CuART - a CUDA-based, scalable Radix-Tree lookup and update engine

Martin Koppehel
Thilo Pionteck
koppehel@ovgu.de
pionteck@ovgu.de
Institute for Information Technology and Communications
Otto-von-Guericke Universität Magdeburg
Germany

Tobias Groth
Sven Groppe
groth@ifis.uni-luebeck.de
groppe@ifis.uni-luebeck.de
Institute of Information Systems (IFIS)
Universität zu Lübeck
Germany

ABSTRACT

In this work we present an optimized version of the Adaptive Radix Tree (ART) index structure for GPUs. We analyze an existing GPU implementation of ART (GRT), identify bottlenecks and present an optimized data structure and layout to improve the lookup and update performance. We show that our implementation outperforms the existing approach by a factor up to 2 times for lookups and up to 10 times for updates using the same GPU. We also show that the sequential memory layout presented here is beneficial for lookup-intensive workloads on the CPU, outperforming the ART by up to 10 times. We analyze the impact of the memory architecture of the GPU, where it becomes visible that traditional GDDR6(X) is beneficial for the index lookups due to the faster clock rates compared to High Bandwidth Memory (HBM).

KEYWORDS

in memory databases, index structures, databases, radix trees, ART

ACM Reference Format:
Martin Koppehel, Thilo Pionteck, Tobias Groth, and Sven Groppe. 2021. CuART - a CUDA-based, scalable Radix-Tree lookup and update engine. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472511

1 INTRODUCTION

The increased demand for deep analysis of huge databases puts new challenges on several components within a modern database system. New approaches in the query interface have been widely evaluated and partially adopted within NoSQL databases [18], as well as optimizations for the storage backends [9]. The index structures used to find specific records within such a database were steadily optimized, but there were no significant improvements in order to speed up index construction and querying. This is particularly problematic since the available main memory for typical server systems increased from a few gigabytes in the 00s into the multi-terabyte range today, enabling more and more database systems to be run entirely within the main memory [15]. From this trend specialized database systems, so called In-Memory-DBs, evolved. While traditional databases spend lots of CPU time waiting for disk I/O, a main memory database can access its entire data within a few hundred CPU cycles because it is entirely backed by main memory. This development causes the main memory index structure used within the database to be a major performance factor, as shown in [16].

While simple queries such as lookups in a key-value store only perform a single lookup in the index, some more complex queries, e.g. index joins across multiple tables access the index structure for each tuple to be joined and hence up to several million times for big data during query processing. In most real-world OLAP (On-Line Analytical Processing) and OLTP (On-Line Transactional Processing) scenarios, such complex queries account for the majority of total queries, since they promise much better performance than fetching and aggregating data within the client software. Consequently, optimizing the used index structures for higher throughput, lower latency and smaller size can yield a significant system performance increase [16].

Basically, three categories of index structures are most widely used today [10]: hash tables, search trees and prefix trees. While a hash table guarantees constant access times for exact queries, utilizing this structure for range queries imposes several limitations on the organization of the data indexed. To use a hash table for range queries the user needs to guarantee that both the upper and the lower limit are contained within the hash table. Additionally the indexed data needs to be stored in a sorted order to enable traversal operations. The performance of a hash table is mostly determined by the underlying hash function, which can be a problem if a sophisticated hash function is used.

If range queries with unordered data are required, linked search trees or prefix trees can be utilized. Such tree structures can be traversed in linear time for given boundaries. The difference between search trees and prefix trees lies in the representation of the search keys. While a search tree stores n-1 complete keys within each node, a prefix tree only stores a subset of the key in each node. In this work we analyze the Adaptive Radix Tree (ART), which is a memory-optimized variant of a prefix-tree. It does so by adapting the node sizes to the number of children and compacting prefixes into inner nodes. By adapting node sizes, ART achieves the same indexing performance compared to hash tables and traditional search trees, with a significant reduction in memory consumption [17]. ART is an efficient index structure for point, range and prefix lookups

This work is licensed under a Creative Commons Attribution-Share Alike International 4.0 License.

ICPP '21, August 9–12, 2021, Lemont, IL, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9068-2/21/08.
https://doi.org/10.1145/3472456.3472511

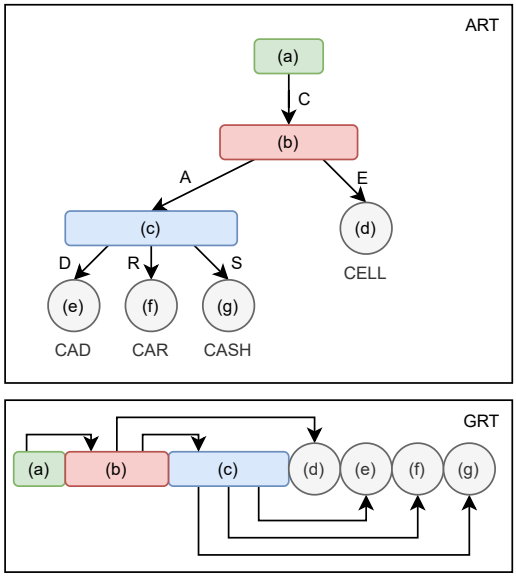

Figure 1: Example ART (upper half) and corresponding mapped GRT (lower half), visualizing the memory layout and the different node sizes, colors represent the different node sizes

as well as updates. Ref. [2] conducted a performance comparison between ART, Judy Trees (another interpretation of adaptive radix trees) and cuckoo hashing. While they found that a "carefully chosen hash table" outperforms radix trees for single queries, ART is significantly faster for range queries.

Another development in modern data centers within the last few years is the increasing usage of specialized hardware accelerators, GPUs and FPGAs. While initially driven by GPU and ASIC solutions for neural network training and inferencing, the increasing programmability of modern GPUs opens up the possibility to accelerate new types of applications such as video/game streaming as well as databases. Those acceleration approaches manifest themselves in off-the-shelf, optimized libraries provided by FPGA and GPU vendors e.g. the Xilinx Vitis Database library [26] or NVIDIA CUDA libraries for general purpose computing [21]. Considering the aforementioned requirements to index structures in modern database systems as well as the fact that lookups can be arbitrarily parallelized for most index structures, it is desirable to accelerate index lookups using GPUs.

1.1 Our Contribution

In this work, we propose memory and algorithmic optimizations which improve the performance of ART when implemented on a GPU. We evaluate the performance in terms of throughput for lookup operations within the ART with several index sizes. We analyze the impact of different GPU memory architectures such as GDDR and HBM. We show that the same optimizations can also be applied towards a CPU-based lookup engine and achieve significant speedups compared to the original ART implementation. Besides lookup operations, we also propose a parallel update engine which is able to atomically replace values within the CuART.

1.2 Structure

The rest of our paper is structured as follows. In the next chapter (2), we present previous and related work to our research. We will not only focus on database applications, but also include algorithmically similar work from other research areas. Chapter 2.1 introduces the concepts of the ART as well as the algorithm we use to find entries within the ART. Chapter 3 presents the optimizations we applied to the existing GPU based ART implementation, highlights challenges in the design and provides deep insights into the implementation. Chapter 4 presents the results of our evaluation, comparing and analyzing several experiments. Finally a short conclusion and an outlook into further research is given.

2 RELATED WORK

In [25] a comprehensive performance case study of several modern index structures with focus on paralellism is published. Since CuART heavily exploits a high degree of paralellism, the index structures compared in [25] can be seen as good comparison candidates. In their study they find that ART is one of the fastest tree-based index structures for querying and updating, making it a good choice for a GPU implementation. Another interesting work presented by Wu et. al. in [24] adapts the well-known database cracking algorithm towards ART, optimizing the memory consumption by constructing the ART during query processing, therefore only indexing keys which are actually queried. Database cracking was first published in [13] as a possible strategy to create and maintain workload-adaptive index structures, yielding a significant performance improvement over traditional indices. The strategy of database cracking was evolved even further by combining the adaptive merging strategy [7] with database cracking to a hybrid indexing solution, effectively reaching a workload-adaptive, low-overhead, low-memory index structure in the context of in-memory databases [14]. The aforementioned publications are complimentary to our work, since they focus on improving index construction, maintenance and memory consumption, whereas our work focuses on performance improvements and memory savings when implemented on a GPU, which opens up space for future work (see section 5.1).

2.1 Radix Trees

Prefix or radix trees have been proposed as a space-efficient data structure quite a few times in literature. Prominent examples include the Patricia Trie introduced by Morrison et. al. in [20], as well as the HAT-Trie in [3] and the Adaptive Radix Tree (ART) introduced by Leis et. al. in 2013 [17] as an extension to traditional Radix Trees. The upper half of figure 1 shows an ART populated with 4 entries, visualizing the different node sizes and pointers between the nodes. Since then, several additions and improvements to this index structure were proposed, e.g. by merging several nodes for sparse tree areas (START, [5]). Another work that builds on top of ART was the GRT proposed in [1], a GPU based implementation of ART which we used as a starting point for our work. The lower half of figure 1 shows the memory layout of the GRT, obviously only

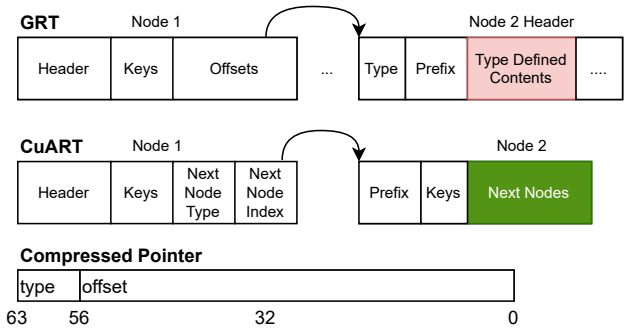

Figure 2: Node Layout of GRT vs. CuART, visualizing the type dependent contents of a node in GRT

one buffer is used and all items are laid out in order. While radix trees in general are suitable for exact, range and prefix lookups, there also have been approaches for running approximate lookups on the GPU by Groth et. al. in [8], making ART also suitable for approximate queries.

2.2 Hardware Accelerated Indexing

Ref. [22] presents a new approach for a hybrid in-memory index tree. Therefore they developed an CPU-GPU B+-tree and HB+-tree. Their implementation is based on heterogeneous hardware platforms and utilizes a hybrid memory structure with simultaneous access from both the CPU and the GPU. They achieve a speedup of 2.4x compared to a plain CPU variant. Ref. [12] also shows new options to improve the performance of B+-trees in a CPU-GPU architecture. They use simultaneous sorting and searching on CPU and GPU to increase the total throughput by 1.8x. It presents an automatic approach to map OpenCL kernels onto heterogeneous multi-cores systems, achieving a speedup up to 1.8x. [6] optimizes the K Nearest Neighbor (KNN) joins by using a hybrid CPU/GPU approach. Apart from specific index structures, ref. [19] evaluates OpenCL optimization techniques which can be used across all hardware vendors and platforms, some of which potentially could be applied to this work too. They show that FPGA platforms have a better performance portability and energy efficiency than GPUs if optimized correctly. ref. [23] evaluates OpenCL optimizations in embedded systems. They show how a multicore algorithm could be extented and optimized. They maximize the throughput by larger batches, create a better GPU resource utilization and limit the CPU and GPU imbalance, a technique we also used in this work.

3 CUART

The following chapter starts by analyzing the problems of GRT and ART, focusing on why the problems discovered in the GRT decrease the achievable performance. Afterwards, we propose several optimizations

3.1 GRT Analysis

An analysis of the runtime and performance metrics of the initial GPU-based ART implementation reveals that the memory access patterns of ART are suboptimal for GPUs, because the node type is

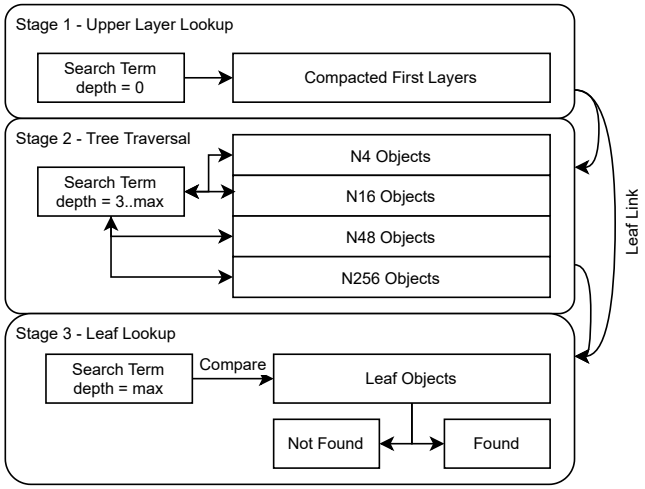

Figure 3: Lookup Algorithm within the optimized buffer structure

encoded within the node structure itself (see figure 2). This leads to at least two memory accesses/transactions towards the local or global memory, because the correct size to read depends on the node type, which is encoded within the header. While modern GPUs have sophisticated memory controllers that will prefetch eagerly, the larger node types of ART are too large (650B for N48 and 2KB for N256) to be prefetched upfront. Therefore it is necessary to read the header first and then read the remaining data depending on the actual node type. Another substantial problem of trees on GPUs in general is that the computational effort required to traverse such tree is typically small, in the case of ART it is at around 20 clock cycles per node, whereas a global memory access typically requires 50 clock cycles at best (NVidia Ampere GPU). This leads to the conclusion that increasing memory bandwidth, decreasing memory latency and improved access patterns will increase the throughput of tree-based structures when implemented on a GPU.

Another problem arises when running mixed read/write workloads such as typical OLTP benchmarks. Due to the control flow intensive algorithm required to update and restructure such trees, a CPU is more suitable to actually perform the update operations. In case of tree modifications like updates and inserts, the CPU most likely needs to perform other actions on the actual data managed by the index structure anyway. However, for a tree-based index structure to be usable on a GPU, the pointer based objects need to be flattened into one or more buffers which can be used on the GPU. In case of frequent updates, preparing the buffers for the GPU needs to happen for almost every update depending on the consistency guarantees of the DBMS.

3.2 Improvements

The following sections provide a detailed overview of our improvements to the ART index structure to improve the performance when implemented on GPUs.

3.2.1 Memory Access Patterns. In the original implementation of the GRT, the authors implemented a mapping step from the pointer-based ART in main memory towards into a single, tightly packed buffer of nodes utilizing an in-order traversal. This buffer is then passed to the GPU as an untyped buffer, and is accessed with offsets in this buffer (see figure 1). While this approach works and is beneficial for hierarchical lookups, it imposes the problem described in section 3.1, that both the alignment and the size to read from global memory is unknown beforehand. In our implementation, we map the index structure into several buffers instead of just one, as shown in figure 3. Namely we utilize one buffer per node type. This change has two significant implications. First of all it allows the implementation to determine the transaction read size before initiating the actual memory request, as shown in figure 2. This improvement combined with a guaranteed alignment of at least 16 bytes allows for efficient memory transactions and therefore a reduced memory controller load. Especially in the case of larger nodes, trading memory bandwidth for access latency should improve the performance significantly. The second implication is that we need to replace the byte offsets into the single buffer used by GRT. We replace the single 64bit offset by a packed 64bit integer containing the next node type in the most significant bits and the node index within the corresponding buffer in the least significant bits. The structure of this 64bit value is shown in figure 2. In our implementation we use the numbers 1 to 4 to represent the different node types (1=N4, 2=N16, 3=N48, 4=N256) and 5 to 7 for the different leaf types (5=leaf8, 6=leaf16, 7=leaf32). For very large index structures, the node type could be further reduced to the actually used bits, leaving even more addressable space for the node arrays. Furthermore, this compression also allows us to save one byte of memory from the node header, because the node type is no longer required there. We reused that byte for an increased maximum prefix length.

In our initial implementation, we replaced the dynamically sized leaf buffer by a fixed size leaf, which can store up to 32 byte keys, again trading memory bandwidth for access latency for small (4 to 16 byte) keys. During the evaluation, we switched from a single sized leaves to several leaf objects of different sizes (8, 16, 32 bytes) to better adapt to dynamic key sizes. A side effect of changing this buffer structure is that we can make a compile-time guarantee, that all nodes are aligned to 16 or even 32 byte addresses, which allows the compiler to generate efficient memory loads. A downside of having fixed-size leafs is that the user has to define a compile-time constant maximum key size up to which lookups can be done. While this restriction is feasible for accelerating traditional columns where indexes are built of 8 (numeric IDs) or 16 (UUIDs) byte keys, we show approaches to overcome this limit in section 3.2.3. A side effect of splitting leafs into multiple leaf buffers of different sizes is that transferring range queries from the accelerator to the host is trivial because it is only required to transmit both the start and the end index within the leaf arrays, because the keys are already strictly ordered within the leaf buffers assuming a lexicographical order, thus speeding up range queries significantly.

3.2.2 Compacting upper Layers. In order to improve the total access latency, we merged the upper layers into a multi-layer ART node, as proposed in ref. [5] for all nodes. This reduces the number of memory accesses required for one layer to complete and also increases cache hit rates at the cost of a slightly higher index memory consumption for large indices. In our case, we merged the first three layers into a lookup table. We realized this optimization by utilizing a dense array of compacted pointers (node links), resulting in 128MB of memory consumption on the device. For large key spaces, this solution accounts for around 30 to 40 megabytes of device memory used atop of the actual index structure. While the lookup table takes up 128MB, the overhead is reduced by the space the nodes in the first three layers take up. Lookups within the compacted root node are realized by using the first three bytes of the key as an index into a dense array, as presented in ref. [5].

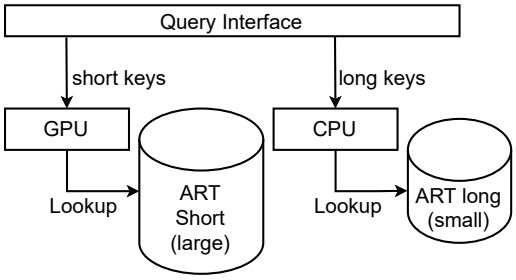

Figure 4: CPU/GPU Hybrid Implementation of long key handling, processing all long keys on the CPU

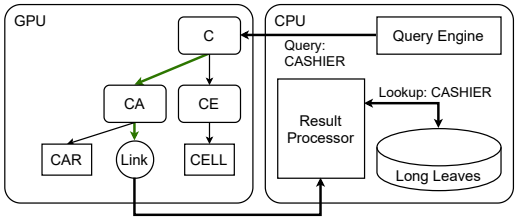

Figure 5: CPU/GPU Hybrid Implementation of long key handling, storing the long keys separately and inserting links to long leaves

3.2.3 Handling long keys. Because our approach can only handle keys up to a compile-time defined length, we introduce three possibilities for handling longer keys than this maximum. The first possibility is to skip the GPU-based processing of long keys entirely and process those keys on the CPU (see figure 4, visualizing the query split). The second possibility is to move the longer keys into a separate leaf buffer on the host memory, inserting links to those leaves into the GPU node buffers (see figure 5). When the GPU encounters such a link, it sets a specific signal in the return value indicating that the CPU needs to compare the complete key along with the address of the actual leaf in main memory. This allows us to handle exceptionally long keys without wasting memory resources in the expensive device memory and still perform the actual lookup on the GPU. The third possibility is the one GRT chose, namely introducing another leaf type which is dynamically sized either at byte, word or 16-byte level and perform a dynamic sized memory

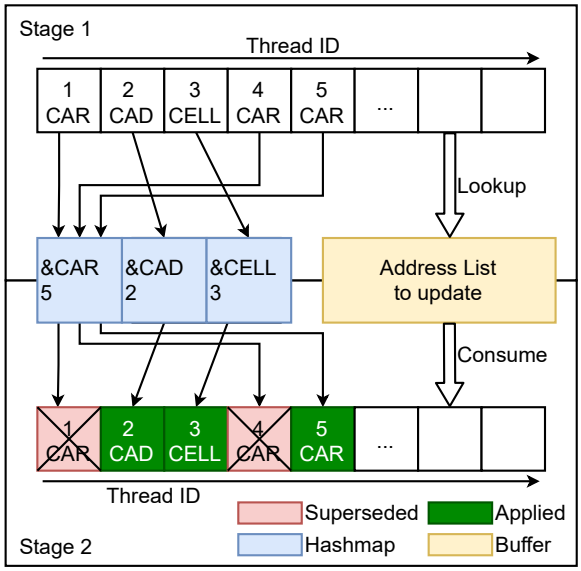

Figure 6: Two-staged update process, visualizing conflict resolution with a hash table and atomic operations, storing the highest index of the update operation, updates for other indices are skipped

comparison. While this approach is good for frequent occurences of keys that are slightly longer than the implementation defined maximum, it can severely hurt the overall lookup performance in case of exceptionally long keys, because the single thread spends a lot of time comparing the actual leaf. The first option (long keys on the CPU, short keys on the GPU) is the most promising as it is able to utilize otherwise unused CPU resources to process even more queries.

3.3 Device Side Deletions

In order to speed up mixed read/write transactions when migrating read queries towards a GPU we utilize CUDA unified memory, which is an implementation of shared virtual memory. This architecture allows us to efficiently process updates and insertions on the CPU and only updating affected memory regions within the buffer structure. To process a deletion directly on the device, the tree is traversed, keeping the last visited offset in local memory. Once a leaf is reached, its contents are cleared and the reference to the leaf is removed from the last visited node. The leaf index is pushed into a list of free leaves which can be used for future inserts. By not modifying the structure of the tree (i.e. not collapsing nodes immediately), the deletion performance can be increased significantly. Chapter 5.1 outlines the work required to perform all tree operations in parallel on a GPU.

3.4 Device Side Updates

Update operations replace the value stored for certain keys. In order to perform an update, again a tree traversal is needed to find the leaf location to update. Since all operations are applied in batches, especially in the case of updates the correct ordering of operations must be considered. We utilize a one-dimensional grid of threads in CUDA, which means that the update operation priority increases along with the thread ID. In order to perform update operations in parallel, duplicate writes to the same key are eliminated by utilizing an atomic hash table (compare Ref. [4]) storing the maximum element index that performs an update to a certain leaf. This process is visualized in figure 6. Stage 1 performs a lookup within the tree, returning the memory location instead of the actual value for the leaf. Afterwards, it updates the hash table if its thread index is larger than the current one or not contained within the hash table. Collisions are handled by simple linear probing as described in ref. [4]. After inserting all indices into the hash table, a thread synchronization is performed. In the end, all threads read back the maximum index for their computed memory location and perform the actual update operation in the global memory if their thread index is equal to the hash table value. As updates and non-structural modifying deletes are quite similar in their functionality, we use the same implementation for both, signaling a deletion through a setting a nil pointer.

4 EVALUATION

The following sections outline our experimental results of evaluating CuART, GRT and the impact of our optimizations. First we present our benchmark setup. We evaluate the lookup and update performance of CuART against GRT and ART in different scenarios varying parameters.

4.1 Setup

In order to evaluate the performance of our algorithm, we build a framework that is capable of generating reproducible trees with data of different characteristics and afterwards generate update, delete, range and exact lookup queries. In contrast to ref. [1] the throughput is measured as an end-to-end manner, including CPU overhead for processing the lookups afterwards, PCIe transfer times and pipelining. Queries are coalesced into batches in order to reduce the compute overhead, typically with a power-of-two size to ease up scheduling and optimal load on the GPUs. We tested against synthetic random test data as well as real world test data from the publicly available BTC dataset [11]. Our host code utilizes a variable amount of command streams for both CuART and GRT, decoupling the GPU dispatch from a specific number of host threads. In general, the benchmarks are performed in three stages:

- Populating the ART index
- Mapping the CPU ART into the buffer structure described in chapter 3.2.1
- Running the actual queries against the GPU, measuring the throughput

In order to prove that our optimizations are generally applicable, we conduct several experiments with a wide range of parameters, varying in tree size, number of host threads, size of the batches sent to the GPU and different key lengths. To prove that our improvements are not only caused by using a different API, we compare CuART against both a CUDA and an OpenCL variant of GRT. To account for different memory options, we present benchmarks of all candidates on the following three machines to rule out architectural differences and mimic a variety of scenarios:

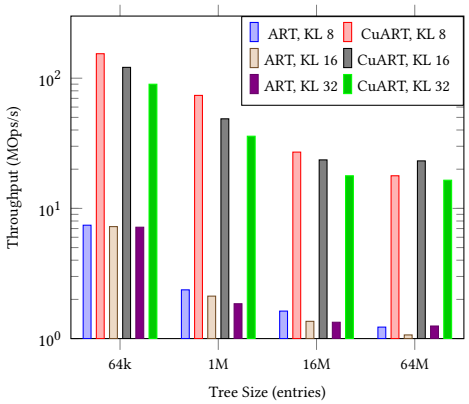

Figure 7: Lookup throughput on classical ART vs CuART memory layout on CPUs (12 threads, 32ki items per batch, KL = Key Length, workstation)

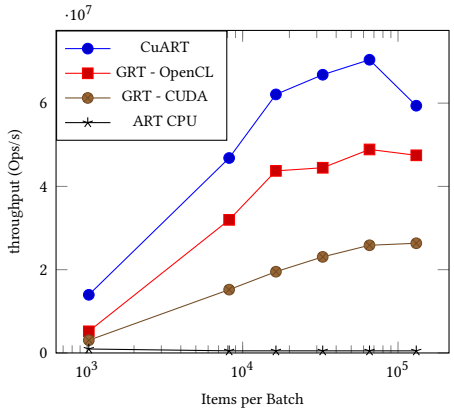

Figure 8: Lookup Throughput with increasing batch size (26Mi entries, 8 threads, 32 byte keys, server)

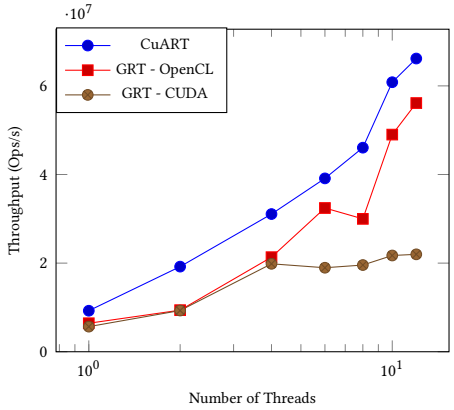

Figure 9: Lookup Throughput with increasing number of threads (26Mi entries, 32 byte keys, 32ki items per batch, server)

- Server - 2x AMD Epyc 7752, 2x NVidia A100 (40GB HBM2), 2TB DDR4-2933
- Workstation - AMD Ryzen 5800X, NVidia RTX3090 (24GB GDDR6X), 128GB DDR4-3200
- Notebook - Intel Core i7 8750H, NVidia GTX1070 (8GB GDDR5), 64GB DDR4-2666

4.2 CPU

In this section we conduct several experiments to prove the benefits of our improvements for exact lookups. We start by verifying our proposed optimizations on the CPU. We varied the tree size (number of items within the tree) as well as the key length (size per item) over typical used ranges. Figure 7 shows the impact of our optimizations on the CPU. For small trees, the CPU has a very good cache efficiency, CuART outperforms the original ART by 2.5 times for small trees, increasing the performance gain to up to 20 times for lookups within larger trees. This experiment reveals that our opimizations are generally applicable to ART and not only tailored towards a specific GPU architecture. While this performance gain is certainly very useful for lookup-intensive workloads (OLAP), it comes at an additional maintenance burden when updating the index structure, because all operations have to be done within the flat buffer. CuART performs and scales significantly better than the original ART because it employs continous pieces of memory. The traditional ART implementation is spread across the main memory. Especially in the case of smaller index structures, our whole tree fits entirely into the CPU cache and utilizes all available cache lines due to its continous memory layout, minimizing cache misses.

4.3 GPU Host Code

To achieve the highest throughput on a GPU, it is crucial to keep as many compute units as possible busy. We performed a design space exploration on two parameters, namely the number of host threads (shown in figure 9) sending work to the GPU and the size of the work batches sent to the GPU (shown in figure 8). While very small batches increase the overhead on the host side, both GRT and CuART achieve a good performance at any batch size between 8192 and 131072 items. For the remaining experiments, we chose a batch size of **32768 items**. Figure 9 reveals that in general, more host threads are preferable for both CuART and GRT. We chose to utilize **8 threads** for the remaining experiments as a tradeoff between dedicating host CPU resources to keep the GPU busy and running query operations in a real world scenario. Compared to GRT, CuART is much more thread agnostic. One reason for this behavior is the inherent asynchronousity of the CUDA API utilizing several streams that can be efficiently mapped onto the GPU. All of those measurements were taken at a medium tree size (26M entries).

4.4 Exact Lookups

In the next experiment we show the impact of varying the tree size, with a fixed key length of 32 bytes, e.g. mimicking the scenario of ART used as an index on a primary key of a growing table. Figure 10 reveals the scalability of CuART compared to GRT. CuART outperforms GRT for all tested index sizes, ranging from 64k items

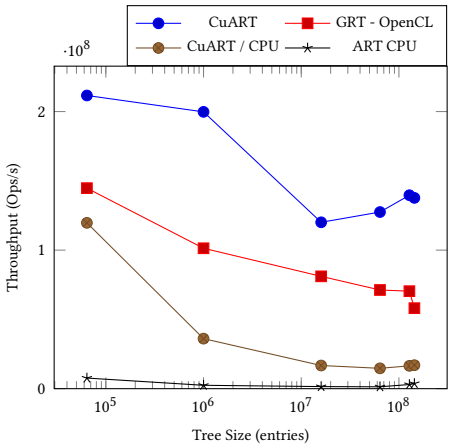

Figure 10: Lookup Throughput with increasing tree size (64k-144M entries, 8 threads, 32byte keys, 16ki items per batch, workstation)

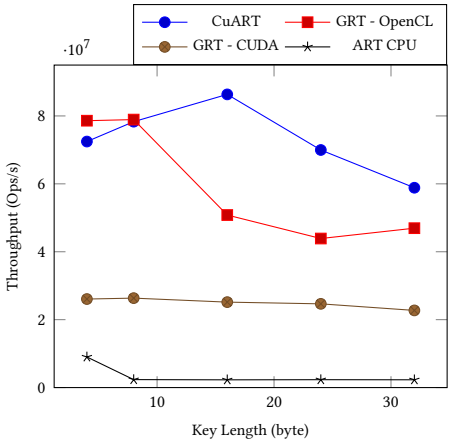

Figure 11: Lookup Throughput with increasing key length (26Mi entries, 8 threads, 32ki items per batch, server)

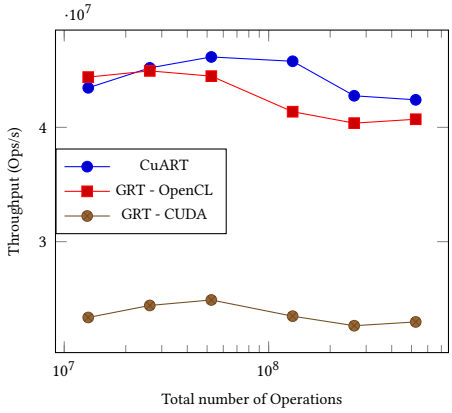

Figure 12: Throughput against the BTC dataset (15.4M keys, 32 byte key length, 32ki items per batch, 8 threads, server)

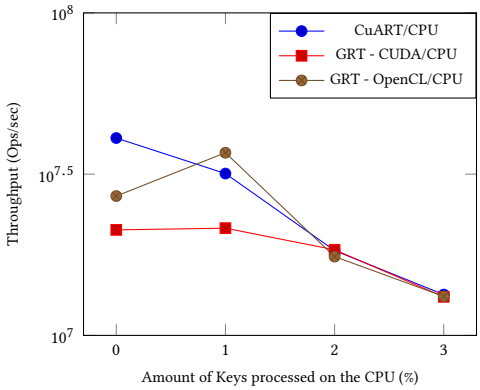

Figure 13: Hybrid CPU/GPU query approach (8 threads GPU / 56 threads CPU, 32+byte keys, 32ki items per batch, 26Mi entries, server)

up to 144M items. An interesting detail is that the throughput of CuART actually increases slightly with increasing tree size. This is due to the fact that larger trees are more densely populated in this test case which means that large nodes occur more frequently, where the prefetch misses affect the GRT implementation much more than CuART (as discussed in section 3.1).

When varying the size of the keys within the tree, the results change quite a bit. In figure 11 we run the benchmark again, this time with a fixed tree size and varying key length from 4 to 32 bytes. The results reveal that CuART outperforms GRT on longer keys while short keys are heavily beneficial for GRT. This is caused by the comparison loops, where GRT adapts to shorter keys byte-oriented compared to CuART which does it word-oriented.

CuART not only outperforms GRT on synthetic test data, but also on real world data, taken from the BTC challenge. Figure 12 visualizes that CuART outperforms GRT by around 20% on the BTC dataset. We extract all keys of 32byte length from the BTC dataset and inserted them into an ART and performed random lookup operations against this tree. The absolute performance and the performance gain in this test are lower than in the synthetic tests due to the fact that long duplicate segments are quite common, which adds computational overhead during prefix compression and increases the overall tree depth.

4.4.1 Long Keys. The need for handling keys longer than the CuART maximum can arise in some specific workloads such as semantic web indexing. In section 3.2.3 we outlined several possibilities to deal with those long keys. In the following experiments we evaluate the first option of processing long keys directly on the CPU and processing short keys in parallel on the GPU. For this scenario we generate a tree with a controlled percentage of long keys and run lookup queries with a controlled amount of long keys afterwards. Figure 13 visualizes the impact of handling those keys on the CPU, in this case all keys longer than 32 bytes were processed on the CPU side and only shorter keys are queried on the GPU. It can be seen that the overall performance drops quite fast with an increasing amount processed on the CPU, yielding around 50% performance impact for only 3% of the keys processed

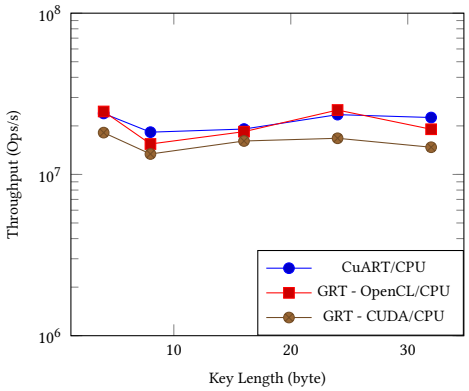

Figure 14: Hybrid CPU/GPU query approach (8 threads GPU / 56 threads CPU, 5% CPU keys, 32ki items per batch, 26Mi entries, server)

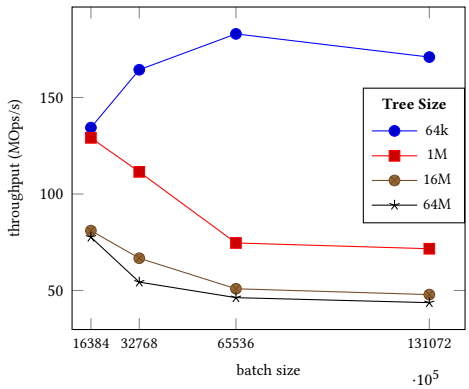

Figure 15: CuART Update throughput with increasing batch size for different tree sizes (16ki items per batch, 8 threads, 16 byte keys, workstation)

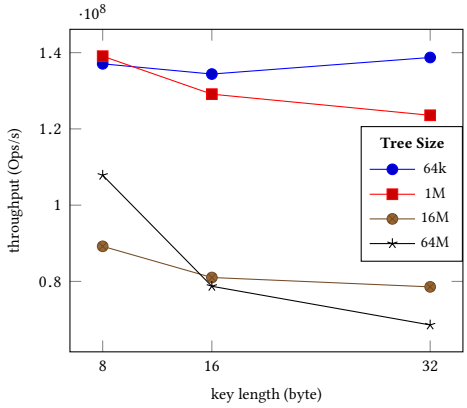

Figure 16: CuART Update throughput with increasing key length for different tree sizes (16ki items per batch, 8 threads, workstation)

on the CPU. This lead us to the next question whether the same optimizations we applied to the GPU algorithm are also applicable to the CPU based implementation. To verify that the bottleneck is indeed the key handling itself, we conducted another experiment where we process a fixed amount of **short** keys on the CPU, in this case 5%. Figure 14 visualizes the result. It can be seen that all GPU implementations are in fact limited by the CPU processing. This leads to the conclusion that the other outlined long key handling options should be evaluated as well in the future.

4.5 Update/Deletions

In this section, we conduct the same experiments outlined in the previous chapter also for update operations. Figure 15 visualizes the update throughput of CuART for different tree sizes and batch sizes. Two observations can be made from this figure, first of all the update throughput drops with increasing batch sizes, which is caused by the fixed size of the temporary hash table for collision resolution, which leads to more conflicts when having large trees and large batches. In our tests, we used a hash table size of 1Mi entries, which means the drop is not visible for a small tree (compare blue 64k tree curve), because the hash table is only partially filled. For larger trees and large batches, hash table collisions become quite frequent and then the linear probing algorithm causes the update throughput to drop. The collisions can be avoided by utilizing a larger hash table, trading memory consumption for speed, although quite a bit of memory is unused. Again, one could use a more sophisticated probing scheme, but simply increasing the hash table size promises better results.

Figure 16 shows the update throughput against different key lengths and tree sizes. While it can be seen that for small trees, caching effects are overwhelmingly large. As the tree size is growing, the influence of caching effects drops. As expected, the update performance drops for larger keys due to more computational overhead to compare the keys. Figure 17 highlights the performance gain for updating entries within CuART against GRT and a CPU based approach. CuART achieves an update throughput of around 20% below its lookup throughput (~120MOps/s) while keeping the atomic update guarantees. In contrast, globally visible, atomic updates cause a significant performance drop in both the GRT and the original ART, where CuART reaches a 10x improvement over the GRT (~13MOps/s) for atomic updates and up to 50x speedup compared to the CPU (~2.5MOps/s). However, the throughput of GRT remains almost constant in GRT, which indicates memory conflicts.

4.6 GPU Memory Impact

Up till now all experiments were run on the NVidia A100 data center GPU, equipped with HBM2. A benchmark across different GPUs show that the NVidia RTX3090 equipped with GDDR6X generally outperforms the A100 due to its higher memory clock (2500MHz on the RTX3090 vs 1215MHz on the A100) and therefore faster random memory access times. Figure 18 visualizes this phenomenon, revealing that CuART outperforms GRT on all tested GPUs. It becomes obvious that CuART profits from a high memory clock more than GRT, which means that the CuART implementation is more efficient and therefore reaches the compute bound later than

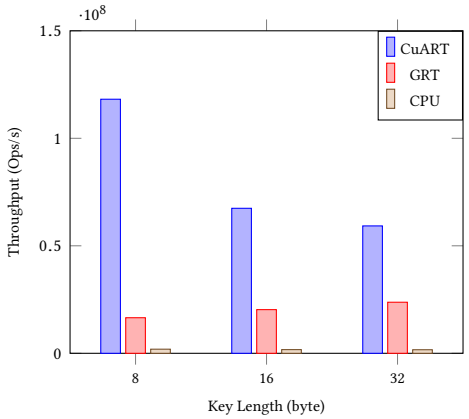

Figure 17: Update throughput of CuART, GRT and the CPU (16Mi entries, 8 threads, 32ki items per batch, workstation)

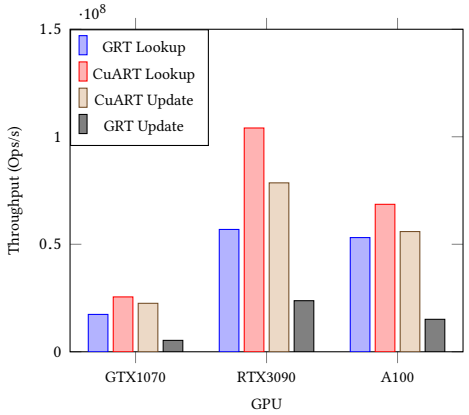

Figure 18: Lookup/Update throughput on different GPUs (16Mi entries, 8 threads, 32ki items per batch, 32 byte keys)

GRT. The figure also includes the throughput for update operations, where CuART outperforms GRT on all tested GPUs. On the NVidia A100, there are 40 independent memory channels available (8 per HBM2 stack), on the RTX3090 there are only 24 channels available (2 channels per GDDR6X chip), where the channel width is 128 bit on the A100 (yielding a total memory width of 5120 bits), whereas the RTX3090 has only a 384bit memory width (24 channels * 16 bit width). When taking this into account, the GDDR6X memory interface is more suitable due to its higher command clock frequency and therefore more commands. Another problem with the HBM within the A100 is the fact that its memory interface is 128bits per channel which means that a typical transaction (i.e. reading a node header) is finished within one single clock cycle, which causes increased command overhead.

5 CONCLUSION

This work presents several optimizations that can be applied in order to improve the lookup performance for the ART index structure on CPUs as well as GPUs. Furthermore we also adapt our algorithm to cover atomic updates on the GPU, opening up more use cases for ART, e.g. in the field of tracking and aggregating metrics with string-based keys, as done e.g. by monitoring software. We show that our optimizations outperform existing ART implementations by up to ten times compared to the traditional CPU ART, up to two times compared to the GPU ART implementation and up to 50 times for atomic update operations compared to the CPU ART. We also show that a GPU utilizing higher-clocked memory is more suitable for this kind of workload, whereas the RTX3090 outperforms the A100 at one quarter of the price tag. Possible real-world uses for our work include ART as an index structure for KV-stores with update/lookup intense workloads or as a traditional database index well-suited for point, range and prefix queries.

5.1 Future Work

Possible future improvements include a full device-based management of the whole ART, implementing structural modifying insertions and deletions. To achieve this, a more sophisticated buffer management needs to be implemented, as the need to allocate new nodes or free old nodes arises. Furthermore, we plan to add a specialized handling for index structures larger than the device memory, by migrating rarely used parts of the key space into host memory and query them in a hybrid manner with both GPU and CPU doing the work. As it became obvious that the implemented long key option is decreasing the achieved throughput, we intent to improve the long key handling in the future. We plan to integrate the multi-layer nodes presented in [5] into our work, further improving the performance. Finally, we plan to evaluate whether the proposed optimizations are feasible to be implemented within an FPGA-based lookup engine.

ACKNOWLEDGMENTS

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 422742661

Funded by
Deutsche Forschungsgemeinschaft
German Research Foundation

REFERENCES

[1] Maksudul Alam, Srikanth B. Yoginath, and Kalyan S. Perumalla. 2016. Performance of Point and Range Queries for In-memory Databases Using Radix Trees on GPUs. In *2016 IEEE 18th International Conference on High Performance Computing and Communications*. IEEE Computer Society, New York, NY, USA, 1493–1500. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0212

[2] Victor Alvarez, Stefan Richter, Xiao Chen, and Jens Dittrich. 2015. A comparison of adaptive radix trees and hash tables. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE Computer Society, New York, NY, USA, 1227–1238. https://doi.org/10.1109/ICDE.2015.7113370

[3] Nikolas Askitis and Ranjan Sinha. 2007. HAT-Trie: A Cache-Conscious Trie-Based Data Structure for Strings. In *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62* (Ballarat, Victoria, Australia) *(ACSC '07)*. Australian Computer Society, Inc., AUS, 97–105.

[4] David Farrell. 2020. *A Simple GPU Hash Table*. https://nosferalatu.com/SimpleGPUHashTable.html

[5] Philipp Fent, Michael Jungmair, Andreas Kipf, and Thomas Neumann. 2020. START — Self-Tuning Adaptive Radix Tree. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. IEEE Computer Society, New York, NY, USA, 147–153. https://doi.org/10.1109/ICDEW49219.2020.00015

[6] Michael Gowanlock. 2019. KNN-Joins Using a Hybrid Approach: Exploiting CPU/GPU Workload Characteristics. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs* (Providence, RI, USA) *(GPGPU '19)*. Association for Computing Machinery, New York, NY, USA, 33–42. https://doi.org/10.1145/3300053.3319417

[7] Goetz Graefe, Stratos Idreos, Harumi Kuno, and Stefan Manegold. 2011. Benchmarking Adaptive Indexing. In *Performance Evaluation, Measurement and Characterization of Complex Systems*, Raghunath Nambiar and Meikel Poess (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–184.

[8] Tobias Groth, Sven Groppe, Martin Koppehel, and Thilo Pionteck. 2020. Parallelizing approximate search on adaptive radix trees. In *Proceedings of the 28th Italian Symposium on Advanced Database Systems, Villasimius, Sud Sardegna, Italy (virtual due to Covid-19 pandemic)*. Villasimius, Sardinia, Italy, 56–67.

[9] The PostgreSQL Global Development Group. 2020. *Future of storage*. https://wiki.postgresql.org/wiki/Future_of_storage

[10] Manoj Gupta and Dharmendra Badal. 2013. A Study on Indexes and Index Structures. 2 (02 2013), 212–222.

[11] José-Miguel Herrera, Aidan Hogan, and Tobias Käfer. 2019. BTC-2019: The 2019 Billion Triple Challenge Dataset. In *The Semantic Web – ISWC 2019*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtěch Svátek, Isabel Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon (Eds.). Springer International Publishing, Cham, 163–180.

[12] Han Huang and Hua Luan. 2020. Optimizing B+-Tree Searches on Coupled CPU-GPU Architectures. In *Algorithms and Architectures for Parallel Processing*, Meikang Qiu (Ed.). Springer International Publishing, Cham, 401–415.

[13] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Proceedings of the 3rd International Conference on Innovative Data Systems Research (CIDR)*. CIDR Conference, Asilomar, California, 68–78.

[14] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *Proc. VLDB Endow.* 4, 9 (June 2011), 586–597. https://doi.org/10.14778/2002938.2002944

[15] Abdullah Talha Kabakus and Resul Kara. 2017. A performance evaluation of in-memory databases. *Journal of King Saud University - Computer and Information Sciences* 29, 4 (2017), 520–525. https://doi.org/10.1016/j.jksuci.2016.06.007

[16] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for in-Memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 468–479. https://doi.org/10.1145/2540708.2540748

[17] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, New York, NY, USA, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[18] Yishan Li and Sathiamoorthy Manoharan. 2013. A performance comparison of SQL and NoSQL databases, In 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM). *IEEE Pacific RIM Conference on Communications, Computers, and Signal Processing - Proceedings*, 15–19. https://doi.org/10.1109/PACRIM.2013.6625441

[19] Umar Ibrahim Minhas, Roger Woods, and Georgios Karakonstantis. 2018. Exploring Functional Acceleration of OpenCL on FPGAs and GPUs Through Platform-Independent Optimizations. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, Nikolaos Voros, Michael Huebner, Georgios Keramidas, Diana Goehringer, Christos Antonopoulos, and Pedro C. Diniz (Eds.). Springer International Publishing, Cham, 551–563.

[20] Donald R. Morrison. 1968. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (Oct. 1968), 514–534. https://doi.org/10.1145/321479.321481

[21] Inc. NVidia. 2016. *Vitis Database Library*. https://docs.nvidia.com/cuda/index.html

[22] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A Hybrid B+-Tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1523–1538. https://doi.org/10.1145/2882903.2882918

[23] Ben Taylor, Vicent Sanz Marco, and Zheng Wang. 2017. Adaptive Optimization for OpenCL Programs on Embedded Heterogeneous Systems. *SIGPLAN Not.* 52, 5 (June 2017), 11–20. https://doi.org/10.1145/3140582.3081040

[24] Gang Wu, Yidong Song, Guodong Zhao, Wei Sun, Donghong Han, Baiyou Qiao, Guoren Wang, and Ye Yuan. 2019. Cracking In-Memory Database Index A Case Study for Adaptive Radix Tree Index. *CoRR* abs/1911.11387 (2019). arXiv:1911.11387 http://arxiv.org/abs/1911.11387

[25] Zhongle Xie, Qingchao Cai, Gang Chen, Rui Mao, and Meihui Zhang. 2018. A Comprehensive Performance Evaluation of Modern In-Memory Indices. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, New York, NY, USA, 641–652. https://doi.org/10.1109/ICDE.2018.00064

[26] Inc. Xilinx. 2019. *Vitis Database Library*. https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-database.html