FORC: A High-Throughput Streaming FPGA Accelerator for Optimized Row Columnar File Decoders in Big Data Engines

Abdul Wadood
Simon Fraser University
Burnaby, BC, Canada
abdul_wadood@sfu.ca

Alec Lu
Simon Fraser University
Burnaby, BC, Canada
alec_lu@sfu.ca

Ken Zhang
Huawei Canada
Markham, ON, Canada
ken.zhang1@huawei.com

Zhenman Fang
Simon Fraser University
Burnaby, BC, Canada
zhenman@sfu.ca

***Abstract*—To improve the file storage efficiency of large datasets, big data analytics usually use some common file formats, such as Apache ORC (optimized row columnar) format, to encode and compress the data. However, this shifts the IO bottleneck (especially with high-bandwidth SSDs) to the computation bottleneck on CPUs to decompress and decode the data. This paper presents FORC, a high-throughput streaming-based FPGA accelerator overlay that supports different ORC file format decoders, and its dataflow integration with Apache ORC. Experimental results show that FORC achieves up to 12.9GB/s decoding throughput on AMD/Xilinx Alveo U280 FPGA, with a geomean speedup of 65x (up to 335x) over the CPU. FORC will be released soon at https://github.com/SFU-HiAccel/FORC.**

I. Introduction

The vast volume of datasets have led to the adoption of various file formats in big data analytics—such as Apache Parquet [1], Avro [2], and ORC [3]—to improve the data storage and management efficiency. These file formats usually encode and compress the raw data to reduce their storage space and facilitate faster data transfers, and improve query performance by retrieving only the relevant data. Among them, the column-oriented Apache ORC has risen to prominence due to its better data compression ratio and query performance [4], as well as its built-in support for ACID (atomicity, consistency, isolation, durability) transactions [3].

Unfortunately, the storage benefits come at the cost of higher computation demand to decompress and decode these file formats for downstream query operations. To demonstrate this, in Fig. 1, we break down the execution time for the CPU to scan an ORC-encoded file from an SSD to a consumable table in memory (detailed setup in Section IV-A). With the improved IO bandwidth from the SSD, IO read only occupies 3% of the execution time. Instead, data decompression and decoding of the ORC file on the CPU occupy 54.2% and 40.3% of the execution time, which become the new bottlenecks and require computation acceleration. While prior studies [5] [6] [7] have well accelerated the data decompression, there is few existing work in accelerating the data decoding stage in ORC.

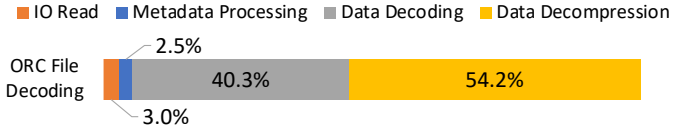

Fig. 1: CPU runtime breakdown of processing ORC file

In this work, we focus on accelerating the data decoding stage in processing ORC files with commodity datacenter FPGAs, which is nontrivial due to the following challenges. First, an ORC file often uses a (different) combination of multiple encoding schemes, including short repeat, direct, patched base, and delta encoding [3]. Designing a separate decoder engine for each encoding would consume resources quickly and leave little room to incorporate downstream query accelerators on the same FPGA. Second, decoding the (encoded) raw data depends on the decoding of the header to reveal the encoding scheme, which takes multiple clock cycles and could prevent the fully pipelined design (i.e., pipeline initiation interval of one, $II = 1$). Moreover, the sequential accumulation of the delta decoder (detailed in Section III-B) poses a challenge for a fully pipelined design to decode multiple data items per cycle from a 512-bit wide stream. Finally, for big data analytics to transparently leverage the benefits of the FPGA-accelerated decoders, there needs a seamless and efficient integration of the FPGA accelerator into Apache ORC, which could well hide the data transfer overhead between the CPU and FPGA, and the IO read.

To address the above challenges, we design and implement FORC, the first high-throughput streaming-based FPGA accelerator for decoding Apache ORC files used in modern big data engines. FORC incorporates the following novel features:

1. A resource-efficient overlay design to share the design of common operations across decoders and support different combinations of ORC decoding schemes (and bit widths).
2. A fully (dynamic) pipelined ORC decoder engine that can process up to the output streaming rate, i.e., four 512-bit wide streaming writes per cycle. For the header decoding obstacle, we trade off pipeline latency for throughput by buffering and delaying the starting of fully pipelined data decoding. For the delta decoding obstacle, we customize a resource-efficient tree-based partial accumulation architecture to achieve a fully pipelined design.
3. An end-to-end dataflow integration of our accelerator into Apache ORC C++ library [3], which well overlaps the IO read, CPU-FPGA data transfer, and FPGA computation.

Our FORC accelerator only uses 17.68% of LUTs and negligible amount of DSPs/BRAMs on the AMD/Xilinx Alveo U280 FPGA. Evaluated on both synthetic datasets and widely used TPC-H datasets [8], our FORC FPGA engine achieves up to 12.9GB/s decoding throughput considering the input data and 48.5GB/s decoding throughput considering the inflated output data, which is on average 65x faster than the Intel

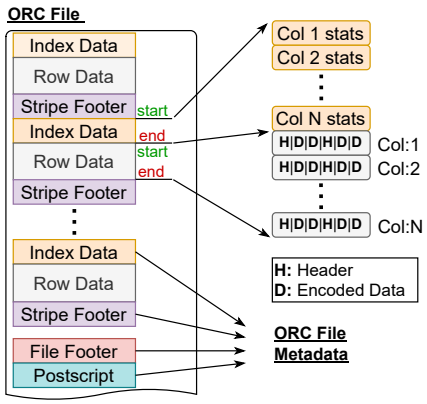

Fig. 2: ORC file structure

Xeon CPU implementation of Apache ORC C++. For the end-to-end integration, FORC achieves a throughput up to the IO bandwidth limit and PCIe bandwidth limit, which is on average 11x faster than the CPU implementation.

II. Apache ORC File Format

The Apache ORC (optimized row columnar) file format [3] is structured to optimize the storage and retrieval of large datasets in big data engines. It stores data in columns (columnar storage) to enhance compression and query performance. Fig. 2 describes the overall ORC file structure [3]. It divides a file into multiple stripes. The file footer and postscript contain critical file metadata, such as the type of compression used, location of each stripe, type schema information, number of rows, and statistics about each column. Each stripe is self-contained and includes three components: index data, row data, and stripe footer. The stripe footer contains information about the location of each column and encoding type in general. The index data contains the statistics (e.g., min and max) of each column, which is used for optimization of the query execution. The row data is stored in multiple columns, where each column stores the encoded (raw) data and the headers indicating the encoding schemes. Note each column can use a combination of multiple encoding schemes. In this paper, we refer to all the file footer, postscript, stripe footer and index data, except the row data, as ORC file metadata.

A. ORC Encoding and Decoding Schemes

Apache Hive 0.12 introduced a new version of run length encoding (RLEv2) in ORC, which enhanced compression capabilities and optimized fixed data width encodings [3]. RLEv2 employs four distinct encoding schemes that are based on the data characteristics: short repeat, direct, patched base, and delta encoding [3]. As shown in Fig. 2, for each column, data is encoded in batches, where each batch uses a dedicated encoding and has a header recording essential encoding information. The first two bits of the header act as identifiers for the chosen encoding scheme. Next we explain the common operations among all these data encoding schemes and thus the decoders, followed by scheme-specific operations.

TABLE I: Examples of four different ORC encoding schemes.

Encode Scheme	Raw Data	Header	Encoded Data
Short repeat (#0)	[500, 500, 500, 500, 500]	1 byte (#0, 16 bits, 5 values)	500
Direct (#1)	[2000, 300, 800]	2 bytes (#1, 16 bits, 3 values)	2000, 300, 8000
Patched base (#2)	[2030, 2000, 2020, 1026010, 2040]	4 bytes (#2, 8 bits, 5 values; base value: 16 bits; PW: 12 bits, PGW: 2 bits, PLL: 1)	2000, 30, 0, 20, 10, 40, (3, 4000)
Delta (#3)	[500, 510, 550, 555, 560, 572]	2 bytes (#3, 8 bits, 6 values)	500, 10, 40, 5, 5, 12

1) Common Encoding and Decoding Operations: There are two common operations among all data encoding schemes:

1. **Changing data endianness**: ORC encodes data in big endian format for compatibility with its Java version. However, the Intel Xeon CPU and FPGA accelerator use little endian to store data. Therefore, before decoding the data according to each encoding scheme, we first need to convert big endian into little endian by shifting the data bytes.
2. **Zigzag and unzigzag for signed integers**: For a signed integer data, ORC always converts it into an unsigned integer representation using a zigzag encoding, which moves the sign bit to the least significant bit using the expression `(val << 1) ^ (val >> [maxBitSize])`. During the decoding stage, an unzigzag operation `(value >> 1) ^ -(value & 1)` is required before applying each specific decoding scheme.

2) Scheme-Specific Encoding and Decoding Operations: Table I gives an example for each data encoding scheme:

1. **Short repeat:** It is used to encode short sequences with 3 to 10 repeated values. Table I shows an example to encode five 500s. For the encoded data, it only records the value once. Its header is one byte long and encodes the following: 1) the first two bits indicate the encoding scheme; 2) the next three bits encode the data width (in bytes) of values; and 3) the last three bits indicate how many times the value repeats (i.e., run length).
2. **Direct:** It is used for random sequences with a fixed data width, up to 512 values. For the encoded data, it directly copies the input data (Table I). Its header is two bytes long and encodes the following: 1) the first two bits indicate the encoding scheme; 2) the next five bits encode the bit width of values; and 3) the last nine bits indicate how many values there are (i.e., run length).
3. **Patched base:** It is used for random sequences with a variable data width, up to 512 values. Table I shows an example of input sequence of 2030, 2000, 2020, 1026010, 2040, where 1,026,010 is 32-bit wide and other values are 16-bit wide. To encode this sequence, it has three steps: 1) a common base value (the smallest number, 16-bit) of 2000 is recorded; 2) the difference (8-bit) of each value subtracting the base value is recorded as 30, 0, 20, 10, 40; 3) for each larger bit width value (1,026,010) whose difference value overflows (in 8-bit), a patch value for the remaining difference (1,024,000) needs to be recorded. Such patch value is recorded in a pair: 1) the first is its index (i.e., 3) in the original sequence, whose bit width is denoted as

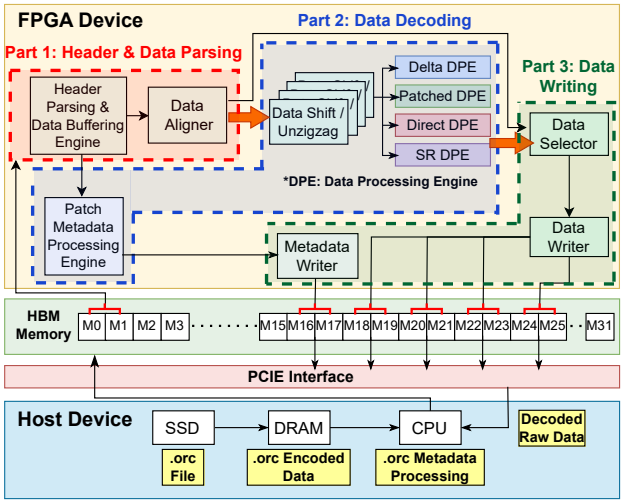

Fig. 3: Overview of FORC hardware architecture

patch gap width; and 2) the second is the high bits of the difference, i.e., 1,024,000 >> 8 = 4,000, whose bit width (excluding leading zeros) is denoted as patch width.
Its header is four bytes long. The first two bytes follow the same encoding as described for direct encoding, except that the bit width (8 bits for the example) is for the subtracted difference value. The remaining two bytes of the header encode the following: 1) the first three bits indicate the data width (in bytes, 16 bits) of the base value; 2) the next five bits encode the patch width (PW=12 bits for the example); 3) the next three bits encode the patch gap width (PGW=2 bits); and 4) the last five bits indicate the patch list length (PLL=1), i.e., number of patches.

4. **Delta:** It is used for monotonically increasing or decreasing sequences with a fixed data width, up to 512 values. For the encoded data, it stores the first input value as the base value; for each remaining input, it stores its difference (delta) to its immediate predecessor value. The bit width of the deltas is usually smaller than that of the original values. The header encoding follows the same encoding as described for direct encoding, except that the bit width is for the delta values. There is a special case when all delta values are the same: the header encodes the delta bit width to zero to indicate this case (Delta_0b), and only one delta value is encoded.

III. FORC Design and Implementation

To avoid frequent CPU-accelerator communication overhead, we decide to offload all ORC decoders onto the accelerator. Moreover, to leave sufficient amount of resources for integrating other query accelerators such as decompression, filtering, and sorting, we propose a resource-efficient overlay design that dynamically supports all combinations of ORC decoders discussed in Section II-A and shares as many common operations as possible among them. Performance wise, we design a fully pipelined streaming accelerator architecture that processes up to the output memory streaming rate, which goes beyond today's SSD and PCIe bandwidth limits and is future-proof for next-generation high-bandwidth IO devices.

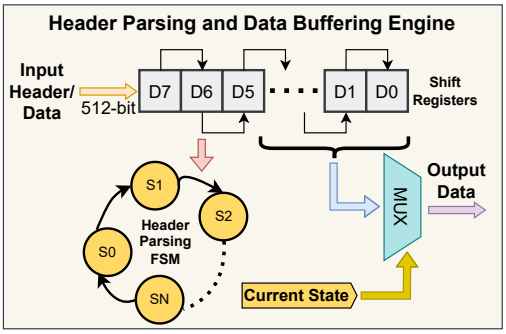

Fig. 4: Header parsing and data buffering engine

Fig. 3 shows an overview of the FORC hardware architecture on a commodity HBM-based FPGA device. First, Section III-A describes **Part 1: Header & Data Parsing Engine**, which streams in 512-bit data per cycle to saturate the off-chip memory bandwidth of one HBM bank [9], dynamically parses the header, buffers and aligns the data for all types of ORC decoders with delayed processing, in a fully pipelined fashion. Second, Section III-B presents fully pipelined **Part 2: Data Decoding** logic of all four decoders and their common data shift/unzigzag unit, including hardware optimization techniques to save resource and improve throughput. Thirdly, Section III-C shows **Part 3: Data Writing** common logic, preparing and streaming out the decoded (and often inflated) raw data into multiple HBM banks. Finally, Section III-D discusses the end-to-end integration of FORC with Apache ORC C++ library [3], overlapping the IO read, CPU-FPGA data transfer, and FPGA computation.

A. Header & Data Parsing Engine

Header Parsing & Data Buffering Engine. As shown in Fig. 4, to saturate the off-chip memory bandwidth [9], it streams in one 512-bit data per cycle and buffers the data in a shift register to be processed in a pipeline fashion. The header parsing finite-state machine (FSM) constantly monitors the header information from this shift register that contains both header and encoded data. Once a header is detected, the header parsing FSM takes six cycles to decode: 1) encoding scheme, 2) data width, 3) run length, and 4) scheme-specific information for patched base and delta encoding schemes. In the next cycle, the buffered (encoded) data along with decoded header information will be streamed to the data aligner. Note the first data cannot be sent until the header is parsed (pipeline depth = 7); after that, all following data are passed in a fully pipelined fashion (II=1). The pipeline overhead is usually small except for a small run length (pipeline tripcount).
Data Aligner. Upon receiving the encoded data (which is unaligned as the header occupied some bytes in the stream) and the decoded header information, data aligner first prepares the encoded data by partitioning and repacking it based on the data width, and then sends it to the data shift/unzigzag units. For the special case of delta encoder (Delta_0b), where all delta values are the same and only one delta value is encoded, the delta value is duplicated and passed to each data shift/unzigzag unit. Data aligner also passes the decoded

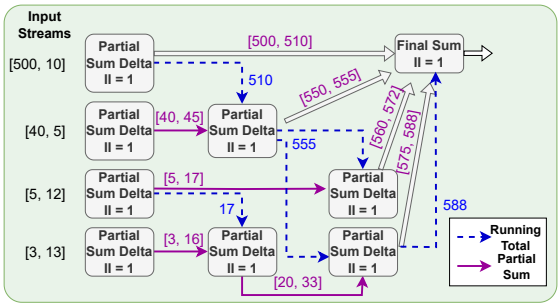

Fig. 5: Fully pipelined delta decoding engine design

header information to the data selector that is used in the data writing logic (Section III-C).

B. Data Decoding Engines

Data Shift/Unzigzag Unit. Upon receiving the aligned data and before sending it to each specific decoder, this unit unpacks the 512-bit data from the stream into multiple items with the decoded data width (e.g., 16 32-bit data items). After that, it converts the endianness of the data to little endian and performs the data unzigzag operation as described in Section II-A1. To make sure this unit can process 512-bit of encoded data per cycle, these operations are coarsely pipelined, each parallelized by a factor of 16. To support the 8-bit encoding data width, we include four of these units.

Delta Decoding PE (Processing Element). In delta decoding, the input delta value is sequentially accumulated with the running sum from its immediate predecessor, which poses a challenge when we need to decode multiple data items (e.g., 16 32-bit data items) per cycle from the wide stream. Fig. 5 shows how the data streams with encoded values from Table I are decoded. To achieve a dynamic initial interval (II) of 1, we optimize our design to compute multiple partial sums on (sub-)streams in parallel and over multiple stages.

In the first cycle, each data stream computes its own partial delta sum (e.g, [500, 10]->[500, 510], [40, 5]->[40, 45], and so on). After this, the partial delta sum from the first data stream goes to the final sum unit to be added with the previous stored running sum (this is 0 in the example as the stream just started with the base value 500) and pass to data writing. Meanwhile, the following partial sum units in the pipeline will add the running total from stream one to the previously calculated partial delta sums from stream two: e.g, running total 510 from stream one is added to partial sums [40, 45] from stream two to get [550, 555] for stream two. This process repeats until all data streams have been accumulated with the running total from previous streams and it is fully pipeline with II=1. Moreover, to reduce the number of pipeline stages, we add partial sum units between every two neighbor streams.

Patched Base Decoding PE. Four summation units work in parallel to add the base value with the difference values from data shift/unzigzag unit and pass the output to data writing.

Patched Metadata PE. As described in Section II-A2, for each value with a larger bit width, a patch index and value pair is encoded. The patched metadata PE directly gets the patch list from the header parsing and data buffering engine,

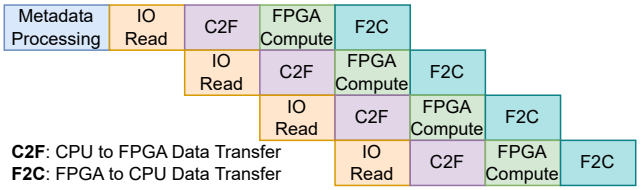

Fig. 6: Dataflow integration of FORC into ORC C++

and decodes it to get each pair of the patch index and patch value (higher bits of the remaining difference). And the patch value is left shifted by the bit width from the decoded header (number of lower bits) to get the full-bits remaining difference. After that, we pass the new pairs to the metadata writer.

Instead of adding this remaining difference with the sum obtained from the patched base decoding PE on the FPGA, we leave it to the host CPU to do this final addition when the CPU reads the data. The reason is that the patched metadata PE can only start reading the patch list after all regular difference values have been read (and processed), due to the streaming nature of our design. Note in practice, patched base encoding is only used if at most 5% of the values in the sequence are of variable bit width; otherwise, direct encoding is used. So this is not a performance issue in our end-to-end integration.

Short Repeat (SR) Decoding PE. Data after shift/unzigzag is duplicated run length number of times and sent to data writing.

Direct Decoding PE. The output data from the shift/unzigzag units is treated as the decoded data and sent to data writing.

C. Data Writing

Data selector maintains the order of the decoded raw data using the decoded header information from the data aligner.

Data writer divides the decoded raw data and writes them onto four parallel off-chip HBM banks, considering the output data inflation.

Metadata writer takes the decoded patch list metadata from the patched metadata PE and writes it to one off-chip HBM bank.

D. End-to-End Integration with ORC C++ Lib

To transparently leverage our accelerated decoders in big data analytics, we integrate FORC into the Apache ORC C++ library [3]. Fig. 6 shows the dataflow integration in the host program: the four stages—IO read, CPU to FPGA input data transfer, FPGA decoding, and FPGA to CPU output data transfer—for each stripe of an ORC file are executed in a dataflow fashion. To achieve this, each input and output port mentioned in prior subsections are connected to two HBM banks to enable the ping-pong overlapping. As a result, FORC utilizes a total of 12 HBM banks: two for input data, eight for inflated output data, and another two for the patch list metadata. Note the metadata (defined in Section II) processing is done only once per ORC file using the ORC software library tools, before initiating the ORC decoding on the FPGA.

We model the end-to-end decoding throughput of FORC as:

$$BW_{E2E} = \min\left(BW_{IO}, \frac{BW_{PCIe}}{CR+1}, Compute_{FPGA}\right) \quad (1)$$

where CR is the compression ratio and $BW_{PCIe}/(CR+1)$ is the effective PCIe bandwidth (BW) shared between transferring both input data and decoded output data (inflated by CR times), BW_{IO} is the disk IO read bandwidth, and $Compute_{FPGA}$ is the FPGA engine's decoding throughput.

IV. Results and Analysis

A. Experimental Setup

Benchmark Datasets. First, we evaluate FORC for each decoding scheme under a range of data widths (8-bit to 32-bit) using synthetic ORC dataset where each column contains only one particular encoding scheme. Then we evaluate FORC on the widely used TPC-H [8] dataset, which uses different combinations of decoding schemes and data widths. We used a 10GB dataset scale for TPC-H tables, and loaded the dataset from a completely cold system. Due to the fully dataflow nature of our FORC design, increasing the dataset size beyond 10GB does not affect the achieved decoding throughput.

Hardware and Software Setup. We evaluate our FORC design on the AMD/Xilinx Alveo U280 datacenter FPGA board (with 32 HBM2 banks) [10]. FORC is developed entirely using Vitis HLS, and the accelerator is synthesized using Vitis 2021.2 [11] and an open source floorplanning optimization tool for task-parallel HLS designs called TAPA/Autobridge (Ver.0.0.20221113.1) [12]. Without TAPA/AutoBridge, FORC achieves 199MHz frequency. With default TAPA/AutoBridge floorplanning, it achieves 208MHz frequency. After we further provide manual placement constraints to TAPA/AutoBridge, it achieves 222MHz frequency for on-board execution.

For the CPU implementation, we evaluate the well-optimized Apache ORC C++ library v1.8.6 [3] on an Intel Xeon Silver 4214 2.20GHz CPU with data stored on a Samsung NVMe SSD with 3.0 to 3.3 GB/s of sequential data read bandwidth. The FPGA board is connected to the Xeon CPU via 16-lane PCIe Gen 3, which achieves around 10GB/s bandwidth using Vitis.

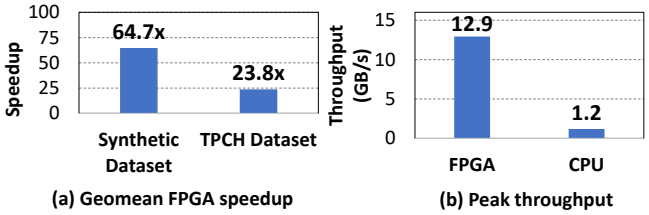

Fig. 7: Geomean speedup and peak throughput of FORC

B. Overall FORC Speedup Over CPU

Fig. 7(a) summarizes the geomean performance speedup of the FORC FPGA engine over the CPU version, which are 64.7x and 23.8x on synthetic dataset and TPC-H dataset. This difference is caused by different combinations of decoding schemes, which we will explain in Section IV-C and IV-D. Fig. 7(b) compares their peak decoding throughput considering input data stream. FORC achieves 12.9GB/s decoding throughput, which fully utilizes the effective bandwidth of one HBM bank characterized in [9] and is 11x higher over the CPU. Note the reported FPGA decoding throughput only considers the FPGA kernel execution time, without considering the IO read and the data transfer between the host and the FPGA, which we will include in the end-to-end throughput report. It demonstrates that our design's throughput goes beyond today's SSD and PCIe bandwidth limits and is future-proof for next-generation high-bandwidth IO devices. Similarly, the reported CPU decoding throughput does not include IO read.

C. FORC Throughput for Each Decoder

Fig. 8 compares the throughput of each decoder on the CPU and FPGA, considering the throughput for both input and output streams. For each decoder, we also show the results for the smallest and biggest bit widths (8-bit and 32-bit) to provide a range of achievable throughput under different compression ratios *(CR = output size / input size)* by the encoders.

Direct Decoding Throughput. For Direct_32b, the compression ratio is one. The input and output throughput of FORC are the same 12.9GB/s, which fully utilizes the bandwidth of one HBM bank and is about 11x faster than the CPU. For Direct_8b, the compression ratio is 4x. The input throughput on the FPGA drops a bit to 11.9GB/s due to more frequent header parsing overhead under the same data size. The output throughput on the FPGA is 47.4GB/s, which is roughly 4x of its input throughput and 42.5x higher than that of the CPU. For intermediate bit widths, the input throughput of FORC will be in the range of 11.9GB/s to 12.9GB/s.

Delta Decoding Throughput. For Delta_32b and Delta_8b, the FPGA input and output throughput is similar to that of Direct_32b and Direct_8b, while the CPU throughput is much lower than direct decoding, which confirm the efficiency of our fully pipelined design. In the extreme of Delta_0b, where all delta values are the same and only one delta value is encoded, the input throughput for both CPU and FPGA significantly drops due to the extremely high compression ratio of ∼238x and thus significant back pressure from the data writing part (i.e., writing ∼238x more output). The key metric to look at here should be the output throughput, for which FORC still achieves 48.51GB/s, 56.5x faster than the CPU.

Patched Base Decoding Throughput. The up range of the bit width for patched base decoding is 24-bit, as 32-bit is reserved for the larger variable bit width. Compared to direct decoding and delta decoding, the FPGA input and output throughput drops due to extra delay of processing the patch list metadata. Nevertheless, the FPGA can still achieve an input throughput of 9.72GB/s to 12.83GB/s, ∼335x faster than the CPU.

Short Repeat Decoding Throughput. Due to the extremely small run length (3 to 10) in short repeat decoding, there is not much room for parallelism and pipeline, and there is too frequent header parsing, i.e., one header parsing per encoded sequence in the 512-bit wide stream. As a result, both CPU and FPGA throughput are limited. Nevertheless, the FPGA is still on average ∼104x faster than the CPU. In addition, we need to include this decoder on the FPGA overlay to avoid frequent CPU-FPGA communication when decoding a real dataset that uses a mix of decoding schemes.

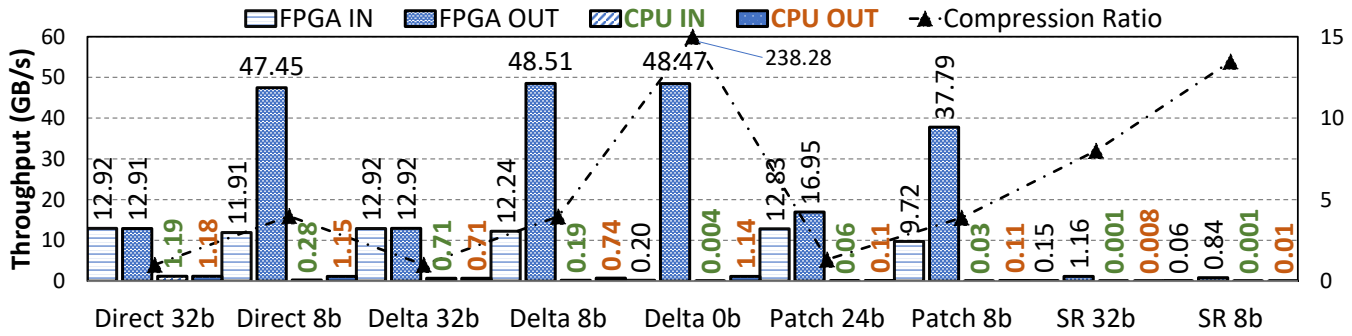

Fig. 8: Throughput comparison of direct, delta, patched base, and short repeat decoders under different bit widths

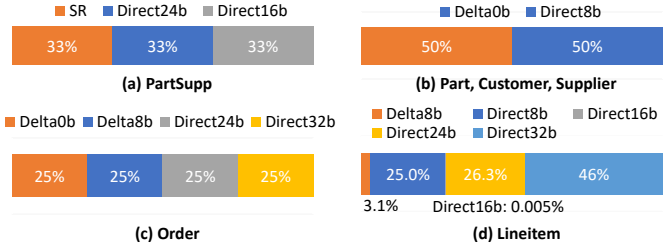

Fig. 9: TPC-H dataset decoding distributions

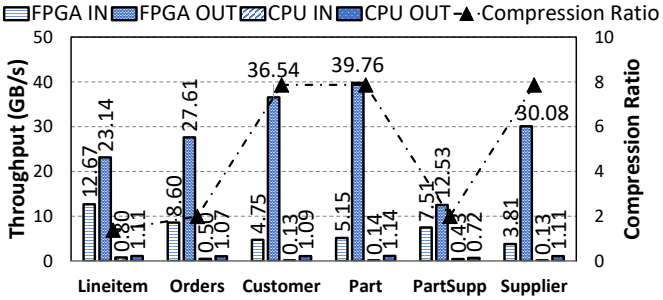

Fig. 10: TPC-H dataset FPGA decoding throughput

D. FORC Throughput on TPC-H Dataset

TPC-H Decoding Combination and Throughput. We also evaluate FORC on the industrial standard TPC-H dataset, which uses a combination of different decoding schemes and bit widths, as shown in Fig. 9. Fig. 10 compares the effective decoding throughput on FPGA and CPU, which is decided by the decoding combination in Fig. 9. On average, FORC achieves 24x speedup over the CPU. This speedup is lower compared to that for the synthetic dataset as TPC-H dataset mostly uses direct and delta decodings which achieve relatively lower speedups. For the lineitem table, it mostly uses direct decoding with different bit widths, and thereby achieves a high input throughput of 12.7GB/s. For the customer, part, and supplier tables, they use a mix of Delta_0b and Direct_8b schemes with a high compression ratio; therefore, they achieve a high output throughput but lower input throughput. For the partSupp table, its throughput is impacted by the slowest short repeat decoder. For the orders table, it has a more balanced mix and achieves something in the middle.

End-to-End Throughput. Fig. 11 compares the end-to-end decoding throughput of the FORC integration into Apache

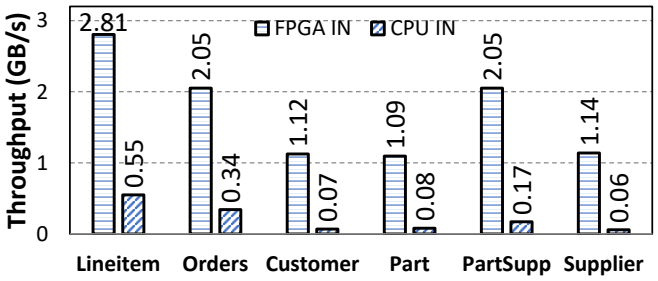

Fig. 11: TPC-H dataset end-to-end decoding throughput

ORC C++ library on the TPC-H dataset. In the end-to-end throughput, we consider both the SSD read and host-FPGA data transfer via PCIe, and its peak throughput is constrained by SSD bandwidth and/or PCIe bandwidth. On average, it achieves a geomean of 11x speedup over the CPU version. For the lineitem table, it has a low compression ratio and it achieves 2.8GB/s end-to-end throughput, close to the SSD read bandwidth. For the customer, part, and supplier tables that have a high compression ratio (CR), they are limited by the PCIe bandwidth, $BW_{PCIe}/(CR+1)$, as analyzed in Section III-D. For the orders and partSupp tables, their end-to-end throughput also drops lower than the SSD read bandwidth due to the presence of much slower decoding schemes, i.e., Delta_0b and short repeat, in some decoding phases.

E. Resource Utilization and Power Consumption

Table II breaks down post place and route resource utilization of FORC on the Alveo U280 FPGA. First, our complete design only uses 17.68% LUTs, 4.17% FFs, and negligible DSP/URAM/BRAM resources, which leaves plenty of resources to integrate other query accelerators. Second, our scheme-specific decoding engines use only 3.93% LUTs and 1.47% FFs and the remaining resources are all used by the common logic shared by different decoders, which confirm the resource-efficiency of our overlay design.

TABLE II: Resource utilization breakdown on Alveo U280

	LUTs	FFs	DSPs	URAM/BRAM
P1: header & data parsing	*6.60%*	*1.27%*	*0.03%*	*0*
P2a: data shift/un-zigzag	*5.30%*	*0.92%*	*0*	*0*
P2b: data decoding engines	*3.93%*	*1.47%*	0	0
P3: data writing	*1.85%*	*0.50%*	*0*	*0*
Overall FORC design	17.68%	4.17%	0.03%	0

Finally, the power consumption is measured using vendor *xbutil* tool. The U280 FPGA board consumes ∼26.2W static power and our design only consumes ∼3.9W dynamic power.

V. Related Work

Previous efforts have proposed FPGA accelerator designs for big data query operations such as decompression [5]–[7], filtering [13], and sorting [14], [15]. Many of these studies are covered in [16], where Fang et al. also provide a comprehensive survey on the status of in-memory database acceleration on FPGAs. However, there is few existing work focusing on big data file format decoding, which is becoming one of the major (orthogonal) computation bottlenecks in big data analytics, especially with today's high-bandwidth SSDs.

The only exception is [17], where Peltenburg et al. present: 1) an FPGA accelerator for decoding Apache Parquet [1] file format into Arrow [18] in-memory data, 2) separate engine designs for different decoding schemes (only direct and delta decoders) and data widths, and 3) a maximum throughput of 8GB/s on AWS F1 instance for direct and delta decoders. To distinguish our efforts from [17], first, FORC decodes ORC file format to in-memory data, which provides better data compression ratio and query performance than Parquet [4], as well as built-in support for ACID transactions that Parquet does not support [3]. Second, FORC designs a unified resource-efficient overlay supporting all ORC decoders explained in Section II-A2—including patched base and short repeat decoders that [17] does not support—and a wide range of data widths. Last but not least, FORC achieves a fully pipelined design and fully utilizes the effective HBM bank bandwidth on the FPGA, achieving up to 12.9GB/s throughput for direct and delta decoding, 61.5% higher than that of [17].

VI. Conclusion and Future Work

In this work we have proposed a high-throughput streaming-based FPGA accelerator overlay called FORC to accelerate widely used ORC file format decoding in big data engines. FORC employs a resource-efficient and fully pipelined overlay design to support different combinations of ORC decoding schemes and bit widths, which can process up to the output streaming rate. To transparently leverage the FPGA acceleration for big data analytics, we also integrate FORC with Apache ORC C++ library to execute in a dataflow fashion, which achieves an end-to-end decoding throughput up to the IO read and PCIe bandwidth limits. Across a wide range of synthetic dataset and industry standard TPC-H dataset, FORC achieves a geomean performance speedup of 65x for the FPGA engine and 11x for the end-to-end performance, over the CPU version. Our FORC framework will be open-sourced soon at https://github.com/SFU-HiAccel/FORC. In future work, we plan to extend it to support decompression operations as well.

VII. Acknowledgement

We thank the anonymous reviewers for their valuable feedback and thank Jiwen Yu, Gilbert Fang, Arpit Malhotra and Benjamin Correia for their help in the end-to-end integration of our FPGA design into the ORC library. We acknowledge the partial support from NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020; CFI John R. Evans Leaders Fund and BC Knowledge Development Fund; Huawei Canada and AMD-Xilinx.

References

[1] Apach Parquet, "Apache parquet is an open source, column-oriented data file format designed for efficient data storage and retrieval." https://parquet.apache.org/, 2023.

[2] Apache AVRO, "Apache avro™ - a data serialization system," https://avro.apache.org/, 2023.

[3] Apache ORC, "Apache orc: the smallest, fastest columnar storage for hadoop workloads," https://orc.apache.org/, 2023.

[4] C. Shanklin, "Orcfile in HDP 2: Better Compression, Better Performance - Cloudera Blog," https://blog.cloudera.com/orcfile-in-hdp-2-better-compression-better-performance/, Sep. 2013.

[5] J. Fang, J. Chen, Z. Al-Ars, P. Hofstee, and J. Hidders, "Work-in-progress: A high-bandwidth snappy decompressor in reconfigurable logic," in *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2018, pp. 1–2.

[6] M. Ledwon, B. F. Cockburn, and J. Han, "Design and evaluation of an fpga-based hardware accelerator for deflate data decompression," in *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*, 2019, pp. 1–6.

[7] Xilinx, "Github - Xilinx/Vitis_Libraries: Vitis Libraries," https://github.com/Xilinx/Vitis_Libraries, 2023.

[8] TPC, "Tpc-h is a decision support benchmark," 2023. [Online]. Available: https://www.tpc.org/tpch/

[9] A. Lu, Z. Fang, W. Liu, and L. Shannon, "Demystifying the memory system of modern datacenter fpgas for software programmers through microbenchmarking," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 105–115.

[10] Xilinx, "Alveo u280 data center accelerator card data sheet (ds963)," 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ds963-u280/Summary

[11] ——, "Vitis unified software platform," 2022. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html#development

[12] L. Guo, Y. Chi, J. Lau, L. Song, X. Tian, M. Khatti, W. Qiao, J. Wang, E. Ustun, Z. Fang, Z. Zhang, and J. Cong, "Tapa: A scalable task-parallel dataflow programming framework for modern fpgas with co-optimization of hls and physical design," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 4, dec 2023.

[13] X. Sun, C. J. Xue, J. Yu, T.-W. Kuo, and X. Liu, "Accelerating data filtering for database using fpga," *J. Syst. Archit.*, vol. 114, no. C, mar 2021.

[14] M. Abdelrasoul, A. S. Shaban, and H. Abdel-Kader, "Fpga based hardware accelerator for sorting data," in *2021 9th International Japan-Africa Conference on Electronics, Communications, and Computations (JAC-ECC)*, 2021, pp. 57–60.

[15] W. Qiao, L. Guo, Z. Fang, M. Chang, and J. Cong, "Topsort: A high-performance two-phase sorting accelerator optimized on hbm-based fpgas," in *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 02. Los Alamitos, CA, USA: IEEE Computer Society, apr 2023, pp. 404–419.

[16] J. Fang, Y. T. B. Mulder, J. Hidders, J. Lee, and H. P. Hofstee, "In-memory database acceleration on fpgas: a survey," *The VLDB Journal*, vol. 29, no. 1, p. 33–59, oct 2019.

[17] J. Peltenburg, L. T. van Leeuwen, J. Hoozemans, J. Fang, Z. Al-Ars, and H. P. Hofstee, "Battling the cpu bottleneck in apache parquet to arrow conversion using fpga," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 281–286.

[18] Apache Arrow, "Apache arrow: A cross-language development platform for in-memory analytics." https://arrow.apache.org/, 2023.