In-memory database acceleration on FPGAs

a survey

Fang, Jian; Mulder, Yvo T.B.; Hidders, Jan; Lee, Jinho; Hofstee, H. Peter

DOI
10.1007/s00778-019-00581-w

Publication date
2019

Document Version
Final published version

Published in
VLDB Journal

The VLDB Journal (2020) 29:33–59
https://doi.org/10.1007/s00778-019-00581-w

SPECIAL ISSUE PAPER

In-memory database acceleration on FPGAs: a survey

Jian Fang[1] · Yvo T. B. Mulder[2] · Jan Hidders[3] · Jinho Lee[5] · H. Peter Hofstee[1,4]

Received: 5 December 2018 / Revised: 8 July 2019 / Accepted: 11 October 2019 / Published online: 26 October 2019

Abstract
While FPGAs have seen prior use in database systems, in recent years interest in using FPGA to accelerate databases has declined in both industry and academia for the following three reasons. First, specifically for in-memory databases, FPGAs integrated with conventional I/O provide insufficient bandwidth, limiting performance. Second, GPUs, which can also provide high throughput, and are easier to program, have emerged as a strong accelerator alternative. Third, programming FPGAs required developers to have full-stack skills, from high-level algorithm design to low-level circuit implementations. The good news is that these challenges are being addressed. New interface technologies connect FPGAs into the system at main-memory bandwidth and the latest FPGAs provide local memory competitive in capacity and bandwidth with GPUs. Ease of programming is improving through support of shared coherent virtual memory between the host and the accelerator, support for higher-level languages, and domain-specific tools to generate FPGA designs automatically. Therefore, this paper surveys using FPGAs to accelerate in-memory database systems targeting designs that can operate at the speed of main memory.

Keywords Acceleration · In-memory database · Survey · FPGA · High bandwidth

1 Introduction

The computational capacity of the *central processing unit* (CPU) is not improving as fast as in the past or growing fast enough to handle the rapidly growing amount of data. Even though CPU core-count continues to increase, power per core from one technology generation to the next does not decrease at the same rate and thus the "power wall" [7] limits progress. These limits to the rate of improvement bring a demand for new processing methods to speed up database systems, especially in-memory database systems. One candidate is *field-programmable gate arrays* (FPGAs), that have been noted by the database community for their high parallelism, reconfigurability, and low power consumption, and can be attached to the CPU as an IO device to accelerate database analytics. A number of successful systems and research cited throughout this paper have demonstrated the potential of using FPGAs as accelerators in achieving high throughput. A commercial example is IBM Netezza [41], where (conceptually) an FPGA is deployed in the data path between *hard disk drives* (HDDs) and the CPU, performing decompression and pre-processing. This way, the FPGA mitigates the computational pressure in the CPU, indirectly amplifying the HDD-bandwidth that often limited database analytics performance.

While FPGAs have high intrinsic parallelism and very high internal bandwidth to speed up kernel workloads, the low interface bandwidth between the accelerator and the rest of the system has now become a bottleneck in high-bandwidth in-memory databases. Often, the cost of moving data between main memory and the FPGA outweighs the computational benefits of the FPGA. Consequently, it is a challenge for FPGAs to provide obvious system speedup, and only a few computation-intensive applications or those with

✉ Jian Fang
j.fang-1@tudelft.nl

Yvo T. B. Mulder
yvo.mulder@ibm.com

Jan Hidders
jan.hidders@vub.be

Jinho Lee
leejinho@yonsei.ac.kr

H. Peter Hofstee
hofstee@us.ibm.com

1 Delft University of Technology, Delft, The Netherlands

2 IBM Research and Development, Böblingen, Germany

3 Vrije Universiteit Brussel, Brussels, Belgium

4 IBM Research, Austin, TX, USA

5 Yonsei University, Seoul, Korea

data sets that are small enough to fit in the high-bandwidth on-FPGA distributed memories can benefit.

Even with higher accelerator interface bandwidth, the difficulty of designing FPGA-based accelerators presents challenges. Typically, implementing efficient designs and tuning them to have good performance requires developers to have full-stack skills, from high-level algorithm design to low-level circuit implementation, severely limiting the available set of people who can contribute.

While some of these challenges also apply to GPUs, GPUs have become popular in database systems. As is the case for FPGAs, GPUs can benefit from their massive parallelism and provide high throughput performance, but also like FPGAs, GPU to system memory bandwidth typically falls well short of the bandwidth of the CPU to system memory. However, compared to FPGAs GPUs support much larger on-device memory (up to 32 GB) that is accessible at bandwidths (more than 800 GB/s) that exceed those of the CPU to system memory. For these reasons, a GPU-accelerated system can provide benefit in a larger number of cases.

Emerging technologies are making the situation better for FPGAs. First, new interface technologies such as OpenCAPI [123], *Cache Coherent Interconnect for Accelerators* (CCIX) [13], and *Compute Express Link* (CXL) [112] can bring aggregate accelerator bandwidth that can exceed the available main-memory bandwidth. For example, an IBM POWER9 SO processor can support 32 lanes of the OpenCAPI interface, supplying up to 100 GB/s for each direction, while the direct-attach DDR4 memory on the same processor provides up to 170 GB/s (2667MT/s $*$ 8 channels) in total [129]. Another feature brought to FPGAs by the new interfaces is shared memory. Compared to using FPGAs as I/O devices where FPGAs are controlled by the CPU, in the OpenCAPI architecture, the coherency is guaranteed by the hardware. FPGAs are peers to the CPUs and share the same memory space. With such a high-bandwidth interface, the computational capability and the parallelism of the accelerator can now be much more effectively utilized.

Apart from new interface technologies, high-bandwidth on-accelerator memory is another enabler for FPGAs. Some FPGAs now incorporate *high bandwidth memory* (HBM) [138] and have larger local memory capacity as well as much higher (local) memory bandwidth. Similar to the GPUs with HBM, such high-bandwidth memory with large capacity allows FPGAs to store substantial amounts of data locally which can reduce the amount of host memory access, and bring the potential to accelerate some of the data-intensive applications that require memory to be accessed multiple times.

In addition, FPGA development tool chains are improving. These improvements range from *high-level synthesis* (HLS) tools to domain-specific FPGA generation tools such as query-to-hardware compilers. HLS tools such as Vivado HLS [38] and OpenCL [115] allow software developers to program in languages such as C/C++ but generate hardware circuits automatically. Other frameworks such as SNAP [136] further automate the designs of the CPU-FPGA interface for developers. In this case, the hardware designer can focus on the kernel implementation, and the software developers do not have to concern themselves with the underlying technology. Domain-specific compilers such as query-to-hardware compilers (e.g., Glacier [86]) can even compile SQL queries directly into FPGA implementations.

Therefore, with these emerging technologies, we believe that FPGAs can again become attractive as database accelerators, and it is a good time to reexamine integrating FPGAs into database systems. Our work builds on [127] which has presented an introduction and a vision on the potential for FPGA's for database acceleration. Related recent work includes [98] which draws similar conclusions with respect to the improvements in interconnect bandwidth. We focus specifically on databases, we include some more recent work, and we emphasize the possibilities with the new interface technologies.

In this paper, we explore the potential of using FPGAs to accelerate in-memory database systems. Specifically, we make the following contributions.

- We present the FPGA background and analyze FPGA-accelerated database system architecture alternatives and point out the bottlenecks in different system architectures.
- We study the memory-related technology trends including database trends, interconnection trends, FPGA development trends, and conclude that FPGAs deserve to be reconsidered for integration in database systems.
- We summarize the state-of-the-art research on a number of FPGA-accelerated database operators and discuss some potential solutions to achieve high performance.
- Based on this survey, we present the major challenges and possible future research directions.

The remainder of this paper is organized as follows: In Sect. 2, we provide FPGA background information and present the advantages of using FPGAs. Section 3 explains the current database systems accelerated by FPGAs. We discuss the challenges that hold back use of FPGAs for database acceleration in Sect. 4. The database, interconnect and memory-related technology trends are studied in Sect. 5. Section 6 summarizes the state-of-the-art research on using FPGAs to accelerate database operations. Section 7 presents the main challenges of using high-bandwidth interface attached FPGAs to accelerate database systems. Finally, we conclude our work in Sect. 8.

System designers may be interested in Sect. 3 for the system architecture overview, Sect. 4 for the system limita-

tions, and Sect. 5 for the technology trends that address these limitations. FPGA designers might want to concentrate on Sect. 6 that discusses the state of the art for high-bandwidth operators relevant to database queries. For performance analysts, Sect. 4 gives a brief comparison between FPGAs and GPUs, as well as the challenges of FPGA regarding database acceleration. For the performance of each operator, a deeper discussion is presented in Sect. 6. For software developers, Sect. 2 provides an introduction to FPGAs, while FPGA programming is discussed in Sect. 4 and 5. We also present lessons learned and potential future research directions in Sect. 7 addressing different groups of researchers.

2 FPGA background

This section gives an introduction to FPGAs, and provides software researchers and developers with background knowledge of FPGAs including architecture, features, programming, etc.

2.1 Architecture

An FPGA consists of a large number of programmable logic blocks, interconnect fabric and local memory. *Lookup tables* (LUTs) are the main component in programmable logic. Each LUT is an n-input 1-output table,[1] and it can be configured to produce a desired output according to the combination of the n inputs. Multiple LUTs together can be connected by the configurable interconnect fabric, forming a more complex module. Apart from the logic circuits, there are small memory resources (registers or flip-flops) to store states or intermediate results and larger block memory (*Block RAMs* or BRAMs) to act as local memory. Recently, FPGA chips are equipped with more powerful resources such as built-in CPU cores, *Digital Signal Processor* (DSP) blocks, *UltraRAM* (URAM), HBMs, preconfigured I/O blocks, and memory-interface controllers.

2.2 Features

The FPGA is a programmable device that can be configured to a customized circuit to perform specific tasks. It intrinsically supports high degrees of parallelism. Concurrent execution can be supported inside an FPGA by adopting multi-level parallelism techniques such as task-level parallelization, data-level parallelization, and pipelining. In addition, unlike the CPU where the functionality is designed for generic tasks that do not use all the resources efficiently for a specific application, the circuit in an FPGA is highly customizable, with only the required functions implemented. Even though building specific functions out of reconfigurable logic is less efficient than building them out of customized circuits, in many cases, the net effect is that space is saved and more processing engines can be placed in an FPGA chip to run multiple tasks in parallel. Also, the capability of customizing hardware leads to significant power savings compared to CPUs and GPUs, when the required functionality is not already directly available as an instruction. The FPGA can also support data processing at low latency due to the non-instruction architecture and the data flow design. In CPUs, the instructions and data are stored in the memory. Executing a task is defined as running a set of instructions, which requires fetching instructions from memory. However, FPGAs define the function of the circuit at design-time, where the latency is dependent on the signal propagation time. Apart from that, the data flow design in FPGAs allows forwarding the intermediate results directly to the next components, and it is often not necessary to transfer the data back to the memory.

2.3 FPGA-related bandwidth

As we focus on the bandwidth impact on this paper, we give a brief introducing of FPGA-related bandwidth and present the summary in Table 1. Similar to the CPU memory hierarchy, the memory close to the FPGA kernel has the lowest latency and highest bandwidth, but the smallest size. The FPGA internal memory including BRAM and URAM typically can reach TB/s scale bandwidth with a few nanoseconds latency. The on-board DDR device can provide tens GB/s bandwidth, While the HBM that within the same socket with the FPGA have hundreds of GB/s bandwidth, and both of them require tens to hundreds nanoseconds latency to get the data. The bandwidth to access the host memory typically is the lowest one in this hierarchy. However, it provides the largest memory capacity.

Hiding long memory latency is a challenge for FPGA designs. Typically, applications with streaming memory access patterns are less latency-sensitive: because the requests are predictable it is easier to hide the latency. However, applications that require a large amount of random access (e.g., as hash join) or unpredictable streaming access (e.g., sort) could get stalls due to the long latency. In this case, we might need to consider using memory with lower latency or transform the algorithms to leverage streaming. We discuss more details based on different operators in Sect. 6.

2.4 Programming

The user-defined logic in the FPGA is generally specified using a *hardware description language* (HDL), mostly

[1] Multi-output LUTs are available now. See Figure 1-1 in https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.

Table 1 FPGA-related bandwidth and latency (from data source to FPGA kernels)

Mem source	Mem type	BW (GB/s)	Latency (ns)	Capacity (MB)
Internal	BRAM	$\geq 10^3$	10^0	10^0
	URAM	$\geq 10^3$	10^1	10^1
On-board	HBM	10^2–10^3	10^1–10^2	10^3
	DRAM	10^1–10^2	10^1–10^2	10^4
Host	DRAM	10^1	$\geq 10^2$	$\geq 10^5$

VHDL or Verilog. Unlike software programming languages such as C/C++ handling sequential instructions, HDLs describe and define an arbitrary collection of digital circuits. It requires the developers to have knowledge on digital electronics design, meaning understanding how the system is structured, how components run in parallel, how to meet the timing requirement, and how to trade off between different resources. This is one of the main reasons that make the software community reluctant to use FPGAs.

High-level synthesis (HLS) tools such as Vivado HLS [38] and Altera OpenCL [115] overcome this problem by supporting software programmers with the feasibility of compiling standard languages such as C/C++ and higher-level hardware-oriented languages like systemC into register-transfer level (RTL) designs. In such a design procedure, HLS users write C code and design the interface protocol, and the HLS tools generate the microarchitecture. Apart from generating the circuit itself, programming frameworks such as OpenCL [121] provide frameworks for designing programs that run on heterogeneous platforms (e.g., CPU+FPGA). These frameworks typically specify variants of standard programming languages to program the kernels and define application programming interfaces to control the platforms. The corresponding *software development kits* (SDKs) are now available for both Xilinx FPGAs [137] and Intel FPGAs [56]. There are also some domain-specific compilers that can compile SQL queries into circuits or generate the circuit by setting a couple of parameters. An example is Glacier [86]) which provides a component library and can translate streaming queries into hardware implementations.

3 FPGA-based database systems

How to deploy FPGAs in a system is a very important question for system designers. There are many ways to integrate FPGAs into database systems. The studies in [82,83] categorize the ways FPGAs can be integrated by either placing it between the data source and CPU to act as a filter or by using it as a co-processor to accelerate the workload by offloading tasks. Survey [57] presents another classification that contains three categories including "on-the-side" where the FPGA is connected to the host using interconnect such as PCIe, "in data path" where the FPGA is placed between the storage/network and the CPUs, and "co-processor" where the FPGA is integrated together with the CPU in the same socket. In this section, we specify three possible database architectures with FPGA accelerators in a logical view and explain their shortcomings and advantages.

Fig. 1 FPGA as a bandwidth amplifier

3.1 Bandwidth amplifier

In a storage-based database system, the bottleneck normally comes form the data transmission to/from the storage, especially the HDD. Compared to hundreds of Gbit/s bandwidth supported by DRAM, the data rate of an HDD device remains at the 1 Gbit/s level, which limits the system performance. In these systems, FPGAs can be used to amplify the storage bandwidth.

As shown in Fig. 1, the FPGA is used as a decompress-filter between the data source (disks, network, etc.) and the CPU to improve the effective bandwidth. In this architecture, the compressed data is stored on the disks, and would be transferred to the FPGA, either directly through the interfaces like SCSI, SATA, Fibrechannel, or NVMe or indirectly, for network-attached storage or protocols like NVMe over Infiniband or Ethernet. In the FPGA, the data is decompressed and filtered according to some specific conditions, after which the data is sent to the CPU for further computation. As the compressed data size is smaller than the original data size, less data needs to be transferred from storage, improving the effective storage bandwidth indirectly.

The idea has proven to be successful by commercial products such as Netezza [41], or a few SmartNIC variants [80,92]. In Netezza, an FPGA is placed next to the CPU doing the decompression and aggregation in each node, and only the data for post-processing is transferred to the CPU. In a few SmartNIC products, an FPGA sits as a filter for the network traffic. By applying compression/decompression or deduplication, they greatly enhance the effective bandwidth

of an network-to-storage applications. A similar idea is also studied by prior research such as the ExtraV framework [72], where the FPGA is integrated into the system in an implicit way. The FPGA is inserted in the data path between the storage and the host, performing graph decompression and filtering. Some research [42,128,139] shows that even without doing the decompression by only performing filtering and aggregation on the FPGA, one can significantly reduce the amount of data sent to the CPU, as well as relieve the CPU computational pressure. This is a good solution for latency-sensitive applications with data stream processing, where the FPGA capability for the high throughput and low latency processing is demonstrated.

3.2 IO-attached accelerator

Attaching FPGAs as an IO-attached accelerators is another conventional way to deploy accelerators in the systems, especially for computational-intensive applications in which the CPUs are the bottleneck of these systems. In this case, FPGAs are used as IO devices performing data processing workloads offloaded by CPUs. Figure 2 illustrates the architecture of using the FPGA as an IO-attached accelerator. In this architecture, the FPGA is connected to the CPU through buses such as PCIe, and the FPGA and CPU have their own memory space. When the FPGA receives tasks from the CPU, it first copies the data from the host memory to the device memory. Then the FPGA fetches data from the memory and writes the results back to the device memory after processing it. After that, the results can be copied back to the host memory.

This approach is illustrated by both industry and academic solutions. Kickfire's MySQL Analytic Appliance [63], for example, connects the CPU with a PCIe-attached FPGA card. The offloaded queries can be processed in the FPGA with a large amount of high-bandwidth on-board memory. Xtremedata's dbX [106] offers an in-socket FPGA solution where the FPGA is pin compatible with the CPU socket. Key database operations including *joins*, *sorts*, *groupbys*, and *aggregations* are accelerated with the FPGA. In recent academic research on accelerating database operators such as sort [18] and join [109], the data is placed in the device memory to avoid data copies from/to the host. This architecture is also present in GPU solutions such as Kinetica [66], MapD [81], and the research work in [51]. A drawback of this architecture is that it requires extra copies (from the host memory to the device memory and the other way around) which leads to longer processing latencies. Also the application must be carefully partitioned, as the accelerator is unable to access memory at-large. The separate address spaces also affect debug, and performance tools. Even today, it is often difficult to get an integrated view of the performance of a GPU-accelerated system for example. Placing the whole database in the device

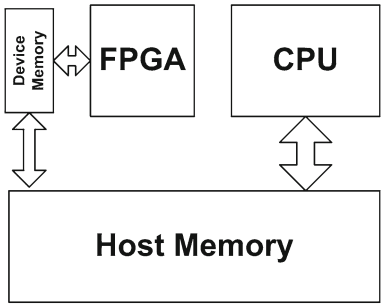


Fig. 2 FPGA as an IO-attached accelerator

memory such as the design from [43] can reduce the impact of the extra copies. However, since the device memory has limited capacity which is much smaller than the host memory, it limits the size of the database and the size of the applications.

3.3 Co-processor

Recent technology allows FPGA to be deployed in a third way where the FPGA acts as a co-processor. As shown in Fig. 3, in this architecture, the FPGA can access the host memory directly, and the communication between the CPU and the FPGA is through shared memory. Unlike the IO-attached deployment, this deployment provides the FPGA full access to system memory, shared with the CPU, that is much larger than the device memory. In addition, accessing the host memory as shared memory can avoid copying the data from/to the device memory. Recently, there have been two physical ways to deploy FPGAs as co-processors. The first way tightly couples the FPGA and the CPU in the same die or socket, such as Intel Xeon+FPGA platform [48] and ZYNQ [27]. The FPGA and CPU are connected through *Intel QuickPath Interconnect* (QPI) for Intel platforms and *Accelerator Coherency Port* (ACP) for ZYNQ, and the coherency is handle by the hardware itself. The second method connects the FPGAs to the host through coherent IO interfaces such as IBM *Coherent Accelerator Processor Interface* (CAPI) or OpenCAPI [123], which can provide high bandwidth access to host memory. The coherency between the CPU and the FPGA is guaranteed by extra hardware proxies. Recently, Intel also announced a similar off-socket interconnect called *Compute Express Link* (CXL) that enables a high-speed shared-memory based interaction between the CPU, platform enhancements and workload accelerators.

DoppioDB [114] is a demonstrated system of this architecture from academia. It extends MonetDB with user-defined functions in FPGAs, along with proposing a Centaur framework [97] that provides software APIs to bridge the gap between CPUs and FPGAs. Other research work studies the acceleration of different operators including compression [104], decompression [35], sort [146] and joins [49], etc.

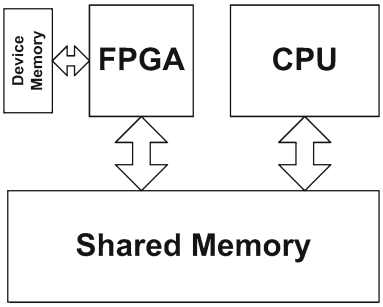


Fig. 3 FPGA as a co-processor

4 What has held FPGAs back?

In this section, we discuss three main challenges that have reduced the interest in using FPGAs to accelerate in-memory database systems in both industry and academia. System designers and performance analysts might be interested in Sects. 4.1 and 4.3 where the system performance limitation and the comparison to GPUs are explained, while software developers can focus on the FPGA programmability in Sect. 4.2. In Sect. 5 we discuss the technology trends that address the challenges discussed here.

4.1 Significant communication overhead

A first obvious challenge for using any accelerator is communication overhead. While many FPGA accelerators discussed in Sect. 6 such as [49,61,125] have demonstrated that FPGAs can achieve high (multi-) kernel throughput, the overall performance is frequenty-limited by the low bandwidth between the FPGA and the host memory (or CPU). Most recent accelerator designs access the host memory data through PCIe Gen3, which provides a few GB/s bandwidth per channel or a few tens of GB/s accumulated bandwidth. This bandwidth is not always large enough compared to that between the CPU and the main memory in the in-memory database systems, and the cost of data transmission from/to the FPGA might introduce significant overhead.

In addition, transferring data from the host to FPGAs that are not in the same socket/die as the CPU increases latency. When the FPGA operates in a different address space, latency is increased even more (a few microseconds is not uncommon). This brings challenges to the accelerator designs, especially for those have unpredictable memory access patterns such as scatter, gather and even random access. To hide the long latency, extra resources are required to buffer the data [88] or to maintain the states of a massive number of tasks or threads [49].

4.2 Weak programmability

Another big challenge is the difficulty of developing FPGA-based accelerators and effectively using them, which has two main reasons.

First, programming an FPGA presents a number of challenges and tradeoffs that software designers do not typically have to contend with. We give a few examples. To produce a highly-tuned piece of software, a software developer occasionally might have to restructure their code to enable the compiler to do sufficient loop unrolling (and interleaving) to hide memory latencies. When doing so there is a tradeoff between the number of available (renamed) registers and the amount of loop unrolling. On the FPGA, the equivalent of loop unrolling is pipelining, but as the pipeline depth of a circuit is increased, its hardware resources change, but its operating frequency can also change, making navigating the design space more complex. Even when we limit our attention to the design of computational kernels, an awareness of the different types of resources in an FPGA may be required to make the right tradeoffs. The implementation in an FPGA is either bound by the number of the computation resources (LUTs, DSP, etc.) or the size of the on-FPGA memory (BRAM, URAM, etc.) or it can be constrained by the available wiring resources. Which of these limits the design informs how the design is best transformed. As can be seen from this example, implementing an optimized FPGA design typically requires developers to have skills across the stack to gain performance While nowadays HLS tools can generate the hardware circuits from software language automatically by adopting techniques such as loop unrolling and array partitioning [26,28], manual intervention to generate efficient circuits that can meet the timing requirements and sensibly use the hardware resources is still required too often. In many cases, the problem is outside the computational kernels themselves. For example, a merge tree that can merge multiple streams into a sorted stream might require prefetching and bufferring of data to hide the long latency. In this case, rewriting the software algorithms to leverage the underlying hardware or manually optimizing the hardware implementation based on the HLS output or even redesigning the circuit is necessary.

Second, generating query plans that can be executed on an FPGA-accelerated system demands a strong query compiler that can understand the underlying hardware, which is still lacking today. The flipside of having highly specialized circuits on an FPGA is that (parts of) the FPGA must be reconfigured when a different set, or a different number of instances of functions or kernels is needed, and FPGA reconfiguration times exceed typical context switch penalties in software by orders of magnitude. In software, the cost of invoking a different function can usually be ignored. Unlike the single-operator accelerators, we survey in Sect. 6, in real-

ity, a query is typically a combination of multiple operators. The query optimizer component in the query compiler optimizes the query plan by reordering and reorganizing these operators based on the hardware. While this field has been well studied in the CPU architecture, it becomes more challenging when taking FPGAs into account. Because of the long reconfiguration times, a query plan for a short running query may look vastly different than an optimized query plan for a long-running query, even if the queries are the same. Thus query compilers need to map the query plan to meet the query execution model in the FPGA. In addition, the FPGA designs may not be optimized and implemented for all required functions or operators. Thus, for some special functions or operators that have not been implemented in the FPGA or where FPGAs do not provide adequate performance gain, the query compiler should drive the query plan back to the pure CPU execution model. Without a shared address space, and a common method to access the data (e.g., the ability to lock memory), a requirement to flexibly move components of a query between the CPU and accelerators is significantly complicated.

4.3 Accelerator alternative: GPU

In recent years, the GPU has become the default accelerator for database systems. There are many GPU-based database systems from both industry and academia such as Kinetica/GPUDB [66,143], MapD [81], Ocelot [53], OmniDB [147], and GPUTx [52]. A comprehensive survey [15] summarizes the key approaches to using GPUs to accelerate database systems and presents the challenges.

Typically, GPUs achieve higher throughput performance while FPGAs gain better power-efficiency [23]. The GPU has a very large number of lightweight cores with fewer control requirements and provides a high degree of data-level parallelism. This is an important feature to accelerate database applications since many database operators are required to perform the same instructions on a large amount of data. Another important feature is that the GPU has large capacity high-bandwidth on-device memory that is typically much larger than the host main memory bandwidth and the CPU-GPU interface bandwidth. Such large local memory allows GPUs to keep a large block of hot data and can reduce the communication overhead with the host, especially for applications that need to touch the memory multiple times such as the partitioning sort [39] that achieves 28 GB/s throughput on a four-GPU POWER9 node.

While FPGAs cannot beat the throughput of GPUs in some database applications, the result might change in power-constrained scenarios. One of the reasons is that the data flow design on FPGAs can avoid moving data between memories. A study from Baidu [96] shows that the Xilinx KU115 FPGA is 4x more power-efficient than the GTX Titan GPU when running the sort benchmark. In addition, the customizable memory controllers and processing engines in FPGAs allow FPGAs to handle complex data types such as variable-length strings and latency-sensitive applications such as network processing that GPUs are not good at (we discuss more detail in Sect. 7.3).

However, when considering programmability and the availability of key libraries, FPGAs still have a long way to go compared to GPUs, as mentioned in Sect. 4.2. Consequently, engineers might prefer GPUs which can also provide high throughput but are much easier to program, debug, and tune for performance. In other words, GPUs have raised the bar for using FPGAs to accelerate database applications. Fortunately, the technology trends in Sect. 5 show that some of these barriers are being addressed, and thus it is worth it to have another look ar FPGAs for in-memory database acceleration.

5 Technology trends

In recent years, various new technologies have changed the landscape of system architecture. In this section, we study multiple technology trends including database system trends, system interconnect trends, FPGA development trends, and FPGA usability trends, and we introduce a system model that combines these technologies. We believe that these technology trends can help system designers to design new database system architectures, and help software developers to start using FPGAs.

5.1 Database system trends

Databases traditionally were located on HDDs, but recently the data is often held in-memory or on NVM storage. This allows for two orders of magnitude more bandwidth between the CPU and stored database compared to the traditional solution. Some of the database operators now become computation-bound in a pure CPU architecture, which demands new and strong processors such as GPUs and FPGAs.

However, the downside is that acceleration, or the offloading of queries, becomes more difficult due to the low FPGA-CPU interconnect bandwidth. When using FPGAs as IO-attached accelerator, typically *PCI Express* (PCIe) Gen 3 is used as an IO interconnect. It provides *limited bandwidth* compared to DRAM over DDRx, and may suffer resource contention when sharing PCIe resources between FPGA and NVM over NVMe, which may impact the performance. While PCIe Gen 4 doubles the bandwidth, it does not solve the communication protocol limitation that using FPGAs as IO-attached accelerators requires to copy the data

between the host memory and the device memory, resulting in extra data transmission overhead and long latency.

These limitations result in a high cost of data movement between the database and the FPGA. This limits the applicability of FPGAs in the data center. In order to accelerate databases with FPGAs (again), the interconnect has to overcome these limitations in order to become a viable alternative.

5.2 System interconnect trends

The bandwidth of system interconnects plateaued for quite a few years, after the introduction of PCIe Gen3. More recently the pace has increased, the PCI Express Consortium released the specification of Gen 4 in 2017 and Gen 5 in 2019, respectively [100], and is expected to release the specification of Gen 6 in 2021 [99]. Because of the long wait for a new generation, other initiatives had started, proposing new interconnect standards to solve the bandwidth bottlenecks mentioned in Sect. 4.

5.2.1 Increase in system interconnect bandwidth

Figure 4 shows collected data regarding DRAM, network and storage bandwidth in 2019 [70] and predicts the future until 2022 (indicated by the diamond-shaped markers). The bandwidth of PCIe was added to act as a proxy for interconnect bandwidth. For each generation of the PCIe standard, the bandwidth of sixteen lanes is plotted, since this is typically the maximum number of lanes per PCIe device. The DRAM data interface is inherently uni-directional and several cycles are required to turn the channel around. A memory controller takes care of this by combining reads and writes to limit the overhead cycles spent in configuring the channel. Therefore, DRAM bandwidth should be interpreted as either a read or write channel with the plotted bandwidth, while attached devices such as network and storage typically have a bi-directional link.

The slope of each of the fitted lines is important here. Clearly, both network and storage bandwidths are increasing at a much faster rate (steeper slope) than DRAM and PCIe. The network and storage slopes are similar and double every 18 months. PCIe doubles every 48 months, while it takes DRAM 84 months to double in bandwidth. A server typically contains one or two network interfaces, but often contains a dozen or more storage devices, lifting the blue line by an order of magnitude. Thus, a shift in balance is expected for future systems where DRAM, interconnect, network and storage bandwidth are about the same.

The fitted straight lines for each of the four data sets shown in Fig. 4 indicate exponential behavior. While it might look like accelerators, such as GPUs and FPGAs, will have to compete for interconnect bandwidth with network and storage, one option is to scale memory and interconnect bandwidth

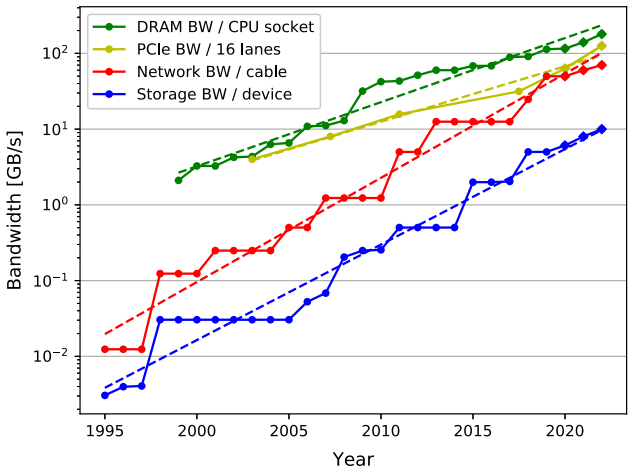


Fig. 4 Bandwidth trends at device-level. Data points were approximated from the referenced figures in order to add the PCI Express standard bandwidth and represent all bandwidths in GB/s [70,88]

accordingly. While scaling is the trend, as becomes apparent from the next paragraph, and works in the short-term, it does not solve the fundamental problem of limited DRAM bandwidth improvements.

The reason that DRAM bandwidth is not increasing at a similar pace is twofold. To increase bandwidth, either the number of channels or the channel frequency must be increased. However, each solution has significant implications. Every additional channel requires a large number of pins (order of 100) on the processor package (assuming an integrated memory controller) that increases chip area cost. Increasing channel frequency requires expensive logic to solve signal integrity problems at the cost of area, and more aggressive channel termination mechanisms at the cost of power consumption [30,47]. If the bandwidth of the interconnect is increased to the same level as DRAM, the same problems that DRAM faces will be faced by attached devices.

5.2.2 Shared memory and coherency

Solely increasing bandwidth will not solve all of our problems, because the traditional IO-attached model will become a bottleneck. Currently, the host processor has a shared memory space across its cores with coherent caches. Attached devices such as FPGAs, GPUs, network and storage controllers are memory-mapped and use a DMA to transfer data between local and system memory across an interconnect such as PCIe. Attached devices can not see the entire system memory, but only a part of it. Communication between the host processor and attached devices requires an inefficient software stack in comparison to the communication scheme between CPU cores using shared memory. Especially when DRAM memory bandwidth becomes a constraint, requir-

ing extra memory-to-memory copies to move data from one address space to another is cumbersome.

This forced the industry to push for coherency and shared memory across CPU cores and attached devices. This way, accelerators act as peers to the processor cores. The Cell Broadband Engine architecture [59] introduced coherent shared system memory access for its Synergistic Processor Element accelerators. A coherent interface between a CPU and GPU has also been adopted by AMD several years ago with their Accelerated Processing Unit (APU) device family [29]. Another example is the *Cache Coherent Interconnect for Accelerators* (CCIX) [13] which builds on top of PCIe and extends the coherency domain of the processor to heterogeneous accelerators such as FPGAs. OpenCAPI is also a new interconnect standard that integrates coherency and shared memory in their specification. This avoids having to copy data in main memory, and coherency improves FPGA programmability.

With shared memory, the system allows FPGAs to read only a small portion of the data from the host memory without copying the whole block of data to the device memory. This can reduce the total amount of data transmission if the application has a large number of small requests. With coherency supported by hardware, programmers can save effort needed to keep the data coherent through software means. In addition, the shared, coherent address space provided by the coherent interface allows programmers to locally transform a piece of code on the CPU to run on the FPGA, without having to understand the full structure of the program and without having to restructure all references to be local to the FPGA. Especially for production code that tends to be full of statements that are very infrequently executed, the ability to focus on performance-critical code without having to restructure everything is essential. The drawback of supporting shared memory and coherency by hardware is that it requires extra hardware resources and can introduce additional latency. Thus, for performance reasons, a developer might need to optimize the memory controller for special memory access patterns.

5.2.3 Concluding remarks

As discussed in this section, both identified bottlenecks will soon belong to the past. This opens the door for FPGA acceleration again. FPGAs connected using a high bandwidth and low latency interconnect, and ease of programming due to the shared memory programming model, make FPGAs attractive again for database acceleration.

5.3 HBM in FPGAs

As shown in Fig. 4, DRAM bandwidth is not increasing at the same rate as attached devices. Even though the latest DDR4 can provide 25 GB/s bandwidth per *Dual In-line Memory Module* (DIMM), for high-bandwidth applications, a dozen or more modules are required. This leads to a high price to pay in *Printed circuit board* (PCB) complexity and power consumption.

The new high-bandwidth memory technologies provide potential solutions, one of which is high-bandwidth memory (HBM). HBM is a specification for 3D-stacked DRAM. HBM has a smaller form factor compared to DDR4 and GDDR5, while providing more bandwidth and lower power consumption [65]. Because HBM is packaged with the FPGA, it circumvents the use of a PCB to connect to DRAM. The resulting package is capable of multi-terabit per second bandwidth, with a raw latency similar to DDR4. This provides system designers with a significant improvement in bandwidth. The latest generation of Xilinx FPGAs supports HBM within the same package [138], providing FPGAs with close to a half TB/s scale bandwidth (an order of magnitude more bandwidth than the bandwidth to typical on-accelerator DRAM). This makes FPGAs also applicable to data intensive workloads.

Due to the area limitation and the higher cost of stacked DRAMs, HBM integrated with the FPGA can not match the capacity of conventional DRAM. Integration with FPGAs results in a competitive advantage for workloads that require, for example, multiple passes over the same data at high bandwidth. Various examples of multi-pass database queries have been studied in this paper. An example is the sort algorithm presented in Sect. 6.3.

5.4 System with accumulated high bandwidth

Today it is possible to have systems with large storage and accelerator bandwidth. Accelerated database systems can leverage these types of heterogeneous systems. A feasible conceptual system model is depicted in Fig. 5, which is based on the IBM AC922 HPC server [55,89]. Note that the numbers shown in Fig. 5 are peak numbers.

This system consists of two nodes that connect to each other via a *Symmetric multiprocessing* (SMP) interface with 64 GB/s bandwidth in each direction. Each node contains one POWER9 CPU and two FPGAs. Each POWER9 CPU has 170 GB/s bandwidth to DDR4 memory in the host side and supports up to two FPGAs. Each FPGA is connected at 100 GB/s rate through two OpenCAPI channels, meaning in total 200 GB/s accelerator bandwidth is provided in each node. The FPGA fabric would be the latest model VU37P [142], where 8 GB HBM is integrated that supports 460 GB/s bandwidth. Each FPGA would have two OpenCAPI interfaces to the I/O that can be used to attach the NVMe storage. In total, eight of these interfaces in a two-node system support 400 GB/s peak storage I/O bandwidth. FPGAs can be connected to each other via 100 GB/s high-end interfaces. This

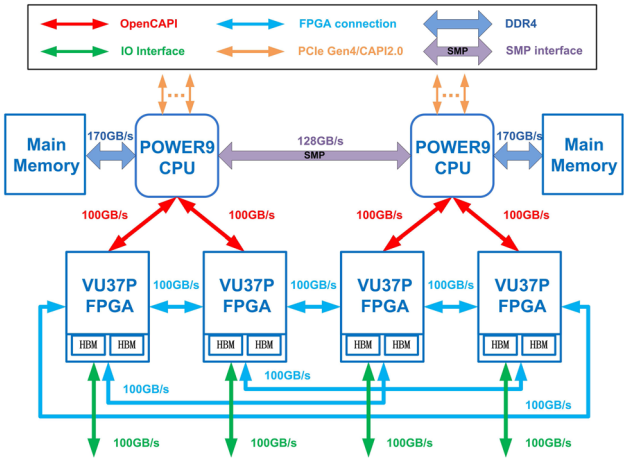


Fig. 5 Proposed FPGA-intensive configuration of POWER9 system (after [89])

example demonstrates that a system with all of the memory bandwidth being available to the accelerators is feasible.

5.5 Programmability trends

Even though the FPGA programmability is a serious and historical challenge that reduces the interest in accelerators for databases from developers, the FPGA community has made great progress and can be expected to keep the current momentum. The number of HLS tools from industry and academia is increasing, and examples include Vivado HLS [38], Altera OpenCL [115], Bluespec System Verilog [93], LegUp [17], DWARV [90], and Bambu [102], etc.

The first step that allowed compiling sequential code in software programming language such as C/C++ into hardware circuits by inserting HLS pragmas and adopting techniques such as loop unrolling, array partitioning, and pipeline mapping [26,28] have proved a milestone contribution. Now these tools are leveraging new techniques to further simplify the programming and enhance the performance. One recent trend is that HLS tools are integrating machine learning techniques to automatically set parameters for performance and controlling resource utilization [75,120,144]. In addition, these techniques can reduce the number of required HLS pragmas which further simplifies FPGA programming in HLS. Another recent change is the support of OpenMP [16,117,134]. OpenMP is one of the most popular languages for parallel programming for shared memory architecture. The support of OpenMP in HLS provides the potential of compiling parallel programs into FPGA accelerators that support shared memory. Because HLS is a big topic study by itself, we can not cover all the aspects. A recent survey [91] comprehensively studies the current HLS techniques and discusses the trends.

The emerging programming framework is another important achievement that contributes to the FPGA programmability, especially for hardware designers. These frameworks help in two different ways. First, such a framework can generate the memory interface for the designers with optimized memory controllers. An example is the SNAP [95] framework for CAPI/OpenCAPI which can take care of the low level communication protocol with the interface and abstract a simple burst mode data request. Another example is the Fletcher [101] framework which can generate interfaces for the Apache Arrow [4] in-memory tabular data format. With these program frameworks, the designer can save time from interface design and focus on the kernel design and performance tuning. Another benefit comes from the support of APIs that can manage the accelerators. These APIs typically wrap up the host accelerator management jobs and communication jobs into a package of software functions. The user only needs to choose and call the right APIs to access the hardware accelerators, which further improves ease of use.

Recently, the above techniques are being applied to the database domain. The prior study in [24,25] gives an overview of how software infrastructure can enable FPGA acceleration in the data center, where the two main enablers are the accelerator generation and the accelerator management. Other work studies SQL-to-hardware compilation. An example is Glacier [86], which can map streaming SQL queries into hardware circuits on FPGAs. Other studies work on the acceleration on database operators such as decompression [73], sort [6], and partitioning [133]. As mentioned before, the frameworks can support FPGA acceleration for databases in two ways, managing the hardware and provide APIs, the Centaur framework [97] is an example of leveraging these ideas for the database domain.

6 Acceleration of query operators

Even though there is not yet a commercial FPGA-accelerated in-memory database, a substantial body of prior work on accelerating database operators or components is pushing progress in this direction. In this section, we summarize the prior work on database operator acceleration including decompression, aggregation, arithmetic, sorts, joins, and others. An overview of the prior work is summarized in Table 2, where FPGA designers and performance analysts can have a quick view on the prior work. We also discuss the potential improvement for operator acceleration, which might be interesting for hardware designers. Most of this work shows that FPGA implementations of the kernels are efficient to the point where performance is limited by the bandwidth from host memory to the FPGAs. In most conventional systems this bandwidth is limited most by the PCIe connection to the FPGA.

Table 2 Summary of database operator acceleration using FPGAs

Operator category	Methods	References	Interface	Bandwidth	Throughput	Data source	FPGA	Frequency
Streaming operator (selection, projection, aggregation, arithmetic, Boolean, etc.)	Median	[84]	DDR*x*	1.6 GB/s	142 MB/s	Off-chip	Virtex-2	100 MHz
	Combined	[31]	PCIe Gen2	2 GB/s	1.13 GB/s	Host	Virtex-6	125 MHz
	Combined	[85]	Ethernet	1 Gbit/s	–	Network	Virtex-5	100 MHz
	Combined	[107]	Ethernet	1 Gbit/s	–	–	Virtex-5	–
	Combined	[124]	PCIe Gen*x*	–	2.31 GB/s	Host	Stratix V	200 MHz
	Combined	[125]	PCIe Gen2	4 GB/s	2.7 GB/s	Host	Stratix IV	200 MHz
	Combined	[126]	PCIe Gen2	4 GB/s	≈4 GB/s	Host	Stratix V	250 MHz
	Rex	[131]	–	–	2 GB/s	–	Virtex-4	125 MHz
	Rex	[113]	QPI	6.5 GB/s	6.4 GB/s	Host	Stratix V	200 MHz
Decompression	RLE	[33]	ICAP	800 MB/s	≈800 MB/s	Off-chip	Virtex-5	200 MHz
	Snappy	[34]	–	–	3 GB/s	–	KU15P	250 MHz
	Snappy	[105]	–	–	0.82 GB/s	–	KU15P	140 MHz
	LZSS	[68]	–	–	400 MB/s	–	Virtex-2	200 MHz
	GZIP/ZLIB	[141]	–	–	3.96 GB/s	–	KU060	165 MHz
	LZW	[74]	–	–	0.5 GB/s (in ASIC)	–	Spartan-3	68 MHz (FPGA)
	LZW	[148]	–	–	280 MB/s	–	Virtex-7	301 MHz
Sort	SN	[87]	–	–	52.45 GB/s	FPGA	Virtex-5	220 MHz
	FMS	[78]	DDR2	–	–	Off-chip	Virtex-5	166 MHz
	FMS	[69]	–	–	2 GB/s	FPGA	Virtex-5	252 MHz
	Merge tree	[69]	–	–	1 GB/s	FPGA	Virtex-5	273 MHz
	Merge tree	[18]	DDR*x*	38.4 GB/s	8.7 GB/s	Off-chip	Virtex-6	200 MHz

Table 2 continued

Operator category	Methods	References	Interface	Bandwidth	Throughput	Data source	FPGA	Frequency
	Merge tree	[79]	–	–	77.2 GB/s	FPGA	Virtex-7	311 MHz
	Merge tree	[118]	–	–	24.6 GB/s	FPGA	Virtex-7	99 MHz
	Merge tree	[119]	–	–	9.54 GB/s	FPGA	Virtex-7	200 MHz
	Merge tree	[145]	–	–	26.72 GB/s	FPGA	VU3P	208 MHz
	Merge tree	[108]	–	–	126 GB/s	FPGA	Virtex-7	506 MHz
	Merge tree	[20]	DDR3	10 GB/s	7.9 GB/s	Off-chip	Virtex-7	250 MHz
Join	SMJ	[18]	DDRx intra-FPGA	115 GB/s 4 GB/s	6.45 GB/s	Off-chip	Virtex-6	200 MHz
	SMJ	[20]	DDR3	3.2 GB/s	0.69 GB/s	Off-chip	Zynq	100 MHz
	Hash join	[110]	DDR3	10 GB/s	–	Off-chip	Virtex-7	200 MHz
	Hash join	[49]	DDR3	76.8 GB/s	12 GB/s	Off-chip	Virtex-6	150 MHz
	Hash join	[50]	–	3 GB/s	18M rows/s	FPGA	Stratix IV	206 MHz
Partitioning	PHJ and Groupby	[21]	DDR3	–	–	Off-chip	Zynq	100 MHz
	Partitioning	[61]	QPI	6.5 GB/s	3.83 GB/s	Host	Stratix V	200 MHz

Operator: *Combined* a combination of multiple streaming operators, SN sorting network, FMS FIFO merge sorter, SMJ sort–merge join, PHJ partitioning hash join.
Data Source: $FPGA$ the data is generated by FPGAs or stored in FPGA memory, $off-chip$ the data is stored in the off-chip memory in the accelerator side, *host* the data is stored in the host memory side

6.1 Decompression

Decompression is widely used in database applications to save storage and reduce the bandwidth requirement. The decompressor works as a translator, reading a compressed stream consisting of tokens, translating the tokens into data itself, and outputting a decompressed stream. There are many different (de)compression algorithms. Since in database applications we do not want to lose any data, we consider lossless (de)compression algorithms in this survey paper. The most popular two types of (de)compression in database systems are the *Run-Length Encoding* (RLE) [111] and the *Lempel-Ziv* (LZ) series. This paper focuses on decompression algorithms instead of compression algorithms, even though there are many studies [1,11,40] on compression acceleration. An important reason is that in database systems, the common case is to compress the data once and to decompress it more frequently.

6.1.1 RLE

RLE is a simple form of a compression algorithm that records a token with a single value and a counter indicating how often the value is repeated instead of the values themselves. For example, a data sequence "`AAAAAAAABBBC`" after RLE compression is "`8A3B1C`". In this case, instead of storing 12 bytes of raw data, we store 6 bytes, or 3 tokens with each token in fixed size (1 byte counter and 1 byte value). The RLE decompression works in reverse. The RLE decompressor reads a fixed size token, translates it into a variable-length byte sequence, and attaches this sequence to the decompressed data buffer built from the previous tokens.

The method proposed in [33] shows that their FPGA-based RLE implementation can help reduce the FPGA reconfiguration time and achieves a throughput of 800 MB/s which is limited by the *Internal Configuration Access Port* (ICAP) bandwidth. It is not difficult to parallelize this translation procedure. As the token has a fixed size, the decompressor can explicitly find out where a token starts without the acknowledgement of the previous token, and multiple tokens can be translated in parallel. The write address of each parallel processed token can be provided in the same cycle by adopting prefix-sum on the repeating counter. Thus, we can imagine that a multi-engine version of this implementation can sufficiently consume the latest interface bandwidth.

6.1.2 LZ77-based

Instead of working on the word level, LZ77 [149] compression algorithms leverage repetition on a byte sequence level. A repeated byte sequence is replaced by a back reference that indicates where the previous sequence occurs and how long it is. For those sequences without duplicates, the original

data is stored. Thus, a compressed file consists of a sequence of tokens including copy tokens (output the back reference) and literal tokens (output the data itself). During the compression, a history buffer is required to store the most recent data for finding a matched sequence. Similarly, maintaining this history buffer is a prerequisite for copy tokens to copy context from during the decompression. Typically, the size of history buffer is on the order of tens of KB level and depends on the algorithms and their settings.

Decompression translates these two types of tokens into the original data. For literal tokens, the decompressor selects the original data stored in the tokens and writes it into the history buffer. For copy tokens, the back reference data including the copied position and copied length is extracted, followed by a read from the history buffer and a write to the history buffer. There are many extensions to this algorithm, e.g., LZ4 [22], LZSS [122], Gzip[2] [32] and Snappy [44].

In an FPGA, the history buffers can be implemented using shift registers [68] or BRAMs [54], and the token decoding and the BRAM read/write can be placed in different pipeline stages. While pipeline design can ensure continuous processing of the compressed data, the throughput declines when data dependencies occur. The LZ4 decompression proposed in [77] uses separate hardware paths for sequence processing and repeated byte copying/placement, so that the literal tokens can always be executed since they contain the original data and are independent of the other tokens. Separating the paths ensures these tokens will not be stalled by the copy tokens. A similar two-path method for LZ77-based decompression is shown in [46], where a slow-path routine is proposed to handle large literal tokens and long offset copy tokens, while a fast-path routine is adopted for the remaining cases. This method is further demonstrated at the system level in [45] to hide the latency of slow operations and avoid stalls in the pipeline.

Even though a single-engine FPGA implementation can outperform a CPU core, it is not easy to exceed a throughput of one token per cycle per engine. To saturate the bandwidth from a high-bandwidth connection, we can either implement multiple decompressor engines in an FPGA or implement a strong engine that can process multiple tokens per cycle. A challenge of implementing multiple engines is the requirement of a powerful scheduler that can manage tens of engines, which also drains resources and might limit the frequency. In addition, the implementation of the LZ77-based decompressor in FPGAs takes much more memory resources [141], especially the BRAMs, limiting the number of engines we can place in a single FPGA. Apart from that, the unpredictable block boundaries in a compressed file also bring challenges to decompressing multiple blocks in parallel [58]. As an alternative, researchers also look for intra-block parallelism. However, the demands of processing multiple tokens in parallel pose challenges including handling the various token sizes, resolving the data dependencies and BRAM bank conflicts. A parallel variable length decoding technique is proposed in [2] by exploring all possibilities of bit spill. The correct decoded streams among all the possibilities are selected in a pipelined fashion when all the possible bit spills are calculated and the previous portion is correctly decoded. A solution to the BRAM bank conflict problem is presented in [105] by duplicating the history buffer, where the proposed Snappy decompressor can process two tokens every cycle with throughput of 1.96 GB/s. However, this method can only process up to two tokens per cycle and is not easy to scale up to process more tokens in parallel due to the resource duplication requirement. To reduce the impact of data dependencies during the execution of tokens, Sitaridi et al. [116] proposed a multiround execution method that executes all tokens immediately and recycles those copy tokens that return with invalid data. The method proposed in [34] improves this method to adopt the parallel array structure in FPGAs by refining the tokens into BRAM commands which achieve an output throughput of 5 GB/s.

For LZ-77-based decompression accelerators that need a history buffer (e.g., 64 KB history for Snappy), a light engine that processes one token per cycle would be BRAM limited, while a strong engine that processes multiple token per cycle might be LUT limited. Even the design [34] with the best throughput cited in this paper is not in perfect balance between LUTs and BRAMs for the FPGA it uses, and there is room for improvement.

6.1.3 Dictionary-based

Dictionary-based compression is another commonly used class of compression algorithms in database systems, the popular ones of which are the LZ78 [150] and its extension LZW [135]. This class of compression algorithms maintains a dictionary and encodes a sequence into tokens that consist of a reference to the dictionary and the first non-matched symbol. Building the dictionary lasts for the whole compression. When the longest string matches the current dictionary, the next character in the sequence is appended to this string to construct a new dictionary record. It is not necessary to store the dictionary in the compressed file. Instead, the dictionary is reconstructed during decompression.

A challenge to designing efficient decompression in FPGAs is to handle the variety of string length in the dictionary. When adopting fixed-width dictionaries, while setting a large width for the string wastes a lot of memory space, using a small string width suffers from throughput decrease since multiple small entries must be inspected to find the match. Thus, a good design for these decompression algorithms demands explicit dictionary mechanisms that can efficiently

[2] Gzip is an implementation of DEFLATE [62].

make use of the FPGA's capability of bit-level processing. A two-stage hardware decompressor is proposed in [74] which combines a parallel dictionary LZW with an Adaptive Huffman algorithm in a VLSI, achieving 0.5 GB/s data rate for decompression. The study in [148] presents an efficient LZW by storing the variable-length strings in a pointer table and a character table separately. The implementation of a single instance of this algorithm consumes 13 18 Kb BRAMs and 307 LUTs in an XC7VX485T-2 FPGA, achieving 300 MHz frequency and 280 MB/s throughput.

6.1.4 Discussion

Memory access patterns Table 3 compares the host memory access patterns of operators discussed in this survey, assuming that the source data is initially stored in the host memory. The decompression has a "sequential" memory access pattern since it has streaming input and streaming output. Typically the decompressor outputs more data then the input.

Bandwidth efficiency As we mentioned before, the RLE algorithms can easily achieve high throughput that can meet the interface bandwidth bound, but the LZ77 and LZ78 series are challenging due to the large number of data dependencies. According to our summary, most of the prior work can not reach the latest accelerator interface bandwidths, but some designs [34] can. This depends on many factors including the algorithm itself, hardware design and its trade-offs, including LUT-to-BRAM balance, and FPGA platforms. For a multi-engine implementation, the throughput is defined by the product of throughput per engine and the number of engines. The challenge is that in an FPGA, we can either have strong decompression engines but fewer of them or more less powerful engines. Thus, a good trade-off during design time is indispensable to match the accelerator interface bandwidth.

6.2 Streaming operators

6.2.1 Streaming operators

Streaming operators are database operations where data arrives and can be processed in a continuous flow. These operators might belong to different categories of database operators such as selections [85,107,124,125,139], projections [85,107,124,126], aggregations (sum, max, min, etc.) [31,84,86,97], and regular expression matching [113,131]. We place them together because they typically act as pre-processing or post-processing in most of the queries and have similar memory access patterns.

Projections and selections are filtering operations that only output the fields or records that match the conditions, while the aggregations are performing arithmetic on all the inputs. Due to the pipeline style design in FPGAs, these operations can be performed in a stream processing model in the FPGA implementation, at high bandwidth and with low overall latency. A multi-engine design of these operators, by adopting parallelism at different levels, can easily achieve throughput that may exceed the accelerator interface bandwidth and even get close to the host memory bandwidth. Regular expression matching can be used to find and replace patterns in strings in databases, such as "REGEXP_LIKE" in SQL. The performance of regular expression matching is bounded by the computation in software due to the low processing rate of software deterministic finite automaton. However, it can be mapped to custom state machines in the FPGA and gain performance from a pipelined design.

Even though the implementation of these kernels in FPGAs is trivial, acceleration of the combination of these operators and other operators is non-trivial. Typically, a query is executed according to a query plan that consists of several operators. In software, the operator order is decided and optimized by the query compiler and the query optimizer. Choosing an order of these operators can reduce the amount of data running in the pipeline and avoid unnecessary loads and stores of the intermediate results. Conversely, an irrational query plan can cause extra data accesses and waste the communication resources. Similarly, to achieve high throughput, the consideration of combining different FPGA-implemented operators in a reasonable order is indispensable. There are many methodologies to optimize the query order, a basic one of which is filtering data as early as possible. This idea has been reported in many publications. For instance, the implementation in [124] executes projections and selections before sorting. The compiler proposed in [85] supports combinations of selections, projections, arithmetic computation, and unions. The method from [107] allows joins after the selection and the projection.

6.2.2 Discussion

Memory access pattern The streaming operators have streaming reads and streaming write or can only write a single value (such as sum). They typically produce less output data compared to the input, especially for aggregation operators that perform the arithmetic. In other words, the streaming operators have sequential access patterns to the host memory.

Bandwidth efficiency A single engine of the streaming operators can easily reach GB/s or even tens of GB/s magnitude throughput. For example, the sum operation proposed in [31] shows a throughput of 1.13 GB/s per engine, which is limited by the connection bandwidth bound of their PCIe x4 Gen 2 connected platform. A combination of decompression, selection, and projection presented in [124] reports a kernel processing rate of 19 million rows/s or 7.43 GB/s (amplified by decompression) which exceeds the bandwidth of PCIe used in their reported platform. Since these opera-

Table 3 Summary of memory access in different operators

Operator category	Methods	Host memory access pattern	In-memory intermediate result	Multi-pass host memory access
Streaming operator	Selection	Sequential	No	No
	Projection			
	Aggregation			
	Arithmetic			
	Boolean			
	Rex			
Decompression	RLE	Sequential	No	No
	LZ77-based			
	Dictionary-based			
Partitioning	Partitioning	Scatter	No	No
	Multi-pass partitioning	Scatter	Yes	Yes
Sort	Sorting network	Sequential	No	No
	FIFO merge sorter	Streaming gather	No	No
	Merge tree	Streaming gather	No	No
	Multi-pass merge tree	Streaming gather	Yes	Yes
	Partitioning sort	Scatter sequential	Yes	Yes
Join	Sort–merge join	Patterns of sort streaming gather	Yes	Yes
	Hash Join (Small Dataset)	Sequential	No	No
	In-memory hash join	Scatter (build) random (probe)	Yes	No
	Partitinging hash join	Scatter sequential	Yes	Yes

For some aggregation and arithmetic algorithms, it only requires a single value as the result to write back
Streaming Gather is similar to Gather but from multiple streams, and the next read depends on the current state

tors do not require resource-intensive functions in FPGAs, an instance of multiple engines can be easily implemented in a single FPGA to achieve throughput that is close to the interface bandwidth. Although multiple engines need to read from and write to different streams, this won't increase the data access control since the streams are independent. Thus, a theoretically achievable throughput upper bound of these operators is the accelerator interface bandwidth.

6.3 Sort

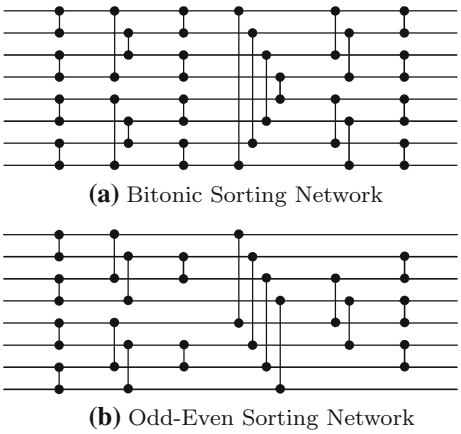
Sorting is a frequently used operation in database systems for ordering records. It is can be used in *ORDER BY* in SQL and in a more complex query to improve the performance. Large scale sort benchmarks are considered key metrics for database performance.

In a CPU-based sort, the throughput is limited by the CPU computation capacity, as well as the communication between computational nodes. For the record holder for a large multi-node sort the per node performance is about 2 GB/s [94]. Single-node sort throughput without the communication overhead may be somewhat larger, but we believe that the single-node performance would be within the same order of magnitude. As network speed increases rapidly, and storage is replaced with NVMe devices where storage bandwidth grows rapidly, sort with CPUs is not going to keep up. Therefore, accelerators are now required for this operation that historically was bandwidth-bound. To reach this goal, many hardware algorithms have been presented using FPGAs to improve performance. In this section, we give an overview of the prior FPGA-based sort algorithms.

6.3.1 Sorting network

A sorting network is a high throughput parallel sort that can sort N inputs at the same time. The compare-and-swap unit is the core element in a sorting network. A compare-and-swap unit compares two inputs and arranges them into a selected order (either ascending or descending), guaranteed by swapping them if they are not in the desired order. Using a set of these compare-and-swap units and arranging them in a specific order, we can sort multiple inputs in a desired order.

A simple way to generate a sorting network is based on the bubble sort or insertion sort algorithm. Thus, these types of sorting networks require $O(N^2)$ compare-and-swap units and $O(N^2)$ compare stages to sort N inputs. More efficient methods to construct the network include the bitonic sorting network and the odd-even sorting network [12]. Figure 6 shows the architecture of the bitonic sorting network (Fig. 6a) and the odd-even sorting network (Fig. 6b) with 8 inputs. We can further pipeline the designs by inserting registers after each stage. Knowing that it takes one cycle for a signal to cross one stage in a pipeline design, both sorting networks take $O(\log^2 N)$ cycles to sort N inputs, while the space complexity is $O(N \log^2 N)$.



(a) Bitonic Sorting Network

(b) Odd-Even Sorting Network

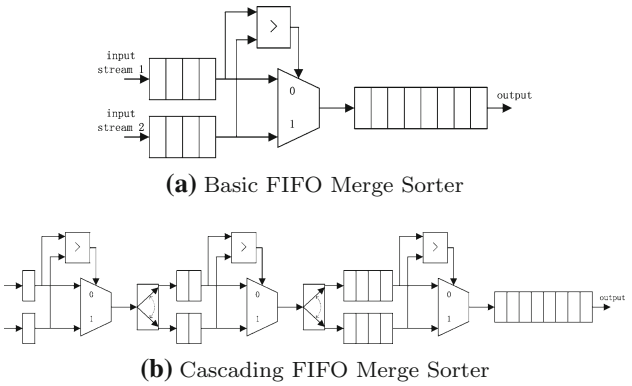
Fig. 6 Architecture of sorting network

A sorting network can sort multiple data sets concurrently by keeping different data sets in different stages. An N-input sorting network is able to process N elements per FPGA cycle. The sorting network proposed in [87] outputs 8 32-bit elements per cycle at 267 MHz, meaning a throughput of 7.9 GB/s. It is not difficult to increase the throughput by scaling up the sorting network for a larger input number. A 64-input sorting network at 220 MHz based on the implementation in [87] can consume data at 52.45 GB/s, approximately equivalent to the bandwidth of two OpenCAPI channels (51.2 GB/s). However, the required reconfigurable resources increase significantly with the increase in the number of inputs. Thus, a sorting network is generally used for the early stages of a larger sort to generate small sorted streams that can be used as input for the FIFO merge sort or merge tree in the later stages.

6.3.2 FIFO merge sorter

The *first-in first-out* FIFO merge sorter is a sorter that can merge two pre-sorted streams into a large one. The key element is the select-value unit. It selects and outputs the smaller (or larger) value of two input streams. The basic FIFO merge sorter is illustrated in Fig. 7a. Both inputs are read from two separate FIFOs that store the pre-sorted streams, and a larger FIFO is connected to the output of the select-value unit. In [78], an unbalanced FIFO merge sorter is proposed which shares the output FIFO with one of the input FIFOs to save FIFO-resources. The proposed architecture is able to sort 32K 32-bit elements at 166 MHz, consuming 30 36 Kb blocks (30 out of 132 from a Virtex-5 FPGA).

A drawback of the FIFO merge sorter is that it takes many passes to merge from small streams to the final sorted stream since it only reduces the number of streams into half each pass, especially when handling large data sets that have to



(a) Basic FIFO Merge Sorter

(b) Cascading FIFO Merge Sorter

Fig. 7 Architecture of FIFO merge sorter

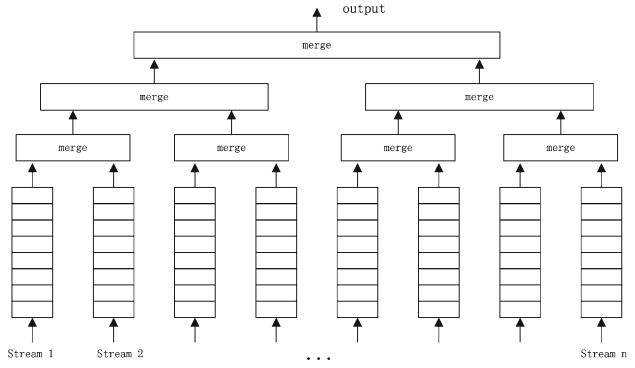


Fig. 8 Architecture of merge tree

store into host memory. Sorting a data set with 1024 small pre-sorted streams requires 10 passes, which means the data needs to travel between the memory and the FPGA multiple times. In this case, the overhead of the data transmission dominates, and the overall performance is limited by the interface bandwidth. This problem can be solved by cascading multiple FIFO merge sorters. As shown in Fig. 7b, FIFO merge sorters with smaller FIFOs are placed in the earlier stages, while those with larger FIFOs are inserted at later stages. An improved method is presented in [69] where the proposed cascading FIFO merge sort can reduce the pipeline filling time and emptying time by starting to process the merge as long as the first element of the second FIFO has arrived. This implementation can sort 344 KB of data at a throughput of 2 GB/s with a clock frequency of 252 MHz. However, in the cascading FIFO merge sorter, the problem size is limited by the FPGA internal memory size, since it needs to store all the intermediate results of each merge sort stage.

6.3.3 Merge tree

For data sets that do not fit in the internal FPGA memory, we need merge trees. The merge tree can merge several sorted streams into one larger stream in one pass. As shown in Fig. 8, a merge tree is constructed by a set of select-value units arranged in multiple levels. In each level, the smaller elements between two streams are selected which will be sent to the next level to select the smallest among these four streams. This process is iterated until the largest element among all the input streams is selected. FIFOs are inserted between the leaves of the tree and the external memory to hide the memory access latency. To pipeline the whole merge and reduce the back pressure complexity from the root, small FIFOs are placed between the adjacent levels. An M-input merge tree merges N streams into $\frac{N}{M}$ streams each pass. Compared to a FIFO merge sorter, it reduces the number of host memory accesses from $\log_2 N$ to $\log_M N$. The merge tree implementation in [69] provides a throughput of 1 GB/s to sort 4.39 million 64-bit elements. Partial reconfiguration was used to configure a three-run sorter (one run of the FIFO merge sorter and two runs of the tree merge sorter) that can handle 3.58 GB of data and achieve an overall throughput of 667 MB/s.

For a multi-pass merge sort, even though we can place multiple merge trees to increase the system throughput, the final pass has to guarantee all the remaining streams are combined into a single sorted stream. This demands a stronger merge tree that can output multiple elements per cycle. The study in [18] presents a multi-element merge unit. This unit compares N elements from the selected stream with N feedback elements produced in the previous cycle and selects the smallest N ones as output, while the remaining N elements will be fed back for the next cycle comparison. The proposed merge tree achieves a throughput of 8.7 GB/s for a two-pass merging which reaches 89% utilization of the interface bandwidth.

A. Srivastava et al. [119] proposed a multi-output merge tree architecture by adopting increasing-in-size bitonic merge units (BMUs). In this architecture, the BMU in the leaf level compares the first element from both input streams and outputs one element per cycle. The next level BMU compares the first two elements from both streams and outputs two elements per cycle. The size of the BMU doubles from one level to the next level until it reaches the root. As a small BMU is much more resource efficient than a large one, as it saves hardware resources in the FPGA, bringing high scalability. A drawback of this method is that the throughput for skewed data drops. A similar architecture [118] where the merge tree is able to output 32 64-bit elements in one cycle reports a throughput of 24.6 GB/s. However, this paper uses a wrapper to produce the input data instead of reading from the external memory, where the control of feeding 32 input streams would consume internal memory of the FPGA as well as potentially reducing the clock frequency. A clock frequency optimization method is illustrated in [79] by deeply pipelining the multi-element merge unit. An instance of this method with 32 outputs per cycle operates at 311 MHz. The

method proposed in [108] further raises the frequency of the merge tree to 500 MHz by breaking the feedback loop.

Based on the observation that it might consume all the BRAM resources in an FPGA to hide the interconnect latency in a wide merge tree, an alternative is presented [145] of an "odd-even" merge sorter combined with a multi-stream interface [88] that can deal with the data skew and supply a stable throughput for any data distribution. The multi-stream interface is constructed as a two-level buffer using both the BRAM and URAM resources. The evaluation shows that the "even-odd" merge sorter kernel can merge 32 streams, providing a throughput of 26.72 GB/s. However, even with the two-level buffer solution, the buffer and the associate logic still require a majority of the FPGA resources.

Even though speedups are gained from the FPGA acceleration, we have to note that the maximum throughput is not going to exceed the interface bandwidth divided by the passes of memory accesses. Thus, a good way to design a sorter needs to trade off between the number of sorting passes and the throughput of each pass.

6.3.4 Discussion

Memory access pattern Different sort algorithms have different memory access patterns. An N-input sorting network typically is used in an early stage of sorting to convert an input stream into an output stream that is a sequence of N-element sorted substreams. The FIFO merge sort has two sorted streams as inputs that will be merged into a larger sorted stream. However, the two inputs are not independent from each other. This is because the next data to be read relies on the comparison result of the current inputs. Since this access pattern is similar with the pattern of "gather" (indexed reads and sequential writes), but the next reads depend on the current inputs, we refer to this as a streaming gather pattern. Thus, the FIFO merge sort has dependent multi-streaming read and streaming write memory access patterns. Similarly, the merge tree has the same access pattern as FIFO merge sort, with the next input might come from multiple streams instead of two.

Bandwidth efficiency All three classes of sort methods mentioned above can sufficiently utilize the interface bandwidth by making stronger engines or deploying more engines. However, a large sort may require multiple passes that each requires the host memory to be accessed. In this manner, the overall throughput of an entire sort depends on both the number of passes and the throughput of each pass, or the overall throughput does not exceed the bandwidth divided by the number of passes. The number of passes can be reduced by a wider merger that merges more streams into one larger stream. However, because each input stream requires buffering, building up a wider merge tree requires a lot of BRAMs, which makes the design BRAM limited.

Buffer challenge Reducing the number of memory access passes is a key point to improve the sort throughput. One way to do this is to sort as much data as possible in one pass. For example, use a wider merge tree to merge more streams. However, in an FPGA, hiding the host memory latency for multiple streams is a challenging problem, let alone for the dependent multi-streaming accesses. In a merge tree, when merging multiple streams, which stream is chosen next cannot be predicted. Even though buffers can be placed at the inputs of each stream, naively deploying buffers for each input to hide the latency would consume all the BRAMs in an FPGA or even more. In the case of a 64-to-1 merge tree that can output 8 16B elements each cycle, to hide the interconnect latency (assume 512 FPGA cycles in a 250 MHz design), 4 MB of FPGA internal memory resources are demanded (which is all the BRAM resources for a Xilinx KU15P).

HBM benefit HBMs on the FPGA card bring the potential to reduce the number of host memory accesses for sort. It provides accelerators with larger bandwidth and delivers comparable latency compared to the host memory. Thus, instead of writing the intermediate results back to the host memory, we can store them in the HBMs. For a data set that fits in an HBM, the analysis from [145] illustrates that it only demands one pass read from and write to the host memory with the help of HBM (the read in the first pass and the write in the final pass), while without HBM it requires five passes access to the host memory. For data sets larger than the HBM capacity, using HBMs for the first several passes of sorting can reduce a remarkable number of host memory accesses.

Partitioning methods The number of host memory accesses goes up fast once the data set size is larger than the HBM capacity. For example, using a 32-to-1 merge tree to sort 256 GB data in an FPGA equipped with 8 GB HBM demands two passes, including one pass to generate 8 GB sorted streams and one pass to merge these 8 GB streams into the final sorted stream. However, with every 32 times increase in the data set size, an extra pass is required.

The partitioning method is a solution to reduce the host memory accesses and to improve the memory bandwidth efficiency. It was previously used to obtain more concurrency by dividing a task into multiple sub-tasks. Another benefit is that it can enhance the bandwidth efficiency.

For sorting large data sets in GPUs, a common method is the partitioning sort [39] where a partitioning phase is executed to partition the data set into small partitions and a follow-up sort phase to sort each of them. This method only demands two passes of host memory accesses and can achieve a throughput of up to one-fourth of the host memory bandwidth (two reads and two writes). Experimental results in [39] illustrate a throughput of 11 GB/s in a single GPU node which is 65% of the interface bandwidth.

This method is feasible in FPGAs if the HBM is integrated. In this way, to sort the whole data set, two passes

accessing the host memory is sufficient. In the first pass, the data is partitioned into small partitions that fit in HBM, and write the partitions back to the host memory. The second pass then reads and sorts each partition and writes the sorted streams back. As this method needs fewer host memory accesses compared to the merge sort, it is interesting to study the partitioning sort in FPGAs and compare it with the multi-pass merge tree method. The partitioning methods are also applicable to hash joins to enhance the memory bandwidth efficiency. This is described in detail in Sect. 6.4.

6.4 Join

Joins are a frequently used operation in database systems, that combine two or more tables into one compound table under specific constraints. The most common one is the equi-join that combines two tables by a common field. In the rest of this paper, we refer to equi-joins as joins. There are many different join algorithms including *nested loop join*, *hash join* (HJ), and *sort–merge join* (SMJ). In this paper, we focus on the HJ and SMJ since they are more interesting to the database community due to their low algorithmic complexity.

6.4.1 Hash join

Hash join is a linear complexity join algorithm. It builds a hash table from one of the two join tables and uses the other table for probing to find matches. The probing time for each element remains in constant time if a strong hash function is chosen. In CPUs, it is difficult for a hash function to have both strong robustness and high speed. In FPGAs, this trade-off is broken because FPGAs allow a complex algebra function implemented in a circuit that calculates faster than the CPU does. Kaan et al. [60] shows a murmur hash FPGA implementation that achieves high performance as well as strong robustness. Research from [67] points out that the indexing takes up most of the time for index-based functions, especially the walk time (traversal of the node list) which accounts for 70%. To solve this problem, the *Widx* ASIC, an index traversal accelerator tightly connected to the CPU cores, is proposed to process the offloaded index-based walker workloads. Widx decouples the hash unit and the walker (the traversal unit) and shares the hash unit among multiple walkers to reduce the latency and to save the hardware resources.

One of the key points to design an FPGA-based hash join algorithm is to have an efficient hash table structure. On one hand, the hash table structure influences the performance of the hash join engine. An inappropriate hash table design might introduce stalls, reducing the throughput. On the other hand, as there is a limited number of BRAMs inside an FPGA, to ensure a multi-engine instance is feasible in one FPGA, a hash table should not consume too many BRAMs. The method in [130] makes use of most of the BRAMs in an FPGA to construct a maximum size hash table. In this method, the BRAMs are divided into groups. The hash table connects the BRAM groups into a chain, and different hash functions are adopted for different BRAM groups. In the case of hash collisions, conflicting elements will be assigned to the next group until the hash table overflows. Even so, due to the skewed data distribution, some of the memory may be wasted. The hash join in [50] uses two separate tables to construct the hash table, including one *Bit Vector* table storing the hash entries and one *Address Table* table maintaining the linked lists. The proposed architecture allows probing without stalls in the pipeline.

For data sets that are too large to store the hash table in the FPGA's internal memory, in-memory hash joins are needed. As the BRAMs can only store part of the hash table, probing tuples demands loading the hash table multiple times from the main memory. However, the latency of accessing main memory is much larger than that of accessing BRAMs. One way to deal with this problem is to use the BRAMs as a cache [110]. However, to achieve high throughput, an efficient caching mechanism is required. If the size of the hash table is much larger than the BRAMs, the cache miss ratio might remain too large to benefit from the cache system. Another way to hide the memory latency is to use multi-tasking. As FPGAs have a large amount of hardware resource for thread states, we can keep hundreds of tasks in an FPGA. An example is shown in [49] where the hash join runs against a Convey-MX system and achieves a throughput of up to 12 GB/s or 1.6 billion tuple/s. However, when applying this method to other architectures, we need to avoid suffering from the granularity effect [36]. Because each access to the main memory must obey the access granularity that is defined by the cache line, every read/write request always acquires the whole cache line(s) of data. If a request does not ask data that covers the whole cache line(s), part of the response data is useless, meaning a waste of bandwidth. Thus, the hash table needs to be properly constructed to reduce or avoid the occurrence of this situation.

Another way to process in-memory hash joins is to partition the data before performing the joins. Both the input tables are divided into non-intersecting partitions using a same partition function. One partition from a table only needs to join with the corresponding partition in the other table. If the hash table of a partition is small enough to fit in the internal FPGA memory, the hash table is required to be loaded only once. The main challenge is how to design a high throughput partitioner. The study in [140] presents a hardware-accelerated range partitioner. An input element is compared with multiple values to find a matched partition, after which this element is sent to the corresponding buffer. In this work, deep pipelining is used to hide the latency of multiple value comparisons. Kaan et al. [61] proposed a hash partitioner that can contin-

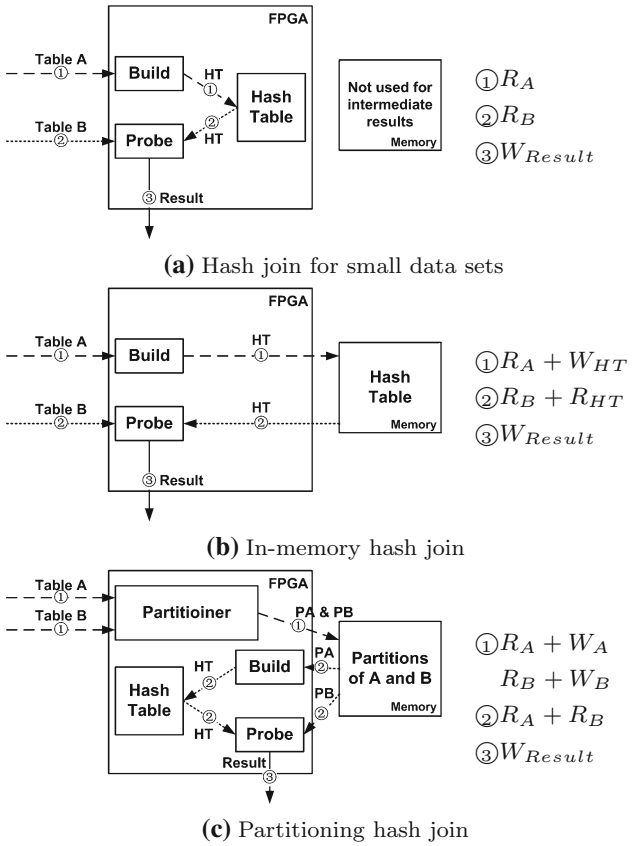


Fig. 9 Data flow regarding main memory accesses in different hash join algorithms. R_x stands for reading x, while W_x stands for writing x. Note that the number of writes to the hash table (W_{HT}) is based on the size of Table A, and the number of reads from the hash table (R_{HT}) is based on the size of Table B

uously output a 64B cache line in a 200 Mhz FPGA design. Write combiners are used to construct a full cache line output to avoid granularity effects [36]. To keep up with the QPI bandwidth, the authors implement multiple engines in an FPGA. Their end-to-end example shows around 3.83 GB/s for partitioning the data in an FPGA and 3 GB/s for a hybrid partitioning hash join (partition in the FPGA and join in the CPU). If the data set is too large and the partition size does not fit in the FPGA's internal memory, a multi-pass partitioning can be adopted. An example is demonstrated in [21] where *LINQits* is proposed to accelerate database algorithms including the hash join and the group-by operator.

6.4.2 Sort–merge join

The sort–merge join is comparable to the hash join algorithm. It first sorts both tables and does a merge step afterward. The most challenging part of a hardware sort–merge join is the sort itself which we have discussed in Sect. 6.3. To further improve the sort–merge join performance, the merge step can be started as long as the first sorted element from the second table is output. The sort–merge join proposed in [18] adopts a similar idea and achieves a throughput of 6.45 GB/s for the join. Ren et al. [20] proposed a sort join in a heterogeneous CPU-FPGA platform. It performs the first few sorting stages in FPGAs and streams the partial results to the CPUs for a later merge sort step, and the merge join afterward.

Others study the comparison between the hash join and the sort–merge join. This topic has been well studied for the CPU architecture [3,8,64], but not too much for FPGA-based accelerators. The work in [130] studies the FPGA-based hash join and the FPGA-based sort–merge join, and claims that the sort–merge join outperforms the hash join when the data sets become larger. However, the in-memory hash join is not included in this comparison which is more suitable for larger data sets. A detailed analysis and comparison is explained in [8] on a multi-core CPU platform. According to the experimental results, the optimized hash join is superior to the optimized sort–merge join, while for large data sets the sort–merge join becomes more comparable. Even though this analysis is based on the CPU architecture, we believe that the principles of the analysis are similar and it would be a good guideline for further analysis based on the FPGA architecture.

6.4.3 Discussion

Memory access pattern Typically hash joins have a streaming read pattern for reading both tables and a streaming write pattern for writing the results back, but accessing the hash table, including establishing the hash table and probing it, is random. If the data set is small enough that the hash table can be stored in the accelerator memory, accessing the hash table becomes internal memory accesses. Thus, from the host memory access aspect, hash joins for small data sets only have streaming read and streaming write patterns. Partitioning a large data set into small partitions before doing the joins allows the hash table to be stored in the accelerator memory, which can avoid the random access to the host memory. The extra memory access passes introduced by the partitioning phase hold scatter patterns (read from sequential addresses and write to indexed/random addresses).

For sort–merge join, the memory access pattern of sorting the two tables is studied in Sect. 6.3.4. Similar with the FIFO merge sorter, the join phase reads both sorted tables in dependent streaming ways and writes the results back as a single stream. Accordingly, the sort–merge join has the same memory access pattern as the sort, plus the streaming gather memory access pattern.

Bandwidth efficiency The data flows of different types of hash joins are illustrated in Fig. 9. The memory shown in the figure is the host memory. However, the data flows can also

be applied in the off-chip memory on the accelerator side. Hash joins on small data sets are streaming-like operations. Both tables are read from the host memory, while the hash table is stored in the FPGA internal memory. Thus, a hash join requires a pass reading the data and a pass writing the results back. As hash joins are not highly computational, this streaming read and streaming write can saturate the accelerator interface bandwidth.

For large data set cases, the hash table needs to be stored in host memory. During the build phase, each tuple in the build table needs to write the hash table. Similarly, each tuple in the probe table during the probe phase generates at least one read to the hash table. Consequently, hash joins on large data sets equivalently demand two passes of host memory accesses for the original tables and the hash table, and one pass writing the results back. However, accessing the hash table results in random access patterns, which suffer performance decrease due to cache misses [14], TLB misses [9,10], the *Non-uniform memory access* (NUMA) effect [71], and the granularity effect [36], etc. The study in [36] shows that only 25% of the memory bandwidth is effective during the access to the hash table in a 64B cache line machine if the tuple size is 16B.

The partitioning hash join can avoid random access by splitting the data into small partitions such that the hash table for each can be stored in the FPGA memory. One drawback of this method is that the partitioning phase introduces extra memory accesses. The cost of partitioning even becomes dominating if multi-pass partitioning is inevitable. However, the partition can be implemented in a streaming processing way, which has streaming read and multi-streaming write patterns. For a one-pass partitioning hash join, two passes of streaming read and streaming write to the host are required. Thus, the throughput can reach up to one-fourth of the bandwidth.

HBM benefit The HBM technology can be adopted in the hash joins as well. For non-partitioning hash joins, HBM can be used as caches or buffers, leveraging the locality and hiding the latency of host memory accesses. It also allows smaller granularity access than DRAM, and therefore can reduce the granularity effect.

For partitioning hash joins, HBMs can help in two ways. One way is to use HBMs to store the hash table of each partition, which allows larger size partitions and can reduce the required number of partitions. The other way is to use HBMs as buffers to buffer the partitions during the partitioning phase, which provides the capability of dividing the data into more partitions in a pass. Both methods can reduce the number of partitioning passes, leading to fewer host memory accesses.

7 Future research directions

In this section we discuss future research directions as well as their challenges for different groups of researchers including database architects, FPGA designers, performance analysts, and software developers.

7.1 Database architecture

We believe that in-memory databases should be designed to support multiple types of computational elements (CPU, GPU, FPGA). This impacts many aspects of database design:

- It makes it desirable to use a standard data layout in memory (e.g., Apache Arrow) to avoid serialization/deserialization penalties and make it possible to invest in libraries and accelerators that can be widely used.
- It puts a premium on a shared coherent memory architecture with accelerators as peers to the host that can each interact with the in-memory data in the same way, thus ensuring that accelerators are easily exchanged.
- It implies we need to develop a detailed understanding of which tasks are best executed on what computational element. This must include aspects of task size: which computational element is best is likely to depend on both task type and task size.
- As noted earlier in the paper, new query compilers will be required that are based on this understanding that can combine this understanding with the (dynamically varying) available resources.

This may seem like a momentous task, but if we assume we are in a situation where all elements have equal access to system memory, it may be possible to build some key operations like data transformations in FPGAs and derive an early benefit.

In addition to shared-memory-based acceleration, there are additional opportunities for near-network and near-(stored-)data acceleration for which the FPGAs are a good match. So in building architectures for in-memory databases, it is also important to keep this in mind.

7.2 Accelerator design

As new interfaces and other new hardware provide new features that break the accelerator-based systems balance, more opportunities arise for FPGAs to accelerate database systems.

As noted in this paper, for several key operators FPGA-based implementation exist that can operate at the speed of main memory. However, less work has been done on designs that leverage both the CPU and FPGA or even a combination of CPUs, FPGAs, and GPUs.

Another opportunity brought to FPGAs is the emerging machine learning workloads in databases. Database systems are extending the support of machine learning operators [76]. In addition, the databases are integrating machine learning methods for query optimization [132]. In both cases, the heavy computation required poses computational challenges to the database systems, and FPGAs can likely help. Thus, new accelerators for these emerging workloads are worth studying.

Lastly, from a system perspective, a single FPGA can only provide limited performance enhancement, while a multi-FPGA architecture should bring remarkable speedup. On the one hand, different functions can be implemented in different FPGAs to enable a wider functionality and more complicated workloads. One the other hand, multiple FPGAs can work together for a single job to enhance the performance. Using FPGAs as a cloud resource can further increase the FPGA utilization rate in both clouds and in distributed database systems [19,103]. Important challenges related to this topic include how to connect the FPGA with CPUs and other FPGAs, and how to distribute the workloads.

7.3 Comparison with GPUs

GPUs have now become a serious accelerator for database systems. This is because the GPU has thousands to ten thousands of threads, which is a good choice for throughput-optimized applications. Besides the high parallelism, being equipped with HBM allowing large capacity internal memory and fast speed to access it is another enabler. While FPGA now is also integrating HBM that makes it more powerful, the parallelism in FPGAs does not typically provide the same order of magnitude of threads in GPUs. Thus, a question we would like to ask ourselves is what kinds of database applications can work better on FPGAs than GPUs.

The first possible answer is the latency-sensitive streaming processing applications such as network processing. The request of both high throughput and low latency in the streaming processing presents challenges to GPUs. Since GPUs need to process data in a batch mode with well-formatted data for throughput gains, this formatting might introduce additional latency. However, this requirement can perfectly meet the features of the FPGA with data flow designs where format conversion is often free. The ability to do efficient format conversion might also apply to near storage processing such as Netezza [41] to relieve the pressure for CPUs and networks. Further improvement can include new features to meet the emerging workloads such as supporting format transformation between the standardized storage formats (e.g., Apache Parquet [5]) and the standardized in-memory formats (e.g, Apache Arrow [4]).

In addition, FPGAs may even beat GPUs in terms of throughput in specific domains that require special functions. For example, if one wants to build databases that operate under an encryption layer, it might need to combine encryption with other operators. FPGAs may be efficient at implementing these special functions in hardware and FPGAs often suffer only suffer negligible throughput impact by combining two function components in a pipeline. These special functions also include provisions for database reliability, special data types like geo-location data processing, or data types requiring regular expression matching.

We can also think about exploring more heterogeneity of building a database system with a CPU-GPU-FPGA combination. Different processors have different features and different strong points. CPUs are good at complex control and task scheduling, GPUs work well in a batch mode with massive data-level parallelism, and FPGAs can provide high throughput and low latency using data flow and pipeline design. Combining these processors together in one system and assigning the workloads to the processors that fit best may well deliver an advantage. However, some challenges need to be addressed to build this system including how to divide tasks into software and hardware, how to manage the computational resources in the system, and how to support data coherency between different types of processors.

7.4 Compilation for FPGAs

The FPGA programmability is always one of the biggest challenges to deploying FPGAs in database systems. To reduce the development effort and the complexity for FPGA-based database users, a hardware compiler that can map queries into FPGA circuits is necessary. Even though there is prior work [85,86,139] and proposed proof-of-concept solutions, there is still a long way to go. The following three directions may improve the hardware compiler, and thus increase FPGA programmability.

One important feature of the recent interfaces such as QPI, CXL, and OpenCAPI is that they support shared memory and data coherency. This development allows leveraging FPGAs more easily for a subset of a query or a portion of an operator in collaboration with CPUs. This enablement is significantly important for FPGA usability because not all the functionality may fit in the FPGA, as the FPGA size may depend on problem size, data types, or the function itself. Thus, a future hardware compiler might take the shared memory feature into account. A first step to start exploring this direction is to support OpenMP for hardware compilation [117,134]. OpenMP has been a popular parallel computing programming language for shared memory. It provides a few pragmas and library functions to enable multi-thread processing in CPUs, which eases multi-thread programming and is very similar to the HLS languages. Some recent enhancements, like task support, are an especially good fit for accelerators. Starting from OpenMP in the hardware compiler allows leveraging

the lessons learned from the study of OpenMP and might be the easier path to prove the benefits of supporting shared memory.

The second potential improvement comes from the support of standards for in-memory representation of data, such as Apache Arrow for columnar-oriented in-memory data. Taking Apache Arrow as an example, this standardized in-memory format allows data exchanges between different platform with zero-copy. Since it eliminates the serialization and deserialization overhead, it significantly enhances performance for data analytics and big data applications. Supporting such standards allows FPGA to be leveraged across languages and frameworks, which also reduces total amount of effort required.

Lastly, for FPGA designers that want to exercise more control over the design of the FPGA kernels for database queries or operators, the approaches that separate interface generation from computational kernel design can further increase productivity. Different FPGA kernels have different memory access patterns [37]. Some of them require gathering multiple streams (e.g., the merge tree), while some others have a random access pattern (e.g., the hash join). Thus simply using a DMA engine that is typically optimized for sequential data cannot meet all requirements, and customized data feeding logic is needed. While the ability to customized memory controllers is a strength of using FPGAs, optimizing this interface logic requires significant work. Thus, such interface generation frameworks can generate the interface automatically and designers can focus on the design and the implementation of kernels themselves. An example of such frameworks is shown in an initial work, the Fletcher framework [101].

8 Summary and conclusions

FPGAs have been recognized by the database community for their ability to accelerate CPU-intensive workloads. However, both the industry and academia have shown less interest in integrating FPGAs into database systems due to the following three reasons. First, while FPGAs can provide high data processing rates, the system performance is bounded by the limited bandwidth from conventional IO technologies. Second, FPGAs are competing with a strong alternative, GPUs, which can also provide high throughput and are much easier to program. Last, programming FPGAs typically requires a developer to have full-stack skills, from high-level algorithm design to low-level circuit implementation.

The good news is that these challenges are being addressed as can be seen through the technology trends and even the latest technologies. Interface technologies develop so fast that the interconnection between memory and accelerators can be expected to deliver main-memory scale bandwidth. In addition, FPGAs are incorporating new higher-bandwidth memory technologies such as the high-bandwidth memory, giving FPGAs a chance to combine a high degree parallel computation with high-bandwidth large-capacity local memory. Finally, emerging FPGA tool chains including HLS, new programming frameworks, and SQL-to-FPGA compilers, provide developers with better ease of use. Therefore FPGAs can become attractive again as a database accelerator.

In this paper, we explore the potential of using FPGAs to accelerate in-memory database systems. We reviewed the architecture of FPGA-accelerated database systems, discussed the challenges of integrating FPGAs into database systems, studied technology trends that address the challenges, and summarized the state-of-the-art research on database operator acceleration. We observe that FPGAs are capable of accelerating some of the database operators such as streaming operators, sort, and regular expression matching. In addition, emerging hardware compile tools further increase the usability of FPGAs in databases. We anticipate that the new technologies provide FPGAs with the opportunity to again deliver system-level speedup for database applications. However, there is still a long way to go, and future studies including new database architectures, new types of accelerators, deep performance analysis, and the development of the tool chains can push this progress a step forward.

References

1. Abdelfattah, M.S., Hagiescu, A., Singh, D.: Gzip on a chip: high performance lossless data compression on fpgas using opencl. In: Proceedings of the International Workshop on OpenCL 2013 & 2014, p. 4. ACM (2014)
2. Agarwal, K.B., Hofstee, H.P., Jamsek, D.A., Martin, A.K.: High bandwidth decompression of variable length encoded data streams. US Patent 8,824,569 (2014)
3. Albutiu, M.C., Kemper, A., Neumann, T.: Massively parallel sort-merge joins in main memory multi-core database systems. Proc. VLDB Endow. **5**(10), 1064–1075 (2012)
4. Apache: Apache Arrow. https://arrow.apache.org/. Accessed 01 Mar 2019
5. Apache: Apache Parquet. http://parquet.apache.org/. Accessed 01 Dec 2018
6. Arcas-Abella, O., Ndu, G., Sonmez, N., Ghasempour, M., Armejach, A., Navaridas, J., Song, W., Mawer, J., Cristal, A., Luján, M.: An empirical evaluation of high-level synthesis languages and tools for database acceleration. In: 2014 24th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–8. IEEE (2014)

7. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., et al.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
8. Balkesen, C., Alonso, G., Teubner, J., Özsu, M.T.: Multi-core, main-memory joins: sort vs. hash revisited. Proc. VLDB Endow. **7**(1), 85–96 (2013)
9. Balkesen, C., Teubner, J., Alonso, G., Özsu, M.T.: Main-memory hash joins on multi-core CPUs: tuning to the underlying hardware. In: IEEE 29th International Conference on Data Engineering (ICDE), 2013, pp. 362–373. IEEE (2013)
10. Balkesen, Ç., Teubner, J., Alonso, G., Özsu, M.T.: Main-memory hash joins on modern processor architectures. IEEE Trans. Knowl. Data Eng. **27**(7), 1754–1766 (2015)
11. Bartík, M., Ubik, S., Kubalik, P.: LZ4 compression algorithm on FPGA. In: IEEE International Conference on Electronics, Circuits, and Systems (ICECS), 2015, pp. 179–182. IEEE (2015)
12. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, pp. 307–314. ACM (1968)
13. Benton, B.: CCIX, Gen-Z, OpenCAPI: overview and comparison. https://www.openfabrics.org/images/eventpresos/2017presentations/213_CCIXGen-Z_BBenton.pdf (2017). Accessed 3 June 2018
14. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core CPUs. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, pp. 37–48. ACM (2011)
15. Breß, S., Heimel, M., Siegmund, N., Bellatreche, L., Saake, G.: GPU-accelerated database systems: survey and open challenges. In: Transactions on Large-Scale Data-and Knowledge-Centered Systems XV, pp. 1–35. Springer (2014)
16. Cabrera, D., Martorell, X., Gaydadjiev, G., Ayguade, E., Jiménez-González, D.: OpenMP extensions for FPGA accelerators. In: 2009 International Symposium on Systems, Architectures, Modeling, and Simulation, pp. 17–24. IEEE (2009)
17. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J.H., Brown, S., Czajkowski, T.: LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 33–36. ACM (2011)
18. Casper, J., Olukotun, K.: Hardware acceleration of database operations. In: Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 151–160. ACM (2014)
19. Caulfield, A.M., Chung, E.S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.Y., et al.: A cloud-scale acceleration architecture. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture, p. 7. IEEE Press (2016)
20. Chen, R., Prasanna, V.K.: Accelerating equi-join on a CPU-FPGA heterogeneous platform. In: 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE, pp. 212–219. IEEE (2016)
21. Chung, E.S., Davis, J.D., Lee, J.: Linqits: big data on little clients. In: ACM SIGARCH Computer Architecture News, vol. 41, pp. 261–272. ACM (2013)
22. Collet, Y., et al.: Lz4: extremely fast compression algorithm. https://code.google.com (2013). Accessed 3 June 2018
23. Cong, J., Fang, Z., Lo, M., Wang, H., Xu, J., Zhang, S.: Understanding performance differences of FPGAs and GPUs. In: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 93–96. IEEE (2018)
24. Cong, J., Huang, M., Pan, P., Wu, D., Zhang, P.: Software infrastructure for enabling FPGA-based accelerations in data centers. In: Proceedings of the 2016 International Symposium on Low Power Electronics and Design, pp. 154–155. ACM (2016)
25. Cong, J., Huang, M., Wu, D., Yu, C.H.: heterogeneous datacenters: options and opportunities. In: Proceedings of the 53rd Annual Design Automation Conference, p. 16. ACM (2016)
26. Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., Zhang, Z.: High-level synthesis for FPGAs: from prototyping to deployment. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **30**(4), 473–491 (2011)
27. Crockett, L.H., Elliot, R.A., Enderwitz, M.A., Stewart, R.W.: The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Strathclyde Academic Media, Glasgow (2014)
28. Czajkowski, T.S., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P., Singh, D.P.: From OpenCL to high-performance hardware on FPGAs. In: 22nd International Conference on Field Programmable Logic and Applications (FPL), pp. 531–534. IEEE (2012)
29. Dan Bouvier Jim Gibney, A.B., Arora, S.: Delivering a New Level of Visual Performance in an SoC. https://www.slideshare.net/AMD/delivering-a-new-level-of-visual-performance-in-an-soc-amd-raven-rdige-apu (2018). Accessed 15 Oct 2018
30. David, H., Fallin, C., Gorbatov, E., Hanebutte, U.R., Mutlu, O.: Memory power management via dynamic voltage/frequency scaling. In: Proceedings of the 8th ACM international conference on Autonomic computing, pp. 31–40. ACM (2011)
31. Dennl, C., Ziener, D., Teich, J.: Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration. In: IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2013, pp. 25–28. IEEE (2013)
32. Deutsch, P.: GZIP file format specification version 4.3. Technical Report, RFC Editor (1996)
33. Duhem, F., Muller, F., Lorenzini, P.: Farm: fast reconfiguration manager for reducing reconfiguration time overhead on fpga. In: International Symposium on Applied Reconfigurable Computing, pp. 253–260. Springer (2011)
34. Fang, J., Chen, J., Al-Ars, Z., Hofstee, P., Hidders, J.: A high-bandwidth Snappy decompressor in reconfigurable logic: work-in-progress. In: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, pp. 16:1–16:2. IEEE Press (2018)
35. Fang, J., Chen, J., Lee, J., Al-Ars, Z., Hofstee, H.P.: A fine-grained parallel snappy decompressor for FPGAs using a relaxed execution model. In: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 335–335. IEEE (2019)
36. Fang, J., Lee, J., Hofstee, H.P., Hidders, J.: Analyzing in-memory hash joins: granularity matters. In: Proceedings the 8th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, pp. 18–25 (2017)
37. Fang, J., et al.: Adopting OpenCAPI for high bandwidth database accelerators. In: 3rd International Workshop on Heterogeneous High-Performance Reconfigurable Computing (2017)
38. Feist, T.: Vivado design suite. White Paper, vol. 5 (2012)
39. Fossum, G.C., Wang, T., Hofstee, H.P.: A 64GB sort at 28GB/s on a 4-GPU POWER9 node for 16-byte records with uniformly distributed 8-byte keys. In: Proc. International Workshop on OpenPOWER for HPC. Frankfurt, Germany (2018)
40. Fowers, J., Kim, J.Y., Burger, D., Hauck, S.: A scalable high-bandwidth architecture for lossless compression on FPGAs. In: IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2015, pp. 52–59. IEEE (2015)

41. Francisco, P., et al.: The Netezza data appliance architecture: a platform for high performance data warehousing and analytics. http://www.ibmbigdatahub.com/sites/default/files/document/redguide_2011.pdf (2011). Accessed 3 June 2018
42. Franklin, M., Chamberlain, R., Henrichs, M., Shands, B., White, J.: An architecture for fast processing of large unstructured data sets. In: IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings, pp. 280–287. IEEE (2004)
43. Ghodsnia, P., et al.: An in-GPU-memory column-oriented database for processing analytical workloads. In: The VLDB Ph.D. Workshop. VLDB Endowment, vol. 1 (2012)
44. Google: Snappy. https://github.com/google/snappy/. Accessed 03 June 2018
45. Gopal, V., Guilford, J.D., Yap, K.S., Gulley, S.M., Wolrich, G.M.: Systems, methods, and apparatuses for decompression using hardware and software (2017). US Patent 9,614,544
46. Gopal, V., Gulley, S.M., Guilford, J.D.: Technologies for efficient lz77-based data decompression (2017). US Patent App. 15/374,462
47. Greenberg, M.: LPDDR3 and LPDDR4: How Low-Power DRAM Can Be Used in High-Bandwidth Applications. https://www.jedec.org/sites/default/files/M_Greenberg_Mobile%20Forum_May_%202013_Final.pdf (2013). Accessed 17 Oct 2017
48. Gupta, P.: Accelerating datacenter workloads. In: 26th International Conference on Field Programmable Logic and Applications (FPL) (2016)
49. Halstead, R.J., Absalyamov, I., Najjar, W.A., Tsotras, V.J.: FPGA-based Multithreading for In-Memory Hash Joins. In: CIDR (2015)
50. Halstead, R.J., Sukhwani, B., Min, H., Thoennes, M., Dube, P., Asaad, S., Iyer, B.: Accelerating join operation for relational databases with FPGAs. In: 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines, pp. 17–20. IEEE (2013)
51. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. ACM Trans. Database Syst. **34**(4), 21 (2009)
52. He, B., Yu, J.X.: High-throughput transaction executions on graphics processors. Proc. VLDB Endow. **4**(5), 314–325 (2011)
53. Heimel, M., Saecker, M., Pirk, H., Manegold, S., Markl, V.: Hardware-oblivious parallelism for in-memory column-stores. Proc. VLDB Endow. **6**(9), 709–720 (2013)
54. Huebner, M., Ullmann, M., Weissel, F., Becker, J.: Real-time configuration code decompression for dynamic FPGA self-reconfiguration. In: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, p. 138. IEEE (2004)
55. IBM: IBM power advanced compute (AC) AC922 server. https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=POD03143USEN&. Accessed 03 Sept 2018
56. Intel, F.: SDK for OpenCL. Programming guide. UG-OCL002 **31** (2016)
57. István, Z.: The glass half full: using programmable hardware accelerators in analytics. IEEE Data Eng. Bull. **42**(1), 49–60 (2019). http://sites.computer.org/debull/A19mar/p49.pdf
58. Jang, H., Kim, C., Lee, J.W.: Practical speculative parallelization of variable-length decompression algorithms. In: ACM SIGPLAN Notices, vol. 48, pp. 55–64. ACM (2013)
59. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev. **49**(4.5), 589–604 (2005)
60. Kara, K., Alonso, G.: Fast and robust hashing for database operators. In: 26th International Conference on Field Programmable Logic and Applications (FPL), 2016, pp. 1–4. IEEE (2016)
61. Kara, K., Giceva, J., Alonso, G.: FPGA-based data partitioning. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 433–445. ACM (2017)
62. Katz, P.W.: String searcher, and compressor using same. US Patent 5,051,745 (1991)
63. Kickfire: Kickfire. http://www.kickfire.com. Accessed 3 June 2018
64. Kim, C., Kaldewey, T., Lee, V.W., Sedlar, E., Nguyen, A.D., Satish, N., Chhugani, J., Di Blas, A., Dubey, P.: Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. Proc. VLDB Endow. **2**(2), 1378–1389 (2009)
65. Kim, J., Kim, Y.: HBM: Memory Solution for Bandwidth-Hungry Processors. https://doc.xdevs.com/doc/Memory/HBM/Hynix/HC26.11.310-HBM-Bandwidth-Kim-Hynix-Hot%20Chips%20HBM%202014%20v7.pdf (2014). Accessed 29 Aug 2018
66. Kinetica: Kinetica. http://www.kinetica.com/. Accessed 3 June 2018
67. Kocberber, O., Grot, B., Picorel, J., Falsafi, B., Lim, K., Ranganathan, P.: Meet the Walkers. PROC of the 46th MICRO pp. 1–12 (2013)
68. Koch, D., Beckhoff, C., Teich, J.: Hardware decompression techniques for FPGA-based embedded systems. ACM Trans. Reconf. Technol. Syst. **2**(2), 9 (2009)
69. Koch, D., Torresen, J.: FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 45–54. ACM (2011)
70. Kruger, F.: CPU bandwidth: the worrisome 2020 trend. https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/ (March 23, 2016). Accessed 03 May 2017
71. Lang, H., Leis, V., Albutiu, M.C., Neumann, T., Kemper, A.: Massively parallel NUMA-aware hash joins. In: In Memory Data Management and Analysis, pp. 3–14. Springer (2015)
72. Lee, J., Kim, H., Yoo, S., Choi, K., Hofstee, H.P., Nam, G.J., Nutter, M.R., Jamsek, D.: ExtraV: boosting graph processing near storage with a coherent accelerator. Proc. VLDB Endow. **10**(12), 1706–1717 (2017)
73. Lei, J., Chen, Y., Li, Y., Cong, J.: A high-throughput architecture for lossless decompression on FPGA designed using HLS. In: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 277–277. ACM (2016)
74. Lin, M.B., Chang, Y.Y.: A New Architecture of a Two-Stage Lossless Data Compression and Decompression Algorithm. IEEE Trans. VLSI Syst. **17**(9), 1297–1303 (2009)
75. Liu, H.Y., Carloni, L.P.: On learning-based methods for design-space exploration with high-level synthesis. In: Proceedings of the 50th Annual Design Automation Conference, p. 50. ACM (2013)
76. Mahajan, D., Kim, J.K., Sacks, J., Ardalan, A., Kumar, A., Esmaeilzadeh, H.: In-RDBMS hardware acceleration of advanced analytics. Proc. VLDB Endow. **11**(11), 1317–1331 (2018)
77. Mahony, A.O., Tringale, A., Duquette, J.J., O'carroll, P.: Reduction of execution stalls of LZ4 decompression via parallelization. US Patent 9,973,210 (2018)
78. Marcelino, R., Neto, H.C., Cardoso, J.M.: Unbalanced FIFO sorting for FPGA-based systems. In: 16th IEEE International Conference on Electronics, Circuits, and Systems, 2009. ICECS 2009, pp. 431–434. IEEE (2009)
79. Mashimo, S., Van Chu, T., Kise, K.: High-performance hardware merge sorter. In: IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, pp. 1–8. IEEE (2017)
80. Mellanox Technologies: Mellanox Innova™-2 flex open programmable SmartNIC. http://www.mellanox.com/page/products_dyn?product_family=276&mtag=programmable_adapter_cards_innova2flex. Accessed 28 Apr 2019
81. Mostak, T.: An overview of MapD (massively parallel database). White Paper, Massachusetts Institute of Technology (2013)

82. Mueller, R., Teubner, J.: FPGA: what's in it for a database? In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 999–1004. ACM (2009)
83. Mueller, R., Teubner, J.: FPGAs: a new point in the database design space. In: Proceedings of the 13th International Conference on Extending Database Technology, pp. 721–723. ACM (2010)
84. Mueller, R., Teubner, J., Alonso, G.: Data processing on FPGAs. Proc. VLDB Endow. **2**(1), 910–921 (2009)
85. Mueller, R., Teubner, J., Alonso, G.: Streams on wires: a query compiler for FPGAs. Proc. VLDB Endow. **2**(1), 229–240 (2009)
86. Mueller, R., Teubner, J., Alonso, G.: Glacier: a query-to-hardware compiler. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 1159–1162. ACM (2010)
87. Mueller, R., Teubner, J., Alonso, G.: Sorting networks on FPGAs. VLDB J. **21**(1), 1–23 (2012)
88. Mulder, Y.: Feeding high-bandwidth streaming-based FPGA accelerators. Master's Thesis, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands (2018)
89. Nallatech: OpenCAPI enabled FPGAs—the perfect bridge to a data centric world. https://openpowerfoundation.org/wp-content/uploads/2018/10/Allan-Cantle.Nallatech-Presentation-2018-OPF-Summit_Amsterdam-presentation.pdf (2018). Accessed 25 Oct 2018
90. Nane, R., Sima, V.M., Olivier, B., Meeuws, R., Yankova, Y., Bertels, K.: DWARV 2.0: a CoSy-based C-to-VHDL hardware compiler. In: 22nd International Conference on Field Programmable Logic and Applications (FPL), pp. 619–622. IEEE (2012)
91. Nane, R., Sima, V.M., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y.T., Hsiao, H., Brown, S., Ferrandi, F., et al.: A survey and evaluation of FPGA high-level synthesis tools. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **35**(10), 1591–1604 (2016)
92. Napatech: Napatech SmartNIC solution for hardware offload. https://www.napatech.com/support/resources/solution-descriptions/napatech-smartnic-solution-for-hardware-offload/. Accessed 28 Apr 2019
93. Nikhil, R.: Bluespec System Verilog: efficient, correct RTL from high level specifications. In: Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-design, 2004. MEMOCODE'04., pp. 69–70. IEEE (2004)
94. Nyberg, C., Shah, M., Govindaraju, N.: Sort benchmark home page. http://sortbenchmark.org/. Accessed 03 Aug 2018
95. OpenPOWER: SNAP framework hardware and software. https://github.com/open-power/snap/. Accessed 03 June 2018
96. Ouyang, J., Qi, W., Yong, W., Tu, Y., Wang, J., Jia, B.: SDA: software-defined accelerator for general-purpose distributed big data analysis system. In: Hot Chips: A Symposium on High Performance Chips, Hotchips (2016)
97. Owaida, M., Sidler, D., Kara, K., Alonso, G.: Centaur: a framework for hybrid CPU-FPGA databases. In: IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, pp. 211–218. IEEE (2017)
98. Papaphilippou, P., Luk, W.: Accelerating database systems using FPGAs: a survey. In: 2018 28th International Conference on Field Programmable Logic and Applications (FPL), pp. 125–1255. IEEE (2018)
99. PCI-SIG: PCI-SIG ® announces upcoming PCI express ® 6.0 specification to reach 64 GT/s. https://www.businesswire.com/news/home/20190618005945/en/PCI-SIG%C2%AE-Announces-Upcoming-PCI-Express%C2%AE-6.0-Specification. Accessed 01 July 2019
100. PCI-SIG: Specifications PCI-SIG. https://pcisig.com/specifications. Accessed 01 July 2019
101. Peltenburg, J., van Straten, J., Brobbel, M., Hofstee, H.P., Al-Ars, Z.: Supporting columnar in-memory formats on FPGA: the hardware design of fletcher for Apache Arrow. In: International Symposium on Applied Reconfigurable Computing, pp. 32–47. Springer (2019)
102. Pilato, C., Ferrandi, F.: Bambu: a modular framework for the high level synthesis of memory-intensive applications. In: 2013 23rd International Conference on Field programmable Logic and Applications, pp. 1–4. IEEE (2013)
103. Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., et al.: A reconfigurable fabric for accelerating large-scale datacenter services. ACM SIGARCH Comput. Archit. News **42**(3), 13–24 (2014)
104. Qiao, W., Du, J., Fang, Z., Wang, L., Lo, M., Chang, M.C.F., Cong, J.: High-throughput lossless compression on tightly coupled CPU-FPGA platforms. In: FPGA, p. 291 (2018)
105. Qiao, Y.: An FPGA-based snappy decompressor-filter. Master's Thesis, Delft University of Technology (2018)
106. Scofield, T.C., Delmerico, J.A., Chaudhary, V., Valente, G.: XtremeData dbX: an FPGA-based data warehouse appliance. Comput. Sci. Eng. **12**(4), 66–73 (2010)
107. Sadoghi, M., Javed, R., Tarafdar, N., Singh, H., Palaniappan, R., Jacobsen, H.A.: Multi-query stream processing on FPGAs. In: 2012 IEEE 28th International Conference on Data Engineering, pp. 1229–1232. IEEE (2012)
108. Saitoh, M., Elsayed, E.A., Van Chu, T., Mashimo, S., Kise, K.: A high-performance and cost-effective hardware merge sorter without feedback datapath. In: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 197–204. IEEE (2018)
109. Salami, B., Arcas-Abella, O., Sonmez, N.: HATCH: hash table caching in hardware for efficient relational join on FPGA. In: IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2015, pp. 163–163. IEEE (2015)
110. Salami, B., Arcas-Abella, O., Sonmez, N., Unsal, O., Kestelman, A.C.: Accelerating hash-based query processing operations on FPGAs by a hash table caching technique. In: Latin American High Performance Computing Conference, pp. 131–145. Springer (2016)
111. Salomon, D.: Data Compression: The Complete Reference. Springer, Berlin (2004)
112. Sharma, D.D.: Compute express link. https://docs.wixstatic.com/ugd/0c1418_d9878707bbb7427786b70c3c91d5fbd1.pdf (2019). Accessed 15 Apr 2019
113. Sidler, D., István, Z., Owaida, M., Alonso, G.: Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 403–415. ACM (2017)
114. Sidler, D., István, Z., Owaida, M., Kara, K., Alonso, G.: doppioDB: a hardware accelerated database. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1659–1662. ACM (2017)
115. Singh, D.P., Czajkowski, T.S., Ling, A.: Harnessing the power of FPGAs using altera's OpenCL compiler. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 5–6. ACM (2013)
116. Sitaridi, E., Mueller, R., Kaldewey, T., Lohman, G., Ross, K.A.: Massively-parallel lossless data decompression. In: 2016 45th International Conference on Parallel Processing (ICPP), pp. 242–247. IEEE (2016)
117. Sommer, L., Korinth, J., Koch, A.: OpenMP device offloading to FPGA accelerators. In: 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 201–205. IEEE (2017)
118. Song, W., Koch, D., Luján, M., Garside, J.: Parallel hardware merge sorter. In: IEEE 24th Annual International Symposium

on Field-Programmable Custom Computing Machines (FCCM), 2016, pp. 95–102. IEEE (2016)
119. Srivastava, A., Chen, R., Prasanna, V.K., Chelmis, C.: A hybrid design for high performance large-scale sorting on FPGA. In: International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2015, pp. 1–6. IEEE (2015)
120. Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 123–134. IEEE Computer Society (2005)
121. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(1–3), 66–73 (2010)
122. Storer, J.A., Szymanski, T.G.: Data compression via textual substitution. J. ACM **29**(4), 928–951 (1982)
123. Stuecheli, J.: A new standard for high performance memory, acceleration and networks. http://opencapi.org/2017/04/opencapi-new-standard-high-performance-memory-acceleration-networks/. Accessed 3 June 2018
124. Sukhwani, B., Min, H., Thoennes, M., Dube, P., Brezzo, B., Asaad, S., Dillenberger, D.E.: Database analytics: a reconfigurable-computing approach. IEEE Micro **34**(1), 19–29 (2014)
125. Sukhwani, B., Min, H., Thoennes, M., Dube, P., Iyer, B., Brezzo, B., Dillenberger, D., Asaad, S.: Database analytics acceleration using FPGAs. In: Proceedings of the 21st international conference on Parallel architectures and compilation techniques, pp. 411–420. ACM (2012)
126. Sukhwani, B., Thoennes, M., Min, H., Dube, P., Brezzo, B., Asaad, S., Dillenberger, D.: Large payload streaming database sort and projection on FPGAs. In: 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2013, pp. 25–32. IEEE (2013)
127. Teubner, J., Woods, L.: Data processing on FPGAs. Synth. Lect. Data Manag. **5**(2), 1–118 (2013)
128. Teubner, J., Woods, L., Nie, C.: XLynx-an FPGA-based XML filter for hybrid XQuery processing. ACM Trans. Database Syst. **38**(4), 23 (2013)
129. Thompto, B.: POWER9: processor for the cognitive era. In: Hot Chips 28 Symposium (HCS), 2016 IEEE, pp. 1–19. IEEE (2016)
130. Ueda, T., Ito, M., Ohara, M.: A dynamically reconfigurable equi-joiner on FPGA. IBM Technical Report RT0969 (2015)
131. Van Lunteren, J., Rohrer, J., Atasu, K., Hagleitner, C.: Regular expression acceleration at multiple tens of Gb/s. In: 1st Workshop on Accelerators for High-Performance Architectures in Conjunction with ICS (2009)
132. Wang, W., Zhang, M., Chen, G., Jagadish, H., Ooi, B.C., Tan, K.L.: Database meets deep learning: challenges and opportunities. ACM SIGMOD Rec. **45**(2), 17–22 (2016)
133. Wang, Z., He, B., Zhang, W.: A study of data partitioning on OpenCL-based FPGAs. In: 2015 25th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–8. IEEE (2015)
134. Watanabe, Y., Lee, J., Boku, T., Sato, M.: Trade-off of offloading to FPGA in OpenMP task-based programming. In: International Workshop on OpenMP, pp. 96–110. Springer (2018)
135. Welch, T.A.: A technique for high-performance data compression. Computer **17**(6), 8–19 (1984). https://doi.org/10.1109/MC.1984.1659158
136. Wenzel, L., Schmid, R., Martin, B., Plauth, M., Eberhardt, F., Polze, A.: Getting started with CAPI SNAP: hardware development for software engineers. In: European Conference on Parallel Processing, pp. 187–198. Springer (2018)
137. Wirbel, L.: Xilinx SDAccel: a unified development environment for tomorrow's data center. Technical Report, The Linley Group Inc. (2014)
138. Wissolik, M., Zacher, D., Torza, A., Da, B.: Virtex UltraScale+ HBM FPGA: a revolutionary increase in memory performance. Xilinx Whitepaper (2017)
139. Woods, L., István, Z., Alonso, G.: Ibex: an intelligent storage engine with support for advanced SQL offloading. Proc. VLDB Endow. **7**(11), 963–974 (2014)
140. Wu, L., Barker, R.J., Kim, M.A., Ross, K.A.: Hardware-accelerated range partitioning. Columbia University Computer Science Technical Reports (2012)
141. Xilinx: GZIP/ZLIB/Deflate data compression core. http://www.cast-inc.com/ip-cores/data/zipaccel-d/cast-zipaccel-d-x.pdf (2016). Accessed 03 Aug 2018
142. Xilinx: UltraScale FPGA product tables and product selection guide. https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf (2018). Accessed 03 Sept 2018
143. Yuan, Y., Lee, R., Zhang, X.: The Yin and Yang of processing data warehousing queries on GPU devices. Proc. VLDB Endow. **6**(10), 817–828 (2013)
144. Zacharopoulos, G., Barbon, A., Ansaloni, G., Pozzi, L.: Machine learning approach for loop unrolling factor prediction in high level synthesis. In: 2018 International Conference on High Performance Computing & Simulation (HPCS), pp. 91–97. IEEE (2018)
145. Zeng, X.: FPGA-based high throughput merge sorter. Master's Thesis, Delft University of Technology (2018)
146. Zhang, C., Chen, R., Prasanna, V.: High throughput large scale sorting on a CPU-FPGA heterogeneous platform. In: Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International, pp. 148–155. IEEE (2016)
147. Zhang, S., He, J., He, B., Lu, M.: Omnidb: towards portable and efficient query processing on parallel cpu/gpu architectures. Proc. VLDB Endow. **6**(12), 1374–1377 (2013)
148. Zhou, X., Ito, Y., Nakano, K.: An efficient implementation of LZW decompression in the FPGA. In: IEEE International Parallel and Distributed Processing Symposium Workshops, 2016, pp. 599–607. IEEE (2016)
149. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)
150. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Trans. Inf. Theory **24**(5), 530–536 (1978)