

A Hardware/Software Approach for Database Query Acceleration with FPGAs

Bharat Sukhwani · Mathew Thoennes ·
Hong Min · Parijat Dube · Bernard Brezzo ·
Sameh Asaad · Donna Dillenberger

Received: 2 January 2014 / Accepted: 25 September 2014 / Published online: 28 October 2014
© Springer Science+Business Media New York 2014

Abstract Complex analytics queries often involve expensive operations that may require large computational runtimes leading to slow query responsiveness and hampering real-time performance. Moreover, running these expensive analytics queries inside traditional online transaction processing (OLTP) systems for real-time analytics can affect the performance of mission-critical OLTP queries. On the other hand, support for real-time analytics is considered vital for important business insights and improved market responsiveness. In this paper, we try to address the needs of real-time analytics by enabling hardware acceleration of complex database query operations such as predicate evaluation, sort and projection. While projection helps reduce the amount of data being processed by subsequent query operations, sort is central to most database queries, even those not involving an explicit sort operation. Our system

B. Sukhwani (✉) · M. Thoennes · H. Min · P. Dube · B. Brezzo · S. Asaad · D. Dillenberger
IBM T. J. Watson Research Center, 1101 Kitchawan Road, Yorktown Heights,
New York, NY 10598, USA
e-mail: bharats@us.ibm.com

M. Thoennes
e-mail: tardis@us.ibm.com

H. Min
e-mail: hongmin@us.ibm.com

P. Dube
e-mail: pdube@us.ibm.com

B. Brezzo
e-mail: brezzo@us.ibm.com

S. Asaad
e-mail: asaad@us.ibm.com

D. Dillenberger
e-mail: engd@us.ibm.com

involves FPGA-based composable accelerator for offloading the analytics queries from the host CPU running the OLTP workload. The FPGA-accelerated database system contains accelerator kernels for various database operations and automatic transformation of query operations into calls to these hardware kernels for seamless integration of the accelerator into the database system. Based on the query semantics, each accelerator kernel can be tailored by software to execute specific database operations and different kernels can be fused together to compose a query accelerator. Our query transformation algorithm creates a query-specific control block to customize the accelerator without requiring FPGA-reconfiguration.

Keywords Database · Sort · FPGA · Analytics · Query transformation

1 Introduction

While the world of enterprises builds up its business analytics capabilities, processing real time transactional data for analytics becomes a very attractive option because it helps improve market responsiveness and customer satisfaction and creates competitive advantages. However, in database management systems (DBMS') that handle online transaction processing (OLTP) for direct revenue generation, querying large amounts of OLTP data for analytics is only acceptable when the OLTP Service Level Agreements (SLAs) are not impacted. Examples of potential impacts include longer response time or reduced throughput caused by resource contention between OLTP and analytics query workloads. Additionally, certain expected level of performance is required on the analytics queries to guarantee real time responsiveness.

To address the above challenges, we explore the acceleration of query processing via field programmable gate arrays (FPGAs). Analytics queries usually involve CPU-intensive operations such as predicate evaluation, column projection, record sorting and joins. Predicate evaluation aims to extract the desired rows from the target database table while projection selects the necessary columns from those rows, effectively removing unnecessary data during the query flow when applied in early stage. Sorting is an expensive operation that not only is needed to provide record ordering for applications, but is also often used in other query operations such as join and grouping. Using FPGAs to process these query operations not only improves the query response time, it also minimizes the impact on OLTP from analytics queries by offloading the expensive analytics queries out of the OLTP resources. In our previous work [1,2], we addressed the acceleration of predicate evaluation, row decompression and table join operations using FPGAs and introduced a query control block (QCB) structure to invoke these tasks on the accelerator. The current paper presents FPGA-based acceleration kernels for column projection and record sorting operations and discusses the enhancements to the query control block to support these operations. The current paper is an extension of our work presented in [3]. Further, in this paper, we also present the required software support for seamless exploitation of the FPGA kernels from a standard database system. Given a standard structured query language (SQL) query, our query transformation routine automatically maps these query operations onto the

FPGA accelerator. For completeness, the current paper also briefly revisits some of the ideas presented in [1].

Our FPGA-accelerated database system consists of an off-the-shelf PCIe-attached FPGA card implementing the different query operations, tightly integrated with a commercial database restructured to exploit the FPGA. The restructured database identifies the SQL operations that can be offloaded to the accelerator and transforms those operations into a query control block structure that is interpreted by the FPGA to customize its logic as per the query. The use of the query control block allows a given FPGA hardware image to support multiple different queries without reconfiguring the FPGA. Moreover, our accelerator directly operates on, and generates outputs in, database-formatted pages, for seamless integration into the standard database flow.

The rest of the paper is organized as follows: Sect. 2 presents related work in database acceleration and Sect. 3 provides an overview of different operations in database queries. In Sect. 4, we discuss the DBMS support to enable query acceleration in hardware and in Sect. 5 we present our query accelerator on FPGA. Section 6 discusses integration of the query accelerator with DBMS. Section 7 presents experiments and results and Sect. 8 concludes the paper.

2 Related Work

With the stagnation of processor frequencies due to power constraints, various parallelization techniques have been applied to improve query performance in databases. These include multi-core and multi-threaded architectures [4, 5], SIMD instructions [6, 7], and accelerators like GPUs [8] and FPGAs [9–11]. Effective use of multi-core requires careful data partitioning and updates to reduce memory conflicts and contention between cores [12] and suboptimal utilization of the cores in large multi-core systems results in power inefficiencies [13]. SIMD exploitation, on the other hand, requires the DBMS data to be laid out in specific formats and packed tightly into 64 or 128 bit registers, thus requiring extra processing or compilation for each database table [14]. The relatively rigid architecture of GPUs results in limitations similar to those of SIMD. Prior work using FPGAs [10, 11] mainly focus on compilation-based approach where queries are first compiled into hardware description and then synthesized into FPGA circuit. In our work, we enable the FPGA to execute different queries without reconfiguration using a software-generated query control block. A recent work [15] proposes the use of partial reconfigurable structures for mapping query operations to FPGA at run-time. The basic idea is to maintain a library of modules implementing single query operations and then assembling them at run-time to create the query logic. The run-time generation of FPGA implementation for a query requires solving a dynamic interconnect problem. Their current implementation is limited to fixed-length columns and supports only projection and predicate evaluation.

Traditional database applications and new data stream management applications have a large performance component dominated by sort [16]. While existing FPGA work for database acceleration target query operations such as predicates, there is little prior work on accelerating sort for databases. Nevertheless, several architectures have been proposed to accelerate stand-alone sort in hardware, e.g., sorting networks

[17, 18] and linear sorters [19–21]. Harkins et al. [22] compared various sort implementations on FPGA and concluded that radix sort is the best-suited algorithm for FPGA implementation. A hybrid between a FPGA insertion sorting unit and a FPGA merge FIFO sorting unit [23] provides a speed-up of between 1.6 and 25 compared to a quicksort CPU implementation. In [24], the FPGA implementation allows continuous sorting where the data arrives in real time, without any need for prior storage of all the data. The above schemes, however, do not target database applications and do not handle large payloads. In our current work, we present an FPGA-based hardware-efficient streaming sort for database operations with support for large payloads and large sorted runs.

3 Operations in Database Queries

Below we provide brief description of the most commonly used operations in database analytics, namely predicate evaluation, projection, record sorting, table join and group-by aggregation. These operations are used to select a subset of the database records and format the result as per the query to present to the user.

3.1 Predicate Evaluation

Among the various operations in relational database processing following the SQL standard, predicate evaluation is one of the most commonly used. It refers to the process of applying query filtering criteria to database rows in order to retrieve a subset of the DBMS table. Queries typically require logical inequality or equality comparisons of one or more fields (columns) from the records (rows) against constants. For a very simple example, “*Select FirstName, LastName From Customers Where ID=1 and LastName = 'Smith'*” is a SQL statement asking for *FirstName* and *LastName* of the rows that qualify the predicate expression “*ID=1*” and “*LastName = 'Smith'*” in a “*Customers*” table. In addition to comparison predicates, queries may also test set containment for a field in a record, for example an in-list predicate “*where ID in (1, 3)*”. Such predicates can also be transformed into individual predicates such as “*where ID =1 or ID=3*”.

In OLTP databases, efficient access to one or few records is usually achieved by means of an I/O efficient B+ tree index data structure based on the data retrieval key. Analytics queries, on the other hand, typically involve scanning the entire table. Although the current paper focuses on predicate evaluation for table scan based analytics queries, our approach may also be applied to predicate evaluation during index scan.

3.2 Record Sorting

As mentioned earlier, sort is a pervasive operation in DBMS. In batch and utility processing, it is invoked by various functions such as data reorganization, index build, and statistics collection. In query processing, the most intuitive use of sort

is when an application, via SQL, requests the query result to be ordered by certain data attributes. A DBMS may also invoke a sort implementation to support other SQL operations such as aggregation. In addition, for a given query operation, a DBMS may provide multiple implementations, some of which utilize sort, and allow the DBMS query optimizer to decide which is the most efficient option for a particular query request. For example, “sort merge join” is one of the most common implementations for table join, an operation that merges fields from two tables by matching common field values from both; the sort step is critical to this overall operation. Another example of the implicit use of sort is for improving data access sequential patterns by sorting unordered record IDs, e.g. as obtained from table indexes, before retrieving the data. Extensive studies and analyses have been undertaken to improve sort performance on modern CPUs [25]. Despite the high level of effort devoted to this task, sort remains an expensive operation whose execution on an accelerator could provide the benefits of performance improvement and cycle offload.

Several characteristics of sort as used in query processing are highly relevant in the context of this paper. First, various data types are supported in DBMS, hence a sort implementation needs to accommodate various data types and data sizes as sort keys. Second, for large amounts of data processing, it is very often the case that sort has to be carried out in two or more phases due to memory constraints. In such a scenario, the first phase generates multiple intermediate sorted subsets (or runs), and the second and subsequent phases merge these subsets into one final sorted result. Third, a sort operation seldom just processes the sort keys. Instead, each key is typically associated with a much larger payload, ranging from a few bytes to hundreds or even more bytes per record. The handling of payload data also impacts the performance of a sort implementation. On the CPU, separating sort key and sort payload only yields benefits when the payload size is very large, in particular considering the potential performance cost of random memory or I/O accesses in the post-sort step to merge the payload data with the sorted keys.

3.3 Column Projection

Projection is the operation that extracts desired attribute fields (columns) from a database row. In high volume and high throughput transaction processing databases, data in a table are stored in row format. A record’s fields are physically co-located for efficient inserts, updates and deletes. However, queries that access many records often request only a few columns from each record for reporting or analytics purposes. Performing projection on data records at an early stage of query processing reduces the amount of data to be processed in the later stages.

For an explicit example, the SQL statement *select LastName, FirstName, AccountID, Balance from Savings_table where Balance > 1000 order by AccountID* illustrates how sort and projection are requested in a DBMS query. The attributes following the “select” clause are projection columns and the attribute following the “order by” clause is the sort key.

3.4 Joins and Aggregation

Besides the above mentioned operations, analytics queries often include table join and aggregation operations. A join operation aims to match records from two or more tables based on a common field. In our previous work [2], we have addressed FPGA-based acceleration of hash joins, the most commonly used approach for table joins. Another commonly applied method for joins is sort-merge join where the main operation is sorting the tables to be joined. Some of the work presented in this paper can be applied for accelerating sort-based table joins.

Aggregation is an operation that is used to gather summary data from query processing. During aggregation, rows with common values on certain columns are grouped together; set functions such as sum, average, count, etc. are performed on other columns of interest across all the rows within each group. The grouping process can take advantage of algorithms such as hashing or sorting.

4 DBMS Support for Acceleration Enablement

Executing database queries in a hardware accelerator requires the accelerator to be aware of the query semantics and the format of the data to be processed. In a given database system, the individual record structure, i.e., the number of fields in the record as well as their types, lengths, and positions within the record, vary from table to table. Moreover, even for a given table, the fields in the *where* clause (predicate), the *select* clause (projection) and/or the key field(s) used to sort the records may vary from query to query. One way to offload different queries is to create multiple different accelerator designs, each purpose-built for a specific query. This, however, is unattractive as it is cumbersome to create and maintain multiple designs, requires prior knowledge of all candidate queries and requires reconfiguration of the FPGA for each query. To create a flexible hardware accelerator that enables reuse of the FPGA design across a large number of queries, we employ a *soft-configuration* approach, where the FPGA engine is designed such that it can be tailored to different queries by loading certain query-specific parameters, without having to reprogram the FPGA. During soft configuration, the structure of the current record and the specifics of the query operations are sent to the FPGA using a data structure called the query control block (QCB). A QCB presents the query semantics to the hardware accelerator in a form that can be interpreted by the accelerator. Transforming a SQL query into this QCB data structure is a preprocessing step and is performed on the host processor. At the start of a new query, the corresponding QCB is downloaded into the FPGA and is parsed to customize the predicate, projection and sort logic on FPGA to execute the target query. Just as DBMS' cache query plans for repeating queries to reduce query optimization cost, QCBs can also be stored and reused.

4.1 Query Control Block

A query control block is a data structure used for mapping different query operations onto the hardware accelerator. It contains query-specific information required to tailor

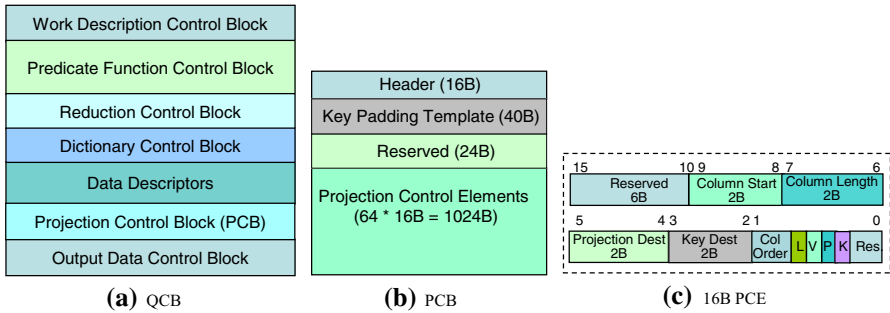


Fig. 1 a Query control block, b projection control block and c projection control element

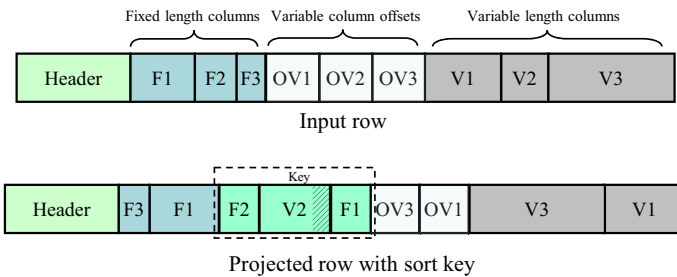


Fig. 2 Examples of input and projected record layout in the target DBMS

the FPGA application logic to execute the query at hand. Different sections of the QCB contain information about different query operations to be executed in hardware (see Fig. 1a). Predicate function control block and reduction control block provide the information about the different predicates to be evaluated and the logical relation between those predicates, respectively. The data descriptors provide the pointers to the database pages to be copied into the accelerator for processing whereas the output data control block provides the memory locations on the host processor to copy the results into. We refer the reader to [1] for details on different regions of the QCB.

4.2 Projection Control Block

In this work, we have enhanced the QCB structure from [1] to include a projection control block (PCB). Similar to the predicate and reduction control blocks, the projection control block is a set of data structure used to indicate to the accelerator the projection and sort tasks to be performed. Since both projection and sort operation require identifying and extracting the desired fields from the database rows, we briefly discuss the format of input and output rows in our target database before presenting the structure of the projection control block.

Figure 2 shows the layout of input and projected rows for our target DBMS. A row may contain fixed length columns, where the column width and its position is the same across different rows, as well as variable length columns, where the length of the

column and its position varies from row to row. The fixed length columns in the row are placed at the beginning, followed by the offsets (pointers) to the starting position of the variable length columns, and finally the variable length column data. The projected row may additionally contain the sort key, inserted between the last fixed length column and the first variable column offset. Any key column with variable length is padded to its maximum length using a column-specific pad character. The order of fixed and variable length columns in the projected row follows the output order in the query.

For a query to be offloaded, the layout of different columns in database rows and the desired projection and sort operation is specified in the projection control block (Fig. 1b). The PCB header contains metadata information about the query, such as the number of fixed and variable length columns to be projected, the starting and ending positions of the variable offsets in the input row, the length of the sort key and the order of sorting. We support composite sort keys of up to 40B in length formed using up to 16 variable and/or fixed length columns arranged in any order. For padding the variable length columns in the key, a 40B key padding template populated with the pad character is provided.

Each column to be projected is specified using a projection control element (PCE) in the PCB (see Fig. 1c). A 16B PCE contains fields for the column start position in the input row and its length as well as its position in the projected row and in the sort key. For variable length columns, the PCE contains the position of the variable offset and the maximum length for that column, along with the column's order in the input row. The order is used to determine the variable column destination in the projected row. Our current implementation supports projection of up to 64 fixed or variable length columns. This limitation exists simply due to the lack of space in the PCB. More than 64 columns can, however, be projected by specifying multiple consecutive columns using a single PCE.

4.3 Query Transformation for Hardware Mapping

The control blocks for predicate evaluation, projection and sort in a QCB are data structures for soft-configuring the FPGA execution. Different operations need to be mapped to the corresponding control blocks in the QCB, e.g., columns in SELECT clause map to PCEs in the PCB, predicates in WHERE clause map to elements of predicate function control block (PFCB) and reduction control block (RCB), and columns in ORDER BY, GROUP BY etc. also map to keys in PCEs for extraction before sorting.

Simply examining the SQL query is not sufficient to provide all the information needed in the QCB for different operations. One reason for this is that a SQL statement does not describe the specific physical data layout information needed by the FPGA operators. For example, information required in the PCEs like the column position and length for fixed length columns or column offset for variable length columns cannot be obtained from a SQL query; similar problem exists for predicate and sort as well. Secondly, a SQL statement does not describe where and how to invoke the accelerator in a query's executions plan. As such, the usage of projection and sort operators is not limited to what appears in the SQL; the DBMS optimizer might use these operators as

part of implementing other operations. For instance, for a particular query with join operation, the DBMS query optimizer may select sort merge join path (sort both tables before joining them) over other join methods, hence sort is needed. Sort may also be used to sort record IDs so the table IO operation is more efficient for a particular query. Finally, some queries may simply not benefit from FPGA offload or may require some restructuring in order to exploit the FPGA engines and this is mostly determined by the optimization component in the database engine before query processing. Due to these reasons, query transformation for FGPA acceleration needs to be done after the DBMS has transformed the SQL statement instead of using an abstract syntax tree (AST) derived from SQL directly.

We devised a methodology to identify the operations of a generic database query that can be offloaded, transform those operations into regular structures that can be mapped onto hardware and automatically create the QCB to represent those operations. The query predicate expression is converted into a perfect binary tree representation where each leaf node represents a single query predicate whereas each non-leaf node represents the reduction operation between its two children. The output at the root of the tree represents the entire predicate expression.

Our query transformation methodology has two phases. In phase 1, database internal structures are processed to efficiently extract relevant information for transforming the query and creating the control blocks in QCB for specific query operations that are candidates for offloading. In phase 2, a determination is made on whether the query operation can be mapped to the FPGA given the hardware resource limitations. For the case when the operation cannot be mapped, a restructuring of query operation may be undertaken such that the resulting query operation can be executed on the available hardware. The restructuring step involves decomposing a query operation into sub-operations. The execution of resulting (sub) operations on the FPGA is orchestrated by the software in such a manner so as to achieve the same net effect as with executing the query operation as a single entity.

There are several advantages of our approach for transforming the query operations into FPGA-specific control blocks. Since it directly exploits database internal structures, it is more efficient as it leverages optimizations and annotations made by the database during query parsing, optimizing and (possibly) rewriting steps that typically precede the transformation. Secondly, our approach identifies early-on in the QCB creation process whether a query operation can be mapped to hardware, thus making the approach much more cost-effective. Finally, our approach overcomes hardware implementation limitations by efficiently partitioning a query operation so as to make it amenable to FPGA execution.

We next showcase our approach through an example query predicate composed of selection clauses. Consider the following query predicate:

$$(f1 = v1 \& ((f3 < v2 \& f6 = v3) \& (f2 > v4 | f8 <= v6))) \quad (1)$$

This predicate has five selection criteria and different AND/OR operators connecting them. The parentheses in the predicate capture the level of nesting for the selection clauses. One can easily look at the predicate and create an appropriate tree representation. However, different databases may store the predicate (with its selection clauses

and parentheses) in different data structures. To automate the process of transforming the query into a hardware-friendly representation such as a binary tree, one needs to identify these structures and process them efficiently to transform into such representations and then create the corresponding QCB for FPGA mapping. In the above example, we can maintain a count of the difference of the number of left parentheses to the number of right parentheses for each selection clause. Thus, for “(f1=v1” this count is 1, for “((f3<v2” this count is 2 and so on. We devised an algorithm called *NodeBuild* which utilizes this information to infer the level of nesting of each selection clause in the tree. This also determines how to combine the results from different selection clauses (leaf nodes) in the corresponding reduction tree (non-leaf nodes).

Algorithm: NodeBuild

1. Create an array of parentheses count for each selection clause in the predicate, moving from left to right in the predicate. For the example shown in equation (1), this array is $\{1, 2, -1, 1, -3\}$.
2. Find the pivot point such that the absolute sum of all the elements to the left of the pivot point is the lowest and is equal to the absolute sum of all the elements to the right of the pivot point. *In the example, the pivot point is between the first and second element because the $abs(sum_left) = 1$ and $abs(sum_right) = abs(2-1+1-3) = 1$.* It is trivial to show that such a pivot point always exists.
3. Divide the array into two sub groups by the pivot point. If there are more than one pivot points, use the one that yields a partition which is balanced in terms of absolute sums of the elements in each sub-array. This is to have shorter path of the predicate network. *For the example, the “right group” is $\{1\}$ and the “left group” is $\{2, -1, 1, -3\}$ which both have the absolute sum of 1.*
4. Recursively perform steps 2 and 3 on each group until the group only has 1 or 2 elements. *In the example, continuing on the “left group”, it gets partitioned into a left group of $\{2, -1\}$ and a right group of $\{1, -3\}$. This completes the step.*
5. Using the recursion level at which an element ends, determine its depth in the binary tree.

In our implementation we maintain an intermediate (tree) data structure to store the information generated at different steps of the algorithm. Figure 3 shows this intermediate tree for the example query and the corresponding binary tree representation for FPGA mapping. The resulting binary tree is used to create the predicate function control block (PFCB) and reduction control block (RCB) of the QCB for executing the query. The PFCB carries information to program the Predicate Elements (PEs) on the FPGA with the individual query predicates. PEs on the FPGA correspond to the leaf nodes in the binary tree representation of the query. The PFCB contains information such as the comparison operation (e.g. less-than, greater-than etc.), the predicate value (v1, v2 etc.) and the starting and ending position of the target column for each PE in the network. The RCB carries information for the logical Reduction Units (RUs) in the FPGA predicate network, to link the individual predicates as per the query. The FPGA RUs implement the non-leaf nodes in the above binary tree and can implement different logical reduction operations. Note that PFCB and RCB combined together must represent a fully populated binary tree in order to map to the corresponding binary tree in hardware. Even if the operations of a query predicate form a left-most

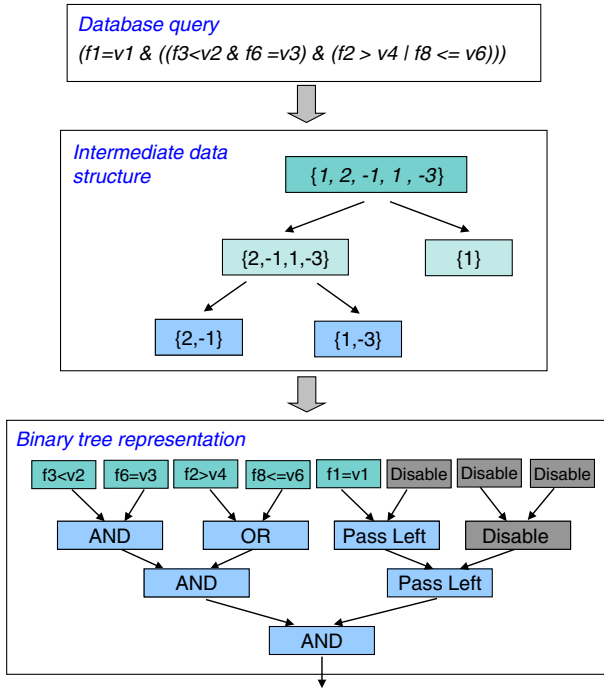


Fig. 3 A predicate query, the intermediate data structure and the final binary tree representation

or right-most tree, all the nodes in the tree need to populated with comparators or pass-through elements.

5 Query Acceleration on FPGA

5.1 Predicate Evaluation Accelerator

The basic unit of computation for evaluating query predicates on FPGA is a predicate element (PE). A PE evaluates a single predicate by comparing a constant predicate value (P) against up to a 64-bit long column of the streaming database rows, generating a 1 bit output indicating whether the predicate evaluates to true for the current row (Fig. 4a). Complex predicate expressions are evaluated on the FPGA using a chain of PEs, each evaluating a different predicate concurrently and independently of each other. A configurable, binary, network of reduction units (RU) combines the individual outputs of the PEs to implement complex filter criteria as per the query, generating the final 1 bit result of evaluating the predicate expression. We refer to the PEs and the reduction network together as a row scanner (Fig. 4c). Note that the structure of the row scanner hardware matches that of the final binary tree representation of the query from the query transformation step (Sect. 4.3).

The number of PEs inside a row scanner and the size of the reduction network are configurable at FPGA synthesis time, to allow for area vs. complexity tradeoffs. As

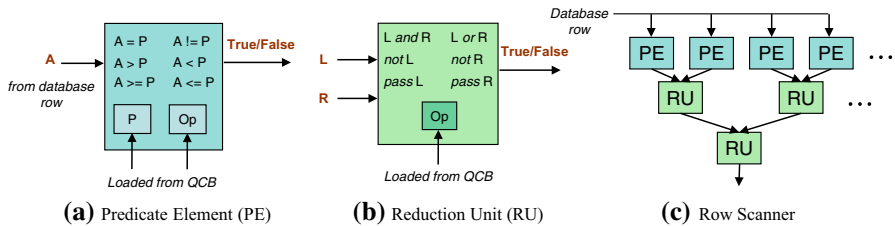


Fig. 4 Predicate evaluation logic on FPGA

discussed earlier, a QCB is used to customize a given FPGA configuration by loading the PE comparison operation, the predicate value and arbitrary reduction patterns into PE's and RU's memories on a per-query basis. One or more PEs in the chain can also be disabled to facilitate the implementation of arbitrary query reduction patterns.

As one might expect, for certain queries, the number of PEs inside the row scanner may be fewer than that required to evaluate the query. For example, this may happen for an in-list query which is handled by the FPGA by flattening the list into individual predicates ORed together. If the size of the list is larger than the number of PEs in the row scanner, such flattening may not be possible. Our query transformation step on CPU can decompose such queries into multi-pass predicate evaluation queries (see Fig. 14; more details in Sect. 6). Depending on the data size and the reduction operation linking the different passes, the final merging of partial results can be performed on the FPGA or on the host CPU.

In addition to the predicate level parallelism achieved by employing multiple concurrent predicate elements, the FPGA design also provides row level parallelism by using multiple instantiations of the row scanner. Each row scanner evaluates one row at a time and operates independently of the rest. Our current design instantiates sixteen row scanners, each with 64 PEs, although the number of row scanners and the number of PEs within each can be configured at synthesis time based on the available FPGA resources.

5.2 Column Projection Accelerator

5.2.1 Overview

Though offloading column projection to the FPGA by itself may not be cost effective, it is beneficial for various reasons to include projection in the accelerator's query processing flow while offloading other query tasks such as predicate and sort. First, it formats the data in a way that the end application requires, thus relieving the host processor of this post-processing task. Second, as will be discussed later in this section, in our FPGA design, projection is performed in parallel with predicate evaluation, thus offloading more computations without adding any latency or affecting the throughput of query processing. Third, projection on FPGA provides bandwidth and storage savings by removing unwanted columns from each record to be reported back or to be temporarily stored locally on the FPGA card during sorting. Besides, projection

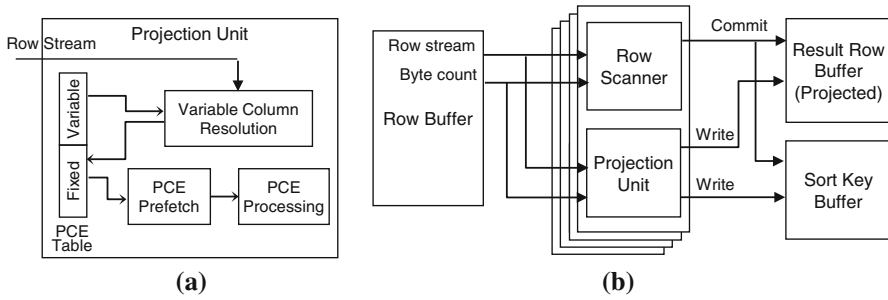


Fig. 5 **a** Projection logic, **b** integration of projection unit with query processing pipeline

is required to extract the columns that form the key for sorting the records, making projection a prerequisite step for the sort operation.

While doing projection on the FPGA has clear advantages for the above reasons, one could argue that these benefits fade for queries that only require sorting of the records based on sort keys. Although projecting the columns on the CPU and sending only the sort keys to the FPGA can provide bandwidth savings in such cases, it also burdens the host with the task of sifting through the database table to extract the sort keys, linking the keys to the original records using a unique tag and then sending the (key, tag) tuples to the FPGA. Furthermore, randomly accessing the records in CPU's memory using the sorted tags returned by the FPGA can result in huge cache miss penalties. These costs can easily overshadow the performance improvements obtained from accelerating the sort operation. Projection on the FPGA would thus be beneficial even for sort-only queries.

5.2.2 Implementation of Projection Logic

Figure 5 shows the block diagram of projection logic and its integration with predicate evaluation flow on FPGA. Each instance of row scanner for predicate evaluation is paired with a projection unit. As the rows are evaluated against the query predicates, they are simultaneously projected in the projection unit, which captures the required columns one byte at a time and forwards to the projected row buffer and the sort key buffer. For each byte streaming through it, the projection unit uses the projection control elements from the PCB to determine whether the byte is to be projected and generates the appropriate write addresses and write enables for the buffers. Once the row is fully streamed through the row scanner, depending on whether it qualifies the predicates, the projected columns are either committed in the buffers or the writes are rolled back by resetting the write pointers.

Projecting fixed length columns To process the streaming rows without stalls, the projection unit must be able to identify the bytes to be projected in a single streaming pass. The use of PCEs to configure the projection units with the required columns' starting byte position in the row and their length enables the projection units to easily determine if a particular byte needs to be projected simply by comparing the row byte

count against the required column's position and length. During soft configuration of the FPGA, the PCEs for the columns to be projected are stored into a PCE table in FPGA memory, (block RAM) in the order in which these columns appear in the input row. During the row streaming phase, the projection unit iterates over these PCEs to capture the required bytes for projection using the column start and length fields in the PCE. The buffer addresses for the captured bytes are generated using the column destination from the PCE. By storing the PCEs according to the order of those columns in the input row, only one PCE needs to be loaded and compared at a time, allowing for very hardware-efficient implementation of the projection logic. A PCE prefetch logic is used to avoid stalls and load a new PCE every cycle in the case of projecting multiple consecutive 1-byte columns (Fig. 5a).

Projecting variable length columns Specifying the position and length information in the PCEs during soft configuration requires that such information remains fixed across different rows and be known a priori. While this is true for fixed length columns, database rows often contain variable length columns (e.g. strings) whose length and position vary from row to row. The starting positions (offsets) of the variable length columns in a row are embedded in the row itself while the lengths must be deduced using the offsets. Projecting variable length columns thus requires extracting and processing the offsets for each variable length column to be projected, on a per row basis. Moreover, as shown in Fig. 2, all the column offsets appear together between the actual fixed length columns and the variable columns, thus requiring access to different parts of the row while projecting a variable length column. This makes it difficult to project such columns in a purely streaming fashion.

One way to handle the projection of variable length columns is to use a two-pass streaming approach, where the first pass through the row solely processes the variable offsets to compute the specific column information and the second pass projects the required columns. An alternative approach is to stage the row in a buffer and access different pieces of the row as necessary in random order. None of these two approaches, however, are desirable. While the first approach essentially reduces the row processing throughput by half, the second approach breaks the streaming model and affects the predicate evaluation logic. We employ a hybrid approach that is similar to the second option of staging the row but also maintains the streaming model for predicate evaluation. We divide the processing of each row into two phases. The first phase, called the *PCE resolution phase*, converts the variable column PCEs into fixed PCEs while the second phase streams the row through the predicate evaluation and projection units, skipping over the already processed columns offsets to reclaim the cycles spent in the resolution step, thereby maintaining the streaming throughput.

To facilitate PCE resolution, we divide the PCE table on FPGA into two sections (see Fig. 6). During soft configuration of the FPGA, the PCEs for the fixed length columns are stored in the lower half of the table, in the order in which these columns appear in the input row, and those for variable length columns are stored in the top half, in the order in which the variable columns must appear in the projected row. Storing the variable PCEs in this order helps us determine the destination position of these columns in the projected row. Just like the input rows, the positions of the variable

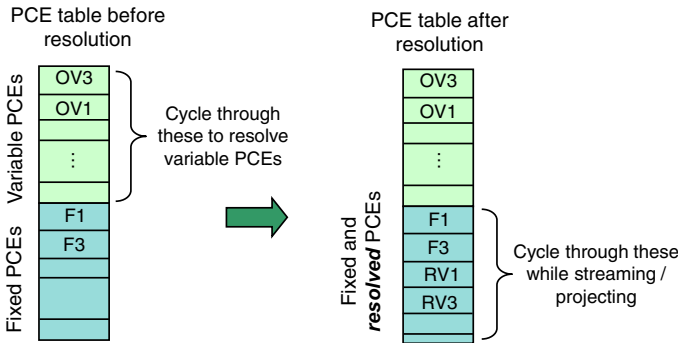


Fig. 6 PCE table before and after variable PCE resolution

columns in the projected row also vary from row to row and must be computed on a per row basis.

During the PCE resolution phase, the row is staged into a temporary buffer and the resolution logic iterates through the variable PCEs to convert them into fixed PCEs. As stated earlier, a PCE for a variable length column contains a pointer to the column’s offset in the row. This position is always fixed irrespective of the length of the variable column. For each variable PCE, the offset pointer is used to read the offsets of the current and the next variable column. The current offset provides the column start position while the length is computed as a difference of the two offsets. The length of the last variable column is computed using the offset and the row length stored in row header. Finally, the destination position of the column in the projected row is computed by using the ordering of the variable PCEs in the PCE table, which is the order in which they appear in the projected row. The destination of the first variable column is immediately after the last variable offset and this position is fixed. For subsequent variable columns, the destination is computed as a sum of the starting position and the length of the previous variable column. The computed column start, length and destination are written in a resolved PCE. The column destination positions are also written into the projected row buffer as the variable column offsets for the projected columns. The resolved PCE is then stored in the lower half of the PCE table below the fixed length PCEs, in the order in which they appear in the input row.

Once all the variable PCEs are resolved, the start and length of each column, including the variable length columns, are known for the current row and the row is streamed over the projection unit. The projection unit iterates over the PCEs in the bottom half of the PCE table and projects the corresponding rows, treating each row as a fixed length row. After the row has been fully streamed, the projected row buffer contains the row and the key buffer holds the sort key. The sort key is copied into the projected row to completely format the projected row as required. This step is overlapped with the PCE resolution step for the next row.

Note that during PCE resolution, the row stream is halted and, depending on the number of variable PCEs to be resolved, this can add significant latency to the processing of each row. Specifically, resolving p_{var} PCEs adds $p_{var} + 4$ extra cycles to the processing of each row, where 4 is the depth of the resolution pipeline. To avoid this

Table 1 Throughput versus size comparison of various hardware sort algorithms

Algorithm	Throughput (keys per cycle)	Number of comparators	Comparators normalized by throughput
Bubble/insertion sorting networks	N	N^2	N
Even-odd/bitonic sorting networks	N	$N \log^2(N)$	$\log^2(N)$
Linear (insertion) sort	1	N	N
Radix sort	Not comparable since this is a multi-pass sort with smaller comparators		
Tournament tree sort	$1/\log(N)$	1	$\log(N) + (\log(N)-1)^{\#}$

[#] Second term for a $\log(N)$ -way merge tree

overhead, we skip all the variable column offsets while streaming the row, since the offsets are not needed for projection or predicate evaluation. Skipping n_{var} 2-byte offsets reclaims $2 * n_{\text{var}}$ cycles, where n_{var} = the number of variable columns in the row ($n_{\text{var}} \geq p_{\text{var}}$). For $p_{\text{var}} \geq 4$, the savings from skipping the offsets more than compensates for the cycles spent for resolving the variable PCEs.

5.3 Sort Accelerator

5.3.1 Overview

A variety of sort algorithms exist and the main considerations while choosing an algorithm for hardware implementation are area requirements, throughput (number of keys sorted in a give time), sorted run size (number of keys to be sorted) and the size of the sort key. The right algorithm may vary based on the requirements of the application. In database applications, sort keys often tend to be very large (tens of bytes or longer), thus requiring large comparators and hence large area resources for each comparator used in the design. Moreover, database tables may contain millions or even billions of records, thus requiring large sorted runs. Additionally, database sort presents two other challenges: the requirement to handle the large payload associated with each key, and the requirement to perform sorting in a streaming fashion. The latter requirement means that all the keys to be sorted might not be available at the same time. Instead, keys might arrive at regular intervals, due to either some pre-processing step or IO or memory limitations. The sort engine, thus, must be able to receive input keys and produce sorted keys in a streaming fashion.

We have evaluated various sort algorithms (see [3] for details) and find tournament tree sort [26] to be well suited to database applications as it supports streaming sort and can generate large sorted runs while requiring the least number of comparators to attain a certain throughput (see Table 1).

Tournament tree sort is a binary-tree based selection and replacement sort algorithm. Keys are entered into the leaf nodes and they exit, in sorted order, from the tree's root node. During the initial tree population phase, one input key is inserted into each leaf node. Pair-wise comparisons are then performed at each tree level. The losers

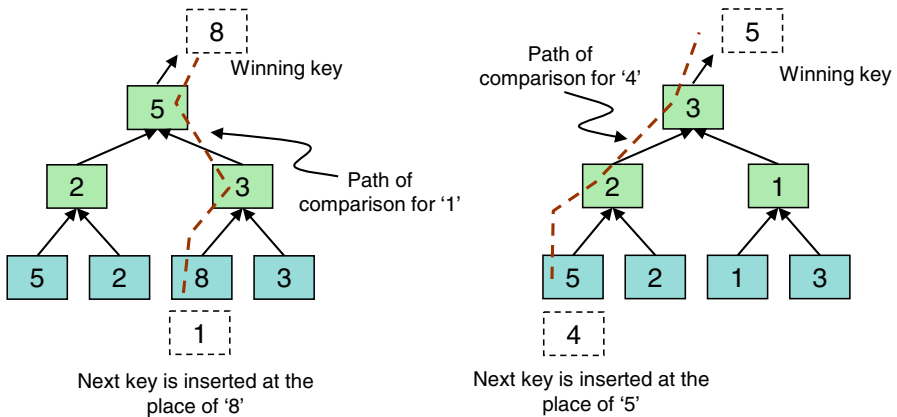


Fig. 7 Tournament tree sort algorithm with four leaf nodes. Tree nodes before and after inserting a new key (1) at the position of the winning key from the previous iteration (8)

are propagated to the next level while the winners are further compared against each other to determine the next higher level (Fig. 7). This is repeated until the tree is fully populated and a winning key emerges at the tree root. At this point, each non-leaf node holds exactly one input key and the tree enters the continuous streaming phase. During streaming, each new input key is inserted at the previous winner’s leaf node and is compared (and swapped, if needed) with nodes along the path of the previous winner (see Fig. 7). In other words, it is compared with exactly one non-leaf node at each level, until the tree root is reached and the next winner is determined, at which point, a new key can be inserted into the tree. For efficient implementation of the tree, it is a common practice to store the keys only in the leaf nodes while the non-leaf nodes store just the pointers to their corresponding leaf nodes.

We chose the tournament tree sort algorithm for database sort implementation on FPGA. To the best of our knowledge, our FPGA implementation is the first hardware implementation of tournament tree sort. Previously, it has been implemented programmatically in IBM System z [27,28], utilizing assist instructions which implement certain functions used in evaluating the tournament tree. Our sort engine is designed to produce minimum sorted runs of thirty two thousand rows, sorting the rows at full PCIe bus bandwidth, while using only a single key comparator, making it very hardware efficient. With minimum run sizes of 32K, sorting a table with a million rows requires a 32-way merge, while smaller tables can be sorted without any further merging. Depending on the available FPGA resources and the required sorting throughput, the 32K sorter can be implemented as a single tree or multiple smaller trees followed by a merge stage. The first option can generate a 32K sorted run using a single comparator, albeit at a throughput lower than the second option of multiple parallel trees. Consider the case of a 200 byte database row. At a PCIe rate of 16B @ 250 MHz, on average, one row is delivered to the FPGA every $200/16 = 12.5$ cycles. In the absence of any predicate filtering, the sort logic must consume a new key every 12.5 cycles in order to sustain full PCIe bandwidth. Since a tree with N nodes can accept a key every $\log_2(N)$ cycles, a single 32K node sort tree, requiring 15 cycles

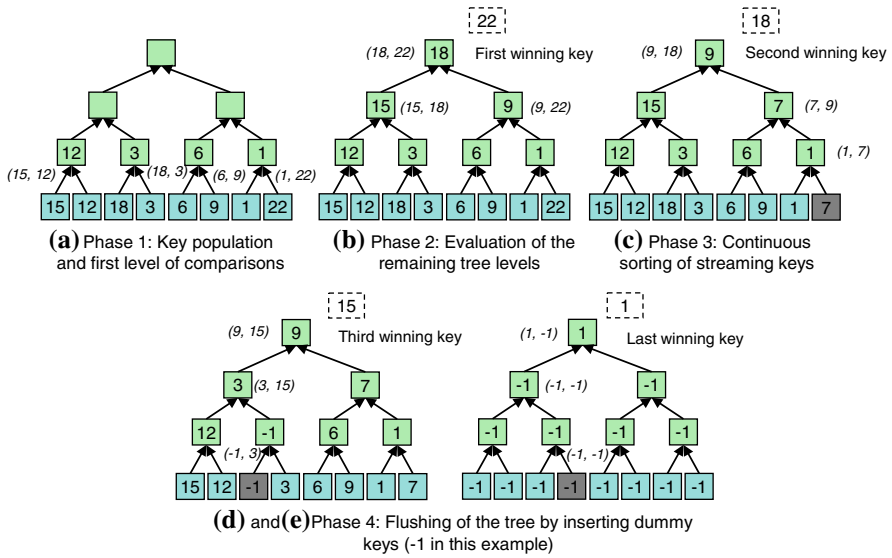


Fig. 8 Different phases of tournament tree evaluation on FPGA (shown for an example tree with 8 leaf nodes and with 9 keys to be sorted). Comparisons being evaluated during each phase are shown with parenthesis next to the corresponding nodes

between two key inserts, cannot support sorting at bus rate. We thus split the tree into two smaller trees, followed by a 2-to-1 merger unit. Each tree consists of 16K leaf nodes and accepts a new key every 14 cycles. Compared to a single comparator of a one-tree design, this design uses three comparators, one for each of the trees and one for the merger, and supports an aggregate sorting throughput of one key every seven cycles. Rows from the predicate evaluation logic are alternately sent to the two trees. For the 200 byte row example, at most one row gets sent to a tree every 23 cycles. In general, for rows larger than 120 bytes, our design can sort the rows without stalling the input stream. In typical queries, rows are usually much longer than that and a large number of rows get filtered out by the predicate evaluation logic. Our two tree configuration would, thus, sustain full bandwidth, except for the cases of very small rows with high qualification ratios. For such cases, further splitting of the trees can be applied and multiple trees can even be implemented on different FPGAs, as discussed later.

5.3.2 Implementation of Tournament Tree Sort

Tournament tree sort implementation on FPGA works in four phases (see Fig. 8). The first phase is the initial population of the keys into the leaf nodes. Since the keys are inserted serially and all the comparisons are made using a single comparator, pipelining the comparisons for all the tree levels during the insertion phase requires complex bookkeeping and scheduling logic to keep track of the winners at different levels and perform pair-wise comparisons in the correct order in a partially-filled tree. We, instead, evaluate only the first level of comparisons during the initial insertion

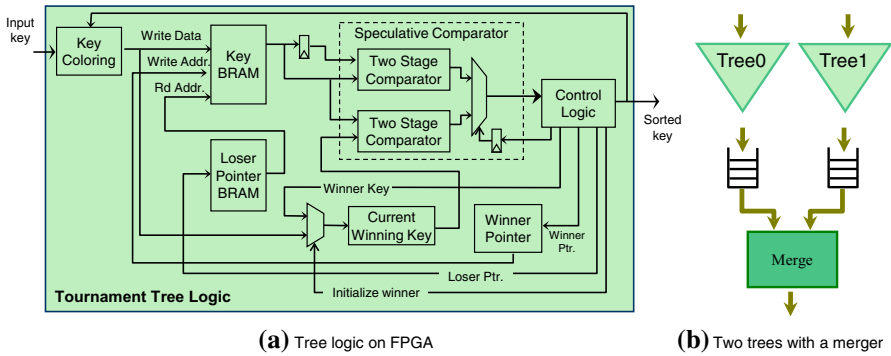


Fig. 9 Tournament tree sort implementation on FPGA

phase to reduce the complexity of the control logic. The second phase then evaluates the remaining levels of the tree and performs initial sorting. The overhead of this extra phase is minimal. Since all the leaf nodes are populated at this time and since the comparisons are being done across one level of the tree at a time, this process can be easily pipelined, with one comparison being performed every cycle. This phase thus adds only about 1 % overhead in the worst case, and even less when the sort run sizes are larger than the tree size.

At the end of the second phase (Fig. 8b), the first sorted key (winner) exits the tree and its position in the leaf node is recorded. The third phase is the continuous sorting phase where each new incoming key is inserted into the leaf node of the previous winner and is initialized as the winner. It is then walked up the tree following the path of the previous winner, comparing with the value at the internal nodes and either keeping the current winner or swapping with the key at that node. Upon reaching the root node, the current outstanding winner is emitted as the next sorted key. Once all the keys have been streamed, the tree enters the fourth and final phase to flush the tree by inserting dummy keys. The dummy keys have the smallest/largest value and hence they always lose in the comparisons and do not exit the tree until all the remaining valid keys have been emitted.

Figure 9 shows the implementation of tournament tree sort on FPGA. There are three main challenges in realizing an efficient implementation of tournament sort in hardware. The first is to allow continuous sorting beyond the minimum guaranteed batch size in the presence of keys that violate the sort order i.e. are larger/smaller (for descending/ascending sort) than the keys already sorted. In the naïve approach, the arrival of such a key would require flushing all the keys from the tree and starting a new batch from an empty tree. This approach thus incurs extra tree setup and teardown cost, significantly reducing the sorting throughput.

To overcome this extra overhead and continue sorting even with violating keys, our approach is to *color* the keys before sorting, i.e. attach a prefix to each key, thereby achieving implicit, on-the-fly binning of the keys into different sorted runs. Coloring starts from the first color (largest value for descending sort/ smallest for ascending) and the color is decremented/incremented to the next value for each new run. Each incoming key is compared against the last sorted key. If it violates the sort order, it

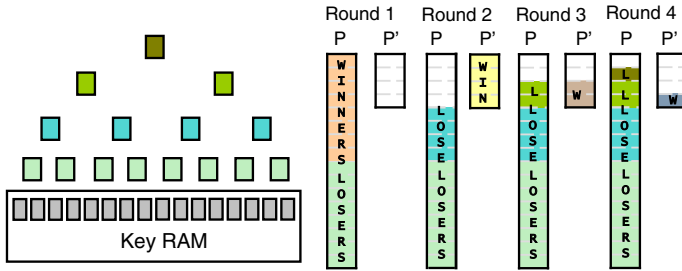


Fig. 10 Tournament tree pointer structure on FPGA

is colored using the next color value, otherwise the current color is used. Due to a smaller/larger color prefix, keys colored with the next color always lose against all the keys of the current color, and thus never get emitted in the current sorted batch. Coloring thus allows the violating keys to participate in the tree, thereby eliminating the need to drain the tree between sorted runs. It effectively enables us to use the violating key and those after that to flush the current tree, without having to flush the tree explicitly using the dummy keys. In our current implementation, we use three bits for the color, supporting seven sorted runs before it is necessary to explicitly drain the tree. The lowest (highest if ascending) color is used as the dummy key to flush the tree with dummy keys. In the absence of coloring, draining a tree with N leaf nodes after each run would require inserting N extra dummy keys, potentially doubling the sorting time for each run.

The second issue is the memory-efficient implementation of the tree structure. A tournament tree implementation utilizes FPGA block RAMs for storing the keys in the leaf nodes and for pointers in the non-leaf nodes. To support large trees with very long sort keys, as well as the buffering of large database pages throughout the pipeline, our design makes heavy utilization of on-chip memory and requires careful use of the block RAMs. In the naïve implementation, a tree with N nodes requires storage for N loser pointers in the tree, $N/2$ winner pointers (for further comparisons) and $N/4$ of secondary pointer storage (for alternating while evaluating the subsequent levels). In our implementation, we reuse the top half of the loser pointer array to store the first level of winners and the top half of the secondary storage for alternating, thus obviating the need for the $N/2$ array. This results in about 29% storage savings for the internal nodes.

Figure 10 shows the structure of our implementation for an example tree size of $N=16$ nodes, with a loser pointer array P (of size N) and a secondary pointer array P' (of size $N/4$). As the initial N keys are inserted, pair-wise comparisons are made and the $N/2$ loser pointers are inserted in the bottom half of P while the top half is used to temporarily store the $N/2$ winner pointers (round 1). The next level of the tree is evaluated by pair-wise comparing the winners stored in array P , walking from bottom to top of the winner section. The loser pointers are placed back in array P , starting from the middle of P , while the winners are stored starting at the bottom of array P' , both moving upwards. Note that the space in array P can be reused for the loser pointers because for each loser pointer to be stored, a pair of winner pointers has

already been consumed. In contrast, writing into the upper quarter of array P would overwrite the pointers to the data being sorted. After examining all the winners in array P, the contents of the RAM are shown as round 2. Fully evaluating all the tree levels requires two more rounds of comparison from array P', eventually generating the winner.

For traversing the tree during subsequent key insertions, the internal nodes can be easily accessed in array P through a simple arithmetic operation on the index of the current position, yielding the index of the parent of the node being examined. As a new key gets inserted at the position of the last winner, it is walked up the tree by comparing it against the internal nodes and updating the array P as needed.

The last implementation challenge is to achieve high operating frequency with single-cycle operation of large key comparators. Once the tree is fully populated, the comparisons at different tree levels to emit the next sorted key cannot be pipelined as the comparison at each level depends on the outcome of the previous level. Single cycle comparisons are thus essential to achieve the target throughput of sorting one key every $\log(N)$ cycles. While a purely combinational comparator for small keys can achieve small cycle time, comparing two forty byte keys using purely combinational logic results in very low operating frequencies. To overcome this issue, we implement a 2-stage comparator that performs speculative comparisons to allow the process to be pipelined. At each tree level, two comparisons are performed in parallel, one with each of the two inputs to the comparison at the previous level. The result from one of these comparisons is chosen based on the winner of the previous level, much like a carry-select adder.

5.3.3 Scalable Tournament Tree Sort

Our implementation of tournament tree sort can be tuned for capacity and throughput by replicating the tree or by creating larger trees, as noted in the previous section. The size of the individual tree and the number of trees that can fit on a given FPGA, however, is limited by the amount of resources available on a single chip. There are various ways to scale the design to sort sizes larger than what can fit on a single FPGA. One approach is to utilize the on-card DRAM present on most commodity FPGA cards to perform multi-phase sort. The intermediate sorted batches generated by the FPGA can be stored in DRAM. A final merge phase on the FPGA can then merge the different batches to generate a single sorted batch.

Another approach is to directly implement the entire tree in DRAM, thus easing the constraints on the size of the tree that can fit in FPGA memory. This approach utilizes the DRAM for storing the keys at different levels of the tree. In this scheme, storing the keys directly in the tree rather than using pointers to the keys eliminates a level of indirection which would otherwise require two memory accesses to fetch a key. For each new key, $\log(N)$ DRAM references are required for comparison at different levels of the tree. Since the path to be evaluated for each new key is known as soon as the previous winning key has been determined, the keys for comparison can be prefetched. Note that since the keys are now stored in the tree, any swap of keys during the evaluation must be written back to the on-card DRAM to keep the tree consistent.

With both of the above approaches that utilize off-chip DRAM, given a moderate sized FPGA, the maximum number of keys is limited only by the size of the on-card DRAM and not by the FPGA resources. Large sort runs and/or higher sorting throughput can also be achieved by composing the sort operation as a multi-phase sort utilizing multiple FPGAs. There are various ways to utilize the coarse level parallelism of multiple accelerators, with the simplest being the host CPU dividing the input data across different accelerators, followed by a final merge phase on the host. As described in Sect. 6, our DBMS integration system supports multi-phase sort with merge on CPU using either a single FPGA or multiple FPGAs. This approach, however, requires explicit partitioning of the input data set.

An alternate approach is to create a multi-tree tournament sort spanning multiple FPGAs and merge the outputs of the trees from different FPGAs on one of the FPGAs. In this case, the data gets implicitly distributed to the trees on the different accelerators without explicit data partitioning on the host. A number of COTS multi-FPGA PCIe cards [29] provide FPGA chips connected to an on-card PCIe switch that provides connectivity between all of the FPGAs and the host. The communication between the FPGA can be either handled by the runtime system provided by the vendor or via logic implemented as part of the design. Note that a multi-FPGA sort tree implementation is feasible on such platforms since the different trees do not need to communicate on a fine granularity. The partial results generated by each tree can be batched and transferred to the merge FPGA via the PCIe switch on a block level, which then finally pushes blocks of sorted data back to the host. The resulting multi-FPGA tournament tree sort provides a $T/\log(N/T)$ throughput, where T is the number of parallel trees, each with N nodes, with some additional overhead due to data transfer between the FPGAs.

5.4 Query Processing Flow on FPGA

Figure 11 shows the query processing flow on the FPGA fusing the different query operation accelerators described in the previous sections. Different accelerator kernels are connected together in a feed-forward pipeline. Though the interconnection between these kernels is fixed, query-specific accelerator can be composed by invoking or bypassing each of those kernels. The operations to be invoked are specified in the software-generated query control block. The FPGA parses the query control block to soft-configure the required accelerator kernels. After soft-configuration, the FPGA uses the DMA addresses specified in the QCB to copy the data pages to be processed from the host memory to the FPGA. The database rows are extracted from the pages and streamed to the predicate evaluation and projection kernels. Qualified rows exiting the projection unit contain the sort key as well as the projected columns that need to be reported. Although the key must be held in the FPGA BRAMs for performance, keeping the entire row on chip during sorting would severely limit the number of rows that could be sorted at one time. We therefore store the rows in on-card DRAM while the keys are being sorted. The data flow bifurcates as the projected row exits the projection logic. The entire record (payload) is stored on the on-card DRAM, while the key, along with a tag, is sent to the sort logic. The tag consists of the record's address

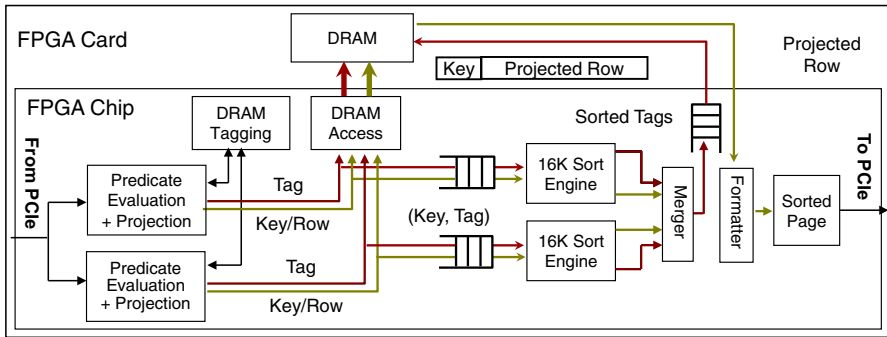


Fig. 11 Query processing pipeline on FPGA

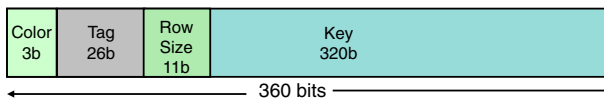


Fig. 12 Composite sort key

in DRAM and its size, and is used to retrieve the record when its corresponding key exits the sort logic. A DRAM tagging unit is shared between multiple instances of the projection logic to ensure the generation of a unique tag for each row. The structure of the composite key is shown in Fig. 12.

As discussed earlier, our design uses two independent sort trees (Fig. 9b). Keys are alternately sent to the two trees and the two sorted key streams are merged by the merger. Once the merger emits the next key in the sequence, the key value is no longer required and is discarded; only the tag is carried forward for accessing the corresponding row. It is, however, necessary to carry forward the color since the rows with different colored keys must be separated into different sorted runs and placed on separate output pages. The page formatter uses the sorted tags to retrieve the rows from the on-card DRAM and writes them in database-formatted output pages. When a color change happens, or when a page gets filled, the page gets scheduled to be written back to host memory via a DMA operation and the page formatter switches to a new page. Since parts of the pipeline may experience variable delay, buffering is used extensively throughout the design. Additionally, back-pressure is implemented to stall the pipeline all the way back to the fetching of the input pages from the host, in the event of buffer overflow.

6 Integration with DBMS

To maximize the benefit of acceleration, it is important to avoid introducing any extra overhead during the query flow. Our FPGA accelerated query operations are seamlessly integrated in the query flow as shown in Fig. 13. An FPGA-initiated DMA transfer operation [1] fetches data pages from the buffer pool into the FPGA for operations such as decompression, projection and predicate evaluation (although these are not

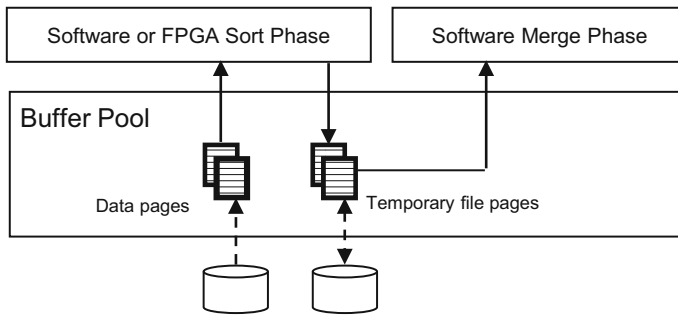


Fig. 13 Query flow integration of FPGA sort

shown in Fig. 13), before feeding data to FPGA sort. The sort run results are directly written, via DMA transfers, to temporary files (pages) on the host. For a large query sort with both sort and merge phases, these DBMS-managed temporary files store the intermediate results between the two phases. Putting FPGA sort results in these temporary files allows a second software or hardware merge operation to use them as input for merge seamlessly. In this flow, the main difference introduced on the DBMS software side for FPGA acceleration is a block level data operation, as opposed to the typical DBMS row-by-row pipelined operations. Interacting with the FPGA to process one row at a time incurs significant overhead. In the block level FPGA-accelerated query operation, a number of data pages and temporary file pages are provided to the FPGA as input and output for each invocation to minimize any possible performance overhead.

For the case when a query cannot be mapped to the FPGA due to hardware limitations (e.g., number of processing elements) we can restructure the query operation by decomposing it into sub-operations which can be mapped to hardware and evaluate these sub-operations on the hardware. Different rules/algorithms can be developed for determining the right order of evaluating these sub-queries in hardware and combining their partial results so as to obtain the same effect as that of the original query operation. Control blocks corresponding to the query operations carry the query operation ID in the sub-operations of the query to facilitate their orchestration on (possibly multiple) hardware and for right combination of their results.

One way to decompose a query operation into sub-operations is through horizontal and/or vertical partitioning of the FPGA blueprint of the query operation. Our example query predicate from Sect. 4.3 can map, without restructuring, to a FPGA predicate network with eight PEs. Consider a scenario where we have a predicate network with only four PEs on the FPGA. In that case, we can vertically partition the predicate network into two sub-predicates and process them sequentially on the same hardware in two different passes. Our QCB enables soft-programming of the FPGAs for the two passes (as shown in Fig. 14), with the intermediate results from the first pass (qualified rows) being stored either in the FPGA BRAM or in the host memory for further processing. For cases where the reduction operation linking the two partitions is a logical conjunction (*AND*), as shown in the example in Fig. 14, the database rows qualified by the first pass can simply be fed into the sec-

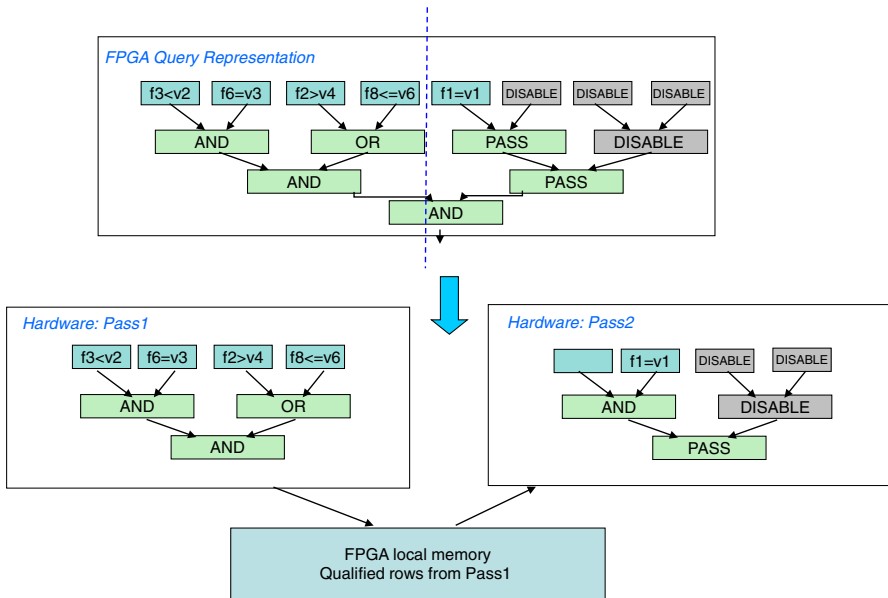


Fig. 14 Query restructuring through vertical partitioning and its execution

ond pass. Depending on the size of the data, the qualified rows from the first pass can be held in local accelerator memory or returned to the host CPU and streamed in again.

For logical disjunction (*OR*), however, simply feeding the output of the first pass into the second pass may result in useful rows being filtered away while re-scanning the original dataset may result in duplicates. There is more than one option to handle this. The first is to pass the entire data set to each partitioned predicate network, one after the other, and perform the merge operation in either host software or FPGA. Each row that qualifies either pass needs to be tagged to uniquely identify the row in the input data set so that the merge operation can remove the duplicates from the final result. The tagged identifier can be from the input row, such as the row ID, or added by the FPGA. A second, more efficient option is to separately save the non-qualifying rows from the first pass and only feed those to the second pass. Again, depending upon the amount of available accelerator memory and the size of the dataset, these non-qualifying rows may be held locally on the accelerator or returned to the host. The final query result is a union of the two sets of qualified rows from the two passes and contains no duplicates.

Overcoming the hardware limitations for sort is different from that of predicates. As discussed earlier, statistically a smaller tournament tree generates runs with less number of sorted rows. For very large input datasets generating multiple sorted runs, each run can be stored in a buffer in the host memory. A software merge function is performed on these runs to generate the final sorted result. The same approach can also be extended to a scalable sort solution when there are multiple FPGAs available and attached to the host to handle large data processing, as

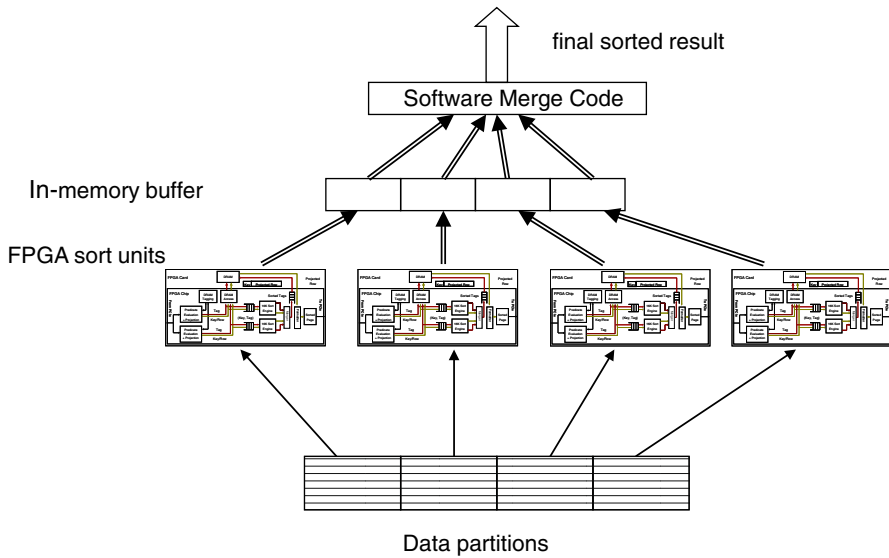


Fig. 15 Integrating scalable sort with multiple FPGAs

shown in Fig. 15. The input data can be partitioned before feeding into a number of FPGA sort units that operate concurrently. The sorted output runs from each FPGA sort unit are stored in host memory before merged by the software merge function.

7 Experiments and Results

We now present the resource requirements for query acceleration on FPGA, followed by the performance results for our FPGA-accelerated database for various queries involving predicate evaluation, projection and sort operations. FPGA performance results are compared against the baseline software measurements taken on a commercial DBMS running on a server with 3.8-GHz multicore superscalar processor and 32 GB of memory. Our target FPGA system is a PCIe gen2 x8 card with an Altera Stratix 5 GXMA7 FPGA and two banks of 4 GB DDR3 @ 666 MHz, each with a peak bandwidth of 10.6 GB/s to the FPGA; our prototype uses only one of these banks. Our design runs at 250 MHz and the FPGA results are post place-and-route throughput estimates and include the data transfer overheads. Both software and FPGA measurements are taken with the data loaded in memory and do not include disk IO.

7.1 FPGA Resource Utilization

Table 2 lists the resource requirements for the acceleration kernels for different query operations and the overall query acceleration pipeline with multiple kernels on the FPGA. As stated earlier, the use of the projection control elements results in very

Table 2 FPGA resource requirements

Module	ALUTs	Registers	M20K BRAM
Projection unit	699 (<1 %)	1,157 (<1 %)	4 (<1 %)
Row scanner	8,780 (2 %)	11,380 (2 %)	-
Tournament tree sort	4,259 (1 %)	2,813 (<1 %)	308 (12 %)
Predicate + projection + sort*	160,801 (34 %)	201,207 (22 %)	847 (33 %)

* 16 row scanners + 16 projection units + two 16K sort trees + merger

efficient implementation of the projection logic. A single projection unit consumes less than 1 % of the FPGA resources while the row scanner consumes somewhat higher resources due to the large number of 8-byte comparators for concurrent evaluation of 64 predicates. Our design replicates 16 instances of predicate and projection units to match the PCIe data rate.

For the tournament tree sort engine, the use of a single comparator keeps the logic utilization very low (to just about 1 %), while the BRAM utilization is high as it is proportional to the size of the tree. Table 2 also lists the total resource requirements for one configuration of our query processing pipeline on FPGA, with 16 parallel instances of predicate evaluation and projection units and two 16K sort trees. This design consumes a total of 34 % logic and 33 % BRAMs.

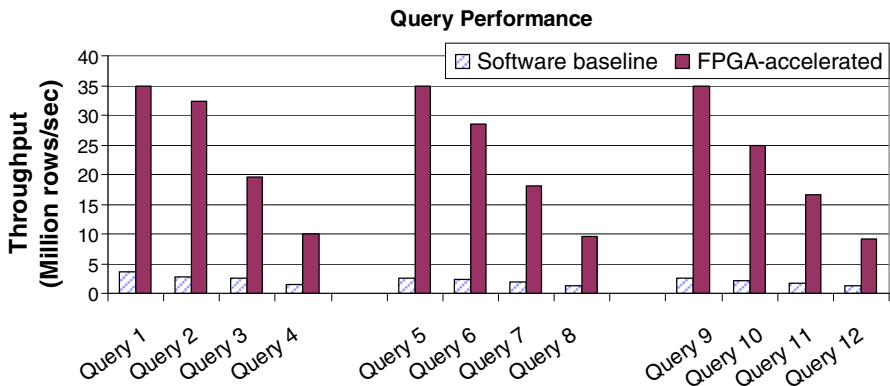
7.2 Performance of Projection-Only Queries

Cost of projection on the CPU can vary from very low to very high depending on the data characteristics and workloads. The factors that impact the performance include the row length, the number of projection columns and their widths and whether the projection columns are fixed or variable length. Moreover, from a cache locality prospective, the relative location of projection columns within a row also impacts the operation's CPU cost. Our experiments indicate, as expected, that projecting multiple columns from a single cache line is cheaper than extracting more than one cache line for wider rows, especially in the case of variable length columns whose location is data driven.

Unlike in software, the performance of our projection implementation on FPGA is not sensitive to factors such as the type and the number of columns to be projected, whether fixed or variable length column. This is because the required columns get projected as the row streams through the FPGA, without requiring any random access through into the row. Projection on FPGA in effect neither adds any latency to the query processing on FPGA nor affects the overall throughput, making the projection operation essentially free on the FPGA. Hence, providing a speedup number for just the projection operation is not appropriate. In terms of the absolute cost of projection on the CPU, for a 169 byte row with 39 columns, we measured the cost for projecting fixed length columns to be between 53 to 159 processor cycles per row. This cost may increase to over a thousand cycles for projecting variable length columns due to potential cache miss penalties while reading from a variable offset of a large row spanning multiple cache lines.

Table 3 Test queries with varying key and payload sizes

Key size	Payload size			
	40	120	200	400
4	Query 1	Query 2	Query 3	Query 4
20	Query 5	Query 6	Query 7	Query 8
40	Query 9	Query 10	Query 11	Query 12

**Fig. 16** Query throughput comparison

7.3 Performance of Predicate and Sort Queries

We now compare the performance of our FPGA-accelerated database against the baseline commercial DBMS while executing queries involving predicate, projection and sort operations. Our targeted baseline database also implements the sort operation using tournament tree sort algorithm.

Other than the computational complexity of the algorithm, a significant factor that impacts the performance of the queries involving the sort operation is the cost of data movement of both key and payload data in the storage hierarchy of the system. Based on the analysis of a typical customer workload, we constructed test queries to ensure that the FPGA sort design is evaluated against realistic scenarios. As shown in Table 3, we exercised 12 queries with key lengths ranging from 4 to 40 B and payload data sizes from 40 to 400 B. Both keys and payloads contain fixed and variable length columns. Our test queries are derived from real customer workloads and resemble the TPC-H Q1 and TPC-DS template three queries.

Figure 16 compares the performance of the FPGA-accelerated queries with the baseline for the above 12 queries for uncompressed rows. For queries 1, 5 and 9, where the combined key size and payload size is less than 120 B, the FPGA throughput is limited by the sort operation. In other words, rows can be delivered to the FPGA over the PCIe bus faster than they can be processed by the sort kernel. For the remaining queries, the key+payload size is larger than 120 B and the throughput is limited by the PCIe bandwidth, with the throughput decreasing with the increase in payload size.

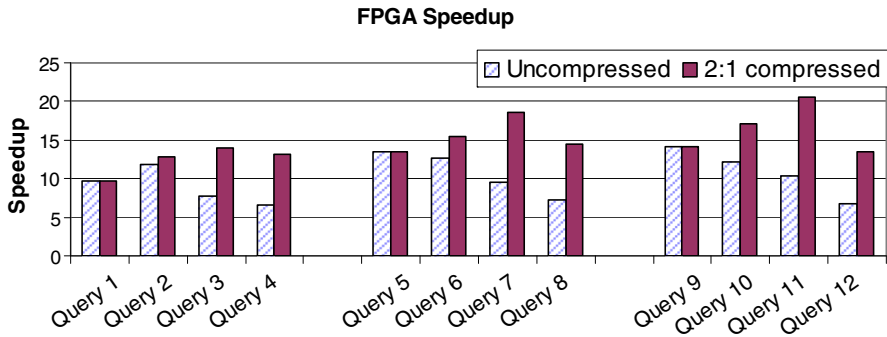


Fig. 17 FPGA speedup for uncompressed and compressed data

Thus the performance of these queries is limited not by the sort logic but by the rate at which the rows can be fed into the FPGA over PCIe.

Figure 17 plots the FPGA speedup over baseline sort for uncompressed rows and rows with 2:1 compression. Our target database uses a variant of Lempel-Ziv compression algorithm with 32 KB dictionary. For uncompressed data, the speedup ranges from $6.5\times$ to $14\times$, while the compressed payload achieves speedup in the range of $10\times$ to $21\times$. As before, for queries 1, 5 and 9, the speedup is limited by the sort trees and having compressed data does not provide any improvement. For the remaining queries, compressed data results in higher speedup compared to uncompressed data, though the gain is query-dependent. For queries 2, 6 and 10, having compressed data alleviates the PCIe limitations, making the sort logic the limiting factor. For queries with larger payloads (queries 3, 4, 7, 8, 11, and 12), however, the sort trees can still process the rows at the rate they are delivered by the PCIe. Datasets with higher compression ratios can thus result even in larger speedups for these queries.

8 Conclusion

We have presented an FPGA-accelerated database system with acceleration kernels for various query operations such as predicate evaluation, sort and projection, including design options, implementation details, and the DBMS integration aspects. We have further presented a query transformation system that automatically maps these query operations, with query restructuring if necessary, onto the FPGA accelerator for seamless exploitation of the FPGA kernels from a standard database system. The design allows fusing together different accelerated kernels for composing query-specific accelerators to support the offload of a wide variety of queries without reconfiguring the FPGA. The performance results demonstrate significant speedup on a wide variety of queries with sort, predicate and projection operations.

The proposed accelerator kernels for different database operations such as predicate and sort, as well as the query transformation routines, are database agnostic and the query control block structure provides a well defined interface between database software and the FPGA accelerator. This allows for easy portability of the current accelerator to other database management systems. Different DBMS', however, typ-

ically use different database page formats and porting to a new DBMS may require some change in the page parsing logic on FPGA. Our logic for projecting variable length columns, on the other hand, is tightly coupled to the representation of variable length columns in our target database and may require some change to handle variable length columns that are formatted differently. Our modular accelerator design, however, keeps the required changes localized without affecting the rest of the design.

References

1. Sukhwani, B., et al.: Database analytics acceleration using FPGAs. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 411–420 (2012)
2. Halstead, R., et al.: Accelerating join operation for relational databases with FPGAs. In: IEEE CS, Proc. IEEE Symp. FCCM, pp. 17–20 (2013)
3. Sukhwani, B., et al.: Large payload streaming database sort and projection on FPGAs. In: Proceedings of 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) (2013)
4. Krueger, J., et al.: Fast updates on read optimized databases using multi core CPUs. In: Proceedings of the VLDB Endowment, vol. 5, No. 1, August (2012)
5. Low, B., Ooi, B., Wong, C.: Exploration on scalability of database bulk insertion with multi-threading. *Int. J. New Comput. Archit. Appl.* **1**(3), 553–564 (2011)
6. Johnson, R., et al.: Rowwise parallel predicate evaluation. In: Proc. Int. Conf. VLDB'08
7. Zhou, J., Ross, K.A.: Implementing database operations using SIMD instructions. In: ACM SIGMOD, pp. 145–156 (2002)
8. Satish N., et al.: Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In: ACM SIGMOD (2010)
9. Jean, J.S., Dong, G., Zhang, H., Guo, X., Zhang, B.: Query processing with an FPGA coprocessor board. In: Proceedings of the 1st International Conference on Engineering of Reconfigurable Systems and Algorithms (2001)
10. Mueller, R., Teubner, J., Alonso, G.: Glacier: a query-to-hardware compiler. In: ACM SIGMOD, pp. 1159–1162 (2010)
11. Mueller, R., Teubner, J., Alonso, G.: Streams on wires—a query compiler for FPGAs. *Proc. VLDB Endow.* **2**(1), 229–240 (2009)
12. Horikawa, T.: An unexpected scalability bottleneck in a DBMS: a hidden pitfall in implementing mutual exclusion. In: Parallel and Distributed Computing and Systems (2011)
13. Scofield, T.C., et al.: XtremeData dbX: an FPGA-based data warehouse appliance. *Comput. Sci. Eng.* **12**, 66–73 (2010)
14. Johnson, R., Raman, V., Sidle, R., Swart, G.: Rowwise parallel predicate evaluation. In: Proceedings of the International Conference on VLDB'08
15. Dennl, C., Ziener, D., Tiech, J.: On-the-fly composition FPGA based SQL query accelerators using a partially reconfigurable module library. In: Proc. IEEE Symp. FCCM, 2012. IEEE, pp. 45–52 (2012)
16. Govindaraju, N., Raghuvanshi, N., Henson, M., Tuft, D., Manocha, D.: GPU Tera-Sort: high performance graphics co-processor sorting for large database management. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, June 26–29, Chicago, IL, USA (2006)
17. Batchner, K.E.: Sorting networks and their applications. In: Proceedings of the AFIPS Spring Joint Computer Conference, vol. 32, pp. 307–314 (1968)
18. Mueller, R., Jens, T., Gustavo, A.: Sorting networks on FPGAs. *Int. J. Very Large Databases* **21**(1), 1–23 (2012)
19. Perez-Andrade, R., Cumplido, R., Feregrino-Uribe, C., Del Campo, F.M.: A versatile linear insertion sorter based on an FIFO scheme. *Microelectron. J.* **40**(12), 1705–1713 (2009)
20. Colavita, A.A., Cicutin, A., Fratnik, F., Capello, G.: SORTCHIP: a VLSI implementation of a hardware algorithm for continuous data sorting. *IEEE J. Solid-State Circuits* **38**(6), 1076–1079 (2003)
21. Hirschil, B., Yaroslavsky, L.P.: FPGA implementations of sorters for non-linear filters. In: Eusipco 2004: Proceedings of the XII European Signal Processing Conference, vol. 1, pp. 541–544. Vienna, Austria (2004)

22. Harkins, J., El-Ghazawi, T., El-Araby, E., Huang, M.: Performance of sorting algorithms on the SRC 6 reconfigurable computer. In: Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology, pp. 295–296, 11–14 Dec (2005)
23. Marcelino, R., Neto, H., Cardoso, J.M.P.: Sorting units for FPGA-based embedded systems. In: Distributed Embedded Systems: Design, Middleware and Resources, vol. 271 (2008)
24. Paul, R., Sau, S., Chakrabarti, A.: Architecture for real time continuous sorting on large width data volume for FPGA based applications. In: Computing Research Repository (CoRR), vol. abs/1206.1567 (2012)
25. Graefe, G.: Implementing sorting in database systems. In: ACM Computer Survey, vol. 38, No. 3. (2006)
26. Knuth, D.E.: The Art of Computer Programming, Vol. 3—Sorting and Searching. Addison-Wesley, Boston (1973)
27. IBM Corporation, z/Architecture Principles of Operations, SA22-7832-09, 7–118–7–124, 7–351–7–356, A-53–A-60, Sept 2012
28. Stack, M.: The Theory Behind the z/Architecture Sort Assist Instructions, Share, Session 8121, Aug 10–15 2008
29. www.picocomputing.com/pdf/SC5_sm.pdf. Pico Computing Inc
30. Sukhwani, B., et al.: Database analytics: a reconfigurable-computing approach. IEEE Micro **34**(1), 19–29 (2014)