Knowledge and Information Systems (2023) 65:4359–4377
https://doi.org/10.1007/s10115-023-01891-w

REGULAR PAPER

Hybrid CPU/GPU/APU accelerated query, insert, update and erase operations in hash tables with string keys

Tobias Groth[1] · Sven Groppe[1] · Thilo Pionteck[2] · Franz Valdiek[2] · Martin Koppehel[2]

Received: 18 January 2023 / Revised: 14 April 2023 / Accepted: 22 April 2023 /
Published online: 26 May 2023

Abstract

Modern computer systems can use different types of hardware acceleration to achieve massive performance improvements. Some accelerators like FPGA and dedicated GPU (dGPU) need optimized data structures for the best performance and often use dedicated memory. In contrast, APUs, which are a combination of a CPU and an integrated GPU (iGPU), support shared memory and allow the iGPU to work together with the CPU on pointer-based data structures. First, we develop an approach for dGPU to accelerate queries in libcuckoo and robin-map and when looking at accelerating insert, updates and erase operations in the original libcuckoo using OneAPI on an APU. We evaluate the dGPU against the CPU variants and our dGPU approach adapted for the CPU and also in a hybrid context by using longer keys on the CPU and shorter keys on the dGPU. In comparison with the original libcuckoo algorithm, our dGPU approach achieves a speed-up of 2.1, and in comparison with the robin-map a speed-up of 1.5. For hybrid workloads, our approach is efficient if long keys are processed on the CPU and short keys are processed on the dGPU. By processing a mixture of 20% long keys on the CPU and 80% short keys on dGPU, our hybrid approach has a 40% higher throughput than the CPU only approach. In addition, we develop a hybrid APU approach for insert, update and erase operations in the original libcuckoo structure focusing on shared memory with iGPU accelerated look-ups of the positions for insert, update and erase operations.

✉ Tobias Groth
t.groth@uni-luebeck.de

Sven Groppe
sven.groppe@uni-luebeck.de

Thilo Pionteck
thilo.pionteck@ovgu.de

Franz Valdiek
franz.valdiek@ovgu.de

Martin Koppehel
martin.koppehel@ovgu.de

1 Institute of Information Systems, University of Lübeck, Ratzeburger Allee 160, 23562 Lübeck, Germany

2 Institute for Information Technology and Communications, Otto-von-Guericke University Magdeburg, Universitätsplatz 2, 39106 Magdeburg, Germany

Keywords Hybrid · Hash table · Strings · GPU · APU · SYCL

1 Introduction

Modern computer hardware has a variety of hardware accelerators for different purposes. For example, computers can have special compute units for video de-/encoding, artificial intelligence or ray tracing. These accelerators allow efficient computing of special tasks which is important if higher performance or low power consumption is the goal. Therefore, modern laptops usually have an APU which combines CPU and GPU for acceleration. Sometimes these laptops also have a dedicated GPU for more performance in tasks like 3D rendering. APUs and dedicated GPUs (dGPUs) differ in performance and memory layout but feature similar compute units. Both hardware can be used in the database context to accelerate hash tables in different ways. Hash tables are widely used as key-value stores. In different applications and scenarios, strings are often used as keys like in semantic web contexts. Therefore, it is beneficial if GPUs could accelerate the querying of string keys in hash tables using high end dedicated GPUs. But there is a problem because these GPUs are optimized for handling fixed size data rather than dynamically sized strings. Thus, handling variable long string keys directly in the dGPU memory organization has the drawback of decreased performance, because the processing of each string takes as long as processing the longest of the currently processed strings. Hence, it is not a surprise that current approaches for GPU hash tables focus on integer keys only and do not support string keys of variable size [1]. To overcome the discussed drawback, we propose a hybrid approach, such that string keys up to a maximum length are stored and handled on the dGPU, and longer string keys are stored in the main memory and processed by the CPU. In addition to dGPUs, the previously mentioned APUs can handle pointer-based structures in shared memory and can be used in a different scenario where CPU and iGPU work together to speed up operations. In this scenario, we look at the insert, update and erase operations in the libcuckoo hash table. The iGPU finds the entries where the operation should take place and this operation can be performed either on the CPU or on the iGPU.

Our dGPU approach takes advantage of both processing architectures because the CPU is best suited to handle variable length data and the GPU is tailored to massive parallelism of fixed size data and it is optimized to achieve the best performance. Our APU approach for insert, update and erase focuses on the important flexibility and adaptability of shared memory based pointer structures which allows operation on the same libcuckoo data structure.

Our main contributions are

- a direct fast look-up of string queries on the GPU,
- hybrid scenarios handling shorter keys up to a fixed maximum length on GPU and longer keys on CPU, and
- an extensive evaluation of these approaches on a high performance computing system running recent many-core CPUs and GPUs for scientific calculations.
- APU-based search with SYCL and APU look-up of the insert position and separate CPU/GPU insertion for libcuckoo.

This contribution is an extension paper of Groth et al. [2]. In this extension, we shift the focus toward heterogeneous computing [3] with an accelerated processing unit (APU). This allows the combination of resources for the search without transferring the index structure. Also, we consider insertions into the original libcuckoo index structure supported by the APU. Thereby we focus on GPU-based look-ups of the insert position in shared memory and

the actual insert is done via the CPU or GPU part of the APU. We also implement update and erase operations with the same position finding.

The remainder is organized as follows. In the Sect. 2, we look at related work in the GPU, APU and hash table acceleration context. Then in Sect. 3, we look at the basics of hash tables and GPU acceleration. Then in Sect. 4, we present our approach for parallel queries in hash tables with string keys. We start with the idea and continue with the details of the data structure and the search. In Sect. 5, we evaluate the data structure on a dedicated GPU. We first explain our benchmark framework and then continue with our benchmark environment before we present our evaluation results. Now we switch our focus to the APU in Sect. 6 and start with the general APU architecture and continue with the shared memory and end with the implementation of insert, update and erase operations. In Sect. 7, we look at the APU results for the query data structure and insert, update and erase operations. We finish with a summary and conclusion in Sect. 8.

2 Related work

Several papers related to hash tables focus on the acceleration of hash table algorithms by using modern hardware and technologies. An overview of GPU accelerated hash tables is shown in Fig. 1, where we add the column about supporting string keys on GPUs and our approach for comparison matters. One important aspect of the hash table performance is efficient hashing. Therefore, [4] describes efficient hashing with SIMD in OpenCL.[1] There are many papers with efficient GPU algorithms like Warpcore [5] which is a fast library of hash tables on GPU. It reaches 1.6 billion inserts and up to 4.3 billion retrievals per second on an Nvidia Quadro GV100. It is also faster than cuDPP [6], SlabHash [7] and NVIDIA RAPIDS cuDF.[2] HashGraph [8] uses sparse graph representations for scalable hash tables. This highly parallel data structure uses information from their value-chain for a high performance collision management. DyCuckoo [9] is a dynamic cuckoo table that has a trade-off between GPU memory size and search performance. It is very efficient and enables fine-grained memory control. There are also hybrid hash tables on CPU and integrated GPU developed with modern program techniques like DPC++ [10]. Reference [11] implements a fully concurrent dynamic hash table that runs on GPUs. Also, they show a warp-cooperative work sharing strategy, which reduces the per-thread assignment and processing overhead.

For APU acceleration, the contribution [12] focuses on accelerating queries in B+-Tress. Reference [13] focuses on accelerating group-by and aggregation with CPU-GPU platforms. Other papers focus on different aspects of APU computing like software transactional memory [14] or hash-joins in DBMS with SYCL [15]. Reference [16] uses OneAPI or SYCL for other tasks like distributed k-nearest neighbors.

To the best of our knowledge, all existing contributions to hash tables accelerated by dGPUs do not support variable length string keys. Hence, our contribution is the first to investigate hash tables with string keys of variable length. Furthermore, we show that a parallel hybrid dGPU/CPU hash table overcomes the drawbacks of dGPUs designed to process data of fixed size and promises high performance. Also to the best of our knowledge, no work uses accelerated hybrid insert, update and erase operations in cuckoo hash tables on the APU using shared memory. Therefore as an additional contribution, we propose APU accelerated look-up based insert, update and erase operation for libcuckoo based on OneAPI and SYCL.

[1] https://www.khronos.org/opencl/, accessed on March 08, 2022.

[2] https://github.com/rapidsai/cudf, accessed on March 13, 2022.

Table 1 Overview of different GPU accelerated hash table algorithms and their features [1] compared with our approach [2]

	GPU Performance Criteria					GPU Hardware Features		
Hashing Technique	String Keys on GPU	Sufficient Parallelism	Memory Coalescing	Control Flow	CPU GPU Data Transfer	Shared Memory	Atomic Operations	Wrap-wide Voting
Open-addressing:								
CoherentHash	✗	✓	✓	✗	✗	✗	✓	✗
CuckooHash1	✗	✓	✗	✗	✗	✓	✓	✗
CuckooHash2	✗	✓	✗	✗	✗	✗	✓	✗
HortonHash	✗	✓	✗	✓	✓	✗	✗	✗
Our Hash Table Approach	✓	✓	✗	✗	✗	✗	✓	✗
MemcachedGPU	✗	✓	✗	✗	✓	✓	✓	✗
StadiumHash	✗	✓	✗	✗	✓	✓	✓	✓
Perfect Hashing:								
PerfectHash	✗	✓	✓	✓	✓	✗	✗	✗
Spatial Hashing:								
BiLevelLSH	✗	✓	✗	✗	✗	✓	✓	✗
EGSH	✗	✓	✗	✓	✓	✗	✗	✗
VoxelHash	✗	✓	✗	✗	✗	✗	✓	✗
Separate Chaining:								
SlabHash	✗	✓	✓	✓	✗	✗	✓	✓

3 Basics

In this section, we look at the basics of hash tables and CPU and GPU acceleration.

3.1 Hash tables

Hashing is a widely used technique to quickly store data inside a structure like for example a hash table. Therefore, we can use one or multiple different hash functions to translate a value into a hash. Hash tables often use a prime number for the size and the hash function includes a modulo operation with the prime number to shrink the range of values into the hash table size.

A hash table without collisions maps the key to a field in O(1) because we just calculate the hash value and access this position. The hash is ideally unique because collisions would lead to overwritten fields. However, hash functions are not collision free for arbitrary input, and we, therefore, need strategies to resolve these conflicts. There exist different strategies like linear probing [17], quadratic probing [17] or double hashing [18]. Another solution is, e.g., to use cuckoo hashing with multiple tables.

Cuckoo hashing [19] uses two or more tables and a hash function for each table: First, we try to insert the key into the first table by using the corresponding hash function. If the position is already occupied, we relocate the element which occupies the space from the first table into the second table and insert the element. If the position in the second table is already occupied, we insert the element there and move the previous element of this table to the next table and so on. Cuckoo hashing has a high memory efficiency [20] which is good for fast data transfer to the GPU. Additionally, cuckoo hashing has an amortized insertion and retrieval time of O(1) [20].

3.2 Hardware acceleration

Modern CPUs and GPUs are designed for high parallelism and high bandwidth. There are many different vendors for graphics hardware and each vendor develops its own API for their hardware, but some also support other hardware from other vendors. In addition, cross-vendor APIs are developed by hardware independent developers. On modern multi-core and many-core systems, there are a variety of different approaches for high-performance computing. The first solution is to use multiple threads and distribute the workload onto the threads. A second idea is to use libraries like OpenMP[3] and Intel Threading Building Blocks[4] for loop and algorithm parallelism. Another technique uses asynchronous computation with tasks and coroutines. Each approach has different advantages and disadvantages like threads have an overhead and need synchronization, but allow manual optimization and custom parallelism. As an additional challenge, there are multiple different implementations for the same concept in different languages. An interface for GPU acceleration is the widely used CUDA Toolkit,[5] but it is limited to NVIDIA GPUs. CUDA offers high performance and extensive tooling and documentation. Another advantage is that many different frameworks and solutions are based on CUDA. For cross-platform GPU acceleration there are OpenCL and Vulkan[6] which are both developed by the Khronos Group. OpenCL is designed for cross-platform computing in contrast to Vulkan primarily designed for 3D rendering, but can also be used for computing. Furthermore, there exist other enterprise solutions like AMD's ROCm[7] for AMD's datacenter cards. Intel's OneAPI is a hybrid concept for CPU, APU, FPGA and GPU computing and also allows cross-platform development for different hardware.

4 Parallel hybrid GPU/CPU hash table for string keys

In this section, we introduce our acceleration design. We start with the approach followed by the data structure and how rollout and search-methods are designed. In the last subsection, we look at hybrid approaches to combine CPU and GPU compute power.

4.1 Approach

For our design, we choose an already implemented and well-known hash table implementation. Therefore, the robin-map[8] is used as a representative for a basic hash table. This implementation is based on robin hood hashing [21] and is also faster than the C++ build-in unordered_map. Robin hood hashing solves collisions by superseding existing elements which are closer to their original index position and these elements have to be relocated to indexes further away. As an example of a high performance well-known variant of hashing, we choose the libcuckoo library which already has been considered by several different scientific contributions like [22] and [20]. This library implements the widely used cuckoo hashing.

[3] https://www.openmp.org/, accessed on March 08, 2022.

[4] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html, accessed on March 08, 2022.

[5] https://developer.nvidia.com/cuda-toolkit, accessed on March 08, 2022.

[6] https://www.vulkan.org/, accessed on March 08, 2022.

[7] https://rocmdocs.amd.com/en/latest/#, accessed on March 08, 2022.

[8] https://github.com/Tessil/robin-map, accessed on March 08, 2022.

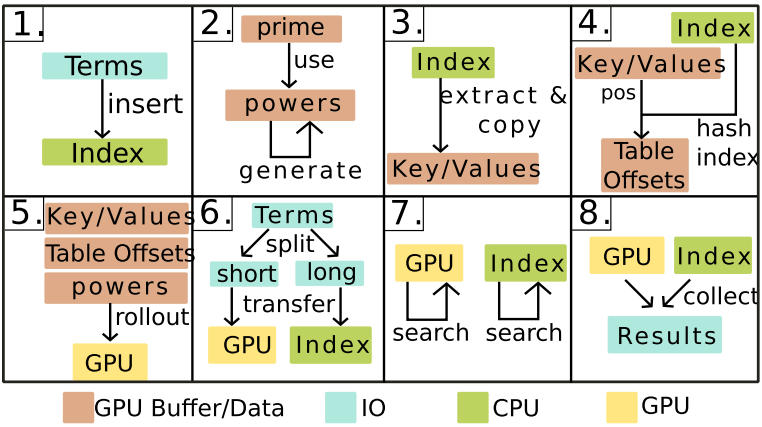


Fig. 1 This figure presents the steps that are necessary to process a search in a hybrid hash table. First terms are inserted in the index, then a prime number and powers are randomly generated. In the next step, the index of the terms is copied to the OpenCL buffer before the hash is mapped to the buffer index and the content is the position in the key/values buffer. In step 5, all buffer content is transferred to the GPU. The terms are split into long and short keys and are transferred to the corresponding computation device. Finally, we start the search and transfer the results back to the host [2]

The main difference between the existing algorithms and our approach is the support of string keys. For these string keys, we need a hash algorithm to convert the keys to a hash. The requirements for this hash algorithm are that it is simple, efficient and has a relatively low collision rate. A simple approach is to use a polynomial rolling hash function like the Rabin–Karp algorithm [23]. Another advantage is that these kinds of algorithms can also be accelerated by GPUs [24]. Therefore, we define and implement the polynomial rolling hash function. The following parameters are provided: $S = [s_0, ..., s_{n-1}]$ is the string key composed of the characters s_i, n is the number of characters in S, $P = [p_0, ..., p_{n-1}]$ is an array of random powers, and M is a prime number. We define the hash $H_M(S, P)$ of S as:

$$H_M(S, P) = \begin{cases} (s_0 * p_0) \; mod \; M & n = 1 \\ (H_M([s_0, ..., s_{n-2}], [p_0, ..., p_{n-2}]) + s_{n-1} * p_{n-1}) \; mod \; M & n > 1 \end{cases}$$

The operation in a hybrid search is shown in Fig. 1. We first fill our hash table and then generate an OpenCL data structure before transferring this structure. When we split the requests into batches and distribute the batches by different metrics on CPU and GPU, one metric can be the length of the keys. Afterward, we start the search on both accelerators and if these searches are finished, then we collect the results of both searches.

4.2 Data structure and search

Our GPU approach provides a generic interface for different hash tables. Therefore, we can use the same OpenCL implementation for robin-map and libcuckoo. To support this, we also need a generic OpenCL data structure for the GPU memory. OpenCL uses 1-dimensional buffers for data transfer and each buffer element can store custom or built-in types. Furthermore, OpenCL has an image type for 2- or 3-dimensional data. For the hash map itself, we work with a 1d-buffer storing the key-value pairs separated by a NULL byte. Because our values are 64 bit long values, every value takes up 8 bytes. We call this the terms buffer as

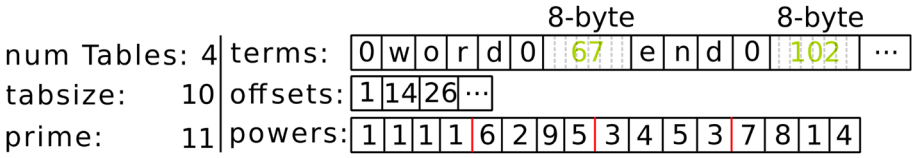


Fig. 2 This is the memory layout of the hash table on the GPU. We use three OpenCL buffers and three values. The values are the number of tables, the maximum size of the table and the prime number for hashing. The first buffer term contains the key-value pairs separated by null bytes. The second buffer contains the start offsets for the key-value pairs and the index corresponds to the hash of a key. The third and final buffer contains the generated powers for the hash algorithm [2]

shown in Fig. 2. Now we have a buffer with data, but for efficiency, we use an offset buffer for random access of each key-value pair. This 1d-buffer for the offsets has a special order which is generated throughout a rollout. The indexes of this buffer correspond to the hash value of the key which is located behind the stored offset. The last buffer contains the powers for the hash algorithm and can be stored and transferred in different ways. We need enough space for the maximum string length multiplied by the number of tables. For this memory, we can use a simple flat 1d-buffer like is shown in Fig. 2. Alternatively, we can use the OpenCL built-in vector types for the number of tables, which has the advantage that we can calculate the hash for a single character in all tables with one vector operation. A disadvantage is that the OpenCL kernel has to be constructed during initialization because we do not know the number of tables beforehand. Furthermore, we also need a prime number for the hash algorithm which is randomly generated and transferred to the OpenCL kernel as an argument alongside the number of tables and the tables size.

After we transferred the buffers to the GPU, we use the hash function to calculate the hash for each key and look in the offsets buffer to find the position in the terms buffer. Then, we compare the keys with each other. If we have a match, then we return the result and otherwise, we continue with the next table. If the last table does not have the key, it is not in the structure and we return not found. A performance benefit can be achieved by the host OpenCL control structure. We are processing the requests in batches and use a simple form of pipelining by splitting the batch into two halves. We start by converting the keys of the first half to a GPU structure which consists of a character buffer and an offset buffer for the start position of each key. Then, we transfer these data and start the search by inserting the write, execute and read operations into a queue. Meanwhile, the second half is prepared and transferred and the OpenCL operations are inserted into a second queue. Afterward, we wait until both queues are finished.

5 Evaluation

In this section, we evaluate our approach. We start with the environment and the different benchmark mods, then we look at the results.

5.1 Benchmark framework

For benchmarking and comparing different hash implementations and other data structures, we develop a hybrid multi-threaded framework called H^2. The reason for this framework is to load or generate random workloads for testing and measure the execution times over

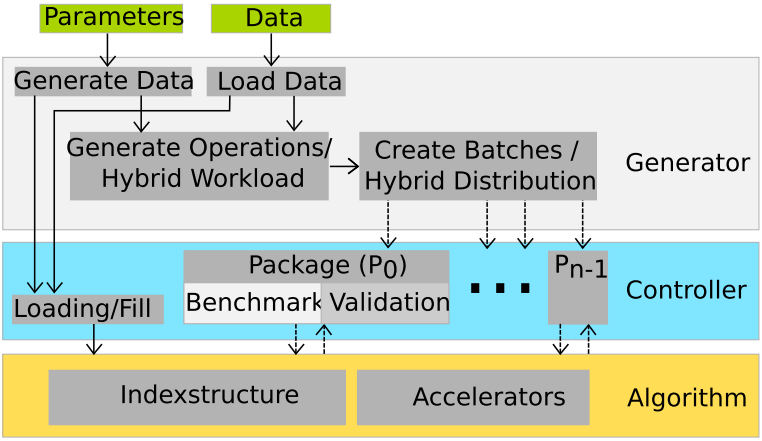


Fig. 3 Benchmark Framework Structure: the framework consists of three different parts, first the data and batch generators followed by the controller for each individual hybrid configuration (package) and then specific algorithms with index and accelerators [2]

multiple runs in different data structures. It is possible to switch between different data structures during runtime to compare different implementations in one run. Also, the H^2 framework can process complete series where each point can be executed multiple times to reduce statistic variance.

For performance purposes, we use modern C++ with Boost for the H^2 framework. For acceleration, H^2 can support different devices and APIs. Currently, CPU, GPU and FPGA are supported devices and OpenCL, CUDA and OneAPI[9] are supported APIs. The general architecture is shown in Fig. 3. The key-value pairs are either generated inside the framework or are loaded from a flat text file with one pair per line. H^2 is optimized for batch processing and supports different hybrid and non-hybrid scenarios. The currently supported scenarios are all batches on CPU, all batches on GPU and split batches according to a certain percentage or a certain maximum length. The H^2 framework has the ability to validate the results and to log resources like RAM, GPU memory and utilization usage. H^2 supports Linux and Windows operating systems.

Each benchmark run is split into two different phases, the loading or fill phase and the run phase. First, the key-value store is initially filled with a certain number of pairs and an initial rollout to the accelerator is done. In the next phase, batch requests are sent multi-threaded to the different accelerators and H^2 waits for the completion of each thread. Afterward, the execution time and throughput are written to an output file and the key-value store and accelerator are reset to the initial state. Now another pass-through or a different configuration can be executed. Also, we use a Node.js[10] application written in TypeScript[11] for result aggregation and diagram generation. The basic configuration and benchmark setup can be handled via one single JSON file. Each file can store multiple series and these are combined into a single diagram.

[9] https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html, accessed on March 08, 2022.

[10] https://nodejs.org/en/, accessed on March 13, 2022.

[11] https://www.typescriptlang.org/.

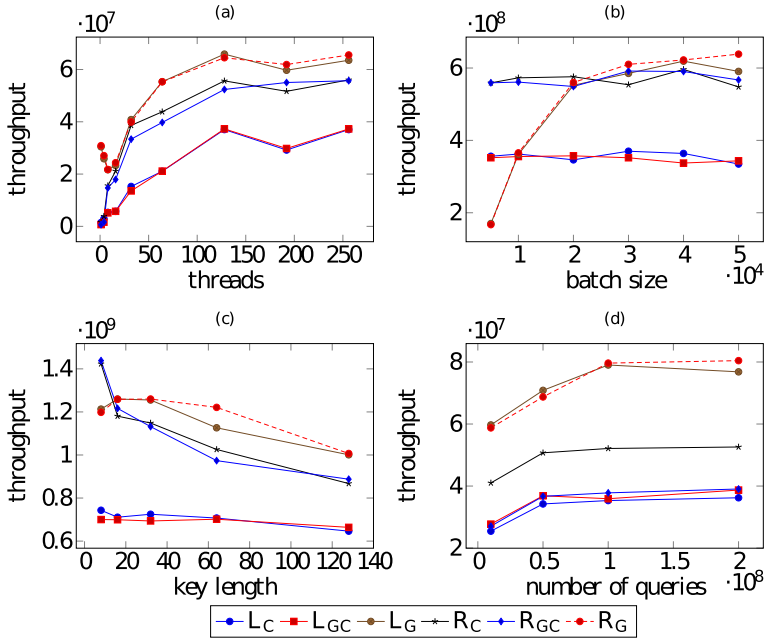


Fig. 4 Benchmark with throughput in case of **a** increasing number of threads, **b** batch size, **c** key length and **d** number of queries with BTC data [2]

5.2 Benchmark environment

We use a modern many-core high-performance computing NUMA systems for scientific computations. This system contains two AMD EPYC 7542 processors with 32-Cores each and a core can handle two threads (Hyper-Threading). Furthermore, the system has 2 TiB of high bandwidth RAM for Big Data computing. For acceleration, we use an NVIDIA A100 graphics card with 40 GiB memory. For real-world data, we use the complete triples data (btc2019-triples.nt.gz) from the 2019 Billion Triple Challenge[12] (BTC) and filter the data after key length requirements.

5.3 Benchmark results

We start by comparing the variants against each other in Fig. 4. Therefore, we use abbreviations to avoid ambiguity as follows:

- libcuckoo CPU/libcuckoo GPU (L_C/L_G),
- robin-map CPU/robin-map GPU (R_C/R_G),
- libcuckoo GPU approach on CPU (L_{GC}),
- robin-map GPU approach on CPU (R_{GC}).

In Fig. 4a, we compare different numbers of threads. We use a thread pool with the same size as the threads but capped it at 128 threads, which is the maximum number of parallel threads. All approaches benefit from more threads, but L_G and R_G have the highest

[12] https://zenodo.org/record/2634588, accessed on March 13, 2022.

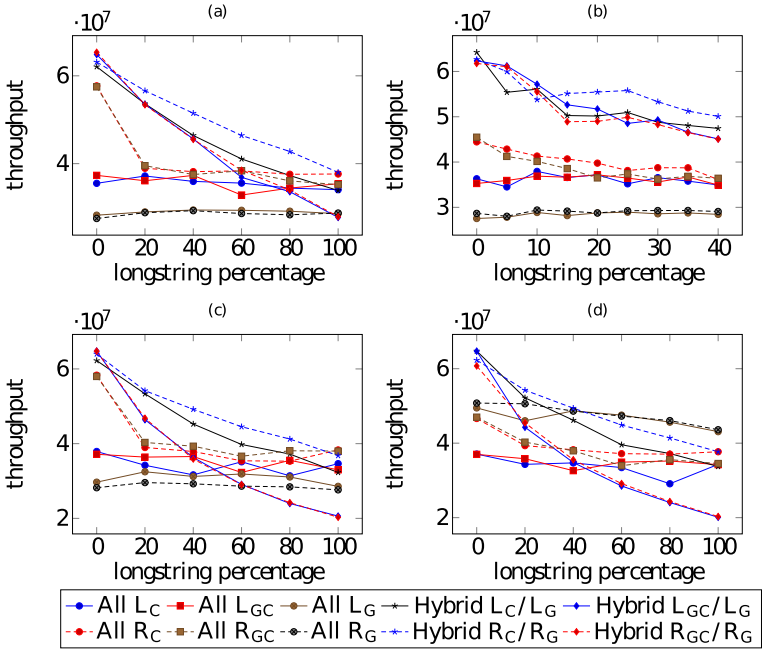
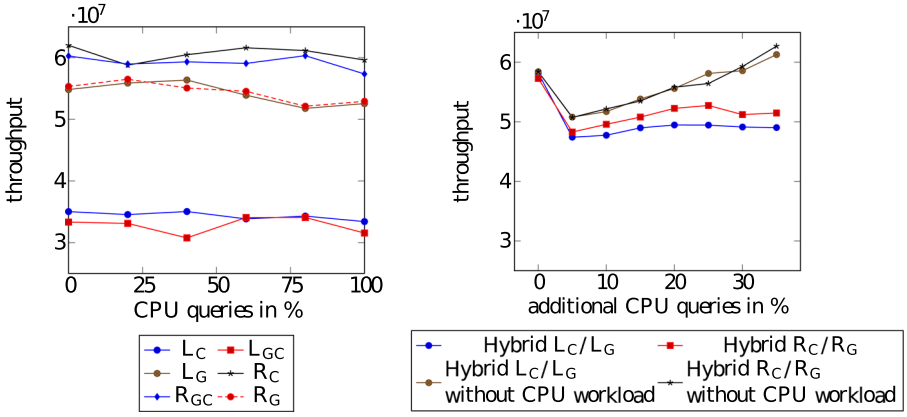


Fig. 5 Hybrid throughput with increase in long string amount, 128 Threads GPU, 128 Threads CPU, **a**, **c** short string length 8 and (**b**, **d**)32, (**a**, **b**) long string length 128 and (**c**, **d**) 256, **b** shows the long key range between 0 and 40% in detail [2]

throughput. L_C is on the same level as L_{GC} and the same is true for R_C and R_{GC}. In Fig. 4b, we consider the batch size. We see that high batch size is important for GPU acceleration because L_G and R_G increase their throughput significantly with larger batches and reach a plateau by 30,000 operations. We can see that we need a certain amount of operations to be efficient because we have some overhead at the start and the end of the pipeline. The third diagram (Fig. 4c) shows the effect of the key length. If the keys get longer, all approaches need more time for comparing the keys and the throughput is decreasing. We can also see that for short strings, R_C and R_{GC} are faster. In the fourth diagram (Fig. 4d), we look at the real-world BTC data with key length 32. We increase the number of queries for better utilization and see that both L_G and R_G are the fastest and that there is also a gap to R_C. L_{GC}, R_{GC} and L_C are even slower as R_C and on the same level. The speed-up is 2.1 between L_C and L_G and 1.5 between R_C and R_G.

Now we compare different query splittings and thread counts in a hybrid scenario. The parameter for splitting is the key length and we have a limit after which we distribute the terms to the CPU approach. In the following chapter, we define long strings as strings that are longer than a certain threshold and are moved to the CPU if not specified differently. We start with zero percent long strings in our query set and increase the amount until all keys are long strings. In Fig. 5, we compare the performance of a single approach with all queries against the hybrid combinations of two approaches split between them. We can see in the graphs that with an increasing amount of long strings all hybrid implementations lose throughput and tend toward the all CPU and GPU performances. But for a medium percentage, the combination of both approaches is faster than CPU or GPU approach alone.



(a) Throughput with increasing CPU amount and decreasing GPU amount

(b) Hybrid throughput with increasing additional CPU amount

Fig. 6 Further hybrid results with a percentage of (**a**) queries processed on the CPU and with a constant load on the GPU and (**b**) additional queries added to the CPU [2]

In Fig. 6a, we simply increase the share of CPU queries in percent but leave the total number of queries constant. We see that this does not change the performance very much, but the throughput of all approaches drops with all queries on the CPU. In Fig. 6b, we have a constant amount of short length keys and add additional longer keys. The throughput of the hybrid variants L_C/L_G and R_C/R_G increases at the beginning, but they reach a maximum at 25% additional queries and make a slight drop after. This is the case because L_C and R_C slow the algorithm down. We check this by simply removing the workload on L_C and R_C and keeping the workload on L_G and R_G. One reason is that the parallelism of GPUs is much more efficient than a high number of CPU threads.

6 APU acceleration

In this section, we look at the architecture of the APU in comparison with the dedicated GPU.

6.1 Overview

APUs are a combination of a GPU and a CPU in one chip and can be used for heterogeneous computing [3, 25]. APUs feature more efficient communication and resource sharing. They are typically used in embedded systems, smartphones and laptops. But some light forms of integrated GPUs are also existing in desktop processors like Intels Raptor Lake or AMDs Ryzen 7000 series. Integrated graphics (iGPU) usually have lower parallelism and raw compute power in comparison with dedicated graphics cards (dGPU) because they are designed for mobile devices with battery power. For example, the Intel 1165g7 can process 512 work items in parallel but the Nvidia A100 can process 1024 work items in 1-dimension and also the number of compute units and the clock speed are reduced. Another performance impact is related to the memory speed because iGPU and CPU use relatively slow memory like DDR4 or LPDDR4 and dGPUs use very fast memory like GDDR6, GDDR6x or HBM2. In

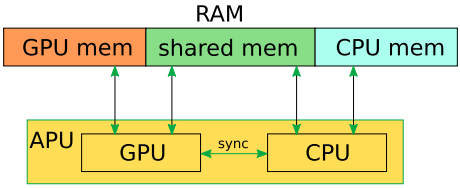


Fig. 7 Simplified APU memory layout, a part of the RAM is exclusively reserved for the iGPU and other parts of the memory can be shared between CPU and iGPU

the APU case, there is usually a part of the main memory assigned dedicated to the iGPU and the rest of the memory can be shared with the CPU.

6.2 Shared memory

Unlike dGPUs which send and receive data through the PCI-Express bus and feature their own dedicated very fast memory on the card, the APU can share the same memory (see Fig. 7). This shared memory can be accessed by the CPU and iGPU without coping or remapping. This enables more possibilities for inter-operation between CPU and GPU like sharing the same address space. If CPU and GPU use the same address space, pointers stay valid between both units. This makes the memory very flexible because it allows using pointer base structures on the iGPU and the CPU by also avoiding data copy or duplication. But naturally shared memory requires synchronization to avoid data races.

6.3 APU implementation

We realize the algorithm for searching in hash tables with the Intel OneAPI Toolkit. It is a platform independent toolkit for different devices like CPU, GPU, APU and FPGA and provides a unified method to use the acceleration. Therefore, it uses the SYCL language which is developed by the Khronos Group. The toolkit features three different memory types host, device and shared. Host memory is located in the host memory, device memory is located on the accelerator device, and shared memory can be located in each of them. Also, SYCL supports universal shared memory (USM) which allows pointers to be valid across host and device. But in our benchmark case, all three memory types and USM are located in the same main memory.

In the case of the implemented libcuckoo hash table, we separate the data structure from the table logic. This data structure can be stored in the USM and we accessed it via the separated logic which is slightly modified for each different accelerator. Because we can use a pointer base structure on the APU we can also support insert, update and erase operations in the libcuckoo data structure but we limit the initial implementation to a single thread to avoid extensive dependency handling. The shared memory handling and operation flow of insertion is shown in Fig. 8 for each accelerator. We also improve the structure of our benchmark framework slightly and group the hardware accelerator and algorithm combinations into shared libraries, because it makes the handling of different compilers and build tool chains easier. Also, it allows special optimization for each accelerator, API and data structure.

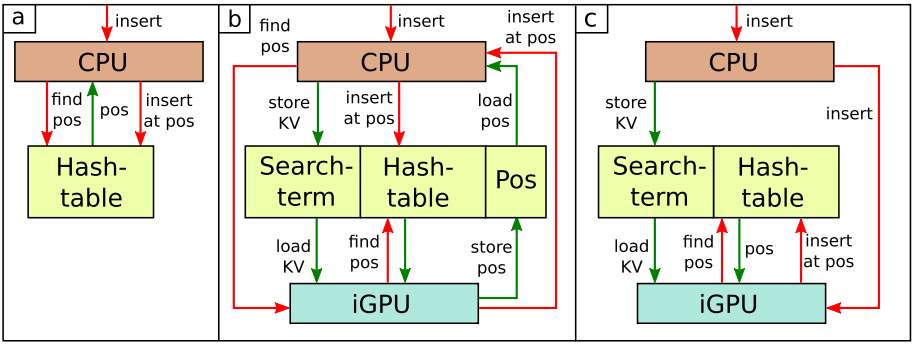


Fig. 8 Insertion in libcuckoo hash table stored in shared memory through the APU in different modes, **a** only through the CPU, **b** finding the insert position on the iGPU and insert on CPU, **c** only through the iGPU

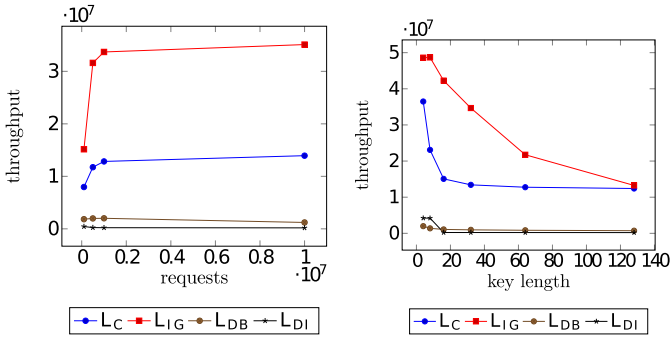
7 APU benchmark results

In this section, we look at the results for each proposed APU idea. We run our experiments on a fast laptop with Intel i7-1165G7, 32 Gigabyte of RAM and an Intel Iris Xe Graphics with 96 compute units. As compilers, we use Clang/LLVM 15 and the Intel DPCPP compiler. We run the following six benchmarks with synthetic data and the ported OpenCL variant libcuckoo DPCPP (L_{DB}) and the original index structure DPCPP variant (L_{DI}). The previous libcuckoo GPU variant (L_G) is measured on the iGPU and called L_{IG} in this context. For libcuckoo inserts (LI), updates (LU) and erases (LE), we have four different algorithms. These are the algorithms explained for inserts and updates/erases are analogous:

- Original insert on CPU (LI_C).
- Finding on iGPU and insert on CPU (LI_{GC}).
- Finding on iGPU and insert on iGPU separate kernels (LI_{GGS}).
- Finding on iGPU and insert on iGPU in one kernel (LI_{GGC}).

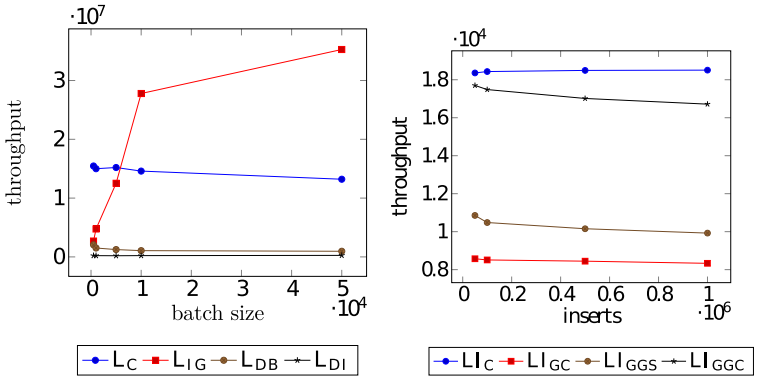
In Fig. 9a, we see that L_{IG} is much faster when L_C is processed on a larger number of requests. Both algorithms (L_{DB}, L_{DI}) are the slowest in the current implementations. The buffer implementation is also faster than the pointer base index structure. Then, we look at the key length in Fig. 9b: we can see that with increase in length the throughput of L_{IG} drops fast to the L_C performance level. L_{IG} is also heavily profiting from larger batch sizes (Fig. 10a). If we look at inserts (Fig. 10b) the CPU is much faster than both the GPU position look-up and insertion on the CPU or GPU. But the insertion on the GPU is faster than the look-up on the GPU and insert on the CPU. For update operations (Fig. 11a), the results are similar: the original CPU variant L_C is the fastest and the iGPU look-up of the position is slower but processing the updates completely on the iGPU is faster as iGPU look-up and CPU update. This result is also true for erase operations (Fig. 11b).

In hybrid benchmarks (see Fig. 12), we can see that at the beginning Hybrid L_C/L_{DB} and Hybrid L_{GC}/L_{DB} are slower as Hybrid L_C/L_G and Hybrid L_{GC}/L_G. If we increase the number of processed long strings Hybrid L_C/L_{DB} passes Hybrid L_{GC}/L_G between 60 and 80% and reaches Hybrid L_C/L_G with 100% long strings. Also, our original hybrid algorithm with CPU and GPU combined (see Fig. 13) is faster than the CPU and GPU alone if the number of long strings is below 20–25%.



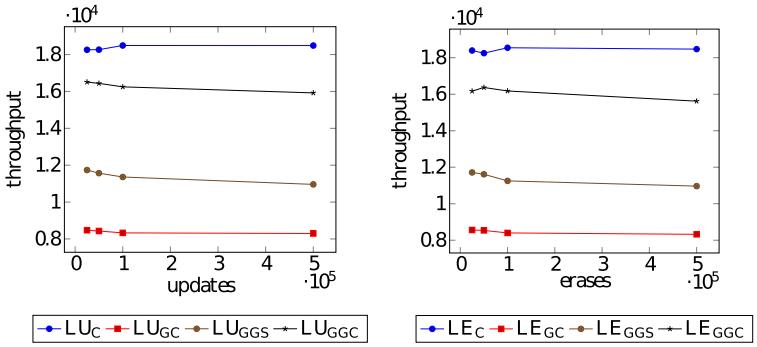
(a) Throughput with increasing search requests (b) Throughput with increasing key length

Fig. 9 Benchmarks with number of requests and key length



(a) Throughput with increasing batch size (b) Inserts through the CPU and iGPU position look-up with inserts on the CPU and iGPU

Fig. 10 Batch size throughput and insert throughput benchmarks



(a) Updates through the CPU and iGPU position look-up with update on the CPU and iGPU (b) Erases through the CPU and iGPU position look-up with erase on the CPU and iGPU

Fig. 11 Update and erase throughput benchmarks

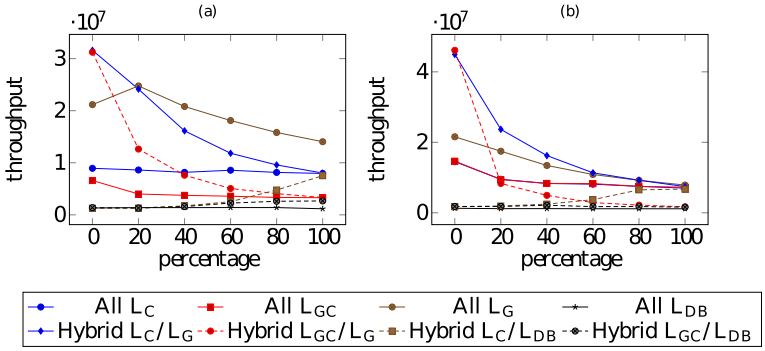


Fig. 12 Hybrid benchmarks in range 0–100% with short strings of length 8 (**a**) and 32 (**b**) and long strings of length 256 (**a**) and 128 (**b**), the amount of short and long strings in the table is 50%

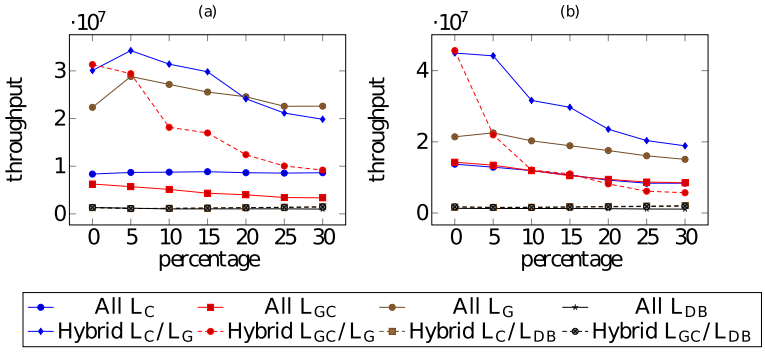


Fig. 13 Hybrid benchmarks in range 0–30% long strings, with short strings of length 8 (**a**) and 32 (**b**) and long strings of length 256 (**a**) and 128 (**b**), the amount of short and long strings in the table is 50%

8 Summary and conclusions

We design a GPU acceleration string-based GPU implementation for robin-map and libcuckoo and integrate it into our hybrid benchmark framework H^2. Furthermore, we propose a parallel hybrid GPU/CPU hash table for string keys of variable length. OpenCL is used for GPU acceleration and we use traditional thread in a thread pool for CPU parallelism. Then, we evaluate our implementation against the original CPU variants. Our approach has a higher throughput than the CPU variant and achieves a speed-up of 2.1 compared to libcuckoo and 1.5 compared to robin-map. For long strings on the CPU and short strings on the GPU, our proposed hybrid approach is faster and has a 40% higher throughput with 20% long keys on the CPU.

In addition, we analyze hybrid approaches with libcuckoo in an APU laptop context both with our buffer structure and also with the original architecture. Also, we implement sequential inserts, updates and erase operations through the APU in shared memory. Further iGPU look-ups and CPU/iGPU inserts, updates and erase operations are implemented.

In future work, the modified hash tables could be merged with the already existing GPU structure to minimize the amount of data that has to be transferred. Another direction is the use FPGAs, and to investigate different types of basic indices for hybrid index structures.

Funding Open Access funding enabled and organized by Projekt DEAL. This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—Project-ID 422742661.

References

1. Lessley B, Childs H (2020) Data-parallel hashing techniques for GPU architectures. IEEE Trans Parallel Distrib Syst 31(1):237–250. https://doi.org/10.1109/TPDS.2019.2929768
2. Groth T, Groppe,S, Pionteck T, Koppehel M, Valdiek F (2022) Accelerated parallel hybrid GPU/CPU hash table queries with string keys. In: The 33rd international conference on database and expert systems applications (DEXA), Vienna, Austria .This paper received the Norman Revell Best Paper Award. https://doi.org/10.1007/978-3-031-12426-6_15
3. Mittal S, Vetter JS (2015) A survey of CPU-GPU heterogeneous computing techniques. ACM Comput Surv. https://doi.org/10.1145/2788396
4. Behrens T, Rosenfeld V, Traub J, Breß S, Markl V (2018) Efficient simd vectorization for hashing in opencl. In: EDBT, pp 489–492
5. Jünger D, Kobus R, Müller A, Hundt C, Xu K, Liu W, Schmidt B (2020) Warpcore: a library for fast hash tables on GPUs. In: 2020 IEEE 27th international conference on high performance computing, data, and analytics (HiPC), pp 11–20. https://doi.org/10.1109/HiPC50609.2020.00015
6. Merrill DG, Grimshaw AS (2010) Revisiting sorting for GPGPU stream architectures. In: Proceedings of the 19th international conference on parallel architectures and compilation techniques. PACT '10. Association for Computing Machinery, New York, pp 545–546. https://doi.org/10.1145/1854273.1854344
7. Ashkiani S, Farach-Colton M, Owens JD (2018) A dynamic hash table for the GPU. In: 2018 IEEE international parallel and distributed processing symposium (IPDPS), pp 419–429. https://doi.org/10.1109/IPDPS.2018.00052
8. Green O (2021) Hashgraph-scalable hash tables using a sparse graph data structure. ACM Trans Parallel Comput. https://doi.org/10.1145/3460872
9. Li Y, Zhu Q, Lyu Z, Huang Z, Sun J (2021) Dycuckoo: Dynamic hash tables on GPUs. In: 2021 IEEE 37th international conference on data engineering (ICDE), pp 744–755. https://doi.org/10.1109/ICDE51399.2021.00070
10. Lupescu G, Tapus N (2021) Design of hashtable for heterogeneous architectures. In: 2021 23rd international conference on control systems and computer science (CSCS), pp 172–177. https://doi.org/10.1109/CSCS52396.2021.00035
11. Ashkiani S, Farach-Colton M, Owens JD (2018) A dynamic hash table for the GPU. In: 2018 IEEE international parallel and distributed processing symposium (IPDPS), pp 419–429. https://doi.org/10.1109/IPDPS.2018.00052
12. Daga M, Nutter M (2012) Exploiting coarse-grained parallelism in b+ tree searches on an APU. In: 2012 SC companion: high performance computing, networking storage and analysis, pp 240–247. https://doi.org/10.1109/SC.Companion.2012.40
13. Luan H, Fu Y (2022) Accelerating group-by and aggregation on heterogeneous CPU–GPU platforms. In: Xie Q, Zhao L, Li K, Yadav A, Wang L (eds) Advances in natural computation, fuzzy systems and knowledge discovery. Springer, Cham, pp 980–990
14. Villegas A, Navarro A, Asenjo R, Plata O (2019) Toward a software transactional memory for heterogeneous CPU–GPU processors. J Supercomput 75(8):4177–4192

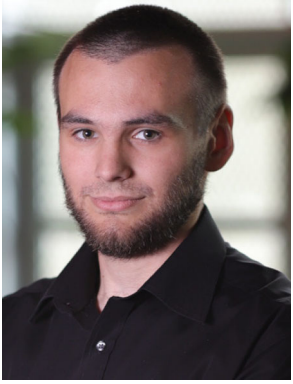
15. Kulikov D, Nikolskaia D, Kurapov P (2021) Efficient hardware-agnostic DBMs operator implementation using SYCL. In: 2021 international conference engineering and telecommunication (En&T), pp 1–5. https://doi.org/10.1109/EnT50460.2021.9681747
16. Breyer M, Daiß G, Pflüger D (2021) Performance-portable distributed k-nearest neighbors using locality-sensitive hashing and SYCL. In: IWOCL'21. Association for Computing Machinery, New York. NY, USA https://doi.org/10.1145/3456669.3456692
17. Knuth DE (1974) The art of computer programming, vol. 3: sorting and searching. Addison–Wesley
18. Guibas LJ, Szemeredi E (1978) The analysis of double hashing. J Comput Syst Sci 16(2):226–274. https://doi.org/10.1016/0022-0000(78)90046-6
19. Pagh R, Rodler FF (2001) Cuckoo hashing. In: auf der Heide, F.M. (ed.) Algorithms—ESA 2001. Springer, Berlin, Heidelberg, pp 121–133
20. Li X, Andersen DG, Kaminsky M, Freedman MJ (2014) Algorithmic improvements for fast concurrent cuckoo hashing. In: Proceedings of the 9th European conference on computer systems. EuroSys '14. Association for Computing Machinery, New York. https://doi.org/10.1145/2592798.2592820
21. Celis P, Larson P-A, Munro JI (1985) Robin hood hashing. In: 26th annual symposium on foundations of computer science (sfcs 1985), pp 281–288. https://doi.org/10.1109/SFCS.1985.48
22. Fan B, Andersen DG, Kaminsky M (2013) Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In: 10th USENIX symposium on networked systems design and implementation (NSDI 13). USENIX Association, Lombard, pp 371–384. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan
23. Karp RM, Rabin MO (1987) Efficient randomized pattern-matching algorithms. IBM J Res Dev 31(2):249–260. https://doi.org/10.1147/rd.312.0249
24. Dayarathne N, Ragel R (2014) Accelerating Rabin Karp on a graphics processing unit (GPU) using compute unified device architecture (CUDA). In: 7th international conference on information and automation for sustainability, pp 1–6. https://doi.org/10.1109/ICIAFS.2014.7069589
25. Daga M, Tschirhart ZS, Freitag C (2015) Exploring parallel programming models for heterogeneous computing systems. In: 2015 IEEE international symposium on workload characterization, pp 98–107. https://doi.org/10.1109/IISWC.2015.16

Tobias Groth is a research assistant at the Institute of Information Systems, University of Lübeck. He received his Master of Science in 2019. His research interests include GPU and APU acceleration, databases, modern technologies and programming languages.

Sven Groppe is a professor at the University of Lübeck. He received 7 project grants from DFG, BMBF and BMWi in the area of data management including the DFG project "Hybrid2-Index Structures for Main Memory Databases", which is the funding source of this contribution. He is the project coordinator of the BMBF funded QC4DB project about accelerating relational database management systems via quantum computing. He published more than 150 journal, conference and workshop papers at top-ranked publication venues including SIGMOD, VLDB and ICPP with over 125 co-authors from 17 countries all around the world. He is member of over 110 program committees of various conferences and workshops and reviewer of over 35 journals. He is a workshop chair of SBD@SIGMOD (2016-2020), BiDEDE@SIGMOD (2021-2023), VLIoT@VLDB (2017-2022) and QDSM@VLDB 2023. He is a general chair of the International Semantic Intelligence Conference (ISIC) (2021-2022), International Health Informatics Conference (IHIC) (2022- 2023) and the International Conference on Applied Machine Learning and Data Analytics (AMLDA) in 2023. More information is available on https://www.ifis.uniluebeck. de/ groppe/.

Thilo Pionteck received his diploma and doctoral degrees in electrical engineering from the Technische Universität Darmstadt, Germany, in 1999 and 2005, respectively. In 2008, he was appointed as assistant professor for Integrated Circuits and Systems at the University of Lübeck, Germany. After a number of substitute professorships, in 2015 he was appointed as a professor at the chair of Organic Computing at the University of Lübeck, with a research focus on adaptive digital systems. Finally, in 2016 he was appointed to the Otto-von- Guericke University, where he currently holds the chair of Hardware-Oriented Technical Computer Science. In his research, Thilo Pionteck focuses on new architectural concepts for the realization of runtime-adaptive, performance- and energy-efficient digital systems. His research interests include NoCs, 3D SoCs, adaptive system design, and dynamic partial reconfiguration of FPGAs.

Franz Valdiek is a research fellow at the Otto-von- Guericke University, Magdeburg. He received his master of science in 2022. Ever since, he focused his research on database management systems and their acceleration. Aside from that, he teaches basics of FPGA programming to bachelor students.



Martin Koppehel is a independent researcher at the Otto-von-Guericke University of Magdeburg. He has a MSc in computer science. Research interests include databases, site reliability engineering and machine learning.