Delft University of Technology

FPGA Acceleration for Big Data Analytics
Challenges and Opportunities

Hoozemans, Joost; Peltenburg, Johan; Nonnenmacher, Fabian ; Hadnagy, Ákos; Al-Ars, Zaid; Hofstee, H. Peter

DOI
10.1109/MCAS.2021.3071608

Publication date
2021

Document Version
Accepted author manuscript

Published in
IEEE Circuits and Systems Magazine

Citation (APA)
Hoozemans, J., Peltenburg, J., Nonnenmacher, F., Hadnagy, Á., Al-Ars, Z., & Hofstee, H. P. (2021). FPGA Acceleration for Big Data Analytics: Challenges and Opportunities. *IEEE Circuits and Systems Magazine*, *21*(2), 30-47. Article 9439431. https://doi.org/10.1109/MCAS.2021.3071608

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

FPGA Acceleration for Big Data Analytics: Challenges and Opportunities

Joost Hoozemans Johan Peltenburg Fabian Nonnenmacher Ákos Hadnagy
Zaid Al-Ars H. Peter Hofstee

***Abstract*—The big data revolution has ushered an era with ever increasing volumes and complexity of data requiring ever faster computational analysis. During this very same era, CPU performance growth has been stagnating, pushing the industry to either scale their computation horizontally using multiple nodes in datacenters, or to scale vertically using heterogeneous components to reduce compute time. However, networking and storage continue to provide both higher throughput and lower latency, which allows for leveraging heterogeneous components, deployed in data centers around the world. Still, the integration of big data analytics frameworks with heterogeneous hardware components such as GPGPUs and FPGAs is challenging, because there is an increasing gap in the level of abstraction between analytics solutions developed with big data analytics frameworks, and accelerated kernels developed with heterogeneous components. In this article, we focus on FPGA accelerators that have seen wide-scale deployment in large cloud infrastructures. FPGAs allow the implementation of highly optimized hardware architectures, tailored exactly to an application, and unburdened by the overhead associated with traditional general-purpose computer architectures. FPGAs implementing dataflow-oriented architectures with high levels of (pipeline) parallelism can provide high application throughput, often providing high energy efficiency. Latency-sensitive applications can leverage FPGA accelerators by directly connecting to the physical layer of a network, and perform data transformations without going through the software stacks of the host system. While these advantages of FPGA accelerators hold promise, difficulties associated with programming and integration limit their use. This article explores the existing practices in big data analytics frameworks, discusses the aforementioned gap in development abstractions, and provides some perspectives on how to address these challenges in the future.**

***Index Terms*—Big data, FPGA, accelerator, heterogenous computing, Apache Arrow, Apache Spark**

I. Introduction

A lot has changed in the world of analytics since the first database management systems were popularized. Highly structured data was carefully administered, making it easy to distill information. Now, the field is confronted with a quintuple challenge: more data to process, more abstraction desired, new computationally intensive paradigms (like AI), and more latency sensitive (interactive) uses. At the same time, transistor performance improvements are slowing down and power efficiency becomes a greater and greater challenge.

Meanwhile, recent developments have also presented us with several new opportunities. Cloud infrastructure has opened up the use of large compute clusters to a far wider range of users by allowing more efficient sharing of hardware. Highly efficient compute units have been designed to target certain workloads in a highly specialized manner. The memory hierarchy has been extended with solid state drives and storage class memory, improving both throughput and latency for access to large datasets. Similarly, networking technology has been advancing rapidly, still keeping pace with Moore's law.

While the performance increase of general-purpose CPUs is not able to keep up with the advances in their periphery, reconfigurable logic found in FPGAs provides several unique strengths that may be able to cope better. For many applications, FPGAs have the ability to transform and filter data at the very high data rates that new storage systems can provide. When attached directly to the network, reconfigurable logic can provide similar functionality at line-rate with very low latency, because data no longer needs to traverse various software layers of the networking and application stack. It allows users to specialize FPGA accelerators found in cloud instances to their applications, without the need to replace hardware. Tailoring the logic mapped on top of the FPGA fabric allows the construction of custom dataflow computers, where there is a minimal need to move data back and forth through the memory hierarchy. Even in cases where such FPGA designs provide only marginal gains in terms of throughput, the computations tend to be more power-efficient because of the reduced data movement and because the FPGA operates at relatively low clock frequencies.

However, FPGA acceleration still has several challenges. In spite of efforts to improve the ease of programming, FPGAs are currently not yet suitable to be used directly by an audience untrained in circuit design. Very few people with proper hardware design training are also active in the field of big data analytics, leading to a scarcity of designers. Furthermore, the use of FPGAs in cloud instances is currently still very explicit, requiring FPGA-aware code paths. Also, data coming from high-level languages running on interpreters and virtual machines needs to be transformed and copied sometimes several times during (de)serialization before it can be used in the accelerator. This often negates any advantage that can be gained by designing the accelerator in the first place. Last, there is still no systematic way to address the long synthesis time for datacenter-class FPGAs (where 12 hours is no exception) and the relatively long reprogramming delay (in the order of 100 ms).

In this article, we investigate these challenges and opportunities and summarize recent progress, and lay out a roadmap for further improvement. Ultimately, our goal is seamless integration of FPGAs in cloud infrastructure for big data analytics. Section II presents a more in-depth background

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works

of the related topics. Then, Section III discusses recent developments (opportunities) and missing links (challenges) for near-future big data analytics systems. Section IV presents a prototype system that features many of the components discussed. Section V provides an analysis and evaluation of our prototype implementation. Section VI provides a discussion of related work. Finally, Section VII concludes the article.

II. Background

A. Heterogeneous computing

Increases in single-thread performance have dropped to less than 10% per year during the last decade [1] due in part to a slowdown in Moore's law. The additional transistors are not directly translating into raw performance anymore, but instead into higher core counts and, more recently, more specialized compute units [2]. Modern processors often contain custom cores for compression, cryptography, media codecs, etc. These are better suited to meet the demands of higher performance at better energy efficiency. Graphics Processing Units (GPUs) have long been made available as a mainstream accelerator, targeting very specific (graphics processing) tasks at first but more recently being applied in a wider variety of workloads. More accelerators have been introduced, e.g. to target machine learning (Google's TPU [3]), or more irregular massively parallel applications (Graphcore's IPU).

In heterogeneous computing, parts of the workload are appropriately divided among multiple different but more specialized and efficient components than just the general-purpose CPU. Not every application will have a compute fabric that provides a perfect match for it. FPGAs, however, create the opportunity to design a custom hardware implementation for a specific application with a fast time-to-market.

B. FPGA computing

While FPGAs have been commonplace in certain application domains, and have been widely used for testing and prototyping ASIC designs, recent offerings from vendors (e.g. [4], [5]) are specifically targeting datacenters and cloud infrastructures.

There is roughly an order of magnitude overhead in both clock frequency and area compared to full-custom IC designs [6], since more transistors are required to implement the same function, due to the reconfigurable nature of the underlying FPGA fabric. In spite of this, FPGAs are becoming an important element in the datacenter strategies of several large cloud providers because often there is still potential for higher performance and/or higher energy-efficiency [7].

This potential comes from several factors:

- Custom datatypes (no need to use a 64-bit ALU for incrementing indexes that would only need a few bits)
- Custom memory hierarchies and connectivity (placing local memories in the most suitable location and providing a more optimal organization)
- No instructions (this means saving caches and memory bandwidth, and it also means no logic required for instruction decoding etc.)
- Very deep pipelining (dataflow designs can have pipeline depths of thousands of stages, all of which are performing useful work at all times)
- Abundant parallelism (pipelines can be duplicated as long as there is FPGA logic available)
- Low clock frequencies (resulting in significant reduction in power consumption)

These factors contrast sharply with a general-purpose processor, which spends the majority of energy and logic moving data around between cores, caches and registers, decoding instructions and resolving dependencies. In comparison, a much larger fraction of FPGAs logic can be utilized for actual operations in a dataflow fashion, minimizing data movements between subsequent compute steps (performed by adjacent compute units on the chip).

However, FPGAs also have a number of drawbacks. Compared to CPUs and GPUs, the development cycles are much longer and require logic design expertise. Furthermore, synthesizing a design for an FPGA can take many hours. The benefits are further reduced when implementing control-heavy and highly complex algorithms.

1) Programming practices: The adoption of this new form of compute fabric has been slow, mainly because of the difficulty associated with programming. The traditional way of programming an FPGA is by using an RTL language, such as VHDL or Verilog, developed to design ASICs. This is a tedious and time-consuming process that requires a very specialized set of skills and knowledge. High-Level Synthesis (HLS) aims to decrease both design time and the level of knowledge required (targeting more traditional software developers by using the C/C++ programming language). However, studies show that the decrease in time to achieve a first functional design is often paired with an increase in time required to tune for high performance. Additionally, unlike conventional compilers for CPUs that now typically outperform hand-written assembly code, HLS compilers still tend to achieve a lower quality result (in terms of performance and FPGA resource utilization) compared to a hand-written RTL version [8]. This could partially be attributed to the mismatch between the programming paradigm and the FPGA fabric. The imperative programming paradigm targets a single sequential machine with a large uniformly accessible memory and aims to provide it with an explicit instruction every cycle (*temporal* programming). The FPGA fabric, on the other hand, is by nature a *spatial* resource. Instead of instructions, programming an FPGA requires specifying the placement of the operations and connections between them. The 'program' is executed implicitly by simply flowing data through the circuit (hence the term 'dataflow' computing).

There exist efforts at various levels of abstraction aiming to provide a more suitable programming method for FPGAs, from high-level parallel constructs [9] to dataflow-oriented [10] and from adding a timing-agnostic layer to an RTL language [11] to using a functional language [12] for circuit generation.

2) Integration: There are two facets to discuss in terms of integration: 1) how to integrate newly created circuits into existing designs and provide re-usability between these

designs, and 2) how the custom-designed FPGA circuits can be employed by software. Although it might seem to be just a matter of practicality, the amount of time a developer needs to spend on integration is alarmingly large. The time a designer needs for writing the actual compute kernel that will perform the operations is, even when considering an RTL design method, not that long. However, the amount of time needed to implement and debug the communication between all the components (both hardware and software) can be considerable.

Luckily, both vendors and the open source community have spent a great deal of effort to provide infrastructure aiming to alleviate this. For example, Gaisler's GRLIB provides open source IP cores for various components needed in a SoC environment, with a bus infrastructure that is easy to interface with. More recently, ARM's Advanced Xtensible Interface (AXI) has become the de-facto bus standard for creating or connecting to existing IP cores, and is supported by FPGA vendor tool-chains.

Although AXI has been a big help in standardizing hardware interfaces, it is still a primarily byte-oriented protocol developed for hardware designs, which means that 1) it is not very accessible for software developers and 2) custom glue logic is almost always necessary to integrate existing IP blocks into new designs or the other way around. As it turns out, the effort of creating glue logic, including for example arbitration and buffering (finding optimal buffer sizes in dataflow graphs is a research topic in itself [13]), is not to be underestimated.

In terms of software integration, the datacenter accelerator cards provide additional infrastructure in the form of a 'shell', which communicates with a standard driver so that designers only need to provide a set of interfaces on their hardware components and can subsequently communicate with them through a software API.

C. Performance scaling

Figure 1 shows how performance of I/O technologies is scaling versus that of main memory (DRAM). It provides two insights that are necessary to illustrate the upcoming challenge in performance scaling: Bandwidths provided by networking and storage technologies are both scaling very rapidly, especially since the introduction of solid-state storage devices such as SSDs. DRAM, however, is not able to deliver. Where networking and storage performance is scaling at rates similar to Moore's law, the throughput delivered by DRAM has been doubling roughly every 26 months, and this rate is decreasing.

Adding more processor cores will not solve this problem. One could already argue that CPU core counts are unable to scale with Moore's law because of the additional logic needed for caches and inter-core communication. The more pressing matter is that all those cores will not be able to access data in main memory fast enough. The result is that "The poor processor is now getting sandwiched between these two exponential performance growth curves of flash and network bandwidth" [14].

In the upcoming years, this discrepancy can be mitigated by scaling out; distributing the growing data volumes over increasing numbers of servers. However, since there is a continuous difference in scaling, this solution will not be viable for long. It seems inevitable that high-performance datapaths with CPUs in them will have to be designed around better alternatives.

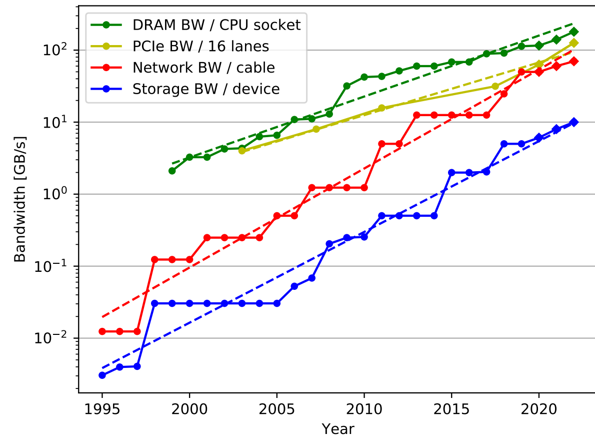


Fig. 1: Performance scaling of DRAM plotted versus storage and networking in terms of bandwidth (BW) [15].

D. Big data analytics

As *vertical* scaling (adding compute resources to a single node) is not able to satisfy the growing demands for performance, *horizontal* scaling is becoming the new approach for tackling very large datasets. Various frameworks that hide the complexity of distributing storage and computations over large numbers of nodes (for example, Spark and Hadoop) quickly grew in popularity. One of the more challenging aspects that these frameworks take care of is resiliency that is needed for this distribution - in some cases this even made it possible to use commodity hardware as opposed to costly server-grade systems. The other catalyst fueling the adoption was the ease of programming. The popular frameworks provide very high-level programming interfaces to their users, making large compute clusters accessible to data scientists without needing lots of programming expertise in programming distributed systems.

1) Programming practices: In contrast to the days of C programming, when very expensive computers were shared by many programmers who were writing highly optimized code, computing hardware has become relatively cheap and programmers relatively expensive. Big data frameworks have made it easy to increase performance by simply adding more nodes. In combination with the increasing importance of time-to-market, this has resulted in the emphasis being placed on productivity and ease of programming, while relying on adding hardware for achieving performance (instead of optimizing code).

Programming productivity is achieved by using modern languages employing increasingly high levels of abstraction. Figure 2 shows an example of a Spark program using its

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Fig. 2: Code for Word Count, a well-known example application counting words with Apache Spark and Scala, from https://spark.apache.org/.

Resilient Distributed Dataset API. The input file can be TeraBytes in size, distributed across several nodes, but this is all hidden from the user. These abstractions come at a cost; they move the computation further away from the bare metal. The consequence is twofold. First the often virtualized runtime environments of modern programming languages (for example, Spark uses Scala, which runs in a Java Virtual Machine) generally achieve less performance than languages compiled to native machine code (although concepts such as Just-In-Time compilation and the use of highly optimized libraries have at least partially addressed this). Second, the data itself is encoded in a way that is specific to the particular compute environment/programming language. While the execution performance limitations of interpreted languages can be largely mitigated, memory management presents a greater challenge, especially in the context of acceleration [16].

Figure 3 shows that about 80% of the frameworks rely on languages that use some form of automatic memory management. This means that many types of data that would have a very straightforward organization in C/C++ would be wrapped in an object (that would, for example, include reference counts for garbage collection). To send an array of these objects to a different analytics framework or to an accelerator, each one would need to be unwrapped and placed into an intermediate array with an encoding that the other party understands (serialization), followed by the reversed process on the other side (deserialization). This could be needed when using different frameworks or languages for certain parts of the computation.

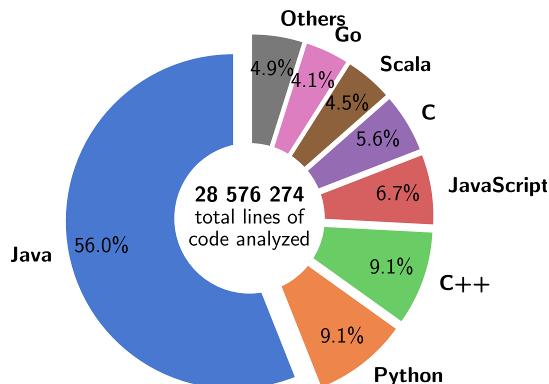
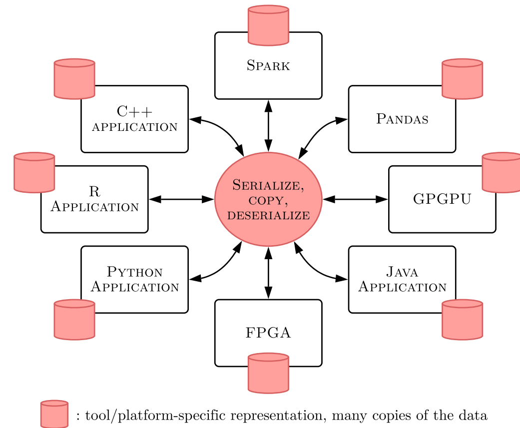
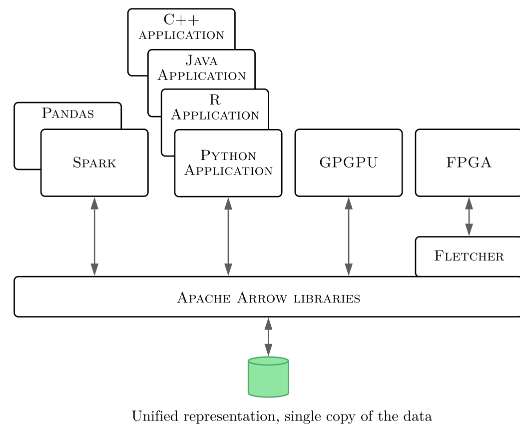


Fig. 3: Programming languages used by 52 various open source big data analytics projects [17].

2) Integration: The use of specific internal data representations makes integration of heterogeneous compute units difficult, because the data needs to be extracted and stored in a format that every accelerator can understand. This may be one of the reasons that adoption of heterogeneous accelerators in analytics has been slow in general, even for GPUs. Even though GPUs have been gaining popularity in datacenters for a variety of workloads including machine learning, there is only a handful of efforts in integrating them into analytics frameworks and supporting them in the mainline code repositories of popular open source frameworks such as Spark. Several research and non-upstream versions exist, but companies will (understandably!) be very cautious in deploying these in production.



(a) (De)serialization and copies are needed for interoperability between analytics frameworks, because every tool or platform uses its own data representation.



(b) Apache Arrow specifies a unified representation, preventing the need for (de)serialization and copies, allowing multiple tools or frameworks to operate on the same data.

Fig. 4: The benefits of a unified data representation such as Apache Arrow.

E. Apache Arrow

A number of industry and academic efforts aim to address the overhead associated with data serialization and deserialization that is needed for interoperability. In this paper we focus on the Apache Arrow project, which proposes that

no matter how you program your application, the data should be represented the same way in memory. This way, data does not need to be (de)serialized when passed between different software technologies (e.g. Java and Python).

In addition, the format uses a *column*-oriented layout as opposed to the *row*-oriented layout used in traditional systems. Although the choice to organize tables in a row- or column-oriented fashion has pros and cons depending on the way it is accessed (e.g. retrieving a full row of data is very efficient using a row-oriented format, but requires several random memory accesses in a columnar format), in the context of data analytics storing the data in a columnar format where data of the same type is stored contiguously has a number of advantages [18]. The first advantage is that this allows computations on this data to take advantage of SIMD instructions that are supported in most contemporary hardware. Several values are loaded into wide SIMD registers in the CPU, after which operations can be performed on all of them simultaneously using a single instruction. The Gandiva execution engine that targets Arrow-formatted data makes use of the LLVM compiler framework to generate highly optimized SIMD code for CPUs using JIT compilation.

The second advantage of using a columnar format is that it allows to send buffers of data containing a column of values of the same type to accelerators such as GPUs and FPGAs. This allows these columns to be streamed into functional units in a straightforward fashion, without the need for decoding values from a row-oriented buffer of entries that can contain complex, nested and even variable-length datatypes. Whereas row-oriented formats continue to be commonplace in transactional systems (that need to process frequent record updates), analytics systems tend to work in a column-oriented fashion.

To summarize, Apache Arrow not only facilitates interoperability between software, but also between different types of compute hardware.

III. Opportunities and Challenges

In our view, the success of an analytics system in the near future will revolve around leveraging a number of key properties listed in Table I. In the following sections, we will discuss recent developments in the areas of Hardware, Programming and Runtime, that we believe will provide opportunities for building these systems. Then, we will provide views on current efforts to address challenges or missing links in the field.

A. Opportunities: Recent developments in big data analytics systems

1) Hardware: In the area of computer hardware, recent developments have included several new components and interconnect technologies that will allow executing parts of the workload in the location and compute fabric that are most suitable for it.

TABLE I: Key system properties

Hardware	• Heterogeneous (various types of compute fabric) • Uniform memory access • Accelerated storage and networking
Programming	• Spatial (functional, declarative, etc.) programming paradigms • Graph representation (DAG) of the application should be scheduled optimally for the (heterogeneous) hardware • Allow easy integration of user-designed accelerators • Allow acceleration using IP libraries • A single common representation of data in memory (no serialization)
Runtime	• Monitor performance of running tasks • Optimize performance by 1) synthesizing new components and storing them in the IP library, 2) updating parameters of accelerators according to the changing data characteristics

Heterogeneous compute fabrics: In addition to the general-purpose GPU offerings that have become popular in recent years, FPGA-based datacenter accelerator cards have been introduced that can be integrated into existing servers [4]. In the near future, these accelerators will contain several types of compute fabric, including not only reconfigurable logic but also digital signal processing units, scalar processors and a grid of VLIW processors connected by a Network-on-Chip [19].

FPGA Overlays: As synthesis can take several hours especially for larger FPGAs, using a programmable overlay can provide a middle ground between performance and compilation time. An overlay is a fixed FPGA design containing compute elements that are highly optimized for a specific application or application domain. These elements can be programmed, and are typically supported by a compiler or other tool-chain that does not require synthesis. Notable examples are the Catapult project [20] that accelerated the Bing search engine in 2012 and various machine learning accelerator overlays such as the Deep Learning Processing Unit (DPU) overlay available in Xilinx' Vitis toolset [21].

Accelerators attached to storage and networking: The IBM Netezza system placed FPGAs between memory and storage to do (de)compression and pre-processing [22]. Similar hardware is now commonly available for custom use, in the form of storage-attached FPGA cards such as the Nallatech 250SoC [23], and 'smart' SSDs available from Samsung [24]. On the networking side, SmartNICs provide custom programmable acceleration and virtualization without the need for the CPU to intervene [25]. These technologies will be key in helping the CPUs cope with the increasing throughput from storage and memory.

Interconnect: For some workloads, connecting an accelerator via PCIe is sufficient, if copying buffers can be overlapped with computation. However, many workloads that could otherwise be accelerated are bound by the limited bandwidth and the high latency. Additionally, if the CPU needs to orchestrate all transfers, latency can become very high for all but the most straightforward memory access patterns.

Next-generation interfaces such as CXL and OpenCAPI

provide transparent, coherent access to main memory, making accelerators a peer of the CPU instead of a second class citizen. The current revision of OpenCAPI (3.0) provides throughput up to 25.6GB/s at a latency of 378 ns [26, p40].

High-Bandwidth Memory (HBM): As FPGAs have limited on-chip memory capacity, buffering enough data can be challenging. Current FPGAs have HBM integrated in the same package, providing several GBs of memory that can be used to buffer large chunks of data. HBM can be accessed by several ports simultaneously, allowing it to feed data into parallel kernels without using precious on-chip FPGA memory resources [27].

2) Programming: In spite of the gap between rising levels of abstraction and increased difficulty to design low-level accelerators, there are some developments that can aid in programming near-future analytics systems.

Parallel programming: Although many efforts in the area of parallel programming have not yet proven to be a silver bullet, and likewise OpenCL did not turn out to be the fit-for-all solution for programming accelerators, we do believe that certain properties of the workloads and the approaches in programming in the big data world provide some opportunities for acceleration. First, we note that a cluster is inherently very spatial in nature, bringing it conceptually closer to the FPGA. This is reflected in the way they are being programmed; declarative (SQL) and functional aspects (Scala) are being used pervasively. MapReduce, a highly popular programming model, maps very naturally to a spatial resource such as a cluster but also to FPGAs [28].

In the meantime, optimization frameworks such as Halide [29], initially developed for image processing, and TVM [30] targeting machine learning, have taken the route of separating the specification of functionality from the implementation (the *schedule*). The actual code is generated for the target platform, allowing design-space exploration and keeping platform-specific optimizations outside of the base program. Taking this concept a step further, efforts such as LIFT & ELEVATE [31], Temporal to Spatial (T2S) [32] and HeteroCL [33] propose a separate mechanism to describe the mapping of the program onto the target platform and the optimizations.

Distributed computing: Analytics frameworks distribute data over a large number of nodes in a resilient way, both in storage and computing [34].

Dataflow representation: Analytics frameworks represent applications as a Directed Acyclic Graph (DAG), which can be used by a scheduler to analyze which parts of the program to execute where. The DAG-representation is a much closer match for the dataflow-oriented nature of FPGAs.

Unified memory layout: By using Apache Arrow, the memory of an analytics program can be accessed by all types of compute fabric without needing to copy or convert.

3) Runtime: Fletcher [35] allows FPGAs to be integrated with big data frameworks, as long as the Arrow unified memory layout is used. As depicted in Figure 5, it generates interfaces based on the *schema*s of the data. The schema describes the layout of a table, including for example the columns and datatypes.

During runtime, the data is organized in 'chunks' of tables named *RecordBatches* in Arrow terminology. Fletcher hides the complexity of having to translate the streams of elements from the RecordBatches to actual memory addresses in the host main memory (or worse, in DRAM on the FPGA board), and requesting the needed transfers. As a result, the kernel developer can work on the level of streams of datatypes such as strings and integers, instead of byte values.

B. Missing links

1) Hardware: We recognize a large and rapidly increasing discrepancy between the high-level software and the low-level hardware. In software, complex datastructures are common with data stored in datatypes of possibly variable length (e.g. strings), using arrays that could even be nested. In contrast, AXI interfaces used in hardware provide a very simplistic way to transfer sequences of bytes, either in a memory-mapped or streaming fashion. It is up to the design to create a mapping between these, extracting the relevant data using custom glue logic that will likely change between applications.

A recently introduced specification called Tydi (Typed Dataflow Interfaces) [37] aims to address this discrepancy. With file formats specifying data layout on disk and Apache Arrow specifying data layout in memory, the Tydi standard specifies data layout 'on the wire' in digital hardware.

2) Programming: Ideally, accelerator kernels should be generated from the application's execution graph. Alternatively, a designer should be able to develop and integrate such a component quickly. One could envision a library of such components, where the transformations that each is capable of is known to a scheduler. This scheduler can then recognize whether certain parts of the execution graph already have an accelerated implementation available in the library.

An open question remains on what granularity to generate and store these components, particularly regarding the synthesis times associated with large FPGAs. Although big data applications can exhibit extensive runtimes, a latency of several hours to generate a bitstream may still prove prohibitive. Design choices include storing complete bitstreams or making use of partial reconfigurability in combination with an on-FPGA Network-on-Chip structure.

3) Runtime:

Partitioning and mapping: With the ability to design kernels to accelerate parts of an analytics application, comes the need to detect which parts to accelerate. For a heterogeneous system to be truly holistic, the scheduler responsible for this mapping needs to take into account not only the application graph in terms of transformations to be applied to the data, but also the amount and location of this data. If accelerating a subgraph or operation on a hardware accelerator means that communication overhead will become a limiting factor, a different partitioning should be considered.

Design-Space Exploration (DSE): Having more parameters means having more choices and in this context this means that the design space increases in size. Scheduling the DAG is only one aspect of the problem. When instantiating

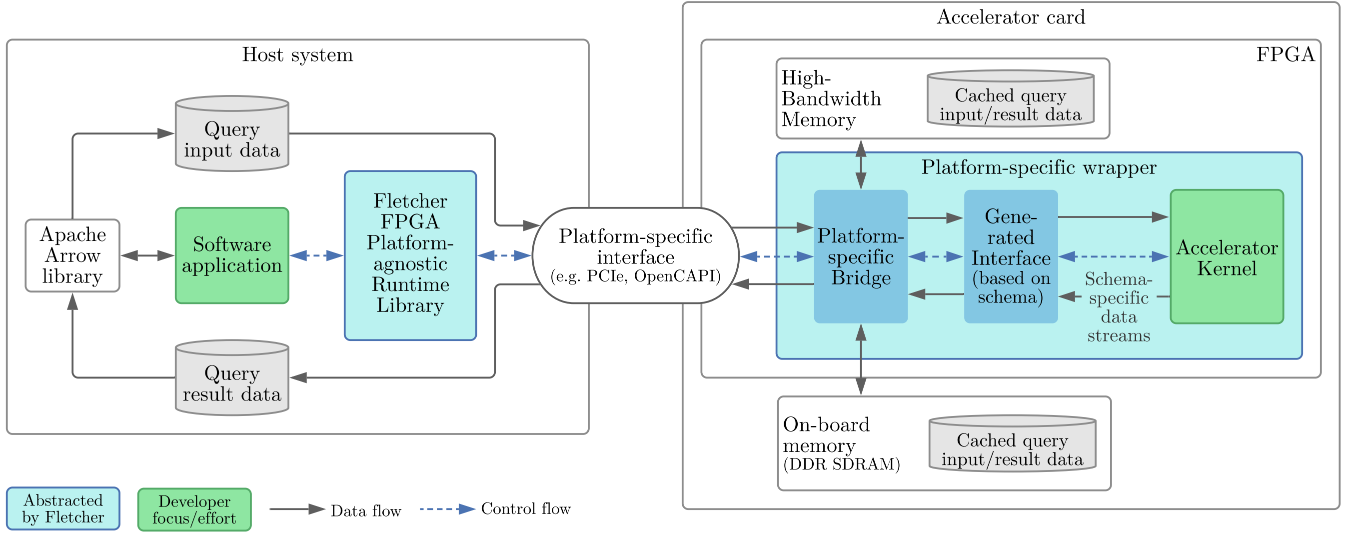


Fig. 5: Overview of the Fletcher HW data interface generator to integrate FPGAs with big data frameworks [36].

a kernel on an FPGA, there is a myriad of choices to be considered, such as the parallelism within each component versus the number of instances [38], the interface widths between them, tiling [39], etc.

Monitoring and auto-adaptation: Big data applications can run for extended amounts of time and the characteristics of the data can vary during this time. Based on these changing characteristics, the system parameters that were chosen by the DSE may no longer be optimal. Therefore, a runtime monitoring and adaptation loop can prevent operating in a mismatching configuration for extended periods of time. By sampling performance counters on the interfaces between kernel components, the runtime can identify which components and interfaces are over/underutilized and trigger a reconfiguration accordingly.

IV. FPGA-ACCELERATED BIG DATA SYSTEM

For the purpose of demonstration, we have implemented a proof-of-concept that incorporates many of the key elements highlighted in the previous section. The system is based on the popular Apache Spark framework on the software side, and on an OpenCAPI-based POWER9 platform with Xilinx VU37P FPGA on the hardware side. First, we will provide a high-level description. Then, we will discuss the system in terms of how it can be programmed, and how its components were integrated.

A. Architecture

The goal of our envisioned system is to provide seamless integration of FPGA accelerated applications in scale-out frameworks for big data analytics, such as Apache Spark. So far, we have illustrated the various gaps, mostly in programming and integration, that need to be closed to achieve this goal. A conceptual depiction of our proposed approach is shown in Figure 6.

On the left side of the figure, various computational steps of an analytics application are shown, represented as a DAG, consisting of multiple *stages*. Each stage performs a certain set of transformations on the data, until it is required to obtain data from partitions that may not be locally available (i.e. the data sits on other nodes). In the figure, the shuffle is required by the `ReduceByKey` transformation, which must gather all data items with the same key. As these items may be spread out over various partitions sitting on various cluster nodes, this transformation is called a *wide* transformation. Spark is especially good in taking care of many issues related to wide transformations, essentially dealing with general issues of scaling out computation over multiple nodes. Some related features are:

- Partitioning large data sets over multiple nodes.
- Distributing and optimizing descriptions of the desired computations.
- Scheduling the desired computations on all nodes.
- Recovering from node failures, through its resilience features.
- Offering an extensive set of libraries with algorithm implementations that perform better at scale.

Because such issues are dealt with by Spark itself, our system focuses only on improving transformations that are *narrow*, i.e. all data required for the computational steps is in a data partition local to the node that hosts the accelerator.

In our proposed system, the scheduler analyzes the DAG to see whether there are (parts of, or sub-)stages that can be efficiently accelerated on FPGA. It does so by searching a library that contains models of components that can implement the same parallel pattern and elemental function, depicted in the top-right corner of Figure 6. In the example, the `Map` pattern is applied, transforming each element in the dataset consisting of strings by applying the elemental function $x \to \{x, 1\}$. In other words, a string is transformed into a tuple of the string and the integer 1. There is a component model in

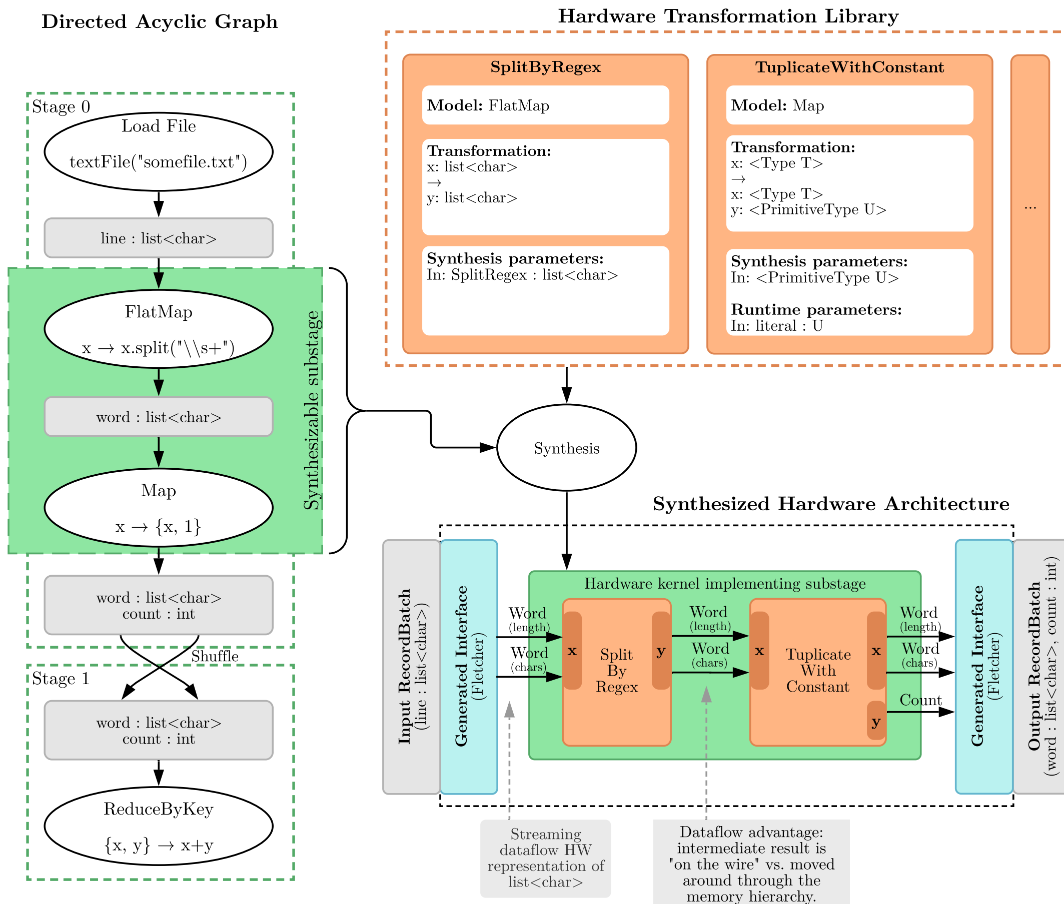


Fig. 6: Overview of a big data analytics system with transparent FPGA offloading. Conceptually, the input and output recordbatches can be streamed in from main memory but also directly from storage, network or other accelerators.

the library that implements the `Map` pattern, with the elemental function 'tuplicating' the element with a constant.

Because the previous transformation (the `FlatMap`) also has a hardware implementation, the synthesis stage can compose components according to the longest possible acceleratable part of the DAG within a single stage, to end up with a hardware accelerator design such as illustrated in the bottom-right corner of Figure 6. If the FPGA is directly attached to storage, it can absorb the file loading step and perform the first parts of the computation before any data has touched main memory yet (similar to predicate pushdown in database systems). Furthermore, OpenCAPI allows peripheral devices (I/O and accelerators) to transfer data between each other without going through main memory. That means that for example a network adapter could send data directly to an FPGA accelerator card.

It may happen that a certain design was previously synthesized to an FPGA bitstream, in which case the FPGA can be quickly reconfigured by loading the bitstream from a cluster-wide library or cache. If it was not previously synthesized, or if some synthesis-time parameters change, it must be resynthesized. We stipulate that in large clusters, the overhead of doing the full fledged FPGA synthesis is relatively low, because it only has to happen on one node out of potentially thousands. Furthermore, we also stipulate that workloads in big data clusters are often either long-running, or recurrent, and as such, the latency of bitstream synthesis may not pose as much of a problem as in other application domains.

After the correct bitstream has been obtained, the scheduler transforms the DAG into an FPGA-accelerated version, where the sub-stage that was implemented in the synthesized hardware accelerator is replaced by a software function that properly interfaces the data into the synthesized accelerator.

In a more advanced setup, the synthesized accelerator would be profiled to measure whether it actually speeds up the computation, such that it could be unloaded when the computation is not faster. In addition, the scheduler should take into account data movements and consider transforming the DAG to place accelerated transformations together. These efforts are left for future work.

We finally emphasize that describing the DAG (as a software programmer would), is closer to dataflow programming than traditional imperative programming, even though it is captured in languages that may also be used in the imperative style. This is useful for our system, because it allows many sorts of sub-stages to be easier to synthesize, as hardware dataflow designs can follow the structure of the DAG.

B. Programming practices

For the high-level programming patterns from the Spark program to be synthesized to an FPGA, the following elements are needed:

- The datasets residing in Spark need to be accessible by the FPGA accelerator kernel. As discussed, this can be done by using the Apache Arrow format for the data sets (i.e. the RecordBatches) in memory, in turn leveraging Fletcher to generate streaming hardware interfaces for the hardware kernel implementation.
- Once on the FPGA, the data needs to flow between the components implementing the DAGs transformations. Since the data may be complex and dynamically sized, a mechanism is required to consistently represent and transport such data structures over hardware streams. The recently introduced Tydi work aims to address this issue. It introduced an interface specification for streaming complex and dynamically sized data structures in hardware.
- The components that implement the parallel patterns (e.g. `Map`) and the elemental function that is to be performed on each element (e.g., the 'tuplication' function), need to be generated and properly connected according to the DAG.

We implemented a hardware description language called Tydal [40], aimed towards structural hardware design with streaming components adhering to the Tydi specification.The Tydal language contains template components that implement the same types of parallel patterns found in many modern cluster computing frameworks such as Spark. Current implementations of template components include `Map`, `FlatMap`, `Reduce`, and `Filter`. Other elementary operations known from streaming dataflow designs are also included, for example clone (to duplicate a stream), split (to extract members of stream with a compound element type into separate streams), and (de)mux. All constructs support the transfer of multiple primitive elements per handshake over their interfaces to be able to scale throughput of the design. More coarse-grained parallelism, by instantiating multiple parallel units, is supported by the language, but automating this is left for future work.

The template components are re-used in the proposed approach to implement hardware structures similar to the Spark DAG to be accelerated. This is done by instantiating the template matching the parallel pattern used in the Spark DAG, together with the appropriate elemental function found in the hardware transformation library. This automated mechanism can re-use the back-end of the language, which is currently written in Rust, to construct the hardware structure necessary to implement the sub-stage of the DAG.

An example of how the `Reduce` parallel pattern hardware template can be implemented is shown in Figure 7. In Tydi, components operating on streams are called *streamlets*. Tydi specifies a type system that allows to express complex and dynamically sized data types (e.g. sum or product types of dynamically-sized lists) flowing over streams, which form the interfaces of such streamlets. There is also a notion of dimensionality, denoted by d, to form dynamically-sized lists ($d = 1$), lists of lists ($d = 2$), etc. To perform the reduce transformation, it is required that the input dimensionality is at least one, such that the elements of the outermost dimension can be reduced using the elemental function component. The elemental function component only has to implement the reduction for two operands (e.g. an addition of two integers, with $d = 0$, a string concatenation where strings are a list of characters, i.e. $d = 1$). The whole parallel reduction can, for a single instance of the elemental function component, be implemented using the Reduction template component. In its most generic form, the Reduction template uses several multiplexers, allowing operands to stream in from the same interface, but split up to both elemental function component inputs. Because the first operand (essentially the first partial result) must be streamed in before the second one can be streamed in, it is required that it is temporarily stored in a FIFO, until the second operand has become available.

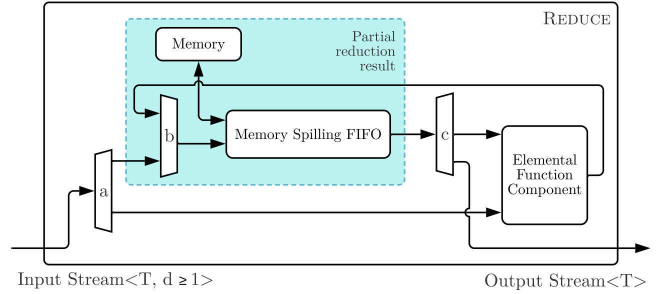


Fig. 7: Hardware architecture of component template for the reduce parallel pattern. Mux a initially splits two operands over the two inputs of the Elemental Function Component. If a partial result is too large to fit in whatever local FIFO size is chosen, it must spill to memory in case it is a dynamically sized data structure (for example, when the type to reduce is a string, i.e. $T = list < char >$). Mux b is optional, but allows for an initial value of the partial result to be set from the input. Demux c is required when there is no additional operand on the input, while the partial result has already begun streaming to the Memory Spilling FIFO before the last input is handshaked.

Since data can be dynamically sized and overflow the chosen dimensions of the FIFO holding the partial result, there must be some mechanism by which the dynamically sized data structure can spill to a larger memory, for example an on-board DRAM. Spilling could also be necessary if it is not possible to keep all needed intermediate data on-chip. The goal of Tydi is to stream data directly between components, to facilitate creating a fully pipelined dataflow design, but in certain complex cases it could be necessary to stream data to and from memory. In such cases, a DMA interface may be generated by Fletcher, since in our setup, all data structures adhere to the Arrow type system. Making sure data fits in a node's memory is a task of the task distribution step in typical analytics frameworks. For a detailed description of the design and implementation of templates for other parallel patterns, we refer the reader to [40].

C. Integration

At the time of writing, Spark does not yet support Arrow directly, although this has been stated as a long-term development goal. However, limited support for columnar processing was added recently, by means of the *ColumnarBatch* data structure (that is equivalent to Arrow's RecordBatch). This allows column-oriented implementations to be defined for physical operators to operate on batches of columnar data. Not many of such operators currently exist, but one example is the Parquet reader. The scheduler has support to combine row-oriented and column-oriented operators, and can insert conversions if necessary. An example of this can bee seen in Figure 8, showing a physical plan (which we will explain in more detail later) where the `FileSourceScanExec` uses the column-oriented Parquet reader, necessitating a conversion to the row-oriented format to allow Spark to perform all the other operations (for which no columnar implementations exist yet).

Spark has internal interfaces that can be extended, for example to implement custom implementations for certain operations. For the proof-of-concept, the relevant components were extended with a custom version of the *columnar* (to prevent column-row conversion) execution engine of certain operations, that we are able to accelerate using one of our kernels. These custom `FletcherExec` operations call the Fletcher runtime to execute the corresponding operation on the FPGA instead of going through normal code generation and execution on the CPU. In addition, an optimization pass was added to the scheduler, that traverses the execution DAG in search of operations or subgraphs that have an accelerated equivalent available. For the proof-of-concept, there is only one such function. Left for future work is adding the automatic synthesis step where the scheduler can also trigger the creation of a new accelerator.

V. Performance evaluation

A. Use-case

To evaluate the proof-of-concept, we have implemented a practical use case in the form of a typical query on a tabular dataset. The publicly available Chicago Taxi Trips dataset [41] loaded from a Parquet file is used as input. Dictionary encoding is disabled, because Fletcher does not yet support dictionaries. The query involves filtering by taxi company names using a regular expression, and accumulating the duration of all of their trips. Figure 8 shows the query on the left side, the physical execution of this query in Spark, and the nodes we have selected to accelerate on FPGA. The optimization pass that was added to the Spark scheduler transforms the DAG by replacing these nodes with a custom `FletcherExec` node that calls the accelerator. The row-column conversion step is not necessary through the use of Arrow; the file reader was replaced by the Parquet reader library function provided by the Arrow library, and the FPGA accelerator is based on Arrow and Fletcher, as discussed in previous sections.

The filter and projection step are composed into an accelerator design using the Tydal language associated with Tydi, shown schematically in Figure 9. Both the summation and regular expression are treated as a primitives. They can be either IP cores, manually implemented by the designer, and should be available in the component library. For the sum, a simple VHDL implementation is used. We use a tool that generates VHDL for the implementation of the regular expression matcher [42].

The regular expression matcher checks the predicate, producing a boolean that is in turn used by the `FilterStream` component. At an operating frequency of 200MHz (matching the interface frequency of OpenCAPI), it can process up to 20 characters per cycle. As the company names in the schema have a maximum length of 40 characters, the matcher can process a string every 2 cycles or less. The other components are fully pipelined and can process an element every cycle. As a result, this accelerator is capable of processing between 100 and 200 Mrecords/s, depending on how many records have a name string longer than 20 characters. As the kernel interfaces are 4 bytes wide for the integer input and 20 bytes wide for the string input, the theoretical maximum throughput is 4.8GB/s.

Important to note is that the presented Proof-of-Concept uses only a single instance of the accelerator. Because the query uses a commutative aggregation operator, it is straightforward to duplicate it, and distribute the input to them using an arbiter. This arbiter does need to keep the integer and string streams synchronized. When processing more complex queries, multiple instances of accelerators need to be managed by software. Development frameworks exist with the purpose of duplicating a kernel design provided by the developer, such as Fleet [43] and TaPaSCo [44]. As the tool-chain is currently in a very early prototyping stage, implementing parallel kernel instances is left as a future development.

B. Performance

Executing the query on the dataset, we measured the accelerator provides a throughput of 135 M records/s, for this particular dataset equaling 2.75 GB/s. The area utilization of the FPGA design is presented in Table II. This kernel can be duplicated several times, especially considering a large fraction of the logic will be shared. This design will in practice saturate the bandwidth of the OpenCAPI interface when using approximately 8 instances, which will likely fit in the available FPGA logic.

The most important result is that this performance was achieved from a very high-level implementation, even while the tool-chain used is in a very early stage of prototyping. This gives a reassuring perspective on the ability of the approach to generate high-performance FPGA circuits from these highly abstract big data applications.

Resource	Used	Available	Utilization
CLBs	17322	162960	10.63%
LUTs	85193	1303680	6.53%
Registers	101305	2607360	3.89%
BRAM tiles	108	2016	5.36%

TABLE II: FPGA resource utilization.

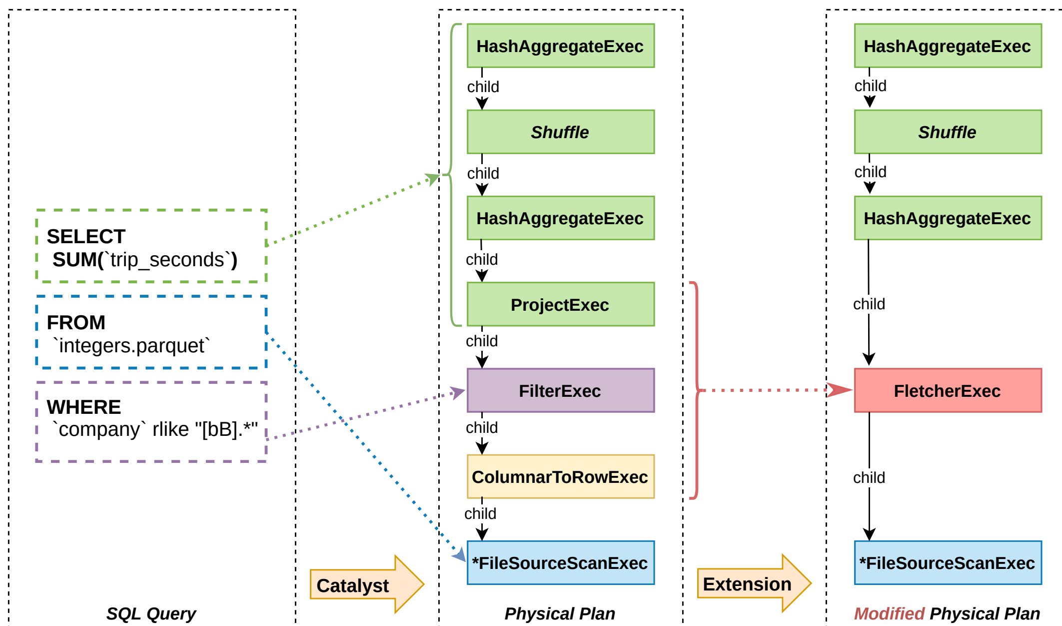


Fig. 8: Spark SQL's physical plan of our use-case and its modification.

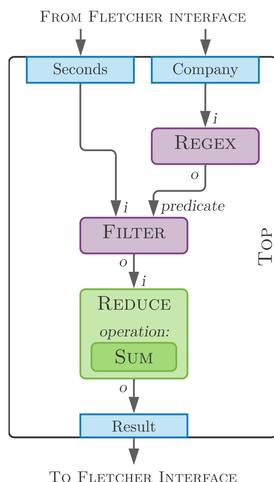


Fig. 9: Schematic of the hardware design created using the composition language.

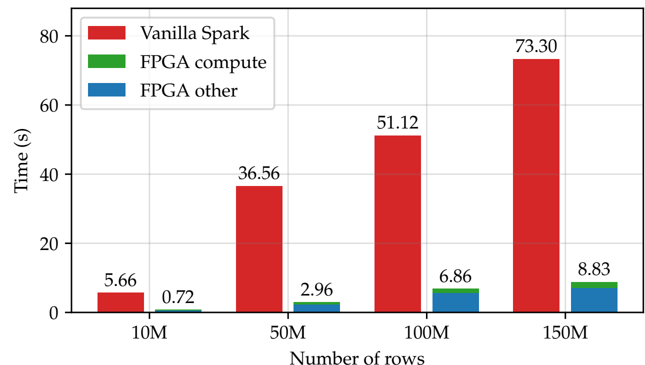


Fig. 10: Comparing the end-to-end run-time of FPGA-accelerated Spark to Vanilla Spark (Chicago taxi data regex use-case using a batch size of 1 million rows, one worker thread vs. one FPGA kernel instance)

In Figure 10, a comparison is plotted of running the query on a single instance of the FPGA accelerator versus a vanilla Spark installation running on a single thread. The Spark framework does not allow differentiating the Parquet reading from the actual execution (because Spark will interleave these steps and only perform reads from the file when blocks of data are being requested by lazy evaluation). However, for the FPGA-accelerated execution we can see that the actual execution takes only a small fraction of the total time.

VI. Related Work

As this work touches upon a wide range of subjects, a discussion of related literature will by no means be complete. However, we will attempt to mention efforts that have similar or alternative approaches.

A. Hardware description languages

With very large FPGAs becoming commonplace and multi-wafer projects falling in costs, exploring highly customized architectures for applications becomes accessible to a broader range of developers. With the ability to create new computer architectures more quickly, comes the desire for tools that can facilitate this [45].

B. High-Level Synthesis

In the domain of FPGA acceleration, High-Level Synthesis (HLS) tools have been introduced by vendors to generate

circuits from C/C++ and OpenCL code. Although this makes creating an initial solution on FPGA more accessible to developers with limited hardware knowledge, significant effort is required to achieve good quality of result. This includes tuning the source code and adding pragmas to guide the tools to make sensible design decisions. Recently, FPGA vendors have introduced their next generation development tools: Xilinx Vitis [21] and Intel's oneAPI [46]. They also lean heavily on a library of accelerated functions but on a different level than we propose in this work; Vitis and oneAPI both provide libraries that contain several examples of HLS implementations that can be easily modified by developers to suit their specific needs. OneAPI is accompanied with a new language called Data-Parallel C++, which aims to target various architectures (including CPUs, GPUs and FPGAs) from a single implementation.

C. Domain-specific hardware generation languages and frameworks

Several researchers believe that somewhere between the low-level RTL design languages and the high-level descriptions, there is a middle ground for describing high-performance circuitry with reasonable design time. To achieve this purpose, several languages have been developed such as CλASH [12], Pyrope [47], TLVerilog [11], and Chisel [48]. Some of these are embedded Domain-Specific Languages (eDSL), that extend an existing programming language such as Scala. eDSLs have the advantage that designs may use any constructs from the host language, often providing a high level of abstraction to generate component designs.

There are several frameworks that generate designs from a higher level description of an algorithm as input. These frameworks often build upon eDSLs, and usually rely on vendor synthesis tools (by generating old-fashioned RTL). Similar to the tools discussed in this work, they often support parallel patterns [49] such as `Map`, `Fold` (reduce), `Groupby`, etc. The DeLite tool [50] is a compiler architecture that can actually generate domain-specific hardware generation tools. It is built on top of MaxCompiler [10], which takes care of scheduling the dataflow graph. Spatial [9] is not only a hardware generation framework but also a full end-to-end acceleration framework including host runtime and infrastructure to communicate with FPGA. It includes design-space exploration functionality to guide the tiling of operations onto the FPGA fabric. Fleet [43] is another Scala eDSL offering a massively parallel streaming model for FPGAs.

D. Functional or declarative languages

Functional languages have properties that are believed to provide a more natural mapping to logic circuits [12]. Besides CλASH, functional descriptions are used for the input for the SPIRAL [51] and Lift [52] frameworks. They apply rewrite rules to lower the high-level sources towards various compute fabrics, including FPGA. The Elevate language adds to this a functional DSL to also allow a performance engineer to steer these transformations in the right direction [31].

In the analytics context, the declarative SQL language has been prevalent since the heyday of database systems. It has been used as a source language for generating logic by, for example, the Glacier project [53], [54].

E. Database acceleration

Accelerating SQL is only a part of the larger area of accelerating databases. Various types of accelerators specifically for databases have been proposed. For example, Netezza placed FPGAs between storage and main memory to do decompression and a number of pre-processing steps [22]. Swarm64 accelerates the open source PostgreSQL using FPGA [55]. Database workloads have several facets such as (de)compression, joining, filtering, and sorting, each having very different properties and suitability for FPGA. An overview of FPGA acceleration of in-memory databases is given in [56] and [57].

F. Big data frameworks employing accelerators

Although not widely supported yet, various academic and industrial efforts exist that aim to incorporate heterogeneous computing (e.g. Falcon computing targeting both GPUs and FPGAs [58]) or at least a certain type of accelerator into big data frameworks.

1) GPU: Compared to FPGAs, more work exists on supporting GPUs from big data frameworks. There are a number of reasons behind this. First, the GPU accelerator has been steadily building popularity for a longer period of time compared to FPGAs, and are more prevalent in datacenters and supercomputers. Second, they have proven to be very efficient in workloads related to machine learning. Third, analytics frameworks, in their turn, often provide very high-level libraries for machine learning due to its rapidly growing popularity (an example is MLlib for Apache Spark [59]). Targeting these high-level libraries directly provides a clear insertion point for GPUs. Although this approach will require serialization/deserialization as is discussed in this work, this approach is taken by several existing efforts [60], [61], [62]

The other approach used is to generate GPU code from the execution graph, similar to what is done in this work. The most well-known work is an industry project by Nvidia called RAPIDS [63], [64], [65]. Academic efforts include TensorFrame [66] and Spark-GPU [67].

2) FPGA: The open-source Blaze runtime system allows implementing FPGA-accelerated algorithms for a Spark data source[68]. Similar to the GPU work targeting high-level Spark libraries, this does not allow accelerating queries at the granularity of sub-stages in the execution graph, giving it less flexibility and only allowing to implement a whole algorithm on FPGA. The Kestrel Runtime from Falcon Computing [58] is built on this runtime.

Related efforts taking the same approach as this work include BigStream [69], which searches the Spark physical execution plan for FPGA acceleration candidates located in a bitstream store, and another novel project integrating Intel's "Spark Native SQL Engine" [70] with FPGAs [71].

As mentioned, FPGA usage in cloud instances is currently very explicit, with clients needing to acquire a full FPGA as opposed to CPU cycles that are much easier to share with other users. An overview of efforts aiming to virtualize FPGAs is presented in [72].

VII. Conclusion

In this article, we discussed our views on FPGA accelerators for near-future big data analytics systems, their challenges and how to address them, as well as the opportunities and how to leverage these. We also presented a proposed architecture for a scalable but still fully programmable system that can be built using technologies that are currently becoming generally available. We discussed the current difficulties involved with designing custom accelerators for FPGAs and how to improve design re-use and IP core connectivity standardization by using Tydi and Tydal. We present an analysis and evaluation of an FPGA-accelerated big data prototype that makes use of several of the discussed components.

While highly experimental, the prototype is able to generate a high-performing FPGA circuit from very high-level code descriptions in Spark. This is made possible by specifically targeting the big data application domain. This allows taking advantage of certain properties, including the DAG representation that is used internally in analytics frameworks and the highly parallel programming language constructs such as Map, filter and Reduce. These patterns are very suitable for large compute clusters, but also map very naturally to a spatial compute resource such as reconfigurable logic. On a final note, within the context of Apache Spark and Apache Arrow, a structured enough environment has been contributed that allows for the automatic generation of infrastructure for FPGA accelerators, without losing potential performance advantage.

Acknowledgements

The authors would like to thank the reviewers and in particular Hongbo Rong for the thoughtful review and valuable comments. The authors also thank Patrick Lysaght and Cathal McCabe from Xilinx for their support. This work is part of the FitOptiVis project funded by the ECSEL Joint Undertaking under grant number H2020-ECSEL-2017-2-783162. [73]

References

[1] Based on data from spec.org, see the attached link for an example of a plot of this data. [Online]. Available: https://liberty.princeton.edu/Projects/AutoPar/

[2] M. D. Hill and M. R. Marty, "Retrospective on Amdahl's Law in the Multicore Era," *Computer*, vol. 50, no. 6, pp. 12–14, 2017.

[3] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[4] Xilinx. Alveo. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo.html

[5] Intel. Intel Accelerator Cards. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/solutions/acceleration-hub/platforms.html

[6] H. Wong, V. Betz, and J. Rose, "Comparing FPGA vs. Custom CMOS and the Impact on Processor Microarchitecture," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 5–14. [Online]. Available: https://doi.org/10.1145/1950413.1950419

[7] Intel newsroom. Brian Krzanich: Our Strategy and The Future of Intel. [Online]. Available: https://newsroom.intel.com/editorials/brian-krzanich-our-strategy-and-the-future-of-intel/

[8] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, May 2019.

[9] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 296–311.

[10] Maxeler. MaxCompiler White Paper. [Online]. Available: https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf

[11] S. F. Hoover, "Timing-abstract circuit design in transaction-level verilog," in *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 525–532.

[12] C. Baaij, M. Kooijman, J. Kuper, W. Boeijink, and M. Gerards, "CλasH: Structural descriptions of synchronous hardware using haskell," in *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*. United States: IEEE Computer Society, 9 2010, pp. 714–721, eemcs-eprint-18376.

[13] M. Hendriks, H. A. Ara, M. Geilen, T. Basten, R. G. Marin, R. de Jong, and S. van der Vlugt, "Monotonic optimization of dataflow buffer sizes," *Journal of signal processing systems*, vol. 91, no. 1, pp. 21–32, 2019.

[14] F. Kruger. (2016, Mar.) Cpu bandwidth – the worrisome 2020 trend. [Online]. Available: https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/

[15] J. Fang, J. Chen, Z. Al-Ars, P. Hofstee, and J. Hidders, "Work-in-progress: A high-bandwidth snappy decompressor in reconfigurable logic," in *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sep. 2018, pp. 1–2.

[16] J. Peltenburg, A. Hesam, and Z. Al-Ars, "Pushing Big Data into Accelerators: Can the JVM Saturate Our Hardware?" in *International Conference on High Performance Computing*. Springer, 2017, pp. 220–236.

[17] J. Peltenburg, "Methods for Efficient Integration of FPGA Accelerators with Big Data Systems," Ph.D. dissertation, Delft University of Technology, 2020.

[18] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really?" in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 967–980. [Online]. Available: https://doi.org/10.1145/1376616.1376712

[19] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: Versal architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 84–93.

[20] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 13–24, doi: 10.1109/ISCA.2014.6853195.

[21] V. Kathail, "Xilinx Vitis Unified Software Platform," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 173–174.

[22] P. Francisco *et al.*, "The Netezza data appliance architecture: A platform for high performance data warehousing and analytics," 2011. [Online]. Available: https://www.ibmbigdatahub.com/sites/default/files/document/redguide_2011.pdf

[23] Bittware. Nallatech 250SoC. [Online]. Available: https://www.bittware.com/fpga/250-soc/

[24] 451 Research. Computational storage? We're the godfather, says Samsung. [Online]. Available: https://samsungsemiconductor-us.com/smartssd/wp-content/uploads/451_SmartSSD.pdf

[25] M. Tork, L. Maudlej, and M. Silberstein, "Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers," in *Proceedings*

of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 117–131. [Online]. Available: https://doi.org/10.1145/3373376.3378528

[26] B. Allison. OpenCAPI: Next generation of accelerartion for the cognitive era. [Online]. Available: https://www.slideshare.net/ganesannarayanasamy/opencapi-next-generation-accelerator

[27] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso, "High Bandwidth Memory on FPGAs: A Data Analytics Perspective," 2020. [Online]. Available: https://arxiv.org/abs/2004.01635

[28] K. Neshatpour, M. Malik, A. Sasan, S. Rafatirad, T. Mohsenin, H. Ghasemzadeh, and H. Homayoun, "Energy-efficient acceleration of MapReduce applications using FPGAs," *Journal of Parallel and Distributed Computing*, vol. 119, pp. 1–17, 2018.

[29] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[30] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[31] B. Hagedorn, J. Lenfers, T. Koehler, S. Gorlatch, and M. Steuwer, "A language for describing optimization strategies," 2020. [Online]. Available: https://arxiv.org/abs/2002.02268

[32] H. Rong, "Programmatic Control of a Compiler for Generating High-performance Spatial Hardware," 2017. [Online]. Available: https://arxiv.org/abs/1711.07606

[33] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 242–251.

[34] M. Zaharia, "An architecture for fast and general data processing on large clusters," Ph.D. dissertation, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2014.

[35] J. Peltenburg, J. van Straten, L. Wijtemans, L. van Leeuwen, Z. Al-Ars, and P. Hofstee, "Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2019, pp. 270–277.

[36] J. Peltenburg, J. van Straten, M. Brobbel, H. P. Hofstee, and Z. Al-Ars, "Supporting Columnar In-memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow," in *Applied Reconfigurable Computing*, C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz, Eds. Cham: Springer International Publishing, 2019, pp. 32–47.

[37] J. Peltenburg, J. van Straten, M. Brobbel, Z. Al-Ars, and H. P. Hofstee, "Tydi: an open specification for complex data structures over hardware streams," *IEEE Micro*, July/August 2020.

[38] G. Zhong, A. Prakash, S. Wang, Y. Liang, T. Mitra, and S. Niar, "Design Space exploration of FPGA-based accelerators with multi-level parallelism," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 1141–1146.

[39] G. Deest, N. Estibals, T. Yuki, S. Derrien, and S. Rajopadhye, "Towards Scalable and Efficient FPGA Stencil Accelerators," in *IMPACT'16 - 6th International Workshop on Polyhedral Compilation Techniques, held with HIPEAC'16*, Prague, Czech Republic, Jan. 2016. [Online]. Available: https://hal.inria.fr/hal-01425018

[40] A. Hadnagy, "Dataflow Hardware Design for Big Data Acceleration Using Typed Interfaces," Master's thesis, Delft University of Technology, 2020. [Online]. Available: http://resolver.tudelft.nl/uuid:38d9b35e-2d75-4cab-b6d1-723c2849badb

[41] Kaggle. (2020) Chicago taxi trips. [Online]. Available: https://www.kaggle.com/chicago/chicago-taxi-trips-bq

[42] J. van Straten, "vhdre: a vhdl regex matcher generator," 2019. [Online]. Available: https://github.com/abs-tudelft/vhdre

[43] J. Thomas, P. Hanrahan, and M. Zaharia, "Fleet: A Framework for Massively Parallel Streaming on FPGAs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 639–651. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/3373376.3378495

[44] J. Korinth, J. Hofmann, C. Heinz, and A. Koch, "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems," in *Applied Reconfigurable Computing*, C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz, Eds. Cham: Springer International Publishing, 2019, pp. 214–229.

[45] L. Truong and P. Hanrahan, "A golden age of hardware description languages: Applying programming language techniques to improve design productivity," *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 136, no. 7, pp. 1–7, 2019.

[46] Intel. oneAPI Programming Model. [Online]. Available: https://www.oneapi.com/

[47] Sheng-Hong Wang, Haven Skinner, Akash Sridhar, Sakshi Garg, Hunter Coffman, Kenneth Mayer, Rafael T. Possignolo, Jose Renau. Pyrope, a modern HDL with a live flow. [Online]. Available: https://masc.soe.ucsc.edu/pyrope.html

[48] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *DAC Design Automation Conference 2012*, June 2012, pp. 1212–1221.

[49] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, vol. 02-06-Apri, pp. 651–665, 2016.

[50] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, pp. 1–25, 2014.

[51] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. Moura, "Spiral: Extreme performance portability," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018.

[52] M. Kristien, B. Bodin, M. Steuwer, and C. Dubach, "High-level synthesis of functional patterns with lift," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 35–45, 2019.

[53] R. Mueller, J. Teubner, and G. Alonso, "Glacier: A query-to-hardware compiler," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 1159–1162. [Online]. Available: https://doi.org/10.1145/1807167.1807307

[54] ——, "Streams on wires: a query compiler for FPGAs," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 229–240, 2009.

[55] Swarm64. PostgreSQL Database Acceleration innovators - Swarm64. [Online]. Available: https://swarm64.com/

[56] P. Papaphilippou and W. Luk, "Accelerating Database Systems Using FPGAs: A Survey," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2018, pp. 125–1255.

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

[58] F. Computing. Kestrel runtime - big data application scheduling and load balancing. [Online]. Available: https://www.falconcomputing.com/falcon-kestrel-runtime/

[59] Apache software foundation. MLlib — Apache Spark. [Online]. Available: https://spark.apache.org/mllib/

[60] P. Li, Y. Luo, N. Zhang, and Y. Cao, "HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms," in *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 2015, pp. 347–348.

[61] R. Bordawekar. (2016) Nvidia GTC Silicon Valley 2016: Accelerating Spark Workloads Using GPUs. [Online]. Available: https://on-demand-gtc.gputechconf.com/gtcnew/sessionview.php?sessionName=s6280-accelerating+spark+workloads+using+gpus

[62] ——. (2016, 08) Accelerating Spark workloads using GPUs. [Online]. Available: https://www.oreilly.com/content/accelerating-spark-workloads-using-gpus/

[63] J. L. Robert Evans. (2020, 07) Spark+AI Summit 2020: Deep Dive into GPU Support in Apache Spark 3.x. [Online]. Available: https://databricks.com/de/session_na20/deep-dive-into-gpu-support-in-apache-spark-3-x

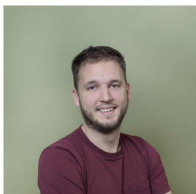
[64] NVIDIA. (2020) Spark RAPIDS plugin - accelerate Apache Spark with GPUs. [Online]. Available: https://github.com/NVIDIA/spark-rapids

[65] RAPIDS. Open gpu data science. [Online]. Available: https://rapids.ai/about.html

[66] Databricks, "TensorFrames: Experimental tensorflow binding for Scala and Apache Spark." April 2017. [Online]. Available: https://github.com/databricks/tensorframes
[67] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang, "Spark-GPU: An accelerated in-memory data processing engine on clusters," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 273–283.
[68] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: ACM, 2016, pp. 456–469.
[69] Bigstream. Hyperacceleration with bigstream technology. [Online]. Available: https://blog.bigstream.co/resources/hyper-acceleration-with-bigstream-technology
[70] Intel. (2020) Spark Native SQL Engine - Optimized Analytics Package for Spark Platform. [Online]. Available: https://github.com/Intel-bigdata/OAP/tree/master/oap-native-sql
[71] W. C. Calvin Hung. (2020) Spark+AI Summit 2020: Accelerating Spark SQL Workloads to 50X Performance with Apache Arrow-Based FPGA Accelerators. [Online]. Available: https://databricks.com/session_na20/accelerating-spark-sql-workloads-to-50x-performance-with-apache-arrow-based-fpga-accelerators
[72] A. Vaishnav, K. D. Pham, and D. Koch, "A Survey on FPGA Virtualization," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 131–1317.
[73] Z. Al-Ars, T. Basten, A. de Beer, M. Geilen, D. Goswami, P. Jääskeläinen, J. Kadlec, M. M. de Alejandro, F. Palumbo, G. Peeren, and et al., "The FitOptiVis ECSEL Project: Highly Efficient Distributed Embedded Image/Video Processing in Cyber-Physical Systems," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, ser. CF '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 333–338. [Online]. Available: https://doi.org/10.1145/3310273.3323437

AUTHOR BIOGRAPHIES

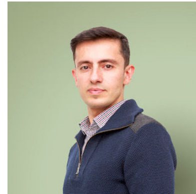
Joost Hoozemans Received his BSc in Computer Science from Utrecht University in 2011 and his MSc and PhD in Computer Engineering from Delft University of Technology in 2014 and 2018, respectively. His research interests include VLIW and TTA processors, reconfigurable computing and FPGA programmability, and dataflow computing.



Johan Peltenburg is a Ph.D. candidate with the Accelerated Big Data Systems Group, TU Delft. He focuses on FPGA accelerators for big data applications, working on the Fletcher accelerator framework. He is a member of IEEE.

Fabian Nonnemacher received his BEng in Computer Science from the DHBW Stuttgart in 2015, and his MSc in Computer Science from the Delft University of Technology in 2020. His research interests include distributed systems, big data analytics and heterogeneous computer architectures.

Ákos Hadnagy received his BSc in Electrical Engineering from the Budapest University of Technology and Economics in 2018, and his MSc in Computer Engineering from the Delft University of Technology in 2020. His research interests include high-level FPGA design, hardware acceleration, and heterogeneous computer architectures.



Zaid Al-Ars is an Associate Professor with TU Delft, where he leads the Accelerated Big Data Systems Group. His work focuses on developing computing infrastructures for efficient processing of big data analytics applications. He is co-founder of a couple of big data companies specialized in high performance analytics solutions and AI. He is also on the advisory board of a number of high-tech start-ups. He published more than 150 peer-reviewed publications and holds two patents. He is a member of IEEE.

H. Peter Hofstee is a distinguished research staff member at IBM and part-time Professor at TU Delft, best known for his contributions to heterogeneous computing as a chief architect of the Synergistic Processor Elements in the Cell Broadband Engine used in PlayStation 3, and the first supercomputer to reach sustained petaflop operation. He currently focuses on optimizing the system roadmap for big data, analytics, and cloud, including the use of accelerated computation. Recent contributions include coherently attached reconfigurable acceleration on POWER7, paving the way for the new coherent attach processor interface on POWER8 through POWER10. He holds more than 100 issued patents. He is a member of IEEE. Contact him at hofstee@us.ibm.com.