FORCv2: A High-Throughput Streaming FPGA Accelerator for Optimized Row Columnar File Format Processing in Big Data Engines

ABDUL WADOOD, ALEC LU, HAIKAI ZHAO, ZHENMAN FANG, Simon Fraser University, Canada

To enhance the storage efficiency of large datasets, big data analytics commonly rely on columnar file formats, such as Apache ORC (Optimized Row Columnar), to encode and compress data. These formats significantly reduce storage requirements and improve query performance by supporting efficient data organization and compression techniques. However, the use of such file formats introduces a new challenge: while high-bandwidth SSDs mitigate the I/O bottleneck, the computational burden of decompressing and decoding these formats shifts the bottleneck to the CPU. This computational overhead becomes a critical challenge in achieving high-throughput data processing. This work introduces FORCv2, a high-performance streaming-based FPGA accelerator overlay designed to overcome these limitations. First, FORCv2 integrates multi-PE Zlib decompression which leverages Xilinx's high-performance single-PE (processing element) implementation and enhances performance with multi-way parallelism. Second, it features fully pipelined data decoding, optimized to achieve throughput close to the memory bandwidth of a single HBM channel, ensuring high-efficiency processing of ORC files. Third, FORCv2 incorporates flexible and efficient filtering capabilities, supporting both pre-stored indices and user-defined filter conditions for versatile data processing. Finally, it provides seamless dataflow integration with the Apache ORC library, enabling efficient end-to-end processing of ORC file formats. The proposed design leverages the inherent parallelism and high-bandwidth memory (HBM) capabilities of AMD/Xilinx Alveo U280 FPGAs to deliver exceptional performance. By employing a fully pipelined architecture and resource-efficient design, FORCv2 achieves up to 2.8 GB/s of overall throughput (input throughput). Considering all operators combined, on average, it provides end-to-end speedups of approximately 52x for the synthetic dataset and 39x for the TPC-H dataset, versus the ORC C++ library running on a dual-socket system with 48 CPU threads. Experimental evaluations demonstrate the effectiveness of FORCv2 across diverse workloads, showcasing its potential to handle real-world big data processing scenarios efficiently. The FORCv2 implementation will be open-sourced at https://github.com/SFU-HiAccel/FORC.

CCS Concepts: • **Hardware → Hardware accelerators**; **Hardware-software codesign**; **Partitioning and floorplanning**; • **Computer systems organization → Data flow architectures**; **High-level language architectures**; **Reconfigurable computing**.

Additional Key Words and Phrases: Big data, Hardware acceleration, File format, Apache ORC, High-level synthesis

ACM Reference Format:
Abdul Wadood, Alec Lu, Haikai Zhao, Zhenman Fang. 2025. FORCv2: A High-Throughput Streaming FPGA Accelerator for Optimized Row Columnar File Format Processing in Big Data Engines . *ACM Trans. Reconfig. Technol. Syst.* 1, 1 (December 2025), 33 pages. https://doi.org/XXXXXXX.XXXXXXX

Author's Contact Information: Abdul Wadood, Alec Lu, Haikai Zhao, Zhenman Fang, abdul_wadood@sfu.ca, Simon Fraser University, Burnaby, BC, Canada.

ACM 1936-7414/2025/12-ART
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The vast volume of datasets have led to the adoption of various file formats in big data analytics—such as Apache Parquet [10], Avro [7], and ORC [8]—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 allowing the application to retrieve 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 [36], as well as its built-in support for ACID (atomicity, consistency, isolation, durability) transactions [8].

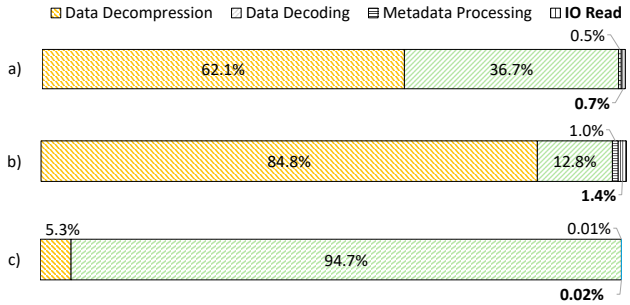

Fig. 1. CPU runtime breakdown of processing ORC file using TPCH dataset: a) Overall Profiling Results, b) Decompression Heavy Columns Profiling Results, c) Data Decoding Heavy Columns Profiling Results

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, Figure 1 presents the breakdown of the CPU execution time for scanning an ORC compressed (Zlib) and encoded TPC-H dataset [41] from an SSD into a consumable table in memory (detailed setup in Section 4.1). The results are shown for three specific cases.

First, Figure 1(a) presents the overall geometric mean breakdown of the CPU time for the TPC-H dataset. In all cases, with the improved I/O bandwidth from the SSD, I/O read occupies less than 1.5% of the total execution time. Most of the processing time is consumed by data decompression, followed by data decoding, indicating these as the primary bottlenecks in the processing pipeline. Second, Figure 1(b) shows the geometric mean breakdown of CPU time for a subset of the TPC-H dataset where data decompression dominates over data decoding. In this case, the ORC encodings used to compress the data provided relatively low compression ratios, with an overall geometric mean compression ratio of only 3x. The results indicate that data decompression consumes a significant 85% of the total processing time, compared to other operations. Finally, Figure 1(c) illustrates the geometric mean breakdown for a subset of the TPC-H dataset where data decoding dominates over data decompression. In ORC, due to the variety of data-specific encodings, some encodings achieve significantly higher compression ratios than others (detailed in Section 4.7). This leads to a shift in the computational bottleneck from decompression to decoding. In our profiling, 28.5% of the dataset was identified as decoding-heavy, with overall compression ratios exceeding 1100x. The results show that in such cases, 95% of the execution time is spent on data decoding, while data decompression and other operations consume minimal time. This issue is also analyzed in the context of Parquet and ORC in [20].

While prior studies [12, 23, 24, 35, 43] have accelerated the data decompression, there is little existing work in accelerating the data decoding stage in ORC and data filtering that is pushed down to file scan stage. This work is the first to integrate all three operators—decompression, decoding, and filtering—and provide end-to-end results for data processing using the ORC file format.

In this work, we focus on accelerating all the data decompression, data decoding and filter stages used in processing ORC files with commodity datacenter FPGAs and its integration with Apache ORC library, which is nontrivial due to the following challenges.

First, ORC supports multiple compression algorithms [8], including Zlib and Zstandard (Zstd), which provide higher compression ratios [25] compared to others. We selected Zlib for our design as Zstd requires significantly larger history sizes and compression block sizes to achieve its compression gains. These larger block sizes can limit the filter pushdown benefits that smaller block sizes provide, which are useful for efficient data processing [18]. Additionally, ORC stores compressed data in a specialized format, adding its own headers to each block while omitting the standard Zlib headers. The offsets of the headers are unknown, introducing a sequential dependency. This introduces a unique challenge in designing a fully pipelined implementation, as the headers must be dynamically processed within a single-port input pipeline. Achieving optimal throughput requires careful resource management to efficiently handle header processing while implementing parallelized decompression at the block level.

Second, an ORC file often uses a (different) combination of multiple encoding schemes, including short repeat, direct, patched base, and delta encoding [8]. 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. Additionally, 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 a fully pipelined design (i.e., pipeline initiation interval of one, $II = 1$). Moreover, the sequential accumulation of the delta decoder (detailed in Section 3.2.2) poses a challenge for a fully pipelined design to decode multiple data items per cycle from a 512-bit wide stream.

Third, filter integration is necessary to reduce the data transfer size of the decompressed and decoded data. However, filter integration into the pipeline adds further complexity.

A fully streaming and pipelined design must ensure that filtering matches the throughput of data decoding or decompression at least; otherwise, the processing elements (PEs) in those operators would be underutilized. The filter module must handle various filter configurations dynamically, including processing data based on pre-stored indices or user-defined filter conditions.

Finally, for big data analytics to transparently leverage the benefits of the FPGA-accelerated file scan, a seamless and efficient integration of the FPGA accelerator into Apache ORC is needed, which could hide the data transfer overhead between the CPU and FPGA, and the IO read.

To address the above challenges, we design and implement FORCv2, the first high-throughput streaming-based FPGA accelerator for processing Apache ORC files used in modern big data engines. FORCv2 incorporates the following novel features:

(1) A resource-efficient, fully dynamic pipelined ORC decoder engine that shares common operations across decoders and supports various combinations of ORC decoding schemes and bit widths. The design achieves an output streaming rate of up to four 512-bit wide streaming writes per cycle. To address the header decoding challenge, it trades off pipeline latency for throughput by buffering and delaying the start of fully pipelined data decoding. For the delta decoding challenge, it employs a resource-efficient tree-based partial accumulation architecture to maintain a fully pipelined design.
(2) A fully dynamic pipelined Zlib ORC decompression header processing engine which trades off pipeline latency for throughput. It leverages a high-performance Xilinx single-PE Zlib decompression PE (processing element) and is extended to a resource-efficient multi-PE architecture for enhanced decompression throughput.

(3) A high-throughput filter design capable of fully utilizing HBM off-chip memory bandwidth, supporting both pre-stored indices and user-defined filter conditions for flexible and efficient data processing.
(4) An end-to-end dataflow integration of our accelerator including decompression, decoder and filter into Apache ORC C++ library [8], which overlaps the IO read, CPU-FPGA data transfer, and FPGA computation.

In this journal paper, we extend our previously developed ORC decoder engine, FORC [42], to include new support for decompression and filtering, enabling end-to-end processing of the ORC file format. Additionally, we integrate FORCv2 with the Apache ORC library, providing a seamless and efficient solution for large-scale data analytics workloads. Our FORCv2 accelerator uses 48.12% of LUTs and a negligible amount of DSPs/BRAMs on the AMD/Xilinx Alveo U280 FPGA. Evaluated on both synthetic datasets and widely used TPC-H datasets [41], our FORCv2 FPGA engine achieves up to ~2.8 GB/s decompression throughput, ~13 GB/s decoding and filtering throughput and overall processing at an input data rate of ~2.8GB/s.

For the end-to-end integration,

FORCv2 achieves a throughput up to the IO bandwidth limit.

Considering all operators combined, on average, FORCv2 achieves end-to-end speedups of approximately 52x for the synthetic dataset and 39x for the TPC-H dataset, versus the ORC C++ library running on a dual-socket system with 48 CPU threads.

2 Background & Motivation

This section provides the necessary background information to understand this work and outlines the motivation behind it. We begin by discussing the fundamentals of

big data analytics, highlighting its significance and impact across various industries. Next, we examine the different storage formats utilized in big data environments, evaluating their strengths and limitations. We then present a comprehensive overview of the Apache ORC file format, focusing on its encoding and decoding schemes, compression and decompression strategies, and its advanced filtering and predicate pushdown capabilities for optimized query performance. Additionally, we explore the motivation for accelerating the ORC format using FPGAs, emphasizing the potential performance gains.

2.1 Overview of Big Data Analytics

We live in the era of big data, with a global population of 8.1 billion. More than 5.44 billion people (~67%) are connected to the internet, generating vast amounts of data every second [2]. By the end of 2024, we will have generated 147 zettabytes (ZB) of data, and this is expected to increase to 181 ZB by the end of 2025 [1]. To provide some perspective, every minute, users generate a staggering amount of data, including 6.3 million Google queries, 41.6 million WhatsApp messages sent, 4 million Facebook posts liked, and 241 million emails sent [38]. These numbers highlight the immense scale of data being created and exchanged globally every single minute.

To handle massive data volumes,

big data analytics, driven by rapid growth in data volume, speed, and variety, enables organizations to extract insights, improve decision-making, and foster innovation [19]. Frameworks like Apache Hive and Apache Spark further support efficient data processing and analysis. However, traditional HDDs often create I/O bottlenecks, limiting performance. To address this, newer technologies such as NVMe and in-memory storage have been adopted, significantly improving data access speeds and reducing latency compared to HDDs [14, 47]. These advancements streamline

data processing workflows, aligning I/O speeds with CPU performance and enhancing the overall efficiency of big data analytics.

2.2 Big Data Formats

Big data storage formats have evolved significantly over time, as shown in Figure 2, to manage structured and unstructured data efficiently. Early formats like CSV and XML provided simplicity and ease of use. However, they fell short in large-scale data processing due to limited compression, lack of schema support, and scalability issues. Introduced in 2001, JSON [30] offered more flexibility in handling semi-structured data, yet it still struggled with large datasets due to storage inefficiencies and slower query performance. To address these problems, formats such as Thrift [37] and Protobuf [11] introduced binary encoding, which provided more compact storage and cross-language compatibility. Although they required predefined schemas, adding some complexity, they delivered higher efficiency for scalable and distributed systems.

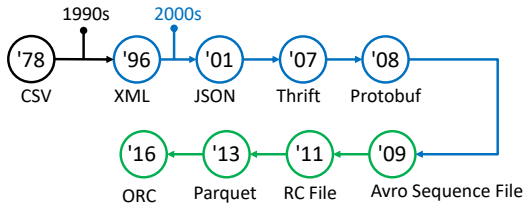

Fig. 2. Big Data Formats Timeline

In today's big data environments, formats such as Avro [7], Parquet [10], and ORC (Optimized Row Columnar) [8] are commonly used. Avro, developed in 2009, is a row-based format optimized for write-heavy tasks. It uniquely combines the schema with the data, allowing for easy schema evolution—a crucial feature in distributed systems where data structures frequently change. By storing the schema in a compact JSON format and the data in binary, Avro ensures both efficiency and compactness. This makes it easy to integrate old and new data without compatibility issues, making Avro ideal for growing datasets and systems with evolving data needs. Additionally, Avro's support for multiple programming languages makes it versatile for various applications, including data transfer.

For analytical workloads that are read-heavy, Parquet and ORC are often preferred because their columnar storage enhances compression and speeds up query performance. Launched in 2013, Parquet is designed for high-performance queries on wide datasets, offering significant compression and quick access to specific columns. ORC, introduced by Hortonworks [15] in 2016, is specifically optimized for Hadoop-based environments, particularly with Hive, and provides even greater storage efficiency than Parquet. Facebook, for instance, uses ORC to save tens of petabytes in its data warehouse, demonstrating a substantial improvement over formats like RCFile or Parquet in terms of speed and efficiency [8]. The excellent compression and query capabilities of ORC make it a popular choice for large-scale data analysis. Both Parquet and ORC support schema evolution, though their effectiveness depends on how they are implemented within the data store.

Table 1 shows the comparison between different big data formats [49].

2.3 Apache ORC File Format

Apache ORC file format was created to enhance the performance of Apache Hive and improve storage efficiency within Apache Hadoop. ORC is a columnar, type-aware and self-describing format specifically designed for large-scale data operations [8]. Using a column-based structure, ORC enables faster data access by allowing systems to only read, decompress, and process the necessary

Table 1. Comparison of AVRO, PARQUET, and ORC formats

	AVRO	PARQUET	ORC
Schema Evolution Support	High	Very Low	Low
Compression Level	Low	Moderate	High
Splitability	Moderate	Moderate	High
Read / Write Heavy	Write	Read	Read
Row / Column	Row	Column	Column

columns for each query, significantly speeding up performance while reducing file sizes. Its type-awareness ensures that the most suitable encoding is used for each data type, and internal indexing further enhances query efficiency. ORC files support predicate pushdown, enabling efficient row filtering using internal indexes. The format is fully compatible with Hive's data types, including complex structures like structs, lists, and maps.

Companies like Facebook, with more than 300 petabytes of data, and LinkedIn were early adopters of the format, which uses ORC with Iceberg and Gobblin for high-performance data queries. Timber uses ORC for its large-scale S3-based logging platform, and HPE Vertica has contributed to the ORC C++ library for SQL-on-Hadoop. Other notable adopters include Apache Gobblin, Apache Nifi, Apache Pig, EEL, and Trino, which leverage ORC for various big data operations, further cementing its place as a leading storage format for performance and scalability in data-driven systems [8].

2.4 ORC File Structure Overview

The ORC file format is structured to provide fast access to specific rows and columns through columnar storage, combined with detailed metadata and indexing mechanisms [8]. An ORC file is divided into stripes, each of which typically contains about 250 MB of data, as shown in Figure 3. This division allows for parallel processing and optimized data access. Each stripe is composed of the following key parts:

(1) **Index Data:** Provides row group-level statistics (e.g., minimum, maximum, sum) for subsets of rows (typically 10,000), along with null flags (HasNull) indicating the presence of null values.
(2) **Row Data (Record Data):** Stores the actual data in a columnar format for efficient access. Present streams indicate null values, while data streams store compressed, encrypted, and encoded values. ORC supports different compression algorithms and uses various encoding schemes tailored to each column's data.
(3) **Stripe Footer:** Contains metadata to decode and navigate data within the stripe, including locations of present and data streams, column encodings, and encryption details, if applicable.

Beyond the individual stripes, ORC files include file-level metadata that summarizes the entire file and helps in its efficient navigation. This metadata includes three main components:

(1) **Metadata:** ORC file metadata contains stripe-level statistics, including minimum, maximum, sum values, and null indicators (HasNull) for each column. These aggregated statistics help optimize queries by identifying relevant stripes for data retrieval, improving query efficiency.
(2) **File Footer:** The file footer provides offsets and lengths for index data, stripe footers, and record data, enabling efficient file navigation. It also includes column-level statistics (e.g., minimum, maximum, sum, HasNull), the total number of rows, and the row index stride, which helps locate specific data efficiently.
(3) **Postscript:** Located at the file's end, the postscript includes information on compression, encryption, ORC writer version, and its length. The last 8 bytes of the file store the postscript length, enabling quick access.

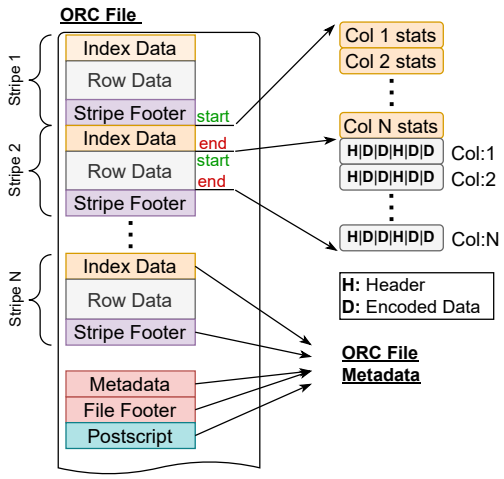

Fig. 3. ORC File Format [8]

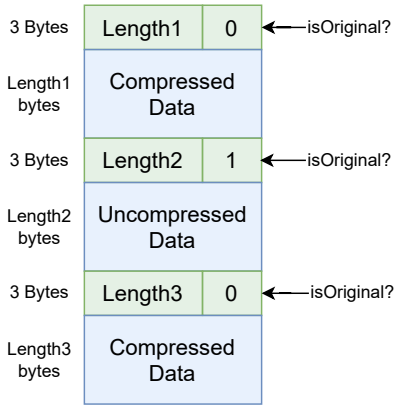

Fig. 4. ORC File Compression Scheme

In this work, we refer to all the file footer, postscript, stripe footer, and index data, except the row data, as ORC file metadata.

2.5 ORC Encoding & Decoding Schemes

Apache Hive version 0.12 introduced an improved run length encoding method, RLEv2, for the ORC file format. This update enhanced compression efficiency and optimized the encoding of data with fixed widths [8]. RLEv2 utilizes four distinct encoding techniques based on the data characteristics: short repeat, direct, patched base, and delta encoding [8]. As shown in Figure 3, the data for each column is divided into batches, each batch employing a specific encoding method and containing a header that records the essential encoding information. The first two bits of this header identify the encoding scheme being used. In the following sections, we will explain the common operations shared by all these encoding schemes and their decoders, followed by operations unique to each specific scheme.

2.5.1 Common Encoding and Decoding Operations. There are two common operations shared by all data encoding schemes [42]:

(1) **Changing Data Endianness**: ORC encodes data in big endian format to maintain compatibility with its Java implementation. However,
x86-64 architectures use little endian for data storage. Therefore, before decoding the data with any encoding scheme, it is necessary to convert the data from big endian to little endian by rearranging the data bytes.
(2) **Zigzag and Unzigzag for Signed Integers**: For signed integer data, ORC converts it into an unsigned integer format using zigzag encoding. This method shifts the sign bit to the least significant bit using the expression `(val<<1)^(val >> [maxBitSize])`. During the decoding process, an unzigzag operation `(value>>1)^(-(value & 1))` is performed before applying the specific decoding scheme.

2.5.2 Scheme-Specific Encoding and Decoding Operations. Table 2 provides examples of each data encoding scheme applied after the common encoding methods have been implemented [42]:

(1) **Short repeat:** It is used to encode short sequences consisting of 3 to 10 repeated values in a batch. Table 2 shows an example of encoding five values, each being 500. In the encoded

Table 2. 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

format, the value is only stored once. The header is one byte in size and contains the following information: (a) The first two bits specify the encoding scheme (with 0 representing short repeat). (b) The next three bits represent the data width (in bytes) of the encoded values. (c) The last three bits denote how many times the value is repeated (i.e., run length).

(2) **Direct:** This encoding method is designed for random sequences with a consistent data width, supporting up to 512 values in one batch. For the encoded data, it directly copies the input data (Table 2). The header consists of two bytes and encodes the following information: (a) The first two bits identify the encoding scheme, with a value of 1 indicating Direct encoding. (b) The next five bits specify the bit width of the encoded values. (c) The last nine bits represent the number of values present in the batch, also known as the run length.

(3) **Patched Base:** This encoding scheme is employed for random sequences that have variable data widths, supporting up to 512 values in one batch. For example, consider the input sequence: 2030, 2000, 2020, 1026010, 2040, where the value 1026010 is 32-bit wide, and the other values are 16-bit wide (refer to Table 2). Patched base encoding is only used if at most 5% of the values in the sequence have variable bit widths; otherwise, direct encoding is used. Encoding this sequence involves three key steps:

(a) **Record a Common Base Value**: Determine and record the smallest number in the sequence, which is 2000 (16-bit).

(b) **Calculate and Record Differences**: Subtract the base value from each number and record these differences as 8-bit values, resulting in the differences: 30, 0, 20, 10, 40.

(c) **Handle Overflowing Differences**: For values like 1026010, where the difference exceeds the 8-bit limit, a patch value for the remaining difference (1024000) must be recorded. This patch value is stored as a pair:

(i) **Index of the Value**: The first element is the index of the value in the original sequence (e.g., 3), referred to as the patch gap width (PGW).

(ii) **High Bits of the Difference**: The second element is the high bits of the difference, calculated by right-shifting 1024000 by 8 bits (resulting in 4000). The bit width of this value, excluding leading zeros, is known as the patch width (PW).

The header for the Patched Base encoding method is four bytes in length and contains following information:

(a) **First Two Bytes**: These follow the same encoding scheme as Direct encoding, with the exception that the bit width (8 bits in this example) pertains to the subtracted difference value.

(b) **Remaining Two Bytes**: These bytes encode additional information:

(i) **Data Width of the Base Value**: The first three bits indicate the data width (DW) of the base value (16 bits in this example).

(ii) **Patch Width (PW)**: The next five bits specify the patch width (PW=12 bits in this example).
(iii) **Patch Gap Width (PGW)**: The subsequent three bits represent the patch gap width (PGW=2 bits).
(iv) **Patch List Length (PLL)**: The final five bits denote the patch list length (PLL=1), indicating the number of patches.

(4) **Delta:** This encoding technique is applied to sequences that are either monotonically increasing or decreasing with a fixed data width, supporting up to 512 values in one batch. In this method, the first value of the sequence is stored as the base value. For each subsequent value, the difference (delta) between it and its immediate predecessor is calculated and stored. Typically, the bit width required for these delta values is smaller than that of the original values, enhancing storage efficiency. The header encoding follows the same structure as described for Direct encoding, except that the bit width specified pertains to the delta values. There are important nuances to consider during delta decoding:
(a) **Decoding the Base Value:** The data width of the base value is not explicitly encoded in the header. The base value is encoded byte by byte, with the MSB acting as a continuation flag. The most significant bit (MSB) of each byte indicates whether more bytes follow. Specifically, the MSB of the highest byte of the base value is set to zero, while the MSBs of all preceding bytes are set to one [8]. During decoding, the decoder reads the base value one byte at a time from the encoded data, continuing until it encounters a byte where the MSB is zero. This process ensures the accurate reconstruction of the base value.
(b) **Decoding the First Delta Value:** Similar to the base value, the first delta value after the base value is encoded using the same MSB-based technique. The MSBs indicate whether additional bytes belong to the current value, allowing the decoder to determine the length of the first delta value without explicit size information in the header.
(c) **Handling Uniform Delta Values (Delta_0b):** In the special case where all delta values are identical, the header encodes the delta bit width as zero, referred to as Delta_0b. This signals that only one delta value needs to be stored. Since the bit width of this delta value is not specified in the header, it is encoded and decoded using the same MSB-based method as the base value.

2.6 ORC Compression & Decompression Schemes

In ORC files, when a generic compression codec is chosen such as Zlib or snappy, everything in the file except for the postscript is compressed using that codec [8]. However, ORC has a requirement that allows readers to skip over compressed sections without needing to decompress the entire stream. To achieve this, ORC compresses data in chunks, each of which has a header, as shown in Figure 4.

For cases where data cannot be effectively compressed (i.e., when the compressed data is larger than the original), ORC stores the original data instead and sets an "isOriginal" flag to indicate this. Each chunk header is 3 bytes long and stores the value of (compressedLength * 2 + isOriginal) in little-endian format. For instance, if a chunk compresses to 256,000 (0x03E800) bytes, the header would be [0x00, 0xD0, 0x07], and for an uncompressed 7-byte chunk, the header would be [0x0F, 0x00, 0x00] [8]. Because each chunk is compressed independently, a decompressor can start decompressing from the beginning of a header without needing to process previous bytes.

By default, the compression chunk size is set to 256 KB, but users have the flexibility to choose a different size. Larger chunk sizes can improve compression efficiency but require more memory. The chunk size is recorded in the postscript, so readers can allocate the right buffer size for decompression, and they are assured that no chunk will expand beyond the defined chunk size.

In cases where no generic compression is used, ORC writes each stream directly without adding headers.

2.7 ORC Filtering & Predicate Pushdown

Predicate pushdown enhances ORC's efficiency by utilizing the metadata to filter out unnecessary data early in the query process. When a query is executed, the system first examines file-level metadata, such as the minimum and maximum values for each column, to determine if entire files or specific stripes should be skipped. This reduces the amount of data read from disk. Additionally, ORC's stripe-level index contains detailed statistics about row groups, allowing further filtering of irrelevant row groups within each stripe. This selective approach minimizes decompression and decoding, leading to faster query execution and reduced I/O overhead. Predicate pushdown filtering also reduces memory consumption, since the filtered data never materializes in the output table.

2.8 Motivation

The ORC file format offers significant storage efficiency, but this advantage comes with the trade-off of increased computational demands for decompressing and decoding the files during downstream query operations. CPU speeds, especially single-thread performance, have remained relatively stagnant, while storage and network bandwidth continue to grow at a pace comparable to Moore's law. In the past, reducing I/O at the expense of additional CPU cycles was almost always worthwhile. Today, however, that trade-off is no longer as effective, and the imbalance will only become more pronounced over time [14]. As illustrated in Figure 1, the I/O read operation accounts for less than 1.5% of the total execution time. However, the real bottlenecks are in the computational tasks: for the TPC-H workload, data decompression takes up 62.1% of the time, while decoding the ORC file consumes another 36.7%. These stages, particularly decompression and decoding, are the primary challenges that call for computational acceleration. An analysis of this issue, specifically in the context of Parquet and ORC, is also presented in [20].

Although prior research has made notable advances in accelerating data decompression, as seen in studies such as [35], [48], [32], [23], and [43], there has been limited progress in accelerating the data decoding phase of ORC files. In [49], Zeng et al. observe that encoding algorithms used in Parquet/ORC are not designed for parallel processing in GPUs as all threads have to wait for the first thread to scan the entire data block to obtain offsets in input and output buffers. Addressing this gap in decoding acceleration is crucial for optimizing overall query performance in ORC-based systems, especially when combined with the gains already achieved in decompression. The use of high-compression algorithms such as Zlib and GZip, while efficient, requires significant CPU resources for decompression. Hardware decompression accelerators on FPGAs usually have single PE implementation and have shown gains ranging from 1x to 3x over CPU-based approaches [32, 43, 48].

Hence, ORC-specific resource and performance efficient hardware acceleration of decompression and decoding kernels to handle the large volume of data are critical for achieving effective performance improvements. It is important to combine data filtering since the data inflates after the decompression and decoding, causing large data transfers.

3 FORCv2 Design & Implementation

In this section, we present the hardware design and implementation of FORCv2, as well as its integration with the Apache ORC C++ software library [9] to decompress, decode and filter ORC files in a fully streaming fashion.

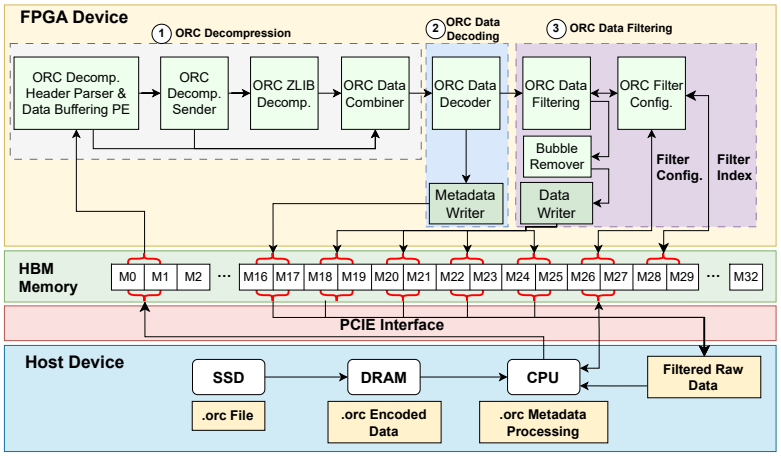

Fig. 5. Overview of FORCv2 Hardware Architecture

To minimize frequent CPU-accelerator communication overhead, we offload the entire ORC processing pipeline onto the FPGA accelerator, including decompression, decoding, filtering, and index processing. To optimize resource utilization for the integration of the full ORC processing pipeline, we employ a resource-efficient overlay design tailored to handle decompression, decoding, and filtering operations within the constraints of the FPGA's hardware resources. Performance-wise, the architecture is fully pipelined and streaming-based, designed to process data at rates comparable to the current limits of SSD. Figure 5 illustrates the full FORCv2 hardware architecture implemented on a commodity HBM-based FPGA device. The architecture consists of three major components:

- **ORC Decompression:** This module, described in Section 3.1, includes the Header Parsing and Data Buffering Engine, which streams 512-bit data per cycle to saturate the memory bandwidth of a single HBM bank. It dynamically parses headers, buffers data, and sends it to the decompression engine. The Decompression Sender and Zlib Decompression modules handle compressed data using 16 parallel decompression engines, while the Data Combiner merges decompressed and uncompressed data in alignment with the parsed headers.
- **ORC Data Decoding:** Covered in Section 3.2, this component handles the decoding of all four data encoders used in ORC. The Data Decoder processes data using pipelined logic for all ORC decoders, including shared operations like unzigzagging and shifting. The Metadata Writer generates and stores metadata related to the decoded data, ensuring compatibility with the ORC format.
- **ORC Data Filtering:** As detailed in Section 3.3, the filtering pipeline integrates the Filter Config and Index Processing module to manage filtering criteria and indices. The Data Filtering PEs perform filtering based on user-defined conditions or stored indices, while tracking the total number of filtered rows. The Bubble Remover consolidates filtered data by eliminating sparsity and preparing it for storage. Finally, the Data Writing module writes the filtered data back into multiple HBM banks using a round-robin strategy to maximize memory bandwidth utilization.

3.1 ORC Decompression

In this section, we present the design and architecture of the ORC data decompression process. We selected the Zlib algorithm due to its superior compression ratio compared to other ORC-supported compression algorithms [13, 31, 40, 48]. To implement this, we leveraged the Xilinx open-source

Zlib library (v2021.2) [43] which achieves up to ~206 MB/s (input) throughput on TPC-H dataset and is on average ~3x faster than single core CPU. We modify it to support ORC-specific decompression configurations. Additionally, we incorporated the implementation of ORC decompression header processing, enabling the efficient decoding of headers. To further enhance performance and optimize storage, we employed 16 parallel decompression units which parallelizes decompression at the block level, facilitating high throughput and improved decompression efficiency.

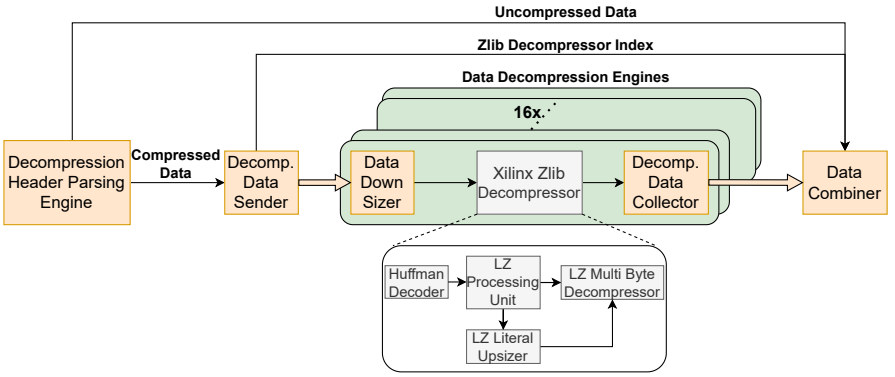

Fig. 6. Overview of ORC Decompression Kernel

The architecture of the ORC data decompression process on a commodity HBM-based FPGA device is illustrated in Figure 6. This architecture is designed to efficiently handle high throughput data processing and maximize memory bandwidth utilization. First, Section 3.1.1 discusses **Decompression Header Parsing Engine**, which streams 512 bits of data per cycle, fully utilizing the off-chip memory bandwidth of a single HBM bank [27, 28]. This engine parses headers and data, sending the processed data to the appropriate modules for further handling. Second, Section 3.1.2 describes the **Decompression Data Sender** engine, which efficiently sends the compressed data to decompression engines in round-robin fashion, enabling the decompression engines to read the data seamlessly. Additionally, it transmits the index of each decompression engine to the Data Combiner, ensuring that the data is correctly ordered in the output. Third, Section 3.1.3 covers the **Data Decompression Engines**, which reduce the data width before feeding it into the decompressor, storing the data in eight 256Kb UltraRAM (URAM) banks to facilitate efficient access and processing during decompression. Finally, Section 3.1.4 describes the **Data Combiner** which aligns and merges the decompressed data based on the header and data sender information, ensuring proper reconstruction of the original ORC file structure.

3.1.1 Decompression Header Parsing Engine. The Decompression Header Parsing Engine is designed to optimize the trade-off between latency and throughput by buffering 1024 bits of incoming data. As the headers are removed from the data sent to the decompression engine, and because a compressed chunk (256KB) can start or end in the middle of the 512 bits read from the HBM bank, data is buffered to enable sending 512 bits of valid data per cycle to the decompression engines. This buffering allows the engine to process headers in parallel while simultaneously streaming data for decompression. The engine begins by reading the first three bytes of the header and examining the last bit to determine whether the data is compressed or uncompressed. It then right shifts these three bytes by one to extract the length of the data. Depending on the compression status, the engine routes the data to the appropriate module for further processing. Additionally, it continuously tracks subsequent headers to ensure efficient and seamless data handling for high-throughput decompression.

3.1.2 Decompression Data Sender. The Decompression Data Sender is responsible for distributing compressed data to each decompression unit in a round-robin fashion. It transmits the data along with information about the valid data bits to ensure accurate processing. Additionally, the engine keeps track of the index of each decompression unit and sends this index to the Data Combiner. This mechanism ensures that the original data order is preserved in the output, maintaining the integrity and correct sequencing of the decompressed data.

3.1.3 Data Decompression Engines. The Data Decompression Engines are designed to handle compressed data efficiently by writing it directly into UltraRAMs (URAMs), allowing the input bandwidth of all 16 decompression engines to be quickly saturated. This setup ensures that the decompression engines can operate in parallel, maximizing throughput. Each decompression engine reads the compressed data from URAM and downsizes it to 16 bits before passing it to the Zlib decompression engine for decompression. The Zlib decompressor produces 64 bits of decompressed data, which are then combined into 512-bit segments. These segments are written back into URAM to prevent stalling and ensure that the decompression engines maintain a continuous and efficient data flow. This parallel processing approach optimizes performance and leverages the FPGA's resources for high-speed decompression.

3.1.4 Data Combiner. The Data Combiner is responsible for aligning the uncompressed and Zlib-decompressed data using the header parsing information and the index details provided by the Decompression Data Sender. By utilizing this information, it ensures the data is correctly ordered and properly sequenced. Once aligned, the Data Combiner outputs 512-bit chunks to the ORC Data Decoder Engine, preserving the original data structure and maintaining high throughput for efficient downstream processing.

3.2 ORC Data Decoding

In this subsection, we will discuss the overall design and architecture of data decoding hardware. We aim to reduce frequent communication between the CPU and accelerator by implementing all ORC decoders in the accelerator. We introduce a resource-efficient overlay design that dynamically accommodates every ORC decoder combination discussed in Section 2.5 and shares as many common operations as possible among them. This strategy helps preserve ample resources for integrating other query accelerators like decompression and filtering. From a performance standpoint, we designed a fully pipelined streaming accelerator architecture that achieves processing rates matching the output memory streaming speed (about 13 GB/s for one HBM channel), which exceeds the current limits of SSD (about 3~3.3 GB/s for our SSD) and PCIe bandwidth (about 10 GB/s for for uni-direction and 5~6 GB/s for bi-direction in our 16-lane gen3 PCIe), ensuring it is ready for future high-bandwidth IO technologies.

Fig. 7 shows an overview of the FORCv2 data decoding hardware architecture on a commodity HBM-based FPGA device. First, Section 3.2.1 covers **Part 1: Header & Data Parsing Engine**, which streams 512-bit data per cycle, maximizing the off-chip memory bandwidth of a single HBM bank [27, 28]. It dynamically parses headers, buffers, and aligns data for every ORC decoder type, incorporating delayed processing in a fully pipelined setup. Second, Section 3.2.2 discusses the fully pipelined **Part 2: Data Decoding** logic for all four decoders and their shared data shift/unzigzag unit, featuring hardware optimization techniques to conserve resources and boost throughput. Third, Section 3.2.3 outlines **Part 3: Data Selector**, used to prepare and stream decoded (and often inflated) raw data for filtering.

3.2.1 Header & Data Parsing Engine. As depicted in Fig. 8, to fully utilize off-chip memory bandwidth [27], the system streams in one 512-bit data per cycle, storing it in a shift register for

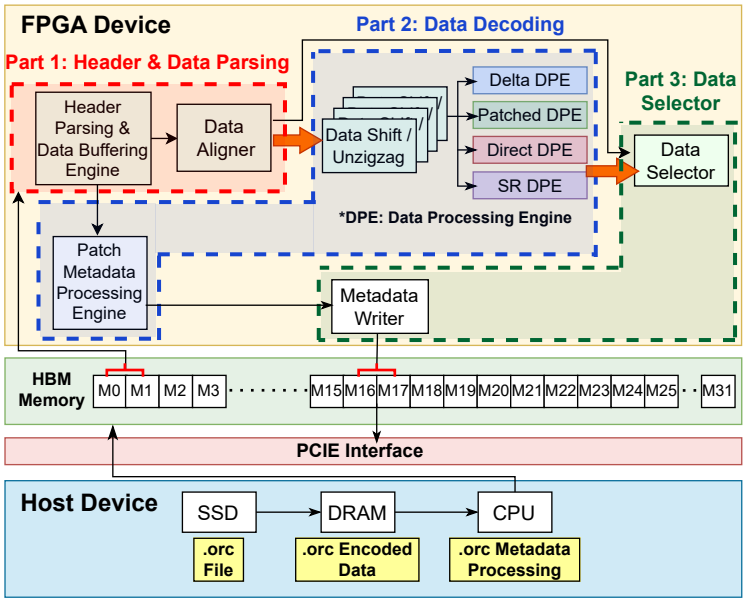

Fig. 7. Overview of FORCv2 data decoding hardware architecture

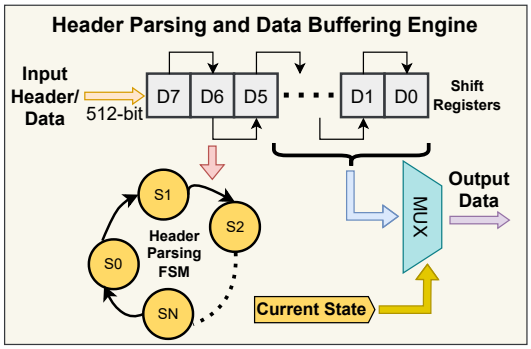

Fig. 8. Header parsing and data buffering engine. Each data (D) block is 512-bit, i.e., 64 bytes.

processing in a pipelined manner. The header parsing finite-state machine (FSM) continuously scans the shift register, which holds both header and encoded data. Upon detecting a header, the FSM requires six cycles to decode the following: 1) encoding scheme, 2) data width, 3) run length, and 4) scheme-specific information for patched base and delta encoding schemes. In the subsequent cycle, the buffered encoded data and decoded header information are forwarded to the data aligner. It is important to note that the first data cannot be transmitted until the header is fully parsed (pipeline depth = 7), but once parsed, all subsequent data flows through in a fully pipelined manner (II=1).

Data Aligner: When the encoded data (which is unaligned due to the header taking up some bytes in the stream) and the decoded header information are received, the data aligner first processes the encoded data by partitioning and repacking it according to the data width. It then forwards the data to the data shift/unzigzag units. In the specific case of a delta encoder (Delta_0b), where all delta values are identical and only one delta value is encoded, the aligner duplicates the delta value

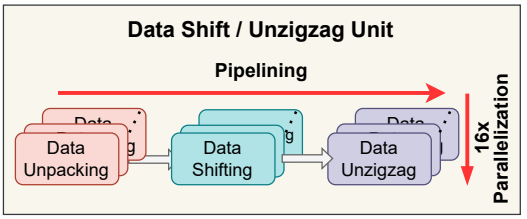

Fig. 9. Data Shift and Unzigzag Unit

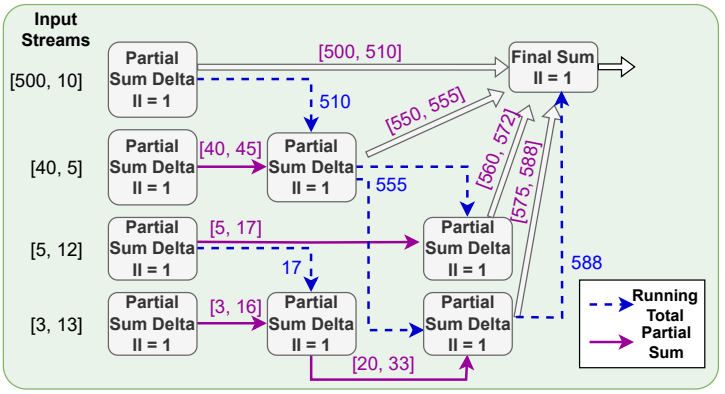

Fig. 10. Fully pipelined delta decoding engine design

and sends it to each data shift/unzigzag unit. Additionally, the data aligner transfers the decoded header information to the data selector for decoded data alignment (Section 3.2.3).

3.2.2 Data Decoding Engines. Here we will discuss the data decoding engines.

Data Shift/Unzigzag Unit: After receiving the aligned data, this unit unpacks the 512-bit data stream into multiple items based on the decoded data width (e.g., 16 items of 32-bit data). It then converts the data's endianness to little-endian format and performs the unzigzag operation, as detailed in Section 2.5.1. To ensure that 512 bits of encoded data can be processed per cycle, these operations are pipelined and parallelized by a factor of 16. To handle 8-bit encoded data widths, four of these units are used, as shown in Figure 9.

Delta Decoding PE (Processing Element): In delta decoding, the input delta values are sequentially accumulated with the running sum from their immediate predecessors, which creates a challenge when decoding multiple data items (e.g., 16 32-bit items) per cycle from a wide data stream. Fig. 10 illustrates how the encoded values from Table 2 are decoded.

To maintain a dynamic initiation interval (II) of 1, the design is optimized to compute multiple partial sums on sub-streams in parallel across several stages. In the first cycle, each data stream calculates its partial delta sum (e.g., [500, 10] becomes [500, 510] and [40, 5] becomes [40, 45]). The partial delta sum from the first stream is then sent to the final sum unit, where it is added to the previously stored running sum (which is 0 at the start of the stream with a base value of 500) before being passed to the data writing stage. Simultaneously, the next partial sum units in the pipeline add the running total from stream one to the partial sums of stream two (e.g., the running total of 510 from stream one is added to stream two's partial sums of [40, 45], resulting in [550, 555]). This process continues for all streams, fully pipelining the accumulation with an initiation interval of 1. To further minimize the number of pipeline stages, partial sum units are added between every two neighboring streams.

Patched Base Decoding PE: Four summation units operate in parallel to add the base value to the difference values received from the data shift/unzigzag unit. The resulting sums are then passed on to the data writing stage.

Patched Metadata PE: As explained in Section 2.5.2, for each value with a larger bit width, a patch index and value pair is encoded. The patched metadata processing engine (PE) receives the patch list directly from the header parsing and data buffering engine, decoding it to extract each patch index and patch value (representing the higher bits of the remaining difference). The patch value is then left-shifted by the bit width (number of lower bits) decoded from the header to reconstruct the full remaining difference. These newly decoded pairs are then sent to the metadata writer.

Rather than adding this remaining difference to the sum obtained from the patched base decoding PE on the FPGA, the final addition is left for the host CPU, which performs this step when reading the data. This approach is taken because the patched metadata PE can only start processing the patch list after all the regular difference values have been read and processed, due to the streaming nature of the system. In practice, patched base encoding is only applied when at most 5% of the values in the sequence have variable bit widths; otherwise, direct encoding is used. As a result, this design choice does not cause a performance issue in the overall integration.

Short Repeat (SR) Decoding PE: In this PE the data is duplicated according to the specified run length. This means that each data item is repeated the required number of times based on its run length, ensuring accurate reconstruction of the original data sequence. Once duplicated, the data are forwarded to the data writing stage for final processing and storage.

Direct Decoding PE: The data output from the shift/unzigzag units is considered the fully decoded data and is then passed on to the data writing stage.

3.2.3 Data Selector: This unit preserves the order of the decoded raw data by utilizing the decoded header information provided by the data aligner. It then sends the decoded 512-bits packets to ORC data filtering.

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

3.3 ORC Data Filtering

The ORC filter engine provides robust support for a wide range of filter operations, including <, <=, >, >=, ==, and <> (i.e., not equal to) on both the left and right operands. This capability efficiently handles common filtering scenarios, covering approximately 85% of the filter conditions used in TPC-H[41] queries, such as conditions of the form "LO < # < RO", which means the number is greater than left operand and is smaller than right operand. The engine also allows for filtering based on either specific filter conditions or leveraging index-based filtering, enhancing flexibility and performance. One of the key advantages of this design is that data indices do not need to be transferred back to the host. Instead, they are stored in High-Bandwidth Memory (HBM) and can be reused for subsequent rounds of filtering if needed, because of the columnar nature of ORC (a selection can be valid for several columns). This approach significantly reduces data movement overhead and improves the efficiency of repeated filtering operations, making the ORC filter engine well-suited for high-performance, data-intensive applications.

Figure 11 illustrates the hardware architecture design for FORCv2 data filtering. The architecture is organized into distinct sections, each focusing on a specific aspect of the filtering process. Section 3.3.1 covers the implementation of the filter configuration and the collection and storage of filtered indices. Section 3.3.2 describes the process of data filtering based on the specified filter conditions. Section 3.3.3 addresses the removal of sparsity in the data using a bubble remover to ensure a

smooth and efficient data flow. Finally, Section 3.3.4 details the round-robin approach for writing the data to HBM banks, optimizing memory bandwidth utilization and balancing data distribution.

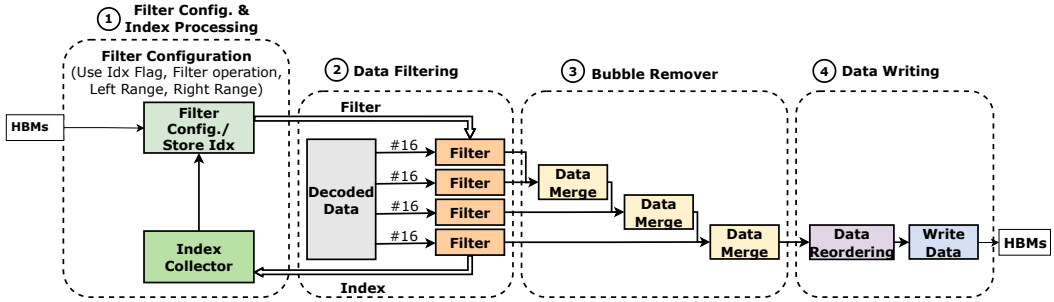

Fig. 11. Overview of FORCv2 data filtering hardware architecture

3.3.1 Filter Config and Index Processing. This module is responsible for reading the user-defined filter configuration from the HBM bank. This configuration consists of two main components: the **index flag** and the **filter condition**. The index flag determines whether the filtering operation will be based on pre-stored indices or the user-specified filter condition. Using this information, the module dynamically adapts its behavior to ensure seamless integration with the data filtering pipeline. The user needs to write a 12-byte configuration field, which determines the functionality of the filter. Table 3 presents the details of the filter configuration.

Table 3. Filter Configuration Register

1bit	15bits	1Byte	1Byte	4Bytes	4Bytes
Idx Flag	Don't Care	Right Range OP (RROP)	Left Range OP (LROP)	Right Range (RR)	Left Range (LR)

If the index flag is set, the module retrieves the pre-stored indices from HBM and streams them directly as filtering criteria to the data filtering modules. In this case, no new indices are written back to HBM as the filtering indices won't change. However, if the index flag is not set, the module streams the user-defined filter condition to the data filtering modules. As the filtering progresses, the module collects the filtered indices generated by the data filtering modules and writes them back to HBM for future use.

3.3.2 Data Filtering PEs. This module is responsible for filtering data based on the decoded data input received from the data decoding module and the filter configuration provided by the Filter Config and Index Processing module. Using this configuration, the data filtering PEs determine the filtering criteria and apply the appropriate operations. If filtering is performed based on the user-defined filter condition, the PEs generate filtered indices and send this information back to the Filter Config and Index Processing module for further handling. Alternatively, if the filtering is based on pre-stored indices, the PEs directly process the data without generating new index information.

Additionally, the Data Filtering PEs monitor and record the total number of rows that have been filtered. This information is written back to the Filter Config port, providing the user with access to the row count as part of the filtering results. This mechanism enables more efficient data transfer to the host by ensuring that only the filtered data is transmitted, reducing unnecessary overhead and optimizing performance.

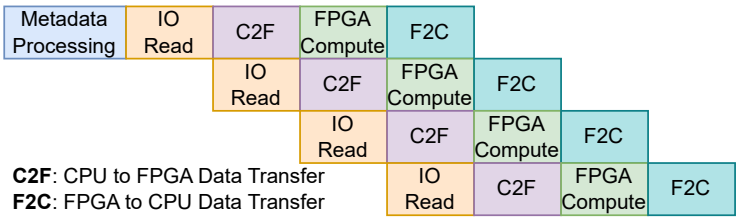

Fig. 12. Dataflow integration of FORCv2 into ORC C++

3.3.3 Bubble Remover. This module is responsible for eliminating sparsity in the data received from the Data Filtering PEs. Using a hierarchical approach, it removes the gaps between the data by shifting them to the end of the stream. As illustrated in the diagram, the module consolidates data streams at multiple levels, progressively merging and compacting the filtered data. This process ensures that valid data is densely packed at the beginning of the stream, with any gaps or unused portions pushed to the end. Once the data is fully compacted and organized, it is forwarded to the Data Writing module for further processing. This approach optimizes data flow and ensures efficient utilization of memory and bandwidth.

3.3.4 Data Writing. This module receives the densely packed data stream from the Bubble Remover module and reorders it for efficient storage. Using a round-robin approach, the module writes the data across four HBM output ports while maintaining the original order of the data. This ensures that the data is distributed evenly across the HBM banks, optimizing memory bandwidth utilization and improving storage efficiency.

3.4 End-to-End Integration

To transparently leverage our accelerator in big data analytics, we integrate FORCv2 into the Apache ORC C++ library [9]. Figure 12 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 are connected to two HBM banks to enable the ping-pong overlapping. As a result, FORCv2 utilizes a total of 16 HBM banks: two for input data, eight for inflated output data, two for filter configuration, two for storing filter indices, and another two for the patch list metadata. Note the metadata 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 FORCv2 as:

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

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.

3.5 Design Discussion

Our FORC design currently supports processing a single column within a stripe and decoding of non-nullable integer RLEv2 encoding, which is used for almost all the data types in ORC [8]. For testing, we validated our design with integer data type columns as it requires RLEv2 decoding only. To fully support all data types, we will need to add Boolean RLE decoding, handling of string data, and unbounded base-128 variant streams in the FPGA as additional data streams, alongside RLEv2

decoding. In our end-to-end evaluation, we process all the stripes of the same column in a dataflow fashion, achieving a throughput close to the I/O and/or PCIe bandwidth.

4 Results & Analysis

4.1 Experimental Setup

Benchmark Datasets. First, we evaluate FORCv2 and its individual operators—ORC Decompression, Decoding, and Filtering—using a synthetic dataset. The synthetic dataset consists of columns, each containing only a single ORC encoding scheme. Data is generated for various data widths, ranging from 8-bit to 32-bit, to assess the achievable throughput across different configurations. Then we evaluate each one of those accelerator kernels on the widely used TPC-H [41] 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. Table 4 shows the TPC-H dataset columns used for collecting the results. Due to the fully dataflow nature of our designs, increasing the dataset size beyond 10GB does not affect the achieved end-to-end throughput.

Table 4. Columns used for getting TPC-H dataset results

TPC-H Tables	Columns Used
Lineitem	C0-C4, C10-C12
Orders	C0-C1, C4, C7
Curstomer	C0, C3
Part	C0, C5
PartSupp	C0-C2
Supplier	C0, C3

Hardware and Software Setup. We evaluate our designs on the AMD/Xilinx Alveo U280 datacenter FPGA board (with 32 HBM2 banks) [46]. FORCv2 is developed entirely using Vitis HLS, and the accelerator is synthesized using Vitis 2021.2 [45] and an open source floorplanning optimization tool for task-parallel HLS designs called TAPA/Autobridge (Ver.0.0.20221113.1) [16][17]. Without TAPA/AutoBridge, Vitis cannot route the design. With TAPA/AutoBridge floorplanning, it achieves 182MHz frequency for on-board execution.

For the CPU implementation, we evaluate the well-optimized Apache ORC C++ library v1.8.6 [9] on two Intel Xeon Silver 4214 2.20GHz CPU sockets with data stored on a Samsung NVMe SSD with 3.0 to 3.3 GB/s of sequential data read bandwidth. By default, Apache ORC C++ library utilizes all the 24 cores with 48 hyper-threads. The FPGA board is connected to the Xeon CPU via 16-lane PCIe Gen 3, which achieves around 10GB/s bandwidth using Vitis.

4.2 Resource Utilization & Power Consumption

Table 5 provides a detailed breakdown of the post-place-and-route resource utilization of FORCv2 on the Alveo U280 FPGA. The complete design utilizes 48.12% of LUTs, 30% of URAMs, 17.28% of FFs, 6.9% of BRAMs, and a negligible amount of DSP resources. Among the components, Zlib decompression accounts for the highest resource usage, consuming 24.61% of LUTs and 30% of URAMs. In contrast, the ORC Decoder and Filter modules utilize moderate amount of resources, collectively consuming 23.52% of LUTs, 5.98% of FFs, and negligible amounts of DSP, BRAM, and URAM resources.

Table 5. Resource utilization breakdown on Alveo U280

	LUTs	FFs	BRAMs	URAM	DSP Blocks
ORC_DECOMP_Header_Proc	1.12%	0.27%	0.00%	0.00%	0.00%
ORC_DECOMP_Data_Sender	1.36%	0.28%	0.00%	0.00%	0.00%
ORC_DECOMP_Decomp_Engine	20.85%	10.57%	6.62%	30.00%	0.00%
ORC_DECOMP_Data_Combiner	1.28%	0.18%	0.00%	0.00%	0.00%
ORC_DECOMP	**24.61%**	**11.30%**	**6.62%**	**30.00%**	**0.00%**
ORC_DECODER_Header&Data_Proc	6.23%	1.28%	0.00%	0.00%	0.03%
ORC_DECODER_Data_Decoding	7.79%	1.78%	0.00%	0.00%	0.00%
ORC_DECODER_Data_Selector	2.04%	0.59%	0.00%	0.00%	0.00%
ORC_DECODER	**16.06%**	**3.65%**	**0.00%**	**0.00%**	**0.03%**
ORC_FILTER_ConfigIndex_Proc	0.27%	0.15%	0.17%	0.00%	0.00%
ORC_FILTER_Data_Filtering	3.62%	1.03%	0.00%	0.00%	0.00%
ORC_FILTER_Bubble_Remover	2.81%	0.92%	0.17%	0.00%	0.00%
ORC_FILTER_Data_Writing	0.77%	0.24%	0.00%	0.00%	0.00%
ORC_FILTER	**7.46%**	**2.34%**	**0.28%**	**0.00%**	**0.00%**
Overall FORC	**48.12%**	**17.28%**	**6.90%**	**30.00%**	**0.03%**

Finally, the power consumption is measured using vendor *xbutil* tool. The U280 FPGA board consumes ~26.2W static power and our design consumes maximum ~10.1W dynamic power. The thermal design power (TDP) of Intel Xeon Silver 4214 is 85W.

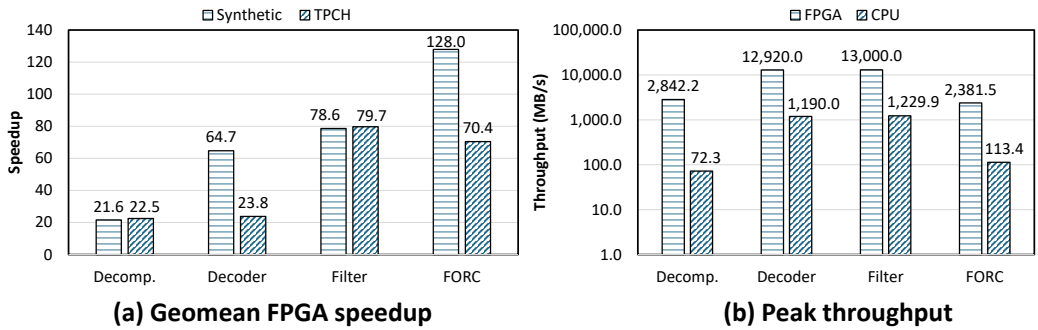

Fig. 13. FPGA execution geomean speedup over CPU and Peak Throughput

4.3 Geomean FPGA Speedup and Peak Throughput

Figure 13(a) summarizes the geometric mean performance speedup of the FORCv2 FPGA engine compared to the CPU implementation. The evaluation highlights the individual speedup gains achieved for each operator—decompression, decoding, and filtering—as well as the overall FORCv2 speedup. From the figure, we observe that the decompression and filtering speedup are similar for both datasets, achieving approximately ~22x and ~79x, respectively. However, the decoding speedup varies between the two datasets, with the synthetic dataset achieving a ~65x speedup compared to ~24x for TPC-H. This difference in decoding speedup is due to the varying encodings used in these datasets, as discussed in Section 4.5. Consequently, this variation also contributes to the difference in overall FORCv2 speedup, with the synthetic dataset achieving ~128x and TPC-H achieving ~70x.

Figure 13(b) illustrates the peak throughput achieved by each operator on the tested datasets. The decompression module achieves a throughput of approximately ~2.8 GB/s, which is close to the bandwidth of a single SSD in the evaluated technology generation, while the CPU achieves a

maximum of ~72 MB/s. This significant performance gain is achieved by leveraging 16x parallelization at the compressed block level, which optimally utilizes the FPGA resources. A single Xilinx kernel achieves up to ~206 MB/s on TPC-H dataset, and through parallel execution at the block level, the throughput scales to ~2.8 GB/s i.e., ~14x faster. The performance gain is not 16x as the data is sent sequentially on block level to all 16 decompression engines.

Both the ORC decoder and filter modules achieve a peak throughput of approximately 13 GB/s, fully saturating the memory bandwidth of one HBM bank. In comparison, the CPU achieves a maximum throughput of only ~1 GB/s. The overall FORCv2 implementation achieves a throughput of approximately 2.4 GB/s, similar to the decompression module. However, it is slightly lower due to the achieved operating frequency of 182 MHz for the full design, compared to 210 MHz achieved by the decompression-only implementation.

These results demonstrate the efficiency of the FPGA design in fully utilizing available hardware resources, delivering throughput close to SSD bandwidth while significantly outperforming the CPU. Note that the reported throughput values represent the FPGA execution time only, excluding I/O reads and data transfers. A detailed analysis of end-to-end throughput, including I/O considerations, will be provided in Section 4.8.

In the following sections, we provide an in-depth discussion of the results for each operator—decompression, decoding, and filtering—as well as the overall performance of the FORCv2 engine.

4.4 ORC Decompression Results

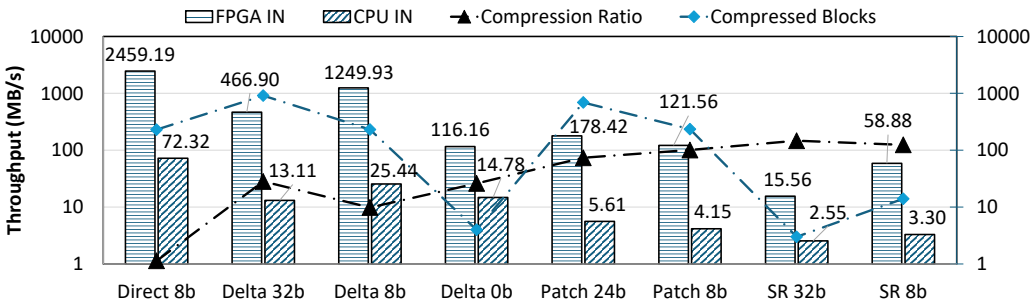

Fig. 14. Decompression throughput comparison for synthetic dataset

We implemented a fully streaming ORC Zlib decompression hardware, as described in Subsection 3.1 achieving 210 MHz frequency, when built individually. Figure 14 shows the decompression throughput achieved by the FPGA compared to the CPU for the synthetic dataset, along with the compression ratios *(CR = output size / input size)* and compressed blocks obtained using ORC Zlib compression. The throughput is calculated based on the input size of the dataset. It is important to note that the Direct 32-bit dataset is not included in the figure, as it was not compressed by the ORC Zlib compression. In this case, the compressed data size was equal to or larger than the original dataset, making compression ineffective. As a result, ORC stored the data in its uncompressed form.

We also observe that in general the decompression throughput decreases as the compression ratio increases. The reduction in throughput is mainly caused by backpressure from the large inflated data size. In addition, when the block size falls below 16 due to higher compression ratios, it prevents efficient utilization of the decompression units, further impacting performance.

In our **synthetic** dataset as shown in figure 14 Direct 8b achieves highest throughput of ~2.5 GB/s. Delta 32b and Delta 0b have similar compression ratios; however, due to the lower number

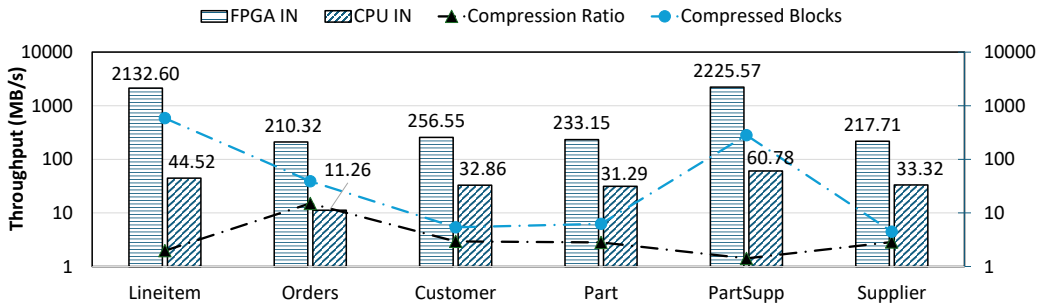

Fig. 15. Decompression throughput comparison for TPCH dataset

of compressed blocks, i.e., 4 in Delta 0b, we observe a lower throughput compared to Delta 32b. Similarly, the SR 32-bit dataset achieves the highest compression ratio of 146x and has a throughput of 15 MB/s. This low throughput is caused by a combination of high backpressure from the large inflated data size and the presence of only three compressed blocks, which limits parallelism and reduces the efficient utilization of the decompression units. Despite these limitations, the SR 32-bit dataset still achieves a 6x speedup on the FPGA implementation compared to the CPU counterpart. It is important to note that the decompression throughput is solely influenced by the compression ratio and the number of compressed blocks, which are data-specific rather than encoding scheme-specific. While SR-24b is not shown in the table, our testing indicates that it achieves a throughput of approximately 1.9 GB/s due to its relatively lower compression ratio and a higher number of compressed blocks.

Similarly, Figure 15 presents the throughput comparison, compression ratio, and compressed block sizes for the **TPC-H** dataset. We observe a similar behavior as seen with the synthetic dataset. We see PartSupp table achieves highest geomean throughput of ~2.2 GB/s. In our profiling with lineitem table, the columns 10, 11 and 12 achieved highest throughput of ~2.8 GB/s and geomean throughput of ~2.1 GB/s across all columns. In contrast with synthetic data, the primary factor contributing to low throughput in TPC-H dataset is the high compression ratio, rather than the number of compressed block sizes. This indicates that the large inflated data size resulting from high compression ratios creates significant backpressure, impacting the throughput more than the block count.

4.5 ORC Data Decoding Results

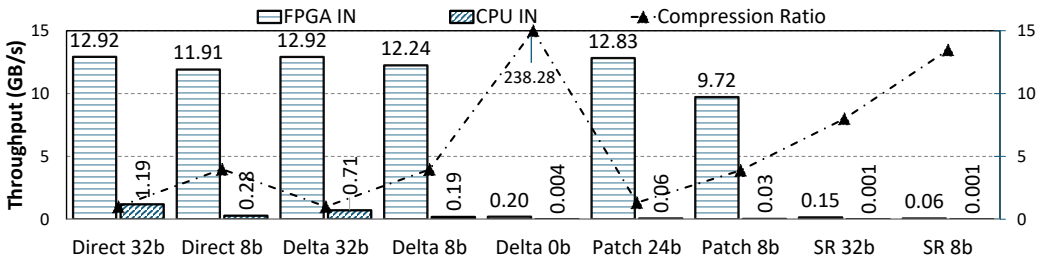

Fig. 16. Throughput comparison of direct, delta, patched base, and short repeat decoders under different bit widths

The ORC data decoding implementation as discussed in Subsection 3.2 achieves a 222MHz frequency, when built individually. Figure 16 compares the throughput of each decoder on the

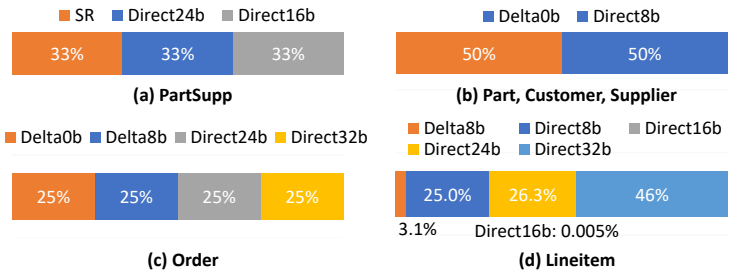

Fig. 17. TPC-H dataset decoding distributions

CPU and FPGA. 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 achieved by the encoders.

Direct Decoding Throughput. For Direct_32b, the compression ratio is one. The input and output throughput of FORCv2 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.

For intermediate bit widths, the input throughput of ORC Decoder 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). Despite this it is still over ~50x 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.

TPC-H Decoding Combination and Throughput. We also evaluate decoder on the industrial standard TPC-H dataset, which uses a combination of different decoding schemes and bit widths, as shown in Fig. 17. Fig. 18 compares the effective decoding throughput on FPGA and CPU, which is decided by the decoding combination in Fig. 17. On average, ORC Decoder 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

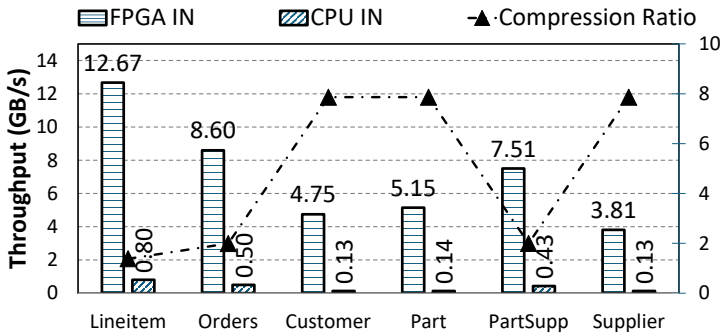

Fig. 18. TPC-H dataset FPGA decoding throughput

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.

4.6 ORC Data Filtering Results

Due to the fully streaming and pipelined nature of our filter design, as discussed in Section 3.3, it achieves a throughput of 13 GB/s for both the datasets across different filtering rates, fully utilizing the FPGA's one HBM bank bandwidth and achieving 225 MHz frequency, when built individually. In contrast, the filtering throughput on the CPU is highly dependent on the filtering rate of the data. This behavior arises from the fact that the ORC format employs filter pushdown at the row group (RG) level which the FPGA does not, allowing it to decide at runtime whether specific row groups need to be decompressed and decoded for filtering after the previous row group has been completely decompressed and decoded. The FPGA does not employ row group level filtering as the whole flow is completely pipelined and skipping a certain row group may degrade the performance due to its sequential nature. It will require additional resource to identify and stop the processes for that row group in all the pipeline stages and then synchronize again for the new incoming row group data. To evaluate this behavior, we profiled our results for different filtering rates ranging from 10% filtering (F10%), where only 10% of the total data is filtered to 90% filtering (F90%) to provide a range of achievable throughput. Due to space constraint only these F10% and F90% are included in the results.

Figure 19 presents the results for the synthetic dataset. For a 10% filtering rate, we observe that the number of row groups processed by the ORC filter remains close to 100% for all cases, except for Delta-encoded data. The achieved throughput in these cases is also similar. This difference in Delta encoding can be attributed to its use for monotonically increasing or decreasing values, where filter pushdown can effectively skip unnecessary row groups. This effect becomes more prominent at the 90% filtering rate, where the percentage of row groups processed drops to approximately 11%, and the CPU's achieved throughput increases to ~1 GB/s.

Figure 20 shows the results for the TPC-H dataset. A similar trend is observed here: as the percentage of processed row groups decreases, the achieved throughput increases.

4.7 Overall FORCv2 Results

FORCv2 design as shown in Figure 5, achieves a frequency of 182 MHz. First, we present the throughput results for the table scan operation, which includes decompression and decoding, without any filtering applied. Since the FPGA throughput is not affected by the filtering rate due to its fully pipelined and streaming design, only the CPU results are impacted by varying filtering

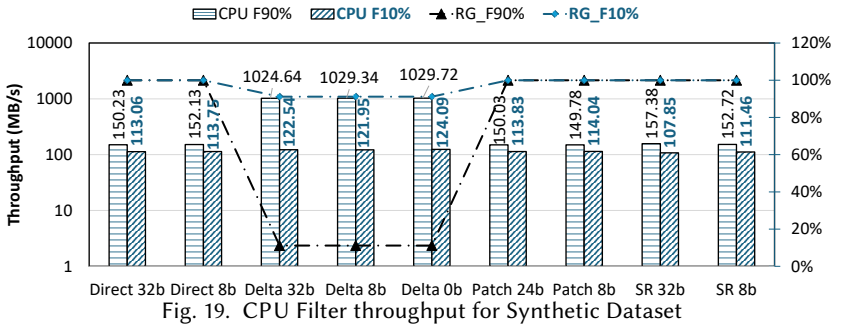

Fig. 19. CPU Filter throughput for Synthetic Dataset

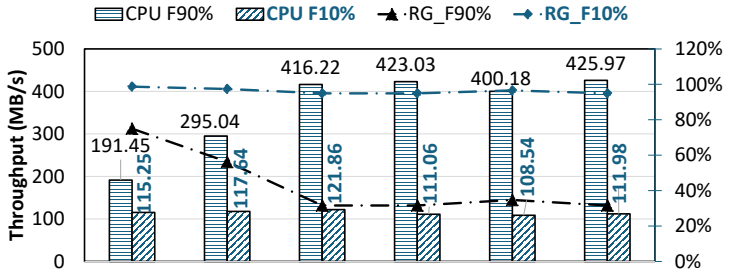

Fig. 20. CPU Filter throughput for TPCH Dataset

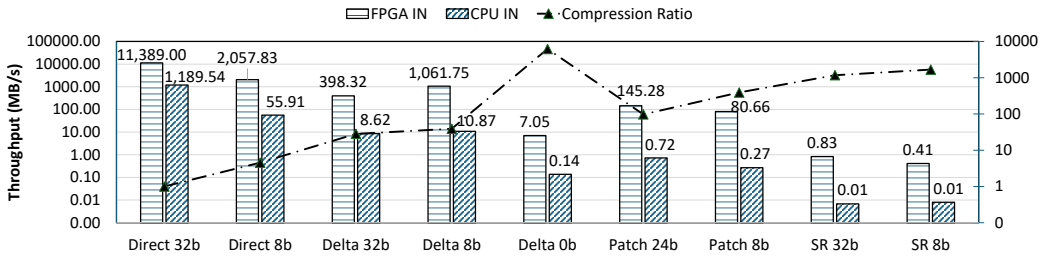

Fig. 21. Table Scan Throughput comparison of FORCv2 for synthetic dataset

rates. Following this, we provide the speedup numbers for different filtering rates to highlight the performance variations.

4.7.1 Table Scan Results Synthetic Dataset: The performance comparison of the table scan operator between the FPGA and CPU for the synthetic dataset, along with the overall compression ratios achieved using Zlib compression and ORC encodings, is shown in Figure 21. From the figure, we observe that the Direct 32-bit dataset (encoding free) achieves a throughput of approximately 11 GB/s on the FPGA. This high throughput is because the data in this case is uncompressed and passes directly to the decoder after decompression header processing. In contrast, the CPU achieves a maximum throughput of only 1 GB/s. The Direct 8b dataset achieves a throughput close to the decompression throughput, delivering a 37x speedup over the CPU implementation.

A general trend is that higher compression ratios tend to result in lower throughput. However, an exception to this behavior is seen in the Delta 32b and Delta 8b datasets. For these datasets, the overall throughput does not align strictly with the compression ratio. This behavior can be explained by the streaming nature of the FORCv2 design, where the operator with the lowest throughput dictates the final throughput of the entire pipeline.

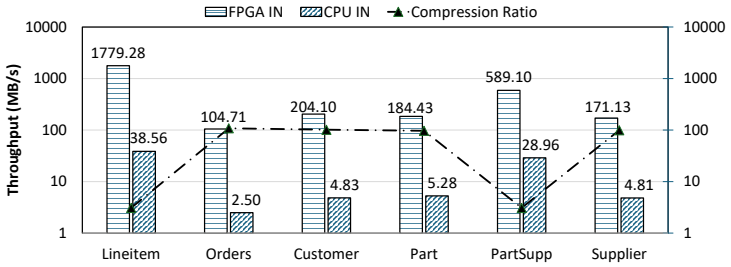

Fig. 22. Table Scan Throughput comparison of FORCv2 for TPCH dataset

In the case of Delta 32b and Delta 8b, the Zlib compression ratios differ significantly. After Zlib compression, Delta 32b achieves a compression ratio of approximately 28x, whereas Delta 8b achieves ~10x. On the other hand, when considering ORC encoding alone, Delta 32b achieves a compression ratio of only ~1x, while Delta 8b achieves around ~4x. Despite these differences in compression ratios, the decoding throughput remains similar for both Delta 32b and Delta 8b datasets. However, the Zlib compression ratio influences the decompression performance, and as a result, the final throughput achieved by the FPGA is primarily determined by the decompression stage.

The Delta 0b dataset achieves the highest overall compression ratio of over 6200x, which introduces significant backpressure on the design. Despite this challenge, the FPGA implementation still achieves an impressive 50x speedup over the CPU implementation. Similarly, the Short Repeat 32b and Short Repeat 8b datasets achieve compression ratios exceeding 1000x. The considerable backpressure caused by both the decompression and decoding stages results in lower throughput for these datasets. Nevertheless, the FPGA implementation delivers 83x and 43x speedup over the CPU for Short Repeat 32b and 8b, respectively. The Patch 24b and Patch 8b datasets achieve compression ratios of approximately ~97x and ~392x, respectively, resulting in ~202x and ~299x speedup over the CPU implementation.

Overall, the FORCv2 design achieves a 68x speedup over the CPU implementation for the synthetic dataset.

4.7.2 Table Scan Results TPC-H Dataset: Figure 22 presents table scan operator performance results for the TPC-H dataset. The lineitem table achieves a throughput of approximately 1.7 GB/s on the FPGA due to its high decoding throughput encodings and greater decompression bandwidth, resulting in a 46x speedup over the CPU. In contrast, the customer, part, and supplier tables contain the Delta 0b encoding, which has a high compression ratio but low decoding and decompression bandwidth. As a result, these tables achieve a lower throughput of approximately 200 MB/s, though they still deliver a significant 100x speedup over the CPU. The partsupp table contains Short Repeat encodings with varying bit widths, which are characterized by low decoding throughput. Consequently, it achieves a lower throughput compared to the Direct-encoded datasets but still performs 20x faster than the CPU. Lastly, the order table contains a mix of encodings, leading to a throughput that falls somewhere in between the other tables, delivering an impressive 42x speedup over the CPU.

4.7.3 Overall Geomean Speedup with Filtering: Figure 23 presents the overall geomean speedup of FORCv2 for both the datasets for table scan operation and various filtering rates. The results clearly show that including the filter operation significantly increases the FPGA's speedup compared to the CPU. This is because the CPU executes all operations sequentially, whereas the FPGA's fully pipelined architecture allows concurrent execution. Additionally, the figure highlights that as the filtering rate increases, the speedup achieved by the FPGA decreases. This is due to the CPU's

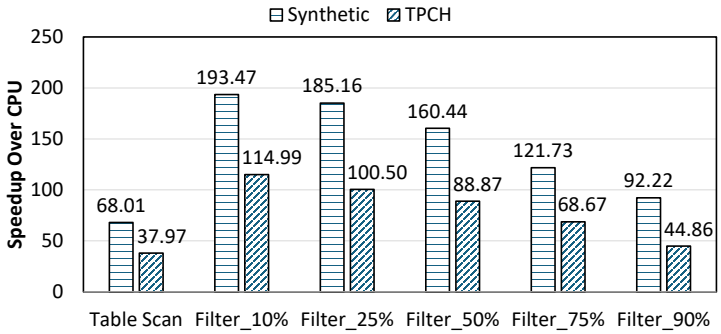

Fig. 23. Geomean FORCv2 Speedup for FPGA execution

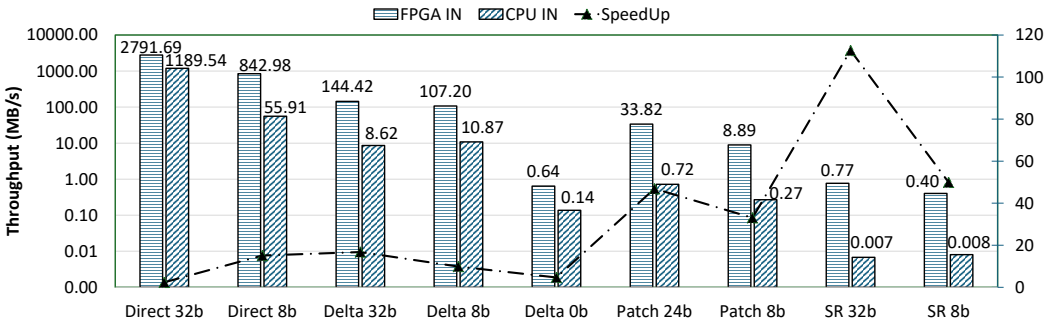

Fig. 24. End-to-End FORCv2 table scan preformance results for synthetic dataset

ability to leverage ORC's row group-level filter pushdown, which allows it to skip certain row groups that do not satisfy the filter condition. This optimization becomes increasingly effective at higher filtering rates, reducing the CPU's workload. Despite this, FORCv2 achieves an impressive geometric mean speedup of approximately 128x over the CPU for the synthetic dataset and 70x for the TPC-H dataset.

4.8 End-to-End Integration Results

In the end-to-end evaluation, we account for SSD read latency and host-to-FPGA data transfer via PCIe. The peak achievable throughput is therefore constrained by a combination of SSD bandwidth, FPGA compute performance, and/or PCIe bandwidth.

4.8.1 Synthetic Dataset Results: Figure 24 presents the end-to-end throughput and speedup of FORCv2 over the CPU for the table scan operator using the synthetic dataset. FORCv2 is integrated into the Apache ORC C++ library, with a dataflow implementation of our design as discussed in Section 3. From the figure, we observe that the Direct 32-bit dataset, being uncompressed, achieves a throughput close to the SSD bandwidth. This results in a 2.4x speedup over the CPU implementation. For the remaining encodings, except Short Repeat (SR), the throughput is lower due to the significant data transfer time caused by the large inflated data size after decompression and decoding. In the case of SR, the throughput is constrained by the FPGA's compute capability, as the compute throughput for SR is inherently lower and becomes the limiting factor. Overall, FORCv2 achieves a geometric mean speedup of approximately 18x over the CPU.

4.8.2 TPC-H Dataset Results: Figure 25 presents the end-to-end results for the table scan operator on the TPC-H dataset. In this case, the throughput for all datasets is primarily constrained by the

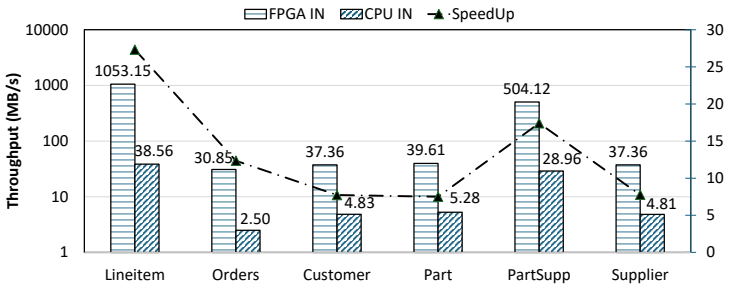

Fig. 25. End-to-End FORCv2 table scan preformance results for TPCH dataset

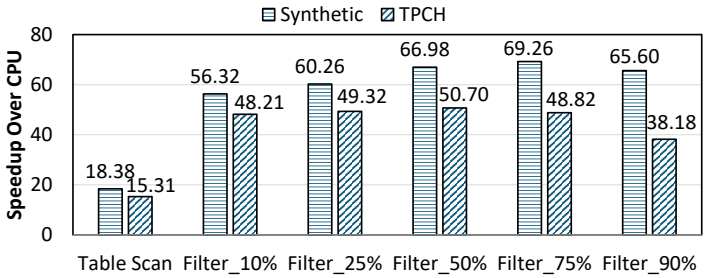

Fig. 26. Overall FORCv2 Speedup for end-to-end execution

PCIe bandwidth, following the relationship $BW_{PCIe}/(CR+1)$, as analyzed in Equation 1. Despite this limitation, the FPGA implementation achieves an impressive 15x speedup over the CPU.

4.8.3 Overall Geomean Results with Filtering: Figure 26 illustrates the overall end-to-end execution speedup for both the table scan operation and various filtering rates across the synthetic and TPC-H datasets. The results clearly show that introducing the filter operation significantly increases the FPGA speedup compared to the CPU.

An interesting observation from the figure is that the end-to-end throughput increases as the filtering rate increases. This is primarily because the data transfer time reduces, allowing the FPGA to benefit from the reduced overhead. However, at a 90% filtering rate, a slight drop in speedup is observed. This occurs because the CPU leverages the ORC row group-level filter pushdown optimization, allowing it to skip processing certain row groups. This benefit offsets the reduction in data transfer time, slightly narrowing the gap between FPGA and CPU performance.

Despite this, the FORCv2 design achieves impressive end-to-end speedups of approximately 52x for the synthetic dataset and 39x for the TPC-H dataset, considering all operators combined. These results demonstrate the efficiency of the FORCv2 pipeline in delivering substantial performance gains for real-world workloads.

4.9 GPU Comparison Results

We compare our results against the NVIDIA V100 GPU that runs Nvidia cuDF C++ version 24.02 for table scan. Figure 27 shows the achieved throughput and the FPGA speedup relative to the GPU using the TPC-H dataset. FORCv2 achieves a geometric mean speedup of 1.94x over the V100. The GPU has a static power consumption of 23.7W and maximum dynamic power consumption of approximately 124W, which uses about 4x more power than our FPGA design.

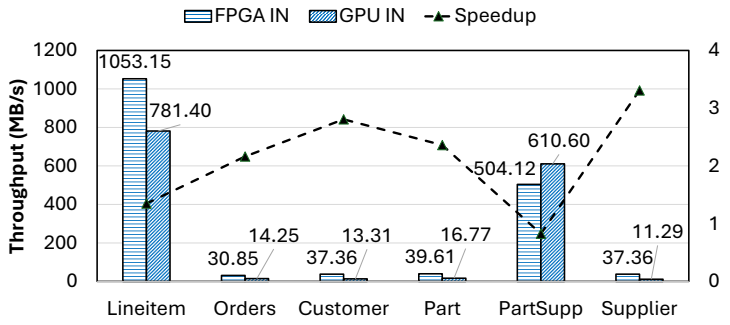

Fig. 27. End-to-end FORCv2 table scan performance results for the TPC-H dataset compared to GPU

4.10 Results Discussion

In our combined synthetic and TPC-H datasets, the throughput is bottlenecked by PCIe in 49% of the cases due to higher compression ratios. For 48% of the cases, the bottleneck is FPGA compute, limited by the lower decompression throughput (maximum 2.8 GB/s). In the remaining 3% of the cases, the bottleneck is I/O, where data is stored uncompressed by default in the case of direct 32-bit encoding. With a Gen5 SSD (14.5 GB/s) and Gen5 PCIe (128 GB/s), the bottleneck shifts to FPGA computation, more specifically our decompression kernel design. However, both the decoder and the filter can achieve approximately 13 GB/s of bandwidth, fully utilizing a single HBM channel, which is close to the bandwidth of the Gen5 SSD. For future SSD and PCIe technologies, we need to further improve the throughput of our decompression kernel design.

Instead of HBM, a disaggregated system with memory pooling can also potentially address the off-chip memory issues as it has higher off-chip memory bandwidth. As long as the CXL interface or some high-bandwidth network interface provides enough data transfer bandwidth between the compute engines and the memory pool, our streaming design can potentially benefit from this.

5 Related Work

Previous efforts have proposed FPGA accelerator designs for big data query operations such as decompression [12, 23, 24, 35, 43], filtering [39], and sorting [4, 34]. Additionally, [22] shows that HBM-based FPGAs can help overcome the bandwidth bottlenecks in data analytics workloads. Many of these studies are covered in [14], where Fang et al. also provide a comprehensive survey on the status of in-memory database acceleration on FPGAs. We design and integrate all the three accelerators in our work.

5.1 Data Decompression

Ledwon et al. [23, 24] propose FPGA-based hardware accelerators for Zlib decompression. The implementation in [23] achieves an average output throughput of 387 MB/s, which is lower than the 518 MB/s throughput reported by Xilinx for their single-PE open-source Zlib decompression, utilized in this work [43]. Their optimized version in [24] achieves an average throughput of 551 MB/s, comparable to the Xilinx open-source implementation. Furthermore, [12] introduces a multi-PE FPGA implementation of Zlib decompression, achieving an average throughput of 867.47 MB/s. However, this is lower than the performance of the multi-PE implementation of Xilinx Zlib decompression employed in this study. The ORC decompression module in our design achieves an average throughput of approximately 1.7 GB/s on the TPC-H dataset. Abali et al. [3]

propose the NXU hardware accelerator, which is integrated into POWER9 and z15 processors, to perform high-throughput data compression and decompression. It achieves up to 6 GB/s of Zlib decompression throughput. It can be utilized alongside with FPGA to improve overall ORC data processing throughput. Qiao et al. [35] proposes snappy decompression engine which achieves 510 MB/s (0.51 GB/s) of throughput which is slower than Xilinx open-source snappy decompression engine performance i.e., 1.97 GB/s [44]. In general, snappy algorithm offers better decompression performance but lower compression ratio compared to Zlib.

5.2 Data Decoding

There is little existing work 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 [33], where Peltenburg et al. present: 1) a pipelined FPGA accelerator for decoding Apache Parquet [10] file format into Arrow [6] 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 [33], first, our ORC Decoder decodes ORC file format to in-memory data, which provides better data compression ratio and query performance than Parquet [36], as well as built-in support for ACID transactions that Parquet does not support [5]. Second, we design a unified resource-efficient overlay supporting all ORC decoders explained in Section 2.5.2—including patched base and short repeat decoders that [33] does not support—and a wide range of data widths. Last but not least, we achieve a fully pipelined design and fully utilizes the effective HBM bank bandwidth (of each bank) on the FPGA, achieving up to 12.9GB/s throughput for direct and delta decoding, 61.5% higher than that of [33].

5.3 Data Filtering

Sun et al. [39] present a data filtering acceleration design implemented on the Amazon EC2 F1 instance. Their approach utilizes a single DDR4 interface for reading and writing data, achieving a bandwidth of 8 GB/s. In contrast, our ORC filter implementation achieves a bandwidth of 13 GB/s by leveraging separate memory ports for reading and writing operations. This design effectively maximizes the utilization of HBM off-chip memory, resulting in significantly higher throughput.

Liu et al. [26] propose a hardware-accelerated Bloom filter implementation on FPGA. Bloom filters are highly efficient data structures for fast data lookup filtering, which is orthogonal to our data filtering implementation. Their design achieves a throughput of 8.8 GB/s on the U280 FPGA. Notably, the ORC file format also supports Bloom filtering, and their work could be leveraged to enhance data lookup filtering within ORC-based processing pipelines.

Jeong et al. [21] present Universal Predicate Pushdown (UPP), a hardware-software co-design that translates diverse database predicates into a compact instruction set architecture (ISA) for efficient in-storage execution on FPGAs. Their approach supports variable-length columns and complex predicates without modifying data formats, achieving up to 7.9x speedup over CPU in query performance on SmartSSD. Our ORC filter implementation, with limited predicate support, achieves a speedup of 38x to 114x over the CPU for filtering rates ranging from 10% to 90%, respectively.

6 Conclusion & Future Work

In this work we have proposed a high-throughput streaming-based FPGA accelerator overlay called FORCv2 to accelerate widely used ORC file format processing in big data engines. FORCv2 employs a resource-efficient and fully pipelined overlay design to support different processes i.e., decompression, decoding and filtering involved in reading data from ORC file format. To

transparently leverage the FPGA acceleration for big data analytics, we also integrate FORCv2 with Apache ORC C++ library to execute in a dataflow fashion, which achieves an end-to-end decoding throughput up to the IO read bandwidth limits.

Considering all operators combined, on average, FORCv2 achieves end-to-end speedups of approximately 52x for the synthetic dataset and 39x for the TPC-H dataset versus the ORC C++ library running on a dual-socket system with 48 CPU threads. Our FORCv2 framework will be open-sourced at https://github.com/SFU-HiAccel/FORC. In future work, we plan to integrate it with our SQL2FPGA [29] and Bloom filter [26] work.

Acknowledgments

This work is partly supported by NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, and CFI John R. Evans Leaders Fund.

References

[1] 2021. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025 (in zettabytes) [Graph], IDC, & Statista. https://www.statista.com/statistics/871513/worldwide-data-created/.

[2] 2024. Worldwide internet user penetration from 2014 to July 2024 [Graph], DataReportal, & We Are Social, & Meltwater. https://www.statista.com/statistics/325706/global-internet-user-penetration/.

[3] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. 2020. Data Compression Accelerator on IBM POWER9 and z15 Processors : Industrial Product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA45697.2020.00012

[4] Maher Abdelrasoul, Ahmed Sayed Shaban, and Hala Abdel-Kader. 2021. FPGA Based Hardware Accelerator for Sorting Data. In *2021 9th International Japan-Africa Conference on Electronics, Communications, and Computations (JAC-ECC)*. 57–60. https://doi.org/10.1109/JAC-ECC54461.2021.9691432

[5] Apach ORC. 2023. ACID support. https://orc.apache.org/docs/acid.html.

[6] Apache Arrow. [n. d.]. Apache Arrow: A cross-language development platform for in-memory analytics. https://arrow.apache.org/.

[7] Apache Avro. 2024. Apache Avro 1.12.0 Documentation. https://avro.apache.org/docs/1.12.0/.

[8] Apache ORC. 2024. Apache ORC Documentation. https://orc.apache.org/docs/.

[9] Apache ORC. 2024. Apache ORC Github. https://github.com/apache/orc.

[10] Apache Parquet. 2024. Apache Parquet Documentation. https://parquet.apache.org/docs/.

[11] Chris Currier. 2022. *Protocol Buffers*. Springer International Publishing, Cham, 223–260. https://doi.org/10.1007/978-3-030-98467-0_9

[12] Haixin Du, Jiankui Zhang, Jin Zhang, Shihao Sha, and Zhaoyang Tang. 2019. The Library for Hadoop deflate compression based on FPGA accelerator. In *2019 Computing, Communications and IoT Applications (ComComAp)*. 282–287. https://doi.org/10.1109/ComComAp46287.2019.9018820

[13] Facebook Engineering. 2018. Zstandard. https://engineering.fb.com/2018/12/19/core-infra/zstandard/.

[14] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. 2019. In-memory database acceleration on FPGAs: a survey. *The VLDB Journal* 29, 1 (oct 2019), 33–59.

[15] Alan Gates. 2013. The Stinger Initiative: Making Apache Hive 100 Times Faster. https://web.archive.org/web/20130328050712/http://hortonworks.com/blog/100x-faster-hive.

[16] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. 2023. TAPA: A Scalable Task-parallel Dataflow Programming Framework for Modern FPGAs with Co-optimization of HLS and Physical Design. *ACM Trans. Reconfigurable Technol. Syst.* 16, 4, Article 63 (dec 2023), 31 pages.

[17] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, New York, NY, USA.

[18] Felix Handte, Yann Collet, and Nick Terrell. 2018. 5 ways Facebook improved compression at scale with Zstandard. https://engineering.fb.com/2018/12/19/core-infra/zstandard/. Accessed: 2024-12-21.

[19] Reihaneh H.Hariri, Erik Fredericks, and Kate Bowers. 2019. Uncertainty in big data analytics: survey, opportunities, and challenges. *Journal of Big Data* 6 (06 2019), 44. https://doi.org/10.1186/s40537-019-0206-3

[20] Joost Hoozemans, Johan Peltenburg, Fabian Nonnemacher, Akos Hadnagy, Zaid Al-Ars, and H. Peter Hofstee. 2021. FPGA Acceleration for Big Data Analytics: Challenges and Opportunities. *IEEE Circuits and Systems Magazine* 21, 2 (2021), 30–47. https://doi.org/10.1109/MCAS.2021.3071608

[21] Ipoom Jeong, Jinghan Huang, Chuxuan Hu, Dohyun Park, Jaeyoung Kang, Nam Sung Kim, and Yongjoo Park. 2025. UPP: Universal Predicate Pushdown to Smart Storage. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 419–433. https://doi.org/10.1145/3695053.3731005

[22] Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. 2020. High Bandwidth Memory on FPGAs: A Data Analytics Perspective. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL50879.2020.00013

[23] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. 2019. Design and Evaluation of an FPGA-based Hardware Accelerator for Deflate Data Decompression. In *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*. 1–6. https://doi.org/10.1109/CCECE.2019.8861851

[24] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. 2020. High-Throughput FPGA-Based Hardware Accelerators for Deflate Compression and Decompression Using High-Level Synthesis. *IEEE Access* 8 (2020), 62207–62217. https://doi.org/10.1109/ACCESS.2020.2984191

[25] LinuxReviews.org. n.d.. Comparison of Compression Algorithms. https://linuxreviews.org/Comparison_of_Compression_Algorithms. Accessed: 2024-12-21.

[26] Kenneth Liu, Alec Lu, and Zhenman Fang. 2024. BitBlender: Scalable Bloom Filter Acceleration on FPGAs with Dynamic Scheduling. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. 325–331. https://doi.org/10.1109/FPL64840.2024.00052

[27] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. 2021. 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* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 105–115.

[28] Alec Lu, Zhenman Fang, and Lesley Shannon. 2022. Demystifying the Soft and Hardened Memory Systems of Modern FPGAs for Software Programmers through Microbenchmarking. *ACM Trans. Reconfigurable Technol. Syst.* 15, 4, Article 43 (jun 2022), 33 pages.

[29] Alec Lu, Jahanvi Narendra Agrawal, and Zhenman Fang. 2024. SQL2FPGA: Automated Acceleration of SQL Query Processing on Modern CPU-FPGA Platforms. *ACM Trans. Reconfigurable Technol. Syst.* 17, 3, Article 39 (Sept. 2024), 28 pages. https://doi.org/10.1145/3674843

[30] Teng Lv, Ping Yan, and Weimin He. 2018. Survey on JSON Data Modelling. *Journal of Physics: Conference Series* 1069, 1 (aug 2018), 012101. https://doi.org/10.1088/1742-6596/1069/1/012101

[31] Alysha Puti Maulidina, Rachel Anastasia Wijaya, Kimberly Mazel, and Maria Seraphina Astriani. 2024. Comparative Study of Data Compression Algorithms: Zstandard, zlib & LZ4. In *Science, Engineering Management and Information Technology*, A. Mirzazadeh, Zohreh Molamohamadi, Babek Erdebilli, Efran Babaee Tirkolaee, and Gerhard-Wilhelm Weber (Eds.). Springer Nature Switzerland, Cham, 394–406.

[32] Jian Ouyang, Hong Luo, Zilong Wang, Jiazi Tian, Chenghui Liu, and Kehua Sheng. 2010. FPGA implementation of GZIP compression and decompression for IDC services. In *2010 International Conference on Field-Programmable Technology*. 265–268. https://doi.org/10.1109/FPT.2010.5681489

[33] Johan Peltenburg, Lars T.J. van Leeuwen, Joost Hoozemans, Jian Fang, Zaid Al-Ars, and H. Peter Hofstee. 2020. Battling the CPU Bottleneck in Apache Parquet to Arrow Conversion Using FPGA. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 281–286. https://doi.org/10.1109/ICFPT51103.2020.00048

[34] W. Qiao, L. Guo, Z. Fang, M. Chang, and J. Cong. 2023. TopSort: A High-Performance Two-Phase Sorting Accelerator Optimized on HBM-Based FPGAs. In *IEEE Transactions on Emerging Topics in Computing*, Vol. 11. IEEE Computer Society, Los Alamitos, CA, USA, 404–419.

[35] Yang Qiao. 2018. *An FPGA-based Snappy Decompressor-Filter*. Master's thesis. Delft University of Technology. https://repository.tudelft.nl/record/uuid:5ed3a03f-b4ac-4513-8f1c-d7583e7626bc

[36] Carter Shanklin. 2013. ORCFile in HDP 2: Better Compression, Better Performance - Cloudera Blog. https://blog.cloudera.com/orcfile-in-hdp-2-better-compression-better-performance/.

[37] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. [n. d.]. Thrift : Scalable Cross-Language Services Implementation. https://api.semanticscholar.org/CorpusID:14349304

[38] Statista. 2023. User-generated internet content per minute 2023. https://www.statista.com/statistics/195140/new-user-generated-content-uploaded-by-users-per-minute/. Accessed: 2025-01-06.

[39] Xuan Sun, Chun Jason Xue, Jinghuan Yu, Tei-Wei Kuo, and Xue Liu. 2021. Accelerating data filtering for database using FPGA. *J. Syst. Archit.* 114, C (mar 2021), 14 pages.

[40] Vadim Tkachenko. 2016. Evaluating Database Compression Methods. https://www.percona.com/blog/evaluating-database-compression-methods-update/.

[41] TPC. 2023. TPC-H is a Decision Support Benchmark. https://www.tpc.org/tpch/

[42] Abdul Wadood, Alec Lu, Ken Zhang, and Zhenman Fang. 2024. FORC: A High-Throughput Streaming FPGA Accelerator for Optimized Row Columnar File Decoders in Big Data Engines . In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/FPL64840.2024.00051

[43] Xilinx. [n. d.]. Xilinx Vitis Libraries - Data Compression. https://github.com/Xilinx/Vitis_Libraries/tree/main/data_compression.

[44] Xilinx. 2021. Vitis Libraries: Data Compression - Snappy Decoder Streaming Parallel Byte 8. https://github.com/Xilinx/Vitis_Libraries/tree/2021.2/data_compression/L2/tests/snappy_dec_streaming_parallelByte8

[45] Xilinx. 2022. Vitis Unified Software Platform. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html#development

[46] Xilinx. 2023. Alveo U280 Data Center Accelerator Card Data Sheet (DS963). https://docs.xilinx.com/r/en-US/ds963-u280/Summary

[47] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '15)*. Association for Computing Machinery, New York, NY, USA, Article 6, 11 pages. https://doi.org/10.1145/2757667.2757684

[48] David C. Zaretsky, Gaurav Mittal, and Prith Banerjee. 2009. Streaming implementation of the ZLIB decoder algorithm on an FPGA. In *2009 IEEE International Symposium on Circuits and Systems*. 2329–2332. https://doi.org/10.1109/ISCAS.2009.5118266

[49] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (Oct. 2023), 148–161. https://doi.org/10.14778/3626292.3626298