and insert operators on MPT. At the same time, many techniques [39–41, 59, 62, 65, 71] have been devised in recent years to accelerate the performance of these immutable data systems by exploiting the massive parallelism of GPU. In this work, we focus on accelerating Merkle Patricia Trie (MPT) with GPU.

However, it is hard to achieve. The key reasons are two-fold. First, as mentioned above, exploiting the massive parallelism of GPU to accelerate the subroutines of these immutable data systems is a common practice. For example, there is a GPU-based cryptographic hash value computation library for Merkle Tree [6, 71], which allows efficient and secure data content verification, and Baldur [40] accelerates the signature verification with GPU [40]. However, none of these techniques can be directly adapted to accelerate MPT with GPU. We will shortly elaborate on the reasons via the technical challenges of our studied problem. Second, it is known that it is not easy to devise a high throughput concurrent index structure on modern hardware (e.g., multi-cores, GPU) [51]. In particular, the traditional fine-grained lock coupling approaches have poor scalability, as the cost of synchronizations via locks dramatically increases with the rising of concurrent data updates; and lock-free approaches scale well but are extremely difficult to implement and require many extra indirections (e.g., Bw-Tree [52]). To the best of our knowledge, none of existing work provides a high throughput concurrent MPT for these immutable data systems.

To accelerate MPT with GPU, we first identify the technical challenges that hinder efficient GPU acceleration. These include **node splitting conflict** and **hash computing conflict** during

† Dr. Bo Tang is the corresponding author.
* indicates equal contribution with alphabet order.

Proceedings of the VLDB Endowment, Vol. 17, No. 8 ISSN 2150-8097.
doi:10.14778/3659437.3659443

concurrent KV pair insertion. To address the node splitting conflict, we propose two GPU-based methods, i.e., PhaseNU and LockNU. In particular, PhaseNU resolves the node splitting conflict by proactively expanding nodes before new KV pairs are inserted. It is a lock-free algorithm as it only requires atomic instructions (e.g., `atomicCAS`) in CUDA to guarantee each node in MPT will be and only be expanded and compressed once. LockNU resolves the node splitting conflict by devising a GPU-based optimistic lock coupling solution. It synchronizes new KV pair insertions from many concurrent threads by locking, but only sparingly. Moreover, we provide a decision model for users to choose the proper one between PhaseNU and LockNU by taking the data distribution and density of the individual workload into consideration. To address the hash computing conflict, we propose PhaseHC, a GPU-based algorithm that supports parallel hash computing in MPT. In particular, PhaseHC consists of two phases: (i) marking phase and (ii) computing phase. In marking phase, it determines the dependency of each node in parallel. In computing phase, it computes the hash value in parallel via the dependency relationship. We introduce a warp-based execution model that improves the performance of PhaseHC as it significantly reduces the thread divergence on GPU.

Interestingly, the core algorithmic ideas of our proposed GPU-based methods to accelerate MPT are generic, which can be utilized by multi-core CPU to improve the performance of MPT. The reason is that the success of accelerating MPT via GPU is achieved by exploiting the single instruction multiple threads (SIMT) execution model of GPU effectively. Nevertheless, it is also supported by multi-core CPU. We will elaborate it shortly in Section 3.3.

In summary, the technical contributions of this work are:

- To the best of our knowledge, this is the first work to accelerate the cryptographically authenticated data index structure MPT with GPU, which can be used to improve the performance of many applications, e.g., blockchain systems, verifiable databases, and collaborative data analysis.
- We propose two GPU-based algorithms (PhaseNU and LockNU) to resolve node splitting conflict and devise a GPU-based hash-compute algorithm (PhaseHC) to avoid hash computing conflict during parallel KV pair insertion in MPT.
- We integrate the GPU accelerated MPT into two real-world systems: a blockchain system Geth and a verifiable database LedgerDB, to verify the effectiveness of our proposed techniques.
- We conduct extensive experiments on two real-world datasets (Wiki and ETHT) and one widely-used synthetic dataset (YCSB), to demonstrate the superiority of our proposed solutions. For instance, our experiments show that our two proposed solutions achieve 19.79X and 29.69X speedup for the insert operator over the deployed MPT in Geth.

2 PRELIMINARIES AND CHALLENGES

2.1 Merkle Patricia Trie

MPT is a variant of traditional radix tries, which is a cryptographically authenticated index structure. It is widely used to store KV pairs in blockchains and verifiable databases [5, 14, 17, 80, 83, 84]. Figure 1 depicts the structure of MPT. For each node in MPT, its key is encoded by hex encoding and appended with "0x10" at the

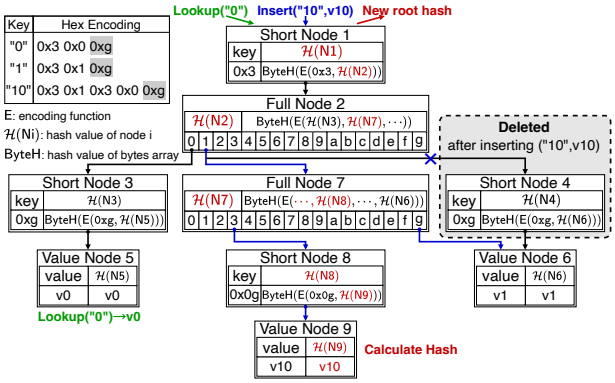


Figure 1: Merkle Patricia Trie

end. For ease of presentation, we use "0xg" to replace "0x10" in the key in this work. MPT consists of three types of nodes: Full Node, Short Node, and Value Node. In particular, a Full Node includes an array of 17 elements, as shown in Figure 1. Each element in Full Node represents a branch that points to a child node, and the 17th (i.e., '0xg'th) element points to a leaf Value Node. A Short Node includes a pointer to the next node and its key field stores a compressed path, as we will elaborate on shortly. A Value Node stores the value of the given key, i.e., Value Node 5 stores the value v0 of key "0", whose hex encoding is "0x3 0x0 0xg". In this work, we interchangeably use "0x3 0x0 0xg" and "0x30g", which only reserves the first "0x" in its hex encoding.

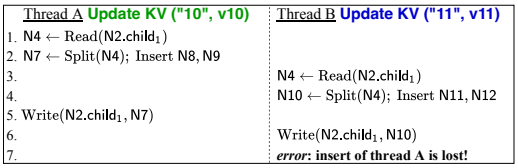
The key difference between MPT and traditional Radix Tries is that the cryptographic hash value of each node in MPT can be used as tamper evidence. The hash value of the root node represents the hash of an MPT. Specifically, the cryptographic hashes of different node types are calculated by different methods. For a Value Node with value V, its hash value is $\mathcal{H}(\text{Value Node}) = V$. The hash value of a Short Node encodes both its key and the hash value of its child node, i.e., $\mathcal{H}(\text{Short Node}) = \text{ByteH}(\text{E}(\text{key}, \mathcal{H}(\text{child})))$, The encoding function E encodes multiple byte arrays together into a new byte array. For example, Ethereum employs the Recursive Length Prefix (RLP) encoding [16]. The byte hash function ByteH computes the hash value of the input bytes array Arr as follows.

$$\text{ByteH}(\text{Arr}) = \begin{cases} \text{Arr}, & \text{if length}(\text{Arr}) < 32; \\ h(\text{Arr}), & \text{otherwise}, \end{cases}$$

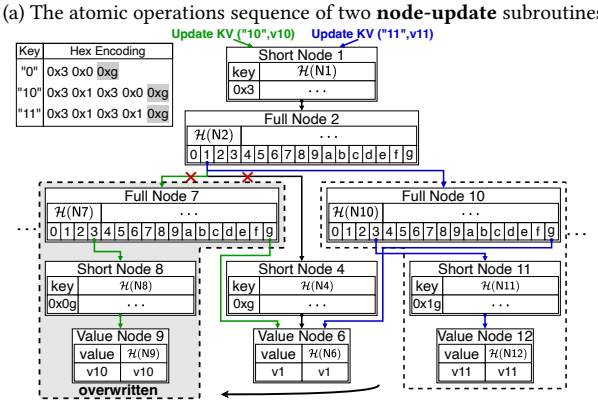
where h maps a bytes array to a fixed-length bytes array, e.g., Ethereum uses *keccak*256 [23] as the hash function h. The hash value of a Full Node encodes the hash value of all its child nodes, i.e., $\mathcal{H}(\text{Full Node}) = \text{ByteH}(\text{E}(\mathcal{H}(\text{child}_0), \cdots, \mathcal{H}(\text{child}_g)))$.

Next, we introduce two fundamental operators in MPT.

(I) Lookup Operator. Given a key, the lookup operator returns its value in MPT. Suppose we want to lookup the key "0" from MPT in Figure 1. First, the key "0" is encoded into "0x3 0x0 0xg", then it starts to lookup from the root. The key of root Short Node 1 is "0x3", which matches the prefix of the search key, and then it moves to the root's child Full Node 2. In the search key, the part after matched "0x3" is "0x0", thus it checks the '0x0'th element at Full Node 2, and moves to its pointed Short Node 3. The last part of the search key is "0xg", it matches the key in Short Node 3, thus, it

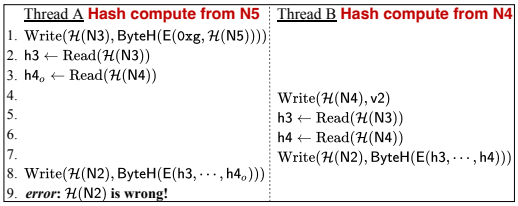


(a) The atomic operations sequence of two **node-update** subroutines

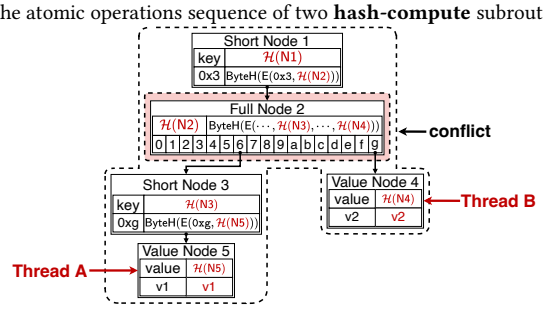


(b) Node splitting conflict in MPT

Figure 2: An example of C1: node splitting conflict



(a) The atomic operations sequence of two **hash-compute** subroutines



(b) Hash value computing conflict in MPT

Figure 3: An example of C2: hash computing conflict

goes to Short Node 3's child node, i.e., Value Node 5. Last, it returns the value v0 as the result of lookup operator with search key "0".

(II) Insert Operator. The insert operator consists of two subroutines: (i) **node-update** and (ii) **hash-compute**. Specifically, the **node-update** subroutine inserts a new KV pair into MPT, while the **hash-compute** subroutine computes the hash value of each affected node during the new KV pair insertion. Next, we illustrate the insert operator by inserting KV pair ("10", v10) into MPT in Figure 1. First, the key "10" is encoded into "0x3 0x1 0x3 0x0 0xg". The **node-update** subroutine follows a similar procedure of lookup operator to find the proper position for new node insertion. In particular, it starts from the root Short Node 1 and moves to Full Node 2. Then access the '0x1'st element of Full Node 2, which points to Short Node 4. Since the key of Short Node 4 is "0xg", which is not "0x3" of the insertion key, a new Full Node 7 is created to replace Short Node 4, and the last element in Full Node 7 points to the child of Short Node 4. Next, the '0x3'rd element in Full Node 7 points to a newly created Short Node 8, which stores the compressed path of "0x0 0xg" of the insertion key, as "0x0g" shown in it. Last, Short Node 8 points to a newly created Value Node 9, which stores the value v10 of the insertion key "10".

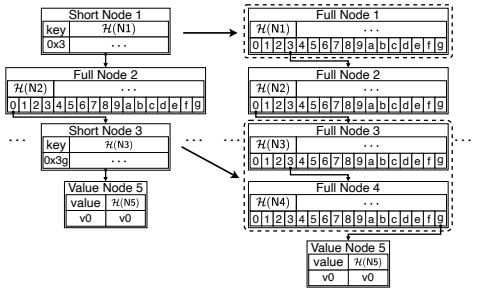
The hash value of every node in the path from the newly added Value Node 9 to root Short Node 1 will be affected (see red colored items in Figure 1). The **hash-compute** subroutine starts from the newly inserted Value Node 9 and updates the hash values of all affected nodes in that path. After that, the insertion of the KV pair ("10", v10) completes.

2.2 Technical Challenges with GPU

It is a common practice to exploit GPU for various applications [18, 32, 40, 42, 66, 71, 74]. In this work, we plan to exploit GPU to accelerate the performance of lookup and insert operators in MPT. For lookup operator, it is straightforward to utilize GPU as it only needs to distribute the search keys into parallel-running threads of GPU. However, it is not trivial for the insert operators, the technical challenges are twofold:

C1: Node Splitting Conflict. Inserting a new KV pair into MPT will affect many nodes. As the above **node-update** example in Figure 1 shows, Short Node 4 is deleted and three new nodes are created (i.e., Full Node 7, Short Node 8, and Value Node 9) during KV pair ("10", v10) insertion. We refer to this as *node splitting*, that is, split Short Node 4 into three new nodes. Parallel insertion of KV pairs into MPT on GPU obviously leads to severe node splitting conflicts. Taking Figure 2 as an example, suppose we insert KV pairs ("10", v10) and ("11", v11) into MPT concurrently. Figure 2(a) illustrates the sequence of atomic operations in two threads A and B. Accordingly, thread A will first split Short Node 4 into Full Node 7 and insert Short Node 8 and Value Node 9 below it. Then thread B will split Short Node 4 into Full Node 10 and insert Short Node 11 and Value Node 12 below it. The insertion of thread A is overwritten by thread B as two parallel insertion threads split Short Node 4 in MPT without conflict control. Therefore, the first technical challenge in this work is *how to avoid node splitting conflict during parallel insertions in* MPT *with GPU?* We are aware that some previous work studied GPU Patricia Trie (or Radix Trie) [19, 45]. However, they only focus on lookup operator and do not provide solutions to handle insert operator on GPU.

C2: Hash Computing Conflict. After **node-update** subroutine for every KV pair inserting, the hash value of every affected node in MPT will be recomputed. A straightforward solution to exploit GPU parallelism for **hash-compute** subroutine is assigning threads to parallel compute the hash value of each node in every path from the newly inserted leaf node to the root node. Figure 3 shows an example. Thread A computes the hash values of the nodes in the path from Value Node 5 to the root Short Node 1, and thread B computes the hash values of the nodes on the path from Value Node 4 to the root Short Node 1. The sequence of the atomic operations in two **hash-compute** subroutines are shown in Figure 3(a). While computing the hash value of Full Node 2, thread A uses the outdated hash value of child g (i.e., $h4_o$) instead of the updated hash



(a) MPT (b) Expanded MPT

Figure 4: Comparison of MPT and expanded MPT

value of child g (i.e., h4) computed by Line 6 in thread B. Thus, the hash value of Full Node 2 is not correct after Line 8 as there is a hash computing conflict when thread A and thread B run in parallel. Moreover, this straightforward solution will compute the hash value of a specific node more than once, i.e., Full Node 2. It incurs significant computing overhead as a single hash value computing is very expensive. Therefore, the second challenge in this work is *how to avoid hash value computing conflict for parallel* ***hash-compute*** *in* MPT *with GPU?* Geth provides a solution for the parallel hash computing in MPT. Specifically, a *Goroutine* is created for each branch and synchronized by a *WaitGroup*. However, it cannot apply to GPU as it does not support dynamic thread creation and synchronizing.

3 OUR PROPOSED SOLUTION

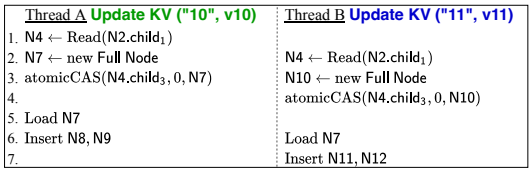
In this section, we present our solution to support parallel KV pair insertions on MPT with GPU, which guarantees sequential consistency [46]. The core idea to achieve parallel insert on MPT with GPU is providing efficient and accurate parallel **node-update** (in Section 3.1) and parallel **hash-compute** algorithms (in Section 3.2).

3.1 GPU-based Node Update Algorithms

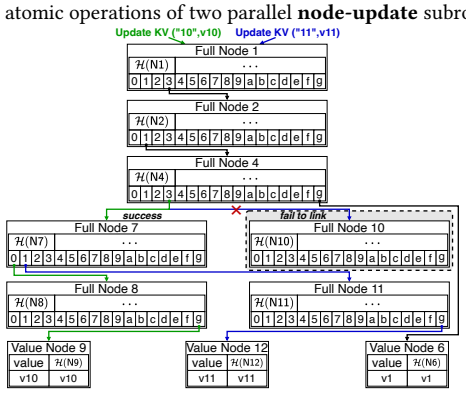
In this section, we present our GPU-based lock-free (PhaseNU) and lock-based (LockNU) algorithms for **node-update** subroutine in insert operator, and the decision model for users to choose the proper **node-update** algorithm for their workloads.

3.1.1 Lock-free ***node-update*** *algorithm* PhaseNU. To devise a lock-free node update algorithm on MPT, we first analyze the properties of different node types in MPT. MPT is a variant of traditional radix tries, which also holds that the path from the root node to a leaf Value Node represents the key of the leaf. In particular, the alphabet of MPT node key is a sized-17 set, i.e., $\{0x0, 0x1, ..., 0xe, 0xf, 0xg\}$. For each Full Node in MPT, the position of the branch in the 17-element array of it represents a character. For each Short Node in MPT, its key field is a substring of the key, which is a compressed path as it can be expanded to one or more Full Nodes and each expanded Full Node has and only has one child. For example, Short Nodes 1 and 3 in Figure 4(a) can be expanded to the Full Nodes, see dotted boxes in Figure 4(b). This compression idea was proposed at [63] to improve the space efficiency of trie structures.

Observation. Node splitting conflict is caused by multiple parallel **node-update** subroutines that want to split the same Short Node simultaneously. Taking Figure 2 as an example, the **node-update** subroutines of inserting ("10", v10) and ("11", v11) both split Short Node 4, which cause a node splitting conflict.



(a) The atomic operations of two parallel **node-update** subroutines



(b) Parallel **node-update** on uncompressed MPT

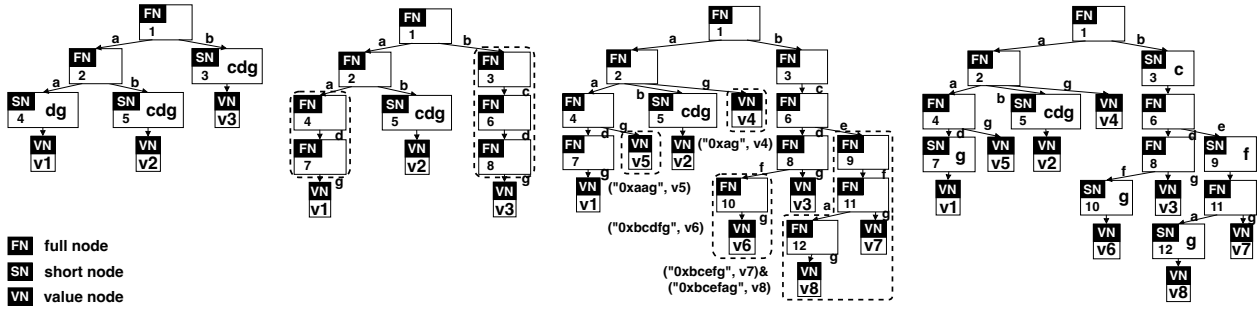
Figure 5: An example of parallel node-update

Combining the above observation and the property of Short Node in MPT, we propose a lock-free node update algorithm PhaseNU, which avoids node splitting conflict by expanding Short Nodes in MPT to the corresponding Full Node sequences.

Example: Figure 5 shows the uncompressed version of MPT in Figure 2. We insert KV pairs ("10", v10) and ("11", v11) into the uncompressed MPT in Figure 5. The atomic operations of two **node-update** threads A and B are shown in Figure 5(a). First, Full Node 7 and 10 are initialized in threads A and B, respectively. Then, the atomic instruction compare-and-swap (`atomicCAS`) determines which one to update MPT, see Lines 3 and 4. It ensures that only one Full Node, see Full Node 7 in Figure 5(b), will be inserted as the '0x3'rd child of Full Node 4. After that, threads A and B load Full Node 7, and insert Full Node 8 and Full Node 11 as the '0x0'th and '0x1'st child, respectively. Last, the '0xg'th element of Full Node 8 and Full Node 11 will point to their corresponding Value Node 9 and Value Node 12. Until now, the parallel **node-update** subroutines finished on uncompressed MPT and the node splitting conflict has been resolved.

We then formally present our devised lock-free **node-update** algorithm PhaseNU, which supports efficient and parallel **node-update** on MPT with GPU. As shown in Figure 6, PhaseNU consists of three phases: (i) expanding phase, it converts MPT to uncompressed version by expanding necessary Short Nodes to corresponding Full Node sequences, see Figure 6(b); (ii) inserting phase, it inserts new KVs pairs to uncompressed MPT, see Figure 6(c); and (iii) compressing phase, it transforms updated uncompressed MPT to general MPT, see Figure 6(d). Unfortunately, it is not trivial to unlock potential power of GPU in each phase. In subsequent, we elaborate on technical challenges and our solutions one by one.

Phase I: Expanding Phase. For a batch of KV pair insertions, a straightforward solution in expanding phase is expanding all Short Nodes in MPT. Obviously, it is inefficient as some of them may not be influenced during this insertion batch, see Short Node



(a) Initial MPT (b) Expanding Phase (c) Inserting Phase (d) Compressing Phase

Figure 6: PhaseNU**: a lock-free based parallel node update algorithm**

5 in Figure 6(a), and expanding them incurs extra overhead. Hence, the main challenge of expanding phase in PhaseNU is how to locate and expand these Short Nodes, which will be influenced, in MPT efficiently. To locate all these Short Nodes quickly, we first devise a node lookup operator (i.e., nodelookup). For a given key, nodelookup returns the first mismatch node in MPT. Taking MPT in Figure 6(a) as an example, if the input key is "0xaag", the result of nodelookup is Short Node 4, as the prefix "0xaa" of given key "0xaag" has been matched at Full Node 1 and Full Node 2, but the character "g" in the key mismatched at Short Node 4.

As discussed in Section 2.2, parallel lookup on MPT can be easily achieved. Hence, we use a similar way to parallel run nodelookup operators for every key in the insertion batch. The result node type of nodelookup operator is either Short Node or Full Node. We directly skip Full Nodes and expand returning Short Nodes to sequences of Full Nodes. To achieve parallel Short Node expansion, We employ atomic compare-and-swap (`atomicCAS`) before expanding them in each thread, which avoids concurrent modification and ensures each Short Node only be split once.

Example: Considering MPT in Figure 6(a), suppose the insert KV pairs are ("0xag", v4), ("0xaag", v5), ("0xbcdfg", v6), ("0xbcefg", v7) and ("0xbcefag", v8)[3]. In expanding phase, it first runs nodelookup for five keys in parallel, and returns Full Node 2, Short Node 3, and Short Node 4. Then we parallel expand Short Node 3 and Short Node 4. Figure 6(b) shows the result after expanding phase.

Phase II: Inserting Phase. The inserting phase of PhaseNU inserts all new KV pairs into the above returned uncompressed MPT. There are only two inserting cases: (i) inserting a Value Node, e.g., for KV pair ("0xaag", v5), it will create a Value Node v5 and insert it as the '0xg'th child of Full Node 4, as shown in Figure 6(c); (ii) inserting a sequence of Full Nodes and a Value Node, e.g., for key value pair ("0xbcdfg", v6), it will create Full Node 10 as the '0xf'th child of Full Node 8, then insert a Value Node v6 as the '0xg'th child of Full Node 10, see Figure 6(c).

The first challenge in the inserting phase is to address the conflict of inserting new nodes. In particular, when the keys of parallel inserting threads share the same prefix, e.g., ("0xbcefg", v7) and ("0xbcefag", v8) share the same prefix "0xbcef", there will incur a conflict when two threads are creating Full Node 9 concurrently. We resolve it by using atomic compare-and-switch (`atomicCAS`), which guarantees each Full Node will only be linked once.

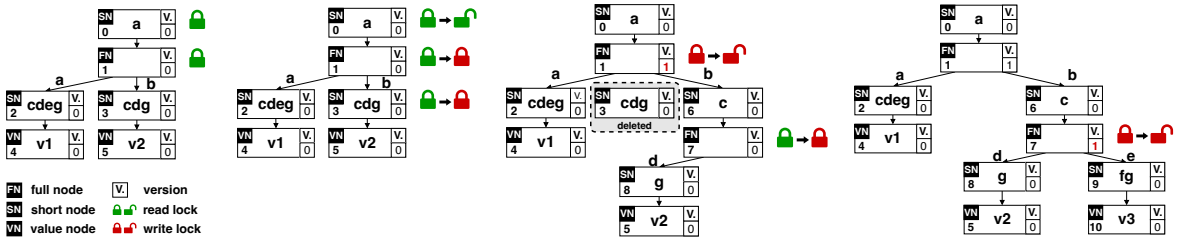
[3]For the sake of presentation, we use hex encoding of those keys directly.

The second challenge in the inserting phase is to efficiently allocate the memory of Full Nodes. The Full Nodes are frequently created and allocated in the global memory of GPU during the inserting phase. Following previous work [22], the memory of each Full Node is allocated from a global memory pool. Each `malloc` operation in it updates its metadata, e.g., the allocated flags. Conflicts of concurrent metadata updating are handled by atomic operations. A straightforward strategy is to call `malloc` when creating each Full Node, we refer to it as *device-based allocator*. Obviously, it incurs severe contention on memory pool since it is shared by all threads in kernel. An alternative strategy is *thread-based allocator*, i.e., a local buffer is allocated for each thread at first, and all Full Nodes are allocated from the local buffer. It minimizes contention in memory pool. However, it is also inefficient as its uncoalesced global memory access results in low memory throughput. In particular, the Full Nodes that are accessed simultaneously by adjacent threads are physically far apart in GPU memory. To improve it, we devise *block-based allocator* based on the shared memory in GPU. Specifically, it invokes `malloc` once to allocate a local memory pool for each block, whose metadata resides in the shared memory and is used by all threads within the block. The benefits of our *block-based allocator* are three-fold: (i) the memory pool contention is limited within a block; (ii) the atomic operations on GPU to access the shared memory are more efficient than those to access the global memory; and (iii) it reduces uncoalesced memory access.

Example: Figure 6(c) shows the result after inserting all KV pairs in the uncompressed MPT in Figure 6(b).

Phase III: Compressing Phase. With the returning uncompressed MPT after the above inserting phase in PhaseNU, the compressing phase converts it to the general MPT, which guarantees memory efficiency. However, the challenges of providing an efficient algorithm for compressing phase are two-fold: (i) it accesses MPT nodes by the path from leaf to root, then how to determine the entry nodes of compressing threads efficiently? and (ii) how to parallel compress the corresponding Full Nodes in uncompressed MPT efficiently?

We first identify the entry nodes of all compressing threads to address the first challenge. On one hand, the correctness of MPT cannot be guaranteed if we miss some of the entry nodes. On the other hand, it incurs expensive compressing overhead and wastes the computing power of the GPU if all leaf nodes are considered entry nodes. Specifically, the entry nodes in uncompressed MPT can be generated in both expanding phase and inserting phase. For



(a) Acquiring optimistic read lock (b) Upgrading to pessimistic write lock (c) Splitting node with mutual exclusion (d) Inserting new nodes with write lock

Figure 7: LockNU**: an example to illustrate GPU-based optimistic lock coupling algorithm, inserting ("0xabcef", v3)**

example, Value Node v1 in Figure 6(c) is an entry node as its parent Short Node 4 was expanded at the expanding phase. Meanwhile, Value Node 7 and Value Node 8 in Figure 6(c) are entry nodes as they are newly inserted in the inserting phase. In PhaseNU, we employ a hash container to on-the-fly record all these generated entries nodes during expanding phase and inserting phase, instead of identifying them at the beginning of the compressing phase. Taking Figure 6 as an example, the hash container includes all entry nodes in compressing phase, i.e., Value Nodes v1, v3, v4, v5, v6, v7, and v8, in which Value Nodes v1 and v3 are obtained from expanding phase, and the rest are from the inserting phase.

We next compress the corresponding Full Nodes to address the second challenge. Specifically, in each compressing thread, a compressing Short Node is created at first, then Full Nodes with one child on the path are continuously marked "compressed", and these marked Full Nodes will be replaced by the compressing Short Node when a Full Node with multiple children is encountered on the path. Taking Figure 6(c) as an example, for the compressing thread with entry Value Node v6, a compressing Short Node is created and Full Node 10 is marked as "compressed" as it only has one child. Then, the compressing Short Node will replace Full Node 10 as the parent of Full Node 10 has multiple children. To parallel run these compressing threads, we employ `atomicCAS` to guarantee all nodes in MPT are only marked and compressed once.

Example: For all entry nodes in the hash container, see Figure 6(c), it runs above compressing thread in parallel, and the compressed MPT is shown in Figure 6(d). Up till this point, all the KV pairs have been inserted into the MPT by our lock-free **node-update** algorithm PhaseNU.

*3.1.2 Lock-based **node-update** algorithm* LockNU. Our above lock-free **node-update** algorithm PhaseNU enjoys excellent performance for node updating in MPT with GPU. However, it incurs overhead to expand and compress MPT. In this section, we propose a lock-based **node-update** algorithm LockNU as an alternative solution. Specifically, we devise a GPU-based optimistic lock coupling algorithm LockNU for parallel node updating in MPT.

The Overview of LockNU**.** Inspired by optimistic lock coupling [49, 51] for concurrent data structures in multi-core CPU, we devise a GPU-based optimistic lock coupling algorithm for parallel **node-update** in MPT with GPU. Returning to the **node-update** subroutine in the insert operator, given a KV pair, it first finds the destination node and then inserts the new nodes accordingly. To resolve the node splitting conflict during parallel **node-update** subroutines on GPU, each node in MPT is augmented with a version counter, which is the building brick of our GPU-based optimistic lock coupling algorithm LockNU. In particular, the core idea of LockNU is two kinds of locks: (i) optimistic read lock and (ii) pessimistic write lock. Each reader or writer in **node-update** subroutine acquires the corresponding lock of the node.

Optimistic read lock: It is a conceptual lock, i.e., it does not actually acquire or release a lock. In fact, it only reads the version counter of a node. For each reader in the GPU thread of **node-update** subroutine, it acquires an optimistic read lock of a reading node by `node.ReadLockOrRestart()` at first. If the lock is not free, i.e., a writer is updating the node, it fails and restarts from the root. Otherwise, it returns the version counter of the node. The optimistic read lock is released by `node.ReadUnlockOrRestart(version)`, which compares the previous version counter, i.e., returned by `node.ReadLockOrRestart()`, with the current version counter. If the previous version counter is obsolete, the thread of **node-update** subroutine fails and restarts from the root node. Besides that, the optimistic read lock of a node may be upgraded to a pessimistic write lock when the node should be updated by itself or the changes of its child nodes in MPT. We will elaborate on it shortly.

Pessimistic write lock: It provides mutual exclusion by physically acquiring the lock. For each writer in the GPU thread of **node-update** subroutine, we guarantee a pessimistic write lock can only be upgraded from an optimistic read lock. The node lock upgrading is achieved by `node.UpgradeToWriteLockOrRestart(version)`. The reason is that, for **node-update** in MPT, it first acquires the optimistic read lock of the node, then reads the node, and checks whether the key of the node matches with the inserting key or not. If it returns false, the node will be modified (e.g., splitting to a Full Node or inserting a new Value Node). Thus, the optimistic read lock will be upgraded to a pessimistic write lock. Specifically, the upgrading function first verifies whether the node is changed or not after it acquires an optimistic read lock by its version counter. If it is changed, the thread of **node-update** subroutine fails and restarts from the root node. Otherwise, it acquires the pessimistic write lock of the node and modifies the node. After that, the node releases its pessimistic write lock by `node.WriteUnlock()` and its version counter is incremented.

Example: Next, we utilize Figure 7 to illustrate how to insert a new KV pair ("0xabcefg", v3) in MPT via GPU-based optimistic lock coupling algorithm LockNU.

Step 1: the thread acquires the optimistic read lock of Short Node 0. Since its key "0xa" matches with the inserting key, the thread then acquires the optimistic read lock of its child, i.e., Full Node 1.

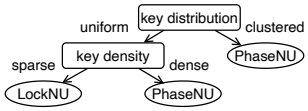


Figure 8: Decision model for node-update algorithms

As shown in Figure 7(a), both nodes have optimistic read locks. The thread visits the '0xb'th element of Full Node 1, which points to Short Node 3. The optimistic read lock of Short Node 0 is released as it will not be updated as its child Full Node 1 will not change.

Step 2: the thread acquires the optimistic read lock of Short Node 3. Its key "0xcdg" differs from the inserting key "0xcefg", thus it will be split. As a result, its parent Full Node 1 will be updated accordingly. Thus, the optimistic read locks of Short Node 3 and Full Node 1 are upgraded to pessimistic write locks, see Figure 7(b). The pessimistic write locks guarantee mutual exclusion, which resolves conflicts from concurrent **node-update** threads.

Step 3. As illustrated in Figure 7(c), Short Node 3 is split into Short Node 6, Full Node 7, and Short Node 8, and the '0xb'th element of Full Node 1 points to the newly created Short Node 6. The '0xd'th element of Full Node 7 points to Short Node 8, which points to Value Node 5. After that, the changing of Full Node 1 completes, and the thread releases the write lock and increments its version counter, see red colored 1 in it.

Step 4: the thread acquires the optimistic read lock of Full Node 7 and upgrades to a pessimistic write lock for inserting the new key "0xabcefg". Accordingly, the '0xe'th element of Full Node 7 points to the newly created Short Node 9, which points to Value Node 10. The thread releases the pessimistic write lock of Full Node 7 and increments its version counter, see Figure 7(d).

The Implementation of LockNU**.** To the best of our knowledge, this is the first work that implements a GPU-based optimistic lock coupling algorithm for parallel **node-update** subroutine in MPT. Specifically, the lock is implemented as a 64-bit number stored in every node. The least significant bit indicates the pessimistic write lock, and other bits form a version counter. In particular, the atomic load and store instructions are not provided by CUDA [4]. We implement them via `__threadfence` and `volatile`.

*3.1.3 Decision model for **node-update** algorithm choosing.* Until now, we have two algorithms for parallel **node-update** in MPT with GPU: PhaseNU and LockNU. They utilize different methods to resolve the conflict of parallel **node-update**. In particular, PhaseNU resolves conflicts by proactively splitting the nodes, and LockNU addresses conflicts by lock coupling. Thus, the overhead of PhaseNU is introduced by splitting and compressing the nodes, but the overhead of LockNU is introduced by the locks. It is not trivial for users to choose the proper one for their workloads. In subsequent, we propose a simple yet effective decision model via our empirical experience. We first analyze the key factors which will influence the overhead of PhaseNU and LockNU as follows:

- **Key distribution.** it captures the probability of occurrence of difference keys, i.e., uniform distributed, or clustered distributed.
- **Key density.** it reveals the number of keys in a given range of key values, i.e., if the bitmap of all values in the given range contains mostly zeroes, it is sparse. Otherwise, it is dense.

Figure 8 depicts the decision model to choose proper **node-update** algorithm for different cases. There are three key observations about the decision model.

Observation I: if keys are clustered distributed, PhaseNU is better than LockNU, as the rightmost path in Figure 8 shown.

LockNU is a GPU-based optimistic lock coupling solution. In particular, when modifying a branch of Full Node via LockNU, it acquires a pessimistic write lock and restarts all the other threads that are accessing this Full Node. Thus, the performance of LockNU will obviously degenerate when the pessimistic write lock is frequently acquired to modify the Full Node. When key distribution is clustered, e.g., Gaussian distribution, Full Nodes in MPT will probably be locked with pessimistic write lock many times as the hex encoding of these clustered keys shares similar prefixes. Thus, the overhead of LockNU is significantly larger than that of PhaseNU.

Observation II: if keys are uniformly and sparsely distributed, LockNU is better than PhaseNU, see the leftmost path in Figure 8.

The reasons why LockNU outperforms PhaseNU when the keys are uniformly and sparse distributed are two-fold: (i) the node splitting conflicts in LockNU are significantly reduced as the hex encodings of keys are quite different; and (ii) for PhaseNU, proactively expanding MPT generates a large number of intermediate Full Node nodes, which incurs lots of unnecessary overhead.

Observation III: if keys are uniformly and densely distributed, PhaseNU is better than LockNU, as shown in Figure 8.

The reasons why PhaseNU is better in this case are similar to the reasons of Observation I. Combining Observation II and Observation III, the superiority of PhaseNU becomes obvious with the rising of key density.

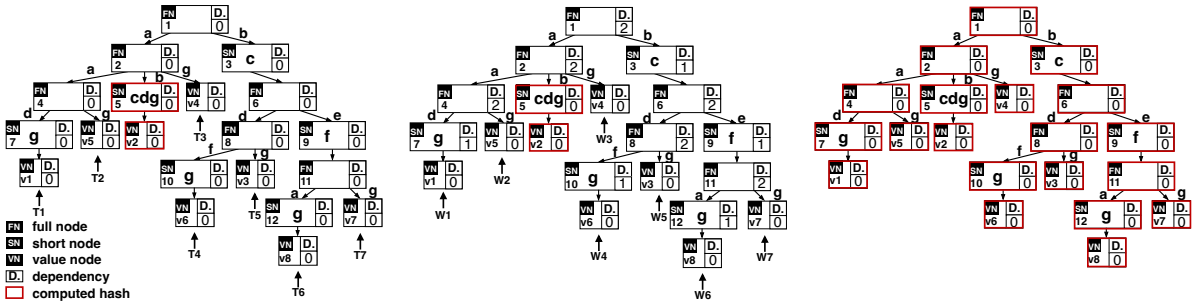
3.2 GPU-based Hash-compute Algorithm

In this section, we present our GPU-based **hash-compute** algorithm PhaseHC to avoid the hash computing conflict for parallel insert operators of MPT on GPU.

We first analyze the underlying reason for hash computing conflict. According to the hash computing equations in Section 2.1, the hash value of each Short Node or Full Node is computed from itself and the hash value(s) of its child(ren). Thus, the correct hash value of a node depends on the updated hash value of every child after inserting new KV pairs. The illustrated example of hash computing conflict in Figure 3 (see Section 2.2) also confirms that, i.e., the wrong hash value of Full Node 2 is caused by the out-of-date hash value of Value Node 4.

To resolve the hash computing conflict, we devise a GPU-based **hash-compute** algorithm PhaseHC. It consists of two phases: (i) marking phase, it parallel marks the dependency of each influenced node during the previous node updating part; (ii) computing phase, it parallelly computes the hash value of these nodes efficiently. Next, we present the technical details of these two phases in PhaseHC.

Phase I: Marking Phase. We augment MPT by attaching a dependency counter (initialized to 0) for every node, which indicates the number of dependent child nodes. As per our previous analysis, the hash value of the newly inserted Value Node will affect all the nodes in the path from it to the root node. To count the number of dependencies of each influenced node, a straightforward



(a) MPT after inserting (b) Marking phase (c) Computing phase

Figure 9: An example of GPU-based Hash-compute algorithm PhaseHC

solution is traversing MPT in a depth-first manner. It is obviously inefficient as it is sequential. To utilize the massive parallelism of GPU, we propose a parallel marking procedure. Specifically, we assign a thread for each entry node in the hash container, which is obtained from **node-update** subroutine, as threads T1 to T7 shown in Figure 9(a). Each thread is responsible for updating the dependency of every node in the path from its entry node to the root node. For each node, it increments the dependency counter of its parent node as the hash value of its parent depends on it. For example, the dependency counter of Short Node 7 will be 1 after thread T1 visits Value Node v1, as shown in Figure 9(b). We use `atomicAdd` in CUDA to guarantee the atomicity of the dependency counter calculated by multiple threads. To guarantee that each node only increments its parent's dependency counter once, we employ an extra atomic flag in every node to indicate whether it has incremented its parent's dependency counter or not. Each thread terminates whenever it reaches the root node. Figure 9(b) depicts the MPT after marking phase, e.g., the dependency counter of Full Node 8 is 2 as its hash value depends on the hash values of Full Node 10 and Value Node v3.

Phase II: Computing Phase. We parallel run hash computing subroutines for every entry node and compute the hash value of every node in the path from the entry node to the root node. For each node, it decrements its parent dependency counter after computing its hash value, which means it is an updated hash value and can be used by its parent. In addition, each hash computing subroutine guarantees that it only computes the node whose dependency counter is 0. Otherwise, it will be terminated. However, it is not trivial to design an efficient computing phase. The reasons are three-fold: (i) the number of memory accesses is large as it incurs many memory accesses to compute the hash value of the node; (ii) thread divergence is obvious, different threads execute different instructions when they are computing hash value of different nodes; and (iii) the cost of the hash function in MPT is naturally expensive.

To reduce (i) the cost of memory accesses, we load the encoding of the node into the shared memory before computing its hash value, and write the hash value to global memory after the computation. We devise a warp-based solution to address (ii) and (iii). Specifically, a warp is a group of threads that can only execute the same instruction at a time on the same GPU core. We assign a warp to each node, instead of a single thread in the marking phase, see W1 to W7 in Figure 9(b). Threads in the same warp collaboratively compute the hash value of the node and synchronize with the `__shfl_sync` primitive. After the computing phase, the hash values of influenced nodes are updated, see the red rectangles in Figure 9(c). Benefits of the warp-based solution are two-fold. First, all the threads in a warp execute the same instruction to compute the hash value of a node, which alleviates the thread divergence problem. Second, the hash computing of a node in MPT can be distributed into a warp of threads with the efficient GPU hash algorithm [26], which improves the performance of node hash value computation.

3.3 Discussion and Extension

The success of accelerating MPT via GPU is achieved by exploiting the single instruction multiple threads (SIMT) execution model on it. Specifically, the execution logic of insert operator is inherently complex, and it is not trivial to provide a high throughput MPT inherently. We divide the complex logic of the insert operator into 5 SIMT-friendly phases, i.e., expanding, inserting, and compressing phases in PhaseNU, and marking and computing phases in PhaseHC, to fully exploit the massive parallelism of GPU. Besides, our proposed techniques (i.e., PhaseNU, LockNU, and PhaseHC) also exploit the following characteristics of GPU: (i) efficient atomic instructions (i.e., `atomicCAS` and `atomicAdd`), (ii) efficient memory allocation, see block-based allocation strategy in Section 3.1, and (iii) warp-based computation optimization in Section 3.2.

We next present the extensibility of our proposed techniques. All these techniques can be extended to improve the performance of MPT on multi-core CPU as the SIMT execution model also is supported on it. To achieve high parallel **node-update** of MPT with multi-core CPU, both LockNU and PhaseNU algorithms can be directly used. For implementation-wise, we use `parallel_for`, which is provided by the Intel Thread Building Block (TBB) [13], to distribute tasks to different threads, and `std::atomic` to control concurrent access of MPT nodes on multi-core CPU. To accelerate **hash-compute** of MPT with multi-core CPU, we slightly revise PhaseHC algorithm as multi-core CPU does not have warp-based execution and shared memory schemes of GPU. In particular, we use CPU-based threads in both marking phase and computing phase of PhaseHC algorithm. In addition, our proposed LockNU algorithm also can be used to provide parallel access to other data structures (e.g., B-Tree [29] and SkipList [70]) on GPU.

4 CASE STUDY ON REAL-WORLD SYSTEMS

In this section, we integrate our proposed solutions into two real-world immutable data systems: Geth [9] and LedgerDB [80].

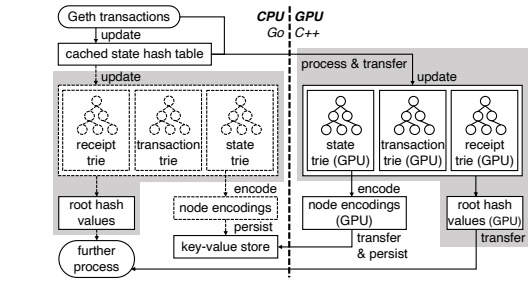
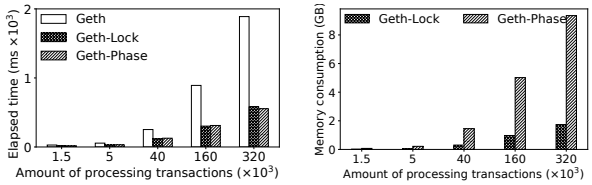


Figure 10: Integrating GPU-accelerated MPT **to** Geth



(a) Performance evaluation (b) GPU Memory consumption

Figure 11: Performance analysis of Geth

4.1 Case Study of Geth

Transaction Processing of Geth**:** We use the left part of Figure 10 to illustrate the transaction processing of Geth. In the beginning, each worker (i.e., miner node) gets a batch of transactions from the transaction pool, which stores all the user-submitted transactions, then executes them. During execution, first, the balance of accounts in cached state hash table will be updated accordingly. To ensure tamper evidence and provide cryptographic authentication, Geth next update three MPTs in the main memory of the worker, i.e., *state trie*, *receipt trie*, and *transaction trie*. Then Geth packs the updated hash value of their roots and the batch of transactions into a block, and broadcasts the block to other workers for further processing. Last, the MPT nodes of *state trie* are asynchronously encoded into RLP encoding [16] and persisted into the KV store.

Integrating GPU-accelerated MPT **to** Geth**:** To integrate GPU-accelerated MPT to Geth, we replace these three in-memory MPTs with our proposed GPU-accelerated MPT, as shown in the right part of Figure 10. For transaction processing procedure on Geth with GPU-accelerated MPTs, we first process input KV pairs (i.e., transactions, receipts, and account states) into continuous arrays and transfer them to GPU global memory. Then, these KV pairs are updated into corresponding MPTs via our proposed algorithms. We employ cross-language library cgo [3] to transfer data and controls between Go on CPU and C++ on GPU. Last, updated hash values of these roots on GPU are transferred to CPU for further processing. Similarly, we asynchronously encode and persist *state trie* on GPU.

Performance Evaluation: We compare the elapsed time of the original Geth and the Geth with GPU-accelerated MPT by running real-world transactions in Ethereum. Specifically, we collect all the regular transactions that start from 15th, November 2022 in blockchain system Ethereum. The number of transactions in each batch ranges from 1.5K to 320K in our experiments. We compare the original Geth with two versions of our GPU-accelerated Geth:

- Geth-Phase: it uses **node-update** algorithm PhaseNU and **hash-compute** algorithm PhaseHC for GPU-accelerated MPT.
- Geth-Lock: it uses **node-update** algorithm LockNU and **hash-compute** algorithm PhaseHC for GPU-accelerated MPT.

We report the time of function `Engine.FinalizeAndAssemble()` in Geth, Geth-Phase, and Geth-Lock, respectively. It measures the elapsed time of gray parts (see Figure 11) of transaction processing, which includes: (i) the cost of processing and transferring the data from CPU to GPU, (ii) the cost of updating three MPTs and (iii) the cost of transferring root hash values from GPU to CPU. We ignore the time cost of encoding and persisting of *state trie* as it works in an asynchronous manner, i.e., its cost will not affect the performance of transaction processing. The results are plotted in Figure 11(a). In summary, the speedup times of the GPU-accelerated Geths (i.e., Geth-Phase and Geth-Lock) over the original Geth range from 1.6X to 3.4X. Even though the improvements in real-world blockchain system Ethereum are good, it can be further improved, see our empirical evaluation results in Section 5, as the data process and transfer cost can be further reduced in GPU-accelerated Geth, e.g., designing both CPU- and GPU-friendly data layout. However, it is out of the research scope of this work and we left it as future work.

GPU Memory Consumption: We measure the peak GPU memory consumption during the processing of a block on the real Ethereum state in 22nd, November 2023. The results are plotted in Figure 11(b). First of all, the memory consumption of PhaseNU and LockNU is less than the GPU memory size even with a large block size of 320K. Secondly, it is not surprising that LockNU outperforms PhaseNU in terms of memory consumption in all tested cases. The reason is that many intermediate Full Nodes are allocated in PhaseNU.

4.2 Case Study of LedgerDB

Transaction Processing and Verification of LedgerDB**:** We use the left part (CPU part) of Figure 12 to illustrate transaction processing and verification in Alibaba LedgerDB. For transaction processing, the client first commits transactions to LedgerDB, which includes querying and updating specific key values. LedgerDB executes the transactions on the *clue index*, which stores the value and the corresponding transaction ids of each key. After that, the executing result (i.e., new "txn id") is returned to the client for further verification. To protect the data integrity, LedgerDB asynchronously collects the processed transactions and updates the authenticated data structures (ADS) in batch. In particular, ADS includes a *batch-Accumulated Merkle Tree* (bAMT) and a *clue-counter MPT* (ccMPT) in LedgerDB. For verification, the client periodically requests the ADS proofs of the committed transactions and keys and then verifies them locally. Only the transactions that persisted in ADSs can be successfully verified.

Integrating GPU-accelerated MPT **to** LedgerDB**:** We replace the ccMPT with our proposed GPU-accelerated MPT, as shown in the right gray part (GPU part) of Figure 12. For transaction processing, the input KV pairs (i.e. info of *clue index*) to ccMPT are updated into the ccMPT in GPU via our proposed algorithms. For verification, we implemented a parallel algorithm to get the proof of each key in ccMPT, which contains all nodes in the search path of the key. The algorithm includes two kernels. The first one searches all keys in parallel, to calculate the size of all proofs and the expected location of each proof in the output buffer. An output buffer is then allocated

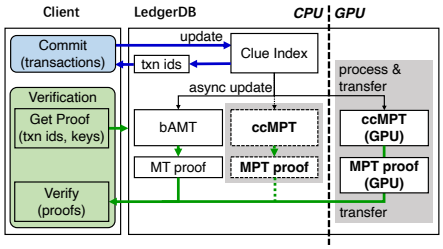
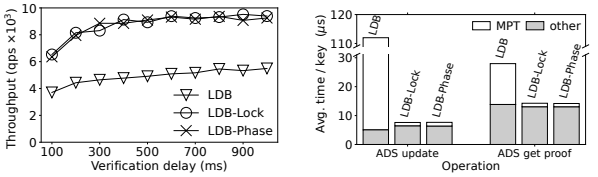


Figure 12: Integrating GPU-accelerated MPT to LedgerDB



(a) End-to-end throughput (b) Operation latency breakdown

Figure 13: Performance analysis of LedgerDB

according to the size. The second kernel searches all keys again and writes the proofs into their pre-calculated positions in parallel. For all operations on GPU-accelerated ccMPT, LedgerDB reorganizes the inputs and outputs, and transfers them among GPU and CPU by following the method in Geth (see Section 4.1).

Performance Evaluation on VeriBench: We use YCSB workload in VeriBench [84] to investigate the performance of LedgerDB with GPU-accelerated MPT. In the experiments, a client keeps committing and verifying transactions to a LedgerDB server. Each transaction consists of operations to 10 KVs with a balanced read-write ratio. A transaction is considered completed only if it is committed and verified successfully. The delay of the periodic verification ranges from 100ms to 1000ms. We compare original LedgerDB (denoted as LDB) with two versions of our GPU-accelerated LedgerDB:

- LDB-Phase: it uses **node-update** algorithm PhaseNU and **hash-compute** algorithm PhaseHC for GPU-accelerated MPT.
- LDB-Lock: it uses **node-update** algorithm LockNU and **hash-compute** algorithm PhaseHC for GPU-accelerated MPT.

We measure the end-to-end throughput by running each experiment for two minutes. As shown in Figure 13(a), The throughput speedup of two GPU-accelerated LedgerDBs (i.e. LDB-Phase, LDB-Lock) over the original LedgerDB ranges 1.47X to 1.78X. Moreover, the gap in the throughput between the original LedgerDB and the GPU-accelerated LedgerDB becomes obvious with the rising of the verification delay. In Figure 13(b), we report the time cost of two key operations in LedgerDB, i.e., ADS *update* and ADS *get proof*. It is clear that the time cost on MPT dominates the total cost of both ADS *update* and ADS *get proof*. Interestingly, our GPU-accelerated MPTs reduces the cost of ADS *update* and ADS *get proof* significantly, which are no longer the performance bottleneck of LedgerDB.

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setting

Datasets. We use the same three datasets as [83] in our experimental evaluation. In particular, Wiki is a real-world dataset. It contains all contents from Wikipedia website. Each KV pair in it represents a particular revision of a page. Each KV pair of ETHT is a transaction in Ethereum, one of the largest blockchains in the world. In particular, the value field is all data in the transaction. We use YCSB to generate data as synthesized KV pairs in this work. Keys in YCSB are uniformly generated. Table 1 summarizes detailed information.

Table 1: Dataset Information

Dataset	Average length		Key distribution	Tested range
	Key (hex)	Value (Byte)		
Wiki	32	10894	clustered	2.5K - 320K
ETHT	65	1073	uniform	5.0K - 640K
YCSB	47	523	uniform	10.0K - 1280K

Experimental Configurations. Our experiments are conducted on a Ubuntu server with two Intel(R) Xeon(R) Gold 6230R CPUs (2.10GHz), 128GB RAM, and an NVIDIA Tesla V100 GPU with 32 GB device memory. In particular, each CPU unit is equipped with 26 cores. Hence, the server could run up to 104 threads. We implemented all our proposed GPU accelerated algorithms in C++ with CUDA 11.7.

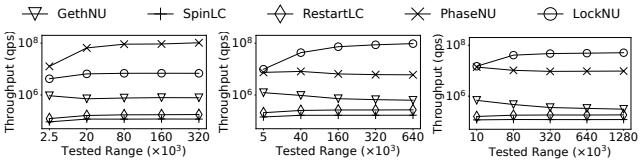
5.2 Performance Evaluation

5.2.1 The evaluation of insert operator in MPT. In this section, the initialized MPT is empty, and we take the KV pairs movement cost from host memory and device memory into account when measuring the end-to-end throughput.

Throughput of Node-update Subroutine. We first evaluate the throughput of **node-update** subroutine in three datasets. We compare three existing or adapted approaches with our proposed lock-free and lock-based **node-update** algorithms on GPU (i.e., PhaseNU and LockNU) by varying the number of inserted key-value pairs. In particular, the first compared method GethNU is the **node-update** approach in real-world blockchain system Geth, which uses a single CPU thread. We also adapt two traditional lock coupling methods (i.e., SpinLC and RestartLC) to GPU as our competitors. Both methods hold at most 2 locks at a time during MPT updating. SpinLC keeps spinning until it acquires the lock while RestartLC restarts from the root when it fails to acquire the lock.

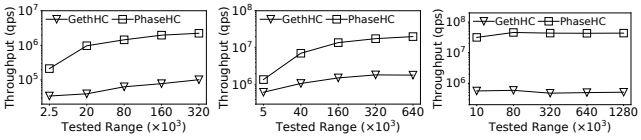
Figure 14 depicts the experimental results. There is no doubt our proposed GPU-based algorithms PhaseNU and LockNU outperform all competitors in every tested case. In particular, the speedup times of PhaseNU and LockNU over GethNU up to 132.28X and 164.68X, respectively. The traditional fine-grained lock-based methods (SpinLC and RestartLC) perform even worse than CPU-based algorithm GethNU, we omit them in subsequent experiments. Interestingly, PhaseNU outperforms LockNU at Wiki, see Figure 14(a). The reason is that the distribution of keys in Wiki is clustered. It confirms the Observation I of the decision model in Section 3.1.3. However, LockNU is better than PhaseNU on ETHT and YCSB, see Figures 14(b) and (c). The reason is that the keys of both ETHT and YCSB are uniformly and sparely distributed, which is Observation II in Section 3.1.3. Specifically, the key of ETHT and YCSB is the hash result of *Keccak*256 [23] and *Fowler_Noll_Vo* [8], respectively.

Throughput of Hash-compute Subroutine. We next evaluate the throughput of **hash-compute** subroutine by varying the number



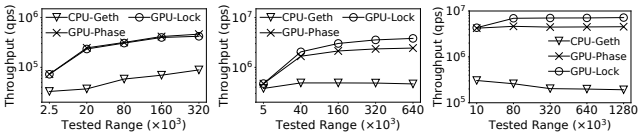
(a) Wiki (b) ETHT (c) YCSB

Figure 14: Throughput of node-update subroutine



(a) Wiki (b) ETHT (c) YCSB

Figure 15: Throughput of hash-compute subroutine



(a) Wiki (b) ETHT (c) YCSB

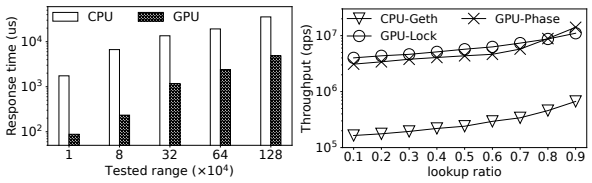
Figure 16: End-to-end throughput of insert operator

of inserted KV pairs. In particular, we compare our proposed GPU-based **hash-compute** method PhaseHC with the **hash-compute** approach in Geth (i.e., GethHC), which is a parallel method via *Goroutine*. It uses all 104 threads of the server in this evaluation.

As reported in Figure 15, our GPU-based algorithm PhaseHC is better than CPU-based algorithm GethHC by up to 25.20X, 10.99X, 91.88X in Wiki, ETHT and YCSB, respectively. Interestingly, the improvement of PhaseHC over GethHC in YCSB (56.38X to 91.88X) is significantly larger than the corresponding improvements it made in both Wiki (6.19X to 25.20X) and ETHT (2.19X to 10.99X). The major reason is that the total number of MPT nodes in YCSB is larger than that in both Wiki and ETHT when we insert the same number of KV pairs into the initial empty MPT.

End-to-end Throughput of Insert Operator. We last measure the end-to-end throughput of insert operator in those three datasets by varying the number of inserted key-value pairs. Since this is the first work that exploits GPU to accelerate the performance of MPT, we use the CPU-Geth, which combines the official implementation of **node-update** and **hash-compute** algorithms (i.e., GethNU and GethHC) in Geth, and run it on a high-end CPU server with 104 threads as a performance indicator to show the superiority of our proposed GPU-Phase and GPU-Lock, which uses PhaseNU and LockNU as the GPU-based **node-update** algorithms, respectively. Both GPU-Phase and GPU-Lock employ the same GPU-based **hash-compute** algorithm PhaseHC.

As reported in Figure 16, GPU-based methods (both GPU-Phase and GPU-Lock) outperform official solution of Geth (i.e., CPU-Geth) in all cases. Specifically, average speedup times of GPU-Phase over CPU-Geth is 5.11X, 3.83X and 19.97X in Wiki, ETHT, and YCSB while average speedup times of GPU-Lock over CPU-Geth is 4.85X, 5.44X and 29.69X in Wiki, ETHT, and YCSB, respectively.



(a) lookup workload (b) Mixed lookup/insert workload

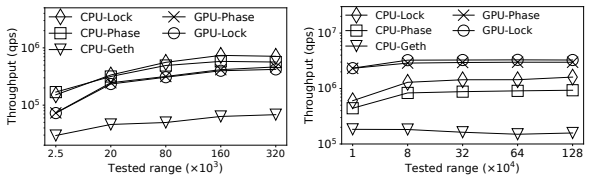
Figure 17: Response time of MPT, on YCSB

5.2.2 The evaluation of lookup operator in MPT. In this section, we evaluate the performance of lookup workload and concurrent lookup and insert workload on CPU-based and GPU-based MPT. In particular, we build an MPT with 640K KV pairs and then perform the corresponding workload with YCSB. **lookup workload.** In this experiment, we compare the response time of CPU-based and GPU-based lookup operators by varying the number of operations from 10K to 1280K. For the GPU-based lookup operator, we implement it as the method we mentioned in Section 2.2. To provide a fair comparison, we implement multi-thread lookup operators as the official implementation of MPT in Geth only provides a single thread lookup operator. Specifically, we uniformly assign all the lookup requests to all 104 threads in the CPU server and parallel process them. As shown in Figure 17(a), the speedup times of GPU-based lookup operator over CPU-based method is 15.05X on average.

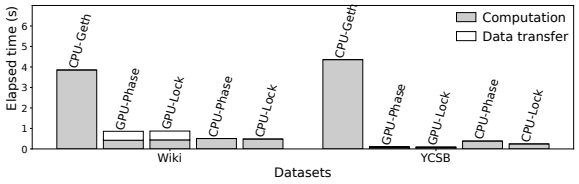
Mixed Lookup/Insert Workload. We next verify the performance of CPU-based and GPU-based MPTs by supporting concurrent lookup and insert operators. The total number of operations is 640K in all tested cases and the ratio of lookup operator in it ranges from 0.1 to 0.9. To support mixed workload on the CPU-based MPT (i.e., CPU-Geth) in the official implementation of Geth, the lookup and the **node-update** subroutine of its insert operators are sequentially executed in the workload. For GPU-Phase, the lookup operators run with the nodelookup operator (see Section 3.1.1) in the expanding phase of PhaseNU of the insert operators to support concurrent read and write operations on it. For GPU-Lock, we employ an optimistic read lock for each lookup operator in the mixed workload as the **node-update** of LockNU also uses optimistic read locks. According to the experimental results in Figure 17(b), we can conclude that both GPU-based methods GPU-Phase and GPU-Lock outperform the CPU-based method CPU-Geth when support concurrent lookup and insert operators. The average speedup of GPU-Phase and GPU-Lock over CPU-Geth is 18.61X and 22.07X, respectively.

5.3 Effectiveness Study

Effect of Multi-core CPU. We extend GPU-based algorithms PhaseNU, LockNU, and PhaseHC to multi-core CPU by following the ideas in Section 3.3. Thus, we have three CPU-based methods: (i) CPU-Geth, the official implementation of Geth; (ii) CPU-Phase, it adapts PhaseNU and PhaseHC to multi-core CPU; and (iii) CPU-Lock, it employs LockNU and PhaseHC to multi-core CPU. As illustrated in Figure 18, our proposed methods are obviously better than CPU-Geth on both Wiki and YCSB. Interestingly, our CPU-based methods (CPU-Phase and CPU-Lock) outperform GPU-based methods (GPU-Phase and GPU-Lock) on Wiki, but GPU-based methods outperform CPU-based methods on YCSB, as shown in

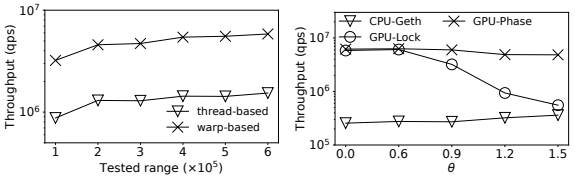


(a) Throughputs on Wiki (b) Throughputs on YCSB



(c) Elapsed time breakdown

Figure 18: Effect of multi-core CPU



(a) Warp-based execution, on ETHT (b) Effect of θ, on YCSB

Figure 19: Effectiveness evaluation

Figures 18(a) and (b). The reason is that the value size of Wiki is large (i.e., 10,894B on average) but that of YCSB is small (i.e., 523B on average). The large value size in Wiki incurs significant data transferring cost from CPU to GPU of the GPU-based methods, as shown in Figure 18(c).

Effect of Warp-based PhaseHC **Execution.** We compare the performance of our proposed warp-based PhaseHC with a thread-based PhaseHC, which uses a single thread to compute the hash value of each MPT node. As depicted in Figure 19(a), the throughput of the warp-based PhaseHC is better than that of the thread-based PhaseHC by up to 3.89X.

Effect of θ in YCSB. We last evaluate the effect of θ in YCSB. The total number of operations is 960K, and MPT has 320K records at the beginning in all tested cases. The ratio of lookup, insert, and update operations are 0.4, 0.2, and 0.4 respectively. The results are plotted in 19(b). It shows that the performance of LockNU falls with the rising of θ, and PhaseNU performs stable as it is a lock-free method. The reason is the LockNU incurs heavy contention of the write lock on the parent of the leaf node when many threads are updating the same key. It confirms our insights in Section 3.1.3 as Zipf distribution is a clustered distribution.

6 RELATED WORK

Immutable Data Management. Immutable data management has many applications, e.g., blockchain systems [5, 7, 14, 15, 17, 64, 69], verifiable databases [2, 20, 80, 82, 85] and collaborative data analysis [10, 24]. The core of immutable data management is providing tampered evidence control and muti-version access. Thus, many cryptographic data structures [5, 12, 54] (a.k.a., Structurally Invariant and Reusable Indexes (*SIRI*) [83]) have been proposed to index immutable data for different applications. Several techniques [6, 71] have been proposed to accelerate Merkle Tree (MT) [61] via GPU. However, these techniques for MT cannot adapt to MPT on GPU as (i) they do not support insert and lookup operations and (ii) the hash-compute algorithm requires all leaves must be the same level.

Parallel Indexes on CPU and GPU. Many studies [21, 22, 45, 48, 50–52, 60, 73] worked on parallel index structures on CPU and GPU. On multi-core CPU, B^{link}-Tree[48] uses additional side links to reduce conflicts in B-Tree. Bw-Tree [52] is a lock-free index which avoids locks by using an indirection layer and appending delta records to nodes. Optimistic lock coupling and read-optimized write exclusion are proposed to synchronize the concurrent Adaptive Radix Trie [50, 51]. On GPU, Harmonia [79] improves cache locality of B+Tree by dividing the tree into a key region and a child region and storing the child region in the cache. Eirene [86] is a concurrency control framework for B+Tree that guarantees linearizability among concurrent requests and decreases the response time. Unfortunately, existing techniques cannot be directly adapted to accelerate MPT on GPU as they cannot efficiently resolve node splitting conflict and hash computing conflict of MPT on GPU.

Query Processing with GPU. Exploiting GPU to accelerate query processing is widely used in database community. For example, optimizing relational operators on GPU (e.g., join [35, 37, 43, 57, 58, 68, 75, 76], aggregation [44, 72] and sort [34, 77]); designing heterogeneous query engines to improve the analytical processing performance with GPU [11, 28, 30–33, 38, 47, 55, 56, 67, 81]; and enhancing the transaction processing performance [25, 36].

7 CONCLUSION

In this paper, we accelerated the performance of MPT by exploiting the high parallelism of GPU. Specifically, we proposed lock-free algorithm PhaseNU and lock-based algorithm LockNU for the **node-update** subroutine and devised a GPU-based hash-compute algorithm PhaseHC for **hash-compute** subroutine. We verified the effectiveness of our proposed GPU-accelerated MPT by two case studies on the real-world system Geth and LedgerDB and extensive empirical experimental studies. The promising directions for future work include (i) devising efficient solutions to process out-of-GPU memory MPT, and (ii) exploiting our proposed techniques to accelerate other cryptographic index structures.

ACKNOWLEDGMENTS

We thank Dr. Xuetao Wei in Southern University of Science and Technology and Mr. Mengde Zhu in Beijing University of Posts and Telecommunications for their insightful discussions and suggestions. We also thank all reviewers for their constructive comments to help us improve the quality of this paper. This work is partially supported by Shenzhen Fundamental Research Program (Grant No. 20220815112848002), the Guangdong Provincial Key Laboratory (Grant No. 2020B121201001) and a research gift from Huawei Gauss department. Dr. Bo Tang is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China.

REFERENCES

[1] 2022. *LedgerDB.* https://www.alibabacloud.com/product/ledgerdb
[2] 2023. *Amazon Quantum Ledger Database.* https://aws.amazon.com/qldb
[3] 2023. *Cgo Library.* https://pkg.go.dev/cmd/cgo
[4] 2023. *CUDA Toolkit.* https://developer.nvidia.com/cuda-toolkit
[5] 2023. *Ethereum.* https://ethereum.org
[6] 2023. *fastmerkle.* https://github.com/shilch/fastmerkle
[7] 2023. *FISCO BCOS.* https://github.com/FISCO-BCOS/FISCO-BCOS
[8] 2023. *FNV hash.* http://www.isthe.com/chongo/tech/comp/fnv/
[9] 2023. *Go Ethereum.* https://github.com/ethereum/go-ethereum
[10] 2023. *Google Docs.* https://docs.google.com
[11] 2023. *Heavy.ai.* https://www.heavy.ai
[12] 2023. *Hyperledger.* https://www.hyperledger.org
[13] 2023. *Intel Thread Building Block.* https://github.com/oneapi-src/oneTBB
[14] 2023. *Near Protocol.* https://near.org
[15] 2023. *Quorum.* https://github.com/ConsenSys/quorum
[16] 2023. *RLP Encoding.* https://ethereum.org/en/developers/docs/data-structures-and-encoding/rlp
[17] 2023. *Zilliqa.* https://github.com/Zilliqa/Zilliqa
[18] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. 265–283.
[19] Md. Maksudul Alam, Srikanth B. Yoginath, and Kalyan S. Perumalla. 2016. Performance of Point and Range Queries for In-memory Databases Using Radix Trees on GPUs. In *HPCC/SmartCity/DSS*. 1493–1500.
[20] Panagiotis Antonopoulos, Raghav Kaushik, Hanuma Kodavalla, Sergio Rosales Aceves, Reilly Wong, Jason Anderson, and Jakub Szymaszek. 2021. SQL Ledger: Cryptographically Verifiable Data in Azure SQL Database. In *SIGMOD*. 2437–2449.
[21] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. 2018. A Dynamic Hash Table for the GPU. In *IPDPS*. 419–429.
[22] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. 2019. Engineering a High-Performance GPU B-Tree. In *PPoPP*. 145–157.
[23] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2013. Keccak. In *Advances in Cryptology – EUROCRYPT*. 313–314.
[24] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. 2015. DataHub: Collaborative Data Science & Dataset Version Management at Scale. In *CIDR*.
[25] Nils Boeschen and Carsten Binnig. 2022. GaccO - A GPU-Accelerated OLTP DBMS. In *SIGMOD*. 1003–1016.
[26] Pierre-Louis Cayrel, Gerhard Hoffmann, and Michael Schneider. 2011. GPU Implementation of the Keccak Hash Function Family. In *Information Security and Assurance*. 33–42.
[27] Jemin Andrew Choi, Sidi Mohamed Beillahi, Peilun Li, Andreas G. Veneris, and Fan Long. 2022. LMPTs: Eliminating Storage Bottlenecks for Processing Blockchain Transactions. In *ICBC*. 1–9.
[28] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating Heterogeneous CPU-GPU Parallelism in JIT Compiled Engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556.
[29] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.
[30] Pengjie Cui, Haotian Liu, Bo Tang, and Ye Yuan. 2024. CGgraph: An Ultra-fast Graph Processing System on Modern Commodity CPU-GPU Co-processor. *Proc. VLDB Endow.* 17, 6 (2024), 1405–1417.
[31] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2007. GPUQP: Query Co-Processing Using Graphics Processors. In *SIGMOD*. 1061–1063.
[32] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *SIGMOD*. 1603–1618.
[33] Henning Funke and Jens Teubner. 2020. Data-Parallel Query Processing on Non-Uniform Data. *Proc. VLDB Endow.* 13, 6 (2020), 884–897.
[34] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: High Performance Graphics Co-Processor Sorting for Large Database Management. In *SIGMOD*. 325–336.
[35] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors. In *SIGMOD*. 511–524.
[36] Bingsheng He and Jeffrey Xu Yu. 2011. High-Throughput Transaction Executions on Graphics Processors. *Proc. VLDB Endow.* 4, 5 (2011), 314–325.
[37] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. VLDB Endow.* 6, 10 (2013), 889–900.
[38] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-Oblivious Parallelism for in-Memory Column-Stores. *Proc. VLDB Endow.* 6, 9 (2013), 709–720.
[39] Ying Huang, Xiaoying Zheng, and Yongxin Zhu. 2023. Optimized CPU–GPU collaborative acceleration of zero-knowledge proof for confidential transactions. *Journal of Systems Architecture* 135 (2023), 102807.
[40] Rares Ifrim, Dumitrel Loghin, and Decebal Popescu. 2023. Baldur: A Hybrid Blockchain Database with FPGA or GPU Acceleration. In *VDBS*. 19–27.
[41] Konstantinos Iliakis, Konstantina Koliogeorgi, Antonios Litke, Theodora Varvarigou, and Dimitrios Soudris. 2022. GPU accelerated blockchain over key-value database transactions. *IET Blockchain* 2, 1 (2022), 1–12.
[42] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating Graph Sampling for Graph Machine Learning Using GPUs. In *EuroSys*. 311–326.
[43] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU Join Processing Revisited. In *DaMoN*. 55–62.
[44] Tomas Karnagel, René Müller, and Guy M. Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation. In *ADMS@VLDB*.
[45] Martin Koppehel, Tobias Groth, Sven Groppe, and Thilo Pionteck. 2021. CuART - a CUDA-based, scalable Radix-Tree lookup and update engine. In *ICPP*. 12:1–12:10.
[46] Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691.
[47] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The Art of Balance: A RateupDB™ Experience of Building a CPU/GPU Hybrid Database Product. *Proc. VLDB Endow.* 14, 12 (2021), 2999–3013.
[48] Philip L. Lehman and s. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.* 6, 4 (1981), 650–670.
[49] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.
[50] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49.
[51] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *DaMoN*. Article 3.
[52] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. 302–313.
[53] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. 2020. A Decentralized Blockchain with High Throughput and Fast Confirmation. In *USENIX ATC*. Article 35.
[54] Qian Lin, Kaiyuan Yang, Tien Tuan Anh Dinh, Qingchao Cai, Gang Chen, Beng Chin Ooi, Pingcheng Ruan, Sheng Wang, Zhongle Xie, Meihui Zhang, and Olafs Vandans. 2020. ForkBase: Immutable, Tamper-evident Storage Substrate for Branchable Applications. In *ICDE*. 1718–1721.
[55] Haotian Liu, Bo Tang, Jiashu Zhang, Yangshen Deng, Xiao Yan, Xinying Zheng, Qiaomu Shen, Dan Zeng, Zunyao Mao, Chaozu Zhang, Zhengxin You, Zhihao Wang, Runzhe Jiang, Fang Wang, Man Lung Yiu, Huan Li, Mingji Han, Qian Li, and Zhenghai Luo. 2022. GHive: Accelerating Analytical Query Processing in Apache Hive via CPU-GPU Heterogeneous Computing. In *SoCC*. 158–172.
[56] Haotian Liu, Bo Tang, Jiashu Zhang, Yangshen Deng, Xinying Zheng, Qiaomu Shen, Xiao Yan, Dan Zeng, Zunyao Mao, Chaozu Zhang, Zhengxin You, Zhihao Wang, Runzhe Jiang, Fang Wang, Man Lung Yiu, Huan Li, Mingji Han, Qian Li, and Zhenghai Luo. 2022. GHive: A Demonstration of GPU-Accelerated Query Processing in Apache Hive. In *SIGMOD*. 2417–2420.
[57] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *SIGMOD*. 1633–1649.
[58] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *SIGMOD*. 1017–1032.
[59] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 2023. GZKP: A GPU Accelerated Zero-Knowledge Proof System. In *ASPLOS*. 340–353.
[60] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*. 183–196.
[61] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO*. 369–378.
[62] Shin Morishima and Hiroki Matsutani. 2018. Accelerating Blockchain Search of Full Nodes Using GPUs. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. 244–248.
[63] Donald R. Morrison. 1968. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (1968), 514–534.
[64] Satoshi Nakamoto. 2009. *Bitcoin: A Peer-to-Peer Electronic Cash System.* https://bitcoin.org/bitcoin.pdf
[65] Ning Ni and Yongxin Zhu. 2023. Enabling zero knowledge proof by accelerating zk-SNARK kernels on GPU. *J. Parallel and Distrib. Comput.* 173 (2023), 20–31.

[66] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703

[67] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving Execution Efficiency of Just-in-Time Compilation Based Query Processing on GPUs. *Proc. VLDB Endow.* 14, 2 (2020), 202–214.

[68] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *SIGMOD*. 1413–1425.

[69] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. 2021. RainBlock: Faster Transaction Processing in Public Blockchains. In *USENIX ATC*. 333–347.

[70] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676.

[71] Ciprian Pungila and Viorel Negru. 2019. Improving Blockchain Security Validation and Transaction Processing Through Heterogeneous Computing. In *CISIS and ICEUTE*, Vol. 951. 132–140.

[72] Viktor Rosenfeld, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2019. Performance Analysis and Automatic Tuning of Hash Aggregation on GPUs. In *DaMoN*. Article 8.

[73] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A Hybrid B+-Tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *SIGMOD*. 1523–1538.

[74] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *SIGMOD*. 1617–1632.

[75] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *ICDE*. 698–709.

[76] Evangelia A. Sitaridi and Kenneth A. Ross. 2012. Ameliorating Memory Contention of OLAP Operators on GPU Processors. In *DaMoN*. 39–47.

[77] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *SIGMOD*. 417–432.

[78] Gavin Wood. 2023. ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. https://ethereum.github.io/yellowpaper/paper.pdf

[79] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. 2019. Harmonia: A High Throughput B+tree for GPUs. In *PPoPP*. 133–144.

[80] Xinying Yang, Yuan Zhang, Sheng Wang, Benquan Yu, Feifei Li, Yize Li, and Wenyuan Yan. 2020. LedgerDB: A Centralized Ledger Database for Universal Audit and Verification. *Proc. VLDB Endow.* 13, 12 (2020), 3138–3151.

[81] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating Data Placement and Query Execution in Heterogeneous CPU-GPU DBMS. *Proc. VLDB Endow.* 15, 11 (2022), 2491–2503.

[82] Cong Yue, Tien Tuan Anh Dinh, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Xiaokui Xiao. 2023. GlassDB: An Efficient Verifiable Ledger Database System Through Transparency. *Proc. VLDB Endow.* 16, 6 (2023), 1359–1371.

[83] Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, Sheng Wang, and Xiaokui Xiao. 2020. Analysis of Indexing Structures for Immutable Data. In *SIGMOD*. 925–935.

[84] Cong Yue, Meihui Zhang, Changhao Zhu, Gang Chen, Dumitrel Loghin, and Beng Chin Ooi. 2023. VeriBench: Analyzing the Performance of Database Systems with Verifiability. *Proc. VLDB Endow.* 16, 9 (2023), 2145–2157.

[85] Meihui Zhang, Zhongle Xie, Cong Yue, and Ziyue Zhong. 2020. Spitz: A Verifiable Database System. *Proc. VLDB Endow.* 13, 12 (2020), 3449–3460.

[86] Weihua Zhang, Chuanlei Zhao, Lu Peng, Yuzhe Lin, Fengzhe Zhang, and Yunping Lu. 2023. Boosting Performance and QoS for Concurrent GPU B+trees by Combining-Based Synchronization. In *PPoPP*. 1–13.

[87] Jie Zhu, Qi Li, Cong Wang, Xingliang Yuan, Qian Wang, and Kui Ren. 2018. Enabling Generic, Verifiable, and Secure Data Search in Cloud Services. *TPDS* 29, 8 (2018), 1721–1735.