SQL2FPGA: Automated Acceleration of SQL Query Processing on Modern CPU-FPGA Platforms

ALEC LU, School of Engineering Science, Simon Fraser University, Canada

JAHANVI NARENDRA AGRAWAL*, International Institute of Information Technology, India

ZHENMAN FANG, School of Engineering Science, Simon Fraser University, Canada

Today's big data query engines are constantly under pressure to keep up with the rapidly increasing demand for faster processing of more complex workloads. In the past few years, FPGA-based database acceleration efforts have demonstrated promising performance improvement with good energy efficiency. However, few studies target the programming and design automation support to leverage the FPGA accelerator benefits in query processing. Most of them rely on the SQL query plan generated by CPU query engines and manually map the query plan onto the FPGA accelerators, which is tedious and error-prone. Moreover, such CPU-oriented query plans do not consider the utilization of FPGA accelerators and could lose more optimization opportunities.

In this paper, we present SQL2FPGA, an FPGA accelerator-aware compiler to automatically map SQL queries onto the heterogeneous CPU-FPGA platforms. Our SQL2FPGA front-end takes an optimized logical plan of a SQL query from a database query engine and transforms it into a unified operator-level intermediate representation. To generate an optimized FPGA-aware physical plan, SQL2FPGA implements a set of compiler optimization passes to 1) improve operator acceleration coverage by the FPGA, 2) eliminate redundant computation during physical execution, and 3) minimize data transfer overhead between operators on the CPU and FPGA. Furthermore, it also leverages machine learning techniques to predict and identify the optimal platform, either CPU or FPGA, for the physical execution of individual query operations. Finally, SQL2FPGA generates the associated query acceleration code for heterogeneous CPU-FPGA system deployment. Compared to the widely used Apache Spark SQL framework running on the CPU, SQL2FPGA—using two AMD/Xilinx HBM-based Alveo U280 FPGA boards and Ver.2020 AMD/Xilinx FPGA overlay designs—achieves an average performance speedup of 10.1x and 13.9x across all 22 TPC-H benchmark queries in a scale factor of 1GB (SF1) and 30GB (SF30), respectively. While evaluated on AMD/Xilinx Alveo U50 FPGA boards, SQL2FPGA using Ver. 2022 AMD/Xilinx FPGA overlay designs also achieve an average speedup of 9.6x at SF1 scale factor.

CCS Concepts: • **Hardware → Hardware accelerators**; **Hardware-software codesign**; • **Computer systems organization → Reconfigurable computing**; **High-level language architectures**.

Additional Key Words and Phrases: Big Data Analytics, Analytical Query Processing, HBM-based FPGA, High-Level Synthesis, Compilation Framework

---

Authors' addresses: Alec Lu, alec_lu@sfu.ca, School of Engineering Science, Simon Fraser University, 8888 University Dr, Burnaby, BC, Canada, V5A1S6; Jahanvi Narendra Agrawal, jahanvi.agrawal@iiitb.ac.in, International Institute of Information Technology, 26/C, Electronic City Phase 1, Bangalore, Karnataka, India, 560100; Zhenman Fang, zhenman@sfu.ca, School of Engineering Science, Simon Fraser University, 8888 University Dr, Burnaby, BC, Canada, V5A1S6.

---

1 INTRODUCTION

With today's ever-growing scale of databases for big data analytics, query engines are struggling to keep up with the rapidly increasing demand for faster processing of more complex workloads, especially for technology companies—such as Amazon [20] and Alibaba [13]—whose business models are highly driven by customer data. Another trend in current database systems is that more data is cached in memory instead of in storage, allowing one to two orders-of-magnitude higher bandwidth between the data and the processor. Such technology trend brings significant speedup for traditional transaction processing workloads. However, complex analytics operations such as join, aggregation, and expression evaluation are becoming computation-bound in the CPU architecture.

Due to the power and utilization walls [9], there is a significant slowdown in CPU performance scaling in datacenters. High-performance, energy-efficient, and fully customizable FPGA accelerators have attracted increasing attention as strong candidates for accelerating query processing from both industry and academia. Several previous works have achieved decent performance speedup and/or energy efficiency improvements by offloading compute-intensive SQL operations onto FPGAs, where they explore the massive parallelism and highly customized architectures in their FPGA accelerators. For example, with nearly 40% of the data analysis workload performed using SQL queries, Baidu developed a suite of software-defined accelerators for SQL operations called SDA [18] and achieved up to a 55x performance speedup over a 12-core CPU server when evaluated on query #3 of the TPC-DS benchmark suite [30]. More recently, to demonstrate FPGA acceleration in query processing, AMD/Xilinx developed an open-source library of query acceleration overlays [37], which achieved an average performance speedup of 26x on TPC-H queries [31] over PostgreSQL [23] running on the CPU.

However, FPGA acceleration does not come for free and typically requires substantial manual programming efforts during development. For example, even with the pre-designed query acceleration overlays, the query acceleration demo code (Ver.2020) from AMD/Xilinx database library [37] takes more than 500 lines of host code (mainly on the configuration and invocation of the undocumented query acceleration overlays) to manually accelerate each of the 22 TPC-H queries on average. This weak programmability and automation support for FPGAs has been a prevailing barrier for software programmers to develop highly efficient FPGA accelerators and/or effectively integrate them into the existing query processing workflow [3, 10]. As a consequence, for example, AMD/Xilinx no longer provides such manual demo code for TPC-H queries in their newer database library versions (e.g., latest Ver.2022). Unfortunately, few studies target the programming and compilation support to automatically map SQL queries onto the FPGA accelerators, as will be discussed in Section 5.1.

In this paper, we propose SQL2FPGA, an automatic compilation framework that translates and maps SQL queries to the heterogeneous CPU-FPGA acceleration platform. To avoid the overwhelming partial reconfiguration overhead on FPGAs (e.g., our experiments show a ~4.8s partial reconfiguration time on Alveo U280, while the average execution time of our queries is only ~5.2s), in this paper, we leverage the AMD/Xilinx open source query acceleration overlays [37] and automatically map SQL queries onto them. Note that this paper does not optimize the query accelerators, but focuses on the compilation support, including query plan optimizations, to automatically compile SQL queries onto existing well-tuned hardware accelerators on FPGAs, i.e., AMD/Xilinx open-source query acceleration overlays.

To ensure the portability of our SQL2FPGA design, we establish a big data engine-agnostic query plan representation such that our optimizations can be leveraged and ported across different big data engine front-ends. To further improve the performance of the CPU-FPGA hybrid query execution, we implement a set of FPGA-aware compiler optimization passes. First, we improve the query operator acceleration coverage on FPGAs by implementing two optimization passes 1) substituting string-type data with an integer-type row id to overcome the accelerator design constraint of having a 32-bit integer datapath and extend more operation offloading to FPGA accelerator; 2) transforming non-natively supported join operations into accelerator supported join operations. Second, to reduce redundant computation, we implement a compiler optimization to merge repeating operations. Third, to optimize for more efficient data transfers, we propose 1) an accelerator fusion optimization to minimize the expensive data exchange between CPU main memory and FPGA device memory; and 2) a join reordering strategy to minimize the intermediate data transfers between the compute-intensive join operations. To identify the optimal platform, either CPU or FPGA, for the physical execution of individual query operations, we utilize machine learning techniques. We train multiple decision tree based classifiers to discern performance patterns based on diverse configurations, including variations in 1) input tables, 2) input and output data sizes, and 3) operation complexity.

We evaluate our SQL2FPGA on all 22 queries from the widely used TPC-H benchmark suite [31] with the AMD/Xilinx Alveo U280 [38] and Alveo U50 [39] datacenter FPGA boards, using AMD/Xilinx database query acceleration overlays [37] (Ver.2020.1 and Ver.2022.1). Compared to Apache Spark SQL execution on CPU, under SF1 and SF30, SQL2FPGA using 2020.1 overlays achieves average speedups of 10.1x and 13.8x across all 22 TPC-H queries; and under SF1 (note 2022.1 overlays does not support SF30), SQL2FPGA using 2022.1 overlays achieves average speedups of 9.8x across all 22 TPC-H queries. Compared with AMD/Xilinx well-optimized manual host code, on average, SQL2FPGA using 2020.1 overlays (note there is no optimized manual host code for 2022.1) incurs around 7% overhead for SF1 dataset and is nearly 1% faster for SF30 dataset for accelerating the 22 TPC-H benchmark queries.

In summary, our paper makes the following contributions:

1. A general framework called SQL2FPGA that enables automatic compilation of SQL queries to be accelerated on the heterogeneous CPU-FPGA platform. SQL2FPGA is open-sourced at: https://github.com/SFU-HiAccel/SQL2FPGA.
2. A set of hardware-aware compiler optimization passes to further improve the performance of the hybrid CPU-FPGA query acceleration.
3. A set of machine learning based classifiers to optimize the platform selection for the physical execution of query operations.
4. A quantitative evaluation and analysis of the experimental results on all 22 TPC-H benchmark queries on two database overlay versions on AMD/Xilinx U280 and U50 FPGAs.

The rest of this paper is organized as follows. Section 2 gives background on SQL query processing, discusses the opportunities and challenges to accelerate query processing using FPGA. Section 3 presents SQL2FPGA, including its overall compilation flow, unified engine-agnostic query plan representation, overlay-based FPGA accelerator designs, query plan optimizations, and machine learning techniques for choosing execution platform. Section 4 presents and analyzes our experimental results. Section 5 summarizes the related work in accelerating SQL query processing using specialized hardware platforms: FPGA, ASIC, and GPU. Lastly, Section 6 presents our concluding statements and discusses future work.

```
1  SELECT l.orderkey, c.orderdate,
2      c.shippriority,
3      sum(l.extendedprice*(1-l.discount))
4      as revenue
5  FROM orders as o, customer as c,
6      lineitem as l
7  WHERE o.mktsegment = 'MACHINERY' and
8      o.custkey = c.custkey and
9      l.orderkey = c.orderkey and
10     c.orderdate < date '1995-03-07'and
11     l.shipdate > date '1995-03-07'
12 GROUP BY l.orderkey, c.orderdate,
13         c.shippriority
14 ORDER BY revenue, c.orderdate
```
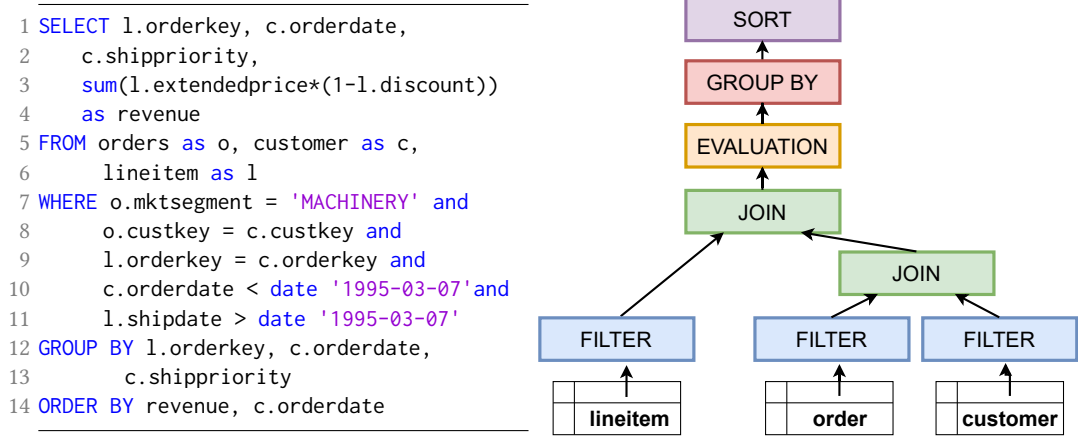
Fig. 1. An example query processing flow using TPC-H query #3: the left listing is the SQL query, the right is a query execution plan generated by Apache Spark SQL.

2 BACKGROUND

2.1 Query Processing

With the ever-increasing scale and workload complexity in today's database management system (DBMS), high-performance query processing engines with efficient optimizations are needed to retrieve and process data from a database, whether stored on disk or in main memory. In this paper, we focus on providing compilation support for accelerating query processing using the heterogeneous CPU-FPGA platform for in-memory database systems.

At a high level, query processing consists of three main stages: 1) high-level query language parsing and translation, 2) query plan optimization, and 3) execution of the generated query plan. For a better illustration, Figure 1 shows an example of how a typical user query is interpreted and executed. The left of Figure 1 shows TPC-H query #3 in SQL commands, and the right presents its corresponding logical execution plan parsed and populated from a query processing engine. The plan is executed in a bottom-up approach, starting from scanning table data (i.e., *lineitem* (l), *orders* (o), and *customer* (c) tables) to retrieve required attributes (e.g., *discount*, *shipdate*, and *orderkey* for *lineitem* (l) table). Then they have processed through a series of relational operations: *filter* operations are for *o.mktsegment* = '*MACHINERY*', *c.orderdate* < *date '1995-03-07'*, and *l.shipdate* > *date '1995-03-07'*; *join* operations reflect *o.custkey* = *c.custkey* and *l.orderkey* = *c.orderkey*; expression *evaluation* operation is for *l.extendedprice * (1 - l.discount)*; *group-by aggregation* operation is for grouping attributes: *l.orderkey*, *c.orderdate*, and *c.shippriority*; and *sort* operation is applied to both attribute *c.orderdate* and aggregation result (*revenue*). For this work, we use Spark SQL, a query processing module from Apache Spark [26], one of the most widely used large-scale big data analytics engines, as our front-end to parse SQL queries and generate the optimized logical plan.

2.2 Potential of FPGA Acceleration for Query Processing

Due to the inherent high parallelism, reconfigurability, and low power consumption characteristics, FPGAs have shown great potential to speed up database systems. Previous research efforts have proposed FPGA accelerator designs for database operators [6–8, 14, 28, 41] to accelerate the entire or part of a query with FPGA [5, 11, 16, 27, 29, 34, 42], and worked on system integration of FPGA accelerators in database systems [19, 25, 32, 33]. Regarding hardware capability,
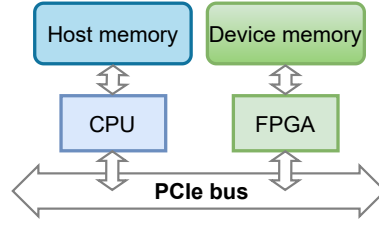
Fig. 2. An overview of heterogeneous CPU-FPGA platform used in SQL2FPGA.

the latest generation of Xilinx Alveo U280 [38] datacenter FPGA board supports HBM2 within the same package, providing close to half TB/s off-chip memory bandwidth, which makes it highly applicable to data-intensive analytical query processing workloads.

Nonetheless, one key factor preventing the wide adoption of FPGA acceleration is the lack of automation support to translate SQL queries to be efficiently accelerated on an FPGA accelerator [10]. In our work, we aim to bridge this gap by providing an automatic compilation framework to accelerate in-memory query processing, specifically for the heterogeneous CPU-FPGA platform as shown in Figure 2, where devices communicate via the PICe interface.

We envision SQL2FPGA to be an FPGA extension plug-in, portable to accelerate different databases in the future. When choosing the FPGA accelerator design, we leverage a set of open-source FPGA accelerator overlay designs from AMD/Xilinx Vitis database library [37]. Even without FPGA reconfiguration, these overlay designs support flexible acceleration for different database operators through runtime parameterization. Although overlay designs have a fixed datapath, Xilinx overlay designs include bypassing logic and SQL2FPGA includes query plan optimizations to extend overlay utilization during query processing as discussed in section 3.5. The dynamic partial reconfiguration approach offers another alternative to support more flexible query operators on the FPGA. However, it comes with reoccurring and overwhelming reconfiguration overheads and is not commonly used nor well supported on datacenter FPGAs.

3 SQL2FPGA SYSTEM DESIGN

In this section, we present the system design of SQL2FPGA, a general framework to enable automatic compilation of SQL queries to be accelerated on the heterogeneous CPU-FPGA acceleration platform. Section 3.1 first gives the compilation flow overview in our framework. Next, Section 3.2 presents details on the vendor-agnostic query plan representation used in our framework. Then, design features of the AMD/Xilinx database accelerator overlay designs are presented in Section 3.3 whereas the CPU operators used in SQL2FPGA are described in Section 3.4. In Section 3.5, we describe the query optimizer of SQL2FPGA, which mainly consists of compiler optimizations to further improve the processing performance of the physical query execution. Last but not least, we present the machine learning based allocator to optimize the platform selection for the physical execution of query operations in Section 3.6

3.1 Compilation Overview

To illustrate the compilation flow of SQL2FPGA, Figure 3 shows our compiler in a three-stage structure.

To ease the development effort, SQL2FPGA is designed to leverage the front-end from different query processing engines. In this work, we leverage the front-end from Spark SQL [2] to first parse user-provided SQL queries; then construct an abstract syntax tree (AST) of database logical operators and expressions; and lastly, to generate an optimized query plan applied with a series of generic static logical plan optimizations such as predicate (e.g., filter and pre-aggregation) pushdown and expression simplification.
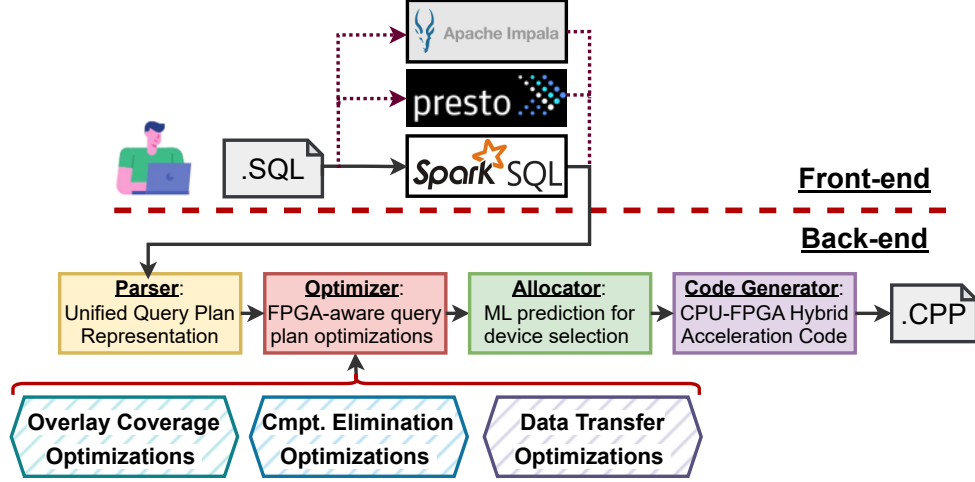
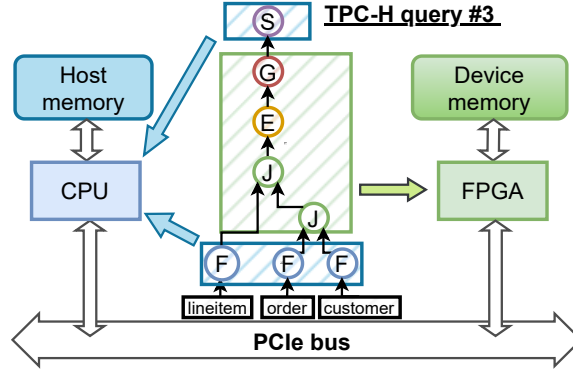Fig. 3. Overview of SQL2FPGA compilation flow.

Fig. 4. Physical execution plan of TPC-H query #3 on heterogeneous CPU-FPGA platform.

On the back-end side, for better design portability and reusability for different query processing engines, our framework first parses the optimized query plan from Spark SQL into a unified query plan representation called *SQL2FPGA-QPlan* to record all necessary information and relations between different logic operators. It is vendor-agnostic and native to SQL2FPGA. Next, our optimizer examines the parsed query plan. It applies a series of compiler optimizations to 1) improve operator acceleration coverage by the FPGA, 2) eliminate redundant computation during physical execution, and 3) minimize data transfer overhead between operators on the CPU and FPGA. Lastly, the code generator outputs the final acceleration code in C++ with AMD/Xilinx OpenCL APIs to interface with the accelerator overlay designs.

Figure 4 presents the corresponding physical execution plan for TPC-H query #3 (depicted in Figure 1) on our heterogeneous CPU-FPGA platform. Data from each table first passes through a filter operation on the CPU before handing over the computation to the FPGA, where *orders* and *customer* tables are first joined, then subsequently joined with table *lineitem*. Next, the results from the second join operation pass through an expression evaluation followed by

a group-by aggregation operation. Finally, aggregation output is transferred back to the CPU main memory and then sorted on the host CPU.

Regarding the device scheduling of operator execution, we explore both the ideal assignment and machine learning guided assignment, and compare their performance. The ideal performance is achieved mostly based on exhaustive empirical experiments. We first determine the operators that can be functionally offloaded to the FPGA overlays. Then, we individually evaluate their CPU and FPGA performance and the required data transfer overhead to decide the platform for the final physical execution. To eliminate the manual effort used in the first approach, the second approach utilizes machine learning techniques to identify the optimal platform, either CPU or FPGA, for the physical execution of each query operator. Specifically, it uses multiple classifiers to discern performance patterns based on diverse configurations, including variations in 1) input tables, 2) input and output data sizes, and 3) operation complexity.

3.2 Vendor-Agnostic Query Plan Representation

Most of today's query processing engines or database systems, such as Spark SQL [2], PostgreSQL [23], and MonetDB [15], store and represent their query plans in a tree structure. However, their naming conventions and implementation details are all different. Furthermore, no unified query plan representation is compatible with all these systems. In general, a query plan can be represented as a tree structure, where each node represents a logical operation while the edges indicate the data flow of the query. In SQL2FPGA, we define a vendor-agnostic query plan representation called *SQL2FPGA-QPlan* to facilitate the compiler optimizations from our query optimizer. *SQL2FPGA-QPlan* is a tree-based data structure where each query plan node contains the following information:

1. A list of operation expressions (e.g., projection expression, aggregation expressions, and keys and payload used for join and group-by operations)
2. A list of input relation tables (column and type of data)
3. A list of output relation tables (column and type of data)
4. A list of children operator nodes
5. A list of parent operator nodes

In this work, we write a parser for interpreting and converting the optimized query plan from Spark SQL since it is one of the most commonly used big data processing engines. When porting SQL2FPGA to work with other query processing engines or database systems, the only required design change is the query plan parser to generate our *SQL2FPGA-QPlan*. We plan to add query plan parsers for other database engines (e.g., PostgreSQL and MonetDB) in future work.

3.3 FPGA Accelerator Overlay Design

To support flexible acceleration for different database operators without FPGA reconfiguration, accelerator overlay design meets the requirement to be dynamically and efficiently configured through runtime parameterization and is commonly used and well-supported on datacenter FPGAs. In this work, we use two sets of open-source streaming-based FPGA accelerator overlay designs from AMD/Xilinx Vitis database library [37]: Ver.2020 and Ver.2022. Moreover, in Ver.2020, AMD/Xilinx also provides manually optimized implementations of acceleration designs for TPC-H queries, which we use as evaluation benchmarks; note in post-2020 versions, AMD/Xilinx no longer provides such manually optimized implementations. In post-2022 versions, the overlay designs remain the same as Ver.2022.

(a) *gqeJoin* FPGA accelerator overlay design

(b) *gqeAggr* FPGA accelerator overlay design

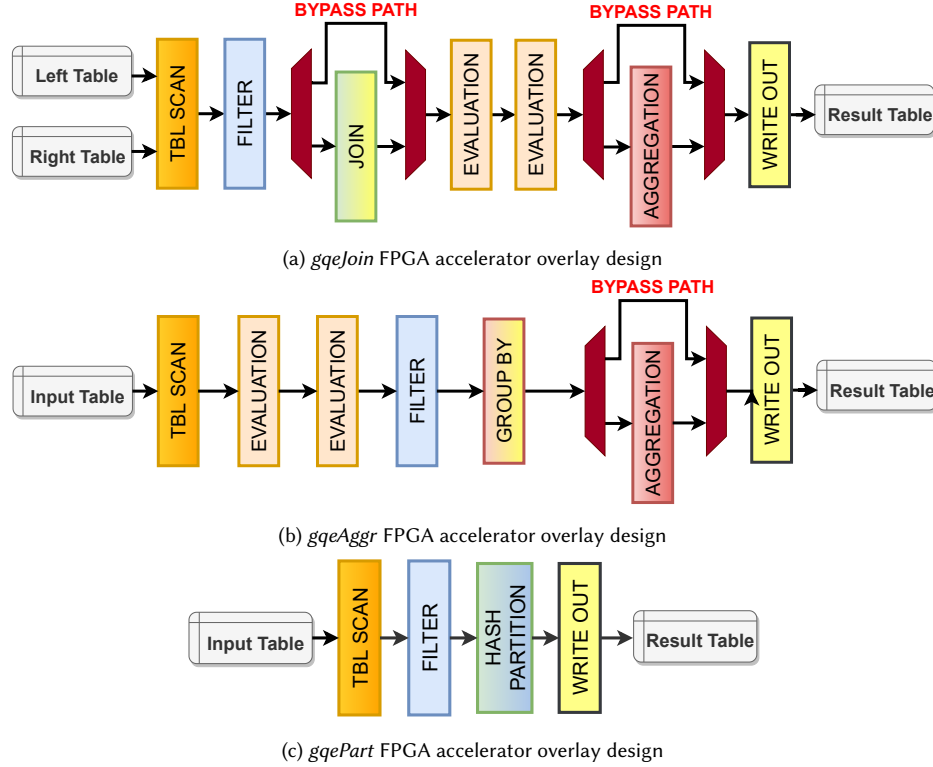(c) *gqePart* FPGA accelerator overlay design

Fig. 5. Overview of overlay designs from Xilinx Vitis database library Ver.2020: top (*gqeJoin*) mainly focuses on different join operations, middle (*gqeAggr*) mainly targets group by aggregation operations, and bottom (*gqePart*) performs hash partition.

3.3.1 ***AMD/Xilinx Database Accelerator Overlay Design Ver.2020***. As shown in Figure 5, the Vitis database library (Ver.2020) consists of two core overlay designs: one focuses on join operations called *gqeJoin* and the other design called *gqeAggr* targets group-by aggregation operations. Each overlay design also contains a separate small prefix accelerator module called *gqePart*, used to perform hash partition when scaling the input table size. Regarding the overall overlay architecture designs, both *gqeJoin* and *gqeAggr* are composed of several accelerator modules constructed in a dataflow fashion. To dynamically gather and redirect column data during the execution of the overlay design, shuffle units are inserted between the adjacent accelerator modules to allow table attributes to switch channel lanes as they flow through the overlay design. All accelerator modules, including their associated shuffle units, are parameterized, meaning they can be configured through a set of user-provided configuration registers.

Figure 6 shows an example of the configuration bit file for the *gqeJoin* overlay design. The configuration file contains a total of nine 512-bit registers. The first 512-bit register records configuration for accelerator modules such as *table scan*, *join*, *aggregation*, *write out*, and shuffle units. The second and third registers record configuration bit for the two *aggregation* module. Lastly, the remaining configuration registers are used for the *filter* operators. For the interested audience, please refer to [37] for details on the overlay design configuration register file format. Regarding each of the accelerator module designs, we summarize their main features as below:

1. *Join*: a hash-based multi-join operator supporting join operations: inner, anti, and semi-join, on up to two keys.

[511-192]	[191-184]	[183-120]	[119-56]		[3-5]	[2]	[1]	[0]
shuffle	output table col id	right table col id	left table col id		join type	dual key	aggr on	join on
evaluation-0 configuration bits (1x512-bit)								
evaluation-1 configuration bits (1x512-bit)								
left table filter configuration bits (3x512-bit)								
right table filter configuration bits (3x512-bit)								

Fig. 6. Configuration register specification of *gqeJoin* overlay design: each register row is 512-bit wide.

2. *Group-by aggregation*: a hash-based operator supporting group-by operation with up to eight unique keys and producing six possible aggregate results: MIN, MAX, SUM, AVERAGE, COUNT, and COUNT-NONZERO.

3. *Filter*: a parallel filter design supporting up to four concurrent boolean-type condition columns.

4. *Evaluation*: a tree-based design with evaluation operation cells in each node, which can be configured to support four kinds of computations for expression evaluations: comparison, boolean algebra, multiplexing, and arithmetic.

5. *Aggregation*: a processing unit that performs calculation of min, max, sum, and count for each input column.

6. *Hash partition*: an operator to distribute a large table into multiple smaller tables based on partition key(s).

7. *Table scan* and *write out*: primitive modules used to facilitate data exchange between device DRAM and FPGA on-chip memories.

We summarize the design constraints imposed by the AMD/Xilinx accelerator overlay designs as follows. First, the overlay designs only support 32-bit integer datapaths, preventing floating-point and variable-length string data types from being accelerated on the overlay designs. For this reason, we conduct floating-point calculations by scaling the floating-point value by a factor of 100, then proportionally scale down and apply type cast for the results after the computation. Please note this is a temporary workaround. For future work, we will explore designing more optimized FPGA accelerators for database operations to resolve this issue. For handling column data of string type, we propose an optimization to extend the acceleration coverage of the overlay design (described in Section 3.5). The second design constraint worth mentioning is that the maximum number of input columns supported by both overlay designs is eight. In contrast, the maximum number of output columns is eight for *gqeJoin* and 16 for *gqeAggr*.

3.3.2 AMD/Xilinx Database Accelerator Overlay Design Ver.2022. As shown in Figure 7, the Vitis database library Ver.2022 consists of two core overlay designs: one focuses on join operations and bloom filter (with partition) operations called *gqeKernel*, and the other *gqeAggr* design that is the same as the one from Ver.2020 library which targets group-by aggregation operations. Similar to Ver.2020 overlay designs, *gqeKernel* is composed of several accelerator modules constructed in a dataflow fashion. Shuffle units are inserted between the adjacent accelerator modules to allow table column data to switch channel lanes as they flow through the overlay. Accelerator modules, including their associated shuffle units, are parameterized, i.e., they can be configured through a set of user-provided configuration registers. However, there are several major design differences in the Ver.2022 overlay designs. First, the datapath of the overlay designs has been extended to 64-bit integer type to support larger scale of data. Second, as the Ver.2022 overlay designs focus on the resource-limited AMD/Xilinx Alveo U50 FPGA [39], the core operations after the filter and before the write out modules of the *gqeJoin* overlay design has been refactored to support hash-join or hash-bloomfilter with hash-partition and retired the bypass paths in the *gqeKernel* overlay design. Third, the *gqeKernel* design takes only up
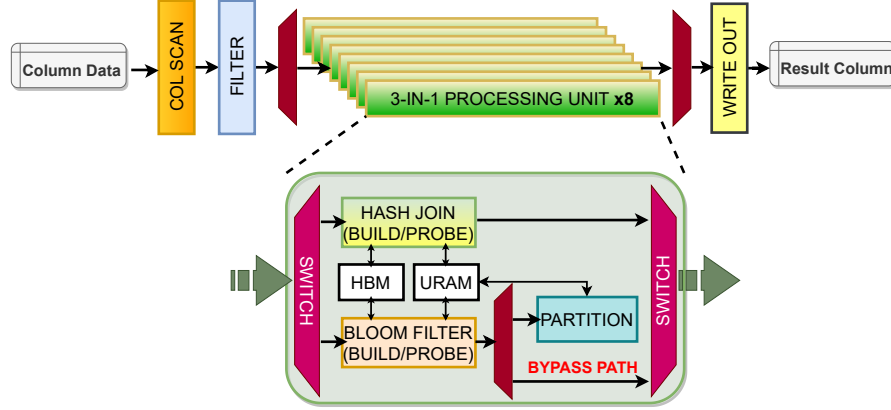
Fig. 7. Overview of *gqeKernel* overlay design from AMD/Xilinx Vitis database library Ver.2022 that replaces the *gqeJoin* overlay design to perform hash-based join and bloom filter (with partition) operation.

to three table columns as input data from each input table, which may consist of 1-2 key column data and 1 payload column. This design choice is constrained by the use of a single 256MB HBM bank for storing all input data, compared to using the higher capacity 4GB DDR4 memory in the Ver.2020 overlay design. With the reduced data movement at a lower number of payload columns, this improves the computation throughput for join operations; however, additional data post-processing is required on the CPU to consolidate the final join results with multiple payload columns. The performance impact is presented and discussed in Section 4.4.

There are two limitations of the Ver.2022 overlays. First, being tailored to the resource-limited Alveo U50 FPGA, the Ver.2022 overlay designs are limited to support low scale factor (e.g., 1GB) datasets. This is because 1) the single 256MB HBM bank for holding all input data and partial hash tables used by the overlay design restricts the support for larger scale factor dataset; and 2) Ver.2022 overlay design do not come with a prefix *gqePart* kernel to performance hash partition on the input tables. Second, unlike Vitis database library Ver.2020, manually optimized implementations of acceleration designs for TPC-H queries using Ver.2022 overlays are no longer provided, which brings significant engineering effort if used without our SQL2FPGA automatic hardware-aware compilation framework.

3.4 CPU C++ Operator Design

In addition to leveraging FPGA overlay designs, we also implement a complete set of CPU C++ query operators, including *filter*, *hash-join*, *group-by aggregation*, *expression evaluation*, and *sort*, based on the C++ operator implementations from PostgreSQL [23]. There are two reasons for this. One is due to the FPGA hardware design constraints summarized in Section 3.3: some operators still have to run on CPU. The other is for verification of the correctness of our framework.

3.5 Compiler Optimizations for Query Plan

As mentioned in Section 3.1, SQL2FPGA optimizer consists of several compiler optimizations targeting different performance aspects in accelerating query processing. First, we propose two optimizations to extend operator acceleration coverage by the overlay design: StringRowIDSubstitution and SpecialJoinTransformation. Next, we eliminate and merge repeated operations in the OperatorPruning optimization pass by traversing the entire query plan. Then, to minimize the data transfer overhead between CPU and FPGA, the FPGAOverlayFusion transformation fuses multiple overlay calls into a single overlay execution. Lastly, to minimize the intermediate data transfer between inner join operations,

Manuscript submitted to ACM

we apply the CascadedInnerJoinReordering optimization. With regard to the generality of our proposed optimizations, strictly speaking, only SpecialJoinTransformation is constrained to the current AMD-Xilinx overlay architecture. All other optimizations are transferable to future accelerator engines. Alos, we will continue to explore and implement more general query plan optimizations through our extended evaluation of TPC-DS [30] and other more complex query benchmarks.

3.5.1　*Acceleration Coverage Extension on FPGA Overlay*.

Opt 1 − StringRowIDSubstitution: due to the limited 32-bit integer datapath support as described in Section 3.3, table columns of variable-length data types such as string cannot be processed using the FPGA overlay design, even though the relation operation does not directly depend on the string-type data. The reason is that Xilinx overlay designs do not support dynamic memory storage for the variable-length attribute data. This design limitation prevents certain operations from being offloaded and accelerated on the FPGA. To overcome this design limitation, in this optimization, we first substitute the string-type attributes with its table row ID for operations that do not require the actual string content. Next, we perform back-substitution to materialize the actual string data by traversing upward of the query plan until it reaches the end of the query plan or the operation requires the string-type data.

Opt 2 − SpecialJoinTransformation: as summarized in section 3.3, the *join* accelerator module supports three types of join (inner, anti, and semi) on up to two keys (using "=" condition, e.g., $left\_table.key1 = right\_table.key1$). Outer join operation is not natively supported. Nevertheless, through relational algebra, outer join is equivalent to the summation of separately conducting an inner join and an anti join. Moreover, by carefully going through the HLS design of the *gqeJoin* overlay design, we have found a specially supported join condition for semi and anti joins when joining on two keys such that $left\_table.key1 = right\_table.key1 \ \&\& \ left\_table.key2 \ != \ right\_table.key2$. In this optimization, we traverse the query plans to incorporate these transformations.

3.5.2　*Redundant Computing Elimination in Query Plan*.

Opt 3 − OperatorPrunning: this optimization is inspired by a series of observations from examining the optimized logical query plan from Spark SQL. First, the same operation expression using the same input columns and producing the output columns are repeatedly called at multiple locations of the query plan. Sometimes, this could also be two query plan nodes sharing the same operation expression while the list of input columns of one plan node is a subset of that from the other plan node. Either way, we could prevent redundant computation by merging the two operations. Second, the optimized logical query plan from Spark SQL sometimes contains *projection* operation performing only attribute aliasing, which can be avoided during actual physical execution.

3.5.3　*Minimizing Data Transfer between Operators*.

Opt 4 − FPGAOveralyFusion: in reducing the expensive PCIe data transfer to exchange input/output data between different FPGA overlay tasks, the objective of this optimization is to minimize the number of overlay tasks invoked throughout the query plan by fusing them based on the pipeline sequence of the overlay designs. The reduced number of overlay tasks issued also lowers the Xilinx API invocation overhead.

Opt 5 − CascadedInnerJoinReordering: join operations are compute-intensive tasks whose execution time could dramatically increase depending on the scale and statistics such as distribution and cardinality of the input data. While it requires extensive effort and in-depth analysis to optimize join operations, based on the details of these input data characteristics, we implement a query plan optimization that could potentially lower the intermediate data transfer

between inner join operations and, thus, improves processing performance. This optimization uses a reordering strategy to compute inner join operations on smaller input table sizes first.

3.6 Machine Learning to Predict Execution Platform

Table 1. Operator classifier training and testing configurations

Operator classifiers	Training features	Classification label	Training set sizes	Testing set sizes
inner join	'num_key', 'key_name', 'left_tbl_num_row',			
semi join	'right_tbl_num_row', 'left_tbl_num_col',		1245	535
anti join	'right_tbl_num_col', 'output_tbl_num_col'	'FPGA' or		
groupby	'num_groupby_key', 'num_evaluation_ops',	'CPU'		
aggregate	'num_aggregation_ops', 'tbl_name',		1100	470
evaluation	'input_tbl_num_row','input_tbl_num_col'			

In addition to optimizing the execution plan of the query, we also need to decide the most suitable platform (CPU or FPGA) for the physical execution of the query operations. Big data query engines like SparkSQL [2] utilize analytical cost models for their CPU-based query operators. However, they are not directly applicable for SQL2FPGA, since we include FPGA-based operators and different implementations of the C++ CPU operators. Also, as suggested in [1], analytical cost models, which are commonly used by optimizers to compare candidate plan costs, are poor predictors of the execution latency.

In turn, to determine the better performing platform between CPU and FPGA for executing query operators, SQL2FPGA employs learning-based modeling and prediction techniques. Specifically, to guide this decision, SQL2FPGA uses the decision tree classifiers [24], which are simple to understand, interpret, and implement using Python Scikit-learn library [22]. Underneath, its core algorithm recursively splits the input training dataset based on certain criteria (e.g., Gini impurity, information gain, or mean squared error) until it reaches a stopping condition. The stopping condition is tuned to prevent model overfitting consists of a mixed factors such as the maximum tree depth and the minimum number of samples leaf nodes.

For the different compute-based operations summarized and listed in Section 3.3, Table 1 shows the classifiers we train based on 1) input/output data size (i.e., 'tbl_num_row' and 'tbl_num_col'), 2) input table specific information (i.e., 'key_name' and 'tbl_name'), and 3) complexity of operation (i.e., 'num_key', 'num_groupby_key', 'num_evaluation_ops', and 'num_aggregation_ops'). In return, the classifiers return a single label identifying the predicted optimal platform (i.e., 'FPGA' or 'CPU'). To generate the training and testing dataset for each operator, we vary the training features for each operator across a wide range, and split 70:30 on the dataset.

Table 2. Evaluated DecisionTreeClassifier model parameters

DecisionTreeClassifier model parameters	Options
criterion	'gini' and 'entropy'
min_weight_fraction_leaf	0.1, 0.2, ... 0.8, 0.9
min_samples_split	1, 2, ... 19, 20
max_depth	1, 2, ... 29, 30
max_features	'sqrt' and 'log2'

For tuning the decision tree classifier, Table 2 lists the model parameters and options we evaluate to train the classifier with Python Scikit-learn library [22]. *criterion* indicates the function to measure the quality of a split; we use criteria 'gini' for the Gini impurity and "entropy" for the information gain. *min_weight_fraction_leaf* presents the minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node; we vary this from 0.1 to 0.9. *min_samples_split* shows the minimum number of samples required to split an internal node; we vary this from 1 to 20. *max_depth* describes the maximum depth of the tree; we vary this from 1 to 30. *max_features* presents the number of features to consider when looking for the best split; we experiment with 'sqrt()' and 'log2()'.

4 RESULTS AND ANALYSIS

First, we present the experimental setup for our evaluations. Second, we evaluate and analyze the overall performance of SQL2FPGA under different design configurations and compare performance results to Spark SQL. Third, we investigate the performance impact of our optimization passes and discuss how they can efficiently accelerate query processing on the hybrid CPU-FPGA platform. Fourth, we summarize the performance difference between the Ver.2020 and Ver.2022 overlay designs. Lastly, we present the performance of the machine learning guided device selection, as well as our preliminary experimental results evaluating on selective TPC-DS benchmark queries.

4.1 Experimental Setup

Benchmark queries. We evaluate SQL2FPGA on all 22 queries in TPC-H Version 2 [31], the de facto industry standard for online analytical processing (OLAP) performance benchmarking. The TPC-H dataset sizes we populate are in scale factors of 1GB (SF1) and 30GB (SF30), demonstrating that SQL2FPGA supports different dataset scales while consistently achieving performance improvements. Note that only SF1 is evaluated using the Ver.2022 overlay designs due to the limited HBM memory space for storing the input data.

Hardware platform and software tool. For our system evaluation, we deploy SQL2FPGA on a CPU-FPGA heterogeneous platform with the 14nm 12-core (24-thread) Intel Xeon Silver 4214 CPU and 128GB DRAM as the host platform, while the FPGA accelerator overlay designs Ver.2020 are deployed on two of the 16nm AMD/Xilinx Alveo U280 (with 32 HBM2 banks and Gen3x16 PCIe interface) [38] datacenter FPGA boards as described in Section 2.2; and accelerator overlay designs Ver.2022 are deployed on two of the 16nm AMD/Xilinx Alveo U50 datacenter FPGA boards.

In terms of optimizations, Spark SQL incorporates various CPU optimizations, including code generation, predicate pushdown, projection pruning, shuffle aggregation, columnar storage utilization, vectorization, and runtime code generation with Whole-Stage Code Generation (WSCG), to enhance query processing performance [2]. On the other hand, our C++ software version (implemented based on the widely used PostgreSQL operators [23]) is compiled with the '-O3' flag, which includes loop unrolling, function inlining, instruction scheduling, aggressive register allocation, loop vectorization, automatic parallelization, and other optimizations to produce faster code. On the other hand, our current choice of the FPGA accelerator design from AMD/Xilinx database library has the potential to be further optimized.

For correctness verification and performance validation purposes, we use the FPGA overlay designs Ver.2020 from AMD/Xilinx Vitis database library 2020.1 [37], the latest version that provides the manually optimized designs for accelerating the TPC-H queries. For the FPGA overlay designs Ver.2022, we use AMD/Xilinx Vitis database library 2022.1 [37], which does not include any manually optimized designs for accelerating the TPC-H queries.

We build the two FPGA accelerator overlay designs: *gqeJoin* and *gqeAggr* using AMD/Xilinx Vitis 2020.1, and they operate at 175MHz and 200MHz, respectively. We also build *gqeKernel* using AMD/Xilinx Vitis 2022.1, and it operates at 178MHz. Tables 3 and 4 below list the resource usage for the Ver.2020 Vitis database overlays on the U280 FPGA and

Table 3. Resource utilization of the Ver.2020 Vitis database overlays on AMD-Xilinx U280 FPGA.

Overlays	LUT	FF	BRAM	URAM	DSP
gqeJoin	248,097 (22.93%)	318,388 (14.17%)	322 (19.96%)	192 (20.00%)	148 (1.64%)
gqeAggr	307,466 (27.96%)	399,958 (17.49%)	263 (16.17%)	256 (26.67%)	626 (6.94%)
gqePart	81,398 (7.52%)	86,471 (3.85%)	77 (4.77%)	256 (26.67%)	10 (0.11%)

Table 4. Resource utilization of the Ver.2022 Vitis database overlays on AMD-Xilinx U50 FPGA.

Overlays	LUT	FF	BRAM	URAM	DSP
gqeKernel	424,564 (48.69%)	434,954 (24.95%)	569 (42.34%)	264 (41.25%)	75 (1.26%)
gqeAggr	298,470 (34.23%)	399,737 (22.93%)	255 (18.97%)	256 (40.00%)	626 (10.52%)

Ver.2022 Vitis database overlays on the U50 FPGA, respectively. As for the evaluation of TPC-H benchmarks, we use Apache Spark 3.1.1 with Scala 2.12. We assume an in-memory database where all data is preloaded in the host CPU's DRAM memory. Thus, we load the entire TPC-H dataset into main memory before executing Spark SQL queries to emulate an in-memory database system. Our execution time includes computation time as well as the data transfer time between host CPU DRAM and FPGA device DRAM. It does not include the query planning time as the scope of this paper focuses on static and automatic compilation.

4.2 Overall Performance Improvement

Figure 8 summarizes the overall performance speedup of SQL2FPGA with AMD/Xilinx FPGA overlay designs Ver.2020 under different design configurations over the Spark SQL execution across all 22 TPC-H queries in SF1 and SF30; and Figure 9 summarizes the overall performance speedup of SQL2FPGA with AMD/Xilinx FPGA overlay designs Ver.2022 across all 22 TPC-H queries in SF1. The baseline Spark SQL design is executed on a 24-thread CPU. To demonstrate the performance improvement between different system configurations in SQL2FPGA, we show three design versions: 1) CPU C++ version implementing all operators used in the query plan using our C++ operator functions implemented based on PostgreSQL [23] optimized C++ operators and execute them entirely on CPU; 2) CPU-FPGA hybrid execution utilizing both CPU and FPGA devices and directly applying the complete set of optimizations, without considering whether an aggressive optimization may lead to performance degradation or not; and 3) best optimized hybrid CPU-FPGA execution plan that exhaustively searches through all optimization combinations to achieve the fastest processing.

Due to the benefits of ahead-of-time compilation, the CPU C++ version designs compiled using the g++ compiler typically perform better than Apache Spark execution with Java virtual machine (JVM). Across all TPC-H queries, the CPU C++ version designs achieve an average of 4.8x and 4.4x performance speedup for SF1 and SF30.

For the hybrid CPU-FPGA execution versions, the exhaustive-search optimized version explores all optimization combinations to obtain the final query execution plan and thus achieves the highest speedups. Note that the searching overhead is rather low with only 32 search space points from our five optimizations, which takes a few minutes to evaluate. With AMD/Xilinx FPGA overlay designs Ver.2020, SQL2FPGA achieves an average performance speedups of 11.3x and 14.6x for SF1 and SF30. SQL2FPGA with AMD/Xilinx FPGA overlay designs Ver.2022, achieves an average
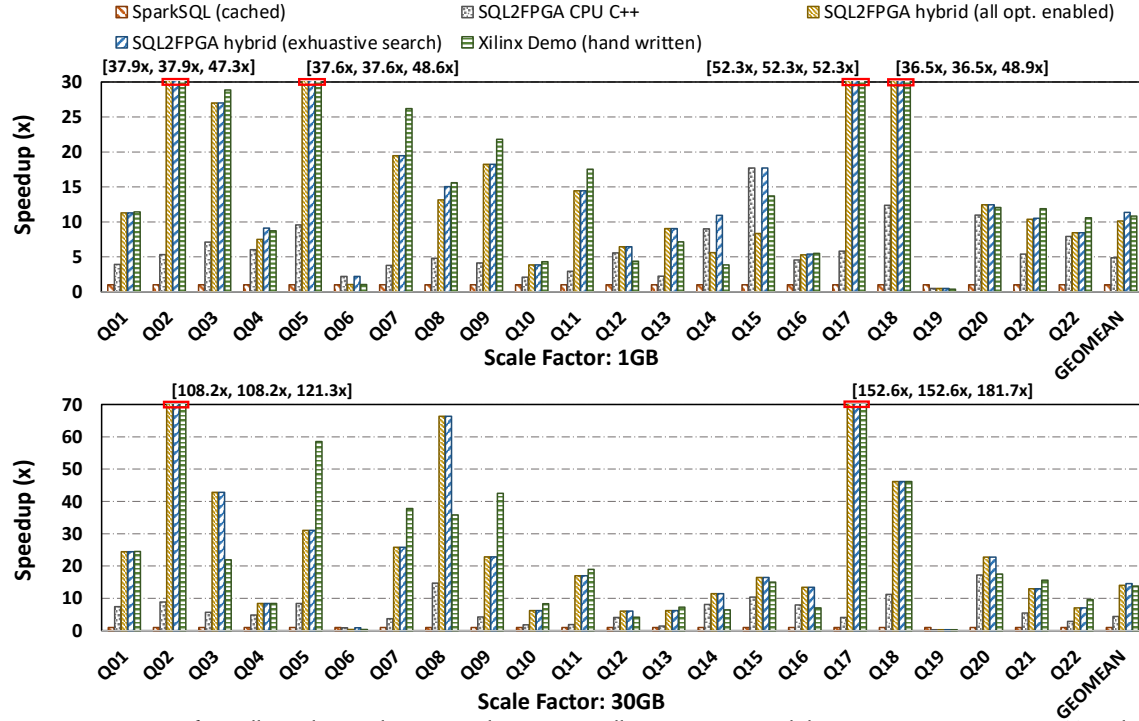
Fig. 8. Comparison of overall speedup results over Spark SQL across all TPC-H queries and their geometric means using AMD/Xilinx FPGA overlay designs Ver.2020.
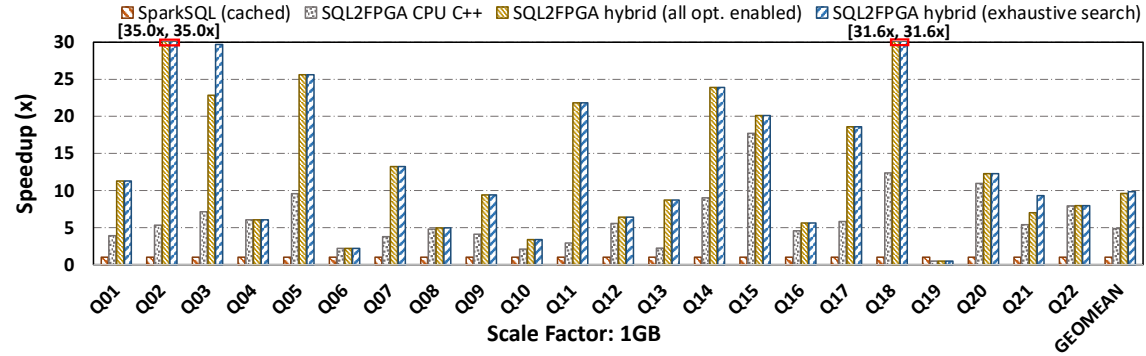

Fig. 9. Comparison of overall speedup results over Spark SQL across all TPC-H queries and their geometric means using AMD/Xilinx FPGA overlay designs Ver.2022.

performance speedup of 9.8x for SF1. On the other hand, the all opt. version designs are optimized with the full set of compiler optimizations (described in section 3.5). SQL2FPGA still consistently achieves 10.1x and 13.9x average speedups over the Spark SQL baseline designs with AMD/Xilinx FPGA overlay designs Ver.2020 for SF1 and SF30; while it achieves an average speedup of 9.6x using AMD/Xilinx FPGA overlay designs Ver.2022 for SF1.

Comparing the CPU C++ implementation and our hybrid execution design enabled with all optimizations, the latter achieves an average speedup of 2.1x and 3.2x over the CPU C++ design for SF1 and SF30 with AMD/Xilinx FPGA overlay designs Ver.2020 and an average speedup of 2.0x using AMD/Xilinx FPGA overlay designs Ver.2022 for SF1,

demonstrating the benefit of hybrid CPU-FPGA acceleration in query processing. For the presented experimental results, we applied our proposed query plan optimizations in the FPGA hybrid mode, but not in the CPU/C++ version as most of our proposed optimizations are tailored for the FPGA execution. Indeed, two of our proposed query plan optimizations, Opt 3-OperatorPruning and Opt 5-CascadedInnerJoinReordering, also benefit the CPU/C++ execution. With these two optimizations applied, on average, our CPU-FPGA hybrid mode remains 2.83x faster than the CPU/C++ execution on the SF30 dataset. Lastly, in term of energy efficiency, for smaller dataset size (e.g., SF1), our approach is 2.14x and 9.21x more efficient than our CPU C++ and Spark SQL designs, respectively.

To further verify the quality of our SQL2FPGA automatic acceleration solution, we also evaluate and compare it with the AMD/Xilinx provided hand-tuned query acceleration designs. The results show that our hybrid execution designs using AMD/Xilinx FPGA overlay designs Ver.2020 achieve similar performance with a marginal 7% performance degradation and around 1% improvement over the AMD/Xilinx provided designs for SF1 and SF30.

Lastly, to better illustrate how the workload is distributed between CPU and FPGA, we examine the following two queries in detail. First, as shown in Figure 4, Query3 from TPC-H retrieves data from three tables and is processed by 8 operators. It takes around 318ms to compute when all operators are executed on CPU for the SF1 dataset. As shown in Figure 4, when the '*FILTER*' and '*SORT*' operations are done on CPU, and FPGA executes '*JOIN*', '*EVALUATION*', and '*GROUPBY*', the execution time is reduced to around 83ms. Second, for Query8 from TPC-H, with a total of 13 operators, pure CPU execution takes around 282ms; hybrid CPU-FPGA execution takes around 83ms with 7 operators done on CPU and 6 '*JOIN*' operators executed on FPGA.

4.3 Speedup for Different Optimization Passes

To quantitatively evaluate the performance impact of the optimization passes, Figure 10 and 11 show performance improvements for AMD/Xilinx overlay design Ver.2020 and Ver.2022, respectively, when incrementally applying the optimization passes as described in section 3.5. For better visualization, Figure 12 and 13 show the exact number of TPC-H queries that benefit from each optimization when using AMD/Xilinx overlay design Ver.2020 and Ver.2022, respectively. Note that the presented performance results are normalized based on the SQL2FPGA CPU C++ version as the baseline design to demonstrate the performance impact of integrating FPGA to accelerate query processing.

no opt – direct offload: Across all TPC-H queries, only nearly half of queries achieve a performance improvement, while the remaining queries show either performance degradation or no performance change, as shown in Figure 12 and 13. The performance degradation is mainly due to the lack of consideration for the acceleration strength of FPGA operators and the performance trade-offs between FPGA acceleration and data transfer overheads. Using the optimized logical query plan directly from Spark SQL without applying any query plan optimizations from SQL2FPGA, *no opt* designs offload every operation supported by the overlay designs to the FPGA device when possible. The no performance change is mainly because these queries do not utilize FPGA overlay designs. This observation also motivates our compiler optimizations. Nevertheless, considering the overall performance impact across all queries, the acceleration benefit still outweighs the performance slowdown, and it achieves around 1.2x performance speedup over the CPU C++ designs for both Ver.2020 and Ver.2022 overlay designs.

opt 1 – strRowIDSub: After enabling the string datatype to row ID substitution optimization pass to extend acceleration coverage on the FPGA overlay designs while reducing the memory footprint between operations, 11 queries from both scale factor datasets using Ver.2020 overlay designs and 10 queries from SF1 dataset using Ver.2022 overlay designs, respectively, achieve improved performance. Especially for queries that were not previously able to offload any operation to the FPGA overlay design, like query #10, could offload three operators to FPGA overlay and thus achieve up to
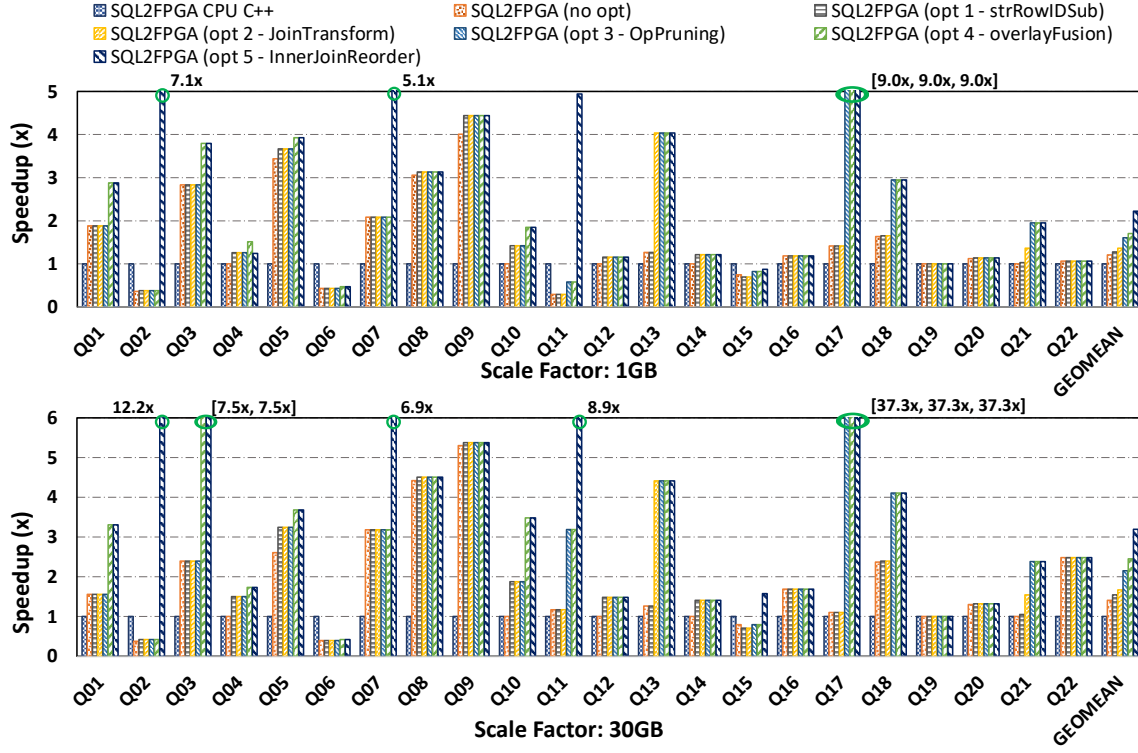
Fig. 10. Performance breakdown of all optimization passes included in SQL2FPGA evaluated across all TPC-H queries and their geometric means using AMD/Xilinx FPGA overlay designs Ver.2020: each optimization is incrementally added based on the prior optimizations.
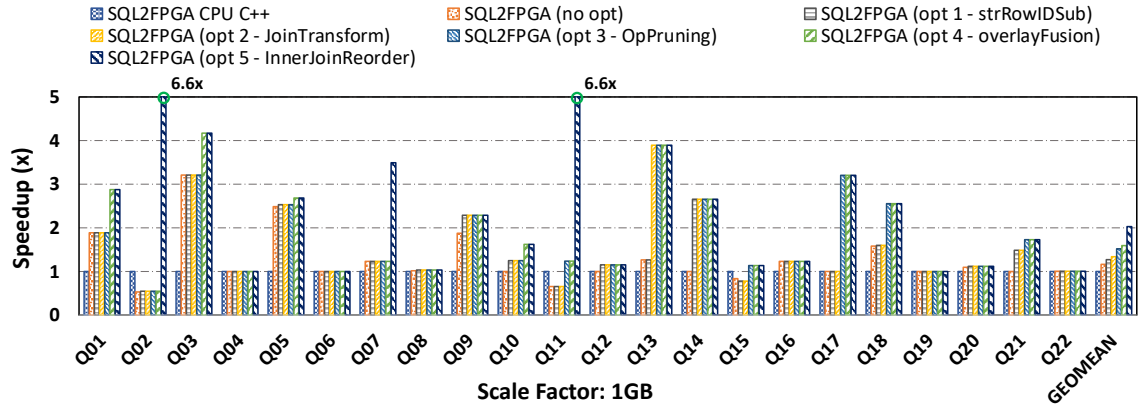

Fig. 11. Performance breakdown of all optimization passes included in SQL2FPGA evaluated across all TPC-H queries and their geometric means using AMD/Xilinx FPGA overlay designs Ver.2022: each optimization is incrementally added based on the prior optimizations.
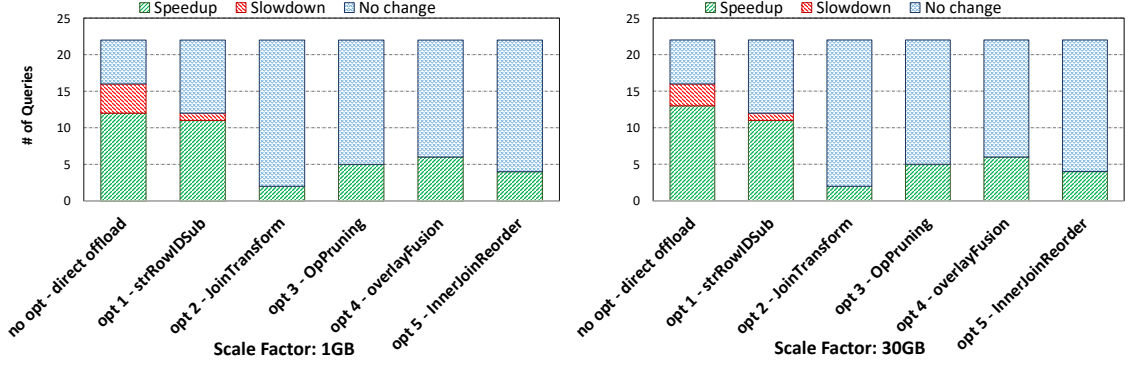
Fig. 12. Statistics for number of TPC-H queries impacted by SQL2FPGA's optimization passes with AMD/Xilinx FPGA overlay designs Ver.2020.
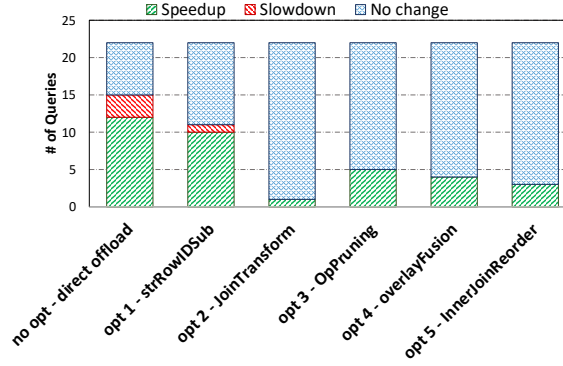

Fig. 13. Statistics for number of TPC-H queries impacted by SQL2FPGA's optimization passes with AMD/Xilinx FPGA overlay designs Ver.2022.

1.42x and 1.88x speedup for SF1 and SF30, respectively. Only one query (query #15) experience performance slowdown (~7% for SF1 and ~13% for SF30), mainly because of the data transfer overhead between CPU host memory and FPGA DRAM, which outweighs the overall FPGA acceleration benefit. Nevertheless, on average, this optimization effectively accelerates query processing by offloading more operators to accelerate on the FPGA overlay designs and lowering the datatype complexity used in operations. Using Ver.2020 overlay designs, it achieves 1.09x and 1.2x speedups over the *no opt* designs for SF1 and SF30 across all affected queries. It also achieves a 1.2x speedup for SF1 with Ver.2022 overlay designs.

opt 2 – joinTransfrom: To extend acceleration coverage on the FPGA overlay designs, the special join transformation improves the processing performance of two queries: #13 and #21, by 3.02x and 1.33x for SF1, and 2.50x and 1.47x for SF30 when accelerated using Ver.2020 AMD/Xilinx overlay designs. For query #13, the optimization transforms and implements the *left outer join* using two separate join operations: a *left anti join* and an *inner join*. For query #21, the optimization detects two special-case dual-key *left anti join* and *left semi join* operations and offloads them to accelerate on FPGA. When accelerated using Ver.2022 AMD/Xilinx overlay designs, this optimization improves processing performance of query #13 by 3.1x, but does not affect query #15 because special-case dual-key *left anti join* and *left semi join* operations are no longer supported. In summary, this optimization extends the FPGA overlay acceleration coverage. Its incremental improvement is around 2.18x and 1.99x better over the previous *opt 1 - strRowIDSub* designs

for SF1 and SF30 across the two affected queries on the Ver.2020 overlay designs; and the incremental improvement is 3.1x improved for one affected query on the Ver.2022 overlay designs.

opt 3 – opPruning: In eliminating redundant computations, the operator pruning optimization removes redundant operations and merges repeating operations in the query plan, thus improving processing performance. Experimental results show that it is effective for five queries.

For query #11, two branches in the query plan carry the same operations, which perform two consecutive join operations: first, join based on the *suppkey* key between *partsupp* and *supplier* tables, then join with the *nation* table on *nationkey*. By merging these two branches and executing the corresponding join operations only once, the performance improves by 1.97x and 2.72x for SF1 and SF30 with the Ver.2020 overlay designs; and by 1.88x for SF1 with the Ver.2022 overlay designs.

For query #15, the same filter operation is called at two separate execution branches in the query plan. Thus, to reduce the execution time, the compiler optimization eliminates one of the function calls and gains 1.20x and 1.13x speedups for SF1 and SF30, respectively, with the Ver.2020 overlay designs; and the gain is 1.46x for SF1 with the Ver.2022 overlay designs.

For query #17, the performance improvement is 6.36x and 33.93x for SF1 and SF30, respectively, with the Ver.2020 overlay designs; and the speedup is 3.21x for SF1 with the Ver.2022 overlay designs.

There are two reasons for the improved performance. First, it removes the group-by aggregation operation on the largest table in the TPC-H dataset, *lineitem* table, which takes around 88% processing time on the SF1 dataset. Second, through the commutative property of join, we merge the previous group-by aggregation operation with the join operation on the same *partkey* key between *lineitem* and *part* tables.

For query #18, the performance speedup is 1.79x and 1.72x for SF1 and SF30, respectively, with the Ver.2020 overlay designs; and the speedup is 1.60x for SF1 with the Ver.2022 overlay designs. This improvement is due to the removal of a duplicated group-by aggregation operation (taking about 36% processing time on SF1 dataset) on the *orderkey* key of *lineitem* table.

Lastly, for query #21, the performance speedup is 1.43x and 1.54x for SF1 and SF30, respectively, with the Ver.2020 overlay designs; and the speedup is 1.16x for SF1 with the Ver.2022 overlay designs. To reduce processing time, the optimization removes three redundant column alias renaming and a duplicated filter operation restricting rows where the *receiptdate* attribute column is less than or equal to the *commitdate* attribute column on the *lineitem* table.

In summary, across the five queries affected by this optimization, the average performance speedup is 2.07x and 3.08x over the *opt 2 - joinTransfrom* designs of the affected queries for SF1 and SF30, respectively, with the Ver.2020 overlay designs; and the average performance speedup is 1.74x for SF1 with the Ver.2022 overlay designs.

opt 4 – overlayFusion: In reducing the expensive PCIe data transfer to exchange input and output data between different FPGA overlay tasks, the overlay fusion optimization maximizes the number of operations carried out within a single overlay design while minimizing the number of overlay acceleration API calls. As listed in Table 5, experimental results show that it is effective for six queries with the Ver.2020 overlay designs, and four queries with the Ver.2022 overlay designs.

For query #1, the number of overlay calls is reduced from 2 to 1, achieving 1.53x and 2.12x speedup for SF1 and SF30, respectively, with the Ver.2020 overlay designs; and the average performance speedup is 1.53x for SF1 with the Ver.2022 overlay designs. For query #3, the number of overlay calls is reduced from 5 to 3, and achieve 1.34x and 3.14x speedup for SF1 and SF30, respectively, with the Ver.2020 overlay designs; and the average performance speedup is 1.30x for SF1 with the Ver.2022 overlay designs. For query #4, the number of overlay calls is reduced from 3 to 1, achieving 1.20x and

Table 5. Number of overlay acceleration API calls before and after applying overlayFusion optimization.

Overlay Design	Version	Q1	Q3	Q4	Q5	Q6	Q10
Ver.2020	Before Fusion	2	5	3	6	2	5
	After Fusion	1	3	1	5	1	3
Ver.2022	Before Fusion	2	5	N/A	6	N/A	5
	After Fusion	1	3		5		3

1.20x speedup for SF1 and SF30, respectively, with the Ver.2020 overlay designs. For query #5, the number of overlay calls is reduced from 6 to 5, achieving 1.07x and 1.14x speedup for SF1 and SF30, respectively, with the Ver.2020 overlay designs; and the average performance speedup is 1.06x for SF1 with the Ver.2022 overlay designs. For query #6, the number of overlay calls is reduced from 2 to 1, achieving 1.09x and 1.09x speedup for SF1 and SF30, respectively, with the Ver.2020 overlay designs. For query #10, the number of overlay calls is reduced from 5 to 3, achieving 1.29x and 1.85x speedup for SF1 and SF30, respectively, with the Ver.2020 overlay designs; and the average performance speedup is 1.30x for SF1 with the Ver.2022 overlay designs.

In summary, the performance results show an average speedup of 1.25x for the SF1 dataset and an average speedup of 1.48x for the SF30 dataset over the *opt 3 - opPruning* designs of the affected queries with the Ver.2020 overlay designs; and the average performance speedup is 1.29x for SF1 with the Ver.2022 overlay designs.

opt 5 – innerJoinReorder: Join operations are intensive in compute and memory, typically occupying a significant chunk of processing time. By reducing the intermediate table data generated based on statistics of the number of input table rows, the inner join reorder optimization aims to reduce data transfer size and compute intensity to improve processing. Our evaluation results show that this optimization is effective for four queries from both scale factor datasets and three queries from SF1 dataset using Ver.2020 and Ver.2022 overlay designs, respectively, to improve processing performance.

This optimization is particularly effective for query #2, achieving 18.87x and 28.76x speedup for SF1 and SF30, respectively, with the Ver.2020 overlay designs; and the average performance speedup is 12.03x for SF1 with the Ver.2022 overlay designs. This is because, in query #2, the original query plan from Spark SQL schedules the most time-consuming join path in an order such that it first joins tables *partsupp* (800,000 rows) and *supplier* (10,000 rows), then table *nation* (25 rows), and lastly, table *region* (5 rows). As a result, the intermediate number of rows generated (for the SF1 dataset) between these join operators are 80,000, 800,000, and 162,880. In contrast, our compiler optimization reorders the join operators to prioritize joining tables with the least number of rows, so we first join between tables *nation* (25 rows) and *region* (5 rows), then *supplier* (10,000 rows), and lastly *supplier* (800,000 rows). The intermediate rows are lowered to 5, 2036, and 162880. As for the SF30 dataset, although the intermediate number of rows is proportionally reduced, the performance increases beyond linear scaling.

For query #7, the optimization swaps the join order between tables *order* (1,500,000 rows) and *nation* (25 rows), and this achieves 2.47x and 2.17x speedups for SF1 and SF30, respectively, with the Ver.2020 overlay designs; and the average performance speedup is 2.84x for SF1 with the Ver.2022 overlay designs. For query #11, the optimization swaps join order between tables *partsupp* (800,000 rows) and *nation* (25 rows), achieving 8.53x and 2.80x speedups for SF1 and SF30, respectively, with the Ver.2020 overlay designs; and the average performance speedup is 6.06x for SF1 with the Ver.2022 overlay designs. For query #15, the optimization treats the number of rows as one from a column aggregation operator and swaps join order with table *supplier* (10,000 rows); this achieves 1.06x and 1.98x speedup for SF1 and SF30 with the Ver.2020 overlay designs.
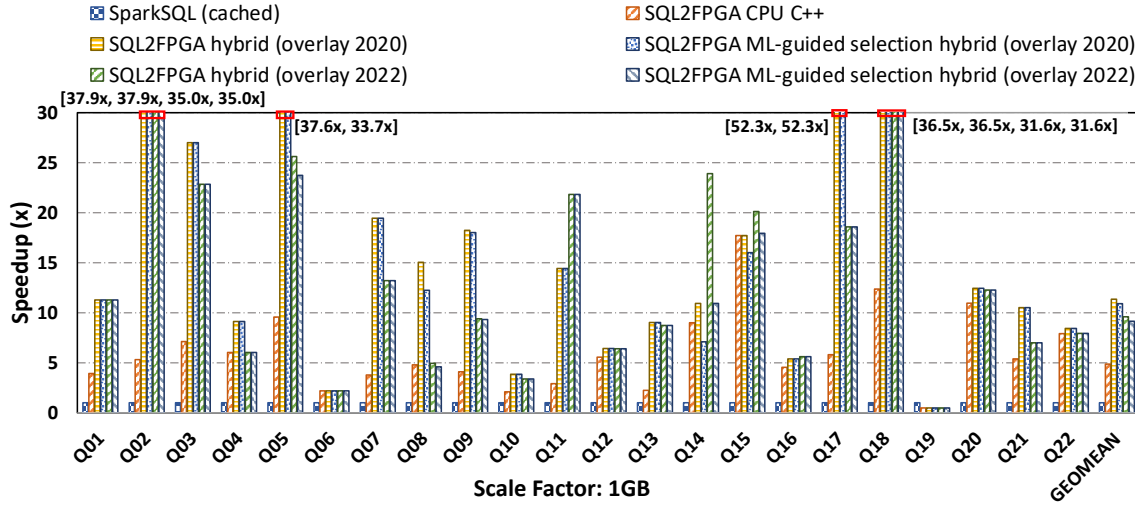
Fig. 14. Overall performance comparison between AMD/Xilinx overlay designs Ver.2020 and Ver.2022.
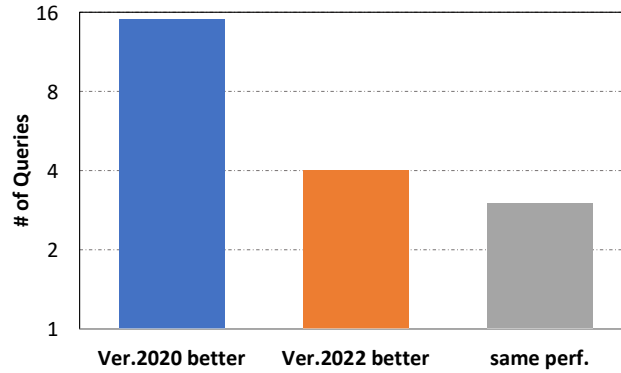

Fig. 15. Statistics for number of TPC-H queries affected between AMD/Xilinx overlay designs Ver.2020 and Ver.2022.

The performance results show an average speedup of 3.22x and 3.58x for the SF1 and SF30 datasets, respectively, with the Ver.2020 overlay designs; and the average performance speedup is 5.91x for SF1 with the Ver.2022 overlay designs.

4.4 Performance Comparison Ver.2020 Vs Ver.2022

To quantitatively evaluate the performance impact between different FPGA overlay designs, Figure 14 compares the performance improvements using AMD/Xilinx overlay design Ver.2020 and Ver.2022 over the SparkSQL execution on CPU. Overall, across all 22 TPC-H queries, SQL2FPGA execution with Ver.2020 overlay design outperforms Ver.2022 overlay design execution by 1.18x. For better visualization, Figure 15 shows the exact number of TPC-H queries that achieve better acceleration between AMD/Xilinx overlay design Ver.2020 and Ver.2022. Ver.2020 overlay design performs better on a majority of queries (15 out of 22): query 2-5, 7-10, 12-13, 17-18, and 20-22; Ver.2022 overlay design outperforms Ver.2020 design on 4 out of 22 queries: query 11 and 14-16; and both overlay versions achieve the same performance for the remaining three queries: query 1, 6, and 20.

Regarding the performance slowdown switching from Ver.2020 to Ver.2022 overlay design, one reason is due to the additional input/output data transfer between host CPU and FPGA for every overlay invocation, as well as the data consolidation process to generate the finalized output table when conducting join operations, described in Section 3.3.2. Across all affected queries, this overhead on average accounts for around 21% of the total FPGA execution time and 5% total hybrid (CPU & FPGA) execution time. Another reason for the lower performance when using Ver.2022 overlay design is that, as the overlay design upgrades to a 64-bit datapath, the *gqeKernel* accelerator hangs due to hash table overflowing and process deadlocking when processing join operations on larger tables (e.g., *lineitem* and *customer* tables). Out of the 15 queries where ver.2022 has the worse performance, eight queries are affected by this; meaning they leverage less FPGA acceleration. Our experiment results show that across all affected queries, this limitation on average accounts for around 42% performance slowdown of the total hybrid (CPU & FPGA) execution time.

For the four queries that achieve performance speedups using Ver.2022 over the Ver.2020 overlay design, only one invocation of the *gqeKernel* is issued to handle a join operation, meaning the the additional input/output data transfer overhead is kept at the lowest level. Although the data consolidation process still remains, the computation speedup from handling only the essential data column (key column) outperforms the Ver.2020 overlay design. Across all affected queries, this achieves around 1.34x performance speedup of the total hybrid (CPU & FPGA) execution time.

Lastly, the remaining three queries that have the same performance between Ver.2020 and Ver.2022 do not use the *gqeKernel* overlay design.

4.5 Machine Learning Guide Platform Selection Performance

Additionally, Figure 14 also shows the performance comparison results of both the ideal performance and the machine learning predicted performance as described in Section 3.1, when selecting the platform (between CPU and FPGA) for the physical execution of query operators. All operator classifiers, except for inner join, have achieved 100% prediction accuracy during training as well for our TPC-H evaluation. For inner join, the classifier has a 96.5% accuracy during training, and is around 93.5% accurate during our TPC-H query evaluation. Compared to the ideal performance, platform selection prediction using machine learning classifiers achieves nearly the same performance for most queries, with an average 4% and 5% performance degradation when evaluated across all 22 TPC-H queries with Ver.2020 and Ver.2022 overlay designs, respectively.

The performance difference is due to incorrect classification for a total of six inner join operations from five queries: query 5, 8, 9, 14, and 15. For query #5, two inner join operations are misclassified. With the small input table sizes, the performance degradation is low, around 11.6% and 7.9% with Ver.2020 and Ver.2022, respectively. For query #8, one inner join operations is mispredicted. The performance degradation is around 22.9% and 7.5% with Ver.2020 and Ver.2022, respectively. For query #9, one inner join operations is misclassified. Due to similar execution performance between CPU and FPGA, the performance degradation is quite low, around 1.2% and 0.64% with Ver.2020 and Ver.2022, respectively. For query #14, one inner join operations is mispredicted. Since the query only consists of three operators and the join operations being most time consuming, mis-prediction has a much higher performance impact, scoring a performance slowdown of 1.54x and 2.18x with Ver.2020 and Ver.2022, respectively. For query #15, one inner join operations is misclassified. The performance degradation is around 10.6% and 12.1% with Ver.2020 and Ver.2022, respectively.

4.6 Performance Evaluation on TPC-DS Benchmark Queries

To further validate the generality of SQL2FPGA, we also test with several TPC-DS queries [30] and show the acceleration performance as below. Compiling the TPC-DS queries using SQL2FPGA is relatively straightforward, but parsing the

database tables into the format that Vitis database accelerator overlay can work with is more challenging. Figure 16 shows the performance speedups for TPC-DS Q1-Q5. Compared to Spark SQL baseline designs, SQL2FPGA CPU C++ designs achieve 8.03x geomean and up to 32.65x speedup whereas the optimized SQL2FPGA hybrid designs achieve 13.36x geomean and up to 46.53x performance speedup.
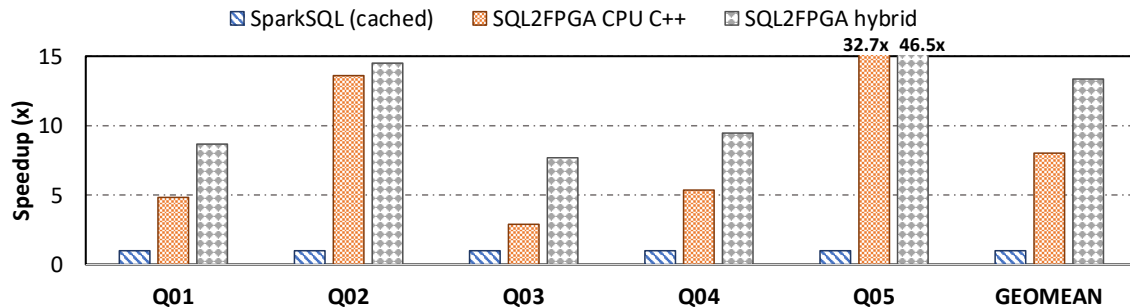

Fig. 16. Comparison of overall speedup results over Spark SQL across TPC-DS queries 1 to 5 and their geometric means.

5   RELATED WORK

5.1   Query Processing Acceleration on FPGA

Previous efforts have proposed FPGA accelerator designs for database operators. Some require reconfiguring the entire FPGA to support different acceleration designs [6, 16, 32]. At the same time, others support a more flexible acceleration of different operators through runtime parameterization [27–29, 34] or partial dynamic reconfiguration [7, 8, 14, 42], which is more commonly used for the embedded FPGA platforms. However, most of these works target near-storage acceleration using "bump-in-a-wire" FPGA accelerators to help reduce data exchange between CPU and disk storage. This differs from our work, where we target to accelerate query processing for in-memory database systems where workloads are mostly computation-bound, which opens new opportunities for FPGA acceleration.

Other works have also developed mechanisms to integrate FPGA acceleration with an existing database system [19, 25, 32, 33]. However, their designs are either not applicable to general database systems or do not provide an automatic compilation to translate queries to FPGA accelerators.

FPGA accelerator for database operations. To accelerate restriction and aggregate operators, Dennl et al. introduced a flexible method to compose the datapath of their accelerator design at runtime through partial dynamic reconfiguration on the FPGA [8]. In [28], Sukhwani et al. implemented a tournament tree algorithm-based FPGA accelerator for sort operation. In [6], Casper et al. proposed efficient hardware designs for selection, merge join, and sort operations and improved memory bandwidth utilization compared to a software version. To dynamically adjust the FPGA accelerator design to match different workload sizes for filter and boolean evaluation, Manev et al. developed a dynamic stream processing accelerator with scalable processing primitives and partial reconfiguration on the FPGA [14].

FPGA acceleration for query processing. In supporting flexible FPGA acceleration for operations, prior efforts have explored partial reconfiguration on FPGA to compose query-specific data paths from pre-compiled components at runtime to accelerate query processing [7, 8, 17, 42]. The series of research conducted by Denn et al. involve transforming an SQL query into a hardware pipeline composed of partially reconfigurable modules, assembled at runtime using a static system that interfaces with the partial modules and the database management system [7, 8, 42]. On the other hand, Mätas et al. focus on the scheduling and providing modular system-level APIs to allow a dynamically reconfigurable

dataflow processing system. This enables independent building, maintenance, and operation of the static shell and reconfigurable modules, enhancing productivity through code reuse and shortening tool runtimes [17]. These work are orthogonal to our work since we focus on the translation and automatic mapping of SQL operations onto existing overlay designs.

While partial dynamic reconfiguration supports flexible switching from one query to the following query using RTL, most datacenter FPGA boards either do not have good partial dynamic reconfiguration tool support or have a high reconfiguration overhead compared to the embedded FPGAs. Targeting datacenter usage, SQL2FPGA uses FPGA overlay designs that could fit entirely onto one FPGA, and we use runtime parameterization for runtime reconfiguration. A similar approach to ours is used in [27, 29], where Sukhwani et al. propose a hardware/software co-design with selection, projection, and sort operations offloaded to an FPGA accelerator, demonstrating the benefits for coupling FPGA-based hardware acceleration with CPU software.

To address the IO bottleneck and relieve the CPU computational pressure, Ibex[34] and IBM Netezza [11] are near storage query processing engines on FPGA performing decompression, restriction, and aggregation operations. [5] further extends [7] and [8] with additional merge-join, sort, and reorder units in the partial reconfiguration module suite. It also developed an energy-aware processing platform that utilizes AXI interfaces for communication with ARM cores.

For integrating FPGA acceleration with real-world database systems, [32] accelerated OpenCL kernel operators on FPGA in a GPU-based database system called OmniDB. [19, 25] modified the in-memory DBMS MonetDB software stack to integrate FPGA accelerators by treating them as user-defined functions (UDF). And [33] integrated an FPGA accelerator with their prototype DBMS system called FCAccel to speed up data extraction from SSDs for SQL processing.

While the research efforts mentioned above show great potential in FPGA-accelerated database operators and queries, they are orthogonal to this paper, where we focus on the automatic compilation of SQL queries onto the CPU-FPGA platform along with the FPGA-aware query plan optimizations. Glacier [16], one of the very few (outdated) query-to-hardware compilers, supports direct translation from SQL queries to RTL code for FPGA. However, to accelerate dynamic analytical processing queries, the repetitive, lengthy hardware synthesis time for every new query limits the applicability of the tool.

5.2   Query Processing Acceleration on GPU

Previous research has also explored query acceleration on GPU. In [12], He et al. presented GDB, a CPU-GPU coprocessing framework for accelerating in-memory relational database systems. Similar to our work, they implemented query plan optimizations to partition operators and data between the CPU and GPU platforms. However, only basic operators such as split and sort are offloaded to the GPU, whereas the FPGA overlay designs used in SQL2FPGA support most query operators except sort. To fill the programming gap between SQL and GPU, Bakkum et al. implemented a subset of the SQLite command processors directly on GPU [4]. In [35], a SQL to GPU compiler called Red Fox is presented and demonstrates an average speedup of 6.48x over an optimized CPU implementation. However, the focus of query plan optimizations targets GPU-only execution instead of a hybrid CPU-FPGA execution model. To improve the GPU resource underutilization issue with the kernel-based execution in CPU-GPU co-processing frameworks, Paul et al. proposed GPL [21], a pipelined execution engine that could achieve up to 48% performance improvement over the kernel-based execution. It mainly focuses on configuration parameter tuning to improve hardware execution, which is orthogonal to our work.

5.3 Query Processing Acceleration on ASIC

While ASIC designs almost always guarantee superior performance and energy efficiency for acceleration, they are inferior to FPGAs concerning the development cost, especially with the rapid changes in today's computing demands. In [36], Wu et al. developed a comprehensive set of ASIC-based operators, defined a domain-specific ISA, and demonstrated a 70x speedup using simulation results over native MonetDB execution on CPU. It devised a programmable spatial-array architecture to support all the basic operators but lacks the consideration of system integration and evaluates only on small dataset size (i.e., 0.01GB). To support scaling for large dataset size, in [40], Xu et al. presented an in-storage query processing engine to enable near SSD processing. It is orthogonal to our work since we target in-memory databases. Also, we propose additional FPGA accelerator-aware compiler optimizations to accelerate query processing.

6 CONCLUSION AND FUTURE WORK

In this paper, we have proposed an automatic compilation framework called SQL2FPGA for translating SQL queries to be processed on the heterogeneous CPU-FPGA acceleration platform. To accelerate compute-intensive in-memory query processing workloads, we first adopted overlay-based accelerator designs from AMD/Xilinx database library [37] that provide flexible operator acceleration through runtime parameterization and are well supported on datacenter FPGAs. To further improve the processing performance of the physical query plan execution, we have implemented a query plan optimizer to 1) extend operator acceleration coverage, 2) eliminate redundant computation, and 3) minimize data transfer overhead. Furthermore, we have utilized machine learning techniques to accurately decide the most efficient platform for physical execution of query operations. Finally, we evaluated our framework by accelerating all 22 TPC-H queries. Experimental results show that SQL2FPGA with AMD/Xilinx Ver.2020 overlay design on average achieves 10.1x and 13.9x performance speedup under SF1 and SF30, respectively, compared to Spark SQL execution on a 24-thread CPU server. Additionally, compared with AMD/Xilinx hand-written optimized acceleration code, SQL2FPGA also achieves similar performance. Furthermore, while evaluated on AMD/Xilinx Alveo U50 FPGA boards, SQL2FPGA using Ver. 2022 AMD/Xilinx FPGA overlay designs also achieve an average speedup of 9.6x at 1GB scale. This work is open-sourced at: https://github.com/SFU-HiAccel/SQL2FPGA.

For future work, we plan to promote wider impact of SQL2FPGA by exploring the integration with an actual big data engine by exploring just-in-time compilation and optimizing the data transfer between CPU and FPGA.

REFERENCES

[1] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *2012 IEEE 28th International Conference on Data Engineering*. 390–401.

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, 1383–1394.

[3] David Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses: The Programmability of FPGAs Must Improve If They Are to Be Part of Mainstream Computing. *Queue* 11, 2 (feb 2013), 40–52.

[4] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. Association for Computing Machinery, 94–103.

[5] Andreas Becher, Florian Bauer, Daniel Ziener, and Jürgen Teich. 2014. Energy-aware SQL query acceleration through FPGA-based dynamic partial reconfiguration. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8.

[6] 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 (FPGA '14)*. Association for Computing Machinery, 151–160.

[7] Christopher Dennl, Daniel Ziener, and Jurgen Teich. 2012. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 45–52.

[8] Christopher Dennl, Daniel Ziener, and Jürgen Teich. 2013. Acceleration of SQL Restrictions and Aggregations through FPGA-Based Dynamic Partial Reconfiguration. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. 25–28.

[9] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. Association for Computing Machinery, 365–376.

[10] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. 2019. In-Memory Database Acceleration on FPGAs: A Survey. *The VLDB Journal* 29, 1 (oct 2019), 33–59.

[11] Phil Francisco et al. 2011. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics.

[12] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.* 34, 4, Article 21 (2009), 39 pages.

[13] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 651–665.

[14] Kristiyan Manev, Anuj Vaishnav, Charalampos Kritikakis, and Dirk Koch. 2019. Scalable Filtering Modules for Database Acceleration on FPGAs. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2019)*. Association for Computing Machinery, Article 4, 6 pages.

[15] MonetDB. 2022. MonetDB: The Database System to speed up your Analytical Jobs. https://www.monetdb.org/ Last accessed December 20, 2022.

[16] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2010. Glacier: A Query-to-Hardware Compiler. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. Association for Computing Machinery, 1159–1162.

[17] Kaspar Mätas, Kristiyan Manev, Joseph Powell, and Dirk Koch. 2022. Automated Generation and Orchestration of Stream Processing Pipelines on FPGAs. In *2022 International Conference on Field-Programmable Technology (ICFPT)*. 1–10. https://doi.org/10.1109/ICFPT56656.2022.9974596

[18] Jian Ouyang, Wei Qi, Yong Wang, YichenTu, Jing Wang, and Bowen Jia. 2016. SDA: Software-Defined Accelerator for general-purpose big data analysis system. In *2016 IEEE Hot Chips 28 Symposium (HCS)*. 1–23. https://doi.org/10.1109/HOTCHIPS.2016.7936221

[19] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. 2017. Centaur: A Framework for Hybrid CPU-FPGA Databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 211–218.

[20] Ippokratis Pandis. 2021. The evolution of Amazon redshift. *Proc. VLDB Endow.* 14, 12 (jul 2021), 3162–3174.

[21] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-Based Pipelined Query Processing Engine. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, 1935–1950.

[22] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, 85 (2011), 2825–2830.

[23] PostgreSQL. 2022. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/ Last accessed December 20, 2022.

[24] J. Ross Quinlan. 1986. Induction of Decision Trees. *Machine Learning* 1 (1986), 81–106.

[25] David Sidler, Muhsen Owaida, Zsolt István, Kaan Kara, and Gustavo Alonso. 2017. doppioDB: A hardware accelerated database. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–1.

[26] Apache Spark. 2022. Unified engine for large-scale data analytics. https://spark.apache.org/ Last accessed December 20, 2022.

[27] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. 2012. Database Analytics Acceleration Using FPGAs. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. Association for Computing Machinery, 411–420.

[28] Bharat Sukhwani, Mathew Thoennes, Hong Min, Parijat Dube, Bernard Brezzo, Sameh Asaad, and Donna Dillenberger. 2013. Large Payload Streaming Database Sort and Projection on FPGAs. In *2013 25th International Symposium on Computer Architecture and High Performance Computing*. 25–32.

[29] Bharat Sukhwani, Mathew Thoennes, Hong Min, Parijat Dube, Bernard Brezzo, Sameh Asaad, and Donna Dillenberger. 2015. A Hardware/Software Approach for Database Query Acceleration with FPGAs. *Int. J. Parallel Program.* 43, 6 (2015), 1129–1159.

[30] TPC. 2022. TPC-DS is a Decision Support Benchmark. https://www.tpc.org/tpcds/ Last accessed December 20, 2022.

[31] TPC. 2022. TPC-H is a Decision Support Benchmark. https://www.tpc.org/tpch/ Last accessed December 20, 2022.

[32] Zeke Wang, Johns Paul, Hui Yan Cheah, Bingsheng He, and Wei Zhang. 2016. Relational query processing on OpenCL-based FPGAs. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–10.

[33] Satoru Watanabe, Kazuhisa Fujimoto, Yuji Saeki, Yoshifumi Fujikawa, and Hiroshi Yoshino. 2019. Column-Oriented Database Acceleration Using FPGAs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 686–697.

[34] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *Proc. VLDB Endow.* 7, 11 (2014), 963–974.

[35] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2014. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. Association for Computing Machinery, 44–54.

[36] 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 (ASPLOS '14)*. Association for Computing Machinery, 255–268.

[37] Xilinx. 2022. Vitis Database Library. https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-database.html Last accessed December 20, 2022.

[38] Xilinx. 2023. Alveo U280 Data Center Accelerator Card Data Sheet (DS963). https://docs.xilinx.com/r/en-US/ds963-u280/Summary Last accessed December 20, 2023.

[39] Xilinx. 2023. Alveo U50 Data Center Accelerator Card Data Sheet. https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds965-u50.pdf Last accessed Dec 12, 2023.

[40] 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)*. 386–399.

[41] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. 2020. FPGA-Accelerated Compactions for LSM-Based Key-Value Store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, 225–238.

[42] Daniel Ziener, Florian Bauer, Andreas Becher, Christopher Dennl, Klaus Meyer-Wegener, Ute Schürfeld, Jürgen Teich, Jörg-Stephan Vogt, and Helmut Weber. 2016. FPGA-Based Dynamically Reconfigurable SQL Query Processing. *ACM Trans. Reconfigurable Technol. Syst.* 9, 4, Article 25 (aug 2016), 24 pages.