



In-Depth Analysis of OLAP Query Performance on Heterogeneous Hardware

David Broneske¹ · Anna Drewes² · Bala Gurumurthy¹ · Imad Hajjar² · Thilo Pionteck² · Gunter Saake¹

Received: 8 June 2020 / Accepted: 28 May 2021 / Published online: 26 July 2021
© The Author(s) 2021

Abstract

Classical database systems are now facing the challenge of processing high-volume data feeds at unprecedented rates as efficiently as possible while also minimizing power consumption. Since CPU-only machines hit their limits, co-processors like GPUs and FPGAs are investigated by database system designers for their distinct capabilities. As a result, database systems over heterogeneous processing architectures are on the rise. In order to better understand their potentials and limitations, in-depth performance analyses are vital. This paper provides interesting performance data by benchmarking a portable operator set for column-based systems on CPU, GPU, and FPGA – all available processing devices within the same system. We consider TPC-H query Q6 and additionally a hash join to profile the execution across the systems. We show that system memory access and/or buffer management remains the main bottleneck for device integration, and that architecture-specific execution engines and operators offer significantly higher performance.

Keywords Heterogeneous database systems · CPU · GPU · FPGA · Overlay architecture

1 Introduction

With the advent of Big Data, the size of generated and transmitted data has grown exponentially. To exploit hidden information in these data, business analysis, graph and stream processing arose as some of the main application ar-

reas for database systems. These applications require a huge amount of processing power from CPUs [33].

Such trends in data-driven applications and modern hardware devices shape the efforts made by the database community and divide it into two main groups: The first focuses on a scaling-out strategy, i.e., on distributing storage and/or processing over clusters of computers that still use CPUs as the main processing element [40, 41]. The second group focuses on scaling-up, i.e., utilizing the advancements made in hardware architecture, specifically GPUs and FPGAs, to build heterogeneous machines with the goal to process data more efficiently than CPU-only machines. The motivation for this strategy is that GPUs and FPGAs have inherently parallel execution models. While GPUs are known for their SIMT model, FPGAs offer spatial parallelism, pipelined execution, low power consumption and inherently enable stream processing. Overall, due to increased demands for efficient processing of large amounts of data, the rise of heterogeneous database systems seems inevitable.

Several projects are published where GPUs are integrated as co-processors in addition to CPUs into one system [10, 16, 30]. Similarly there are other attempts to integrate FPGAs and CPUs [27, 29, 32]. Since most systems focus on these two individual pairings, few research groups investigated systems consisting of all three.

Authors are in alphabetical order due to equal contributions.

David Broneske
David.Broneske@ovgu.de

Anna Drewes
Anna.Drewes@ovgu.de

✉ Bala Gurumurthy
Bala.Gurumurthy@ovgu.de

Imad Hajjar
Imad.Hajjar@ovgu.de

Thilo Pionteck
Thilo.Pionteck@ovgu.de

Gunter Saake
Gunter.Saake@ovgu.de

¹ Databases and Software Engineering Group, Otto von Guericke University, Magdeburg, Germany

² Institute for Information Technology and Communications, Otto von Guericke University, Magdeburg, Germany

In this paper, we focus on analyzing the performance of all three devices (CPU-GPU-FPGA). In order to draw fair conclusions, we run the same workload from TPC-H [42] on all of them using common primitive-based operator implementations. Our performance analysis include an end-to-end analysis to cover all overheads incurred by the underlying system. We not only compare performance figures, but also provide insights into portability and potential barriers and bottlenecks. These insights provide a better understanding of the scale-up strategy.

This paper is structured as follows: In Sect. 2, we discuss related work. In Sect. 3, we provide background information on device characteristics and used *primitives*. Then in Sect. 4, we describe how we achieve operator portability across different hardware architectures, with special focus on the FPGA architecture. In Sect. 5, we explain our experimental setup and present the experimental results in Sect. 6. Finally, in Sect. 7, we conclude the paper by pointing out limitations and future work.

2 Related Work

In this section, we discuss a selective list of Online Analytical Processing (OLAP) systems, which we divide into two groups: CPU-GPU and CPU-FPGA hybrids.

CPU-GPU Hybrid: GPUQP [16] is one of the first attempts to introduce GPUs as query co-processors in order to exploit their massive data parallelism. He et al. [22] introduced an in-memory relational query co-processing system called GDB. This system uses an optimized set of primitives such as split and sort for query processing. Similarly, CoGaDB [10] is a main memory DBMS targeting a heterogeneous CPU-GPU system. It uses two sets of operators, one for the CPU, and the other for GPU. Further, it uses a device-aware optimizer – HyPE [9]. HyPE takes advantage of CPUs and GPUs and extends the physical optimization process in a cost-learning-based engine. Alternative to CoGaDB, a hardware-oblivious processing engine – Ocelot [23] is developed as an extension of MonetDB. In contrast to previously mentioned hardware-aware systems, its operators were developed using OpenCL [25]. Similarly, OmniDB [48] exploits hardware characteristics in OpenCL based CPU-GPU DMBSs.

Other than these, few commercial implementation comprises of (but not limited to) Kinetica [26], OmniSci (a.k.a. MapD) [28], SQreamDB [38] and BlazingDB [6].

CPU-FPGA Hybrid: Researchers also investigated heterogeneous DBMSs combining CPU and FPGA. One example of such systems is doppioDB [36], an extension of MonetDB that uses Hardware User Defined Functions (HUDFs) for query acceleration on FPGAs. The system was built on Intel’s Xeon+FPGA platform showing advantages and

limitations of HUDFs. It is built on specialized hardware, while our FPGA system is designed for off-the-shelf FPGA accelerator cards.

Centaur [29] is a framework for hybrid CPU-FPGA databases that allows the dynamic allocation of pipelined FPGA operators as well as hybrid execution on CPU and FPGA. Centaur utilizes UDFs and FThreads to allow seamless integration of FPGA operators in the database engine. While our system is designed with Dynamic Partial Reconfiguration (DPR) and portability as main goals, Centaur builds on the Intel Xeon+FPGA technology, and they aim to distribute and queue commands to a set of static workers.

In [39], the authors use a PCIe-attached FPGA to offload predicate evaluation for heavy analytical query processing, while the CPU handles transaction processing. They stream compressed row-based data through a configurable accelerator over PCIe for executing different queries without reconfiguration. The reported results indicate the system offers an order of magnitude speedup in the offloading computations compared to a software baseline.

AxleDB [34] is an FPGA-based database query processor with SSDs directly attached to the processing units. Static accelerators are arranged in a streaming-based ring bus and are operated by a global state machine with host-generated instruction words. In addition to the accelerators, it implements an index structure. The authors achieve a speedup of 2x over standard DBMSs by implementing data parallel accelerators. While being based on standard FPGA hardware, AxleDB is also a static system, which limits its flexibility in both allocation of compute resources and communication compared to a system with DPR and configurable data-flow routing.

The ReProVide project [4] presents a hybrid system consisting of CPU-FPGA, where the FPGA is connected over the network. This system uses the dynamic partial reconfiguration (DPR) capability of modern FPGAs to exchange large accelerators to do pre-processing of data to be sent to the host. The authors also propose a model for operator placement and buffer allocation.

The DPR-based FPGA system presented in [49] cannot just stream rows through processing modules, but can also do sort and join operations using heterogeneous reconfigurable partitions. The authors focus only on the FPGA: The host assigns partial queries to the FPGA and manages the accelerator. In addition to the special function partition for sort and join, the system consists of 4 banks of 16 small daisy-chained reconfigurable modules with data flow being routed only between banks. This work is the closest in design to our system, but we designed a fully column-based system and focus on more flexible routing of data-flow between compute units and a more uniform grid of compute tiles.

```

void red_add(int *in, int *out, size_t size) {

    int temp_add = 0;

    #pragma omp shared(temp_add)
    #pragma omp parallel for reduction(+:temp_add)
    for(size_t i=0; i<size ; i++) {
        temp_add += in[i];
    }
    *out = temp_add;
}

```

Fig. 1 OpenMP code of the reduce-add primitive

Overall, in this work, we evaluate query processing over a heterogeneous processor system containing three different processor architectures: CPU, GPU and FPGA. For a detailed discussion on integrating FPGAs into DBMSs, we refer to our previous work [3]. There we present an extensive literature review, identify the challenges and propose approaches to tackle them. To the best of our knowledge, there is no existing work that compares query processing performance on CPU, GPU and FPGA processing from a single holistic setup.

3 Background

Current processor architectures are branching out into two broad spectra: (1) *homogeneous cores*, where multiple CPU cores are tightly coupled for processing. (2) *heterogeneous cores*, where tailor-made compute architectures are used for specific functions. DBMSs running on top of these architectures have to be aware of the features of the underlying hardware to have improved performance [10, 36].

In order to run database operations on different hardware architectures, an abstract representation of the operations is necessary to reduce re-implementation work and increase re-usability [8].

3.1 Primitives

We use primitives as basic building blocks of DBMS operations. Primitives in this context are atomic functions that are combined together to form an SQL operator. They can be parallelized and arranged together to execute partial or full SQL queries [7, 22]. We surveyed the primitives and detailed their alternative implementations in our paper [19]. Below are the primitives considered in this work:

- Map: A one-to-one mapping from input to output.
- Scan: Prefix sum on the given input.
- Materialize: Synthesizing selected column values.
- Aggregate: Grouped or single reduction on an array of inputs.
- Scatter & Gather: Data shuffling operations.

```

__kernel void red_add
(__global unsigned int* in,
 __global unsigned int* out,
 unsigned int INTENT) {
    int tmp_sum = 0;
    size_t pos = INTENT * get_global_id(0);
    for (unsigned int i=0; i < INTENT; i++){
        tmp_sum += in[pos + i];
    }
    atomic_add(&out[0], tmp_sum);
}

```

Fig. 2 OpenCL code of the reduce-add primitive

- Hash build: Creating a hash table.
- Hash probe: Hash table look up.
- Sort: Sorting a given input based on some criteria.

Using these primitives, we build a Directed Acyclic Graph (DAG) for a given SQL query.

3.2 Devices and Characteristics

CPU: the key characteristics of CPUs are instruction pipelining, hierarchical caches and SIMD [24]. With pipelining, CPUs provide instruction-level parallelism but the overall execution in a system is delayed due to data and control-flow dependencies across instructions. This delay is hidden using out-of-order execution and branch prediction strategies. Another improvement in current CPU generations are the SIMD units i.e., vectorized execution of a single instruction across multiple values [12, 37]. These units can execute an instruction on a vector of data instead of just on a single value.

GPU: A GPU contains thousands of physical cores to support embarrassingly parallel operations. GPUs use a higher degree of Thread Level Parallelism (TLP) compared to CPUs in order to hide memory access latency. A GPU is capable of executing a large number of simple instructions in parallel, but does not have the capability of branch prediction. Therefore, control-flow-heavy operations cannot exploit the full parallelism in a GPU due to the serialization of diverging branches [21]. Finally, the biggest challenge in a GPU is the transfer time spent to move data from the CPU's main memory to the GPU's. This transfer overhead can over-shadow the performance benefits of GPU-acceleration [11].

FPGA: FPGAs are capable of realizing arbitrary logic circuits by mapping computation, i.e., mathematical (pure, stateless) functions, to hardware lookup tables (LUT). Additionally, modern FPGAs contain specialized hardware for more efficient processing, including large blocks of SRAM (BRAM), carry chains connecting neighboring LUTs and multiply-accumulate units (DSP). Finally, *dynamic partial reconfiguration* (DPR) allows the designer to exchange parts of the circuit at runtime. Traditionally, FPGAs are

```

typedef ap_axis<32, 2, 1, 1> axis_t;
typedef hls::stream<axis_t> stream_t;
void red_add(stream_t& in, stream_t& out) {
#pragma HLS INTERFACE axis register both port=in
#pragma HLS INTERFACE axis register both port=out
#pragma HLS INTERFACE ap_ctrl_none port=return
    axis_t in_val, out_val, dummy_val; out_val.data = 0;
    do {
#pragma HLS PIPELINE
        in.read(in_val);
        out_val.data += in_val.data;
    } while (!in_val.last);
    dummy_val.data = 0;
    dummy_val.last = 0;
    dummy_val.keep = 0xf;
    out.write(dummy_val);
    out_val.last = 1;
    out_val.keep = 0xf;
    out.write(out_val);
}

```

Fig. 3 HLS code of the reduce-add primitive

programmed using hardware description languages (HDL) that allow for low-level design specification on the register transfer level (RTL). As compiler technology improved, the concept of high-level synthesis (HLS) was developed. Here, the kernel is programmed in a higher-level language such as C++, and is then transformed automatically into a circuit. HLS provides a more software-like development environment, but can sometimes be less flexible and also incur a slight resource overhead compared to an RTL design [44].

4 Execution Models

In this section, we discuss the execution model of the three devices (CPU, GPU and FPGA), i.e. how to execute a primitive on these different devices.

4.1 CPU

For executing primitives on a CPU, we follow the operator-at-a-time execution model proposed by Boncz et al. [7]. This model executes one primitive at a time for all input data, in a tight loop. When one primitive in the query plan finishes execution, the next one starts. Though vectorized

execution of these primitives provide performance benefits [50], this is not suitable for a heterogeneous environment, due to high data transfer delay.

Figure 1 shows an OpenMP implementation of the reduce-add primitive to be run on a multi-core CPU.

For data transfer in OpenMP, we simply forward the input pointer to the target function for execution.

4.2 GPU

Similar to CPUs, GPUs also follows operator-at-a-time execution. For each primitive (or kernel in the context of OpenCL), the runtime loads data as well as creating result space in the GPU based on the target kernel. The result space is estimated based on the input size, intents (data processed per thread) [31]. Once set, the kernel is executed.

A task sets the number of parallel local items and the work groups required for parallelizing the task on the GPU. These values are computed from the input size and parameters present in the task. Note that consecutive operations in GPU simply forward the data buffer without any expensive data routing.

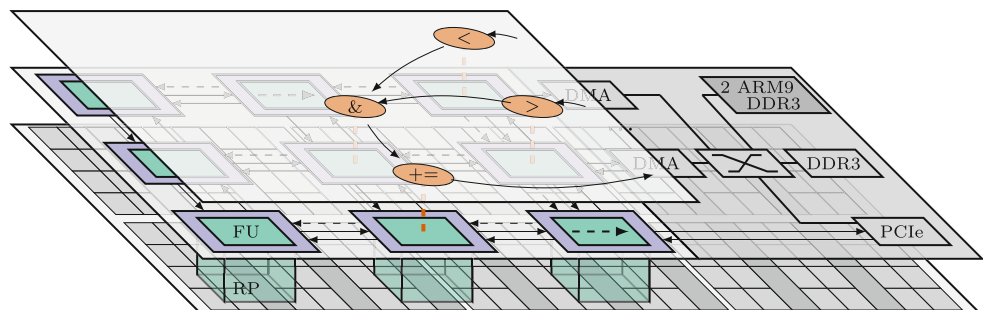
In Figure 2, we show an example OpenCL implementation for the reduce-add primitive. It is worth mentioning, that the same kernel code can be used to run on a CPU instead of a GPU. Note the kernel's additional parameter INTENT defining the number of inputs processed by a single work item.

Data transfer mechanism is different based on the target OpenCL device. For OpenCL execution on CPU, we simply forward the host pointer. In case of GPUs, we consider the case of a cold-store, where initially no data is available in GPU and data is transferred explicitly according to the incoming query. However, all intermediate results are stored in the GPU memory and only the final result is transferred back to the CPU.

4.3 FPGA

While HLS solves most of the complexities in programming a FPGA, the software integration problem still re-

Fig. 4 General structure and abstraction levels of our FPGA overlay architecture



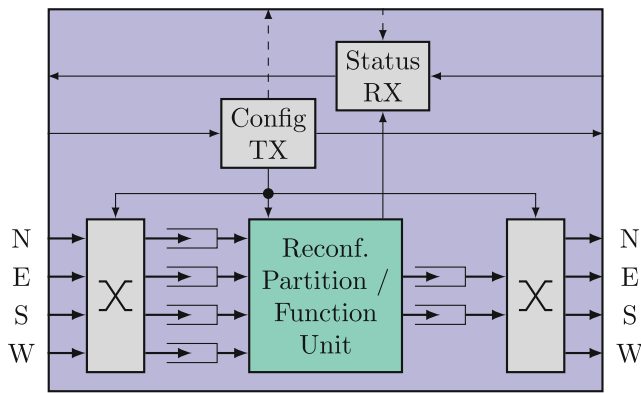


Fig. 5 A single tile of our FPGA overlay architecture

mains. To connect FPGAs to a host server, there exist products such as Xillybus over PCIe [46] which provide both hardware interfaces and OS drivers, but any user program still has to be adapted specifically to every FPGA design. Thus, single-function accelerators are common in the industry [27, 32, 43]. Another option is to implement multiple functions in the same FPGA design, which limits the amount of resources every functional unit can occupy, and therefore potentially reduces performance. We have already analyzed these shortcomings with respect to Intel OpenCL for FPGAs [15]. To combat this, modern FPGAs allow for smaller parts of the fabric to be reconfigured with different circuits during runtime, instead of just loading a complete design into the FPGA at power up. Dynamic partial reconfiguration of reconfigurable partitions (RP) allows for function units (FU) to be exchanged at runtime, and therefore greatly increases flexibility, but also forces higher complexity upon the designer. One possibility to hide this additional complexity is to specify the design in such a way that a simpler representation of it can be devised: An overlay architecture abstracts away the raw image of FPGA hardware resources into a user representation more specific to their application.

4.3.1 User Model of the Overlay Architecture

As shown in layer two of Fig. 4, our overlay architecture consists of an array of locally interconnected FUs, where each FU performs a set of streaming operations based on the primitives introduced in Sect. 3.1. These primitives are implemented using HLS, synthesized for each FU, and then integrated into the overall design transparently to the programmer via DPR. Figure 3 shows the reduction primitive for addition as an example of an HLS primitive: After defining the data types for the hardware interface using vendor libraries, we add a stream of 32-bit-integers. The logic for accessing the stream hardware is generated by the HLS tool chain.

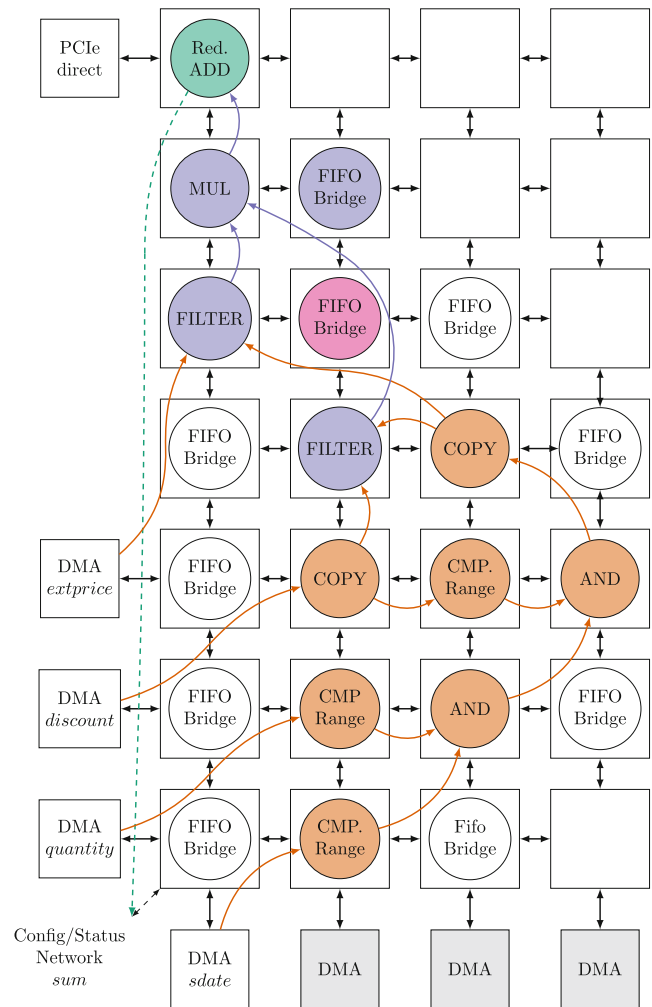


Fig. 6 Data-flow graph of TPC-H query Q6 mapped onto the FPGA overlay architecture

Query processing (partially) on FPGA using the reconfigurable overlay architecture starts with mapping the input data-flow graph to the graph describing the available FUs. While finding perfect matchings is complex, efficient approximative approaches such as simulated annealing [5] are generally available and effective. Based on the matching, the user space driver constructs the necessary configuration data, loads the required primitives via DPR and configures data-flow routing within the overlay architecture. Finally, after input data is copied to the DDR memories on the FPGA card, the user instructs the overlay architecture to process the required columns and waits for it to finish.

4.3.2 Structure and Infrastructure

We implemented a custom FPGA overlay architecture focused on hardware-pipelined execution of data-flow graphs using DPR to provide diverse functionality. As Fig. 4 shows, the FPGA is divided into a static part (on the right) and

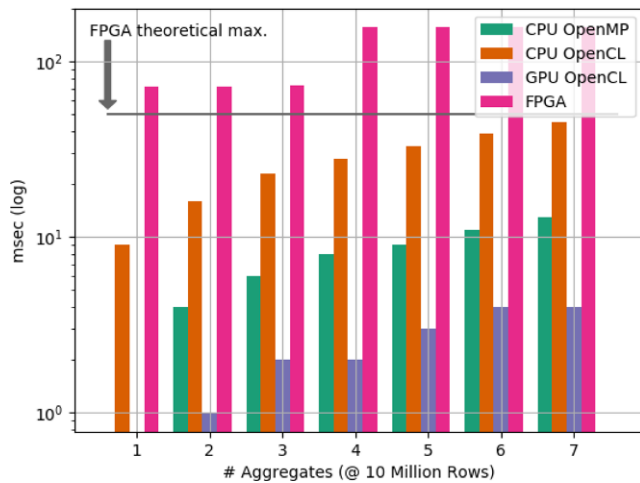


Fig. 7 Pure execution time of increasing numbers of reduction primitives at 10 million rows

a set of reconfigurable partitions. Only the actual function units (FUs) are placed inside the reconfigurable partitions (in green). The generic infrastructure elements of the static partition are in the second layer (in gray). They consist of the logic required for PCIe connectivity, a DDR3 controller and direct memory access (DMA) blocks for data access and transfer from/to the host.

The RPs are arranged in a regular grid pattern across the FPGA fabric and are grouped together with their supporting logic into a grid of tiles. The tiles are the fundamental building block of the overlay architecture and their inner structure is described in Fig. 5. Each tile has high-bandwidth data-flow connections to a 4-neighbourhood. Furthermore, there is a packet-switched configuration and status network for short message exchanges between the host program and any FU without the need for a static data-flow route. The input and output crossbars within each tile enable flexible data transport and are also set up via these messages. In addition, to support random memory accesses for example for the hash table primitives, a few tiles are also attached to the static memory bus directly, not just through DMA engines.

Since data-flow graphs are likely to not map perfectly to a 2D-grid, a special pass-through or bridge FU can be loaded, which instead of an operation just contains two FIFOs. This allows forwarding of up to two data streams through any unused tile. One exemplary use is highlighted in pink in Fig. 6, where two data streams need to cross over.

The whole design is freely scalable and flexible with respect to the targeted application domain. This is especially interesting, since the amount of general and specialized FPGA resources provided is defined one level lower, by the shape and placement of the reconfigurable partitions. This allows not only for different designs to set higher or lower resource limits, but also, the partitions do not necessarily

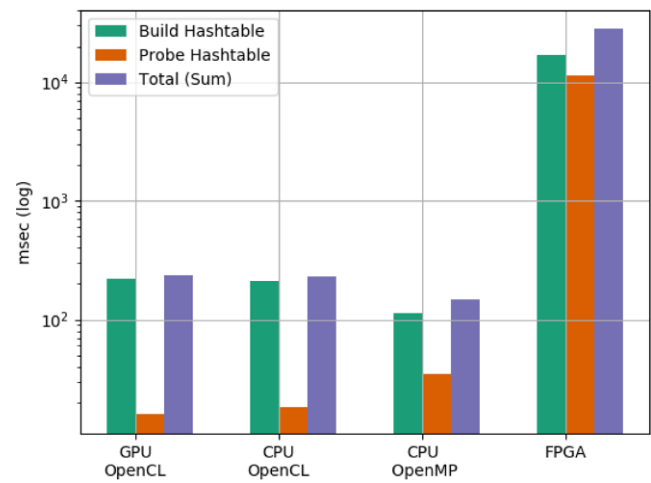


Fig. 8 Pure execution time of the different hash join primitives at 25 million rows

have to have the same size. Again, all of this is transparent to the user, who only perceives an array of functional units, each with its own set of supported operations.

In conclusion, our overlay architecture allows for fast and dynamic composition of query datapaths. Using small configurable local crossbars allows for higher flexibility than a statically wired set of RPs while allowing for many more RPs than a globally interconnected system. This flexibility in data-flow routing can also help reduce the cost of re-configuration. Also, the structure of the tiles fits perfectly to a column-based system, chosen to reduce bandwidth by accessing only the necessary columns. Finally, the structure of our overlay architecture enables both functional and data parallelism since different parts of the system are always similar.

4.3.3 Physical Design

The connections between the overlay architecture and the memory subsystem are located along the left and bottom sides in our FPGA, as shown in Fig. 6. Again, such placement is due to the physical layout of the generic infrastructure components. The DMA engines are used to scan or store columns located in the FPGA's DDR3 DRAM. In addition, there is one bidirectional connection directly to the PCIe core, which allows one column to be streamed to or from the host server directly.

5 Experimental Setup

This section covers the setup of our experiments and describes the hardware used to gather our results. Since we focus on analytical query processing instead of transactions, we choose the TPC-H benchmark [42] for our comparisons.

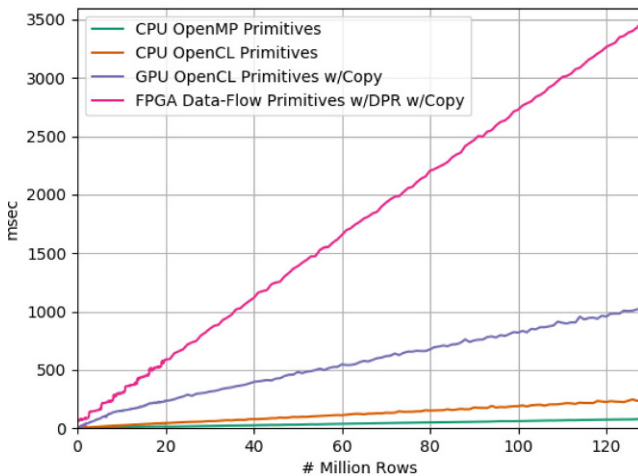


Fig. 9 TPC-H Q6: Execution time, including transfer overhead, for all devices

Due to limited space, we report a detailed analysis of one query, which we carefully selected. One important goal was to have a similar query execution plan for CPU and GPU. Such an operator-oblivious comparison is required to provide a fair comparison of both performance and system integration. Our secondary goal was to select either a complete query or a meaningful partial query pipeline that could be implemented with the finite hardware resources of common FPGAs. Furthermore, our current evaluation excludes algorithmic variants and computationally expensive multi-dimensional sorting operations. Such evaluation is extensively studied on their own already [13, 18, 20, 47].

Fig. 6 shows the data-flow graph of Q6 with its different types of operations. The selection primitives, consisting of three range comparisons, are connected via logical operations. This is shown in orange in the bottom half of Fig. 6. At the tail end Q6 contains two arithmetic operations, which are shown in green and violet. After the reduction, only a scalar result remains. For execution on FPGA, all primitives are mapped to the tiles of our FPGA overlay architecture. After selection, for CPU and GPU, results are materialized by gathering column values according to the 1-values in the resulting bitmap. In contrast, the data-flow processing using hardware-pipelined function units on the FPGA allows for in-place materialization using the filter primitive. The filter and post-processing primitives are shown in violet in Fig. 6. Since this tree-shaped query execution plan does not neatly fit onto a 2D-grid, non-compute FIFO-only Bridge FUs are used to allow flexible placement of the query's data-flow graph.

Overall Q6 contains enough optimization potential to allow all of the three architectures to apply their own strengths, but the execution plans on different devices are still similar enough to reach comparable results.

5.1 Device Setup

All measurements are taken on a server equipped with an AMD Epyc 7351P 16-Core Processor, an AMD Radeon RX Vega 56 with 8GiB HBM and a Xilinx Zynq-7000 ZC706 evaluation card with 2GiB DDR3 DRAM and Debian Linux as the operating system.

FPGA – Physical Setup: Our FPGA provides 218600 LUTs, 437200 FFs, 545 BRAMs, 900 DSPs [45]. The SoC comprises of Zynq PS: two ARM9-cores and one DDR memory controller. Based on the SoC and PCIe endpoint (i.e. the fixed-function units) layout, we implement the I/O-pins to DDR memory controller. Similarly based on the layout as well as the locations and number of DSP resources, we choose a 4×7 tiles grid for .

Each FU can at most use 1200 LUTs, 2400 FFs and 20 DSP units. Fig. 6 shows an abstract representation of both the logical and physical design, the latter of which is constrained by placement requirements for the PCIe core and DDR3 memory controller. For the same reason, only the two leftmost tiles on the bottom row are equipped with an extended interface, which in addition to the normal data streaming connections also allows random memory access. These tiles are sized slightly larger and occupy 1600 LUTs, 3200 FFs, 10 BRAMs, and 20 DSP units, but they still fit within the regular grid structure. All streaming connections within the overlay architecture are set to 32 bits. The complete design requires 73949 LUTs, 75203 FFs, 37.5 BRAM and no DSP units. In relation to the available resources, this is 34% of LUTs, 17% of FF and 7% of BRAM. Given the requirements of the DDR3 memory controller, the overlay architecture is clocked at 200Mhz.

6 Evaluation

In this section, we compare execution of the same TPC-H workload on different devices in our system. At first we present performance results for a micro-benchmark followed by a detailed end-to-end comparison of query 6 and hash join. To ensure comparable performance, we run our experiments on CPU, GPU and FPGA, all of which are available in the same environment and share the same system resources.

6.1 Micro-benchmark

We test the devices' general processing throughput by launching an increasing number of aggregate operators, with each input column consisting of 10 million integers. CPU and GPU process the aggregates sequentially, but with wide data parallelism within each primitive, while on the FPGA, functional parallelism between the aggregates

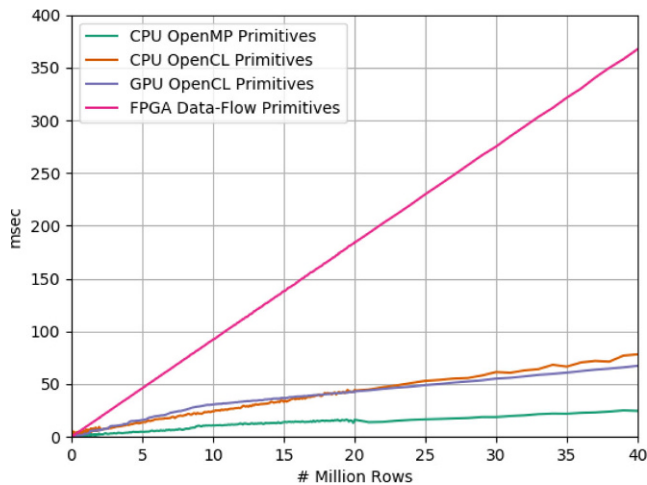


Fig. 10 TPC-H Q6: Pure execution time, showing details for smaller data sizes

is employed. The resulting execution times for the four primitive implementations on our three devices are shown in Fig. 7. The data shows that GPU, just as expected, by far outperforms CPU and FPGA due to its high memory bandwidth and optimized atomic operations. With its lower clock rate and limited throughput per DMA channel, the FPGA is much slower for a single aggregate than the other devices. However since all FUs can run completely independently from each other (or in a hardware pipeline), with statically allocated bandwidth per connection, the FPGA handles functional parallelism efficiently: Performance is nearly constant when increasing the number of concurrent reduce operations until memory controller, or rather, system interconnect bandwidth limits are reached. The step down in performance after three concurrent aggregates happens because at that point, the faster memory controller becomes saturated and additional aggregates have to fall-back to the slower memory. Finally, the grey line in Fig. 7 shows the theoretical maximum throughput for the FPGA, which is almost reached.

Due to their operator-at-a-time execution, performance of both CPU and GPU degrades linearly with increasing numbers of concurrent reduce operations. In case of the single aggregation, the runtime is less than 1ms for CPU OpenMP and GPU OpenCL versions. Hence, they are not visible in the Fig. 7. The GPU shows a near linear growth in execution time, as it executes one reduce after another. Still, the GPU performs well for reduce compared to other implementations. Even with the OpenMP variant of CPU being nearly twice as fast as the OpenCL version, they are slower compared to GPU.

6.2 Hash Joins

Unlike the primitives of query 6, a hash join implementation requires large amounts of intermediate data to be stored in order to determine the join pairs. Such random memory accesses are particularly expensive in FPGAs. To analyse the performance of joins, we can limit ourselves to just the primitives required for them, since they will be the most expensive primitives in the overall query pipeline.

We split the hash join into its two primitives: hash-table build and hash-table probe. We use linear probing for collision resolution and multiplicative hashing as hash function. On the FPGA, the join primitives can be loaded into the extended tiles equipped with random memory access. Since these tiles still need to fit within the regular grid of the physical design, their resources are constrained in similar ways as the pure streaming tiles. To circumvent these limitations, we could break these primitives out of the grid or implementing them in the static part. This would allow for higher-performance implementations at the cost of flexibility, for example by implementing large domain-specific caches. Another performance limitation is that the memory interface generated by the HLS compiler is not well-optimized for concurrent random memory access.

The execution time for hash join is given in Fig. 8. Due to the given limitations, we see a larger performance penalty for FPGA in comparison to the other devices.

Based on our results, the performance of OpenCL-based CPU hash-build is comparatively slower than that of OpenMP based implementation (cf. Fig. 11). However the OpenCL hash-build in CPU is faster than that of the OpenMP implementation. Such performance difference comes from threading strategies present in the implementations – OpenMP is coarse grained and OpenCL is fine grained [35]. Such performance from OpenCL and OpenMP are still worse compared to the GPU. Here, the massive parallel nature of the GPU is useful in probing multiple input values thereby having better performance than CPU.

6.3 Query 6

We report performance measurements for table sizes from 1000 rows up to 128 million rows which almost fills up the entire memory on the smallest device.

The communication overhead to start computation in non-CPU devices is included in the results. Looking at the data from this angle allows us to analyze end-to-end latency from the database system optimizer's point of view, including performance and quality of the host drivers. The figures detailing query execution time, including data transfer to PCIe devices, are organized as follows: Fig. 9 covers the whole range of input rows and includes all transfer times

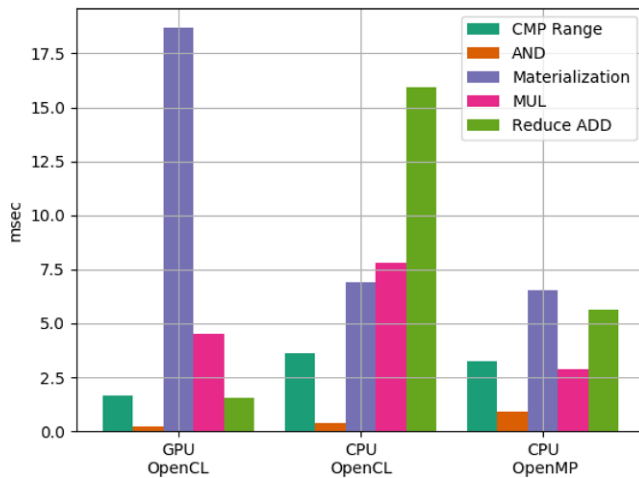


Fig. 11 TPC-H query 6: Pure execution time of the different primitives at 25 million rows

and extra overhead, in this case just the reconfiguration time for the FPGA. To facilitate a comparison focused on pure compute performance, Fig. 10 shows execution time only.

Finally, Fig. 11 shows the execution time of the single primitives for the platforms that do operator-at-a-time execution. Since the FPGA overlay architecture executes all primitives in parallel in a hardware pipeline, it is not included. This data shows strong evidence for implementing architecture-specific primitives. This is especially clear for the materialization primitive, which underperforms on GPU due to architectural disadvantages. In contrast, for the reduction, the GPU can bring its specialized hardware for atomic operations to bear.

6.4 Discussion

6.4.1 Access Bottleneck

Comparing the results shown in Fig. 9 and Fig. 10 it is clear that devices without direct access to system memory are at a disadvantage. In the case of a cold-start without most required data already present in the device-local memory, transfer times over PCIe are the dominating factor in execution time for both GPU and FPGA, something which can only be improved with newer system interconnects such as NVLink [17]. This effect is exacerbated for the FPGA card used, as both its internal memory bandwidth and PCIe throughput are fairly low compared to GPUs. Improving this with more powerful logic cores is possible, but requires a much higher amount of FPGA resources. However, if device-resident data is used, the situation is improved. For Q6, OpenCL on GPU is faster than the CPU OpenCL driver for more than 20 million rows, see Fig. 10.

6.4.2 Hardware-Sensitive vs. Hardware-Oblivious Primitives

A surprise is that for query execution on CPU, there is a big difference between the OpenCL primitives and OpenMP parallelized versions, despite using zero-copy buffer management for the OpenCL variant. This is due to the fact that OpenCL code written for massively parallel execution on a GPU does not translate to an optimal execution on a multi-core CPU. So even with the current state of OpenCL packages, the effort of implementing native primitives for CPU remains a sensible investment. Our evaluation supports existing evidence, that OpenCL is in a disadvantage compared to OpenMP for multi-core CPUs [35].

Due to the limited throughput and low memory bandwidth of the resource-efficient generic infrastructure components on the FPGA, the DMA interfaces into the overlay architecture reach only roughly half of their maximum bandwidth. Still, the FPGA processes query 6 at an input throughput of about 1.75 GB/s. Such execution performance can be improved by fusing multiple operator kernels together for FPGAs [14]. Finally, the stepping observed for small table sizes during data transfer onto the FPGA card are due to kernel-level I/O scheduling.

6.4.3 Power Comparison

While the overlay architecture is flexible and scalable, it must be emphasized that limited performance in system integration components is the major bottleneck for the FPGA. For a fair comparison, the power consumption differences should be minded: While the CPU generates up to 155W of waste heat and the GPU has a heat output of up to 210W, the FPGA evaluation card only draws about 12W. These numbers are based on the original equipment manufacturer's specifications [1, 2] and in addition for the FPGA, power estimates from the vendor design tools. For Query 6 with 128 million input rows, these power requirements translate to ca. 12J on the CPU using OpenMP-based primitives, 37J for the CPU OpenCL implementation, 32J for OpenCL on the GPU and finally, 14J for the FPGA. Even without data-parallel processing, the FPGA is competitive with the near-optimal CPU-based primitive implementation, while both OpenCL-based sets of primitives require a lot more energy.

6.4.4 Data-Parallelism on the FPGA

Setting the processing capabilities on the FPGA into context, the performance measurements reported are for hardware-pipelined tuple-at-a-time processing. On a card with higher memory bandwidth only $\times 2$ or $\times 4$ SIMD parallelism would be required to reach parity with CPU and GPU.

7 Conclusion

In this paper, we present an in-depth analysis of running OLAP queries on CPU, GPU and FPGA. We consider not only execution time alone, but also the overhead incurred by I/O and scheduling.

Our experiments consist of running the same micro-benchmark using aggregation, hash joins and Q6 from TPC-H on all three devices using portable operators (primitives) to obtain fair and comparable results. With our experiments, we observed that architecture-specific primitive implementations and execution engines are needed to better exploit the underlying architectures of CPUs, GPUs, and FPGAs. While direct access to main memory gives CPUs in general a large advantage over coprocessors, we again confirm the related work that OpenMP primitive implementations still outperform OpenCL on CPUs.

Regarding FPGAs, our overlay architecture offers a high scalability for analytical query processing, but the low memory bandwidth of the FPGA card's memory controller and system interconnect limit real-world performance. We argue that FPGAs are still to be considered for query processing due to its low power consumption compared to CPUs and GPUs.

Finally, with recent advancements in system interconnect technologies (e.g., NVLink), we argue that heterogeneous coprocessor-based database systems can become more competitive with CPU-only systems for general analytical database query processing.

Acknowledgements This work was partially funded by the DFG (grant no.: SA 465/51-1 and PI 447/9)

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. AMD: EPYC 7351P. <https://www.amd.com/en/products/cpu/amd-epyc-7351p>. Accessed 25 May 2021
2. AMD: Radeon RX Vega 56. <https://www.amd.com/en/products/graphics/radeon-rx-vega-56>. Accessed 25 May 2021
3. Becher A et al (2018) Integration of FPGAs in database management systems: challenges and opportunities. *DB Spektrum* 18(3):145–156
4. Becher A et al (2019) ReProVide: towards utilizing heterogeneous partially reconfigurable architectures for near-memory data processing. *BTW Workshops*, p 51
5. Betz V, Rose J (1997) VPR: a new packing, placement and routing tool for FPGA research. *Proceedings of the 7th International Conference on Field-Programmable Logic and Applications*, pp 213–222
6. BlazingDB (2020) BlazingSQL: high performance SQL engine on RAPIDS AI. <https://blazingsql.com/>. Accessed 28 May 2020
7. Boncz PA et al (1999) MIL primitives for querying a fragmented world. *VLDB J* 8(2):101–119
8. Boncz PA et al (2019) Database architectures for modern hardware. *Dagstuhl Seminar*, vol 18251. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern
9. Breß S (2013) Why it is time for a HyPE: a hybrid query processing engine for efficient GPU coprocessing in DBMS. *Proc VLDB Endow* 6(12):1398–1403
10. Breß S (2014) The design and implementation of CoGaDB: a column-oriented GPU-accelerated DBMS. *Datenbank Spektrum* 14(3):199–209
11. Breß S et al (2014) GPU-accelerated database systems: survey and open challenges. In: *Transactions on large-scale data- and knowledge-centered systems XV*. Springer, Berlin, Heidelberg https://doi.org/10.1007/978-3-662-45761-0_1
12. Broneske D, Breß S, Heimele M, Saake G (2014) Toward hardware-sensitive database operations. *Proceedings 17th International Conference on Extending Database Technology (EDBT)*, pp 229–234
13. Chen R, Prasanna VK (2016) Accelerating equi-join on a CPU-FPGA heterogeneous platform. *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp 212–219 <https://doi.org/10.1109/fccm.2016.62>
14. Drewes A, Joseph JM, Gurumurthy B, Broneske D, Saake G, Pionteck T (2020) Optimising operator sets for analytical database processing on FPGAs. In: Rincón F, Barba J, So H, Diniz P, Caba J (eds) *Applied reconfigurable computing. Architectures, tools, and applications ARC 2020. Lecture notes in computer science*, vol 12083. Springer, Cham, pp 30–44 https://doi.org/10.1007/978-3-030-44534-8_3
15. Drewes T, Joseph JM, Gurumurthy B, Broneske D, Saake G, Pionteck T (2018) Efficient inter-kernel communication for opencl database operators on FPGAs. *2018 International Conference on Field-Programmable Technology (FPT)*, pp 266–269 <https://doi.org/10.1109/fpt.2018.00050>
16. Fang R, He B, Lu M, Yang K, Govindaraju NK, Luo Q, Sander PV (2007) GPUQP: query co-processing using graphics processors. *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data - SIGMOD '07*, pp 1061–1063 <https://doi.org/10.1145/1247480.1247606>
17. Foley D et al (2017) Ultra-performance pascal GPU and NVLink interconnect. *IEEE Micro* 37(2):7–17
18. Grozea C, Bankovic Z, Laskov P (2010) FPGA vs. multi-core CPUs vs. GPUs: hands-on experience with a sorting application. In: Keller R, Kramer D, Weiss JP (eds) *Facing the multicore-challenge. Lecture notes in computer science*, vol 6310. Springer, Berlin, Heidelberg, pp 105–117 https://doi.org/10.1007/978-3-642-16233-6_12
19. Gurumurthy B et al (2018) Cooking DBMS operations using granular primitives. *Datenbank Spektrum* 18(3):183–193
20. Halstead RJ, Absalyamov I, Najjar WA, Tsotras VJ (2015) FPGA-based multithreading for in-memory hash joins. *7th Biennial Conference on Innovative Data Systems Research (CIDR '15)*.
21. Han TD, Abdelrahman TS (2011) Reducing branch divergence in GPU programs. *Proceedings of the Fourth Workshop on General*

- Purpose Processing on Graphics Processing Units - GPGPU-4. <https://doi.org/10.1145/1964179.1964184>
22. He B et al (2009) Relational query coprocessing on graphics processors. *ACM Trans Database Syst* 34(4):1–39
 23. Heimel M et al (2013) Hardware-oblivious parallelism for in-memory column-stores. *Proc VLDB Endow* 6(9):709–720
 24. Hennessy JL, Patterson DA (2011) *Computer architecture: a quantitative approach*. Elsevier
 25. Khronos Group (2020) OpenCL. <https://www.khronos.org/opencl/>. Accessed 25 May 2020
 26. Kinetica (2020) Kinetica high performance analytics database. <http://www.kinetica.com/>. Accessed 28 May 2020
 27. Hemsoth N (2016) Baidu takes FPGA approach to accelerating SQL at scale. <https://www.nextplatform.com/2016/08/24/baidu-takes-fpga-approach-accelerating-big-sql/>. Accessed 28 May 2020
 28. OmniSci (2020) Accelerated analytics platform. <http://www.omnisci.com/>. Accessed 28 May 2020
 29. Owaida M, Sidler D, Kara K, Alonso G (2017) Centaur: a framework for hybrid CPU-FPGA databases. 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp 211–218 <https://doi.org/10.1109/fccm.2017.37>
 30. Pinnecke M et al (2017) Are Databases Fit for Hybrid Workloads on GPUs? A Storage Engine’s Perspective. Proceedings of the 33rd International Conference on Data Engineering, pp. 1599–1606. <https://doi.org/10.1109/ICDE.2017.237>.
 31. Pirk H, Moll O, Zaharia M, Madden S (2016) Voodoo – a vector algebra for portable database performance on modern hardware. *Proc VLDB Endow* 9(14):1707–1718. <https://doi.org/10.14778/3007328.3007336>
 32. Putnam A, Caulfield AM, Chung ES, Chiou D, Constantinides K, Demme J et al (2014) A reconfigurable fabric for accelerating large-scale datacenter. 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), pp 13–24 <https://doi.org/10.1109/isca.2014.6853195>
 33. Saecker M, Markl V (2013) Big data analytics on modern hardware architectures: a technology survey. In: Aufaure MA, Zimányi E (eds) *Business intelligence eBISS 2012*. Lecture notes in business information processing, vol 138. Springer, Berlin, Heidelberg https://doi.org/10.1007/978-3-642-36318-4_6
 34. Salami B, Malazgirt GA, Arcas-Abella O, Yurdakul A, Sonmez N (2017) AxleDB: a novel programmable query processing platform on FPGA. *Microprocess Microsyst* 51:142–164. <https://doi.org/10.1016/j.micpro.2017.04.018>
 35. Shen J, Fang J, Sips H, Varbanescu AL (2012) Performance gaps between OpenMP and OpenCL for multi-core CPUs. 2012 41st International Conference on Parallel Processing Workshops, pp 116–125 <https://doi.org/10.1109/icppw.2012.18pp>
 36. Sidler D, Owaida M, Istvan Z, Kara K, Alonso G (2017) doppioDB: a hardware accelerated database. 2017 27th International Conference on Field Programmable Logic and Applications (FPL), pp 1659–1662 <https://doi.org/10.23919/fpl.2017.8056864>
 37. Siewert S (2009) Using intel streaming SIMD extensions and intel integrated performance primitives to accelerate algorithms
 38. SQreamDB (2020) SQream – GPU Data Warehouse. <https://sqream.com/product/>. Accessed 28 May 2020
 39. Sukhwani B, Min H, Thoennes M, Dube P, Iyer B, Brezzo B et al (2012) Database analytics acceleration using FPGAs. Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques - PACT ’12, pp 411–420 <https://doi.org/10.1145/2370816.2370874>
 40. The Apache Software Foundation (2018) ApacheSpark. <https://spark.apache.org/>. Accessed 14 May 2020
 41. The Apache Software Foundation (2019) ApacheStorm. <https://storm.apache.org/>. Accessed 14 May 2020
 42. Transaction Processing Performance Council (2014) TPC benchmark H (decision support). Tech. Rep. 2.17.1
 43. Turan F, Roy SS, Verbaughede I (2020) HEAWS: an accelerator for homomorphic encryption on the Amazon AWS FPGA. *IEEE Trans Comput*. <https://doi.org/10.1109/tc.2020.2988765>
 44. Xilinx (2014) SDAccel development environment backgrounder. <https://www.xilinx.com/support/documentation/backgrounders/sdaccel-backgrounder.pdf>. Accessed 3 June 2020
 45. Xilinx (2018) Zynq 7000 SoC data sheet: overview. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. Accessed 3 June 2020
 46. Xillybus (2020) An FPGA IP core for easy DMA over PCIe with Windows and Linux. <http://xillybus.com/>. Accessed 4 June 2020
 47. Zhang C, Chen R, Prasanna V (2016) High throughput large scale sorting on a CPU-FPGA heterogeneous platform. 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp 148–155 <https://doi.org/10.1109/ipdpsw.2016.117>
 48. Zhang S, He J, He B, Lu M (2013) OmniDB: towards portable and efficient query processing on parallel CPU/GPU architectures. *Proc VLDB Endow* 6(12):1374–1377. <https://doi.org/10.14778/2536274.2536319>
 49. Ziener D et al (2016) FPGA-based dynamically reconfigurable SQL query processing. *ACM Trans Reconfigurable Technol Syst* 9(4):25:1–25
 50. Zukowski M, van de Wiel M, Boncz P (2012) Vectorwise: a vectorized analytical DBMS. 2012 IEEE 28th International Conference on Data Engineering, pp 1349–1350 <https://doi.org/10.1109/icde.2012.148>