A High Throughput B+tree for SIMD Architectures

Weihua Zhang, Zhaofeng Yan, Yuzhe Lin, Chuanlei Zhao, and Lu Peng, *Senior Member, IEEE*

Abstract—B+tree is one of the most important data structures and has been widely used in different fields. With the increase of concurrent queries and data-scale in storage, designing an efficient B+tree structure has become critical. Due to abundant computation resources, SIMD architectures provide potential opportunities to achieve high query throughput for B+tree. However, prior methods cannot achieve satisfactory performance results due to low resource utilization and poor memory performance. In this paper, we first identify the gaps between B+tree and SIMD architectures. Concurrent B+tree queries involve many global memory accesses and different divergences, which mismatch with SIMD architecture features. Based on this observation, we propose Harmonia, a novel B+tree structure to bridge the gaps. In Harmonia, a B+tree structure is divided into a key region and a child region. The key region stores the nodes with its keys in a breadth-first order. The child region is organized as a prefix-sum array, which only stores each node's first child index in the key region. Since the prefix-sum child region is small and the children's index can be retrieved through index computations, most of it can be stored in on-chip caches, which can achieve good cache locality. To make it more efficient, Harmonia also includes two optimizations: partially-sorted aggregation and narrowed thread-group traversal, which can mitigate memory and execution divergence and improve resource utilization. Evaluations on a 28-core INTEL CPU show that Harmonia can achieve up to 207 million queries per second, which is about 1.7X faster than that of CPU-based HB+Tree [1], a recent state-of-the-art solution. And on a Volta TITAN V GPU, it can achieve up to 3.6 billion queries per second, which is about 3.4X faster than that of GPU-based HB+Tree.

Index Terms—SIMD, B+tree, high-throughput

1 Introduction

B+tree [2] is one of the most important data structures, which has been widely used in different fields, such as web indexing, database, data mining and file systems [3], [4]. In the era of big data, the demand for high throughput processing is increasing. For example, there are millions of searches per second on Google while Alibaba processes 325,000 sale orders per second [5]. Meanwhile, the data scale on server storage is also expanding rapidly. For instance, Netflix estimated that there are 12 PetaByte data per day moved upwards to the data warehouse in stream processing systems [6]. All these factors put tremendous pressures on applications which use B+tree as the index data structure.

single instruction multiple data (SIMD) architectures have been one of the most popular accelerators in modern processors, such as different SIMD extensions in CPUs and its variant SIMT in GPUs. Due to high processing capability, they provide a potential opportunity to accelerate query throughput of B+tree. Many previous works [1], [7], [8], [9], [10] have used SIMD architectures to accelerate the query performance of B+tree. However, those designs have not achieved satisfactory results, due to low resource utilization and poor memory performance.

In this paper, we perform a comprehensive analysis on B+tree and SIMD architectures, and identify several gaps between the characteristics of B+tree and the features of SIMD architectures. For traditional B+tree, a query needs to traverse the tree from root to leaf, which would involve many indirect memory accesses. Moreover, two concurrent executed queries may have different tree traversal paths, which would lead to different divergences when they are processed in a SIMD unit simultaneously. All these characteristics of B+tree are mismatched with the features of SIMD architectures, which impedes the query performance of B+tree on SIMD architectures.

Based on this observation, we propose Harmonia, a novel B+tree structure, to bridge the gaps between B+tree and SIMD architectures. In Harmonia, the B+tree structure is partitioned into two parts: a key region and a child region. The key region stores the nodes with its keys in a breadth-first order. The child region is organized as a prefix-sum array, which only stores each node's first child index in the key region. The locations of other children can be obtained based on these index numbers and the node size. With this compression, most of the prefix-sum array can be stored in caches. Therefore, such a design matches memory hierarchy of modern processors for good cache locality and can avoid indirect memory accesses.

To further improve the query performance of Harmonia, we propose two optimizations including partially-sorted

- *W. Zhang, Z. Yan, Y. Lin, and C. Zhao are with the Software School, Shanghai Key Laboratory of Data Science, Institute for Big Data, and Shanghai Institute of Intelligent Electrontics & System, Fudan University, Shanghai 200433, China. E-mail: {zhangweihua, zfyan16, yzlin14, clzhao16}@fudan.edu.cn.*
- *L. Peng is with the Division of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. E-mail: lpeng@isu.edu.*

Manuscript received 4 Apr. 2019; revised 8 Sept. 2019; accepted 11 Sept. 2019. Date of publication 23 Sept. 2019; date of current version 10 Jan. 2020. (Corresponding author: Weihua Zhang.)
Recommended for acceptance by R. Ge.
Digital Object Identifier no. 10.1109/TPDS.2019.2942918

aggregation (PSA) and narrowed thread-group traversal (NTG). For PSA, we sort the queries in a time window before issuing them. Since adjacent sorted queries are more likely to share the same tree traversal path, it increases the opportunity of coalesced memory accesses when multiple adjacent queries are processed in a SIMD unit. For NTG, we reduce the number of threads for each query to avoid useless comparisons. When the thread group for each query is narrowed, more queries can be combined into a SIMD unit, which may increase execution divergence. To mitigate the execution divergence problem brought by query combinations, we design a model to decide how to narrow the thread group for a query.

Evaluations on a 28-core INTEL CPU show that Harmonia can achieve up to 207 million queries per second, which is about 1.7X faster than that of CPU-based HB+Tree [1], a recent state-of-the-art solution. On a Volta TITAN V GPU, it can achieve up to 3.6 billion queries per second, which is about 3.4X faster than that of GPU-based HB+Tree. The main contributions of our work can be summarized as follows:

- Analysis on the gaps between B+tree and SIMD architectures.
- A novel B+tree structure which matches memory hierarchy well with good locality.
- Two optimizations to reduce divergences and improve computation resource utilization of SIMD architectures.

The rest of this paper is organized as follows. Section 2 introduces the background and discusses our motivation. Section 3 gives out Harmonia structure and tree operations. Section 4 introduces two optimizations applied on Harmonia tree structure. Section 5 introduces the implementation of the Harmonia. Section 6 shows the experimental results. Section 7 surveys the related work. Section 8 concludes the work.

2 Background and Motivation

This section first introduces the background. Then, the gaps between SIMD architectures and B+tree are discussed.

2.1 Modern SIMD Architectures

SIMD architectures have been one of the most popular accelerators in modern processors, including vector units in CPUs and its variant, i.e., SIMT in GPUs. In a SIMD unit, multiple processing units (PUs) or lanes are driven by the same instruction to process different data simultaneously. Note that SIMT can be seen as an execution model, where SIMD is combined with multithreading. Therefore, SIMT is more flexible. For example, it is not necessary for SIMT to assemble the data into a fixed-length SIMD register and SIMT threads can be scheduled to overlap long-latency memory accesses.

To utilize the computation resources in a SIMD unit, a candidate loop is partitioned based on the SIMD width, i.e., the PU number, and the consecutive iterations are grouped together. Each iteration is executed on a PU and different iterations are processed in a SIMD manner. In each step, if the instructions from different iterations are the same, they can be combined into a SIMD instruction and are processed simultaneously. Otherwise, they will be organized into several sub-groups based on their instruction types and are processed one by one. When these sub-groups are executed, only part of PUs in the SIMD unit are used. Therefore, if the execution paths among different iterations are not the same, there exists execution divergence, which would lead to computation resource waste. Moreover, If a batch of memory addresses requested by a SIMD instruction fall within one cache line, which is called coalesced memory access [11], [12], they can be served by single memory transaction. Therefore, the coalesced memory access pattern can improve the memory load efficiency and throughput. Otherwise, multiple memory transactions will be required, which leads to memory divergence.

SIMD architectures provide powerful computation resources. To fully utilize them, an application should have the following characteristics.

Reducing Global Memory Accesses. Global memory accesses are performance bottlenecks for SIMD architectures. Therefore, increasing on-chip data locality and reducing global memory accesses are critical to improving performance.

Avoiding Execution Divergence. All the lanes of a SIMD unit execute the same instruction at a time. Conditional code blocks, such as if-else, would cause execution divergence because some SIMD lanes may execute along the if path while the others may execute along the else path depending on the conditional result of each lane. Since the codes in different execution paths cannot be executed at the same time, they have to be partitioned into several sub-groups based on the execution path. While the instructions of a sub-group are executed, the instructions in the other sub-groups have to wait, which leads to low resource utilization.

Avoiding Memory Divergence. Memory divergence leads to multiple memory transactions which imposes long memory overhead. Therefore, avoiding memory divergence is important for SIMD performance.

2.2 Gaps Between SIMD and B+tree

B+tree is a self-balanced tree [2] where the largest number of children in one node is called fanout. Each internal node can store no more than $fanout - 1$ keys and $fanout$ child references. There are two kinds of B+tree organizations: regular B+tree and implicit B+tree [13]. For regular B+tree, each node in B+tree contains two types of information: key information and child reference information. Child references are used to get the child locations. For implicit B+tree, the tree is complete and only contains key information, which is arranged in an array with the breadth-first order. Implicit B+tree achieves the child locations using index computations. It has to restructure the entire tree for some update operations, such as insert or delete. Since restructuring tree structure is very time-consuming, we mainly focus on regular B+tree in this paper.

For B+tree, when a query is performed, it traverses the tree from the root to a leaf level by level. At each tree level, the query visits one node. It first compares the target with the keys held by current node to find a child whose corresponding range contains the target key. Then it accesses the child reference to fetch the target child's position as the next level node to visit.

Because of high query throughput and the support of order operations, such as range query, B+tree has been

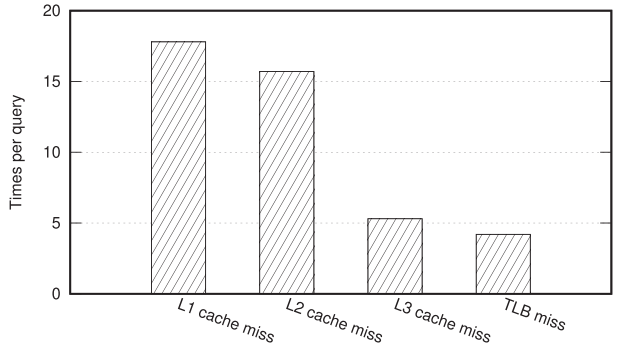


Fig. 1. Memory performance per query.

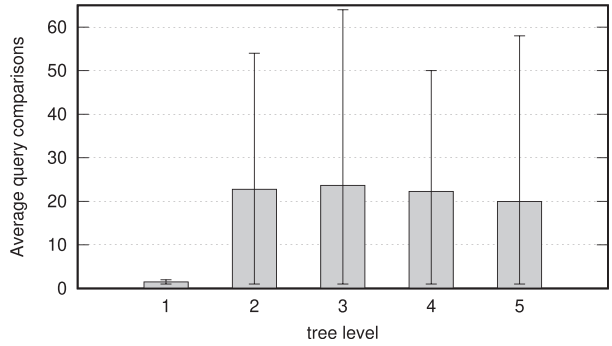


Fig. 3. Query divergence for different levels.

widely used in different fields like web indexing, file systems, and databases. Since search performance is more important for lookup-intensive scenario, such as online analytical processing (OLAP), decision support systems and data mining. [1], [14], [15], [16], B+tree systems typically use batch update instead of mixing search and update operations to achieve high lookup performance. With data scale increasing, it has become more and more critical to further improve B+tree query performance.

It seems that SIMD architecture is a potential solution to accelerating search performance of B+tree due to its powerful computation resources. However, prior SIMD-based B+tree methods cannot achieve satisfactory performance results. To understand the underlying reasons, we perform a detailed analysis and uncover three main sources of the performance gaps between B+tree and SIMD architectures. In the following analysis, we use the CPU configuration in Section 6 as our target hardware platform, the tree size is 2^{23} with 64-fanout and we randomly generate 100 queries for analysis.

Gap in Memory Access Requirement. Each B+tree query needs to traverse the tree from root to leaf. This traversal brings lots of indirect memory accesses, which is proportional to tree height. For instance, if the height of a B+tree is five, there are four indirect global memory accesses when a query traverses the tree from root node to its target leaf node. To illustrate this problem, we use PAPI (Version 5.6.1.0) [17] to collect average results of four memory metrics including L1/L2/L3 cache misses and TLB miss. As the data in Fig. 1 show, the memory performance is poor. There are 17.8 L1 cache misses, 15.7 L2 cache misses, 5.3 L3 cache misses and 4.3 TLB misses on average for a query.

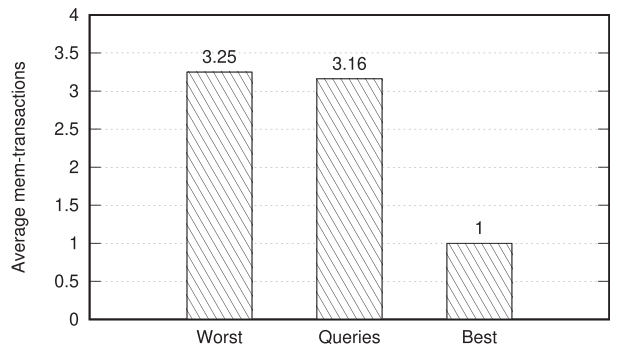


Fig. 2. Average memory transactions for a SIMD unit.

Gap in Memory Divergence. Since the target leaf node of a query is generally random, multiple queries may traverse the tree along different paths. When they are processed simultaneously, such as in a SIMD unit, the memory accesses are disordered, which would lead to memory divergence and greatly impede the performance. To illustrate it, we collect the average number of memory transactions for a SIMD unit when concurrent queries traverse. For a 4-height and 8-fanout B+tree, each SIMD unit processes 4 queries concurrently and the input query data are randomly generated based on uniform distribution. As shown in Fig. 2, the average number of memory transactions for a SIMD unit (illustrated in the second bar) is 3.16, which is about 97 percent of the worst case (3.25) shown in the first bar of Fig. 2. For the worst case, 4 queries access the root node in a coalesced manner, so it just needs 1 memory transaction. For the other levels, 4 queries access different nodes for the worst case, so it requires 4 memory transactions for each level. Therefore, the memory divergence is very heavy for an unoptimized B+tree.

Gap in Query Divergence. Since the target leaf node of a query is random, multiple queries may require different amounts of comparison operations in each level traversal, which would lead to query divergence. To illustrate this problem, we collect the average comparison number, the largest one and the smallest one in each tree level. As shown in Fig. 3, the comparison numbers of each level for different queries have a large fluctuation although average comparison number is close except level 1 because it's root node with fewer keys. We also collect the SIMD unit utilization of different tree sizes using PAPI. As the data in Fig. 4 show, the computation resource utilization of a SIMD unit is only 66 percent in average due to query divergence.

3 Harmonia Tree Structure

To make the characteristics of B+tree match the features of SIMD architectures, we propose a novel B+tree structure called Harmonia. In this section, we first present Harmonia tree structure. Then we introduce its operations.

3.1 Tree Structure Overview

In a traditional regular B+tree structure, a tree node consists of keys and child references as shown in Fig. 5a. A child reference is a pointer referring to the location of the corresponding next level child. In this organization, the size of each node is large. For example, the size of a node is about 1 KB for a 64-fanout tree. Since the target of each query is random, it is

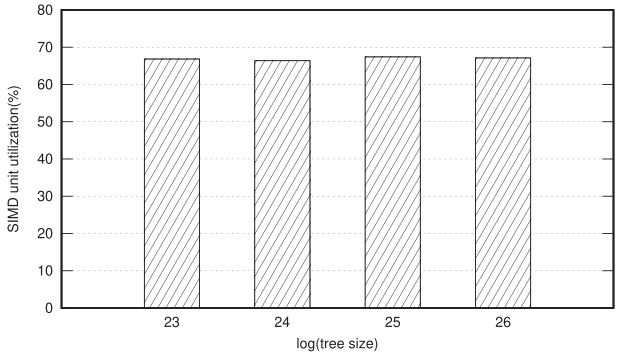


Fig. 4. Computation resource utilization for a SIMD unit.

difficult to utilize the memory hierarchy to explore different types of locality. Moreover, the next child location is obtained through the reference pointer, which will involve many indirect global memory accesses. Therefore, the memory performance of traditional regular B+tree is poor.

To overcome these constraints and fit memory hierarchy better, the tree structure is partitioned into two parts in Harmonia: a key region and a child region. The key region is a one-dimensional array which holds the key information of original B+tree nodes in a breadth-first order. The key region is organized in node granularity and the size of each item (a node) is fixed ($(fanout - 1) * keysize$). The child region is organized as a prefix-sum array. Each item in the array is the index of the node's first child in the key region, which equals to the node number in the key region before its first child. For example, the prefix-sum child array of the regular B+tree in Fig. 5a is $[1, 4, 6, 7, 9 \ldots]$. It means the first child index of node 0 (root) is 1, and the first child index of node 1 is 4 and so on. The number of children in a node can be obtained by the prefix-sum value of its successor node minus its prefix-sum value. Moreover, the index of any child in the key region can be obtained through simple index computation.

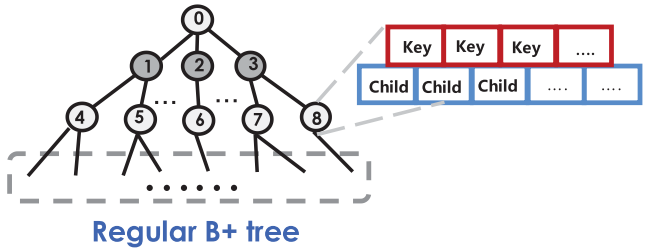
In this organization, the size of the prefix-sum child array is small. For example, for a 64-fanout 4-level B+tree, the size of its prefix-sum array at most is only about 16 KB. Therefore, most of the prefix-sum child array, even a very large B+tree, can be saved in low-latency on-chip caches, which can improve memory locality.

3.2 Tree Operation

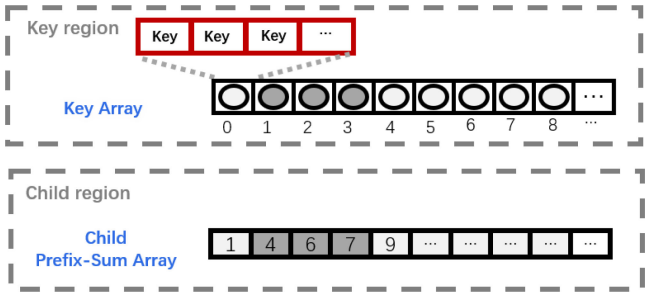
Based on the above design, we further describe how Harmonia handles common B+tree operations in a batch update scenario, including search, range query, update, insert and delete. Batch update scenario is phase-based because updates are relatively infrequent [18] and can be deferred [19], [20]. For example, it was reported that there is a high read/write ratio (about 35:1) in TPC-H [18]. Therefore, in Harmonia, operations are separated into two phases, query phase and update phase. In Harmonia's query phase, SIMD units are used to accelerate query performance. In the update phase, batched updates are processed and the B+tree is synchronized after the update phase.

3.2.1 Search and Range Query

To traverse a B+tree, a query needs to search from the tree root to the target leaf level by level. For each level of B+tree,



(a) Regular B+tree structure



(b) Harmonia tree structure

Fig. 5. Regular B+tree and Harmonia B+tree.

the query first compares with the keys in the current node (an item of the key region) and finds the child whose corresponding range contains the target key. Suppose the ith child is the target child and current node index is $node\_idx$. Since the prefix-sum child array contains the first child's index, the ith child's index can be computed through Equation (1) and the next level node can be obtained through accessing the key region

$$child\_idx = PrefixSum[node\_idx] + i - 1. \quad (1)$$

For example, when we are at the root node whose $node\_idx$ is 0 and try to visit its second child ($i = 2$), we will calculate $child\_idx$ with Equation (1), so the child index of root in the key region is 2. Therefore, we can get the next level node based on its index (2) in the key region.

After the target leaf node is reached and the target key is found, a query is finished. For a range query, it can use the basic query operation to get the first target key in the range, and scan the key region from the first target key to the last target key in the query range. Since the key region is a consecutive array, range queries can achieve high performance.

3.2.2 Update, Insert and Delete

For an update (update an existing record's value) operation, it is similar to a query. After the target key is acquired, the value is updated. Compared with update, insert (insert a new record) and delete (delete an existing record) are more complex because they may change the tree structure. Since insert and delete are a pair of inverse operations, we mainly discuss the details of insert here.

For a single insert operation, it needs to retrieve the target leaf node through a search operation. If the target leaf node is not full, the record will be inserted into the target node. When the target node is full, the target node needs to be split and a new node will be created. Because the current key region is organized in a consecutive way, when a new node is created, the key region has to be reorganized. The nodes

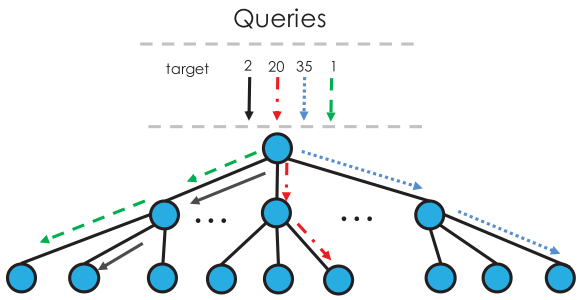


Fig. 6. B+tree.

after the created node must be moved backward so the new node can be inserted into the key region, while the corresponding prefix-sum array items need to be updated due to the change of key region item location.

When multiple updates are processed in a parallel manner, thread safe must be guaranteed. In our current design, a simple locking strategy is used.

Algorithm 1. Syn for Tree Update

```
1: if Operations == updates without split or merge then
2:     //Locking strategy of updates without split or merge
3:     LOCK(coarse_lock)
4:     global_count++
5:     RELEASE(coarse_lock)
6:
7:     LOCK(node.fine_lock)
8:     operation_without_split_or_merge()
9:     RELEASE(node.fine_lock)
10:
11:    LOCK(coarse_lock)
12:    global_count–
13:    RELEASE(coarse_lock)
14: else
15:    //Locking strategy of updates with split or merge
16:    RETRY:
17:    LOCK(coarse_lock)
18:    if global_count == 0 then
19:      operation_with_split_or_merge()
20:      RELEASE(coarse_lock)
21:    else
22:      RELEASE(coarse_lock)
23:      goto RETRY
24:    end if
25: end if
```

If an operation leads to a change of tree structure like split (in insert) or merge (in delete), a coarse-grained lock is used to protect the entire tree. Otherwise, a fine-grained lock is used to protect the particular target leaf node. Moreover, there needs a mechanism, as shown in Algorithm 1, to avoid conflicts between the coarse-grained lock and fine-grained locks. To achieve this goal, a global counter is used to record the number of in-process updates with fine-grained locks. The coarse-grained lock is also used to protect global counter accesses. When an operation needs to update the tree, it needs to first get the coarse-grained lock in order to update the global counter or check whether it is zero. If it is an insert operation without split or a deletion operation without merge (Lines 3-13), it increases the global counter by one after acquiring the coarse-grained lock, then releases the coarse-grained lock. Then, it locks the target leaf using the corresponding fine-grained lock. After the operation is completed, the fine-grained lock is released and the global counter is decreased by one with the protection of the coarse-grained lock; If an insert operation leads to a split or an deletion operation leads to merge (Lines 16-24), it needs to get the coarse-grained lock and check whether the global counter is zero. If so, it will finish its operations and release the lock. Otherwise, it will release the lock first to avoid deadlock and retry the step. Through such a design, the thread safety can be guaranteed.

Although this design can process the update operations, the memory movement of key region due to node splitting or merging will involve an enormous overhead. To reduce the overhead, the memory movements are performed after a batch of update operations are finished. To support such a design, Harmonia uses auxiliary nodes to update the tree structure for node splitting or merging. Here, we use node splitting as an example to illustrate how it works. When an insert causes one node to split, an auxiliary node is created for its father node. The information of keys and children references of its father node is copied into the auxiliary node and the father node is tagged as shadowed. After that, the split is processed on the auxiliary node and the reference pointer to the newly created node is saved in the auxiliary node. In such a design, when a node is traversed, its status is first checked. If the status is shadowed, its auxiliary node will be used for children retrieval. Otherwise, the original prefix array is used for children retrieval.

After all update operations in a batch are done, the tree structure might not follow the rules of Harmonia. Therefore, we need to update the auxiliary node's information into Harmonia to maintain the tree structure of Harmonia. The key region is extended first and some original items in the key region are moved backward to make room for the newly created nodes. And then put the auxiliary nodes in the right location. Since the locations of all these data movements can be known in advance, some of them can be processed in parallel.

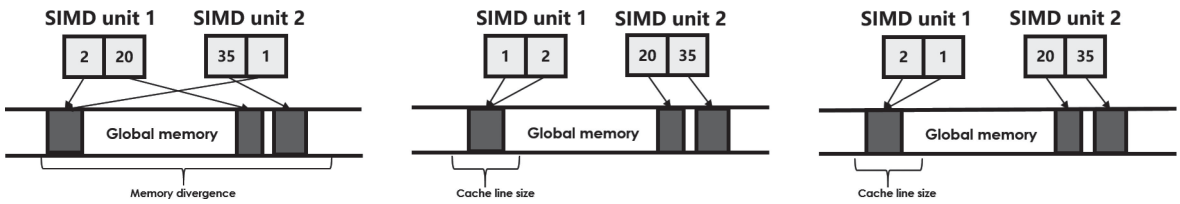
Movements after batch can improve update throughput significantly and achieve comparable performance with those of the multi-thread traditional B+tree and the state-of-the-art GPU B+tree as the data shown in Section 6 (Fig. 22).

4 Harmonia Optimizations

To reduce divergences and improve computation resource utilization of SIMD units, we further introduce two optimizations for Harmonia including partially-sorted aggregation (PSA) and narrowed thread-group traversal (NTG).

4.1 Partially-Sorted Aggregation (PSA)

When an application is executed, the most efficient memory access manner is coalesced. For B+tree, the targets of multiple queries are generally random. When adjacent queries are processed in a SIMD instruction, it is difficult to achieve a coalesced memory access because they would traverse the tree along different paths. Fig. 6 shows an example. Four query targets are 2, 20, 35 and 1 individually. When they traverse the tree and two adjacent queries are combined into a SIMD instruction, there is no coalesced memory access after they leave the root node, as shown in Fig. 7a. Therefore,



(a) Queries (b) Sorted queries (c) Partially-sorted queries

Fig. 7. An example of memory access pattern for queries.

processing these concurrent queries in a SIMD instruction would lead to poor performance due to memory divergence. In this section, we will propose a partially-sorted aggregation for better memory performance.

4.1.1 Sorted Aggregation

If multiple queries have shared part of the traversal path, the memory accesses can be coalesced when they are processed in a SIMD unit. For instance, if the queries with target 1 and target 2 in Fig. 6 are processed in a SIMD unit, there are coalesced memory accesses for their first two level traversals.

For two concurrent queries, they will have more opportunities to share a traversal path if they have closer targets. To achieve this goal, a solution is to sort the queries in a time window before they are issued. For the example in Fig. 6, the query target sequence becomes 1, 2, 20, and 35 after sorting as Fig. 8 shows. When two adjacent queries are combined into a SIMD unit, the SIMD unit with the first two queries will have coalesced memory accesses for their shared traversal path as shown in Fig. 7b. Moreover, because the queries in the same SIMD unit will go through the same path, it can also mitigate execution divergence among them.

Although sorting queries can reduce memory divergence and execution divergence, it brings additional sorting overhead. To illustrate this problem, we evaluate the overhead using radix sort [21] to make a batch of queries sorted before assigning them to the B+tree concurrent search kernel. As the data in Fig. 9 show, the kernel performance has about 22 percent improvement compared with that of the original one. However, there is about 7 percent performance slowdown for the total execution time. The reason behind this is that complete sorting will generate more than 25 percent overhead.

4.1.2 Partially-Sorted Aggregation

To achieve a coalesced memory access, multiple memory accesses in a SIMD unit only need to fall into the address space of a cache line even they are unordered. As shown in Fig. 7c, although the query to target 2 is before the query to target 1, we can still achieve coalesced memory accesses for their shared path, which has the same effect with that of the completely sorted queries as shown in Fig. 7b. Therefore, to achieve the goal of coalesced memory access, there is no need to sort the queries within a group; a partial sorting among groups can achieve the effect similar to the complete sorting for coalesced memory access. Moreover, bit-wise sorting algorithms, such as radix sort, are commonly used algorithms on SIMD architectures because they can provide stable performance for a large workload [21], [22]. For these bit-wise sorting algorithms, the execution time is proportional to the sorted bits as the data shown in Fig. 10, which are normalized statistic data collected on GPU, using radix sort algorithms supported by CUB [21] library. Therefore, partial sorting can also be used to reduce the sorting overhead. As the data in the third bar of Fig. 9 show, the sorting overhead is brought down after partial sorting is applied and the search performance is comparable to that of the completely sorted method. The overall performance has about 10 percent improvement compared with that of the original one.

For a partial sorting, the queries will be sorted based on their most significant N bits. If N is large, there is a high probability that the targets of sorted queries are closer. However, it will lead to a higher sorting overhead. Here, we discuss how to decide the PSA size to achieve a better trade-off between the number of coalesced memory accesses and the sorting overhead. Suppose each key is represented by B bits, the size of traversed B+tree is T and a cache line can save K keys. In this condition, the key range is 2^B and each existing key in the tree can averagely cover the key range of $2^B/T$. The keys in a cache line can cover the key range of $2^B/T * K$ and the bits to represent this range is $\log_2(2^B/T * K)$ on

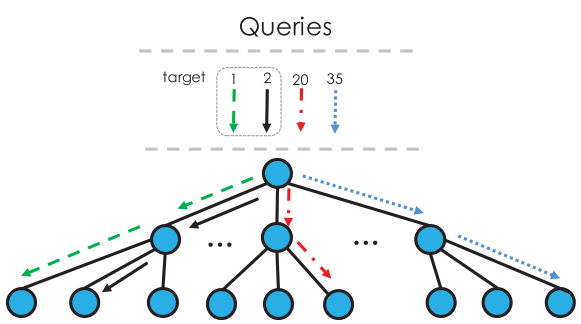


Fig. 8. Queries share traversal path.

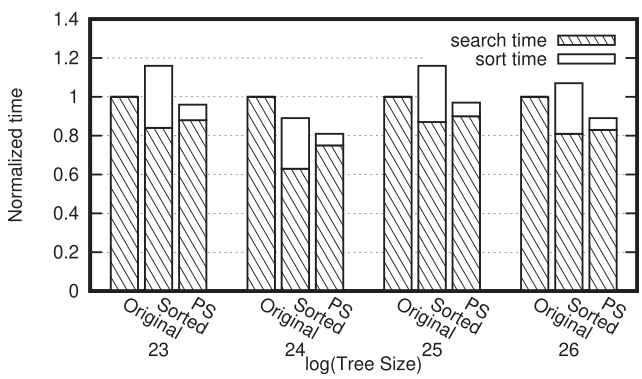


Fig. 9. Sorted queries (sorted) and partially-sorted queries (PS) search time normalized to the search time of original queries (original).

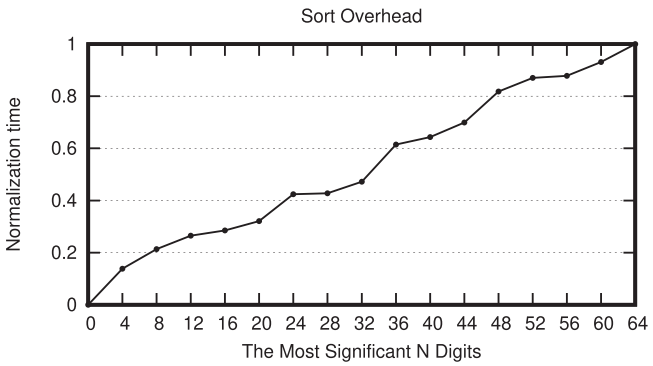


Fig. 10. Normalized time for different sorting bits.

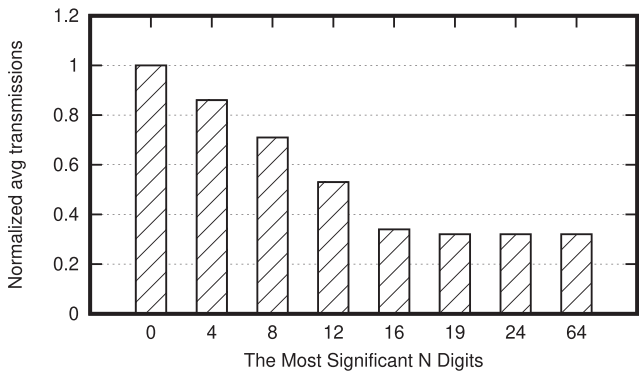


Fig. 11. Normalized avg transmissions for different sorting bits.

average. If the memory requests of multiple queries in a SIMD unit fall in the covering range of a cache line, no matter whether they are sorted or not, they are coalesced memory accesses. Therefore, it is unnecessary to sort the queries when their target keys fall in the same cache line. Based on the above analysis, the value of N can be calculated using Equation (2). Note, our analysis is conservative because we suppose the key value is full in its space. In reality, the key number is smaller than its space size. Therefore, it is possible that the targets exceeding the covering range of a cache line achieve a coalesced memory access

$$N = B - \log_2\left(\frac{2^B}{T} * K\right). \tag{2}$$

As an example, suppose the key is represented by 64 bits ($B = 64$), the tree size is 2^{23} ($T = 2^{23}$) and the size of a cache line is 128-byte, which can store 16 keys ($K = 16$). Based on Equation (2), the value of N equals to 19. To verify its efficiency, we collect average memory transactions per warp for different partially-sorted bits and the normalized sorting time for completely sorting on GPU. As the data shown in Fig. 11, only sorting 19 bits can achieve the similar optimization effort as that of completely sorted. Moreover, its overhead is about 35 percent of the completely sorted method. The data also illustrate that the design can achieve a better trade-off. We also evaluate other tree sizes and find it can draw the same conclusion. Due to the space constraint, the data are not given out here.

4.2 Narrowed Thread-Group Traversal (NTG)

Traditional methods [1], [8], [9], [23] generally use the fanout number of threads to serve a query.[1] Based on our observation, it has insufficient resource utilization problem due to many unnecessary comparisons. When a query traverses the B+tree, the comparison goal in one tree level is to find a child whose range contains the query target. In a sequential comparison method, only the keys before the target child are compared. However, in a fanout-based parallel comparison manner, all the keys in a node are compared. Although the fanout-based approach simplifies the design, it will lead to computation resource waste because the comparisons with the keys after the target child is useless. Fig. 12a shows an example. Suppose the tree fanout is 8 and the PU number of a SIMD unit is 8 as well. The fanout-based thread group will use the whole SIMD unit to serve a query. So for the query whose target is 2, only the first 3 threads make the useful comparisons, and the rest of comparisons are useless.

In many situations, lots of comparisons are not needed. To illustrate it, we divide the key region into 4 parts evenly for different fanout trees and collect the comparison distribution in these four regions, which means the proportion of queries falling within the four parts. As the data in Fig. 13 show, for different tree fanouts, about 80 percent of queries can find the target child through searching the front 50 percent part of the key segment. The reasons behind it are two folds. First, it is a high probability that a B+tree node is half full, which means a query only needs to compare with a front half fanout number of keys at most for these nodes. Second, data distribution also influences it. Therefore, most comparisons in the fanout-based method are useless, which leads to the waste of computation resources.

To avoid useless comparisons, the thread group for each query should be narrowed. The more the thread group is narrowed, the fewer useless comparisons are involved. After the thread group for a query is narrowed, multiple groups for different queries will be combined into a SIMD unit. Due to the query divergence discussed in Section 2.2,

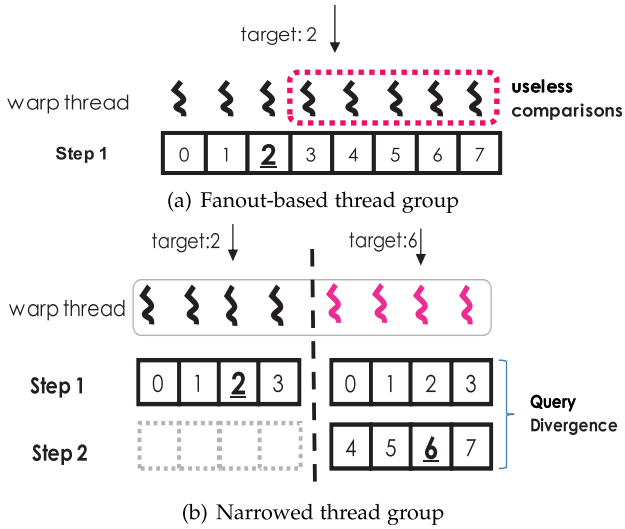


Fig. 12. Example of different thread groups.

1. Due to the scale of data stored in the tree, the tree fanout is typically a large number such as 64 or 128. If the fanout is larger than the PU number of a SIMD unit, all PUs in a SIMD unit are used for a query.

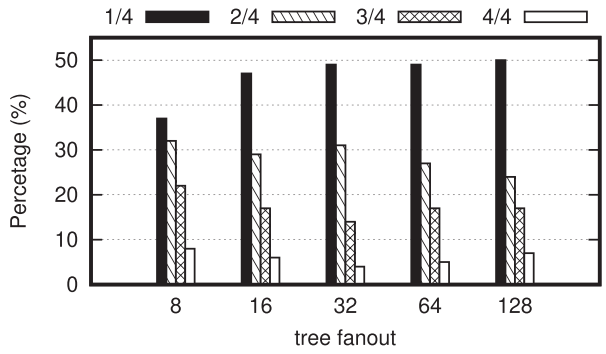


Fig. 13. The proportion of queries accessing the different node parts.

the SIMD unit's traversal time for one tree level will be decided by the thread group that will cost the most comparison operations, which will hurt the performance. Fig. 12b shows an example. If we use 4 threads to serve a query, the useless comparisons for target 2 will be reduced. However, since two queries are combined into a SIMD unit, the threads for target 2 has to execute two steps due to the query divergence brought by target 6, although it only needs one step.

Therefore, when the thread group for a query is narrowed, the overhead involved by query divergence must be considered. To achieve a better trade-off, we propose a model to decide how to narrow the thread group.

Assume the size of the thread group is GS, the number of queries processed by one SIMD unit equals to $SIMDsize/GS$ ($SIMDsize$ is a fixed number). And the throughput (TP) of a SIMD unit for one tree level equals the number of queries per SIMD unit divided by SIMD unit execution time T, which is shown in Equation (3)

$$TP \approx \frac{SIMDsize}{GS * T}. \tag{3}$$

The SIMD unit execution time T is composed of two parts of time: comparison time and memory access time. Because some varieties of SIMD architectures , like GPUs, has a mechanism to hide memory access time by scheduling the warps on the same SM, and the PSA can also alleviate the memory divergence in a warp, the influence of memory access time can be neglected in the throughput equation. Since comparison time is proportional to the comparison execution step, warp execution time T is also positively related to S (the max comparison step that the warp needs to execute.)

When we narrow the thread group, the waste of computation resources is reduced. However, the query divergence will increase. To check whether narrowing thread group can still get performance improvement, we compare the SIMD unit throughput before narrowing (TP_b) and after narrowing (TP_a) in Equation (4) and substitute the T with SIMD unit max comparison steps S. G is the narrowing proportion each time

$$\frac{TP_a}{TP_b} \propto \frac{S_b * GS_b}{S_a * GS_a} = \frac{S_b}{S_a} * G. \tag{4}$$

The SIMD unit size is the multiple of 2, so the GS is generally reduced by 2 each time, which means we can consider G as a constant. Therefore, to find the appropriate narrow thread-group size, we only need to approximately check the change ratio of S after narrowing the thread group in practice and decide whether there is a performance gain based on Equation (4).

Because PSA increases the opportunity of queries sharing a traversal path, major query divergence happens at the last several levels tree traversal. So to decide the best NTG size, we only need to have some simple profiling to know the change ration of S for the last several levels after narrowing thread group. That data can be collected on CPU easily. Here we applied a static profiling method. Before processing the data, some data (for example, 1,000 queries) are used to collect the average SIMD unit execution steps for different NTG sizes. Then, the best NTG size is decided based on Equation (4). If its value is greater than 1, it means narrowing the thread group can further improve performance. This step is repeated until its value is smaller than 1. To verify the accuracy of this model, we collect the performance data of different NTG size for different tree fanouts including 8, 16, 32, 64 and 128 on different GPU (Tesla K80 and Volta TITAN V). Experimental results show the NTG size of this model is basically consistent with the NTG size of the best performance. For example, on Tesla K80, the NTG size for the best performance is 2 for 64-fanout B+tree, and the NTG size for the best performance is 4 threads for 128-fanout B+tree.

5 Implementation

We implement Harmonia B+tree structure with two optimizations (PSA and NTG), which can be used to further improve performance. HB+tree [1] is an optimized B+tree. In its design, a coarse-grained index structure is involved into the inner nodes to speed up traversal. In HB+tree, each inner node consists of a key region and an index region. For each inner node, the keys are partitioned into multiple sub-regions in sequence and there is an index in the index region corresponding to each sub-region. The index value is the maximal key value for its sub-region. When a query traverse a node, the operation first traverses the index area to determine which part of the key sub-region includes the target key value. And then, the keys in the corresponding sub-region are compared. Such a manner can achieve better traversal performance when tree fanout is large, such as 64-fanout or 128-fanout.

In our current implementation, we also use coarse-grained index structure, which is the same as that of HB+tree, to improve the performance of tree traversal. The index structure in each inner node is stored in the key region. We implement our system on both CPU through using its SIMD unit (Intel AVX [24]) and GPUs (Nvidia GPUs).

CPU Implementation. For CPU implementation, we use OpenMP [25] to exploit thread-level parallelism and use the SIMD units in each core to implement our design. Since Harmonia child information is much smaller than regular B+tree, it can also be saved on on-chip cache, which can achieve better cache locality on CPU platform.

For the overall search procedure, the queries are first sorted using radix sort from boost [26]. Then, the sorted queries are partitioned into sub-groups and these sub-groups are distributed to different OpenMP threads executed on different physical cores. For each sub-group, the queries in it are finally processed in a SIMD manner. They

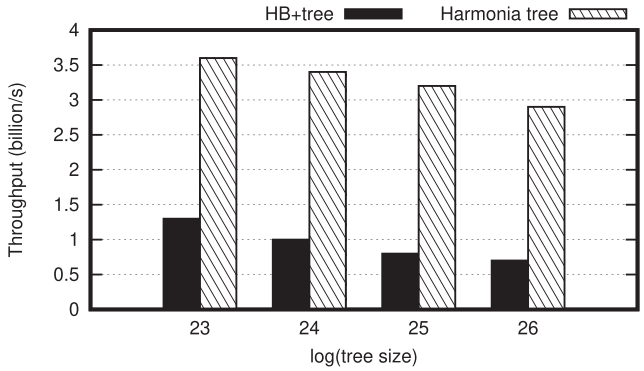


Fig. 14. GPU query performance.

are merged together based on the width of the SIMD instructions for the NTG optimization.

GPU Implementation. For GPU implementation, we put the whole tree structure on GPU. In a batch scenario, we implement the queries on GPU and updates on CPU, and B+tree will be synchronized between GPU and CPU after the update.

In our current design for GPU, the top level of the prefix-sum child array is stored in the constant memory,[2] and the rest is fetched into the read-only cache (texture cache) on each SM when they are used. The shared memory is not used to cache the prefix array because its size is not big enough for prefix-sum child array. Moreover, the prefix-sum array can be shared among different SMs when it is stored in texture cache. In this way, the prefix-sum array accesses will be more efficient than the child references of regular B+tree.

On GPU, we first use CUB [21] library to implement the partially sort for the PSA method. After that, we dispatch queries to the warps on GPU in order to process them in parallel. The number of queries processed by a warp is decided by the NTG optimization.

6 Evaluation

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

- Can Harmonia deliver better performance than the state-of-art GPU-based B+tree?
- Does Harmonia solve the issues discussed in Section 2.2?
- How does each design choice affect the performance?
- Can Harmonia achieve good update performance?
- Does Harmonia maintain a good performance in different situations?

6.1 Experimental Setup

We conduct all experiments on a 28-core server (Intel Xeon CPU E5-2680 v4 @ 2.40 GHz) with a Volta TITAN V GPU. Each CPU core has a private 32 KB L1 data cache, 32 KB L1 instruction cache, 256 KB L2 cache, and a shared 35 MB L3 cache. We use OpenMP 4.0 [25], Intel AVX2 to implement the CPU version of Harmonia. Harmonia implementation is compiled by GCC 5.4.0 and CUDA 10 on Ubuntu 16.04 (kernel 4.15.0) using O3 optimization option. We evaluate the performance of Harmonia using a throughput metric.

HB+Tree [1] is a state-of-the-art CPU-GPU hybrid B+tree. It supports search by using both CPU and GPU, and batch update on CPU. To compare the performance of different platforms, a CPU-based HB+tree and a GPU-based HB+tree are implemented. For GPU-based HB+tree, we put the whole tree structure on GPUs to make evaluation results fair and all the tree traversal of a query request is proposed on GPUs. We also compare the update performance with HB+Tree. Moreover, we also implement a parallel Regular B+tree [27] with Pthread [28] and OpenMP 4.0 [25] using the same thread-safe strategy as that of Harmonia for evaluation.

For overall performance experiment, in order to get stable performance, we use the data set which size is 100 million. These input data are queried on different tree sizes including 2^{23}, 2^{24}, 2^{25} and 2^{26} and the key size is 64 bits. For the scalability experiments, we evaluate different key sizes (32 bits versus 64 bits), different GPUs (Volta TITAN V versus Pascal TITAN XP) and different distributions (uniform, gamma, normal). All results are averaged by 5-time executions and the sorting time is included for Harmonia evaluation.

6.2 Overall Evaluation

In this section, we evaluate the performance of query, range query and update for Harmonia.

6.2.1 Query Performance

To see whether Harmonia can achieve a better performance, we conduct an experiment on Harmonia and HB+tree [1] on CPU and GPU, and an experiment on Regular B+tree on CPU.

As the data in Fig. 14 show, the performance of Harmonia can reach up to 3.6 billion queries per second. It outperforms HB+tree under different tree sizes on GPU. Its performance is about 3.4X faster than that of GPU-based HB +tree. As the data in Fig. 15 show, the CPU-Harmonia tree can reach up to 207 million queries per second which is about 1.7X faster than CPU-based HB+tree and 2.8X faster than CPU-based Regular B+tree.

To see whether Harmonia solves the gap issues discussed in Section 2.2, we first use nvprof [29] to collect the three metrics on GPU: the number of global memory transactions, memory divergence, and warp coherence[3] for HB+tree and Harmonia. As the data in Fig. 16 show Harmonia only issues 22 percent global memory transactions of HB+tree on GPU with 34 percent less memory divergence and 13 percent higher warp coherence (less warp divergence) than HB+tree on GPU.

We also use PAPI [17] to collect four performance metrics on CPU: L1 miss rate, L2 miss rate of L1, TLB miss rate and the number of branch per query of HB+tree and Harmonia. As shown in Fig. 17, Harmonia significantly reduces the cache misses and branches per query when compared to HB+tree. Harmonia only issues about 28 percent L1 misses,

2. The constant memory on GPU is read-only and faster than global memory, and it doesn't need to reload after current kernel finish, but it has a limit size (64 KB in Nvidia Volta) which is usually smaller than the prefix-sum child array.

3. Warp coherence metric means the proportion of the coherent step in the warp execution period. It is anti-correlation with warp divergence. We profile the metric *warp_execution_efficiency* as warp coherence, the metric *gld_transaction* as global memory transaction and the metric *gld_transaction_per_request* as memory divergence.

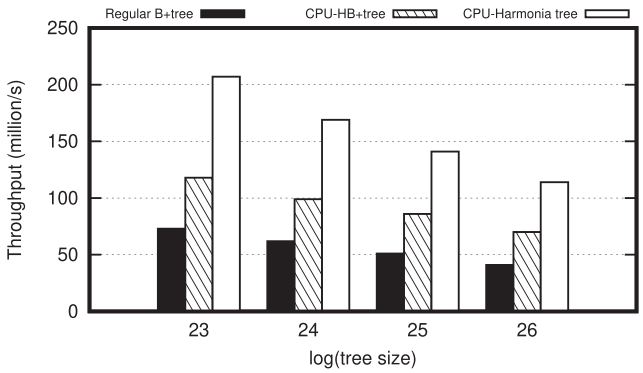


Fig. 15. CPU query performance.

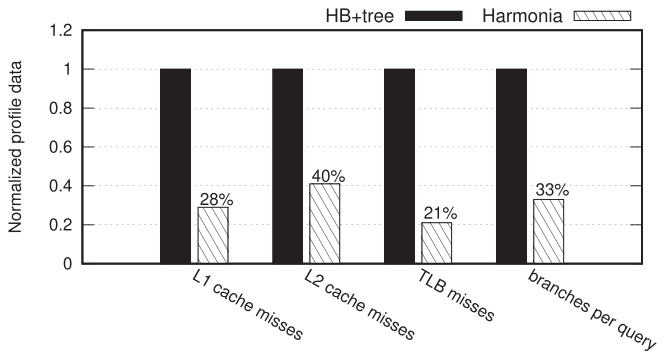


Fig. 17. CPU profile data normalized to those of HB+tree.

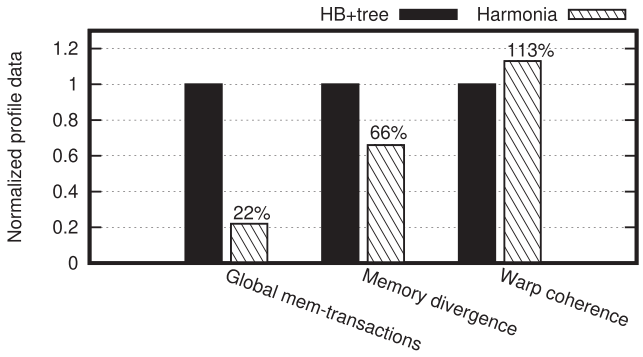


Fig. 16. GPU profile data normalized to those of HB+tree.

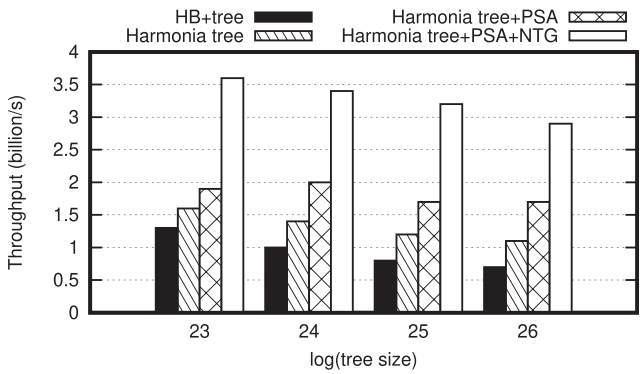


Fig. 18. Impact of different design choices on GPU.

40 percent L2 misses, 21 percent TLB misses and 33 percent branches per query of HB+tree.

The reasons for these results are because the size of prefix-sum child array is small and most of it can be stored in on-chip caches. Therefore, the global memory accesses can be significantly reduced. Moreover, as the Section 4.1 discussed, the design of PSA can reduce memory divergence and branch divergence because the adjacent sorted queries share more traversal paths, which brings a higher possibility of coalesced memory accesses. Therefore, Harmonia bridges the gaps between B+tree and SIMD architectures by effectively reducing the high latency of global memory transactions, memory divergence and warp divergence, which results in better performance than the state-of-art SIMD-based B+tree.

6.2.2 Impact of Different Design Choices

To understand the performance improvement from various factors, we evaluate different design choices using uniform distributions as input data set. The baseline refers to HB+tree. We evaluate the Harmonia B+tree structure (Harmonia tree), Harmonia B+tree structure with PSA, and the whole Harmonia (Harmonia tree + PSA + NTG). The GPU results and the CPU results are shown in Figs. 18 and 19 respectively.

As the data in Figs. 18 and 19 show, the three design choices respectively achieve about 1.4X, 1.9X and 3.4X speedup for GPU, and 1.1X, 1.5X and 1.7X speedup for CPU. Those results illustrate these design choices are efficient on both GPU and CPU. Applying the NTG method achieves much more performance improvement on GPU than that of CPU. The reason behind it is because traditional methods generally use the fanout number of threads to serve a query, which leads to insufficient resource utilization problem due to many unnecessary comparisons as analyzed in Section 4.2. The larger the PU number in a SIMD unit is, the more resource waste would be. In our current evaluation environments, there are 32 PUs (threads) in a GPU SIMT warp and 8 32-bit PUs in a CPU SIMD unit. The PU number in a GPU SIMT warp is about 4X compared to that in a CPU SIMD unit. Therefore, the resource utilization problem of GPU is much more serious than that of CPU in prior methods, which is also the reason why the performance improvement of CPU is not as dramatic as that of GPU.

6.2.3 Range Query Performance

To evaluate the range query performance of Harmonia, we also conduct an experiment. As the data in Figs. 20 and 21 show, the range query throughput of Harmonia can achieve about 1.1 billion per second on GPU and 90 million per second on CPU which is about 2.4X and 1.8X faster than GPU-based HB+tree on CPU-based HB+tree respectively. Generally, the range query performance is mainly depended on two parts: query operation and horizontal traversal in the leaf layer. In Harmonia, the query operation performance is improved by the tree structure and its two optimizations. The horizontal traversal process can also achieve a better access performance because the contiguous key layout in Harmonia.

6.2.4 Update Performance

To analyze the update performance, we evaluate the Harmonia update performance by comparing it with those of the Regular B+tree and HB+tree. We evaluate the update

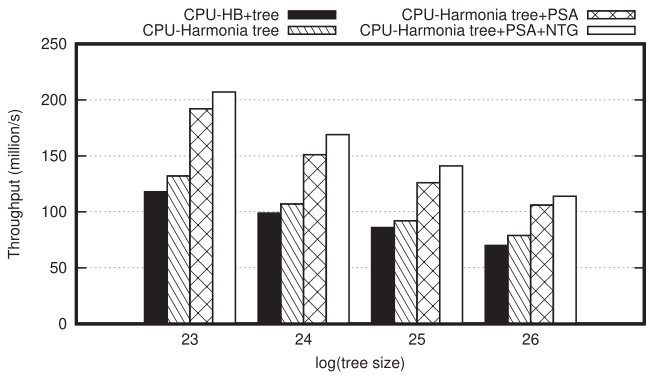


Fig. 19. Impact of different design choices on CPU.

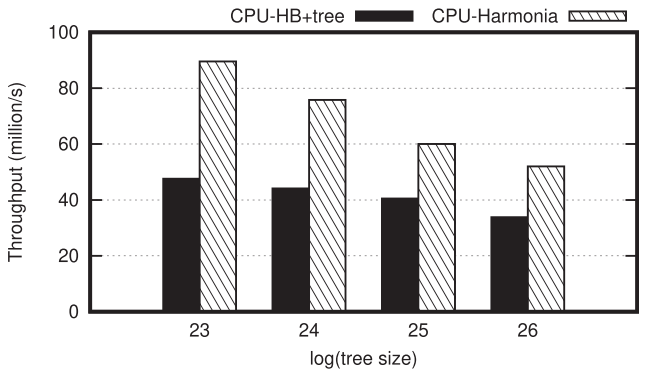


Fig. 21. Range query throughput on CPU.

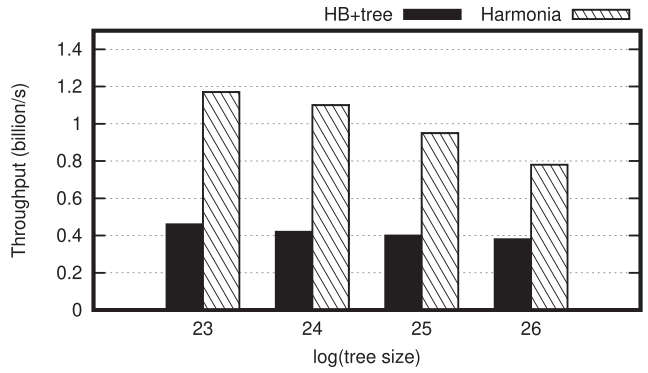


Fig. 20. Range query throughput on GPU.

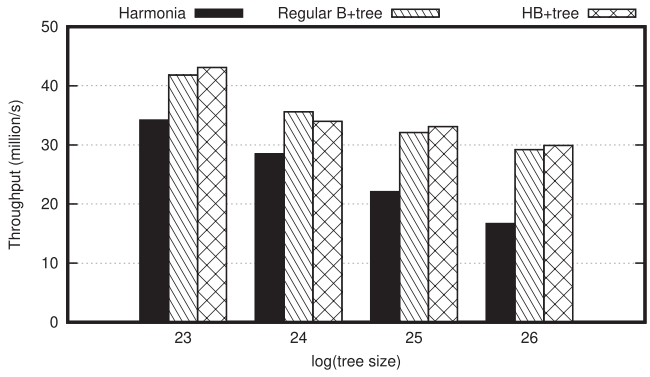


Fig. 22. Update throughput.

performance for different tree sizes and different update ratios. The update requests include update, insertion and deletion. The update ratio is the proportion of the update operations in these requests. For example, a 95 percent update ratio means that the update operations occupies 95 percent and the other parts (5 percent) are insertion and deletion. In the evaluation, the update ratios are from 20 to 95 percent, and insertion and deletion are equally proportional.

As the data in Fig. 22 show, for 95 percent update ratio, the update throughput performance of Harmonia can achieve 71 percent that of Regular B+tree and 70 percent of HB+ tree for different tree sizes on average. This is because the batch process can avoid many unnecessary data movements. We also test the throughput of different update ratios. As the data in Fig. 23 show, Harmonia has higher throughput than regular B+tree when update operation ratio is less than 70 percent, and the throughput of HB+tree is slightly higher than that of Harmonia. Since updates are relatively infrequent in the batch update scenario as described in [18], the performance of the proposed CPU-based batch update method is acceptable.

6.3 Scalability of Harmonia Design

The final performance of a B+tree can be influenced by various factors, including the different key sizes, the distribution of input data and the different hardware configurations. Therefore, we evaluate the scalability of Harmonia under different factors. Due to the space constraint, we only use the performance results of GPUs to illustrate the efficiency. We also collect the performance data on CPUs, they can get the same conclusions.

6.3.1 Performance of Different Key Sizes

To understand the performance impact of different key sizes, we conduct an experimental on 32-bit key and 64-bit key. As the data in Fig. 24 show, Harmonia can achieve better query performance than HB+tree, no matter which key size is. This is because Harmonia can take full use of the memory hierarchy and reduce the execution and memory divergence efficiently. For different key sizes, 32-bits keys can achieve a better performance than those of 64-bits keys. The reason behind it is that the smaller the key is, the more keys can be stored in the cache hierarchy, which can reduce the number of long latency memory accesses.

6.3.2 Performance of Different Distributions

In order to understand the impact of different input distributions, we use different data distributions as input data for tree queries, including uniform distribution, normal distribution ; ($\mu = 0.5, \sigma^2 = 0.125$) and gamma distribution ($\kappa = 3, \theta = 3$). As the data in Fig. 25 show, for these three distributions, the performance of Harmonia and HB+tree has the same trends. The performance data of GPU-based Harmonia are about 3.4X faster than those of GPU-based HB+tree.

6.3.3 Performance of Different GPUs

To see the performance impact of different GPUs, we conduct an experiment on NVIDIA Pascal TITAN XP and NVIDIA Volta TITAN V. As the data in Fig. 26 show, query throughput can reach up to 1.9 billion and 3.6 billion per second on NVIDIA Pascal TITAN XP and NVIDIA TITAN V respectively. Such results illustrate that the performance of

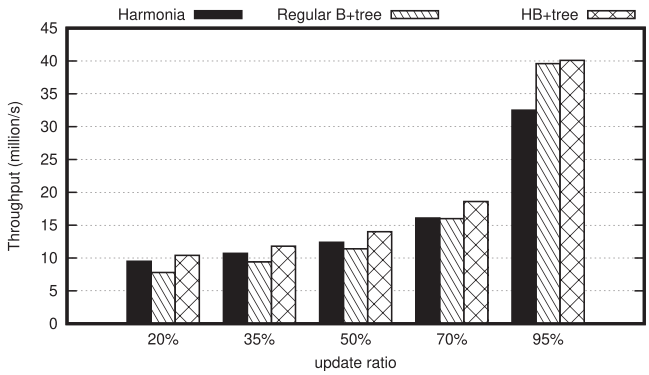


Fig. 23. Throughput of different update ratios.

Fig. 25. Performance for different distributions on Volta TITAN V.

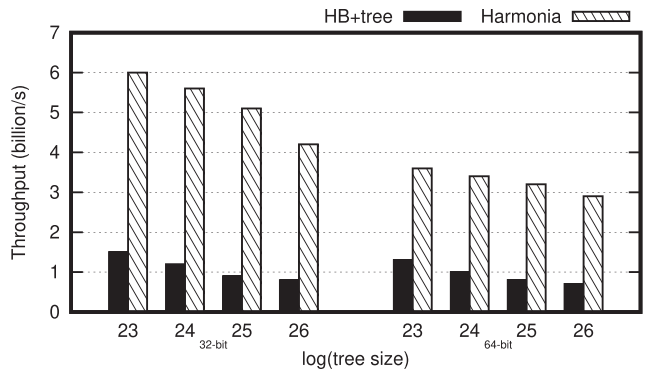


Fig. 24. Performance for different key sizes on GPU.

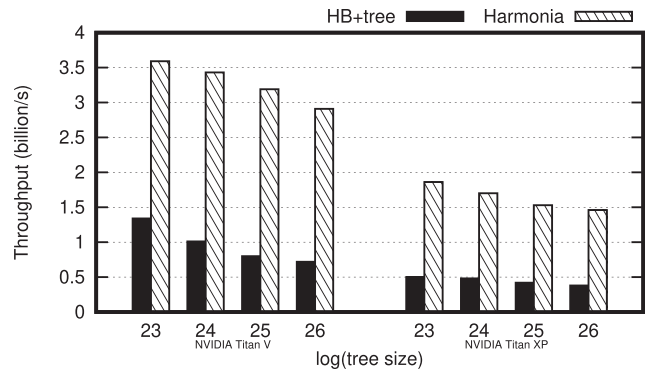


Fig. 26. Performance for different GPUs.

Harmonia scales well when the hardware platforms become more powerful (80 SMs on Volta TITAN V versus 30 SMs on Pascal TITAN XP).

7 Related Work

With the popularity of parallel hardware, such as multi-core CPUs and GPUs, there have been many efforts to accelerate B+tree search performance. Rao et al. propose a cache line conscious B+tree structure, called CSS-tree [30], to achieve better cache performance. CSS-tree is further extended to CSB+-tree [31] to provide an efficient update. Prior works [32], [33], [34] analyzed the influence of B+tree node size to search performance. They find the cache performance can be improved when the node size is larger than the cache line size. Kim et al. propose FAST [35], a configurable binary tree structure for multi-core systems. Its tree structure can be defined based on CPU cache-line size, memory page size and SIMD width. Besides, several works optimize the concurrent B+tree performance on distribution system aim to improve the concurrency and provide consistency [36], [37], [38].

GPUs have been widely used to improve application performance in different fields, such as matrix manipulation [39], [40], [41], [42], [43], [44], [45], [46], Stencil [47], [48], [49], [50], [51] and so on. To utilize the computation resources of GPUs, FAST [35] and HB+ tree [1] utilize the heterogeneous platform to search B+tree. HB+tree [1] also discusses several heterogeneous collaboration modes to make CPU and GPU cooperation more efficient such as CPU-GPU pipelining, double buffering. Kaczmarski [8], [9] proposes a GPU-based B+tree, which can update efficiently, and also discusses several methods for single-key search or batch search on GPU. Since GPU resides across the PCIe bus, Fix et al. [23] present a method that reorganizes the original B+tree of database into a continuous layout before uploading onto GPU, and search the B+tree using braided method parallelism. Daga et al. [10] accelerate B+tree on an APU to reduce the cost of transmission between GPU and CPU and overcome the irregular memory representation of the tree. Awad et al. [52] design a GPU B-Tree for batch update performance with a warp-cooperative work-sharing strategy. In contrast, we design a novel tree structure with two optimizations, which can bridge the gaps between B+tree and GPUs to achieve high query performance.

8 Conclusion

In this paper, through a comprehensive analysis of the characteristics of B+tree and SIMD architectures, we identify several gaps between B+tree and SIMD architectures, such as the gap in memory access requirements, memory divergence, and query divergence. Based on this observation, we proposed a novel B+tree structure called Harmonia. In Harmonia, the B+tree structure is divided into a key region and a prefix-sum child region. Due to the small size of prefix-sum array, the Harmonia B+tree structure can fully utilize the memory hierarchy to decrease the number of high latency memory accesses via cache accesses on chip. There are also two optimizations in Harmonia to alleviate the different divergences on SIMD architectures and improve the resource utilization: partially-sorted aggregation and narrowed thread-group. As a result, Harmonia performs average 1.7X speedup to CPU-based HB+tree and 3.4X speedup to GPU-based HB+tree.

ACKNOWLEDGMENTS

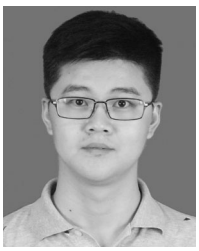
We are very grateful to the anonymous reviewers for their valuable feedback and comments. This work is supported in part by the National Natural Science Foundation of China (No. 61672160), Shanghai Municipal Science and Technology Major Project (No.2018SHZDZX01) and ZJLab, Shanghai Technology Development and Entrepreneurship Platform for Neuromorphic and AI SoC.

REFERENCES

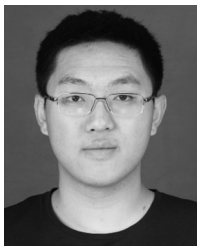
[1] A. Shahvarani and H.-A. Jacobsen, "A hybrid B+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1523–1538.
[2] D. Comer, "Ubiquitous B-tree," *ACM Comput. Surveys*, vol. 11, no. 2, pp. 121–137, 1979.
[3] V. Srinivasan and M. J. Carey, "Performance of B+ tree concurrency control algorithms," *VLDB J.*, vol. 2, no. 4, pp. 361–406, 1993.
[4] P. Kieseberg, S. Schrittwieser, L. Morgan, M. Mulazzani, M. Huber, and E. Weippl, "Using the structure of B+-trees for enhancing logging mechanisms of databases," *Int. J. Web Inf. Syst.*, vol. 9, no. 1, pp. 53–68, 2013.
[5] R. C. Teresa Lam and C. Li, "Sales performance of Alibaba in 2017 single's day," 2017. [Online]. Available: https://www.fbicgroup.com/sites/default/files/CREQ_04.pdf
[6] My data is bigger than your data! 2018. [Online]. Available: https://lintool.github.io/my-data-is-bigger-than-your-data
[7] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proc. 3rd Workshop Gen.-Purpose Comput. Graph. Process. Units*, 2010, pp. 94–103.
[8] K. Kaczmarski, "Experimental B+-tree for GPU," in *Proc. East-Eur. Conf. Advances Databases Inf. Syst.*, 2011, pp. 232–241.
[9] K. Kaczmarski, "B+-tree optimized for GPGPU," in *Proc. OTM Confederated Int. Conf.*, 2012, pp. 843–854.
[10] M. Daga and M. Nutter, "Exploiting coarse-grained parallelism in B+ tree searches on an APU," in *Proc. SC Companion: High Perform. Comput. Netw. Storage Anal.*, Nov. 2012, pp. 240–247.
[11] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*, W. Zhang, Ed. Hoboken, NJ, USA: Wiley, 2014.
[12] *CUDA C Programming Guide*, 2018. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
[13] J. I. Munro and H. Suwanda, "Implicit data structures for fast search and update," *J. Comput. Syst. Sci.*, vol. 21, no. 2, pp. 236–250, 1980.
[14] G. Graefe, et al., "Modern B-tree techniques," *Found. Trends® Databases*, vol. 3, no. 4, pp. 203–402, 2011.
[15] A. Vaisman and E. Zimányi, "Data warehouses: Next challenges," in *Proc. Eur. Bus. Intell. Summer School*, 2011, pp. 1–26.
[16] P. Vassiliadis and A. Simitsis, "Near real time ETL," in *New Trends in Data Warehousing and Data Analysis*. Berlin, Germany: Springer, 2009, pp. 1–31.
[17] S. Browne, J. J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, pp. 189–204, 2000.
[18] Microsoft Redmond and Microsoft Research Cambridge, "DBMS workloads in online services," 2009. [Online]. Available: http://www.tpc.org/tpctc/tpctc2009/tpctc2009-10.pdf
[19] K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylonen, "Concurrency control in B-trees with batch updates," *IEEE Trans. Knowl. Data Eng.*, vol. 8, no. 6, pp. 975–984, Dec. 1996.
[20] B. Shneiderman, "Batched searching of sequential and tree structured files," *ACM Trans. Database Syst.*, vol. 1, no. 3, pp. 268–275, 1976.
[21] D. Merrill, NVIDIA Research Group, "CUB Documentation," 2018. [Online]. Available: https://nvlabs.github.io/cub/index.html#sec9
[22] E. Stehle and H.-A. Jacobsen, "A memory bandwidth-efficient hybrid radix sort on GPUs," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 417–432.
[23] J. Fix, A. Wilkes, and K. Skadron, "Accelerating braided B+ tree searches on a GPU with CUDA," in *Proc. 2nd Workshop Appl. Multi Many Core Processors: Anal. Implementation Perform.*, 2011, pp. 1–11.
[24] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel AVX: New frontiers in performance improvements and energy efficiency," *Intel White Paper*, vol. 19, 2008, Art. no. 20.
[25] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan.–Mar. 1998.
[26] B. Schäling, *The Boost C++ Libraries*, XML Press; 2nd ed. (Sep. 22, 2014), 2011.
[27] A. Aviram, "B+ tree implementation," 2018. [Online]. Available: http://www.amittai.com/prose/bpt.c
[28] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming: The Sockets Networking API*, vol. 1. Reading, MA, USA: Addison-Wesley, 2004.
[29] *Profile User's Guide*, 2018. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html
[30] J. Rao and K. A. Ross, "Cache conscious indexing for decision-support in main memory," in *Proc. 25th Int. Conf. Very Large Data Bases*, 1999, pp. 78–89.
[31] J. Rao and K. A. Ross, "Making B+-trees cache conscious in main memory," *ACM SIGMOD Rec.*, vol. 29, no. 2, pp. 475–486, 2000.
[32] R. A. Hankins and J. M. Patel, "Effect of node size on the performance of cache-conscious B+-trees," *ACM SIGMETRICS Perform. Evaluation Rev.*, vol. 31, no. 1, pp. 283–294, 2003.
[33] S. Chen, P. B. Gibbons, and T. C. Mowry, "Improving index performance through prefetching," *ACM SIGMOD Rec.*, vol. 30, no. 2, pp. 235–246, 2001.
[34] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin, "Fractal prefetching B+-trees: Optimizing both cache and disk performance," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2002, pp. 157–168.
[35] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "FAST : Fast architecture sensitive tree search on modern CPUs and GPUs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 339–350.
[36] X. Wang, W. Zhang, Z. Wang, Z. Wei, H. Chen, and W. Zhao, "Eunomia: Scaling concurrent search trees under contention using HTM," in *Proc. 22nd ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2017, pp. 385–399.
[37] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, "Fast and general distributed transactions using RDMA and HTM," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 26.
[38] W. Zhang, X. Wang, S. Ji, Z. Wei, Z. Wang, and H. Chen, "Eunomia: Scaling concurrent index structures under contention using HTM," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 8, pp. 1837–1850, Aug. 2018.
[39] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on GPUs," in *Proc. 29th ACM Int. Conf. Supercomput.*, 2015, pp. 99–108.
[40] N. Sedaghati, A. Ashari, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Characterizing dataset dependence for sparse matrix-vector multiplication on GPUs," in *Proc. 2nd Workshop Parallel Program. Analytics Appl.*, 2015, pp. 17–24.
[41] J. Li, X. Li, G. Tan, M. Chen, and N. Sun, "An optimized large-scale hybrid DGEMM design for CPUs and ATI GPUs," in *Proc. 26th ACM Int. Conf. Supercomput.*, 2012, pp. 377–386.
[42] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast implementation of DGEMM on Fermi GPU," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 35.
[43] Y. Nagasaka, A. Nukada, and S. Matsuoka, "Cache-aware sparse matrix formats for Kepler GPU," in *Proc. 20th IEEE Int. Conf. Parallel Distrib. Syst.*, 2014, pp. 281–288.
[44] Y. Nagasaka, A. Nukada, and S. Matsuoka, "High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA pascal GPU," in *Proc. 46th Int. Conf. Parallel Process.*, 2017, pp. 101–110.
[45] Y. Nagasaka, A. Nukada, and S. Matsuoka, "Adaptive multi-level blocking optimization for sparse matrix vector multiplication on GPU," *Procedia Comput. Sci.*, vol. 80, pp. 131–142, 2016.
[46] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs for graph applications," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 781–792.
[47] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *Proc. 26th ACM Int. Conf. Supercomput.*, 2012, pp. 311–320.
[48] T. Endo, Y. Takasaki, and S. Matsuoka, "Realizing extremely large-scale stencil applications on GPU supercomputers," in *Proc. IEEE 21st Int. Conf. Parallel Distrib. Syst.*, 2015, pp. 625–632.
[49] G. Jin, T. Endo, and S. Matsuoka, "A parallel optimization method for stencil computation on the domain that is bigger than memory capacity of GPUs," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2013, pp. 1–8.

[50] G. Jin, T. Endo, and S. Matsuoka, "A multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of GPU," in *Proc. IEEE 27th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, 2013, pp. 1080–1087.
[51] P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Register optimizations for stencils on GPUs," in *Proc. 23rd ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2018, pp. 168–182.
[52] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens, "Engineering a high-performance GPU B-tree," in *Proc. 24th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, Feb. 2019, pp. 145–157.

Weihua Zhang received the PhD degree in computer science from Fudan University, in 2007. He is currently a professor of Parallel Processing Institute, Fudan University. His research interests include compilers, computer architecture, parallelization, and systems software.



Zhaofeng Yan is now working toward the graduate degree in the Software School, Fudan University and working in the Parallel Processing Institute. His work is related to parallel processing, GPU computing, transaction memory, systems software and so on.



Yuzhe Lin is now working toward the graduate degree in the Software School, Fudan University and working in the Parallel Processing Institute. His work is related to parallel processing, transactional memory, GPU computing and so on.

Chuanlei Zhao is now working toward the undergraduate degree in the Software School, Fudan University and working in the Parallel Processing Institute. His work is related to parallel processing, GPU computing, system performance improvement and so on.

Lu Peng received the PhD degree in computer engineering from the University of Florida, in 2005 and joined the Electrical and Computer Engineering Department, Louisiana State University where he is currently the Gerard L. Jerry Rispone professor. His research focuses on computer architecture, reliability, and big data analytics etc. He received an ORAU Ralph E. Powe Junior Faculty Enhancement Award in 2007. He is a senior member of the IEEE and ACM.