# Fletcher
## A framework to efficiently integrate FPGA accelerators with apache arrow

Peltenburg, J.W.; Van Straten, Jeroen; Wijtemans, Lars; Van Leeuwen, Lars; Al-Ars, Zaid; Hofstee, Peter

**DOI**
10.1109/FPL.2019.00051

**Publication date**
2019

**Document Version**
Accepted author manuscript

**Published in**
Proceedings - 29th International Conference on Field-Programmable Logic and Applications, FPL 2019

# Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow

Johan Peltenburg* Jeroen van Straten* Lars Wijtemans* Lars T.J. van Leeuwen* Zaid Al-Ars* H. Peter Hofstee*†
**Accelerated Big Data Systems, Delft University of Technology,* Delft, The Netherlands. Contact: j.w.peltenburg@tudelft.nl
†*IBM,* Austin TX, USA. Contact: hofstee@us.ibm.com

***Abstract*—Modern big data systems are highly heterogeneous. The components found in their many layers of abstraction are often implemented in a wide variety of programming languages and frameworks. Due to language implementation differences, interfaces between these components, including hardware accelerated components, are often burdened by serialization overhead. Serialization bandwidth of many high-level language frameworks is an order of magnitude lower than contemporary FPGA accelerator interface bandwidth, especially when objects are small but numerous. Therefore, serialization bounds the effective end-to-end performance of FPGA-accelerated solutions integrated with applications written in high-level languages. The Apache Arrow project defines a language agnostic columnar in-memory format optimized for big data applications, preventing the need to serialize or even make copies during communication between components. To enable FPGA accelerators to benefit from the approach of Arrow, we first investigate the properties of its format in relation to hardware interfaces and establish that the format is usable. Second, we present the Fletcher framework, that automatically generates highly efficient hardware interfaces to access data of potentially complex, nested Arrow data types. Our approach allows 11 of the languages supported by Apache Arrow libraries to efficiently communicate large data sets with FPGA accelerators at system bandwidth. Furthermore, on the hardware side, the generated interfaces deliver any data type that Arrow can represent as groups of streams, providing a better starting point for data-flow-oriented kernel development, compared to manually creating custom interfaces to address issues related to pointer arithmetic, bus word misalignment and latency. For example applications, as measured on an AWS EC2 F1 and CAPI2-enabled POWER9 system, accelerated end-to-end application performance improves by 1.3× - 49× compared to a hardware accelerated solution that still requires serialization.**

***Keywords*—FPGA acceleration, Apache Arrow, big data systems, serialization, accelerator bandwidth**

## I. Introduction

In terms of both hardware and software, the increasing heterogeneity in (cluster) computing frameworks built for big data analytics causes major challenges [1]. One challenge is that different system components that consume the same data may use different representation of that data in memory. This introduces a serialization requirement whenever data is passed from one component to another, if they are not implemented using the same technology.

Serialization is generally an unwanted necessity, as it merely transforms the form rather than the contents of the data, and is therefore a non-functional aspect. In applications built on top of these analytics frameworks, serialization may take up a large portion of the run-time of the full application [2]. Examples of where serialization takes place between components of a heterogeneous framework such as Apache Spark [3] can be seen in Figure 1(a).

The Apache Arrow project was launched to (among other contributions) overcome this bottleneck [4], and has already seen integration in several well known tools and frameworks from the data analytics community, such as Spark, Parquet and Pandas. The Arrow project defines a common columnar in-memory format for data sets and provides zero-copy inter-process communication libraries for various languages, including (at the time of writing) C, C++, Java, Python, R, Matlab, Go, C#, JavaScript, Ruby and Rust. For a schematic overview, see Figure 1(b).

In this paper, we first establish that the Apache Arrow format is also usable in the context of FPGA acceleration, where serialization bottlenecks can also be present. This can tremendously improve end-to-end accelerated application throughput, because host-side serialization throughput from various high-level languages can generally be several orders of magnitude lower than contemporary accelerator interface throughput [5].

A second advantage to using Arrow's standardized format exists. Because the in-memory format is derived from meta-data about the data sets, called *schemas*, we may also derive highly optimized hardware interfaces automatically from these schemas. From the perspective of an accelerator developer, these interfaces provide an easier starting point to interface with Arrow data sets, and in turn, to any of the languages supported by Arrow.

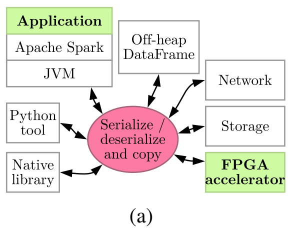
Access to objects/records and their fields can be expressed through tabular data set indices rather than the usual byte addresses, preventing the need to manually design units that perform tedious pointer arithmetic and perform the required requests on a memory interface. After supplying an index range of objects or records to process, the interface delivers streams of the exact data types expressed through the schema, rather than bus words. This allows the FPGA accelerator developer to fully focus on implementing the actual computational path of the accelerator only, rather than having to bother with the interface as well. This can normally be a cumbersome exercise, especially for data sets that consist of not just primitives such as *ints* or *floats*, but also contain more complex data types such as structure, lists and dictionaries (and any nested combination thereof).

Additionaly, an advantage from building on top of the Apache Arrow ecosystem is that through Fletcher, high-performance FPGA acceleration is made available to all supported languages. Finally, a resulting advantage from delivering object or record fields as streams is that this integrates more naturally with HLS-tools, without having to write HLS-code that is, again, interface specific.

We contribute the first implementation of these ideas in the form of a fully open-sourced (including experiments, see [6]), vendor agnostic FPGA acceleration framework called *Fletcher*. We elaborate the problem of framework heterogeneity and serialization overhead in relation to FPGA acceleration in Section II. We investigate the Arrow format in Section III. The Fletcher framework is discussed in Section IV. We present the results for four example applications using Fletcher in Section V. An overview of related work is discussed in Section VI. Section VII concludes this paper.

## II. Background

When processing a data set with an external accelerator, the data must be moved from host memory to accelerator over its interface. The bandwidth of this data transfer is maximized when the data resides in a large contiguous memory buffer (CMB) because it may be transferred using large contiguous bursts. Thus, a developer who wants to use an FPGA accelerator to speed up some application



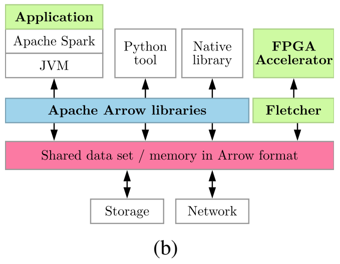


Fig. 1. (a) Examples of where serialization can take place and (b) how Arrow attempts to prevent serialization through the use of a common data layer.

must first make sure the data resides in a CMB, lest many short transfers with the associated overhead must be initiated. However, most commonly used containers and objects in various languages do not store the data in a CMB. The in-memory formats for such containers and objects are often designed for efficient use within the language run-time itself, or to provide some sort of abstraction that suits the language paradigm well. To prevent accelerators from having to traverse objects graphs, possibly incurring memory latency several times *per object*, serialization must be applied. However, serialization negatively impacts the effective bandwidth to the accelerator.

Take the example of a C++ Standard Template Library (STL) string. While it is possible to allocate the string in an STL vector such that the string objects themselves reside in a CMB, the string object constructor allocates memory for its character array using `malloc()` separately for each string. Thus, the characters (data of interest) of the string *are not guaranteed to reside in a CMB*. This is a general problem in case objects hold variable length data that is allocated by the object itself.

To continue with the example of a C++ STL string object, STL constructors can (in contemporary versions of the C++ language) be provided with custom *allocators* that could (albeit in an arguably counter-productive manner) place them in a CMB. However, if the strings are sufficiently short, the characters are actually placed in the string root object space itself (by both the LLVM and GCC implementation of the STL). This is defined in the behaviour of the constructor, and is an optimization that prevents a second memory allocation from taking place. This effectively breaks up any CMB of characters. It is therefore not possible to guarantee that the data is stored contiguously using an STL string, as we can only dictate that, *if* it allocates, it should use our custom allocator. Without rewriting the string implementation, we cannot change *when* it allocates. Thus, a developer must create some custom representation and implementation of a string, requiring extra effort.

A similar case can be made for even more abstract languages like Python or Java, where this problem is generally worse and less trivial to mitigate for the programmer, as no direct control exists over object layouts in memory. Even if this effort is spent, a data set built up like this will still suffer from more drawbacks.

Even when objects with equivalent fields are stored in a CMB, their in-memory representations are not equivalent among different language run-times, especially due to the presence of run-time specific metadata (e.g. JVM: class references, C++: virtual function tables, Python: reference counters). This (to an accelerator useless) metadata may be of significant size, especially when objects are small and numerous, as commonly seen in big data analytics. Therefore, even if the data may be stored in a CMB, *effective* bandwidth is decreased. Furthermore, it is required for an accelerator to implement a filtering step before processing, to make it a true CMB, i.e. not a CMB that also contains language-specific meta-data. This filter step furthermore would depend on the host-side language run-time used, while the function of the accelerator is essentially not different.

Even worse, object layouts are not guaranteed to be consistent inside a language itself. E.g. both the Java Virtual Machine [7] and C++ [8] do not specify or restrict how an object is laid out in memory—it is left to the implementation of the JVM and the compiler, respectively. Also, compilers may choose to optimize the lay-out, e.g. to improve alignment w.r.t. cache lines in different ways.

Summarizing, to effectively integrate an accelerator hardware design targeting a heterogeneous environment, the design must:

– be adjusted for every host-side run-time language,
– be adjusted for every compiler implementation,
– put a restriction on the application compiler/run-time, and filter language-specific metadata,
– invent a custom in-memory format for every non-primitive data type, in every language involved, or
– apply the costly act of serialization.

If one standardizes an as-contiguous-as-possible in-memory format and provides interfaces to produce/consume this data in various languages, all these options become unnecessary or irrelevant. Such a standardized solution for software is provided by the Apache Arrow project.

## III. Apache Arrow in-memory format

We investigate the general use case of Arrow data sets, where they appear to a programmer in tabular form, called RecordBatches. A RecordBatch is accompanied by a *schema* that specifies the types of the fields of the objects/records stored in the table. Each record field is stored in a separate table column.

The fact that there is a higher level description of the data structure (the schema) already provides an advantage. While designing the functional aspects of an FPGA accelerator can already be challenging, a significant portion of design time involves structural aspects of the interface. Interface design often deals with converting data on very wide hardware buses (the platforms used in this work both use 512 bits) to something more usable at the input of the accelerator. This includes pointer arithmetic to determine which bytes are the bytes of interest, parallelizing or serializing words into larger or smaller chunks, and shifting them into the right positions before turning them into data streams to be absorbed by some kernel.

The relation between the raw bytes of a RecordBatch are known from the schema and the format specification. It is therefore possible to automatically generate circuits that perform the required pointer arithmetic and pre-processing of raw bus words into streams that are more meaningful and usable to an accelerator developer. More specifically, based on the schema, an interface may be generated that as a command takes a range of object/records indices of a RecordBatch and streams out the requested fields as exactly the data types expressed in the schema. Furthermore, parts of the control and data flow on the host-side may also be automated (e.g. passing buffers addresses and potentially moving data to accelerator on-board memory).

With such a setup, is it possible to operate at system bandwidth? In general, any serialized format suitable for FPGA processing causes as few pointer traversals as possible, requires as little pre-processing or reordering in the accelerator as possible and is streamable. With this in mind, we investigate two forms of data that can be generalized to all data structures; fixed-width data fields and variable-length data fields.

### A. *Fixed-width fields*

RecordBatch columns with fixed-width elements (e.g. `floats`, `booleans` or `ints`) are in Arrow format stored in one contiguous *values buffer*, equivalent to a C-like buffer. Given some index of data to obtain, an offset has to be calculated, the specific data word (or words) have to be loaded. Upon receiving the raw bytes, the bus words have to be shaped into the correct type, before they can be presented on a streaming output. If kernels can absorb multiple elements per cycle, or if multiple kernels want to read from the same column in parallel, it is possible to match system bandwidth on such an interface. Assuming a kernel requests the full range of objects from the table, only one "pointer" is traversed to read this field for all objects of interest with maximum size pipelined bursts on the memory interface.

This is much more efficient than if the accelerator would have to traverse a pointer for each fixed-width element. For a C programmer it may seem far fetched for a collection of integers to be stored as a list of pointers to integers. However, some high-level languages (such as Python and R) box every integer into an object (hence the need for e.g. *Numpy*). Any interface dealing with such an in-memory lay-out will quickly be bounded by memory latency if such a collection of integers is to be traversed through pointers to the integer objects.

### B. Variable-length fields

More interesting are Arrow columns of variable length types (e.g. a UTF-8 string). They are referred to as lists of some other type (e.g. a `List<Char>` or `List<List<Int>>`). They contain at least two buffers, an *offsets buffer* and the values buffer. An offset at some index in the offsets buffer corresponds to the index of the first element of the list in the values buffer. The values buffer contiguously holds all primitive list elements. This format offers some advantages that an interface generation framework may exploit over what HLS-compilers can assume about this data structure.

More formally, consider the case where a variable length object is represented through two Arrow buffers; the offsets buffer $O = \{o_1, o_2, ..., o_N\} \in \mathbb{Z}^{\geq}$ and values buffer $V = \{v_1, v_2, ..., v_M\}$. $O$, in the C-language, will be represented as an unsigned integer array. A C-based HLS compiler may not make assumptions about the values of $o_i$, as they are defined during run-time. More specifically, it cannot assume that in the case of an Arrow offsets buffer, $o_{i+1} - o_i \in \mathbb{Z}^{\geq}$; the outcome of this calculation might also yield a negative integer. Therefore, not to lose generality it must request each run of value buffer elements $v_{o_i} ... v_{o_{i+1}}$ separately, and any data path consuming the data is subject to memory latency.

Hardware pre-fetching (such as explored in [9]) or using spatial locality in caches may improve this behavior, but these constructs are costly, especially when, in the case of the Arrow format, they are not required. To elaborate, when requesting a range $j...k$ of variable length objects, in fact the whole range of values of interest $v_{o_j} ... v_{o_{k+1}}$ can be requested from the contiguous buffer. This can be bursted into a FIFO, ready to be delivered on the output stream synchronized with a length stream resulting from subtracting two consecutive offsets. Thus, memory latency for pointer traversal is only paid three times independent of the amount of variable length objects that are requested; once to obtain $o_{k+1}$ from the offsets buffer, once to obtain all offsets of interest $o_j ... o_{k+1}$, and once to obtain all values of interest. No dynamic hardware pre-fetching or caches are required to deliver throughput that is close to system bandwidth. This approach also generalizes to nested lists.

Furthermore, with these assumptions, this interface can be generated automatically, without the need to manually write an HDL-based interface or the need to write special HLS functions that mimic this optimal behavior. HLS templates for transformation functions used in higher-order functions such as map, filter and reduce, can immediately be provided with length stream and value stream as arguments. Again, this approach generalizes to nested types.

Arrow also supports other convenient data types such as *structs*, sparse and dense unions and dictionaries, which are discussed in its format specification. Furthermore, a special type of fixed-width field that contains a validity bit to allow entries to be nullable is supported.

### C. Limitations

Some limitations to the Arrow approach exist. First, once data sets have been built in memory, it is not trivial to mutate them without breaking contiguousness. Therefore, Arrow is best at storing immutable data sets in memory but less powerful when working with algorithms that aim to mutate data sets in place. Second, at the time of writing, no data format is specified for graph-based data sets, or other more exotic non-tabular formats. Still, graphs can generally be represented through tables, although there is, at the time of writing, no Arrow standard specification. A final limitation is that because a different in-memory format is used than some language run-time is used to, code that accesses data (accessors) must go through an additional layer (e.g. some Arrow language specific library) rather than being able to use default ways of accessing object or record fields. While investigating this drawback, we did not find any significant performance degradation. We have investigated C++ (a case where code is compiled to native instructions), where the performance of accessing Arrow based containers is similar and sometimes faster than accessing STL containers, as Arrow exposes raw pointers to the data buffers. For Java (a case where code is compiled to virtual machine bytecode), access to Arrow based data is done through calls to the Unsafe library, as the data is stored outside the VM managed heap. Fortunately, widely-used implementations of the JVM inline these calls during JIT compilation, providing similar performance to normal object field accessors. In Python (a case where code is interpreted), it is common for high performance libraries to use native code underneath (e.g. NumPy) written in Cython. This involves extra developer effort but is a common trade-off made in the Python ecosystem.

Establishing that aside from these limitations, the Arrow in-memory format is indeed suitable since it is highly contiguous and streamable, the next section will discuss the implementation of an interface generation framework based on the Arrow format.

## IV. Fletcher

### A. Overview

A high-level overview of our FPGA acceleration framework that exploits the benefits of the Arrow format, called Fletcher, is seen in Figure 2. In this figure, the general compile-time and run-time flow is depicted. At compile-time, a developer starts with an Arrow schema. From the schema, a default HDL or HLS template for the accelerated function implementation and an interface that will provide streams of requested data from the Arrow table are generated (see Section IV-B). These sources are synthesized, placed and routed to provide the FPGA bitstream. At run-time, the enumerated steps in Figure 2 are taken:

1) Starting with a data source (e.g. a Parquet [10] file on disk), the data is loaded into memory.

2) Rather than loading the data set into a language native container (that would incur serialization overhead as soon as the data is needed in the accelerator), the application will ingest the data into memory formatted as an Arrow-based data set (e.g. a RecordBatch). Arrow library functions will place the data in host memory according to

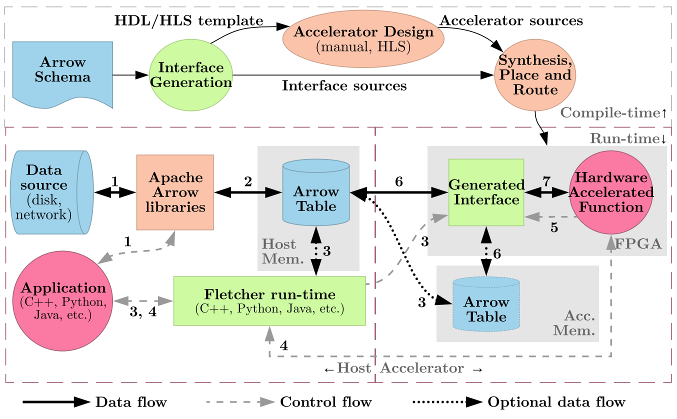


Fig. 2. Architectural overview of Fletcher. Upper part of the figure shows the compile-time (development) flow, lower part of the figure shows the run-time flow for host system (left) and accelerator (right).

the schema and the format specification (if not already in the Arrow format).

3) The application can request the Fletcher run-time libraries to prepare the Arrow data set for processing on the accelerator. For some platforms this simply means passing virtual addresses of the buffers [11], and for other platforms this means a copy of the buffers must be made to accelerator on-board memory. This process is fully automated in the Fletcher run-time libraries. Basic use requires the user to only claim the platform / accelerator card, create a context in which the on-board memory is managed by the run-time, bind a host-side abstract representation of the Hardware Accelerated Function to a context, and provide the input RecordBatches as an argument to the Hardware Accelerated Function. Advanced users may use lower-level API calls to the Fletcher run-time system to e.g. place other data in the accelerator memory and control other data paths not generated through Fletcher.

4) The application can now issue commands to the functional part of the accelerator, the Hardware Accelerated Function (HAF). Commands include setting arguments, reset, start, stop and poll for completion.

5) After the HAF receives the commands from the application, it can request a row or ranges of rows from the generated interface through a pipelined command stream.

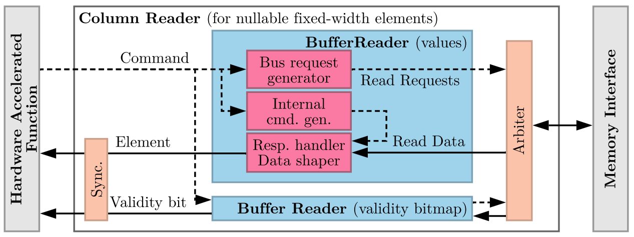
6) The generated interface will request the desired data from the host memory or the accelerator on-board memory.

7) After receiving the data from the memory, the interface provides streams of data back to the HAF, containing the data from the requested rows and fields, in the form specified in the schema.

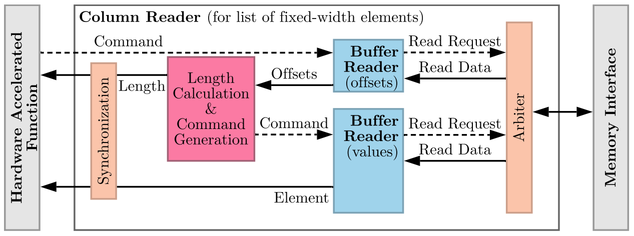
The last two steps can be reversed in case the HAF wants to write to an Arrow data set in memory.

## B. Interface generation

Since a RecordBatch in Arrow is an abstraction of a group of columns, a Fletcher generated top-level interface is called a ColumnReader (CR). The CR internals are generated in pure VHDL by parsing a *configuration string* that conveys information about the schema required to generate the hardware structure. Thus, to generate the hardware interface from a schema involves transforming the schema into a string which is done through a command-line tool called *Fletchgen*. The core logic of the interfaces that are generated are based on a vendor-agnostic pure HDL streaming primitives library as a part of the Fletcher framework.



(a) ColumnReader for nullable, fixed-width elements. The outputs of two BufferReaders are combined to deliver the field value with its validity bit synchronously.



(b) List of fixed-width elements (non-nullable). Two BufferReaders are combined, where the offsets BufferReader provides a command for the values BufferReader. A list length stream and list element stream is provided to the computational kernel.

Fig. 3. Generated internal architecture of ColumnReaders for two examples.

The configuration string causes CRs to internally configure for different column types. A fixed-width type (as seen in Figure 3(a) will result in a CR configuration to read a single values buffer through a component called a BufferReader (BR). BRs include bus request generation and response reshaping logic and deliver exactly the fixed-width type of the schema.

Developers may add metadata to the Arrow schema and its fields to generate interfaces that, e.g. deliver multiple elements per cycle, contain more or less register slices in data paths, contain shallower or deeper FIFOs or even ignore schema fields altogether if they are not of interest. This allows the developer to make trade-offs between area, power and performance.

Reading variable-length data chains multiple BRs as seen in Figure 3(b), where one BR reads from an offsets buffer and through a specialized component generates new commands for a second BR that reads the values buffer. Although not shown in the figures, CRs may recursively instantiate themselves to support, for example, lists of lists; combining an offsets BR on its own top level and another offsets BR and a values BR in the level below. Other options, such as *struct* types are also supported, that instantiate a CR for each struct field and synchronize their output streams on a top level CR. While this paper focuses on the general motivation and overview from the application-level, previous work has described in detail the multitude of specific digital design challenges solved by to the streaming primitives library and how they are combined into a CR [12].

The Fletchgen tool creates a wrapper around the whole design (where multiple CRs can be instantiated), and generates an HDL or HLS template for the HAF, abstracting away all non-Arrow related interfaces. The top-level of the hierarchical design that Fletchgen generates currently provides the commonly used AXI4 (for data paths) and AXI4-lite (for control paths) interfaces. A schematic overview of all the components involved in this higher-level hierarchy are shown in Figure 4. Since this generation step only involves structural aspects of the design, it is implemented to perform the

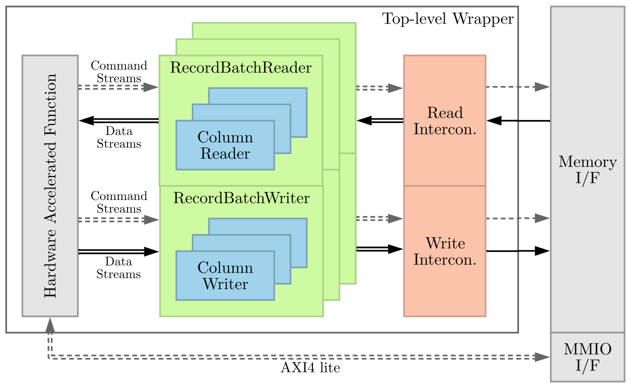


Fig. 4. Schematic overview of the upper layers of a Fletcher-based design, fully generated through Fletchgen. The ColumnReader/Writer, RecordBatchReader/Writer, Read/Write Interconnect and Top-level Wrapper components are all derived from the supplied Arrow schema.

following three stages:

*1) Construction:* The first stage converts Arrow more software-oriented data type descriptions to Fletcher-specific data type descriptions. These descriptions are augmented with hardware-specific traits, e.g. to allow nested or variable length data types to be moved as a bundle of streams operating in a specific clock domain. These hardware-oriented type descriptions can then be used to instantiate signals, ports and components in an abstract, graph-based intermediate representation of the structural design. Through this intermediate representation, Fletchgen instantiates and hierarchically groups multiple ColumnReaders and ColumnWriters according to the supplied set of Arrow schemas. Each schema results in a schema-specific component called a RecordBatchReader/Writer (where each field of a record in a RecordBatch can still be individually accessed). All RecordBatchReaders/Writers are combined into a top-level wrapper in which also an appropriate bus infrastructure is generated.

*2) Transformation:* In the second stage of generation, the intermediate representation is supplied to a back-end, where the abstract structural representation is transformed to a version suitable for emission as source files for downstream tools. Currently, there is a VHDL back-end, and a back-end for DOT graphs [13] that allow fast visual inspection and debugging. For example, since VHDL does not allow port-to-port connections of instantiated components, the VHDL back-end will resolve this by inserting a signal in between. Other transformations include the expansion of abstract types to physical types. One example for the VHDL back-end is to expand a *stream* of some other type to contain physical valid/ready handshake signals.

*3) Emission:* In the final stage, the transformed graph-based representation is emitted as source code.

The combination of the ColumnReaders and Fletchgen elevates the level of abstraction up to the point where a developer can simply provide a set of Arrow schemas, obtain a template for the HAF and work on the functional aspects of the accelerator right away. Cooperative design efforts between software and hardware designers can benefit from the schema representation of the data as well, as a means of defining a (data-oriented) interface between hardware and software, agnostic of the software language run-time framework used. In other words, using Fletcher to create an FPGA accelerator design enables the efficient use of this accelerator in any of the (at time of writing) 11 languages that Apache Arrow supports.

### *C. Simulation*

Fletchgen allows conversion of existing Arrow RecordBatches to a memory model for simulation that mimics a host interface and memory. In this way, a designer may perform hardware/software co-design of the HAF in simulation, agnostic of the final implementation platform.

To validate the correctness of the CRs themselves, schemas were generated randomly, where at each schema nesting level (within structs and lists) the complexity decreases on average such that eventually the nesting ends. Data sets based on this schema were generated randomly and random ranges of data are requested. The resulting stream outputs are checked with the expected outcome. Using this method, over ten thousand generated interfaces are validated in simulation.

Although only reading from Arrow data sets has been discussed so far, Fletcher allows the generation of interfaces that write to Arrow data sets in host memory as well. Components such as ColumnWriters, BufferWriters, etc. are implemented that reverse the streams shown in Figure 3.

## V. Results

We implement four applications using Fletcher to investigate its characteristics. We benchmark on an Amazon Web Services EC2 F1 system equipped with a proprietary card that contains a Xilinx XCVU9P (AWS/F1) and a POWER9 Barreleye system equipped with an AlphaData ADM-9V3 equipped with a Xilinx VU3P attached through CAPI 2.0 using the SNAP framework (P9/SNAP) [14]. These systems can simultaneously run 8 and 144 hardware threads on their CPU(s), respectively. The P9/SNAP system allows Fletcher to read and write directly from and to host memory using virtual addresses, hence implementations using this system do not require copies to accelerator on-board memory.

Each application is implemented in C++ (GCC), Python (3.6) and Java (OpenJDK 8). The application software-only throughput is measured. The bandwidth of serialization from a language specific container to a format usable by FPGA (in our case the Arrow format) is also measured. Finally, the FPGA throughput and copy bandwidth are measured.

### *A. Regular expression matcher*

In this example, a large collection of (tweet-sized) strings is matched to a set of sixteen regular expressions. The number of matches are counted for each regular expression. It is an application that is fully streamable and generally performs extremely well on FPGA — hence any serialization overhead can penalize its potential performance tremendously. With this example application, we can measure the performance of CRs that fetch variable-length objects (UTF-8 strings). The software kernels use the fastest regex matching libraries we could find. In C++ we use the RE2 library [15] and spread the workload over all available hardware threads. We use the Python wrappers for the RE2 library as well and the standard multiprocessing module to spread the workload over all hardware threads. In Java the built-in regex matcher is fastest, and has also been parallelized over all hardware threads.

In the FPGA implementation, we place multiple streaming regex matching units in parallel where each unit has a CR configured to deliver four characters per cycle at 250 MHz. This setup matches the peak theoretical throughput of 16 GB/s for the on-board DDR interface. A data set with random length strings between 0-255 with a total size of 1 GiB is used as an input.

From the run-time measurements, shown in Figure 5 and throughput measurements, shown in Table I, we find that the FPGA kernel

TABLE I
REGULAR EXPRESSION MATCHING RESULTS

| | | Throughput (GB/s) | | | | Speedup | | |
|---|---|---|---|---|---|---|---|---|
| System | Language | Native data set w/ CPU | Serialization | FPGA Copy | FPGA Kernel | w/ Serialization | Arrow/Fletcher | Improvement |
| AWS/F1 (16 regex units) | C++ | 0.08 | 0.55 | 7.13 | 14.27 | 6.18 | 59.73 | 9.67 |
| | Python | 0.04 | 0.83 | 7.17 | 14.28 | 15.93 | 107.73 | 6.76 |
| | Java | 0.03 | 0.27 | 7.13 | 14.27 | 8.24 | 152.91 | 18.56 |
| P9/SNAP (8 regex units) | C++ | 0.43 | 0.81 | n/a | 7.61 | 1.70 | 17.78 | 10.44 |
| | Python | 0.11 | 0.81 | n/a | 7.61 | 6.77 | 70.72 | 10.45 |
| | Java | 0.16 | 0.16 | n/a | 7.61 | 0.95 | 46.49 | 48.69 |

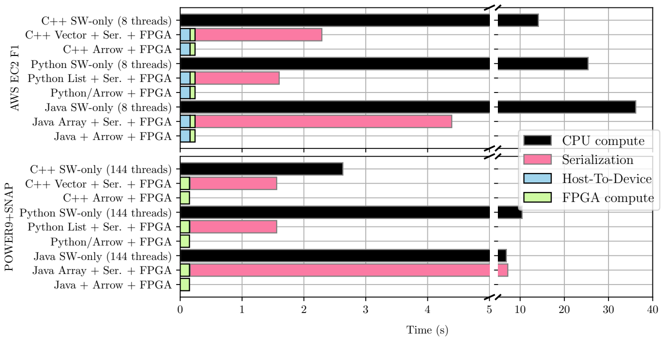


Fig. 5. Run-time components of regular expression matching accelerator

vastly outperforms the CPU implementation, as expected. However, to get the data set to the accelerator *we must first serialize it*. The serialization throughput for each language is below 1 GB/s, while the EC2 F1 platform has a copy bandwidth of over 7 GB/s. Once the data is on the on-board memory, the parallel CRs are able to stream the data to the regex units at over 14 GB/s (achieving almost 90% of the peak bandwidth). Through the use of the Arrow in-memory format and interfacing with the data through the use of Fletcher, the end-to-end speedup improves by over 9×. A similar advantage may be observed in the POWER9/SNAP case. In the particular case of the Java implementation on this platform, serialization dominates so much, that even though the accelerator exhibits an almost two orders of magnitude higher throughput, it would not be worthwhile to use the accelerator when serialization has to take place. This is mainly due to a very low serialization throughput, and as a result, using Fletcher yields a very high improvement factor.

Additionally, we find the FPGA area utilization of each CR for this example to be 1.45% CLBs and 0.21% BRAM tiles for the XCVU9P). Further details on area utilization of a wide variety of ColumnReader/Writer configurations can be found in [12]. In summary, the CLB utilization ranges from 0.02% for primitive types with the width of the data bus to 2.34% for ColumnReaders for lists of primitives able to deliver 64 elements per cycle.

## B. String writer

In this example, we consider *writing* to Arrow RecordBatches from FPGA. Use cases include the FPGA being the data source or being in the data-path from another source to host memory (e.g. data coming from a network interface or storage). The data source contains a set of string lengths and a set of string characters (similar to e.g. how uncompressed Parquet files store strings). Our intent is to measure how fast ColumnWriters can write variable-length objects into a format that is usable by the software-language run-times that Arrow supports.

TABLE II
STRING WRITER RESULTS

| | | Throughput (GB/s) | | | |
|---|---|---|---|---|---|
| System | Language | To native container | To Arrow RecordBatch | FPGA copy | Total (Arrow RecordBatch) |
| AWS/F1 | C++ | 0.85 | 2.53 | - | 2.53 |
| | Python | 0.96 | 2.60 | - | 2.60 |
| | Java | 0.59 | 1.81 | - | 1.81 |
| | FPGA | - | 9.76 | 2.75 | 2.15 |
| P9/SNAP | C++ | 0.76 | 7.52 | - | 7.52 |
| | Python | 1.60 | 7.68 | - | 7.68 |
| | Java | 0.28 | 3.96 | - | 3.96 |
| | FPGA | - | 9.76 | - | 9.76 |

Because connecting the FPGA to an actual flash drive or network interface is outside the scope of this work, we mimic such an input in FPGA by generating a character stream with 64 characters per cycle (at 250 MHz) and another stream with pseudo-random lengths between 0-255, resulting in a total data size of approximately 1 GiB. The length stream is generated uniformly random between 0-255. This results in the 64-character input stream where every handshake on average only has 75% valid input data, resulting in a peak input rate of 12 GB/s. In software, the time to deserialize the same data source to a language native container (C++: `vector<string>`, Python: `list` of strings (using Cython), Java: `Array<String>`, all pre-allocated where applicable) as well as to an Arrow RecordBatch is measured (in the Python case by wrapping the C++ implementation).

From the measurement shown in Table II, it can be concluded that the Arrow format itself already gives a performance benefit because it does not require the need to allocate memory for each string object separately. The ColumnWriters of the FPGA implementation are able to generate the Arrow RecordBatch at an even higher throughput of almost 10 GB/s, slightly over 80% of the average input bandwidth of 12 GB/s. The device-to-host bandwidth of the AWS/F1 system only delivers 2.53 GB/s, causing a bottleneck for the FPGA implementation. This is expected to be increased while the AWS/F1 system is further developed. For the P9/SNAP system, a more modest speedup of 1.3× is observed.

## C. K-means clustering

We perform K-means clustering (only on AWS/F1) of a data set of integer feature vectors; a common kernel in data analytics that is computationally intensive. The algorithm is of a more iterative nature; it is not fully streamable and therefore the impact of serialization is expected to be less dominant. At the same time, using Fletcher we may generate an easy-to-use interface that delivers streams of vectors of which the lengths is defined during run-time.

The C++ implementation uses a vector of feature vectors as input data and performs the clustering using a parallelized implementation on all hardware threads. The Python implementation wraps the C++ implementation through Cython. The Java implementation uses an `ArrayList` of `ArrayLists` as an input dataset and is also multi-threaded. The FPGA implementation processes one feature vector per clock cycle, where up to 16 features can be processed in parallel from the input stream received from the ColumnReader. For every iteration of the K-means algorithm, the whole data set is requested through the CRs. For our dataset of aprox. 1 GiB of feature vectors, the number of iterations was 25, and thus we may calculate the average bandwidth per iteration for all implementations.

The results of this measurement are shown in Table III. It can be seen that the bandwidth of the ColumnReader grows close to the peak bandwidth (delivering up to 70%, although computational aspects of the implementation are also included in this measurement). The results show that even for a computational intensive algorithm like

TABLE III
K-MEANS CLUSTERING RESULTS

| | Avg. GB/s | | Total run-time (s) | | |
|---|---|---|---|---|---|
| Language | CPU | FPGA | CPU | FPGA (w/ ser) | FPGA (w/o ser) |
| C++ | 1.40 | 11.15 | 19.24 | 6.08 | 2.55 |
| Python | 1.29 | 11.15 | 20.77 | 8.07 | 3.03 |
| Java | 1.00 | 11.15 | 26.92 | 3.88 | 2.55 |

K-means, the benefit can be substantial (up to 2.7× in this particular case).

### D. HLS-based filter

In this example, augmenting an existing commercial HLS tool (Vivado HLS) with Fletcher is investigated. An Arrow RecordBatch was created with two columns containing a string and a third column containing an integer. On this RecordBatch, an SQL-like query is be performed that exactly matches the contents of one string and the integer, and returns the other string column.

Initially, the kernel is described as a C++ function that has pointer arguments to the used Arrow buffers. The HLS tool initially cannot compile this kernel as there is no static information about the size of the buffers. After adding a pragma for each of the buffer pointers we are able to compile an implementation that communicates with a memory bus. Assuming an off-chip memory latency in the order of a hundred nanoseconds ($\sim$25 cycles at 250MHz), this kernel incurs memory latency for the outer loop that iterates over all strings. This results in an outer loop iteration latency of at least 49 cycles with an inner loop iteration latency of two cycles. Only a fraction of the cycles are spent on actual work; the kernel is memory latency bound. Additional pragmas and rewriting the kernel in a specific way would allow to optimize this behavior, even as so far to possibly write additional functions that mimic Fletcher's approach. However, Fletcher helps automate this process and overcomes the need for rewriting the kernel.

In a second implementation, using *Fletchgen*, an interface is automatically generated based on an Arrow schema. The interface provides the ability to write the kernel as a C++ function with `hls::stream<type>` arguments. The input streams provide the properties of the string; a length and character stream for each string, and a stream with the integer. After the filter step has been performed, the kernel may push characters and lengths into the output stream. Again an outer loop over all strings and an inner loop over all characters is created. The HLS tool is immediately able to compile the kernel without the use of any pragmas. Because there are no bus requests, the minimum latency of the outer loop is much smaller; only 5 cycles. In this example, the iteration latency is improved by almost 10×. This means that our approach enables users to skip the tedious step of writing HLS-oriented C++ code to interface more efficiently with the data, while providing better performance at the same time (for reasons explained in Section III-B). Developers are allowed to immediately focus on the computational aspect of the kernel.

## VI. Related work

Previous works have acknowledged serialization bottlenecks within the context of FPGA acceleration and big data processing frameworks, see [16][17][18][5].

Integrating big data processing frameworks with GPGPU and FPGA have been explored for Spark with Scala (JVM based) and OpenCL [19], although the focus is on programmability rather than attempting to alleviate serialization bottlenecks. Research to optimize interfaces with specific languages such as Java have been attempted [20], but still require a software serialization step. Other work shows highly configurable hardware templates based on common SQL-like operations written in a C# variant [21]. Because storing structured tabular data in Arrow is one of its main use cases, it would be interesting to merge the lessons learned from this language specific approach and bring them to the more language agnostic Fletcher. Other works have shown that analytics applications, e.g. [22][23], and database applications, e.g. [24], can benefit from FPGA acceleration in general

## VII. Conclusion

The increasing heterogeneity in big data analytics frameworks is burdened by serialization overhead. The Apache Arrow project provides methods to overcome serialization bottlenecks during inter-process communication for various software languages by providing a common columnar in-memory format for data sets. We show that the format is highly suitable for FPGA accelerators because it is highly contiguous in memory and requires a minimum amount of pointer traversals to access a collection of data objects. Based on descriptions of the type of data set in an Arrow schema, highly efficient and easy to use FPGA accelerator interfaces may be generated automatically. These interface can lift the level of abstraction without losing performance. This idea is implemented in Fletcher; the first FPGA accelerator framework to make use of the Arrow format. Fletcher is open-source and vendor-agnostic. Three use-cases show that the combination of Arrow and Fletcher can be beneficial to the end-to-end throughput of an FPGA accelerated application, especially when the accelerated operation is streamable. For these cases, the benefit was shown to range from 1.3× - 49×, depending on the characteristics of the applications and the implementation platform. For a fourth use case that uses an HLS-based design flow, Fletcher allows the kernel to be expressed using stream arguments rather than buffer pointer arguments, increasing the ease of use and integrated performance of a commercial HLS tool.

Future work will focus on two aspects. First of all, a more extensive set of benchmarks should be performed to characterize the approach for different applications in more detail. Secondly, in the code generation step, profiling components could be inserted to provide run-time information about the utilization of different generated hardware resources. When fed back to the code generation step itself, it would be interesting to investigate if these profiles could be automatically used by the code generation step itself to optimize its output.

In summary, Fletcher enables fast and efficient integration of FPGA accelerators into any software language that Apache Arrow supports or any analytics or database framework that uses Apache Arrow in its back-end.

## Acknowledgment

This work has been supported by the Fitoptivis European ECSEL project no. ECSEL2017-1-737451. The authors thank Xilinx for their additional support.

## References

[1] M. Maas, K. Asanović, and J. Kubiatowicz, "Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: ACM, 2017, pp. 138–143. [Online]. Available: http://doi.acm.org/10.1145/3102980.3103003

[2] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making Sense of Performance in Data Analytics Frameworks," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, 2015, pp. 293–307.

[3] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A Unified Engine for Big Data Processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[4] The Apache Software Foundation, "Apache Arrow," 2018. [Online]. Available: https://arrow.apache.org/

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

[6] Accelerated Big Data Systems Group, "Fletcher - A framework to integrate FPGA accelerators with Apache Arrow," https://github.com/abs-tudelft/fletcher, 2018.

[7] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification*. Pearson Education, 2017.

[8] I. O. for Standardization, "ISO/IEC 14882:2017, Programming languages–C++," *ISO/IEC*, vol. 14882, 2017.

[9] T. Chen and G. E. Suh, "Efficient data supply for hardware accelerators with prefetching and access/execute decoupling," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[10] The Apache Software Foundation, "Apache Parquet," 2018. [Online]. Available: https://parquet.apache.org/

[11] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, "CAPI: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 1–7, 2015.

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

[13] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph static and dynamic graph drawing tools," in *GRAPH DRAWING SOFTWARE*. Springer-Verlag, 2003, pp. 127–148.

[14] OpenPOWER foundation, "CAPI SNAP Framework Hardware and Software," 2018. [Online]. Available: https://github.com/open-power/snap

[15] Google, "RE2, a regular expression library," 2018. [Online]. Available: https://github.com/google/re2

[16] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei, "When Apache Spark meets FPGAs: a case study for next-generation DNA sequencing acceleration," in *The 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[17] E. Ghasemi and P. Chow, "Accelerating Apache Spark Big Data Analysis with FPGAs," in *UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld*, July 2016, pp. 737–744.

[18] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: ACM, 2016, pp. 456–469.

[19] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala, "Sparkcl: A unified programming framework for accelerators on heterogeneous clusters," *arXiv preprint arXiv:1505.01120*, 2015.

[20] J. Cong, P. Wei, and C. H. Yu, "From JVM to FPGA: Bridging abstraction hierarchy via optimized deep pipelining," in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. Boston, MA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/hotcloud18/presentation/cong

[21] E. S. Chung, J. D. Davis, and J. Lee, "Linqits: Big data on little clients," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 261–272.

[22] R. Mueller, J. Teubner, and G. Alonso, "Data processing on FPGAs," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 910–921, 2009.

[23] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, "Database Analytics Acceleration Using FPGAs," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 411–420.

[24] P. Papaphilippou and W. Luk, "Accelerating database systems using fpgas: A survey," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2018, pp. 125–1255.