Sailfish: Exploring Heterogeneous Query Acceleration on Discrete CPU-FPGA Architecture

Xing Wei[1,2], Yaofeng Tu[1,2], Yinjun Han[1,2], Zhenghua Chen[2], Xuecheng Qi[2], Daojun Hua[2]
{wei.xing6, tu.yaofeng, han.yinjun, chen.zhenghua, qi.xuecheng, hua.daojun}@zte.com.cn
[1]State Key Laboratory of Mobile Network and Mobile Multimedia Technology, China
[2] ZTE Corporation, China

***Abstract*—The hardware of modern server is being increasingly heterogeneous as advanced accelerators, such as FPGAs, are used together with multicore CPUs to meet the computing requirement of analytical query workloads. Unfortunately, the earlier database engines are designed for homogeneous servers, where query execution is only parallelized across CPUs, but ignores the prized FPGA resources. To exploit the available heterogeneous resources, emerging work try to construct the cross-device query pipeline, wherein a part of the operators run on the FPGA-end and the rest run on the CPU-end respectively. However, when running such a pipeline, the CPU-end and FPGA-end operators could obtain mismatched computing resources, which easily cause uneven processing performance between two ends, thereby limiting whole pipeline performance or wasting computing resources. Of note, it is nearly impossible for query optimizers to assign the matched computing resources to make CPU-end and FPGA-end have the similar processing performance, as several factors (e.g., operator selectivity) affect processing performance. To tackle this problem, we propose the *heterogeneous motion operator*, which can adjust the runtime computing resources (e.g., parallelism) of CPU-end, so as to match FPGA-end processing performance. In addition, we further implement an FPGA accelerator that supports parallel processing of hash join. By integrating our motion operator into PostgreSQL with the above FPGA accelerator, we build a prototype called Sailfish, whose experimental performance exceeds the native acceleration scheme by a huge margin.**

***Index Terms*—Database, Heterogeneous System, FPGA**

I. Introduction

The past few years have witnessed the rapid transformation of Field Programmable Gate Array (FPGA) from the specific processor to the advanced multi-function accelerator adopted by various analytical, data-intensive applications. Comparing to CPU and GPU, FPGA can organize its processing units into the specific hardware circuit for target acceleration scenarios, thereby omitting the costs of loading and parsing instructions. Hence, FPGAs are being used in many deployment scenarios, ranging from the supercomputing used for HPC applications to platform-as-a-service that provides FPGA-accelerated virtual machines. Until now, the widely-used CPU-FPGA platform is still discrete, in which the FPGA board with many computing and private memory resources is attached via PCIe bus as the peripheral of CPU.

Unfortunately, traditional analytical DBMSs solely operate on CPUs. In the past decades, to meet the strict performance requirement of big data analysis, database engines attempt to exploit the CPU parallelism (e.g., multi-thread and SIMD) and node parallelism (e.g., massively parallel processing) to speed up the query execution, yet FPGA has not attracted attentions. Recently, a part of emerging database engines [1]–[3] are being increasingly deployed on a heterogeneous platform with discrete CPU and FPGA, which aim at utilizing FPGA-end computing resources to facilitate the query execution. To achieve the heterogeneous query acceleration, earlier works [4]–[6] try to offload those CPU-heavy operators (e.g., HashJoin) into FPGA-end, which can undertake a part of CPU-end computing burden. Instead of speeding up an independent operator, later works [2], [7]–[9] aggressively hand over the contiguous operators within a pipeline to FPGA-end with the help of reconfigurable capacity in advanced FPGA (e.g., Xilinx FPGA Virtex-II). It is worth noting that deploying a group of contiguous operators on FPGA-end can not only offload more computing tasks from CPU-end, but also amortize the overheads of cross-device data transfer and/or synchronization into more operators, especially on discrete CPU-FPGA architecture.

Despite the progresses made in leveraging FPGA to speedup database, there still exists a critical issue about the mismatched computing resources between CPU and FPGA during runtime. More precisely, for a pipeline that leaves a part of contiguous operators to FPGA-end, the rest of pipeline running on CPU-end could take too many or too few computing resources (i.e., worker thread or process) to make its processing performance go beyond or lag behind FPGA-end. If CPU-end provides the higher performance, the whole pipeline will be limited by the FPGA-end and waste the extra CPU-end computing resources. Otherwise, the whole pipeline will be restricted to CPU-end. Consider the processing performance of CPU-end and FPGA-end are affected by several factors (e.g., operator selectivity), it is very difficult for query optimizer to allocate the properly computing resources that match the FPGA-end.

In this paper, we propose the *heterogeneous motion operator* to address the above issues. Such an operator is in charge of adjusting the runtime parallelism of CPU-end operators within the same pipeline, so as to match the processing performance of FPGA-end. In addition, the motion operator also plays as the coordinator to manage the cross-device data flow so that the pipeline across CPU-end and FPGA-end can run on the iterator model. Based on above efforts, we further implement a prototype called as **Sailfish** that integrates the motion operator into PostgreSQL, and take only about 30% onboard resources

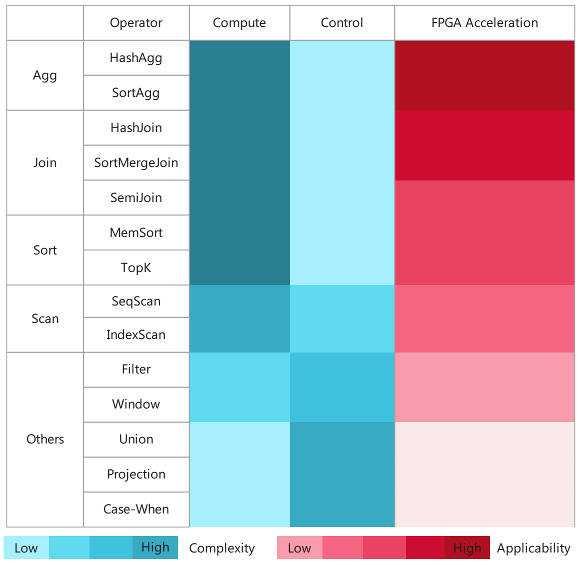
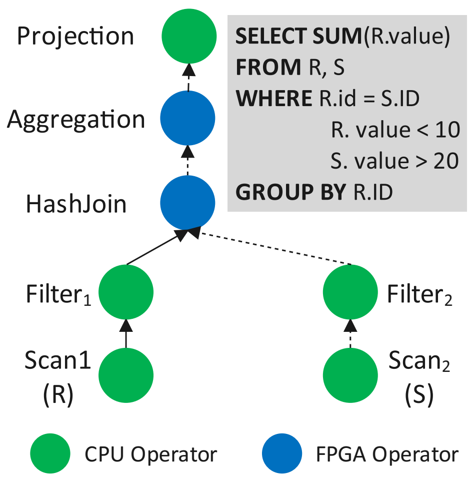


Fig. 1. A Taxonomy of Operators' Fitness to FPGA Acceleration.



```
SELECT SUM(R.value)
FROM R, S
WHERE R.id = S.ID
            R. value < 10
            S. value > 20
GROUP BY R.ID
```

Fig. 2. Query Execution with FPGA Acceleration.

to construct the FPGA-end accelerator that facilitates parallel hash join. Notably, the remaining onboard resources could still speedup other operators (e.g., Aggregation and Scan). We also conduct the experiment to verify that our design could always take proper CPU-end computing resources to fit the FPGA-end accelerator, thereby eliminating the wasted resources.

Outline. In the rest of paper, we first describe the background and related work about FPGA-based database acceleration in section II. Then, we state the design of Sailfish and detail the heterogeneous motion operator in section III. In the following, we evaluate our design in section IV. Finally, we conclude our paper and discuss further work in section V.

II. Background and Related Work

A. *Revisiting Heterogeneous Query Acceleration*

In this section, we firstly revisit the existing strategies that leverage FPGA to accelerate query execution, and then identify the potential performance issue on heterogeneous environment.

① ***Operator Acceleration:*** In DBMS, a SQL can be usually translated into the execution plan including several operators, such as $Join$ and $Aggregation$. To enable them to run faster, FPGA as a candidate accelerator can offer a large amount of programmable computing resources (i.e., FPGA Logic Cells) as specific kernels or processing engines (PEs) to execute those operators. But considering the characteristics of FPGA [10], some compute-intensive operators are friendly to FPGA, while they belong to the control-intensive area, i.e., having many branch predication, FPGA could perform worse than CPU since they need to consume many hardware resources to form the specific control logic [11]. As depicted in Fig. 1, we conduct a taxonomy of existing query operators in terms of computing and control perspectives, and derived several operators fitting to the FPGA acceleration. Specifically, $HashJoin$ and $HashAgg$ are both typical compute-intensive operators, and there exist massive implementation methods for these two operators. For example, Halstead et al. [4] implemented an end-to-end multithreading hash join on FPGA, whose experimental results can show the speedup between $2\times$ and $3.4\times$ over the multi-core approaches on the uniform and skewed datasets. Eryilmaz et al. [6] further implemented an aggregation operator that could be configured for the different number of groups, which is $2.2\times$ faster than the $HashAgg$ on CPU.

② ***Pipeline Acceleration:*** Recently, emerging work [2], [7], [8], [12] try to leave a set of contiguous operators within a pipeline to FPGA. For example, as shown in Fig. 2, the query pipeline $\{Scan_2 \rightarrow Filter_2 \rightarrow HashJoin \rightarrow Aggregation\}$ can be split into two parts, wherein the part including $HashJoin$ and $Aggregation$ can be handled by FPGA device. Considering the pipeline design [11] in FPGA, the FPGA-end operators can be mapped into a specific state machine and avoid unnecessary loads and stores of the intermediate results [13]. Even though the implementation of these operators in FPGA is trivial, the acceleration of combination of these operators is non-trivial in FPGA. To pipeline the FPGA-end operators, some typical researches [7], [8] utilize the query-to-hardware compilers to generate the hardware circuit logic automatically, and leverage the low-latency reconfigurable function of advanced FPGA to deploy the logic in the real-time fashion. Meanwhile, the other works, such as doppioDB [2], build a shared data buffer for multiple FPGA-end operators, and coordinate the data flow within the buffer to pipeline the specific operators rapidly.

B. *Achilles' Heel of Existing Acceleration Schemes*

As stated earlier, the advantages of taking FPGA as database accelerator are quite significant (e.g., replenishing computing resources), but how to make full of them is a really tough task. When pushed to the limit, in the FPGA-attached heterogeneous database bottlenecks can be attributed to the *data transmission* and *mismatched computing resources*.

① ***Massive Data Transmission:*** Unlike the computing tasks with abundant processing resources (e.g., CPU cores or FPGA Logic Cells), the cross-device data transmission is still limited by the PCIe bandwidth in the heterogeneous database system. More concisely, for the PE assembled in FPGA, its execution must keep its input available, which could cause massive data transmission if its precursor operator runs on the CPU-end. To make matters worse, if the successor operator still runs on the CPU-end, the PE should further leverage the PCIe to transfer its output to CPU-end, thereby causing the severe "data ping-pong" [11]. Meanwhile, comparing to the local memory access, the overheads of cross-device data transmission are very costly. To remedy this situation, there are several methodologies [1], [2], [5] to optimize the order of query operators, the basic two of which are filtering data as early as possible and compressing intermediate results as much as possible. But, when facing the scenario of massive data processing, above efforts will be not always enough.

② ***Mismatched Computing Resources:*** The key idea [2] of pipelining a part of contiguous operators in FPGA is relatively new, which could avoid the unnecessary cross-device loads and stores of intermediate results, and markedly lighten the burden of data transmission. For instance, as depicted in Fig. 2, there are two operators $HashJoin$ and $Aggregation$ laying on the FPGA-end, where the join results could be directly treated as input of aggregation without any transmission costs. However, when conducting a query pipeline having CPU-end operators (e.g., $Scan_2$ and $Filter_2$ in Fig. 2) and FPGA-end operators (e.g., $HashJoin$ and $Aggregation$ in Fig. 2), there is a main concern arises here: during the execution of such a pipeline, the CPU-end and FPGA-end should have matched computing resources, thereby making each end obtain similar processing performance and avoiding surplus and starvation of computing resources happening on any end. Notably, it is very challenging for existing query optimizers to foresee all related factors (e.g., operator selectivity, data distribution and etc) and generate the optimal query plan with proper computing resources allocation at both ends.

III. Design

Based on the analyses presented earlier, we firstly introduce the Sailfish design from the CPU-end and FPGA-end aspects, and then detail how the heterogeneous motion operator works to coordinate the query pipeline across different devices.

A. Framework of Sailfish

1) CPU-end Design: As illustrated in Fig. 3, the CPU-end Sailfish reuses the original PostgreSQL's processing flow, i.e., after parsing a specific SQL into the abstracted tree, the query optimizer firstly transforms the tree into a query plan and then hands the query plan over to the execution engine for further processing. But, to make the query execution benefit from the FPGA acceleration, Sailfish does some modification on query optimizer and execution engine. The details are as follows.

$$S_r = \frac{W_s + W_p}{W_s + W_p/R + W_e}$$

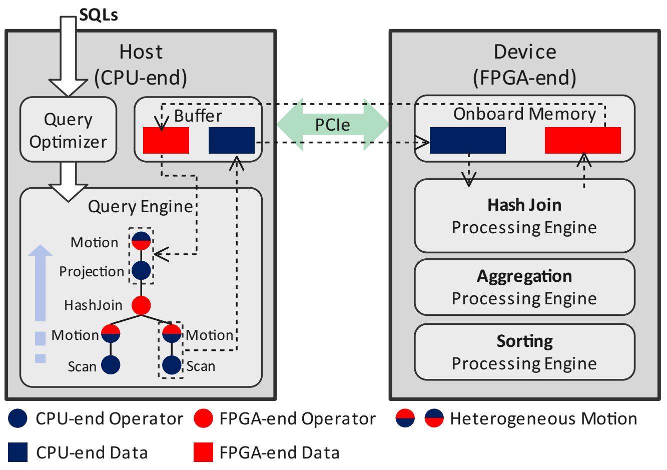


Fig. 3. The Design Overview of Sailfish.

✲ Query Optimizer: Actually, facing the abstracted tree generated by parser, the query optimizer usually picks up the specific query operators to replace algebraic operators according to the cost model. Notably, in a heterogeneous environment, if the query operators could be handed over to FPGA acceleration, they could be treated as FPGA-end operators. Otherwise, they will still be seen as CPU-end operators. In fact, to check whether an operator could be offloaded to FPGA or not, we rely on the Amdahl's law [14] to derive above equation, and integrate it into the cost model for evaluating the execution gain of an FPGA-end operator. Where, W_s and W_p are the serial-only and parallelizable workloads of the specific operator respectively.R is the speedup ratio of parallel processing on the FPGA compared to the CPU, and W_e represents the extra overhead incurred by passing the operation to FPGA. When the speedup ratio $S_r > 1$, it indicates that the FPGA-end operator is a better choice.

✲ Execution Engine: Alike the traditional PostgreSQL, Sailfish also adopts the iterator model, which means that the iteration primitive (i.e., *open*, *next* and *close*) of any operator can be directly passed to its precursor operators when they belong to the same pipeline. For example, the primitive *next* of operator $Filter_2$ (in Fig. 2) can deliver to operator $Scan_2$, and $Scan_2$ will transfer the data from table S to operator $Filter_2$. During the execution of such a pipeline, adjacent operators can have same computing capacity and similar processing throughput. But under the heterogeneous environment, those operators within same pipeline (i.e., heterogeneous pipeline) could be deployed on different device, which cannot pass any primitive and own different processing capacity. To enable the pipeline across devices to run efficiently on the iterator model, we introduce the heterogeneous motion operator (see details in section III-B), which uses host memory as a buffer to coordinate the data flow between CPU and FPGA.

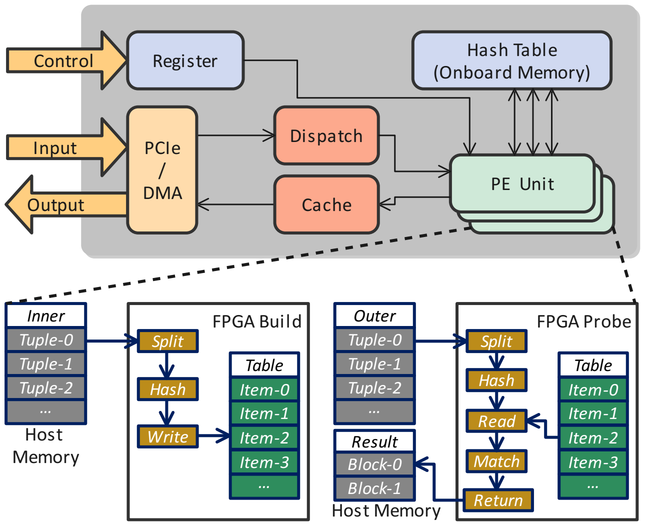


Fig. 4. The Design of FPGA-end Accelerator. Taking hash join as an example.

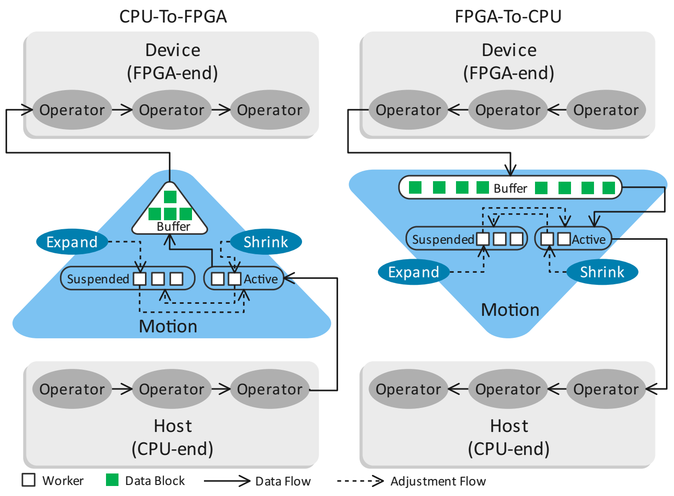


Fig. 5. The Mechanism of Heterogeneous Motion Operator.

2) FPGA-end Design: The core components of FPGA-end design is illustrated in Fig. 4. The register component can be directly accessed by CPU-end, which provides the storage for metadata and control data. The PCIe/DMA component offers the PCIe interfaces to transmit data to FPGA-end via DMA. The dispatch module is responsible for distributing the input to specific PE. Each PE can independently conduct the hash join processing based on the input. While the cache component is in charge of temporary storage of join results for CPU-end. To establish a better idea of what the FPGA-end acceleration is doing, we will take the hash-join query as an example to show the detailed workflow.

Example. To speedup a hash-join query, ① the CPU-end firstly inquires the FPGA-end registers to preempt the available PEs for hash join. ② Once getting the proper number of PEs, the CPU-end will flush the metadata (including inner/outer table schema and memory address, join conditions and etc) into the specific FPGA registers so as to notify acquired PEs to begin acceleration. ③ In the following, the CPU-end motion operator will take the interfaces of PCIe/DMA module to transmit inner tuples to FPGA-end. ④ For each received tuple, the FPGA-end dispatch module will distribute it to target PE. ⑤ Subsequently, the PE will extract the join column from each tuple according to the schema, calculate its hash value and write it into onboard hash table according to the hash value. ⑥ When completing the transmission of inner tuples, the CPU-end will continue to send the outer tuples to FPGA-end. ⑦ Meanwhile, facing the constantly coming outer tuples, the PE will keep on extracting the join columns and figure out its hash values. ⑧ Based on the hash value, the PE will retrieve the target item from onboard hash table, and check the join condition. ⑨ Once meeting the join condition, the join results will be cached in a buffer (4KB size), and then return to CPU-end in a batch.

B. Heterogeneous Motion Operator

In essence, the heterogeneous motion operator is a specific variant of the conventional gather[1] operator in the PostgreSQL version for parallel query execution. In terms of PostgreSQL's gather operator, it must be the root of a query pipeline, which creates several parallel workers to process the pipeline iteration (i.e., *open*, *next* and *close*) and collects the processing results for successor pipeline. Under the heterogeneous environment, Sailfish's motion operator will replace above gather operator to act as the root of CPU-end pipeline. Compared to the gather operator, the motion operator not only supports the parallel execution of CPU-end pipeline, but also dynamically adjusts the CPU-end parallelism to fit FPGA-end throughput. Since the FPGA-end only has limited dynamic adjustment capability, the motion operator just adjusts the CPU-end parallelism, and leaves the FPGA-end adjustment to further work.

As depicted in Fig. 5, when opening a heterogeneous motion operator, it firstly creates the worker processes with maximum parallelism, and then makes its related CPU-end pipeline split into independent tasks according to the maximum parallelism. It is worth noting that there is only a part of new workers used for the execution of CPU-end pipeline tasks at the beginning. For example, the motion operator in Fig. 5 initially creates 5 workers, but only activates 2 workers for the execution of CPU-end pipeline tasks. Soon afterwards, the motion operator still needs to apply for a piece of memory as the specific buffer that stores the data to be sent by CPU-end or received from FPGA-end. By analyzing the buffer occupancy periodically, this operator can check whether there exists a computing resource mismatch between CPU-end and FPGA-end, and then decide to expand or shrink the computing resources (i.e., worker) at the CPU-end.

1) Expand: There are two scenarios that need the motion operator to expand its CPU-end computing resources. ① When

[1]PostgreSQL's Gather Operator: https://www.postgresql.org/docs/current/how-parallel-query-works.html

the motion operator only uses less than half buffer space to store the data blocks to be sent to FPGA-end, it means there should be more computing resources contributing to the execution of CPU-end pipeline. Hence, as depicted in Fig. 5, the operator itself will wake up a worker from the suspended queue and make it involved in the execution of CPU-end pipeline. Above phase will be ongoing until the CPU-end processing speed catches up with the FPGA-end consumption rate. ② If the buffer is filled with the data blocks received from FPGA-end, the corresponding motion operator also needs to expand the computing resources and makes more workers participate in the execution of CPU-end pipeline, so as to avoid the data accumulating in the buffer.

2) Shrink: Unlike the expanding operation, there is only a scenario that will trigger a shrinkage operation. Specifically, as motion operator's buffer is filled with the data blocks that will be sent to FPGA-end, it indicates that there are many workers involved in the execution of CPU-end pipeline, whose the rate of generating data exceeds the consumption rate of FPGA-end. Under such a scenario, the shrinkage operation will continue to move the worker from the active queue into the suspended queue until the buffer is not full. Meanwhile, considering that the invocation of shrinkage operation reflects that the CPU-end will exist many idle workers, the motion operator will migrate the operators in parallel pipelines to CPU-end, thereby making the best of CPU-end computing resources.

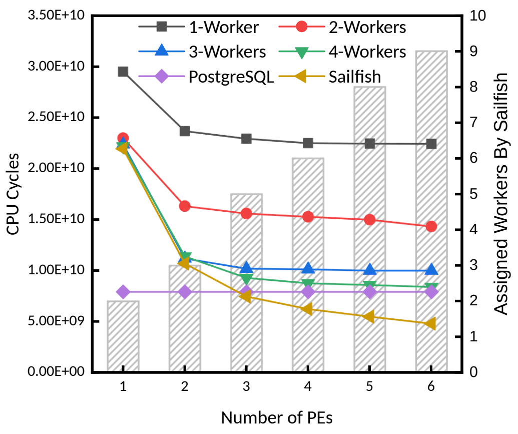
IV. Experiment

A. Experimental Setup

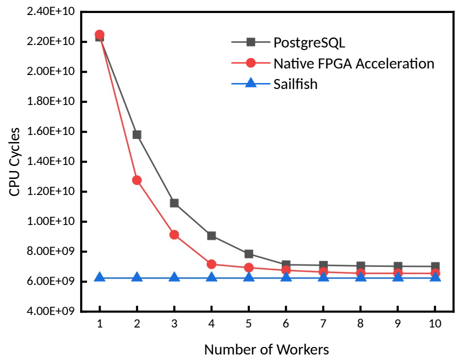
1) Hardware Environment: In default, all the experiments are deployed on a 18-core (36-hyperthread) server equipped with two Intel(R) Xeon(R) Gold 6150 CPUs clocked at 2.70 GHz with 25MB cache. This server is populated with 754 GB 2666 MT/DRAM DIMMs for the maximum bandwidth. The FPGA device Xilinx VM1802 at 250MHz is outfitted to server via PCIe generation 3. To ensure stable test performance, the CPU frequency was stabilized at 2.70GHz by turning off the Intel pstate driver so that the latency of 1 CPU cycle is fixed at 0.37ns. Compared to CPU, FPGA frequency is also fixed, and one FPGA cycle consumes 4ns. Hence, one FPGA cycle is equal to about 10.8 CPU cycles. The server runs Arch Linux with kernel 5.4.0.112.

2) Evaluated Systems: We take two representative systems as the baselines compared to our Sailfish.

- **PostgreSQL:** We take the PostgreSQL 14.0 as a baseline. Here the hash join operation is executed using only CPU processors without the FPGA acceleration.
- **Native FPGA Acceleration:** We implement a variant of PostgreSQL 14.0, which can offload the execution of hash join operator into FPGA-end, but does not support adjust the computing resources of both ends during the runtime. Hence, it is treated as the Native FPGA Acceleration (of PostgreSQL).
- **Sailfish:** We further implement the prototype of Sailfish, which can also offload the hash join operator into FPGA device and introduce our proposed heterogeneous motion operator to dynamically adjust the computing resources of CPU-end.



(a) Varying PE Number.



(b) Varying Worker Number.

Fig. 6. The Effectiveness of Heterogeneous Motion.

Meanwhile, we implement the FPGA-end hash join accelerator for Sailfish and native FPGA acceleration, whose utilization of FPGA resources is shown in Table I.

TABLE I
Utilization of FPGA Resources for Hash Join Accelerator

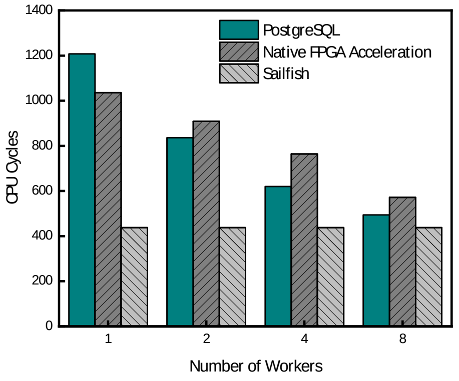
LUT	REG	BRAM
143298(15.92%)	128471(7.14%)	463(47.9%)

3) Evaluated Workloads: We take the dataset used by [15], which owns inner and outer tables. We assume that the inner table R has 1M[2] tuples and outer table has 10M tuples, which follows the Zipf distribution ($\alpha = 0.99$). Each tuple is a simple $< Key, Playload >$ pair, wherein each element occupies 4B. Based on the setting of inner and outer tables, we try to take a two-table hash join query to evaluate the effectiveness of our Sailfish's design.

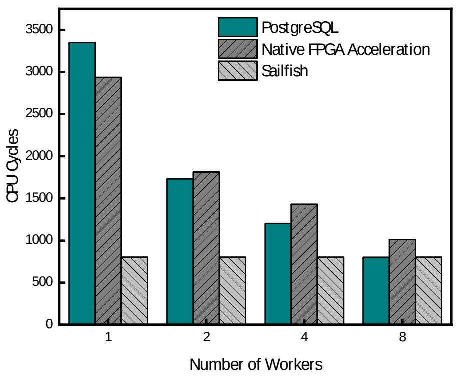
B. Effectiveness of Heterogeneous Motion

1) Varying PE Number: Here we try to vary the PE number to evaluate the effectiveness of our Sailfish. For the original

[2] $M = 1 \times 10^6$



(a) Build Phase.



(b) Probe Phase.

Fig. 7. Performance of Build and Probe Phases.

PostgreSQL, it does take any acceleration methods, and only uses the CPU workers (assigned by query optimizer) to execute query. As illustrated in Fig. 6(a), when PE number exceeds 2, the latency of Native FPGA acceleration does not decrease with the increase of PE number, the latency of Sailfish, on the other hand, continues to reduce. This is due to that Native FPGA acceleration way is not able to dynamically increase the CPU-end computing resources (i.e., worker number) when increasing the PE number, thereby making overall system performance limited by the CPU-end processing capacity. When comparing the PostgreSQL, the Native FPGA acceleration with lower worker number (e.g., 4) even perform worse since PostgreSQL could fully use the computing resource on CPU-end. For Sailfish, it can dynamically adjust the computing resources (i.e., the number of worker represented by chart in Fig. 6(a)) to fit the computing resources in FPGA-end.

2) Varying Worker Number: In contrast to change the PE number, we try to set the PE number as 5 in default. As shown in Fig. 6(b), when increasing the worker number from 1 to 4, the query latency of PostgreSQL and Native FPGA acceleration reduce quickly, but are still higher than that of Sailfish, and then become stable. Notably, the latency of Sailfish is consistently low and remains stable as the number of threads increases. This is because that the Sailfish has the ability to dynamically adjust the CPU-end worker number to match the FPGA-end processing throughput so that it cannot be limited by the CPU-end bottleneck. While for the traditional PostgreSQL and Native FPGA Acceleration, they do not have the ability to increase their assigned CPU-end worker number, thereby suffering from the limited CPU-end performance.

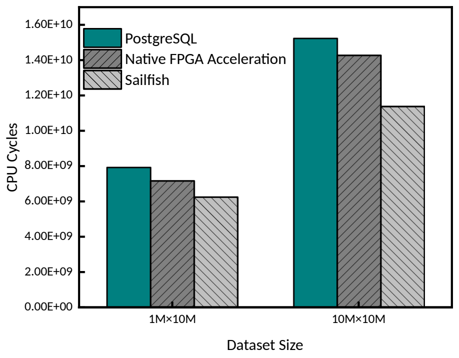


Fig. 8. Overall Performance under Different Dataset Sizes

C. Performance of Build and Probe

1) Build Phase: As depicted in Fig. 7(a), we measure the performance of build phase of the three evaluated systems by varying the worker number. For fairness, we use the fixed PE number (i.e., 5). In build phase, when the worker number increases, the latency of three systems decreases and then tends to be stable. When the worker number is small (about $1 - 4$), the build cost of Sailfish is the lowest. When the worker number is 1, the build time cost of Sailfish, Native FPGA acceleration and PostgreSQL are 432 CPU cycles, 1010 CPU cycles and 1200 CPU cycles respectively, and the latency of sailfish is only 42.8% and 36% of the latter two systems. The reason is that Sailfish can dynamically adjust the assigned CPU-end worker number to make the processing rate of CPU-end catch up the FPGA-end. Notably, only when the worker number increases to 8, the build performance of PostgreSQL can catch up Sailfish.

2) Probe Phase: Now we further evaluate the performance of probe phase by varying worker number. The measurement configuration is the same as that of build phase. As shown in Fig. 7(b), Sailfish performs the best in all systems when using the lower worker number (about $1-4$). When the worker number is 1, the latency of Sailfish, Native FPGA acceleration and PostgreSQL are 789 CPU cycles, 2916 CPU cycles and 3450 CPU cycles respectively, and the latency of Sailfish is only 27.1% and 23% of the latter two systems. For the same reason as in Build Phase, sailfish can dynamically adjust the resource schedule effectively between CPU and FPGA to reduce CPU idle time and lower system latency of probing. Meanwhile, the probe performance of PostgreSQL and Sailfish will go to deuce when the worker number increases to 8.

D. Performance under Different Dataset Sizes

At last, we measure the latency under small size of dataset (1M $\times$ 10M) and medium size of dataset (10M $*$ 10M). As shown in Fig. 8, the latency of the three systems do not differ significantly on the small dataset because that both

CPU and FPGA resources are abundant. When the dataset volume increases, that is, under the medium dataset scale, the advantage of sailfish is obvious compared with Native FPGA acceleration and PostgreSQL, because at present the CPU or FPGA resources are at full load, and the resource usage of the another one needs to be dynamically adjusted to improve the overall system performance.

V. Conclusion and Further Work

Incorporating the FPGA accelerator into a real-world query engine is a non-trivial mission, which requires thinking of the equilibrium of CPU-end and FPGA-end computing resources on the discrete architecture. Tackling this issue leds us to design the heterogeneous motion operator that can adjust the runtime computing resources (i.e., multicore parallelism) of CPU-end according to the processing performance of FPGA-end. Based on this, the query performance can be maximized by using as few CPU-end computing resources as possible. By integrating the motion operator into PostgreSQL, our experimental results reveal that it can achieve the significant performance advantages when comparing to the native FPGA acceleration.

In the further, we try to explore the following directions for more performance improvement of database engines deployed on the heterogeneous platform.

Heterogeneity-aware Optimizer: For the query optimizer in a heterogeneous database, it is a challenge to find out the optimal query plan from the huge enumeration space including lots of candidate operators for CPU and FPGA, different granularities for those operators and so on. Therefore, it is time to integrate a learning model into query optimizer so that it could quickly pick up the proper query plan from the huge search space. In addition, the query optimizer can also use the learning model to guide the resource scheduling at the compile so that CPU-end and FPGA-end have the matched processing performance.

Compute Express Link: Recently, Intel as the leader proposes the off-socket interconnect called Compute Express Link (i.e., CXL) that offers the high-speed shared memory between CPU and accelerators (e.g., FPGA and CPU), and supports the cache coherency across devices [16]. Although there is still no CXL-capable device, such a promising technique drives the industry and academia to propose several prospects that totally remove the overheads of cross-device data transmission and enable the accelerator to become a co-processor in collaboration with the CPU. But, under such a tightly-coupled architecture, CPU and FPGA are still independent devices, which have own computing resources respectively, while reserving the issue of mismatched computing resources as before.

Acknowledgment

This work was supported by the National Key Research and Development Program of China (2021YFB3101100).

References

[1] L. Woods, Z. István, and G. Alonso, "Ibex - an intelligent storage engine with support for advanced SQL off-loading," *Proc. VLDB Endow.*, vol. 7, no. 11, pp. 963–974, 2014.

[2] D. Sidler, Z. István, M. Owaida, K. Kara, and G. Alonso, "doppiodb: A hardware accelerated database," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds. ACM, pp. 1659–1662.

[3] S. Huang, L. Chang, I. E. Hajj, S. G. D. Gonzalo, J. Gómez-Luna, S. R. Chalamalasetti, M. El-Hadedy, D. S. Milojicic, O. Mutlu, D. Chen, and W. W. Hwu, "Analysis and modeling of collaborative execution strategies for heterogeneous CPU-FPGA architectures," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 7-11, 2019*. ACM, 2019, pp. 79–90.

[4] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras, "Fpga-based multithreading for in-memory hash joins," in *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[5] X. Chen, Y. Chen, R. Bajaj, J. He, B. He, W. Wong, and D. Chen, "Is FPGA useful for hash joins?" in *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*, 2020.

[6] Z. F. Eryilmaz, A. Kakaraparthy, J. M. Patel, R. Sen, and K. Park, "FPGA for aggregate processing: The good, the bad, and the ugly," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 1044–1055.

[7] C. Dennl, D. Ziener, and J. Teich, "Acceleration of SQL restrictions and aggregations through fpga-based dynamic partial reconfiguration," in *FCCM 2013, Seattle, WA, USA, April 28-30, 2013*. IEEE Computer Society, 2013, pp. 25–28.

[8] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Brezzo, S. W. Asaad, and D. Dillenberger, "Database analytics: A reconfigurable-computing approach," *IEEE Micro*, vol. 34, no. 1, pp. 19–29, 2014.

[9] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh, "In-rdbms hardware acceleration of advanced analytics," *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1317–1331, 2018.

[10] P. Papaphilippou and W. Luk, "Accelerating database systems using fpgas: A survey," in *FPL 2018, Dublin, Ireland, August 27-31, 2018*. IEEE Computer Society, 2018, pp. 125–130.

[11] J. Hoozemans, J. Peltenburg, F. Nonnemacher, A. Hadnagy, Z. Al-Ars, and H. P. Hofstee, "Fpga acceleration for big data analytics: Challenges and opportunities," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 30–47, 2021.

[12] M. Owaida, D. Sidler, K. Kara, and G. Alonso, "Centaur: A framework for hybrid CPU-FPGA databases," in *25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017, Napa, CA, USA, April 30 - May 2, 2017*. IEEE Computer Society, 2017, pp. 211–218.

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

[14] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, ser. AFIPS Conference Proceedings, vol. 30, pp. 483–485.

[15] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core cpus," in *SIGMOD 2011, Athens, Greece, June 12-16, 2011*, T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, Eds. ACM, 2011, pp. 37–48.

[16] Intel, "Compute express link," https://www.computeexpresslink.org/.