A Glass Half Full: Using Programmable Hardware Accelerators in Database Analytics

Zsolt István
IMDEA Software Institute, Madrid, Spain
{first.lastname@imdea.org}

Abstract

Even though there have been a large number of proposals to accelerate databases using specialized hardware, often the opinion of the community is pessimistic: the performance and energy efficiency benefits of specialization are seen to be outweighed by the limitations of the proposed solutions and the additional complexity of including specialized hardware, such as field programmable gate arrays (FPGAs), in servers. Recently, however, as an effect of stagnating CPU performance, server architectures started to incorporate various programmable hardware components, ranging from smart network interface cards, through SSDs with offloading capabilities, to near-CPU accelerators. This availability of heterogeneous hardware brings opportunities to databases and we make the case that there is cause for optimism. In the light of a shifting hardware landscape and emerging analytics workloads, it is time to revisit our stance on hardware acceleration.

In this paper we highlight several challenges that have traditionally hindered the deployment of hardware acceleration in databases and explain how they have been alleviated or removed altogether by recent research results and the changing hardware landscape. We also highlight that, now that these challenges have been addressed, a new set of questions emerge around deep integration of heterogeneous programmable hardware in tomorrow's databases, for which answers can likely be found only in collaboration with researchers from other fields.

1 Introduction

There is a rich history of projects aiming to specialize parts of, or entire, computers to databases. Notable examples include the Database Machine from the seventies [1], Gamma [2], the Netezza data appliance [3], the Q100 DB processor [4], and Oracle Rapid [5] most recently. These works demonstrate dramatically increased energy efficiency and better performance thanks to a hardware/software co-design approach. However, CPUs, until very recently, enjoyed a performance scaling in line with Moore's law and the time and effort of designing and delivering specialized hardware was not economical. This changed with the stagnation in CPU performance [6] in the last decade and the simultaneous increase in networking speeds that has created a clear need for hardware acceleration.

Initially, the move to the cloud worked against hardware acceleration for databases due to the cloud's reliance on commodity hardware and the need to cater to many different users and applications. In the meantime, however, new data-intensive workloads emerged in the cloud (most notably machine learning), that suffered from stagnating CPU performance and could benefit from various types of compute or networking acceleration. If we look at today's cloud offering and datacenters, an exciting, heterogeneous, landscape emerges: Machine learning workloads in the Google Cloud are accelerated with Tensor Processing Units (TPUs) [8], increasing energy efficiency by at least an order of magnitude when compared to GPUs. Amazon, Baidu and Huawei all offer Field Programmable Gate Arrays (FPGAs) by the hour in their cloud to users[1] to implement custom accelerators. Microsoft has been also deploying FPGAs in the Azure Cloud, to accelerate their infrastructure and machine learning pipelines, in their Project Catapult [7]. Furthermore, Intel has been experimenting with including small programmable elements on their Xeon CPUs [9] that can be tailored to the compute-intensive task at hand.

The recent developments discussed above mean that multi-purpose programmable hardware accelerators are entering the mainstream and, from the point of view of the database, they can be exploited without having to incur additional cost for deployment. Specialized hardware is most often used to accelerate compute-bound operations and the ongoing shift in the analytical workloads ran on databases towards machine learning[2] brings significantly more compute-intensive operations than the core SQL operators. What's more, there are proposals for using machine learning methods to replace parts of the decision making and optimization processes inside databases [10]. These emerging operators bring new opportunities in hardware acceleration both inside databases and for user workloads. It is important to note however that, now that hardware acceleration of real-world workloads is economically feasible, new challenges emerge in the area of deep integration of programmable hardware in databases.

In this paper we make the case that the two trends mentioned above, namely, datacenters becoming increasingly heterogeneous and workloads opening towards machine learning, combined with the state of the art in hardware acceleration for databases tackle most of the past hindrances of programmable hardware adoption and are a cause for optimism. We will focus on Field Programmable Gate Arrays (FPGAs) as a representative example and discuss how several significant challenges have been alleviated recently. In the second part of this paper we highlight open questions around the topics of resource management and query planning/compilation in the presence of programmable hardware accelerators.

2 Background

2.1 Programmable Hardware in the Datacenter

The wide range of programmable hardware devices proposed and already deployed in datacenter can be categorized depending on their location with regards to the data source and CPU into three categories (see Figure 1): on the side, in data-path and co-processor.

The most traditional way we think about accelerators is as being *on the side*, attached to the processor via an interconnect, for instance PCIe. Importantly, in this deployment scenario the CPU owns the data and explicitly sends it to the accelerator, resulting typically in significant additional latency per operation (due to communication latency and data transformation overhead). This encourages offloading operations at large granularity and without requiring back and forth communication between the CPU and the accelerator. GPUs are a common example of these kinds of accelerators, and were shown to be useful, for instance, to offload LIKE-based string queries [11]. There have also been proposals that deploy FPGAs this way for data filtering and decompression, e.g., in the work by Sukhwani et al. [12].

Another way of placing acceleration functionality in the architecture is *in data-path*. This can be thought of as a generalized version of near-data processing [13], and the goal of the accelerator is to filter or transform

[1]At the moment of writing it costs around \$1.65/h to rent an Amazon EC2 F1 instance.

[2]For instance, Microsoft SQL Server now includes machine learning plug-ins. `https://docs.microsoft.com/en-us/sql/advanced-analytics/what-is-sql-server-machine-learning`

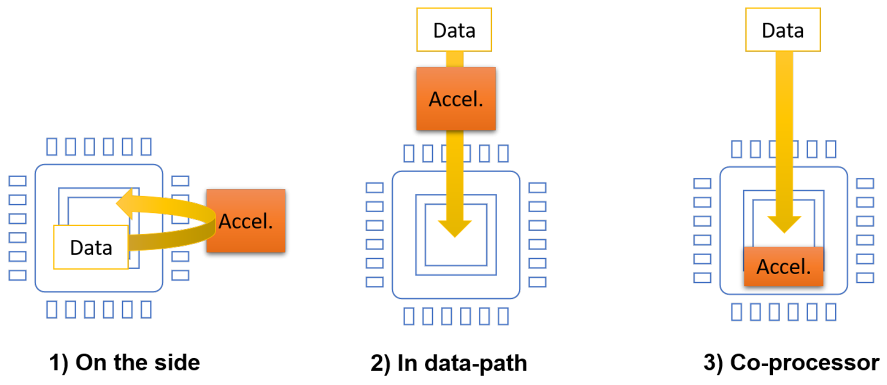


Figure 1: Programmable hardware accelerators can be deployed either as "on the side" accelerator (e.g., GPUs), as "in data-path" accelerator (e.g. smart NICs, smart SSD), or as co-processor (e.g. in Oracle DAX or Intel Xeon+FPGA).

data at the speed that it is received from the data source (designs that can't guarantee this could end up slowing the entire system down [14]). Much of the research effort in this space has been centered around in-SSD processing [15][27], but more recently, there has been efforts in using RDMA network interface cards (NICs) to accelerate distributed databases [17][18]. These NICs are limited to data manipulation acceleration, but there are efforts to make NICs and networking hardware in general, more programmable [19]. This will allow in the future to offload complex, application-specific, operations.

The third deployment option, namely, *co-processor*, is also becoming increasingly available in the form of CPUs that integrate domain-specific or general-purpose programmable co-processors: The Oracle DAX [20] is an example of the former because it implements database-specific operations (data decompression, scan acceleration, comparison-based filtering) on data in the last level cache. Thanks to its specialized nature, it occupies negligible chip space and does not increase the cost of the CPU. As opposed to the DAX, the Intel Xeon+FPGA [9] platform offers an FPGA beside the CPU cores for general-purpose acceleration. The FPGA has high bandwidth cache coherent access to the main memory of the CPU and can be reprogrammed in different ways. This creates acceleration opportunities without the usual overhead of the on the side accelerators.

2.2 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are chips that can be programmed to implement arbitrary circuits and historically have been used to validate designs that later would be turned into Application-Specific Integrated Circuits (ASICs). They have recently become a target for implementing data processing accelerators in datacenters thanks to their flexibility (their role can change over time, as opposed to an ASIC) and order of magnitude better energy efficiency than that of traditional CPUs [21]. FPGAs are composed of look-up tables (LUTs), on-chip memory (BRAM) and digital signal processing units (DSPs). All these components can be configured and interconnected flexibly, allowing the programmer to implement any hardware logic on top (Figure 2). It is not uncommon to have small ARM cores integrated inside the programmable fabric either (e.g., in Xilinx's Zynq product line).

FPGAs offer two types of parallelism: First, pipeline parallelism means that complex functionality can be executed in steps 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 that is often exploited on FPGAs is data-parallel execution. This is like SIMD (single instruction multiple data) processing in CPUs, but it can also implement a SPMD (single program multiple data) paradigm if the operations are coarser grained. What makes FPGAs

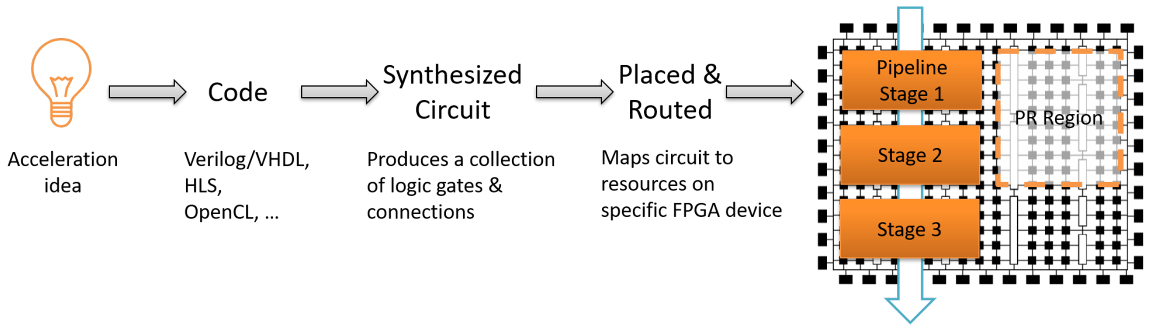

Acceleration idea
Code
Verilog/VHDL, HLS, OpenCL, …
Synthesized Circuit
Produces a collection of logic gates & connections
Placed & Routed
Maps circuit to resources on specific FPGA device
Pipeline Stage 1
Stage 2
Stage 3
PR Region

Figure 2: 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.

interesting for acceleration is that these two types of parallelism can be combined even inside a single application module to provide both complex processing and scalable throughput.

As Figure 2 shows, FPGAs are programmed by synthesizing a circuit from a hardware definition language, such as Verilog or VHLD, and creating a "bitstream" for a specific device type that defines the behavior of every logic resource on the chip. This is an expensive step as it requires the tool to lay out the circuit on the "chip surface" and define connections and routing of these connections between circuit elements. Since FPGAs have flexible clocking options and the programmer is free to define a target frequency (e.g., 300MHz), the tools have to set up routing such that signals are propagated within the clock periods (which can become impossible with too high frequencies).

It is also possible to perform partial reconfiguration (PR), meaning that only a portion of the FPGAs resources are reprogrammed (illustrated on the right-hand side of Figure 2). This means that, for instance, in a database use-case a hardware-accelerated operator can be replaced with another one without having to bring the device offline. PR, however, comes with limitations: the regions can only be defined at coarse granularity, their size can't be redefined at runtime and their reprogramming requires milliseconds.

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. This also applies to the "state" of an algorithm that is often stored as data in the on-chip BRAM memories. These can be accessed in a single clock cycle, but if the data doesn't fit in the available BRAM, high latency off-chip DRAM has to be used.

3 Sources of Pessimism

Many early projects of FPGA-based database acceleration propose deploying them as on the side accelerators for row stores [12][22][23] and they demonstrate that FPGAs are able to successfully accelerate selection, projection, group-by aggregation, joins and even sorting by an order of magnitude when compared to MySQL and Postgres, for instance. However, the **benefits are significantly reduced once one factors in the cost of communication over PCIe and the software overhead** of preparing the data for the FPGA to work on (sometimes pre-parsing, often copying pages).

In traditional, on the side deployments, the high latency communication (microseconds over PCIe) often forces designs to move entire operators onto the FPGA, even if only parts of the operator were a good match for the hardware. This leads to complications, because even though FPGAs excel at parallel and pipelined execution, they behave poorly when an algorithm requires iterative "loops" or has widely branching "if-then-else" logic. In the case of the former, CPUs 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**.

In parallel with accelerator-based efforts, there have been numerous advances in the space of analytical databases. Today, column-oriented databases, such as MonetDB [24], are widely deployed and typically outperform row-oriented ones by at least an order of magnitude and can take advantage of many-core CPUs efficiently. As a result, the **speedups that FPGAs offer when targeting core SQL operators have shrunk**[3] and often are not enough to motivate the additional effort of integrating specialized hardware in the server architecture.

For the above reasons, FPGA-based acceleration ideas are often received with pessimism. However, changes in the hardware available in datacenters and the cloud, as well as the changes in database architecture and user workloads, create novel opportunities for FPGA-based acceleration. In the next section we discuss these in more detail and provide examples of how they can be exploited.

4 Reasons for Optimism

4.1 Changing Architectures

With the increasing adoption of distributed architectures for analytical databases, as well as the disaggregation efforts in the datacenter [25], there are numerous opportunities for moving computation closer to the data source to reduce the data movement bottlenecks. **These bottlenecks arise from the fact that the access bandwidths are higher closer to the data source than over the network/interconnect and they can be eliminated by pushing filtering or similar data reduction operations closer to source. Thus, the main goal of having the accelerator in the data-path is to maintain the data access bandwidth high while reducing the amount of data sent to the processor.**

The data source is often (network-attached) flash storage and recent projects, for instance, YourSQL [15], BlueDBM [16] and Ibex [27], show that it is possible to execute SQL operations as the data is moving from storage to processing at high bandwidth. Another use-case that can benefit from data reduction in a similar way is ETL. Recent work [26] has demonstrated that specialized hardware can be used to offer a wide range of ETL operations at high data rate, including: (de)compression, parsing from formats such as CSV or JSON, pattern matching and histogram creation.

In Ibex we deployed an FPGA between an SSD and the CPU, offering several operators that can be plugged into MySQL's query plans. As Figure 3 shows, these include scans, projection, filtering and group-by aggregation, and were chosen in a way that ensures that the processing in hardware will reduce the final data size for most queries. For this reason, Ibex does not accelerate joins, since these would potentially result in larger outputs than the input and slow down the system this way. The rest of the operations are all performed at the rate of the data arriving from storage.

As opposed to on the side accelerators, in this space there are two possible options for who "owns" the data. In the case of smart SSDs, data is typically managed by the host database [27][15]. In contrast, in the case of distributed storage accessed over the network, it is possible to explore designs where the data is both processed and managed by the specialized hardware device as, for instance, in Caribou [28][29], our distributed key-value store that is built using only FPGAs. In Caribou, the FPGAs, in addition to network line-rate data processing, implements the hash table data structure and memory allocator necessary for managing large amounts of data, as well as, data replication techniques to ensure that no records are lost or corrupted in case of device failures or

[3]Using specialized hardware can still compete with multi-cores if we factor in energy efficiency (Operations/s/Watt) but in many cases the metric that is of interest is database throughput and response time.

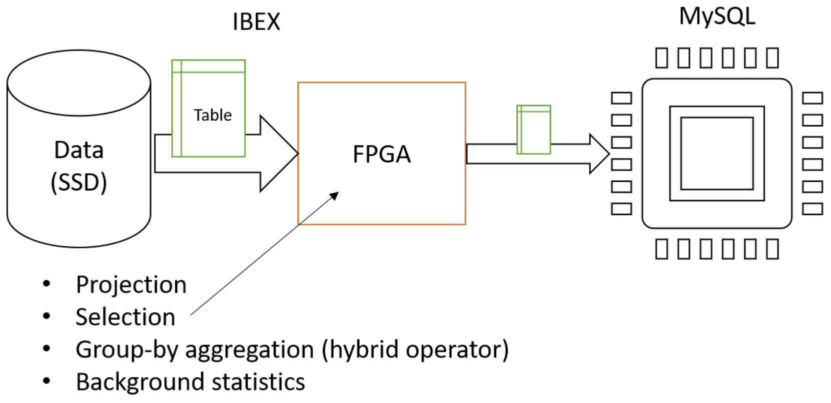


Figure 3: In Ibex we showcase several operations that can be performed on the data as it is read from storage with the goal of reducing the number of tuples that arrive at the CPU.

network partitions. This results in a high throughput energy efficient distributed storage layer that, even though is built using FPGAs, can be used as a drop-in replacement for software-based solutions [29].

In many ways, in data-path accelerators provide similar acceleration options as the on the side ones because data is still moved over a network (similarly to an interconnect in the case of the latter) that requires processing it in batches large enough to warrant the latency overhead. However, if FPGAs are deployed as co-processors, this overhead is drastically reduced and new opportunities open up, since the latency to the FPGA is in the same order of magnitude as a cross-socket memory access. The Centaur platform [30], for instance, exposes the FPGA of an Intel Xeon+FPGA platform using an efficient “hardware thread” API. **As a result, in this co-processor scenario, the database can offload functionality as if spawning a parallel thread and the FPGA can be used for processing even just a handful of tuples** – as we point out in the next subsection, there are emerging use-cases where this low latency acceleration is a game-changer.

4.2 Emerging Compute-Intensive Workloads

The examples in the previous subsection showed how to reduce the data access bottleneck with an in data-path accelerator targeting common SQL operators. It is unclear, however, if this strategy can be applied for co-processors as well. Modern database engines, that make use of the multi-core CPUs and their wide SIMD units, are rarely compute bound once the data is loaded into main memory. **Unless reading data from storage, offloading core SQL operators is unlikely to bring orders of magnitudes improvements in performance. There is, however, cause for optimism if we look beyond such operators and in the direction of machine learning, both training and inference.**

A significant portion of machine learning pipelines operate on relational data and the case has been made that there is a benefit in integrating these directly in the database [31]. Furthermore, there is also interest in including such components in the internal modules of the databases [10], to perform optimizations depending on the workload characteristics and a model. Since this could require on-line retraining that, without hardware acceleration, could hurt user throughput significantly, new opportunities open up for FPGAs. Acceleration of training as part of user workloads is being explored, for instance in Dana [31]. The iterative and computation-heavy nature of training operators makes them less sensitive to the latency issues introduced by using on the side accelerators and therefore could revive the interest in these acceleration platforms. Amazon, for instance, is already offering FPGAs running Xilinx’s OpenCL-based compute framework as PCIe-attached accelerators.

In the “ML-backed” database scenario it will also be paramount to be able to take decisions with low latency using learned models – this further motivates the use of FPGAs. Even though GPUs are a de-facto standard for machine learning acceleration, when it comes to low latency inference, FPGAs can offer benefits since they do not require batching in their processing modules: recent work by Owaida et al. [32] and Umuroglu et al. [33] demonstrates, for instance, how FPGAs can be used very efficiently to accelerate inference using decision trees, respectively, neural networks.

4.3 Hybrid Approaches to Acceleration

Since all functionality, regardless whether used or not, occupies chip space on the FPGA, **corner cases often can't be efficiently handled in hardware. For this reason, it is important to design accelerators such that they behave predictably even if the particular instance of the problem can't be fully handled.** As we illustrate below with two examples from our work, state of the art solutions overcame such cases by splitting functionality between FPGA and software, such that the part on the FPGA remains beneficial to execution time regardless of the input data contents or distribution.

In Ibex [27] we used a hybrid methodology to implement a group-by operator that supports `min`, `max`, `count` and `sum` (in order to compute `avg`, we used query rewriting to compute the count and sums). This operator is built around a small hash table that collects the aggregate values. In line with FPGA best-practices, the hash table is of fixed size and is implemented in BRAM. The reason for this is that this way it is possible to guarantee fixed bandwidth operation, regardless of the data contents, because the FPGA doesn't have to pause processing to resize the table. Unfortunately, this approach has a drawback: if a query had just one more group than the size of the hash table, the FPGA couldn't be used – and this information is often not available up front. We overcome this situation by post-processing the results of the group-by operator on the FPGA in software. The hardware returns results from the group by aggregation unit in a format that allows the database to perform an additional aggregation step on top without having to apply projections on the tuples or parse them in the first place (see Figure 4). If, during the hash table operations collisions are encountered that can't be solved, a partial aggregate is evicted from the table and sent to the software post-processor. Once all the data has been processed on the FPGA, the contents of the hash table are sent to the software post-processor to compute the final groups. This results in a behavior where, if all the groups could be computed on the FPGA, the final software step has to perform virtually no work (assuming that the number of resulting groups is significantly smaller than the cardinality of the table), and otherwise the software executes the group by aggregation as if there wouldn't be any FPGA present (though still benefits from projections and selections).

The regular expression-based LIKE operator that we implemented in MonetDB [34] running on top of the Intel Xeon+FPGA platform is another example of the hybrid operator methodology. If the expression could not be encoded in its entirety on the FPGA, because, for instance, it contains too many characters (such as the

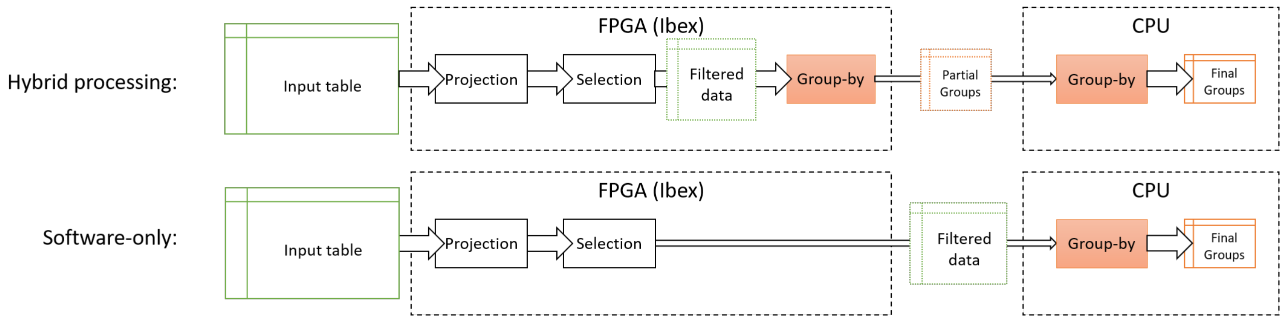
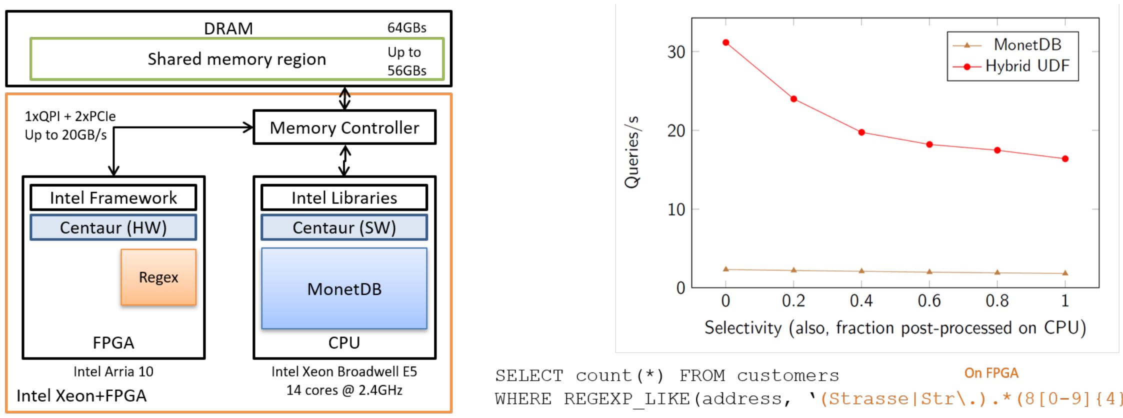


Figure 4: By implementing operators in a way that allows hybrid computation, the FPGA accelerator can reduce data sizes over the bottleneck connection to the CPU in most cases. In this example of Ibex's group-by operator, if we would choose an “all or nothing approach”, moving the data to be aggregated to the CPU could become the bottleneck.



```
SELECT count(*) FROM customers
WHERE REGEXP_LIKE(address, '(Strasse|Str\.).*(8[0-9]{4}).*delivery')
```

Figure 5: Even if only part of the regular expression fits on the FPGA it is worth to offload it because the post-processing becomes cheaper, resulting in an overall faster execution.

bottom example in Figure 5), we cut it at the last possible wildcard and process the first part of the expression on the FPGA and the second part in software. For each string, the FPGA operator returns an index that signifies the end of the location where the regular expression matched the string. The software can pick up processing from this point in case of hybrid processing and match the rest of the expression. In any case entire expression fits on the FPGA, however, the software has no additional work to do. In Figure 5 we illustrate how, when compared to a single-threaded execution in MonetDB, the hybrid solution is always faster than the software-only one (for more details see [34]).

One aspect that makes the integration of programmable hardware in databases challenging is the change in the predictability of query runtimes. Therefore, in our work we aim to design circuits whose throughput is not affected by the problem instance they work on. This way the query optimizer can predict the rate at which data will be processed/filtered on the FPGA and with this information it can reliably decide when to offload. One example of such a design is the regular expression module we presented above. Since the overhead of compiling regular expressions to circuits and then performing partial reconfiguration (PR) could take longer than executing an entire query, we took a different approach: we created a "universal" automaton that could implement any expression within some limits on the number of distinct characters to detect and the number of states. Small on-chip memories are used to describe the state machine and the characters of the regular expression, and their contents can be loaded at runtime in nanoseconds. We laid out this state machine as a pipeline, that processes one character per clock cycle, regardless of the contents of the on-chip memories. The conversion from a regular expression written by a user to the configuration parameters is performed in software but is orders of magnitude cheaper than circuit synthesis.

5 The Road that Lies Ahead

5.1 Managing Programmable Hardware

How to best integrate hardware that, while reprogrammable, will never be as flexible as software? Should the operating system/hypervisor control it, or can future database learn to manage it?

Even though there are efforts in the FPGA community to speed up the process of partial reconfiguration, it is unlikely that the overhead of this operation will ever be as small as that of a software context switch. As a result, databases must find ways to adapt to the idea of running on programmable hardware that, even though, can change over time, doesn't have the flexibility of software. The main question that needs to be answered in this space is who will "own" the acceleration functionality, because this also defines whether the database needs only to be able to compile its queries to take advantage of the accelerators, or whether it could also synthesize fully custom accelerators depending on the workload.

If it is the OS/hypervisor that controls the accelerator, then the database still has to be able to adapt to different underlying hardware acceleration functionality, that will be likely both designed and managed by the infrastructure/cloud provider. In this scenario, the database has to create query plans that take advantage of the specific machine's acceleration opportunities. For this, it is likely that we can reuse techniques that are already present in databases for compiling code for different target CPU features such as SIMD units [39].

Alternatively, if the database takes full ownership of the accelerator, it will have more responsibility but also greater opportunities. Instead of relying on the cloud provider to design general-purpose acceleration units that might or might not match the database's needs, the database developer can design and synthesize the right ones and integrate them tighter with the database. What's more, the database could even generate and synthesize workload-specific accelerators at runtime.

In DoppioDB [35][30] we explored the case where the database manages the accelerator. The role of the operating system is to set up a basic infrastructure on the FPGA, configuring it 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). Once the database has started, the FPGA gets access to the process's virtual memory space and the database can explicitly manage what tasks the different slots perform, choosing, in our prototype, from a small library of available operators. In DoppioDB, instead of focusing only on the usual SQL operators like selection or joins, we began exploring how one could extend what the database is capable of, targeting machine learning type of operators, such as training a model using stochastic gradient descent or running inference with decision trees. This functionality was exposed using a UDF mechanism, but in the future could be integrated much tighter with the database. The research question that emerges is how to populate the hardware operator library and what granularity these operators should have. Recent work by Kara et al. [41] shows that it is possible to offload sub-operators successfully to the FPGA. However, the identification of generic enough sub-operators that can be deployed on an accelerator and parameterized/composed at runtime remains an open challenge.

5.2 Compilation/Synthesis for Programmable Hardware

Are there reusable building blocks that would benefit query compilation for programmable hardware? Should databases have their own DSLs from which to generate hardware accelerators?

The second big question is how to express acceleration functionality for database use-cases in an efficient way. 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 [39][40].

The side effect of bringing more ideas from compilers into databases is that it will likely be also easier integrate a DSLs for hardware accelerators [36][37][38] into the database. However, many of these solutions are targeting compute kernels written in languages such as OpenCL [36], that are a better fit for HPC and machine learning type functionality than database operations. 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 [37]. Spatial approaches the problem of writing parallel code for accelerators in a way that accounts for the fact that FPGA circuits 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 Conclusion

Even though it has been approached pessimistically for a long time, we argue that the use of specialized hardware in analytical databases has a positive outlook. To support this argument, we discussed the past and future challenges of including a specific kind of hardware accelerator, namely FPGAs, in databases.

To address fears that deploying FPGAs always brings high overheads that reduce their “raw” speedup, we highlighted how, in today’s distributed database landscape, they can be used to reduce bottlenecks of data movement by positioning them in data-path. Since they can process data at the rate at which it is retrieved from the data source, they never slow down data access, even if there is no opportunity for acceleration. We also discussed the opportunities that novel, machine learning, workloads bring. Their operators are typically compute bound on CPUs and using FPGAs we can achieve significant speedups even when compared to an entire socket with multiple cores. Finally, to demonstrate that it is possible to design FPGA-based operators that behave gracefully even if the entire functionality of the operator doesn’t fit on the device, we discussed two examples from our previous work that implement hybrid computation across FPGA and CPU (a group-by operator and a regular expression matcher).

We also identify two areas in which significant progress is required before the inclusion of heterogeneous hardware in databases becomes the norm rather than the exception. One is finding ways to actively manage the programmable hardware underneath the database, shaping it to workloads using partial reconfiguration and parameterizable circuits. The second question is about finding the right programming primitives for hardware accelerators in the context of databases operators, to avoid designing from scratch each new accelerator idea, and to allow the database to offload parts of a query more flexibly at runtime. It is unlikely that we can provide answers for both questions only from inside the database community and will have to instead collaborate with researchers working in the areas of operating systems, programming languages and compilers.

Acknowledgments

Our cited work and many of the lessons learned are a result of the author’s collaboration with current and past members of the Systems Group at ETH Zürich, in particular, Gustavo Alonso, David Sidler, Louis Woods and Jana Giceva.

References

[1] J. Banerjee, D. Hsiao and K. Kannan. DBC: A Database Computer for Very Large Databases. *IEEE Transactions on Computers*, 6, pp. 414-429, IEEE, 1979.

[2] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.I. Hsiao, R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and data engineering*, 2(1), pp. 44-62, 1990.

[3] P. Francisco. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics. *IBM Red Books*, 2011.

[4] L. Wu, A. Lottarini, T.K. Paine, M. Kim, K.A. Ross Q100: The Architecture and Design of a Database Processing Unit. *ASPLOS'14*, pp. 255-268, 2014.

[5] S.R. Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, E. Sedlar et al. A many-core architecture for in-memory data processing. *MICRO'17*, pp. 245-258, ACM, 2017.

[6] H. Esmaeilzadeh, E. Blem, R.S. Amant, K. Sankaralingam and D. Burger. Dark silicon and the end of multicore scaling. *ISCA'11*, pp. 365-376, IEEE, 2011.

[7] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. *NSDI'18*, USENIX, 2018.

[8] K. Sato, C. Young, D. Patterson. An in-depth look at Google's first Tensor Processing Unit (TPU). *Google Cloud Big Data and Machine Learning Blog, 12*, 2017.

[9] P.K. Gupta. Accelerating datacenter workloads. *FPL'16*, 2016.

[10] T. Kraska, M. Alizadeh, A. Beutel, E. Chi, J. Ding, A. Kristo, V. Nathan, et al. SageDB: A learned database system. *CIDR'19*, 2019.

[11] E. Sitaridi, K. Ross. GPU-accelerated string matching for database applications. *Proceedings of the VLDB Endowment*, pp. 719-740, 2016.

[12] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Brezzo, S. Asaad, D.E. Dillenberger. *Database analytics: A reconfigurable-computing approach*, IEEE Micro, 34(1), pp. 19-29, 2014.

[13] M. Oskin, F.T. Chong, T. Sherwood. Active pages: A computation model for intelligent memory. *IEEE Computer Society*, Vol. 26, No. 3, pp. 192-203, 1988.

[14] G. Koo, K.K. Matam, H.V. Narra, J. Li, H.W. Tseng, S. Swanson, M. Annavaram. Summarizer: trading communication with computing near storage. *MICRO'17*, pp. 219-231, ACM, 2017.

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

[16] S.W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu. BlueDBM: Distributed Flash Storage for Big Data Analytics. *ACM TOCS* 34(3), 7, 2016.

[17] C. Barthels, S. Loesing, G. Alonso, D. Kossmann. Rack-scale in-memory join processing using RDMA. *SIGMOD'15*, pp. 1463-1475, ACM, 2015.

[18] A. Dragojevic; D. Narayanan; M. Castro. RDMA Reads: To Use or Not to Use?. *IEEE Data Eng. Bull.*, vol. 40, no 1, pp. 3-14, 2017.

[19] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3), pp. 87-95, 2014.

[20] K. Aingaran, S. Jairath, D. Lutz. Software in Silicon in the Oracle SPARC M7 processor. *Hot Chips Symposium (HCS'16)*, pp. 1-31, IEEE, 2016.

[21] J. Teubner and L. Woods. Data processing on FPGAs. *Synthesis Lectures on Data Management*, 5(2), pp. 1-118, 2011.

[22] J. Casper, K. Olukotun. Hardware acceleration of database operations. *FPGA'14*, pp. 151-160, ACM, 2014.

[23] C. Dennl, D. Ziener, J. Teich Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration. *FCCM'13*, pp. 25-28, IEEE, 2013.

[24] P.A. Boncz, M. Zukowski, N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. *CIDR*, Vol. 5, pp. 225-237, 2005.

[25] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, S. Kumar. Flash storage disaggregation. *EUROSYS'16*, 2016.

[26] Y. Fang, C. Zou, A.J. Elmore, A.A. Chien. UDP: a programmable accelerator for extract-transform-load workloads and more. *MICRO'17*, pp. 55-68, ACM, 2017.

[27] L. Woods, Z. Istvan, G. Alonso. Ibex: an intelligent storage engine with support for advanced SQL offloading. *Proceedings of the VLDB Endowment*, 7(11), pp. 963-974, 2014.

[28] Z. Istvan, D. Sidler, G. Alonso. Caribou: intelligent distributed storage. *Proceedings of the VLDB Endowment*, 10(11), pp. 1202-1213, 2017.

[29] Z. Istvan. Building Distributed Storage with Specialized Hardware Doctoral dissertation, ETH Zurich, 2018.

[30] M. Owaida, D. Sidler, K. Kara, G. Alonso Centaur: A framework for hybrid CPU-FPGA databases. *FCCM'17*, pp. 211-218, IEEE, 2017.

[31] D. Mahajan, J.K. Kim, J. Sacks, A. Ardalan, A. Kumar, H. Esmaeilzadeh. In-RDBMS Hardware Acceleration of Advanced Analytics. *Proceedings of the VLDB Endowment*, 11(11), 2018.

[32] M. Owaida, H. Zhang, C. Zhang, G. Alonso. Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms. *FPL'17*, IEEE, 2017.

[33] Y. Umuroglu, N.J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, K. Vissers. Finn: A framework for fast, scalable binarized neural network inference. *FPGA'17*, pp. 65-74, ACM, 2017.

[34] D. Sidler, Z. Istvan, M. Owaida, G. Alonso. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. *SIGMOD'17*, pp. 403-415, ACM, 2017.

[35] D. Sidler, Z. Istvan, M. Owaida, K. Kara, G. Alonso. doppioDB: A hardware accelerated database. *SIGMOD'17*, pp. 1659-1662, ACM, 2017.

[36] M. Wong, A. Richards, M. Rovatsou, R. Reyes. Khronos's OpenCL SYCL to support heterogeneous devices for C++, 2016.

[37] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, K. Olukotun. Spatial: a language and compiler for application accelerators. *PLDI'18*, pp. 296-311, ACM, 2018.

[38] O Mencer. Maximum performance computing for Exascale applications. *ICSAMOS'12*, 2012.

[39] H. Pirk, J. Giceva, P. Pietzuch. Thriving in the No Man's Land between Compilers and Databases. *CIDR*, 2019.

[40] H. Pirk, O. Moll, M. Zaharia, S. Madden Voodoo - A vector algebra for portable database performance on modern hardware. *Proceedings of the VLDB Endowment*, 9(14), pp. 1707-1718, 2016.

[41] K. Kara, J. Giceva, G. Alonso. Fpga-based data partitioning *SIGMOD'17*, pp. 433-445, ACM, 2017.