Battling the CPU Bottleneck in Apache Parquet to Arrow Conversion Using FPGA

Peltenburg, Johan; Van Leeuwen, Lars T.J.; Hoozemans, Joost; Fang, Jian; Al-Ars, Zaid; Hofstee, H.Peter

DOI
10.1109/ICFPT51103.2020.00048

Publication date
2021

Document Version
Accepted author manuscript

Published in
2020 International Conference on Field-Programmable Technology (ICFPT)

Citation (APA)
Peltenburg, J., Van Leeuwen, L. T. J., Hoozemans, J., Fang, J., Al-Ars, Z., & Hofstee, H. P. (2021). Battling the CPU Bottleneck in Apache Parquet to Arrow Conversion Using FPGA. In *2020 International Conference on Field-Programmable Technology (ICFPT)* (pp. 281-286). Article 9415570 IEEE. https://doi.org/10.1109/ICFPT51103.2020.00048

Battling the CPU Bottleneck in Apache Parquet to Arrow Conversion Using FPGA

Johan Peltenburg*, Lars T.J. van Leeuwen*, Joost Hoozemans*, Jian Fang*†, Zaid Al-Ars*, H. Peter Hofstee*‡
*Accelerated Big Data Systems, Delft University of Technology, Netherlands, Contact: j.w.peltenburg@tudelft.nl
†National Innovation Institute of Defense Technology, China
‡IBM, USA, Contact: hofstee@us.ibm.com

***Abstract*—In the domain of big data analytics, the bottleneck of converting storage-focused file formats to in-memory data structures has shifted from the bandwidth of storage to the performance of decoding and decompression software. Two widely used formats for big data storage and in-memory data are Apache Parquet and Apache Arrow, respectively. In order to improve the speed at which data can be loaded from disk to memory, we propose an FPGA accelerator design that converts Parquet files to Arrow in-memory data structures. We describe an extensible, publicly available, free and open-source implementation of the proposed converter that supports various Parquet file configurations. The performance of the converter is measured on an AWS EC2 F1 system and on a POWER9 system using the recently released OpenCAPI interface. A single instance of the converter can reach between 6 and 12 GB/s of end-to-end throughput, and shows up to a threefold improvement over the fastest single-thread CPU implementation. It has a low resource utilization (less than 5% for all types of FPGA resources). This allows scaling out the design to match the bandwidth of the coming generation of accelerator interfaces. The proposed design and implementation can be extended to support more of the many possible Parquet file configurations.**

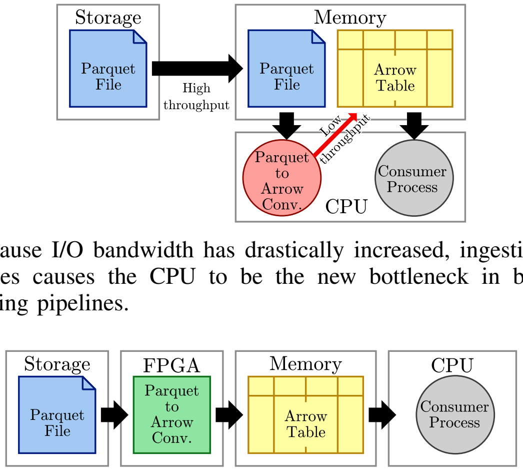
***Index Terms*—FPGA, Accelerator, Apache Parquet, Apache Arrow**

I. Introduction

In the context of big data analytics, the bandwidth associated with reading data from persistent storage is increasing rapidly due to the availability of non-volatile memory solid-state drives (e.g. NVMe SSDs). In the past, database system were often designed with the assumption that *CPUs are fast* and *I/O is slow*. However, this relationship is turning around over recent years. CPUs are no longer able to parse, decompress, and deserialize files at data rates close to I/O bandwidth, sometimes lacking over an order of magnitude in performance.

To improve the performance of analytics systems, we propose performing part of the decompression and deserialization of files to in-memory data structures with an FPGA accelerator (as shown in a contextual overview of this work in Figure 1). FPGA accelerators provide the following benefits within this context.

First, it is possible to place the FPGA on the data path from storage to memory. Commercial FPGA accelerator cards with interfaces to SSDs are readily available today from various vendors. Second, FPGA systems can implement specialized data-flow designs with deeply pipelined datapaths. This allows FPGAs to provide high performance and energy-efficiency [1].



(a) Because I/O bandwidth has drastically increased, ingesting Parquet files causes the CPU to be the new bottleneck in big data processing pipelines.

(b) To alleviate the bottleneck, a heterogeneous system with an FPGA accelerator is proposed, where the FPGA accelerator performs an ingress transform of the stored file.

Fig. 1: FPGA acceleration of the Parquet-To-Arrow converter

In this paper, we contribute an open source design of an FPGA accelerator that takes files with large tabular data structures encoded in the Apache Parquet file format as input. It provides a streaming interface to optionally insert a decompression engine such as a Snappy or GZip IP core [2] [3]. The accelerator converts these files into tables according to the Apache Arrow format in memory.

II. Background

A. Related Work

Previous research has acknowledged CPU processes to become the new bottleneck in big data processing pipelines, because I/O bandwidth is increasing [4][5][6][7][8]. An analysis of this problem specific to Parquet and ORC, and a proposal of an improved format is presented in [9]. The format results in a lower compression rate, but the implementation is not freely available or widely used at the time of writing. In more recent work [10], the bottleneck is acknowledged, and FPGA-based solutions are provided at the level of the file system itself. The limitation also holds for network I/O, relevant to this paper in case distributed file chunks are shuffled, which is discussed in [11]. Previous work on reducing data duplication explored a specific combination of FPGA accelerators and Apache Parquet files [12].

B. Apache Parquet

Parquet [13] is a storage format intended to store large tabular data structures in a column-oriented format, often used in distributed environments. Each Parquet file has a complex hierarchical structure described by metadata in the footer of the Parquet file. This metadata describes the data types of the columns, and what compression and encoding schemes are used. The data itself is divided over *row groups*, containing one chunk of each column in the table, useful for distributed storage systems. The size of these row groups can be set when writing the Parquet file to allow for longer sequential reads in the same column chunk. The columnar format can be advantageous, e.g. only the relevant columns required by some computational transformation need be accessed without having to decode irrelevant columns. Column chunks are in turn divided into *pages*. Each page is compressed according to a specific compression codec, and its values are encoded using a specific encoding scheme. The locations of the pages and column chunks are found in the file footer. Every page can be independently decompressed and decoded, such that every page can be randomly accessed and processed in parallel.

C. Apache Arrow

Typical to the big data framework ecosystem, Parquet is used in the context of a wide variety of software languages and run-times. When implementing a fast converter from Parquet files to in-memory data structures, it must be decided what (software) language and run-time engine will be at the consuming end of the data. The choice for a specific language, e.g. C++, rules out immediate use in another language, e.g. Python, unless one would perform the tedious work of implementing wrappers and/or serializers/deserializers. Fortunately, the Apache Arrow project provides a *common data layer*, where the common in-memory representation of large tabular data structures is the same for any of the 11 supported languages [14]. The project furthermore provides language-specific APIs to access the data [14]. Applications in any programming language supported by Arrow may immediately benefit from an accelerated implementation of the conversion.

Furthermore, an FPGA accelerator framework built on top of Arrow exists, called Fletcher [15][16], and is used in this work. Fletcher generates DMA engines with streaming dataflow interfaces to and from Arrow *RecordBatches* in memory. In Arrow, RecordBatches are column-oriented tabular data structures, typically containing a large amount of data records. The interfaces are generated based on Arrow *schemas* — descriptions of the data types of the values in the columns of the RecordBatches.

III. Design and implementation

A. Input Data Structure

Because the typical size of row groups is in the order of hundreds of MBs to multiple GBs, the overhead of parsing row group or even column metadata on a CPU and performing host-to-accelerator communication is relatively small. Pages, however, are in the order of megabytes (the default is one MiB, although they can be chosen to be much larger). Therefore, we chose to implement page processing fully in hardware. This includes metadata parsing as well as decompressing and decoding the actual data.

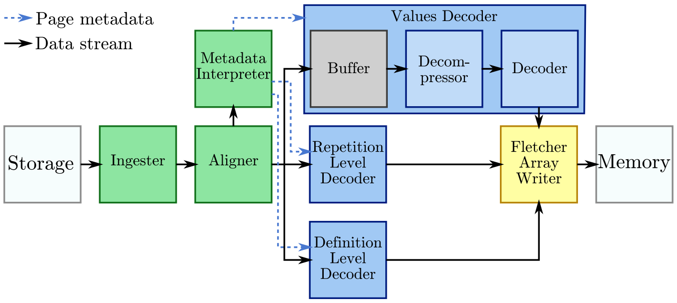


Fig. 2: Architectural overview of the proposed accelerator. Control flow is omitted for clarity.

A Parquet page itself consists of four variable-length sections. First, a header with page metadata, serialized according to Apache Thrift's Compact Protocol. Second and third, blocks containing the so called repetition and definition levels, used for nested data types (e.g. lists of lists) and/or nullable types. The last block contains the actual values. Because values of columns are stored contiguously and have the same data type, encoding techniques such as (among others) delta encoding with binary packing are used. They can also be compressed with codecs such as Snappy [17] and gzip.

B. Architecture

We propose the top-level architecture of the Parquet-to-Arrow converter as shown in Figure 2. The converter should perform its function in a streaming fashion, at a throughput close to the I/O bandwidth on either side of the accelerator. That is, either the SSD interface or accelerator to host memory interface. For contemporary and near-future systems, this is in the order of tens of GB/s.

The design consists of the following components. The Ingester, Aligner and Metadata Intepreter are always the same and required to convert any Parquet file. The implementation of the Values Decoder, Repetition Level Decoder and the Definition Level Decoder depend on the compression and encoding scheme used by the file.

Ingester: The Ingester initiates the loading of pages from memory or storage in large bursts. It produces two streams with raw bytes, and initial alignment information within the raw byte stream.These streams are fed into the Aligner component.

Aligner: The Aligner implements a pipelined barrel shifter to align the raw bytes for the next stages. Because one of the three variable-length blocks within a page may be aligned differently, but could start within a streamed word of the previous block, the Aligner holds the unaligned words in a history buffer to be able to immediately restart the pipeline for the next page, without having to request the data again from memory or storage. The downstream components report back the amount of used bytes to provide the necessary control information for this functionality.

Metadata Interpreter: Parsing the metadata involves a complex state machine, because it must implement the used features of the Apache Thrift serialization protocol. This protocol uses dynamic features, such as variable-length integers, causing the metadata interpreter to absorb one byte per cycle. Because the page metadata is only a fraction of the total page data, the overhead of this relatively low-throughput process is negligible. After interpreting the header, the compressed and uncompressed size of the page and the number of values are known and streamed to the appropriate parts of the design.

Fletcher ArrayWriter: The Fletcher ArrayWriter is a component generated by the Fletcher framework, serving as a DMA engine that can write from hardware streams to in-memory arrays of complex data structures (e.g. nested lists) formatted by the Apache Arrow format specification. It must be noted that in this context, Arrow arrays are not C-like arrays, but can consist of multiple buffers holding data with specific relations expressed through the Arrow type. We feed the various streams emerging from the decoding of the values, and the repetition and definition levels, into the ArrayWriter. In turn, it will write the data into memory in the Arrow format.

Values Decoder: The internals of the Values Decoder depend on the data types used in the column, since that influences what encoding schemes can be used. Furthermore, the values can be compressed, and therefore must be decompressed for reading.

Decompressor: When compression is used, the Values Converter should contain a decompressor component.with a streaming interface, such as e.g. can be found in an open-source implementation of Snappy [18] (performing up to ≈ 8 GB/s) or GZip [19]. For files that are uncompressed, the decompressor may simply pass through the data stream.

Decoders: The presented prototype supports the following datatypes:

(A) Plain encoding of fixed-size primitives.
(B) Bit-packed delta encoding of fixed-size primitives.
(C) Mixed encoding of UTF8 strings.

In case A, the raw byte representation of the mentioned data types is used.

In case B, an initial value is given and then for each value only the difference (delta) with respect to the previous value is stored in bit-packed encoding. When the deviation between values is low, the deltas can be encoded with a small number of bits.

In case C, Parquet supports a string format that stores sequences of strings as (bit-packed, delta encoded) lengths and (plain) characters separately within a page.

Delta Decoder: To decode bit-packed delta-encoded values to raw values (used in aforementioned cases B and C), we implement the *Delta Decoder* component, also shown in Figure 3, to be used within the Values Decoder.

It consists of a Delta Header Reader, responsible to read metadata related to the delta encoded values, such as the initial value. After parsing, the Delta Header Reader aligns the input stream to the start of a block of values. Each block contains more metadata that is parsed by the Block Header Reader.After parsing the Header of such a block, the stream is again aligned to the first delta encoded value. Through a component called Bit Unpacker, delta values are finally fed into the Delta Accumulator. This unit performs the final parallel prefix sum on the initial value, minimum delta and unpacked deltas to obtain the actual values.

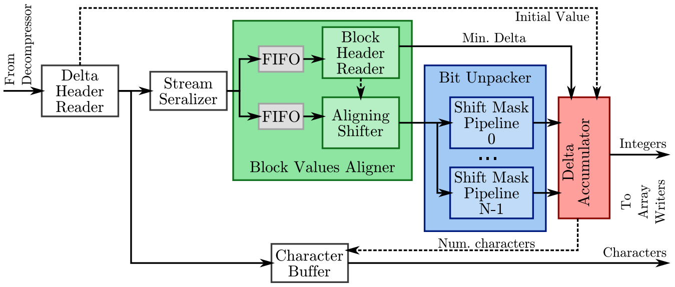


Fig. 3: Delta decoder

IV. Results

A. Experiment setup

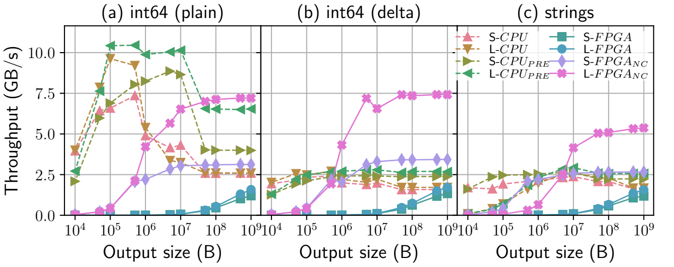


Fig. 4: AWS EC2 F1 throughput versus Arrow RecordBatch output size

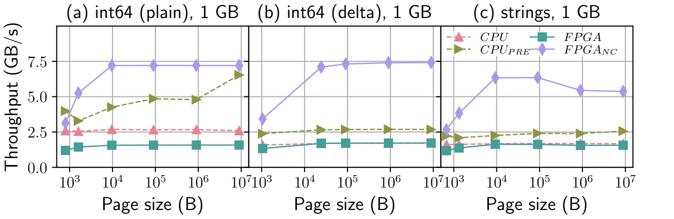


Fig. 5: AWS EC2 F1 throughput versus Parquet page size

The Parquet-To-Arrow converter is implemented on two platforms; the Amazon EC2 F1 platform using an Intel Xeon E5-2686 v4 CPU and a Xilinx XCVU9P FPGA (hereafter *F1*), and an Inspur FP5290G2 with a dual-socket POWER9 Lagrange 22-core CPU and OpenCAPI interface to an ADM-PCIE-9H7 with a Xilinx XCVU37P (hereafter *OpenCAPI*). The FPGA implementation of the F1 system runs at 250 MHz, while the FPGA implementation of the OpenCAPI system runs at 200 MHz. The implementation is publicly available, free, and open-sourced, including all benchmarks performed to reproduce the result in this section [20].

The FPGA implementation in the F1 system requires the Parquet file to be copied from host memory to on-board memory, because it can only access the on-board DDR memories of the accelerator card. During this transfer, the system may perform other tasks, if they can be overlapped. We therefore present two flavors of measurements for the F1 system. In the first, denoted by $FPGA$, we measure the end-to-end solution, where the Parquet file starts in host memory and ends up decoded as an Arrow RecordBatch in host memory again (i.e. a full round-trip). In the second, we measure *no copy* time, denoted by $FPGA_{NC}$. Here, we measure only the FPGA processing time with the Parquet file already in the on-board DDR memory and the Arrow RecordBatch ending up in the on-board DDR memory as well.

The OpenCAPI system allows to access host memory directly. This transparency closely mimics the operation of a storage-attached FPGA (which would be an ideal candidate for this design).

To obtain the absolute best result on CPU, it was necessary to re-implement the Parquet subset supported by the FPGA implementation in `C++` and compile it using `GCC` with `-Ofast`. Our `C++` implementation outperformed the existing software implementations for this subset, providing us with the fastest CPU implementation.

We create a second `C++` implementation where the virtual memory pages[1] of the Arrow buffers are touched, to make sure the TLB is 'warm', consequently removing the overhead of a 'cold' TLB from the measurements. These measurements are denoted as CPU_{PRE}. For both the CPU implementation and for the FPGA implementation, we measure the performance of one thread and one kernel, respectively.

We measure three combinations of data type and encoding, from the cases described in the previous section: *int64 (plain)*, *int64 (delta)*, and *strings*. Data for *int64 (plain)* was randomly generated. Data for *int64 (delta)* was randomly generated, but with a random modulo that changes every 256 elements. This modulo creates a mix of data requiring different packing lengths, instead of almost always requiring the full width as is expected for fully random data. The *strings* were randomly generated with random lengths between 1 and 12 characters. Note that the length of the strings determines the mix between delta-packed and plain data, and very long strings will result in behavior more similar to that of *int (plain)*.

B. Performance

1) Throughput vs. RecordBatch size: We first measure the throughput versus the size of the resulting Arrow RecordBatch, shown in Figure 4. The figures display two Parquet page sizes; small pages, prefixed with "S-", where the pages are approximately 1 kB in size, and large pages, prefixed with "L-", where the pages sizes are approximately 10 MB in size.

Since plain encoded 64-bit integers require no further decoding, they correspond to performing a plain copy. Figure 4a therefore gives a good indication of the overhead associated with processing the Parquet files on FPGA. When the output size is very small (i.e. small Arrow RecordBatches), the overhead of initializing the FPGA to start operating becomes evident. As the output size grows, we see that the FPGA accelerated solution increases in bandwidth, since it has to spend relatively less time on initialization. The CPU solutions initially increase in throughput, but later decrease around the tens of megabytes range, most likely due to running out of cache space.

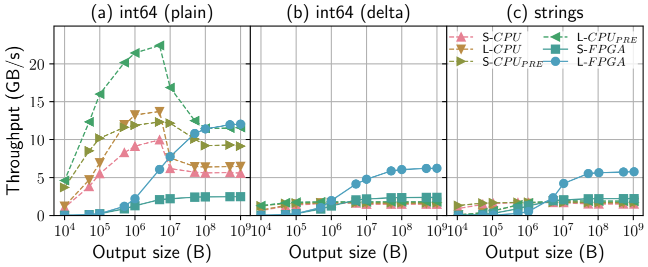


Fig. 6: POWER9/OpenCAPI/9H7 throughput versus Arrow RecordBatch output size

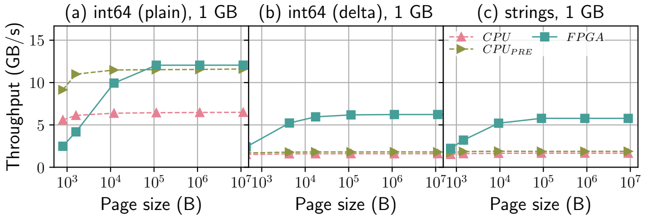


Fig. 7: POWER9/OpenCAPI/9H7 throughput versus Parquet page size

In the delta-encoded and bit-packed integer case, shown in Figure 4b, the integers also need to be unpacked, better revealing the value of the proposed system. Now that calculations have to be performed, we quickly find the $FPGA_{NC}$ to outperform the CPU implementation. However, the end-to-end measurement ($FPGA$) reveals that the overhead associated with making copies from and to the on-board DDR memory still prevents the FPGA accelerated solution from achieving better performance. Finally, in the *strings* case, shown in Figure 4c, the same conclusions as for the *int64 (delta)* case can be drawn.

On the OpenCAPI system, the FPGA implementation is able to outperform the CPU implementation for all data types and encodings, as long as the page and total output sizes are sufficiently large. For the plain encoded integers, the benefit is rather small, but for configurations where actual work has to be performed to decode the data, the FPGA implementation shows its value, resulting in a speedup of around 3×. For the delta encoded integers and the strings, the throughput of the FPGA implementation levels off at around 6 GB/s. This is due to the parallel prefix sum component, for which it is difficult to achieve timing closure at wide configurations.

2) Throughput vs. Parquet page size: We measure the throughput for a 1 GB RecordBatch at various Parquet page

[1] Not to be confused with Parquet pages.

Data type	**Encoding**	**Input stream width** (bits)	**LUTs** (%)	**Registers** (%)	**BRAM** (%)
Int64	Plain	512	1.18	1.27	2.13
Int32	Delta	64	1.46	1.50	2.85
Int32	Delta	128	1.55	1.61	2.99
Int64	Delta	64	1.68	1.64	2.66
Int64	Delta	128	1.76	1.76	2.99
Int64	Delta	256	1.90	1.99	3.24
UTF8	Mixed	128	2.79	2.92	4.47

TABLE I: Resource utilization. Device: Xilinx XCVU9P.

sizes, shown in Figure 5.

For all data types, we observe that interface bandwidth can already be saturated around the default Parquet page size of 1 MiB. When the page sizes are set to be much smaller, we observe that the overhead of decoding headers becomes a bottleneck. However, even for non-realistic page sizes of around one kB, the $FPGA_{NC}$ measurement throughput is better for delta and string data.

C. Resource utilization

In Table I, we find the resource utilization statistics of a single Parquet-to-Arrow converter for the Xilinx XCVU9P. For clarity, this excludes the F1 platform-specific resources. The area utilization is modest, with most resources staying under 5%. This allows for multiple converter cores to be implemented in contemporary FPGAs, that could work on converting the Parquet file in parallel, leveraging its parallel-friendly format. Timing closure for all designs was reached for a 250 MHz clock rate.

D. Discussion

From the results presented, we find the Parquet-to-Arrow converter accelerator to be an interesting alternative to a CPU-only solution. We stipulate the following general observations.

First, our measurements indicate that CPUs will become the bottleneck when loading data from the Parquet file format, rather than I/O bandwidth, for modern storage systems with increased I/O bandwidth. This is most pronounced by first looking at the difference between the CPU measurements of plain encoded integers in Figures 4a and 6a. Since the operation to decode the Parquet file only requires parsing page headers and otherwise only performing `memcpy`, the CPUs are bottlenecked by memory I/O bandwidth, achieving close to 7 GB/s and 12 GB/s when the output size is larger than the caches. However, continuing to look at Figures 4b,4c and Figures 6b,6c, where actual work on decoding has to take place, the CPU performance never reaches above 3 GB/s anymore. The CPU has become the bottleneck. In server-grade systems equipped with many NVMe drives operating in parallel, the available storage bandwidth is much higher than what all CPU cores working in parallel would be able to process.

Second, the main figure of merit being throughput, the FPGA accelerator always outperforms the CPU implementations in terms of processing throughput. To increase the end-to-end bandwidth of the F1 system, it would be interesting to explore overlapping data copy and FPGA computation, where a single instance of our proposed architecture is already able to saturate a PCIe interface.

The OpenCAPI interface provides up to 25 GB/s of full-duplex bandwidth. Reaching this bandwidth is not easy to achieve with a single kernel, since closing timing for the delta decoding step is too hard when the interface is very wide. However, since various parts of a Parquet file may be decoded in parallel, multiple instances of our proposed design would be able to saturate the interface bandwidth with ease. Such a design is very feasible since the area footprint is relatively small and could by estimation fit more than sixteen times in the VU37P, where four instances should saturate the bandwidth in practice. As such, our design should be able to saturate interface bandwidths of current and upcoming storage-attached FPGA accelerator platforms.

Finally, as the Parquet format is of a very dynamic nature, it is challenging to support all possible potential configurations. A Parquet file can only be converted by the accelerator if the facilitated configurations of data type, decompressor and decoder match the columns of interest in the file. If switching between different configurations is required, it is required to extend the decoding components in such a way that the area overhead may be rather large. It would be more beneficial to maintain a library of pre-synthesized configurations that can be (partially) reconfigured into the converter depending on the file. This would leverage the reconfigurability advantage of the FPGA while allowing a Parquet-To-Arrow converter to maintain a relatively small footprint.

V. Conclusion

As I/O bandwidth of storage and network continues to increase, the use of traditional exchange formats for large data structures leaning on the premise of fast CPUs and slow I/O will begin to see CPU bottlenecks. We observed that this bottleneck is also present when loading data from Apache Parquet files into Apache Arrow in-memory data structures at I/O bandwidths of modern storage and network solutions, where the CPUs of our systems are only able to support a throughput in the order of several GB/s.

We proposed an FPGA accelerator design, in which the Parquet files are converted to Apache in-memory data structures, only using the CPU to parse high-level metadata that does not impact performance. We provide a modular and extensible architecture that is able to parse lower-level but more performance-critical Parquet page metadata, in addition to being able to decompress and further decode stored values. The architecture allows users to insert their own decompressors and decoders, based on the many possible encodings that Parquet files may employ. We present an implementation for three data types and encodings in this paper, for plain encoding, and for bit-packed delta-encoded values for both integers and UTF8 strings.

When using encoding schemes more complex than *plain*, results clearly show the merit of using FPGAs. For a POWER9 system with an FPGA connected via OpenCAPI, a single instance of the proposed architecture is able to process realistically configured Parquet files at up to 6 GB/s versus just over 2 GB/s for an optimized CPU implementation. On an

Amazon EC2 F1 system, similar advantages are measured for cases where data transfer between host and FPGA can be overlapped. Otherwise, interface bandwidth becomes a bottleneck and performance does not surpass the CPU. The implementations use a small amount of resources (below 5%), which allows multiple instance of the Parquet-To-Arrow converter to be instantiated. This will allow processing Parquet pages in parallel, increasing throughput as long as there are resources and I/O bandwidth available.

In addition to the encodings, Parquet supports compression standards such as GZip, Snappy or Brotli. Decompression of these algorithms on FPGA is beyond the scope of this paper and has been discussed by various literature. However, if compression would be applied on top of the presented encodings, the FPGA accelerator benefits are expected to become more pronounced. The reason is that decompression will only decrease throughput on the CPU, while there are high-throughput, fully streamable implementations with low resource utilization available for decompression of various algorithms on FPGA. The design includes support for adding a decompression component and integration and evaluation of such designs are envisioned for future work.

In conclusion, the Parquet-To-Arrow converter is a promising heterogeneous alternative to CPU-only based processing of Parquet files into Arrow in-memory data structures. Based on our measurements, saturating the bandwidth provided by current and upcoming storage-attached FPGA acceleration platforms using our proposed design seems feasible.

Acknowledgments

This work is part of the FitOptiVis project [21] funded by the ECSEL Joint Undertaking under grant number H2020-ECSEL-2017-2-783162. The authors thank Xilinx for their additional support.

References

[1] 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.

[2] J. Fang, J. Chen, J. Lee, Z. Al-Ars, and H. P. Hofstee, "Refine and recycle: A method to increase decompression parallelism," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160-052X, 2019, pp. 272–280.

[3] Xilinx. Vitis snappy implementation. [Online]. Available: https://xilinx.github.io/Vitis_Libraries/data_compression/2020.1/source/L2/snappy.html

[4] M. Nanavati, M. Schwarzkopf, J. Wires, and A. Warfield, "Non-volatile storage," *Queue*, vol. 13, no. 9, pp. 20:33–20:56, Nov. 2015. [Online]. Available: http://doi.acm.org/10.1145/2857274.2874238

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

[6] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance Analysis of NVMe SSDs and Their Implication on Real World Databases," in *Proceedings of the 8th ACM International Systems and Storage Conference*, ser. SYSTOR '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2757667.2757684

[7] B. Kim, J. Kim, and S. H. Noh, "Managing Array of SSDs When the Storage Device Is No Longer the Performance Bottleneck," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, Jul. 2017. [Online]. Available: https://www.usenix.org/conference/hotstorage17/program/presentation/kim

[8] J. Peltenburg, A. Hesam, and Z. Al-Ars, "Pushing Big Data into Accelerators: Can the JVM Saturate Our Hardware?" in *High Performance Computing*, J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf, Eds. Cham: Springer International Publishing, 2017, pp. 220–236.

[9] A. Trivedi, P. Stuedi, J. Pfefferle, A. Schuepbach, and B. Metzler, "Albis: High-performance file format for big data systems," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 615–630. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/trivedi

[10] M. Ajdari, P. Park, J. Kim, D. Kwon, and J. Kim, "CIDR: A Cost-Effective In-Line Data Reduction System for Terabit-Per-Second Scale SSD Arrays," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2019, pp. 28–41.

[11] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: It's time for a redesign," *Proc. VLDB Endow.*, vol. 9, no. 7, p. 528–539, Mar. 2016. [Online]. Available: https://doi.org/10.14778/2904483.2904485

[12] L. Kuhring, E. Garcia, and Z. István, "Specialize in Moderation – Building Application-aware Storage Services using FPGAs in the Datacenter," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotstorage19/presentation/kuhring

[13] Apache Software Foundation. Apache Parquet. [Online]. Available: https://parquet.apache.org/

[14] ——. Apache Arrow. [Online]. Available: https://arrow.apache.org/

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

[16] 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.

[17] Google. Snappy. [Online]. Available: http://google.github.io/snappy/

[18] Accelerated Big Data Systems, Delft University of Technology. Hardware snappy decompressor. [Online]. Available: https://github.com/abs-tudelft/vhsnunzip

[19] Xilinx. Vitis gzip implementation. [Online]. Available: https://github.com/Xilinx/Applications/tree/master/GZip

[20] L. van Leeuwen and Delft University of Technology. fast-p2a. [Online]. Available: https://github.com/abs-tudelft/fast-p2a

[21] Z. Al-Ars, T. Basten, A. de Beer, M. Geilen, D. Goswami, P. Jääskeläinen, J. Kadlec, M. M. de Alejandro, F. Palumbo, G. Peeren, L. Pomante, F. van der Linden, J. Saarinen, T. Säntti, C. Sau, and M. K. Zedda, "The fitoptivis ecsel project: Highly efficient distributed embedded image/video processing in cyber-physical systems," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, ser. CF '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 333–338. [Online]. Available: https://doi.org/10.1145/3310273.3323437