Supporting Columnar In-memory Formats on FPGA
The Hardware Design of Fletcher for Apache Arrow

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

DOI
10.1007/978-3-030-17227-5_3

Publication date
2019

Document Version
Accepted author manuscript

Published in
Applied Reconfigurable Computing

Supporting Columnar In-Memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow

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

[1] Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
[2] IBM Research, 11500 Burnet Rd, Austin TX 78758, USA
j.w.peltenburg@tudelft.nl

Abstract. As a columnar in-memory format, Apache Arrow has seen increased interest from the data analytics community. Fletcher is a framework that generates hardware interfaces based on this format, to be used in FPGA accelerators. This allows efficient integration of FPGA accelerators with various high-level software languages, while providing an easy-to-use hardware interface for the FPGA developer. The abstract descriptions of data sets stored in the Arrow format, that form the input of the interface generation step, can be complex. To generate efficient interfaces from it is challenging. In this paper, we introduce the hardware components of Fletcher that help solve this challenge. These components allow FPGA developers to express access to complex Arrow data records through row indices of tabular data sets, rather than through byte addresses. The data records are delivered as streams of the same abstract types as found in the data set, rather than as memory bus words. The generated interfaces allow for full system bandwidth to be utilized and have a low area profile. All components are open sourced and available for other researchers and developers to use in their projects.

Keywords: FPGA · Apache Arrow · Fletcher

1 Introduction

The domain of data analytics is becoming increasingly mature. Various solutions for e.g. scalable computing on large distributed data sets, easy to use data structuring interfaces, storage and visualization exist (e.g. respectively Spark [12], Pandas [3], Parquet [10], etc.). At the same time, the demand to process this data in a more efficient manner increases as well. To overcome limitations with serialization bottlenecks for heterogeneous software systems, an Apache project named Arrow [9] was launched to provide a common in-memory format for big data. The project provides libraries for (at the time of writing) 11 different languages to consume or produce data sets in the common in-memory format. This

This work has been supported by the Fitoptivis European ECSEL project no. ECSEL2017-1-737451.

elevates the need to serialize data stored as language-native run-time objects when performing inter-process communication between application components running in different language run-times. Zero-copy inter-process communication is made possible through the common data layer that is offered by Arrow.

FPGA accelerators may also benefit from this format. The no-serialization advantage has been exploited in an open-source, hardware-agnostic FPGA acceleration framework called Fletcher [6] for which an overview and general motivation may be found in [7]. The goal of the project is to generate interfaces based on Arrow meta-data called *schemas* that provide an abstract description of the type of data in a tabular data set. Because the in-memory representation follows from the schema, an interface can be generated based on the schema that fetches the data based on a table index rather than a byte address, delivering exactly the data object expressed through the schema, rather than a bus word. This increases the programmability for the hardware developer - they can focus on the accelerator implementation rather than spending time on the platform specific interface and host-side software to shape data into a format useful for the accelerator.

In this paper we describe the internals of the hardware solution of Fletcher to support a set of common Arrow data types. This is challenging because on the one hand, schemas can widely vary, and on the other hand, platform specific interfaces can widely vary. Section 2 introduces the background. Next, we list some requirements for the hardware components of Fletcher in Section 3. The main contributions of this work can be found in sections Section 4 through Section 5. A vendor-agnostic hardware library that is used in Fletcher is introduced in Section 4. Section Section 5 shows how the components from the library are combined into designs that can read from Arrow data sets through a host-memory interface and reshape the data into a format desired by the schema. Functionality and performance for a large variety of schemas are verified in Section 6. Related work not discussed throughout the paper is discussed in Section 7. We conclude this paper in Section 8.

2 Background

2.1 Problem definition

To explain why the use of Arrow with FPGA accelerators is relevant, consider an example use-case of matching regular expressions to a column of UTF8-strings (a common operation performed on strings that are stored in databases or event logs). Evaluating regular expressions in hardware is known to be efficient and streamable with state-of-the-art work shows a throughput of 25.6 GB/s [8]. This significantly exceeds the available interface bandwidth (e.g. 8GB/s for PCIe Gen3 x8). However, to attach such an FPGA accelerator to a high-level language, language native strings need to be serialized to a usable format. The throughput of serializing approximately 1 GiB data set of language native strings in C++, Java and Python in software on an Intel Xeon machine and an IBM POWER9 machine are shown in Tab. 1. From this table, it can be seen that the serialization

Tab. 1: Serialization throughput of various language run-times of 1 GiB of strings

Language:		**C++** (gcc)	**Java** (OpenJDK)	**Python** (CPython 3.6)
Throughput (GB/s):	Xeon E5-2686	0.55	0.83	0.27
	POWER9 Lagrange	0.81	0.81	0.16

throughput of language-native string objects to a usable format in FPGA is not in the same order of magnitude as host-to-accelerator bandwidth.

Using Fletcher framework for FPGAs allows exploiting the more efficient in-memory format of Arrow and allows large data sets to be streamed in at system bandwidth. Fletcher is operational on two major FPGA platforms meant for data-center and cloud applications; the OpenPOWER CAPI SNAP framework [4] and the Amazon Web Services (AWS) EC2 F1 instances [1].

2.2 Apache Arrow

Arrow data sets are typically tabular and stored in an abstraction called a RecordBatch. A RecordBatch contains several columns for each field of a record, that are in Arrow called *arrays*. These arrays can hold all sorts of data types, from strings to lists of integers, to lists of lists of time-stamps, and 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. In this work we consider *validity* buffers, *offset* buffers and *value* buffers.

Validity buffers store a single bit to signify if a record (or deeper nested) element is valid or *null* (i.e. there is no data). Value buffers store actual values of fixed-width types, similar to C 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) Schema:

```
Field A:
Float (nullable)
Field B:
List(Char)
Field C:
Struct(E: Int16, F: Double)
```

(b) RecordBatch:

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

(c) Arrow Buffers:

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				!		

Fig. 1: An example schema (a) of a RecordBatch (a) and resulting Arrow buffers (c).

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. An example of a *schema*, a corresponding *RecordBatch* (with three *arrays* and the resulting *buffers* are seen in Fig. 1.

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.

Because schemas can express a virtually infinite number of type combinations an implementation of the mechanisms must meet a challenging set of requirements. In the next section, we first describe the requirements of such an interface.

3 Requirements

Consider an accelerator to be the data sink in case an Arrow RecordBatch is being read. From the description in the previous section, a set of requirements for the generated interface can be constructed.

1. **Row indexing:** The data sink is able to request table elements by using Arrow table row indices as a reference. In turn, the data sink will receive the requested elements only.
2. **Streaming:** The elements will be received by the sink in an ordered stream.
3. **Throughput:** The interface can be configured to supply an arbitrary number of elements in a valid transfer.
4. **Bus interface:** The host-memory side of the interface can be connected to a bus interface of arbitrary power-of-two width.

The first requirement allows developers to work with row indices rather than having to perform the tedious work of figuring out the byte addresses of data (including potentially deeply nested schemas with multiple layers of offset buffers). Furthermore, it implies that elements are received in the actual binary form of their type, and not, e.g., as a few bytes in the middle of a host memory bus word (that are often 512 bits wide for contemporary systems). This allows the developer to not have to worry about reordering, serializing or parallelizing the data contained in one or multiple bus words.

The second requirement maps naturally to hardware designs that often involve data paths with streams of data flowing between functional units.

The third requirement allows multiple elements of a specific data type to arrive per clock cycle. For example, when a column contains elements of a small type (say a Boolean), it is likely the accelerator can process more than one element in parallel. This differs from Requirement 2 in the sense that the elements that will be delivered in parallel are part of the same request mentioned

in Requirement 1. Furthermore, it can be that the top level element is a list of small primitive elements. Thus, one might want to absorb multiple of the nested elements within a clock cycle.

The last requirements allows the interface to be connected to different platforms that might have different memory bus widths. In the discussions of this work, we will generally assume that this width is set to 512 bits, since the platforms that Fletcher currently supports both provide memory bus interfaces of this size. However, Fletcher can also operate on wider or narrower bus interfaces.

4 Vendor-agnostic hardware libary

Fletcher aims to be vendor-agnostic in order to thrive in an open-source setting. All designs are based on data streams. This requires custom streaming primitives that can perform the basic operations on streams. Commercial tools contain IP cores to support some (but not all) of these operations as well. However, to engage with an open-source oriented community, it is important to not force designs to use vendor-specific solutions. This causes the need for a custom streaming operations library that is maintained alongside Fletcher.

The most important streaming components are discussed in this subsection. The most basic primitives on which all other components are built, are as follows:

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.

The throughput requirement mentioned in the previous section dictates that streams must be able to deliver multiple elements per cycle (MEPC). To support this, and other operations, the previously mentioned primitives are extended by the set of following stream operators:

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

Reshaper A component that absorbs an arbitrary number of valid elements of an MEPC stream and outputs another arbitrary number of elements. This element is useful for serializing wide streams into narrow streams (or vice versa, parallelizing narrow streams into wide streams). The element 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 (especially the Arbiter and Buffer), a light-weight bus infrastructure has been developed to allow multiple masters to use the same memory interface. This bus infrastracture is similar to (and includes wrappers for) AXI-4, supporting independent read/write request and data channels and bursts.

Read/Write Arbiter Arbitrates multiple masters onto a single slave.
Read/Write Buffer Allows buffering of at least a full maximum sized burst to relieve the arbiter of any back-pressure.

5 Components to match Arrow abstractions

5.1 Implementation alternatives

Designing an interface to Arrow data could follow different approaches. A flexible approach would have a small customized soft processor generate the requests based on a schema or some bytecode that is compiled on the host. In this way, any schema (reasonably limited in size) could be requested, and schemas can be changed during run-time.

However, this approach would have several drawbacks. First of all, it would introduce more latency as it takes multiple instructions to calculate addresses and generate requests. Moreover, as developers can create schemas with fixed-width types of arbitrary length, allocating streams for the "widest" case is impractical. If one would supply the implementation with support for some very wide fixed-width type (effectively limiting the schemas that can be expressed already), it would cause a relatively large amount of area overhead for schemas with narrow primitives. For example, consider a hard-coded 1024-bit stream of which some schema only uses one bit. As schema data can be of many varieties, the streams would require run-time reordering of the elements coming from bus words. This involves relatively expensive parametrizations of the Stream Reshaper to support all possible cases of aligning arbitrary elements. Elements themselves must be restricted to be smaller than 1024 bits and only a fraction of RAM spent on FIFOs in the data paths is effectively used.

The aggregate of these drawbacks causes the proposed interface generation framework to completely configure the generated interface during compile-time. For this purpose, we introduce highly configurable components that correspond to abstractions seen in the Arrow software-language specific counterparts.

5.2 Buffers

Readers As explained in Section 2, Arrow *buffers* hold C-like arrays of fixed-width data. We 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 fixed-width type MEPC streams. It performs the following functions:

- Based on the properties of the bus interface and the data type, perform the pointer arithmetic to locate elements of interest in the Arrow buffer.
- Perform all the bus requests desired to obtain a range of elements.
- Align received bus words.
- Reshape aligned words into MEPC streams with fixed-width data types.

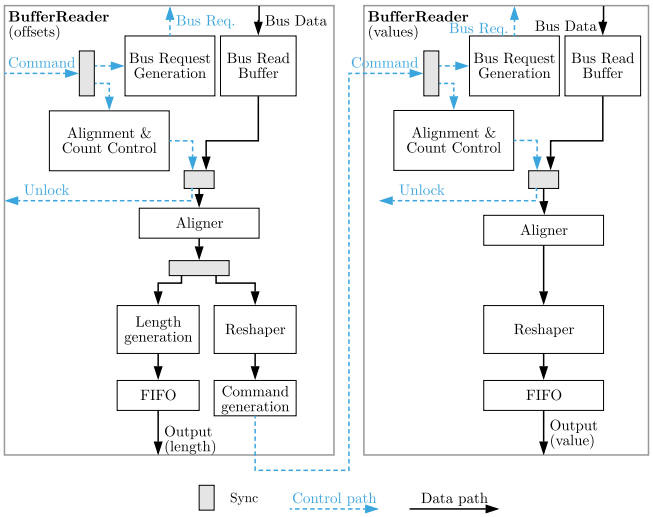


Fig. 2: A BufferReader for an offsets buffer (left) and a values buffer (right)

An architectural overview of the proposed implementation of two BRs (in combination providing a setup to read variable-length types) is shown in Fig. 2.

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) A MEPC stream of data corresponding to an Arrow data type.

Reading from a values buffer, and reading from validity bitmap buffers (by instantiating a BR with element size one) is supported by the rightmost configuration of the BR as shown in Fig. 2.

Here, a command stream is absorbed by two units: a bus request generation unit and an alignment and count controller. The bus request generator performs all pointer arithmetic and generates bus burst requests. The alignment and count controller calculates, based on the width of the bus and the type of elements, how much a bus word must be shifted (especially for the first bus word received), since some first index in the command stream might point to any element in a buffer. It also generates a count of valid items in the MEPC stream resulting from alignment. This is also useful when last bus words in a range contain less elements than requested.

Even though first and last bus words might not be aligned or do not contain all requested elements, after aligning and augmenting the stream with a count, the reshaper unit will shape a non-full MEPC stream into a full MEPC stream.

Furthermore, when the last bus word has been streamed to the aligner, an unlock stream handshake is generated to notify the accelerator that the command has been completed in terms of requests on the bus.

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 substracted 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.

Command (out) Used to generate commands for other buffers. This is useful when this BR reads from an Arrow offsets buffer.

Writers Complementary to BRs, we also implement BufferWriters (BW) that, given some index range can write to memory in the Arrow format. They contain the same interface streams as BR, except the data flow is inverted. An architec-

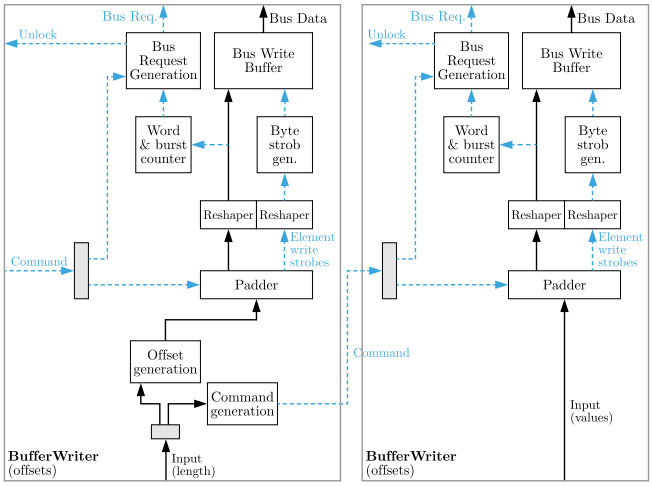


Fig. 3: A BufferWriter for an offsets buffer (left) and a values buffer (right)

tural overview of the proposed implementation of two BW is observed in Fig. 3. Writing to a validity bitmap buffer or a values buffers requires the buffer writer to operate as follows (as seen on the right side of the figure).

When a command is given to the BW, the MEPC input stream is delivered to a unit that pre- and post-pads the stream to force the stream to be aligned

with a minimum bus burst length parameter. Furthermore, it generates appropriate write strobes (only asserting strobes for valid elements). The elements and strobes are then reshaped to fit into a full bus word and sent to a bus write buffer. Note that sometimes it is unknown how long an input stream will be when the command is given. Therefore the command to the BW supports both no range or with range commands. At the same time this requires counting accepted bus words into the BusBuffer. A bus request generation unit uses this count to generate bus requests preferably when full bursts are ready, but if the input stream has ended, bus words are bursted out with minimum burst steps until the buffer is empty.

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.

5.3 Arrays

To support Arrow *arrays*, that combine multiple buffers to deliver any field type that may be found in an Arrow schema, we implement special components called ColumnReaders and Writers.

These ColumnReaders- and Writers instantiate the BRs and BW resulting from a schema field. They furthermore support:

- Attaching command outputs of offsets buffers to values or validity bitmap buffers.
- Arbitration of multiple buffer bus masters onto a single slave.
- Synchronization of unlock streams of all buffers in use.
- Recursive instantiations of themselves. This, in turn, supports:
  - Nested types, such as `Lists<List<Type>>`.
  - Adding an Arrow validity bit to the output stream.
  - Support Arrow *structs*, such as `Struct<List<Int16>, Float>`.

The ColumnReaders and ColumnWriters are supplied with a configuration string that conveys the same information as an Arrow schema. 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. 1 is shown in Fig. 4. Reading from the example RecordBatch (corresponding to the schema) will require three ColumnReaders. The manner in which they are recursively instantiated is shown in the figure. Here one can discern four types of ColumnReader configurations:

Default A default ColumnReader only instantiates a specific ColumnReader 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 ColumnReaders.

Prim A ColumnReader instantiating a BR for fixed-width (primitive) types.

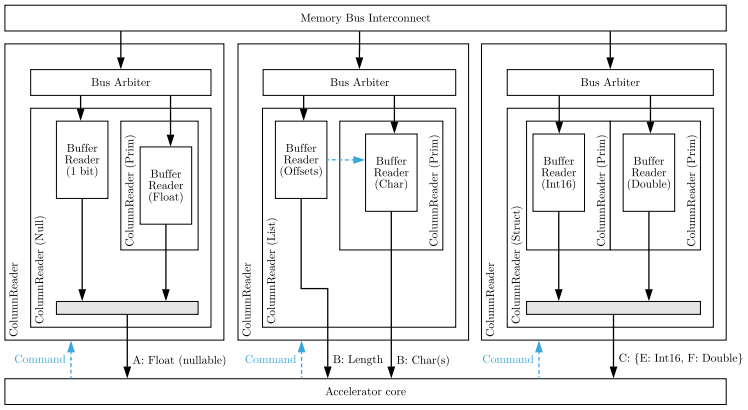


Fig. 4: Resulting ColumnReader configuration from the Schema in Fig. 1

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

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

Through the List and Struct type ColumnReaders, nested schemas may be supported. On the top level all streams that interface with the accelerator core are concatenated. A software tool named *Fletchgen* generates top levels for various platforms (including AWS EC2 F1 and OpenPOWER CAPI SNAP) that wraps around the ColumnReaders and ColumnWriters and splits the streams that are concatenated onto single signals vectors into something readable (using the same field names as defined in the schema) for the developer. A discussion of the inner workings of *Fletchgen* and the support for these platforms is outside the scope of this paper but the implementation may be found in the repository online [6]. The complement (in terms of data flow) of this structure is implemented for ColumnWriters. One additional challenge to ColumnWriters 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.

5.4 Continuous integration

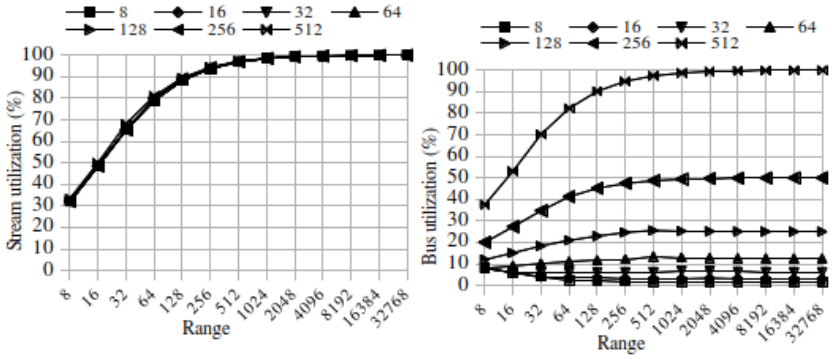
All parts of Fletcher are open sourced. This allows all interested parties to submit changes to the hardware design. Part of improving the maintainability of the project includes bootstrapping of the build and test process in a continuous integration framework, where the simulator used is also an open-source project [2]. By using fully open-sourced tools in the collaborative development process, the threshold to get started with FPGA accelerators and Fletcher is lowered.

6 Results

6.1 Functional validation

Because the number of schema field type combinations is virtually infinite (through nesting), it is not trivial to validate the functionality of the framework. To obtain good coverage in simulation, a Python script is used to generate random schemas with supported types. The types decrease in complexity the deeper their nesting level, such that at some point the nesting ends with a primitive type. The resulting buffers are deduced from the schema, random content is generated and a host memory interface is mimicked. Random indices are requested from the simulated ColumnReaders, and their output streams are compared to the expected output. In this way, the correct functioning of over ten thousand different generated structures was validated.

6.2 Throughput



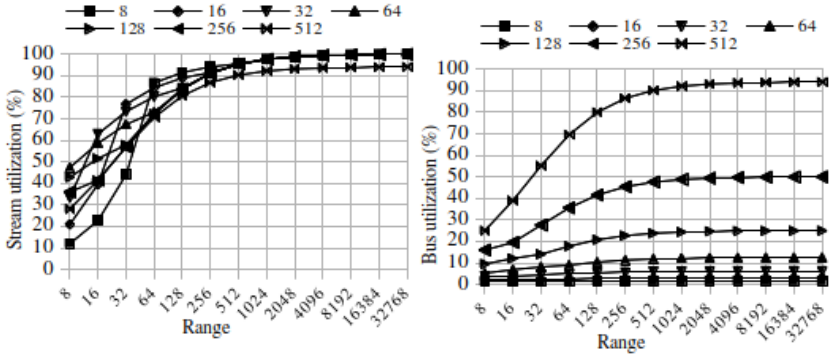
(a) Near-optimal output stream utilization

(b) Memory bus pressure

Fig. 5: Utilization for a ColumnReader for various fixed-width types versus command range (each line represents a different fixed-width type).

ColumnReaders/Writers for fixed-width types The main goal of the hardware components of Fletcher is to provide the output streams with the same bandwidth as the system bandwidth, if the accelerator core can consume it. In other words, the generated interfaces should not throttle the system bandwidth because of a sub-optimal design choice (like a sub-optimal in-memory format or a sub-optimal hardware component).

We simulate the throughput of ColumnReaders and ColumnWriters, assuming that we have a perfect bus interconnect, i.e. the bus delivers/accepts the requested bursts immediately and at every clock cycle a valid bus word can



(a) Near-optimal output stream utilization

(b) Memory bus pressure

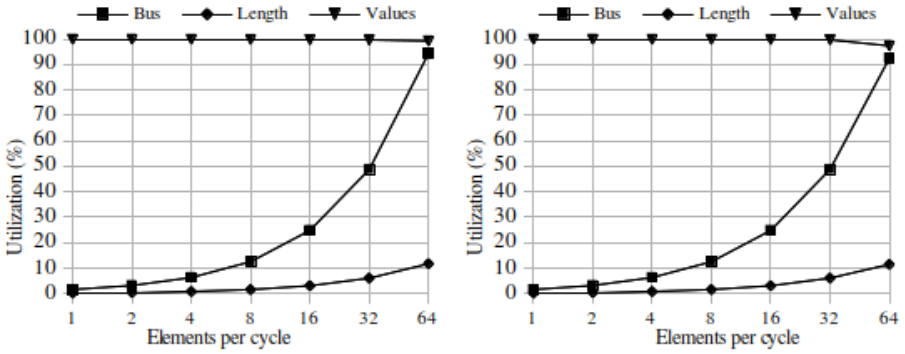
Fig. 6: Utilization for a ColumnWriter for various fixed-width types versus command range.

be produced. We measure the bus utilization and stream output utilization (in handshakes per cycle during the processing of a command) for different fixed-width types, as a function of the range of Arrow array entries requested through the command stream. We furthermore assume the accelerator core can handshake the ColumnReader output or ColumnWriter input stream every cycle. The results of this simulation for a data bus width of 512 bits (as both platforms, AWS EC2 F1 and OpenPOWER CAPI SNAP, that Fletcher currently supports use this memory bus width) are shown in Fig. 5b and Fig. 5a, where the bus utilization and output stream utilization is shown, respectively, for various fixed-width types. Similar measurements for the ColumnWriters are seen in Fig. 6.

Initialization overhead and latency of both the ColumnReader and ColumnWriter is present when the command only requests a short range of entries. However, once the range grows larger (a likely scenario in most big data use cases where massively parallel operators on data sets such as maps, reductions and filters are applied), the stream utilization becomes near optimal. As long as the element width is smaller than the bus width, maximum stream throughput is achieved, and as long as the element width is equal to the bus width, maximum bus bandwidth is achieved. We may conclude that a ColumnReader for fixed-width types does not create a bottleneck if the accelerator core can absorb data at the system bandwidth rate. A developer using a ColumnReader can now express access to an Arrow Array in terms of RecordBatch indices and will receive the exact data type as specified through the schema on the stream, without degradation of the system bandwidth.

ColumnReaders/Writers for variable-length types We simulate the throughput of a ColumnReader/ColumnWriter for an Arrow Array where the items in the Array are lists of primitive types. We choose the type to be a character (8 bits). We generate random lists between length 1 and 1024 and, in Fig. 7,

plot the utilization of the bus and the input/output streams as function of the elements-per-cycle parameter of this ColumnReader/ColumnWriter. From these figures, we may observe that the value stream utilization is near-optimal, independent of the number of elements per cycle that it is configured for; as long as the memory bus can deliver the throughput, the accelerator core is fed at maximum throughput.



(a) ColumnReader (b) ColumnWriter

Fig. 7: Bus and input/output stream utilization for an increasing elements-per-cycle parameter demonstrating utilization near 100%.

6.3 Area utilization

Tab. 2: Area utilization statistics for a Xilinx XCVU9P device

Type	Resource	W=8	W=16	W=32	W=64	W=128	W=256	W=512
Column Reader Prim(W)	CLB LUTs	0.30%	0.28%	0.26%	0.24%	0.22%	0.20%	0.21%
	CLB Registers	0.20%	0.20%	0.20%	0.20%	0.22%	0.24%	0.26%
	Block RAM (B36)	0.65%	0.65%	0.65%	0.65%	0.65%	0.65%	0.65%
	Block RAM (B18)	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%
Column Reader List of Prim(W)	CLB LUTs	2.34%	1.81%	1.46%	1.32%	1.03%	1.04%	0.78%
	CLB Registers	1.01%	1.01%	1.01%	1.01%	1.00%	1.00%	1.00%
	Block RAM (B36)	1.30%	1.30%	1.30%	1.30%	1.30%	1.30%	1.30%
	Block RAM (B18)	0.09%	0.09%	0.09%	0.09%	0.09%	0.09%	0.09%
Column Writer Prim(W)	CLB LUTs	0.20%	0.19%	0.19%	0.20%	0.20%	0.22%	0.23%
	CLB Registers	0.28%	0.28%	0.28%	0.28%	0.29%	0.31%	0.33%
	Block RAM (B36)	0.37%	0.37%	0.37%	0.37%	0.37%	0.37%	0.37%
	Block RAM (B18)	0.02%	0.02%	0.02%	0.02%	0.02%	0.02%	0.02%
Column Writer List of Prim(W)	CLB LUTs	1.03%	0.97%	0.91%	0.87%	0.80%	0.78%	0.52%
	CLB Registers	1.18%	1.12%	1.11%	1.11%	1.06%	1.06%	0.73%
	Block RAM (B36)	1.11%	1.11%	1.06%	1.06%	1.06%	1.06%	0.74%
	Block RAM (B18)	0.07%	0.05%	0.07%	0.07%	0.07%	0.07%	0.05%

For the same memory bus width as the supported platforms (512 bits), we synthesize ColumnReaders and ColumnWriters for various fixed-width types (W=8,16,...,512) and for various variable-length types (W=8 with EPC=64, W=16 with EPC=32, etc.) for a Xilinx XCVU9P device (that used in AWS EC2 F1 instances). The area utilization statistics are shown in Tab. 2.

The ColumnReaders/Writers require little area. Most configurations utilize less than one percent of the resources. Interestingly, ColumnReaders/Writers for small elements require more LUTs than wider elements on a wide bus. This is due to the reshaper and aligner units discussed in Section 5, requiring aligning and reshaping more MEPC stream element count combinations, increasing mux sizes. Designers may chose to reduce this number in the ColumnReaders and Writers themselves, but this requires an asymmetric connection to the memory bus interconnect, effectively moving the alignment functionality to the interconnect. Register usage increases when element size increases, since register slices on the path to the accelerator core match the width of the elements. Block RAM usage is the same for all configurations, because this depends on the maximum burst length that has been fixed to 32 beats for all configurations.

7 Related work

While Arrow is not the only framework following the trend of in-memory computation for big data frameworks (an overview can be found in [13]), it is a framework that is especially focused on providing efficient interoperability between different tools/languages. This allows the 11 languages supported by Arrow to quickly and efficiently transfer data to the FPGA accelator using Fletcher.

Several solutions to abstract away memory bus interfaces are commercially available and integrated into HLS tools (such as Xilinx' SDAccel and Intel's FPGA SDK for OpenCL). However, they have no inherent support for nested types that Arrow schemas can represent, and usually work well only with simple, C-like primitive types and arrays. Loading data from nested structures involves pointer traversal and arithmetic which HLS tools do not deal with efficiently [11]. At the same time, after Fletcher generates an interface that delivers streams which HLS tools can operate on very well.

State-of-the-art frameworks to integrate FPGA accelerators with specific databases exist [5], although interface generation specific to the schema data type and serialization overhead are not discussed.

8 Conclusion

The goal of the Fletcher framework is to ease integration of FPGA accelerators with data analytics frameworks. To this end, Fletcher uses the Apache Arrow in-memory format to leverage the advantages of the Arrow project, including no serialization overhead and interfaces to 11 different high-level languages. To support the wide variety of data set types that Arrow can represent, and to convert these data sets into hardware streams that are desirable by an FPGA

developer, this work has presented a bottom-up view of a library of vendor-agnostic and open-source components. These components allow reading from tabular Arrow data set columns, by providing a range of table indices, rather than byte addresses, to refer to records stored in the tables. Fletcher is effective at generating these interfaces without compromising performance. It takes very little area to create an interface that provides an accelerator core with system bandwidth for any configuration of the Arrow data set. Fletcher significantly simplifies the process of effectively designing FPGA-based solutions for data analytics tools based on Arrow.

References

1. Amazon Web Services: AWS EC2 FPGA Hardware and Software Development Kits (2018), https://github.com/aws/aws-fpga
2. Gingold, T.: Ghdl vhdl 2008/93/87 simulator (2018), https://github.com/ghdl/ghdl
3. McKinney, W.: Python for data analysis: Data wrangling with Pandas, NumPy, and IPython. " O'Reilly Media, Inc." (2012)
4. OpenPOWER foundation: CAPI SNAP Framework Hardware and Software (2018), https://github.com/open-power/snap
5. Owaida, M., Sidler, D., Kara, K., Alonso, G.: Centaur: A framework for hybrid cpu-fpga databases. In: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). pp. 211–218 (April 2017). https://doi.org/10.1109/FCCM.2017.37
6. Peltenburg, J., van Straten, J.: Fletcher: A framework to integrate Apache Arrow with FPGA accelerators. (2018), https://github.com/johanpel/fletcher
7. Peltenburg, J., van Straten, J., Wijtemans, L., van Leeuwen, L., Al-Ars, Z., Hofstee, H.: (UNDER REVIEW) Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow. In: 2019 56th ACM/ESDA/IEEE Design Automation Conference (DAC) (June 2019)
8. Sidler, D., István, Z., Owaida, M., Alonso, G.: Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 403–415. SIGMOD '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3035918.3035954
9. The Apache Software Foundation: Apache Arrow (2018), https://arrow.apache.org/
10. The Apache Software Foundation: Apache Parquet (2018), https://parquet.apache.org/
11. Winterstein, F., Bayliss, S., Constantinides, G.A.: 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 (Dec 2013). https://doi.org/10.1109/FPT.2013.6718388
12. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache Spark: A Unified Engine for Big Data Processing. Commun. ACM **59**(11), 56–65 (Oct 2016)
13. Zhang, H., Chen, G., Ooi, B.C., Tan, K.L., Zhang, M.: In-memory big data management and processing: A survey. IEEE Transactions on Knowledge and Data Engineering **27**(7), 1920–1948 (2015)