

Accelerating the merge phase of sort-merge join

Philippos Papaphilippou, Holger Pirk, Wayne Luk
Dept. of Computing, Imperial College London, UK
{pp616, pirk, w.luk}@imperial.ac.uk

Abstract—We present an efficient, high-throughput and scalable hardware design for accelerating the merge phase of the sort-merge join operation. Sort-merge join is one of the fundamental join algorithms and among the most frequently executed operations in relational databases. It has been the focus of various recent pieces of research, each having different shortcomings and usually only focusing on the sort phase. In this paper, a new parallel sort-merge join architecture is developed, that provides data streaming functionality and high throughput. The key idea of the paper is the use of a novel design for a co-grouping engine, with which the input data are summarised on-the-fly. In this way, the operation is performed on streams of data, preserving the linear access pattern for faster data movement and also eliminates the need for a replay buffer/cache. In contrast to related work, our approach does not make assumptions about the value distribution of the input data and applies to any input size and width. We evaluate the design on a Zynq UltraScale+ based platform, and show that there is up to 3.1 times speedup over the host processor, even without accelerating the sort phase, while still taking into account the data transfers from and to main memory in Linux.

Index Terms—Sort-merge join, stream processing, operators, database acceleration, high-throughput, FPGA, MPSoC, big data

I. INTRODUCTION

Joins, in particular equi-joins (those with an equality predicate) are one of the key operations in relational database systems. It is, therefore a rewarding target for hardware acceleration.

The two most widely applied (equi-)join algorithms are sort-merge join and hash-join (in various forms). Neither of these algorithms strictly dominates the other in terms of performance making an efficient implementation of both important for optimal performance and on different architectures [1]. The common wisdom in data management research is to apply hash-join when one table is much larger than the other and the data are unsorted. Sort-merge join is applied when the table sizes are in the same order of magnitude or when the input tables are sorted. In this case, the dominant performance factor is the merge phase. While there is published research on the hardware-acceleration of the sort phase, there is no significant progress on the merge phase.

Equijoin is the task to join two tables by combining all rows that share the same value in the join attributes. If keys appear more than once in the two tables, then the join operation would need to create a Cartesian product of the matching rows from the two tables.

Hash-join implementations require some form of a hash-table [2]. The input data do not need to be sorted, and for this

reason they produce a random access pattern. CPUs are an attractive platform for hash-joins, since they are equipped with mechanisms to cope with the additional latency of accessing data in a non-linear pattern, and their main memory is also designed for lower latency.

On the other hand, co-processors such as FPGAs shine at processing data with linear access patterns, such as with sort-merge join. Our goal is to accelerate equijoin using sort-merge join on FPGAs, to benefit from the algorithm’s streaming nature, while overcoming a number of related challenges found in current research.

The key contributions of this work are as follows.

- A fully-streaming sort-merge join approach, where the merge phase is on the programmable logic.
- Design decisions that allow merge-join to be performed on data of arbitrary length, width and key distribution.
- The high-throughput building blocks for co-grouping, round-robin and join.

II. CHALLENGES

Before discussing our approach, let us describe the challenges. How these are addressed by our solution is presented in subsection III-D.

C1. Input properties

An input feature that complicates sort-merge join accelerators is the presence of duplicate keys in a table. A Cartesian product needs to be performed on the two sets of rows with matching keys from the two inputs. For many duplicate keys, a naive solution needs to look back on the stream, such as by having a cache/replay buffer of the recent values. This complicates the design and impacts the performance [1].

Another challenging aspect of the input is its size (length and width). It would be ideal to avoid keeping input-related states inside the FPGA, to make it applicable to arbitrarily long tables. With respect to the data width, the presence of long data fields can restrict the applicability of a join accelerator or impact performance.

In related work, there are assumptions about these input properties, such as by only targeting databases of limited scale, and they need to be eliminated in order to provide a solution with real-world applications and scalability in mind.

C2. Data movement

The data movement is usually the bottleneck for hardware-accelerated databases and big data analytics [3]. Considering that the throughput and latency ‘inside’ the FPGA is

many orders of magnitude faster than the ports to the host processor or DRAM, research is leaning towards complicated designs for minimising the inter-chip communication. Common workarounds include the implementation of cache hierarchies, data prefetching and compression. The bandwidth constraints become more limiting when transferring wide table rows alongside the keys to be compared.

C3. Scalability

For keeping up with technological advances in the data center, high-throughput designs can be challenging to develop. The main considerations are the high hardware resource utilisation and the timing constraints. As with sorting [4], [5], it is desirable for a pipelined stream processor to have narrow feedback datapaths. Otherwise, the operating frequency will need to be lowered to allow complex pipeline stages operating under one cycle. Related to the merge phase, the Cartesian product is currently known to limit the output rate [1].

III. PROPOSED APPROACH

Our proposed design is for a single-attribute join. It takes two sorted input streams, A and B , each containing rows of the form $\langle index, key \rangle$. The resulting output contains rows of the form $\langle i_{start,A}, i_{end,A}, i_{start,B}, i_{end,B}, key \rangle$, where $i_{start,A}$, $i_{end,A}$ and $i_{start,B}$, $i_{end,B}$ represent the index ranges for A and B respectively, where the value of the key is the same.

The merge-join accelerator is a modular design and consists of two **co-grouping engines** (CG), for compacting multiple tuples with same keys into row ranges, a **join module**, to match ranges with same keys in the two streams, two circular queue buffers, to synchronize the co-grouping engines with the join module, and finally multiple **round-robin modules** (RR), responsible for removing invalid ($Null$) entries while keeping the data correctly aligned in banked memories. In the following subsections we are going to see these building blocks in detail.

Figure 1 presents an overview. P denotes the degree of parallelism, or the maximum number of elements that can be processed per cycle. This design receives either 0 or P elements per cycle from each of the 2 inputs, while the number of rows exported in output ranges from 0 to $P/2$ per cycle.

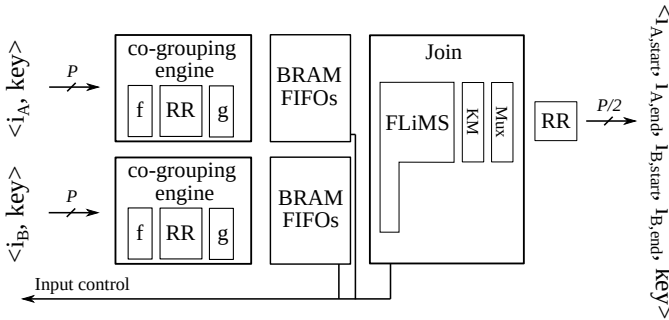


Fig. 1. Building blocks of the proposed solution

A. Co-grouping engine

The data first pass through the co-grouping engine. There are two co-grouping engines, one for each input stream. They are responsible for summarising multiple consecutive entries sharing the same key (duplicates). The input stream is a series of $\langle index, key \rangle$ pairs and the output consists of $\langle index_{start}, index_{end}, key \rangle$ tuples.

The goal is to be able to provide a high-throughput by processing many $\langle index, key \rangle$ pairs per cycle. For this reason, we assume that the data are organised in a round-robin fashion, such as when loading consecutive main memory locations to banked memory. In order to do this, we use the parallel architecture as shown in figure 2.

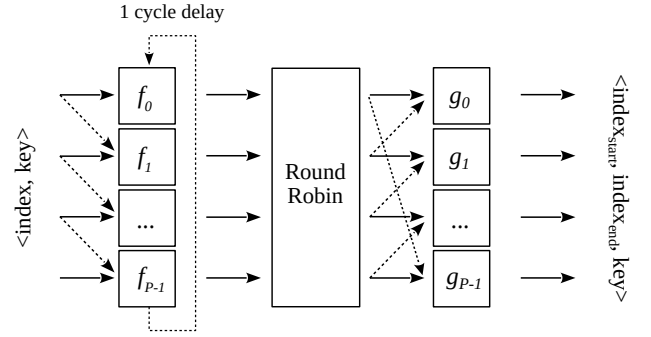


Fig. 2. Co-grouping engine

In the first pipeline stage, we have P entities, denoted with f , that invalidate any tuples whose key does not appear for the first time. The resulting tuples are of the form $\langle index, key, valid \rangle$ where the $\langle index, key \rangle$ part is the same as the input (one cycle delayed). The $valid$ bit is the result of the condition $(key \neq key_{Left})$, where key_{Left} is the immediately previous key. In order to provide the correct previous key to the top entity (f_0), the last one (f_{P-1}) also works as a shift register, to delay the last key of the current batch by a cycle, for comparing it with the top of the next batch.

After the first pipeline stage, the round-robin module permutes the $\langle index, key, valid \rangle$ tuples according to their valid bit and the last offset, so that the valid tuples will appear as they are produced in order (more details in III-B).

During the last pipeline stage, there are P entities responsible for resolving the $index_{end}$ field of the ranges for the duplicates and finalise the output of the co-grouping engine. The output tuples are of the form $\langle index_{start}, index_{end}, key \rangle$. Each one of the g_i entities receives both the tuples in positions i and $(i+1)\%P$. Note that this does not require delayed input, although there are internal state registers. The actions taken based on input and states are illustrated in algorithm 1.

B. Round-robin module

The round-robin module is responsible for reordering sparse data (with $Nulls$), such that it will appear as the valid portion was written serially in a round-robin fashion. This is done in parallel to enable high-throughput processing. Examples of the

```

1 registers  $index', key', valid'$ ;
2 input  $\langle index, key, valid \rangle$ 
3 input  $\langle index, key, valid \rangle_{Right}$ 
4 output  $\langle index_{start}, index_{end}, key \rangle$ 
5 while forever do
6   receive (positive clock edge);
7   if ( $valid \ \&\& \ valid_{Right}$ ) then
8     if ( $key > key_{Right}$ ) then
9       output  $\langle index, index_{Right} - 1, key \rangle$ ;
10    else
11      if  $valid'$  then
12        output  $\langle index', index_{Right} - 1, key' \rangle$ ;
13      else
14        output  $Null$ ;
15      end
16       $index', key', valid' \leftarrow index, key, True$ ;
17    end
18  else if ( $\neg valid \ \&\& \ valid_{Right}$ ) &&  $valid'$  then
19    output  $\langle index', index_{Right} - 1, key' \rangle$ ;
20     $valid' \leftarrow False$ ;
21  else if ( $valid \ \&\& \ \neg valid_{Right}$ ) then
22     $index', key', valid' \leftarrow index, key, True$ ;
23    output  $Null$ ;
24  else
25    output  $Null$ ;
26 end

```

Algorithm 1: Entity g_i algorithm pseudocode

output destination are a set of P block RAM banks, or another processing module, such as in the co-grouping case.

The pipelined solution here is based on a bitonic sorter, an adder tree and barrel shifters. The bitonic sorter [6] is a parallel algorithm for sorting a list of N elements in $(\log_2(N) \cdot (\log_2(N) + 1))/2 \in \Theta(\log^2(N))$ steps. It consists of multiple compare-and-swap (CAS) units, which are sorters of 2 elements. Here, the bitonic sorter is used to bring down the entries with their valid bit equal to 0. This is achieved by appending the valid bit to the most significant bit (MSB) position of the key. The adder tree is to perform a popcount (addition of the valid bits) operation in a pipelined way. The latency of the adder tree is $\log_2(N)$ steps, which is conveniently shorter than that of the bitonic sorter. There is an additional addition with the previous sum, as a way to update the last offset (SR). The output of this addition is then fed to the barrel shifters, that perform the final rotation in another $\log_2(N)$ steps. This is summarised in figure 3.

This functionality can be seen as a parallel form of a round robin arbiter [7], and the sorting network a self-routing permutation network [8]. Our approach can be used to build such a design for a custom P , though there are alternative design choices, such as different sorting networks [9], [10]. For a small number of inputs, some components here could be simplified. For example, the adder tree is not necessary for $P=4$, as the full addition fits in the specified peripheral cycle.

C. Join module

The purpose of the join module is to get the $\langle index_{start}, index_{end}, key \rangle$ tuples from the two co-groupers, one for each input stream A and B, and encode the Cartesian product whenever the key matches. This results in the output stream, which consists of tuples of the form $\langle i_{start,A}, i_{end,A}, i_{start,B}, i_{end,B}, key \rangle$.

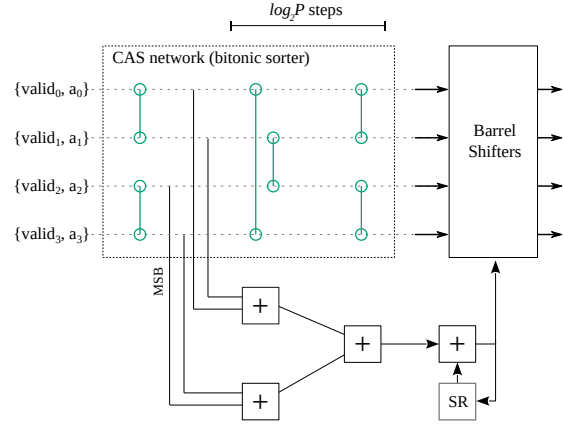


Fig. 3. Round robin module for $P=4$

The join module is based on FLiMS [5], a recent work that provides a building block for designing a high-throughput H/W merge sorter [11]. This block merges 2 sorted streams into one sorted stream, with a steady output rate of P entries per cycle. The idea of using an efficient merger of 2 streams is based on the fact that, since we essentially eliminate the duplicate keys, we will have a maximum of two identical keys consecutively, after we sort the two input lists together. Then, we can match the grouped keys by just comparing consecutive entries in the sorted-by-key output. Figure 4 provides an overview.

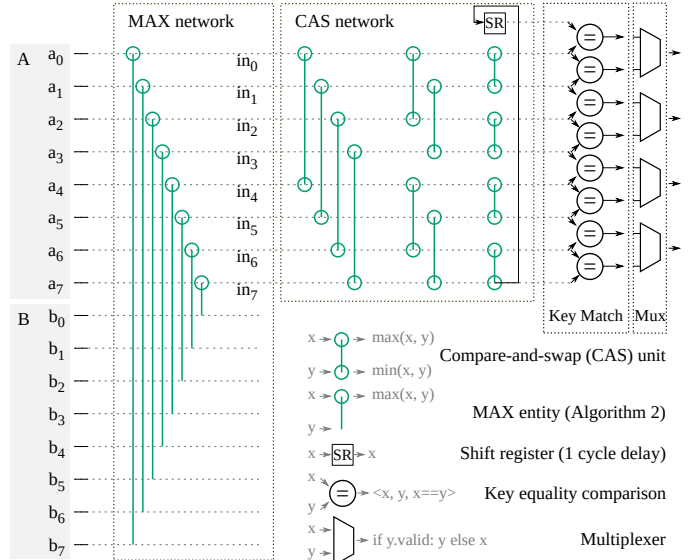


Fig. 4. Join module for $P=8$

Beforehand, the two data streams of $\langle index_{start}, index_{end}, key \rangle$ tuples are stored in two circular queue buffers, one for input A and one for input B. The circular queue buffers are divided into P banks each, to provide a degree of parallelism P . On each cycle, FLiMS dequeues a total of P heads, denoted a_i and b_i from inputs A and B respectively. The MAX entities allow only the top P out of the current $2P$ values residing in the internal registers ca_i

and cB_i to pass through. The functionality of these entities is described in algorithm 2. The results of these comparisons are also used for input control, as they are used to calculate the empty space in the circular queue buffers.

```

1 int i;                                ▷ i is the entity tag
2 registers cAi, cBi, ini;
3 while forever do
4     receive (positive clock edge);
5     if cAi.key > cBi.key then
6         ini ← < 0, cAi >;           ▷ 0 for source A
7         cAi ← dequeue(ai);
8     else
9         ini ← < 1, cBi >;           ▷ 1 for source B
10        cBi ← dequeue(bP-i);
11    end
12 end

```

Algorithm 2: Entity MAX_i algorithm pseudocode

We modified FLiMS in the following ways: 1) Every data unit needs to carry the payload, which are the $index_{start}$ and $index_{end}$ fields, as well as an extra bit to remember each tuple’s source, for inputs A and B. 2) The comparators inside the CAS and MAX_i entities now operate on the key fields only. The topology is based on the bitonic partial merger, which is a subset of the bitonic sorter [5], [12].

After FLiMS, there is the key matching stage (KM), where each consecutive pair of tuples are compared based on their keys. The last entry/tuple is also delayed for an additional comparison to the top of the next batch, in the cases where a matching pair is halfway through. When there is a match, the resulting $\langle i_{start,A}, i_{end,A}, i_{start,B}, i_{end,B}, key \rangle$ tuple is produced, and the position of the i fields are arranged according their source bit.

Next, there are $P/2$ multiplexers (Mux) to reduce the output width to the maximum output rate ($P/2$), based on the valid bits. This is because there should not be three consecutive identical keys, and therefore no two consecutive matches. Finally, the join module output goes through another round-robin module for $P/2$ inputs (see figure 1) to remove sparsity and the results are saved in a banked buffer for immediate consumption.

D. Properties overview

In regard to the input/output, the approach to use indices instead of passing full rows of tables minimises the inter-chip data movement. This is consistent with a strategy known as “late materialization” that is implemented by virtually all modern in-memory database systems [13]. Here, it reduces the bandwidth consumption (challenge C2) and makes it applicable to arbitrarily wide tables. Additionally, this fully-streaming functionality is agnostic to the input length (challenge C1).

Another input challenge is the data skewness problem, which is traditionally found in hardware merge sorters [11]. It arises when the input queues are not long enough and the merger may underutilise the bandwidth of one of the inputs because of the key distribution. In our design, this can be solved efficiently at the circular buffer queue level, since it is able to hold many co-grouped tuples.

The total pipeline length is equal to $\log_2^2(P) + 2 \cdot \log_2(P) + 4 \in O(\log^2(P))$, (ignoring the queue buffers).

In terms of hardware resources, it requires 2 sets of P BRAM circular queue buffers of data width $3 \cdot 32$ -bit (for the fields in $\langle index_{start}, index_{end}, key \rangle$) and of depth greater than $1 + (\log_2(P) \cdot (\log_2(P) + 1)/2 + \log_2(P)) + 1 \in O(\log^2(P))$, the latency of the co-grouping engine. Regarding the LUT/FF/routing complexity, the fastest growing resources are in the CAS network of RR. This is in $O(P \cdot \log^2(P))$ and is an achievement over related work (V).

The H/W complexity with respect to the key-size is linear. However, if the key size is hundreds of bits wide, the CAS units might not be able to fit under a narrow FPGA cycle. In such cases, there are solutions for scaling arithmetic operations on wider fields [14], which could be evaluated in this context.

With scaling the degree of parallelism P , the building blocks remain ‘feedback-less’. We conducted a scalability analysis (not shown) that validated the expected complexities in FF/LUT using Vivado, as well as some of the timing properties.

IV. EVALUATION

The evaluation was done on Avnet’s Ultra96 development board. The board has 2GB of LPDDR4 RAM and features the Zynq ZU3EG MPSoC, which is based on the Xilinx UltraScale+ architecture. The 4 ARM CPU cores run Linux at 1.2GHz. In comparison with earlier iterations of this architecture [15], one of its database-related advantages is that it provides 128-bit-wide ports for faster communication between the processing system (PS) and the programmable logic (PL).

A program¹ written in C has two functions for the same task, one is a software implementation and the other is hardware-accelerated. The code for the latter is simple, as it just initiates some DMA copies and busy-waits when necessary. The input of the program is a text file containing two integers per line, which represent keys from the inputs A and B. These data are saved twice, once in a memory-allocated (malloc) space and once in a memory-mapped (mmap) space. The first will perform better for the software implementation, since it is cached, for a fair comparison. The latter is a continuous DRAM memory location outside the kernel space, where the Direct Memory Access (DMA) engine will move data from and into the programmable logic. After calling the two functions individually, the expected result is the same, an array of tuples, each carrying 5 32-bit values (see section III). Our current evaluation focuses on comparing single-thread (software) and one-port (hardware) performance.

In order to minimize any runtime noise, measures that were taken include pinning processes, running in isolation with high priority and averaging tens of runs per data point. The C program is compiled using `gcc -O3 -march=native`.

The implementation of the merge phase accelerator is for $P=8$ (256-bit/cycle), and has a clock frequency of 115 MHz. This means the peripheral itself is able to process input entries at a maximum overall rate of 6.85 GiB/s.

One basic performance factor is the input size. Longer DMA transfers are more efficient in such a heterogeneous system.

¹Source available: <http://philippos.info/mergejoin>

In figure 5, we can see how the wall time is affected for the non- and hardware- accelerated versions of the merge phase for different input sizes (of similar input characteristics). This is also compared with measurements for the DMA copy throughput. Here, the FPGA speedup peaks at 2.8 times, when the DMA throughput reaches 1.4 GiB/s.

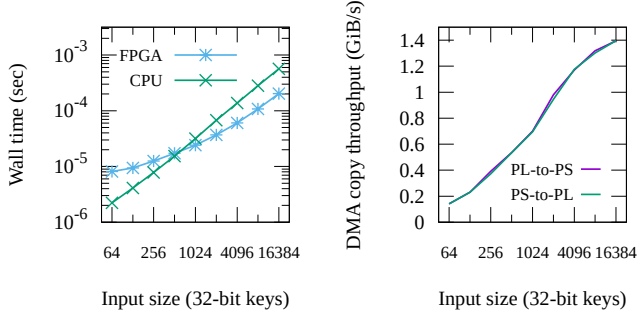


Fig. 5. Wall time and DMA AXI copy speed

The performance of both S/W and H/W implementations depends on the key value distributions. In figure 6, we present experimental results on how the FPGA speedup varies across different random inputs. The data are automatically generated to have some specified characteristics. We vary the percentage of distinct keys in both inputs together, as well as the number of key matches. The input size is fixed (16384 32-bit keys per table). As an observation, FPGA favours inputs that yield more ‘internal’ workload, i.e. many co-groupings of duplicate keys, with fewer rows in output for less inter-chip communication.

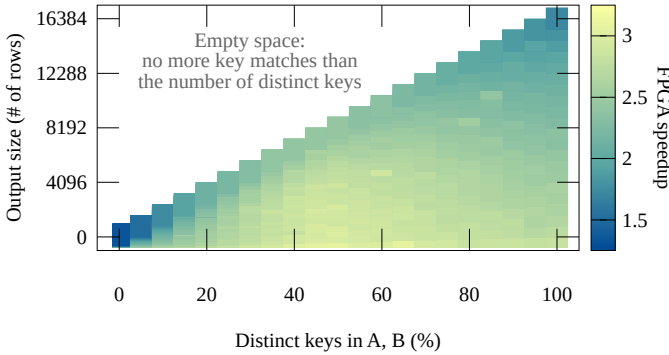


Fig. 6. FPGA speedup for different key distributions

The FPGA speedup for this input design space varied from 1.3 to 3.1, with 2.3, 2.6 and 2.8 being the lower quartile, the median and the upper quartile respectively. This peripheral consumes about 18K LUTs and 20K FFs, which take 26% and 14% of the respective capacity for this device.

V. RELATED WORK

To the best of our knowledge, there is only one work that provides a hardware merge phase accelerator, by Casper et al. [1]. The idea is to use multiple FPGAs for task-pipelining. The main drawback is the presence of a replay buffer, which is explained to be a bottleneck on certain inputs, as it makes the access pattern less linear. Additionally, the maximum output

rate is restricted because the hardware complexity of the join part grows quadratically, as it needs N^2 comparators for matching keys from N rows in each input at a time [1].

Another sort-merge join accelerator focuses on a heterogeneous platform [15]. This work combines a sort accelerator with a merge phase algorithm for the CPU. It does not provide a merge accelerator in hardware, as it is left for software.

There are also approaches that use different join algorithms, mainly based on hash-join. In such cases, there need to be expensive mechanisms to hide the induced latency for random accesses. Examples include emulating multiple threads on a single FPGA [16] or accelerating data partitioning for parallel hash joins [17], to saturate the main memory bandwidth.

Another category of related research focuses on small scale databases, such as for embedded systems. A recent work [2] uses a parallel tree-based join (PTJ) structure that resides on the FPGA. The solution performs well for medium-sized input, but the tree structure slows down for bigger inputs and it also has to fit in BRAM, thus making it inapplicable for big data.

VI. FUTURE WORK AND CONCLUSIONS

There are many ways this work can be expanded. The unused FPGA resources could be spent on high-throughput [4], [5] or large scale sorters [18] for implementing a full H/W sort-merge join. Different stream processors could be attached for accelerating more complex analytics, such as for filtering and machine learning. For better evaluating the potential of the FPGA, we could also test inputs greater than the CPU cache size. The design could also be evaluated for the multi-threaded case and its equivalent for hardware (i.e. using multiple ports). It would be interesting to evaluate on high-end systems with high inter-chip communication and memory bandwidth for datacenter use. This modular design is expected to scale well.

We provide a modular design for accelerating the merge phase of sort-merge join. The architecture is designed to be scalable to provide very high throughput for high-end and future architectures. The building blocks developed here could be used in the development of novel stream processors. The evaluation on an MPSoC system yields a performance speedup of up to 3.1 over the host processor, while still taking into account the data transfers under Linux. The co-grouping engine and the index range approach remove restrictions on data, such as the input size, duplicate key distributions and row size, that are found in previous solutions. The summarised output is also desirable when extending its functionality to more complex queries, such as selection/filtering on joined tables, where the full output would be unnecessary and wasteful. The appeal of the merge-join accelerator is its streaming nature and scalability, as it can be combined with sorting or other stream processors to provide efficient co-processors for accelerating big data analytics.

VII. ACKNOWLEDGEMENTS

The support of dunnhumby and the United Kingdom EPSRC (grant numbers EP/L016796/1, EP/N031768/1, EP/P010040/1 and EP/L00058X/1) is gratefully acknowledged. The authors would like to thank Chris Brooks from dunnhumby for his insightful comments.

REFERENCES

- [1] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, ACM, 2014, pp. 151–160.
- [2] H. Zhang, B. Zhao, W.-J. Li, Z.-G. Ma, and F. Yu, "Resource-Efficient Parallel Tree-Based Join Architecture on FPGA," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 66, no. 1, pp. 111–115, 2019.
- [3] P. Papaphilippou and W. Luk, "Accelerating database systems using FPGAs: A survey," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2018, pp. 125–130. DOI: 10.1109/FPL.2018.00030.
- [4] M. Saitoh, E. A. Elsayed, T. Van Chu, S. Mashimo, and K. Kise, "A high-performance and cost-effective hardware merge sorter without feedback datapath," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2018, pp. 197–204.
- [5] P. Papaphilippou, C. Brooks, and W. Luk, "FLiMS: Fast Lightweight Merge Sorter," in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 78–85. DOI: 10.1109/FPT.2018.00022.
- [6] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ACM, 1968, pp. 307–314.
- [7] J. Chao, "Saturn: A terabit packet switch using dual round robin," *IEEE Communications Magazine*, vol. 38, no. 12, pp. 78–84, 2000.
- [8] D. M. Koppelman and A. Y. Oruç, "A self-routing permutation network," *Journal of Parallel and Distributed Computing*, vol. 10, no. 2, pp. 140–151, 1990.
- [9] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on FPGAs," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 21, no. 1, pp. 1–23, 2012.
- [10] D. Bundala and J. Závodný, "Optimal sorting networks," in *International Conference on Language and Automata Theory and Applications*, Springer, 2014, pp. 236–247.
- [11] W. Song, D. Koch, M. Luján, and J. Garside, "Parallel hardware merge sorter," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2016, pp. 95–102.
- [12] A. Farmahini-Farahani, A. Gregerson, M. Schulte, and K. Compton, "Modular high-throughput and low-latency sorting units for FPGAs in the large hadron collider," in *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, IEEE, 2011, pp. 38–45.
- [13] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden, "Materialization strategies in a column-oriented DBMS," in *2007 IEEE 23rd International Conference on Data Engineering*, IEEE, 2007, pp. 466–475.
- [14] H. D. Nguyen, B. Pasca, and T. B. Preußner, "FPGA-specific arithmetic optimizations of short-latency adders," in *2011 21st International Conference on Field Programmable Logic and Applications*, IEEE, 2011, pp. 232–237.
- [15] R. Chen and V. K. Prasanna, "Accelerating Equi-join on a CPU-FPGA heterogeneous platform," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2016, pp. 212–219.
- [16] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras, "FPGA-based Multithreading for In-Memory Hash Joins," in *CIDR*, 2015.
- [17] K. Kara, J. Giceva, and G. Alonso, "FPGA-based data partitioning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ACM, 2017, pp. 433–445.
- [18] K. Manev and D. Koch, "Large utility sorting on fpgas," in *2018 International Conference on Field-Programmable Technology (FPT)*, IEEE, 2018, pp. 334–337.