Harmonia: A High Throughput B+tree for GPUs

Zhaofeng Yan
Software School, Fudan University
Shanghai Key Laboratory of Data Science, Fudan University
Shanghai Institute of Intelligent Electronics & Systems, Shanghai
zfyan16@fudan.edu.cn

Yuzhe Lin
Software School, Fudan University
Shanghai Key Laboratory of Data Science, Fudan University
yzlin14@fudan.edu.cn

Lu Peng
Division of Electrical and Computer Engineering
Louisiana State University
lpeng@lsu.edu

Weihua Zhang
Software School, Fudan University
Shanghai Key Laboratory of Data Science, Fudan University
zhangweihua@fudan.edu.cn

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, GPUs 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 GPUs. Concurrent B+tree queries involve many global memory accesses and different divergences, which mismatch with GPU features. Based on this observation, we propose Harmonia, a novel B+tree structure to bridge the gap. 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 warp divergence and improve resource utilization. Evaluations on a TITAN V GPU show that Harmonia can achieve up to 3.6 billion queries per second, which is about 3.4X faster than that of HB+Tree [39], a recent state-of-the-art GPU solution.

CCS Concepts • **Information systems → Data structures**; • **Computer systems organization → Parallel architectures**.

Keywords GPU, B+tree, High-throughput

1 Introduction

B+tree [10] 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 [23, 41]. 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 [44]. 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 system[2]. All these factors put tremendous pressures on applications which use B+tree as index data structure.

Graphics Processing Units (GPUs) have become one of the most popular many-core processors. Due to abundant computation resources [26], GPUs have occupied about 80% of the accelerator market share in the high-performance computing (HPC) market [27], and have been widely used in cloud computing environments. They also provide a potential opportunity to accelerate query throughput of B+tree. Many previous works [6, 11, 21, 22, 39] have used GPUs 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 GPUs, and identify several gaps between the

PPoPP '19, February 16–20, 2019, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295704

characteristics of B+tree and the features of GPUs. 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 GPU warp simultaneously. All these characteristics of B+tree are mismatched with the features of GPUs, which impede the query performance of B+tree on GPUs.

Based on this observation, we propose Harmonia, a novel B+tree structure, to bridge the gaps between B+tree and GPUs. 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 number and the node size. With this compression, most of the prefix-sum array can be stored in GPU caches. Therefore, such a design matches GPU memory hierarchy 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 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 warp. 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 warp, which may increase warp divergence. To mitigate the warp divergence problem brought by query combinations, we design a model to decide how to narrow the thread group for a query.

Experimental results show Harmonia can achieve a throughput of up to 3.6 billion queries per second on a TITAN V GPU, which is about 3.4X higher than that of HB+tree [39], a recent state-of-the-art GPU solution. The main contributions of our work can be summarized as follows:

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

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 shows the experimental results. Section 6 surveys the related work. Section 7 concludes the work.

2 Background and Motivation

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

2.1 General-Purpose GPUs

Graphics Processing Units (GPUs) are one of the most popular many-core processors and have been widely used in different fields, such as HPC and big data processing. A typical GPU architecture is shown in Figure 1. There are multiple stream multiprocessors (SMs) in a GPU. Each SM has multiple CUDA cores with software configurable shared memory (L1 cache) and read-only data cache. GPUs also have several memory hierarchy layers shared by all SMs including L2 cache, constant memory, texture memory, and off-chip global memory. Among them, global memory is the slowest one, which generally requires several hundred cycles to access.

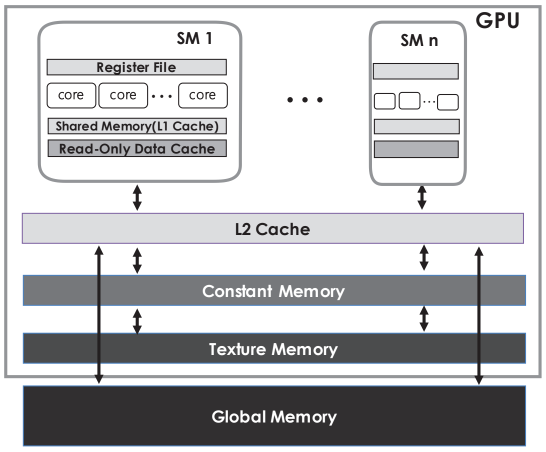


Figure 1. An overview of Nvidia GPU Architecture

To utilize the computation resources in an SM, multiple consecutive threads, e.g., 32 threads, are organized into a warp. Each thread is executed on a CUDA core and the threads in a warp are executed in a single instruction multiple thread (SIMT) manner. Multiple warps compose a thread block that can be dispatched to a specific SM. The warps on the same SM are scheduled to hide long memory access latency. The thread blocks further constitute a GPU kernel, which is a parallel function that can be invoked by the programmer and executed on all the SMs in a GPU.

Since the instructions are issued and executed at a warp granularity, executing different instructions among the threads in a warp will bring warp divergence [1]. Moreover, memory operations are also issued per warp. If a batch of memory addresses requested by a warp fall within one GPU cache line, which is called coalesced memory access [1, 20], 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 [11, 47].

GPUs provide powerful computation resources and high memory bandwidth. To fully utilize them, an application should have the following characteristics.

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

Avoiding warp divergence. When warp divergence happens, the threads in a warp execute different instructions. Since the threads in a warp are executed in the SIMT mode, they have to be partitioned into several sub-groups based on the instructions. When the threads in a sub-group are executed, the other threads have to wait, which leads to low resource utilization.

Avoiding memory divergence. Memory divergence leads to multiple memory transactions which impose long delays on GPU applications. Therefore, avoiding memory divergence is important for GPU performance.

2.2 Gaps between GPUs and B+tree

B+tree is a self-balanced tree [10] 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 [29]. 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 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 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. [15, 39, 45, 46], 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 that how to further improve B+tree query performance.

It seems that GPU is a potential solution to accelerating search performance of B+tree due to its powerful computation resources. However, prior GPU-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 GPUs.

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. However, a large number of global memory accesses in tree traversal would lead to poor memory performance on GPUs.

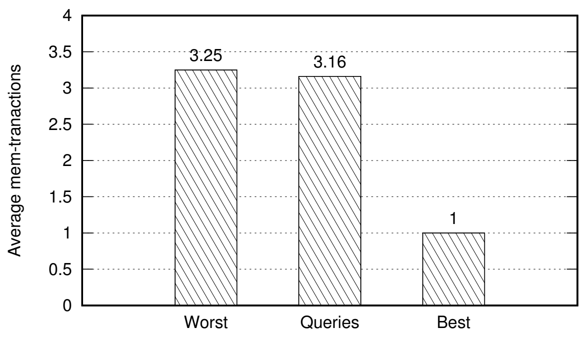


Figure 2. Average memory transactions per warp

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 GPU warp, the memory accesses are disordered, which would lead to memory divergence and greatly impede the GPU performance. To illustrate it, we collect the average number of memory transactions for each warp when concurrent queries traverse B+tree. For a height-4 and fanout-8 B+tree, each warp processes 4 queries concurrently and the input query data are randomly generated based on uniform distribution. As shown in Figure 2, the average number of memory transactions for each warp (illustrated in the second bar) is 3.16, which is about 97% of the worst case (3.25) shown in the first bar of Figure 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 GPU-based 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 comparison numbers of 100 queries in each level based on the above experiment setting and computed the average number, the largest one and the smallest one. As shown in Figure 3, the comparison numbers of each level for different queries have a large fluctuation although average comparison number is close to 4. Therefore, processing these queries simultaneously in a warp would lead to warp divergence and memory divergence.

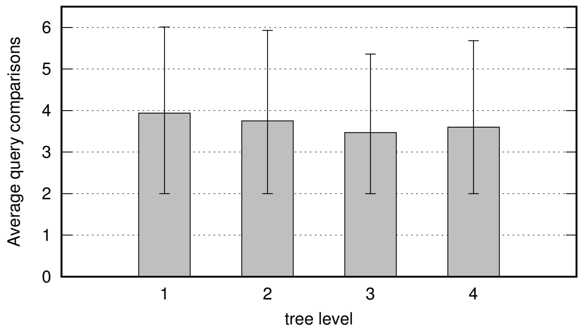


Figure 3. Query divergence

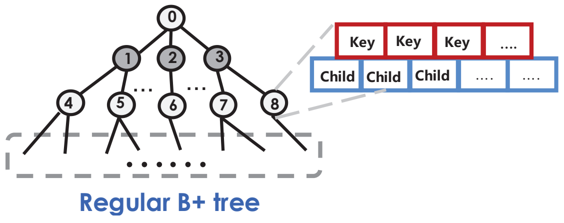
3 Harmonia Tree Structure

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

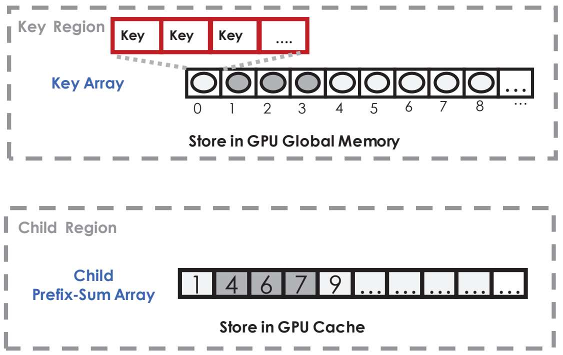
3.1 Tree Structure Overview

In a traditional regular B+tree structure, a tree node consists of keys and child references as shown in Figure 4(a). 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 1KB for a 64-fanout tree. Since the target of each query is random, it is difficult to utilize the GPU 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 on GPUs.

To overcome these constraints and fit GPU 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



(a) Regular B+tree structure



(b) Harmonia tree structure

Figure 4. Regular B+tree and Harmonia B+tree

region is organized in node granularity and the size of each item (a node) is fixed ((fanout-1)*key size). The child region is organized as a prefix-sum array. Each item in the array is the node's first child index 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 Figure 4(a) is [*1*, *4*, *6*, *7*, *9*...]. 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 children number 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 16KB. Therefore, most of the prefix-sum child array, even a very large B+tree, can be saved in low-latency on-chip caches in GPU memory hierarchy, such as constant memory, which can improve memory locality.

In our current design, the top level of the prefix-sum child array is stored in the constant memory [1], and the rest is fetched into the read-only cache on each SM when they are used. In this way, the prefix-sum array accesses will be more efficient than the child references of regular B+tree.

[1]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 (64KB in Nvidia Kepler) which is usually smaller than the prefix-sum child array.

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 [28] and can be deferred [33, 40]. For example, it was reported that there is a high read/write ratio (about 35:1) in TPC-H [28]. Therefore, in Harmonia's query phase, the GPU is used to accelerate query performance. In the update phase, batched updates are processed on CPUs; The B+tree on the GPU is synchronized after the updates.

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, 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.

$$child\_idx = PrefixSum[node\_idx] + i - 1 \tag{1}$$

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 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.

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
```

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

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 update without split or 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 operation leads to a split or 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. When an insert causes one node to split, an auxiliary node is created and the node status is marked as split. The auxiliary node contains the entire child references, and the split is processed on the auxiliary node. During the period of batch update, we need to first check whether or not the leaf node status is split for an insert. If it is, a new insert or update will use the information of its auxiliary node. Otherwise, it will use the original node in the key region.

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 shows in Section 5 (Figure 14).

4 Harmonia Optimizations

To reduce divergences and improve computation resource utilization on GPUs, 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 on a GPU, 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 warp, it is difficult to achieve a coalesced memory access because they would traverse the tree along different paths. Figure 5 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 warp, there is no coalesced memory access after they leave the root node, as shown in Figure 6(a). Therefore, processing

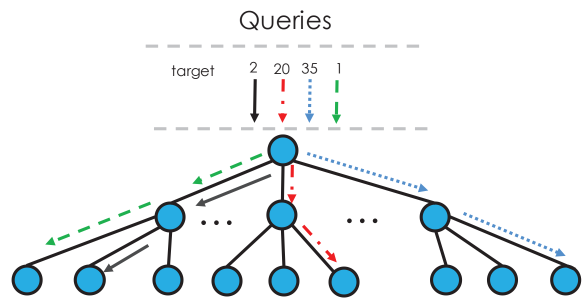


Figure 5. B+tree

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

4.1.1 Sorted Aggregation

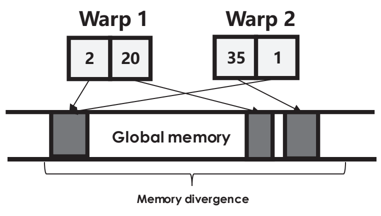
If multiple queries have shared part of the traversal path, the memory accesses can be coalesced when they are processed in a warp. For instance, if the queries with target 1 and target 2 in Figure 5 are processed in a warp, there are coalesced memory accesses for their first two level traversals.

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 Figure 5, the query target sequence becomes 1, 2, 20, and 35 after sorting as Figure 7 shows. When two adjacent queries are combined into a warp, the warp with the first two queries will have coalesced memory accesses for their shared traversal path as shown in Figure 6(b). Moreover, because the queries in the same warp will go through the same path, it can also mitigate warp divergence among them.

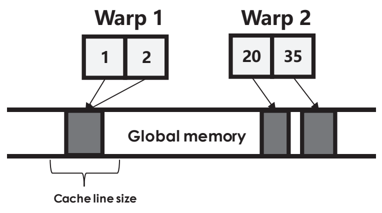
Although sorting queries can reduce memory divergence and warp divergence, it brings additional sorting overhead. To illustrate this problem, we evaluate the overhead using GPU radix sort [12] to make a batch of queries sorted before assigning them to the B+tree concurrent search kernel. As the data in Figure 8 show, the kernel performance has about 22% improvement compared with that of the original one. However, there is about 7% performance slowdown for the total execution time. The reason behind this is that complete sorting will generate more than 25% overhead.

4.1.2 Partially-Sorted Aggregation

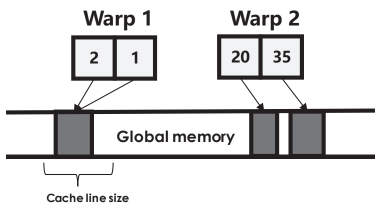
To achieve a coalesced memory access, multiple memory accesses in a warp only need to fall into the address space of a cache line even they are unordered. As shown in Figure 6(c), although the query to target 2 is before the query to target 1, we can still achieve coalesced memory accesses for their shared path, which has the same effect with that of the completely sorted queries as shown in Figure 6(b). Therefore,



(a) Queries



(b) Sorted queries



(c) Partially-sorted queries

Figure 6. An example of memory access pattern for queries.

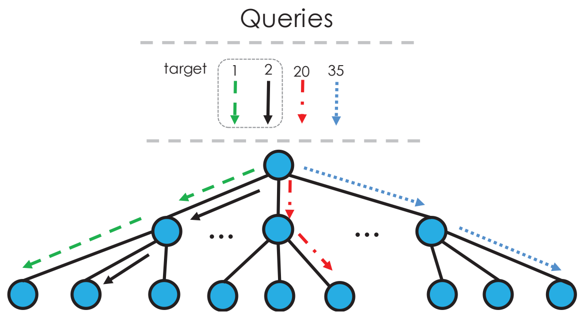


Figure 7. Queries share traversal path

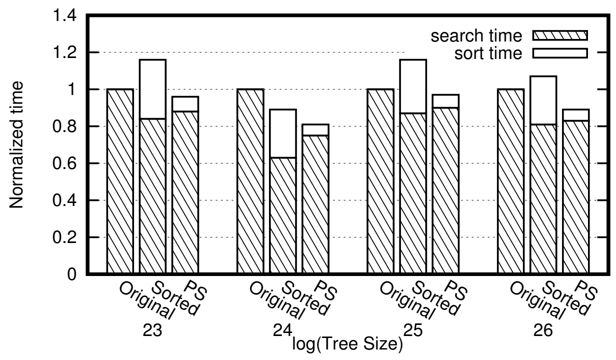


Figure 8. Sorted queries (sorted) and partially-sorted queries (PS) search time normalized to the search time of original queries (original)

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 the most commonly used algorithms on GPUs because they can provide stable performance for a large workload [12, 42]. For these bit-wise sorting algorithms, the execution time is proportional to the sorted bits. Therefore, partial sorting can also be used to reduce the sorting overhead. As the data in the third bar of Figure 8 shows, 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% improvement compared with that of the original one.

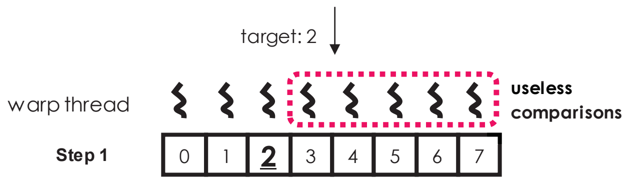
For a partial sorting, the queries will be sorted based on their most significant N bits. If N is large, there is a high probability that the targets of sorted queries are closer. However, it will lead to a higher sorting overhead. Here, we discuss how to decide the PSA size to achieve a better trade-off 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 average. If the memory requests of multiple queries in a warp 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.

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

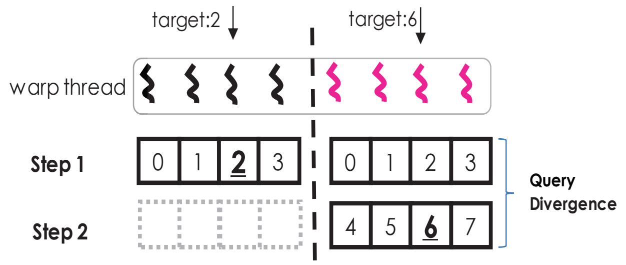
As an example, suppose the key is represented by 64 bits ($B = 64$), the tree size is 2^{23} ($T = 2^{23}$) and the size of a cache line is 128-byte, which can store 16 keys ($K = 16$). Based on Equation 2, the value of N equals to 19. To verify its efficiency, we collect average memory transactions per warp for different partially-sorted bits and the normalized sorting time for completely sorting. Experimental results show only sorting 19 bits can achieve the similar optimization effort as that of completely sorted. Moreover, its overhead is about 35% 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 [14, 21, 22, 39] generally use the fanout number of threads to serve a query[2]. Based on our observation, it has insufficient resource utilization problem due to many unnecessary comparisons. When a query traverse 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. Figure 9(a) shows an example. Suppose the tree fanout is 8 and the GPU warp size is also 8. The fanout-based thread group will use the whole warp to serve a query. So for the query which target is 2, only the first 3 threads make the useful comparisons, and the rest of comparisons are useless.



(a) Fanout-based thread group



(b) Narrowed thread group

Figure 9. Example of different thread groups.

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 Figure 10 show, for different tree fanouts, about 80% of queries can find the target child through searching the front 50% 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.

[2]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 GPU warp size, all threads in a warp are used for a query.

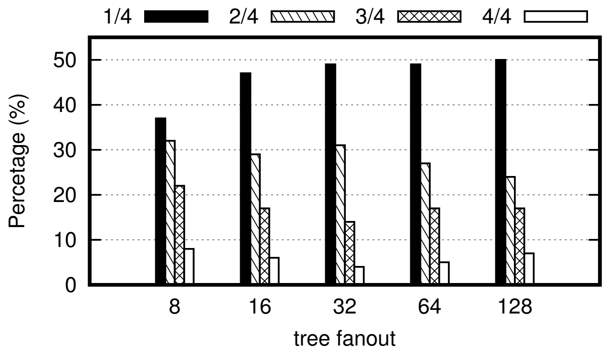


Figure 10. The proportion of queries accessing the different node parts.

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 warp. Due to the query divergence discussed in Section 2.2, the warp'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. Figure 9(b) 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 warp, the threads for target 2 has to execute two steps due to the query divergence brought by target 6, although it only need 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 warp equals to $warpsize/GS$ ($warpsize$ is a fixed number). And the throughput (TP) of a warp for one tree level equals the number of queries per warp divided by warp execution time T, which is shown in Equation 3.

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

The warp execution time T is composed of two parts of time: comparison time and memory access time. Because GPU 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 warp throughput before narrowing (TP_b) and after narrowing (TP_a) in Equation 4 and substitute the T with warp 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 warp 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 on GPUs, some data (for example, 1000 queries) are used to collect the average warp 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 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 Evaluation

We implement Harmonia in CUDA and C++. To evaluate the prototype performance, we 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?

5.1 Experimental Setup

We conduct all experiments on a 28-core server (Intel Xeon CPU E5-2680 v4 @ 2.40GHz) with a TITAN V GPU. Each CPU core has a private 32KB L1 data cache, 32KB L1 instruction cache, 256KB L2 cache, and a shared 35MB L3 cache. 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 [39] 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. For search performance evaluation, we compare Harmonia with the GPU part of HB+Tree. For update evaluation, we compare Harmonia with HB+tree.

For the input data sets in search evaluation, we choose the most commonly used distributions in prior B+tree evaluations [24, 39] (uniform distribution). The size of each data set is 100 million in order to get stable performance. These input data are queried on different tree sizes [3] including 2^{23}, 2^{24}, 2^{25} and 2^{26}. For update evaluation, we do the evaluation using a data set mixed by 5% inserts and 95% updates with a batch size of 4096K.

5.2 Overall Evaluation

To see whether Harmonia can achieve better performance, we conduct an experiment on Harmonia and the GPU part of HB+tree [39].

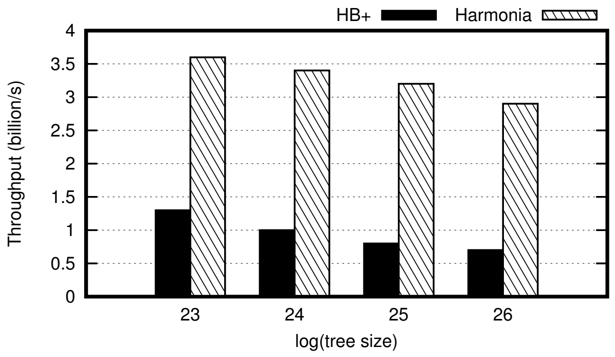


Figure 11. Overall query performance comparison.

As the data in Figure 11 show, the performance of Harmonia can reach up to 3.6 billion queries per second. It outperforms HB+tree under different tree sizes and input distributions. Its performance is about 3.4X faster than that of HB+tree. The primary reason behind the performance improvement is that the design of Harmonia bridges the gaps between traditional B+tree and GPUs well, which can reduce global memory transactions and divergences. Moreover, it

[3] the number of key-value pairs are inserted into the B+tree, and each key is 64 bits.

can also take full utilization of GPU computation resources and memory hierarchy.

To see whether Harmonia solves the gap issues discussed in Section 2.2, we use nvprof [3] to collect the three metrics: the number of global memory transactions, memory divergence, and warp coherence[4] for HB+tree and Harmonia. The profile data are shown in Figure 12.

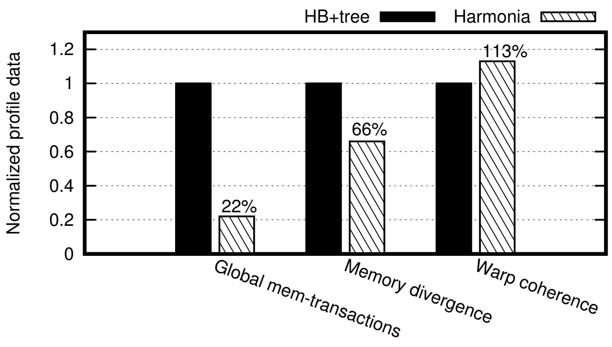


Figure 12. Profile data normalized to those of HB+tree

As the data (global memory-transactions) in Figure 12 show, Harmonia only issues 22% global memory transactions of HB+tree. This is because the size of prefix-sum child array is small and most of it can be stored in on-chip GPU caches. Therefore, the global memory transactions can be significantly reduced. As the data (memory divergence and warp coherence) in Figure 12 show, the memory divergence of Harmonia is 34% less than that of HB+tree, and Harmonia has 13% higher warp coherence (less warp divergence) than HB+tree. Just as the Section 4.1 discussed, the design of PSA can reduce memory divergence and warp divergence because the adjacent sorted queries share more traversal paths, which brings a higher possibility of coalesced memory accesses.

According to the profile data, Harmonia bridges the gaps between B+tree and GPUs 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 GPU-based B+tree.

5.3 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 results are shown in Figure 13.

First, the Harmonia tree structure gets about 1.4X speedup due to better memory locality and fewer global memory transactions. Second, only applied PSA can get about 2X speedup because of the reduction of warp divergence and memory divergence. Third, after PSA and NTG are applied, Harmonia gets about 3.4X speedup because NTG reduces the useless comparisons and takes full advantage of the computation resources of GPUs.

[4]Warp coherence metric means the proportion of the coherent step in the warp execution period. It is anti-correlation with warp divergence.

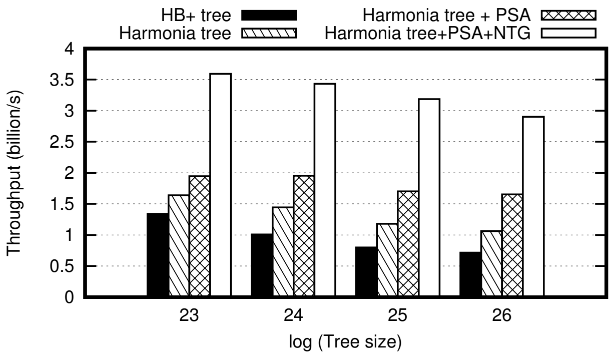


Figure 13. Impact of different design choices

5.4 Update Performance

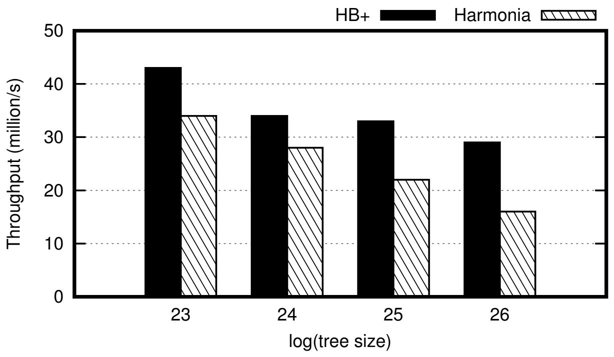


Figure 14. Update throughput

To analyze the update performance, we evaluate the Harmonia update performance by comparing it with the HB+tree. As the data in Figure 14 show, the update throughput performance of Harmonia can achieve average 70% of HB+ tree. This is because the batch process can avoid many unnecessary data movements. Since updates are relatively infrequent in the batch update scenario as described in [28], the performance of the proposed CPU-based batch update method is acceptable.

6 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 [34], to achieve better cache performance. CSS-tree is further extended to CSB+-tree [35] to provide an efficient update. Prior works [7, 8, 16] 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 [24], 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 [9, 48, 49].

GPUs have been widely used to improve application performance in different fields, such as matrix manipulation [4, 25, 30–32, 37, 38, 43], Stencil [13, 17–19, 36] and so on. To utilize the computation resources of GPUs, FAST [24] and HB+ tree [39] utilize the heterogeneous platform to search B+tree. HB+tree [39] also discusses several heterogeneous collaboration modes to make CPU and GPU cooperation more efficient such as CPU-GPU pipelining, double buffering. Kaczmarski [21, 22] 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. [14] 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. [11] 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. [5] 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.

7 Conclusion

In this paper, through a comprehensive analysis of the characteristics of B+tree and GPUs, we identify several gaps between B+tree and GPUs, 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 GPU memory hierarchy to decrease the number of high latency memory accesses via cache accesses on GPU chip. There are also two optimizations in Harmonia to alleviate the different divergences on GPUs and improve the resource utilization: partially-sorted aggregation and narrowed thread-group. As a result, Harmonia performs average 3.4X speedup to HB+tree, a state-of-art GPU-based B+tree.

Acknowledgments

We thank our shepherd, Eshcar Hillel, and the anonymous reviewers for their constructive comments. We also appreciate Travis D LeCompte for suggestions to improve the paper. We are grateful to support from the National Key Research and Development Program of China (No. 2017YFB0202105), the National Natural Science Foundation of China (No. 61672160) and Shanghai Science and Technology Development Funds (17511102200).

References

[1] 2018. *CUDA C Programming Guide*. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[2] 2018. My data is bigger than your data! https://lintool.github.io/my-data-is-bigger-than-your-data

[3] 2018. *Profile User's Guide*. https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[4] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. IEEE Press, 781–792.

[5] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. 2019. Engineering a High-Performance GPU B-Tree. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2019)*.

[6] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 94–103.

[7] Shimin Chen, Phillip B Gibbons, and Todd C Mowry. 2001. *Improving index performance through prefetching*. Vol. 30. ACM.

[8] Shimin Chen, Phillip B Gibbons, Todd C Mowry, and Gary Valentin. 2002. Fractal prefetching B+-trees: Optimizing both cache and disk performance. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 157–168.

[9] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 26.

[10] Douglas Comer. 1979. Ubiquitous B-Tree. *Acm Computing Surveys* 11, 2 (1979), 121–137.

[11] Mayank Daga and Mark Nutter. 2012. Exploiting Coarse-Grained Parallelism in B+ Tree Searches on an APU. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 240–247. https://doi.org/10.1109/SC.Companion.2012.40

[12] Duane Merrill,NVIDIA Research Group. 2018. CUB Documentation. https://nvlabs.github.io/cub/index.html#sec9

[13] Toshio Endo, Yuki Takasaki, and Satoshi Matsuoka. 2015. Realizing extremely large-scale stencil applications on GPU supercomputers. In *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*. IEEE, 625–632.

[14] Jordan Fix, Andrew Wilkes, and Kevin Skadron. 2011. Accelerating braided b+ tree searches on a gpu with cuda. *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC)* (2011), 1–11.

[15] Goetz Graefe et al. 2011. Modern B-tree techniques. *Foundations and Trends® in Databases* 3, 4 (2011), 203–402.

[16] Richard A Hankins and Jignesh M Patel. 2003. Effect of node size on the performance of cache-conscious B+-trees. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 31. ACM, 283–294.

[17] Justin Holewinski, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 311–320.

[18] Guanghao Jin, Toshio Endo, and Satoshi Matsuoka. 2013. A multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of GPU. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 1080–1087.

[19] Guanghao Jin, Toshio Endo, and Satoshi Matsuoka. 2013. A parallel optimization method for stencil computation on the domain that is bigger than memory capacity of GPUs. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. IEEE, 1–8.

[20] Ty McKercher John Cheng, Max Grossman. 2014. *Professional CUDA C Programming*.

[21] Krzysztof Kaczmarski. 2011. Experimental B+-tree for GPU. *ADBIS (2)* 11 (2011).

[22] Krzysztof Kaczmarski. 2012. B+-tree optimized for GPGPU. In *OTM Confederated International Conferences*. Springer, 843–854.

[23] Peter Kieseberg, Sebastian Schrittwieser, Lorcan Morgan, Martin Mulazzani, Markus Huber, and Edgar Weippl. 2013. Using the structure of b+-trees for enhancing logging mechanisms of databases. *International Journal of Web Information Systems* 9, 1 (2013), 53–68.

[24] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott a Brandt, and Pradeep Dubey. 2010. FAST : Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. *Sigmod '10* (2010), 339–350.

[25] Jiajia Li, Xingjian Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2012. An optimized large-scale hybrid DGEMM design for CPUs and ATI GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 377–386.

[26] Hang Liu and H Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*. IEEE, 1–12.

[27] Addison Snell Michael Feldman. 2015. Accelerated computing: a tipping point for HPC. (2015). https://www.nvidia.it/content/EMEAI/images/tesla/tesla-server-gpus/accelerated-computing-at-a-tipping-point.pdf

[28] Microsoft Redmond and Microsoft Research Cambridge. 2009. DBMS workloads in online services. http://www.tpc.org/tpctc/tpctc2009/tpctc2009-10.pdf

[29] J Ian Munro and Hendra Suwanda. 1980. Implicit data structures for fast search and update. *J. Comput. System Sci.* 21, 2 (1980), 236–250.

[30] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2014. Cache-aware sparse matrix formats for Kepler GPU. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*. IEEE, 281–288.

[31] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2016. Adaptive multi-level blocking optimization for sparse matrix vector multiplication on GPU. *Procedia Computer Science* 80 (2016), 131–142.

[32] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 101–110.

[33] Kerttu Pollari-Malmi, Eljas Soisalon-Soininen, and Tatu Ylonen. 1996. Concurrency control in B-trees with batch updates. *IEEE Transactions on Knowledge and Data Engineering* 8, 6 (1996), 975–984.

[34] Jun Rao and Kenneth A Ross. 1999. Cache conscious indexing for decision-support in main memory. In *VLDB*, Vol. 99. Citeseer, 78–89.

[35] Jun Rao and Kenneth A Ross. 2000. Making B+-trees cache conscious in main memory. In *ACM SIGMOD Record*, Vol. 29. ACM, 475–486.

[36] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P Sadayappan. 2018. Register optimizations for stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 168–182.

[37] Naser Sedaghati, Arash Ashari, Louis-Noël Pouchet, Srinivasan Parthasarathy, and P Sadayappan. 2015. Characterizing dataset dependence for sparse matrix-vector multiplication on GPUs. In *Proceedings of the 2nd workshop on parallel programming for analytics applications*. ACM, 17–24.

[38] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P Sadayappan. 2015. Automatic selection of sparse matrix representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 99–108.

[39] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A Hybrid B+-tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16* (2016), 1523–1538.

[40] Ben Shneiderman. 1976. Batched searching of sequential and tree structured files. *ACM Transactions on Database Systems (TODS)* 1, 3 (1976), 268–275.

[41] V Srinivasan and Michael J Carey. 1993. Performance of B+ tree concurrency control algorithms. *The VLDB Journal* 2, 4 (1993), 361–406.

[42] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 417–432.

[43] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. 2011. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 35.

[44] Renne Chan Teresa Lam, Christy Li. 2017. Sales performance of Alibaba in 2017 Single's Day. (2017). https://www.fbicgroup.com/sites/default/files/CREQ_04.pdf

[45] Alejandro Vaisman and Esteban Zimányi. 2011. Data warehouses: Next challenges. In *European Business Intelligence Summer School*. Springer, 1–26.

[46] Panos Vassiliadis and Alkis Simitsis. 2009. Near real time ETL. In *New Trends in Data Warehousing and Data Analysis*. Springer, 1–31.

[47] Bin Wang. 2015. *Mitigating GPU Memory Divergence for Data-Intensive Applications*. Ph.D. Dissertation.

[48] Xin Wang, Weihua Zhang, Zhaoguo Wang, Ziyun Wei, Haibo Chen, and Wenyun Zhao. 2017. Eunomia: Scaling concurrent search trees under contention using htm. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 385–399.

[49] Weihua Zhang, Xin Wang, Shiyu Ji, Ziyun Wei, Zhaoguo Wang, and Haibo Chen. 2018. Eunomia: Scaling Concurrent Index Structures Under Contention Using HTM. *IEEE Transactions on Parallel and Distributed Systems* 29, 8 (2018), 1837–1850.