IEICE TRANSACTIONS

on Fundamentals of Electronics, Communications and Computer Sciences

DOI:10.1587/transfun.2023EAP1135

Publicized:2024/05/28

A PUBLICATION OF THE ENGINEERING SCIENCES SOCIETY

The Institute of Electronics, Information and Communication Engineers

Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

High-Parallelism and Pipelined Architecture for Accelerating Sort-Merge Join on FPGA

Meiting XUE[†], **Wenqi WU**[††], **Jinfeng LUO**[††], **Yixuan ZHANG**[†††,††††], ***Nonmembers***, ***and*** **Bei ZHAO**[††], ***Member***

SUMMARY Join is an important but data-intensive and compute-intensive operation in database systems. Moreover, there are multiple types of join operations according to different join conditions and data relationships with diverse complexities. Because most existing solutions for accelerating the join operation on field programmable gate arrays (FPGAs) focus only on the easiest join application, this study presents a novel architecture that is suitable for multiple types of join operation. This architecture has a modular design and consists of three components that are executed sequentially and in pipeline. Specifically, the top-K sorter is used instead of the full sorter to reduce resource utilization and advance the merge processing. Further, the architecture is perfectly compatible with both N-to-1 and N-to-M join relationships, and can also adapt well to both equi-join and band-join. Experimental results show that this design, which is implemented on an FPGA, achieved a high join throughput of 242.1 million tuples per second, which is better than other reported FPGA implementations.

key words: *FPGA-based acceleration, database, join operation, sort.*

1. Introduction

With the development of significant breakthroughs in field programmable gate array (FPGA) design, FPGA-based architectures have become an attractive option for accelerating various database systems [1, 2]. FPGAs provide enormous flexibility for users and can be custom-designed to suit individual requirements and data qualities because of their programmable characteristics. Moreover, FPGAs are particularly appropriate for solving problems that can be naturally parallel and pipelined. It can not only be used as a co-processor for existing systems, but also as a separate system and platform for independent work due to its fully programmable characteristics. One such problem is the acceleration of the join query, an essential database operation that combines tables characterized by given conditions.

Based on the join condition, join queries can be categorized into two types: equi-join with equality conditions, and non-equi-join (also called θ-join) with inequality conditions. Here, θ refers to a binary relational operator belonging to the set $\{<, \leq, =, \neq, >, \geq\}$. If θ refers to the equality sign (=), then this join degrades into an equi-join [3, 4]. Most research efforts on join acceleration have been dedicated to the equi-join because of its easy implementation that can be achieved with efficient acceleration through a number of advanced strategies. At the same time, the θ-join is not developed with equi-join synchrony. It is significant to integrate them into one architecture.

Join operations can also be divided into three types based on the data model relationship: as one-to-one (1-to-1), many-to-one (N-to-1), and many-to-many (N-to-M) joins. The main difference between these three relationships is whether there are duplicate keys in one table.

On the other hand, the three fundamental algorithms for performing a join operation are the nested-loop join, hash join, and sort-merge join [5–8]. Among these, the nested-loop join has a relatively simple algorithm; however, it is usually time-consuming and inefficient. Therefore, the hash join and sort-merge join are more widely applied in join queries than the nested-loop join is. A hash join is implemented in two phases: a build phase for constructing hash tables, and a probe phase for finding match joins in hash tables. Because of the advantages of the hash function, a hash join is theoretically able to support fast searches, yet involves random accesses. It also requires another storage to store hash key-value pairs, making it more suitable for situations where one table is much smaller than the other. The hash join is well suited to equi-join operations; however, because of its random accesses, it is not as suitable for θ-join operations. Moreover, when hash join is applied to N-to-M joins, the join performance is greatly affected because there will be more hash collisions with complete traversal. The sort-merge join also consists of two phases: a sort phase for sorting elements, and a merge phase for exporting match joins. However, unlike the hash join, the sort-merge join performs better when two tables are in the same order of magnitude and requires no additional storage. Moreover, the sort-merge join can be competent for any join type regardless of the join condition and data model relationship, without performance degradation. Thus, this study focuses on a general architecture for the join operation, exploiting the sort-merge join algorithm as our basic skeleton.

In this paper, we present a novel architecture that has a modular design and is suitable for multiple types of join

[†]The authors are with the Department of Cyberspace, Hangzhou Dianzi University, Hangzhou 310018, China (e-mail:munuan@hdu.edu.cn).

[††]The author is with the College of Biomedical Engineering and Instrument Science, Zhejiang University, Hangzhou 310027, China (e-mail:winkywow@zju.edu.cn (Corresponding Author), luojf9@zju.edu.cn).

[†††]The author is with Institute of information engineering, CAS, Beijing 100085, China (e-mail:zhangyixuan234@mails.ucas.ac.cn).

[††††]The author is with School of Cyber Security, University of Chinese Academy of Sciences, Beijing 101408, China (e-mail:zhangyixuan234@mails.ucas.ac.cn).

[†]The author is with the School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou 310018, China (e-mail:zhaobei@hdu.edu.cn).

operation. This architecture consists of three main components: partition, sorter, and comparison, which are executed sequentially and in pipeline. Each of these components is flexible. Moreover, the architecture can easily realize parallelism, leading to a further improvement of performance. The main contributions of this study are as follows:

- A hardware architecture with high parallelism and a deep pipeline is constructed, such that it accommodates the characteristics of FPGA.
- The architecture is created with a modular serial design that can be easily implemented and can be tailored to suit individual requirements.
- Its high-throughput, FPGA-based implementation for accelerating sort-merge join operations outperforms other approaches.
- The proposed solution provides for all scenarios of the join query, including equi-join and θ-join in N-to-1 and N-to-M data model relationships.

The remainder of this paper is organized as follows: Section 2 provides information regarding closely related work. Section 4 describes the basic algorithm in the sort and merge phases. Section 5 introduces the architecture implementation. Section 6 presents the experimental details and a comparison of the results with those of existing methods. Finally, section 7 concludes this study.

2. Related work

2.1 FPGA-Based Database Accelerator

In the era of big data, as accelerators continue to be designed and used on databases, several FPGA-based algorithms have been performing well in specific operations [1, 9]. For example, compaction plays a critical role in Log-Structured-Merge-tree (LSM-tree) based key-value stores, which are widely used in applications with write-intensive workloads. To this end, Sun et al. [10] designed and implemented an FPGA-based compaction engine with key-value separation and index-data block separation strategies, while fully utilizing the bandwidth of the FPGA chip to improve the compaction performance. On the other hand, with regard to storage, to solve the severe bandwidth bottlenecks that occur when vast amounts of data are moved from storage to query processing nodes, Woods et al. [11] developed Ibex, a prototype of a hybrid intelligent storage engine that supports off-loading of GROUP BY aggregation and projection- and selection-based filtering operators. Taking advantage of the programmability offered by the OpenCL HLS tool, Wang et al. [12] proposed an FPGA-specific cost model that includes two components, namely unit cost and optimal query plan generation, to determine the optimal query plan in less than one minute.

Meanwhile, sorting is essential for many scientific and data processing problems. For example, combining the parallelization of algorithms and the use of designs based on specialized hardware structures, Li et al. [13] proposed an extended nonstrict partially ordered set-based configurable linear sorter for FPGAs, which consists of multiple customizable micro-cores as sorting units. The sorting units in their architecture package the storage and comparison of tuples, are connected and communicated into a chain, and act the same way in each clock cycle to improve the realized frequency of the sorter. As a basic data structure, hash tables are widely used in the database, as in the hash join operation mentioned earlier. To address the possibility of hash collision, which will involve several probes on item relocation, Li et al. [14] proposed an efficient Cuckoo hashing scheme, called Multi-copy Cuckoo or McCuckoo, to foresee ways of successfully kicking items by using multiple copies, to check for and avoid expensive rehashing during insertion failures. Jinyu Zhan et al. [15] designed a novel data processing framework to prefilter data, and then propose a CPU-FPGA co-design to accelerate the queries with a workload-aware task scheduler.

2.2 Hardware-Based Equi-Join Accelerator

Hardware-accelerated equi-join operations have been receiving much attention from the engineering community in recent years [16–19]. In relevant research studies, two targeted platforms, namely FPGA and GPU, have been considered. For instance, Chen et al. [20] used a hybrid CPU-FPGA heterogeneous platform by operating "folded" bitonic sorting networks in parallel on the FPGA and merging the partial results on the CPU to benefit from both high parallelism and large available memory. Based on this hybrid sorting design, they developed two streaming join algorithms by optimizing the classic CPU-based nested-loop join and sort-merge join algorithms, thus obtaining good throughput improvement. On the other hand, Zhou et al. [8] utilized hierarchical indexes to identify result-yielding regions in a solution space to take advantage of result sparseness and achieve acceleration with low match rates on an FPGA. Further, in addition to one-dimensional equi-join query processing, their solution supports processing of multidimensional similarity join queries. Meanwhile, Papaphilippou et al. [21] proposed a co-grouping engine on an FPGA for the merge phase of the sort-merge join, with which the input data are summarized on-the-fly, preserving the linear access pattern for faster data movement and eliminating the need for a replay buffer/cache.

With the rapid development in the GPU world, many accelerators have also been designed based on GPU. For instance, Rui et al. [22] overhauled the popular radix hash join and redesigned sort-merge join algorithms on GPUs by applying a series of novel techniques to utilize the hardware capacity of the latest Nvidia GPU architecture and the new features of the CUDA programming framework. They took advantage of revised hardware arrangement, larger register file and shared memory, native atomic operation, dynamic parallelism, and CUDA Streams to improve the performance

of the operation. Meanwhile, Sioulas et al. [23] designed and implemented a family of novel, partitioning-based GPU-join algorithms that work around the limited memory capacity and slow PCIe interface in GPU. Lin Qian et al. [24] designed a write-optimized edge storage system via concurrent microwrites merging. They implement a two-level cache structure that dispatches the concurrent write threads efficiently and significantly mitigates the cache block competition problem with a flexible merging scheme. Tarikul et al. [25] proposed Relational Fabric. It is a near-data vertical partitioner that allows memory or storage components to perform on-the-fly transparent data transformation. Relational Fabric also pushes vertical partitioning to the hardware. Huan Zhang et al. [26] proposed a resource-efficient join architecture based on a tree model with a parallel implementation. It needs build and probe phase to achieve. However, when the data is skewed, the binary tree may be quite unbalanced.

2.3 Theta-Join Accelerator

By contrast, with regard to θ-joins, although these operations are widely used in various applications, very minimal research related to their hardware-based implementation, especially FPGA-based acceleration, has been conducted. Therefore, queries containing such joins are notably slow because of their inherent quadratic complexity. In an attempt to mitigate this problem, Koumarelas et al. [3] proposed an ensemble-based partitioning approach to save on communication cost and reduce the total execution time of θ-joins in a massively parallel setting. The key idea is to cluster join key values following matrix re-arrangement and agglomerative clustering techniques in isolation or combination. On the other hand, Wang et al. [4] used Hadoop with GPU to speed up nested-loop join, hash join, and θ-join operations used on big data. They used MapReduce and HDFS in Hadoop to handle large data tables, and estimated the number of results accurately and allocated the appropriate storage space without unnecessary costs.

Besides, there are some methods accelerated on CPU. For example, Okcan et al. [27] derived a simple randomized algorithm, referred to as 1-Bucket-Theta, for implementing arbitrary joins in a single MapReduce job, which requires minimal statistics. Moreover, they proposed the M-Bucket class of algorithms, which can further improve runtime as long as sufficiently detailed input statistics are available. Meanwhile, Li et al. [28] proposed a prefix tree index with a holistic pruning ability on multiple attributes, in conjunction with a cost model, to quantify the prefix tree. Based on this, they further devised a filter-verification framework to support similarity search and join on multi-attribute data. On the other hand, Khayyat et al. [29] introduced an inequality join algorithm based on sorted arrays and space-efficient bit-arrays and devised a simple method for estimating the selectivity of inequality joins used to optimize multiple predicate queries and multi-way joins.

3. Symbol description

The symbol description is shown in table 1

Table 1: The meaning of each symbol in paper.

Symbol	Meaning
R, S	Name of table
R_{rid}	The rid^{th} record in table R
$R_{rid}.key$	The key value of rid^{th} record in table R
K	The number of elements in a single sort
P	The number of parallelisms to divide data in average
N_R	The size of table R
n_r	The match size in table R

4. Proposed solution

Our objective is to design a hardware-based accelerating architecture suited for both equi- and θ-joins. The skeleton of our architecture is based on sort-merge join, which consists of sort and merge phases. Accordingly, we aim to design pipelined algorithms for both phases. In this section, we describe the proposed basic algorithms in the order of execution.

4.1 Join Example

It is assumed that a join query operates between two tables, namely table R and table S. Tuples in $\{rid, key\}$ format are the records streaming into the join architecture, and matching tuples are exported in $\{(r_{rid}, s_{rid})\}$ format. Here, rid is used to identify the row address, key is the join attribute, and r_{rid} (resp., s_{rid}) is the corresponding tuple of table R (resp., S). As an example, for tables R and S in Fig. 1, an equi-join query Q_e can be applied, as follows:

Q_e: SELECT $r.rid$, $s.rid$
FROM R r, S s
WHERE $r.key = s.key$.

Query Q_e returns a set of pairs $\{(r_i, s_j)\}$, where $r_i.key = s_j.key$; here, the result is $\{(r_3, s_2), (r_5, s_1)\}$. On the other hand, an example of a θ-join query, Q_θ, can be as follows:

Q_θ: SELECT $r.rid$, $s.rid$
FROM R r, S s
WHERE $ABS(r.key - S.key) < 5$.

Here, θ is the binary relational operator '>', and query Q_θ returns a set of pairs $\{(r_i, s_j)\}$, where $r_i.key > s_j.key$. The result here is $\{(r_1, s_1), (r_1, s_2), (r_1, s_3), (r_1, s_4), (r_2, s_1), (r_2, s_3), (r_2, s_4), (r_3, s_1), (r_3, s_3), (r_3, s_4), (r_4, s_1), (r_4, s_3), (r_4, s_4)\}$.

TABLE R

RID	KEY
1	25
2	14
3	19
4	18
5	7

TABLE S

KEY	RID
7	1
19	2
13	3
12	4

Fig. 1: Example of join tables.

4.2 Sort Phase

One of the critical parts of the sort-merge join is the sorting of the two tables. For hardware implementation, it should be noted that executing a full sorter is naturally more resource-intensive and time-consuming than executing a partial sorter. Specifically, when a full sorter is applied, the merge phase has to wait until the whole sorting is completed. By contrast, if a partial sorter is applied, the merge phase for each part can start immediately after the sorting for that given part is finished. That is to say, the partial sort for the next part and the merge phase for the previous part can run simultaneously and independently, resulting in further improvement of efficiency. Thus, in this study, we apply the top-K sorter, as a partial sorter, in the sort phase instead of a full sorter, unlike in other sort-merge join algorithms.

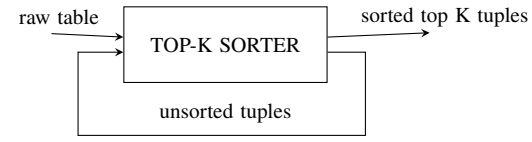
The top-K sorter in our architecture can process continuous sequences of any variable length and output the sorted top K elements for each sequence together with the remaining unsorted elements. In this way, we realize the full sorting of the whole table, which sort-merge join requires, by calling the top-K sorter multiple times. Specifically, we input the whole table into the top-K sorter in the first round and obtain the sorted top K tuples with the other unsorted tuples. Then, we input the unsorted tuples as a new sequence and obtain the secondary top K sorted tuples. This step is repeated until no unsorted tuples remain, i.e., the whole table is sorted out. In this way, the fully sorted result is eventually formed as needed. For example, as shown in Fig. 2, we are given a table that has five tuples listed as $\{(1,25),(2,14),(3,19),(4,18),(5,7)\}$, where the first number in each tuple is the rid, and the second is the key. We then use a top-2 sorter in the sort phase and sort the tuples in descending order. The procedure of the sort phase is as follows:

(a) In the first round, we input the whole table into the top-2 sorter, and obtain $\{(1,25),(3,19)\}$ correctly sorted (highlighted in gray cells).

(b) In the second round, we input the remaining three tuples, and obtain $\{(4,18),(2,14)\}$ correctly sorted.

(c) Lastly, in the final round, we input the last tuple $\{(5,7)\}$ and obtain it as is.

As a result, we obtain the fully sorted table as $\{(1,25),(3,19),(4,18),(2,14),(5,7)\}$. The sort order of the top-K sorter, i.e., in ascending or descending order, should be selected according to the θ operator.



RID	KEY
1	25
2	14
3	19
4	18
5	7

(a)

RID	1	3	2	4	5
KEY	25	19	14	18	7

(b)

RID	1	3	4	2	5
KEY	25	19	18	14	7

(c)

RID	1	3	4	2	5
KEY	25	19	18	14	7

Fig. 2: Algorithm and example of the top-K sort.

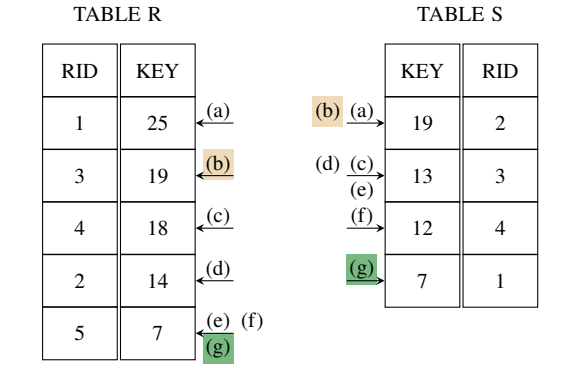
4.3 Merge Phase

Another essential part of the sort-merge join is the merging of two sorted tables. Herein, we designed two types of merge strategies for the merge phase: one is the equi-merge, for equi-join; and the other is the range-merge, aimed for use with θ-join. Although the only difference between the two strategies is in the comparison condition, which are equality and non-equality, respectively, the range-merge has a higher complexity than that of equi-merge.

4.3.1 Equi-merge

In equi-merge, we read one tuple separately from each of the two sorted tables and check if they have equal keys. If not, we move the larger (resp., smaller) one to the next tuple when the table is sorted in descending (resp., ascending) order. Otherwise, the common key and equal tuples are recorded in a temporary storage, and then we enter into the comparison subroutine. In this subroutine, we operate the two tables separately. For each table, we obtain the next tuple to determine if the key of the new tuple is equal to the common key, and store the equal one in the temporary storage; this step is repeated until the key of the new tuple is no longer equal to the common key or if we reach the end of the table. After these independent comparisons are completed for both tables, we make a Cartesian product of tuples in the temporary storage and export the join matches. Afterward, we continue to read one tuple from each of the two tables and check for equality. Equi-merge is terminated once at least one of the tables is empty. In the example shown in Fig. 3, the sorted keys of table R are $\{25,19,18,14,7\}$,

and the sorted keys of table S are $\{19, 13, 12, 7\}$. Note that we have omitted the rid for description convenience. The procedure of equi-merge can be executed as follows and the pseudo-code is shown below:



Temporary Storage	Step	b	g
	Common key	19	7
	Table R	(3, 19)	(5, 7)
	Table S	(2, 19)	(1, 7)
	Join Match	(r_3, s_2)	(r_5, s_1)

Fig. 3: Example of equi-merge.

Algorithm 1 Equi-merge

```
Require: table S, table R
  if unsort(L) then
      sort(L)
  if unsort(R) then
      sort(R)
  i ← 0, j ← 0, result ← Φ
  n_l ← length of L, n_r ← length of R
  while i != n_l or j != n_r do
      if s_i.key == r_j.key then
          result ← result + {s_i, r_j}
          j++
      elseif s_i.key > r_j.key
          i++
      else
          j++
  while i != n_l do
      if s_i.key == r_j.key then
          result ← result + {s_i, r_j}
      i++
  while j != n_r do
      if s_i.key == r_j.key then
          result ← result + {s_i, r_j}
      j++
  return result
```

(a) Read the top tuple from each of the two tables, i.e., 25 from table R, and 19 from table S, then compare them, and find that they are not equal. Note that the step letters with arrows in the figure indicate that the tuple pointed at by the arrow is read at the corresponding step; the same applies to subsequent figures.

(b) Read the next tuple, i.e., 19, from table R (because $25 > 19$), compare it with the tuple 19 from table S, and find that they are equal. Enter the comparison subroutines (highlighted in light yellow):

i. Record the common key and equal tuples in the temporary storage.

ii. Read the next tuple, i.e., 18, from table R, compare it with the common key, and find that they are not equal; thus, exit the comparison subroutine for table R.

iii. Read the next tuple, i.e., 13, from table S, compare it with the common key, and find that they are not equal either; thus, exit the comparison subroutine for table S.

iv. After both comparison subroutines are finished, make a Cartesian product from the temporary storage and export the join match $\{(r_3, s_2)\}$.

(c)-(f) Continue to compare the tuples of the two tables to check if they are equal, and find that they are not all equal (namely, 18 vs. 13; 14 vs. 13; 7 vs. 13; 7 vs. 12).

(g) Read the next tuple, i.e., 7, from table S, compare it with 7 of table R, and find that they are equal. Enter the comparison subroutines as in step (b) (highlighted in green). Because the two tables are now both empty, the equi-merge of these two tables is terminated.

Finally, after equi-merging of the two tables is completed, we obtain join matches $\{(r_3, s_2), (r_5, s_1)\}$. It is easy to determine that the time complexity of the equi-merge algorithm is

$$O_{equi\_merge} = O(N_R + N_S) + O_{export\_match}, \quad (1)$$

because each step will take a new tuple from either table R or table S. Here, N_R and N_S are the sizes of tables R and S, respectively. For exporting matches, the time complexity depends on the number of results for each match and the interval of each adjacent match. For each common key, the match exporting has an $O(n_r \times n_s)$ time complexity, where n_r and n_s denote the numbers of tuples equal to this common key in tables R and S, respectively, and the symbol $\times$ means Cartesian product; the product here is also the number of matches to this common key. Moreover, because of the use of a temporary storage, we can parallel the exporting of the temporary storage and the comparison of the next pair of tuples to hide the exporting cost and further improve the performance. However, if two common keys are found to be excessively close, i.e., the interval between two adjacent matches is excessively small, the parallelism will be affected because of the recycling of the temporary storage. Nonetheless, overall, the match-exporting time complexity

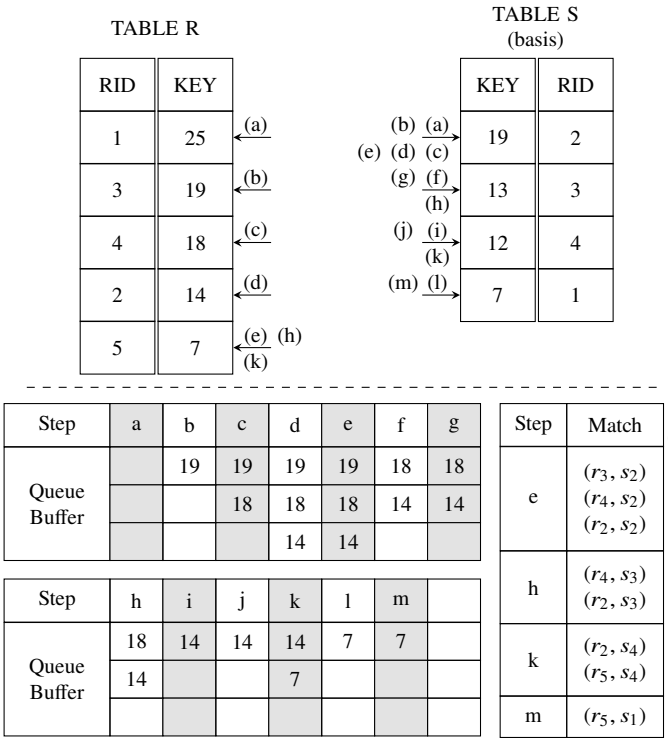


Fig. 4: Example of range-merging.

in equi-merge can be neglected for most applications.

4.3.2 Range-Merge

With regard to the range-merge, the procedure is a slightly more complex. Herein, we consider one table as the comparison basis, from which tuples will be compared sequentially with tuples from the other table. Furthermore, we record the match tuples for the basis table with the help of a Queue buffer, to which each tuple of the other table will be inserted only once at most. Specifically, for each tuple from the basis table (referred to as $tuple_{cur}$ temporarily), we follow these three steps to execute a comparison:

① Compare $tuple_{cur}$ with the head of the Queue buffer (if existing) to check whether they match the join condition or not. If not, pop the head out and repeat this step; otherwise, turn to the next step.

② Compare $tuple_{cur}$ with the new tuple from the other table to check if there is a new match. If yes, insert the new tuple into the Queue buffer and repeat this step; otherwise, finish the comparison for $tuple_{cur}$.

③ Pair $tuple_{cur}$ with all the tuples from the Queue buffer and export the join matches.

Fig. 4 shows an example of range-merge based on the tables from the previous examples. The sorted keys of table R are $\{25, 19, 18, 14, 7\}$, whereas the sorted keys of table S are $\{19, 13, 12, 7\}$. We consider table S as the comparison basis (i.e., the basis table), and compare each of its tuples with the whole table R using a Queue buffer. The match condition is that the absolute value of the difference between the keys of the two tuples is smaller than 5, i.e., $|key_R - key_S| \leq 5$. The procedure of range-merge is executed as follows:
Initially, the pointers and Queue are all $null$s.

(a) Read the first tuple, i.e., 19, of table S as $tuple_{cur}$, and begin the range-merge for this tuple. Because the Queue is presently empty, we skip step ① mentioned earlier, and read the first tuple, i.e., 25, of table R. Compare the two tuples, and find that they are unmatched (19 vs. 25). At this point, the Queue is still $null$.

(b)-(d) Read the next tuple, i.e., 19 (resp. 18, 14), of table R, and compare it with $tuple_{cur}$; find that they are all matched (i.e., 19 vs. 19; 18 vs. 19; and 14 vs. 19), and insert them (i.e., 19, 18, and 14) into the Queue.

(e) Read the next tuple, i.e., 7, of table R, and compare it with $tuple_{cur}$; find that they are unmatched, and thus the comparison for this $tuple_{cur}$ is finished. Then, as in step ③ described earlier, we export the join matches as $\{(r_3, s_2), (r_4, s_2), (r_2, s_2)\}$.

(f) Read the next tuple, i.e., 13, of table S as $tuple_{cur}$, and begin the range-merge for this tuple. As in step ① described earlier, we first check if the head tuple of the Queue satisfies the match condition for current $tuple_{cur}$, and find that 19 is unmatched, which should be popped out.

(g) Repeat checking the head tuple of the Queue, and find that 18 and 14 also matches with the new $tuple_{cur}$; turn to step ② for $tuple_{cur}$.

(h) Read the new tuple, i.e., 7, of table R, and compare it with $tuple_{cur}$. Find that they are unmatched, and thus the comparison for this $tuple_{cur}$ is finished. Then, the join matches listed as $\{(r_4, s_3), (r_2, s_3)\}$ are exported.

(i)-(k) Read the next tuple, i.e., 12, of table S as $tuple_{cur}$. After the execution of steps ①-②, we pop 18 out of and insert 7 into the Queue, and export the join matches as $\{(r_2, s_4), (r_5, s_4)\}$.

(l) Read the next tuple, i.e., 7, of table S as $tuple_{cur}$, and compare it with the head of the Queue (i.e., 14); find that they are unmatched, and pop the 14 out of the Queue.

(m) Repeat the comparison between $tuple_{cur}$ and the head of the Queue (i.e., 7), and find that they are matched.

(n) Now that no more tuples remain to be compared, export the last join match as $\{(r_5, s_1)\}$, and finish the whole merge procedure.

Finally, eight pairs that match the join condition are exported. The theoretical time complexity of the range-merge algorithm is

$$O_{range} = O(N_R + N_S) + O_{export\_match}. \quad (2)$$

The first part, $O(N_R + N_S)$, can be obtained easily based on calculating the amounts of time for reading the tuples, because the tuples of table S are traversed once, whereas each tuple of table R is inserted into or popped out of the Queue only once at most. On the other hand, the second part, i.e., the match-exporting time complexity, depends on the number of results per match, and can be expressed as follows:

$$O_{export\_match} = \sum O(n_r \times n_s). \quad (3)$$

Note that this method performs better in band-range merges, i.e., where the match condition is that the difference/distance between two elements should be smaller than a threshold. In problems involving matches where one element is smaller/bigger than or not equal to the other element, the time complexity of range-merge will increase to $O(N_R \times N_S)$ because the n_r and n_s for each matching will approximate N_R and N_S, respectively.

5. Proposed hardware architecture

Because FPGA is good at realizing high-parallelism and deep-pipelined work, we propose a new architecture for accelerating the sort-merge join operation based on the algorithms described in Section 4.

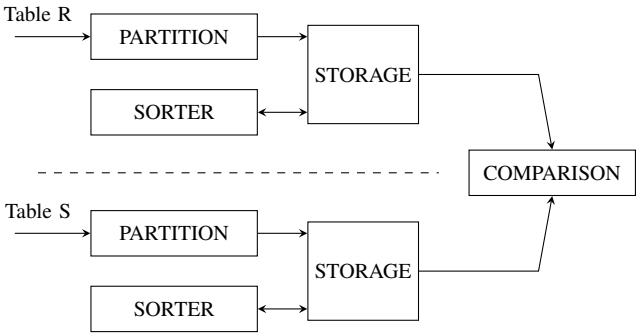


Fig. 5: The proposed join architecture.

5.1 System Components

The join architecture, shown in Fig. 5, consists of three components apart from **storage**, namely the **partition**, **sorter**, and **comparison** components.

5.1.1 Partition Component

The partition component reads the data from data sources and divides the data into several parts according to user-defined coarse-grained partition rules, such as feature-based partition methods, data-distribution-based partition manners, and hash-based partition functions. A good partition component can help further accelerate the join operation because it performs the preliminary filtering and reduces the number of tuples that should be sorted and merged for each part. Additionally, the parallelism of our architecture depends on the partition component. More specifically, the parallelism level is equal to the number of partition modules in the partition component, and that number can be arbitrary.

5.1.2 Sorter Component

The sorter component consists of several top-K sorters, each of which processes the data of one partition part correspondingly after the partition component. Therefore, the number of top-K sorters is equal to the number of partition parts. After a thoroughgoing research of top-K sorter architecture, we adopt the serial pipelined sorting architecture proposed in [30] as the default for each top-K sorter composed of K identical sorting cells. We also implement the top-K sorter architecture proposed in [31] to demonstrate the modular design of our architecture. Note that the selection of the parameter K is a trade-off between resource utilization and sort efficiency. A larger value for K denotes fewer repetitions of the rounds mentioned in Section 4.2 but requires a higher logic usage.

5.1.3 Comparison Component

The comparison component is the last step in the process and generates the final join match results. The sorted tuple lists for the corresponding parts of the two tables (for example, both the first parts of the two tables) are compared and merged in one comparison module to find matches according to the merge algorithms. Therefore, the number of comparison modules is equal to the number of top-K sorters, the same as for the partition modules.

Above all, the entire architecture we proposed is of a modular design. The specific method for each component can be selected and optimized according to the characteristics of the tables waiting to be joined, to obtain better performance in different application scenarios.

5.2 Join Process

The processing by the proposed architecture is fully pipelined, and tuples go through the three aforementioned components sequentially. Furthermore, the proposed algorithm can be paralleled in arbitrary threads to improve performance. Specifically, the tuples of two tables are partitioned and sorted simultaneously in parallel, followed by comparisons between corresponding sorted parts to produce the join matches.

Fig. 6 provides an example of the architecture for a parallelism of three. The partition, sorter, and comparison components are marked with yellow, green, and red rectangular blocks, respectively. Because of the limitation in figure size, we show only the detailed modules for the processing of table R, which is indicated with a dashed line; the omitted hardware design, for the processing of table S, is the same.

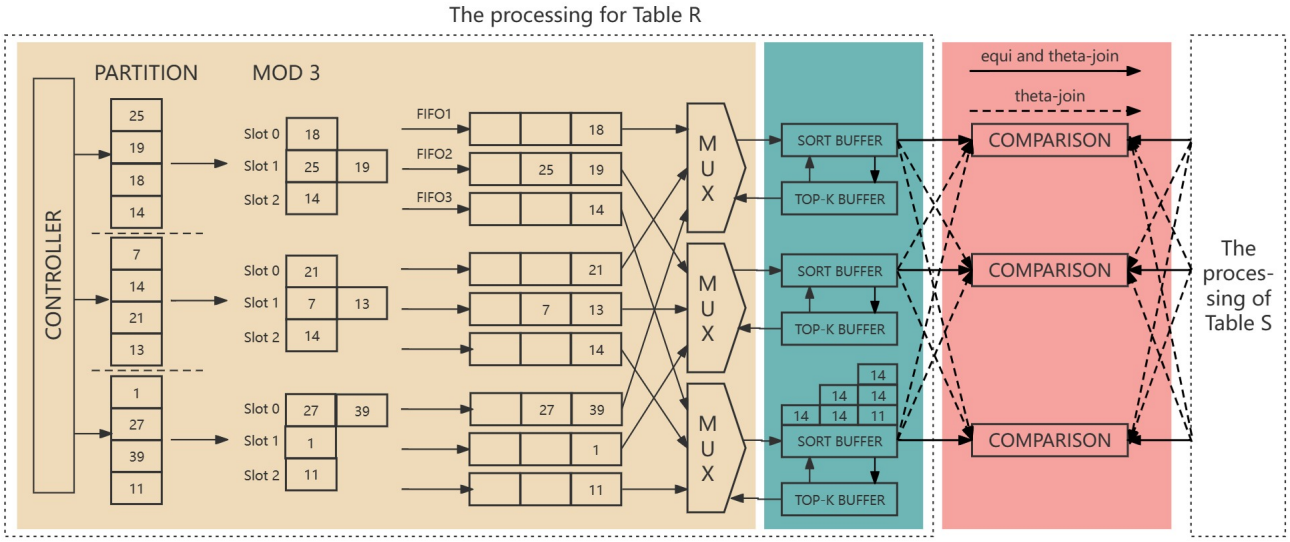


Fig. 6: Example of the architecture when parallelism is three.

As described in Section 5.1, the partition component continuously fetches tuples from the data sources and distributes the tuples to their partition First In First Outs (FIFOs) according to pre-defined partition rules. In this example, the value of data in table R is shown in partition module. The hash function is the value mod 3. After the partition, the value in table R will be hashed into three different slots. And that is also the evidence of partition FIFOs. Here, the partition FIFOs are used to catch the tuples after the partition functions to resolve the time consumed on partitioning and sorting inconsistency. Because there are three identical partition modules for each table, the central controller will cyclically call the three partition modules to balance the number of tuples for three partition branches. After the partitioning, three multiplexers (MUXs) wait to collect the tuples from the same parts and send them to the corresponding top-K sorter. In this example, we can see there are two 14 in value list, they are divided into part1 and part2, but they also get into slot2. That means they are both sent to the MUX2. After the MUX, the top-k buffer will sort the same FIFO from differnet parts. And the processing is from 14 to 14,14, and 14,14,11, finally. The top-K sorters yield the sorted tuples to sort buffers for subsequent comparison and output the remaining unsorted tuples to unsorted storage for re-sorting. Here, sort buffers are where the tuples are stored originally; we recycle the storage to store the sorted and unsorted tuples. The sorted tuples in the sort buffers will be merged with the corresponding sorted buffers of the other table. The comparison and merging of two sort buffers can start when they are both non-empty. Each comparison module fetches tuples from corresponding paired sort buffers, checks if they fulfill the join conditions, and outputs the join matches. In addition to these, we use dual-port block RAMs (BRAMs) for the storage to execute the operation of reading and writing independently and ensure a pipelined workflow.

In comparison, it have different mode for equi-join and theta-join. In equi-join comparison, the data only need to compare with the corresponding part in table S. In theta-join part, the data need to compare with all the parts in table S. The two situations are shown in different arrows.

5.3 Complexity Analysis

5.3.1 Time Complexity

The time complexity of the proposed architecture consists of three parts, which correspond precisely to the three components.

For the partition component, it is easy to pipeline the partition function, e.g., hash functions. Thus, the time complexity of the partition component can be readily obtained as $O(N)$. Here, N refers to the total number of data points for each table which means $N = N_R + N_S$.

On the other hand, for the sort component, as described in [30], if $N > K + 1$, the top-K sorter can produce the top K elements of the input sequence after $N + K$ clock cycles, and if $N \leq K + 1$, the sorted elements will come out after a constant delay of $2K$ clock cycles. Thus, the time complexity of sorting out the top K elements in N tuples is $O(N)$, leading to a total time complexity for our sort component of $O(N \times \frac{N}{K})$, because of the multiple execution times (i.e., $\lceil \frac{N}{K} \rceil$) of the top-K sorter.

Based on the analysis presented in Section 4.3.1 and 4.3.2, the time complexity of comparison in equi-merge and range-merge both approximate $O(N) + O_{export\_match}$.

However, the architecture is fully pipelined, and thus the time complexity should be the maximum among all components, which can be expressed as

$$\begin{aligned} O_{overall} &= \max\left(O_{partition}, O_{sorter}, O_{comparison}\right) \\ &= \max\left(O(N), O(\frac{N^2}{K}), O(N) + O_{export\_match}\right) \\ &= O(\frac{N^2}{K}). \end{aligned} \tag{4}$$

If there are P parallelisms taken into account, the total number of each branch should be $\frac{N}{P}$. Thus, the time complexity of the parallel architecture should be similar to $O(\frac{N^2}{P^2 \times K})$.

5.3.2 Space Complexity

The space complexity (i.e., resource utilization complexity) of the proposed architecture is much easier to analyze. The main part of the logical resource is used on the sort component, of which the space complexity is $O(P \times K)$. Because the original storage of tables can be recycled during sorting, and because no additional storage is needed, the space complexity of storage is also $O(N)$, which is not mentioned later, like other sort-merge join approaches.

6. Experimental results and discussion

This section discusses the complexity analysis of the proposed architecture, and the performance evaluation in terms of throughput and resource utilization. Further, the proposed method is compared with existing join approaches.

6.1 Experimental Setup

Our design was implemented on a Xilinx Virtex-7 FPGA using Xilinx Vivado 2019.2. We use v7 690t as the experiment platform, and the detail model is xc7vx690t, in particular. A $\{rid, key\}$ generator was used; the rid and key were set to integers for simplification, and the key was set to 32 bits wide. The generator is used in both basis and compared tables. Tuples were streamed into the architecture one by one. Table sizes were varied from $4K$ to $64K$ for performance evaluation. A CRC hash function was used as the partition method. The depths of the partition FIFOs, which are used as caches, were set to 16. We use the result of hash functions to make the partition, that means the same value will be divide into the same partition. We realized and experimented with two sorting architectures to demonstrate the modular design, as described in Section 5.1. On the other hand, the sorter component adopted the sorting architecture proposed in [30] as its default and obtained the performance correspondingly, unless otherwise stated. Because there is a trade-off in setting the K-values, we tested different K-values, i.e., 32, 64, 128, 256, and 512, for the sort component. With regard to the comparison component, we implemented two merge methods, namely equi- and range-merges, yielding matches for equi- and θ-joins, respectively. For θ-join, we assumed a match condition such that the absolute value of the difference between the keys of two tuples is smaller than 5, as has been demonstrated previously in Section 4.3.2. However, regardless, the throughput of our architecture is calculated based on the equi-join without additional statement. Moreover, we constructed parallelisms P=2, 4, 8 for comparison and observed the performance improvement. We conducted these experiments under the condition that the two tables have the same size, for simplification and fair comparison (i.e., N_R=N_S=N). Meanwhile, we measured the performance based on how many millions of tuples can be processed in one second by dividing the total number of tuples by the total time required to obtain the average throughput.

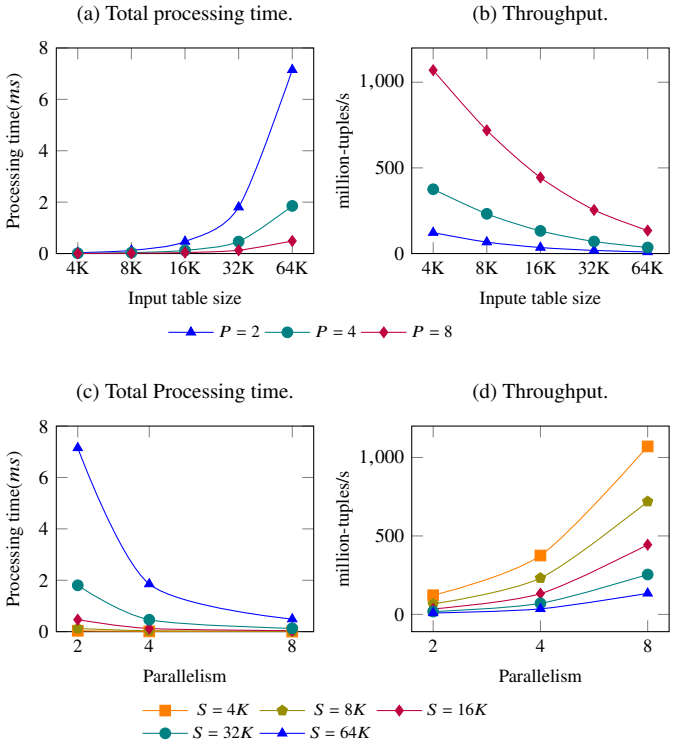


Fig. 7: Performance of different parallelisms with various input table sizes when $K = 64$.

6.2 Results for Different Parallelisms

In this part of the study, we varied the parallelism to check for performance improvement from increasing the parallelism level. Because the length of each tuple is 96 bits, we can use million-tuple per second as the unit. Fig. 7a shows the total processing times for various parallelisms with respect to input table sizes when K-value = 64, whereas Fig. 7b shows the corresponding throughputs. For comparison, Fig. 7c and Fig. 7d visualize the same performance, except that the horizontal axis denotes the parallelism, to demonstrate the performance improvement with respect to parallelism more intuitively. We can observe that when the processing tables are of the same size, then as the parallelism doubles while the K-value is kept unchanged, the total processing time is reduced to nearly a quarter, and approaches

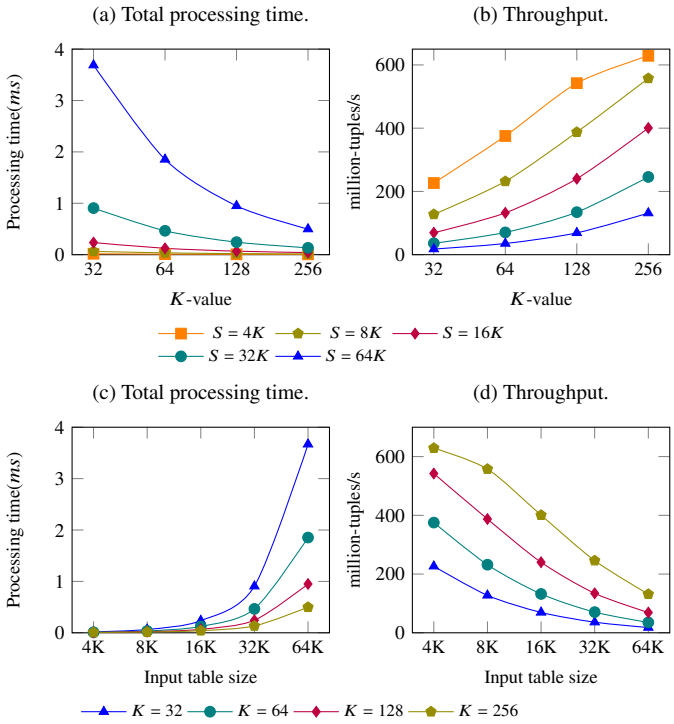


Fig. 8: Performance of different K-values with various input table sizes when $P = 4$.

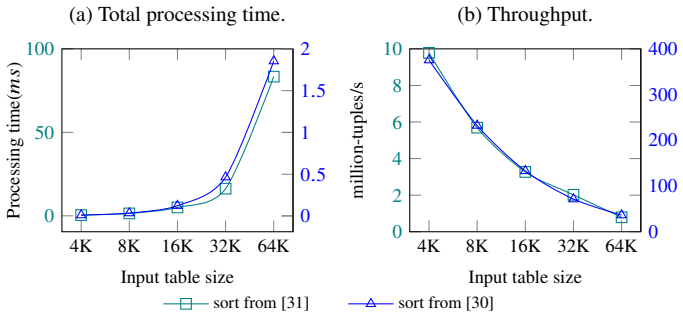


Fig. 9: Performance of different sort architecture which has been proposed in [30] and [31] with various input table sizes.

a quarter as the table size grows. Correspondingly, the improvements in throughput are similar, i.e., as the parallelism doubles while the K-value is kept unchanged, the throughput quadruples. Moreover, based on the first two subfigures in Fig. 7, when the parallelism is kept unchanged, the processing time nearly quadruples as the input table size grows doubly, while the throughput is reduced to half. The results can be expressed as

$$\begin{aligned} time &\sim \frac{N^2}{P^2}, \\ throughput &\sim NP^2. \end{aligned} \tag{5}$$

These formulas are consistent with the time complexity analyzed in Section 5.3, i.e., $O(\frac{N^2}{P^2 \times K})$.

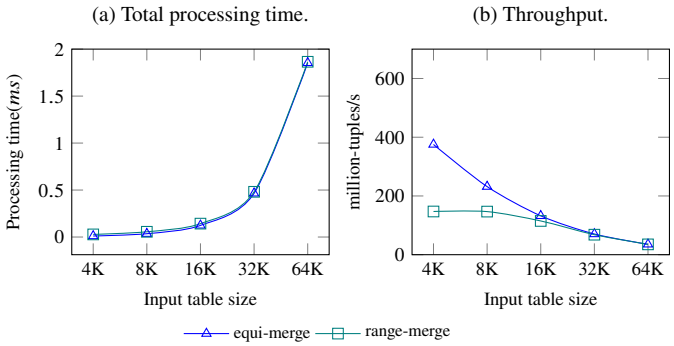


Fig. 10: Performance of different merge methods with various input table sizes when $P = 4$ and $K = 64$.

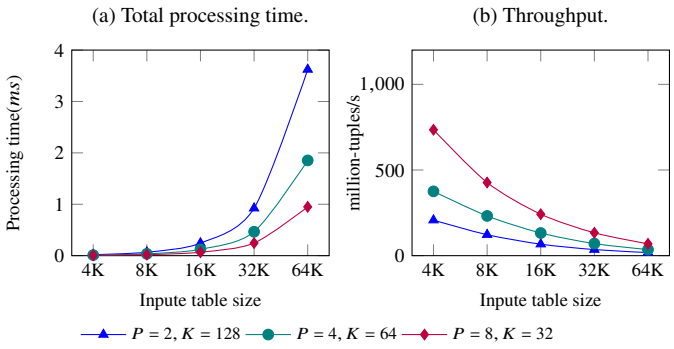


Fig. 11: Performance of different combination of parallelism and K-values with various input table sizes.

Table 2: Resource utilization and processing time consumption for each component.

Config	Component	LUT	FF	Time(s)
$P = 2, K = 128$	Partition	269	62	15.55
	Sorter	35795	62043	244.38
	Merge	1990	416	229.70
	Total	38054	62521	244.40
$P = 4, K = 64$	Partition	612	62	7.67
	Sorter	38662	65262	123.67
	Merge	3979	832	116.14
	Total	43253	66156	123.88
$P = 8, K = 32$	Partition	1146	62	3.88
	Sorter	48198	76226	67.50
	Merge	7961	1664	53.17
	Total	57305	77952	67.69

6.3 Results for Different Sorters

First, we adjusted the setting of the K-value and observed the performance change trends therewith. Fig. 8 shows the performance for different K-values with respect

Table 3: Comparison between the proposed method and previous methods.

Method	Platform	Frequency (MHz)	LUT	FF	Slice[1]	BRAM (KB)	Power[2] (W)	Throughput (million-tuples/s)	Ratio1[3] ($\times 10^{-3}$)	Ratio2[4]
[32], NLJ	Xilinx Virtex-6	238	1362	-	341	203	0.343	0.12	0.35	0.35
[32], AHJ	Xilinx Virtex-6	200	1200	-	300	135	0.283	1.8	6.00	6.36
[32], SHJ	Xilinx Virtex-6	167	1509	-	377	203	0.295	1.25	3.32	4.24
[20]	Xilinx Zynq	100	42609	27838	10653	104	0.397	85	7.98	214.11
[26], $T\overset{5}{=}4$	Xilinx Zynq	349	1706	2238	427	186	0.599	14.9	34.89	24.87
[26], $T=8$	Xilinx Zynq	261	3744	4842	936	208	0.774	28.9	30.88	37.34
[26], $T=16$	Xilinx Zynq	261	7820	10050	1955	250	1.302	62.4	31.92	47.93
[33], $K=128$	Xilinx Virtex-7	276	18176	7296	4544	-	0.514	38.8	8.54	75.51
[33], $K=256$	Xilinx Virtex-7	266	36352	14592	9088	-	0.746	65.6	7.22	87.91
[33], $K=1024$	Xilinx Virtex-7	248	145408	58368	36352	-	2.028	162.1	4.46	79.92
Ours, $P=2$, $K=128$	Xilinx Virtex-7	300	38054	62521	9514	-	1.213	67.1	7.05	55.32
Ours, $P=4$, $K=64$	Xilinx Virtex-7	300	43253	66156	10814	-	1.315	132.3	12.23	100.61
Ours, $P=8$, $K=32$	Xilinx Virtex-7	300	57305	77952	14327	-	1.576	242.1	16.90	153.62

[1] It was assumed that slices are fully used as LUTs and registers.
[2] Power was estimated using the Xilinx Power Estimator based on the platform, clock frequency, and resource consumption provided in the table.
[3] Ratio1 refers to the ratio of throughput to slices.
[4] Ratio2 refers to the ratio of throughput to power.
[5] The number of binary-tree model in parallel tree-based join architecture.

to various input table sizes while the parallelism is set to 4. Specifically, Fig. 8a illustrates the processing times for different K-value settings, whereas Fig. 8b shows the corresponding throughputs. It can be concluded that the processing time for the same table is reduced to half as the K-value is doubled, and thus the throughput is doubled accordingly. Fig. 8c shows the processing times for different input table sizes for each K-value, whereas Fig. 8d shows the corresponding throughputs. Similar to the results shown in the previous subsection, when the K-value is kept unchanged, the processing time nearly quadruples as the input table size grows doubly, while the throughput is reduced to half. Thus, the overall inference can be expressed as follows:

$$\begin{aligned} time &\sim \frac{N^2}{K}, \\ throughput &\sim NK. \end{aligned} \tag{6}$$

Eq. 6 further proves the theoretical time complexity $O(\frac{N^2}{P^2 \times K})$.

In addition, to verify the design criteria for the modularity, we also implemented the top-K sorter proposed in [31] and tested its performance. Fig. 9 shows a performance comparison of the two different sort architectures. Note that the two curves in the same plot use the opposite sides of the graph for their respective ordinates. The difference in value is caused by differences in the sort architecture, which led to different realized frequencies for the hardware, i.e., 300 MHz and 6.67 MHz, respectively, for the two aforementioned sort architectures. As shown in the figure, if the subtle numerical differences could be ignored, these two curves will be almost identical. This phenomenon indicates that any top-K architecture can replace the detailed implementation of the sorter component. Also, the performance of the top-K sorter architecture will greatly influence the performance of the entire system because the sorter component is the most time-consuming part of the architecture.

6.4 Results for Different Merge Methods

In this part of the study, we implemented the two proposed merge algorithms and tested their performance in different join scenarios. Fig. 10 shows the performance of equi-merge and range-merge in terms of processing time and throughput, respectively. As shown in Fig. 10a, the change pattern in the processing time of range-merge for θ-join is similar to that of equi-merge for equi-join. Furthermore, as the input table size gets larger, the gaps between the processing times in Fig. 10a and between the throughputs shown in Fig. 10b for the two merge methods become smaller. This is because, as the tables grow, the bottleneck is transferred from merging to sorting, i.e., when big tables are processed, the cost of merging can be covered by that of sorting.

6.5 Utilization and Performance Comparisons

Based on the observation that the performance increases as the parallelism and/or K-value is increased, combined with the fact that the resource utilization also increases as the parallelism and/or K-value is increased, there should be a trade-off between resource utilization and hardware performance. Therefore, in this section, we tested the performance of different combinations of parallelism and K-values such that the product of these two parameters is kept unchanged, to be able to observe the real benefits of increasing parallelism. Based on the results shown in Fig. 11, we can conclude that when the product of parallelism and K-value is kept constant, the decrease in processing time and the increase in throughput equally becomes nearly close to a factor of P.

Similarly, this conclusion can be derived from the theoretical time complexity $O(\frac{N^2}{P^2 \times K})$. We list the resource utilization and time consumed for each component in Table 2. As shown in Table 2, the sorter component is always the most dominant part in terms of complexity, whereas the merge component is the second, because it needs to wait for the sorting results in order to perform the merging.

Table 3 shows a detailed comparison of the proposed method and several state-of-the-art methods for FPGA-based join accelerators. To ensure fair comparisons, we present the throughputs of these methods for a table size of $16K$. The ratio of throughput to slices and that to power are considered as indicators of effective resource utilization. As mentioned before, sort-merge-based join architectures do not require additional storage, leading to zero BRAM usage. From Table 3, it can be observed that our design achieved a throughput of 242.1 million tuples per second when $P = 8, K = 32$, which is the highest among all evaluated methods. The ratio1 and ratio2 for the proposed method are the second-highest. It also takes a great trade-off between power consumption and throughput, although the power consumption is larger than previous works. For comparison, the researchers in [32] implemented different join algorithms in large semantic web databases and exhibited the lowest throughput with the least resource utilization. The approach of [20] operated only the sort phase on FPGA with the help of merging on CPU, and thus obtained the highest ratio of throughput to power. The researchers in [33] also used the top-K sorter instead of full sorting to improve the performance of their method; however, unlike in our method, their proposed architecture cannot benefit from parallelism. The approach of [26] obtained the highest ratio of throughput to slices, but the performance is not stable enough for the skewed input data.

7. Conclusion

This paper presents a novel architecture that is highly paralleled and deeply pipelined to accelerate the sort-merge join operation used in database systems. Unlike traditional sort-merge algorithms, we use a top-K sorter to reduce resource consumption and achieve synchronization with the merge phase. For the merge phase, we propose two comparison units compatible with different types of join conditions and realize a general join operation accelerator. The proposed architecture was shown to provide high throughput and superior resource utilization efficiency. Furthermore, the architecture is of a modular design, which can be tailored as needed, thereby allowing performance enhancement in specific scenarios after targeted adjustment or even replacement.

Acknowledgement

This paper has been supported by the Fundamental Research Funds for the Provincial Universities of Zhejiang under Grant GK219909299001-22.

References

[1] P. Papaphilippou and W. Luk, "Accelerating database systems using fpgas: A survey," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 125–1255, IEEE, 2018.

[2] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 13–24, IEEE, 2014.

[3] I. Koumarelas, A. Naskos, and A. Gounaris, "Flexible partitioning for selective binary theta-joins in a massively parallel setting," *Distributed and Parallel Databases*, vol. 36, no. 2, pp. 301–337, 2018.

[4] H. Wang, N. Li, Z. Wang, and J. Li, "Gpu-based efficient join algorithms on hadoop," *The Journal of Supercomputing*, vol. 77, no. 1, pp. 292–321, 2021.

[5] H. Roh, M. Shin, W. Jung, and S. Park, "Advanced block nested loop join for extending ssd lifetime," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 4, pp. 743–756, 2017.

[6] A. Nguyen, M. Edahiro, and S. Kato, "Gpu-accelerated voltdb: A case for indexed nested loop join," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 204–212, IEEE, 2018.

[7] W.-Q. Wu, M.-T. Xue, Q.-J. Xing, and F. Yu, "High-parallelism hash-merge architecture for accelerating join operation on fpga," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2021.

[8] Z. Zhou, C. Yu, S. Nutanong, Y. Cui, C. Fu, and C. J. Xue, "A hardware-accelerated solution for hierarchical index-based merge-join," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 1, pp. 91–104, 2018.

[9] J. Fang, Y. T. Mulder, J. Hidders, J. Lee, and H. P. Hofstee, "In-memory database acceleration on fpgas: a survey," *The VLDB Journal*, vol. 29, no. 1, pp. 33–59, 2020.

[10] X. Sun, J. Yu, Z. Zhou, and C. J. Xue, "Fpga-based compaction engine for accelerating lsm-tree key-value stores," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 1261–1272, IEEE, 2020.

[11] L. Woods, Z. István, and G. Alonso, "Ibex: An intelligent storage engine with support for advanced sql offloading," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 963–974, 2014.

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

[13] D. Li, L. Huang, T. Gao, Y. Feng, A. Tavares, and K. Wang, "An extended nonstrict partially ordered set-based configurable linear sorter on fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 5, pp. 1031–1044, 2020.

[14] D. Li, R. Du, Z. Liu, T. Yang, and B. Cui, "Multi-copy cuckoo hashing," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1226–1237, IEEE, 2019.

[15] J. Zhan, W. Jiang, Y. Li, J. Wu, J. Zhu, and J. Yu, "Accelerating queries of big data systems by storage-side cpu-fpga co-design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 7, pp. 2128–2141, 2022.

[16] Ç. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on modern processor architectures," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1754–1766, 2014.

[17] K. Huang, "Multi-way hash join based on fpgas," 2018.

[18] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *Proceedings of the VLDB Endowment*, vol. 7, no. 1, pp. 85–96, 2013.

[19] J. Paul, B. He, S. Lu, and C. T. Lau, "Revisiting hash join on graphics processors: a decade later," *Distributed and Parallel Databases*,

vol. 38, no. 4, pp. 771–793, 2020.
[20] 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)*, pp. 212–219, IEEE, 2016.
[21] P. Papaphilippou, H. Pirk, and W. Luk, "Accelerating the merge phase of sort-merge join," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 100–105, IEEE, 2019.
[22] R. Rui and Y.-C. Tu, "Fast equi-join algorithms on gpus: Design and implementation," in *Proceedings of the 29th international conference on scientific and statistical database management*, pp. 1–12, 2017.
[23] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "Hardware-conscious hash-joins on gpus," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 698–709, IEEE, 2019.
[24] L. Qian, Z. Qu, M. Cai, B. Ye, X. Wang, J. Wu, W. Duan, M. Zhao, and Q. Lin, "Fastcache: A write-optimized edge storage system via concurrent merging cache for iot applications," *Journal of Systems Architecture*, vol. 131, p. 102718, 2022.
[25] T. I. Papon, J. Hyoung Mun, S. Roozkhosh, D. Hoornaert, A. Sanaullah, U. Drepper, R. Mancuso, and M. Athanassoulis, "Relational fabric: Transparent data transformation," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pp. 3688–3698, 2023.
[26] 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, 2018.
[27] A. Okcan and M. Riedewald, "Processing theta-joins using mapreduce," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 949–960, 2011.
[28] G. Li, J. He, D. Deng, and J. Li, "Efficient similarity join and search on multi-attribute data," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1137–1151, 2015.
[29] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and P. Kalnis, "Fast and scalable inequality joins," *The VLDB Journal*, vol. 26, no. 1, pp. 125–150, 2017.
[30] T. Chen, W. Li, F. Yu, and Q. Xing, "Modular serial pipelined sorting architecture for continuous variable-length sequences with a very simple control strategy," *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 100, no. 4, pp. 1074–1078, 2017.
[31] S. Dong, X. Wang, and X. Wang, "A novel high-speed parallel scheme for data sorting algorithm based on fpga," in *2009 2nd International Congress on Image and Signal Processing*, pp. 1–4, 2009.
[32] S. Werner, S. Groppe, V. Linnemann, and T. Pionteck, "Hardware-accelerated join processing in large semantic web databases with fpgas," in *2013 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 131–138, IEEE, 2013.
[33] W. Chen, W. Li, and F. Yu, "Modular pipeline architecture for accelerating join operation in rdbms," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 11, pp. 2662–2666, 2020.

Meiting Xue received the PhD degree in instrument science and technology from Zhejiang University, China, in 2020. She is an instructor with the Department of Cyberspace, Hangzhou Dianzi University. Her research interests include database acceleration embedded computer systems and data security governance.

Wenqi Wu received the B.S. degree from the College of Biomedical Engineering and Instrument Science, Zhejiang University, in 2017, where she is currently pursuing the Ph.D. degree. Her research interests include machine learning and novel acceleration architecture on hardware, especially for biomedical data processing.

Jinfeng Luo is pursuing the Ph.D. degree from the College of Biomedical Engineering and Instrument Science, Zhejiang University. His interests include hardware acceleration, network switching technology, network security technology, and digital signal processing.

Yixuan Zhang received the B.S. degree from the Department of Cyberspace, Hangzhou Dianzi University, in 2017, and he is currently working toward the master degree in the School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China. His research interests include approximate membership queries and network traffic identification.

Bei Zhao received the B.S. degree in electrical engineering, and the Ph.D. degree in instrument science and technology from Zhejiang University, Hangzhou, China, in 2000 and 2009, respectively. Currently, he is working as a lecturer with the School of Computer Science and Technology, Hangzhou Dianzi University, China. His research interests include wireless sensor networks, mobile computing, high performance embedded computing, digital signal processing and related aspects.