Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators

Peltenburg, Johan; Hadnagy, Ákos; Brobbel, Matthijs; Morrow, Robert ; Al-Ars, Zaid

DOI
10.1109/ICFPT52863.2021.9609833

Publication date
2021

Document Version
Accepted author manuscript

Published in
2021 International Conference on Field-Programmable Technology (ICFPT)

Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators

Johan Peltenburg*, Ákos Hadnagy*, Matthijs Brobbel†, Robert Morrow‡ Zaid Al-Ars*
*Accelerated Big Data Systems, Delft University of Technology, Netherlands, Contact: j.w.peltenburg@tudelft.nl
†Teratide B.V., Netherlands ‡Sigmax.ai Inc., U.S.A.

***Abstract*—JSON is a popular data interchange format for many web, cloud, and IoT systems due to its simplicity, human readability, and widespread support. However, applications must first parse and convert the data to a native in-memory format before being able to perform useful computations. Many big data applications with high performance requirements convert JSON data to Apache Arrow RecordBatches, the latter being a widely-used columnar in-memory format for large tabular data sets used in data analytics. In this paper, we analyze the performance characteristics of such applications and show that JSON parsing represents a bottleneck in the system. Various strategies are explored to speed up JSON parsing on CPU and GPU as much as possible. Due to performance limitation of the CPU and GPU implementations, we furthermore present an FPGA accelerated implementation. We explain how hardware components that can parse variable-sized and nested structures can be combined to produce JSON parsers for any type of JSON document. Several fully integrated FPGA-accelerated JSON parser implementations are presented using the Intel Arria 10 GX and Xilinx VU37P devices, and compared to the performance of their respective host systems; an Intel Xeon and an IBM POWER9 system. Result show the accelerators achieve an end-to-end throughput close to 7 GB/s with the Arria 10 GX using PCIe, and close to 20 GB/s with the VU37P using OpenCAPI 3. Depending on the complexity of the JSON data to parse, the bandwidth is limited by the host-to-accelerator interface or available FPGA resources. Overall, this provides a throughput increase of up to 6x, compared to the baseline application. Also, we observe a full system energy efficiency improvement of up to 59x more JSON data parsed per joule.**

***Index Terms*—JSON, parsing, Apache Arrow, FPGA, accelerator**

I. Introduction

Today, many systems exchange large amounts of data using a human-readable format described in the JavaScript Object Notation (JSON) Data Interchange Standard. These systems include web and cloud services, non-relational databases, Internet-of-Things devices, and many others. A consuming system has to deserialize JSON data before it can be used, which can take up a significant amount of time and energy, sometimes causing bottlenecks in systems that require high performance in terms of throughput, latency, or energy efficiency.

Recent state-of-the art contributions have shown to squeeze out the last bit of CPU performance possible for JSON parsing using SIMD extensions [1]. Yet, many applications still require the parsing step to be performed with even higher throughput and/or lower latency. Considering that nowadays FPGA accelerators are readily available in data centers worldwide, we explore parsing JSON data with FPGA accelerators to provide high performance and energy-efficiency [2].

In this paper, we focus on a real-world application that receives a large amount of JSON-formatted data. The application converts the JSON data to Apache Arrow inter-process communication (IPC) messages, which is a format that is immediately usable by downstream consuming processes without parsing. Finally, these messages are published to the distributed streaming platform Apache Pulsar [3].

We consider two realistic fully integrated use-cases with data following a simple model and a more complex model. We then contribute the following:

- We present and measure several approaches to achieve the best possible software performance for our use cases, as a fair point of comparison for the FPGA accelerator implementation.
- We present a vendor-agnostic open-source FPGA implementation of parsing components for various JSON value types.
- We present and measure parsing and conversion throughput of the FPGA accelerator within a fully integrated application setup working on the two use-cases.
- We show that within the context of our application, the FPGA accelerated implementation can achieve close to 20 GB/s of parsing and conversion throughput when the accelerator host-to-device interface provides enough bandwidth and when enough FPGA resources are available.

In Section II, we briefly introduce the JSON format, the Apache Arrow in-memory format to which the JSON data needs to be converted, and we discuss related work. Section III gives an overview of the application and explores the run-time profile for the two use-cases. The architecture of the FPGA accelerator design and details of the parser implementation are described in Section IV. Section V presents measurements of the accelerated solution. We conclude the paper in Section VI.

II. Background

A. JSON

JSON is a human-readable format for data exchange or storage. Data serialized in the JSON format is typically called a JSON document. Once a JSON document is successfully parsed, its DOM (Document Object Model) is known. The DOM is a tree that represents the logical structure of the

data. Each node of the tree represents one of the 7 possible types of JSON values: objects with members that are key-value pairs, arrays, strings, numbers, boolean values ”true”, ”false”, and the value ”null” indicating there is no data. An example of a JSON document is shown in Listing 1. The structural characters `{` and `}` are used to represent object boundaries. The members of an object consist of key names delimited by quotation marks (`"`) placed before the structural character `:` (colon), and values placed after the colon. Keys and string values are enclosed in `"` (quotes). Array values are separated by `,` (comma).

```
{ "id": 11,
  "message": "Hi FPT!",
  "read": false,
  "meta": {
    "refs": [42, 1337],
    "tag": null
  }
}
```

Listing 1: An example of a JSON document.

A JSON document is a form of human-readable serialized data that is typically exchanged between two software systems. Due to its human-readable nature, it is not known to be the most compact data serialization format, yet it is so ubiquitous, well-supported, and simple, that it is used even in systems exchanging massive amounts of information.

Software applications consuming JSON documents typically use libraries to parse them, of which there are many (at the time of writing, https://www.json.org shows a non-exhaustive list of 172 JSON parser implementations across 61 software languages). The most typical use case of such libraries is to either provide a way to construct JSON documents from native representations of objects (serialization), or to parse JSON documents and navigate their DOM to reconstruct native representations of (parts of) objects (deserialization) before continuing to operate on the data. More advanced JSON parsers allow pushing filter predicates into the parser to operate on the data while it is being parsed [4].

B. Apache Arrow

For the application presented in this article, we are interested in deserialization of JSON documents. The application takes a JSON document and converts it to the Apache Arrow in-memory format [5]. Arrow is a project aiming to unify the in-memory representation of data sets found in data analytics applications across multiple software languages. This provides a means to exchange information between heterogeneous software systems without making copies in memory. It is (being) integrated in widely used (big) data analytics frameworks such as Apache Spark [6], Pandas [7], Dask [8], Dremio [9], and Polars [10].

The Arrow in-memory format is a hardware-friendly format, allowing to store tabular data sets in a columnar-oriented fashion (rather than row-oriented) in an abstraction called a

Schema fields	**Name**	id	message	read	meta	
					refs	tag
	Type	uint64	string	bool	list(uint64)	string
Data		11	Hi FPT!	false	42, 1337	∅
		404	Beans	true	11	coffee
		...	...	...	...	...

Fig. 1: Example Arrow RecordBatch. The first row contains the data from the JSON example in Listing 1.

RecordBatch. In Figure 1, an example of an Arrow RecordBatch is shown, where the first row holds the same data as the JSON document shown in the example of Listing 1. The names and data types of the values in each column of a RecordBatch are described by the *schema* of the RecordBatch.

In a random-access memory, each column consists of a typically small number of large contiguous buffers holding raw data such as integers, UTF8 characters, or, in the case of variable-sized lists, offsets into other buffers that are part of the column. This allows to traverse a minimum amount of pointers to be able to access data randomly and in parallel.

The Arrow format furthermore allows accelerating many operations by SIMD instructions due to its contiguous nature. As a mental exercise, one can imagine having to sum all values by some constant in a table column, where the values of interest are placed between values from other columns (row-oriented). Now consider the column-oriented approach, where the values in a column are placed contiguously in a buffer. Here, it is easy to accelerate the sum operation using SIMD by loading multiple values from the column into an SIMD register at once, while this is not practical for the row-oriented approach.

Arrow also defines a data serialization format for inter-process communication (IPC) of RecordBatches, which, apart from some metadata in the header of an IPC message, holds the same column-oriented data as in a RecordBatch. This prevents the need to make copies of the data to some other in-memory format once it has arrived in the memory of the receiver.

C. Related work

We emphasize the need of our application to convert JSON documents to Apache Arrow RecordBatches. The Arrow libraries already provide JSON parsing functionality that is of interest to our application. Internally, the official Arrow C++ implementation uses *RapidJSON* [11], one of the more popular and faster JSON parsing implementations.

The state-of-the-art CPU implementation for JSON parsing is *simdjson* [1]. The implementation makes use of the SIMD extensions of x86, ARM and POWER processors. *simdjson* applies specific byte and bit-level manipulations on consecutive raw JSON bytes in SIMD vectors in order to speed up lexical analysis (including UTF8 validation) and parsing.

Few JSON parsing FPGA accelerator implementations have been published in academic literature. Recent work on the *Fleet* language for streaming hardware designs shows a JSON-related application as an example design [12]. This part of the work focuses on extracting byte strings of object field values according to some schema. These values still need to

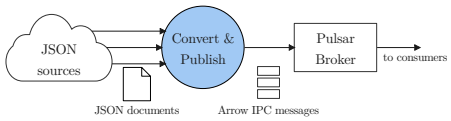


Fig. 2: Application context

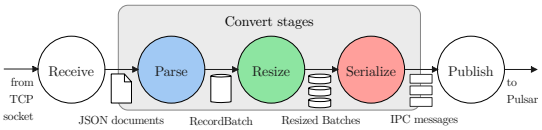


Fig. 3: Application stages

be converted, e.g. to two's-complement binary, before being able to use them in some downstream process. In addition, for the specific object model of the JSON parsing experiment, the work reports a throughput of 21.39 GB/s. However, this is the throughput with the accelerator card on-board DDRs as source and sink; the data is not communicated from/to the host system, which is typically required by applications.

In a recent survey paper [13], the potential for FPGA accelerators in non-relational database systems is discussed, where the JSON format is widely used as well. The authors also express the lack of academic literature of JSON parsing implementations in FPGA. At the same time, the work provides a good overview of FPGA implementations related to (and sometimes parsing) Extensible Markup Language (XML) documents, that share some similarities to JSON documents. In the XML space, previous work explored performing queries on XML documents, where the queries involve computing only on specific parts of the XML DOM values [14]. Filtering uninteresting parts of the XML DOM is accelerated by FPGA through the use of *skeleton automata*, allowing run-time reconfiguration of the filter functionality; a technique that could be used in future work of our JSON parsing accelerator implementation.

III. Application

A. Overview

In this paper, a design is shown of an FPGA accelerator for JSON parsing that immediately writes the deserialized data into host memory in the Apache Arrow columnar in-memory format. The accelerator operates in a real-world application context, shown in Figure 2. Here, many sources send JSON documents via TCP to the application, followed by a convert and publish service, for this article considered to run on a single server. The application accepts the TCP connections, receives the JSON documents, deserializes the JSON data, turns it into Apache Arrow formatted data, serializes it into Arrow IPC messages, and publishes them to a broker instance of the distributed low-latency streaming platform Apache Pulsar (which runs as a separate process either on the same or an external server). The application and its sources are freely available [15].

The application consists of five stages, also shown in Figure 3:

```
{ "voltage": [1337, 1024, 768, 384, 42] }
```

Listing 2: Example of the simple JSON use-case

```
{ "timestamp":"2021-03-09T13:37:42Z",
  "odometer":80201,
  "hypermiling":true,
  "avgspeed":69,
  "sec_in_band":[14,22,28,205,110,261,302,260,
                 220,100,30,10],
  (+7 other fields omitted for brevity) }
```

Listing 3: Partial example of the complex JSON use-case

A Receive - handles incoming connections and places incoming JSON documents into a set of buffers, separated by newline characters.

B Parse - parses JSON documents from the buffers and deserializes data into an Arrow RecordBatch.

C Resize - The Arrow RecordBatch is potentially split up into multiple Arrow RecordBatches, not to exceed the maximum message size of Pulsar messages.

D Serialize - The Arrow RecordBatches are serialized into Arrow IPC messages.

E Publish - The IPC messages are published to a Pulsar *topic* via a Pulsar *broker* instance.

The receive and publish stages are handled by third-party programs (respectively the operating system and the Pulsar client library), placing these stages out of scope of acceleration. We therefore focus on the middle three stages, which we call the conversion stages. These three stages are multi-threaded.

B. Use-cases

We study the application and potential for FPGA acceleration in the context of two use-cases, where both schemas do not change for months or even years while the system operates. For the first use-case, the application receives JSON documents with voltage values of batteries in an IoT environment. The JSON documents have only one field that is a variable-length array of integers. An example is shown in Listing 2.

For the second use-case, the application receives JSON documents from motorized vehicles with usage statistics, which is more complex. Each document consists of twelve fields; two boolean fields, four integer fields, one string field and five arrays of integers. As an example, a JSON with some of those fields is shown in Listing 3.

C. Run-time profile

To get a clear overview of which stages form bottlenecks in the full pipeline, we profile the application thoroughly by varying the number of JSONs to parse, as well as the number of threads used to perform the parsing.

Figure 4 shows the average run-time of each of the convert stages of 8 repetitions of the experiment. Here, we observe that irrespective of the total input size of JSON data bytes, the parse

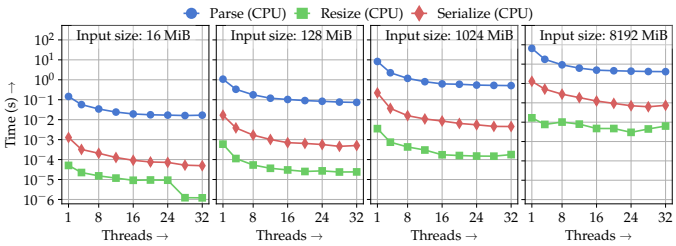


Fig. 4: Baseline run-times of application processing stages on an Intel Xeon Silver 4215R CPU.

stage dominates the total run-time of the conversion stages, taking roughly one to two orders of magnitude longer than serialization and resizing. The implementation of this stage uses Arrow's JSON parser library of the C++ implementation, which uses *RapidJSON* internally.

Resizing is expected to be fast, since for Arrow RecordBatches it is a zero-copy operation (only objects managing metadata have to be reconstructed in memory, but the actual bulk data stays in the same memory location without being copied in order to construct smaller batches). Serialization does involve a copy of the data, but only of a few contiguous buffers, since, in the Arrow IPC format, no compression or other type of encoding of the data is required. Observe that for the smallest input size, as the number of threads increases, the resizing time drops significantly because the multi-threaded implementation schedules the JSON documents over the available threads, effectively resizing at the input of the convert stages.

D. Parsing

The application profile exposes the parsing stage to be a bottleneck. Since Arrow uses *RapidJSON* internally while *simdjson* claims better performance, we investigate speeding up the parse stage in software first as much as possible, before moving to an FPGA accelerated solution. To this end, we tried the following approaches:

1) *RapidJSON* — A previously discussed framework, which is used by Arrow internally. We use *RapidJSON* in a stand-alone manner to parse JSONs specific to our schemas of interest only.

2) *simdjson* — Currently regarded as the state-of-the-art software framework to quickly parse JSON data. We use it in a similar manner to *RapidJSON*.

3) *Boost Spirit.X3* — A parser-combinator framework [16], which we use to construct parsers specifically for the schemas of interest (not the whole JSON language). The motivation to include *Boost Spirit.X3* is that we take an approach similar to our hardware accelerated solution (presented in Section IV), where we aim to (automatically) create schema-specific JSON parsers. These parsers do not parse the whole JSON language to construct a traversable DOM tree, but only a subset specific to an expected schema, and construct an Arrow RecordBatch. The expectation for this approach is that by constructing recursive descent parsers for only a schema-specific sub-set, faster implementations can be realized. We stipulate that some

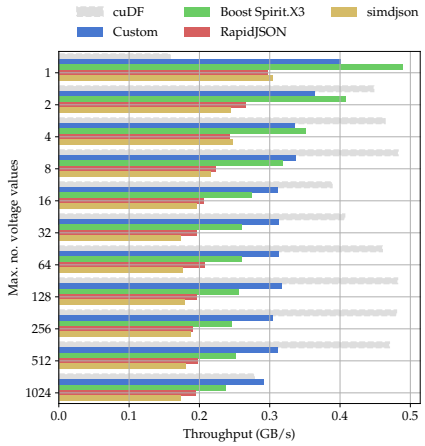


Fig. 5: Throughput of various JSON parsing strategies for CPU and GPU.

assumptions about the input are made for this implementation. First, it assumes the input only contains well-formed UTF8 characters, and we do not need to validate UTF8 correctness. Second, we assume the order of object fields is fixed and predefined, such that fields cannot appear in a seemingly random order. These two assumptions are not made by *RapidJSON* and *simdjson*, and are likely to impact performance in favor of this parser.

4) Custom — A completely hand-written and optimized recursive-descent solution with the same assumption as the *Boost Spirit.X3* parser.

5) *cuDF* — A relatively new CUDA dataframe library as part of the RAPIDs GPU acceleration framework that also works with Apache Arrow data sets in memory [17]. At the time of writing, the framework only supports primitives and strings as JSON member values, and therefore incorrectly infers JSON arrays with integers (which are used heavily throughout our use-cases) as strings.

The motivation to create the *Boost Spirit.X3* and custom hand-optimized solution is that the widely-used (production-ready) JSON parsing frameworks *RapidJSON* and *simdjson* include many features that are required by its general use in the field (e.g. a generic API to walk the DOM tree of any type of JSON document at run-time which is inherently schema-less), and by production environments (e.g. UTF8 validation). While such features are paramount, we consider that they are not necessary to study the highest CPU performance possible in a controlled experimental environment. We also tried parser-generator frameworks such as ANTLR4 [18], but these implementations only provided several megabytes per second of throughput. These frameworks are not designed to deliver high performance, but to be highly flexible for language front-end design (e.g. performing lexing and parsing to build up easily manipulable abstract syntax trees).

In Figure 5, we show measurements of the single-threaded throughput (or fully utilized GPU throughput) of the mentioned parsing implementations for two schemas on an Intel Xeon Silver 4215R CPU running at 3.20 GHz (which is one of our target systems for FPGA acceleration as well). For all

approaches, we created several variants that e.g. pre-allocate output buffers rather than growing them dynamically, and use different APIs provided by the frameworks that have different performance characteristics. For clarity and brevity, we report only the fastest variants for each of the approaches (while the sources of the other variants are publicly available [19]).

For the simple schema, we vary the maximum number of values that the "voltage" array holds. When the maximum number of values is low, more time is spent in processing keys and structural characters (e.g. "voltage" and {) than in values, and vice versa when the number of values is high.

The measurements show that the throughput of *simdjson* and *RapidJSON* is similar and does not vary much with the maximum number of values. We find *RapidJSON* to be slightly faster than *simdjson* for our specific purpose, which is likely due to the difference between the event-driven API of the former, which better fits our use-case, and the DOM-walking API of the latter, that builds a DOM tree in memory first before allowing the DOM to be traversed and building up the Arrow RecordBatch to take place.

When mainly parsing structural characters in case the maximum number of array values is low, *Boost Spirit.X3* provides almost two-thirds additional throughput over *simdjson* and *RapidJSON*, although the improvement is only one-third when the maximum number of array values is high. The custom implementation performs best when the number of voltage values is eight or higher, becoming slightly faster than the *Boost Spirit.X3* implementation. To get an idea of the GPU-accelerated *cuDF* performance, we report our measurement using an NVIDIA Tesla T4 GPU, although, as explained, the output has an incorrect schema, yet the JSON document structure is still parsed. *cuDF* parses arrays of integers as strings, and did not yet convert these text-based integers to binary integers, which will require additional processing.

Concluding, the custom implementation is on average the fastest functionally correct implementation of a JSON-to-Arrow converting parser that we could produce. Although it is unlikely to be used in a production-ready system, we include it in our experiments for evaluation within a controlled environment. For our use cases, *simdjson* on average does not provide an improvement over *RapidJSON*. Since Arrow uses *RapidJSON* internally, we consider the Arrow implementation to be the baseline for evaluation as the best production-ready implementation.

IV. FPGA ARCHITECTURE AND IMPLEMENTATION

A. System architecture

Designing a JSON parsing FPGA accelerator knows several challenges and opportunities. For clarity, we reiterate the context in which our proposed system operates: the system must deserialize many JSONs with the same DOM to the Arrow in-memory format for a long period of time. Therefore, the system does not require parsing of JSON documents with different (perhaps at compile-time unknown) DOMs without reconfiguration of the architecture implemented by the FPGA. This provides the opportunity to construct the JSON parsing circuit at compile time.

One challenging aspect is that both JSON and Arrow allow nested and variable-length data such as objects holding other objects and arrays of arbitrary size. Communicating such data between hardware components requires an interface specification that supports such constructs. Industry-standard solutions such as e.g. *AXI4-Stream* and *Avalon-ST* only focus on variable-length streams of bytes, and do not describe how nested structures should be signaled from source to destination. To solve this challenge, we use Tydi [20], which provides a specification for streaming dataflow interfaces between hardware components that transfer complex (e.g. with nested records, unions, etc.) and dynamically-sized data structures over hardware streams.

Once Tydi streams of parsed JSON data are available, another challenge is that the data has to be transferred to host memory (in the Arrow format) for downstream processing. To communicate between the accelerator and host memory, we use Fletcher [21]. Fletcher generates DMA engines based on schemas of RecordBatches that on the one side can read and write from random-access memory, and on the other side provides streams for accelerator components to work with. These streams are (with a small amount of glue logic) compatible with Tydi streams.

We contribute support to Fletcher for the accelerator platform of one of our systems, the Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA [22] (which we will abbreviate as Intel PAC henceforth), to the open-source community [23]. The platform is based on the Open Programmable Acceleration Engine (OPAE) project.. The Intel PAC card is connected to the host system using PCIe x8 Gen3.

The second system on which we evaluate the accelerator is an Inspur Power Systems FP5290G2 with two POWER9 CPUs and an Xilinx VU37P on an AlphaData ADM-PCIE-9H7 accelerator card which (despite its name) is connected using OpenCAPI 3. The accelerator has direct access to the shared virtual memory of the host system. We use the *OC-Accel* project and Fletcher to interface between the accelerator kernel and host CPU.

A high-level overview of the accelerator architecture and the host application is shown in Figure 6. From the network, the TCP client receives raw JSON data and places it in one of multiple buffers (residing in pinned huge pages for the Intel PAC-based design) that are for both systems directly accessible by the accelerator, managed by the converter threads. Since the data is fully streamable through the design, it would also be possible to stream in data through e.g. a TCP offload engine, directly from a network interface that many contemporary commercial FPGA accelerator cards have. When a converter encounters a buffer that is not empty, it starts one of the multiple available hardware parsers through the control path provided by Fletcher, the OPAE/OC-Accel libraries/drivers, the OPAE/OC-Accel and Fletcher shells and the generated interconnect and DMA engines specific to the schema of the output RecordBatch(es). The parser first requests all bytes of

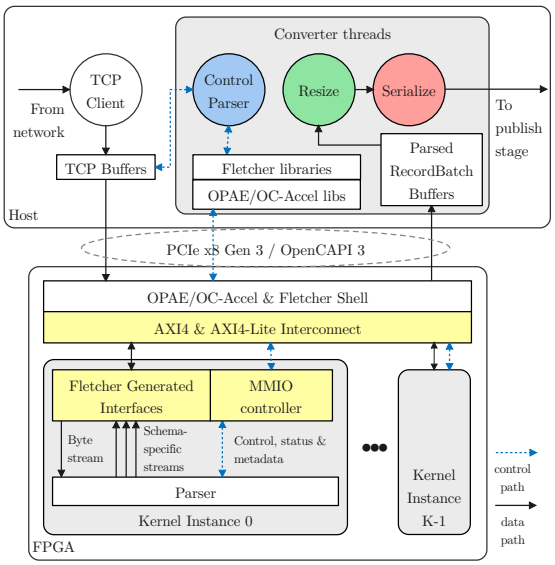


Fig. 6: Heterogeneous architecture showing how the FPGA accelerator is integrated in the application pipeline. AXI4 data path width is 1024 bits for OpenCAPI, and 512 bits for OPAE.

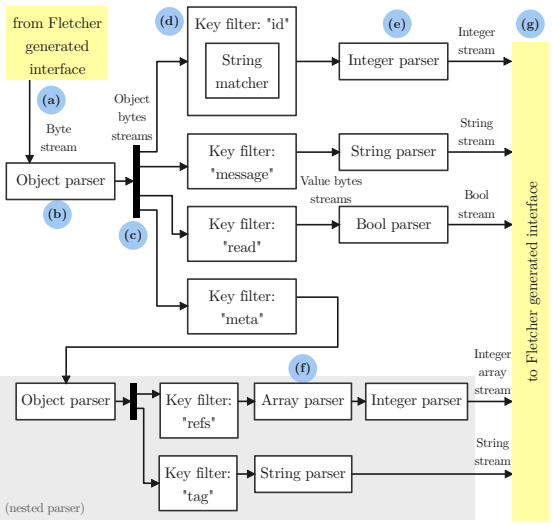


Fig. 7: Example parser design for JSONs from Listing 1

the input buffer. It then parses and converts the raw bytes to Tydi streams corresponding to the Arrow schema of the destination RecordBatch. These schema-specific streams are consumed by the Fletcher-generated interface and placed in the CPU's main memory over the PCIe or OpenCAPI interfaces.

B. JSON parser construction

To construct FPGA-based JSON parsers corresponding to specific Arrow schemas, a modular approach is taken. This way, it becomes easy to construct or even automatically generate JSON parsers. Figure 7 shows the architecture of a parser implementation for JSON documents that follow a DOM of Listing 1. The figure explains how basic components can be combined to construct a parser for specific JSON DOM / Arrow schema. We explain each of the components and their interfaces with this example, as annotated in Figure 7:

(a) A stream of raw bytes enters the parser, with an additional signal marking the last byte from the input buffer.

(b) The *object parser* component expects an object to be delimited by the structural character `{` at the start, and by `}` at the end. All bytes in between, the object content bytes, are passed onto the output stream, together with signals marking whether the byte is part of the key or the value part of a field. It also provides a signal for the last byte in the field value, the last byte in the whole object, and the last byte from the input buffer. We will continue to name these signals the *last* signals, which are inherently supported by Tydi interfaces.

(c) A stream synchronizer duplicates the object content bytes stream with the object contents for each of the four object members of the example schema.

(d) The object contents streams are filtered by *key filter* components, one for each expected object member. To do so, the key filter first attempts to match the expected key string (e.g. `"id"`). Once this key appears on the input stream, all subsequent bytes after the structural character `:`, the value bytes, are passed to the output, until the structural character `,` is encountered (which denotes the end of the member's value). The output stream again signals the end of the value, but also passes on the *last* signals from the upper levels of the hierarchy. When a different key appears on the input stream, the component simply discards the bytes. Note that it therefore does not matter what the order of the fields in an object is.

(e) The streams of value bytes are passed on to subsequent parser, depending on the expected value type. For example, the *integer parser* converts a sequence of numerals to its two's-complement binary representation (e.g. a 64-bit integer). The *string parser* simply strips the outer quotation marks (`"`) and adds a signal denoting the end of a string.

(f) The *array parser* component is used to parse array values. It strips the array delimiters `[` and `]`, the element separator `,` and adds signals to denote the end of an element in addition to the *last* signals from the upper levels of the hierarchy.

(g) Finally, streams for all JSON values for all object members appear at the output of this schema-specific parser. With the addition of a small amount control logic, these streams are made compatible to work with the Fletcher-generated interface. From this point onward, the Fletcher-generated interface takes care of placing the data as an Apache Arrow RecordBatch in the main memory of the host CPU. Using the *last* signals from all levels of the hierarchy, the control logic knows when to close a stream flowing into the Fletcher generated interface, such that Fletcher may make the RecordBatch available to the downstream software processes.

In the manner of the example given above, we constructed accelerator designs for both the simple and the complex schema.

The parsers are able to absorb multiple bytes per handshake at the input stream, which can happen every cycle. All implementations currently use a maximum of eight bytes per

handshake, which gives each parser a peak theoretical input throughput of 1.6 GB/s when running at 200 MHz. In practice, this throughput is often not reached, as some transfers can hold multiple JSON values at the same time. For example, a single transfer on the input byte stream could hold three array values: `10,20,30`. The output of a parser component delivers at most one value per transfer, causing back-pressure of the input (for two cycles in this example).

Note that the design is a dataflow design that does not require intermediate results to be written/read to/from memory. This means no on-board accelerator memory is needed, and the performance will mostly depend on the host-to-device interface bandwidth, which can be saturated as long as there are FPGA resources available to increase the number of parsers.

C. Schema size and resource utilization considerations

For the complex schema, we experienced a high utilization initially, because the Fletcher-generated interface becomes relatively large for each kernel instance, as for each field a separate DMA engine is generated. Naively duplicating the kernel to achieve more parallelization yielded designs with too high utilization already after one instance (e.g. with ALM utilization already exceeding 45% for the Arria 10 GX for a single kernel instance). To deal with this problem, for the complex schema, we multiplex the output streams of multiple parser instances onto a single bundle of streams to the Fletcher generated interface, such that for N fields and P parser instances, we do not have to place and route $N * P$ DMA engines, but only N. These N DMA engines provide ample bandwidth to keep up with the input streams. In terms of throughput, one DMA engine for one single field already provides enough bandwidth to prevent back-pressure on the JSON byte input stream of a parser when it is operating at its peak throughput, even for small integer or boolean values.

V. Results

A. Throughput

We measure the end-to-end throughput (JSON bytes parsed per second) of the parsing stage of the application. The end-to-end throughput is the throughput as seen by the software application, which includes data movement from the host main memory to the accelerator, and vice versa. We vary the amount of total JSON input bytes since, sometimes there will be a large workload, but often there will be a small workload, so we are interested in both cases. We also vary the amount of threads (up to the system maximum hardware threads), and report the end-to-end throughput. The number of values in the JSON array fields are uniformly random, picked between 1 and some maximum, where a maximum of 1, 8, 64, and 512 array values was measured. These measurements are aggregated and the average throughput over all maximums is reported. We present an overall summary of the results in Table I, where the mean throughput is calculated over the best performing configuration (in terms of CPU threads or FPGA parser instances) for each implementation and each input size.

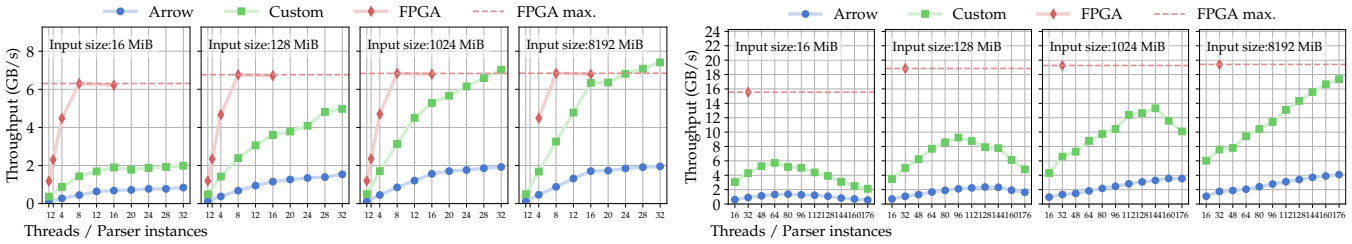
	System	Intel Xeon + Arria 10 GX		POWER9 + VU37P	
	Schema	Simple	Complex	Simple	Complex
System power (J/s)	Idle	218.00	218.00	299.00	299.00
	Arrow	427.47	423.94	611.26	641.92
	Custom	406.48	402.18	651.39	633.15
	FPGA	274.40	245.22	385.96	320.91
Mean throughp. (GB/s)	Arrow	1.68	1.64	2.93	2.70
	Custom	5.60	5.07	11.84	9.77
	FPGA	6.70	5.12	18.37	10.10
Speedup	vs. Arrow	3.99	3.12	6.27	3.74
	vs. Custom	1.20	1.01	1.55	1.03
Energy eff. (MiB/J)	Arrow	7.64	7.60	8.94	7.51
	Custom	28.33	26.26	32.05	27.90
	FPGA	113.30	179.43	201.42	439.85
Energy improv.	vs. Arrow	14.83	23.61	22.53	58.58
	vs. Custom	4.00	6.83	6.28	15.77
Peak throughp. (GB/s)	Arrow	2.10	2.11	4.11	2.70
	Custom	7.81	7.04	17.60	14.69
	FPGA	6.85	5.49	19.42	10.60
FPGA no. parsers		8	6	32	8
FPGA util. (%)	ALM/CLB	39	67	47.28	54.94
	BRAM	30	42	62.8	29.66
	DSP	0	0	0	0

TABLE I: Summary of performance, power consumption, energy efficiency, and resource utilization.

In Figure 8, the measurements for the simple schema of Listing 2 are shown. From the figure, it can be concluded that on the Intel Xeon + Arria 10 GX setup, the accelerator saturates the PCIe x8 Gen3 bandwidth already when eight hardware parsers are instantiated. Increasing the number of parser instances to sixteen does not help to increase the throughput. In both the case of the Intel system, the accelerator outperforms the baseline Arrow implementation, but does not always outperform the experimental custom implementation when the thread count is high and the workload is large. Still, considering all input sizes, the mean throughput of the FPGA accelerator has better throughput than the experimental custom implementation. For all systems, it outperforms the Arrow baseline by $4\times$ to $6\times$, also shown in the summary of Table I.

In Figure 9, the measurements for the complex schema of Listing 3 are shown. Here, for the Intel system, the PCIe link is not completely saturated yet, but are limited by the maximum number of parsers that can be instantiated in hardware while closing timing. On the POWER9 system, that has a higher amount of threads available compared to the Intel system, we may conclude that spawning many threads only makes sense if the workload is high enough, since some overhead is associated with managing all threads, and performance drops when spawning too many. Otherwise, similar observations can be made, i.e. the mean throughput of the FPGA accelerator is higher by around $3.5\times$ versus the Arrow baseline, and approximately equal to the custom implementation.

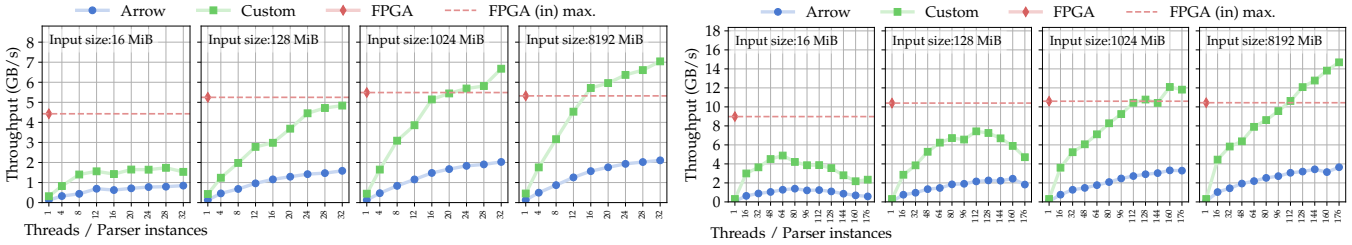
Overall, we stipulate that while the CPU implementation can achieve a throughput close to that of the FPGA implementation, for both systems, it requires the use of all available hardware threads of both CPU sockets of these machines. Offloading the JSON parsing to the FPGA accelerator will free up these threads such that they can be used to e.g. perform downstream analysis on the deserialized data or run an instance of a Pulsar broker.



(a) Intel Xeon + Arria 10GX

(b) POWER9 + VU37P

Fig. 8: Throughput for simple schema



(a) Intel Xeon + Arria 10GX

(b) POWER9 + VU37P

Fig. 9: Throughput for complex schema

B. *Energy efficiency*

We also measure the full system power consumption using the Data Center Manageability Interface that both systems support. In Table I, we show the power consumption, the aforementioned mean performance and energy efficiency for all implementations and schemas. To calculate the energy efficiency, we use the power consumption difference between the full system being idle and under load. For both systems and schemas, we find the FPGA accelerated solution to perform very well in terms of energy efficiency. We observe an increase in energy efficiency of $14\times$ to $59\times$ over the baseline Arrow implementation, and $4\times$ to $16\times$ over the experimental custom implementation. The highest energy efficiency improvement is achieved for the complex schema, where the accelerator is controlled by one CPU thread only. This is because the parser output stream multiplexing to the Fletcher-generated DMA engines, that from an API perspective deliver just a single Arrow RecordBatch, and are therefore controlled by a single thread.

C. *Resource utilization*

Table I also shows the resource utilization of the FPGA accelerator for the respective schemas and devices. Overall, it seems that more parser instances may fit, since resource utilization of the main resources used varies around 50% for most implementations. However, in our design, we often ran into timing issues due to local congestion, which is something that can be further investigated in the future. In this way, more parsers may fit, although as explained before, enough host-to-accelerator interface bandwidth must be available to be able to deliver enough data to work on. Other interesting approaches with the remaining resources would be to reduce the latency of the overall application by directly connecting the accelerator card to the network, and by using a TCP offload engine to stream the TCP payload with JSON data directly into the parsers.

VI. Conclusion

We investigated an application that deserializes large amounts of JSON data and converts it to the Apache Arrow in-memory format before publishing it to the Pulsar streaming platform. By profiling the application, we found JSON parsing to be the bottleneck, being one to two orders of magnitude slower than subsequent stages in the application pipeline. We presented multiple approaches to accelerate this stage of the application, including multiple JSON parsing frameworks and custom recursive descent parsers for CPUs, and a GPU acceleration framework. We furthermore thoroughly explained how FPGA accelerators can be used to parse and convert JSON data to Apache Arrow data, by using modular streaming dataflow components for various JSON value types that can be combined in a way that corresponds to the expected schema of the JSON data. The FPGA accelerator design was implemented for an Intel Arria 10 GX and a Xilinx VU37P FPGA, hosted by an Intel Xeon and POWER9 server, respectively. Results show that the parsing and conversion throughput of the FPGA accelerator is between $3\times$ and almost $6\times$ higher, with an energy efficiency increased of between $14\times$ to $59\times$ compared to the baseline implementation that is based on *RapidJSON*. A challenge that remains is how to deal with large schemas that may cause the accelerator design to grow beyond available FPGA resources. In conclusion, the presented FPGA accelerator provides an interesting alternative to traditional methods of JSON parsing in terms of throughput and energy efficiency.

References

[1] G. Langdale and D. Lemire, "Parsing gigabytes of JSON per second," *The VLDB Journal*, vol. 28, no. 6, pp. 941–960, Dec 2019. [Online]. Available: https://doi.org/10.1007/s00778-019-00578-5

[2] K. Neshatpour, M. Malik, M. A. Ghodrat, A. Sasan, and H. Homayoun, "Energy-efficient acceleration of big data analytics applications using FPGAs," in *2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 115–123.

[3] "Apache Pulsar - distributed pub-sub messaging system," https://github.com/apache/pulsar, 2021.

[4] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann, "Mison: A fast json parser for data analytics," *Proc. VLDB Endow.*, vol. 10, no. 10, p. 1118–1129, Jun. 2017. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.14778/3115404.3115416

[5] "Apache Arrow - A cross-language development platform for in-memory analytics," https://arrow.apache.org, 2021.

[6] 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, p. 56–65, Oct. 2016. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/2934664

[7] W. McKinney *et al.*, "Pandas: a foundational Python library for data analysis and statistics," *Python for high performance and scientific computing*, vol. 14, no. 9, pp. 1–9, 2011.

[8] Matthew Rocklin, "Dask: Parallel Computation with Blocked algorithms and Task Scheduling," in *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra, Eds., 2015, pp. 126 – 132.

[9] "Dremio - the missing link in modern data," https://github.com/dremio, 2021.

[10] R. Vink, "Polars - Fast multi-threaded DataFrame library in Rust and Python," https://github.com/pola-rs/polars, 2021.

[11] Tencent and M. Yip, "RapidJSON - A fast JSON parser/generator for C++ with both SAX/DOM style API," https://rapidjson.org/, 2021.

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

[13] J. Dann, D. Ritter, and H. Fröning, "Non-Relational Databases on FPGAs: Survey, Design Decisions, Challenges," *arXiv preprint arXiv:2007.07595*, 2020.

[14] J. Teubner, L. Woods, and C. Nie, "Skeleton automata for fpgas: Reconfiguring without reconstructing," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 229–240. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/2213836.2213863

[15] Teratide, "Bolson - A JSON-to-Arrow IPC conversion and Pulsar publish tool," https://github.com/teratide/bolson, 2021.

[16] J. de Guzman and H. Kaiser, "Boost Spirit X3," https://www.boost.org/, 2018.

[17] "cuDF - GPU DataFrame Library," https://github.com/rapidsai/cudf, 2021.

[18] T. J. Parr and R. W. Quong, "ANTLR: A predicated-LL(k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250705

[19] Johan Peltenburg, "Experiment sources accompanying the paper High-performance FPGA acceleration of JSON parsing," https://github.com/johanpel/j2a, 2021.

[20] J. Peltenburg, J. Van Straten, M. Brobbel, Z. Al-Ars, and H. P. Hofstee, "Tydi: An Open Specification for Complex Data Structures Over Hardware Streams," *IEEE Micro*, vol. 40, no. 4, pp. 120–130, 2020.

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

[22] Intel, "Intel Programmable Acceleration Card (PAC) with Intel Arria 10 GX FPGA Data Sheet," https://www.intel.com/content/www/us/en/programmable/documentation/hhf1507759304946.html, 2020.

[23] Teratide B.V., "Fletcher-OPAE - Fletcher Open Programmable Acceleration Engine platform support." https://github.com/teratide/fletcher-opae, 2021.