Generating high-performance FPGA accelerator designs for big data analytics with Fletcher and Apache Arrow

Peltenburg, Johan; van Straten, Jeroen; Brobbel, Matthijs; Al-Ars, Zaid; Hofstee, H. Peter

DOI
10.1007/s11265-021-01650-6

Publication date
2021

Document Version
Final published version

Published in
Journal of Signal Processing Systems: the journal of DSPtechnologies

Citation (APA)
Peltenburg, J., van Straten, J., Brobbel, M., Al-Ars, Z., & Hofstee, H. P. (2021). Generating high-performance FPGA accelerator designs for big data analytics with Fletcher and Apache Arrow. *Journal of Signal Processing Systems: the journal of DSPtechnologies*, *93*(5), 565-586. https://doi.org/10.1007/s11265-021-01650-6

Journal of Signal Processing Systems (2021) 93:565–586
https://doi.org/10.1007/s11265-021-01650-6

Generating High-Performance FPGA Accelerator Designs for Big Data Analytics with Fletcher and Apache Arrow

Johan Peltenburg[1] · **Jeroen van Straten**[1] · **Matthijs Brobbel**[1] · **Zaid Al-Ars**[1] · **H. Peter Hofstee**[1,2]

Received: 2 May 2020 / Revised: 28 January 2021 / Accepted: 10 February 2021 / Published online: 1 March 2021

Abstract

As big data analytics systems are squeezing out the last bits of performance of CPUs and GPUs, the next near-term and widely available alternative industry is considering for higher performance in the data center and cloud is the FPGA accelerator. We discuss several challenges a developer has to face when designing and integrating FPGA accelerators for big data analytics pipelines. On the software side, we observe complex run-time systems, hardware-unfriendly in-memory layouts of data sets, and (de)serialization overhead. On the hardware side, we observe a relative lack of platform-agnostic open-source tooling, a high design effort for data structure-specific interfaces, and a high design effort for infrastructure. The open source Fletcher framework addresses these challenges. It is built on top of Apache Arrow, which provides a common, hardware-friendly in-memory format to allow zero-copy communication of large tabular data, preventing (de)serialization overhead. Fletcher adds FPGA accelerators to the list of over eleven supported software languages. To deal with the hardware challenges, we present Arrow-specific components, providing easy-to-use, high-performance interfaces to accelerated kernels. The components are combined based on a generic architecture that is specialized according to the application through an extensive infrastructure generation framework that is presented in this article. All generated hardware is vendor-agnostic, and software drivers add a platform-agnostic layer, allowing users to create portable implementations.

Keywords FPGA · Accelerator · Big data · Analytics · Fletcher · Apache Arrow

1 Introduction

In terms of both hardware and software, the increasing heterogeneity in big data analytics systems causes major challenges [1]. FPGA accelerators are the next near-term and widely available alternative for higher performance in the data center. While such accelerators can perform well under specific circumstances, designing for such systems still requires a relatively large amount of effort, increasing the total cost of FPGA accelerated solutions. Within the context of big data analytics pipelines, this article will give an overview of some of the challenges from the FPGA developer point of view.

✉ Johan Peltenburg
j.w.peltenburg@tudelft.nl

H. Peter Hofstee
hofstee@us.ibm.com

[1] Delft University of Technology, Delft, Netherlands

[2] IBM, Austin, TX, USA

The challenges are mainly related to integration, where there are two sides to the coin. On the one hand, it is time consuming for developers to set up accelerator designs that integrate easily with big data analytics pipelines. On the other hand, big data analytics pipelines know many layers of abstraction and are built on technologies that are not designed to work well with heterogeneous components.

In this article, we first contribute a high-level overview and software language analysis of contemporary big data analytics frameworks. We then stipulate the challenges associated with such software components that a hardware developer faces when integrating FPGA accelerators into big data analytics pipelines. To deal with the challenges described, we contribute a thorough description of the internals of an open-source project called Fletcher [2] [3]; an FPGA accelerator framework designed to deal with these challenges. Fletcher is built on Apache Arrow, which provides a common in-memory format for tabular data sets found in big data analytics applications, to be used by any sort of (mainly software) technology. By providing a common format, moving data between heterogeneous processes such as Python or Java can be done

without serialization overhead, something that also FPGA accelerator implementations can benefit from through the use of Fletcher.

To this end, we furthermore contribute in this article:

- A set of open-source domain-specific hardware components for streaming dataflow design; vhlib [4].
- An open-source hardware construction library; *Cerata* [5],
- An open-source tool based on Cerata and Apache Arrow, used to generate easy-to-use high-throughput hardware interfaces to Apache Arrow formatted data sets in memory; *Fletchgen* [6]
- An open-source tool to generate AXI4-lite MMIO infrastructure for register files; *vhdMMIO* [7].

Together, these components form a tool chain named Fletcher that decreases the development time of FPGA accelerator implementations that are to be integrated with big data analytics frameworks using Apache Arrow. The user only defines an Apache Arrow description of the (potentially nontrivial, e.g. variable-length and nested) column types in the tabular data structure. The user then passes this description to Fletchgen, which generates a platform-agnostic FPGA accelerator infrastructure, completely tailored and optimized for the supplied schema, and fully integrated with the host system software, where the only thing that remains is the implementation of the application-specific hardware kernel. To this end, we contribute also the run-time software stack associated with this mechanism [8]. The kernel is supplied with easy-to-use interfaces, where rather than a memory bus with raw bytes, the user can request access to data based on tabular row indices, and is supplied with streams corresponding to the Apache Arrow types. We finally give an example of how the tools are used to decrease the development effort of FPGA accelerated solutions in big data analytics pipelines.

The remainder of this article is structured as follows. In Section 2, we discuss the background of this article, discuss the challenges for FPGA accelerators to become feasible alternatives to general-purpose or GPGPU computing in big data analytics systems, and provide some lessons learned throughout the development of the Fletcher framework. In Section 3, we discuss some of the basic hardware components that lie at the foundation of the Fletcher framework. These are used in an automated infrastructure generation step to create custom, easy-to-use interfaces, based on high level descriptions of potentially relatively complex tabular data structures that the FPGA accelerator must operate on. This infrastructure generation step will be explained in detail in Section 4, where we also describe the underlying toolchain with three novel implementations of the tools required to generate designs of such a dynamic nature. We will proceed to explain the toolchain by examples in Section 5. Section 6 will discuss the lessons learned and provide an outlook on the future of Fletcher. Section 7 concludes this paper.

2 Background

Big data systems are reaching maturity in terms of squeezing out the last bits of performance of CPUs or even GPUs. The next near-term and widely available alternative for higher performance in the data center and cloud may be the FPGA accelerator.

Coming from the embedded systems and prototyping-oriented market, FPGA vendors have broadened their focus towards the data center by releasing accelerator cards with similar form factors and interfaces as GPGPUs. Various commercial parties offer cloud infrastructure nodes with FPGA accelerator cards attached. FPGA accelerators have also been successfully deployed at a large scale in commercial clusters of large companies (e.g. [9]).

Whether the FPGA accelerator in the data center will become an implementation platform as common as other accelerators, such as GPGPUs, is still an open question. The answer will depend on the economic advantages that these systems will offer; will they provide a lower cost per query? Will they provide more performance per dollar?

In an attempt to answer these questions, valid reasons to be skeptical about embracing FPGA accelerators in the data center exist. We stipulate three disadvantages within this context:

1. Technological disadvantage: FPGAs run at relatively low clock frequencies and require more silicon to implement the same operation compared to a CPU or GPGPU, requiring the specialized circuits they implement to be orders of magnitude more efficient at whatever computation they perform before they provide an economically viable alternative.
2. Hard to program: A notorious property of FPGAs is that they are hard to program, incurring high nonrecurring engineering costs; a higher cost per query or more dollars to achieve decent performance.
3. Vendor-specific: Relative to the software ecosystem in the field of big data analytics, one could observe a lack of reusable, vendor-agnostic, open-source tooling and standardization. The big data analytics community has shown to thrive and rely specifically on open-source frameworks, as this provides more control over their systems and prevents vendor lock-in.

On the other hand, valid reasons to be optimistic exist as well, because of the following advantages.

1. Specialization: FPGAs are able to implement specialized data flow architectures that, contrary to load-store architecture-based machines, do not always require the intermediate results of fine-grained computations to spill to memory, but rather pass them to the next computational stage immediately. This often leads to either increased performance or to increased energy efficiency, both of which may provide an economic advantage.
2. Hardware integration: FPGAs have excellent I/O capabilities that help to integrate them in places the GPGPU cannot (yet) go, for example, between the host CPU and network and storage resources. This can help to build solutions with very low latency compared to CPUs and GPGPUs.

2.1 Hardware Design Challenges

The two mentioned advantages have the potential to mitigate the first disadvantage in specific cases, which leads us to mainly worry about the problem of productivity. One branch of approaches that the research and industrial community takes to increase productivity is to say: hardware is hard to design while software is easy to program, therefore we should be able to write software resulting in a hardware design. While the term has become ambiguous, this approach is called High-Level Synthesis (HLS), which we interpret here as; using a description of a software program to generate a hardware circuit performing the same function, hopefully with better performance. A thorough overview of HLS tools can be found in [10].

The HLS approach can (arguably) lead to disappointment on the side of the developer, since it is easy to enter a state of cognitive dissonance during programming. A user with a software design background may find many constructs and libraries not applicable or synthesizable in a language that s(he) thinks to understand. Hardware-specific knowledge must be acquired, and often vendor-specific pragmatism must be applied to end up with a working implementation. A user with a hardware design background may experience a lack of control that may result in a suboptimal design, hampering the intended performance that they know could be achieved using a HDL. Software languages are designed with the intent to abstract CPU instructions, memory, and I/O, but not the gates, connections, and registers that hardware-oriented users desire to express more explicitly than what is allowed by most software-oriented languages. A recent meta-analysis of academic literature [11] shows that designs created with HLS techniques at a reduced design effort of about 3× still show only half the performance compared to HDL designs, although the meta-study includes designs in frameworks that would classify as an HDL approach (e.g. Chisel) more than HLS, according to our definition. Since the direct competitor is the server-grade CPU and the GPGPU, it is in many cases unlikely that losing half the performance is acceptable.

For the reasons mentioned above, we argue (together with [12, 13]) for a different approach to attack the ”hard-to-program” problem; hardware is hard to design, therefore we need to provide hardware developers with abstractions that make it easier to design hardware. Such abstractions are easier to provide when the context of the problem is narrow, leading to domain-specific approaches. We must increasingly take care that these abstractions incur zero overhead, since technologically, we are getting close to an era where the added cost of abstraction cannot be mitigated by more transistors, due to the slowdown of Moore’s law.

We stipulate three FPGA-specific challenges from the hardware development point of view when designing FPGA-based hardware accelerators for big data systems that cause a substantial amount of development effort.

H1 **Portability**: Highly vendor-specific styles of designing hardware accelerators prevent widespread reuse of existing solutions, often leading hardware developers to ‘roll their own’ implementations. It also makes it hard to switch implementations to different FPGA accelerator platforms of different vendors.

H2 **Interface design**: Developers spend a lot of time on designing interfaces appropriate for their data structure, since they are typically provided with just a byte-addressable memory interface. This involves the tedious work of designing appropriate state machines to perform all pointer arithmetic and handle all bus requests.

H3 **Infrastructure**: Hardware developers spend a lot of time on the infrastructure or sometimes colloquially called ‘plumbing’ around their kernels, including buffers, arbiters, etc., while their focus is the kernel itself.

2.2 Big Data System Integration

Not only FPGA-based designs themselves can be very complex — the big data analytics frameworks in which they need to be integrated are very complex as well. For the sake of the discussion in this article, we are going to assume that there is a hardware developer wanting to alleviate some bottlenecks in a big data analytics pipeline implemented in software through the use of an FPGA accelerator. In such a context, it is safe to assume that there is a lot of data to be analyzed. The FPGA accelerator must have access to this data.

Assuming the analytics pipeline to be implemented in the C programming language, a programmer may point to

their efficiently packed, hand-crafted `structs`, `unions`, arrays, pointers to nested dynamically-sized data structures, and eventually the primitive types of data that makes up the data structure of interest. Were this data structure to be somewhat inefficiently laid out in memory in terms of feeding it to the accelerator, the programmer would be able to easily modify the exact byte-level layout of the data structure in memory, typically causing the data to reside in regions of memory that are as contiguous as possible, such that they can be loaded into the FPGA using large bursts, preventing interface latency from becoming a bottleneck when many pointers need to be traversed. These assumptions are reasonable and describe a common design pattern in hardware acceleration of software written in low-level languages such as C. However, we will show that in the domain of big data analytics, these assumptions usually do not hold.

We have analyzed the code bases of many active and widely used open-source projects related to big data analytics. The goal is to answer the question: what languages are mostly used in the big data ecosystem? While there are hundreds of candidates in the open-source space alone, we have selected projects that are commonly found in the middleware of the infrastructure. This is where accelerators are most likely to be integrated. We therefore do not include frameworks focused on specific applications or end-users (e.g., deep learning or business intelligence), since they are often built on top of the middleware frameworks that we analyzed.

The overview of the frameworks that were analyzed is as follows:

- 8 query engines: PrestoDB, Cloudera Hue, Dremio, and Hive, Drill, Impala, Kylin, Phoenix
- 7 stream processing engines: Heron, Samza, Beam, Storm, Kafka, Druid, Flink
- 15 (in-memory) data storage engines: MongoDB, CouchDB, Cassandra, CockRoachDB, CouchDB, OpenTSDB, Accumulo, Riak, HBase, Kudu, Redis, Memcached, Hadoop-HDFS, Sqoop, Arrow
- 9 management and security frameworks: Airflow, ZooKeeper, Helix, Atlas, Prometheus, Knox, Metron, Ranger
- 6 hybrid general-purpose frameworks: Mesos, Hadoop, Tez, CDAP, Spark, Dask
- 4 logging frameworks: Flume, Fluent Bit, Fluentd, Logstash
- 2 search frameworks: ElasticSearch, Lucene-Solr
- 3 messaging / RPC frameworks: RocketMQ, Akka, Thrift

A pie chart of the analysis is shown in Fig. 1. From the figure, we may find that the vast majority of the codebase is written in Java, followed by Python, with C/C++ taking up about 15% of the lines of code. These figures indicate the most widely used run-time technologies in big data analytics pipelines.

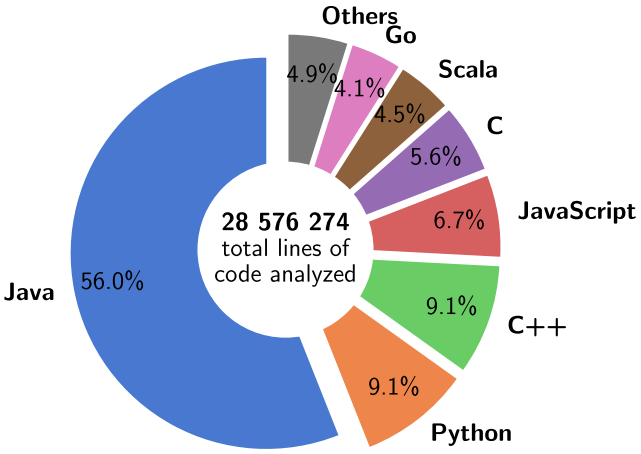

Figure 1 Language analysis of 52 open-source projects from the big data ecosystem.

About 80% of the code found in the ecosystem is written in languages that typically alleviate the burden of low-level memory management by various methods that cause several problems. First, garbage collection (GC) is applied to prevent memory leaks, sometimes causing data to move around the memory, invalidating any pointers to the data, causing the need to halt the software run-time when FPGA accelerators would be operating on the data. Second, extensive standard libraries with containers for many typical data structures (e.g., strings, dynamically sized arrays, hash maps) are commonly used. This decreases the development effort and provides a form of standardization within a language. However, the language-specific in-memory formats of these containers often do not correspond well to how it would be preferable for FPGA accelerators to access the data. Finally, data is often wrapped into objects (e.g., in Python), although the native architecture supports a specific data type in hardware. While in C an array of a thousand integers is relatively simple in memory, in Python this looks like an array with a thousand pointers to boxed integer objects, which is not very efficient to access with high throughput, as it is potentially highly fragmented. Furthermore, these objects contain language and run-time specific metadata that are of absolutely no use to an accelerator, such as pointers to the class of the object in Java.

Discussing the details of all these techniques is outside the scope of this article, but we summarize the discussion to the following challenges for developers wanting to integrate an FPGA accelerator solution into a software-oriented big data analytics pipeline:

S1. **Complex run-time systems**: it is hard to get to the data, because it is hidden under many layers of automated memory management.
S2. **Hardware-unfriendly layout**: the data is laid out in a way that is most practical for the language run-time system, with a lot of additional bytes containing data that is uninteresting to the FPGA accelerator. A more FPGA-friendly in-memory format of the data structure must be designed to make it accessible to the FPGA accelerator.
S3. **(De)serialization**: Even if one would handcraft such a format, one would have to *serialize* the input data for the accelerator into that format, and then *deserialize* the result back into a format that the language run-time would understand. The throughput of (de)serialization is relatively low compared to modern accelerator interfaces, and can easily lead to performance bottlenecks [14].

2.3 Apache Arrow

Due to the nature of this article, the challenges S1, S2, and S3 from the previous section were described mainly from an FPGA acceleration developer point of view. However, even within the software ecosystem of big data analytics pipelines, such challenges exist. When heterogeneous processes interact (e.g., when there is inter-process communication between a pure Java program offloading some computation to a very fast C library), there needs to be one common (in-memory) format that both programs agree on. Several projects have provided such a common format for generic types of data, such as Google's Protobuf [15]. The project provides a code generation step to automatically generate serialization and deserialization functions that help produce and consume data in the common format, turning it back into language-native in-memory objects, such that programmers can continue to work with them in the fashion of their language.

Later, it was realized that serialization and deserialization itself can cause bottlenecks, since copies have to be made twice; first, when serializing the data to the common format at the producer side, and again, when deserializing it on the consumer side. In many cases, providing specialized functions to access the data in its common format turns out to be faster than applying serialization and deserialization, since data may be passed between processes without making any copies to restructure it into a language-specific format. This has led to what is called a zero-copy approach to inter-process communication. Through the help of libraries such as Flatbuffers [16], such functions are provided to several languages. Producing processes immediately use the common format for their data structure, and then only share a pointer to the data with the consuming process. No copies are made because both processes work with the common format as much as possible from the same location in memory. Programmers are provided with language-specific libraries that make it easy for them to interact with the data structure according to the fashion of their language.

An approach similar to Flatbuffers, but specifically tailored to big data analytics, is found in the Apache Arrow project [17]. Apache Arrow is specifically tailored to work with large tabular data structures that are stored in memory in a column-oriented fashion. While iterating over column entries in tables, the columnar format causes more efficient use of CPU caches and vector instructions than a row-oriented format. It also provides a memory management daemon called Plasma, that allows to place data structures outside the heaps of garbage collected run-time systems, providing interfaces for zero-copy inter-process communication of Arrow data sets.

Thus, Arrow specifically solves the challenges S1, S2, and S3 by, respectively:

1. Allowing data to be stored off-heap, unburdened by GC.
2. Providing a common in-memory format and language-specific libraries to access the data, preventing the need for serialization.
3. Tailoring the format to work well on modern CPUs by being column-oriented.

2.4 Fletcher

Previous studies have shown that inefficiencies in serialization of data from language run-times with automated memory management may cause more than an order of magnitude decrease in throughput compared to modern accelerator interfaces to host memory, that contemporary protocols such as PCIe, CXL, CCIX, or OpenCAPI (intend to) provide [14]. Therefore, the benefits of Apache Arrow can help alleviate bottlenecks in the context of FPGA accelerators as well.

Fletcher is an open-source FPGA accelerator framework specifically built on top of Apache Arrow, with the intent to not only solve challenges S1, S2, and S3 on the big data analytics framework integration side, but also to solve challenges H1, H2, and H3 on the hardware development side. This is illustrated in Fig. 2.

Previous articles have discussed, at a very high level, the idea behind the framework, and have shown several use cases [2]. These use cases have shown that through the use of Arrow and Fletcher, serialization overhead can be prevented, since the Arrow format is highly suitable for hardware accelerators, allowing the accelerators to perform at the bandwidth of the accelerator interface to host memory.

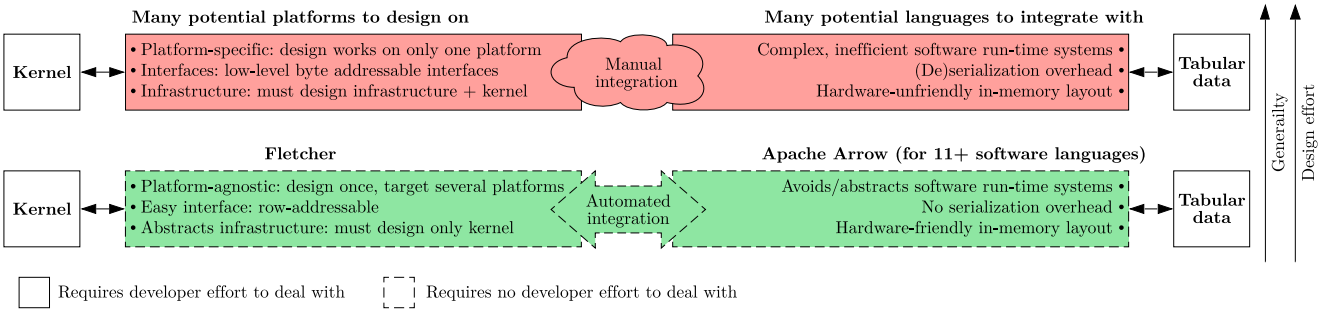

Figure 2 Advantages of the Fletcher FPGA accelerator framework for big data systems, based on apache arrow.

An brief summary and overview of the framework as used by the developer during compile time and during run time is shown in Fig. 3.

When accessing tabular data, one would prefer to do so through row indices rather than byte addresses. This has led the Fletcher project to construct specific low-level hardware components with streamable interfaces, that allow to provide a range of row indices, returning one or multiple streams of data corresponding to the types of Arrow tables. In contrast to a byte-addressable memory interface, this addresses the challenge H2. We will briefly reiterate the design of these components in Section 3.

In the remainder of this article, we will describe how Fletcher deals with the challenges H1, the problem of portability, and H3, the problem of the infrastructure design effort.

2.5 Related Work

While many commercial tools exist that automate infrastructure design, most of them are geared towards the HLS approach, but provide little help to users that, for the reasons mentioned above, prefer to work with HDLs to describe their solutions. HLS tools are also known to have problems dealing with dynamic data structures, as described in [18], that Arrow allows to express. In Section 3, we show a method specific to Arrow for traversing the dynamic structures efficiently. Previous research has extensively investi-

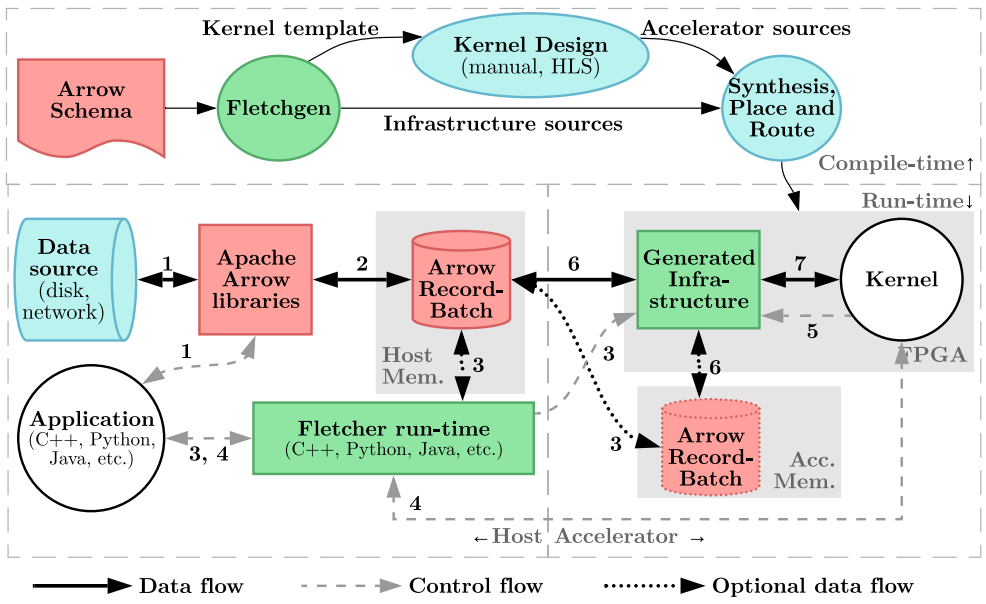

Figure 3 Fletcher overview. During compile time (top), the user specifies an Arrow Schema, runs Fletchgen to generate hardware infrastructure sources and a kernel template, implements the kernel, and finally places and routes their design. During run-time (bottom), applications use Apache Arrow to load data from some source (1) into memory in the form of RecordBatches (2). They can then use the Fletcher run-time library to make RecordBatches available to the FPGA accelerator (3) or to obtain new RecordBatches from the FPGA (4). The hardware kernel sends reads or write requests (5) to send or receive data to/from host- or on-board memory (6) through the generated infrastructure. The generated infrastructure deals with the byte-oriented memory interface, turning the data into hardware streams (and vice versa) (7) such that the streams match the kernel interfaces with the potentially complex and nested types of the Arrow schema.

gated hardware interfaces for more generic C-style dynamic data structures through specialized DMA engines [19], but does not focus on integration with modern software frameworks from the big data analytics ecosystem analysis. To the best of our knowledge, Fletcher is the only open source FPGA accelerator framework that deals with challenge H3, in the context of big data analytics on tabular data sets for those that prefer an HDL design flow specifically.

A number of frameworks do exist that help deal with challenge H1. We first give an overview of related work regarding challenge H1, also shown in Table 1. This helps us compare Fletcher to existing frameworks and stipulate the differences. We use the following criteria to include specific frameworks in our comparison:

- The framework is active and publicly available open-source.
- The framework targets datacenter-grade accelerator cards/platforms.
- The framework provides abstractions that provide some form of portability between such cards/platforms.

As shown in the table, there are currently a small number of other frameworks that adhere to these criteria. TaPaSCo [20] allows designers to easily set up systems that perform several hardware-accelerated tasks in parallel. It is in some sense complementary to Fletcher, since (as will be discussed again later) Fletcher provides an AXI4 top level for memory access, alongside an AXI4-lite for the control path of the kernel, exactly fitting the integration style of TaPaSCo's processing elements. TaPaSCo furthermore allows design-space exploration to find optimal macroscopic configurations of the parallel kernels, a feature that Fletcher does not have. It also allows to target a wide variety of (mainly embedded-oriented, but some datacenter-grade) FPGA accelerator cards, although currently only those that contain Xilinx FPGAs.

Spatial [21] is mainly a domain-specific language embedded in Scala, tightly connected to the Chisel hardware description language [23]. The language provides a very high level of abstraction to design accelerators and targets not only various FPGA accelerator platforms (of both Intel and Xilinx), but also CGRA-like and ASIC targets. Aside from not being a language itself, Fletcher differs from Spatial in the sense that it is less generic, and focuses only on abstractions to easily and efficiently access tabular data structures described in Arrow.

OC-Accel [22], the successor of CAPI SNAP, does adhere to the criteria described, although it is still somewhat platform-specific, since it allows to target FPGA accelerator systems that have an OpenCAPI [24] enabled host system, typically found only in contemporary POWER systems. OC-Accel is a target for Fletcher, aside from AWS EC2 F1 and Xilinx Alveo cards. We conclude the comparison by mentioning that Fletcher is a more domain-specific solution that only works for the tabular data structures of Apache Arrow. This prevents Fletcher from being used in other domains, although the lessons learned are of value when creating similar frameworks for other domains.

3 Hardware Internals

In this section, we will briefly summarize the core hardware components of the Fletcher framework as presented in [3]. To understand them, we must first introduce the way Apache Arrow stores data in memory.

3.1 Arrow Columnar In-memory Format

An example of the Apache Arrow in-memory format and how it relates to a schema is shown in Fig. 4. Arrow tabular data is stored in an abstraction called a RecordBatch. A RecordBatch contains several columns for each field of a record, that are in Arrow called *Arrays* (not to be confused with C-like arrays). These arrays can hold all sorts of data types, from strings to lists of integers, to lists of lists of timestamps, and various others.

Arrays consist of several Arrow contiguous *buffers*, that are related, to store the data of a specific type. There are several types of buffers, such as *validity* buffers, *value* buffers and *offset* buffers. Validity buffers store a single bit to signify if a record (or deeper nested) element is valid or

Table 1 Overview of open-source FPGA accelerator development frameworks

Framework	Focus	Targets	Ref.
Fletcher	HLL software integration, tabular data	AWS EC2 F1, OC-Accel, Xilinx Alveo	[2]
TaPaSCo	Parallel kernels, DSE	AWS EC2 F1, Various Xilinx-centric	[20]
Spatial	HDL (eDSL), DSE	AWS EC2 F1, Various Xilinx-centric, Intel Arria 10, other non-FPGA targets	[21]
OC-Accel	OpenPOWER/OpenCAPI systems	Alphadata 9V3, 9H3, 9H7	[22]

a

Field A: Float32 (nullable)
Field B: UTF8 String
Field C: Struct(E: Int16, F: Float64)

b

A	B	C
0.5f	"fpga"	(42, 0.125)
0.25f	"fun"	(1337, 0.0)
∅	"!"	(13, 2.7)

c

Index	Buffers for:					
	Field A		Field B		Field C	
	Validity (bit)	Values (float)	Offsets (int32)	Values (char)	Values E (int16)	Values F (double)
0	1	0.5f	0	f	42	0.125
1	1	0.25f	4	p	1337	0.0
2	0	×	7	g	13	2.7
3			8	a		
4				f		
5				u		
6				n		
7				!		

Figure 4 An example Arrow schema (**a**) of an Arrow RecordBatch (**b**) and resulting Arrow buffers (**c**).

null (i.e. there is no data). Value buffers store actual values of fixed-width types, similar to C-like arrays. Offset buffers store offsets of variable length types, such as strings (which are lists of characters), where an offset at some index points to where a variable-length item starts in another buffer.

A RecordBatch contains specific meta-data called a *schema* that expresses the types of the fields in the records, therefore defining the types of the arrays, in turn defining which buffers are present. When a user wants to obtain (a subset of) a record from the RecordBatch, through the schema, we may find out what buffers to load data from to obtain the records of interest. Normally, an FPGA developer designs an accelerator that has to interface with a memory bus to get to the data set. That means the accelerator must typically request a bus word from a specific byte address. However, in the case of a tabular data set stored in the Arrow format, it is more convenient to express access to the data by supplying a table index, or a range of table indices, and receiving streams of the data of interest in the form of the types expressed through the schema, rather than as a bus word.

3.2 Vendor-Agnostic Streaming Library

To be able to access Arrow Arrays as such, we continue to describe Fletcher's hardware internal from the bottom up. In the design of these hardware components, we apply a streaming-oriented design methodology (using ready/valid handshaking mechanism) as much as possible. When these streams are Arrow data streams, we allow users to *scale* the throughput of these streams, by handshaking a user-specified number of elements per transfer. We call such streams multiple-element-per-handshake (MEPH) streams. To support this style of streaming-oriented design, we have developed a vendor-agnostic streaming library that contains the following components:

Slice — A component to break up any combinatorial paths in a stream, typically using registers.

FIFO — A component to buffer stream contents, typically using RAM.

Sync — A component to synchronize between an arbitrary number of input and output streams.

Barrel — A pipelined component to barrel rotate or shift MEPH streams at the element level.

Reshaper — A component that absorbs an arbitrary number of valid elements of an MEPH stream and outputs another arbitrary number of elements. This component is useful for serializing wide streams into narrow streams (or vice versa, parallelizing narrow streams into wide streams). The component can also be used to reduce elements per cycle in a single stream handshake or to increase (e.g. maximize) them. The implementation of the Reshaper uses the Barrel component.

Arbiter — A component to arbitrate multiple streams onto a single stream.

Buffer — An abstraction over a FIFO and a sync with a variable depth.

On top of the streaming components, a light-weight bus infrastructure has been developed to allow multiple masters to use the same memory interface. This bus infrastructure is similar to (and includes wrappers for) AXI4, supporting independent request and data channels, and bursts.

3.3 BufferReaders/Writers

We use the aforementioned streaming library to construct components matching the concepts of Apache Arrow. The smallest unit we need to access is an Arrow Buffer, which is basically a C-like array of primitive data. We therefore implement a component called a BufferReader (BR). The BR is a highly configurable component to support turning host memory bus burst requests and responses into streams of potentially fixed-width types. Based on an Arrow buffer address and a range of items to obtain from the buffer, the component performs the appropriate pointer arithmetic

to locate the elements of interest in the Arrow buffer. It then requests the data on the memory interface, and handles all responses, aligning and reshaping the bus words into MEPH streams with fixed-width data types corresponding to the Arrow primitive type contained in the buffer. An architectural overview of the implementation of two BRs (in combination providing a setup to read variable-length types) is shown in Fig. 5a.

The top-level of a buffer reader contains the following interfaces, that are all pipelined streams:

Command (in)	Used to request a range of items to be obtained from host memory by the BR. Also contains the Arrow buffer address and a special *tag*.
Unlock (out)	Used to signal the completion of a command, handshaking back the command's original *tag*.
Bus read request (out)	Used to request data from memory.
Bus read data (in)	Used to receive data words from memory.
Data (out)	An MEPH stream of data corresponding to an Arrow data type.

Offset buffers require the consumer of the data stream to turn an offset into a length. In this way, the consumer (typically the accelerator core logic) can know the size of a variable length item in a column. Therefore, for offset BRs, two consecutive offsets are subtracted to generate a length. Furthermore, BRs support the generation of an output command stream for a second BR. To generate this command stream, rather than generating a command for the child buffer for each variable length item, the BR requests both the last offset and the first offset in the range of the command first, before requesting all offsets in a large burst. The first and last offset can then be sent as a single command to the child BR, allowing it to request the data in the values buffer using large bursts.

Complementary to BRs, we also implemented Buffer Writers (BWs) that, given some index range can write to memory in the Arrow format. They have the same interfaces as BR, except the data flow is inverted, also shown in Fig. 5b. If the BW writes to an offsets buffer, it can be configured to generate offsets from a length input stream. This length input stream can optionally be used to generate commands for a child buffer. To achieve maximum throughput, the child command generation may be disabled, otherwise the child buffer writer will generate padding after the ending of every list in an Arrow Array containing variable length types.

3.4 Arrays

To support Arrow *Arrays*, that combine multiple Buffer Readers/Writers to deliver any field type that may be found in an Arrow schema, we implement specialized components called Array Readers and Array Writers. They furthermore support, attaching command outputs of offsets buffers to values or validity bitmap buffers, arbitration of multiple buffer memory interface masters onto a single slave, synchronization of unlock streams of all buffers in use, and finally, recursive instantiation. The recursive instantiation allows support for nested types, such as `Lists<List<(Type)>>`, adding an Arrow validity bit to the output stream, and support for Arrow *structs*, such as `Struct<Int16, Float64>` as shown in the example.

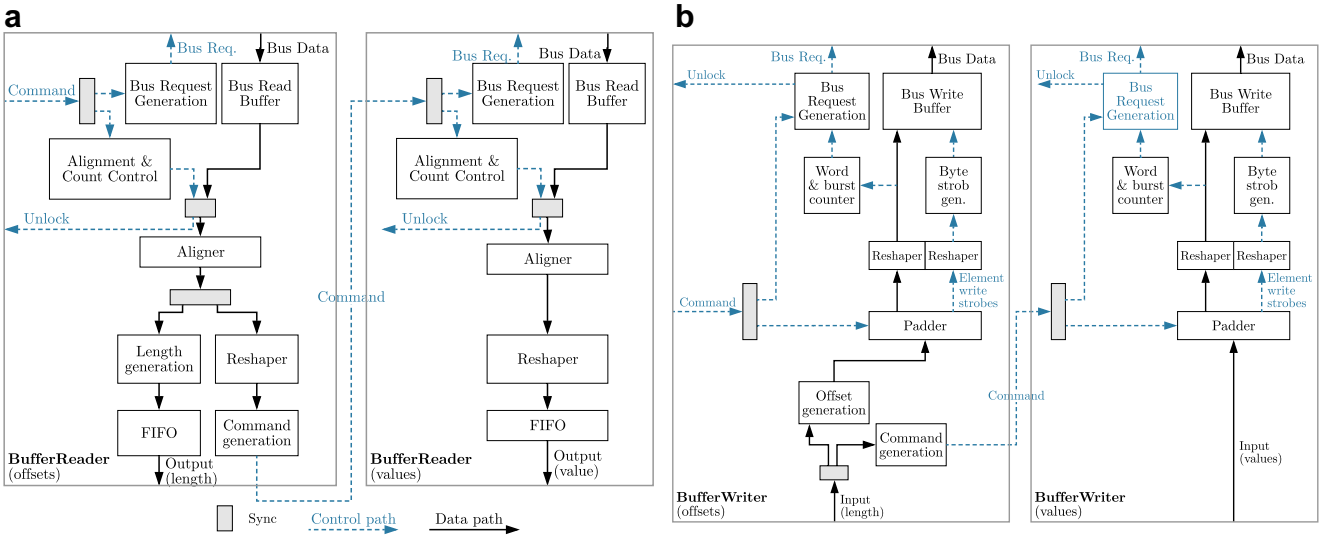

Figure 5 The Bufferreader/Writer components of Fletcher.

The Array Readers and Writers are supplied with a configuration string that conveys the same information as an Arrow field type. By parsing the configuration string, the components are recursively instantiated according to the top level type of the field in a schema. An example for the schema from Fig. 4 is shown in Fig. 6.

Reading from the example RecordBatch (corresponding to the schema) will require three Array Readers. The manner in which they are recursively instantiated is shown in the figure. Here one can discern four types of Array Reader configurations:

Default A default Array Reader only instantiates a specific Array Reader of the top-level type of the corresponding schema field, but provides a bus arbiter to share the memory interface amongst all BRs that are instantiated in all child Array Readers.

Prim An Array Reader instantiating a BR for fixed-width (primitive) types.

Null Used to add a validity (non-null) bitmap buffer and synchronize with the output streams of a child Array Reader to append the validity bit.

List Used to add an offsets buffer that generates a length stream and provides a first and last index for the command stream of a child Array Reader.

Struct Used to instantiate multiple Array Readers, synchronizing their output streams to couple the delivery of separate fields inside a struct into a single stream.

The complement (in terms of data flow) of Array Readers are also implemented as Array Writer. One additional challenge to Array Writers is that they require dynamically resizable Arrow Buffers in host memory, because it cannot always be assumed that the size of the resulting Arrow Buffers is known at the start of some input stream. This is an interesting challenge for future work.

4 Fletcher Toolchain

4.1 Generic Fletcher High-Level Architecture

We have so far described how Array Readers and Array Writers are generated, still using VHDL only. Although the components can be rather complex in nature, already, they allow to merely access a single Arrow Array in a RecordBatch; one column in the tabular data structure. However, many applications require access to *multiple* columns, as well as *multiple* RecordBatches. Furthermore, accelerator kernels require a control path from the host software as well.

With these requirements, a generic architecture of a Fletcher-based accelerator design is presented in Fig. 7. In this figure, the Array Readers/Writers (hereafter ArrayR/Ws) as described in the previous section are shown. We continue to explain the new components shown in the figure.

– **RecordBatch Reader/Writer** RecordBatch Readers and Writers (hereafter RecordBatchR/Ws) are com-

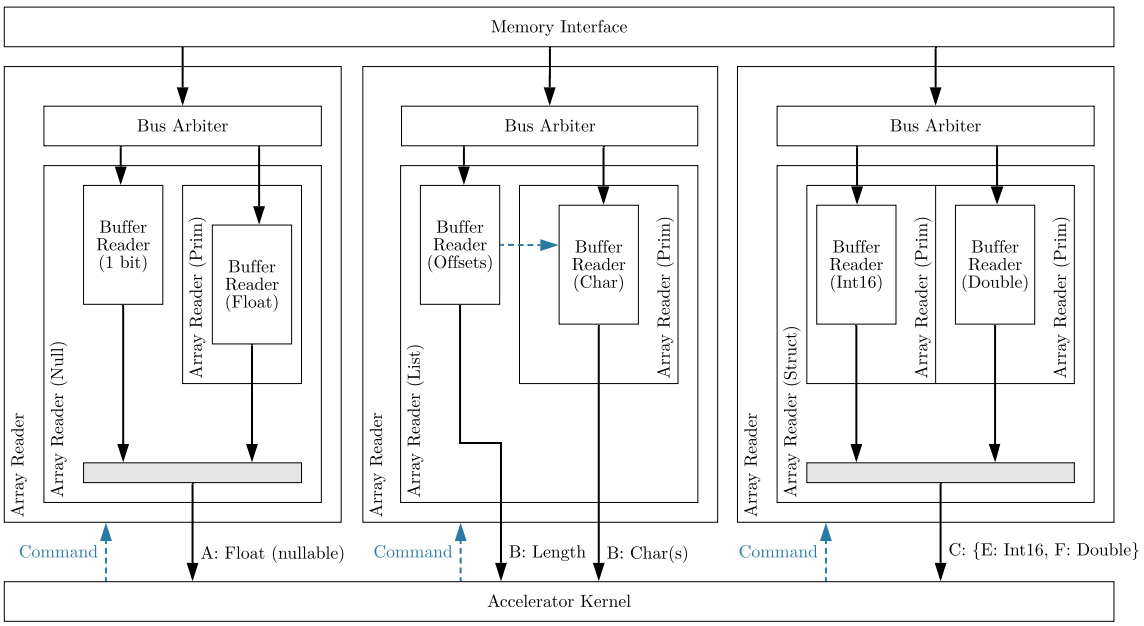

Figure 6 Three resulting Array Reader configurations for each field of the Schema in Fig. 4.

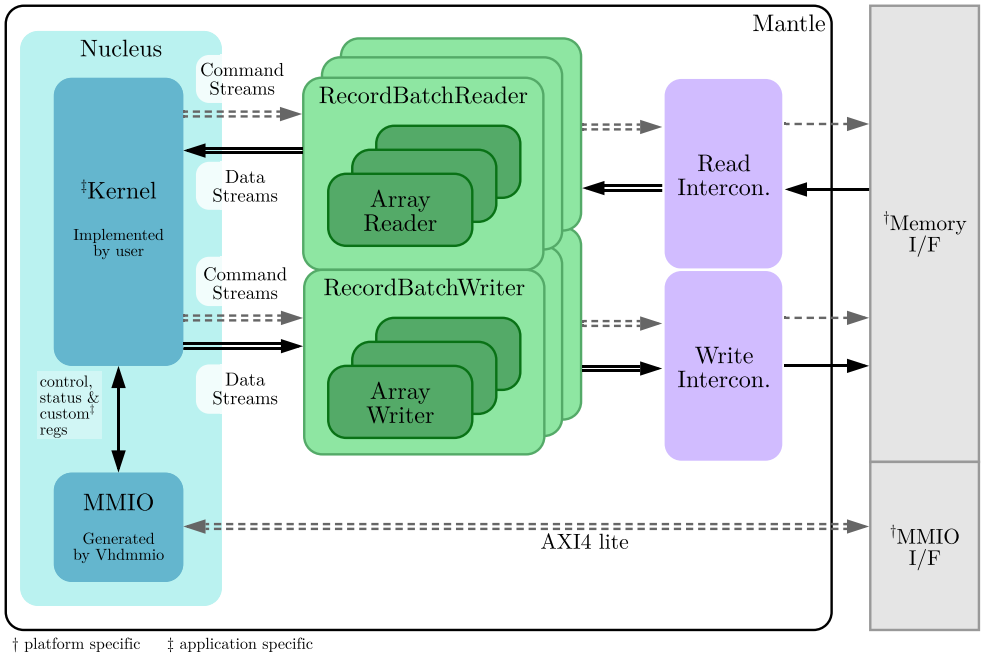

Figure 7 Generic top-level architecture of Fletcher accelerator designs.

ponents that wrap around multiple ArrayR/Ws of a single RecordBatch. It may seem that the level of hierarchy that the RecordBatchR/Ws introduce does not necessarily have to exist, since the ArrayR/Ws can be operated independently of each other. However, a user may not want to issue a separate command to each ArrayR/W, but rather a single command to all ArrayR/Ws in a RecordBatch. The RecordBatchR/W allows to duplicate a single command stream into multiple command streams for each ArrayR/W, and allows to merge command responses into a single response stream as well. This ultimately adds support to access multiple columns.

– **Read/Write interconnect** The Read/Write Interconnect components manage all memory interfaces coming from the ArrayR/Ws. Since the memory interfaces of ArrayR/Ws may have various configurations, but the top-level memory interface typically only supports one configuration, serializers and parallelizers will automatically be inserted here. Furthermore, round-robin arbiters and buffers are instantiated in this component.
– **Nucleus** The Nucleus component directly interfaces with the Arrow data streams, the command streams to the RecordBatchR/W, and with an AXI4-lite bus for memory-mapped I/O. Users may choose to implement their kernel at this level of abstraction, requiring them to insert their own MMIO controllers and fully manage the information on the command streams to the RecordBatchR/Ws themselves, including the addresses of the Arrow buffers in the memory. However, the philosophy of the Fletcher framework is to allow developers to express access to their data in terms of row indices, not having to worry about pointers (and pointer arithmetic). Therefore, by default, the Nucleus level abstracts the command streams of the RecordBatchR/Ws in such a way that the Arrow buffer addresses are hidden. To do so, it instantiates an MMIO controller that is used to pass information about buffer addresses from the host to the Nucleus. The MMIO controller is furthermore used to pass metadata about the specified RecordBatches and run-time information about the workload, such as the number of rows that a RecordBatch has, and a range of row indices for the kernel to operate on. Users may also pass or return application-specific information through these registers from/to the host machine.
– **Mantle** The Mantle component wraps around all other components, resulting in a top-level design that always has the same interface. In this way, supporting Fletcher on a new FPGA acceleration platforms is a matter of integrating the Mantle with the existing subsystems. Any generated Fletcher design can from that point onward be mapped onto that platform.

4.2 Fletcher Tool Chain

Through specialization of the generic architecture shown in Fig. 7, based on Arrow schemas, Fletcher faces challenge H3. However, to automate the specialization itself is a challenge on its own. Because of the large number of variations of designs that may be generated to accommodate multiple Arrow Arrays, multiple RecordBatches and the

control path thereof, it is infeasible to implement a generic version of the design shown in Fig. 7 in HDLs that vendor tools support.

To provide an agile and open-source hardware development experience to the users of Fletcher, a tool is required that is able to generate application-specific flavors of the generic architecture. It must furthermore be able to generate a platform-agnostic simulation environment, such that kernel implementations can be functionally verified independent of the target platform.

We therefore develop three new tools:

- Cerata; a generic hardware construction library providing high-level abstractions for structural hardware design.
- Vhdmmio: a generic MMIO controller generation tool taking a simple description of a register map, outputting VHDL sources with MMIO controller components that can be connected to an AXI4-lite bus.
- Fletchgen: an Arrow-specific tool built on top of Cerata, using the abstractions provided to describe the generic architecture as shown in Fig. 7, including the RecordBatchR/Ws, the Nucleus, the interconnect infrastructure and the Mantle. It furthermore uses Vhdmmio for the control path from the host system through memory-mapped I/O.

These tools are part of the Fletcher hardware generation toolchain, which we will continue to explain in more detail. A high-level overview of the toolchain is shown in Fig. 8.

4.2.1 Cerata

Cerata is an open source hardware construction library written in modern C++17. It is intended to be used only for structural hardware design, providing many abstractions for structural hardware generation. Structural designs can be described as a graph by connecting nodes representing ports, signals, parameters, literals and expressions. The graphs are hierarchical, such that they represent either components or instances. Cerata allows the expression of advanced interface types, supporting in particular nested streams that often emerge when converting nested Arrow data types into a form suitable for hardware. These can be connected with single lines of code, similar to how Chisel and SystemVerilog allow bulk connections.

Like Chisel that is hosted in Scala, Cerata allows already generated designs to be inspected programmatically through its host language C++, resulting in what could be viewed as introspection. For example, it is possible to describe a component X, and during generation of another component Y that uses X, to inspect what ports X has in order to generate some structure that properly supports

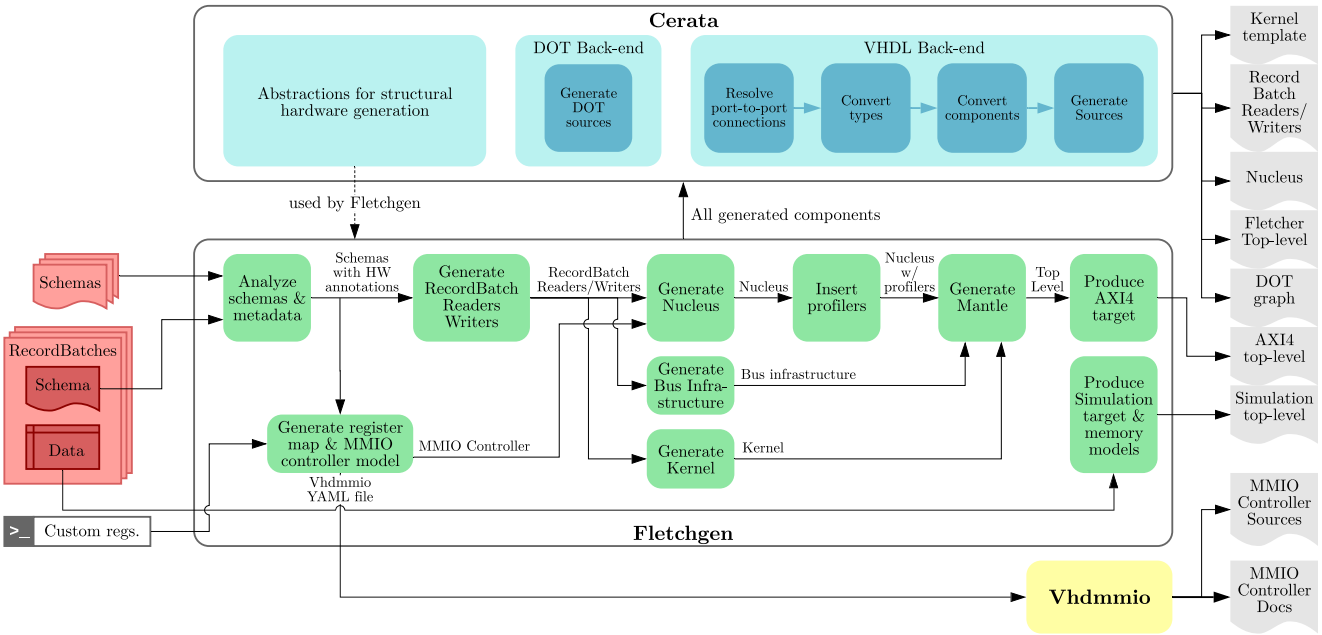

Figure 8 Overview of hardware generation components in the Fletcher tool-chain. The goal of these components is to automatically generate hardware interfaces to Arrow RecordBatches (tabular data structures). The user prepares the *schemas* (descriptions of the column types in tabular data structures) and RecordBatches (schema plus actual data for simulation) (in red). In the center, the general flow of the automatic hardware interface generator tool Fletchgen is shown. It produces graph-like descriptions of the hardware structure, and passes it to the Cerata library that generates structural VHDL from the graphs. All outputs on the right are automatically generated by the tool. The user only needs to implement the kernel template.

the instantiation of X. Note that this allows for a bottom-up generation approach, which is very impractical in any traditional hardware description languages like VHDL[1]. There, all information has to be known at the top-level, trickling down to the lower levels of the hierarchy. It also allows for specific transformations to be implemented for graphs, one of which is to insert stream profilers, as we will discuss later. Since the rest of the Fletcher tool-chain is written in VHDL and C++, we have not used Chisel for this capability, since it outputs Verilog and is hosted in Scala.

After constructing and transforming graphs according to the needs of the user, Cerata can target two back-ends, a DOT [25] back-end to visualize the constructed graphs, and a VHDL back-end to generate structural VHDL.

4.2.2 Fletchgen

The input of Fletchgen are Arrow schemas and RecordBatches (that contain their schema plus data). All schemas are first checked for *required* metadata that is Fletcher-specific, and optional metadata. An overview of all *schema-level* metadata that Fletchgen understands is as follows:

- A schema name (required), used in the generation of HDL sources.
- A schema access mode (required), specifying whether a user would like to read from a RecordBatch, or write to a RecordBatch.
- Memory interface specification (optional). This defines the properties of the bus infrastructure at the memory side of the interface. Properties include (amongst others) data width, address width, and maximum burst length.

Additionally, schema *fields* can be annotated with the following metadata attributes:

- Ignore (true/false); a user not interested in a specific column of the RecordBatch can choose to ignore it. No hardware support or interface will be generated for this column. Note that this is an advantage of a columnar data storage system. Columns can be accessed completely independently of other columns.
- Elements per cycle; the maximum number of elements that can be handshaked in a single cycle on the output stream of this field.
- Length elements per cycle; the maximum number of list lengths that can be handshaked in a single cycle on the length stream of this field.
- Profile (true/false); a user may choose to insert stream profilers - units that gather statistics about the handshaking mechanism of (multi-element-per-cycle) streams that can be translated into throughput. This helps users make performance/area trade-offs.
- Tag width; the number of bits used to identify commands and command responses.

After analysis of the metadata and after verification that specific properties (such as schema names) do not cause any conflicts, the hardware may be structurally described. Fletchgen generates the design from the bottom-up, starting with the instantiation of all required ArrayR/Ws inside their corresponding RecordBatchR/Ws. In this step, the configuration string for the ArrayR/Ws is derived from the Arrow schema, using API calls provided by the Arrow library itself to traverse the tree of potentially nested field types.

The ArrayR/Ws are considered to be 'primitive' components as far as Fletchgen is concerned, i.e. they do not consist of other components that have to be generated (although their implementation is described with a very generative style of VHDL). ArrayR/Ws are described with VHDL, but this language does not allow port names to be generated. The data and control streaming interfaces therefore have nondescript names that are not easy to recognize for kernel developers.

The ports would preferably be named after the Arrow schema fields such that they are easy to recognize for kernel developers. Furthermore, because ArrowR/Ws can cause a variable number of streams to appear, these streams are concatenated with port vectors for each type of stream, while it is more pleasing to get separate interface ports for every stream related to a specific Arrow field. We have therefore equipped Cerata with abstractions to concatenate multiple streams onto single ports and vice versa. These abstractions are used to eventually generate streaming RecordBatchR/W interfaces that have names corresponding to what Arrow field they were derived from, such that they become easily recognizable by users.

Additionally derived from the schema and its metadata is the memory-mapped I/O register map. Furthermore, users may supply additional arguments to reserve more custom registers in the register map through the command-line interface of the Fletchgen tool. All registers are 32-bits, controlled over an AXI4-lite interface from the host side. Four categories of registers are mapped; default registers, schema-derived registers, custom registers, and profiling registers, as follows:

- **Default registers**; control, status, and two return value registers for results up to 64-bits wide. Since most of the target platforms have 64-bit addresses, this allows to pass a pointer to some resulting data structure, or a primitive return value. Resulting data structures laid out in the Arrow format would typically be passed to

[1] Or is arguably incredibly esoteric in slightly more modern languages like SystemVerilog by writing low-level C support functions against the Verilog Procedural Interface.

Fletchgen as a separate schema with access mode set to write.

- **Schema-derived registers**; the range of operation on each RecordBatch (first and last row index), followed by all Arrow buffer addresses, which we will call *RecordBatch metadata*. Because these registers are automatically set by the Fletcher run-time library, it is imperative that there is a unique order to the metadata, that is consistent between the hardware implementation and the software run-time. This is done by first sorting all schemas by name, and then stable sorting them by access mode. Since schema names must be unique for each access mode, the resulting unique ordering will make sure the hardware implementation corresponds to how the run-time library will set all metadata.
- **Custom registers**; the set of registers supplied by the user, for whatever purpose.
- **Profiling registers**; the registers that contain the results of profiling Arrow data streams. These include a control register to start and stop profiling, as well as six measurement results; the number of elements transferred on the stream, the number of cycles the stream valid signal was asserted, the number of cycles the stream ready signal was asserted, the number of cycles both were asserted, the number of 'last' signals handshaked (to count the number of stream packets transferred on variable-length types such as strings), and the number of cycles the profiler was enabled.

After the whole register map is known, Fletchgen generates a human-readable YAML-file that is passed to the Vhdmmio tool. This tool then generates an implementation of an MMIO controller according to the register map described above, and outputs user-friendly documentation about the register map. Since the implementation of the MMIO controller is generated by this external tool, inside Fletchgen, it is considered to be a primitive component. Only a model of its interface is constructed, which is passed onto the next generation step.

The generated RecordBatchR/Ws and the MMIO controller now contain all the information necessary to generate three more components. First, the memory bus infrastructure that has to connect to the memory interface side of the RecordBatchR/Ws. Second, the Nucleus, that forwards the Arrow data and control streams to the third component; the Kernel. Note that the Kernel component is not implemented by Fletchgen, but must be implemented by the user.

The Nucleus is at first constructed without taking the stream profiler components into account, since inserting stream profilers is one of the *transformation* functions available in Cerata. As illustrated in Fig. 9, after tagging streams with the profiling option, the profiling transformation function may be called by supplying the component implementation to transform. Optionally, references to signals where the stream profiler measurement outputs need to be connected can be given. When they are not given, the transformation will extend the component interface to contain the profiler measurement output signals. In the case of Fletchgen, we tag the streams between the Nucleus external interface and the Kernel corresponding to the field metadata supplied through the Arrow schema, and we supply the MMIO controller's profiling registers as output signals.

Now, a Nucleus instantiating the MMIO controller and the Kernel component with profiling registers is constructed. Together with the bus infrastructure that was generated, everything is tied together to form the Mantle — the Fletcher generic top-level.

Through the use of the Cerata hardware construction library, we have now implemented everything as shown in Fig. 7, apart from the Kernel, which is left to the user. The design achieves the goal of the Fletcher framework; providing the user with hardware interfaces that correspond to the abstractions of Apache Arrow. They may now access RecordBatches through a command stream by only

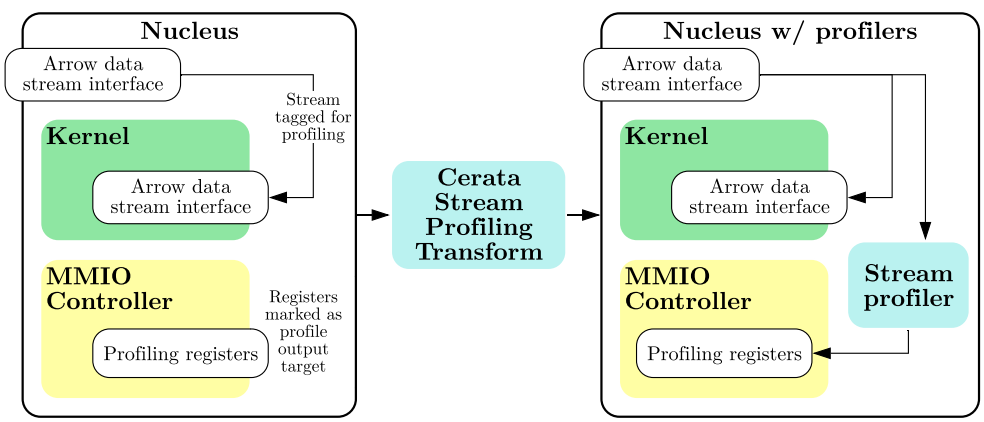

Figure 9 Profiling transformation applied to the Nucleus.

supplying row indices, and will read or write data over streams that correspond with Arrow's types.

4.2.3 Run-Time Integration

We continue to describe how Fletcher is integrated during run-time, where challenge H1 related to portability must also be solved. An overview of the approach is seen in Fig. 10, where two of the supported platforms, AWS EC2 F1 and OC-Accel are shown. Because the top-level component, the Mantle, has the same interface for any Fletcher design, supporting multiple FPGA accelerator platforms is done creating platform-specific wrappers for the Mantle that are maintained in a separate open-source repository to prevent platform-specific code from contaminating the Fletcher code base. The platform-specific low-level drivers to interact with the accelerator framework are abstracted, first through a low-level library in C, providing a common API for all platforms to the higher-level Fletcher platform-agnostic run-time libraries that are intended for users. Fletcher run-time library dynamically searches and loads platform-specific versions of its low-level drivers, depending on what platform is available.

The currently supported languages include C++ and Python. The language-specific Fletcher libraries contain an API leaning heavily on Apache Arrow's abstractions. An example of how the accelerator is operated from Python is found in Fig. 11.

During run-time, users only have to provide references to the RecordBatches of interest. The users only need to manually start the kernel and write and read values to their custom registers. These are placed into a queue, and automatically made available to the FPGA accelerator.

4.2.4 Simulation

To support users of Fletchgen with functional verification through simulation, note that in Fig. 8, users may also supply Arrow RecordBatches. The Arrow schema that is contained within the RecordBatch will be handled like any other schema, except the data in the RecordBatch will be used to produce a simulation top-level, wrapping the Mantle, and instantiating simulation-only memories that contain the RecordBatch data. The simulation top-level sets the buffer addresses and RecordBatch metadata automatically through the MMIO interface. It continues to send the start signal to the kernel, such that when the user is ready to debug the kernel, all data and control signals flowing in from the upper layers of the hierarchy are already handled. Only the custom registers are to be set appropriately by the user in the simulation top-level.

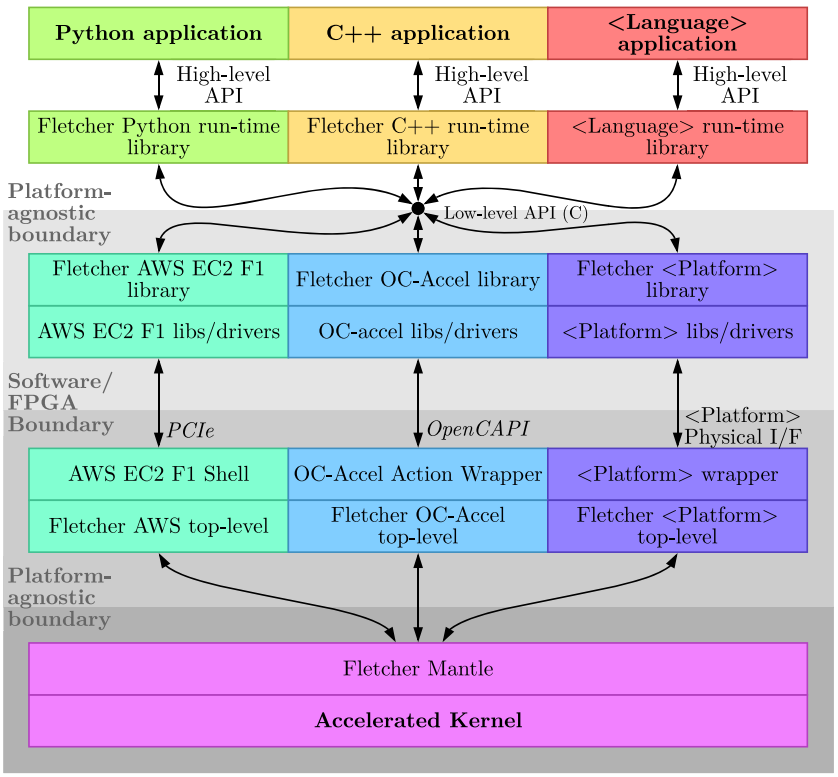

Figure 10 Platform-agnostic run-time stack.

Figure 11 Example of using the Python run-time library to control the accelerator.

```python
import pyfletcher as pf

# Set up an auto-detected platform.
platform = pf.Platform()
platform.init()
# Create a Context for the data
context = pf.Context(platform)
# Queue input and output RecordBatches.
context.queue_record_batch(in_batch, out_batch)
# Enable the Context, causing data transfer
#   and MMIO registers to be set appropriately
context.enable()
# Set up an interface to the Kernel,
#   supplying the Context.
kernel = pf.Kernel(context)
# Start the kernel.
kernel.start()
# Wait for the kernel to finish.
kernel.wait_for_finish()
```

5 Examples

To provide an example of the functionality described in the previous section, consider the following example application. Suppose we have two tables, where one table called `people` contains a unique key, names, ages and favorite food. The last item refers to a second table named `foods`, containing a unique key and food names. Suppose dinner must be cooked for all children based on their favorite food, we may query the tables for the names of all people under 12, and look up their favorite food.

We stipulate that in the upcoming example, we intend to reduce the design time of the FPGA accelerator infrastructure required to load and store the resulting RecordBatches, since this all we can do based on Arrow schemas. The actual implementation of the function of the computational kernel cannot be derived from a schema. As such, this is still left to the users to implement however they see fit, e.g. by using traditional HDL or by using high-level synthesis techniques to perhaps reduce that part of the design time.

When using the Fletcher framework to set up an accelerator implementation to solve this problem, we first have to define the Arrow schemas that describe the types of data each table will hold. An example of how this is done in Python is shown in Fig. 12a. Note that one could use any language supported by Apache Arrow libraries to produce the schema, but for this article we choose Python because it is relatively succinct. The 30 lines of Python code hold enough information to produce Arrow schemas for our example. They can be passed to Fletchgen to generate a customized, application-specific version of the architecture presented in Fig. 7.

On lines 3-15, the developer does not only define a Schema for the 'foods' table, but also specifies some data it contains, and places the data inside an Arrow RecordBatch.

a

```python
import pyarrow as pa

# RecordBatch describing the 'foods' input table:
foods = pa.RecordBatch.from_arrays(
    # RecordBatch data for simulation:
    [pa.array([10, 31, 32, 70], pa.uint16()),
     pa.array(['apple', 'pear', 'banana', 'melon'])],
    # RecordBatch schema:
    schema=pa.schema([
        pa.field('id', pa.uint16()),
        pa.field('name', pa.string())
          .with_metadata({'fletcher_profile': 'true'}),
    ]).with_metadata({'fletcher_mode': 'read',
                      'fletcher_name': 'foods'})
)
# Schema describing the 'people' input table:
people = pa.schema([
    pa.field('id', pa.uint32())
      .with_metadata({'fletcher_ignore': 'true'}),
    pa.field('name', pa.string()),
    pa.field('age', pa.uint8()),
    pa.field('food_id', pa.uint16()),
]).with_metadata({'fletcher_mode': 'read',
                  'fletcher_name': 'people'})
# Schema describing the 'dinner' output table:
dinner = pa.schema([
    pa.field('name', pa.string()),
    pa.field('food', pa.string()),
]).with_metadata({'fletcher_mode': 'write',
                  'fletcher_name': 'dinner'})
```

b

```
entity Kernel is
  -- .. (generics omitted for brevity)
  port (
    -- The command stream to access the 'id'
    -- column of the 'foods' table.
    foods_id_cmd_valid     : out std_logic;
    foods_id_cmd_ready     : in  std_logic;
    foods_id_cmd_firstIdx  : out std_logic_v..
    foods_id_cmd_lastIdx   : out std_logic_v..
    -- The incoming Arrow data stream for the
    -- 'id' column of the 'foods' table.
    foods_id_valid         : in  std_logic;
    foods_id_ready         : out std_logic;
    foods_id_last          : in  std_logic;
    foods_id               : in  std_logic_v..
    -- .. (other streams omitted for brevity)
    -- Kernel control signals:
    start, stop, reset     : in  std_logic;
    idle, busy, done       : out std_logic;
    -- Generic result signal going to the MMIO
    -- controller:
    result                 : out std_logic_v..
    -- RecordBatch metadata coming from the
    -- MMIO controller:
    foods_firstidx         : in  std_logic_v..
    foods_lastidx          : in  std_logic_v..
    -- .. (other metadata omitted for brevity)
    -- Custom MMIO register input
    age_threshold          : in  std_logic_v..
  );
end entity;
```

Figure 12 Examples of input and output of Fletchgen.

As explained in the previous section, the data can be used to generate simulation models. The developer also supplies metadata on the `names` field, effectively tagging the resulting hardware streams to be profiled. Finally, the mandatory metadata are added; the access mode of the schema, in this case set to *read* from it, and the name of the schema to generate appropriate component and interface names.

On lines 16-24, the developer defines a schema, but since the question does not involve returning the unique key of the people, merely their name, we have no use for this field. By supplying Fletcher-specific metadata, we may ignore this field, and no hardware will be generated to access this column.

Finally, on lines 25-30, the output schema is defined, with the access mode set to be able to *write* to the RecordBatch. For brevity, we have omitted 5 more lines involving saving the schema and RecordBatch to a file.

After providing Fletchgen with these schemas, we obtain many files that encompass the whole design as described in the previous section, corresponding to Fig. 7, but specialized for the supplied schema. The only thing that the hardware developer has to do now is implement the kernel, for which a template was generated. For our example, the template is shown in Fig. 12b. Note that we have compacted the template for reasons of brevity, leaving out several rather detailed signals, and only show the code related to the `foods` table's `id` field. The interfaces provided by the template allow the hardware developer to reason about the tabular data structures they are working with in terms of row indices, easing the development process.

In Fig. 13a and b, we find the graphical representation of the design that was generated from the schemas in Fig. 12a. This is the specialized version of the generic architecture presented in Fig. 7.

To demonstrate the method of operation, suppose the kernel implementation requires all food `names` from the table. The user would first start the host-side application, which could be written in any of the software languages that Apache Arrow and Fletcher support. If we assume the language is Python, the host-side application for this example will look like Fig. 11, although line 9 would be replaced with the variable names of the RecordBatches of this example. Following Fig. 11, the platform is first initialized on line 5 (note that Fletcher auto-detects the vendor-specific platform, hence host-side software is portable across supported platforms). Then, on line 7, a context is created on this platform for this specific application. We add RecordBatches to the context by queuing them on line 9. We then enable the context, meaning the FPGA accelerator will be able to access the supplied RecordBatches. For some platforms (e.g. the supported AWS EC2 platform), this means any RecordBatches that are read will be transferred to accelerator on-board memory. For other platforms, the accelerator can read directly from the host memory (e.g., the supported OC-Accel platform). We then continue to construct the kernel abstraction with the enabled context on line 15. On line 17, the kernel is started, and on line 19, we wait for it to finish. Metadata, e.g. Arrow buffer addresses and RecordBatch dimensions, as well as control, e.g. starting the kernel and polling for completion, is abstracted and made vendor-agnostic by the Fletcher runtime system depicted in Fig. 10. In hardware, metadata and control information travels over the AXI4 lite MMIO interface as shown in Fig. 7.

Focusing on the example of obtaining all food names from the table, after all metadata and control information is passed, the kernel can use this information to start operating and issue a command to obtain all data from the relevant column. Fletchgen has generated a streaming interface appropriate for string data found in this column, using two streams; one for lengths, and another for the characters.

We show the simulation waveforms of the access mechanism in Fig. 14. Note that for brevity we have left out signals of the kernel component that are unrelated to the discussion, and have highlighted the main points of interest in the figures. The RecordBatch metadata is automatically supplied through the MMIO controller before the kernel is given the start signal (A). The kernel can use the `foods_lasttidx` input to know the total size of the RecordBatch, to prevent reading out of bounds. However, if the developer wishes to parallelize the kernel, it is possible to also supply a `foods_firstidx`, such that each instance of this kernel can operate on its own part of the input tables and output tables. The kernel may send a command over the `foods_name_cmd` stream (B) to request the Arrow data, in this case all entries from the `name` column. Arrow data will start flowing into the kernel through the `foods_name` stream, that supplies string lengths (C), followed by the characters on `foods_name_chars` (D). Note that the first two food names appear on the character stream.

Starting off with interfaces that make sense w.r.t. the data structures the developer has to access contrasts heavily with the normal HDL-flow experience, where a developer typically starts off with a byte-addressable memory interface and a memory-mapped I/O interface. This demonstrates the Fletcher's ability to face challenge H2 as described in Section 2.

We demonstrate the ability to fine-tune the generated interface by making simple modifications to the Arrow schema. As we can see from Fig. 14, the throughput of the character stream is relatively low, since only one character can be handshaked per cycle. Fortunately, the developer has annotated the field, as shown in Fig. 12a,

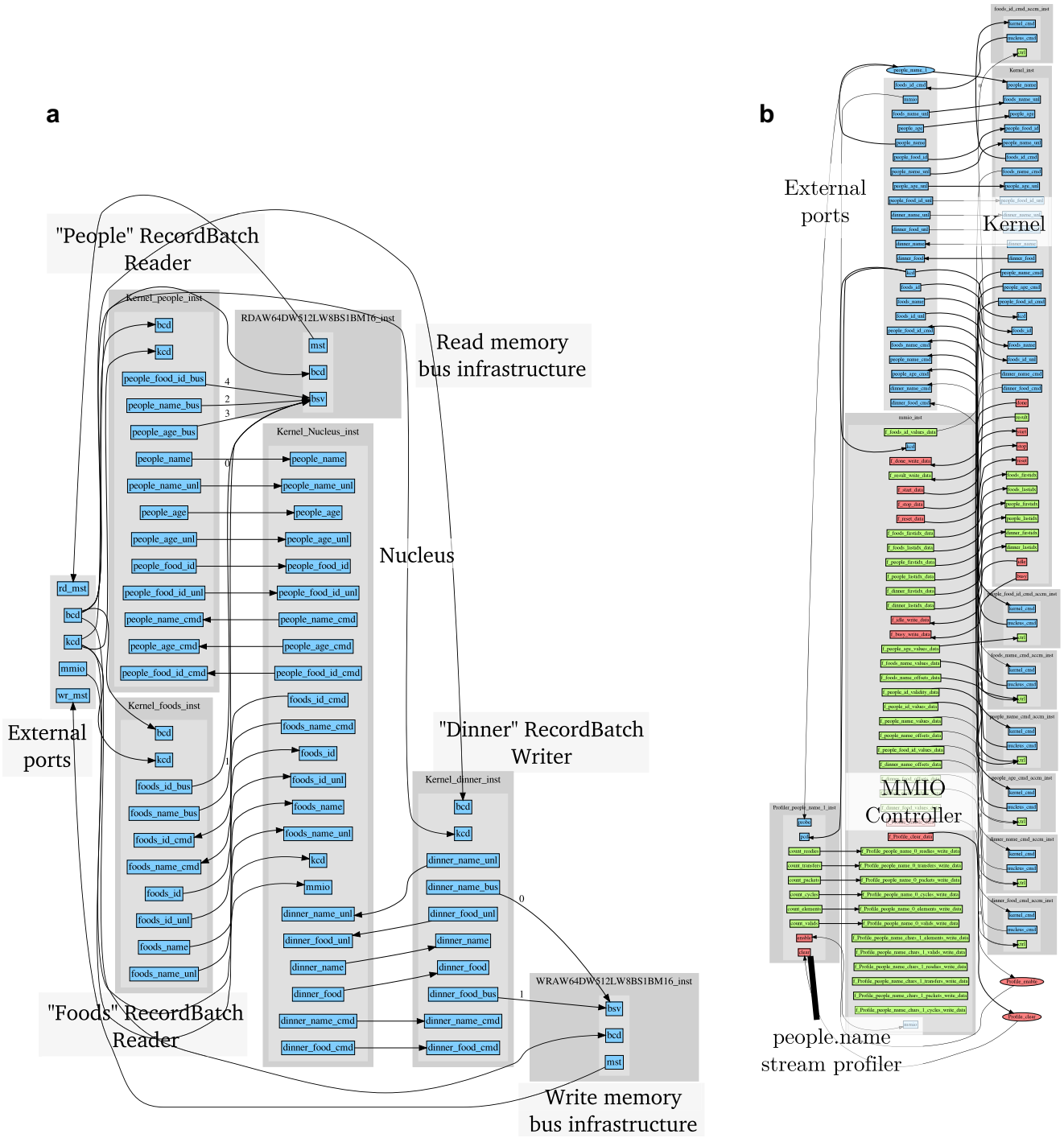

Figure 13 Graphical representation of the generated design resulting from the example schema in Cerata. This is a specialization of Fig. 7, tailored to implement the interface required for the Arrow Schema as defined in Fig. 12a.

with the stream profiling option. After running simulation or the real implementation of this kernel, the developer may study the stream profile to find that the character stream provides a bottleneck to the whole system. In that case, the developer may simply annotate the Arrow field with the previously described option to provide multiple elements per handshake. Regenerating the design and making slight modifications to only the kernel will cause a `count` field to appear on the stream, as shown in Fig. 15. The stream now allows to handshake four elements per transfer, with the count field indicating how many are valid. Note that the same amount of food names are handshaked as in

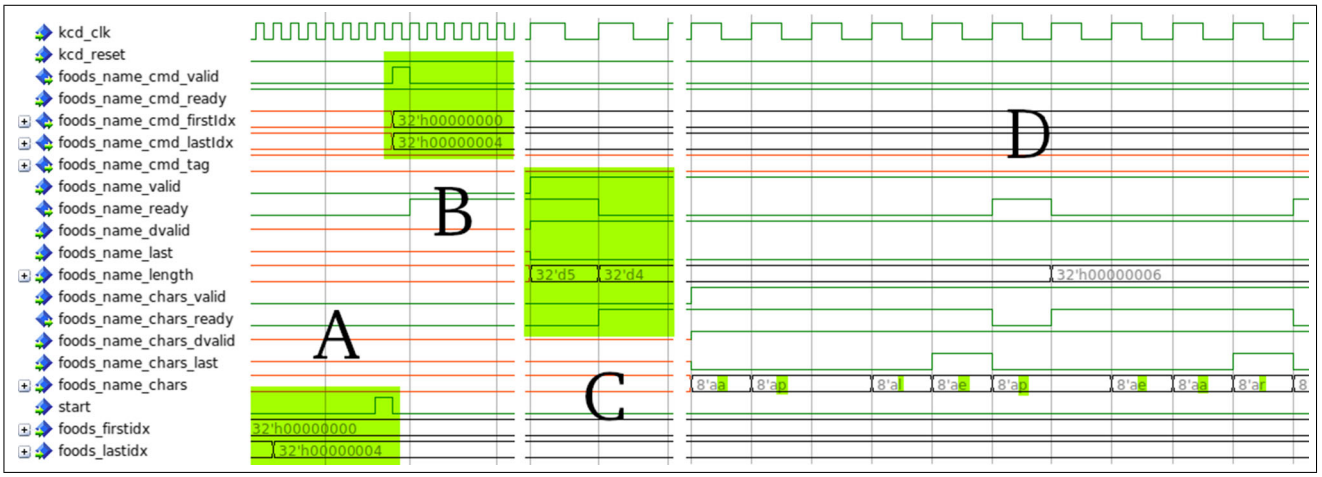

Figure 14 Streaming interface example, accessing the `foods name` field.

Fig. 14, although rather than taking ten cycles, they are now handshaked over three cycles, increasing the throughput of the stream at the cost of additional wires and control logic to support the wider interface.

Related to development effort, it is hard to properly quantify the reduction in development effort because of a large human dimension to such a measurement. To give a slight indication of the development effort saved, we could still look at the lines of code that were generated. From the 30 lines of Python seen in Fig. 12a, Fletchgen and Vhdmmio generate 6304 lines of VHDL (not counting blank or comment lines), that are arguably human-readable and modifiable. This excludes the components that, as far as Cerata is concerned, are 'primitive'. The support library of hardware primitives and ArrayR/W's amounts to approximately 30K lines of code. To test a more extensive design, we have also captured all table schemas of the TPC-H benchmark suite in 120 lines of Python, resulting in Fletchgen to generate 33K lines of code necessary to provide access to all its tables.

Finally, for throughput and area measurements, we refer the reader to prior work [3]. In this paper, it was shown that as the number of requested rows from a RecordBatch grows (as is typical in big data applications), a simulated ideal memory interface would be utilized between 95-100% for the read paths (from memory to FPGA) and close to 95% for the write paths (from FPGA to memory). In real applications presented in [2], designs are shown that are able to saturate the available system bandwidth through Fletcher generated interfaces. More applications of Fletcher

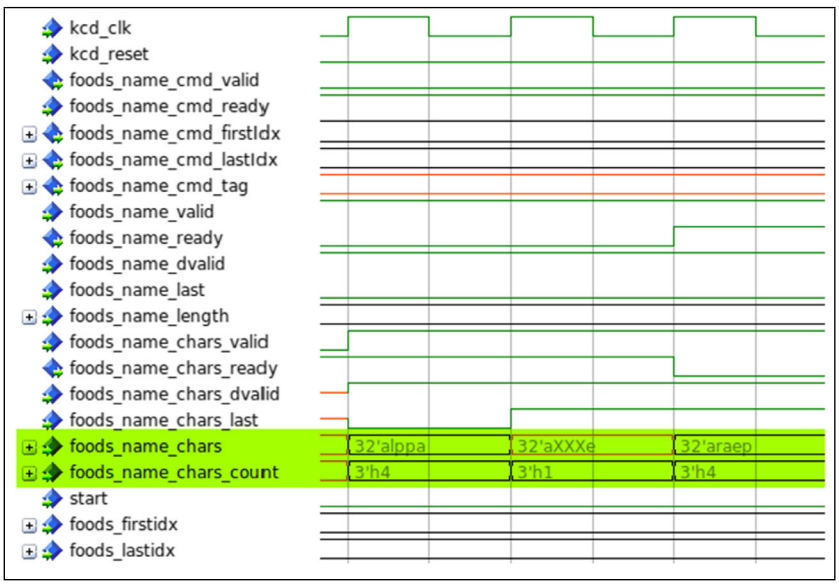

Figure 15 Accessing the `foods name` field with MEPH.

are found in [26, 27], where the focus is on the end-to-end application performance rather than the generated interfaces themselves.

6 Discussion

Now that we have discussed the methods used in the Fletcher framework, and given examples of how users interact with it, we finally discuss the lessons learned during the development of the framework and give an outlook on future work.

6.1 Lessons Learned

We have learned that the interfaces to dynamically sized and deeply nested data structures can become relatively complex. An interesting direction of research would be to investigate methods to standardize streaming hardware interfaces for complex data structures. Although Fletcher specifies a streaming interface for tabular data sets, it only touches the tip of an iceberg, since it only deals with Arrow data sets, but not other sorts of structures, such as graphs (on which studies have already started in [19]). Developers still have to spend a lot of time designing interfaces *between* hardware components, not necessarily communicating through a byte-addressable memory, for specific data structures as no standardized 'container' types with associated access behavior as known from software standard libraries exist.

We hypothesize that the level of abstraction of modern software languages evolves faster than high-level synthesis tools can catch up with. This causes an increasing gap in the skill set between at least a part of the target audience of the tool and the tool itself, especially in the domain of big data analytics. Focus should be given to the development of new hardware description languages and techniques that provide high-level, meaningful abstractions for decades of digital circuit design and computer architecture knowledge, such that the hardware development experience can be made as agile as that of the software development experience without the loss of expressiveness and performance. Embedded domain-specific languages provide promising alternatives, e.g. [21, 23] on the near-term, while valiant efforts to support designing new HDLs without being embedded in another language have very recently been presented, e.g. in [28].

Finally, researchers working on FPGA accelerators in big data analytics should take care to not only measure kernel or accelerator on-board computational performance, but should consider the implications of end-to-end integration into the intended big data analytics pipeline as well. If the overhead associated with the data path to the accelerator is too large (e.g. because of serialization), it can turn out to not be worthwhile to use the accelerator at all. In economically successful systems, the data will be following the path of least resistance.

6.2 Future Work

The following points are of interest in future work on Fletcher.

- **Optimization for large numbers of columns** When designs use a large number of columns, the current generic architecture is relatively inefficient on the infrastructure side, since it instantiates an equal amount of ArrayR/Ws as there are columns. In the future, it is interesting to investigate how access to data of multiple Arrow columns could be served by the same ArrayR/W to decrease the area overhead of the system.
- **Dynamically resizable buffers** Filter transformations on columns and tables produce a data-dependent amount of output data. This means that the size of the resulting Arrow buffers is unknown at compile/synthesis time. The ability to automatically resize buffers for Array Writers would therefore be a valuable addition to the tool chain. This research direction could lean heavily on reallocation techniques used in software systems oriented towards big data analytics, such as in [29], which can be used in Apache Arrow itself.
- **Automated profile-driven architectural optimization** Since Fletchgen does not have information about the kernel implementation during infrastructure generation, static optimization of the generated infrastructure is limited. As demonstrated in the example of Fig. 12a, the user has to tag a stream to enable profiling, generate and run the design, obtain the profile, and manually modify the Arrow field metadata for the stream to allow for increased throughput. Note that in the case of the example, when strings are very small, we have a relatively high number of lengths to handshake, compared to a low number of characters, and vice versa when strings are very large. If we want to iterate over all entries in that column with high throughput, we should make the length stream wider in the first case, but the character stream in the second. The characteristics of the data may therefore dictate what the best hardware configuration is, hence it is required to profile the system during run-time to obtain statistical information about the characteristics.

  While designing the kernel, a developer may not exactly know in what context the accelerator will be used, and thus, these statistics will be unknown as well. Tailoring the design to perform well in one case may cause lower performance when the accelerator is used

in another. We envision a system where a developer designs the kernel not only based on the resulting streaming interface, but also on the outcome of profiling the streaming interfaces, allowing them to generate more optimal designs based on the profile. When an accelerator is in operation and the profile changes above a certain threshold, the system can then automatically re-synthesize the design to achieve higher performance without a human in the loop. This can be seen as a fine-grained form of automated design-space exploration and optimization.

7 Conclusion

In this article, we have discussed challenges for FPGA accelerators to become widespread alternatives to existing computational solutions in the domain of big data analytics. We have stipulated three challenges from the software integration side: complex run-time system, in-memory layouts of data sets unfriendly to hardware implementations, and (de)serialization overhead. On the side of designing FPGA accelerators, we discussed three challenges as well: a relative lack of accelerator platform-agnostic open-source tooling geared towards an HDL style flow, a high design effort because of a lack of interfaces tailored towards the data structure needing to be accessed, and a high design effort because the need to design a large amount of infrastructure. We have discussed the Fletcher framework that aims to deal with these challenges. Fletcher is built on top of Apache Arrow, providing a common, hardware-friendly in-memory format, allowing developers to communicate large tabular data sets between over eleven software languages without the need for copies, preventing (de)serialization overhead. Fletcher adds hardware accelerators to the list. Several low-level hardware components were designed to deal with the mentioned challenges for table columns, providing easy-to-use, high-performance interfaces to hardware-accelerated kernels. The lower-level components are combined into a larger design, based on a generic architecture for FPGA accelerators that have tabular inputs and outputs. Through an extensive infrastructure generation framework, specialized, data type-driven specializations of the generic architecture are generated, automating the tedious work of infrastructural design. The infrastructure generation tool made specifically for Arrow is built on top of a generic C++17 structural hardware construction library called Cerata, and on an MMIO controller generator framework called Vhdmmio. Developers can focus on the design of their kernels that are supplied with easy-to-use and high-performance hardware interfaces. The Fletcher toolchain and run-time libraries drastically reduce the design and integration effort of FPGA accelerators into big data analytics pipelines while allowing the tabular data structures to be accessed at interface bandwidth.

Acknowledgements This work is part of the FitOptiVis project [30] funded by the ECSEL Joint Undertaking under grant number H2020-ECSEL-2017-2-783162. The authors thank Patrick Lysaght and Cathal McCabe from Xilinx for their additional support.

Funding Open access funding provided by Delft University of Technology.

References

1. Maas, M., Asanović, K., Kubiatowicz, J. (2017). Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era. In *Pproceedings of the 16th Workshop on Hot Topics in Operating Systems, ser. HotOS '17* (pp. 138–143). New York: ACM, https://doi.org/10.1145/3102980.3103003.
2. Peltenburg, J., van Straten, J., Wijtemans, L., van Leeuwen, L., Al-Ars, Z., Hofstee, P. (2019). Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 270–277).
3. Peltenburg, J., van Straten, J., Brobbel, M., Hofstee, H.P., Al-Ars, Z. (2019). Supporting columnar in-memory formats on fpga: The hardware design of fletcher for apache arrow. In *Applied Reconfigurable Computing* (pp. 32–47): Springer International Publishing.
4. Delft University of Technology (2020). vhlib: a vendor-agnostic VHDL IP library. [Online]. Available: https://github.com/abs-tudelft/vhlib.
5. Delft University of Technology (2020). Cerata: a Hardware Construction Library written in C++17. [Online]. Available: https://github.com/abs-tudelft/cerata.
6. Delft University of Technology (2020). Fletchgen: The Fletcher Design Generator. [Online]. Available: https://github.com/abs-tudelft/fletcher/tree/develop/codegen/cpp/fletchgen.
7. Delft University of Technology (2020). vhdMMIO: a fully vendor-agnostic tool to build AXI4-lite MMIO infrastructure. [Online]. Available: https://github.com/abs-tudelft/vhdmmio.
8. Delft University of Technology (2020). Fletcher platform-specific libraries. [Online]. Available: https://github.com/abs-tudelft/fletcher/tree/develop/platforms.
9. Caulfield, A.M., Chung, E.S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J., Lo, D., Massengill, T., Ovtcharov, K., Papamichael, M., Woods, L., Lanka, S., Chiou, D., Burger, D. (2016). A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM*

international symposium on microarchitecture (MICRO) (pp. 1–13).
10. Nane, R., Sima, V., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y.T., Hsiao, H., Brown, S., Ferrandi, F., Anderson, J., Bertels, K. (2016). A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *35*(10), 1591–1604.
11. Lahti, S., Sjövall, P., Vanne, J., Hämäläinen, T.D. (2019). Are We There Yet? A study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *38*(5), 898–911.
12. Hennessy, J.L., & Patterson, D.A. (2019). A new golden age for computer architecture. *Communications ACM*, *62*(2), 48–60. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/3282307.
13. Truong, L., & Hanrahan, P. (2019). A golden age of hardware description languages: Applying programming language techniques to improve design productivity. In *3rd Summit on advances in programming languages (SNAPL 2019) ser. Leibniz International Proceedings in Informatics (LIPIcs)*, (Vol. 136 pp. 7:1–7:21). Dagstuhl: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10550.
14. Peltenburg, J., Hesam, A., Al-Ars, Z. (2017). Pushing big data into accelerators: Can the jvm saturate our hardware? In *High performance computing* (pp. 220–236): Springer International Publishing.
15. Google Inc. (2020). Protocol buffers. [Online]. Available: https://developers.google.com/protocol-buffers.
16. Google Inc. (2020). Flatbuffers: Memory efficient serialization library. [Online]. Available: https://github.com/google/flatbuffers.
17. The Apache Software Foundation (2020). Apache Arrow. [Online]. Available: https://arrow.apache.org/.
18. Winterstein, F., Bayliss, S., Constantinides, G.A. (2013). High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *2013 International conference on field-programmable technology (FPT)* (pp. 362–365).
19. Weisz, G., & Hoe, J.C. (2015). CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing. In *2015 25th International conference on field programmable logic and applications (FPL)* (pp. 1–8).
20. Korinth, J., Hofmann, J., Heinz, C., Koch, A. (2019). The tapaSCo Open-Source Toolflow for the automated composition of task-based parallel reconfigurable computing systems. In *Applied reconfigurable computing* (pp. 214–229): Springer International Publishing.
21. Koeplinger, D., Feldman, M., Prabhakar, R., Zhang, Y., Hadjis, S., Fiszel, R., Zhao, T., Nardi, L., Pedram, A., Kozyrakis, C., Olukotun, K. (2018). 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* (pp. 296–311). New York: Association for Computing Machinery.
22. Castellane, A., & Mesnet, B. (2019). Enabling fast and highly effective fpga design process using the capi snap framework. In *High performance computing* (pp. 317–329): Springer International Publishing.
23. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K. (2012). Chisel: Constructing hardware in a Scala embedded language. In *DAC Design automation conference 2012* (pp. 1212–1221).
24. Stuecheli, J., Starke, W.J., Irish, J.D., Arimilli, L.B., Dreps, D., Blaner, B., Wollbrink, C., Allison, B. (2018). IBM POWER9 Opens up a new era of acceleration enablement: OpenCAPI. *IBM Journal of Research and Development*, *62*(4/5), 8:1–8:8.
25. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G. (2002). Graphviz— open source graph drawing tools. In *Graph Drawing* (pp. 483–484). Berlin: Springer.
26. van Dam, L., Peltenburg, J., Al-Ars, Z., Hofstee, H.P. (2019). An accelerator for posit arithmetic targeting posit level 1 blas routines and pair-hmm. In *Proceedings of the conference for next generation arithmetic 2019, ser. CoNGA'19*. New York: Association for Computing Machinery. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/3316279.3316284.
27. Peltenburg, J., Van Leeuwen, L.T., Hoozemans, J., Fang, J., Al-Ars, J., Hofstee, H.P. (2020). Battling the CPU bottleneck in apache parquet to arrow conversion using FPGA. In *2020 international conference on Field-Programmable technology (ICFPT)*.
28. Schuiki, F., Kurth, A., Grosser, T., Benini, L. (2020). LLHD: A multi-level intermediate representation for hardware description languages.
29. Evans, J. (2006). scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference, Ottawa, Canada*.
30. Al-Ars, Z., Basten, T., de Beer, A., Geilen, M., Goswami, D., Jääskeläinen, P., Kadlec, J., de Alejandro, M.M., Palumbo, F., Peeren, G., et al. (2019). 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* (pp. 333–338). New York: Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/3310273.3323437.