Design and Implementation of High-Throughput Parallel Hash Join Architecture on FPGA Based SRAM

Mr.K.Ch.Malla Reddy [(1)] Mr.A.Prasad [(2)] Rayalla Siva [(3)] Vallela Srinivasa Reddy [(4)] Yerva Sai Kumar Reddy [(5)] Tadi Venkata Sravana Kumar [(6)]

[1,2] *Krishna Chaithanya Institute Of Technology & Sciences, Ece Department, Markapur, Andhra Pradesh.*
[3,4,5,6,] *Krishna Chaithanya Institute Of Technology & Sciences, UG Student-ECE, Markapur, Andhra Pradesh.*

Abstract. In this paper, one of the most significant relational operations in a database is the hash join operator. For a long time, there has been growing interest in the technique of offloading and acceleration of this operation on hardware. However, because of the difference in the number of hash table accesses needed for each search, the non-uniform distribution of data resulting from hash collisions adversely impacts the throughput of the hash join technique. A non-collision parallel static random-access memory (SRAM)-based hash join architecture is provided to address this problem. To prevent hash collisions and guarantee a worst constant memory access for every stage of the hash join method, this design makes use of several hash functions and content addressable memories (CAMs). As a result, the hash join performance is increased. The experimental results demonstrate that our design achieved a high hash join throughput of 153.6 million tuples per second, a speedup factor of at least 2.5 with the best existing FPGA-based hash join architecture, and a match rate of 50%. The proposed architecture was implemented on a Xilinx Vivado field programmable gate array (FPGA).
Keywords: Ternary Content address memories(TCAMs), FPGA, Verilog HDL, Hash Join Table and Static Random Access Memory(SRAM).

I.INTRODUCTION

As programmable integrated circuit technology have advanced, there has been a growing movement to shift some CPU-intensive database operations—such as sorting [1], [2], aggregation [3], and [4]—to hardware, combine [5] with [7]. Join, a crucial database action, joins tuples from two or more relations using a common key. The sort-merge join, hash join, and nested-loop join are the three common join algorithms. Numerous studies have focused on the hash join approach since it can retrieve tuples in,time [8]– [11]. This algorithm's primary flaw is the erroneous data distribution brought about by hash collisions, which results in a mapping of distinct keys to the same index, which has a detrimental effect on the algorithm's throughput. Many hash join techniques, each with a different collision resolution strategy, have been proposed to address the aforementioned problem. One tactic like this is to put the identically indexed tuples into a chained list. A tuple is searched by scanning the chain until the key that corresponds to the tuple is located [5, 12, 13, 14]. A hash-join engine that distributes the first table across several channels using various hash functions takes advantage of parallelism, as demonstrated by Halstead et al. [12]. In order to lessen the imbalance between the left and right sub-trees of the root, H. Zhang et al. suggested a parallel tree-based join architecture in [13]. This architecture involves processing a binary tree in parallel. wherein a binary tree is processed concurrently to lessen the disparity between the root's left and right sub-trees. This strategy's query timings are dependent on in the worst situation, on the longest chain list.In a different

approach, robust hash functions were employed to generate balanced distributions; however, they are computationally costly [10], [14], and because totally uniformly distributed hash functions do not exist, whether they require reprocessing the tuples depends on the table load. Two distinct hash join accelerators were presented by Werner et al. [14], who used two hash tables running concurrently to reduce latency and use comparatively little resources. A fully parallel hash join was built by N. Devarajan et al. [10], and they obtained good performance; nonetheless, the design. We describe a non-collision parallel hash join design, inspired by [15]. A multiple hash table method is used in this design to distribute the tables in addition to a CAM to prevent hash collisions. To enhance the efficiency of the hash join technique, we investigate two crucial concerns in this design: memory utilization and the non-collision parallel hash join strategy. Tuples are kept on a separate SRAM to take advantage of pipelining and paralleling for hash join execution. Constant precise matching is accomplished by using hash table access time in parallel and the constant time exact matching operation in CAM. The main contributions of this research are as follows, in general a collision-avoidance parallel hash join technique is proposed. The proposed method, in contrast to [15], manages hash collisions and provides a worst-case constant latency for the insert and query operations of the hash join algorithm. A parallel hash join design incorporates a CAM along with several channels. This architecture, as opposed to [12], distributes the tuples among multiple hash channels, doing away with the need for redundant storage. We theoretically analyze our architecture's memory consumption, each hash channel's collision rate, and the capacity needs of the CAM.

II.PROPOSED SYSTEM

There are usually two stages to a hash join operation: the build phase and the probe phase. In the former, a hash table is filled with the data from the first table once it has been scanned using the hash function. Pairs. In the latter, to locate matched results, the hash table is probed, and the second table is read. The first and second tables in this work are taken to have N and M rows, respectively (N<M). The join architecture receives the tuples, which are a component of the integrated records, in the format {rid1,2, key}. The join attribute serves as the key, while the ride is utilized to uniquely identify a row. The second table streams in, producing the join results {Rid1, Rd2, Key}.

A. BUILD PAHSE

Consequently, throughout the build phase, each hash channel's memory access time stays constant.The FPGA fetches the first table's tuples in parallel during the build phase. Each channel's FIFO is used to cache incoming tuples from other channels or local storage. Each channel hashes the tuple's key, and the hash function of each channel saves the tuple's location at the corresponding address on the hash table. The hash tables store tuples in the pattern {Status, Rid, Key}. The status (1 bit) of each tuple indicates whether the row in the hash table is occupied. Each channel's conflicting hash values are shifted to the next, and this shift procedure continues until the pair is successfully placed into a hash table or crosses the shift threshold. after which it is transferred in a {rid, key} format to the CAM. The second table's key and the keys in the hash tables and CAM will be compared in the investigation stage. There are several hash channels, as Fig. 1 illustrates, and each channel has resources allotted to it, enabling parallel distribution in the first table during the build stage. Non-conflicting tuples in each hash channel

need two clock cycles to update the hash table (i.e., to write data and check if the corresponding address is available). In the event of a hash collision, conflicting tuples are sent to the FIFO for additional handling.

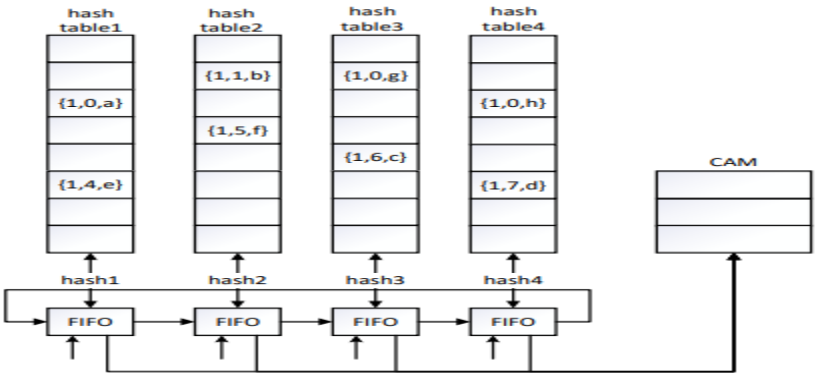

Fig.1. Example of hash join architecture with four hash channels.

B. PROBE PHASE and DATA SHIFT STRATEGY

The tuples in the second table are streamed in and compared with those in the first table to see if any results match during the probe phase. The rows are linked upon the discovery of a match. Combined and organized in a {Rid1, Rid2, Key} structure. In the pipeline, tuples from the second table flow through the hash channel. Until they are joined or invalidated, several hash channels run concurrently.

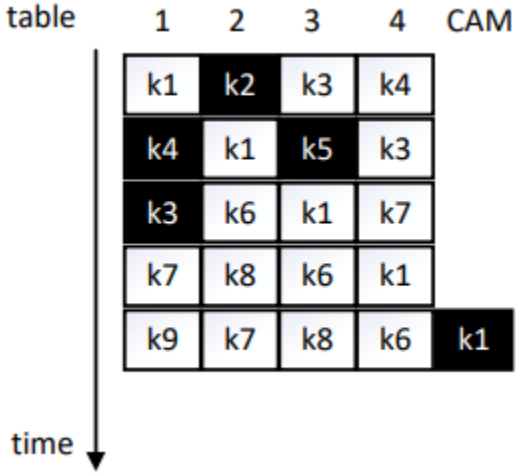

Fig.2. Probe and build phase data shift strategies.

Keys from the second table flowed concurrently over several hash channels, just like during the build phase. Each hash channel directs the tuple to the appropriate address. computer to discover a match by the hash function of each channel. In a pipeline, this is done concurrently on every channel. This tuple moves to the next channel for the next step lookup if no match is found in the first channel. If a match is found in one channel, the join result is generated for additional processing. A tuple would be submitted to the CAM for query operation if it couldn't discover a

match in any of the channels. The pair is dropped when there is a mismatch in any of the channels and CAM. In this work, we concentrate on the "N-to-1" join connection, wherein the search is stopped as soon as the key of the second table matches. There is no pipeline stall since some of the components in each channel's probe phase take a fixed number of cycles to finish. On CAMs, exact table matching procedures need a single clock cycle. As a result, during the probing phase, the memory access time is also constant. The primary goal of our solution is to distribute the first table using a tiny CAM using various hash channels, guaranteeing that the tuples that cannot be entered into all by moving the hash tables to the CAM, a non-collision hash join technique can be produced. A timeline of memory access operations for the data shift non-collision strategy is shown in Fig. 2. As can be seen, items (K1, K2, K3, K4) stream and item K2 is inserted (or probed) simultaneously during the first time slot; in the second time slot, the remaining items (K1, K3, K4) are shifted, and items (K4, K5) are successfully inserted (or probed) while a new item (K5) replaces the inserted (or probed) item. Consequently, two additional items (K6, K7) can be processed in the third time slot; No object is successfully inserted (or probed) in the fourth time slot; Since the worst-case access time happens in the fifth time slot and the shift threshold (the hash channel number) has been achieved, item (K1) is transferred to the CAM. This approach functions similarly during the build and probe phases implemented in Verilog hdls.

III. RESULTS AND ANALYSIS DISCUSSION

This section suggests a novel and efficient approach for implementing finite-field multipliers. By analyzing the theoretical limits of area and delay for conventional, Karatsuba, and overlap-free methods, the proposed implementation plan is formulated. The identified trends provide a framework for constructing finite-field multipliers of varying sizes. Additionally, the section evaluates the combinational latency and hardware resource requirements through theoretical gate-based analysis.

IV. RESULTS AND ANALYSIS DISCUSSION

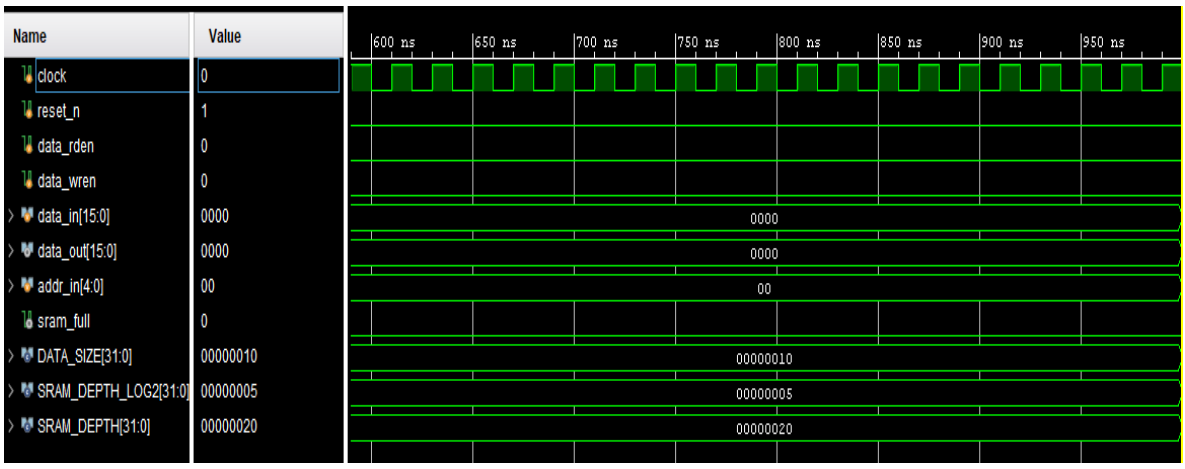

Fig.3. TCAM Simulation output of Hash function.

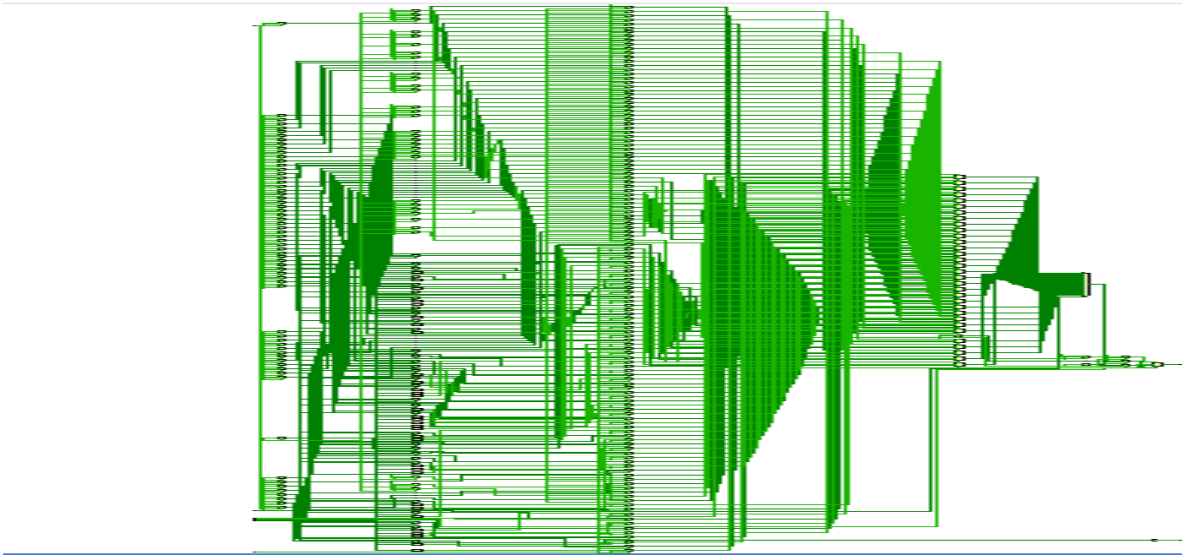

Fig.4. TCAM RTL output of Hash function.

Fig.5. TCAM Synthesized output of Hash function.

Resource	Utilization	Available	Utilization %
LUT	365	117120	0.31
FF	528	234240	0.23
IO	47	204	23.04
BUFG	1	352	0.28

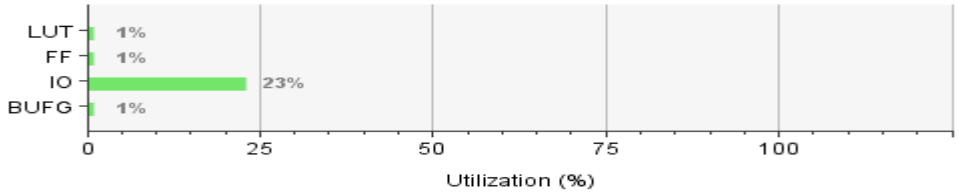

Fig.6. TCAM utilization of LUTs of Hash function.

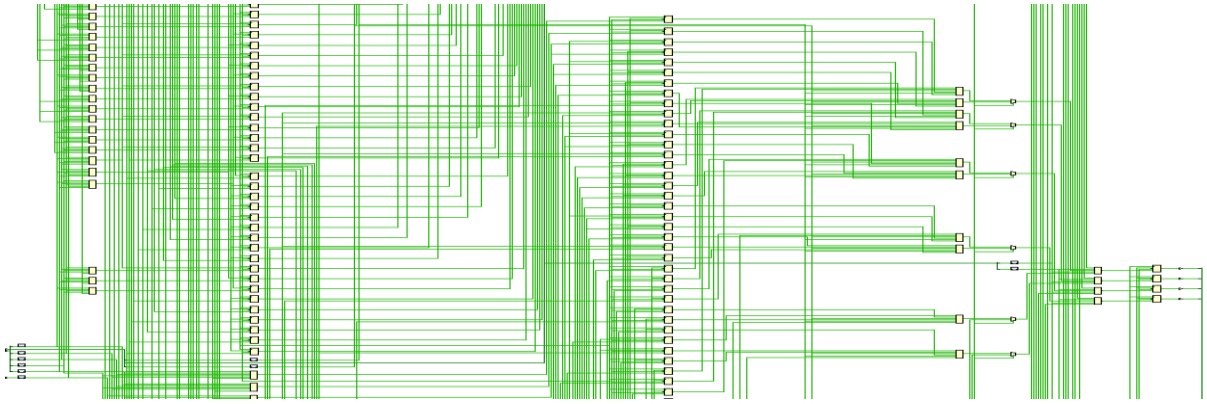

Fig.7. TCAM optimized output of Hash function.

Resource	Utilization	Available	Utilization %
LUT	361	117120	0.31
FF	528	234240	0.23
IO	47	204	23.04
BUFG	1	352	0.28

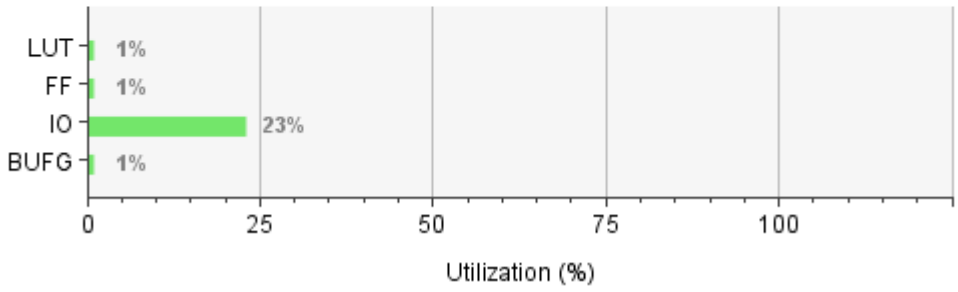

Fig.8. TCAM optimized utilization LUTs output of Hash function.

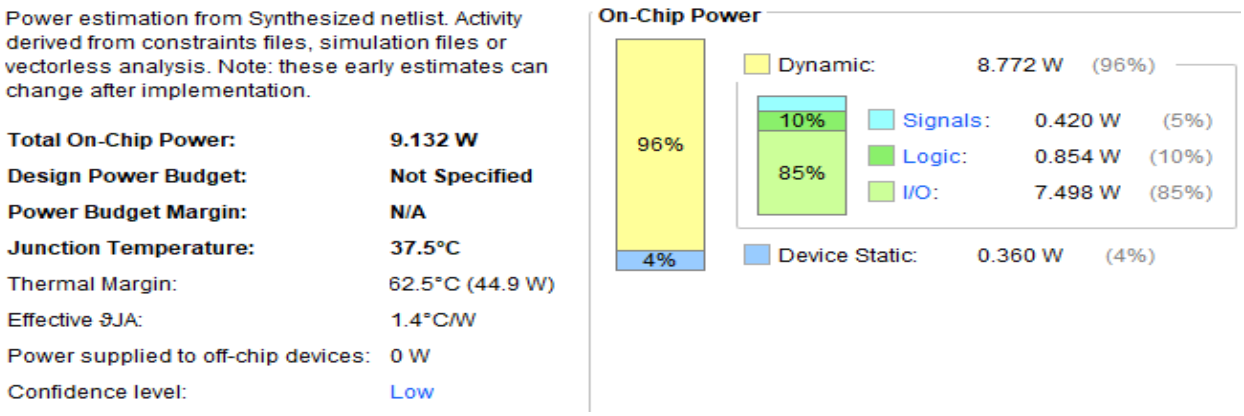

Fig.9. TCAM power of Hash function.

CONCLUSION

This paper, a non-collision parallel SRAM-based hash join architecture that enhances the speed of relational database join operations was presented in this study. Several hash channels are offered in our architecture to spread the same table, and tuples are each hash channel functions in parallel and processed in each channel inside a pipeline. The tuples that cannot be entered into all the channels are stored in a tiny CAM, which helps to resolve the hash collision. To lessen FPGA stalling, a data shift approach is applied throughout the build and probe stages. Our solution achieves a constant time for the CAM insert operation throughout the build phase, memory access time for the hash table for each phase, and a single clock cycle for the In order to increase hash join throughput and provide a predictable worst-case query time, CAM search procedure is used in the probe phase. The experimental findings show that in comparison to the current SRAM-based join accelerators.

REFERENCES

1. Ren, Chen and Prasanna, Viktor K. Accelerating Equi-Join on a CPUFPGA Heterogeneous Platform. IEEE Symp. FCCM, 2016.
2. Chen, W. and Li, W. and Yu, F. A Hybrid Pipelined Architecture for High Performance Top-K Sorting on FPGA. IEEE Trans. Circuits and Systems II: Express Briefs, pp. 1–1, 2019.
3. L. Woods, Zsolt Istvan and G. Alonso. Ibex: An intelligent storage engine with support for advanced SQL off-loading. Proc. VLDB Endowment, Vol. 7, No. 11, pp. 963–974, 2014.
4. M. Owaida et al. Centaur: A Framework for Hybrid CPU-FPGA Databases. IEEE Symp. FCCM, pp. 211–218, 2017.
5. B. Sukhwani et al. Database Analytics: A Reconfigurable-Computing Approach. Micro, IEEE, Vol. 34, No. 1, pp. 19–29, 2014.
6. Casper, Jared and Olukotun, Kunle. Hardware Acceleration of Database Operations. Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays, pp. 151–160, 2014.

7. Wang, Zeke et al. Relational query processing on OpenCL-based FPGAs. International Conference on Field Programmable Logic & Applications, 2016.
8. H. I. Hsiao et al. Parallel Execution Of Hash Joins In Parallel Databases. IEEE Trans. Parallel & Distributed Systems, Vol. 8, No. 8, pp. 872–883, 1997.
9. R. D. Gopal et al. Criss-cross hash joins: design and analysis. IEEE Trans. Knowledge & Data Engineering, Vol. 13, No. 4, pp. 637–653653, 2001.
10. N. Devarajan et al. GPU accelerated relational hash join operation. International Conference on Advances in Computing, 2013.
11. Wozniak, Kinga Anna and Schikuta, Erich. Classification Framework for the Parallel Hash Join with a Performance Analysis on the GPU. IEEE Trans. Knowledge & Data Engineering, pp. 675–682, 2017.
12. R. J. Halstead et al. Accelerating Join Operation for Relational Databases with FPGAs. IEEE International Symposium on Field-programmable Custom Computing Machines, 2013.
13. H. Zhang et al. Resource-Efficient Parallel Tree-Based Join Architecture on FPGA. IEEE Trans. Circuits and Systems II: Express Briefs, Vol. 66, No. 1, pp. 111–115, 2018.
14. S. Werner et al. Hardware-accelerated join processing in large Semantic Web databases with FPGAs. International Conference on High Performance Computing & Simulation, 2013.
15. S. Pontarelli et al. Parallel d-Pipeline: a Cuckoo Hashing Implementation for Increased Throughput. IEEE Trans. Computers, Vol. 65, No. 1, pp. 1–1, 2016.