A Hybrid B+-tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms

Amirhesam Shahvarani
Fakultät für Informatik
Technische Universität München
shahvara@in.tum.de

Hans-Arno Jacobsen
Fakultät für Informatik
Technische Universität München

ABSTRACT

An in-memory indexing tree is a critical component of many databases. Modern many-core processors, such as GPUs, are offering tremendous amounts of computing power making them an attractive choice for accelerating indexing. However, the memory available to the accelerating co-processor is rather limited and expensive in comparison to the memory available to the CPU. This drawback is a barrier to exploit the computing power of co-processors for arbitrarily large index trees.

In this paper, we propose a novel design for a B$^+$-tree based on the heterogeneous computing platform and the hybrid memory architecture found in GPUs. We propose a hybrid CPU-GPU B$^+$-tree, – HB$^+$-tree, – which targets high search throughput use cases. Unique to our design is the joint and simultaneous use of computing and memory resources of CPU-GPU systems. Our experiments show that our HB$^+$-tree can perform up to 240 million index queries per second, which is 2.4X higher than our CPU-optimized solution.

CCS Concepts

•Information systems → Data management systems; Data structures; •Computer systems organization → *Multicore architectures;*

Keywords

Heterogeneous Computing; Indexing; In-memory Database; B$^+$-tree

1. INTRODUCTION

The B$^+$-tree is a well known dynamic data structure, widely used as index in database management systems, data warehouses, online analytical processing (OLAP), decision support systems and data mining [10, 26, 4, 15]. Since the memory capacity of modern servers is sufficiently large, in many databases today, indexing information is kept in main memory in order to eliminate performance limitations arising from expensive disk I/O [35, 2]. Due to different characteristics of main memory, implementing an efficient in-memory B$^+$-tree involves different constraints [42].

Approaches that leverage GPUs to accelerate processing have become popular in many domains due to the superior computing power to price ratio offered by many GPUs [39, 41]. Also, in databases, several approaches have emerged to demonstrate the benefits of using GPUs to accelerate processing, such as, GPUTeraSort for sorting billion-record wide-key databases [17] and GPU-accelerated relational join processing [23][6]. Also, tree-based indexing, as a critical operation in databases, has been in the focus of recent approaches [27, 28, 13].

A GPU offers a higher memory bandwidth as compared to a CPU, which makes the GPU an attractive choice for database indexing. However, the efficient utilization of both memory bandwidth and computation resources of a GPU is a challenging endeavor because of distinct architecture of the GPU, which forces programmers to use the same arrangement of parallel threads for both computation and data transfer [7]. In addition, leveraging the GPU as a processing accelerator necessarily involves data transfer between main memory and GPU memory, resulting in additional latency [20].

In this paper, we present HB$^+$-tree (Hybrid B$^+$-tree), a modified B$^+$-tree, jointly leveraging CPU and GPU resources of the same compute platform. Our design is geared towards lookup intensive applications where tree updates are performed through bulk update processing, applicable to index updates in online analytical processing (OLAP), decision support systems and data mining [47, 48, 18].

Realizing indexing operations based on either CPU or GPU is subject to different trade-offs. CPU performance is bounded by *memory bandwidth* as the index grows beyond the size of the last level cache (LLC), while GPU performance is bounded by *memory capacity*. Although GPU's memory architecture is efficient enough enabling the GPU to reach higher throughput, the memory available to the GPU is more limited than CPU's main memory. Intuitively speaking, our design objective is to combine these characteristics of CPU and GPU memory to achieve high throughput for index tree operations over high volumes of data. We explore a hybrid design that scatters index data among CPU and GPU memory according to the volume and the frequency of accesses. Complementing this design, we proposed a heterogeneous CPU-GPU algorithm for searching the HB$^+$-tree. We develop a task pipelining method between CPU and GPU to

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2882918

overcome the communication cost between them. For better computation resource utilization, the task pipeline is further extended by double buffering, which is a concurrency design pattern for avoiding delay in data transfer [19]. We also design a load balancing scheme to improve resource utilization for systems with different GPU to CPU computation power ratios.

Furthermore, we propose two versions of HB$^+$-tree in this paper. In addition to the *regular* HB$^+$-tree, capable of efficiently performing bulk updates, we propose an array representation, referred to as *implicit* HB$^+$-tree, which is more efficient for high-throughput search-only applications. Moreover, we develop a bulk update mechanism for the regular HB$^+$-tree, which deals with the challenges of utilizing a hybrid memory architecture. For both regular and implicit HB$^+$-tree, we develop and evaluate a 64-bit and a 32-bit key versions of the tree. The data structure and algorithm designs we describe are based on using 64-bit keys; the design differences for the 32-bit tree version are summarized at the end of each section.

To the best of our knowledge, HB$^+$-tree is among the first indexing approaches to *jointly* leverage the heterogeneous computing power of CPU and GPU as well as *jointly* utilize their separate memories to achieve a higher aggregate bandwidth than using either memory alone.

Our approach has two main advantages over previous heterogeneous implementations of B$^+$-tree employing, for example, an APU (Accelerated Processing Unit) [13]. First, our design benefits from the hybrid memory architecture to improve memory bandwidth instead of relying on the CPU's main memory alone. Heterogeneous platforms increase the potential computing power of a system, but applications which are bandwidth bounded cannot leverage the extra compute resources unless the memory bandwidth is also improved [29]. Second, we accelerate the index search using a discrete GPU which provides higher computing power than an integrated GPU, as found in APUs. An APU is a system processor equipped with additional processing resources such as a FPGA (Field-Programmable Gate Array) or an integrated GPU to accelerate a specific kind of computation. However, the computing power of these integrated components are not comparable to high-end discrete FPGAs or GPUs that are interconnected with the system via the PCIe bus.

We opted to develop our approach using CUDA, which is the widely-used parallel computing platform and programming model developed by Nvidia for general purpose computing on GPUs [44]. However, there is no technical limitation in our design that prevents porting our approach to other GPU platforms, such as OpenCL [46].

To highlight the advantages of our hybrid solution, we further develop a CPU-optimized, multi-threaded B$^+$-tree, as baseline for comparison with our HB$^+$-tree. For this design, we also develop regular and implicit as well as 64-bit and 32-bit tree versions.

To ensure our CPU-optimized B$^+$-trees exhibits adequate performance, we also implement FAST – the fastest reported indexing performance of a comparable solution running on a single CPU [29] – and compare our implementations against it. Our CPU-optimized B$^+$-tree attains 1.3X higher throughput than FAST on average. Furthermore, for our CPU-optimized tree, we propose a novel tree structure optimization based on cache blocking and SIMD-enabled parallel search algorithm, and show how the use of huge pages help to increase the throughput of index search operations using a single CPU. Several components of the CPU-optimized B$^+$-tree are used in the implementation of our HB$^+$-trees.

We evaluate our solutions for varying number of tuples from 8M to 1B and show how each design decision affects the indexing performance.

HB$^+$-tree achieves up to 240 and 210 million queries per second for implicit and regular tree versions, respectively, which is 2.4X times higher than the results for our CPU-optimized B$^+$-tree.

The remainder of the paper is organized as follows. Section 2 surveys related approaches. Section 3 provides background information on B$^+$-tree. Section 4 presents our CPU-optimized design and implementation of B$^+$-tree. Section 5 introduces our HB$^+$-tree, including implementation details. Section 6 presents our experimental evaluation. Section 7 gives our conclusions and identifies future works.

2. RELATED WORK

A large body of work has been developed to optimize B-tree-like indexing. In this section, we focus on analyzing related work on in-memory indexing employing the power of parallel computing platforms.

B$^+$-tree is an indexing structure originally designed for systems with small main memory and comparatively large hard disks [8][15]. To optimize for costly disk I/O, B$^+$-tree operations are performed for entire disk blocks yielding fewer I/O transactions.

Flash-aware indexing trees such as BF-tree, FD-Tree, and LA-Tree have been proposed to reap performance benefits from the superior bandwidth and latency of solid state drives [5, 34, 1]. Furthermore, there exist many approaches for in-memory indexing in order to exploit the superior bandwidth and latency of system main memory. For example, Zhang et al. [22] provide a comprehensive review of data structures for in-memory data management such as for time-/space efficient indexing and concurrency control.

T-tree was proposed for databases where both indexing information and data records reside in main memory [31]. Lu et al. [36] showed that B$^+$-tree outperforms T-tree when concurrency control mechanisms are enforced. Rao et al. [42] introduce a cache-conscious indexing data structure, called Cache Sensitive Search Tree (CSS-tree), which is designed for predominantly static data. Later, Rao et al. [43] extended CSS-tree to CSB$^+$-tree to support incremental updates.

The Bw-tree is designed to exploit the caches of modern multi-core chips and the superior bandwidth of flash storage [33]. Zhou et al. [50] present an access buffering technique for in-memory tree-structured indexes that avoids cache thrashing. Mao et al. [37] introduced Masstree, a shared concurrent data structure combining B$^+$-tree and tries tailored to multi-cores. Hankins et al. [21] studied the effect of node size on cache misses, instruction count, and TLB misses for the CSB$^+$-tree. Based on their experiments, using nodes with sizes of 512 bytes and above, resulted in fewer TLB misses and better performance, while setting nodes size equal to the cache line width produced fewer cache misses but higher TLB misses. Chen et al. [11] explored how prefetching could improve operations in B$^+$-tree, also concluding that nodes wider than a single cache-line resulted in better performance. ART (Adaptive Radix

Tree) and FAST (Fast Architecture Sensitive Tree) are the latest data structures targeting high throughput in-memory indexing [32] [29]. ART is an adaptive radix tree (trie) for high speed in-memory indexing which exhibits better memory usage than previous radix trees. Alvarez et al. [3] compared the lookup throughput and memory footprint of ART to different data structures including B$^+$-tree and hash indexes. FAST is a static binary-tree developed for multi-core systems, which is configurable according to system characteristics such as cache-line size, memory page size and SIMD width. Sewall et al. [45] introduced PALM, a parallel latch-free modification of a B$^+$-tree designed for multi-core processors which is capable of concurrent search and update processing.

All these approaches are developed to utilize either a single-core or a multi-core CPU expect FAST which is capable to be configured for many-core GPU accelerators. But it is only able to operate on GPU resident data and assumes that the data fits into the GPU memory; it is therefore bounded by the GPU memory capacity. There are other GPU-accelerated index structures, which suffer from the same limitations. Fix et al. [16] presented an approach for a GPU-accelerated B$^+$-tree by proposing to modify the memory layout of the B$^+$-tree optimized for GPU memory. No GPU search throughput is reported, but a 9.4X to 19.2X speedup over a single-threaded CPU implementation is shown. Kaczmarski [27] proposed a GPU-specific implementation of B$^+$-tree which is capable of performing efficient updates. Although this approach performs bulk insertions faster than a CPU implementation, search throughput does not surpass 25.6 Kilo Queries Per Second (including copying keys from CPU to GPU and returning values back). Also in [28], the authors proposed a p-ary search with the goal of improving response time of query search using GPUs.

The limited capacity of GPU memory is addressed by other approaches. Daga et al. [13] utilize an APU (Accelerated Processing Unit) to accelerate search in a B$^+$-tree. Since the integrated many-core processor is directly accessing system main memory, their approach does not suffer from the penalty of having to move data between CPU and GPU memory and the limited capacity of GPU memory does not constitute a problem. However, it is still bounded by system main memory bandwidth, which is a critical problem for tree traversal as the tree grows [29]. Their implementation achieved up to 18 MQPS for a 6-core CPU and 70 MQPS for an APU (operating on system memory). Although, an APU is a heterogeneous computing platform, our solution based on a similar platforms has two main advantages. First, we are jointly utilizing two memory components, which is critical to improve the overall system memory bandwidth, while the APU approach is still relaying only on system main memory. Second, the computing power of high-end discrete GPUs is significantly higher than the integrated accelerator available in APUs.

3. B$^+$-TREE BACKGROUND

B$^+$-tree is a variation of B-tree which stores values only in leaf nodes, while inner nodes only comprise keys [15]. Hence, inner nodes and leaf nodes are represented by different data structures. Beside the characteristics adapted from B-tree such as height-balance and optimized memory access, B$^+$-tree offers faster range query support because of its sorted linked leaf nodes. The branching factor of the inner nodes

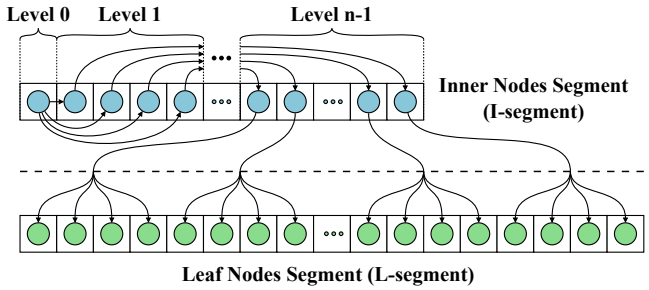

Figure 1: Arrangement of nodes in I-segment and L-segment.

is called the *order* of the B$^+$-tree. Inner nodes of order m, store up to $m-1$ keys and m child references.

Search in B$^+$-tree is a step-wise process, traversing the tree from the root node, each step consists of two parts: first, the search detects the child node associated with an interval which holds the targeted key, and, second, traversal proceeds to the next node. The process continues until the target key-value pair is found in a leaf node. To perform a range query on a B$^+$-tree, one can simply search for the first key in the range and then traverse leaf nodes until the last key is found.

Data structures in which the structural information is implicitly preserved in the way data is stored rather than explicitly through pointers, are called *implicit data structures* [38]. In implicit representation of B$^+$-tree, nodes are arranged in a breadth-first fashion in a one dimensional array. Since a node's child locations are known, and there is no need to store pointers, an implicit B$^+$-tree requires less memory and provides higher search throughput as compared to a regular B$^+$-tree. However, using an implicit representation leads to a linear time penalty for insert and delete operations. To distinguish the implicit representation from the one with pointers, we refer to the latter as the *regular* B$^+$-tree and the former the *implicit* B$^+$-tree in the rest of this paper.

The notations we use in this paper is summarized below.

H : Height of root node (leaves are at height zero).
S : Size of a variable (a key or a value) in bytes.
S_I : Size of an inner node in bytes.
S_L : Size of a leaf node in bytes.
F_I : Maximum fanout of an inner node.
P_L : Maximum capacity of key-value pairs in a leaf node.

4. CPU-OPTIMIZED B$^+$-TREE

In this section, we describe our parallel design of both, the implicit and the regular B$^+$-tree, optimized to exploit the features of a multi-core Intel CPU. Our CPU-optimized solutions serve as baseline in the evaluation of our hybrid solution, the HB$^+$-tree, described in the next section. The three main optimization we applied for the CPU-optimized solutions are : (1) an SIMD-enabled search algorithm based on the Intel AVX extension, (2) cache blocking to minimize cache misses, (3) huge page utilization to reduce TLB misses.

4.1 Tree Layout

The node structures of the CPU-optimized B$^+$-tree are designed with regards to minimizing both cache and TLB misses during search operations. We make use of huge pages by developing our own memory allocator which allows deter-

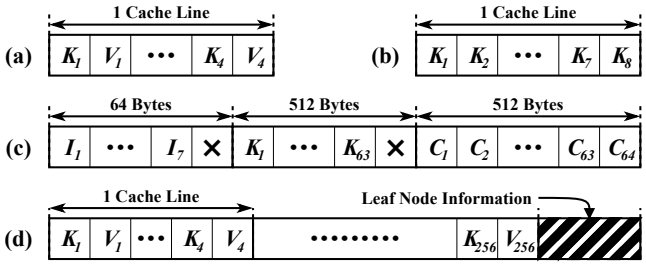

Figure 2: Node structure for CPU-optimized B$^+$-tree. (a) leaf node on implicit B$^+$-tree (b) inner node of implicit B$^+$-tree (c) inner node of regular B$^+$-tree (d) leaf node of regular B$^+$-tree (× indicates the element in fixed to maximum value).

mining whether a node resides on a huge page or not. Coupled with our tree node segmentation, which separates inner and leaf nodes into different segments, our approach minimizes the cost of TLB misses. We also use cache-conscious node structures for better cache data utilization.

The nodes are split into two segments: Inner node segment (I-segment) and leaf node segment (L-segment). The I-segment is always allocated to huge pages, while the L-segment could be allocated to either a huge page or a 4KB page, depending on the total size of the B$^+$-tree. For a given set of N tuples, the space needed for the I-segment (I_{space}) and L-segment (L_{space}) is given in Equation 1 (assuming the tree is full). Since there are only four entries in the last level TLB for 1GB pages and to assure that accessing inner nodes cause no TLB misses, the I-segment must not be larger than 4GB.

$$I_{space} = \frac{N}{P_L(F_I - 1)} \times S_I,\ L_{space} = \frac{N}{P_L} \times S_L \qquad (1)$$

Query search starts from the I-segment, where the root resides, and after passing all inner nodes, continues in the L-segment to determine the target key-value pair in the leaf.

The total number of TLB misses depends on whether the L-segment is placed in a 1GB or a 4KB page. In case of using a 4KB page, since accessing the I-segment causes no TLB miss and each leaf node resides within its individual 4KB page, there is at most a single TLB miss per lookup. If the required memory to store both segments is not more than 4GB, the best option is to also allocate the L-segment on the huge page. Using such a configuration causes no TLB miss for the entire search operation. If the size of the tree exceeds 4GB and the L-segment is allocated to a huge page, the total number of misses depends on the sequence of input queries and the TLB replacement policy.

We design different inner node data structures for our implicit and regular B$^+$-tree, as detailed in Figure 2.

Implicit B$^+$-tree: Since in this tree organization, nodes are arranged in a breadth first fashion, the child node locations are implicitly known. If the node A is the i_{th} node of a tree at level m in breadth first order, then the j_{th} child of A is at position $Offset[m+1]+i\times F_I+j$, where $Offset[l]$ is pointing to the beginning of the l_{th}-level. As a result, it is possible to achieve a higher fanout using the same amount of memory in comparison to the regular B$^+$-tree. We dedicate one cache line per each inner or leaf node ($S_I = S_L = 64$). The only content of leaf nodes are key-value pairs as it shown in Figure 2(a). Since all nodes are fully occupied and they are

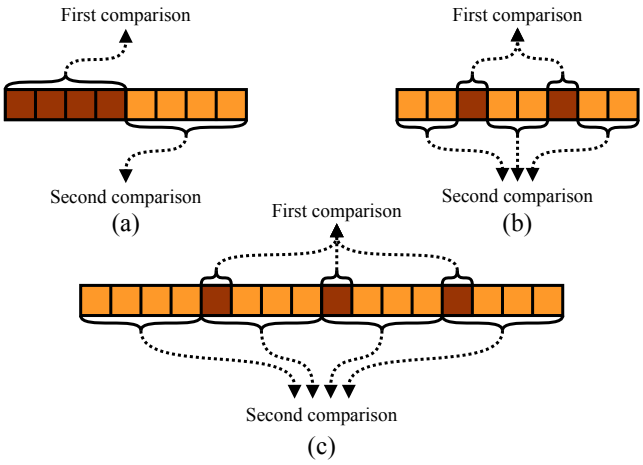

Figure 3: Node search using AVX unit. (a) linear 64-bit (b) hierarchical 64-bit (c) hierarchical 32-bit.

placed in order, there is no need to explicitly maintain node size as well as next and previous node pointers for the linked list of leaves. Each inner node filled with eight keys as illustrated in Figure 2(b). The number of cache lines required per search query is $H + 1$, where $H = \lceil \log_9 (N/4 + 1) \rceil$.

Regular B$^+$-tree: The minimum amount of space required for an inner node with m children is equal to $(m-1)S+mP$ bytes, where P is the reference size in bytes. Considering $S = P = 8$, the maximally achievable fanout using a single cache line is limited to 4. Such a small fanout leads to many random memory accesses during search operations. For better lookup performance, we propose a structure for inner nodes consisting of indexes, keys and child references. As illustrated in Figure 2(c), each inner node consist of 17 cache lines ($S_I = 1088$), where the first one is dedicated to indexes, while keys and references are arranged in the following sixteen cache lines ($F_I = 64$). Each index is assigned to the maximum value of the corresponding cache line ($I_s = K_{8s}$). Utilizing indexes, only three cache lines are retrieved to find the successor node. The search algorithm first searches indexes and, based on the comparison result, fetches the corresponding cache lines, which includes the targeted child reference.

We apply inner node fragmentation in order to achieve better memory and cache line data utilization. The data of each inner node is broken up into two fragments. One fragment contains key-value pairs and child references, and the other one contains the node size, parent and sibling references. Whenever an inner node is needed, our node memory allocator dedicates one of each fragment from two separated data structures in such a way that both fragments share the same index which can be used to retrieve both fragments later on. Also, we set all empty keys of each inner node to the maximum value in our implementation ($2^n - 1$ for an n-bit integer), so that the lookup algorithm is able to perform node search without knowing the inner node size.

The size of a leaf node in the regular B$^+$-tree impacts the range query performance. Moving to the successor leaf node in the implicit B$^+$-tree can be done very efficiently, as leaf nodes are arranged sequentially. But the small leaf node capacity of the regular B$^+$-tree causes a series of cache misses during range query execution and, thus, decreases

performance, while leaf nodes bigger than a cache line lead to slower leaf node search. To address this problem, we designed bigger leaf nodes and make use of a dedicated memory pool manager for allocating leaf nodes and last level inner nodes so that both point and range queries can be realized efficiently. We pack 64 small leaf nodes into a bigger node, which we extend with another cache line to store leaf node information. Each last level inner node is only related to one big leaf node. Similar to the node fragmentation technique we used, here, our memory pool manager allocates leaf nodes and last level inner nodes from two different memory pools in such a way that both nodes share the same index. Consequently, the tree lookup algorithm can directly retrieve the cache line in the leaf node, where the targeted key is located, by using the index of the last inner node and the inner node search result. Moreover, we set all empty elements of a leaf node to the maximum value, which enables the lookup algorithm to search a leaf node without knowing the size of the leaf node. In case the search key is the maximum value, the lookup algorithm must read the node size to perform leaf node search. Although the capacity of the bigger leaf node is 256 key-value pairs, we consider $P_L = 4$ in our analysis since the addressable units from a last inner node are cache lines with a capacity of 4. The structures of implicit and regular B$^+$-tree are illustrated in Figure 2(a) and (d), respectively. The total number of cache lines needed for each query is $3H + 1$, where H (height of tree) is:

$$\left\lceil \log_{32}\left(\frac{N}{4}+1\right)\right\rceil \leq H \leq \left\lfloor \log_{16}\left(\frac{N/2+1}{2}\right)\right\rfloor + 1 \qquad (2)$$

Using 32-bit variables, 16 keys or values can fit into a cache line. Consequently, an inner node's fanout increases to 17 and 256 for implicit and regular B$^+$-tree, respectively. The capacity of each cache line in leaf nodes increases to 8.

4.2 Utilizing SIMD Unit for Search

The Advanced Vector Extensions 2 (AVX2) is the latest enhancement to Intel x86 processors for SIMD operations. AVX2 is capable of operating on 256-bit registers, which is equivalent to eight 32-bit or four 64-bit integers.

Since the size of AVX registers is half the size of a cache line, it is not feasible to compare an entire cache line using a single AVX comparison operation. We propose two different approaches to employ the AVX unit: linear and hierarchical.

The linear approach divides the cache line into two equal parts and separately searches each one. In contrast, the hierarchical approach divides the array into three equal parts and uses the boundary keys to locate the part where the target is placed.

The hierarchical approach needs less data loaded into AVX2 registers, while the linear approach is control dependency free, which is safe for out-of-order execution. We also implemented sequential search as a baseline to measure the resulting speedup. Our AVX-enabled search algorithms are illustrated in Figure 3.

Software Pipelining is a method to improve loop performance by rearranging instructions such that the instructions of the modified loop are chosen from different iterations of the original loop [24]. Using this method, dependent instructions from a single iteration are scattered among multiple loop iterations, so that the CPU pipeline can be scheduled to reduce instruction stalls caused by memory latency. To this end, each CPU thread loads a batch of queries and resolves them concurrently. Using this configuration, the thread switches to resolving another query whenever the current search operation is blocked by a data access. The optimal size for batches depends on the system configuration. Small batch sizes cannot provide reasonable overlap, while large overlap leads to inefficient CPU register utilization. In our experiments, a size of 16 resulted in the best performance. The total number of concurrent queries is up to $16 \times CPU\ Threads$.

5. HYBRID CPU-GPU B$^+$-TREE

In this section, we introduce the design of our CPU-GPU hybrid B$^+$-tree. We describe the tree's memory layout and the heterogeneous CPU-GPU search algorithm. Finally, we present a load balancing method, as technique for fine-tuning our hybrid tree across systems with different GPU-to-CPU computation power ratios.

5.1 Overview

Current multi-purpose processors are heavily relying on cache units to mitigate the memory wall problem [49]. For trees which would fit entirely into the last level cache (LLC), caching is very effective and memory latency would almost vanish. However, search throughput drops noticeably as the tree size surpasses LLC capacity and becomes memory bound [29]. Although techniques such as prefetching and software pipelining are applicable for tree-based index search to alleviate the memory latency problem, the system performance is still bounded by the memory bandwidth [11, 29].

The results from previous efforts of implementing a B+-tree on GPUs demonstrate the realizable performance benefits [29]. Instead of relying on caching, GPUs use high degrees of multi-threading and fast context switching logic with near zero overhead, to hide memory latency [12]. Since this mechanism is not affected by the volume of data, the throughput of tree indexing using GPUs is more resilient against tree growth. As result, GPU-based approaches outperform CPU-based approaches for tree sizes larger than the LLC [29]. However, GPUs cannot maintain their performance advantage, because the amount of their memory is limited in comparison to CPUs. It is not feasible to make use of the computational capabilities of GPUs, when the tree grows beyond the GPU memory capacity using previous methods [13].

To address this dilemma, we propose a new B$^+$-tree, called HB$^+$-tree, leveraging the hybrid memory architecture and heterogeneous computing model of today's computing platforms. Here, we employ the computing power of discrete many-core accelerators for index searching on trees larger than the accelerator's dedicated memory. We design HB$^+$-tree based on a compute platform accelerated by GPUs. To achieve higher total memory bandwidth, we scatter the nodes across GPU *and* CPU memory in a way which enables the index search algorithm to utilize *both memories* concurrently. As a result, the effective system memory bandwidth is the aggregate of both memory units. Also, we design a heterogeneous search algorithm to minimize the communication overhead between processors and *utilize both* – GPU and CPU – simultaneously.

Since our target use cases are lookup-intensive and batch update processing dominated scenarios (e.g., data ware/-

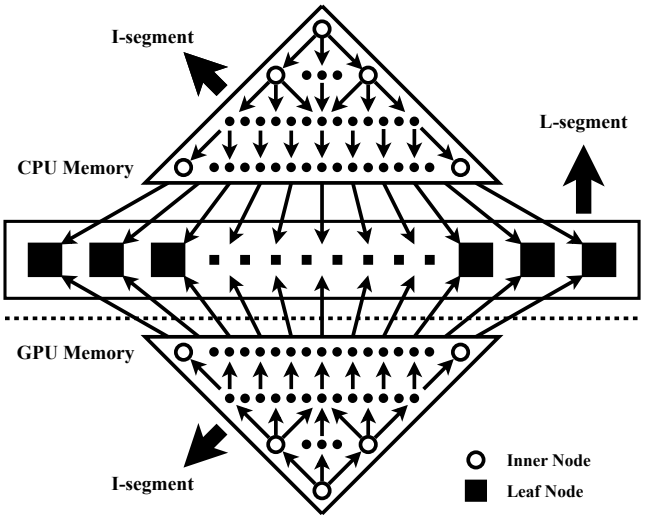

Figure 4: HB$^+$-tree node arrangement: The triangular area is the I-segment which is duplicated on both CPU and GPU memory and the rectangular area is the L-segment which resides only in the CPU memory.

houses), we envision that our indexes are integrated into existing systems based on passing query input and index output via CPU main memory. Also, all our HB$^+$-tree versions exhibits the same interface as our CPU-optimized B$^+$-tree; both follow the conventional B+-tree interface.

5.2 Tree Layout

Similar to the CPU-optimized tree, HB$^+$-tree also consist of I-segments and L-segments. The L-segment is configured based on the CPU search algorithm and only resides in CPU memory, while the I-segment resides in both GPU and CPU memory, i.e., it is mirrored across both memory units. The rationale for this design is that leaf nodes require more space than inner nodes for storage and are less frequently accessed; thus, we place them in CPU memory, which has a higher capacity but lower bandwidth.

Figure 4 illustrates the placement of inner and leaf nodes in the HB$^+$-tree. The leaf nodes of both, the regular and implicit HB$^+$-tree, are identical to the ones of the CPU-optimized version of the tree.

Unlike main memory, the GPU memory architecture does not have a fixed unit of transfer. As a warp executes an instruction accessing GPU memory, the GPU translates the access into one or more aligned data transfers of size 32, 64 or 128 bytes [40]. This limitation is a consequence of coalesced memory access, which results in higher bandwidth as well as higher latency.

We discovered that the best balance between thread scheduling efficiency and bandwidth utilization results from using transfers of size 64 bytes. Since our CPU-optimized B$^+$-tree nodes are also based on 64 byte transfers, the inner node structures of HB$^+$-tree are similar to our CPU-optimized B$^+$-tree. For the regular version, the inner nodes are identical, but we reduce fan-out of inner nodes in implicit HB$^+$-tree to 8, so that we can utilize the same thread hierarchy for both data access and node search and avoid warp divergence, and we set the last key (K_8) to the maximum representable value.

For the 32-bit version, F_I is increased to 16 and 256 for implicit and regular HB$^+$-tree, respectively.

5.3 Parallel Node Search on GPU

In this section, we first describe our search algorithm for an arbitrary sized array and then explain how it is used for search in HB$^+$-tree.

For a given key and a sorted $array$ of s elements such that key is not bigger than the last element ($key \leq array[s]$), the parallel search algorithm finds the maximum index i such that $key \leq array[i]$. The possible values for i are $[1..s]$. To find the target index i, the search algorithm initializes s threads ($t_j : 1 \leq j \leq s$), where each thread is assigned to a single result value. First, each thread (t_j) compares key to the associated value ($array[j]$) to check whether key is less than or equal to $array[j]$ and stores the result ($r_t : 0, 1$) in a shared array. Based on the thread's local comparison result (r_t) and the result from the prior thread (r_{t-1}), each thread determines if it is assigned the final answer. If so (i.e., $r_t = 1$ and $r_{t-1} = 0$), the thread sets the final answer to its own index.

Because the last keys of all inner nodes of HB$^+$-tree are always set to the maximum ($2^n - 1$ for an n bit number), it is assured that all queries are less than or equal to the last key, and our search algorithm always returns a valid result.

Searching an inner node in the regular HB$^+$-tree is slightly different and requires three memory accesses instead of one and involves three steps. First, the parallel search algorithm is applied on indexes to determine the interval of keys containing the search query. Then, the corresponding interval is fetched from GPU memory and searched using the parallel algorithm to identify the next node position. Finally, the address of the next level node is retrieved using an extra memory transfer.

The total number of concurrent queries at the GPU is equal to $GPU\_Threads/T$, where the optimal number of $GPU\_Threads$ depends on the GPU specification and T is the number of threads dedicated per each query (8 for a 64-bit implementation and 16 for a 32-bit implementation).

5.4 Search Query Execution

Since we considered that the input queries are given in CPU memory, the first step is to transfer them into GPU memory, before the GPU starts executing a search operation. After GPU finished its task, the intermediate results, – references to nodes where the search operation must be resumed, – are transfered into main memory after the GPU completes the search operation. In the last step, the CPU continues the search operation to reach the target tuple. The execution of a search on the CPU is analogous to the implementation for the CPU-optimized B$^+$-tree.

The given queries are broken into buckets of size M which are processed independently according to the following steps, where $T_i, i = 1..4$ are times required for each step in our cost model.

1. Transfer bucket to GPU memory.
   $T_1 = T_{init} + (M \times S)/Bandwidt$
2. GPU traversal of all inner nodes of tree per each query.
   $T_2 = K_{init} + (M/SIMD_G) \times P_{GPU}$
3. Transfer of intermediate results to CPU memory.
   $T_3 = T_{init} + (M \times R)/Bandwidt$
4. CPU continues search in leaf nodes.
   $T_4 = (M/SIMD_C) \times P_{CPU}$

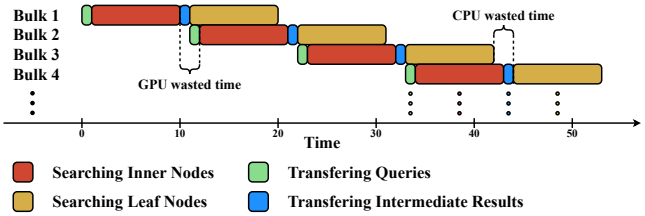

Figure 5: CPU-GPU pipelining.

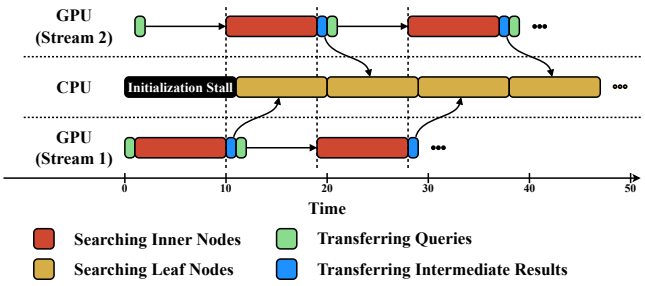

Figure 6: CPU-GPU pipelining with double buffering.

R	: Size of an intermediate result in bytes.
T_{init}	: Data transfer initialization time between main memory and GPU memory.
K_{init}	: GPU initialization time for search operation.
$SIMD_G$	: GPU SIMD width.
$SIMD_C$	: CPU SIMD width.
P_{GPU}	: Average processing time for a query on GPU.
P_{CPU}	: Average processing time for a query on CPU.
$Bandwidth$	: Data transfer bandwidth between main memory and GPU memory.

Assigning the proper value for M is important since both performance parameters, throughput and latency, are controlled by M. Small bucket sizes increase the influence of overhead constants (K_{init} and T_{init}) against effective computation time, leading to lower throughput; increasing M increases the cost of each step (T_i), resulting in higher latency.

Apart from how each bucket is processed, bucket scheduling is also important for optimal utilization of resources. The simplest approach is to load and resolve each query bucket sequentially. The drawbacks of using this approach are two-fold: (1) it is not feasible to utilize both processors concurrently, and (2) there is no opportunity of overlapping communication and computation to eliminate data transfer overhead. Thus, the cost for resolving each bucket is the aggregate of all steps ($T_S = \sum_{i=1}^{4} T_i$). We propose CPU-GPU pipelining and employ a double buffering technique to eliminate these drawbacks.

CPU-GPU pipelining improves system performance by overlapping the execution of buckets. As illustrated in Figure 5, the next bucket is loaded as soon as the intermediate result of the current bucket is transferred into CPU memory. In this way, CPU and GPU can be utilized concurrently. The average time needed to resolve queries within a bucket is reduced to $T_P = T_1 + \max(T_2 + T_3, T_4)$ (ignoring pipeline initialization stalls). Considering $T_2 = T_4$, T_P is equal to $T_1 + T_2 + T_3$, then CPU processing time has been eliminated.

Furthermore, we extend pipelining with double buffering to eliminate the data transfer time. The timeline for the enhanced pipelining approach is given in Figure 6. We initiate two GPU threads which are working on separated buffers but share the same processors, where each thread operates as a CPU-GPU pipelined approach. The average cost of processing each bucket is $T_P = \max(T_2, T_4)$, considering that data transfer time is smaller than computation time.

Although double buffering improves overall system throughput, it also increases processing latency because of prefetching of buckets. The average latency of the pipelined approach is $T_1 + T_2 + T_3 + \frac{T_4}{2}$, which increases to $2 \times T_2 + \frac{T_4}{2}$ by applying double buffering.

5.5 Load Balancing Scheme

Our HB$^+$-tree design is primarily targeting systems which are accelerated using sufficiently powerful GPUs and the system throughput is bounded by the CPU. Therefore, HB$^+$-tree devotes only a small share of the query load to the CPU, which is only searching leaf nodes while all inner nodes are processed by the GPU.

To offer a more generally applicable solution, we enhance HB$^+$-tree with a load balancing mechanism, which improves resource utilization on systems with an arbitrary GPU-to-CPU computation power ratio, refereed to as *load balanced* HB$^+$-tree.

With the load balancing scheme, the CPU starts traversing inner nodes up to a specific depth (D) and transfers the query and the intermediate inner node index to GPU memory. Then, the GPU resumes traversing up to the final inner node level and returns the leaf node index to the CPU. Finally, the CPU searches the leaf node to determine the target key-value pair. We prefer to dedicate the top inner nodes to the CPU since the space required for them is comparably lower than the inner nodes at the bottom of tree resulting in better cache utilization and lookup performance.

Let $I_{G,i}$ and $I_{C,i}$ be the average cost of searching at depth i for GPU and CPU, respectively, and let L_C be the average cost of searching a leaf node, then the average cost of a single search (C) is given according to Equation 3. Adjusting the parameter D is required to minimize C_{inner}.

$$C = \max(L_C + \sum_{0}^{D} C_{C,i}, \sum_{D+1}^{H} C_{G,i}) \tag{3}$$

Moreover, to provide a finer granularity for work load distribution, we divide each bucket into two parts. For the first part, $R \times M$ queries ($0 \leq R \leq 1$) of a bucket, the CPU searches only D levels of inner nodes, while for the rest of the queries ($M \times (1 - R)$), the CPU searches $D + 1$ levels. Using the new parameter R, the search cost C is updated to Equation 4.

$$C = \max(L_C + \sum_{0}^{D-1} C_{C,i} + R\, C_{C,D}, (1-R) C_{G,D} + \sum_{D+1}^{H} C_{G,i}) \tag{4}$$

We develop a discovery algorithm to determine the values for D and R that minimize C. The algorithm starts from $D = 0$ and $R = 1$, where it dedicates the maximum possible load to the GPU. First, it linearly searches for the optimal value of D (coarser parameter). Then, it adjusts R (finer

parameter) using binary search. The discovery algorithm is given in Algorithm 1.

Algorithm 1 Discovery algorithm

```
1: D ← 0, R = 1
2: (Time_GPU, Time_CPU) = getSample(D, R)†
3: while Time_GPU > Time_CPU do
4:     D ← D + 1
5:     (Time_GPU, Time_CPU) = getSample(D, R)
6: R ← 0.5
7: for step ← 2 to 5 do
8:     (Time_GPU, Time_CPU) = getSample(D, R)
9:     if Time_GPU > Time_CPU then
10:        R ← R + 1/(2^step)
11:    else
12:        R ← R − 1/(2^step)
```

†*getSample* runs the program for given D and R; it returns the time GPU and CPU require to perform their work share.

We also change the bucket handling strategy which is advantageous only for GPU bounded systems. The GPU must perform thread scheduling prior to starting effective kernel execution as a new kernel program is submitted to the GPU. Pre-submitting of a successor kernel before the current one is finished, enables the GPU to perform scheduling of the next kernel, concurrently to the previous kernel execution. For this to work, we require at least three concurrently operating buckets. Since this optimization technique is not effective for CPU bounded systems, we restrict the number of query buckets in the not-load-balanced version of HB$^+$-tree to two in order to reduce latency. However, we increase the number of query buckets to three in the load balanced implementation of the HB$^+$-tree for better GPU utilization.

5.6 Batch Update

The implicit B$^+$-tree is not capable of processing individual updates. Whenever an update is required, the entire tree must be re-built. The algorithm first builds both I-segment and L-segment in main memory based on the new dataset and, subsequently transfers the I-segment to GPU memory.

Efficiently processing concurrent batch updates with the regular HB$^+$-tree faces two challenges: (1) I-segment synchronization and (2) concurrency handling. The former is specific to HB$^+$-tree, while the latter is a general challenge for tree indexing. We propose two different tree update methods; their performances depends on the batch size.

We design an asynchronous parallel update method which first performs updates in main memory in parallel and then transfers the entire I-segment to GPU memory. The given update queries are processed in groups of size 16K. Each thread takes a query and searches the tree up to the last level inner node. At this point, the thread checks if the query execution causes any node merge or split. If not, it requests the lock assigned to the inner node and performs the update. Because of HB$^+$-tree's big leaf nodes (256 entries), more than 99% of the update queries can be resolved this way, on average. The remaining unresolved queries are processed subsequently using a single thread. When all queries are executed, the I-segment in GPU memory is updated. This method is more efficient for bigger batch sizes which often result in many inner node modifications. In these cases, it is more beneficial to transfer the entire I-segment once, instead of performing many small transfers for each inner node.

For smaller batch sizes, we propose a synchronized update method which is performed by two threads, a *modifying* and a *synchronizing* one. The modifying thread executes update queries and submits a request for each modified inner node to a shared queue. Upon receiving a request, the synchronizing thread updates the inner node in GPU memory according to the node's replica in main memory. Using this method, tree update and node synchronization proceed concurrently. Although, it is feasible to implement this method with multiple modifying and synchronizing threads, we found the performance of this method is bounded by the communication initialization latency between main memory and GPU memory which was not reduced by parallelism.

6. EVALUATION

We now present the performance evaluation of both CPU-optimized B$^+$-tree and HB$^+$-tree. First, we describe the experimental setup and workload. Then, we demonstrate the impact of various optimizations on the individual approaches, and finally, we compare the search operation performance of CPU-optimized B$^+$-tree and HB$^+$-tree considering latency and throughput.

6.1 Experimental Setup

We used two system setups for evaluating our approaches. The first machine (M_1) is equipped with Intel Xeon E5-2665 accelerated by the Nvidia Geforce 780 GTX. The second machine (M_2) is an Intel Core-i7 4800MQ accelerated by the Nvidia Geforce 770M GTX.

For all experiments except the experiment on skewed data, we generated multiple sets of key-value with 8M (2^{23}) to 1B (2^{30}) tuples, where keys and values are randomly generated according to a uniform distribution on $[0 - MAX]$ ($MAX = 2^n - 1$, n is number of bits: 32 or 64). After constructing the B$^+$-tree using this set, we randomly permuted the pairs using the Knuth shuffle [30]. Finally, we use the new sequence as the input for the search operation.

Our multi-threaded implementation is using OpenMP, an API for parallel computing based on the shared memory programming paradigm [14]. We also made use of PAPI to better understand the performance of our implementation. PAPI is an API for accessing available hardware counters inside the CPU [9].

6.2 CPU-optimized B$^+$-tree Evaluation

Memory Page Configuration. In this experiment, we aim to determine the memory page configuration that maximizes the search operation throughput. We evaluated our B$^+$-tree using three different configurations: (1) both I-segment and L-segment on small pages, (2) I-segment on huge pages and L-segment on small pages, (3) both I-segment and L-segment on huge pages.

To examine our expectation about the average TLB misses per query, we evaluated a single-threaded implementation of all three configurations and counted the TLB misses during search operations using PAPI. Since OpenMP library causes extra TLB misses, we excluded multi-threading to obtain more accurate measurement. We plot the average TLB miss per each query in Figure 7(a). Without utilizing huge pages, the misses increase as the tree grows. Also, it can be seen that searching in the implicit tree causes more TLB misses

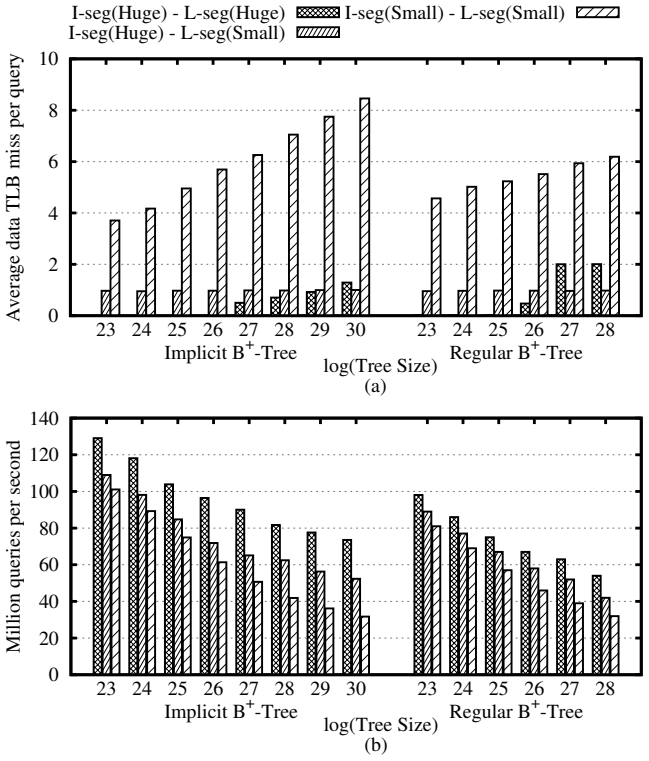

Figure 7: Memory page configuration evaluation. (a) TLB misses (b) Throughput.

than for the regular B$^+$-tree. The reason is the fanout of inner nodes in the implicit tree is inferior to the regular tree, consequently, the tree depth is higher. Allocating only inner nodes on huge pages, significantly reduces the number of misses. In this case, misses are independent of tree depth and they are bounded to one TLB miss per query. Allocating the entire tree on huge pages eliminates misses for smaller trees which do not need more than 4GB of space. As the required space exceeds this amount, the average miss rate increases and surpasses one miss per query. We conclude from Figure 7(a) that in terms of TLB misses, the second configuration is more robust against tree growth, while the third one is best for trees less than 4GB in size.

To determine the effect of TLB misses on tree search performance, we evaluated the multi-threaded tree search using the same configurations. The results are in Figure 7(b). As expected, the first configuration is the least performing. The fastest configuration is the third one, although, it generates more TLB misses than the second configuration for bigger trees. According to our analysis, this behavior is the consequence of the different costs of misses for 4K and 1G pages. As a TLB miss occurs, a page walk is required to retrieve the requested physical address. For 4K pages, five memory accesses are required to translate logical to physical address, while three accesses are sufficient for 1G pages [25]. Even if the TLB miss rate is higher in the third configuration, the penalty of a page walk is less significant, which results in better performance. This experiment indicates the superiority of using huge pages in this application.

SIMD Accelerated Node Search. We now examine the node search algorithms to determine the fastest one and measure the resulting improvements. A query search operation is evaluated using three different search algorithms: (1) sequential, (2) linear SIMD, and (3) hierarchical SIMD; software pipelining is applied in all of these configurations. To indicate the effectiveness of software pipelining in this application, we also evaluated sequential search without software pipelining. Since AVX2 support is required for the evaluation, we evaluated this experiment on M_2 (M_1 does not support AVX2). The result of the experiments are given in Figure 8.

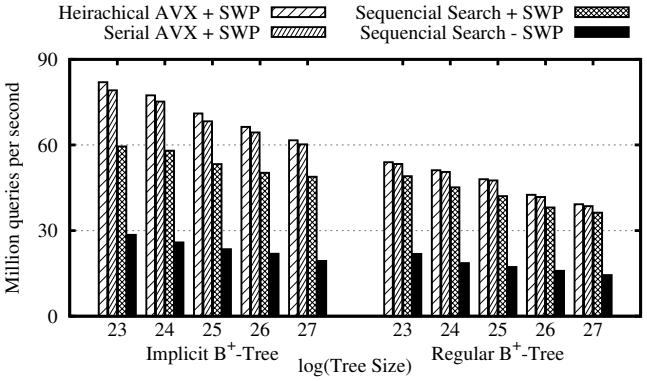

Figure 8: Software pipelining and node search comparison.

Enabling software pipelining is highly effective and improves the system throughout between 108%-152%, while it increased latency by 6X on average. Among the node search algorithms, the hierarchical SIMD approach, achieved the best result; it is slightly faster than linear search. Both SIMD implementations lose their advantage to sequential search as the tree size grows. This behavior confirms that tree processing becomes memory latency bounded for bigger trees, and memory optimization techniques are ever more important in this case.

Comparison with FAST. We compare our CPU-optimized implicit B$^+$-tree to FAST [29], the fastest reported indexing tree in the literature, – also an implicit structure, – to assure our CPU-optimized B$^+$-tree design is competitive enough to be used as a performance baseline. As shown in Figure 9, our B$^+$-tree achieved 1.3X higher throughput on average than FAST. Our different SIMD-enabled node search, which allows us to reach higher node fan-out and, consequently, better cache line utilization, is the source for this improvement. Even though our implementation achieves better performance than what FAST reported, we do not aim to challenge FAST in this work, since FAST is designed to be a configurable data structure, able to adapt to different hardware configurations, while our design is specifically tuned for the Intel architecture.

6.3 HB$^+$-tree Evaluation

Bucket Handling Strategies. In this experiment, we study three different bucket handling techniques: (1) sequential, (2) pipelining, and (3) pipelining with double buffering. With sequential bucket handling, it is neither feasible to employ CPU and GPU simultaneously, nor overlap communication and computation. This approach is the simplest; we use it as baseline in our evaluation. Resolving buckets using pipelining allows us to partially overlap CPU and GPU computations. Double buffering helps to overlap

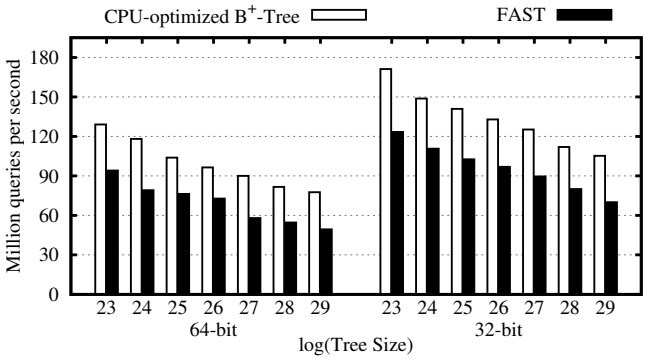

Figure 9: Comparison of FAST and implicit CPU-optimized B$^+$-tree.

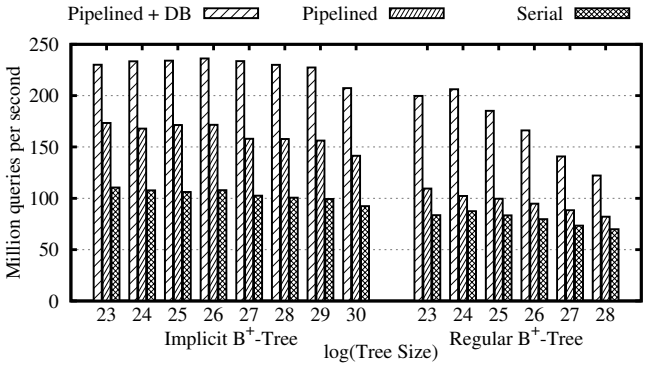

Figure 10: Bucket handling strategy evaluation.

data transfer and search, so that it improves resource utilization. We show the results for search using these techniques in Figure 10.

Sequential bucket handling is the least efficient. Pipelining is more effective for the implicit B$^+$-tree. It increases the throughput by 56% for implicit and by 20% for regular B$^+$-tree. The double buffering technique is effective for both tree versions. Using bucket pipelining extended by double buffering improves throughput by 110% over the baseline technique. Gaining twice the throughput in comparison to the sequential approach indicates that we successfully managed to simultaneously exploit the computation capabilities of both processors.

Bucket Size. The goal of this experiment is to determine the optimal bucket size considering both throughput and latency. Increasing the bucket size, diminishes the influence of communication and GPU initialization overheads, resulting in better system throughput, while at the same time, increases the system latency. We evaluated the search operation using M_1 for different bucket sizes: 8K, 16K, 32K, and 64K. As shown in Figure 11, search throughput grows, as bucket size increases for the implicit B$^+$-tree, while for the regular B$^+$-tree, the throughput is nearly the same for bucket sizes 16K, 32K, and 64K. Considering that the average latency also increases as the bucket size grows (2.7X for 64K and 1.7X for 32K), we use 16K as the optimal bucket size for the rest of our experiments.

Impact of Skewed Data. We studied HB$^+$-tree for several input data distributions, including Uniform, Normal($\mu = 0.5$, $\sigma^2 = 0.125$), Gamma($k = 3$, $\theta = 3$) and Zipf($\alpha = 2$). The generated random values are in the range

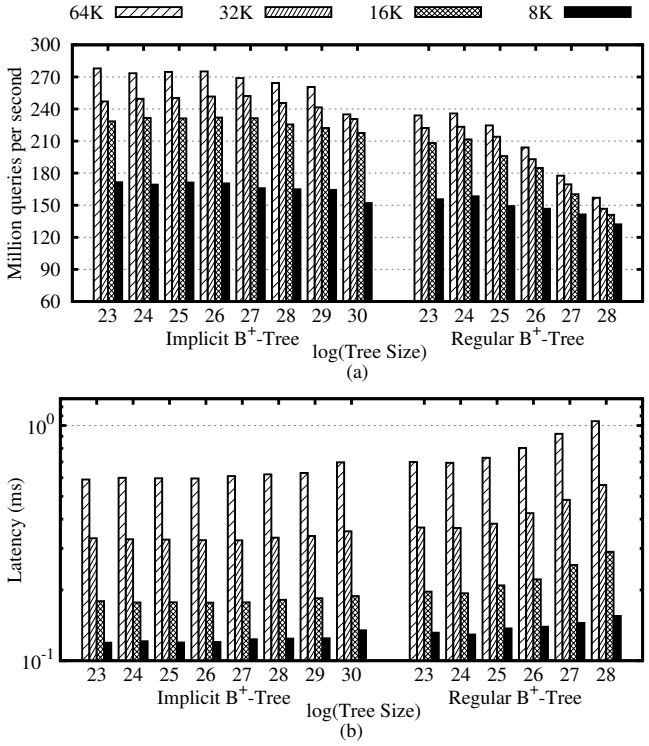

Figure 11: Experiment on varying size of buckets (a) throughput (b) latency.

$[0, 1]$. Before the values are given to search queries, they are linearly mapped to $[0, MAX]$. We used the Uniform distribution as the baseline and scaled the results of other distributions accordingly. The normalized results are illustrated in Figure 12.

The performance on all distributions, except Zipf, is within 1.1X of the Uniform distribution, while the performance for Zipf input data increases by up to 2.2X. When the data becomes more skewed, the same portion of the tree is accessed more frequently, which results in a higher cache hit rate. This behavior is even more pronounced for highly skewed data, such as the Zipf distribution.

Update Performance. In this experiment, we evaluate the performance of update query execution on HB$^+$-tree as compared to CPU-optimized B$^+$-tree for both regular and implicit tree versions.

We first present evaluations of the different update query execution methods for the regular HB$^+$-tree including both the single- and multi-threaded versions of the synchronous and asynchronous approach. Figure 13(a) illustrates the throughput of these methods for various tree sizes; the I-segment transfer time is excluded for the asynchronous approaches. Parallel execution is more effective in the asynchronous approach which results in 3X higher throughput in comparison with the single-threaded approach. The synchronous approach is only 30% faster than the multi-threaded one, which is bounded by the data transfer latency between CPU and GPU memory.

The I-segment synchronization times for different tree sizes are illustrated in Figure 13(b). To examine the effect of I-segment synchronization overhead, we measure the time required to perform batch updates with different batch sizes in a tree of size 64M. The results are shown in Figure 14.

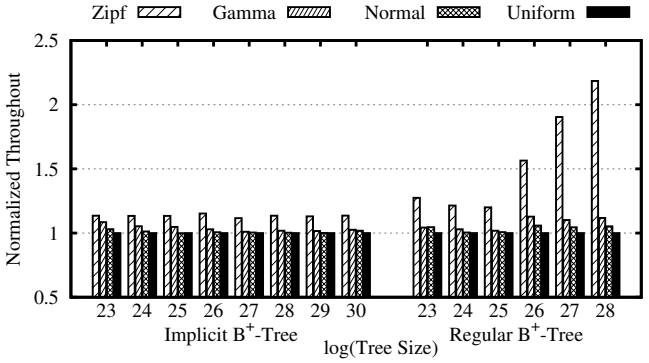

Figure 12: Experiment on different distributions.

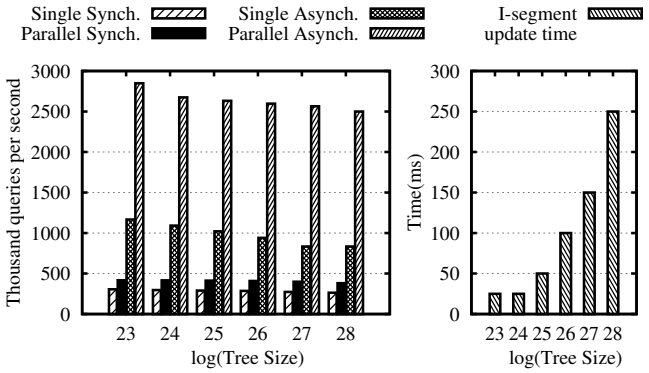

Figure 13: Evaluation of Regular B$^+$-tree update.

Up to a batch size of 64K, the synchronous approach performs better because of the slow I-segment transfer in the asynchronous approach. But for batches larger than 128K, the asynchronous is more effective, as the I-segment transfer cost is amortized by the larger number of queries processed. This experiment shows that the choice of update depends on the batch size. A synchronous update is more efficient for smaller batches while an asynchronous one performs better for larger batches.

To update implicit CPU-optimized B$^+$-tree, the entire tree has to be rebuilt, including the I-segment and L-segment. For implicit HB$^+$-tree, it is additionally required to transfer the I-segment to GPU memory. To compare the cost of updating these two trees, we measure the cost of each phase including L-segment rebuilding, I-segment rebuilding, and I-segment transfer separately as shown in Figure 15. The cost of transferring the I-segment is only 3 to 7 percent of tree reconstruction.

6.4 HB$^+$-tree vs. CPU-optimized B$^+$-tree

We now compare the search performance of HB$^+$-tree against the CPU-optimized B$^+$-tree in terms of throughput, latency and selectivity using M_1.

Throughput. Figures 16(a) and 16(b) show the search performance of both trees (for 64-bit and 32-bit variable sizes). The throughput of the implicit HB$^+$-tree is almost constant for different tree sizes, which indicates that the amount of time the GPU requires for traversing inner nodes is inferior to the time, the CPU requires for scheduling and searching leaf nodes. Consequently, the search performance is bounded by the computational power of the CPU. However, the regular HB$^+$-tree does not show similar behavior;

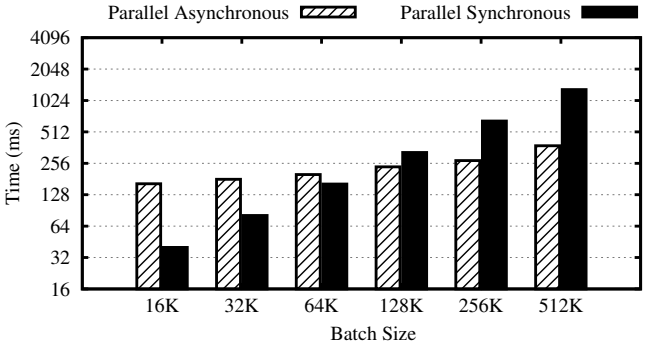

Figure 14: Regular B$^+$-tree update for different batch sizes.

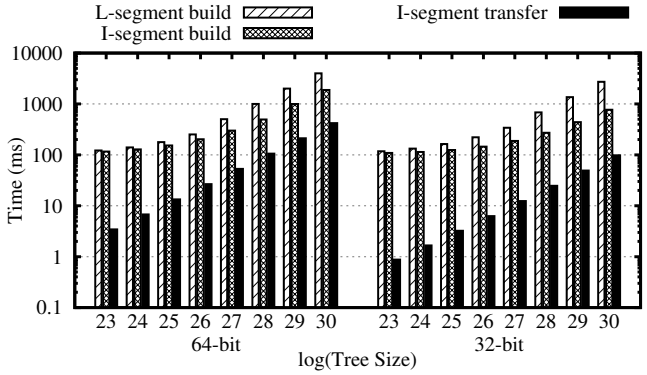

Figure 15: Implicit HB$^+$-tree update.

similar to the CPU-optimized tree, its performance declines as the tree grows. The GPU accelerated approach outperforms the CPU-optimized approach by 2.4X and 2.1X higher throughput on average for 64-bit and 32-bit variables, respectively.

Latency. Figure 16(c) illustrates the query search latency for both HB$^+$-tree and CPU-optimized B$^+$-tree. The hybrid approach exhibits comparably higher latency, 67X on average, than the CPU-optimized one. The higher latency is the consequence of a different number of queries required for an effective utilization of each platform. The number of concurrent queries for CPU and GPU are 2^8 and 2^{14}, respectively, where the ratio (64) is almost the same as the latency ratio. The average latency of the hybrid approach is less than 0.18ms for the implicit B$^+$-tree and 0.25ms for regular the B$^+$-tree.

Range queries. In this experiment, we compare HB$^+$-tree against CPU-optimized B$^+$-tree in performing range queries for different numbers of matching keys per query for a total of 128M keys. Figure 17 shows the performance of range queries for the retrieval of 1 to 32 keys. Since range queries require more leaf node traversal, the ratio of the search time in inner nodes to the entire lookup time decreases for these queries. As a result, the lookup performance of implicit and regular tree versions becomes similar; also, HB$^+$-tree looses its advantage as more keys per query match. HB$^+$-tree is more than 80% faster than the CPU-optimized B$^+$-tree up to 8 matching keys per query and the performance advantage decreases to 22% for 32 matching keys per query.

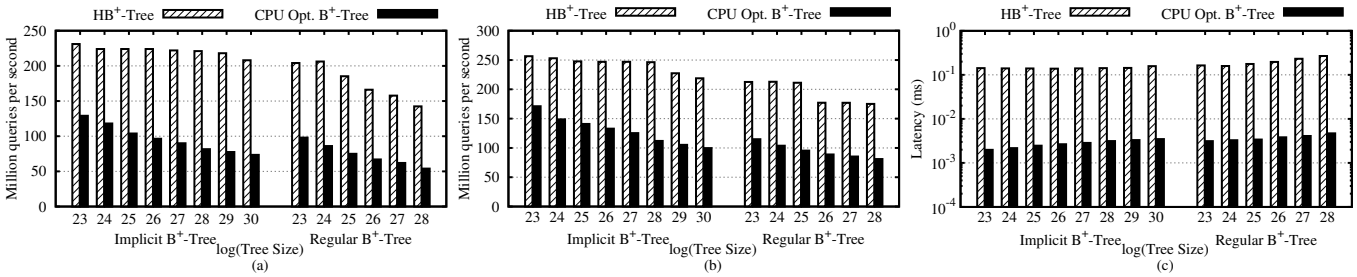

Figure 16: Evaluation of CPU-optimized B$^+$-tree and HB$^+$-tree. (a) Throughput (64-bit) (b) Throughput (32-bit) (c) Latency (64-bit).

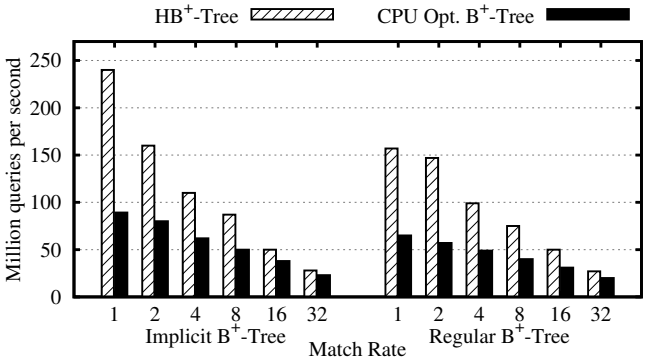

Figure 17: Throughput of Range queries.

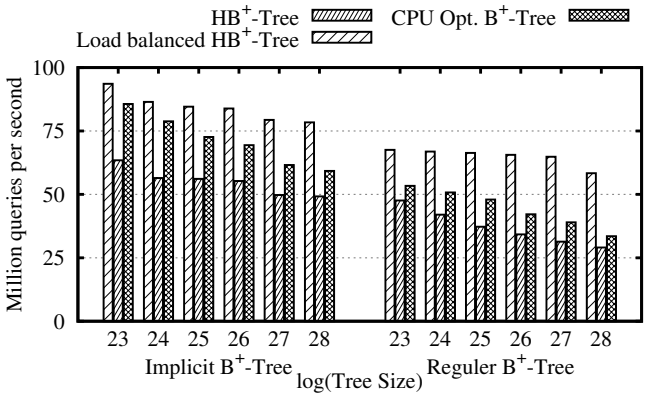

Figure 18: Evaluation of load balancing scheme.

6.5 Load Balancing Evaluation

We now examine the effectiveness of our load balancing scheme on heterogeneous platforms where the computation power is not bounded by the CPU. To this end, we used M_2 which is, relatively speaking, equipped with a less powerful GPU accelerator. Figure 18 shows the results. Without load balancing, HB$^+$-tree performs 25% slower than our CPU-optimized tree, on average. This indicates that the communication overhead between both processors is far higher than the acceleration provided by the GPU.

Applying load balancing scheme is highly effective and improves HB$^+$-tree throughput by 65% on average. In comparison to the CPU-optimized tree, the load balanced HB$^+$-tree performs up to 32% and 65% better for the implicit and regular approach, respectively.

7. CONCLUSIONS

In this paper, we presented an indexing structure, called HB$^+$-tree, specifically tailored to a heterogeneous computing platform with a hybrid memory architecture. Index search is accelerated by utilizing the resources of the hybrid GPU-CPU platform to aggregate the processing resources and memory bandwidth of both processing units. These improvements empower our approach to perform search faster for trees where the tree traversal performance approaches the memory bandwidth limit. In such situations, HB$^+$-tree performs on average 2.4X faster search than the CPU-optimized B$^+$-tree, with individual measurements improving performance by up to 2.9X.

The directions for our future work are two-fold: (1) Further support for parallel update queries and (2) development of a general leaf-stored tree processing framework using a CPU-GPU hybrid platform. In this paper, we primarily focused on realizing efficient search. So far, updates are performed sequentially by the CPU with asynchronous data transfer to the GPU; this could be further improved by employing GPU cycles in support of parallel update query execution. The other direction is to develop a general framework which enables the use of a CPU-GPU hybrid platform for any arbitrary leaf-stored tree structure, such that using the node structure and search/update function as input, the framework would determine the parameters for an approach that best utilizes the resources of both CPU and GPU.

8. REFERENCES

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *Proc. VLDB Endow.*, 2009.

[2] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *VLDB*, 2012.

[3] V. Alvarez, S. Richter, X. Chen, and J. Dittrich. A comparison of adaptive radix trees and hash tables. In *ICDE*, 2015.

[4] J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: the definitive guide.* " O'Reilly Media, Inc.", 2010.

[5] M. Athanassoulis and A. Ailamaki. Bf-tree: Approximate tree indexing. *Proc. VLDB Endow.*, 2014.

[6] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *GPGPU*, 2010.

[7] M. Bauer, H. Cook, and B. Khailany. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *SC*, 2011.

[8] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *SIGFIDET*, 1970.

[9] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *IJHPCA*, 2000.

[10] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD*, 1997.

[11] S. Chen, P. B. Gibbons, and T. C. Mowry. *Improving index performance through prefetching*. ACM, 2001.

[12] S. Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2013.

[13] M. Daga and M. Nutter. Exploiting coarse-grained parallelism in b+ tree searches on an APU. In *SCC*, 2012.

[14] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 1998.

[15] R. Elmasri. *Fundamentals of database systems*. Pearson Education India, 2008.

[16] J. Fix, A. Wilkes, and K. Skadron. Accelerating braided b+ tree searches on a GPU with CUDA. In *A4MMC*, 2011.

[17] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, 2006.

[18] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 2011.

[19] M. Grand. *Patterns in Java: a catalog of reusable design patterns illustrated with UML*, volume 1. John Wiley & Sons, 2003.

[20] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *ISPASS*, 2011.

[21] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious B+-trees. In *ACM SIGMETRICS Performance Evaluation Review*, 2003.

[22] B. C. O. K.-L. T. M. Z. Hao Zhang, Gang Chen. In-memory big data management and processing: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 2015.

[23] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, 2008.

[24] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[25] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-052US. September 2014.

[26] M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis. *Fundamentals of data warehouses*. Springer Science & Business Media, 2013.

[27] K. Kaczmarski. B+-tree optimized for GPGPU. In *OTM*. 2012.

[28] T. Kaldewey, J. Hagen, A. Di Blas, and E. Sedlar. Parallel search on video cards. In *HotPar*, 2009.

[29] 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 *SIGMOD*, 2010.

[30] D. E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms, The*. Addison-Wesley Professional, 2014.

[31] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *VLDB*, 1986.

[32] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, 2013.

[33] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, 2013.

[34] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 2010.

[35] J. Lindström, T. Niklander, P. Porkka, and K. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Databases in Telecommunications*. 2000.

[36] H. Lu, Y. Y. Ng, and Z. Tian. T-tree or b-tree: Main memory database index structure revisited. In *ADC*, 2000.

[37] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[38] J. I. Munro and H. Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 1980.

[39] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE micro*, 2010.

[40] C. Nvidia. NVIDIA CUDA programming guide (version 6.5). *NVIDIA Corporation*, 2014.

[41] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. 2008.

[42] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. VLDB, 1999.

[43] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *SIGMOD*, 2000.

[44] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[45] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proc. VLDB Endowment*, 2011.

[46] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 2010.

[47] A. Vaisman and E. Zimányi. Data warehouses: Next challenges. In *Business Intelligence*. Springer, 2012.

[48] P. Vassiliadis and A. Simitsis. Near real time ETL. In *New trends in data warehousing and data analysis*. Springer, 2009.

[49] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH*, 1995.

[50] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *VLDB*, 2003.

APPENDIX

Here, we provide more details on our implementations of the SIMD-enabled node search using AVX unit, the GPU search kernel, and more evaluations.

A. SIMD ENABLED SEARCH

In this section, we present our SIMD enabled search algorithm in more detail. Considering `Node[0..7]` is an array of keys in an inner node and `query` is the given search query, Snippets 1 and 2 show the implementation of the linear and hierarchical approaches for 64-bit keys, respectively.

Snippet 1 Linear AVX search (64-bit)[§].

```
1:  __m256i†Vquery =_mm_set1_epi64x(query);
2:  __m256i vec =_mm256_set_epi64x(node[0],
3:  node[1], node[2], node[3]);
4:  __m256i Vcmp =_mm256_cmpgt_epi64(Vquery,
        vec);
5:  int cmp = _mm256_movemask_epi8(Vcmp);
6:  cmp = cmp & x10101010;
7:  cmp = __builtin_popcount‡(cmp);
8:  int k = cmp;
9:  Vec = _mm256_set_epi64x(node[4], node[5],
10: node[6], node[7]);
11: Vcmp = _mm256_cmpgt_epi64(Vquery, vec);
12: cmp = _mm256_movemask_epi8(Vcmp);
13: cmp = cmp & x10101010;
14: cmp = __builtin_popcount (cmp);
15: k += cmp;
16: // k is the minimum i s.t. query <= node[i]
```

[§]Functions starting with `_mm` are SIMD instructions

[†]256-bit data type as four 64-bit integer values

[‡]method by GNU's Compiler Collection (GCC) determines the number of ones in the binary representation of a number

Snippet 2 Hierarchical AVX search (64-bit).

```
1:  __m128i†Vquery = _mm_set1_epi64x(query);
2:  __m128i Vec =_mm_set_epi64x(node[2], node
        [5]);
3:  __m128i Vcmp =_mm_cmpgt_epi64(Vquery, Vec);
4:  int cmp = _mm_movemask_epi8(cmpRes);
5:  cmp = cmp & 0x00001010;
6:  cmp = __builtin_popcount (cmp);
7:  int k = cmp * 3;
8:  Vec = _mm_set_epi64x(node[k], node[k + 1]);
9:  Vcmp = _mm_cmpgt_epi64(Vquery, Vec);
10: cmp = _mm_movemask_epi8(Vcmp);
11: cmp = cmp & 0x00001010;
12: cmp = __builtin_popcount (cmp);
13: k += cmp;
14: // k is the minimum i s.t. query <= node[i]
```

[†]128-bit data type as two 64-bit integer values

The linear approach first loads the `query` into an AVX vector in Line 1. In Lines 2-4, the first half of the key array (`Node[0..3]`) is loaded into a vector and compared to `key`. Then, the number of keys smaller or equal to the input are stored in variable `k` (cf. Lines 5-8). This process repeats for the second half of the key array adding the comparison result to `k` in Lines 9-15. At the end, `k` is the index of the child to resolve the query.

The hierarchical approach first compares the boundary keys which are `node[2]` and `node[5]` to `query` in Lines 2-4 and based on the comparison results, the search algorithm calculates the index of keys for the second comparison and stores it in `k`. Finally, it compares the `query` to the `node[k]` and `node[k+1]` to find the right child index.

B. ADDITIONAL EVALUATIONS

In this section, we provide further experimental results for our HB$^+$-tree and our CPU-optimized B$^+$-tree.

B.1 HB$^+$-tree lookup using CPU

Figure 19 shows a comparison of the lookup performance of CPU-optimized B$^+$-tree and HB$^+$-tree only using the CPU. The performance of the regular tree versions are identical since they are based on the same node structures. The CPU-optimized implicit B$^+$-tree results in better performance, due to better cache line data utilization. The fan-out of inner nodes in HB$^+$-tree is decremented by one for the benefit of faster search with GPU.

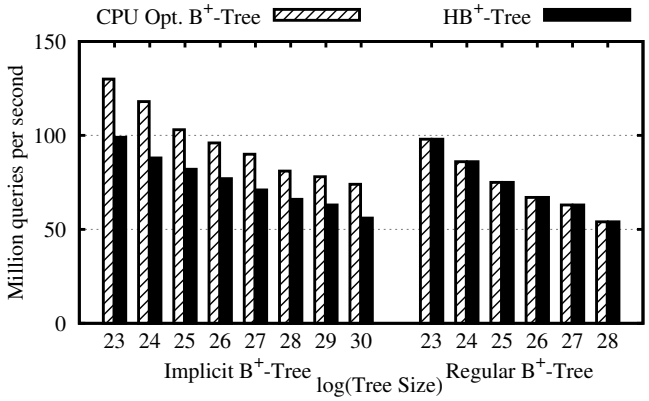

Figure 19: Evaluation of lookup in HB$^+$-tree using CPU.

B.2 Software Pipelining

In this experiment, we study the effect of software pipelining on lookup performance, that is, on throughput and latency. Software pipelining helps the processor to overlap execution with data fetching by executing multiple queries simultaneously, but at the same time, increases processing latency. Algorithm 2 shows how we apply software pipelining with prefetching for CPU-optimized B$^+$-tree lookup. Here, as a thread finishes searching a node, instead of waiting for the child node to be loaded into the cache, it switches to processing another query. Then, when the thread switches back to the same query, the child node is already loaded into cache.

Figure 20(a) illustrates the lookup throughput for various numbers of simultaneously processed queries ranging from 1 to 32. Increasing the number of queries from 1 to 16 continuously improves the throughput, which results in 2.5X better performance than without software pipelining. But due to the limited cache size, increasing the number of simultaneously processed queries from 16 to 32 is not effective and performance remains almost the same. The lookup latency for different software pipeline lengths is illustrated in Figure 20(b) which indicates the latency is quickly increasing with number of queries per thread. On average, the lookup

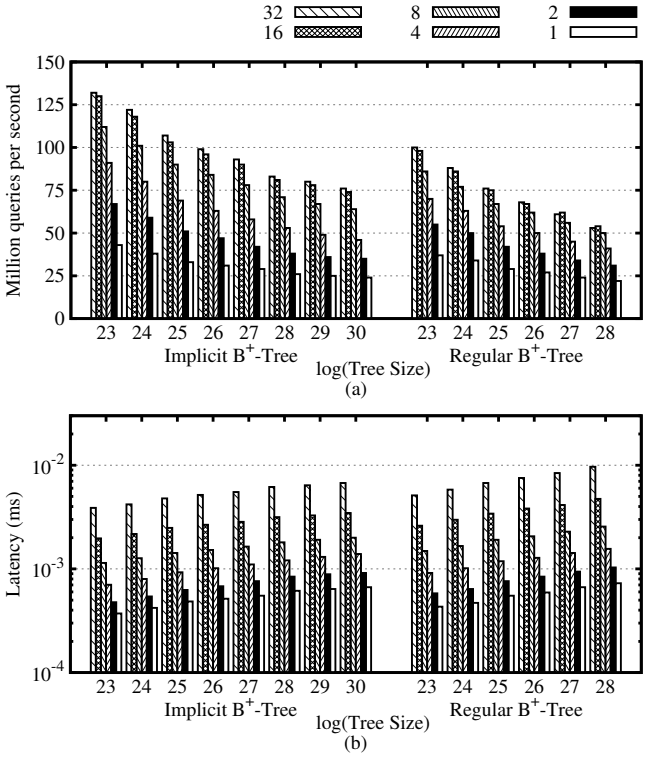

Figure 20: Evaluation of various software pipelining lengths: (a) throughput (b) latency.

latency using a software pipeline of length 16 is 6X higher than without software pipelining.

Algorithm 2 Software pipelining-enabled tree search

Input: P : length of software pipeline
Input: $keys[P]$: search queries
Input: H : height of B$^+$-tree
1: **for** $i \leftarrow 1$ to P **do**
2: $node[i] \leftarrow root$
3: **for** $step \leftarrow 1$ to $H-1$ **do**
4: **for** $i \leftarrow 1$ to P **do**
5: $node[i] \leftarrow getNextNode(\text{I-seg}, node[i], keys[i])$
6: **prefetch**$(node[i])$
7: **for** $i \leftarrow 1$ to P **do**
8: $value[i] \leftarrow getValue(\text{L-seg}, node[i], key[i])$

B.3 Concurrent Search and Update Queries

We now examine the performance of parallel search/update query execution in HB$^+$-tree utilizing only the CPU. We evaluate both synchronous and asynchronous approaches, where the I-segment transfer time is excluded for the asynchronous approach.

The synchronous approach consists of one synchronizing thread, which continuously updates the inner nodes in GPU memory and multiple query processing threads, while the asynchronous approach only consists of query processing threads. The query processing threads are based on the update algorithms given in Section 5.6, which are also capable of resolving search queries. The results are shown in Figure 21. As the ratio of update queries increases, the throughput of the synchronous approach decreases faster than the asynchronous one which is due to the high communication initialization overhead between GPU and main memory. The execution of buckets with 100% search queries in this evaluation is not as fast as our previously evaluated lookup methods which is due to the mutex locking and synchronization overhead in the query processing threads.

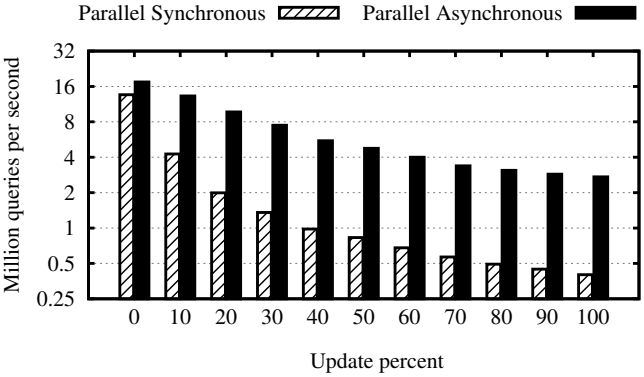

Figure 21: Evaluation of concurrent search/update queries.

C. CUDA PROGRAMMING MODEL

The CUDA programming model is based on hierarchical threading including the notions of *grid*, *block* and *thread*. At the top of the hierarchy is the grid which consists of *blocks* and each block is a group of threads. Threads are distinguished by two index values, `BlockIdx` and `ThreadIdx`, which define the runtime behavior of each thread.

Implementing efficient programs for CUDA requires good understanding of both the GPU memory architecture and the thread scheduling model. The unit of scheduling in CUDA is the *warp*, which is a set of 32 threads. Threads within the same warp are able to execute only a single instruction at a time. The situation called *warp divergence* results when threads of the same warp divert into different code paths, typically occurring for if-then-else statements. As a result, the warp has to be scheduled for both paths separately, which increases the total execution time. Proper index assignment, which directs threads of each warp into a single code path avoids this situation.

Since there is no message passing mechanism supported by CUDA, communication is only feasible via read/write operations on memory. Therefore, efficient bandwidth utilization is of great significance. The CUDA memory architecture is composed of different memory types, where the important ones for our work are *shared memory* and *device memory*.

Device memory, or GPU memory, is the biggest and slowest memory; it can be accessed by an entire grid. All accesses to device memory have to be done through either 32-, 64-, or 128-byte memory transactions. As a warp issues a device memory access, the GPU coalesces this access into transactions of these sizes. In the worst case, each access is translated into 32 separate memory transactions, which divide the device memory performance by 32. Relocating data, such that threads within a warp access adjacent memory locations, leads to more efficient device memory utilization. Shared memory is comparably faster than device memory but it is limited to a block. To provide higher

Snippet 3 GPU kernel code for searching inner nodes in implicit tree ($F_I = 8$)[§].

```
1:  // teamQuery is the requested key
2:  long teamQuery;
3:  __shared__† char flag[9], result;
4:  char selfFlag, i;
5:  long selfKey, nodeIndex;
6:  flag[threadIdx.x] = 0;
7:  nodeIndex = 0; // root index
8:  __syncthreads()‡;
9:  for (i = 0; i < tree_depth; i++) {
10: // levelOffsets is an array stores the
11: // offsets of each level in tree
12: selfKey = tree[levelOffsets[i] + nodeIndex
        + threadIdx.x];
13: flag[threadIdx.x+1] = 0;
14: selfFlag = 0;
15: if (teamQuery <= selfKey) {
16: flag[threadIdx.y][threadIdx.x+1] = 1;
17: selfFlag = 1;
18: }
19: __syncthreads();
20: if (selfFlag == 1 &&
21: flag[threadIdx.x] == 0) {
22: result = threadIdx.x;
23: }
24: __syncthreads();
25: // threads traverse to the next node
26: nodeIndex = (nodeIndex + result)<<3;
27: }
28: // nodeIndex is the index of target leaf
        node
```

[§]This program is executed by eight threads concurrently with `threadIdx.x` = 0 to 7

[†]Shared variables are declared by `__shared__`

[‡]`__syncthreads` is a barrier synchronization primitive

bandwidth, shared memory is composed of multiple memory banks which can be accessed simultaneously. The highest bandwidth is achieved when accesses are equally separated among the memory banks. For inter-block communication, shared memory is the better option in comparison to the device memory due to lower access latency.

The Nvidia Tesla platform is explicitly targeting high performance computing and offers faster double precision floating point operations which is a critical requirement for many applications. However, we opted for the Nvidia Geforce processor family in this work, since index tree search algorithms only require integer operations, either 64-bit or 32-bit, and the Geforece platform offers a better computation-power-to-price ratio for these operations.

D. GPU SEARCH KERNEL

The Snippet 3 represents the GPU kernel function for searching the I-segment of the implicit HB$^+$-tree. The input parameters are `I-seg`: the reference to the I-segment in GPU memory, `levelOffsets`: offsets of each level in I-segment, and `teamQuery`: the given search query. Shared variables are declared by the keyword `__shared__` and barriers `__syncthreads` are used to avoid race conditions when accessing shared variables.

Search starts from the root node (`nodeIndex = 0`) and performs the parallel node search per each inner node. Each thread loads a key from the current node and stores it into local register (`selfKey`) in Line 12. After initializing flags, threads compare their local register to `teamQuery` and store the result in both local and shared flags. In Line 19, it is required to synchronize threads before they check the shared flag to avoid race conditions. In Lines 20-23, each thread deduces if it is assigned to the next level node. If so, the thread update the shared result variable. The operation is repeated for each level of inner nodes. At the end, `nodeIndex` is referring to the desired leaf node.