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

Alec Lu, Zhenman Fang
School of Engineering Science, Simon Fraser University
{alec_lu, zhenman}@sfu.ca

***Abstract*—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. 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—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.

I. 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 and Facebook—whose business models are highly driven by customer data. Another trend in current database systems is that more data are 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 and expression evaluation are becoming computation-bound in the CPU architecture.

Due to the power and utilization walls [1], 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 [2] 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 [3]. More recently, to demonstrate FPGA acceleration in query processing, AMD/Xilinx developed an open-source library of query acceleration overlays [4], which achieved an average performance speedup of 26x on TPC-H queries [5] over PostgreSQL [6] 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 from Xilinx database library [4] 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 [7], [8]. Unfortunately, few studies target the programming and compilation support to automatically map SQL queries onto the FPGA accelerators, as will be discussed in Section V-A.

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 $\sim$4.8s partial reconfiguration time on Alveo U280, while the average execution time of our queries is only $\sim$5.2s), in this paper, we leverage the AMD/Xilinx open source query acceleration overlays [4] 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 database-agnostic query plan representation such that our optimizations can be leveraged and ported across different database 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.

We evaluate our SQL2FPGA on all 22 queries from the widely used TPC-H benchmark suite [5] with the AMD/Xilinx Alveo U280 [9] datacenter FPGA board. Compared to Apache Spark SQL execution on CPU, under SF1 and SF30, SQL2FPGA achieves average speedups of 10.1x and 13.8x across all 22 TPC-H queries. Compared with AMD/Xilinx well-optimized manual host code, on average, SQL2FPGA 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.
2. A set of hardware-aware compiler optimization passes to further improve the performance of the hybrid CPU-FPGA query acceleration.
3. A quantitative evaluation and analysis of the experimental results on all 22 TPC-H benchmark queries.

II. Background

A. 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 much required to retrieve and process data from a database, whether stored on disk or in main memory. For the interest of this paper, we focus on providing compilation support for accelerating query processing using the heterogeneous CPU-FPGA platform for in-memory database systems.

On 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 top of Figure 1 shows TPC-H query #3 in SQL commands, and the bottom 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 [10], 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.

```
SELECT l.orderkey, c.orderdate,
    c.shippriority,
    sum(l.extendedprice*(1-l.discount))
    as revenue
FROM orders as o, customer as c,
      lineitem as l
WHERE o.mktsegment = 'MACHINERY' and
      o.custkey = c.custkey and
      l.orderkey = c.orderkey and
      c.orderdate < date '1995-03-07'and
      l.shipdate > date '1995-03-07'
GROUP BY l.orderkey, c.orderdate,
        c.shippriority
ORDER BY revenue, c.orderdate
```

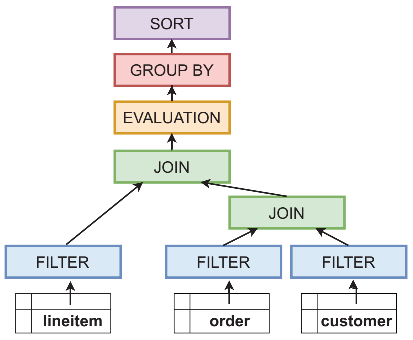


Fig. 1: An example query processing flow using TPC-H query #3: the top listing is the SQL query, the bottom is a query execution plan generated by Apache Spark SQL.

B. Potential of FPGA Acceleration for Query Processing

Due to the inherent high parallelism, reconfigurability, and low power consumption characteristics, FPGA has shown great potential to speed up database systems. Previous research efforts have proposed FPGA accelerator designs for database

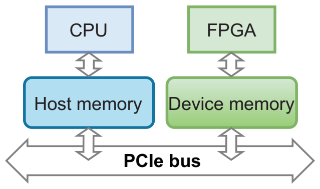


Fig. 2: An overview of heterogeneous CPU-FPGA platform used in SQL2FPGA.

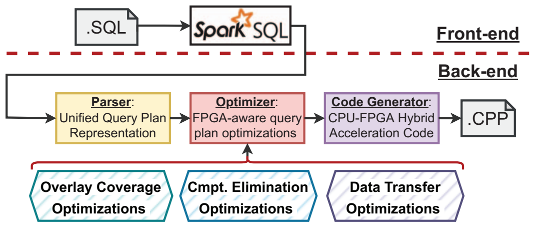
operators [11], [12], [13], [14], [15], [16] to accelerate the entire or part of a query with FPGA [17], [18], [19], [20], [21], [22], [23], and worked on system integration of FPGA accelerators in database systems [24], [25], [26], [27]. Regarding hardware capability, the latest generation of Xilinx Alveo U280 [9] 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 [8]. 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 [4]. 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 III-E. 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.

III. 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 III-A first gives the compilation flow overview of the framework. Next, Section III-B presents details on the vendor-agnostic query plan representation used in our framework. Then, design features of the Xilinx database accelerator overlay designs are presented in Section III-C whereas the CPU operators used in SQL2FPGA are described in Section III-D. Last but not least, in Section III-E, we describe the query optimizer of SQL2FPGA, which mainly



.SQL
Spark SQL
Front-end
Back-end
Parser: Unified Query Plan Representation
Optimizer: FPGA-aware query plan optimizations
Code Generator: CPU-FPGA Hybrid Acceleration Code
.CPP
Overlay Coverage Optimizations
Cmpt. Elimination Optimizations
Data Transfer Optimizations

Fig. 3: Overview of SQL2FPGA compilation flow.

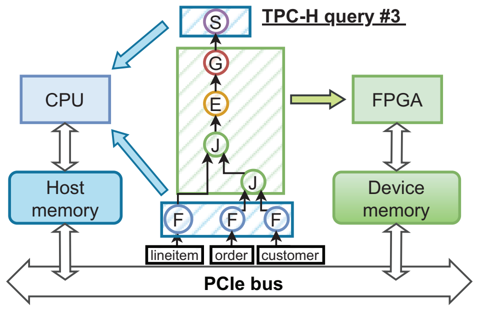


Fig. 4: Physical execution plan of TPC-H query #3 on heterogeneous CPU-FPGA platform.

consists of compiler optimizations to further improve the processing performance of the physical query execution.

A. 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 [28] 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.

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 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 pass through a filter operation on the CPU before handing over the computation to 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, our current approach is mostly based on 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 execution. We plan to build a performance model to guide a proper device selection for operators in future work.

B. Vendor-Agnostic Query Plan Representation

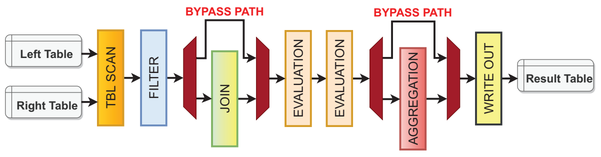
Most of today's query processing engines or database systems, such as Spark SQL [28], PostgreSQL [6], and MonetDB [29], store and represent their query plans in a tree structure. However, their naming conventions and implementation details are all uniquely different. Furthermore, no unified query plan 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

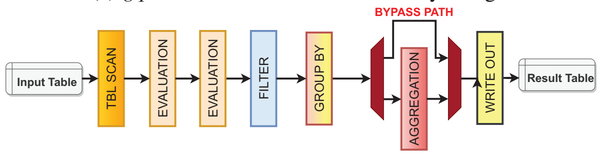
In this work, we write a parser for interpreting and converting the optimized query plan from Spark SQL since it is one of the most commonly used big data processing engines. When porting SQL2FPGA to work with other query processing engines or database systems, the only required design change is the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

C. FPGA Accelerator Overlay Design

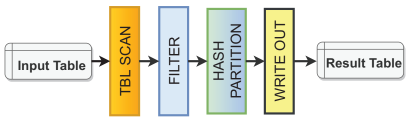
To support flexible acceleration for different database operators without FPGA reconfiguration, accelerator overlay design meets the requirement to be dynamically and efficiently configured through runtime parameterization and is commonly used and well-supported on datacenter FPGAs. In this work, we leverage a set of open-source streaming-based FPGA accelerator overlay designs from Xilinx Vitis database library [4]. Moreover, Xilinx also provides manually optimized



(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: top (*gqeJoin*) mainly focuses on different join operations, middle (*gqeAggr*) mainly targets group by aggregation operations, and bottom (*gqePart*) performs hash partition.

[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.

implementations of acceleration designs for TPC-H queries, which we use as evaluation benchmarks.

As shown in Figure 5, the Vitis database library 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 [4] 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.
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 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. For handling column data of string type, we propose an optimization to extend the acceleration coverage of the overlay design (described in Section III-E). 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*.

D. 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 [6]. The reason for this is two folds. One is due to the FPGA hardware design constraints summarized in Section III-C: some operators still have to run on CPU. The other is for verification of the correctness of our framework.

E. Compiler Optimizations for Query Plan

As Section III-A mentions, SQL2FPGA optimizer consists of several compiler optimizations targeting different performance aspects in accelerating query processing. Firstly, 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, we apply the CascadedInnerJoinReordering optimization.

1) ***Acceleration Coverage Extension on FPGA Overlay****:* Opt 1 – StringRowIDSubstitution: due to the limited 32-bit integer datapath support as described in Section III-C, 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 III-C, 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 $left\_table.key1 = right\_table.key1$ and $left\_table.key2$!= $right\_table.key2$. In this optimization, we traverse the query plans to incorporate these transformations.

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. Secondly, 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) ***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.

IV. Results and Analysis

In this section, first, we present the experimental setup for our design evaluations. Secondly, we evaluate and analyze the overall performance of SQL2FPGA under different design configurations and compare performance results to Spark SQL. Thirdly, we investigate the performance impact of our optimization passes and discuss how they can efficiently accelerate query processing on the hybrid CPU-FPGA platform.

A. Experimental Setup

Benchmark queries. We evaluate SQL2FPGA on all 22 queries in TPC-H Version 2 [5], 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.

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 are deployed on two of the 16nm Xilinx Alveo U280 (with 32 HBM2 banks and Gen3x16 PCIe interface) [9] datacenter FPGA boards as described in Section II-B. For correctness verification and performance validation purposes, we use the FPGA overlay designs from Xilinx Vitis database library 2020.1 [4], the latest version that provides the manually optimized designs for accelerating the TPC-H queries. We build the two FPGA accelerator overlay designs: *gqeJoin* and *gqeAggr* using Xilinx Vitis 2020.1, and they operate at 175MHz and 200MHz, respectively. As for the evaluation of TPC-H benchmarks using real-world, large-scale data analytics framework, we use Apache Spark 3.1.1 with Scala 2.12. For a fair comparison, the entire TPC-H datasets are loaded in the host main memory before the Spark SQL execution to emulate an in-memory database system.

B. Overall Performance Improvement

Figure 7 summarizes the overall performance speedup of SQL2FPGA under different design configurations over the Spark SQL execution across all 22 TPC-H queries in SF1 and SF30. 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 [6] 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 of 11.3x and 14.6x for SF1 and SF30, whereas for designs optimized with the full set of compiler optimizations (described in section III-E) still consistently achieves 10.1x and 13.9x average speedups over the Spark SQL baseline designs.

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, demonstrating the benefit of hybrid CPU-FPGA acceleration in query processing.

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

C. Speedup for Different Optimization Passes

To quantitatively evaluate the performance impact of the optimization passes, Figure 8 shows performance improvements when incrementally applying the optimization passes as described in section III-E. For better visualization, Figure 9 shows the exact number of TPC-H queries that benefit from each optimization. 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 9. The performance

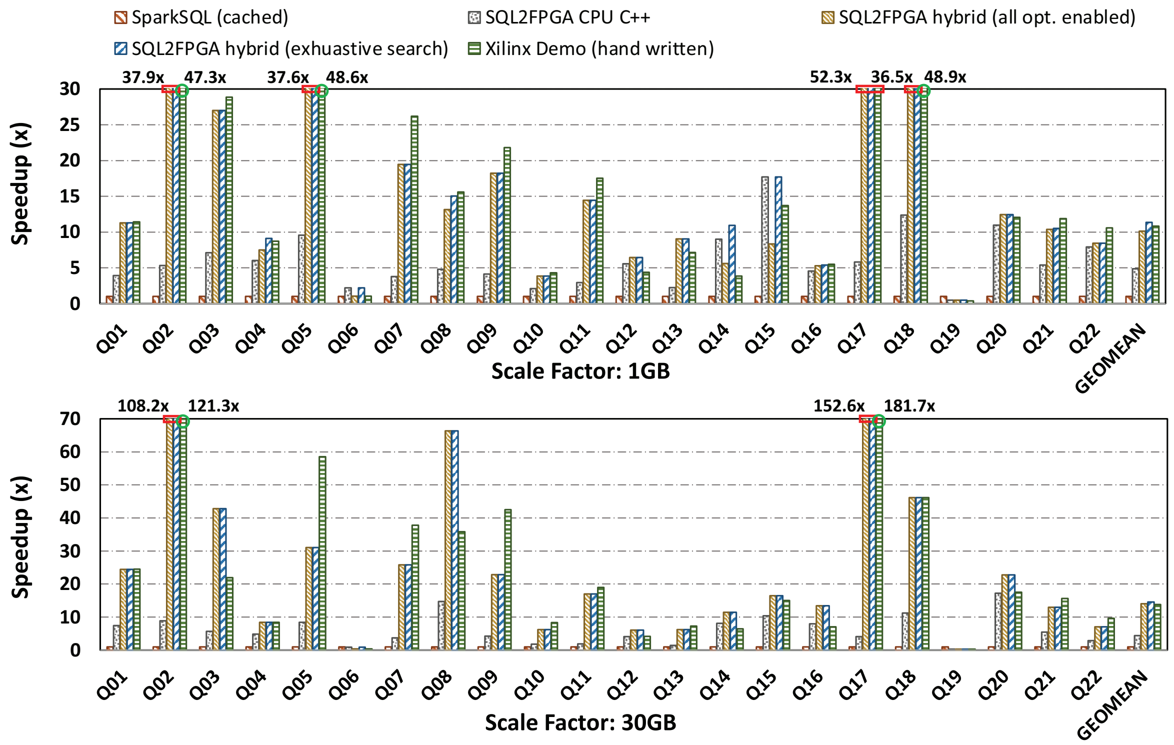


Fig. 7: Comparison of overall speedup results over Spark SQL across all TPC-H queries and their geometric means.

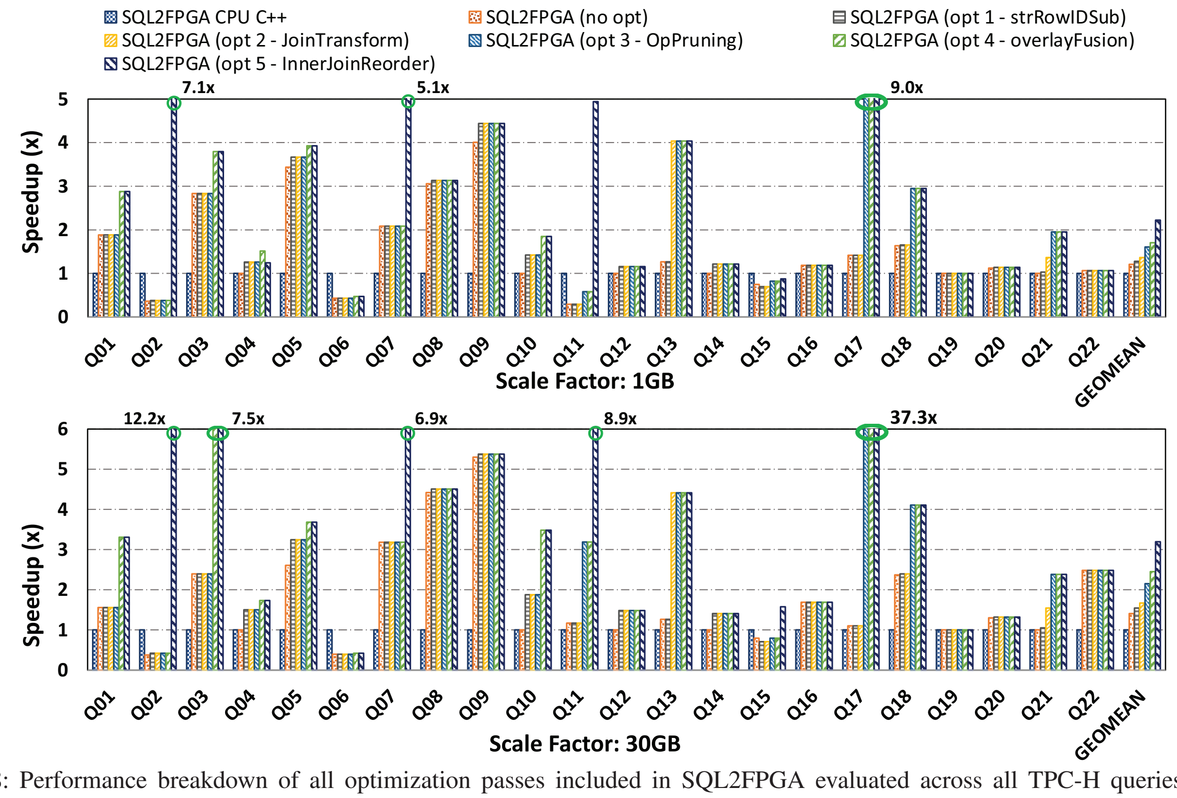


Fig. 8: Performance breakdown of all optimization passes included in SQL2FPGA evaluated across all TPC-H queries and their geometric means: each optimization is incrementally added based on the prior optimizations.

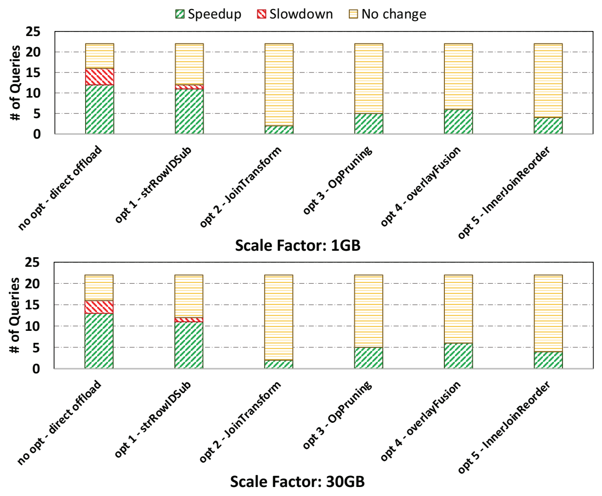


Fig. 9: Statistics for number of TPC-H queries impacted by SQL2FPGA's optimization passes.

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.

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 improve 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 1.42x and 1.88x speedup for SF1 and SF30, respectively. Only one query (query #15) experience performance slowdown ($\sim$7% for SF1 and $\sim$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. It achieves 1.09x and 1.2x speedups over the *no opt* designs for SF1 and SF30 across all affected queries.

opt 2 – joinTransfrom: Also, 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. 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. 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.

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.

Namely, 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.

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.

For query #17, the performance improvement is 6.36x and 33.93x for SF1 and SF30, respectively. The significant performance gain comes from two folds. 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. 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. 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.

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. Experimental results show that it is effective for six queries.

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. 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. For query #4, the number of overlay calls is reduced from 3 to 1, achieving 1.20x and 1.20x speedup for SF1 and SF30, respectively. 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. 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. 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.

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.

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 to achieve performance speedup for SF1 and SF30.

This optimization is particularly effective for query #2, achieving 18.87x and 28.76x speedup for SF1 and SF30, respectively. 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. 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. 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.

The performance results show an average speedup of 3.22x and 3.58x for the SF1 and SF30 datasets, respectively.

V. Related Work

A. 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 [17], [14], [24]. At the same time, others support a more flexible acceleration of different operators through runtime parameterization [19], [13], [20], [22] or partial dynamic reconfiguration [11], [12], [18], [15], 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 [24], [25], [26], [27]. 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 [12]. In [13], Sukhwani et al. implemented a tournament tree algorithm-based FPGA accelerator for sort operation. In [14], 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 [15].

FPGA acceleration for query processing. In supporting flexible FPGA acceleration for operations, Dennl et al. explored partial reconfiguration on FPGA to compose query-specific data paths from pre-compiled components at runtime to accelerate query processing [11], [12], [18]. While dynamic partial reconfiguration supports flexible switching from one query to the following query using RTL, most datacenter FPGA boards either do not have good dynamic partial reconfiguration tool support or have a high reconfiguration overhead compared to the embedded FPGAs. SQL2FPGA uses FPGA overlay designs that could fit entirely onto one FPGA, and we use runtime parameterization for runtime reconfigurations. A similar approach to ours is used in [19], [20], 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[22] and IBM Netezza [21] are near storage query processing engines on FPGA performing decompression, restriction, and aggregation operations. [23] further extends [11] and [12] 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, [24] accelerated OpenCL kernel operators on FPGA in a GPU-based database system called OmniDB. [25], [26] modified the in-memory DBMS MonetDB software stack to integrate FPGA accelerators by treating them as user-defined functions (UDF). And [27] 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 [17], one of the 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 makes the tool impractical.

B. Query Processing Acceleration on GPU

Previous research has also explored query acceleration on GPU. In [30], He et al. presented GDB, a CPU-GPU co-processing 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 [31]. In [32], 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 [33], 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.

C. 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 [34], 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 [35], 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.

VI. 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 [4] 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. Finally, we evaluated our framework by accelerating all 22 TPC-H queries. Experimental results show that SQL2FPGA 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 Xilinx hand-written optimized acceleration code, SQL2FPGA also achieves similar performance. For future work, we plan to improve SQL2FPGA by first adding configuration support to leverage the latest overlay designs from the Xilinx database library and extending SQL2FPGA to support more FPGA platforms and query benchmarks. Second, we plan to leverage machine learning techniques to guide the efficient selection of the CPU or FPGA for operator execution. Lastly, to promote wider impact, we will explore the integration with an actual database engine by exploring just-in-time compilation and optimizing the data transfer between CPU and FPGA. This work will be open-sourced at: https://github.com/SFU-HiAccel/SQL2FPGA.

Acknowledgements

This work is partly supported by NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020; CFI John R. Evans Leaders Fund; Huawei Canada and AMD-Xilinx. We thank AMD-Xilinx Vitis DB team, Prof. Jiannan Wang and Dr. Jinglin Peng from Simon Fraser University, for their insightful discussion and technical support. We also thank the anonymous reviewers for their valuable feedback.

References

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. Association for Computing Machinery, 2011, p. 365–376.

[2] J. Ouyang, W. Qi, Y. Wang, YichenTu, J. Wang, and B. Jia, "Sda: Software-defined accelerator for general-purpose big data analysis system," in *2016 IEEE Hot Chips 28 Symposium (HCS)*, 2016, pp. 1–23.

[3] TPC, "Tpc-ds is a decision support benchmark," 2022, last accessed December 20, 2022. [Online]. Available: https://www.tpc.org/tpcds/

[4] Xilinx, "Vitis database library," 2022, last accessed December 20, 2022. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-database.html

[5] TPC, "Tpc-h is a decision support benchmark," 2022, last accessed December 20, 2022. [Online]. Available: https://www.tpc.org/tpch/

[6] PostgreSQL, "Postgresql: The world's most advanced open source relational database," 2022, last accessed December 20, 2022. [Online]. Available: https://www.postgresql.org/

[7] D. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses: The programmability of fpgas must improve if they are to be part of mainstream computing." *Queue*, vol. 11, no. 2, p. 40–52, feb 2013.

[8] J. Fang, Y. T. B. Mulder, J. Hidders, J. Lee, and H. P. Hofstee, "In-memory database acceleration on fpgas: A survey," *The VLDB Journal*, vol. 29, no. 1, p. 33–59, oct 2019.

[9] Xilinx, "Alveo u280 data center accelerator card data sheet (ds963)," 2022, last accessed December 20, 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ds963-u280/Summary

[10] A. Spark, "Unified engine for large-scale data analytics," 2022, last accessed December 20, 2022. [Online]. Available: https://spark.apache.org/

[11] C. Dennl, D. Ziener, and J. Teich, "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*, 2012, pp. 45–52.

[12] ——, "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*, 2013, pp. 25–28.

[13] B. Sukhwani, M. Thoennes, H. Min, P. Dube, B. Brezzo, S. Asaad, and D. Dillenberger, "Large payload streaming database sort and projection on fpgas," in *2013 25th International Symposium on Computer Architecture and High Performance Computing*, 2013, pp. 25–32.

[14] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '14. Association for Computing Machinery, 2014, p. 151–160.

[15] K. Manev, A. Vaishnav, C. Kritikakis, and D. Koch, "Scalable filtering modules for database acceleration on fpgas," in *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, ser. HEART 2019. Association for Computing Machinery, 2019.

[16] T. Zhang, J. Wang, X. Cheng, H. Xu, N. Yu, G. Huang, T. Zhang, D. He, F. Li, W. Cao, Z. Huang, and J. Sun, "Fpga-accelerated compactions for lsm-based key-value store," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, ser. FAST'20. USENIX Association, 2020, p. 225–238.

[17] R. Mueller, J. Teubner, and G. Alonso, "Glacier: A query-to-hardware compiler," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. Association for Computing Machinery, 2010, p. 1159–1162.

[18] D. Ziener, F. Bauer, A. Becher, C. Dennl, K. Meyer-Wegener, U. Schürfeld, J. Teich, J.-S. Vogt, and H. Weber, "Fpga-based dynamically reconfigurable sql query processing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 4, aug 2016.

[19] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, "Database analytics acceleration using fpgas," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. Association for Computing Machinery, 2012, p. 411–420.

[20] B. Sukhwani, M. Thoennes, H. Min, P. Dube, B. Brezzo, S. Asaad, and D. Dillenberger, "A hardware/software approach for database query acceleration with fpgas," *Int. J. Parallel Program.*, vol. 43, no. 6, p. 1129–1159, 2015.

[21] P. Francisco *et al.*, "The netezza data appliance architecture: A platform for high performance data warehousing and analytics," 2011.

[22] L. Woods, Z. István, and G. Alonso, "Ibex: An intelligent storage engine with support for advanced sql offloading," *Proc. VLDB Endow.*, vol. 7, no. 11, p. 963–974, 2014.

[23] A. Becher, F. Bauer, D. Ziener, and J. Teich, "Energy-aware sql query acceleration through fpga-based dynamic partial reconfiguration," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.

[24] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang, "Relational query processing on opencl-based fpgas," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–10.

[25] M. Owaida, D. Sidler, K. Kara, and G. Alonso, "Centaur: A framework for hybrid cpu-fpga databases," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 211–218.

[26] D. Sidler, M. Owaida, Z. István, K. Kara, and G. Alonso, "doppiodb: A hardware accelerated database," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–1.

[27] 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, pp. 686–697.

[28] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. Association for Computing Machinery, 2015, p. 1383–1394.

[29] MonetDB, "Monetdb: The database system to speed up your analytical jobs," 2022, last accessed December 20, 2022. [Online]. Available: https://www.monetdb.org/

[30] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM Trans. Database Syst.*, vol. 34, no. 4, 2009.

[31] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. Association for Computing Machinery, 2010, p. 94–103.

[32] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili, "Red fox: An execution environment for relational query processing on gpus," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. Association for Computing Machinery, 2014, p. 44–54.

[33] J. Paul, J. He, and B. He, "Gpl: A gpu-based pipelined query processing engine," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. Association for Computing Machinery, 2016, p. 1935–1950.

[34] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "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*, ser. ASPLOS '14. Association for Computing Machinery, 2014, p. 255–268.

[35] S. Xu, T. Bourgeat, T. Huang, H. Kim, S. Lee, and A. Arvind, "Aquoman: An analytic-query offloading machine," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 386–399.