KPU-SQL: Kernel Processing Unit for High-Performance SQL Acceleration

Hao Kong
konghao@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China

Haishuang Fan
fanhaishuang20z@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China

Jingya Wu
wujingya@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Liyun Cheng
chengliyun21s@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China

Yan Chen
yanchen@yusur.tech
YUSUR Technology Co., Ltd.
Beijing, China

Wenyan Lu
luwenyan@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Guihai Yan
yan@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Xiaowei Li
lxw@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

ABSTRACT

Application-specific accelerator is a prominent way for analytic query processing. To achieve a substantial improvement over the state-of-the-art in performance while maintaining programmability, we propose a kernel processing unit (KPU) framework and apply it to SQL acceleration. Kernel customization and data transmission are two critical bottlenecks, we separately optimize them in the key core and shadow core with a self-designed data management system. A software stack named RACE with a performance model and function simulator is also introduced. The experiments demonstrate that KPU-SQL outperforms the CPU and GPU by 24.5x and 8.75x on average, respectively.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; **Hardware-software codesign**.

GLSVLSI '23, June 5–7, 2023, Knoxville, TN, USA

ACM ISBN 979-8-4007-0125-2/23/06.
https://doi.org/10.1145/3583781.3590268

KEYWORDS

database; application-specific accelerator; programmable; hardware/software co-design; SQL analytics

ACM Reference Format:
Hao Kong, Haishuang Fan, Jingya Wu, Liyun Cheng, Yan Chen, Wenyan Lu, Guihai Yan, and Xiaowei Li. 2023. KPU-SQL: Kernel Processing Unit for High-Performance SQL Acceleration. In *Proceedings of the Great Lakes Symposium on VLSI 2023 (GLSVLSI '23), June 5–7, 2023, Knoxville, TN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3583781.3590268

1 INTRODUCTION

Computing accelerators have been witnessed in many domain-specific computing offloading engines implemented with either ASIC or FPGA substrates. Even though the accelerator-equipped systems usually deliver attractive performance and energy profiles, the extremely high complexity and non-recurring engineering cost render the successful stories far from prevalent. In database offloading, it's not difficult to design and implement a single-functional computing logic for analytic query processing, such as *sort* [11] and *join* [3], but how to make a flexible framework that can engage these "coarse-grained" units in a readily way, without incurring extra overheads from compilation, runtime, and OS, is a grand challenge.

The practice of the instruction-extension approach in general-purpose CPU is a good example to integrate special accelerators into the CPU architecture in a unified way. Such a tightly-coupled

style can optimally preserve the transparency of the programmer to the underlying hardware variants. However, the drawback is also prominent. Modern CPUs are already tens of billions of transistor monsters, integrating any more special-purpose accelerators, especially those with considerable silicon footprint, is not a trivial issue. This leads to not only larger chip area, but also design, verification, and testing cost, which directly contributes to the time-to-market and ultimately the product profitability. The above concern steers us to a clear direction: can we organize these accelerators in a synergistic way? The answer should be an architecture framework that holds the following merits:

- High performance with customization design.
- Unified ISA, reinforce easy programming.
- Cooperative, loosely coupled but easily cooperated with CPU pipeline.

We propose a novel accelerator-centric architecture, called *Kernel Processing Unit* (KPU), to fulfill this purpose. The key elements of a KPU are highly optimized hardware micro-computing units, i.e. kernels, which are designed for executing complex operators, such as hash, sort, join and so on, with a few semantic-rich instructions.

To achieve the above aims, we present a dedicated FPGA-based reference platform shown in Fig. 1. The KPU consists of a set of **table cores**, in which shadow cores imitate the behavior of the key core to process the satellite columns. Each core embraces the combination of simple pipeline and trigger instruction architecture[9], which helps to eliminate the program counter and avoid over-serialized execution. The data management system (**DMS**) is built from scratch to reach high bandwidth utilization and break the barriers of multi-level cache. We also propose a hardware-acceleration-friendly KPU instruction set architecture (**KISA**) that unifies the kernel interface and facilitates the integration of state-of-the-art designs.

To plug KPU into different relational DBMS non-intrusively and hybridize CPU-KPU to co-operate together with task dispatching strategy, a software stack called *Real-time Accelerate Computing Environment* (**RACE**) is proposed which has the ability to validate KPU design, evaluate its system level performance and abstract the boundary between the application and hardware device.

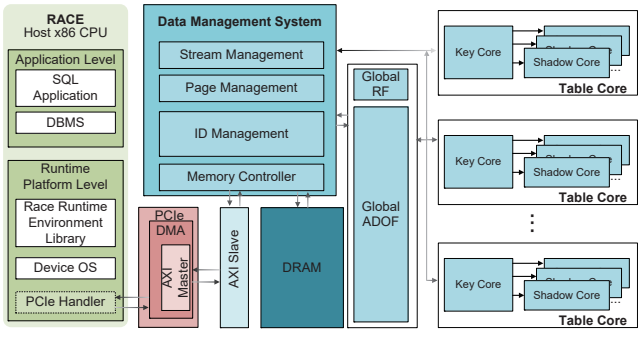

Figure 1: KPU-SQL architecture overview.

2 BACKGROUND AND MOTIVATION

General-purpose processors are often unable to meet real-time response needs and cost requirements for analytic query processing. As such, heterogeneous computing has emerged as the predominant method for SQL acceleration, often utilizing customized computation kernels that leverage complete application characterization. SQL operators are generally classified into two groups: scan-intensive and join-intensive[12].

For scan-intensive operators like filter, the computing capability is often the bottleneck, as continuous memory access can overwhelm the core computing power. To process such a large amount of data, PC-based instructions must execute serially to represent the producer-consumer relationship between SQL operators. However, this approach can introduce various types of hazards and result in significant idle time.

For join-intensive operators like sort and join, random memory access can cause serious bandwidth utilization decay. To mitigate this issue, one approach is to convert random memory access to continuous access, such as merge tree in sorting[10], but the satellite data which refer to the non-key column data still face the gather/scatter pattern problem. Another option is to break through the traditional multi-level cache structure and design a storage structure for streaming data, enabling computation to be embedded into the data stream. In column-oriented databases, data are available in streaming form with good locality and a low reuse rate, necessitating a unique memory management system.

Customized designs typically target a single kernel individually, such as sort [11], join [3], aggregation [5], and so on. While such designs offer high performance, they require significant alteration when integrated into a complete system and can create compatibility issues with each other. Additionally, these designs often lack a unified external interface, making integration challenging and limiting the possibility of flexible kernel expansion. By adopting a unified external interface, we can easily integrate any state-of-the-art design and detach it to be upgraded individually at low cost.

Database processing unit works as one way to solve the integration problem with a certain level of programmability. It gets design wins mainly based on its ability to provide an attractive point in the space of power-delay-cost-productivity trade-offs. A Previous study proposed RAPID that achieves a good performance/power design, by omitting features such as branch prediction, cache hierarchy, and a floating-point unit [1]. RAPID's initial prototype consists of 1440 database processing units, each with 32 dpCores and an 8GB of DDR3 memory. While programmability extends the lifetime of a product, it can come with a performance overhead. When comparing the performance of a multi-core chip design without RDMA, RAPID does not rank among the top-performance tier, as shown in table 1.

As the traditional processing architecture is based on individual data addressing, which brings two problems: 1) complex control, addressing control accounts for most of the instructions; 2) data access delay uncertainty, data transmission conflicts between the continuous data delay uncertainty, will introduce complex synchronization control. To improve bandwidth utilization, the addressing method of individual data is abandoned and the stream addressing

Table 1: The Performance of Different Database Processing Unit

Related Work	Data Size(GB)	Average Speedup
Q100[13]	0.01	70X
AQUOMAN[14]	1000	1.5~2X
RAPID[1]	1024	3.4X(From Total HW.)
DOE[7]	1	10~100X

method is adopted. Therefore, we designed the DMS, where the entire data stream is treated as an addressing unit. The entire data stream is stored continuously in memory, greatly simplifying addressing control, and shares a single entry in the address lookup table. The addressing method is inherently "burst" transfer, and the entire data stream is transferred continuously, improving data transfer efficiency and fixing the delay between consecutive data.

DOE[7] goes some distance towards realizing these objectives. To fully realize the requisite gains, DOE works as a co-processor, while the host CPU is only responsible for the global scheduling of data and generating query plans with the inner cost models. Instead, KPU avoids this pitfall by interleaving/hybridizing the CPU and KPU to cooperate together rather than omitting the computing power of the CPU.

3 KPU INSTRUCTION SET ARCHITECTURE

The KPU Instruction Set Architecture (KISA) follows the classical load/store RISC-like execution model. This model preserves a semantically simple interface with the other components outside of the KPU domain. Before delving into the details, the expression conventions are introduced first, as illustrated in Algorithm 1. A basic instruction consists of the operator type and source and destination operands, which are contained by either one or more immediate operands, register or *Atomic Data Object (ADO)* entries, or memory locations. Based on this, each kernel can share the unified external interfaces.

Algorithm 1 Trigger Instruction Mode

#**Rule** A
when (A.condition) **do**
OPTYPE [src.][src.]··· [dest.] (Status Config)

3.1 KISA Design Issues

As most application-specific instructions take multiple cycles to execute, the traditional multi-stage pipeline experiences significant stall time. To address this challenge, we introduce a new approach inspired by triggered instructions [9]. This approach performs guard-actions on status registers and is designed to work with the load-kernel-store and operator-at-a-time mode. Each kernel adopts a data-flow-driven execution method, and the input port of the kernel is equipped with a synchronous buffer. When data arrive at each port and activate the trigger conditions, the computation is automatically started without requiring external control intervention.

Aside from the register file designed for storing scalar data, an ADO is also used to buffer a batch of register elements with one single index. However, rather than simply extending the dimension of a register, ADO is considered the minimum unit of transmission and processing. The operator imposed on each ADO has to be carried out in an "atomic" manner to exploit data parallelism. For example, given an N-element ADO, and two types of operators, OP_α and OP_β, it is legal to apply either OP_α or OP_β to all elements of the ADO, but illegal to apply OP_α to some elements and OP_β to others. To facilitate the communication between ADOF(ADO File) and RF(Scalar Register File), augmented ***MOVE*** instructions are provided.

To address the inefficiencies in executing ***LOAD/STORE*** instructions, some auxiliary functions have been integrated into the design. For instance, *LOADS* is designed to compare the heads of two sorted sequences, and determines which one should be updated. Additionally, *STORF* has been implemented to filter *NULL* values when the intermediate results are stored into DRAM.

To improve performance in the OLAP domain, the need for ***branch*** and ***jump*** operations has been eliminated through accurate computational pattern analysis. Instead of using BEQ and BNE for loops to manipulate data recursively, trigger conditions are now used to facilitate the flow of data.

3.2 KISA Use Case

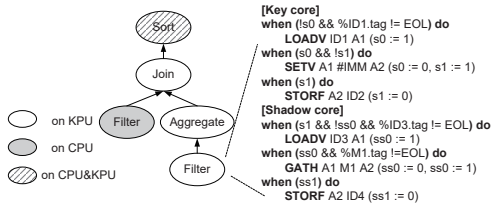

Figure 2: Query plan tree of TPC-H query-1 and partial KISA expression.

The KPU supports two offloading modes: 1) Partially offloading for large volume of data, in which CPU-bound and KPU-friendly operators are offloaded without interfering with the CPU-side query plan execution. This saves the transmission time and utilizes the residual computing power of the host. 2) Fully offloading for small volume of data, which is ideal for situations where most data reside on-board and are analyzed multiple times.

Using TPC-H Q1 as an example, the query plan can be represented as a tree structure, as shown in Fig. 2, with the order of operators being determined by the query optimizer. In fully offloading mode, each node in the tree can be translated into a set of KISA instructions, such as the filter shown in Fig. 2. In the case of partially offloading mode, CPU-bound operators can divide the query plan into multiple sub-trees, with different sub-trees typically operating on independent data tables. The low-cost sub-trees can still be executed by the CPU, and their results can be consumed by the blocked operators through the KPU. To reduce the frequency of DRAM reads and writes, the non-blocked node outputs can be directly fed into the parent node via ADOF, and the optimizer can also decide to cache high-value intermediate results as necessary.

4 KPU ARCHITECTURE

The KPU architecture is depicted in Fig. 3, and its key features can be summarized as follows:

- Control-lite. The control is largely in the hands of the host while the KPU is responsible for processing the data flow

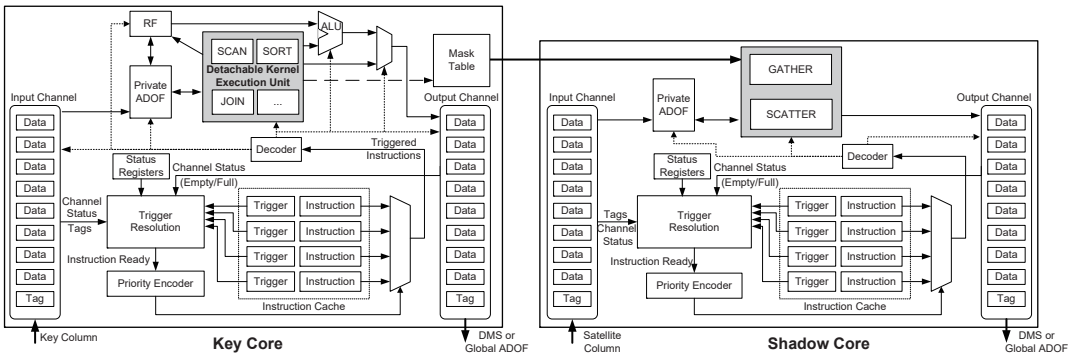

Figure 3: The micro-architecture of table core with one shadow core.

through a customizable kernel execution unit (KEU). This separation of control and data reduces centralized data synchronization control among multiple cores, branch prediction, and cache hierarchy.

- Memory-heavy. To adapt to the stream processing and column-oriented storage format, DMS is our key design, which focuses only on the core functions, including storage format, data pre-fetching, dynamic space allocation, and batch processing.

4.1 Table Core Level Parallelism

Regarding multi-core designs, there are two methods for deploying workloads: assigning one column to each core or distributing different batches of the same column to different cores. The former guarantees data integrity for each core and facilitates continuous memory access, but can cause mismatches between the column and core. The latter can circumvent this problem but force the need for a homogeneous core design, which results in wasted resources and functional redundancy. Both ways have their places in KPU, and KPU is predominantly in the former way with assistance in the latter one.

The SQL computational workload involves computation-intensive operations (such as comparison and arithmetic during filter) and memory-intensive operations (such as gather and scatter during projection). The latter can cause data-cache misses and occupies more cycles than the former, especially in the case of large joins [12]. Therefore, a versatile key core is designed specifically to process key columns, which is surrounded by a set of shadow cores for moving data of satellite columns. Both constitute a table core, which can save hardware utilization and improve performance by taking different optimization measures independently for different cores. Key cores can focus on the KEU design, while shadow cores emphasize data movement, reducing KEU idle cycles. For example, key cores process the key column to output a boolean mask, which shadow cores can use to filter satellite columns. When column-to-core mismatch problem occurs, the table core can be assigned to another task while the shadow cores continue processing redundant satellite data.

Traditional processor architectures typically use a centralized control module to manage data synchronization across multiple processing modules. However, this requires a large number of dynamic control instructions and can obstruct the execution process of each processing module. In our design, communication between multiple table cores is rare and data volume is small. Therefore, we divide ADOFs into two parts: private and global. Private ADOFs are only accessible within a single core, while global ADOFs are accessible by all cores. Global ADOFs are limited in number and modified by only one core at a time, avoiding data consistency problems. Operations, such as aggregation, that require collecting intermediate results from multiple cores, can also be accessed using global ADOFs/RF. In summary, KISA can precisely express all of these operations through global ADOFs/RFs.

4.2 Kernel Execution Unit

Macro SQL operators, such as filter, aggregation, sort, and join, are customized in the KEU, while other auxiliary functions are embedded in the KPU's ALU. GroupBy is usually a preemptive operation for filter and aggregation and is not tested separately. By using unified interfaces and incorporating state-of-the-art designs with some adjustments, we focus on optimizing the kernel design, especially sort and join, which are the two most costly relational operators and are commonly used in TPC-H queries [2, 4]. When evaluating the kernel's individual performance with data sizes varying from 1MB to 100GB, the results are all competitive, as shown in Fig. 4. Next, let's take a closer look at the internal architecture design:

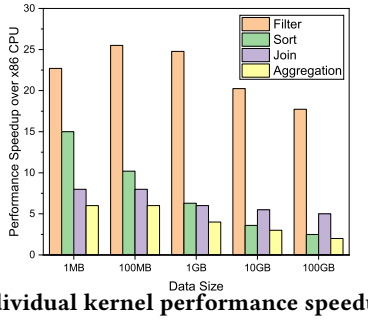

Figure 4: Individual kernel performance speedup over CPU.

Sort. Partitioning is excessively resource-consuming and requires multiple rounds of data scanning [14], so it is not deployed on the KPU board. Instead, to support large-scale data sorting, the KPU is responsible for sorting a specific volume of data, while the CPU merges all the sorting results. Pre-sorting is accomplished using bitonic sort, and the sorted sequences are merged using techniques derived from Bonsai[11]. As illustrated in Fig. 5(a), the *MERGE* instruction corresponds to a merge tree used to merge ordered sequences from A1 to A8, enabling ADO-length output per clock

cycle. The merge tree is comprised of a series of mergers (k-M) and couplers. The k-M is a bitonic partial merger that outputs k data per cycle, as shown in Fig. 5(b). The process of multiple merging passes is scheduled by *LOADV* instructions using different stream sources. To sort satellite data, we simply stitch together the row index with the satellite data, and in the final pass, the relative index of the output is sent to shadow cores.

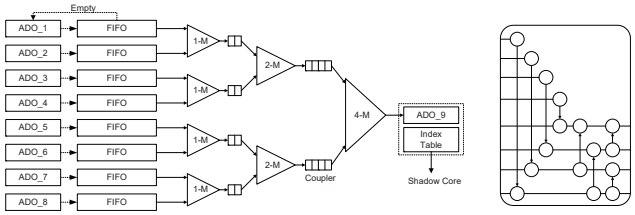

Figure 5: Merge sort architecture overview.

Join. To tackle semi-, anti-, and outer-joins, as well as address poor random memory accesses and read/write conflicts, we use the sort-merge join based on [8]. As shown in Fig. 6, *LOADS* is used to determine which ADO needs to be updated. Then, the adjacent data in the ADO are compared to identify the starting and ending positions of duplicate data, and to filter out non-duplicated data, thereby reducing the number of comparisons required in the final *Join* stage. When deduplicating, boolean comparison results are concatenated in the MSB of the data, which then goes through a bitonic sorter and the adjacent data in the sorting results are then compared with each other. The start and end positions are stored in a repetition table, along with the matching mask table. Both tables are then sent to shadow cores to re-arrange the join results.

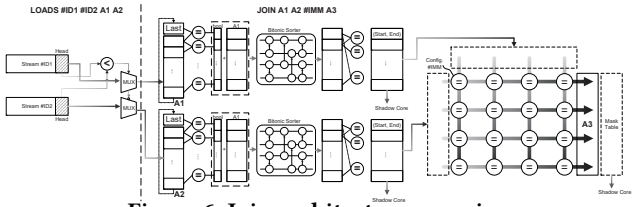

Figure 6: Join architecture overview.

4.3 Data Management System

Massive data processing and multi-core computing often lead to memory becoming a bottleneck when it comes to improving application performance, particularly in light of the following challenges in memory access and data interaction: 1) access conflicts caused by multiple cores simultaneously accessing memory require arbitration; 2) dynamic memory allocation, as SQL cannot predict result size in advance; and 3) complex data object management, with each processing core required to handle multiple data objects, such as integer, floating point, and string.

To address the aforementioned problems, a data management system has been designed, as shown in Fig.7. Data is accessed by ID, which serves as an addressing unit, instead of being identified with a physical address. Each column can be appointed with a unique ID, and LOAD/STORE instructions can be triggered with this ID without the need for repeatedly updating physical addresses. The outputs of each kernel are also assigned an ID. The bandwidth utilization has been tested with varying page sizes and burst lengths, and it can be achieved to nearly 90%, as shown in Fig.8. Performance decreases slightly when accessing non-continuous pages.

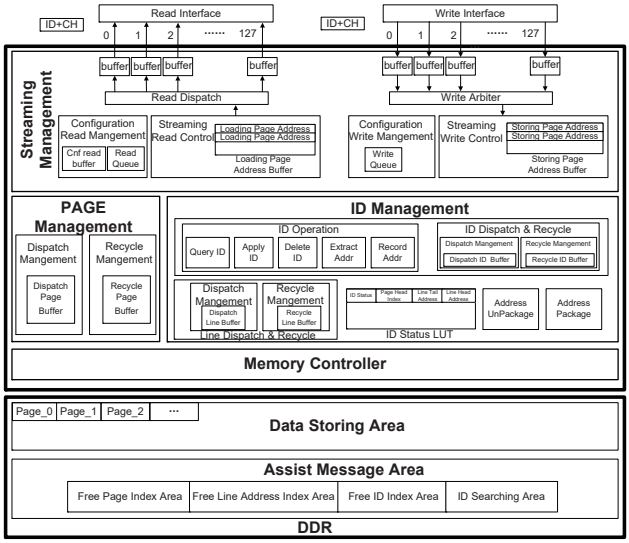

Figure 7: Architecture of data management system (DMS).

The DMS integrates page management, ID management, and streaming management, with page management being responsible for dynamic memory allocation and recovery, ID management for ID application and release, state querying, data loading and storage, and physical address extraction, and streaming management for the efficient loading and storage of multi-channel data. The following are some of the techniques and innovative strategies employed by DMS:

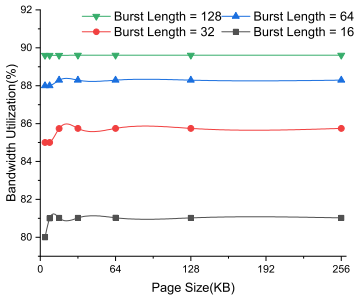

Figure 8: DDR Read/write performance with DMS.

1) To address the dynamic memory allocation problem, memory space is divided into multiple pages of fixed size (configurable as 4KB, 8KB, 16KB or more) with a linked-list-based memory management system for dynamic allocation and recycling of pages. The linked list is constituted by the ID index, line and page index, which is stored in the DDR assist message area. To improve efficiency, the ID status LUT is designed to store frequently used information. 2) A pre-allocation strategy is adopted to improve the response time of page requests. When the number of free pages in the buffer cache falls below a certain threshold, the command to fetch from the free page index area of the DDR is activated. 3) Due to most operators shrink, except for constant sort and expanding join, there exists a bandwidth mismatch between the kernel input and output. To

Table 2: The Accuracy of the Performance Model

Kernel	No. of Records	Real(ms)	Estimated(ms)
Filter	100k	2.66	2.5(-6.4%)
	1M	24.75	25(+1%)
	10M	245.83	250(+2%)
Join	10k	2.324	2.3(-1%)
	100k	28.048	25(-10%)
	1M	325.77	312(-4%)

ensure fast page recycling, a queued page recycling mechanism is adopted, with the queue recording the next address to be stored or loaded, and recycled pages being written to the DDR when the number is higher than a certain threshold, reducing the number of DDR accesses and improving efficiency of reading and writing. 4) Additionally, FPGA development boards usually have multiple DRAM chips, which naturally support multi-channel concurrent read/write. As multiple banks can open or precharge a row concurrently, a bank-awareness page allocation measure is taken to improve data concurrency bandwidth and reduce page walk latency.

5 RACE SOFTWARE SYSTEM

After describing the KPU architecture and extracting the supported operations, the question arises how to develop it in a disciplined and efficient way, to ease the verification and correctness of time-consuming and error-prone HDL constructions. RACE is designed to allow easy migration and adoption of current open-source DBMS, taking SparkSQL and PostgreSQL as an example shown in Fig. 9.

KPU software emulator enables a function simulator and performance model, which is created for evaluating each point in the design space and can predict the runtime with more than 95% accuracy, the partial results are shown in Table 2. The simulator also helps to find the correct mix of functional units, memories and forwarding paths, and to validate the behavior of KISA.

To address the high synchronization overhead introduced by CPU-to-device data copying and the high latency of on-board PCIe bus, the memory space is divided into separate parts for input/output. Double-buffering is used to overlap transfers with computation and arrange input in a constant address space, allowing the host to easily partition data using KISA instructions and making hardware memory partition management unnecessary. Additionally, RACE only transfers the columns mentioned in the queries to the KPU, eliminating unneeded I/O.

6 EVALUATION

6.1 Experimental Setup

Baselines. All evaluations were conducted on CPU (12th Gen Intel Core i5-12400) with 16GB RAM and 500GB Samsung SSD, PG-Strom[6] on NVIDIA P100 GPU with Apache Arrow. We implemented the KPU in Verilog and built it on the Xilinx Alveo U200 FPGA card (XCU200), which worked at 270MHz. The sweet-spot of ADO's size is 64 by carefully measuring multiple burst lengths to satisfy the memory bandwidth and computation capacity of KPU cores. Observed from the KISA expression, 16 private ADOs are enough for each table core, and the instruction cache size of each core is 1 KB. According to the statistical results of the related column in TPC-H, the number of shadow cores is set as 4.

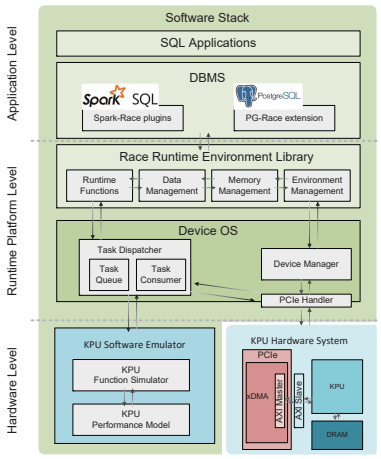

Figure 9: The overview of RACE software system.

Benchmark. TPC-H data are generated with a scale factor of 100. To transfer all 22 queries of TPC-H into KISA, as the compiler is not supported currently, function-mapped methods are used as an alternative. The data width is 64-bit to cover the actual data range. The decimals are represented as 64-bit integers without dot, which is allowed by TPC-H rules and is faster than the variable-length numerical strings methods, more accurate to avoid the rounding errors than double. String manipulations are costly, although there are some regular expression match work, only simple prefix search (WHERE p_name like '*%green%*'.) is supported, leaving the rest on the host.

6.2 Overall Results

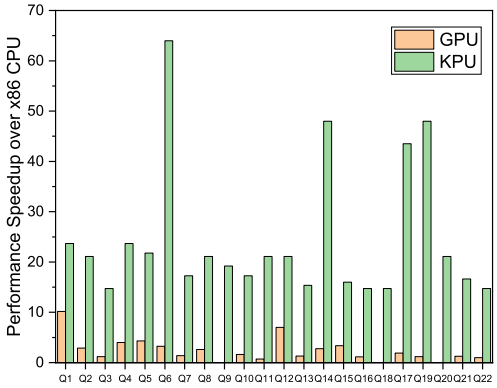

Figure 10: KPU execution time results of TPC-H.

Hardware resource utilization is shown in Table 3. Due to the limitation of resources on Alveo U200 and the TPC-H schema, a 16-cores design was implemented. It should be noted that more cores can be added if more resources allow, leading to better performance. The performance of all TPC-H 22 queries are shown in Fig. 10, 24.5x and 8.75x speedup over the CPU and GPU respectively on average.

To verify the compability of RACE system, we also test the TPC-DS queries for SparkSQL on 10GB local mode, as shown in Fig. 11. Because there are 99 queries in total and it's too difficult to do compilation, TPC-DS is only emulated at current, which shows a 15.62x speedup on average.

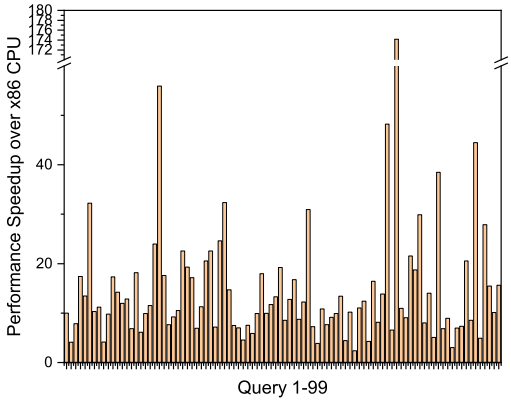

Figure 11: Results of TPC-DS.

Table 3: Hardware Resource Usage

Hierarchy	LUT(%)	FF(%)	BRAM(%)
JOIN	27.44	0.88	-
SORT	24.33	32.52	< 1
FILTER	< 0.01	0.4	-
AGGREGATION	4.7	1.1	-
ALU	< 0.01	< 0.01	-
DMS	8.73	4.51	26.71
ADOF	4.16	3.52	-
RF	5.12	2.88	-
xDMA	2.21	1.29	1.71
Inst-Cache	-	-	10.72
others	4.27	3.08	6.91
Total	80.96%	50.18%	46.05%
Available	1182240	2364480	25.8MB

7 CONCLUSION

In this paper, we proposed an accelerator-centric architecture, KPU and implemented it on the analytic query processing. To deal with the table-mode workload, each table core is split into a key core, which handles the key column, and a shadow core, which mimics the behavior of the key core. The self-designed DMS is used to alleviate the bandwidth bottleneck to 90% utilization ratio. Our 16-core KPU-SQL outperforms the CPU and GPU by 24.5x and 8.75x on average respectively.

ACKNOWLEDGMENTS

This paper is supported in part by National Natural Science Foundation of China (NSFC) under grant No. 62002340, 61872336 and 62090020, in part by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDB44030100, and in part by Youth Innovation Promotion Association CAS No. Y201923. The corresponding author is Guihai Yan and Xiaowei Li.

REFERENCES

[1] Cagri Balkesen, Nitin Kunal, Georgios Giannikis, Pit Fender, Seema Sundara, Felix Schmidt, Jarod Wen, Sandeep Agrawal, Arun Raghavan, Venkatanathan Varadarajan, Anand Viswanathan, Balakrishnan Chandrasekaran, Sam Idicula, Nipun Agarwal, and Eric Sedlar. 2018. RAPID: In-Memory Analytical Query Processing Engine with Extreme Performance per Watt. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1407–1419. https://doi.org/10.1145/3183713.3190655

[2] Peter Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer International Publishing, Cham, 61–76.

[3] Jared Casper and Kunle Olukotun. 2014. Hardware Acceleration of Database Operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '14)*. Association for Computing Machinery, New York, NY, USA, 151–160. https://doi.org/10.1145/2554688.2554787

[4] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1206–1220. https://doi.org/10.14778/3389133.3389138

[5] Zubeyr F Eryilmaz, Aarati Kakaraparthy, Jignesh M Patel, Rathijit Sen, and Kwanghyun Park. 2021. FPGA for aggregate processing: The good, the bad, and the ugly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1044–1055.

[6] HeteroDB 2018. *PG-Strom*. Retrieved April 1, 2023 from https://github.com/heterodb/pg-strom

[7] Wenyan Lu, Yan Chen, Jingya Wu, Yu Zhang, Xiaowei Li, and Guihai Yan. 2022. DOE: Database Offloading Engine for Accelerating SQL Processing. In *2022 IEEE 38th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 129–134.

[8] Philippos Papaphilippou, Holger Pirk, and Wayne Luk. 2019. Accelerating the merge phase of sort-merge join. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 100–105.

[9] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 142–153. https://doi.org/10.1145/2485922.2485935

[10] Makoto Saitoh, Elsayed A Elsayed, Thiem Van Chu, Susumu Mashimo, and Kenji Kise. 2018. A high-performance and cost-effective hardware merge sorter without feedback datapath. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 197–204.

[11] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. 2020. Bonsai: High-performance adaptive merge tree sorting. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 282–294.

[12] Utku Sirin and Anastasia Ailamaki. 2020. Micro-Architectural Analysis of OLAP: Limitations and Opportunities. *Proc. VLDB Endow.* 13, 6 (feb 2020), 840–853. https://doi.org/10.14778/3380750.3380755

[13] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 255–268. https://doi.org/10.1145/2541940.2541961

[14] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Kim, Sungjin Lee, and Arvind Arvind. 2020. Aquoman: An analytic-query offloading machine. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 386–399.