DOE: Database Offloading Engine for Accelerating SQL Processing

Wenyan Lu[1], Yan Chen[2], Jingya Wu[1,3], Yu Zhang[2], Xiaowei Li[1], Guihai Yan[1]
[1]*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences*
[2] *YUSUR Technology Co., Ltd.*
[3]*University of Chinese Academy of Sciences*
Beijing, China
[1,3](luwenyan, wujingya, lxw, yan)@ict.ac.cn, [2](yanchen, zy)@yusur.tech

Abstract—**The CPU-Accelerator heterogeneous systems have demonstrated performance and efficiency benefits on DBMSs. However, it is non-trivial to customize efficient domain-specific accelerators. Even if high-performance accelerators are available for DBMS, it is challenge to integrate the software with accelerator non-intrusively. To address these problems, we proposed a hardware-software co-designed system, DOE, which contains hardware accelerator architecture - Conflux for effective SQL operation offloading, and a software programming platform - DP2 for application integration non-intrusively. The experiment results show that DOE achieves more than 100x and 10x performance improvement compared with PostgreSQL and MonetDB respectively.**

Index Terms—**DOE, DP2, Database, Heterogeneous system**

I. Introduction

To meet stringent performance requirement and power budget of data processing systems, it is not uncommon to offload some computations from host CPU to a dedicated processing engine such as GPU or FPGA [12], [13]. The computing tandem of CPU-Accelerator platform, as a typical heterogeneous system, is believed to be a promising paradigm to unlock the advantage of domain-specific computing.

To reap the potential of "offloading", it does not come for free because of the two challenges. 1) Integrating an accelerator into software application requires massive and intrusive developments. Most applications do not have the flexibility to port to heterogeneous computing. So the integration with an accelerator requires massive changes inside the application. 2) Accelerator customization is highly complex. Offloading a job from host CPU to general-purpose designed processors does not yield better performance, but overhead. To tackle this challenge, the key is to hardwire the computing logic according to task features, i.e. customization.

Keeping the two fundamental challenges in mind, we explore the database offloading engine (DOE), a new computing system for general DBMSs. We try to answer the question: Can the CPU-Accelerator paradigm boosts a general database performance, without intrusive modifications to the core logic of the target database? An ideal DOE should be transparent to the target database and applications.

This work is supported by National Natural Science Foundation of China No. 61872336, 62002340, 62090020, 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.

Although it's intuitive to resort to novel hardware for performance, prior development work for database accelerator suffers critical limitations. The first is that the prior only focused on the SQL computing logic, but left vacancy in how to enable data sharing with the host [4], [7]. The second is missing the synergy with the full functional database.

To overcome the above limitations, we take another perspective for DOE development. First, DOE is not a database, but an offloading engine. The accelerator serves as a co-processor to the host CPU system running DBMS, the filter and aggregation queries are forward to the customized co-processor to exploit ultra-high parallelism and low latency, while the other control-intensive and IO-intensive DB routines, such as index building and data duplication, still resides in CPU domain. Second, DOE should be independent to the host DBMS and therefore applicable to speeding up query executions. Hence, DOE should be engaged with a set of simple but comprehensive APIs for offloading specific queries as transparent as possible.

This paper demonstrated that integrating an accelerator without intrusive modification into database is possible. Clearly, only part of operations of SQL are suitable for offloading. The offloading candidates share the common features such as computing intensiveness, fine-grained parallelizability, and involving a large volume of random memory access. By contrast, other offloading-unfriendly operations should be handled by host CPU. By thoroughly profiling the TPC benchmarks, we developed a set of design rules to make the DOE system be an ideal complement with the host database. We built a hardware accelerator architecture to maximize memory bandwidth and fine-grained query executing parallelism, as well as a programming platform for non-intrusive integration between accelerator and database application. DOE, as far as we know, is the first general database accelerating engine to emphasize the synergy between the host database and the accelerator.

II. Background & Preliminary

A. The potential of database offloading

1) CPU-ACC Architecture: As Fig. 1(a) shows, there are three layers in a typical heterogeneous computing architecture.

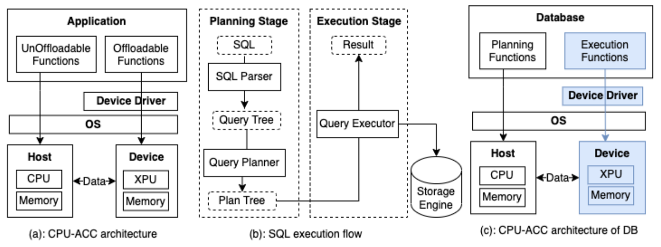

Fig. 1. CPU-DOE architecture

i) Device, that accelerates the specific functions through a customized accelerator (e.g., XPUs). ii) Device Driver, that abstracts the device capabilities for application programming. iii) Application, that uses the device to accelerate the offloadable functions.

2) The SQL execution flow of database system: The query operations are complex but have similar execution flow among different database systems. Fig. 1(b) shows a simplified query processing flow, which includes two stages, *Planning Stage* and *Execution Stage*. Planning stage translates the SQL to query plan. The query plan is a binary tree structure, where the tree nodes are the query operators. Execution stage computes the result by executing plan tree recursively.

3) The opportunity of database offloading: In database applications, the execution stage is composed of high CPU-cost computations, especially in the high-throughput scenario (like OLAP). So, offloading these computations to a heterogeneous computing device has become an effective solution for the performance improvement of data query. As Fig. 1(c) shows, this offloading solution is systematic engineering of database applications, drivers and device development.

B. Offloading-friendly SQL Operations

We make a detailed analysis on the six offloading-friendly SQL operations in data access pattern to guide the DOE design, as shown in Fig.2.

1) Selection & Projection: In the column-oriented storage, the data in the same column are stored in continuous address space of DRAM. Both selection and projection are conducted on specific data columns. So, a huge amount of continuous DRAM access is dominated in selection and projection.

2) Group-by & Aggregation: It is common to perform group-by functions based on the ordered sequence. In this situation, it is intuitive to get the groups by scanning the entire data column. Similar to selection & projection, this kind of operations access DRAM from continuous address space.

3) Hash Join: Hash join is one of the most complex and time-consuming operations. It is composed of two phases: *build* and *probe*. The core operation of both build and probe phases are hash table lookuping, where the random DRAM access dominates both phases, as shown in Fig. 2(3).

4) Sort: Sort operations reorder the records of one table according to the values of a key column in ascend or descend. It frequently involves a large number of random data access and movement. Randomly accessing DRAM for small data is dominated in sort operations.

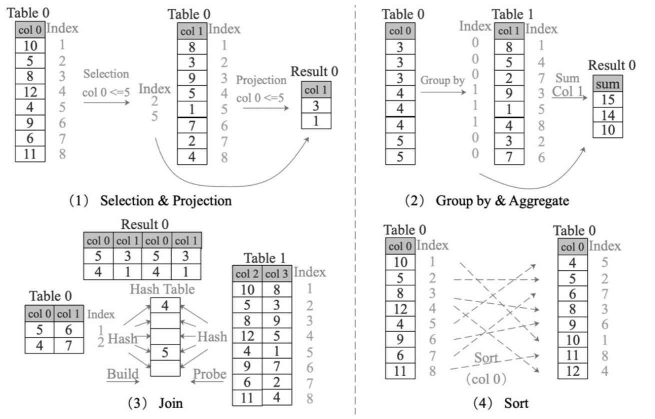

Fig. 2. Basic Operations vs. Data Access Pattern

In summary, the above operations can be classified into two categories: i) scan-like, continuous memory access, including *selection & projection*, and *aggregation & group-by*; ii) join-like, random memory access, including *hash join* and *sort*.

C. The "Tricky" DRAM bandwidth for DOE customization

In DBMS, it is not uncommon that ultra low DRAM bandwidth utilization is achieved by both scan-like and join-like operations. DRAM bandwidth optimization becomes a key breakthrough for performance improvement. Next, we make analyses on DRAM characteristics to motivate the DOE customization.

1) Continuous memory access overwhelms the CPU computing power.: Pipeline processing of memory controller conceals the DRAM access latency. The CPU will undergo bandwidth over-provisioning, because it cannot consume such amount of data volume. This case was observed in many column-based operations such as scan-then-filter on a deep column. In this situation, the bottleneck is not in IO-end but in CPU-end, so the DOE should be able to consume such high data volume.

2) Random memory access causes serious bandwidth utilization decay.: Long stall cycles are inevitable when accessing random address space. Even worse, while only a small portion data of memory access is required in one operation, for example, 4 bytes out of 64 bytes, the bandwidth wasting will get more prominent. This case was observed in join and sort operations which randomly access DRAM for small data. Avoiding these stalls is one of the most important optimization methodology for the effective use of DRAM bandwidth. In this situation, the processing bottleneck is at IO-end.

III. DP2: DOE Programming Platform

We designed a platform - DOE Programming Platform (DP2) for the integration of the DOE device and database applications. The platform aims to enable the programming capability and computation offloading, while the application is responsible for the implementations of query plans and operators.

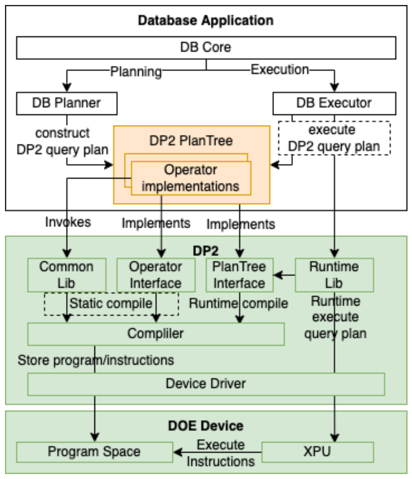

Fig. 3. DP2 Architecture

The boundary between DOE platform and database applications is the query plan and operators. DP2 abstracts and encapsulates the DOE database operators, including the instruction execution and data transmission. It also provides the programmable interfaces for the implementations of offload-friendly operators. Thereby, rewriting the query plan becomes easier in database applications.

Fig.3 shows the high-level architecture of DP2, which is consisted of four core components.

A. Operator Interfaces

We design the operator interface based on the concept from open-source databases. Generally, the operator interfaces are defined by the platform, and implemented by applications.

1) **Scan interface** that should be implemented for loading target data into device memory with filtering conditions.
2) **Join interface** that should be implemented for joining multiple columns based on the join conditions.
3) **Project interface** that should be implemented for choosing target columns based on the SQL context.
4) **GroupBy interface** that should be implemented for grouping the query result by the given column name.

B. PlanTree Interface

The query plan is a binary tree, in which the tree node is the *Operator Interface*. The query plan is defined by platform, and constructed by the application, and eventually executed by the platform in the runtime. The PlanTree node contains 4 key elements, i) left child node, ii) right child node, iii) the operator implementation, iv) the output of operator implementation. Fig. 4 shows a sample of plan tree, which is constructed for offloading SQL “SELECT a.y, b.z FROM A as a, B as b WHERE a.x=b.x GROUP BY a.x” to DOE device.

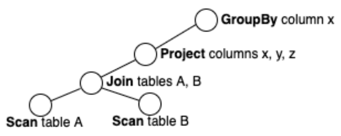

Fig. 4. PlanTree Structure

TABLE I
COMMON FUNCTION LIB

Function	Purpose
columnCreate	Create the column metadata and allocate storage space in the DOE device
columnRelease	Release the storage space in the DOE device
columnWrite	Write column data from host to DOE device storage
columnRead	Read column data from DOE device to host storage
columnJoin	Execute a Join operation for multiple columns
rowFilter	Execute a filtering operation on a column with conditions

C. Common Function Lib

The common function lib provides the column-oriented operation functions based on the device capabilities, and is used for the implementations of operator interface by application developer. Table 1 lists part of functions in this lib. As an example, while implementing a ‘Scan operator’, we use ‘columnWrite’ for copying data from host to device, then use ‘rowFilter’ to filter out the rows.

D. DP2 Compiler

The DP2 compiler is built for compiling the offload-able programs to device instructions based on the DOE instruction set. DP2 compiler supports two access modes. i) Static compilation. The Common Function Lib and Operator Implementations do not depend on runtime context. Instead, they are compiled to device in build-time for better performance. ii) Runtime compilation. The PlanTree varies based on different SQL context and optimization strategies. And it is compiled in runtime.

E. DP2 Runtime Lib

The DP2 runtime lib provides runtime APIs for applications to execute the offloaded query plan. The most important API is ExecutePlan, which takes PlanTree as input and returns the query results.

IV. CONFLUX: A NOVEL MICRO-ARCHITECTURE FOR DOE

We proposed a novel accelerator architecture design, Conflux, with four optimized mechanisms for efficient SQL operation offloading. There are three goals when designing Conflux:

1) **Memory bandwidth efficient.** The Conflux should on one hand avoid random addresses as much as possible when the data are scattered across the memory space, and on the other hand be able to reach line-rate processing when data bursting. We propose two mechanisms, streaming segmentation and random caching, to make efficient use of the memory bandwidth.
2) **High performance.** Performance is the top priority. High performance comes from two mechanisms: i) computing logic customization, which is represented by hardwired operators such as hash join, sort, aggregation, and ii) computing session management, which enables massively fine-grained task parallelism.
3) **Programmable.** Conflux is not statically configured for a given specific query expression, but can be programmed to executed any complex queries. We build a instruction set able to express tasks derived from the original SQL queries.

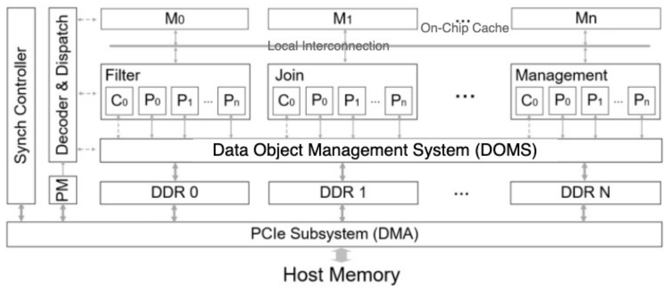

Fig. 5. Conflux Accelerator Architecture

A. Conflux Overview

The overview of Conflux architecture is shown in Fig.5. To achieve high task-level parallelism, Conflux is built as a many-core system. Meanwhile, to continuously access DRAM as much as possible and facilitate full pipelined data processing for each core, streaming data objects (SDO) are used as the basic data structure. The key components are as following:

1) **Data object management system (DOMS)** provides an efficient shared-memory pool for SDO. It has two main functions: i) Efficient DRAM space management, dynamic DRAM pages allocation and re-collection for SDO; ii) Flexible SDO management, which supports the recording, segmenting and merging functions for streaming objects and can transparently serve hundreds of concurrent data accesses for multi-processing cores.
2) **Data synchronization system** lays a high-speed data transmission channel between the CPU-end memory and the ACC-end DOMS based on PCIe DMA mechanism, to efficiently load source data from CPU-end memory to DOMS, or directly fetch back the results from the DOMS to CPU-end memory.
3) **Data processing system** is consist of several groups of processing cores, such as selection, projection, join, and so on. Each group has a configuration core (C-core) and multiple homogeneous processing cores (P-core). P-cores run in stateless mode across different streaming objects. And C-core can distributes tasks to each P-core as balanced as possible, to maximize task-level parallelism.
4) **On-chip cache system** is responsible for random data buffering, of which are multiple groups of SRAMs shared by multi-cores.
5) **Local interconnection system** is NOC-based busses designed for local data exchange between P-Cores.
6) **Control system** consists the instruction decoder and task dispatcher translating DOE ISA to various tasks for processing cores.

B. Memory bandwidth optimization

1) Streaming Segmentation for Continuous Memory Access: For scan-like operations, data transmission is efficient by accessing continuous address space of DRAM. The key optimization is how to improve the capability of data consumption. We need to maximize data-level parallelism to make full use of DRAM bandwidth. The key approach is to pre-segment each SDO, and all segments are processed in parallel. Note that the SDO pre-segment and result merge are trivial, which is transparently implemented in DOMS.

Fig.6(1) shows an example of *selection*. The original input column col0 is divided into two parts before selection and are processed in parallel.

2) Random Caching for Random Memory Access: For join-like operations, data transmission is inefficient when accessing random address space of DRAM. The key optimization is how to reduce the amount of random access to DRAM. The on-chip SRAM has high random access performance. The key optimization for join-like operations is to cache random data on SRAM to ensure the address continuity when accessing DRAM. However, it is challenge to fit a volume-varied SDO into capacity-limited SRAMs. Additionally, We pre-segment the SDO into blocks based on memory pages, and then processing block-by-block.

Fig. 6(2) shows an example of *hash join*. Firstly, Table0 are segmented into two parts. And then hash tables are built in parallel and cached in SRAMs. After that, random hash table lookup is performed efficiently on SRAMs during probe phase.

C. Execution Optimization for Performance

1) Stateless Execution Simplifying Parallel Tasks Scheduling: After the streaming segmentation, tasks can be concurrently conducted on multiple P-Cores. However, the number of logical tasks and the number of free physical cores often mismatches, which makes the task to core mapping complicate. To simplify task to core management, we adopt a stateless execution method, and all processing cores work in data-driven style.

Fig. 6(3) shows an example of two tasks (Task A and Task B). After segmentation, all sub-processing of TaskA has been mapped to processing units (*Sel(x, x)*). Due to no data dependency between P-Cores, parts of Task B are allocated to free P-Cores. By doing so, new tasks can be immediately allocated for processing on free cores, such as (new task ->*Sel(2,3)*), which greatly reduces the time of the core stall.

2) In-place Exchange Enabling Pipeline Execution: Minimizing the volume of data transmission between P-Cores and DRAM is critical for bandwidth optimization. A SQL query plan often contains multiple processing phases, where contain lots of *producer ->consumer* relationships between consecutive phases. Therefore, a huge amount of data transmission can be saved by in-place data exchanging between P-Cores.

Fig.6(4) shows an example of "*selection ->projection ->join*" task processing. The result of *Sel0* is directly passed to *Proj1* through the local bus, and the result of *Proj1* is used as the input of *Join0* immediately.

D. Conflux Instruction Set Architecture

Flexible and efficient ISA helps to fully stimulate the acceleration engines. We design a basic set of instruction expression, and parts list in Table II. The instruction set architecture (ISA) consists of basic instructions (e.g., *Selection, Projection, Groupby*, etc), and auxiliary instructions (e.g., *StreamSeg, StreamMerge*, etc).

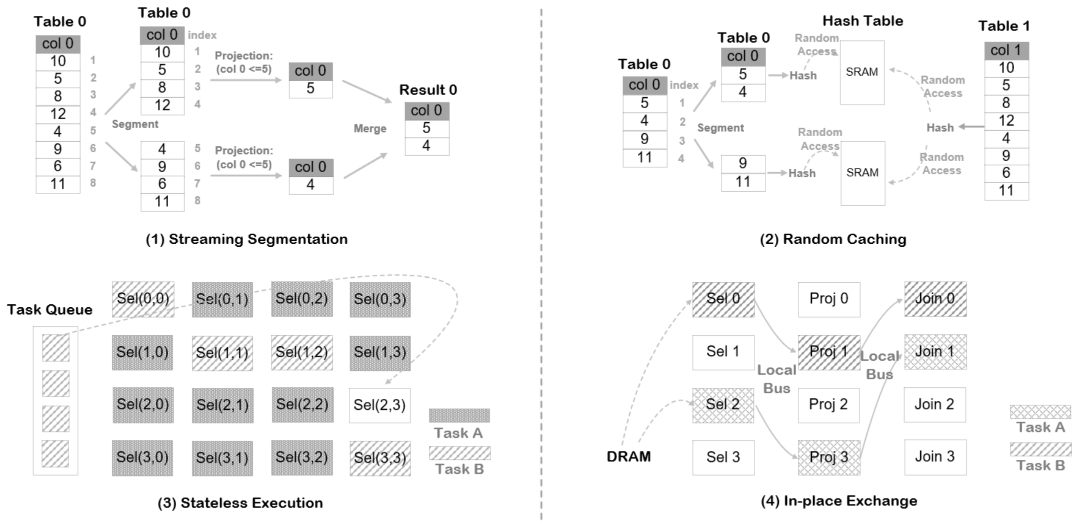

Fig. 6. Key Optimizations

TABLE II
INSTRUCTION LIST

Class	Instruction	Type	Comments
Cacheless	Selection	functional	-
Cacheless	Projection	functional	-
Cacheless	Groupby	functional	-
Cacheless	Aggregate	functional	-
Cacheless	IndexMerge	auxiliary	Merge multi-condition results of selection
Cacheless	DataMerge	auxiliary	Merge segmented results
Cacheless	StreamSeg	auxiliary	Segment streaming into sub-streamings
Cacheless	StreamMerge	auxiliary	Merge multi-streaming
Cacheleness	Join	functional	-
Cacheleness	Sort	functional	-
Cacheleness	TreeIndex	functional	Index-based selection
Cacheleness	Projection-random	functional	Random data

V. Experimental Evaluation

In this section, we evaluate the proposed DOE system in terms of DRAM bandwidth utilization, performance, and resource consumption.

A. Experimental Methodology

1) Baselines: All evaluations were conducted on a Dell Precision 7920 tower server featuring two intel Xeon Gold 5218 CPU @2.30GHz, with 64GB of memory (four 64-bit 16GB DDR4-3200). Four DRAMs have a bandwidth of 100GB/s. We selected two representative RDBMSs (PostgreSQL and MonetDB) as our baselines.

PostgreSQL. a widely used open-source RDBMS PostgreSQL 11.7. We use the function"pg_prewarm" to prefetch data tables into DRAM, to ensure that all queries are executed on in-memory data.

MonetDB. a column-oriented DBMS MonetDB 11.31. We confirmed that MonetDB did not read data from SSDs during query executions, that is, all data were stored in the DRAMs.

DOE. We integrate the DOE to PostgreSQL system through the proposed DP2 platform. And, we implemented the Conflux accelerator in Verilog and build it on the Intel Arria 10 FPGA (10AX115N2F45E1SG), And, we customized the PCB system board with 16GB DRAM (two 64-bit 8GB DDR4-2400), which can deliver 37.5GB/s of bandwidth.

2) Benchmark: We used the TPC-H dataset as the test data. To make all tables reside in DRAM, we set the scale factor as 1 for most of evaluations. We used the 22 test queries as the benchmark.

B. Configuration Exploring

Otherwise specified, the P-Core configurations, i.e., the numbers of each type of P-Core, are detailed in Table III.

An implementation can be either computation-bounded or memory-bounded. In DOE system, DRAM bandwidth utilization is the key optimization object. Firstly, considering communication constraint, we evaluate each type of P-Core isolately to bound the maximum number. Then, we carry out a complete design space exploration considering multiple P-Cores under computation resource (ALMs and operating frequency) constraints. Out of 508 viabale instances, we selected the configuration with highest performance configuration detailed in Table III.

TABLE III
KERNEL CONFIGURATION

Selection	Projection	Groupby	Projection-random
32	16	32	8
Join	**Sort**	**Aggregate**	**Tree-Index**
8	6	32	6

C. Experimental Results

1) Resource consumption in FPGA: Table IV details the resource consumption of each P-Core, DOMS, data synchronization system, on-chip cache, control system, and other basic components (Others). The "Others" entry in Table IV includes, PCIe, DMA, and DRAM controllers. Note that some auxiliary instructions are not list in the table, which are implemented in DOMS. The whole DOE system consumes 328089 ALMs, and 34004088 BRAM bits, which occupies 77% ALMs and 61% BRAM of whole FPGA on-chip resources respectively.

2) DRAM bandwidth Utilization: The DRAM bandwidth utilization is a key metric for evaluating the database system, which is the key optimization in this paper. Fig.7 shows the bandwidth utilization of Conflux accelerator equipped

TABLE IV
RESOURCE CONSUMPTION IN FPGA CHIP

Systems	Number of ALMs (%)	Number of BRAM (%)
Selection	21383 (5%)	1%
Projection	8689 (2%)	1%
Groupby	16732 (4%)	2%
Aggregate	33689 (8%)	3%
Join	40960 (10%)	4%
Sort	39321 (9%)	5%
TreeIndex	13100 (3%)	3%
Projection-random	21048 (5%)	2%
DOMS	30758 (7%)	6%
Synch system	7663 (2%)	3%
on-chip cache	1102 (<1%)	18%
control system	942 (<1%)	1%
Others	92702(22%)	12%
Total	328089 (77%)	61%

with PostgreSQL. DOE stably achieves over 60% DRAM bandwidth utilization across most of queries. Two reasons contribute to the superior DRAM bandwidth utilization: a) strong parallelism-harvesting capability, and b) attractive random data caching capability for join-like operations.

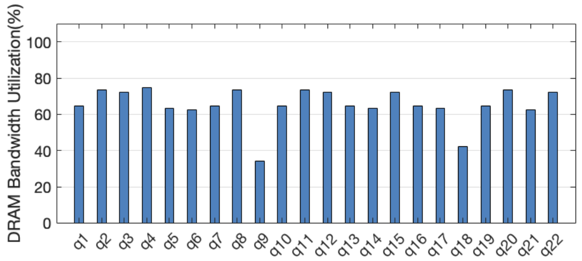

Fig. 7. DRAM Bandwidth Utilization

3) Performance: Fig.8 shows the performance of DOE (PostgreSQL database with Conflux accelerator), PostgreSQL, and MonetDB. DOE provides more than 100x performance speedup over PostgreSQL, and 10x over MonetDB across most queries, except q9 and q18. Proposed four optimization mechanisms contribute to the superior performance of DOE. Full DRAM bandwidth utilization, coupled with in-place data exchange enable massively fine-grained parallelism. The columnar database MonetDB achieve good performance in the OLAP scenario. The join operations across multiple tables decays the task-level parallelism, so DOE held a slender lead than MonetDB in q9 and q18.

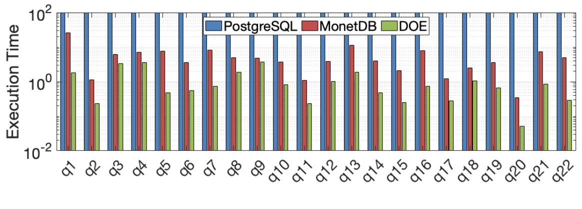

Fig. 8. Performance

VI. Related Work

There are three common directions to explore the database "offloading" approaches: (1) GPU-based methods. GPUs integrated huge amount of cores become one of most popular platforms for application accelerations, such as image processing, scientific calculation, and models. However, using GPU to accelerate some operations in DBMS only get limited performance improvement [1]–[3]. (2) Customized hardwares for basic SQL operations. To accelerate partial basic operations in DBMS, various of subtle micro-architectures have been proposed [4]–[9], [16]. To accelerator the whole operators in DBMS, the ASIC-based Q100 and a domain-specific ISA were proposed [8], [9]. It is a ponderable reference in the design of independent programmable acceleration engines for database. However, these works only focused on basic operators, but ignore the consideration of system integration. (3) Customized systems for a specific database. A whole database system were proposed in [10] to optimize the original data format DBMS. A column-based acceleration system were proposed in [11] to optimized SSD accessing. Besides, some customized systems were proposed to apply on distributed data processing systems [14], [15]. However, these works only targeted at a specific database system.

VII. Conclusions

We designed a whole stack database offloading system, DOE. And, we proposed a flexible DOE programming platform DP2, and designed an efficient Conflux accelerator architecture. Our design provides more than 100x and 10x performance improvement compared with PostgreSQL and MonetDB respectively.

References

[1] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda." 2010, pp. 94–103.

[2] C. Kim, J. Chhugani, N. Satish, E. Sedlar, and P. Dubey, "Fast: fast architecture sensitive tree search on modern cpus and gpus," in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010.

[3] E. A. Sitaridi and K. A. Ross, "Gpu-accelerated string matching for database applications," The VLDB Journal, 2016.

[4] P. Vaidya and J. J. Lee, "A novel multicontext coarse-grained join accelerator for column-oriented databases." in International Conference on Engineering of Reconfigurable Systems Algorithms, 2009.

[5] K. Kara and G. Alonso, "Fast and robust hashing for database operators," in International Conference on Field Programmable Logic Applications, 2016, pp. 1–4.

[6] Z. Zhou, C. Yu, S. Nutanong, Y. Cui, and C. J. Xue, "A hardware-accelerated solution for hierarchical index-based merge-join(extended abstract)," in 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019.

[7] K. Manev, A. Vaishnav, C. Kritikakis, and D. Koch, "Scalable filtering modules for database acceleration on fpgas," in the 10th International Symposium, 2019.

[8] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "The q100 database processing unit," IEEE Micro, vol. 35, no. 3, pp. 1–1, 2015.

[9] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Brezzo, S. Asaad, and D. E. Dillenberger, "Database analytics: A reconfigurable-computing approach," IEEE Micro, vol. 34, no. 1, pp. 19–29, 2014.

[10] B. Sukhwani, H. Min, M. Thoennes, P. Dube, and S. Asaad, "Database analytics acceleration using fpgas," 2012.

[11] S. Watanabe, K. Fujimoto, Y. Saeki, Y. Fujikawa, and H. Yoshino, "Column-oriented database acceleration using fpgas," in 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019.

[12] G. Yan, W. Lu, X. Li, and N. Sun, "Comparative study of the domain-specific processors" SCIENTIA SINICA Informationis 52.2:358-375,2022.

[13] G. Yan, X. Li, and N. Sun, "Study of design methodology of software-defined accelerators" Communications of CCF 14.11:58-63, 2018.

[14] M. Hemmatpour, B. Montrucchio, M. Rebaudengo, and M. Sadoghi,"Analyzing in-memory nosql landscape," IEEE Transactions on Knowledge and Data Engineering, vol. PP, no. 99, pp. 1–1, 2020.

[15] M. Najafi, K. Zhang, M. Sadoghi, and H. A. Jacobsen, "Hardware acceleration landscape for distributed real-time analytics: Virtues and limitations," in ICDCS, 2017.

[16] M. Najafi, M. Sadoghi, and H.-A. Jacobsen, "Flexible query processor on fpgas," Proc. VLDB Endow., vol. 6, no. 12, p. 1310–1313, 2013.