Foundations and Trends® in Databases

FPGA-Accelerated Analytics: From Single Nodes to Clusters

Suggested Citation: Zsolt István, Kaan Kara and David Sidler (2020), "FPGA-Accelerated Analytics: From Single Nodes to Clusters", Foundations and Trends® in Databases: Vol. 9, No. 2, pp 101–208. DOI: 10.1561/1900000072.

Zsolt István
IMDEA Software Institute
Spain
zsolt.istvan@imdea.org

Kaan Kara
Oracle Labs
Switzerland
kaan.kara@oracle.com

David Sidler
Microsoft Corporation
USA
david.sidler@microsoft.com

now
the essence of knowledge
Boston — Delft

Contents

FPGA-Accelerated Analytics: From Single Nodes to Clusters

Zsolt István[1], Kaan Kara[2] and David Sidler[3]

[1]*IMDEA Software Institute, Spain; zsolt.istvan@imdea.org*
[2]*Oracle Labs, Switzerland; kaan.kara@oracle.com*
[3]*Microsoft Corporation, USA; david.sidler@microsoft.com*

ABSTRACT

In this monograph, we survey recent research on using reconfigurable hardware accelerators, namely, Field Programmable Gate Arrays (FPGAs), to accelerate analytical processing. Such accelerators are being adopted as a way of overcoming the recent stagnation in CPU performance because they can implement algorithms differently from traditional CPUs, breaking traditional trade-offs. As such, it is timely to discuss their benefits in the context of analytical processing, both as an accelerator within a single node database and as part of distributed data analytics pipelines. We present guidelines for accelerator design in both scenarios, as well as, examples of integration within full-fledged Relational Databases. We do so through the prism of recent research projects that explore how emerging compute-intensive operations in databases can benefit from FPGAs. Finally, we highlight future research challenges in programmability and integration, and cover architectural trends that are propelling the rapid adoption of accelerators in datacenters and the cloud.

Zsolt István, Kaan Kara and David Sidler (2020), "FPGA-Accelerated Analytics: From Single Nodes to Clusters", Foundations and Trends® in Databases: Vol. 9, No. 2, pp 101–208. DOI: 10.1561/1900000072.

1

Introduction

Big Data has been instrumental to our lives in the last decade and has lead to scientific insights, the boom of Machine Learning and the emergence of novel on-line services. Datacenters that are hosting such data-intensive applications are facing however an important challenge: the amount of data that needs to be stored and processed is increasing at an exponential rate whereas traditional processor performance has been stagnating for years. As Moore's law and Dennard scaling taper off, CPU transistor counts are growing less year-on-year and, due to heat dissipation issues, it is becoming challenging to power on all parts of CPUs at high clock rates at the same time, resulting in "dark silicon". The stagnation of CPU performance, however, presents opportunities as well: CPUs can now incorporate more heterogeneous components without having to keep them powered on at all times. In addition, for many workloads, replacing CPUs partially or entirely with specialized hardware has become the more economic choice. These devices can utilize transistors more efficiently for the tasks at hand, delivering more performance for the same energy budget.

Driven by the above described trends, and in order to keep up with growing data sizes, data processing and management applications have

been becoming increasingly distributed. Today, applications built with platforms such as Apache Spark, routinely span hundreds of server nodes. While helpful in reducing compute bottlenecks, this distribution brings new data movement bottlenecks at various levels of the software and hardware architecture. In this monograph we focus on how specialized hardware accelerators can provide an answer to the compute stagnation problem and showing how they can be also helpful in reducing data movement bottlenecks by placing them in the right location within the computer architecture.

There are many different technologies one could use for building accelerators but one particular technology stands out as a middle ground between energy efficiency and versatility: Field Programmable Gate Arrays (FPGAs). FPGAs make it possible to express algorithms in ways that are fundamentally different from CPUs or GPUs: FPGAs have no instruction sets and functionality is laid out directly as circuitry. As opposed to traditional CPUs that concentrate on-chip memory into layers of caches, FPGAs have small SRAM memories distributed throughout the device that can be configured flexibly and can be co-located with computation. These differences to CPUs result in a higher performance in the same or lower energy footprint, making FPGAs attractive both as accelerators and as energy-efficient replacements of software solutions. FPGAs are *re*configurable meaning that they can implement different functionality over time and can be reprogrammed from software. Once programmed, they act as integrated circuits (ASICs), bringing significant improvements in energy efficiency when compared to CPUs.

Today, FPGAs are available in most clouds (e.g., Amazon F1 Instances, n.d.; Firestone, 2016; Putnam, 2014; Weerasinghe *et al.*, 2016) as accelerators. As a prominent example, the Microsoft Catapult project uses FPGAs to create programmable network-interface cards (NICs) to offload tenant network functions from the CPU. Meanwhile, programmable co-processors based on FPGAs are also becoming more common, for instance, with projects such as Intel Xeon+FPGA (Gupta, 2015). Storage devices/nodes are also increasingly more programmable,

with examples such as Samsung's SmartSSD[1] and Amazon Redshift AQUA.

Several early projects of FPGA-based database acceleration proposed PCIe-attached accelerator cards a decade ago, demonstrating that FPGAs are able to speed up projection and selection (Dennl *et al.*, 2012; Salami *et al.*, 2017; Sukhwani *et al.*, 2013; Wang *et al.*, 2016a; Woods *et al.*, 2013), aggregation (Dennl *et al.*, 2013; Salami *et al.*, 2017), joins and even sorting (Casper and Olukotun, 2014; Sukhwani *et al.*, 2013; Zhang *et al.*, 2016), by an order of magnitude when compared to commonly used row-stores, such as MySQL and PostgreSQL. However, in many previous systems, once all integration costs were factored in, in particular the cost of data transfers over PCIe, the benefits were significantly reduced. In the meantime, however, the bandwidth of interconnects (PCIe, NVMe, etc.) and networks has been increasing at a steady pace, making FPGA acceleration once again an attractive option. Furthermore, due to increased data sizes, there are more and more distributed databases that suffer from various data movement bottlenecks, for instance, when retrieving data from distributed storage nodes or when shuffling tuples between compute nodes for a join operation. Specialized hardware can be used to move computation closer to the source, and by reducing data sizes through filtering or transformations close to storage, memory, etc., reduce data movement bottlenecks.

In this monograph we explore what is required for integrating these accelerators with software systems (focusing on database management systems) and how one should design the acceleration functionality to ensure that the overall speedups in the system are worth the additional complexity. The monograph provides a detailed look into several representative examples of FPGA-accelerated databases and the internals of accelerated SQL operators, Machine Learning operators and data shuffling operators.

Monograph Structure. In this section (Section 1), we outlined the reasons why FPGAs and similar specialized hardware have become not only economically feasible for database acceleration but, in many

[1] https://samsungsemiconductor-us.com/smartssd/index.html.

cases, even necessary to keep up with data growth. We also provided an intuition on how, thanks to a fundamentally different execution model from CPUs and GPUs, these devices offer benefits in efficiency. The rest of the monograph is structured as follows:

- In Section 2 we present a background on FPGAs and highlight their differences with ASICs. Readers already familiar with FPGAs and their development work flow might consider skipping this section.
- Section 3 covers salient aspects of integrating FPGAs in data processing systems, in various locations of the software and hardware architecture. We describe several representative examples of how FPGAs are integrated with databases and what features the frameworks that enable this need to provide.
- In Section 4 we turn our attention to individual operators from the domain of core SQL acceleration, Machine Learning and Distributed Joins. We present both high level guidelines to help future FPGA programmers design circuits adequate to the task and deep dive into the design and performance characteristics of representative operators, implemented by the authors.
- Section 5 describes the remaining challenges related to programming and sharing FPGA accelerators more easily in datacenters and the cloud. This section also provides a peek into the future of re-programmable hardware and the opportunities this will bring for database management systems.

2

Background

2.1 What Are FPGAs?

Field Programmable Gate Arrays (FPGAs) are reconfigurable hardware chips. Thanks to their programmability, they can be used to implement a wide range of functionality. Traditionally, FPGAs have been used to prototype and validate Application-Specific Integrated Circuit (ASICs) designs or to facilitate firmware updates of various hardware devices, such as routers or switches. Recently, however, they have used increasingly often as data processing accelerators in datacenters thanks to their orders of magnitude better energy efficiency than that of traditional CPUs (Teubner and Woods, 2013). Even though ASICs require even less power for the same functionality, FPGAs provide flexibility because their role can change over time, as opposed to an ASIC that has fixed functionality.

Internally, FPGAs are composed of look-up tables (LUTs), on-chip block memory (BRAM) and digital signal processing units (DSPs). All these components can be configured and interconnected flexibly, allowing the programmer to implement custom processing elements (Figure 2.1). It is not uncommon to have small ARM cores integrated inside the programmable fabric, e.g., in Xilinx's Zynq product line.

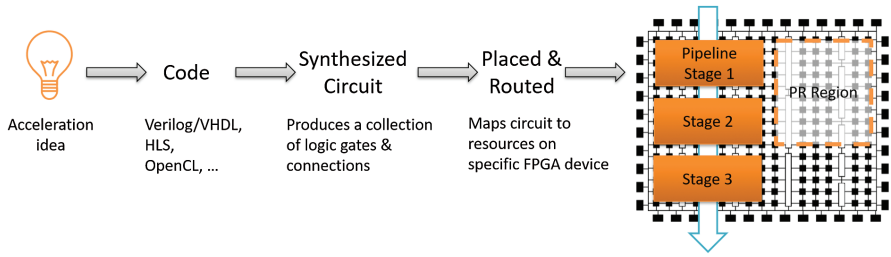


Figure 2.1: The typical steps of programming FPGAs are shown above. The tools spend most of their time mapping the synthesized circuit onto the FPGA. This is because the chip is composed of many programmable gates and memories that have to be configured and connected together in a 2D space, ensuring that signals can propagate correctly within clock periods.

As Figure 2.1 shows, FPGAs are programmed as follows: functionality is expressed in a hardware description language, such as Verilog or VHDL, or using a high level language. The former has the benefit that it allows programmers to define registers, wires and logical functions directly, giving them fine-grained control but the latter increases productivity since functionality is expressed in a C-like syntax. Based on the source code, the design is synthesized into a logic gate-based representation. Based on the synthesized design, that is compatible with a wide range of FPGAs, a device-specific "bitstream" is created. To produce this bitstream, the tools have to define the location of each logical gate on the "chip surface" (placement) and define connections and route of these connections between circuit elements (routing). Since FPGAs have flexible clocking options and the programmer is free to define a target frequency (e.g., 300 MHz), the tools have to set up routing such that signals can be propagated within the clock periods. This is not always possible and will require the programmer to either lower the target frequency, or to segment the design into simpler pipeline stages.

In addition to reprogramming the entire device, it is also possible to perform partial reconfiguration (PR), meaning that only a portion of the FPGA's resources are reprogrammed (illustrated on the right-hand side of Figure 2.1). This operation takes in the order of milliseconds depending on the size of the region and comes with two limitations: the

regions can only be defined at coarse granularity, their size can't be redefined at runtime.

When programming FPGAs, there are two types of parallelism to take advantage of: pipeline parallelism and data-parallel execution. Pipeline parallelism means that complex functionality can be split up into stages executed in parallel, without reducing throughput. The benefit of FPGAs in this context is that the communication between pipeline stages is very efficient thanks to the physical proximity and availability of on-chip memory to construct FIFO buffers. The second type of parallelism to be exploited is data-parallel execution. This is similar to SIMD (single instruction multiple data) processing in CPUs but operations are coarser grained and often it's possible to have diverging control flows. Mixing these two types of parallelism, combined with the ability to express computation directly as a hardware circuit, without having to encode them through an instruction abstraction layer, makes FPGAs into compelling accelerators.

One important limitation of FPGAs is that all application logic occupies chip space and there is no possibility of "paging" code in or out dynamically. This means that the complexity of the operator that is being offloaded is limited by the available logic resources (area) on the FPGA. In the database context, even though, it is possible to change the set of operators running on the FPGA dynamically, even if using partial reconfiguration, it is not practical to have operators that do not fit entirely on the FPGA. The chip area limitation applies to the "state" of an algorithm as well, because for high performance, this often has to be stored in on-chip BRAM that can be accessed in a single clock cycle. If the data doesn't fit in the available BRAM, high latency off-chip DRAM has to be used, which will degrade performance significantly.

Even though FPGAs excel at parallel and pipelined execution, they behave poorly when an algorithm requires iterative execution with data dependencies or has widely branching "if-then-else" logic. In the case of the former, CPUs will almost always deliver higher performance thanks to their higher clock rates. In the case of the latter, the branching logic needs to be mapped to logic gates that encode all outcomes, resulting in very large circuits. Since the space on the FPGA is limited, the larger circuits result in reduced parallelism, which in turn leads to lower

throughput. This means that even though FPGAs could be successful in accelerating the common case of an algorithm, they might not be able to handle corner cases, and in practice this leads to uncertainty in the query optimizer or even to wasted work, if an unexpected corner case is encountered during execution.

2.2 FPGAs vs. ASICs

FPGAs are by no means the only type of specialized hardware that could be deployed in modern datacenters. As illustrated in Figure 2.2, these devices span a large trade-off space ranging from energy efficiency and to flexibility. Traditional CPUs represent the peak of flexibility but for this a high energy cost is paid. GPUs, for instance, achieve a better compute efficiency than CPUs because they restrict the instruction set used. More specialized processors, such as the Google TPU,[1] achieve even better efficiency by limiting the types of operands used for computations (e.g., tensors only in the TPU case).

FPGAs are different from the previously mentioned microprocessors because they do not provide an instruction set but rather, allow the programmer to express the application logic directly as circuits, removing a layer of abstraction. For this reason, they achieve at least an order of magnitude lower power consumption than CPUs for the same

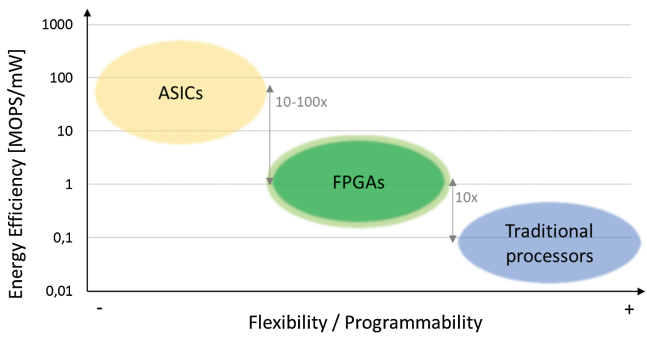


Figure 2.2: FPGAs constitute a compute efficiency middle ground between fixed-function ASICs and fully general purpose CPUs.

[1] https://cloud.google.com/tpu/.

performance target. The cost of increased efficiency is reduced flexibility: an FPGA will be able to perform only the task that it has been programmed for and changing its behavior will interrupt processing.

There are devices that offer even higher compute efficiency at the cost of further reduction of flexibility. Coarse Grained Reconfigurable Arrays (CGRAs) are similar to FPGAs in that they can be reprogrammed but their underlying logic resources are not general purpose but are designed exclusively for a specific application domain (Gao and Kozyrakis, 2016). The overall best compute efficiency can be achieved, of course, if there is no requirement for the hardware device to be re-programmable. Application-specific Integrated Circuits (ASICs) are fixed purpose and cannot be changed after production but achieve several orders of magnitude higher compute efficiency even when compared with FPGAs. They are used abundantly in network interface cards, network switches, disk drives, etc., because the underlying functionality and the responsibilities of the devices does not change.

Finally, FPGAs allow for a faster time to market in comparison to ASICs, measured in weeks instead of months. When combined with the ability to adjust or entirely change their functionality directly in a production environment has made them an attractive choice. As a good example, programmability of the accelerator was a main criteria when choosing the device type for Microsoft's Catapult (Putnam *et al.*, 2014) platform. The latest public figures put the number of FPGAs deployed in Microsoft's datacenters within the Catapult project in the millions – a significant proof that such devices can be used in production workloads. The low power usage of FPGAs allowed Microsoft to deploy them in a regular server chassis with minor impact on cooling and power consumption, without having to give up flexibility or having to commit to a specific accelerator domain. For instance, when used as a network accelerator, the Catapult devices operate independently, but when used as part of Brainwave (Chung *et al.*, 2018) to offload Bing ranking, these devices are connected over the network to form a pipeline of seven FPGAs. Each one is programmed dynamically to execute a different stage of the processing pipeline. If, instead, one would use ASICs to accelerate the Bing document ranking then it would require either different ASICs for the different pipeline stages, an economically infeasible choice, or

that each device is able to process the full pipeline, likely increasing the complexity of the design and reducing its throughput significantly. This demonstrates the benefit of the programmability of FPGAs in environments with changing workloads. Of course, in time, if a specific workload matures and is not expected to change again in the lifetime of a server machine, the corresponding functionality could be "hardened" into an ASIC.

One example of a project that opted for ASICs instead of FPGAs, or similar re-configurable hardware, is the Oracle DAX (Oracle, 2015) co-processor. This co-processor puts small "cores" close to the cache of an Oracle SPARC CPU to implement database-specific data manipulation operations, such as lightweight data decompression, scan acceleration, and comparison-based filtering. Since the CPUs are designed for use with databases and the data manipulation primitives are a mature and non-changing part of the Oracle database engine, it was more economical to turn the DAX functionality into an ASIC that is part of the processor chip. As a result, the DAX occupies negligible chip space and does not increase the cost or energy consumption of the CPU while offering significant performance boost for a wide selection of queries.

2.3 Where to Deploy Acceleration?

The wide range of accelerators proposed for and already deployed in datacenters can be categorized in three categories, depending on their location with regards to the data source and CPU: *on-the-side*, *in data-path* and *co-processor*.

Perhaps the most typical way we think of accelerators, or programmable hardware in general, is as being *on-the-side*, attached to the processor via an interconnect, for instance PCIe (shown visually in Figure 2.3). In the case of such accelerators the CPU owns the data and explicitly sends it to the accelerator for processing, resulting usually in significant additional latency per operation due to communication latency and data transformation overhead. These additional costs encourage offloading operations at large granularity and without requiring excessive communication between the CPU and the accelerator. GPUs are a common example of this kind of accelerator and, in the database

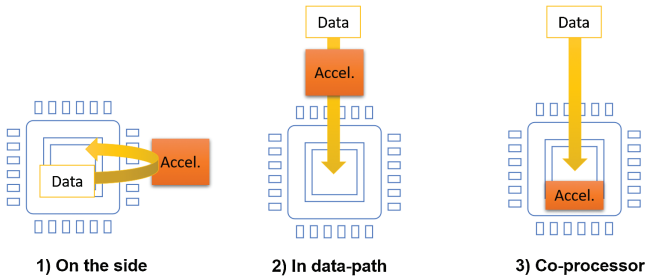


Figure 2.3: Programmable hardware accelerators can be deployed either as "on-the-side" accelerator (e.g., GPUs), as "in data-path" accelerator (e.g., smart NICs, smart SSD), or as co-processor (e.g., in Oracle DAX or Intel Xeon+FPGA).

context, were shown to be useful, for instance, to offload LIKE-based string queries (Sitaridi and Ross, 2016), general filtering and spatial joins (Root and Mostak, 2016). There have also been proposals that deploy FPGAs this way for data filtering and decompression, e.g., in the work by Sukhwani *et al.* (2012).

Another way of deploying acceleration functionality is *in data-path* (Figure 2.3). This deployment option can be thought of as a generalized version of near-data processing (Oskin *et al.*, 1998): the goal of the accelerator is to filter or transform data at the speed that it is received from the data source, positioned ideally before communication bottlenecks. The main challenge in this context is that designs that can't guarantee line-rate processing (that is, at the speed of the network/storage device) can slow down the entire system (Koo *et al.*, 2017). Much of the research effort regarding in data-path accelerators has been centered around in-Flash and in-SSD processing (Jo *et al.*, 2016; Woods *et al.*, 2014). More recently, however, there have been efforts in using programmable network interface cards (NICs) to accelerate distributed databases (Barthels *et al.*, 2015; Dragojević *et al.*, 2014). NICs are useful for data manipulation acceleration at the end-host, but there are also efforts in making networking hardware more programmable (Bosshart *et al.*, 2014) and used to make distributed operations, such as aggregation of data from multiple nodes, more efficient (Mai *et al.*, 2014).

In addition to the previous two deployment options, there is a third one, driven by the slowdown in Moore's law and Dennard Scaling

(Esmaeilzadeh *et al.*, 2011; Taylor, 2012). Given the limits of multi-core performance scalability, in the future it can be beneficial to include heterogeneous *co-processors* in CPUs, instead of adding more cores (Figure 2.3). An other co-processor platform, the Intel Xeon+FPGA (Gupta, 2015), incorporates a general purpose FPGA "core" beside the CPU cores that can be used for acceleration of various workloads. The FPGA has high bandwidth cache-coherent access to the main memory of the machine and this creates acceleration opportunities without the usual overhead of the on-the-side accelerators.

The above presented three integration options bring different benefits and limitations. As a rule of thumb, the choice of what to offload to programmable accelerators is dictated by the way these are connected to the machine and the database management system running on it. For instance, an on-the-side accelerator favor offloading very compute-intensive operators that require little coordination from the CPU, while a smart storage engine would favor selection predicates that result in the highest possible selectivity, to alleviate data movement bottlenecks. With the emergence of shared-memory systems, such as the Intel Xeon+FPGA or IBM CAPI, the overhead of performing work on the specialized hardware has been reduced. This opens up acceleration opportunities across the stack. As a result, the choice of what to offload can be driven by the workloads and lead to better overall utilization of resources.

3

FPGAs in Datacenters and Clouds

3.1 The Path to Adoption

Warehouse-scale datacenters, as established by Google and others in the early 2000's, were driven by the deployment of cost-effective commodity hardware. In the meantime, the scale and number of datacenters increased and cloud providers have to market their services aggressively to compete, changing the economics of operating a datacenter. As such, from an economical point of view, multiple factors emerged that support the deployment accelerators and specialized hardware:

- Energy consumption contributes significantly to the operating costs of a datacenter and offloading task from the general-purpose CPU to specialized hardware not only increases performance but also decreases power usage.
- To gain a competitive advantage in cost, performance, or functionality over their competitors, datacenter provides need to develop in-house accelerators and cannot rely on commodity hardware.
- The increased scale and number of datacenters makes it feasible to invest into the necessary infrastructure and know-how to develop, deploy, and operate accelerators in production.

While the change in economical factors benefits accelerators in general, the increased adoption of FPGAs is also due to their rapid improvements over the years. The size of FPGAs was increasing according to Moore's law and, even as CPUs fail to scale further, FPGAs, with their relatively simple and homogeneous internal architecture continue to benefit. They are used as a process-driver by the foundries, allowing them to adopt the new process sooner than other more complex accelerators.

The improvements in FPGA area is illustrated by the following example: the Xilinx flagship device in 1999, the Virtex XCV1000, had 27,000 logic cells. In comparison, the Virtex 7 flagship introduced in 2010 featured over one million logic cells. In 2020, Xilinx is selling high-end chips with as many as nine million logic cells. While in the past area was very scarce, suddenly a wide range of applications could easily be mapped to an FPGA. This allowed manufactures to divert part of the transistors on the device to dedicated logic blocks that could benefit specific application domains significantly, without introducing significant cost if unused. These dedicated blocks offer functionalities such as memories, multipliers, high-speed transceivers, microprocessors, Ethernet MAC, PCIe, and floating point arithmetics. The introduction of dedicated logic blocks made FPGAs more powerful, increased their flexibility, and improved their interoperability with other devices. The increased I/O capabilities and improvements in routing (i.e., the ability to lay out very large circuits on the device in a way that it can reach high clock frequencies), led to a wide-spread adoption of FPGAs in datacenters.

Apart from the interfaces to get data in and out of the FPGA, such as PCIe and Ethernet, additional hard- and software infrastructure was necessary to deploy FPGAs in the datacenter. On the hardware side, a reusable "shell" was implement to provide common abstractions to the raw I/O interfaces. The idea of a "shell" was first introduced by Microsoft's Catapult (Putnam *et al.*, 2014), and was also adopted by the manufactures to support the deployment of OpenCL kernels on FPGAs. As we described in Subsection 3.2, today many different integration frameworks exist, building on a similar shell idea. In addition to helper- and management-logic on the FPGA, there has been a great deal of

progress in software-side drivers and libraries. In the early 2000s, these were very limited in functionality and often used more as a debug tool, not focusing on performance. Today, however, software libraries such as Intel's ONEAPI, Xilinx SDAccel, etc., can efficiently move data on and off the device, monitor its health, and even recover from failures.

3.2 Interfacing with FPGA Accelerators

In addition to the differences in terms of execution model, FPGAs are also different from CPUs in that there is no operating system or firmware running on the device by default. Modules responsible for communication with the outside components have to be implemented and deployed alongside the functionality that is being accelerated. Depending on the physical placement of the FPGA, the ways of communicating with software will vary and so will the management mechanisms. For instance, if deployed as a co-processor, it is expected that operating-system-like features will be present on the FPGA but if deployed as a PCIe-attached accelerator card, perhaps it will interface only with very specific applications and will lack a proper driver.

In the following we present a small selection of frameworks and integration mechanisms used on FPGAs, organized based on the physical location of the FPGA.

3.2.1 On-the-Side FPGAs and Co-Processors

To avoid re-implementing the basic communication and management modules as well as, features expected to be present by an operating system, there have been several frameworks appearing (Agron and Andrews, 2009; Ismail and Shannon, 2011; Iturbe *et al.*, 2013; Lübbers and Platzner, 2009; Owaida *et al.*, 2017). These typically abstract away the PCIe bus or memory controller interfaces into generic streaming interfaces. Figure 3.1 shows a high level overview of how different components interact.

As for exposing the acceleration functionality to software, often a POSIX-like thread abstraction is applied, such as in ReconOS (Lübbers and Platzner, 2009) and hthreads (Agron and Andrews, 2009). These

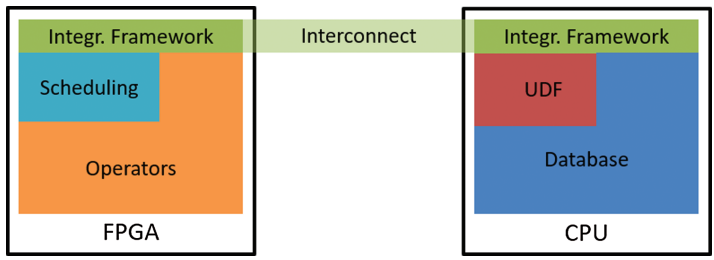


Figure 3.1: Accelerators deployed on FPGAs require an Integration Framework to communicate with software. Optionally, an on-FPGA Scheduler can be also used that allows dynamic deployment of operators. On the database side, integration can be done either directly with the Query Execution Engine or through a UDF.

frameworks target embedded devices and system-on-chip platforms where the FPGA is next to an embedded CPU, so they are often not suitable for bandwidths typical for large server machines. Emerging frameworks, such as the Intel Acceleration Stack, focus on providing access to FPGAs in a shared-memory CPU-FPGA system (Oliver *et al.*, 2011). Centaur (Owaida *et al.*, 2017) is a framework that builds on top of the Intel Acceleration Stack, focusing on database use-cases and exposing a thread-like interface.

In addition to the choice of interface with the acceleration modules on the FPGA, there is also the question of sharing the device across multiple queries, in a database setting, or between multiple tenants, in a cloud setting. There are proposals for virtualization of FPGA resources in cloud computing (Asiatici *et al.*, 2017; Byma *et al.*, 2014; Chen *et al.*, 2014; Zhang *et al.*, 2017b; Zhu *et al.*, 2018) – Vaishnav *et al.* (2018) provide an extensive survey of this field.

Most solutions for sharing FPGAs across workloads rely on partial reconfiguration (PR), which involves reprogramming a pre-defined region of the FPGA at runtime. The PR regions are treated as dynamically scheduled, time-shared computing units, as a way of virtualizing FPGA-based processing. The main drawback is the high cost of PR, which can be in the order of seconds depending on the size of the region, making fine-grained time-sharing impractical. Preemptive scheduling has also been proposed for FPGAs (Happe *et al.*, 2015; Knodel *et al.*, 2017;

Morales-Villanueva *et al.*, 2016) via PR to capture the logic and on-chip memory contents as a way of context saving and then restoring. These methods are designed around the lack of support for capturing/restoring fabric state in FPGA devices. Although these systems provide a non-disruptive way of context switching, the proposed methods have several limitations: First, the methods are specific to a given FPGA device and vendor as they are based on low-level bitstream manipulation; second, the cost of PR is increased since preemptive scheduling requires more frequent (bidirectional) reconfiguration; and third, the states residing on specialized computation resources (DSPs) cannot be captured, limiting the designs in fundamental ways. Due to these limitations, most frameworks offering dynamic scheduling of operators (functional modules) do so without allowing preemption.

3.2.2 Accelerators Near the Storage

Modern data analytics applications, in addition to computational bottlenecks, often face various data movement bottlenecks as well due their distributed nature. These arise in various places in the architecture, e.g., when moving data to the main-memory for processing, from the disk drive, or a network source.

The communication bottleneck between disk and CPU has been addressed in over a decade's worth of database research (Do *et al.*, 2013; IBM/Netezza, 2011; Jo *et al.*, 2016; Weiss, 2012). Today, however, datacenter and cloud architectures often decouple compute and storage resources for better scalability and, while this enables elastic scale-out, it introduces data movement bottlenecks over the network. To alleviate these, computation can be moved closer to the storage nodes, in effect offloading processing into the storage layer.

Figure 3.2 illustrates the benefit of in-storage processing with a query that aims to determine the aggregate spending by customers whose name contains "John", broken down by region. As shown in the figure, if we move all data to the client for filtering, network will be the bottleneck. If we use in-storage processing to offload the filtering and perform the group-by aggregation on the client, the bottleneck can be reduced or removed altogether.

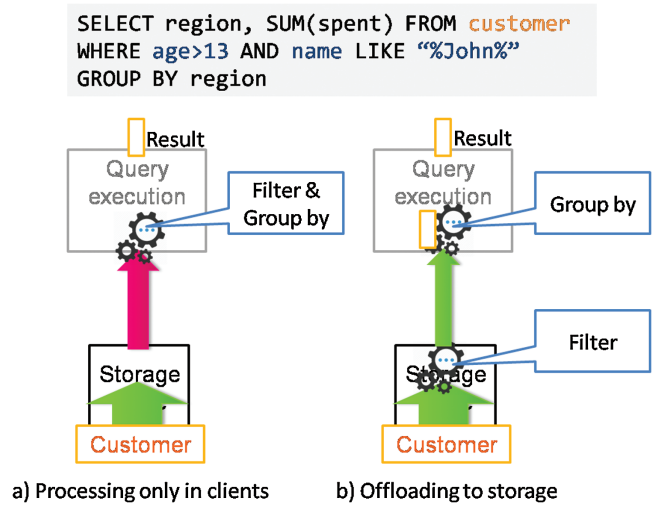


Figure 3.2: Example of offloading filtering for an SQL-style query to the storage layer.

FPGAs present opportunities in the context of near-data processing because many of today's "smart storage" solutions struggle to strike a balance between compute capacity and the bandwidth of the network or storage device. Solutions based on traditional servers suffer from a bandwidth mismatch among storage, CPU, and network. At the same time, they often over provision compute capacity, resulting in power-hungry nodes. Network storage devices (Jun *et al.*, 2014; Klimovic *et al.*, 2016) are better balanced in terms of bandwidth but provide limited processing capabilities. With FPGAs and similar hardware, one can explore an alternative design that offers rich offloading functionality while balancing the bandwidth requirements of storage and network. In addition to the bottlenecks present at the storage layer level, distributed databases and data analytics solutions face communication overheads when having to exchange data across many nodes, such as for shuffle operations or distributed joins. With emerging programmable network interface cards (SmartNICs) there are also opportunities in reducing these kinds of data movement bottlenecks and in this section we look at the design principles behind in-network and near-data accelerators,

how they are integrated with the rest of the software and deep dive into several examples.

Intelligent storage engines such as in IBM Netezza (Francisco, 2011) or the YourSQL project (Jo *et al.*, 2016) have been suggested to support early filtering of data to both increase query performance and reduce energy consumption. Koo *et al.* (2017) evaluated pushing query processing into smart SSDs, concluding that the idea has a lot of potential but that the hardware inside current SSDs is too limited, and that the processor quickly becomes a bottleneck. Thanks to the guarantee of line-rate processing, FPGAs are a possible point in the design space to consider for smart drives.

To extend a database with an intelligent storage engine, the parts of the code base that communicate with the disk driver need to be replaced by code that interfaces with the FPGA. For instance, MySQL features a *pluggable* storage engine architecture that allows multiple storage engines to co-exist in a single database instance. It even allows combining different storage engines. For example, in an existing database, one could migrate several tables, for which query off-loading makes sense, to the smart drive, and leave other tables unchanged. While the migrated tables now would benefit from hardware acceleration, higher-level operations like joins across tables associated with different engines would still be possible.

An other way of interfacing with distributed data storage is through a key-value interface. While not specific to databases, these interfaces are common with services such as Amazon S3. In contrast to smart drives, where the database retains the task of managing the data, in the case of key-value stores, the data management is carried out by the FPGA itself. This means that such solutions will incorporate a data management layer inside the FPGA in addition to the data processing operators.

In recent years we have seen a proliferation of FPGA-based key value stores (KVSs) (Blott *et al.*, 2013; Chalamalasetti *et al.*, 2013; Fukuda *et al.*, 2014; István *et al.*, 2017; Lavasani *et al.*, 2014; Li *et al.*, 2017; Xu *et al.*, 2016) driven by the need for more efficient large-scale data management and storage solutions. In this context, FPGAs are useful because they offer network-bound performance even with small

key-value pairs and near-data processing in a fraction of the energy budget of regular servers. The design of FPGA-based key-value stores has matured throughout the years moving from modest PoCs targeting caching scenarios (Blott *et al.*, 2013; Chalamalasetti *et al.*, 2013; Fukuda *et al.*, 2014; Lavasani *et al.*, 2014) towards more general purpose solutions that could replace KVSs such as Redis or S3 (István *et al.*, 2017, 2018; Li *et al.*, 2017; Xu *et al.*, 2016). Even if the state-of-the-art solutions prioritize different performance metrics or use different storage medium (e.g., DDR, Flash, NVDIMMs), typically they adopt a variation of the pipelined architecture. The processing pipeline is typically split into two parts: one dealing with the keys and a hash table and one that manages the values. The pipeline parallelism ensures that the latency of memory/storage access is hidden (while one operation is waiting for data to come back, others can be processed in previous pipeline stages). Depending on the minimum allowed value size, such a single pipeline can saturate 10 Gbps (István *et al.*, 2017) and even 40 Gbps networks (Li *et al.*, 2017).

3.2.3 Accelerators Near the Network

Until recently, there was a significant performance gap between processor interconnects, i.e., the network between sockets and cores, and LANs, both in terms of bandwidth and latency. With the emergence of specialized high-speed networks (e.g., Infiniband), this gap has been significantly reduced. In particular, Remote Direct Memory Access (RDMA) has been shown to make the gap almost negligible (Barthels *et al.*, 2015; Zamanian *et al.*, 2017) enabling new distributed designs that were not feasible just a few years ago.

So called Smart Network Interface Cards (SmartNICs) have emerged in recent years, incorporating FPGAs. For instance, Microsoft Catapult (Putnam, 2014) deployed an FPGA-based device as a bump in the wire in front of a NIC (Figure 3.3(C)) to offload virtual network functions for cloud tenants. The FPGA is managed over a PCIe-based interface, which allows the cloud provider to re-purpose it as an on-the-side accelerator if the network offloading is not necessary. It has to be pointed out that in such a design, if one would want to offload

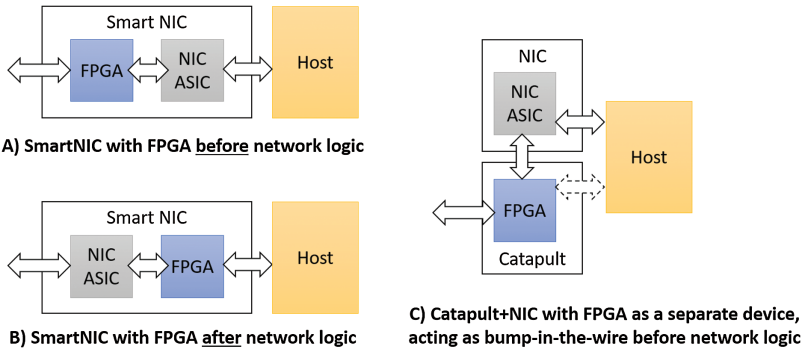


Figure 3.3: Network interface cards can incorporate FPGA-based acceleration in several ways.

application-specific network-level processing, parts of the network stack handling would have to be reimplemented inside the FPGA, since it is located *before* the NIC proper (see Figure 3.3(A)).

A representative design for an alternative approach is NICA (Eran *et al.*, 2019): the FPGA is deployed *after* the NIC ASIC and before the CPU (see Figure 3.3(B)). This allows it to work on packets that have been already processed in the Ethernet stack for instance. Such SmartNIC designs fit also very well with RDMA-based designs, since the FPGA, to some extent, is agnostic to the network protocol handled by the NIC. Once limited to specialized networks such as Infiniband, RDMA has become a commodity with the widespread adoption of 40 G Ethernet and the introduction of RDMA over Converged Ethernet (RoCE), leading to large-scale deployments in data centers (Dragojević *et al.*, 2015). Network bandwidth is expected to increase, with 100 G and 400 G Ethernet being rolled out. The latter already approaching the memory bandwidth available in Intel x86 server-grade systems. Such a change has led to the design of new distributed, shared memory systems that take advantage of the features of RDMA (Dragojević *et al.*, 2014; Kalia *et al.*, 2014; Mitchell *et al.*, 2013; Zamanian *et al.*, 2017).

In the scope of databases and data analytics, the ability to directly access remote memory through RDMA at a high bandwidth and low latency has made distributed operators more viable, especially in the

case of in-memory databases where previously the amount of memory was limited to a single machine. Yet, in data processing, much of these performance gains can be lost if the data moved (regardless of how fast) is discarded because it is not needed (e.g., through selection, projection, filtering, etc.). Thus, it would be desirable not only to use RDMA but also have the possibility of pushing filtering and sorting functionality down to the NIC to make the data transfer enabled by RDMA smarter.

3.3 Database Integration Case Studies

Depending on the level of integration of FPGA-based accelerators with the query execution engine, we can differentiate between designs that access the FPGA using User Defined Functions (UDFs) and those that create physical operators running on the FPGA as part of a query plan. While the former approach requires less engineering and adds no complexity to the database, it also defines the data access patterns the accelerator can perform. In databases such as PostgreSQL, for instance, UDFs expose only a tuple-by-tuple scan interface with no option for random lookups. There are databases that pass entire columns or tables to the UDF, such as MonetDB, but it is typically not possible to implement operations over multiple tables. If using a deeper integration of the FPGA in the database, there is more flexibility in data access but the programming effort is also greater. Ibex (Woods *et al.*, 2014) is an example of such deep integration where the FPGA is used to offload operations at the storage engine level and therefore is integrated as a custom storage engine with its special scan implementation as part of MySQL.

Apart from the data access aspects, the choice of integration also defines who is responsible of managing and exposing the acceleration functionality. For UDFs, this responsibility falls on the user/OS, whereas in the other case, the accelerator is under the management and full control of the database.

In the following we present an overview of frameworks that offer management and scheduling functionality inside an FPGA deployed in different locations of the server architecture.

3.3.1 Co-Processor: DoppioDB

DoppioDB (Alonso *et al.*, 2019; Kara *et al.*, 2020) is an FPGA-accelerated research database built on top of MonetDB. It's main goal is to extend the capabilities of MonetDB with machine learning operators that, due to their compute intensive nature, can significantly benefit from specialized hardware. As shown in Figure 3.4, DoppioDB runs on an Intel Xeon+FPGA platform. The FPGA, an Intel Arria 10 device, has cache coherent access to the main memory of the machine through two types of interconnects: QPI, for low latency, cache-line granularity accesses, and PCIe for bulk accesses. Cache coherence is guaranteed, however, over both physical interconnect technologies and the Intel Framework takes care of scheduling memory accesses over these different interconnects in a transparent manner.

DoppioDB incorporates Centaur by Owaida *et al.* (2017) as a management layer that exposes operators on the FPGA as hardware threads to the software. The abstraction allows software to create new hardware threads, similarly to POSIX threads, passing them a pointer to a function to be their start routine. In the case of the hardware threads, these functions are stubs for the functionality on the hardware side and simply define the number and types of arguments the hardware operator excepts. Once started, the threads can be waited for and joined when they finish execution on the FPGA. Centaur uses shared memory data structures and its own FIFO buffers on the FPGA to queue invocations

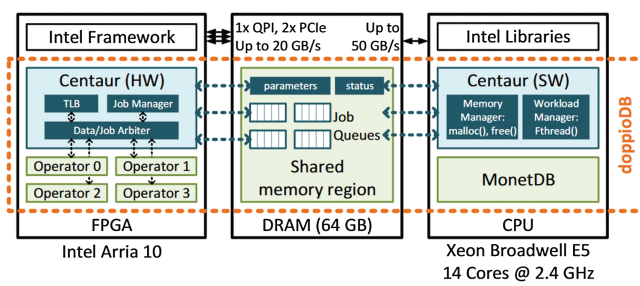


Figure 3.4: DoppioDB extends MonetDB with UDFs that offload computation to the FPGA. Thanks to the cache-coherent access to the main memory of the machine from the FPGA-side, the communication overhead is minimized.

and match them to the operators deployed on the FPGA. Depending on its configuration, the number of operators can vary, but in DoppioDB, four reconfigurable regions are used to allow for four operators.

DoppioDB is built on top of MonetDB, a column oriented relational storage engine. Its functionality has been extended with hardware-based operators using the user-defined function (UDF) interface of the database. The choice of UDF is fitting for the DoppioDB use-case, that is, for offering new types of machine learning operators for the database. These are typically ran towards the top of query plans and do not require deep integration with the query engine. One important difference between traditional analytics in databases and novel machine learning workloads is that while the former benefits greatly from a column-oriented layout, the latter often requires to operate on rows (tuples). For this purpose, DoppioDB includes a module for column-to-tuple transformation on the FPGA side. This module performs batched reads to several columns and combines them in an on-chip buffer. The benefit of performing this operation on the FPGA side, as opposed to on the CPU, is not increased performance but, instead, the fact that caches are not polluted unnecessarily. The module is able to perform transformations at peak memory bandwidth on the FPGA side and, for this reason, is transparent to the actual operators implemented on the FPGA.

3.3.2 Smart Drives: Ibex Case Study

Ibex (Woods *et al.*, 2014) is an example of such a smart drive in which the FPGA has a direct SATA link to an SSD (Figure 3.5) and is integrated with MySQL as a custom storage engine.

The hardware engine transfers data to/from the SSD via a SATA core instantiated on the FPGA. Thus, the hardware engine has to operate on blocks of raw data. In answering block requests for an upstream system, it is not immediately clear how an accelerator could filter or modify individual tuples within those blocks.

The semantics of any filtering or aggregation task have to be expressed on the tuple level, which is why these tasks are typically handled above the storage engine. To let the hardware engine perform query

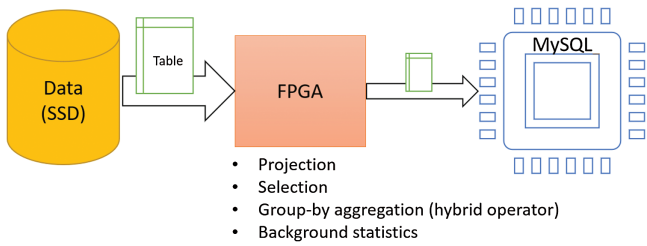


Figure 3.5: Ibex showcases several operations that can be performed on the data as it is read from storage with the goal of reducing the number of tuples that arrive at the CPU.

processing tasks, the mismatch problem between block-oriented disk access and the tuple-oriented semantics of these tasks needs to be solved. Since Ibex is designed for hybrid query processing, the Ibex hardware supports both *block-level* and *tuple-level* access to base tables. This is achieved by incorporating in Ibex a parser module that can transform database pages internally to tuples as they are retrieved from the drive.

When sub-queries are offloaded to the FPGA, the hardware switches to a tuple-based interface, i.e., disk blocks are parsed in hardware and processed by a parameterizable query pipeline, and the result tuples are forwarded to the host system as a sequence of tuples, which are directly fed into the query evaluation pipeline of the database. For a Volcano-style execution engine, such as the one of MySQL, tuples provide the right abstraction.

In all other cases, data is accessed in the conventional, block-oriented mode, i.e., using a raw data path. This includes not only un-predicated table scans but also any operation that does not require hardware acceleration can use block-level access just like an off-the-shelf system would, e.g., update operations, maintenance tasks (backup/recovery, etc.), or index-based plans.

Ibex implements three main operations that can be used directly in query plans, namely, Projection, Selection and Group-by Aggregation. As seen in Figure 3.6, tables read from the disk drive go through these transformations and, in the process, decrease the result set size. As explained later, since the group-by aggregation is a blocking operator whose results might not fit on the memory of the FPGA, Ibex provides a

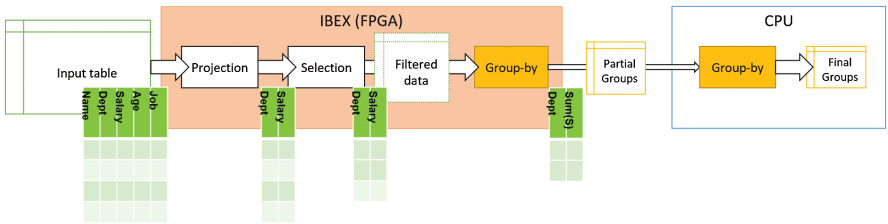


Figure 3.6: Ibex implements three main data processing elements which aim to reduce the data leaving the FPGA.

hybrid implementation that uses the CPU to compute final groups based on partial aggregates from the FPGA. In addition to these operators, it has been also proposed to implement additional logic that can refresh table statistics as a side-effect to full table scans (Istvan *et al.*, 2014).

In this monograph we explore in more detail the implementation of the Aggregation module in Subsection 4.3 as an example of a module that respects all three design principles of near-data processing elements.

3.3.3 Smart Storage Nodes: Caribou Case Study

Caribou (István *et al.*, 2017) implements a key-value store accessible over the network using a network/application co-design approach on an FPGA. It is designed such that its internal data structures and pipelines can handle virtually any mix of random-access operations that can be received over a 10 Gbps network link. As a result, Caribou's performance reaches over 11 million requests/s on regular TCP/IP connections, which is almost 3× higher than what can be achieved with software over the same commodity network. Caribou, at the same time, offers 3–5× lower power consumption per node when compared to regular servers. To mitigate the data movement bottleneck between storage nodes and the rest of the application, it can offer different ways data can be filtered inside the storage node while matching the streaming read bandwidth of the underlying storage.

Caribou is open source[1] and, thanks to its modular design, new functionality can be easily added. It is being actively extended with

[1] https://github.com/fpgasystems/caribou.

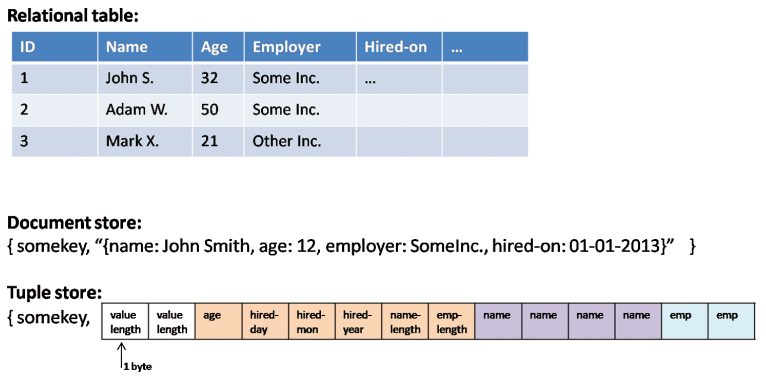


Figure 3.7: Two ways of mapping relational tuples to a key-value store.

functionality such as performance- and data-isolation for multi-tenant deployments (István *et al.*, 2018), as well as, better integration with clients written in Golang and Python (Kuhring *et al.*, 2019). As a result, it can be used as a platform for prototyping near-data processing solutions and to explore integration with different applications.

For database use-cases, the data stored in Caribou could be represented in two different ways (Figure 3.7), both of which are compatible with the processing modules on the FPGA:

- Documents – It is possible to express database tuples as JSON documents, explicitly encoding the value of all columns (attributes) in the original table.
- Binary tuple – This representation is closer to how tuples are stored in traditional database engines. In this case the schema of the tuple is implicit and the clients have the knowledge of which column spans which bytes. Variable length columns are assumed to be at the end of the value. The length of each value is encoded in its first two bytes.

Caribou exposes the following operations to clients:

Insert – Stores a value in memory, under a key specified in the request. It can be performed both in a replicated or a node-local way. The response encodes the success/failure state of the operation.

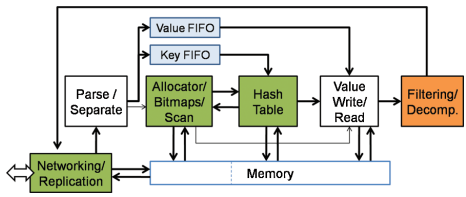


Figure 3.8: Modules in Caribou and their connections.

Read – Retrieves a value based on a key. If the key is not present in the system it will return failure.

Conditional Read – Retrieves a value corresponding to the key in the request and evaluates a predicate on it. The parameters for the predicate evaluation are provided with the request. If the predicate matches, the value is returned. Otherwise a failure header (16 B) is returned.

Delete – Removes a key's entry from the hash table and frees the memory holding the value.

Update – Given a key and a value, updates value stored in the system. This operation does not allocate memory if the new value size is the same[2] as the existing one. Updating a non-existent key will fail.

Scan Query – Scans internally over all[3] values in the storage and evaluates a predicate on each. Only the values that fulfill the predicate are sent to the client.

Figure 3.8 depicts the high level architecture of a single Caribou node. The logic is centered around a hash table that (1) can handle reads and writes in constant time, so that the storage has predictable performance, and (2) can store variable sized or non-ordered keys, so that Caribou can be used in different use-cases. A Cuckoo hash table is used because it provides constant time lookups. Insertions and deletions

[2]To be more precise, as long as the new value and the old value are the same multiple of 64 Bs.

[3]The scan operation retrieves all values stored in the same tablespace. In our current implementation there is only a global tablespace, but as later explained the system can be extended to support multiple tablespaces. This means that scans on one relation will not be influenced by the size of other relations.

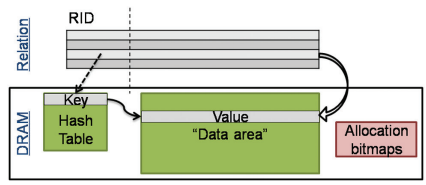


Figure 3.9: Mapping tables to storage in Caribou.

might need variable time to be carried out but our implementation can finalize these operations in the background while still providing constant time answers to clients. In order to make sure that memory space is used efficiently, the location of hash table entries and values needs to be completely decoupled. Figure 3.9 depicts this decoupled design, where the hash table holds pointers to the so called Data Area that holds values. Memory for the values is allocated by a separate module. The same module is responsible for carrying out scans as well. Caribou implements a slab-based allocation strategy (similar to the one found in memcached).

Near-memory processing is carried out by the module at the end of the pipeline. It receives values retrieved either as a result of a lookup or a scan operation. This module supports selection on both structured and less structured data. Thanks to hardware parallelism, selection can be combined with other transformations, e.g., decompression. Functionality could be extended in the future with, e.g., group-by aggregation (Woods *et al.*, 2014), with statistics (Istvan *et al.*, 2014), skyline queries (Woods *et al.*, 2015) or complex pattern detection (Woods *et al.*, 2010).

The networking component of Caribou is taken from earlier work on TCP/IP stacks in hardware (more details can be found in Sidler *et al.*, 2015, 2016). An important feature of the networking stack is that it has low latency (<10 μs RTTs), can saturate 10 Gbps even with small packets, and can support up to thousands of connections at the same time.

Zookeeper's atomic broadcast (István *et al.*, 2016; Junqueira *et al.*, 2011) is used to replicate data from the node that acts as master copy to the replica nodes. Writes are directed to the master copy and they are successful if a majority of nodes perform them. Reads can be directed

to any node, but an external commit manager or coordinator (Loesing *et al.*, 2015; Plattner and Alonso, 2004) should be used by the processing nodes to make sure that they are not reading from nodes with stale data (e.g., after recovering from a network partition). The implementation in hardware comes from previous work (István *et al.*, 2016). This algorithm has been chosen because it allows for pipelining of proposal sending and taking commit decisions, fitting well with hardware execution. Furthermore, low latency TCP/IP networking and reliable response times of other FPGA nodes help to maintain high performance while using ordinary sockets instead of special purpose networking or dedicated links.

4

Designing Accelerators with FPGAs

The focus of this section is the internal architecture of different accelerators in the database context and begins with a short summary of the recommendations for designing FPGA-based acceleration. After that, we discuss in detail the following types of operators:

1. Core SQL operators:
    - Filtering with "LIKE" and Regular Expressions – Subsection 4.2
    - Group-by Aggregation – Subsection 4.3
    - Filtering with "WHERE" predicates based on comparisons – Subsection 4.4
2. Machine Learning operators:
    - K-Means clustering – Subsection 4.5
    - Stochastic Gradient Descent – Subsection 4.6
3. Distributed SQL operators:
    - Distributed Join acceleration – Subsection 4.7

4.1 Design Guidelines

In this section we highlight the design principles that guide successful database accelerator designs. These principles emerged from studying the state of the art related work, as well as, the projects the authors contributed to.

Overall, the main purpose of FPGAs is to accelerate computation offloaded from CPUs but depending on the physical location of the FPGAs, the designs will be subject to different optimization conditions. When deployed as co-processors or on-the-side accelerators, they need to be explicitly invoked by a CPU which means that they need to provide high factor speedups to be worth deployment. In contrast, when deployed in the data path, compute offload translates to potential data reduction, alleviating this way data movement bottlenecks. In such scenarios, the network or interconnect rate will also define an upper bound on the useful throughput inside the FPGA.

4.1.1 Guidelines for Co-Processors and On-the-Side Accelerators

A1 **Target compute-intensive operations**: Not surprisingly, the use of FPGAs and similar accelerators pays off when used to offload operations that are otherwise expensive to compute on the CPU.

A2 **Minimize data transfer costs**: The benefit of any accelerator will be reduced by communication costs. Therefore it is important to adjust the way the database communicates with the accelerator to the characteristics of the interconnect. In a PCIe-based setting, for instance, streaming large amounts of data is more efficient than accesses at tuple granularity. In addition to the data movement cost, the integration with the database engine can lead to further overheads if memory copy, reformatting and transformations are not avoided.

A3 **Don't try to do everything in hardware**: Research projects have repeatedly found that the benefit of FPGAs is highest when they offload functionality that does not have a complex control

flow (no excessive logical branching). This means that algorithms with many corner-cases could be slower on an FPGA than on a CPU – what's more, including the logic to handle corner-cases on the device might even reduce the performance of the common case. Therefore, it is important to design accelerators that can gracefully transition to software-based execution, or apply post-processing, when necessary, in an efficient manner.

4.1.2 Guidelines for Near-Data and In-Data-Path Accelerators

B1 **Place offloading functionality as close as possible to the data source**: Not surprisingly, in order to reduce data movement bottlenecks, the offloading functionality has to be placed at locations where data from a high bandwidth device is transferred through a lower bandwidth connection. This way, if the FPGA can reduce the data size by offloading filtering, transformations or aggregations, it can directly reduce the bottleneck imposed by the lower bandwidth side.

B2 **Design computation that is running at line-rate**: As opposed to on-the-side accelerators where the goal is to make hardware modules as fast as possible, when designing for data-movement related use-cases, it is only required to reach the bandwidth of the input/output medium. Nonetheless, it can be challenging to design operators that guarantee line-rate regardless of the workload parameters and we will discuss examples later in this section.

B3 **Design with runtime-parameterizable circuits**: It is important to design the FPGA functionality such that it ensures that a wide range of workloads and operations can benefit from the "bump in the wire" processing. The overhead of reprogramming the FPGA very often is not feasible in the near-data processing case and having functionality that can be re-purposed at-runtime by setting on-chip registers or memories is preferable.

4.2 Core SQL: Like and Regular Expressions

Filtering on string fields is an important operation in SQL but when searching more complex expressions than substrings, CPUs can become compute bound. In general, databases implement more complex filter patterns through regular expressions (Aho and Ullman, 1992). In this section we focus on a subset of the features of POSIX Regular Expressions[1] that are most relevant for database use-cases, including characters, ranges, sets, choices, repetitions and wild cards, but excluding recursion and other advanced features.

In software regular expressions are usually translated to deterministic finite state automaton (DFA) (Aho and Ullman, 1992). The advantage of DFAs is that, by construction, only a single state can be active at any time and, determined by the next parsed symbol, the choice of next state is also deterministic. In software this means that with each input symbol, only the transitions of the current state have to be considered (usually a small number), which helps with cache locality and reduces the compute overhead. The main drawback of using DFAs, however, is that removing non-determinism from the pattern can require in essence listing all possible states, which leads to a so called "state explosion", yielding very large DFAs.

Figure 4.1 shows a simple regular expression, `a.*(b|c)d`, translated to an NFA. Even this simple example shows non-determinism in that at the same time both state S_1 and S_2 might be active. In contrast, DFAs have additional states to avoid nondeterminism, which leads to the already mentioned state explosion problem (Woods *et al.*, 2010).

Non-deterministic finite state automaton (NFA) avoid the state explosion problem but evaluating transitions from multiple active states can become compute intensive in software. Furthermore, locality is also lost because different parts of the NFA might need to be looked at for the same input. For this reason, NFAs are less often used in software, but are a good match for hardware. This is because in FPGAs parallel execution comes for free – on the other hand, space for storing the automaton is more scarce.

[1] https://en.wikibooks.org/wiki/Regular_Expressions/POSIX-Extended_Regular_Expressions.

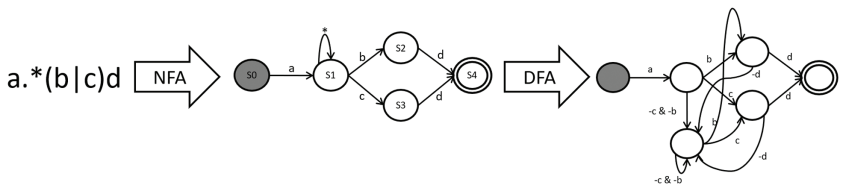


Figure 4.1: When mapping regular expressions to Nondeterministic Finite State Automaton (NFA), the number of states is small but several of them could be active at the same time. Deterministic Finite State Automaton (DFA) introduce additional states that ensure that only one is active at a time but as a result increases the complexity significantly.

On FPGAs there is rich related work for implementing NFAs as circuits (Bispo *et al.*, 2006; Nakahara *et al.*, 2011; Teubner *et al.*, 2012; Yang *et al.*, 2008), targeting mainly networking use-cases, such as packet inspection. In that context, the main challenge is how to efficiently combine many different regular expressions, all known apriori, in a way that they can all be matched against the same input. In database settings, instead, the focus is on answering a single regular expression, not known apriori, aiming at much higher data rates than what are common in networking related work (10–25 GB/s vs. 1–10 Gbps).

In the following we present one possible design for FPGAs that (1) ensures constant rate processing, regardless of the contents of the expression or input data and (2) enables runtime parameterization of the circuit (Sidler *et al.*, 2017), that is, the core processing logic remains unchanged on the FPGA but by changing the contents of several registers at run-time, it can be configured to match any user-defined expression that fits within its complexity limits.

Parametrization. Since the time to recompile hardware circuits can take minutes or even hours, in database use-cases it is beneficial to design with parameterization in mind. The Regular Expression Processing Units (PUs), described in Sidler *et al.* (2017), demonstrate one way of achieving flexibility. As can be seen in Figure 4.2, a PU is like a "skeleton NFA". It consists of Character Matchers and a generic State Graph. These components are parametrized by the three sets of registers: Tokens, Triggers, and State Transitions. The number of characters and states is

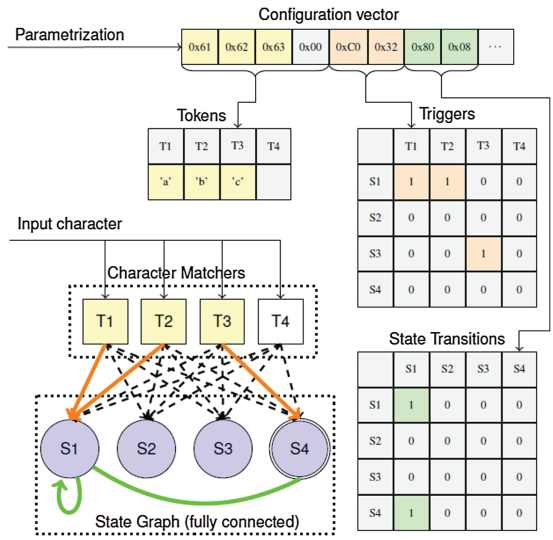


Figure 4.2: Internal structure of a PU with resources for matching four characters and four states. The configuration vector at the top parametrizes the registers (Tokens, Triggers, and State Transitions) on the right, implementing the regular expression `(a|b).*c`.

fixed at compile time and they limit the space of regular expressions that can be mapped to a specific deployment, either by the length of an expression or its complexity.

Figure 4.2 also includes an example configuration vector encoding the example expression `(a|b).*c`. The configuration vector contains the characters, the bits representing the Triggers and State Transitions. Apart from that, it also contains flags which indicate if two Character Matchers are coupled together to evaluate a range instead of two separate characters. There are three parameter registers: Tokens, Triggers, and State Transitions. They are parametrized through the above mentioned configuration vector, and as a result the deployed PU implements the regular expression `(a|b).*c`. The Tokens parameter register defines the characters to be matched in the Character matchers, the Trigger defines which token triggers which state, and the State Transition register defines which state triggers which state. Being able to parametrize at

runtime Tokens, Triggers, and State Transitions results in high flexibility when mapping regular expressions to the hardware.

Compacting Expressions to Fit on the FPGA. The main goal of the above described PU design is to deliver the same throughput as a hardcoded NFA while being able to evaluate any regular expression that fits into the deployed circuit (number of characters and states). To achieve these two seemingly conflicting goals, a structure corresponding to complete graph is created, where each state is a node in the graph and the edges which represent state transitions can be enabled or disabled at runtime through the State Transitions registers. The choice of which token triggers which state can be imagined as a bipartite graph, configured through the Triggers register. The circuit is implemented as synchronous logic. This means that all states will evaluate their inputs in parallel, make a decision, and then update their outputs all at once, based on a common clock. The same holds for the Character Matchers which update their outputs in sync with the same clock signal. Figure 4.2 shows how the State Transitions register is configured for the example expression `(a|b).*c`. The start states are implicitly defined by not having any activating edge, the end state is explicitly defined as the state with the highest index. Once it becomes active, a signal indicating a match is activated, this signal also contains the match location as a 16 bit unsigned integer.

To achieve flexibility, the State Graph is implemented as a fully connected graph. This however requires significant hardware resources as the graph size increases, since each node needs to able to propagate their signal to all other nodes within a single clock cycle. To reduce the required size of the State Graph, it is possible to take advantage of the fact that in database use-cases filtering expressions will often contain natural-language related terms. Therefor, it is possible to match on sequences of characters in the regular expression instead of individual characters. Extracting these sequences allows building a compacted NFA on so called *tokens* instead of individual characters. Inside a PU is a series of Character Matchers, as shown in Figure 4.2, that can be chained together to match a sequence of characters.

Complex Expressions and Hybrid Execution. In general, more complex regular expressions translate to NFAs with more states, and longer expressions require more character matchers. For instance, the expression `(Blue|Gray).*skies` translates to three NFA states and 11 characters. Even though the PUs are runtime parameterizable, the number of supported states and characters is decided at compile time and have an upper bound based on the size of the FPGA chip. As a result, it could happen that a regular expression in the workload either requires more states or characters than available and therefore cannot entirely be mapped onto the PU. To resolve this issue and still benefit from hardware acceleration, the design principle of "trying not to do everything in hardware" can be applied by combining hardware and software execution to evaluate the regular expression. It is possible to split regular expressions into two pieces at a suitable point, e.g., at the occurrence of a wildcard `'.*'`. If one of these pieces fits into the regular expression matcher on the FPGA, we can pre-process all tuples on the hardware, and post-process the ones that matched (the remainder of the expression) on the CPU.

Data Parallel Execution. FPGAs can offer high levels of parallelism and, in the case of the regular expression matching PUs, this can be used to increase throughput. Figure 4.3 shows how several PUs can

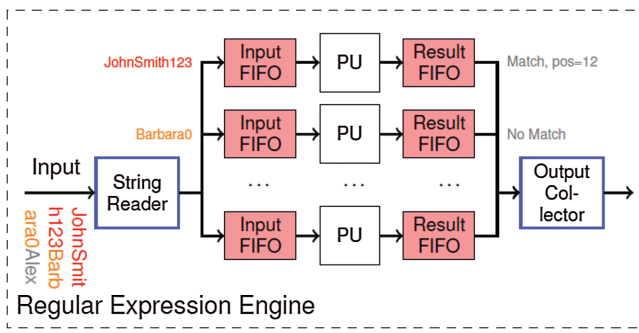


Figure 4.3: To reach high throughput, Processing Units (PUs) are grouped into an Engine. They operate on a scatter-gather fashion on the strings for the query.

be assembled into an Engine that works on different string inputs in parallel. Such architectures are very common in FPGA design.

The Engine is composed of several modules that form a pipeline. They work in parallel and communicate through cache-line wide buses. The first module in this pipeline is the String Reader which fetches the strings from memory, belonging to different tuples. The strings are parsed, aligned to the internal 512 bit data bus and forwarded in round-robin fashion to the cache-line wide Input FIFOs. At the end of the pipeline, the Output Collector collects the 16 bit match indexes from the Result FIFOs in round-robin fashion to guarantee that the results are in the same order as the input. Specifically, it collects 32 results and writes them into a cache line of 512 bit. These cache lines can then be written sequentially to the result column. The pointer to the result column is also provided as a runtime parameter.

Integration. The regular expression matching operator could be integrated with a wide range of software applications and, in the following, we detail how it has been integrated as part of DoppioDB (Subsection 3.3.1). Overall, there are two aspects that have to be considered: one is the right sizing of the circuit in terms of a performance target, and the other is the workflow required for offloading expressions to the FPGA at run-time.

In terms of performance, there are several variables to take into account. First of all, each PU can process one input character (one byte) per clock cycle. The clock frequency can be chosen by the FPGA programmer but cannot be larger than the F_{max} of the PU, that is the maximum frequency at which all signals can propagate safely without timing uncertainties. Figure 4.4 illustrates the performance/logic resource consumption of the Regular Expression Engine depending on the number of parallel PUs and the choice of clocking frequency. The PUs have been configured in this example with eight states and 32 characters. The metric for resource consumption is in adaptive logic modules (ALMs) on the FPGA. The figure shows that for the same nominal throughput, the resource consumption of the PUs can be halved by doubling the clock. The resource overhead for the String Reader and

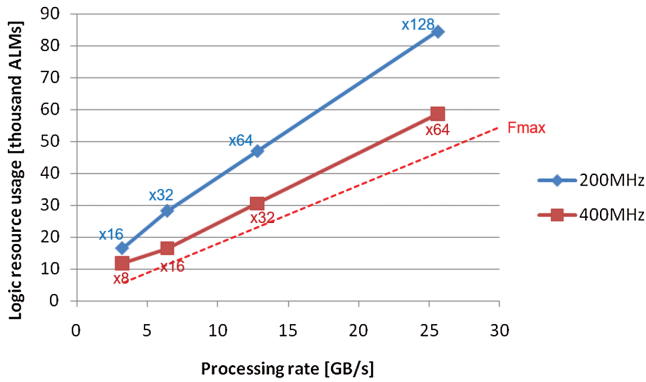


Figure 4.4: When designing operators for FPGAs, in addition to the Throughput/Frequency trade-off, if they support data-parallel execution, there is also a Throughput/Resource consumption trade-off to be considered.

Output Collector is constant, and when clocked at 400 MHz, these can handle the memory bandwidth in the Xeon+FPGA machines (20 GB/s).

The other important part of integration is the workflow required for parameterizing the Engine at runtime. Figure 4.5 depicts the two different flows, one done at design/compile time, and one done at run-time. In the former, the FPGA designer decides the internal parallelism of the Engine, defining this way the nominal throughput of the accelerators, and the size of the skeleton NFA inside the PUs. After the circuit has

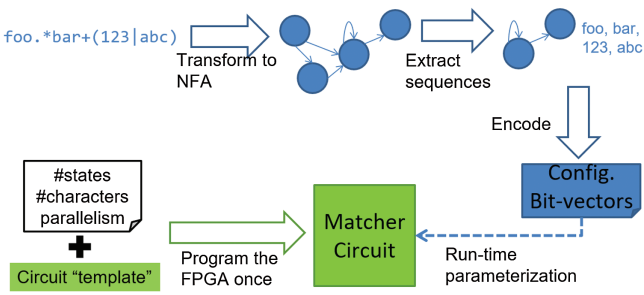


Figure 4.5: The Regular Expression Engine is configured with compile-time parameters and deployed on the FPGA. At runtime, regular expressions in user queries are translated to the internal NFA format and the parameter registers on the FPGA are updated.

been compiled it can be either statically or dynamically deployed as an operators. At run-time, filtering expressions that are to be offloaded to the FPGA, undergo the following steps: (1) they are transformed into an NFA, (2) sequences of characters are extracted from this NFA, reducing its state count, and (3) the resulting NFA and Tokens are encoded as bitvectors that are used to update the registers within PUs. This operation takes in the order of microseconds, orders of magnitude lower than partial reconfiguration and incomparably lower than the cost of compilation.

4.3 Core SQL: Group-by Aggregation

One of the innovations in Ibex (Subsection 3.3.2) was the inclusion of a group-by component, in addition to the projection and filtering ones. This component handles *grouping* using a specially designed hardware *hash table* on the FPGA and, as a result, reduces the data amount between disk and CPU for an even wider range of queries. A typical group-by query is illustrated below.

```
SELECT col2, col7, MAX ( col1 ), MIN ( col1 )
  FROM table                                        (Q1)
group-by col2, col7;
```

The grouping criteria that defines the individual groups is specified in the group-by clause. This can be a single column or a combination of several columns, as in Q_1. In the following, we refer to this grouping criteria as the *group key*.

In the presence of a group-by clause, only columns that are part of the group-by clause can be projected without an aggregation function, all other columns need to be part of an aggregate since for every group there will be a single result tuple. Notice that the same column can appear in multiple aggregates, e.g., `MAX ( col1 )` and `MIN ( col1 )` in Q_1.

In the following, we present the design of the aggregation module used in Ibex in more detail. Since this module is situated on the data path between disk and processor, it has to ideally operate at a fixed bandwidth, corresponding to that of the disk. Furthermore, given the limited resources on FPGAs, it is important to ensure that, in case not

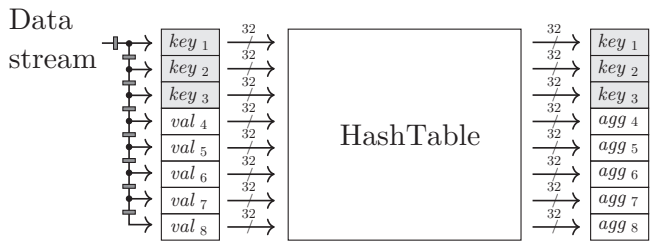


Figure 4.6: High-level architecture of the group-by component. Data is loaded in a pipelined fashion (pipeline registers → ▯).

all groups fit on the FPGA's memory, the system can gracefully fall back to a software operator.

Design Overview. The high-level design of the group-by component is depicted in Figure 4.6. After a projection and selection stage, relevant data in the form of key-value pairs is loaded into a wide input buffer (here, 256 bits wide), which is divided into multiple 32-bit slots. The keys can be formed from appending values from several database columns together, and the same applies to the values.

At runtime, a bit mask determines how many slots belong to the group key and how many slots are used for aggregation. In Figure 4.6, the first three slots are used for the group key and the remaining five for aggregation. Notice that we do not require all slots to be used. It is perfectly valid to use, say, only the first two slots, one for the group key, and the other for a single aggregate. Our design exhibits flexibility not only to support combined group keys, as in Q_1, but also group keys on columns that are wider than 32 bits. For group keys that are smaller than 32 bits, we simply add a padding of zeros to the 32-bit slot.

Besides assigning slots for *grouping* and *aggregation*, each slot can be mapped to a column at runtime. Furthermore, also the type of aggregation (`COUNT, SUM, MIN, MAX`)[2] can be set for every slot. Data is loaded into the slots in a pipelined fashion. This allows us to compute multiple aggregates on the same column. Moreover, even the order

[2]For the average, we compute `SUM` and `COUNT` in hardware, and perform the division in software.

of aggregates can be specified already in hardware, i.e., reordering in software is not necessary. Another advantage is the scalability of the technique. In Ibex, we set the buffer width to 256 bits (eight slots), because it matches the DRAM word-width of the prototype design but it is possible to increase the width at compile-time without negative side-effects on performance.

The hash table operates by performing atomic "probe and update" operations. The group key slots are used to probe the hash table, upon which the hash table returns an entry the size of the input buffer, containing the group key, as well as the current running aggregates. All aggregates are updated in parallel and the entry is written back to the hash table.

In order to provide a predictable throughput independent of table contents, hash collisions are detected but not resolved in hardware. When a colliding tuple is detected, it *bypasses* the hash table and is forwarded to the host. Thus, during query processing *bypassed* tuples of the form $\{key_1, \ldots, key_i, val_{i+1}, \ldots, val_n\}$ are forwarded to the output buffer. After the entire table has been read, the hash table contents are flushed and aggregated tuples of the form $\{key_1, \ldots, key_i, agg_{i+1}, \ldots, agg_n\}$ are forwarded to the output buffer.

An alternative design would be to resolve collisions of the FPGA by either chaining keys in buckets of the hash table or rehashing. Both approaches introduce additional processing that, in the case of the former, increases the access latency for large buckets, and in case of the latter, interrupts common case processing for lengthy intervals. Unpredictable behavior could lead to slowdowns in queries and discourage the use of the accelerators – hence, the decision to not resolve collisions has not been taken in order to simplify the FPGA logic but rather to ensure that it behaves in a predictable manner and it is possible to fall back to software more gracefully.

If there are bypass tuples, an additional group-by and aggregation step is required in software. This can be performed naively by rewriting the original query to wrap the group-by operation in an additional one, the inner executed in Ibex, the outer by MySQL's native operator. Experiments confirmed, however, that such an approach introduces significant overhead when there are many groups or many bypass tuples,

mainly due to the cost of crossing the interface between the storage engine and the rest of the query execution pipeline. To avoid the high cost of running the outer query, Ibex implements an aggregation step inside the software side of the storage engine to deal with bypass tuples directly.

Fully-Pipelined Hash Table. At the heart of the group-by component is a hardware implementation of a hash table. While hash tables on FPGAs have been studied in prior work (Dhawan and DeHon, 2015; István *et al.*, 2015; Thinh *et al.*, 2007), the hash table design is specially tailored at supporting group-by queries at line-rate. The hash table allows the execution of group-by aggregation in a single pass, without having to first sort the input data. Figure 4.7 illustrates how this works: ① First, the group key is hashed. ② The bits of the hash value (or a subset thereof) serve as the memory address, and the corresponding memory word is read from that address. ③ A special flag indicates whether a particular group is being processed for the first time. If this is the case, an *insert* is performed, i.e., completely overwriting the memory word that has been just read. Otherwise, an *update* is performed on the running aggregates in the memory word. ④ Finally, the processed memory word is written back to memory.

As mentioned previously, in in-data-path deployments, it is important to achieve line-rate performance for operators. For the group-by aggregation inside the FPGA in Ibex this translates to SATA II rate of 300 MB/s (or 16 bits per clock cycle at 150 MHz). However, to achieve this, there are several challenges. First of all, constant time lookup is only guaranteed if there are no *hash collisions*, i.e., if no two groups

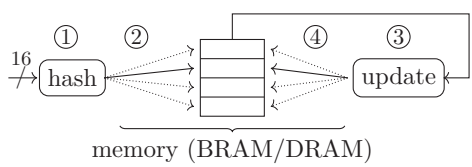


Figure 4.7: Abstract view of our hash table with the four fundamental operations: ① *hash*, ② *read*, ③ *update*, and ④ *write*.

map to the same memory address. Instead of *stalling* the input stream to perform collision resolution on the FPGA, colliding tuples are sent to the CPU as *bypass*. The hash table implements a first-come-first-served policy, that is, each memory address will "belong" to the first inserted group key. Getting rid of collision resolution, however, only partially solves the problem since the four steps depicted in Figure 4.7, each require at least one clock cycle. Performing these four steps in a sequential way would again cause stalling of the input stream. Therefore, it is required to pipeline the hash table, similarly to the designs mentioned already in Subsection 3.2.2, but at a much finer granularity.

The pipelined version of the hash table is displayed in Figure 4.8. The four stages *hash*, *read*, *update*, and *write* are executed in parallel for different tuples in the pipeline. The number of clock cycles that can be spent in each stage while ensuring line-rate processing depends on the size of the tuples in the database table – the larger the tuples the more clock cycles can be spent. The smallest tuple that Ibex supports is 32 bits wide. This means that in the most extreme case, the hash table has only two clock cycles for every stage.

Hashing of group-by keys happens using a *multiplicative* hash function that is itself fully pipelined, and consumes 16 bits per clock cycle matching the input rate. Using BRAM, reading and writing each take one cycle (the hash table can be moved to DRAM for increased capacity, this option is discussed next). Finally, updating all the aggregates of a tuple is done in parallel and can also be handled in a single cycle. Thus, with BRAM line-rate processing is guaranteed. Nevertheless, care needs to be taken when two tuples of the same group follow closely after each other since this could lead to *data hazards*.

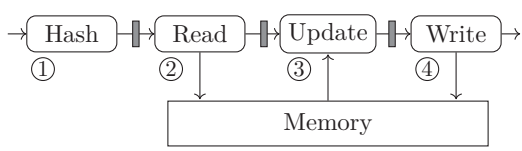


Figure 4.8: Pipelined hash table hides latencies, allowing concurrent processing of four tuples.

When two consecutive tuples access the same memory location because they belong to the same group, it is essential that the second tuple sees the modifications of the preceding one, otherwise updates will get lost. The design in Figure 4.8 cannot guarantee this when tuples are small.

One approach is to detect situations where potential data hazards could occur, and then stall the pipeline for an appropriate amount of time (this has been done in some early Key-value Store designs on FPGAs, such as in the work by Blott *et al.*, 2013). However, such stalling is only acceptable if it happens infrequently. Unfortunately, tuples of the same group stored close together is not an unlikely scenario. Moreover, pre-sorted tables would exhibit the worst performance, which is counter-intuitive.

To solve this problem, it is customary to introduce a caching layer between memory and the pipelined hash table. This layer implements a *write-through cache* that holds the n last writes to memory. As shown in Figure 4.9, all read requests are logged temporarily in a queue, and the following writes to memory are cached during the *write* stage. When a new read is performed, the address of the key is first checked in the queue of recently accessed memory locations. For this purpose, the queue exposes a CAM-like[3] interface for reading, which returns the index inside the queue of the most recent access to the memory address in question. Using this index, the write cache, holding the data recently written to memory, can be accessed. Thus, in the *read* stage, actual read requests

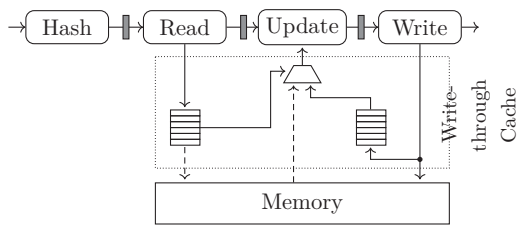


Figure 4.9: Pipelined hash table with *write-through cache* to avoid data hazards.

[3]Content-addressable memory (CAM) is a storage device in which the information is identified by content rather than by an address.

to memory are only issued for memory addresses that have *not* been accessed in the recent past. The logic in the *read* stage, then instructs the subsequent *update* stage to either fetch the next tuple from the write cache, or wait for it to be delivered from memory.

DRAM-Based Hash Table. FPGAs incorporate several megabytes' worth of Block RAM (BRAM) that is low latency SRAM on the device. It is clear, however, that there is a trade-off between the size of the hash table and the amount of bypass tuples with increasing number of groups. Therefore, as an alternative, it can be an option to consider off-chip DRAM, in the order of hundreds of megabytes or gigabytes, depending on the underlying platform.

In the case of Ibex, the prototyping board[4] had a 256 MB DDR2 SODIMM (a somewhat low performance component by today's standards). Thanks to the caching layer, discussed above, DRAM can be used instead of BRAM transparently with the hash table pipeline. The only required modification to the hash table logic is the correct sizing of the memory access cache. The size of the cache depends on the actual memory latency, because items in the cache can be evicted only once they have been written to memory for effective protection against the data hazards discussed earlier. For instance, with BRAM a capacity of eight is sufficient, while with the DRAM on the Ibex platform the cache needs to hold at least 32 entries.

Performance Properties. This section describes a subset of the experiments presented in Woods *et al.* (2014) that best summarizes the benefits of hybrid processing between FPGA and CPU. Experiments were conducted on a Desktop PC featuring a Quad-Core Intel (i7-2700K, 3.50 GHz) processor with 8 GB of main memory. We ran MySQL 5.5.25 on a 64-bit Windows 7 platform, which was installed on a 256 GB OCZ

[4] https://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf.

Vertex 4 SATA III 2.5" SSD. An identical SSD was connected directly to the FPGA (Virtex 5, XC5VLX110T).[5]

As a summarizing experiment, we can compare how fast two of MySQL's storage engines, namely MyISAM and INNODB, can execute Group-by queries in comparison to Ibex. In particular, we want to show how far MyISAM and INNODB are from SATA II (300 MB/s) line-rate query processing. To this end, the following simple Group-by query was ran on a synthetic workload:

```
SELECT col1, COUNT ( * )
  FROM table                    (Q2)
GROUP BY col1;
```

We varied the table size between one and 1024 megabytes and every table always consisted of exactly 16 groups, each containing an equal amount of tuples stored in unsorted order. In Figure 4.10, on the x-axis we plot the table size and on the y-axis performance as execution time.

MyISAM stores tables in files managed by the operating system, i.e., a notion of database pages does not exist. By contrast, INNODB has

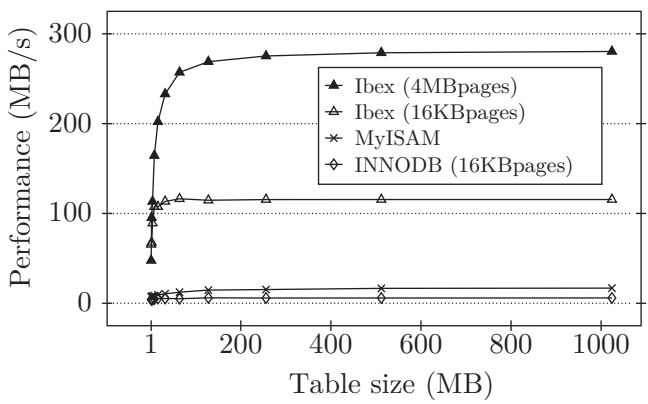


Figure 4.10: Performance of Ibex versus MyISAM and INNODB for table sizes ranging from one to 1024 megabytes.

[5]The OCZ Vertex 4 SSD is SATA III compatible but we used it exclusively in SATA II mode since our FPGA only supports SATA II, allowing a theoretical maximum bandwidth of 300 MB/s.

all the bells and whistles of a full-fledged storage engine, including a buffer pool to cache database pages, with a default page size of 16 KB.[6]

As can be seen in Figure 4.10, MyISAM performs better than INNODB. In all of our experiments, MyISAM always exhibited better performance than INNODB, which is due to its simplicity compared to INNODB. For example, MyISAM does not support database transactions and therefore employs a simpler locking mechanism. Since our goal is to compare performance of Ibex against the fastest common MySQL storage engines, we sometimes omit INNODB results in the following plots for readability reasons.

Ibex, configured to use 16 KB pages (≙), performs significantly better than both MyISAM (∗) and INNODB (⋄). However, the throughput is still far below the 300 MB/s of SATA II. Throughput can be increased using larger database pages. Thus, Ibex with 4 MB pages (▲) saturates the SATA II link, and shows that Ibex can sustain SATA II line-rate. While the maximum bandwidth of SATA II is 300 MB/s, note that actual transfer rates of stored data are around 280 MB/s due to protocol overhead and latencies within the SSD.

The previous experiment assumed that the entire group-by aggregation could be off-loaded to the FPGA. However, for most workloads hash collisions will occur, or the predetermined size of the hash table may be chosen too small to hold all groups. Thus, we need to quantify how *bypass* tuples impact performance. For this purpose, query Q_3 was ran on the 64 MB table from the previous experiment with modified group keys for each run to produce a varying number of collisions and bypass tuples:

```
SELECT col1, SUM ( col2 )
  FROM table                    (Q3)
GROUP BY col1;
```

[6] According to the documentation, by recompiling the code, one can set the page size to values ranging from 8 KB to 64 KB. However, this is not officially supported, and for our version of MySQL, we could not compile the INNODB code with pages larger than 16 KB.

As explained earlier, one way of handling bypass tuples it to rewrite queries, for instance as in Query Q_3:

```
SELECT col1, SUM(s)
  FROM (SELECT   col1, SUM(col2) AS s
        FROM     table                    (Q'3)
        GROUP BY col1) AS t1
GROUP BY col1;
```

The inner query is executed by the Ibex storage engine, returning a result table that contains bypass tuples, as well as partial aggregates. The outer query, on the other hand, is evaluated completely by the MySQL query processor.

The results are depicted in Figure 4.11. The y-axis shows execution time, and the x-axis displays the percentage of tuples that are bypassed and aggregated in software. When running query Q_3 with MyISAM, there are of course no bypass tuples but since we modified the group keys for every run, we show a separate measurement for each workload also for MyISAM. Not surprisingly, execution time is relatively constant for all workloads and takes roughly 8.5 seconds ($\ast$).

When running query Q'_3 with Ibex, the execution time depends on the number of bypass tuples. With no bypasses and 4 MB pages ($\blacktriangle$) query execution takes only 0.26 seconds, which is 32 times faster than

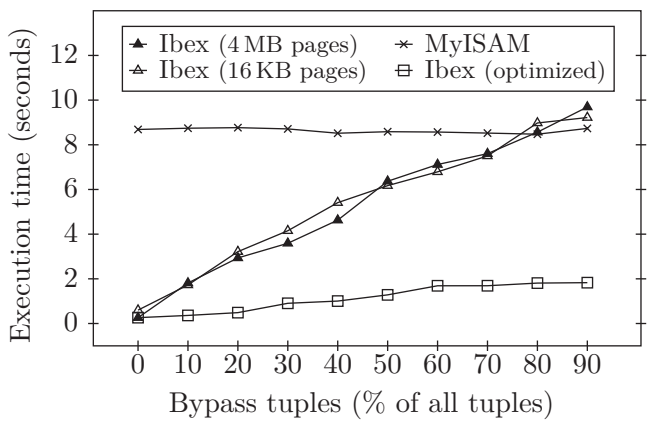


Figure 4.11: Impact of bypasses on execution time.

MyISAM. Execution time then increases linearly with respect to the number of tuples bypassed. At 90% bypassed tuples, the performance of Ibex is slightly worse than that of MyISAM since query Q_3' actually consists of two queries, while query Q_3 is a single query.

We ran query Q_3' both with 4 MB pages (▲) and 16 KB pages (△). Figure 4.11 shows that the page size here does not significantly affect execution time. Hence, execution time is dominated by the overhead of grouping and aggregating in the MySQL query processor. When, instead of the rewrite approach, the software hash table inside Ibex is used to deal with the bypass tuples, performance increases significantly (⊟).

4.4 Core SQL: Where Predicates and Filtering

Caribou (István *et al.*, 2017) implements two types of in-storage processing: comparison-based filters and regular expressions. The module used for regular expression processing on FPGAs has been already covered in Subsection 4.2. In addition to regular expressions, it can be beneficial to implement conditional expressions on numbers and bytes. This is useful if relational tuples are mapped to byte-arrays in the storage. Tuples can contain a combination of fixed and variable size columns, with the fixed size columns usually stored at the beginning of the record. The regular expression matcher can be used to filter out string-based columns, and to be able to filter based on the fixed-size ones more efficiently, Caribou uses so called "condition matchers" that compare a specific offset in the value to a constant. This enables clients to push down the filtering expression such as the following one to the storage: `where col2 = 123 and col3 < 4 and col4 = 'abcd'`. All three parts can be handled by the condition matchers in Caribou, but it is up to the client to map the name of a column (attribute of the tuple) to a specific byte-offset in the value at runtime.

In the following we present a short overview of how such condition matchers can be implemented with the goal of demonstrating that, in order to achieve line-rate performance, data parallel execution is not always a must. In the case of the condition matchers, a wide processing bus and deep pipelining is used to ensure that data can be processed at fixed bandwidth, without having to parallelize execution across several

tuples. In terms of their resource consumption, these modules are very small when compared, for instance, to regular expression matchers and can be included in virtually any data processing pipeline to pre-filter data.

Design Overview. A single conditional matcher compares a 32 bit word at a specific offset of the value to a constant. The comparison can be done using different functions. We have implemented ==, !=, < and >, but this could be easily extended. Each matcher is parametrized at runtime. This takes one clock cycle and does not degrade performance, even if every lookup uses predicates. The parameter word encodes the predicate for each comparator unit as a triplet: (1) a byte offset in the value representing the start of the column, (2) a comparison function, and (3) a 32 bit value that is evaluated against the value at the given offset with the given function. For the previous example, for instance, `col2` might map to byte 0–3 in the value, `=` is function number 0 and the constant `123` is provided as-is.

To be able to execute filters that contain more than a single comparison, we instantiate multiple matchers. They are assembled into a pipeline, as shown in Figure 4.12. This allows data to be simply streamed through these steps and be tagged in the process with a match-bit. The match-bit will determine whether a value is to be sent to the client, or

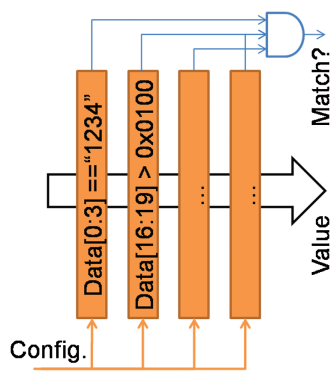


Figure 4.12: Conditional matchers compare a given offset in the value to an input constant. By chaining multiple of them, more complex conditions can be expressed.

to be dropped inside the storage node. In our current implementation we only allow the logical "and" operation between different conditions, but this could be extended to arbitrary combinations using the same lookup table technique we used in Ibex (Woods *et al.*, 2014).

Internal Architecture. Figure 4.13 illustrates the internal of each conditional matcher. Data flows through from left to right in 512 bit words (this is the memory word width). This circuit is designed to process values at one word/cycle rate, and it requires a single cycle pause between values.

In addition to the valid/ready signals, that have been omitted from the figure for simplicity, the data is tagged with a last and drop signal as well. The last word of the value is marked by the last flag, and in case drop is also enabled with the last flag, it means that the value has not met one of the conditions and will not be sent to the client.

Even though comparing for equality is not expensive in hardware, less-than and more-than comparisons require significantly more circuitry. Since a conditional matcher needs to be able to compare its constant to

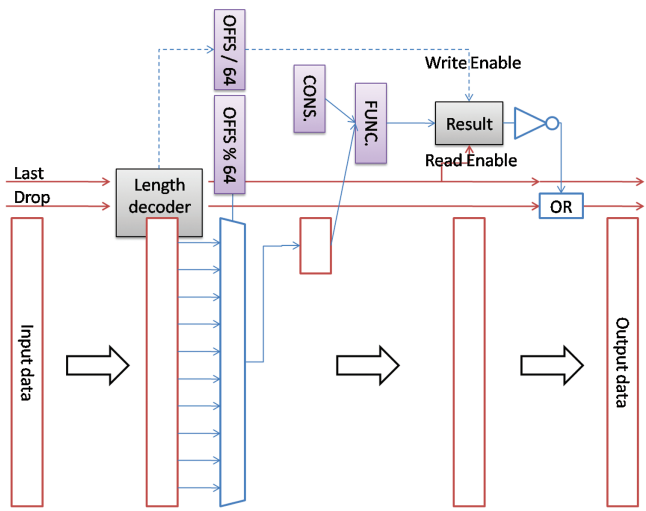


Figure 4.13: Conditional matchers are pipelined and contain several parameterizable elements (offset, function to compare with and constant to compare to).

any offset in the value, two implementation options are possible: one could replicate the constant and the comparison circuit to perform a parallel comparison starting at each byte in the input in parallel, and then discard all but one result, or have a single comparison circuit and "bring" the desired subset of bytes to it. We chose the second option, and use a multiplexer to select the part of the value that is of interest.

The lower bits of the offset parameter are used to drive a multiplexer, as can be seen in the upper part of Figure 4.13. The result of this comparison is written into a register. On the output side, the last signal is used to read out this register and to output the drop signal.

Performance Properties. In our current design the conditional matchers run at the network clock (156.25 MHz) and use a 64 B wide data bus. One of the expensive parts in their architecture is the registering of wide data words and the multiplexing from every possible byte-offset to the comparators. It would be possible to save resources by doubling the clock frequency of this module and halving its internal data lines to 32 B (the overall throughput would stay constant).

Table 4.1 illustrates what is the throughput of this module with increasing value sizes, as the one cycle overhead between values becomes less and less significant. It also shows how its behavior would change with the faster clock/narrower bus modification. While at first thought the throughput should stay the same, the one cycle overhead is reduced in absolute terms thanks to the faster clock and as a result the achievable throughputs are higher.

Table 4.1: Throughput of the conditional matchers for different value sizes

Value Size [B]	Tput. at 156 MHz
64	4992 MB/s
128	6656 MB/s
256	7987 MB/s
512	8874 MB/s
1024	9397 MB/s

4.5 Machine Learning: K-Means Clustering

K-means clustering is used as a building block for unsupervised learning tasks spanning several application domains (Dhanachandra *et al.*, 2015; Jamesmanoharan *et al.*, 2014). Since K-means is iterative and computationally intensive, it is a common target for software and hardware optimizations (Böhm *et al.*, 2017; Tang and Khalid, 2016; Wang *et al.*, 2016b). In software, techniques such as vectorization (Böhm *et al.*, 2017) lead to significant speedups. Using GPUs to accelerate K-means has also shown promise (Li *et al.*, 2013), often combined with pruning techniques (Hadian and Shahrivari, 2014; Tang *et al.*, 2017) to avoid comparing points to distant centroids.

There is a wide range of research on accelerating K-means using FPGAs, typically focusing on image data (Saegusa and Maruyama, 2006; Wang and Leeser, 2007) and on offloading only parts of the algorithm such as the assignment (Estlick *et al.*, 2001; Gokhale *et al.*, 2003). The benefit of offloading the entire algorithm, and thereby reducing communication with the CPU, was shown by Liu *et al.* (2005). In the following, we will present a design that uses state-of-the art K-means pipelines for multi-dimensional fixed point data on the FPGA and adds flexibility (He *et al.*, 2018), in that the same circuit can be used either to maximize bandwidth usage to the memory, or to maximize computation per accessed byte. This flexibility is important because, even though, traditionally, FPGA-based designs do not consider memory bandwidth conservation a design goal, in hybrid CPU-FPGA architectures, memory bandwidth can become a scarce resource because it is shared between concurrent operators running in both software and hardware. In order to ensure that hardware operators do not prevent others from making progress, they need to be designed so that their memory bandwidth requirements can be adjusted in steps. Instead of simply slowing down processing when memory bandwidth is scarce, some operator designs can offer more computation on the same data: one example is running the clustering algorithm with different parameters concurrently. Thanks to the parallelism of the FPGA, it is possible to trade off chip space for increased compute bandwidth per input byte without slowing processing

input : D, k, t_{max}
// ① Initialization Step
$t \leftarrow 0$;
Randomly initialize k centroids $\mu_{\mathbf{1}}^{\mathbf{t}}, \mu_{\mathbf{2}}^{\mathbf{t}}, \ldots, \mu_{\mathbf{k}}^{\mathbf{t}}$;
while $t < t_{max}$ **do**
  $t \leftarrow t + 1$;
  $C_j \leftarrow \emptyset$ for all $j = 1, \ldots, k$;
  // ② Cluster Assignment Step
  foreach $\mathbf{x_j} \in \mathbf{D}$ **do**
    $j^* \leftarrow \text{argmin}_i \{\|\mathbf{x_j} - \mu_{\mathbf{i}}^{\mathbf{t}}\|^2\}$;
    $C_{j^*} \leftarrow C_{j^*} \cup \mathbf{x_j}$;
  end
  // ③ Centroid Update Step
  foreach $i = 1$ *to* k **do**
    $\mu_{\mathbf{i}}^{\mathbf{t}} \leftarrow \frac{1}{|C_i|} \sum_{x_j \in C_i} \mathbf{x_j}$
  end
end

Algorithm 1: K-means algorithm

down (whereas in a CPU, once the algorithm is compute bound, such parallelism will lead to slower processing).

Running the algorithm with different parameters addresses a key challenge of clustering operators, namely determining the "right" number of clusters. The most common method to detect the value of k is called the "elbow method". This traditionally requires multiple runs over the same dataset using a different numbers of clusters until adding another cluster does not significantly decrease the squared sum of errors within the clusters. On an FPGA, this computation can be efficiently parallelized, resulting in predictable runtime.

Algorithm Overview. K-means is an unsupervised clustering algorithm that groups data points into a predefined number of clusters k. Each data point has a number of dimensions d. To assign data points to a certain cluster, a distance metric is used. For instance, the Euclidean

distance:

$$dist^2(\mu, x) = \sum_{i=1}^{d} |x_i - \mu_i|^2 = \|\mathbf{x} - \mu\|^2$$

where x and μ are two points of d dimensions.

The algorithm (Algorithm 1) consists of three steps: ① the initialization step, which picks k random centroids, ② the assignment step, where each point is assigned to its closest centroid, and ③ the update step, where the centroids are recalculated as the mean of the points assigned to them. The algorithm is either executed for a fixed number of iterations or until the centroid assignments no longer change.

As an input parameter, the K-means algorithm requires the value k. However, if no prior information about the data set is given, it is difficult to choose the most suitable value of k. In order to find the optimal number of clusters for a data set, the algorithm is run on a range of k values until convergence. The sum squared error (SSE) between each data point and its assigned centroid is calculated as follows:

$$SSE = \sum_{i=1}^{k} \sum_{x \in C_i} dist^2(\mu_i, x)$$

where k is the number of clusters, x is a data point in cluster C_i with the centroid μ_i. Since increasing the number of clusters will always reduce the average distance of the centroids to the data points, increasing k will decrease this metric to the extreme of reaching zero when k is the same as the number of data points. Therefore, simply minimizing this metric cannot be used as the target of finding an optimal cluster number. Instead, the sum squared error as a function of increasing k is measured and the "elbow point", where the rate of decrease sharply shifts, can be used as a guide to determine k.

Operator Design Overview. The K-means operator consists of three main parts: the *Controller* which requests data from the main memory and writes back the results, the *Assignment* pipeline which calculates the distance between a data point and all the centroids and assigns it to the closet centroid, and the *Update* module which updates the centroids in-between iterations. The operator processes 32 bit fixed point values.

When the operator is started, the *Controller* reads the initial centroids from the main memory. The distance processors in the *Assignment* pipeline are then initialized with these centroids. After initialization, the *Controller* starts fetching the data points from memory and, depending on the operational mode, either broadcasts or distributes them in round robin fashion to the pipelines.

The input data points are expected in row format meaning all dimensions of a data point arrive in order. For this reason, when integrating with systems, such as, DoppioDB, the column-to-row converter module is required. After the algorithm is executed for the intended amount of iterations, the *Controller* writes the final centroids and the corresponding SSE back to main memory where it can be read by the user application.

The *Assignment* pipeline implements the assignment step of the K-means algorithm which is computationally intensive since it calculates the distances between each data point and all the centroids. To address this, we deploy a series of distance processors that exploit the parallelism of the FPGA. As shown in Figure 4.14, the data points are streamed one dimension at a time from the *Controller* through the array of distance processors. Each processor stores a centroid and calculates the Euclidean distance between its centroid and the data points streaming through. To find the minimum distance between a point and all the centroids, each processor receives the minimum distance calculated from the previous

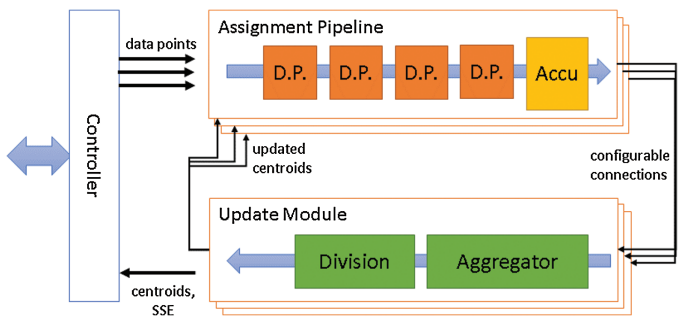


Figure 4.14: Architecture of the operator with multiple *Assignment* pipelines and *Update* modules which can be dynamically interconnected at runtime.

processors and compares it to the distance to its centroid. It then forwards the minimum of the two values and the corresponding cluster assignment. In this way, the minimum distance and the corresponding cluster assignment can be obtained at the end of the array when the last dimension of a data point is output.

The output of the distance processor array is fed into the Accumulator (*Accu*) module which accumulates the data points per cluster and maintains counters indicating how many data points are assigned to each cluster. After all data points have passed through the processor array, the accumulator is triggered to push its results to the *Update* module.

The *Update* module contains an *Aggregator* and *Divider* module. The *Aggregator* can aggregate the data points, count of assignments, and the SSE from multiple pipelines. The aggregated values are then forwarded to the *Divider* module which calculates the new centroids. The *Update* module then pushes the new centroids to the corresponding pipelines to update the centroids stored in the distance processors.

Both the number of dimensions per data point and the number of clusters can be parametrized at runtime. The distance processors receive with each dimension a flag indicating if this is the last dimension of the data point. This flag is set by the *Controller*, therefore the distance processors themselves are oblivious to the number of dimensions.

The connections between pipelines and *Update* modules are configured at runtime and depend on the operational mode: *High Bandwidth* or *Elbow*. Figure 4.15 depicts how, in the *High Bandwidth* mode, they are all assigned and parametrized with the same number of clusters k. This operational mode provides the highest bandwidth for a specific number of clusters and maximizes memory bandwidth usage. In the *Elbow* mode, the user can partition and assign the pipelines to concurrently run different numbers of k for faster exploration of the space. This mode reduces the required amount of memory bandwidth. Both modes fully utilize all pipelines deployed on the FPGA.

For the *High Bandwidth* mode, all but one *Update* module are disabled because all pipelines work together and their results need to be merged after each iteration. Similarly, the updated centroids are loaded at the beginning of each iteration to all pipelines. In contrast,

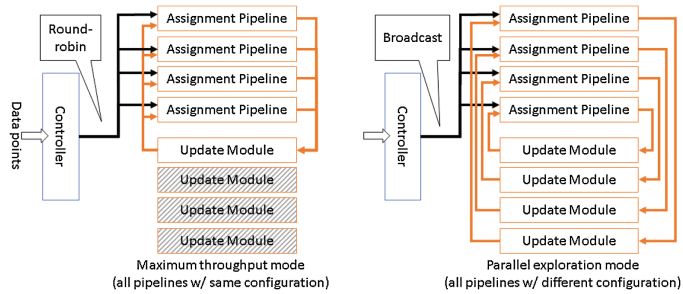


Figure 4.15: The operator can be parametrized at runtime for either high throughput meaning all pipelines execute the same configuration or for parallel exploration where each pipeline has possibly a different configuration.

for the *Elbow* mode, pipelines work in multiple parallel configurations (PC) independently, therefore they all use an *Update* module each.

Performance and Flexibility. As showcased in the previous section, the performance of FPGA-based operators that are designed in a fully pipelined manner depends on the clock frequency and the data bus width. In the case of the K-means operator, processing happens at 32 bit granularity and if one uses, for instance, a 200 MHz clock, each assignment pipeline can operate at 800 MB/s rate. Assuming 16 parallel pipelines inside the operator the nominal throughput is 12.8 GB/s. The operator has been deployed as part of DoppioDB (Subsection 3.3.1) on the Intel Xeon+FPGA machine by He *et al.* (2018). In this section we sample experiments from that work and compare the FPGA to a multi-core K-means implementation (Böhm *et al.*, 2017) which is highly optimized using SIMD and MIMD parallelism as well as minimizing data transfers between registers, cache, and main memory.

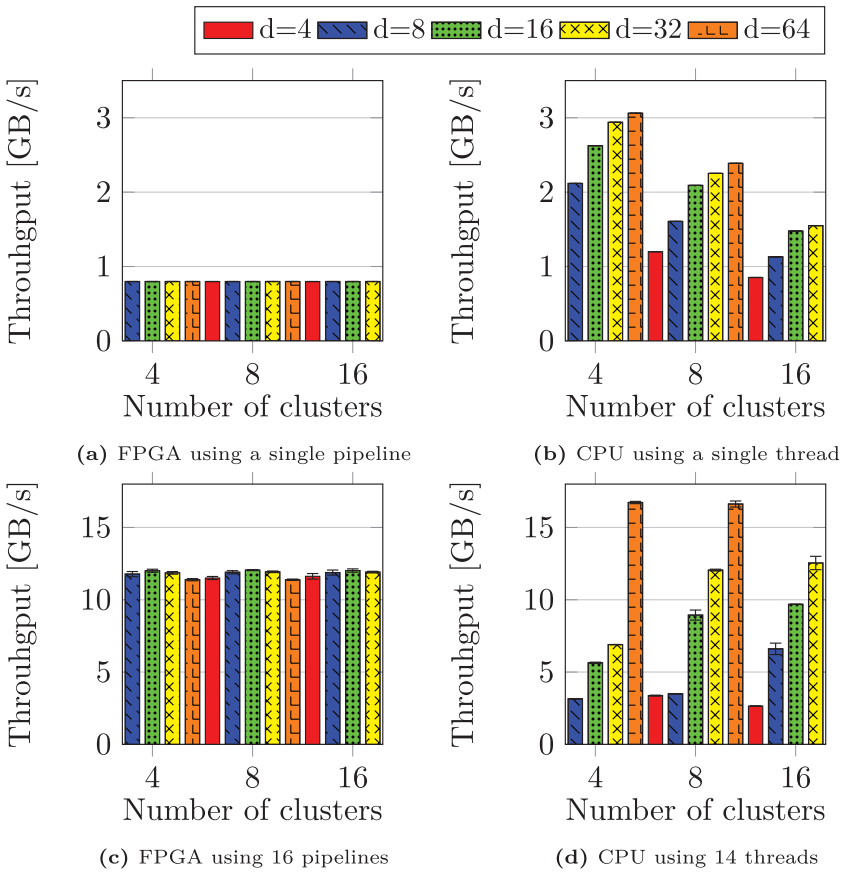
In terms of resource usage, both the *Assignment* pipeline and the *Update* module have a very low resource usage allowing the design to scale out in the number of pipelines and thereby increasing the bandwidth. The resource usage on the FPGA in the Intel Xeon+FPGA machine increases linearly from one to 16 pipelines, using up to 40% of logic resources. DSPs (small arithmetic cores on the FPGA) are mainly used by the *Assignment* pipeline to calculate the Euclidean distance in

each distance processor. For 16 pipelines up to 58% of the DSPs are occupied, making it the critical resource for further scalability.

The throughput of our operator is evaluated in the *High bandwidth* mode. This means all active pipelines are configured with the same centroids and number of clusters (each pipeline supports up to 16 clusters and 64 dimensions) and input is distributed in a data-parallel fashion. A single *Aggregator* module is active to aggregate and update the centroids. The throughput of the circuit increases linearly with the number of parallel pipelines used. With 16 pipelines our design reaches 11.4 GB/s close to the theoretical peak of 12.8 GB/s when operating at 200 MHz. This throughput scaling shows that the *Controller* fetches data at a sufficient rate and distributes it efficiently to all pipelines.

As a comparison, the multi-core K-means software implementation is ran with increasing number of threads. As can be seen from the previous experiment, the chosen configuration ($k = 8$, $d = 64$) suits the software implementation. The FPGA peak performance is slightly higher than the 8-threaded execution. By using 14 threads the CPU benefits from the high parallelism and memory bandwidth and reaches up to 16.7 GB/s. From these results, it can be concluded that the FPGA can match at least 10 cores in terms of absolute performance while providing fully predictable response times for all parameter ranges.

To showcase the flexibility of the operator, the number of dimensions is varied in a synthetic data set and change the number of centroids to cluster the data around. The performance is measured for a single and 16 pipelines on the FPGA, and a single and 14 threads in software. Figure 4.16 shows very stable throughput independently of the choice of d and k. In both cases the throughput by the FPGA implementation is close to its theoretical maximum of 0.8 GB/s for single pipeline (32 bits per cycle at 200 MHz) and 12.8 GB/s for 16 pipelines. It is clear that a software based solution can provide the same amount of flexibility, however its performance varies depending on the input parameter and is less predictable. Figure 4.16(b) shows that in case of a single thread the throughput decreases with increasing number of clusters. However in the case of 14 threads, the software implementation benefits from more clusters and dimensions, since this allows better parallelization of the work among multiple threads (Figure 4.16(d)). At the same time



(a) FPGA using a single pipeline

(b) CPU using a single thread

(c) FPGA using 16 pipelines

(d) CPU using 14 threads

Figure 4.16: Throughput using synthetic data, with a varied number of clusters and dimensions.

for a low number of dimensions and clusters increasing the number of threads only shows a marginal benefit.

Given the flexibility of the operator, multiple k can be executed concurrently when running in *Elbow mode*. To illustrate this, the seminal

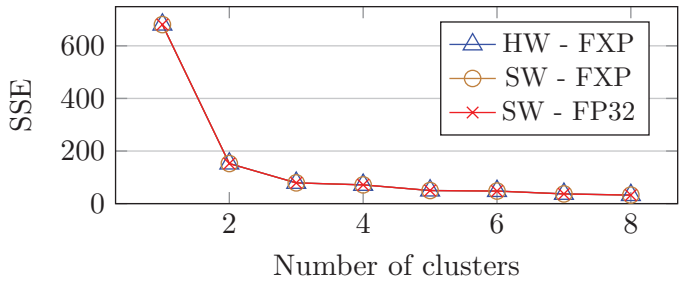


Figure 4.17: Elbow method applied to the Iris data set.

Iris data set has been expanded[7] to 15 Mio. data points and the K-means algorithm on the hardware has been configured to compute the clustering for eight different cluster numbers in parallel.

For each configuration the operator returns the SSE after the final iteration. The SSEs calculated by the hardware operator are plotted in Figure 4.17. For comparison, the error metrics computed by the software using either floating point or fixed point arithmetic are plotted as well. It is clearly visible that the three different implementations lead to the same results. The Iris data set contains three types of flowers, therefore it is expected to cluster well into three clusters as confirmed by the elbow method in Figure 4.17.

When our K-means operator operates in the *Elbow mode*, it can calculate the SSE for multiple number of clusters concurrently. This allows for a faster exploration of the optimum k while reducing the overall required memory bandwidth. In other words, it reduces the required amount of memory bandwidth by performing more computation on the data read from main memory.

Figure 4.18 compares the concurrent evaluation of eight clusters using two pipelines each on the FPGA to a single threaded execution of

[7]The Iris data set, https://www.kaggle.com/uciml/Iris/data, describes different types of flowers and has four data dimensions. Since it has only 150 data points in its original form, we expand it by generating multiple data points from each original point by adding uniform noise within 10% of the original values. Since the operator uses fixed point arithmetic, the Iris data set is converted from floating point to fixed point. Correspondingly the result returned by the operator is converted back to floating point.

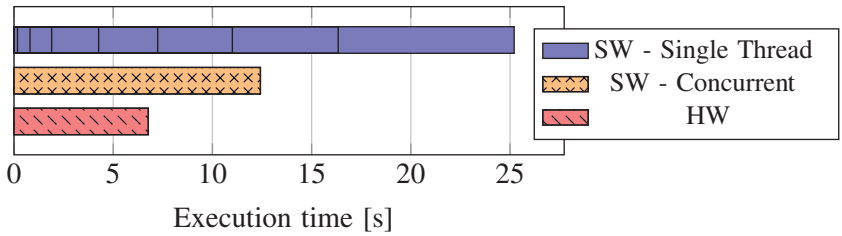


Figure 4.18: Evaluation of multiple k, both sequentially and concurrently on software and concurrently in hardware.

the software implementation which evaluates the eight different clusters sequentially. For sequential execution each k is run till convergence, while in the concurrent mode, the number of iterations is chosen such that convergence for all k's is ensured. The concurrent evaluation of multiple k's shows a clear performance benefit. For a fair comparison, all eight clusters are ran concurrently on the CPU each assigned to a different core (single-threaded). By using more cores, the software achieves a lower execution time over a single thread but remains inferior to the FPGA solution.

4.6 Machine Learning: Stochastic Gradient Descent

In recent years Machine Learning has revolutionized many fields and it is increasingly common for databases to include such operators in the their analytics toolkits, to provide classification or clustering operations, for instance. In this section we explore how FPGAs can be used to implement *training* with Stochastic Gradient Descent. One prominent feature of machine learning algorithms, especially those trained with stochastic gradient descent, is that they can tolerate certain types of noise and errors incurred during execution and still return statistically the same answer. This observation has enabled a range of system optimizations such as lock-free (Recht *et al.*, 2011) and asynchronous execution (You *et al.*, 2016). Recently, one emerging line of research has focused on developing machine learning algorithms that can tolerate a different type of noise—low-precision data representation and computation (Gupta *et al.*, 2015; Kim *et al.*, 2011; Lesser *et al.*, 2011; Perez-Garcia

et al., 2014). FPGAs are especially interesting devices for custom precision computation thanks to their architectural flexibility (Lesser *et al.*, 2011; Mücke *et al.*, 2010; Rabieah and Bouganis, 2016). This subsection based on recent work by Kara *et al.* (2017) presents the advantages of using an FPGA-based accelerator to process low-precision data directly.

The FPGA-based computation focuses on training one of the simplest class of machine learning models – dense linear models. Despite their simplicity, dense linear models are fundamental for applications such as regression and classification, compressive sensing, and image reconstruction. For applications such as human-in-the-loop analytics and feature selection, one often needs to train hundreds or thousands of models. Thus, the training speed is important.

SGD is an iterative algorithm that performs multiple passes over the data (so called *epochs*). There are two decoupled metrics to assess the performance of SGD: (1) *statistical efficiency*, the number of epochs (N_{epochs}) the algorithm needs to converge, and (2) *hardware efficiency*, the time the algorithm requires to execute each epoch (T_{epoch}). The objective of FPGA-based offloading is to increase the hardware efficiency, by lowering T_{epoch}, and maintain statistical efficiency, by keeping N_{epochs} the same.

We present the following two designs to explore the performance objective in the given machine learning scope:

1. `floatFSGD`: An FPGA-based SGD implementation working on 32-bit floating-point data. Apart from being scalable (handling high dimensionality) and resource-efficient, `floatFSGD`'s performance is on par with a 10-core CPU, despite being bound on the available memory bandwidth on our current platform.

2. `qFSGD`: An FPGA-based SGD implementation working on quantized data (1,2,4, or 8-bit fixed-point). `qFSGD` mainly showcases how to increase internal computation parallelism (i.e., vectorization width) thanks to the architectural flexibility of an FPGA. `qFSGD` is up to 11× faster than `floatFSGD` and up to 10.6× faster than the fastest 10-threaded CPU version of SGD.

Background: SGD and Quantization. The scope is the following optimization problem: Given a dataset $(a_i)_{i=1,N}$ of D-dimensional data points, each with its own label $(b_i)_{i=1,N}$, we wish to identify the dense linear model $\mathbf{x}$ which minimizes the classification loss over this dataset:

$$\arg\min_x Q(x) = \frac{1}{N}\sum_{i=1}^{N} \text{loss}_i(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} \frac{(\mathbf{a_i}\mathbf{x} - b_i)^2}{2} \tag{4.1}$$

- $\mathbf{a_i} \in \mathbb{R}^{1\times D}$, a single sample of data set $\mathbf{a} \in \mathbb{R}^{N\times D}$,
- $b_i \in \mathbb{R}$, the corresponding true inference value to $\mathbf{a_i}$,
- $\mathbf{x} \in \mathbb{R}^{D\times 1}$, the model to be trained and used for inference,
- $N \in \mathbb{N}$, the number of samples,
- $D \in \mathbb{N}$, the number of features per sample.

A standard tool for solving this problem is stochastic gradient descent (SGD), consisting of the iterative process in Algorithm 2. For a small enough step size γ, SGD will converge to the *optimal* solution (Bottou, 2010).

Data: dataset $a_1, a_2, \ldots, a_N$ of D-dimensional data
Result: optimal value of the model $\mathbf{x}$
Initially, $\mathbf{x}$ is zero;
while *not converged* **do**
  for *i from* 1 *to* N **do**
    $\mathbf{g_i} = \frac{\partial \text{loss}_i(\mathbf{x})}{\partial \mathbf{x}} = (\mathbf{a_i}\cdot\mathbf{x} - b_i)\mathbf{a_i}$;
    $\mathbf{x} \leftarrow \mathbf{x} - \gamma\mathbf{g_i}$;
  end
end

Algorithm 2: Stochastic Gradient Descent

Recent work by Zhang *et al.* (2017a) shows that SGD convergence can be guaranteed even if the data undergoes a compression process called *stochastic quantization* before it is used in the gradient update. We now provide a brief explanation of this procedure. Assume that the data consist of floating-point values contained in an interval $[L, U]$.

We *quantize* each data point $a_{i,j}$ to one of s levels, as follows. First, we split the interval $[L, U]$ into $s-1$ intervals of equal length $\Delta = (U-L)/(s-1)$. Then, each data point is rounded stochastically to one of the endpoints of its interval:

$$Q_s^{L,U}(a_{i,j}) = \begin{cases} \left(\left\lfloor \frac{a_{i,j}}{\Delta} \right\rfloor + 1\right)\Delta & \text{with prob. } a_{i,j} - \left\lfloor \frac{a_{i,j}}{\Delta} \right\rfloor \Delta \\ \left\lfloor \frac{a_{i,j}}{\Delta} \right\rfloor \Delta & \text{otherwise.} \end{cases} \tag{4.2}$$

Example: Quantize value 0.7 between [0,1] with two levels. $\Delta = 0.\bar{3}$

$$Q_2^{0,1}(0.7) = \begin{cases} 1 & \text{with prob. } 0.7 \\ 0 & \text{with prob. } 0.3 \end{cases}$$

This quantization procedure is chosen so that the *expected* quantized value returned equals the value itself, that is:

$$\mathbf{E}[Q(a_{i,j})] = a_{i,j}. \tag{4.3}$$

In other words, if we iterate the quantization procedure on a sample, the average of the returned values would converge to the value of the sample. The key observation by Zhang *et al.* (2017a) is that SGD still converges even if samples are quantized in this way. However, to preserve correctness, we must take two independent quantizations Q' and Q'' for each sample a_i, and update the gradient value to:

$$\hat{\mathbf{g}}_i = (Q'(\mathbf{a_i})^T \mathbf{x} - b_i) Q''(\mathbf{a_i}). \tag{4.4}$$

This choice of update ensures that $\mathbf{E}[\hat{\mathbf{g}}_i] = \mathbf{g}_i.$, i.e., the update is an *unbiased estimator* of the true gradient, which in turn ensures convergence of SGD.

FPGA-SGD on Float Data (`floatFSGD`). We first present an SGD implementation that works on 32-bit floating-point data (Figure 4.19), a common data representation in machine learning. As the data is accessed in a 64 B cache-lines, the circuit is designed to work on this

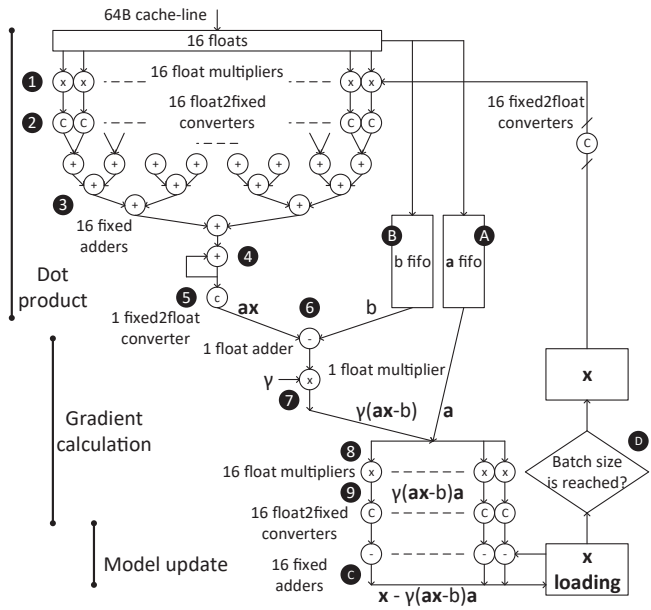


Figure 4.19: Computation pipeline for `floatFSGD`, latency: 36 cycles, data width: 64 B, processing rate: 64 B/cycle.

data width. It is able to consume a cache-line at every clock cycle (200 MHz), resulting in the processing rate of 12.8 GB/s.

Scale to # of features: The challenging part of the design is to make it capable of handling a number of features D that is larger than 16, which is the default width of the pipeline. This is possible since all vector algebra in Algorithm 2 can be performed iteratively, where each portion contains 16 values. To stay cache-line aligned, we use zero-padding if $D \bmod 16 \neq 0$. Thus, we can calculate how many cache-lines it takes for $\mathbf{a_i}$ (one row in the set) to be completely received:

$$\#\mathbf{a_i} \text{ cache-lines} = \begin{cases} D/16 & \text{if } D \bmod 16 = 0 \\ \dfrac{D + (16 - D \bmod 16)}{16} & \text{if } D \bmod 16 \neq 0. \end{cases}$$

The only parameter determining the scalability of `floatFSGD` is the maximum dimensionality D_{max}, because it determines the amount of BRAM needed for storing the model $\mathbf{x}$. We choose D_{max} to be 8192,

which is more than enough for most existing linear dense model training examples. The design can handle any number of samples, N, since training is done in a streaming fashion.

Walk-through of computation pipeline: In the following, we explain each stage of the computation pipeline. The first stage is the dot product $\mathbf{a_i} \cdot \mathbf{x} = \sum_{j=1}^{D} a_{i,j} x_j$. When a cache-line containing a part of vector $\mathbf{a_i}$ arrives, it is first multiplied (❶) with the corresponding part of vector $\mathbf{x}$, in floating-point with multiplier IPs,[8] which have 5-cycle latency and throughput of one result per cycle. Then, the values are converted to a 32-bit integer (❷) by multiplication with a large constant that is configurable during runtime, depending on the value range desired. This *float2fixed* conversion takes one cycle. After that, an adder tree (❸) accumulates 16 values, each layer taking one cycle. At the output of the adder tree is an accumulator (❹). It accumulates the results coming out of the adder tree for the pre-calculated number $\#\mathbf{a_i}$ cache-lines, building the final value for the dot product. The dot product result is converted back to floating-point (❺), since the next stages of the calculation will be performed with floating-point data. In the next part, the rest of the gradient calculation takes place. First, the scalar value b (the true inference value in the data set), is subtracted from the dot product (❻) using a floating-point adder IP, which has 7-cycle latency and throughput of one result per cycle. The b value is received some cycles before the subtraction takes place and is placed into a FIFO (Ⓑ), waiting there until the dot product result is ready. After the subtraction, a floating-point multiplication takes place with step size γ (❼), which can be configured to any *float* value. At the end of this step, we have a scalar value $\gamma(\mathbf{a_i} \cdot \mathbf{x} - b_i)$, which needs to be multiplied with vector $\mathbf{a_i}$. At this stage, a FIFO (Ⓐ) already contains all parts of vector $\mathbf{a_i}$, because the incoming cache-lines are written to this FIFO simultaneously as they were sent to the dot product calculation. The scalar-vector multiplication (❽) takes place in floating-point, where all parts of $\mathbf{a_i}$ are multiplied with the same scalar value. This gives the gradient $\mathbf{g_i}$ one part at a time, which undergoes *float2fixed* conversion (❾), so that a cycle-by-cycle update of the model $\mathbf{x}$ (Ⓒ)

[8] All floating point IPs are created via Altera Quartus II 13.1.

can take place. This would not be possible with a floating-point adder having 7-cycle latency, since the result of the current calculation is needed in the next cycle. The gradient is applied to the corresponding part of the model as it becomes available. After the last part of the gradient is subtracted from the model, the update for $\mathbf{a_i}$ is completed. After all rows go through the same calculation, one epoch is completed (Algorithm 2).

Staleness vs. batch size (as a result of pipelined execution): The model updated and the model read for the dot product are separate (Figure 4.19). Only when a certain batch size (the number of already processed $\mathbf{a_i}$) is reached, the updated model is carried on to the actual model (❶). The reason is the latency introduced by the computation pipeline: in theory, the whole gradient calculation and the update to the model as in Algorithm 2 should be an atomic operation. However, to exploit deep-pipelining, we don't perform this operation atomically. Instead, we keep the actual model and the updated model separate and carry out the accumulated update only when a certain batch size is reached (called a mini-batch SGD). The batch size is a configurable parameter, which should be set to the latency of the pipeline (36 cycles) to avoid any so-called stale updates.

End-to-end float vs. hybrid computation: We choose a hybrid (*float+fixed*) over end-to-end *float* computation, because a 7-cycle floating-point addition latency leads to: (1) A high latency adder tree (❸) that imposes a larger batch-size to avoid staleness, slowing down the convergence rate, (2) not being able to do a cycle-by-cycle accumulation (❼), since the result of an ongoing addition is required in the next cycle. Thus, to keep the processing rate at 64 B/cycle, we choose a hybrid design that eliminates both these disadvantages.

FPGA-SGD on Quantized Data (`qFSGD`). We explain how we change the `floatFSGD` design to work on quantized data. The main purpose is simple: Instead of reading *float* data (only 16 values in a cache-line), we quantize the data beforehand, so that more than 16 values fit into a cache-line, thus reading less volume of data in total. There is one main challenge in making the FPGA-SGD work on quantized data: scaling

out the `floatFSGD` pipeline so that it can work on more than 16 values in parallel. Before we explain how this is achieved, we first review the quantization options we consider and how the data layout looks like.

Quantization for `qFSGD`: Equation (4.2) shows that, given a non-integer value, the quantization still might produce a non-integer value. However, floating-point arithmetic induced by non-integer values are hard to implement on the FPGA and scaling out such a design would be difficult. We take advantage of the fact that we can select the quantization variables $[L, U]$ and s aptly, so that only integer values are produced. Table 4.2 shows our choices for these values.

After selecting a quantization precision (one of *Q1*, *Q2*, *Q4* or *Q8*; powers of two to stay cache-line aligned), the data set (the values in matrix $\mathbf{a}$) must be normalized to the selected quantization's corresponding $[L, U]$. At this stage, the sign of the data set is considered for the normalization: we do not normalize a negative data set into a positive interval in order to keep existing zeros (maintain sparsity). Thus, we do not use *Q1* for negative datasets.

The layout of quantized data: As mentioned earlier in relation to Equation (4.4), to calculate the correct gradient, we need two quantization samples of the same data point. That is, if we, for example, select *Q8*, a quantized sample has eight bits and we need two of them to calculate the gradient; the actual amount of bits we use is 16. That's why, when we perform quantization on a data set, we always create two samples and store them in memory next to each other. Thus, we can calculate how many quantized values can fit into one cache-line, a value we call K, in Table 4.3. The value K dictates the amount of

Table 4.2: Choice of quantization levels, lower and upper bounds, so that only integer values are produced

Levels	Data Set Positive	Data Set Negative	Needed Bits
$s = 2$	$[L, U] = [0, 1]$	N/A	1, *Q1*
$s = 3$	$[L, U] = [0, 2]$	$[L, U] = [-1, 1]$	2, *Q2*
$s = 9$	$[L, U] = [0, 8]$	$[L, U] = [-4, 4]$	4, *Q4*
$s = 129$	$[L, U] = [0, 128]$	$[L, U] = [-64, 64]$	8, *Q8*

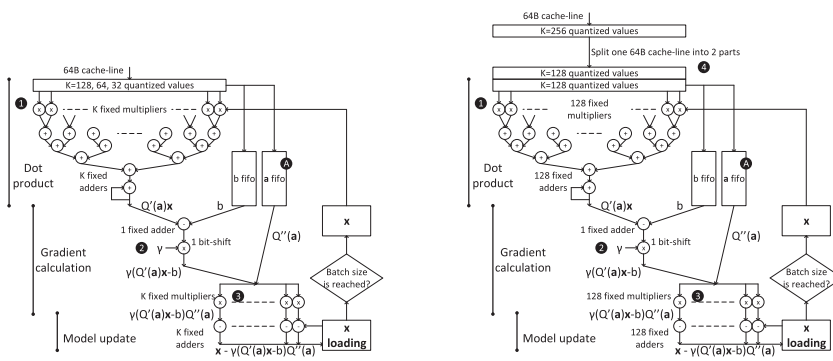
Table 4.3: Number of received values in a single cache-line

Data Type	*Q1*	*Q2*	*Q4*	*Q8*	*Float*
# of values in a cache-line, K	256	128	64	32	16
Processing rate (GB/S), PR	6.4	12.8	12.8	12.8	12.8

zero-padding we need to perform, in order to be cache-line aligned (similarly to how it is done for `floatFSGD`). Thus, the number of cache-lines required to receive one quantized row $Q(\mathbf{a_i})$ can be calculated:

$$\#\mathbf{a_i} \text{ cache-lines}(K) = \begin{cases} D/K & \text{if } D \bmod K = 0 \\ \dfrac{D + (K - D \bmod K)}{K} & \text{if } D \bmod K \neq 0. \end{cases} \tag{4.5}$$

Computation pipeline for quantized data (Figure 4.20): The selection of Qx, which determines the width of the pipeline is a generic parameter that can be set before synthesis. Thus, each Qx results in a different bitstream. The explanation here only focuses on the



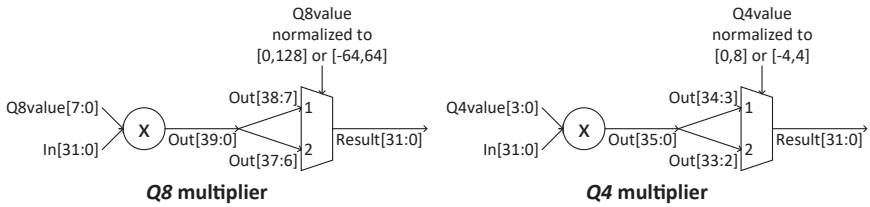
(a) *Q2*, *Q4* and *Q8* `qFSGD`,latency: $log(K)$+5 cycles, data width: 64B, processing rate: 64B/cycle.

(b) *Q1* `qFSGD`, latency: 12 cycles, data width: 32B, processing rate: 32B/cycle.

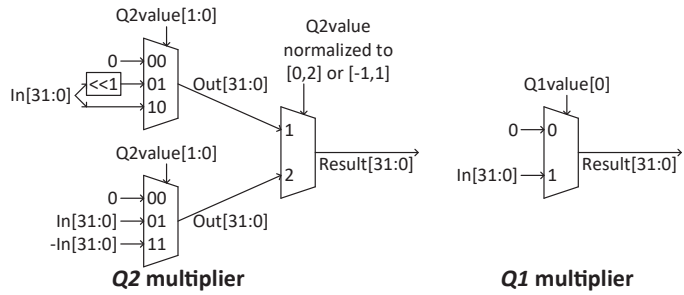
Figure 4.20: Computation pipelines for all quantizations. Although for *Q2*, *Q4* and *Q8*, the pipeline width scales out and maintains 64 B width, for *Q1* it does not scale out and the pipeline width needs to be halved, making *Q1* `qFSGD` compute bound.

differences from `floatFSGD` and how the pipeline is scaled out. The first thing to note is that *Qx* pipelines work only on integer data, so there is no need for converters. This is because the arriving quantized data $Q(\mathbf{a_i})$ is already in integer form, as explained previously, and the inference values b are also converted to integer by multiplication with a large constant. Another difference here is that for a given value in vector $\mathbf{a_i}$, two quantized samples arrive due to the double sampling method. The first sample is given to the dot product calculation (❶) and the second sample is put into a FIFO (🅐), where it is kept until the dot product result is ready, as depicted in Figure 4.20. The last difference is applying the step size γ (❷), which is actually a division. Since integer data is consumed here, γ can be applied simply as a bit-shift operation. By how many bits the value is shifted to the right is a runtime configurable parameter, allowing adjustments according to the data set characteristics.

Scaling out for quantized data and trade-offs: Scaling out the pipeline for *Q8* and *Q4* is straightforward using conventional signed multipliers (❸) implemented by DSP resources, followed by a bit-shift, to keep the data width at 32-bit (Figure 4.21(a)). However, for *Q2* and *Q1*, multiplication can be performed using only multiplexers (Figure 4.21(b)), since one of the multiplicands is only 2-bit and 1-bit, respectively. Doing this efficient multiplication allows the *Q2* pipeline to scale to 128-value parallelism, which would have otherwise required a 100% usage of the available DSP resources on the target FPGA (see Table 4.4). However, the pipeline shown in Figure 4.20(a) does not scale to 256-value parallelism at the target frequency of 200 MHz, even though the *Q1* multiplier is just one multiplexer. The main issue here is (1) the bus for propagating the model from the BRAM to compute units becomes too wide (8192 signals), (2) the adder tree becomes too wide and deep. Note that, we still have to perform addition in full-precision, therefore we cannot simplify the addition as we have done with the multiplication. Using compressor trees (Kumm and Zipf, 2014) instead of standard adder trees also does not help in meeting timing constraints. An alternative solution at the expense of the processing rate is to halve the `qFSGD` pipeline to process *Q1* data (Figure 4.20(b)). To do so, an



(a) Signed integer multiplication for *Q8* and *Q4* data, implemented with DSPs.



(b) Signed integer multiplication for *Q2* and *Q1* data, implemented with combinational logic.

Figure 4.21: Multiplication implementations depending on the quantization type.

arriving cache-line is split into two parts (❹), which are processed sequentially by the pipeline shown in Figure 4.20(b). The processing rate of this pipeline is 32 B/cycle, or 6.4 GB/s.

Experimental Evaluation. The main hypothesis to experimentally validate is: (1) low-precision data representation created via stochastic quantization can be used for training dense linear models while maintaining quality, and (2) since the processor doing the training needs

Table 4.4: Resource consumption for computation pipelines

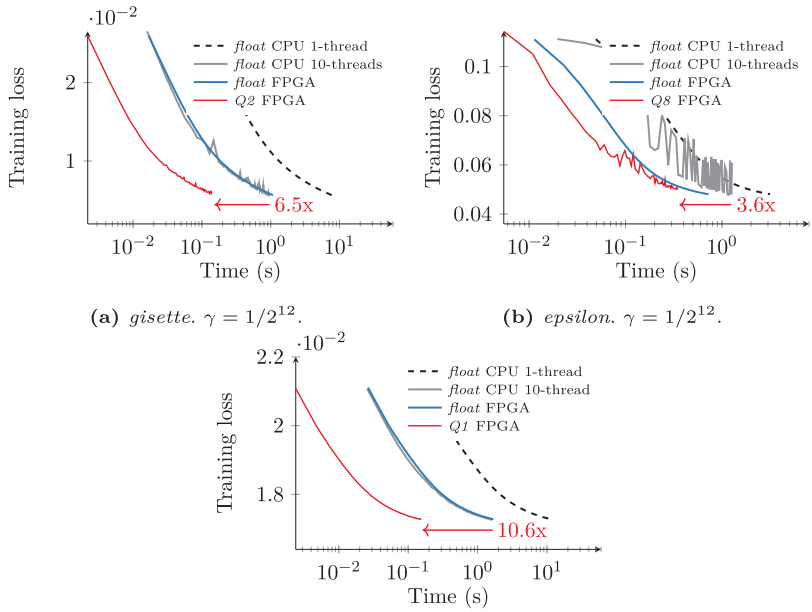
Data Type	Logic (ALMs)	DSP	BRAM (bits)
float	38% (89194)	12% (33)	7% (3.471 K)
Q8	35% (82152)	25% (64)	6% (3.145 K)
Q4	36% (84500)	50% (128)	6% (3.145 K)
Q2, *Q1*	43% (100930)	1% (2)	6% (3.145 K)

Table 4.5: Datasets used in experimental evaluation

Name	Training Size	Testing Size	# Features	# Classes
cadata	20,640		8	Regression
synthetic100	10,000		100	Regression
synthetic1000	10,000		1000	Regression
mnist	60,000	10,000	780	10
gisette	6000	1000	5000	2
epsilon	10,000	10,000	2000	2

to read less data per epoch, using quantized data provides speedup. To validate this, the FPGA-based SGD is used on various datasets having different characteristics (see Table 4.5). As the CPU baseline, we show both a single-threaded SGD doing exactly the same calculation as `floatFSGD` and a high performance parallel library called "Hogwild!" (Recht *et al.*, 2011) working on *float* data, with a mini-batch size of 36 (equivalent to `floatFSGD`). The multi-thread parallelism in Hogwild! is achieved through asynchronous updates: each thread works on a separate portion of the data and applies gradient updates to a common model without any synchronization. The asynchrony might reduce the statistical efficiency, especially if the data set is dense. Both CPU baselines make use of vectorized instructions and are compiled with GCC 4.8.4, with -O3 enabled.

Methodology: Since the step size is applied as a bit-shift operation, one of the following step sizes are used, which results in the smallest loss for the full-precision data after 64 epochs: $(1/2^6, 1/2^9, 1/2^{12}, 1/2^{15})$. With a given constant step size, FPGA-based SGD is performed with all precision variations that are implemented (*Q1*-only for classification data-, *Q2*, *Q4*, *Q8*, *float*). For each data set, we present the loss function over time in Figures 4.22 and 4.23, showing four curves: single-threaded and a 10-threaded CPU-SGD for *float*, `floatFSGD`, and `qFSGD` for the **smallest** precision data that **has converged within 1% of the original loss**. To goal is to show the difference in time for all



(a) *gisette*. $\gamma = 1/2^{12}$.

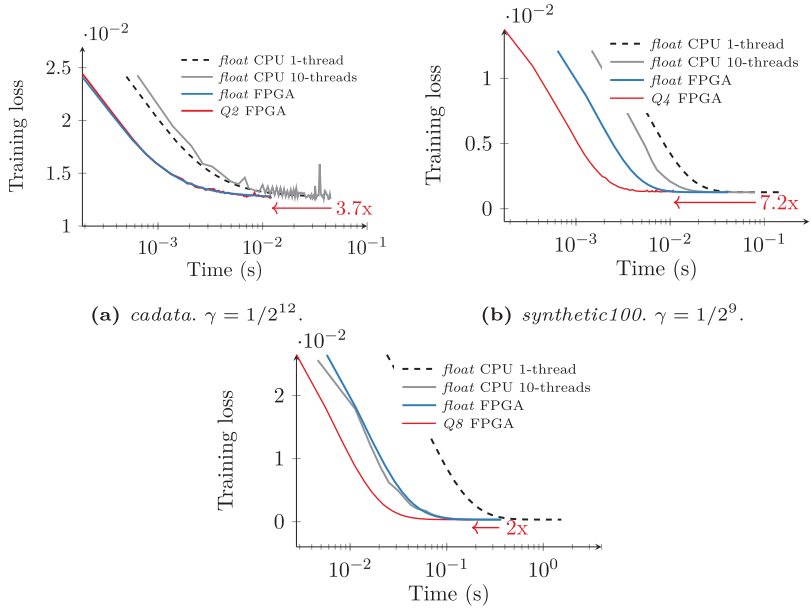
(b) *epsilon*. $\gamma = 1/2^{12}$.

(c) *mnist*, trained for digit 7. $\gamma = 1/2^{15}$.

Figure 4.22: SGD on classification data. All curves represent 64 SGD epochs. Speedup shown for Qx FPGA vs. *float* CPU 10-threads.

implementations to converge to the same loss, emphasizing the speedup that is achieved with `qFSGD` compared to full-precision variants.

Main results: In Figure 4.22 we observe that for all classification datasets, `qFSGD` achieves a speedup while maintaining convergence quality. For *gisette* in Figure 4.22(a), *Q2* reaches the same loss 6.9× faster than Hogwild!. Due to the high variance data in *epsilon*, both Hogwild! and Qx curves seem to be unstable (Figure 4.22(b)). We can see that `floatFSGD` in this case behaves well, providing both 1.8× speedup over Hogwild! and better convergence quality. On the *mnist* data set (Figure 4.22(c)), *Q1* can be used without losing any convergence quality, showing that the characteristics of the data set heavily effect the choice of quantization precision, which justifies having multiple Qx implementations. For *mnist*, *Q1* `qFSGD` provides 10.6× speedup over Hogwild! and 11× speedup over `floatFSGD`. In Figure 4.23(a),



(a) *cadata*. $\gamma = 1/2^{12}$.

(b) *synthetic100*. $\gamma = 1/2^{9}$.

(c) *synthetic1000*. $\gamma = 1/2^{9}$.

Figure 4.23: SGD on regression data. All curves represent 64 SGD epochs. Speedup shown for Qx FPGA vs. *float* CPU 10-threads.

`floatFSGD` converges as fast as *Q2* for *cadata*, because they read the same number of cache-lines. The reason is the cache aligned zero padding (set $D = 8$ and respective Ks for *Q2* and *float* into Equation (4.5)). For *synthetic100* and *synthetic1000*, we need to use *Q4* and *Q8*, respectively, to achieve the same convergence quality as *float* within the same number of epochs. In Figure 4.23(c), Hogwild! convergence is slightly faster than that of `floatFSGD`, and *Q8* provides 2× speedup over that. Hogwild! becomes faster with higher dimensionality (compare Figures 4.23(b) and (c)), because with lower dimensionality cache pollution occurs more frequently (Recht *et al.*, 2011).

The outcome of the main results: The same loss can be reached using quantized data within the same number of epochs as with full-precision data; thereby achieving better hardware efficiency while maintaining statistical efficiency. To achieve this, the precision has to be selected carefully.

Table 4.6: Multi-class classification on *mnist*. Run times are for training 10 models with 100 iterations and $\gamma = 1/2^{15}$

Precision	Accuracy for 10 Digits	Training Time (s)
float CPU-SGD	85.82%	19.0354
Q1 `qFSGD`	85.87%	2.4083

Classification accuracy: In Table 4.6 we show the accuracy on the training set for *mnist*, for which 10 separate models (for each digit) are trained. This can be parallelized with the CPU very well since each model can be trained completely independently. The CPU reaches a processing rate of 10 GB/s for this test, the highest observed for the CPU. On the FPGA, 10 models need to be trained one after the other, since there is only one SGD instance. *Q1* achieves 8× speedup against the highest performing CPU implementation, while maintaining the same multi-classification accuracy.

4.7 Distributed Joins: Data Partitioning

Data partitioning is a common step in data parallel algorithms running on distributed systems. Partitioning data among multiple machines improves locality and facilitates parallel processing. In database engines, operators such as joins (Schuh *et al.*, 2016; Schuhknecht *et al.*, 2015), aggregations, and sorting (Polychroniou and Ross, 2014) make use of data partitioning to parallelize the execution among multiple cores or machines. Interestingly, existing results show data partitioning is often the most expensive part of the operator, with most of the overhead arising from the *data shuffling* part which moves the data around while the hash function is generally very inexpensive (e.g., as used in radix partitioning). Data partitioning is not only a common operation in database engines, but also a key part in big data frameworks such as Apache Spark or Hadoop.

Recent publications have investigated join algorithms in the context of multi-core machines. Comparisons of sort-merge and hash join algorithms (Balkesen *et al.*, 2013; Barber *et al.*, 2014; Lang *et al.*, 2013; Manegold *et al.*, 2002) have resulted in several efficient algorithms for

both strategies and shown that the radix hash join outperforms sort-merge based algorithms. With respect to distributed hash joins over modern RDMA-capable networks, Barthels *et al.* (2017) reported one of the highest throughput numbers to date. The authors scale their implementation to thousands of cores reaching a throughput of 48.7 $\dot{\text{B}}$ tuples/s. Furthermore, previous work (Kara *et al.*, 2017; Wang *et al.*, 2015) has shown that FPGA-based high fanout data partitioning implementations can deliver high performance thanks to the algorithm's suitability to a pipelined design. Based on these works, we will present in the following an implementation making use of an FPGA-based SmartNIC to accelerate the radix hash join by offloading data partitioning.

Algorithm Overview. The radix hash join is a partitioned hash join (equi-join) in which both input relations are first divided into a large set of small partitions before, in a second stage, a hash table is constructed for each partition using the tuples of the smaller (inner) relation. In a third stage, these hash tables are probed using the partitioned data of the larger (outer) relation in order to find matching pairs. The goal of the partitioning phase is to ensure that the resulting partitions fit into the processor caches in order to speed up the subsequent build and probe phases. However, using a large partitioning fan-out can result in either cache thrashing, if the number of generated partitions is larger than the number of cache lines, and/or a high TLB miss ratio if the number of partitions is larger than the number of TLB entries (Manegold *et al.*, 2002). To that end, the radix hash join uses a multi-pass partitioning scheme. Each pass over the data looks at a subset of bits of the join key to determine the partition to which a tuple belongs. Each pass looks at a different non-overlapping subset of bits, and the amount of bits controls the fan-out of the pass and in turn limits the amount of TLB misses and cache thrashing. With these scheme, even large input relations can be subdivided into small cache-sized chunks within a couple of passes. In the radix hash join, the most expensive part of the computation is the partitioning, rather than the join itself.

Distributed Radix Hash Join. In the distributed case of the radix hash join the two input relations are equally partitioned across several

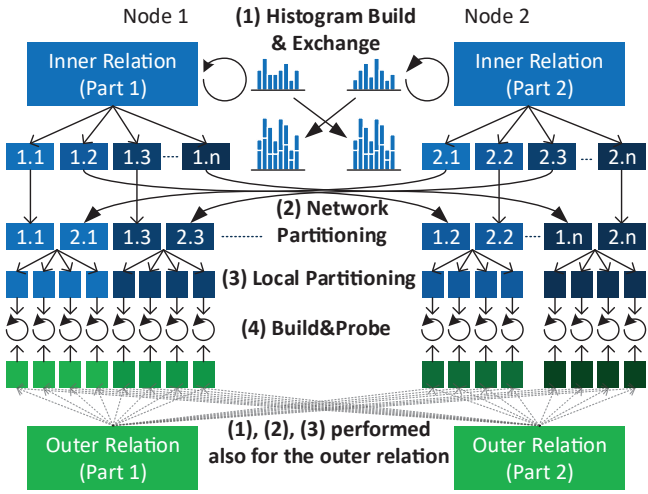


Figure 4.24: Distributed radix hash join over two nodes.

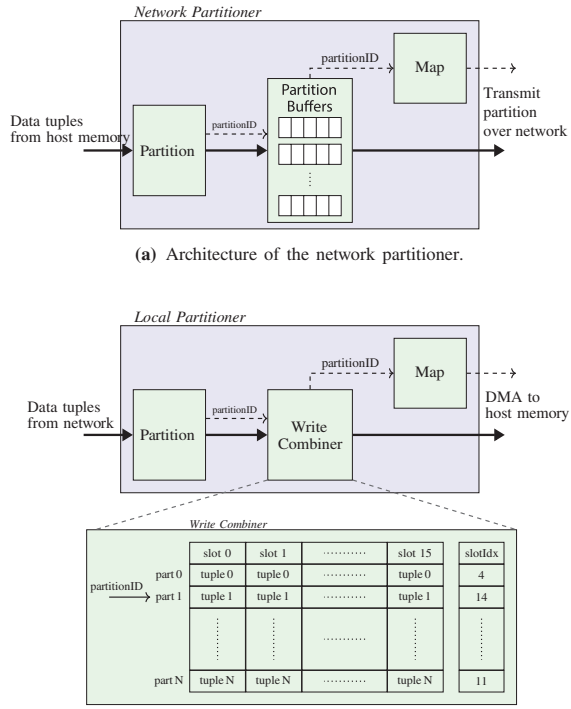
nodes such that for N nodes each node has $\frac{1}{N}$ data tuples. We consider an implementation using one-sided RDMA (Remote Direct Memory Access) (Barthels *et al.*, 2017) where nodes can directly access the memory of other nodes. The execution of the algorithm consists of four steps: (1) histogram computation, (2) network partitioning, (3) local partitioning, and (4) build and probe, as illustrated in Figure 4.24.

In the first step, each node calculates the histogram over the local data partitions. This local histogram is then exchanged with all other nodes to calculate a global histogram for each relation. The global histogram provides the size of every partition that will be created in the next step. In the second step, *network partitioning*, each node partitions its local data tuples. Thanks to the global histogram calculated in the first step, each node can process its input data independently and write the resulting partitions directly to their local or remote target memory address. This approach avoids unnecessary data copies and synchronization across the nodes. After all partitions of the first pass (*network partitioning*) have been created, the third step does another pass over the local partitions (*local partitioning*). This second pass looks at a different subset of bits in the join key to generate partitions that can fit into the cache of the processor to speed up the final step. After the

third step matching tuples from both the inner and outer relation have to be present in the same partition, since join keys belonging to different partitions differ in at least one of the bits used during the two radix partitioning phases. Because there is no dependency between partitions, in the forth step the construction and probing of the hash tables does not require thread synchronization nor inter-node communication such that all cores can be used efficiently. Matching tuples are outputted to a local buffer, which is given as input to the next down-stream operator in the query pipeline.

Design Overview. The two partitioning steps and the data exchange over the network represent a large part of the execution time of the distributed hash join. Using an FPGA-based SmartNIC supporting RDMA as presented by Sidler *et al.* (2020), the data partitioning can be implemented as a 'bump on the wire" operation. In particular data can be partitioned on-the-fly either when transmitted or received. The baseline implementation by Barthels *et al.* (2017) partitions the data into local buffers and transmits them to the target node when they reach a threshold. Due to the cheap hash function, this step is mostly bound by the memory bandwidth and the I/O to the NIC. This step can be offloaded to the NIC and data can be partitioned on-the-fly by the NIC during transmission. Figure 4.25(a) illustrates the *Network Partitioner* implemented on the FPGA-based SmartNIC. The incoming data tuples are partitioned using the same radix hash as in the baseline. For each partition tuples are accumulated in buffers and transmitted once they reach the size of an MTU (Maximum Transmission Unit). This buffering is necessary to achieve reasonable goodput when transmitting the tuples over the network. Offloading the data-intensive data partitioning to the SmartNIC eliminates one complete pass over the input data and data copies into the transmission buffers. As an additional benefit offloading frees up CPU cycles that can be assigned to other database operators.

To accelerate the third step of the algorithm, the *local partitioning*, data can be partitioned on-the-fly when received by the SmartNIC and before written to the host memory. In comparison to offloading of the *network partitioning*, less buffering is required to efficiently utilize the bandwidth of the PCIe link to the host. As such a similar approach to



(a) Architecture of the network partitioner.

(b) Architecture of the local partitioner.

Figure 4.25: Hardware modules to offload the *network partitioning* (a) and the *local partitioning* (b) phase to an FPGA-based NIC.

software implementations is used to combine tuples belonging to the same partition before writing them to host memory. Software implementations make use of SIMD registers in the CPU to accumulate up to 64 B of tuples. On the FPGA-based SmartNIC on-chip memory can be used to implement custom "SIMD" registers. As shown in Figure 4.25(b), the *Write Combiner* accumulates up to 128 B of tuples before writing them to host memory over PCIe.

In the baseline implementation the two partitioning steps are strictly sequential and cannot be overlapped. When they are offloaded to the SmartNIC data is partitioned while being transmitted and received and therefore the two steps are naturally overlapped increasing the efficiency

and utilization of the system. In addition the offloading reduces the load on the CPUs and the pressure on the memory subsystem.

Performance Properties. To evaluate the performance improvements of offloading the two partitioning steps to the NIC. We implemented the two partitioning modules (Figure 4.25) on an FPGA-based SmartNIC (Sidler *et al.*, 2020), supporting one-sided RDMA operations over RoCE v2. This SmartNIC represents the setup illustrated in Figure 3.3(C) such that the *Network Partitioner* and *Local Partitioner* are deployed on the data path between the RoCE stack and the DMA engine that accesses the host memory over PCIe. The card used is an Alpha Data ADM-PCIE-7V3 FPGA card. For the valuation the card was connected over PCIe 3 ×8 to the host machine and over 10 G Ethernet to the network. The distributed radix hash join was executed using two machines each equipped with two Intel Xeon E5-2630 with 8 CPU cores each. The input relations consists of 16 B tuples with an 8 B key and 8 B rid. The data is uniformly distributed across the two machines. The fan-out for the network partitioning is set to the number of nodes, in our case two, while the fan-out of the local partitioning is set to 1024 leading in total to 2048 partitions.

In a first experiment we compare the execution time of the radix hash join with and without offloading the two partitioning steps for input relation sizes ranging from 80–640 Mio. tuples corresponding to 2.4–19 GB, see Figure 4.26. The *network partitioning* is in both cases bound by the network link which is limited to 10 Gbit/s. As such it takes up a significant part of the execution time. We expect that a higher bandwidth network would decrease this phase proportionally. Given the overlap of the *network partitioning* and *local partitioning* in the case of offloading, the latter is completely hidden by the former. This overlap of the two steps directly translates into an overall reduction of execution time. The histogram and build&probe step of the algorithm are executed on the CPU as in the baseline implementation and therefore their execution time remains the same.

Similar to most related work, in the previous experiment we dedicated a full CPU socket with eight cores to the execution of the hash join. However, in an actual database system many queries have to be

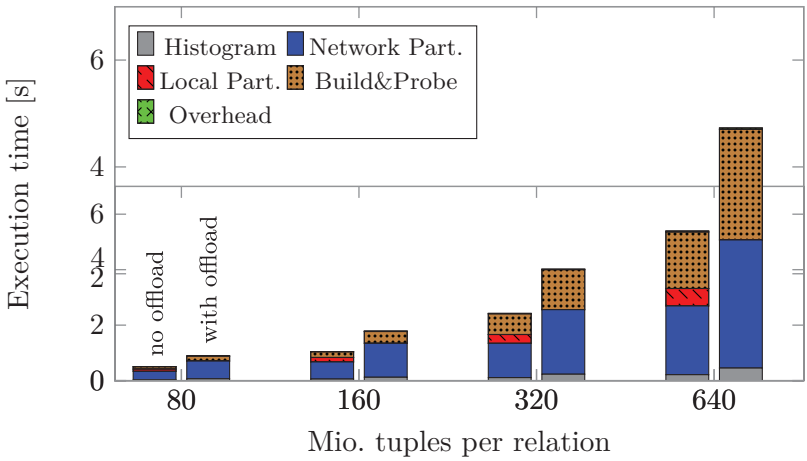


Figure 4.26: Distribute radix hash join using RoCE 10 G with and without offload.

served concurrently and it is unlikely that a full socket can be dedicated to the execution of a single hash join. We reduce the number of CPU cores assigned to the hash join and observe the impact on the execution time, see Figure 4.27. When the two partitioning phases are offload a reduction in CPU cores has a smaller impact, since only the build&probe phase can benefit from the parallelism on the CPU. Another observation is that in case of offloading the same performance can be achieved with half the CPU cores in comparison to the implementation without offload. This demonstrates the reduction in CPU load thanks to offloading.

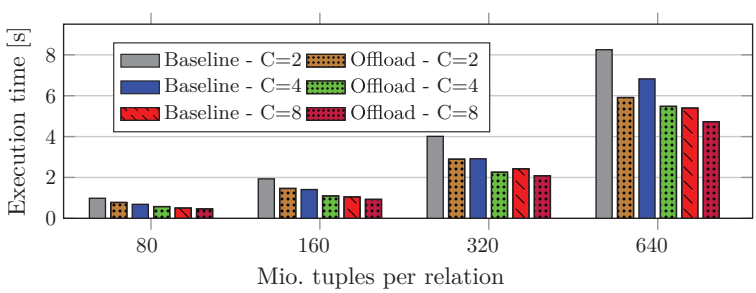


Figure 4.27: Distributed radix hash join execution with and without partitioning offload and varying the number of CPU cores.

5

Future Challenges and Architectures

5.1 Datacenter and Cloud Architecture Trends

There is a significant push in datacenters for disaggregated architectures, breaking up resources that traditionally resided within a single server, to be distributed over the network. This applies to storage (Do *et al.*, 2013; Klimovic *et al.*, 2016), memory (Dragojević *et al.*, 2014; Ousterhout *et al.*, 2010) and, now that they are gaining mainstream adoption, accelerators (Caulfield *et al.*, 2016; Weerasinghe *et al.*, 2015). The benefit of such disaggregation is increased elasticity for resource allocation and increased efficiency by physically specializing racks to different hardware types.

In the future it is expected that on-the-side accelerators, instead of being connected through PCIe in specific hosts, will be moved into more energy and space-efficient dense racks and accessed over fast interconnects. Projects such as the CloudFPGA at IBM (Weerasinghe *et al.*, 2015) shows that with fast networking, FPGAs can be accessed already with similar overheads to PCIe. With disaggregation, however, it is possible to assign such resources more flexibly to workloads. By densely packing FPGAs into dedicated racks, it is also possible to reduce energy consumption, deployment and maintenance costs.

In order to successfully integrate such disaggregated compute accelerators into analytics applications, the cloud/resource management software will have to provide methods of programming the FPGAs. There are already efforts in this direction and we expect that in the future cloud servers will have access on demand to a potentially large number of FPGAs working together on complex problems, similar to how Brainwave (Chung *et al.*, 2018) does it today behind the scenes.

5.2 Device Trends

The internal device technologies for reconfigurable processors keep evolving at a rapid pace and FPGAs are moving towards internal architectures with increased heterogeneity. For instance, hardened arithmetic units such as DSPs become more suitable for high-performance-computing, by inclusion of floating-point arithmetic capabilities in recent Intel FPGAs (Nurvitadhi *et al.*, 2019) and newer Xilinx FPGAs include high capacity dedicated memory blocks called Ultra-RAM (Boppana *et al.*, 2015), in addition to regular BRAM, with the purpose of increasing on-chip caching capability.

The recent Xilinx ACAP (Adaptive Computing Platform) (Gaide *et al.*, 2019) device is a move towards more course-grained reconfigurability: ACAP contains so called AI engines, which can be seen as the next step in the evolution of the DSP, with SIMD and VLIW capabilities. In addition to the AI engines, there is the usual FPGA-fabric providing fine grained reconfigurability. The course-grained architecture sacrifices some of the flexibility of an FPGA, but provides increased efficiency for dedicated compute tasks such as machine learning or video processing.

A further area where reconfigurable accelerators are rapidly being improved is better interconnectivity. This is mainly driven by the increasing usage of FPGAs in datacenters (Putnam *et al.*, 2014) and the requirements on consuming and producing larger amounts of data. The data interconnectivity of FPGAs used in datacenters can be categorized in three classes, where current advances are focusing on:

(1) Host CPU facing interface: Although the majority of the systems still rely on PCIe to attach an FPGA to a host CPU, recently

cache-coherent interfaces have been developed to lower the latency and increase the bandwidth for this interface. One early example of this is the Intel Xeon+FPGA (Gupta, 2015) that we have used in multiple projects as shown in previous sections. OpenCAPI from IBM (OpenCAPI OC-Accel, n.d.) is another example that aims to attach FPGAs coherently to a POWER CPU.

(2) Local main memory: This interface is used mainly to store intermediate data and usually realized as off-chip DIMMs attached to the FPGA. Recently, to provide much higher bandwidth to this interface, Xilinx has released FPGAs with in-package HBM with a capacity of up to 8 GBs. Theoretically, the HBM provides up to 400 GB/s bandwidth to a single FPGA, nearly 20× faster than single channel DDR4-based solutions. To take advantage of HBM that exposes a very wide bus to the FPGA (8192-bits), Xilinx includes a hardened crossbar that is clocked much faster (900 MHz) than typical FPGA fabric clock frequencies.

(3) Network facing interface: FPGAs in datacenters are often used as network offload engines and vendors are emphasizing this capability by improving the hardened network functionality. For instance, recent ACAP devices from Xilinx contain 100 G Ethernet blocks. However, network and transport layer functionality is still missing as hardened circuitry and must be implemented using the FPGA fabric itself (Sidler *et al.*, 2015). Other FPGA-based networking solutions include SmartNICs from Mellanox, however in those devices the FPGA-fabric is not meant for high-end compute acceleration, but rather for packet preprocessing tasks.

While the interconnectivity of FPGAs is important especially within the datacenter, the on-chip connectivity is another area that requires improvement. It is essential to be able to feed the compute resources on the FPGA with enough data while keeping the clock frequencies high. As FPGAs get larger and interconnectivity busses get wider (e.g., 8192-bits from HBM), bit-wise routing becomes challenging and tends to limit the clock frequency for many applications. One example of this is the so-called Super-Logic-Regions (SLR) in recent Xilinx

FPGAs. An SLR denotes a same-die FPGA fabric and the final FPGA package is built from multiple SLRs that are connected with each other. While the final FPGA can thus be larger without suffering from decreasing manufacturing yields, designs crossing between SLRs are forced to have lower clock frequencies, because signals take longer time to propagate through SLR crossings. A recent FPGA architecture from Achronix (Achronix Vectorpath with Speedster7t FPGAs, n.d.) provides a hardened network-on-chip and vector-routing, promising to alleviate this problem. Similarly, Xilinx's ACAP has a network-on-chip to move data efficiently between hardened AI engines and the FPGA fabric.

5.3 Sharing FPGAs and Accelerators

Many of the applications described in this monograph take full control of the underlying FPGA accelerator device, even if in time, multiple applications can use it (time multiplexing). The challenge of sharing FPGAs and similar devices across multiple cloud tenants or data-center applications at the same time is the "spatial" aspect of their programming.

Figure 5.1 depicts the two main ways that FPGAs could be space multiplexed. First, by relying on several programmable regions, one can deploy several different acceleration modules, which keeps the FPGA

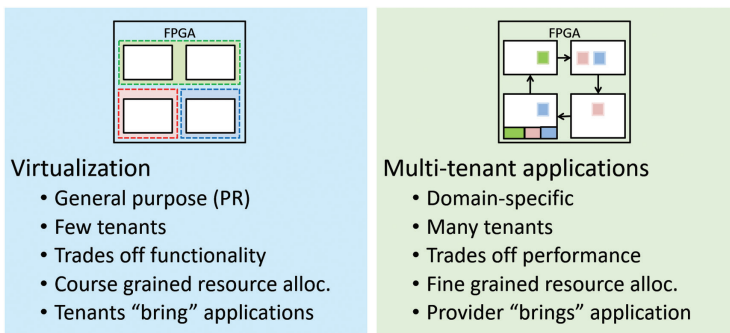


Figure 5.1: FPGAs can be shared by multiple tenants or applications at the same time in two ways: by partitioning the available programmable area (virtualization) or by designing functionality that is fixed but can be shared (multi-tenant modules).

general purpose. There are several examples of such approached, including, Centaur (Owaida *et al.*, 2017), the work by Vaishnav *et al.* (2019), ReconOs (Lübbers and Platzner, 2009), etc., but they typically suffer from the limitation that by accommodating more programmable regions, the size of each region shrinks, limiting the functionality of each tenant. Allocation of the FPGA is also very coarse grained and even though we refer to it as "virtualization", the modules still need to be compiled for the specific device and partial reconfiguration region characteristics. Overall, this approach also puts the burden on the application developers to provide the blocks that will be run on the FPGA.

In contrast, FPGAs could also be shared by designing functionality on them that multiplexes the on-chip state across several tenants but keeps the processing the same. One such example is Multes (István *et al.*, 2018) that is a multi-tenant key-value store offering data and performance isolation for tenants. As Figure 5.1, such an approach permits functionality-rich solutions to be shared by potentially tens of tenants, it also restricts the accelerator a specific type of application (e.g., key-value processing, inference accelerator, etc.). To some extent it can also result in nominally lower performance than single-tenant variants of the same logic due to meta-data swapping but, when compared to the cost of multi-tenancy in software applications, the overhead is negligible. This approach also allows a service provider to design and maintain the FPGA functionality, exposing it to a large number of client applications.

In DoppioDB (Kara *et al.*, 2020) we explored the "virtualized" approach for sharing the FPGA across several queries of the same database instance. The FPGA is configured with several "slots" that can be filled in using partial reconfiguration (we call these slots hardware threads because the interface to them in software is similar to a function call on a new thread). In the context of DoppioDB, the operators that we designed were able to saturate the memory bandwidth without occupying the full area of the FPGA, so having several hardware threads allowed the database to use the FPGA in a more varied set of queries without having to reconfigure the device often. Ideally, however, what most acceleration scenarios would benefit from, is much cheaper partial reconfiguration (PR). If the time to perform PR could be reduced to the sub-millisecond level, more acceleration opportunities would open

up and the barrier to entry would also be lowered. For this, however, the major FPGA vendors will have to improve their tools and libraries, as well as, potentially change the way the FPGA is programmed at the lowest level.

5.4 Programming FPGAs

It is clear that, in the long run, databases must find ways to adapt to the idea of incorporating specialized hardware whose functionality will change over time, preferably even with the arrival of queries. This brings us to the second big challenge of better FPGA integration in databases, namely how to express acceleration functionality that allows quicker development time, or perhaps even techniques such as code generation and just-in-time compilation.

As opposed to CPUs or GPUs where the architecture (ISA, caches, etc.) is fixed, in an FPGA it is not. This adds a layer of complexity to the problem of compiling operators, as well as query planning in general. Given even just the heterogeneity of modern CPUs and their different SIMD units, there is already a push for databases to incorporate more and more compiler ideas (Pirk *et al.*, 2016, 2019).

The side effect of bringing more ideas from compilers into databases is that it will likely also be easier to integrate DSLs for hardware accelerators (Da Silva *et al.*, 2016; Koeplinger *et al.*, 2018; OneAPI, 2020) into the database. However, many of these solutions are targeting compute kernels written in languages that are a better fit for HPC and machine learning type functionality than database operations (OneAPI, 2020). Therefore, novel ideas are needed that bridge the space between databases and languages/compilers for specialized hardware. One possible direction to explore is related to the design of the Spatial language and compiler (Koeplinger *et al.*, 2018). Spatial approaches the problem of writing parallel code for accelerators in a way that accounts for the fact that circuits are physically laid out on the chip. Given that query plans are often composed by a set of sub-operators that are parameterized differently to implement, for instance, different join types, these could be an intermediate step between SQL and hardware circuit that

allows the database to offload a pipeline of such sub-operators to the FPGA in an automated manner.

Another aspect that makes translating operators to hardware-based accelerators challenging comes from the fact that not all functionality will fit on the device. This is true regardless whether we target an FPGA, a P4-based switch or SmartNIC, or an ASIC-based solution such as the DAX. Therefore, even if the best case of an operator can be efficiently translated to hardware, corner cases will have to be handled without significantly impacting performance. For this reason, the challenge of compilation is also related to the ideas discussed before around hybrid execution and query planning. Frameworks that compile queries to such platforms will have to provide software-based post-processing functionality to ensure that corner cases are gracefully handled. The challenge in this hybrid computation is to find suitable points where to split the functionality in an automated way.

6

Closing Remarks

This monograph provided a high level overview of FPGA-based accelerators and their role within analytical databases. With the slowdown of Moore's law, specialized hardware is increasingly attractive as a way of reducing computational bottlenecks and, thanks to recent technological advances, FPGAs are becoming easier to integrate in large software systems.

The monograph highlights several representative database designs that incorporate FPGAs, either as traditional accelerator or as an in-data-path ones. The deep dive into operator design summarized the main design guidelines for FPGAs and illustrated how they can be used in the context of core SQL operator offloading and to improve the performance of emerging machine learning operators.

The outlook for FPGAs, and accelerators in general, is optimistic. In the light of the increasing disaggregation trends in datacenters, emerging memory technologies and the better software support from vendors to deploy applications across heterogeneous architectures, in the future, it will be easier to integrate FPGAs in software systems. Nonetheless, as we highlight, there are still several challenges to be overcome, most notably that of sharing FPGAs more efficiently and programming them more easily.

References

Achronix Vectorpath with Speedster7t FPGAs (n.d.). https://www.achronix.com/vectorpath/.

Agron, J. and D. Andrews (2009). "Building heterogeneous reconfigurable systems with a hardware microkernel". In: *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. ACM. 393–402.

Aho, A. V. and J. D. Ullman (1992). *Foundations of Computer Science*. Computer Science Press.

Alonso, G., Z. Istvan, K. Kara, M. Owaida, and D. Sidler (2019). "doppioDB 1.0: Machine learning inside a relational engine". *IEEE Data Engineering Bulletin*. 42(2): 19–31.

Amazon F1 Instances (n.d.). aws.amazon.com/ec2/instance-types/f1/.

Asiatici, M., N. George, K. Vipin, S. A. Fahmy, and P. Ienne (2017). "Virtualized execution runtime for FPGA accelerators in the cloud". *IEEE Access*. 5: 1900–1910.

Balkesen, C., J. Teubner, G. Alonso, and M. T. Özsu (2013). "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware". In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 362–373.

Barber, R., G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe (2014). "Memory-efficient hash joins". *Proceedings of the VLDB Endowment*. 8(4): 353–364.

Barthels, C., S. Loesing, G. Alonso, and D. Kossmann (2015). "Rack-scale in-memory join processing using RDMA". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 1463–1475.

Barthels, C., I. Müller, T. Schneider, G. Alonso, and T. Hoefler (2017). "Distributed join algorithms on thousands of cores". *Proceedings of the VLDB Endowment*. 10(5): 517–528.

Bispo, J., I. Sourdis, J. M. Cardoso, and S. Vassiliadis (2006). "Regular expression matching for reconfigurable packet inspection". In: *2006 IEEE International Conference on Field Programmable Technology*. IEEE. 119–126.

Blott, M., K. Karras, L. Liu, K. Vissers, J. Bär, and Z. Istvàn (2013). "Achieving 10 Gbps line-rate key-value stores with FPGAs". In: *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 13)*.

Böhm, C., M. Perdacher, and C. Plant (2017). "Multi-core K-means". In: *Proceedings of the 2017 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics. 273–281.

Boppana, V., S. Ahmad, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig (2015). "UltraScale+ MPSoC and FPGA families". In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE. 1–37.

Bosshart, P., D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker (2014). "P4: Programming protocol-independent packet processors". *ACM SIGCOMM Computer Communication Review*. 44(3): 87–95.

Bottou, L. (2010). "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer. 177–186.

Byma, S., J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow (2014). "FPGAs in the cloud: Booting virtualized hardware accelerators with Openstack". In: *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 109–116.

Casper, J. and K. Olukotun (2014). "Hardware acceleration of database operations". In: *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 151–160.

Caulfield, A. M., E. S. Chung, A. Putnam, H. Angepat, I. Fowers, M. Haselman, S. Heil, M. Humphret, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger (2016). "A cloud-scale acceleration architecture". In: *MICRO-49: The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 1–13.

Chalamalasetti, S. R., K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala (2013). "An FPGA memcached appliance". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM. 245–254.

Chen, F., Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang (2014). "Enabling FPGAs in the cloud". In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM. 3.

Chung, E., J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, and M. Abeydeera (2018). "Serving DNNs in real time at datacenter scale with project brainwave". *IEEE Micro*. 38(2): 8–20.

Da Silva, H. C., F. Pisani, and E. Borin (2016). "A comparative study of SYCL, OpenCL, and OpenMP". In: *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE. 61–66.

Dennl, C., D. Ziener, and J. Teich (2012). "On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library". In: *2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 45–52.

Dennl, C., D. Ziener, and J. Teich (2013). "Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration". In: *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 25–28.

Dhanachandra, N., K. Manglem, and Y. J. Chanu (2015). "Image segmentation using K-means clustering algorithm and subtractive clustering algorithm". *Procedia Computer Science*. 54: 764–771.

Dhawan, U. and A. DeHon (2015). "Area-efficient near-associative memories on FPGAs". *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*. 7(4): 30.

Do, J., Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt (2013). "Query processing on smart SSDs: Opportunities and challenges". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 1221–1230.

Dragojević, A., D. Narayanan, O. Hodson, and M. Castro (2014). "Farm: Fast remote memory". In: *11th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association. 401–414.

Dragojević, A., D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro (2015). "No compromises: Distributed transactions with consistency, availability, and performance". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 54–70.

Eran, H., L. Zeno, M. Tork, G. Malka, and M. Silberstein (2019). "NICA: An infrastructure for inline acceleration of network applications". In: *2019 USENIX Annual Technical Conference*. 345–362.

Esmaeilzadeh, H., E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger (2011). "Dark silicon and the end of multicore scaling". In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 365–376.

Estlick, M., M. Leeser, J. Theiler, and J. J. Szymanski (2001). "Algorithmic transformations in the implementation of K-means clustering on reconfigurable hardware". In: *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*. ACM. 103–110.

Firestone, D. (2016). "SmartNIC: Accelerating azure's network with FPGAs on OCS servers". In: *Open Compute Summit 2016*.

Francisco, P. (2011). "The netezza data appliance architecture: A platform for high performance data warehousing and analytics." *IBM Redbooks*.

Fukuda, E. S., H. Inoue, T. Takenaka, D. Kim, T. Sadahisa, T. Asai, and M. Motomura (2014). "Caching memcached at reconfigurable network interface". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 1–6.

Gaide, B., D. Gaitonde, C. Ravishankar, and T. Bauer (2019). "Xilinx adaptive compute acceleration platform: VersalTM architecture". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 84–93.

Gao, M. and C. Kozyrakis (2016). "HRL: Efficient and flexible reconfigurable logic for near-data processing". In: *IEEE International Symposium on High-Performance Computer Architecture*. IEEE. 126–137.

Gokhale, M., J. Frigo, K. Mccabe, J. Theiler, C. Wolinski, and D. Lavenier (2003). "Experience with a hybrid processor: K-means clustering". *The Journal of Supercomputing*. 26(2): 131–148.

Gupta, P. K. (2015). "Xeon+FPGA platform for the data center". In: *Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*. Vol. 119.

Gupta, S., A. Agrawal, K. Gopalakrishnan, and P. Narayanan (2015). "Deep learning with limited numerical precision". In: *International Conference on Machine Learning*. 1737–1746.

Hadian, A. and S. Shahrivari (2014). "High performance parallel K-means clustering for disk-resident datasets on multi-core CPUs". *The Journal of Supercomputing*. 69(2): 845–863.

Happe, M., A. Traber, and A. Keller (2015). "Preemptive hardware multitasking in ReconOS". In: *International Symposium on Applied Reconfigurable Computing*. Springer. 79–90.

He, Z., D. Sidler, Z. István, and G. Alonso (2018). "A flexible K-means operator for hybrid databases". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 368–3683.

IBM/Netezza (2011). "The netezza data appliance architecture: A platform for high performance data warehousing and analytics". http://www.redbooks.ibm.com/abstracts/redp4725.html.

Ismail, A. and L. Shannon (2011). "FUSE: Front-end user framework for O/S abstraction of hardware accelerators". In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 170–177.

Istvan, Z., L. Woods, and G. Alonso (2014). "Histograms as a side effect of data movement for big data". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM. 1567–1578.

István, Z., G. Alonso, M. Blott, and K. Vissers (2015). "A hash table for line-rate data processing". *ACM TRETS*. 8(2): 13.1–13.15.

István, Z., D. Sidler, G. Alonso, and M. Vukolic (2016). "Consensus in a box: Inexpensive coordination in hardware". In: *13th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association. 425–438.

István, Z., D. Sidler, and G. Alonso (2017). "Caribou: Intelligent distributed storage". *Proceedings of the VLDB Endowment*. 10(11): 1202–1213.

István, Z., G. Alonso, and A. Singla (2018). "Providing multi-tenant services with FPGAs: Case study on a key-value store". In: *28th International Conference on Field Programmable Logic and Applications*. IEEE. 119–124.

Iturbe, X., K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, I. Martinez, T. Arslan, and J. Perez (2013). "R3TOS: A novel reliable reconfigurable real-time operating system for highly adaptive, efficient, and dependable computing on FPGAs". *IEEE Transactions on Computers*. 62(8): 1542–1556.

Jamesmanoharan, J., S. H. Ganesh, M. L. P. Felciah, and A. K. Shafreenbanu (2014). "Discovering students' academic performance based on GPA using K-means clustering algorithm". In: *World Congress on Computing and Communication Technologies*. IEEE. 200–202.

Jo, I., D.-H. Bae, A. S. Yoon, J. U. Kang, S. Cho, D. D. Lee, and J. Jeong (2016). "YourSQL: A high-performance database system leveraging in-storage computing". *Proceedings of the VLDB Endowment*. 9(12): 924–935.

Jun, S.-W., M. Liu, and K. E. Fleming (2014). "Scalable multi-access flash store for big data analytics". In: *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 55–64.

Junqueira, F. P., B. C. Reed, and M. Serafini (2011). "Zab: High-performance broadcast for primary-backup systems". In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE. 245–256.

Kalia, A., M. Kaminsky, and D. G. Andersen (2014). "Using RDMA efficiently for key-value services". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. 295–306.

Kara, K., D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang (2017). "FPGA-accelerated dense linear machine learning: A precision-convergence trade-off". In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 160–167.

Kara, K., Z. Wang, C. Zhang, and G. Alonso (2020). "doppioDB 2.0: Hardware techniques for improved integration of machine learning into databases". *Proceedings of the VLDB Endowment*. 12(12): 1818–1821.

Kim, J. K., Z. Zhang, and J. A. Fessler (2011). "Hardware acceleration of iterative image reconstruction for X-ray computed tomography". In: *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 1697–1700.

Klimovic, A., C. Kozyrakis, E. Thereksa, B. John, and S. Kumar (2016). "Flash storage disaggregation". In: *Proceedings of the Eleventh European Conference on Computer Systems*. ACM. 1–15.

Knodel, O., P. R. Genssler, and R. G. Spallek (2017). "Migration of long-running tasks between reconfigurable resources using virtualization". *ACM SIGARCH Computer Architecture News*. 44(4): 56–61.

Koeplinger, D., M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun (2018). "Spatial: A language and compiler for application accelerators". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 296–311.

Koo, G., K. K. Matam, I. Te, H. V. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram (2017). "Summarizer: Trading communication with computing near storage". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-50 '17*. Cambridge, Massachusetts. 219–231.

Kuhring, L., E. Garcia, and Z. István (2019). "Specialize in moderation—building application-aware storage services using FPGAs in the datacenter". In: *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. USENIX Association.

Kumm, M. and P. Zipf (2014). "Pipelined compressor tree optimization using integer linear programming". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 1–8.

Lang, H., V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper (2013). "Massively parallel NUMA-aware hash joins". In: *In Memory Data Management and Analysis*. Springer. 3–14.

Lavasani, M., H. Angepat, and D. Chiou (2014). "An FPGA-based in-line accelerator for memcached". *IEEE Computer Architecture Letters*. 13(2): 57–60.

Lesser, B., M. Mücke, and W. N. Gansterer (2011). "Effects of reduced precision on floating-point SVM classification accuracy". *Procedia Computer Science*. 4: 508–517.

Li, Y., K. Zhao, X. Chu, and J. Liu (2013). "Speeding up K-means algorithm by GPUs". *Journal of Computer and System Sciences*. 79(2): 216–229.

Li, B., Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang (2017). "KV-direct: High-performance in-memory key-value store with programmable NIC". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 137–152.

Liu, W.-C., J.-L. Huang, and M. S. Chen (2005). "Kacu: K-means with hardware centroid-updating". In: *Conference, Emerging Information Technology 2005.* IEEE.

Loesing, S., M. Pilman, T. Etter, and D. Kossmann (2015). "On the design and scalability of distributed shared-data databases". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 663–676.

Lübbers, E. and M. Platzner (2009). "ReconOS: Multithreaded programming for reconfigurable computers". *ACM Transactions on Embedded Computing Systems (TECS)*. 9(1): 8.

Mai, L., L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf (2014). "NetAgg: Using middleboxes for application-specific on-path aggregation in data centres". In: *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies*. 249–262.

Manegold, S., P. Boncz, and M. Kersten (2002). "Optimizing main-memory join on modern hardware". *IEEE TKDE*. 14(4): 709–730.

Mitchell, C., Y. Geng, and J. Li (2013). "Using one-sided RDMA reads to build a fast, CPU-efficient key-value store". In: *USENIX Annual Technical Conference*. USENIX Association. 103–114.

Morales-Villanueva, A., R. Kumar, and A. Gordon-Ross (2016). "Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable FPGAs". In: *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 1505–1508.

Mücke, M., B. Lesser, and W. N. Gansterer (2010). "Peak performance model for a custom precision floating-point dot product on FPGAs". In: *European Conference on Parallel Processing*. Springer. 399–406.

Nakahara, H., T. Sasao, and M. Matsuura (2011). "A regular expression matching circuit based on a decomposed automaton". In: *Reconfigurable Computing: Architectures, Tools and Applications*. Springer.

Nurvitadhi, E., D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar, and R. Krishnamurthy (2019). "Evaluating and enhancing intel® stratix® 10 FPGAs for persistent real-time AI". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 119–119.

Oliver, N., R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, and H. Mitchel (2011). "A reconfigurable computing system based on a cache-coherent fabric". In: *2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE. 80–85.

OneAPI (2020). "The OneAPI specification". URL: www.oneapi.com/spec/ (accessed 06/2020).

OpenCAPI OC-Accel (n.d.). https://github.com/OpenCAPI/oc-accel.

Oracle (2015). "Software in silicon: What it does and why". https://community.oracle.com/docs/DOC-932216.

Oskin, M., F. T. Chong, and T. Sherwood (1998). *Active Pages: A Computation Model for Intelligent Memory*. Vol. 26. IEEE Computer Society.

Ousterhout, J., P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Maziéres, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, and S. M. Rumble (2010). "The case for RAMClouds: Scalable high-performance storage entirely in DRAM". *ACM SIGOPS Operating Systems Review*. 43(4): 92–105.

Owaida, M., D. Sidler, K. Kara, and G. Alonso (2017). "Centaur: A framework for hybrid CPU-FPGA databases". In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 211–218.

Perez-Garcia, A. N., G. M. Tornez-Xavier, L. M. Flores-Nava, F. Gómez-Castañeda, and J. A. Moreno-Cadenas (2014). "Multilayer perceptron network with integrated training algorithm in FPGA". In: *2014 11th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*. IEEE. 1–6.

Pirk, H., J. Giceva, and P. Pietzuch (2019). "Thriving in the no man's land between compilers and databases".

Pirk, H., O. Moll, M. Zaharia, and S. Madden (2016). "Voodoo-a vector algebra for portable database performance on modern hardware". *Proceedings of the VLDB Endowment*. 9(14): 1707–1718.

Plattner, C. and G. Alonso (2004). "Ganymed: Scalable replication for transactional web applications". In: *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Berlin, Heidelberg: Springer. 155–174.

Polychroniou, O. and K. A. Ross (2014). "A comprehensive study of main-memory partitioning and its application to large-scale comparison and radix-sort". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM. 755–766.

Putnam, A. (2014). "Large-scale reconfigurable computing in a microsoft datacenter". In: *Hot Chips 26 Symposium (HCS), 2014 IEEE*. IEEE. 1–38.

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

Rabieah, M. B. and C.-S. Bouganis (2016). "FPGASVM: A framework for accelerating kernelized support vector machine". In: *Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*. 68–84.

Recht, B., C. Re, S. Wright, and F. Niu (2011). "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in Neural Information Processing Systems*. 693–701.

Root, C. and T. Mostak (2016). "MapD: A GPU-powered big data analytics and visualization platform". In: *ACM SIGGRAPH 2016 Talks*. 1–2.

Saegusa, T. and T. Maruyama (2006). "An FPGA implementation of K-means clustering for color images based on Kd-tree". In: *2006 International Conference on Field Programmable Logic and Applications*. IEEE. 1–6.

Salami, B., G. A. Malazgirt, O. Arcas-Abella, A. Yurdakul, and N. Sonmez (2017). "AxleDB: A novel programmable query processing platform on FPGA". *Microprocessors and Microsystems*. 51: 142–164.

Schuh, S., X. Chen, and J. Dittrich (2016). "An experimental comparison of thirteen relational equi-joins in main memory". In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 1961–1976.

Schuhknecht, F. M., P. Khanchandani, and J. Dittrich (2015). "On the surprising difficulty of simple things: The case of radix partitioning". *Proceedings of the VLDB Endowment*. 8(9): 934–937.

Sidler, D., G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley (2015). "Scalable 10 Gbps TCP/IP stack architecture for reconfigurable hardware". In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 36–43.

Sidler, D., Z. István, and G. Alonso (2016). "Low-latency TCP/IP stack for data center applications". In: *26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 1–4.

Sidler, D., Z. István, M. Owaida, and G. Alonso (2017). "Accelerating pattern matching queries in hybrid CPU-FPGA architectures". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM. 403–415.

Sidler, D., Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso (2020). "StRoM: Smart remote memory". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM.

Sitaridi, E. and K. Ross (2016). "GPU-accelerated string matching for database applications". *The VLDB Journal*. 25: 719–740.

Sukhwani, B., H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad (2012). "Database analytics acceleration using FPGAs". In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. ACM. 411–420.

Sukhwani, B., H. Min, M. Thoennes, P. Dube, B. Brezzo, S. Asaad, and D. E. Dillenberger (2013). "Database analytics: A reconfigurable-computing approach". *IEEE Micro*. 34(1): 19–29.

Tang, Q. Y. and M. A. Khalid (2016). "Acceleration of K-means algorithm using Altera SDK for OpenCL". *ACM TRETS*. 10(1): 6.

Tang, Z., K. Liu, J. Xiao, L. Yang, and Z. Xiao (2017). "A parallel k-means clustering algorithm based on redundance elimination and extreme points optimization employing MapReduce". *Concurrency and Computation: Practice and Experience*. 29(20): e4109.

Taylor, M. B. (2012). "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse". In: *DAC Design Automation Conference 2012*. IEEE. 1131–1136.

Teubner, J. and L. Woods (2013). "Data processing on FPGAs". *Morgan & Claypool Synthesis Lectures on Data Management*. 5(2): 1–118.

Teubner, J., L. Woods, and C. Nie (2012). "Skeleton automata for FPGAs: Reconfiguring without reconstructing". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 229–240.

Thinh, T. N., S. Kittitornkun, and S. Tomiyama (2007). "Applying cuckoo hashing for FPGA-based pattern matching in NIDS/NIPS". In: *2007 International Conference on Field-Programmable Technology*. IEEE. 121–128.

Vaishnav, A., K. D. Pham, and D. Koch (2018). "A survey on FPGA virtualization". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 131–1317.

Vaishnav, A., K. D. Pham, and D. Koch (2019). "Heterogeneous resource-elastic scheduling for CPU+FPGA architectures". In: *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. 1–6.

Wang, X. and M. Leeser (2007). "K-Means clustering for multispectral images using floating-point divide". In: *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. IEEE. 151–162.

Wang, Z., B. He, and W. Zhang (2015). "A study of data partitioning on OpenCL-based FPGAs". In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 1–8.

Wang, Z., J. Paul, H. Y. Cheah, B. He, and W. Zhang (2016a). "Relational query processing on OpenCL-based FPGAs". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

Wang, Z., S. Zhang, B. He, and W. Zhang (2016b). "Melia: A MapReduce framework on OpenCL-based FPGAs". *IEEE Transactions on Parallel and Distributed Systems*. 27(12): 3547–3560.

Weerasinghe, J., F. Abel, C. Hagleitner, and A. Herkersdorf (2015). "Enabling FPGAs in hyperscale data centers". In: *2015 IEEE 12th Intl. Conf. on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl. Conf. on Autonomic and Trusted Computing and 2015 IEEE 15th Intl. Conf. on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. IEEE. 1078–1086.

Weerasinghe, J., R. Polig, F. Abel, and C. Hagleitner (2016). "Network-attached FPGAs for data center applications". In: *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE. 36–43.

Weiss, R. (2012). "A technical overview of the oracle exadata database machine and exadata storage server". *Oracle White Paper*. Oracle Corporation, Redwood Shores.

Woods, L., J. Teubner, and G. Alonso (2010). "Complex event detection at wire speed with FPGAs". *Proceedings of the VLDB Endowment*. 3(1–2): 660–669.

Woods, L., G. Alonso, and J. Teubner (2013). "Parallel computation of skyline queries". In: *IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 1–8.

Woods, L., Z. István, and G. Alonso (2014). "Ibex – An intelligent storage engine with support for advanced SQL off-loading". *PVLDB*. 7(11): 963–974.

Woods, L., G. Alonso, and J. Teubner (2015). "Parallelizing data processing on FPGAs with shifter lists". *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*. 8(2): Article 7.

Xu, S., S. Lee, S.-W. Jun, M. Liu, and J. Hicks (2016). "BlueCache: A scalable distributed flash-based key-value store". *Proceedings of the VLDB Endowment*. 10(4): 301–312.

Yang, Y.-H. E., W. Jiang, and V. K. Prasanna (2008). "Compact architecture for high-throughput regular expression matching on FPGA". In: *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM. 30–39.

You, Y., X. Lian, J. Liu, H.-F. Yu, I. S. Dhillon, J. Demmel, and C.-J. Hsieh (2016). "Asynchronous parallel greedy coordinate descent". In: *Advances in Neural Information Processing Systems*. 4682–4690.

Zamanian, E., C. Binnig, T. Harris, and T. Kraska (2017). "The end of a myth: Distributed transactions can scale". *Proceedings of the VLDB Endowment*. 10(6): 685–696.

Zhang, C., R. Chen, and V. Prasanna (2016). "High throughput large scale sorting on a CPU-FPGA heterogeneous platform". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 148–155.

Zhang, H., J. Li, K. Kara, D. Alistarh, J. Liu, and C. Zhang (2017a). "Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR.org. 4035–4043.

Zhang, J., Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, and T. Moscibroda (2017b). "The feniks FPGA operating system for cloud computing". In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. ACM. 22.

Zhu, Z., A. X. Liu, F. Zhang, and F. Chen (2018). "FPGA resource pooling in cloud computing". *IEEE Transactions on Cloud Computing*. DOI: 10.1109/TCC.2018.2874011.