Performance of Point and Range Queries for In-memory Databases using Radix Trees on GPUs

Maksudul Alam*, Srikanth B. Yoginath†, and Kalyan S. Perumalla†
*Department of Computer Science, Virginia Tech, Blacksburg, VA
†Oak Ridge National Laboratory, Oak Ridge, TN
Email: maksud@vbi.vt.edu, yoginathsb@ornl.gov, perumallaks@ornl.gov

***Abstract*—In in-memory database systems augmented by hardware accelerators, accelerating the index searching operations can greatly increase the runtime performance of database queries. Recently, adaptive radix trees (ART) have been shown to provide very fast index search implementation on the CPU. Here, we focus on an accelerator-based implementation of ART. We present a detailed performance study of our GPU-based adaptive radix tree (GRT) implementation over a variety of key distributions, synthetic benchmarks, and actual keys from music and book data sets. The performance is also compared with other index-searching schemes on the GPU. GRT on modern GPUs achieves some of the highest rates of index searches reported in the literature. For point queries, a throughput of up to 106 million and 130 million lookups per second is achieved for sparse and dense keys, respectively. For range queries, GRT yields 600 million and 1000 million lookups per second for sparse and dense keys, respectively, on a large dataset of 64 million 32-bit keys.**

I. Introduction

A. In-Memory Databases

In the age of big data, modern data systems need to support high throughput, low latency operations. Of late, main memory capacity is adequate to store the entire contents of some databases in the computer's main random access memory (RAM). In line with Jim Gray's observation, "Memory is the new disk, disk is the new tape", RAM has become sufficiently inexpensive to make in-memory databases a viable choice for large data-intensive enterprise applications [1]. Responding to these trends, commercial in-memory systems have emerged, such as HyPer [2], SAP HANA [3], and Microsoft Hekathon [4], to name a few. These systems are heavily used in data-intensive tasks in the fields of information retrieval, machine learning, data mining and analysis [5]–[7].

Index search is one of the most critical functions of an in-memory database. It has been shown that the greatest amount of time used for a query operation by an in-memory database system is the time spent on searching the index [8]. Therefore, the best performance of such a database system depends on efficient index searching. The efficiency of index searching largely depends on the choice of index data structures. Traditional disk-optimized schemes for index searching are either inefficient or not easily generalized for different search types. For example, binary search trees are inefficient [9], and hash-tables cannot support range queries.

Modern computer systems include accelerators such as graphical processing units (GPUs) to offload computationally intensive work from main processors that offer great potential for fast index searching. Because such GPU accelerators are specialized for compute-intensive, data-parallel computation compared to traditional multi-core CPU systems, some data intensive tasks have been ported to GPU platforms in the past [10], [11]. In the same fashion, index searching operations can exploit a GPU platform to improve bulk index lookup if appropriate algorithms are designed and implemented efficiently.

B. Index Search Approaches

Indexes are used in modern database systems to facilitate faster lookups of keys for data. In these systems, a list of $\langle key, id \rangle$ tuples are organized in such a way that, given a specific key $key(i)$ or a range of keys $[key(i), key(j)]$, the index search subsystem returns the corresponding identifier(s) $id(s)$ of the key(s). The goal of any index searching algorithm is to deliver the identifiers as quickly as possible, corresponding to a given set of keys.

Broadly speaking, there are two main approaches to fast index searching: (1) hash-based approaches and (2) tree-based approaches. Within the latter, an important sub-category is the radix tree-based approach.

Hash-based Systems. Hash-based systems are mainly used in key-value storage systems such as Memcached [12], Redis [13], and RAM-Cloud [14]. Hash-based systems are very efficient and perform well for point queries. However, these systems do not support range queries, which is a crucial functionality in many systems such as finding the transactions between two dates or finding the top k values within some range. These systems also do not have good cache utilization in general [15].

Tree-based Systems. Tree-based systems, such as B+-tree, have been the de-facto standard for traditional disk-based databases [16] designed to obtain the best I/O throughput. However, the B+-tree does not utilize the increasing cache sizes properly and is deemed unusable for in-memory systems [9]. Consequently, many variations of B+-tree have been proposed, such as the T-Tree [17], Cache Sensitive Search Trees (CSS-Trees) [18], Cache Sensitive B+-Trees (CSB+-Trees) [19], Δ-Tree [20], BD-Tree [21], Fast Architecture Sensitive Tree (FAST) [22], and Bw-tree [23]. Although these systems achieve better cache utilization, they are computationally expensive. Furthermore, the order of comparison operations in

tree structures is harder to predict and results in undesirable stalling in processor pipelines [9].

Radix Tree-based Systems. Radix trees, such as the Judy Tree [24] and the Adaptive Radix Tree (ART) [9], do not rely on hashing or comparison. Instead, they create index trees based on the digital representation of search keys. One key property of searching with the radix tree is that the search operation requires $\mathcal{O}(k)$ operations, where k is the length of the key, and it is independent of the number of keys to index. A radix tree has many advantages over comparison-based trees:

- The height of a radix tree depends on the length of the keys, not on the number of keys;
- Keys are kept in a lexicographic order, therefore any order of insertions results in the same tree;
- Keys can be retrieved in sorted order, allowing fast range queries; and
- Keys are not stored explicitly, thus giving memory savings, yet can be re-constructed by tree traversal.

C. Organization

The rest of the paper is organized as follows. The details of GPU-based radix tree implementation for index searching is discussed in Section II. Benchmarks for the performance evaluation are discussed in Section III. Experimental results are presented in Section IV. Finally, we summarize and conclude in Section V.

II. GPU-based Radix Tree for Index Searching

To the best of our knowledge, radix-tree based index searching system has not been implemented and/or evaluated on the GPU. In this paper, we implement and evaluate the performance of a radix tree-based index searching on the GPU. Furthermore, our implementation also supports range queries on the GPU, for which we identify radix trees as ideally suited. We present GRT (GPU-based Radix Tree), an efficient index searching implementation based on radix trees customized and optimized for single instruction multiple data (SIMD) operation, different key lengths, various key distributions, and point queries as well as range queries.

The adaptive radix-tree (ART) is the most efficient radix-tree based index searching system in the CPU [9]. ART has been demonstrated to be very memory efficient and yields high lookup throughput [9]. Our basic approach builds on the structures used by ART and adapts them for modern GPUs. A comprehensive performance study of GRT is presented here using benchmarks that index some data sets containing millions of keys. We also compare the runtime performance against other existing competitive CPU and GPU-based methods. Performance evaluation demonstrates the high throughput achieved by our implementation, clocking over 100 million lookup operations per second on large data sets of over 64 million 32-bit keys. We provide benchmarks and detailed runtime data to support the results and findings.

A. Radix Tree Nodes

A radix tree has two types of nodes: inner nodes and leaf nodes. Inner nodes map partial keys to other nodes. Each leaf node represents a distinct key and stores the index value of the corresponding key. A lookup operation on a radix tree starts from the root. For each inner level, a partial key is extracted from the given key, and it is used to navigate to the child node of the next level. This process is repeated until a leaf node is reached. Therefore the lookup operation depends on the number of levels of the radix tree.

The number of levels of a radix tree depends on the size of the partial keys. If the size of a partial key is s and the size of a key is k, then the radix tree requires $\lceil \frac{k}{s} \rceil$ levels. For example, for 32-bit keys, if the size of a partial key is 4, then it would require 8 levels. Therefore, for faster lookup, it is desirable that the length of the partial key is large.

For a partial key of size s-bits, an inner node may store up to 2^s child pointers. In conventional radix trees, all inner nodes store the same number of pointers, that is, all inner nodes have 2^s pointers. When most of the child pointers are not used, the space required by conventional radix trees can be excessive [9]. To reduce the space required and utilize the benefits of a radix tree, Leis et al. proposed an adaptive radix tree (ART) [9], which adaptively uses different sizes of inner nodes with the same partial key size. In their implementation, they used 8-bit partial keys and four different types of inner nodes with fan out of 4, 16, 48, and 256 branches. On CPUs, ART has been demonstrated to deliver high throughput lookups in high performance in-memory database systems such as HyPer [2]. However, adaptive radix trees have never been implemented or evaluated on GPUs. To exploit the performance gain from adaptivity, in our implementation, we build on adaptive radix trees as our index searching data structure and map them to the GPU architectural elements characterized by memory idiosyncrasies and single instruction multiple data (SIMD) processing style.

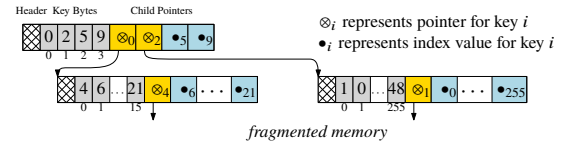
We implemented index searching on modern GPU architectures, with support for three basic operations of index searching: (1) inserting $\langle key, id \rangle$ tuples in a bulk mode, (2) performing the search for a single given key $\langle key \rangle$, a list of keys $\langle key_1, key_2, \ldots key_n \rangle$, or a range of keys $\left[key_{\min}, key_{\max}\right]$ and (3) modifying any given tuple $\langle key, id \rangle$ by changing the id of an already present key.

B. Tree Interface and Serialization

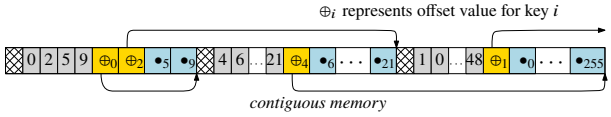
Before using the radix tree for searching, the tree has to be built from the keys and stored. The tree can be built with the CPU in main memory or by the GPU in GPU (device) memory. Building the radix-tree in the CPU is relatively easier as the CPU efficiently deals with dynamic memory allocation. After the radix-tree is built on the CPU, it must be transformed into another memory efficient structure, suitable for efficient GPU lookup operations. The original ART structure uses memory pointers, which are inefficient to use in a GPU. For efficient memory accesses, we use offset values instead of pointers as shown in Fig. 1. The offset value denotes how far the

child is located from the start of the pointer. Therefore, a tree traversal operations ($\texttt{root} \rightarrow \texttt{child}[i]$) becomes the simpler ($\texttt{root} + \texttt{offset}$) operation on the GPU, where `root` is the starting point of the tree. The tree is serialized using a depth-first search (DFS) traversal into a contiguous byte array for efficient lookup operations on the GPU. Therefore, in our first implementation, the main instance of the radix tree is stored in main memory and a GPU-efficient copy is stored in a contiguous chunk of the GPU memory. However, whenever a new key is inserted in the radix-tree, we have to serialize the structure again and copy it to the GPU memory. For applications where insertions and deletions are frequent, this may take a large amount of time. To avoid this, we implemented another efficient version of GRT where the radix-tree is built only using the offset-based structure on a pre-allocated memory. In this case, we allocate a large chunk of GPU memory in advance and build the radix-tree in this chunk of memory. We can determine the approximate amount of memory required to store the radix-tree based on the key distribution as shown in the experimental evaluation section.

For efficiency, the full tree is copied to the GPU memory. As the first level of the tree is always accessed for any lookup query, many concurrent lookups would cause memory contentions. Such memory contentions are avoided by caching the first level of the tree in the shared memory at the block level. The individual threads work on different parts of memory and, therefore, do not have contentions. Since the shared memory of a GPU is significantly faster than the global memory, the lookup time is reduced.



(*a*) Original Radix Tree



(*b*) Serialized Radix Tree

Fig. 1. Serializing the radix tree for GPU

C. Query Types

There are two main query categories that require index searching support.

Point Queries. The most typical index searching operation in a database system is to find the index from a single key, known as point queries. When tree-based structures are used for point query searching, the average runtime for a search operation is proportional to the depth of the tree. For faster searching, hash tables are often used, which have an amortized $\mathcal{O}(1)$-time search. The actual performance of the hash tables is highly dependent on the table structure and hash functions [9], [25]. Another popular method for point queries is the radix-tree which offers $\mathcal{O}(k)$-time search operations, where k is the size of the key.

Range Queries. A range query is a searching operation that retrieves all indices where the value of the key is between an upper and lower boundary. Range queries appear in many applications, such as using timestamps as keys and finding the records contained between two timestamps.

Many indexing systems that have been tuned to deliver fast point queries are unsuitable for range queries. For example, hash-based systems support fast point queries, but they have limited support for range queries. One simple way to perform range queries is to repeat the point query operation for all keys in the range. This may yield acceptable performance where the range is small, but for large ranges, this will not do well because it wastes computation in the form of many duplicate traversals of the tree. Tree-based systems such as comparison-based and radix-based systems, use an ordering of keys and, therefore, support efficient range queries.

In many cases, such as for finding names or phone numbers with a common prefix, a variation of range queries called prefix queries is also used. As the name suggests, prefix query searches for the indices of keys with a common prefix. Hash-based and comparison-based tree systems are not well suited for prefix queries. Only radix tree-based systems natively support prefix queries.

III. Benchmarks

To evaluate the performance of index searching on GPUs, we vary the key sizes, key distributions, and lookup workload patterns. In this section, an overview of the key and lookup benchmarks is provided, followed by additional details of the different lookup benchmarks.

A. Overview

The following are the key and lookup variants used for experiments in the performance study.

- **Key Variants.** The lookup operations of any radix-based tree is dependent on the key structure. In this work, we considered multiple variants of the key structure in terms of lengths and values. We considered both synthetic and real-world key values to measure the performance of GRT.
  GRT is capable of handling any key length. However, for the purposes of the performance study, we considered three commonly used key lengths in our experiments: 4-byte, 8-byte, and 16-byte keys. Most of the previous works on index searching supported 4-byte keys for benchmarking; therefore, we follow the convention and employ 4-byte keys for the benchmarking. We also used 8-byte keys for benchmarking to compare with [8], which uses 8-byte keys. Many real-world keys are larger than these sizes. To measure the performance of real-world keys on the GPU, we also evaluated 16-byte keys. We

varied the number of keys across experimental scenarios. Four distinct sets of keys were considered in this work: 64 thousand, 1 million, 16 million, and 64 million keys. All sizes are similar to ones covered in the prior literature [8], [9], [22]. A summary of the keys used in the experiments is shown in Table I.

TABLE I
DATA SETS USED IN THE EVALUATION

Key Type	# of keys	Length
Sparse	64**K**, 1**M**, 16**M**, 64**M**	4 Bytes
Dense	64**K**, 1**M**, 16**M**, 64**M**	4 Bytes
ISBN	7**M**	8 Bytes
Music Id	9**M**	16 Bytes

- **Lookup Variants.** To measure the performance of GRT, multiple benchmarks are considered. Three types of lookup operations are covered in these benchmarks: (1) a singular lookup for a key, (2) a bulk lookup for an array of keys, and (3) a range lookup between two keys. Singular lookup is useful in applications residing completely in the GPU. Bulk lookup is useful in applications residing either on a GPU or a CPU. Singular lookup provides the best latency while bulk lookup provides the best throughput. Range lookup is used in many database applications, and it can be effectively utilized on both CPU and GPU. A summary of all the benchmarks is provided in Table II.

TABLE II
LIST OF BENCHMARKS

Code	Description
`1S`	**Bulk** (**1**) lookup with ***Sparse*** keys
`1D`	**Bulk** (**1**) lookup with ***Dense*** keys
`1A`	**Bulk** (**1**) lookup with 8-byte (**A**) *real-world* keys
`1B`	**Bulk** (**1**) lookup with 16-byte (**B**) *real-world* keys
`2S`	**Singular** (**2**) lookup with ***Sparse*** keys
`2D`	**Singular** (**2**) lookup with ***Dense*** keys
`3S`	**Hierarchical** (**3**) lookup with ***Sparse*** keys
`3D`	**Hierarchical** (**3**) lookup with ***Dense*** keys
`4S`	**Range** (**4**) lookup with ***Sparse*** keys
`4D`	**Range** (**4**) lookup with ***Dense*** keys

The main performance metric used here is the throughput, measured as the number of lookups completed per second. Average time per lookup can be readily inferred as the inverse of the throughput.

B. Lookup Benchmarks

- **Bulk Lookup Benchmarks.** The bulk lookup is used to compare the performance of GRT with other efficient implementations reported in the literature. In the bulk lookup, the indices to be looked up are provided in an array of keys. The following variations of these bulk lookup arrays are considered: (1) *Sparse distribution*, where each key is unique and chosen uniformly at random from $[1, 2^k)$ where k is the size of the key in bits, and (2) *Dense distribution*, where every key is in the range $[0, 1, 2, \ldots, n-1]$ where n is the number of keys to index. Further, in our performance evaluation with the bulk lookup benchmarks, we have also exercised two real-world keysets: (1) ISBN numbers from a book database [26] and (2) Music Identifiers from a music database [27].

- **Singular and Hierarchical Lookup Benchmarks.** In many GPU-based parallel applications, the GPU threads independently perform many lookup operations to fetch data. The singular lookup benchmarks evaluate the lookup performance of such systems. In this benchmark, each GPU thread independently generates many random keys and performs point queries for each key. We also use both sparse and dense keys in this experiment.
  So far, the keys are considered to be unrelated to each other. But, in many applications, such as genealogy [28], communication threads in social network analysis, and phylogenetic trees [29], the keys have a hierarchical relationship resulting into self-referenced foreign keys. Such applications require the information or data from its parents leading up to the root of the tree. In such cases, an efficient lookup is needed to trace back the parent using a single composite lookup rather than having multiple singular lookups to achieve the same. To evaluate such scenarios, we introduce an additional benchmark that exercises the hierarchical lookups. In this benchmark, let $\mathcal{K}$ be the set of keys. Then, for any key $k \in \mathcal{K}$, we have a parent key $P(k) \in \mathcal{K}$. The root key does not have a parent key. Since we use an m-ary tree, each key has m child keys. Given a key $k \in \mathcal{K}$, the hierarchical benchmark traces every node in the path back to the root of the m-ary tree.

- **Range Lookup Benchmarks.** In a range lookup, when the user provides a pair of keys $k_{\min}$ and $k_{\max}$, all the indices of keys k are returned such that $k_{\min} \leq k \leq k_{\max}$, where the size of range r is defined as $r = k_{\max} - k_{\min} + 1$. In this experiment, the number of keys is kept to the maximum available and the range size is varied. We consider both the *sparse distribution* and the *dense distribution* of keys, as mentioned earlier. We also measure the performance of range search as the number of range queries processed per second and also in terms of throughput as the number of indices retrieved per second.

IV. PERFORMANCE STUDY

In this section, we evaluate the performance of GRT under different workloads and configurations and observe that GRT achieves excellent throughput for all workloads.

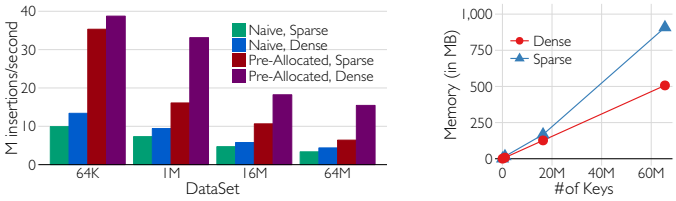
In the experiments, the CPU is a *Xeon E5* with a 2.5 GHz clock and a memory of 16 GB with a clock rate of 2.1 GHz with a peak bandwidth of 68 GB/s. The GPU is an *nVidia Tesla K80* with processor clock rate of 562MHz, memory rate of 2.5

GHz, and peak bandwidth of 240 GB/s. The operating system is `Ubuntu 12.04.5 LTS`, and all software on this machine was compiled with GNU `gcc 4.6.3` with optimization flags `--O3`. The `CUDA compilation tools V6.5` were used for the GPU code along with the `nvcc` compiler. All CPU based experiments were performed using a single core.

A. GPU Overhead

Here we present performance results regarding the creation and serialization performance of GRT. As mentioned earlier, we use two approaches to building the GPU radix tree. In the naive approach, the pointer based radix-tree is built on the CPU and then serialized and copied to the GPU memory. In the pre-allocation based approach, we allocate a big chunk of memory in the GPU and directly build the radix tree on that memory, which supports further insertions and deletions of indices in the GPU. However, our experiments suggest that building the tree only using the GPU memory takes more time than using the CPU memory. For example, creating a radix tree with $16M$ sparse keys require about 1.54 seconds using the CPU memory, whereas using only the GPU memory requires about 16 seconds. Therefore, we use a hybrid approach, where the initial pointer less offset-based radix-tree is built in the CPU and copied to the GPU memory. Future updates to the radix-tree could be performed directly in the GPU memory without using the CPU.

Insertion Throughput. Fig. 2a demonstrates the insertion throughput of building the radix tree using both approaches for sparse and dense keys. Note that the insertion throughput of the naive approach is similar to the one reported in [9]. The pre-allocate based hybrid approach yields better throughput than the naive scheme for all data sets. Also, note that dense keys require less time to build the tree.



(a) Insertion Throughput (b) Required Memory

Fig. 2. Replicating Radix Tree for the GPU

Memory requirement. Fig. 2b shows the memory required to store the radix tree in the GPU. The required memory increases linearly with the data size. Adaptive radix trees use lazy expansion; for this reason, dense keys expand more slowly and have better space efficiency. For 64M entries, about 900 MB of GPU memory is required for sparse keys, which is about 14 bytes per search key. However, for dense keys, only 531 MB of GPU memory is required, which is about 8 bytes per key. For the pre-allocation based approach, we use 16 and 8 bytes per sparse and dense key, respectively.

B. Bulk Lookup

We performed bulk lookup operations on the sparse and dense keys[1] from the data sets listed in Table I. The index searching on the CPU with adaptive radix trees was performed using the unaltered source code of the ART system [9]. We used 512 threads for the GPU implementations. In each experiment, more than 100 million lookup operations were performed, in order to eliminate noise. The results of the experiments are shown in Fig. 3a.

From the figure, it is clear that the dense key lookups have better throughput in most of the cases. With the increase in the number of keys, sparse keys become dense, and this advantage diminishes. For example, for 64K dense keys, only two-byte levels are sufficient for all the search operations (when bytes are accessed in little endian format), whereas many sparse keys would require more than two levels. Therefore, the performance of dense keys is significantly better. For 64 million entries, we achieve a throughput of about 100 million lookups per second for the sparse keys and 130 million lookups per second for the dense keys.

C. Bulk Lookup with Real-World Keys

To test the performance of GRT, we performed bulk lookup operations on real world keys. We collected two sets of keys from two popular open source databases. The first set of keys comes from `OpenLibrary.com` [26] containing approximately 7 million book entries indexed by ISBN-13 numbers. A 13-digit ISBN number fits into 8 bytes. We also collected a set of keys from the `MusicBrainz.com` [27] database, which contains approximately 9 million song entries. Each entry has a 16-byte id.

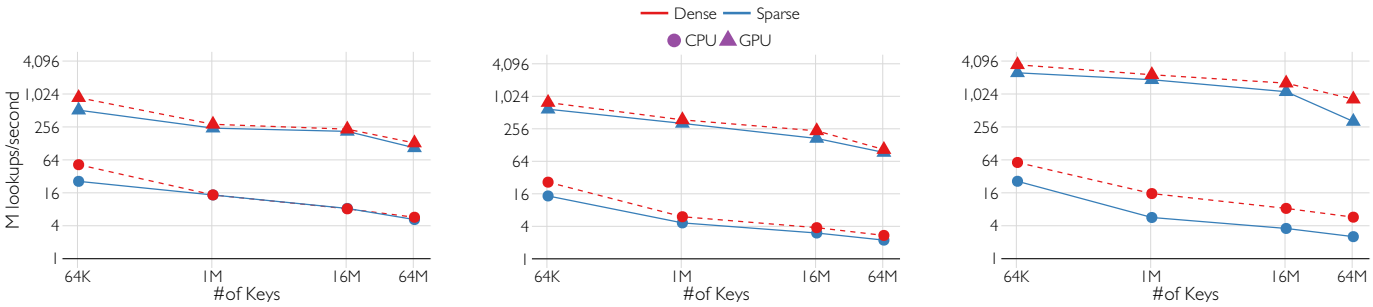
Fig. 4 exhibits the throughput and required memory for the lookup operations. As observed from the figures, we achieved a throughput of 258.34 million lookups per second on the Tesla K80 GPU.

ISBN-13 keys are structured, so that the first three digits represent a prefix (either 978 or 979) as per the standard ISBN specification, the next two digits represent the registration group element, the four digits next to it represent the publisher identifier, and the title and check digit is represented by the next three digits and one digit respectively. For these kinds of structured keys, the radix-based approach is well suited.

We expect similarly suitable lookup behavior for analogous key structures found in many other domains such as international article number (EAN) barcodes, electronic product codes (EPC), stock keeping unit (SKU), social security numbers (SSN), and telephone numbers.

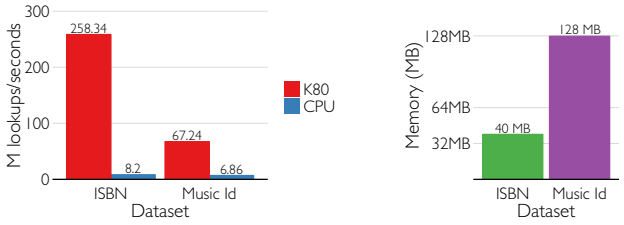
The music-identifying keys are randomly generated unique identifiers, and the distribution of keys in the lookups is very sparse. As a result, we observe approximately 67 million lookups per second. Nevertheless, GRT still performs better than the CPU version, providing a speedup of approximately $10\times$.

[1] dense keys are shuffled randomly for the experiments.



(a) Bulk Lookup (b) Singular Lookup (c) Hierarchical Lookup

Fig. 3. Bulk, Singular, and Hierarchical lookup throughputs for 4-byte keys



(a) Throughput (b) Required GPU Memory

Fig. 4. Bulk lookup for ISBN (8 Bytes) and Music Id keys (16 Bytes)

D. Singular and Hierarchical Lookup

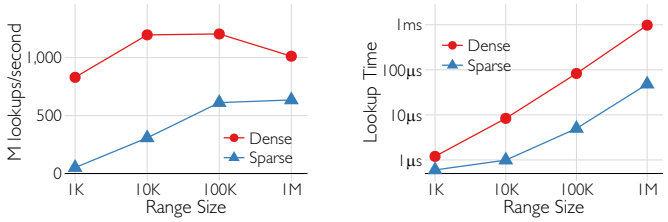
In the singular lookup benchmark, each GPU kernel thread generates a random key and performs a point query. In this experiment, each GPU thread performs millions of point queries to eliminate noise. Fig. 3b exhibits the singular lookup throughputs for both sparse and dense keys. The throughputs are similar to the bulk lookup because the singular lookup simply generates random keys instead of using a given array in the bulk lookup.

In the hierarchical lookup benchmark, we assume that there is an m-ary parent-child relationship among the keys. Therefore, each parent has m children keys. In this experiment, we set the value $m = 10$. Each GPU thread randomly selects a key and repeatedly performs searches towards the parent, until the root key is reached. An m-ary tree has depth $\mathcal{O}(\log_m N)$ for N keys. Hence, in the worst case, there would be $\mathcal{O}(\log_m N)$ such search operations per key. To identify how the hierarchical relationship affects the search performance, we investigate yet another lookup variant. In this variant, each GPU thread selects a random key and searches the index of the key.

Fig. 3c shows that hierarchical lookup achieves about two times greater throughput than the singular lookup operations. This behavior is observed across both CPU and GPU as a similar trend. The main reason is the way searches are performed. The keys closer to the root are searched more often than the keys that are closer to the leaves. Therefore, they are stored in the cache for a longer time, effectively improving the throughput.

E. Range Lookup

In this experiment, we evaluate the performance of range queries for GRT. To the best of our knowledge, there is no range query implementation on the GPU. All the experiments were performed with 64 million keys. We used both sparse and dense keys for this experiment. Here, we used a set of four range sizes $\mathcal{R} = \{10^3, 10^4, 10^5, 10^6\}$. In each GPU thread, a random key k is picked, and a range query operation is performed over the range $[k, k + r - 1]$ for each $r \in \mathcal{R}$.

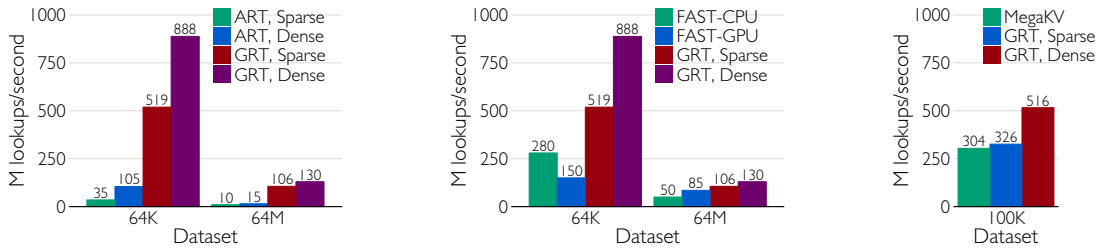


(a) Lookup Throughput (b) Time Required

Fig. 5. Performance of range queries

Fig. 5a shows the throughput performance of range queries on the GPU. As the figure shows, the throughput of dense keys is much higher than the corresponding throughput of sparse keys. The range queries are implemented using depth first search tree traversal. For any given range, the inner nodes traversed by the algorithm for dense and sparse keys are approximately the same. However, as the dense distributions contain more keys (leaf nodes) than the sparse distributions in the same range, dense distributions yield better throughput. Also observe that for sparse keys, throughput increases when the range is increased, but for the largest range (1M) the throughput does not increase much. A similar trend is also observed in dense keys, but in this case, the throughput decreases by a significant amount for the largest range. This is due to the fact that, the traversal algorithm has to process more inner nodes compared to the number of leaf nodes. For range size of 10^5, range query operation processes an average 394 inner nodes for dense keys, whereas, for range size of 10^6, the operation requires processing 3925 inner nodes.

Fig. 5b exhibits the time required to perform a single range query by our algorithm. The number of range queries



(a) ART (b) FAST (c) MegaKV

Fig. 6. Comparing GRT with other systems

per second decreases with increasing range size as expected. However, range queries for sparse keys run much faster. This is due to the fact that, sparse distributions contain fewer keys in a given range as compared to dense distributions and, therefore, have to process fewer leaf nodes.

As seen from the above figures, our algorithm is capable of performing fast range queries with high throughput, which is not possible for hash-based systems.

F. Update Operations

Our pre-allocation based approach provides support for delete, modify, and insert operations directly on the GPU. However, if one wants to update/insert a lot of indices, the CPU should be used in conjunction with the GPU. Our experiments show that for an already constructed $16M$ sparse keys, we achieve an update throughput of 0.8 million index operations per second using only the GPU, whereas along with the CPU yields a throughput of 8 million index operations per second.

G. Comparing GRT with Other Index Searching Systems

We compare the throughput of GRT with existing CPU and GPU based index searching systems. Here we only consider ART [9], FAST [22], and Mega-KV [8]. Due to the unavailability of source codes of these systems, we use the best performance numbers from their corresponding articles.

We show the performance results on three data sets with 64K, 100K, and 64M keys. The 64K and 64M data sets consist of 4-byte keys; they are used in performance evaluation on both ART and FAST. Therefore, the comparison remains fair. We also consider both the sparse and dense key distribution as was done in the previous literature. The 100K keys with a key length of 8-byte data set is used specifically to compare with Mega-KV, considering that the largest block size in Mega-KV is also the same.

GRT vs. ART. Fig. 6a shows the throughput comparison between ART and GRT. GRT obtained a magnitude of $15\times$ and $8\times$ higher throughput over ART for sparse and dense keys, respectively. We also observe a similar pattern for sparse and dense keys, as this pattern emerges because of the radix-tree structure.

GRT vs. FAST. Fig. 6b exhibits the throughput comparison between GRT and FAST (both CPU and GPU). For smaller key sizes, the CPU implementation of FAST is faster than the GPU version because of caching effects. Both the sparse and dense key versions of GRT have approximately two to three fold performance gain over the FAST implementations. For larger key sizes, such as 64M keys, however, the performance of GRT decreases rapidly compared to the FAST version. Nevertheless, the actual throughput of GRT always remains better than that of FAST. Note that the comparison may not be entirely fair here: as the source codes for FAST on the GPU is not available online, we were unable to determine how FAST would perform in modern GPUs.

GRT vs. Mega-KV. Mega-KV is the latest GPU-based system that uses a hash table for index searching. Fig. 6c shows the throughput comparison between GRT and Mega-KV. For 100K keys, both systems have similar throughput, with GRT performing better than Mega-KV by a small margin. Note that both the corresponding GPUs for GRT and Mega-KV have almost similar FLOPs and memory bandwidth, making the comparison fair.

Performance across different GPUs. In additional experiments, we used three different GPUs that differ in their capacities in terms of processor clock rate, memory capacity, and peak GFLOPs. The GTX 580 model was released in 2010, the Tesla K20X in 2012, and the Tesla K80 in 2014. The only common feature among the models is the bandwidth, as the bandwidth of GPUs has not increased significantly over the years. We obtained nearly similar throughput rates from all of the GPUs in the bulk lookup performance. The same phenomenon is observed in Fig. 3. Although the throughput is almost doubled for the singular lookup performance, this improvement happened in every GPU. Therefore, the difference in GPU architecture is not contributing significantly in index searching. This is probably because the index searching systems require extensive memory access operations and are memory-bound by nature. Therefore the determining factor for index searching operations seems to be the bandwidth. Also, note that dense keys use less memory and therefore perform better.

V. Conclusion and Future Work

In this work, we developed an efficient GPU-based Radix Tree (GRT) for index searching with high throughput on

GPUs. We serialized the radix tree to a low memory footprint data-structure to achieve extremely efficient lookup operations on GPUs. We augmented additional index searching performance benchmarks to the set of well-known existing ones and evaluated GRT on all the benchmarks. The performance study included a comprehensive analysis, taking into account the diversity of keys, lookup operations, and runtime effects with real-world data. We compared GRT to the most recent and the best available index searching systems. Our experiments indicate that GRT meets or betters the performance of other index searching systems currently available both on the CPU and the GPU. For a large data set of 64 million keys, GRT achieved a throughput of 106 million lookups per second for sparse keys and 130 million lookups per second for dense keys. For the same data set, excellent range query performances are achieved, yielding over 1000 million lookups per second for sparse keys and over 600 million lookups per second for large range sizes for dense keys.

Since index searching is a bottleneck for databases, especially for main memory systems, GRT can be directly applied to such systems to achieve significant performance gains. Additionally, it is also possible for GRT to be extended to high-performance key-value storage systems, where values are stored in memory in conjunction with the keys.

In our study, the implementation was restricted to a single GPU for storing and managing index structures that fit within a GPU's memory. In future, we plan to explore index searching operations on larger data sets with a greater number of keys on multiple GPUs.

Acknowledgment

This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Dept. of Energy. Accordingly, the U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

References

[1] S. Robbins, "Ram is the new disk..." Online, 2008.
[2] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *IEEE Data Engineering*, 2011.
[3] F. Färber, S. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: Data management for modern business applications," *SIGMOD Rec.*, 2012.
[4] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL server's memory-optimized OLTP engine," in *ACM Management of Data*, 2013.
[5] M. Steinbrecher and J. Boese, *Real-Time Data Mining with In-Memory Database Technology*. Springer, 2013, pp. 275–284.
[6] D. Schwalb, M. Faust, J. Krueger, and H. Plattner, "Leveraging in-memory technology for interactive analyses of point-of-sales data," in *IEEE Data Engineering Workshops*, 2014.
[7] K. Herbst, C. Fähnrich, M. Neves, and M.-P. Schapranow, "Applying in-memory technology for automatic template filling in the clinical domain," in *CLEF Evaluation Labs and Workshop, Online Working Notes*, 2014.
[8] K. Zhang, K. Wang, Y. Yuan, L. Guo, and R. Lee, "Mega-KV: a case for GPUs to maximize the throughput of in-memory key-value stores," in *VLDB Endowment*, 2015.
[9] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *IEEE Data Engineering*, 2013.
[10] R. Wu, B. Zhang, and M. Hsu, "GPU-accelerated large scale analytics," *IEEE/ACM UnConventional High Performance Computing*, 2009.
[11] J. Canny and H. Zhao, "Big data analytics with small footprint: Squaring the cloud," in *ACM Knowledge Discovery and Data Mining*, 2013.
[12] B. Fitzpatrick and A. Vorobey, "Memcached: a distributed memory object caching system," Online, 2003. [Online]. Available: http://memcached.org/
[13] S. Sanfilippo and P. Noordhuis, "Redis," Online, 2009. [Online]. Available: http://redis.io
[14] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for DRAM-based storage," in *USENIX Conf. on File and Storage Tech.*, 2014.
[15] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent memcache with dumber caching and smarter hashing," in *USENIX Conf. on Networked Systems Design and Implementation*, 2013.
[16] D. Comer, "Ubiquitous B-tree," *ACM Computing Surveys*, 1979.
[17] T. Lehman and M. Carey, "A study of index structures for main memory database management systems," in *Very Large Data Bases*, 1986.
[18] J. Rao and K. Ross, "Cache conscious indexing for decision-support in main memory," in *Very Large Data Bases*, 1999.
[19] J. Rao and K. A. Ross, "Making B+-trees cache conscious in main memory," in *ACM Management of Data*, 2000.
[20] B. Cui, B. C. Ooi, J. Su, and K.-L. Tan, "Contorting high dimensional data for efficient main memory KNN processing," in *ACM Management of Data*, 2003.
[21] ——, "Main memory indexing: the case for BD-tree," *IEEE Trans. Knowl. Data Eng.*, 2004.
[22] C. Kim, J. Chhugani, N. Satish, and E. Sedlar, "FAST: fast architecture sensitive tree search on modern CPUs and GPUs," in *ACM Management of Data*, 2010.
[23] D. B. Lomet, S. Sengupta, and J. J. Levandoski, "The Bw-tree: A B-tree for new hardware platforms," in *IEEE Data Engineering*, 2013.
[24] D. Baskins, "Judy arrays." [Online]. Available: http://judy.sourceforge.net/
[25] V. Alvarez, F. M. Schuhknecht, J. Dittrich, and S. Richter, "Main memory adaptive indexing for multi-core systems," in *Works. on Data Management on New Hardware*, 2014.
[26] OpenLibrary, "Open library data dumps." [Online]. Available: https://openlibrary.org/developers/dumps
[27] MusicBrainz, "Musicbrainz database / download." [Online]. Available: https://musicbrainz.org/doc/MusicBrainz_Database/Download
[28] F. Chen, A. J. Mackey, C. J. Stoeckert, and D. S. Roos, "OrthoMCL-DB: querying a comprehensive multi-species collection of ortholog groups," *Nucleic Acids Res.*, vol. 34, no. 1, pp. D363–D368, 2006.
[29] G. Perrière and M. Gouy, "WWW-query: An on-line retrieval system for biological sequence banks," *Biochimie*, vol. 78, no. 5, pp. 364 – 369, 1996.