High-Throughput Big Data Analytics Through Accelerated Parquet to Arrow Conversion

L.T.J. van Leeuwen

TU Delft

High-Throughput Big Data Analytics Through Accelerated Parquet to Arrow Conversion

A Thesis

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

and

Embedded Systems

by Laurentius Theodorus Johannes van Leeuwen

To be defended on Tuesday the 27th of August

Q&CE-CE-MS-2019-17
by L.T.J. van Leeuwen (4239784)

Faculty of EEMCS
Delft University of Technology
Mekelweg 4
Delft, The Netherlands

Committee:

Chair:

Dr. ir. Z. Al-Ars

Members:

Prof. dr. H.P. Hofstee
Dr. J.S. Rellermeyer
Ir. J.W. Peltenburg

Abstract

With the advent of high-bandwidth non-volatile storage devices, the classical assumption that database analytics applications are bottlenecked by CPUs having to wait for slow I/O devices is being flipped around. Instead, CPUs are no longer able to decompress and deserialize the data stored in storage-focused file formats fast enough to keep up with the speed at which compressed data is read from storage. In order to better utilize the increasing I/O bandwidth, this work proposes a hardware accelerated approach to converting storage-focused file formats to in-memory data structures. To that end, an FPGA-based Apache Parquet reading engine is developed that utilizes existing FPGA and memory interfacing hardware to write data to memory in Apache Arrow's in-memory format. A modular and expandable hardware architecture called the ParquetReader with out-of-the-box support for DELTA_BINARY_PACKED and DELTA_LENGTH_BYTE_ARRAY encodings is proposed and implemented on an Amazon EC2 F1 instance with an XCVU9P FPGA.

The ParquetReader has great area efficiency, with a single ParquetReader only requiring between 1.18% and 2.79% of LUTs, between 1.27% and 2.92% of registers and between 2.13% and 4.47% of BRAM depending on the targeted input data type and encoding. This area efficiency allows for instantiating a large number of (possibly different) ParquetReaders for parallel workloads. Multiple Parquet files of varying types and encodings were generated in order to measure the performance of the ParquetReaders. A single engine has achieved up to 2.81× speedup for DELTA_LENGTH_BYTE_ARRAY encoded strings and 2.79× speedup for DELTA_BINARY_PACKED integers when compared to CPU-only Parquet reading implementations, attaining a throughput between 2.3 GB/s and 7.2 GB/s (limited by the interface bandwidth of the testing system) depending on the input data. The high throughput and low resource utilization of the ParquetReader allow for the interface bandwidth to be saturated using multiple ParquetReaders utilizing only a small amount of the FPGA's resources.

Preface

The nine months of research and writing that have led to this thesis have been an amazing learning experience. While the scale of the thesis project may have seemed daunting when starting out, I am very glad to finally be able to present this report as the result of my work. This could of course not have been accomplished without the help of my friends, family and colleagues.

First I want to thank my supervisor Zaid Al-Ars, my daily supervisor/colleague/friend Johan Peltenburg, and the other amazing people at the Accelerated Big Data Systems group for their support, advise and shared knowledge during my project. I have learned so much from you, and through the laughs we shared you have also made the past nine months a lot more fun than I thought they were going to be.

I also want to thank my friends Robin Hes and Cornelis de Mooij for their invaluable feedback on my thesis report. You have given me a lot more confidence in finalizing my writing.

Last, but definitely not least, giving my all everyday in order to successfully complete this project would not have been possible without the love and support of those close to me. To my parents, my friends and Anissa, thank you for being in my life. I dedicate this work to you.

—Lars van Leeuwen, 15 august 2019

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Objective & motivation

1.1.1 Speeding up database analytics

Every day, over 2.5 exabytes of data are created by humanity [1]. As the digitization of the world continues, this staggeringly large number is bound to grow even larger. This deluge of data provides many opportunities, but also many challenges. The ability to analyze large amounts of data in a database within a short amount of time is very valuable to an enterprise's short term decision making process. Knowing how often a brand name is mentioned in tweets over the course of a day is of great value to a company, but only if the large amounts of data generated by Twitter users can be processed efficiently. The performance of big data analytics applications dealing with problems of this nature is of great importance.

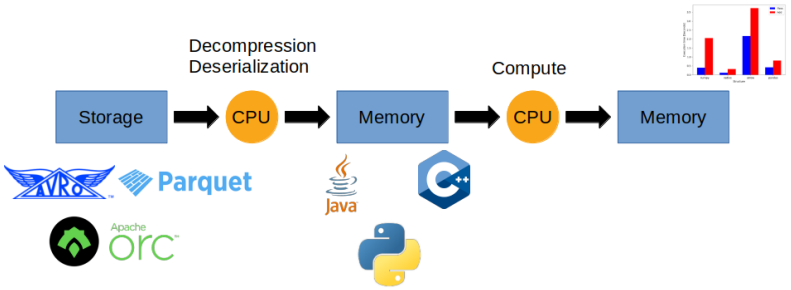


Figure 1.1: Schematic overview of a database analytics process with examples of file formats and in-memory data structures

Figure 1.1 shows a schematic overview of the basic process associated with a database analytics application. First, the data to be analyzed must be read from persistent storage and loaded into memory. Data will often be stored in a file format that provides compression while also including metadata aimed at providing structure to the data to help with query performance. In memory, the data structure is dependent on the tools that will be used for processing, with C++ vectors, Java vectors and Python Pandas dataframes requiring different memory layouts for the data than the serialized form found in the storage-focused file format. Therefore, reading data from storage requires a decompression and deserialization step before the data can be used. The second step in the database analytics process is the compute step, where the actual analysis takes place. The operations that need to be done on the data during this step are very dependent on the particular analysis that is being performed, typically including filtering, mapping and/or matching steps.

In order to speed up the process as previously described, the bottleneck needs to be found and alleviated. The nature of the analysis determines the time spent on the compute step, in turn determining whether or not the compute will be the bottleneck. So, what if we only need to do simple computations on a large amount of data to get our results? If compute is not the

bottleneck, what is? A classical assumption in computing is that I/O devices are slow while CPUs are fast, suggesting that the bandwidth of the storage medium will be the limiting factor in the process. However, this assumption is starting to become outdated [2].

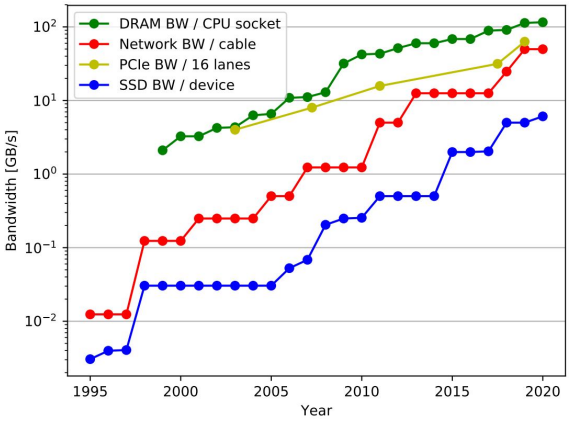
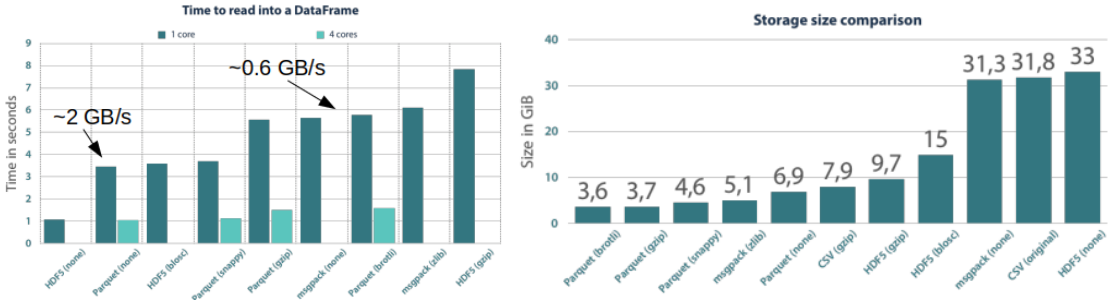


Figure 1.2: Trends in bandwidth [3]

Figure 1.2 shows how bandwidth for storage and network has developed over the years compared to DRAM or PCIe bandwidth. It can be seen that storage and network bandwidth have grown significantly quicker than PCIe or DRAM. NVMe SSDs can already be found on the market with sequential read speeds of more than 4 GB/s, which begs the question: what is being done with the ever increasing storage bandwidth?



(a) Time to read different file formats into a Pandas dataframe

(b) Comparing compression capabilities of different file formats for the same data set

Figure 1.3: Comparison of read speed and storage size for different file formats storing the same data set [4]

In [4] a benchmark has been performed comparing the read and storage capabilities of different file formats with varying compression codecs storing the New York Taxi & Limousine Commission Trip Data dataset. The benchmarks measure the size of the file once the dataset has been fully written to storage and the speed at which a file can be read into a Python Pandas dataframe, Pandas being a popular Python package for data analysis [5]. Figure 1.3 shows the results of these benchmarks. In order to make sure that storage bandwidth would not be a limiting factor in the benchmarks, the files were loaded into memory before the actual conversion to be measured took place. Figure 1.3a shows that even Parquet without extra compression, the fastest file format to read that provides any reduction in storage size, only attains a throughput of 2 GB/s on a single core. The best file format in terms of storage size, Parquet with Brotli compression according to Fig. 1.3b, only manages up to 600 MB/s single-core performance. Parquet as a file format allows for splitting the file up into multiple parts that can be read in parallel, so a multi-core solution is possible [6]. But the slow single-core performance means that, especially

for the Parquet file with Brotli compression, a lot of cores are needed to saturate the increasing storage and network bandwidth.

If CPUs are having trouble keeping up with modern trends in storage and network bandwidth, a solution outside of general processing should be investigated. With the development of specialized hardware in a field-programmable gate array (FPGA), it may be possible to significantly increase the speed at which persistent storage-focused file formats can be converted to in-memory data structures, while also improving energy efficiency. The main goal of this thesis will therefore be to develop and implement digital logic in an FPGA for use in accelerating the conversion of storage-focused file formats to in-memory data structures and to investigate its performance in comparison to that of a general processor.

1.1.2 Choice of formats

Considering the typically long development times required to produce a functional hardware design for FPGAs, it is infeasible for an MSc thesis to encompass the development a system that can freely convert a large selection of file formats to a large selection of in-memory data structures. Therefore, the project will focus only on converting the Apache Parquet file format to the Apache Arrow in-memory data structure. In this section, the reason for choosing these specific formats will be explained. A technical description of these formats will follow in Chapter 2.

Apache Arrow is a columnar memory format that aims to be language-independent, meaning that the same Arrow data should be accessible from multiple language runtimes without copying or serialization [7]. This particular quality of Arrow makes it a good fit for the target in-memory data structure of the conversion hardware because it will make the project usable by tools written in any of the languages Arrow supports, which currently include (among others) C, C++, Java and Python. Arrow being a columnar format is also an advantage. Having data of the same column stored contiguously in memory is beneficial to many database analytics applications, allowing sequential reads if only the data of a single column is required for analysis. A final advantage to using Arrow is the existence of Fletcher, a hardware framework for interfacing between FPGAs and Arrow [8]. Based on an Arrow schema describing the structure of the Arrow data, Fletcher can be used to generate hardware that will either read or write Arrow data to or from the FPGA. This pre-existence of open source Arrow writing hardware means efforts can be focused solely on the Parquet reading part of the converter.

Apache Parquet is a columnar data storage format [6]. Because Parquet and Arrow are both columnar formats no significant data reordering needs to be performed during conversion. The Apache Software Foundation has been setting up Parquet and Arrow as on-disk and in-memory counterparts, even merging development of Parquet's C++ library into the same repository as Arrow's development [9]. An advantage of this connection between Parquet and Arrow is that multiple software tools already exist for converting between the two formats, which can be helpful during development of the converter for purposes of debugging and verification. One of the basic requirements defined for Arrow is that the format should always be capable of representing fully-materialized decoded and decompressed Parquet data [10]. Finally, the structure of Parquet files allows for dividing columns into parts allowing parallel conversion of the data in a column. This is important because it allows multiple Parquet reading cores to be instantiated in the same FPGA, making optimal use of available area for parallel performance.

1.1.3 Research questions

The main research question can be formulated as follows:

- Can FPGA-based hardware acceleration allow for better utilization of increasing I/O bandwidth when converting storage-focused data formats to in-memory data formats?

Answering this question requires the design and development of a hardware accelerated Parquet to Arrow conversion system. During design, development and testing of this system, the following questions need to be answered:

1. Which forms, variants and parts of the Parquet file format are well suited for FPGA-based, high-throughput conversion to Arrow format?
2. Which factors impact the performance of the system?
3. How does the performance of the hardware accelerated system compare to that of a CPU-only system?

1.1.4 Contributions

The main contribution of this thesis is the design and development of an FPGA-based Parquet reading engine that can dissect Parquet pages and extract the relevant data. Modules are included that can decode common Parquet encodings for floats, doubles, integers and strings. The data generated by the Parquet readers can be written to Arrow format using Fletcher, or alternatively, this data can be operated on by custom analytical kernels. The four most important features of the Parquet reading engine are as follows:

1. **Area efficiency.** In order to allow for parallel Parquet reading workloads and to leave room for custom analytical FPGA kernels, the engines are designed to be very area efficient.
2. **High-throughput.** In order to provide a benefit to using FPGA-based Parquet reading engines over a CPU-only implementation, the engines are designed to maximize throughput. The combination of high-throughput and low resource utilization allow a collection of ParquetReaders to saturate the interface bandwidth while using only a small amount of the resources in the FPGA.
3. **Modularity.** Because a Parquet file can come in many forms with a multitude of different encoding or compression schemes, the design allows for easy replacement and addition of modules to facilitate future expanded support for different parts of the Parquet file format.
4. **Vendor agnosticism.** The hardware should be usable with any FPGA, in any system. Therefore the design does not use any vendor specific IP cores.

The behavior of the hardware is studied through extensive testing and benchmarking. Extensive discussion is provided on the results, allowing for future implementations of the hardware to make educated design choices for an area/performance trade-off. The ParquetReader has been designed with flexiblity in mind, allowing this trade-off to be made through the setting of VHDL generics. By measuring and discussing the properties of the system used for testing the hardware, insights are gained about the possible performance of the hardware in a practical, real-world application.

The full VHDL project can be found on the fast-p2a GitHub repository under the Apache License 2.0 [11].

1.2 Report structure

Chapter 2 provides all the necessary background for understanding the designs and design process discussed in Chapter 3 and Chapter 4. Technical discussions of Apache Parquet, Apache Arrow, Fletcher and AXI are included. These discussions are limited to only the parts of those projects that are relevant to the work presented in this thesis. Chapter 2 also explores related work that provides different perspectives on the increasing storage bandwidth and the implications for database systems. Finally, the general development strategy used for the work presented in this thesis is briefly explained.

In Chapter 3 the general design of a hardware accelerated Parquet reader is discussed from a high-level perspective, defining the steps involved in the Parquet to Arrow conversion process and dividing these steps between the FPGA and the CPU. System configurations are proposed for both a theoretical, ideal system, and for the system available for testing. A high-level hardware architecture is introduced that forms the basis for a single Parquet reading engine, of which multiple could be included in a single FPGA. A fully functional ParquetReader module for simple

non-nested, non-nullable Parquet files containing fixed-width primitives is implemented and benchmarked. The benchmark results are extensively discussed, both in context of the system configuration used for testing and theoretically possible performance.

Chapter 4 opens with a discussion on the challenges with including string reading support in the ParquetReader, and explains the decision to develop decoders for delta and delta length encoded integers and strings. A general design for a decoder is then proposed that makes use of the similarities between the integer and string encoding to enable optimal reuse of VHDL modules. The VHDL associated with this design is written in such a way that key design decisions can be changed by the setting of generics in the higher-level VHDL modules. After implementation, the implications of changing these design parameters are explored through extensive benchmarking. Special attention is also paid to the effect of the nature of the input data on the performance of the different implementations.

Finally, some concluding remarks are provided in Chapter 5, summarizing this report and reflecting back on the research question formulated in this chapter. Possible avenues for future development of the Parquet to Arrow conversion hardware are also discussed.

Chapter 2

Background

2.1 Perspectives on the decompression bottleneck

As discussed in Section 1.1.1, storage and network bandwidth have increased to the point where decompression and deserialization are at risk of becoming a bottleneck in many database analytics applications. In this section, multiple perspectives on this problem from solutions proposed in the literature are gathered and briefly evaluated in comparison to the proposed solution in this thesis.

Nanavati et al. [2] provide an extensive evaluation of the developing imbalance between I/O bandwidth and CPU performance. The authors argue that new storage class memories (SCMs) have such high bandwidth that CPUs risk becoming completely overwhelmed. Because existing paradigms for the structuring of database systems are still based on old assumptions considering IO and CPU performance, they will not be able to efficiently utilize faster non-volatile memory without rethinking not only the ratio of CPUs to SCMs in the system, but also the way CPUs interact with the SCMs. Having a fixed number of outstanding read requests to storage is a sensible way to deal with slow storage for example, but with SCMs this strategy risks flooding the RAM.

The authors take a high-level view of the subject, and the proposed solutions are accordingly general. Through careful balancing of storage with computing power, I/O centric scheduling, rethinking datacenter organization and workload-aware storage tiering, the authors foresee a future where the increasingly inefficient utilization of SCMs will be attacked at all layers of the datacenter infrastructure.

The work in this thesis is considerably more specific and low-level than that of Nanavati et al., but fits well with their aim of better utilization of the growing storage bandwidth. If CPUs are able to offload work to an FPGA accelerator, they can spend the freed up time doing other useful work. The end result will be a change in the optimal ratio of SCMs to CPUs, with fewer CPUs per SCM required to optimally utilize the SCM bandwidth.

Sukhwani et al. [12] describe an FPGA implementation aimed at accelerating database analytics for transactional databases. The authors argue that offloading analytical jobs from the CPU to an FPGA allows for fast execution of an otherwise time-consuming task and frees up CPU resources for "mission-critical transactional workloads". The hardware has been designed to take pages from the database management system's (DBMS) memory, extract the rows and evaluate predicates from SQL queries. These queries can be supplied to the FPGA at runtime, avoiding having to synthesize a new design for every single SQL query that might by used in the system.

The authors report impressive results for the performance of their prototype when compared with the baseline software executed on the CPU. When the data in the rows is compressed, the FPGA attains a speedup of 10.7x compared to the CPU for the same data. For uncompressed data the speedup is a more modest 1.2x. It makes sense that compressed data results in a larger speedup because more computations, namely those required for decompression, can be offloaded from the CPU to the FPGA. These attained results are very promising in regards to the value of a Parquet to Arrow converter with decompression capabilities.

The difference between the work performed by Sukhwani et al. and the work performed in this thesis is that their implementation is focused on transactional, row based databases. Parquet

on the other hand is a column based format more suited for analytical databases. A working Parquet to Arrow converter in an FPGA would be of great value to any application (implemented in an FPGA or otherwise) that performs analysis on data in a columnar fashion, or any toolchain that has multiple tools access the same (Arrow format) data.

Trivedi et al. [13] provide a very different perspective on the decompression bottleneck. The authors present an entirely new file format called Albis. The decompression bottleneck is removed by simply not compressing the data to begin with. Secondly, Albis provides a binary API that aims to avoid object materialization unless needed, reducing the number of objects. The third and final pillar of the Albis file format is to separate the metadata from the actual data, meaning readers can sequentially scan the data in the file without encountering unwanted metadata.

When the read performance of Albis is compared to common file formats, the differences are very significant. When reading a dataset consisting of integers and floats, Albis attains a "goodput" (incoming data size divided by runtime) of 59.9 GB/s, whereas ORC and Parquet reach 19.9 GB/s and 12.5 GB/s respectively. The authors also include Arrow in their benchmarks when used as file format instead of an in-memory format, reaching 30.1 GB/s goodput.

This performance does come at a price. The dataset used for the benchmarks when stored as an Albis file requires 94.5 GB, whereas ORC and Parquet (without extra compression) require 72.0 GB and 58.6 GB respectively. Storage space isn't free, which is especially true for the fast NVMe storage devices used in the benchmarks. Finally, the benchmarks performed in the paper are very sensitive to inefficiencies in the software tasked with reading the files, making them less of measure for the performance of the file format than the performance of the reader. Especially for Arrow, which is a relatively young project with a not yet fully optimized software library, this skews the performance benchmarks.

2.2 Apache Parquet

As mentioned on the Apache Parquet website, Parquet as a file format aims to be a compressed and efficient columnar data representation [6]. To that end, Parquet files have a complex hierarchical structure and support for multiple compression and encoding schemes. The technical explanation of Parquet in this section will be limited to structural details and encodings schemes that are relevant to the work. The Apache Parquet project is in active development and implementation details of the file format tend to change quickly. Because of this, some discrepancies have developed between the documentation and the different implementations of Parquet. All information in this section is accurate at the time of writing this document and was gathered by comparing documentation on the Apache Parquet website and parquet-format GitHub repository with the parquet-mr and parquet-cpp project source code [6][14][15][16]. For Java related testing and development of the Parquet to Arrow converter a fork of a snapshot release of parquet-mr version 1.12.0 was used [17]. For C++ related testing and development the parquet-cpp version that comes with Arrow version 0.13.0 was used.

2.2.1 File structure

Figure 2.1 shows the structure of a Parquet file with two distinct parts of the file visible. The left half of the figure shows how the columns and their values are structured, while the right half shows the FileMetaData. This metadata provides information such as the location of data in the file and type schemas.

Within the left half three different hierarchies can be distinguished: the row group, the column and the page. The row group is how Parquet ensures that values within the same row are stored near each other. Columns are divided into chunks with row groups containing one column chunk of each column in the Parquet file, stored contiguously. Because of this chunking, Parquet is not a fully columnar format. The degree to which Parquet stores values in a columnar fashion is dependent on the size of the row groups. Larger row groups (row group size can be specified when Parquet writing Parquet files) allow for larger column chunks which means that more values in the same column will be stored contiguously in memory.

More important to software or hardware tasked with reading Parquet files are the Parquet pages. Each column chunk contains one or more pages stored back to back that contain the

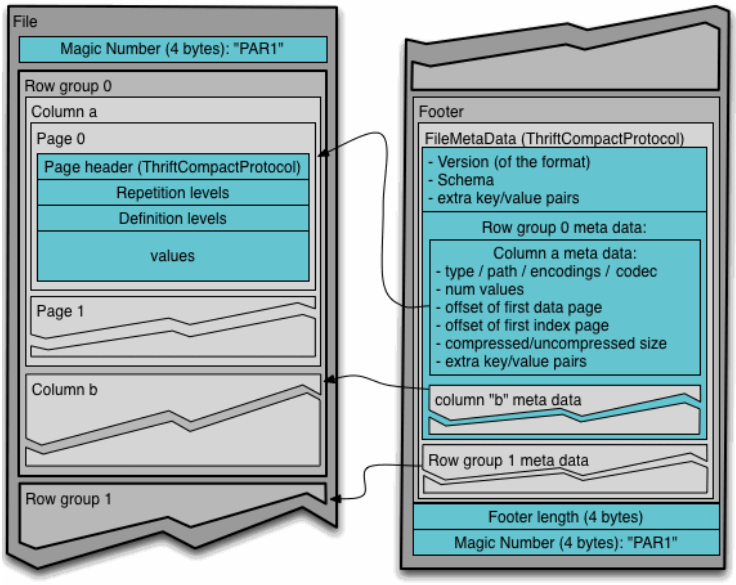


Figure 2.1: Parquet file structure [6]

compressed and encoded values with accompanying metadata. As can be seen in Fig. 2.1, there are four distinct parts in a Parquet page. The first is the page header. The page header stores information necessary for reading the page such as the number of values it contains and the encoding scheme used. Page headers are serialized using Apache Thrift which will be discussed further in Section 2.2.2. Directly following the page headers are the repetition levels and the definition levels. Repetition levels store the structure of nested types such as lists of lists while the definition levels store the nulls in the page. The encoding used for storing this information is Google's Dremel encoding [18]. If the type of the values in the page is not nested or nullable no repetition or definition levels will be encoded in the page. Finally, after the repetition and definition levels, the values are stored. These values can be encoded with any encoding that Parquet supports for the type. After this, the data can optionally be compressed on a byte level with any of the supported compression codecs. This will be further discussed in Section 2.2.3. Like the row groups, a preferred size for the pages can be selected before writing the file. Larger pages avoid the space and processing overhead of having many page headers in the Parquet file. Small pages allow for more fine grained reading, because fetching a single value in a Parquet file requires processing the entire page that contains that value.

The bulk of the metadata in a Parquet file is stored as a footer. By reading this FileMetaData structure the location and types of all the column chunks can be determined. Depending on the version of Parquet being used the metadata at the end of the file can also be used to locate individual pages in the file. This will be discussed briefly in Section 2.2.4. According to the Parquet website, the rigid separation of column data and metadata allows for splitting columns into multiple files with the metadata for all of them only having to be read from one location. This is a useful feature for the Hadoop filesystem that Parquet was designed for, but it does mean that loss of the file footer means loss of the entire file.

2.2.2 Metadata serialization

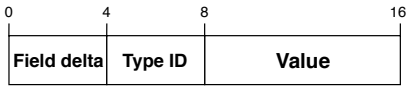
In Parquet files, metadata is serialized using Apache Thrift's "Thrift Compact Protocol". The specification for this protocol can be found on the Apache Thrift GitHub repository [19]. The features of Thrift that are important for this project will be explained in this section by using an example from Parquet's metadata structures.

Figure 2.2 shows a diagram representation of the Thrift definition of the PageHeader metadata structure. The diagram shows that a page header contains 9 fields of which 3 are required and 6 are optional. The last 6 fields are other Parquet metadata thrift structures that are (option-

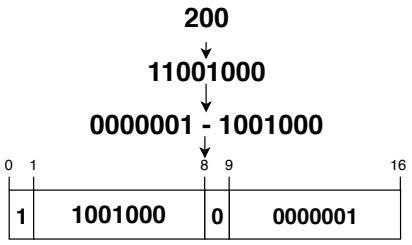
PageHeader
1: required PageType type
2: required i32 uncompressed_page_size
3: required i32 compressed_page_size
4: optional i32 crc
5: optional DataPageHeader data_page_header
6: optional IndexPageHeader index_page_header
7: optional DictionaryPageHeader dictionary_page_header
8: optional DataPageHeaderV2 data_page_header_v2
9: optional BloomFilterPageHeader bloom_filter_page_header

Figure 2.2: Thrift definition of PageHeader metadata structure

ally) nested within this one. PageType is defined as an enumeration with 32 bit integers so the first four field are all Thrift "i32" type.



(a) Thrift field encoding (short form) with a single byte value (bit numbers annotated)



(b) Example of VarInt encoding for the number 200

Figure 2.3: Two concepts in the Thrift Compact Protocol: field encoding and VarInt encoding

Two encoding concepts in the Thrift Compact Protocol shown in Fig. 2.3 are needed in order to create a Thrift binary structure for the PageHeader definition in Fig. 2.2. Figure 2.3a shows how each field is encoded with a field header consisting of a field delta and a type id, followed by the field value. The field delta (4 bits) shows which field in the definition is encoded here as a delta from the previous field. If the field delta is 1 for example, this field is the one directly following the previously encoded field in the definition. If the delta is 2 one optional field after the previously encoded field has been skipped. The first field in the structure gets field delta 1. Type id (4 bits) encodes the type of this field. The two types in PageHeader, "i32" and "struct", have id's 5 and 12 respectively. After the type id, the value of the field is encoded as defined for that particular type, with the number of bytes in the value being determined either by the type or being encoded in the value itself. Once all bytes for the values are written, either a new field delta or a null byte indicating the end of the Thrift binary structure follows.

Integers are encoded as variable length integers (VarInt) shown in Fig. 2.3b. The binary representations of the integers are divided into 7 bit groups and put into little endian order. Before each 7 bit group a 1 bit is written to indicate another byte will follow, or a 0 bit if it is the last byte. Upon decoding, the resulting number is sign extended with 0. Because this would cause problems with negative numbers, all negative and positive numbers are mapped into the natural number space using zigzag encoding with field values 0, 1, 2, 3, 4, 5 decoding to 0, -1, 1, -2, 2 and so forth.

To put the encodings from Fig. 2.3 into practice, an example PageHeader with type 3, a page size of 668 bytes (compressed and uncompressed) and no optional fields except field 8 is shown in hexadecimal representation in Fig. 2.4. This particular header requires 9 bytes to store in the file. After the field header "5C", an encoding of a DataPageHeaderV2 structure would follow in the same fashion.

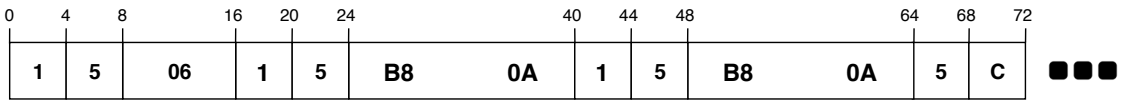


Figure 2.4: Example hexadecimal representation of a Thrift Compact Protocol encoded PageHeader with bit number annotations

2.2.3 Delta encoding

Encoding	Supported types
PLAIN	All
DICTIONARY	All
DELTA_BINARY_PACKED	INT32, INT64
DELTA_LENGTH_BYTE_ARRAY	BYTE_ARRAY
DELTA_BYTE_ARRAY	BYTE_ARRAY

Table 2.1: Encodings in Parquet with supported types

Table 2.1 shows all the supported encodings in Parquet. In the table, BYTE_ARRAY refers to any array of bytes, thus notably including strings. Of these, PLAIN encoding is the simplest. Integers are encoded with their full little-endian 32 bit or 64 bit representation, written consecutively. Strings are encoded as their length followed by their characters, again written back to back. Other encodings in Table 2.1 are more complex, often providing a significantly larger compression ratio. Implementing support for all these encodings is outside the scope of this thesis, so a selection had to be made in Parquet's encodings. For reasons further explained in Section 4.1 it was decided to include support for the DELTA_BINARY_PACKED and DELTA_LENGTH_BYTE_ARRAY encoding schemes. Therefore, this section will include a technical explanation on the delta encoding scheme as it is implemented in Parquet.

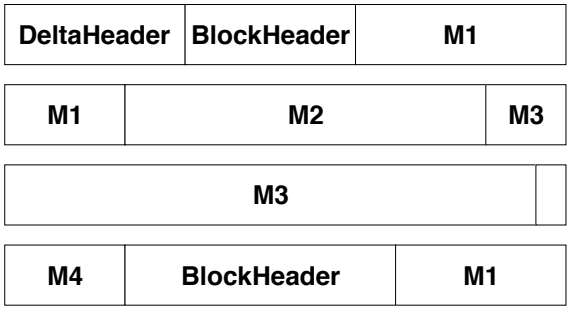


Figure 2.5: Schematic layout of DELTA_BINARY_PACKED encoded integers in a Parquet page

The general idea behind delta encoding schemes is that each piece of data is not stored in a way that directly represents the value, but rather as a difference in respect to the previously stored value. This difference is referred to as the delta. The benefit of such encodings schemes is that a set of large numbers with only small differences between them can be stored as very small deltas. If deltas are small, they can be represented with only a few bits. This allows for efficient bitpacking. Delta encoding works especially well for sorted data sets, because subsequent numbers in a data set are guaranteed to be as close to each other as possible through the sorting. Data sets whose subsequent values have large differences between them may not be efficiently storable using delta encoding schemes.

As a demonstration of the general layout of the values section in a Parquet page that has been encoded with DELTA_BINARY_PACKED encoding (hereafter referred to as delta encoding) a schematic representation of the start of such a section can be found in Fig. 2.5. A delta encoded page always starts with a delta header, the only one in the page. After that a number of blocks

follow consisting of a block header followed by a number of miniblocks. Figure 2.5 shows a delta encoding with four miniblocks per block referred to as M1, M2, M3 and M4. The delta header consists of four integers back to back:

1. **Block size in values:** The number of values in the block, always a multiple of 128.
2. **Number of miniblocks:** The number of miniblocks in a block. Upon dividing "block size in values" by "number of miniblocks" the result (the number of values in a miniblock) should always be a multiple of 32.
3. **Total value count:** The total number of values in the page. The same as in the page header.
4. **First value:** The first integer in the page.

All four are encoded as VarInts (see Section 2.2.2), but only "first value" is encoded as a zigzag integer because it is the only one that can be negative. In the parquet-mr file writer (the Java implementation of Parquet), "block size in values" is hardcoded to be 128 and "number of miniblocks" is hardcoded to be 4, resulting in 32 values per miniblock [15].

Within a block, integers are encoded as deltas with respect to the previous value. The first delta in the page uses the "first value" encoded in the delta header as a starting point. All deltas in the block are bit packed with each miniblock having its own set bit width as defined in the block header. Because bit packing requires all deltas to be positive (the packed bits are sign extended with 0), each block header includes a "minimum delta" that first needs to be added to the unpacked deltas to create the final (possibly negative) delta. This results in a block header with the following fields:

1. **Minimum delta:** The minimum delta in the block added to unpacked deltas to allow for negative deltas. Encoded as a zigzag VarInt.
2. **Bit widths of miniblocks:** The bit widths of each miniblock, encoded as one byte per miniblock bit width.

This block and miniblock structure exists in order to dynamically react to changes in the encoded data set. Every new miniblock has new bit packing width so a contiguous group of integers that are numerically close can have their small deltas packed very tightly. Every new block has a new minimum delta, meaning that a contiguous group integers with large, diverging deltas can be "escaped" upon the start of a new block.

If there are not enough values to fill the final miniblock it is padded to its expected length (based on the bit width). The format specifies the miniblock should be padded with 0, but Parquet readers should be able to deal with other padding values as well. If there are so few values in the final block that an entire miniblock can be omitted that miniblock will not be encoded. The bit width for that miniblock encoded in the block header should be set to 0 according to the format, but (like with the miniblock padding), Parquet readers should be able to deal with other values as well.

Based on the delta encoding, the DELTA_LENGTH_BYTE_ARRAY encoding (hereafter referred to as delta length encoding) can be used to encode strings. In delta length encoding the delta encoding as described above is used to encode the string lengths. After all the lengths have been written, a complete sequence of all the characters in the strings will be written back to back.

2.2.4 Parquet v1.0 and Parquet v2.0 compared

Two major versions of the Parquet file format exist: version 1.0 and version 2.0. The differences between these two versions are significant enough to require a choice in which version to support in the Parquet to Arrow converter. The most important differences that may be relevant for making this choice are listed in this section. The actual decision on which version to support is explained in Section 3.1.2.

1. **Writer support:** Of the two major Parquet implementations, parquet-cpp (C++) and parquet-mr (Java), only parquet-mr currently supports writing version 2.0 files.

2. **Page headers:** The page header for version 2.0 includes more data. Notably the number of nulls in the page and the size of the definition and repetition level parts of the page in bytes. (In version 1.0 these sizes are found in the definition and repetition level parts of the page themselves).

3. **Compression:** In Parquet version 1.0, selecting a compression codec during Parquet file writing will cause the byte level compression to be applied to the definition levels, the repetition levels and the values. In Parquet 2.0, this compression is only applied to the values.

4. **PageIndex and OffsetIndex:** Starting from Parquet 2.4, extra (Thrift Compact Protocol encoded) metadata structures are included in the footer of the file that store information on the location of Parquet pages in the file based on column values and row indices. This allows for more efficient scans and deprecates some of the now superfluous information previously stored in page headers and column metadata. Column metadata is now no longer written after the columns by the parquet-mr filewriter.

2.3 Apache Arrow

On the Apache Arrow web page the project is described as “a cross-language development platform for in-memory data” [7]. Arrow includes not just a format for in-memory data, but also computational libraries and built-in functionality for interprocess communication with support for many programming languages. Because Arrow aims to have its data structures accessible to multiple language runtimes without serialization, the format has a well defined physical memory layout. This layout will serve as the basis for the technical explanation of Arrow in this section [10]. All testing and development in this work is based on Arrow version 0.13.0.

2.3.1 Arrow structures

The Arrow data structure with which this thesis is most concerned is the Arrow Array, which will be the focus of the technical explanations in this section. Arrow defines an array as a sequence of values with known length that all have the same type, making it the most direct translation of a Parquet column chunk in the Arrow project and a natural target for a Parquet to Arrow converter. Arrays are immutable, meaning a process is not allowed to change values in the array. This is an important guarantee to have when multiple processes can access the same data. According to the specification, "all array slots are accessible in constant time, with complexity growing linearly in the nesting level" [10]. This fast access is very beneficial to the performance of analytics applications.

Arrays can be collected into higher level data structures, most notably record batches and tables. Record batches are collections of multiple equal length arrays whose (possibly differing) types are defined by an Arrow schema. Tables consist of columns that in turn consist of array chunks of the same type. Essentially, record batches and tables are both collections of arrays defined by a schema, but only tables allow for chunking.

2.3.2 Primitive type arrays

Figure 2.6 shows what an Arrow array containing 32 bit integers would look like in memory. Each of the values is stored contiguously in a buffer of values padded to a multiple of 64 bytes. In the value buffer, both the padding bytes and the bytes belonging to a null can have any value. The null bitmap buffer shows which values in the values buffer are null. If a bit in the null bitmap buffer is 0, the corresponding value in the value buffer is null. In the case of a 1 it is a valid value. Note that bytes in Arrow’s null bitmap buffer have their bits ordered from right to left. If the type of an Arrow array is set as non-nullable, the inclusion of a null bitmap buffer is optional.

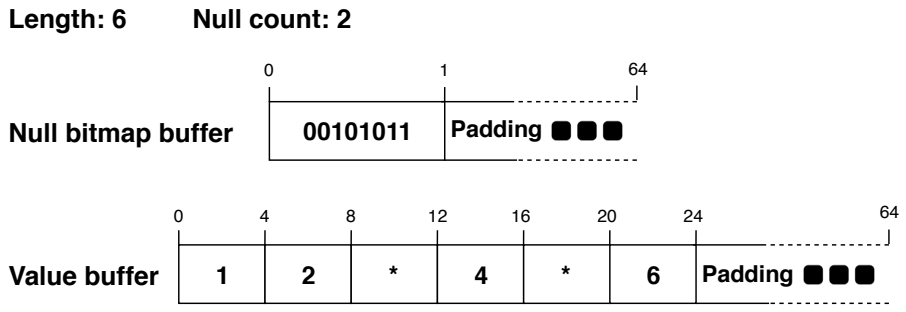


Figure 2.6: Schematic representation of the memory layout of an Arrow array containing the 32 bit integers [1, 2, null, 4, null, 6], with byte numbers annotated

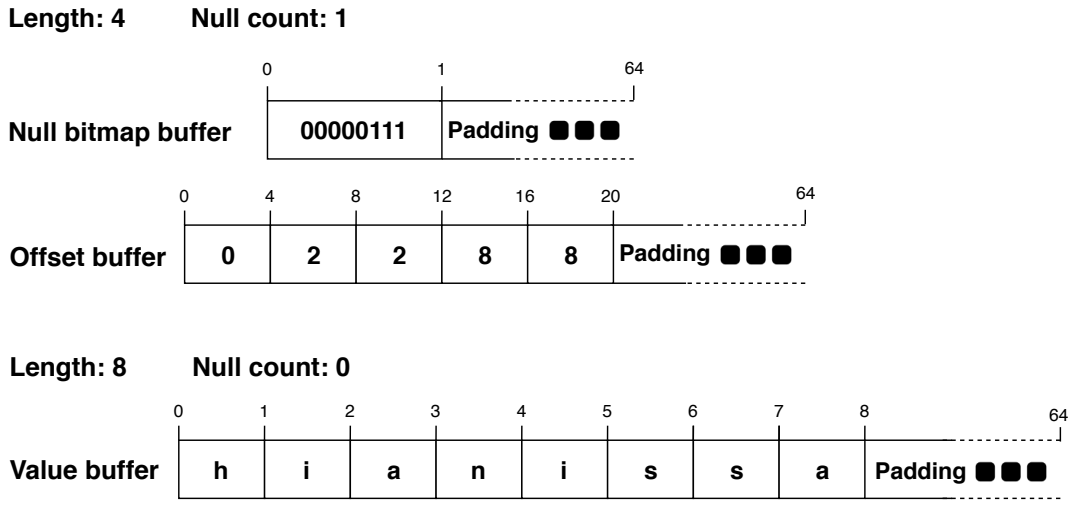


Figure 2.7: Schematic representation of the memory layout of an Arrow "List<char>" array (strings) containing [["hi"], [""], ["anissa"], null], with byte numbers annotated

2.3.3 List type arrays

The simplest nested structure has the schema "List<T>", which is an array of lists of the type specified by "T". This can be any type, so when nested types are chosen, deeply nested structures can be made with Arrow. Figure 2.7 shows the memory layout of an Arrow array with the schema "List<char>", making it essentially an array of strings. In order to make this nested structure, an offset buffer of 32 bit integers is added. A distinction can now be made between a parent array (offsets) and a child array (values). The offset buffer encodes the start of each string in the value buffer. For example: the third string in the array is "anissa" and the third offset in the offset buffer is 2. This means the string "anissa" starts at index 2 in the value buffer. The length of any string can be calculated directly from the offset buffer by reading the offsets of the string whose length needs to be calculated and the offset of the string following it. Subtracting the first offset from the second offset results in the length of the string that starts at the first offset. This property is ensured by the offset buffer always having a length one larger than the top-level array, with the full length of the value buffer written to the last slot in the offset buffer. Note that the values (child) array does not include a null bitmap buffer in Fig. 2.7 due to the fact that no characters are null. If any character were null, a null bitmap buffer would have to be included.

2.4 Fletcher

With Apache Arrow aiming to be accessible to analytical tools in any computing environment, having a way to interface between Arrow data and FPGAs is a natural extension of that idea. To that end, Fletcher has been created by the Accelerated Big Data Systems group at the TU Delft Computer Engineering department [8][20]. Fletcher is an easy to use and efficient hardware interface for reading or writing values to or from an Arrow array. By allowing FPGAs and software tools access to the same data set in a format suitable for both hardware and software, serialization overheads normally associated with FPGA accelerators are alleviated [8].

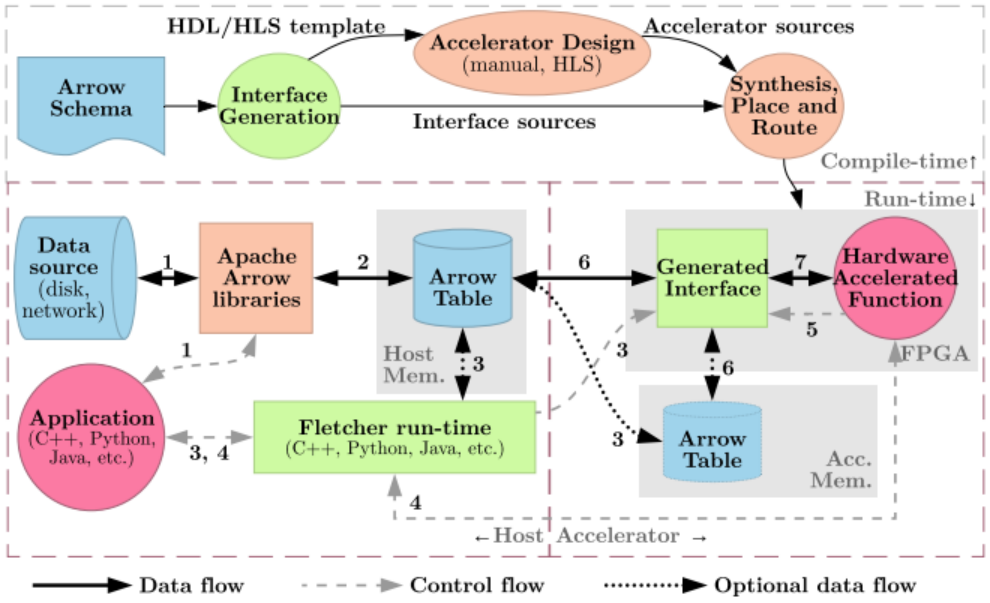


Figure 2.8: Fletcher overview [8]

Figure 2.8 shows the architectural overview of Fletcher. Based on an Arrow schema, Fletcher will generate a VHDL description for a hardware interface used to read or write Arrow arrays. One of the main focuses of Fletcher is that it should be easy to use. Therefore, values in the Arrow arrays are addressable with row indices. This avoids any need for pointer arithmetic. Through the use of a template provided by Fletcher, the end user is able to implement Fletcher compatible kernels for accelerated computing. On the host side, the communication with the accelerator is done via a platform agnostic runtime currently supporting Amazon Web Services and CAPI SNAP.

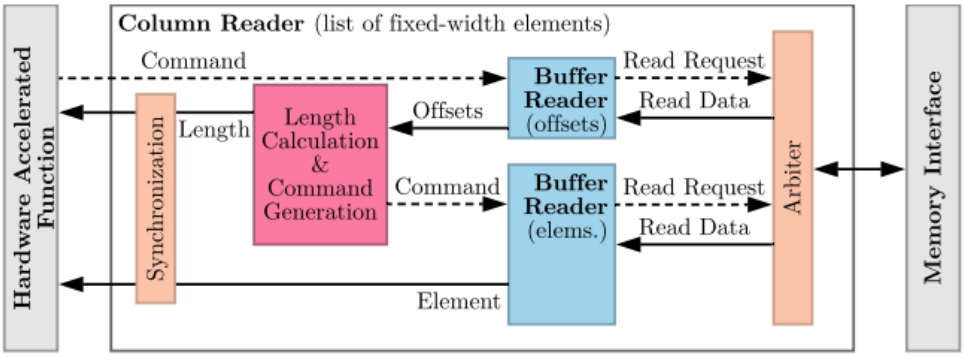


Figure 2.9: Fletcher array reader architecture for an array containing non-nullable lists of fixed-width elements [8]

The workhorses of the Fletcher hardware interface are the ArrayReaders and ArrayWriters. Figure 2.9 shows an example configuration of a Fletcher ArrayReader, configured for reading an array containing non-nullable lists of fixed-width elements. The BufferReaders in the diagram are the components that are responsible for reading the Arrow buffers. The values read by the BufferReader for offsets are used to calculate string lengths, after which the lengths and characters are streamed to the Hardware Accelerated Function through a synchronizer. For BufferWriters the process is reversed.

Through the use of Fletcher's ArrayWriters, the development of a Parquet to Arrow converter is essentially reduced to the development of a Parquet reader, as the "'to Arrow" part of the project is already covered. Having Fletcher as a dependency also allows for the usage of the many VHDL packages that Fletcher (and the associated vhlib library) provides in the areas of stream manipulation, bus architecture and simulation utilities [21].

2.5 AXI

The Advanced eXtensible Interface (AXI) is an interconnect protocol defined in ARM's Advanced Microcontroller Bus Architecture (AMBA) specification for connecting functional blocks or peripherals in FPGA, System on a Chip (SoC) or embedded designs. The FPGA designs discussed in this thesis rely on AXI for interfacing with external memory, therefore the basics of the AXI protocol will briefly be discussed in this section.

2.5.1 Handshaking protocol

AXI relies on a ready-valid handshaking protocol for transmitting data between AXI interconnected modules. Figure 2.10 shows a diagram of the connection between two modules communicating via such handshakes. The data on the data port is valid once the producer asserts the valid signal. If the consumer asserts the ready signal it is ready to receive data. If at the rising edge of the clock signal both the valid and ready signals are asserted, both the producer and the consumer know a transaction has taken place on this clock edge and they both deassert their ready and valid signals until they are ready to produce or consume again (which may be in the same clock cycle). In AXI style handshakes, a consumer may wait for a producer to assert valid before he asserts ready. A producer may however never wait for a consumer to assert ready before he asserts valid.

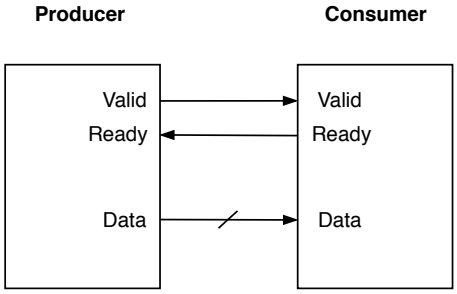


Figure 2.10: AXI style communication

2.5.2 AXI4 master/slave interface

Figure 2.11 show the channels between master and slave defined by the AXI4 protocol [22]. In each of these channels, data is transmitted using the handshaking protocol described in Section 2.5.1. From a high-level view the channels serve the following purposes:

- The write address channel is used by the master to signal to the slave it wants to write data. It provides the address it wants to write to and the amount of data it is going to write.

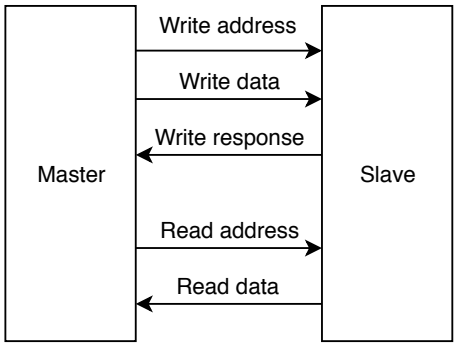


Figure 2.11: Channels between master and slave

- The write data channel is used by the master to transfer the data to the slave after signaling an incoming write transfer on the write address channel.
- The write response channel is used by the slave to signal to the master it has completed a write transfer. This channel is unused by the designs in this thesis.
- The read address channel is used by the master to request a read transfer from the slave of a specified size, starting at a specified address.
- The read data channel is used by the slave to transfer the data requested on the read address channel to the master.

The AXI4 protocol defines many different signals in each of these channels. For the work performed in this thesis the following signals are relevant:

- Write address channel
  - **AWADDR:** The address to write to
  - **AWLEN:** The number of transfers in the write burst
  - **AWSIZE:** The number of bytes in each transfer
- Write data channel
  - **WDATA:** The data written in the current transfer
  - **WLAST:** Indicates the last transfer in a burst
  - **WSTRB:** Indicates which bytes in **WDATA** are valid
- Read address channel
  - **ARADDR:** The address to read from
  - **ARLEN:** The number of transfers in the read burst
  - **ARSIZE:** The number of bytes in each transfer
- Read data channel
  - **RDATA:** The data read in the current transfer
  - **RLAST:** Indicates the last transfer in a burst
  - **RRESP:** Indicates the status of the read transfer

Included with Fletcher is the Interconnect package. This packages provides useful components for creating a bus architecture that allows multiple modules communicating via an AXI4 like protocol to use a single AXI4 interface to the external memory via arbitration. This package can be used to create a design that instantiates multiple parallel computation engines that

work independently from each other. The only differences between the AXI4 like protocol (hereafter called Fletcher bus) used by the Interconnect package and the AXI4 subset described above are the absence of the AWSIZE and ARSIZE signals and a slightly different interpretation of the ARLEN and AWLEN signals.

2.6 Hardware development methodology

Developing custom logic for an FPGA is done by creating register-transfer level (RTL) descriptions of the circuitry. An RTL description models the circuit as a sequence (or network) of hardware registers with data flowing between them. Operations can be done on the data by inserting combinatorial digital logic in between these registers. The two most commonly used hardware description languages (HDLs) are VHDL and Verilog. VHDL was chosen as the HDL to be used for the Parquet to Arrow converter. This choice was made mainly because of better familiarity of the developer with this language. Another reason for choosing VHDL is because it is the same language that Fletcher was written in. Although mixed-language Verilog and VHDL projects are well supported in modern synthesis tools, using the same HDL as Fletcher will result in a more cohesive total design.

The hardware will be designed by first sketching a high level view of the circuit, specifying multiple independent modules that have a well defined job and well defined inputs and outputs. Design parameters in these modules will be implemented as VHDL generics as much as possible to facilitate a flexible and customizable design. Based on Fletcher's development style, the communication between modules will be done using AXI style ready-valid handshakes as discussed in Section 2.5.1. The benefit of using AXI style handshakes is that data flows dynamically through the pipeline. If a module can't consume data from its producer, it will simply deassert ready, which will let the producer know it can't proceed. Depending on the design of the producer, it can decide to either consume data from upstream anyway or block its inputs as well, propagating the backpressure up the pipeline. In this way, each module (depending on its buffering capabilities) can decide for itself how much data it can consume before downstream modules need to consume data.

As an overarching VHDL development methodology, a strategy resembling the two-process method described by Gaisler will be used [23]. Each module will have a VHDL architecture with two processes, one for sequential signals and one for combinatorial signals. In the combinatorial process, all the inputs and registers will be evaluated and new values for the registers and outputs will be determined. The sequential process takes these values and propagates them to the registers and outputs upon the rising edge of the clock signal. This design methodology is a good fit for the handshaking strategy discussed previously because of the combinatorial nature of the valid and ready signals. If all inputs and outputs of a module were synchronous signals, one single sequential VHDL process in the architecture would suffice. Valid and ready can change in the middle of a clock period however, which can be taken care of by a separate combinatorial process.

Debugging hardware designs can be a difficult and time consuming task, so special attention needs to be paid to setting up a comprehensive set of unit tests for functional verification. Depending on the complexity of the module concerned, one of three testing strategies will be chosen. A self-checking testbench will be made for very complex modules. Using Python, input data is randomly generated together with the expected output data. A VHDL testbench will insert this data into the module and compare its output to the expected output. This VHDL testbench will randomly insert pauses into the input and output streams to check if the module can correctly deal with backpressure or sparse input streams. If the module to be tested is relatively simple with a limited set of possible behaviours, a testbench will be made that does not directly check the output, but whose output needs to be verified by visual inspection. Finally, if the input of a module is very specific and hard to generate with a Python script, the module can be tested by integrating it into a higher level module whose other internal modules have already been verified to work correctly. This higher level module can then be tested to verify operation of the contained module.

Chapter 3

Hardware accelerated Parquet reading

3.1 High-level design

3.1.1 System overview

The first step in designing a system for hardware accelerated Parquet to Arrow conversion is to determine the steps involved in reading a set number of values from a Parquet column starting at a certain index. Then, a decision has to be made on which tasks will be offloaded to the FPGA and which tasks are better suited for execution on the CPU. As discussed in Section 2.2.1, a column is divided into column chunks. Each column chunk consists of a number of pages stored back to back that contain the stored values. This means the longest sequential read a Parquet reader can do is the full length of a column chunk. After a column chunk has been processed, the starting location of the next column chunk belonging to the same column in the file needs to be found before more values can be read. The starting locations of pages and column chunks in the file are stored in metadata structures in the footer of the Parquet file. To summarize, when reading a set number of values from a Parquet column starting at a certain index, the following steps need to be performed for each column chunk encountered in the column until the specified number of values have been read:

1. Fetch the location of the column chunk in the file and the number of values contained within from the Parquet metadata in the footer
2. Repeat for each page in the column chunk:
    a) Read page header
    b) Decode repetition levels (in case of nested data types)
    c) Decode definition levels (in case of nullable types)
    d) Extract values from page with optional decompression and decoding depending on the Parquet file
3. Write values to Arrow format in memory

When considering how to divide these steps between the CPU and the FPGA, the strengths of both computing platforms need to be considered. FPGAs are famously good at parallel computations. Well designed, deeply pipelined hardware in an FPGA can do a lot of very specialized computations on either the same or different data in the same clock cycle. The trade-off is that the clock frequency is low when compared to that of a CPU, in the order of hundreds of MHz. A CPU core, being a general processor, does not specialize for certain computations and instead relies on a limited set of instructions executed sequentially to perform the required operations on the data. Unlike an FPGA however, a CPU can do these sequential computations very quickly with a clock frequency in the order of multiple GHz.

Due to the nature of the Thrift compact protocol used to serialize the metadata structures in Parquet (see Section 2.2.2), fetching values from the Parquet metadata is an inherently sequential process. Fields in Thrift structures are often of variable length, with 32 bit integers for example

taking up between 1 and 5 bytes once encoded with VarInt encoding. Because optional fields in Thrift structures can be omitted and fields are identified by their location in the Thrift definition relative to the previously encoded field, each field header in the structure needs to be inspected to read the value from one specific field stored towards the end of the structure. This requires inspecting bytes one by one, something the FPGA with its low clock frequency is not especially good at. Also, in order to find the requested information about the column chunk, multiple Thrift structures need to be traversed. This pointer chasing process would require an FPGA to either wait for the memory latency every time a new structure needs to be read, or to cache the entire file footer in block RAM on-chip, the size of which grows significantly for larger and more complex Parquet files.

The page reading process is much more suitable for execution on FPGA. Decompressing and decoding the values stored in each page is a computationally intensive task, giving a lot of room for improvement over the CPU. Page headers are serialized using Thrift like the other metadata structures. According to the argumentation used in this section, deserializing these on the FPGA is not optimal. These page headers are quite small however, and when a larger page size is selected when writing a Parquet file these headers can be very sparse in a Parquet file. The performance impact of having to read fewer or more page headers on the FPGA will be investigated in Section 3.3.2.

Taking the strengths of both the FPGA and the CPU into account it was decided to perform the fetching of column chunk metadata on the CPU, which can then provide a starting address in storage or memory of a column chunk and a request for a certain number of values to the FPGA. The FPGA will then traverse all the pages in the column chunk until it has extracted the requested number of values. The CPU can then start a new job for the next column chunk if it needs more values from the column. The values read from the column chunks are written to Arrow format by the FPGA using Fletcher.

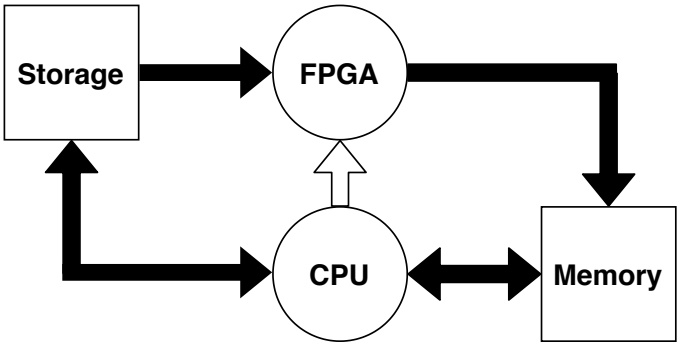


Figure 3.1: System overview of the hardware accelerated Parquet to Arrow converter

Figure 3.1 shows what a complete system might look like. The CPU reads the Parquet metadata from storage and provides control information to the FPGA telling it where to find the start of the column chunk in the file, the size of the column chunk in bytes, and how many values to read from it. The FPGA can then directly read the Parquet data from storage and write the data (in Arrow format) to the host memory using DMA. However, because the hardware necessary to build such a system was unavailable during this project, a different system configuration based on Amazon EC2 F1 instances had to be used for testing [24].

Figure 3.2 shows the system used for testing and performance evaluation. The CPU reads a Parquet column chunk from storage and loads it into host memory. This column chunk is then transferred to the accelerator on-board memory (AOM) using DMA. The custom logic in the FPGA can be configured and started by the CPU using an AXI-Lite MMIO interface. Once the custom logic has finished processing the column chunk the resulting Arrow array is copied back to host memory and compared to the expected result. As an end-to-end system this configuration is less optimal than the one shown in Fig. 3.1, because it requires copying the Parquet and Arrow data back and forth between the host memory and the AOM. However, because the time spent copying data and the time required to convert the Parquet data to Arrow format using the FPGA can be measured individually, it is perfectly functional as a way of testing the performance of

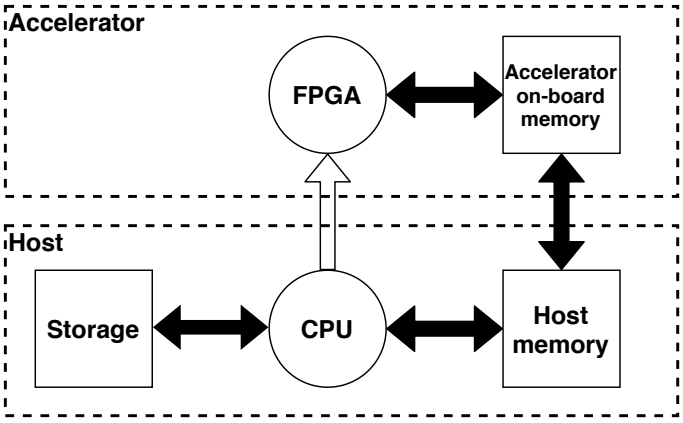


Figure 3.2: System overview of the Amazon EC2 F1 instances used for testing

the custom logic. It should be noted that the system in Fig. 3.2 could still be used as an efficient end-to-end system when the copies to and from the AOM and the accelerator processing are pipelined. Further discussion on the implications of using Amazon EC2 F1 instances for testing and performance evaluation can be found in Section 3.3.

3.1.2 Hardware architecture

Section 3.1.1 defined the work to be performed on the FPGA as the conversion of a contiguous sequence of Parquet pages to Arrow format based on the size of the column chunk and the number of values to be read, both parameters supplied by the CPU. The high-level architecture of the ParquetReader VHDL module targeting Parquet v2.0 files is shown in Fig. 3.3.

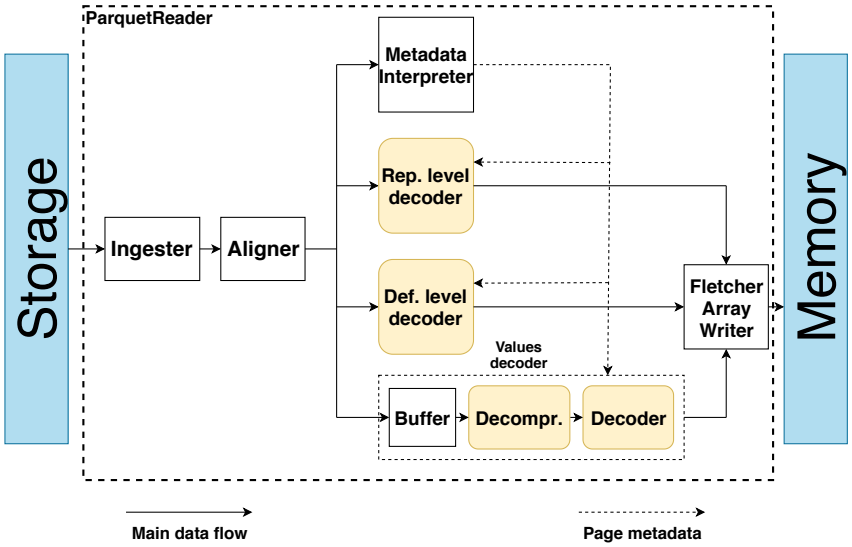


Figure 3.3: High-level architecture of the ParquetReader hardware design with yellow blocks signifying optional or replaceable modules

With pages streaming into the ParquetReader back to back, the design needs to cycle between reading the page header, the repetition levels, the definition levels and the values of the page, in that order. Each of these four blocks gets its own module for parsing the contained data. The aligner has the job of ensuring that each of these modules gets the associated part of the Parquet pages, with the data correctly aligned. Because the lengths of the four blocks in the Parquet pages are not the same for each page, the aligner requires its consumers to report back the number of bytes in the last consumed bus word that they actually used. The aligner uses this information

to realign the data for the next module. The ingester generates read requests on the Fletcher bus and buffers the data response.

A design where the four main modules were connected in series has also been considered. Each module would have to read only the data of its own block, after which it would have to realign the incoming bytes and pass the remaining data in the page to the next module. This idea was abandoned because this decentralized aligning strategy would result in a more complicated design than the design seen in Fig. 3.3. The task of dividing the Parquet page into parts is now given solely to the aligner, avoiding the need to implement very similar logic in multiple modules.

The yellow blocks in Fig. 3.3 show the modularity of the design. The definition levels are not encoded in pages with non-nullable types and the repetition levels are not encoded in pages containing non-nested types. This means that these modules can be left out of the design when dealing with these simple types. The prototype developed for this thesis only supports reading non-nullable fixed-width primitives and strings, so no repetition and definition level decoders were implemented. The decompressor and decoder in the ValuesDecoder module are included in the design during synthesis based on the chosen compression codec or encoding for the column the ParquetReader is tasked with reading.

As discussed in Section 2.2.4, there are significant differences between Parquet v1.0 and Parquet v2.0. Parquet v2.0 was chosen as a target not only because it is the newest version of Parquet, but also because it allows for the regular structure seen in Fig. 3.3. In Parquet v1.0, the definition levels and repetition levels are compressed together with the values. A design targeting Parquet v1.0 would therefore require the decompressor to be directly connected to the aligner, after which another aligner would be required to divide the resulting uncompressed data among the level and value decoders.

The ParquetReader can be configured at synthesis time for different types, compressions and encodings by setting the generics in the VHDL ParquetReader component. In order to improve usability, types can be set using the same configuration strings that Fletcher uses, supported by Fletcher's ArrayConfig package. Parquet encodings and compressions can be selected via separate generics.

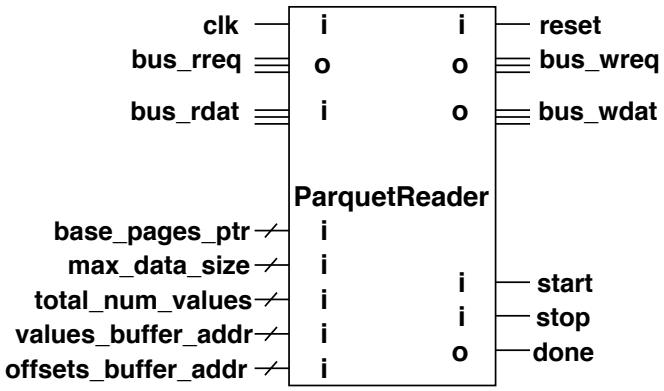


Figure 3.4: Top-level view of the ParquetReader module

The inputs and outputs of the ParquetReader module as described in Fig. 3.3 can be seen in the top-level view of Fig. 3.4. Through `bus_rreq` and `bus_rdat` the ParquetReader can connect with the Fletcher bus architecture for accessing the AXI4 read address and read data channels respectively. The internal Fletcher ArrayWriter does the same for the AXI4 write channels with `bus_wreq` and `bus_wdat`. The parameters for a command to the ParquetReader can be set with the signals in the bottom left of the figure. `Base_pages_ptr`, `values_buffer_addr` and `offsets_buffer_addr` set the addresses for the Parquet column chunk, Arrow values buffer and Arrow offsets buffer respectively. `Max_data_size` limits the number of bytes the ParquetReader will read from memory and `total_num_values` sets the number of values to read from the column chunk in the current command. Once the parameters have been set the ParquetReader can

be started and stopped via the signals in the bottom right of the figure. The ParquetReader asserts its `done` output upon completion of the command.

Figure 3.5 shows how Fletcher's bus architecture can be used to instantiate ParquetReaders for a parallel workload. The ParquetReaders can be configured at synthesis time to each process a different column from a Parquet file with different types, encodings and/or compression codecs. The ParquetReaders could also be configured to work on the same Parquet column in order to divide the workload of a single Parquet to Arrow conversion job over multiple Parquet reading engines. In order to facilitate a flexible Parquet to Arrow conversion system, multiple configurations and combinations of ParquetReaders could be synthesized beforehand for different datasets or files. An FPGA can be reconfigured with such a configuration in just a few seconds. Through the controller module, each command can be sent to each ParquetReader independently of the others. The prototype configurations developed for testing and performance evaluation are single engine designs requiring no special controller set-up. Instead they rely on Fletcher's UserCoreController as a stand-in.

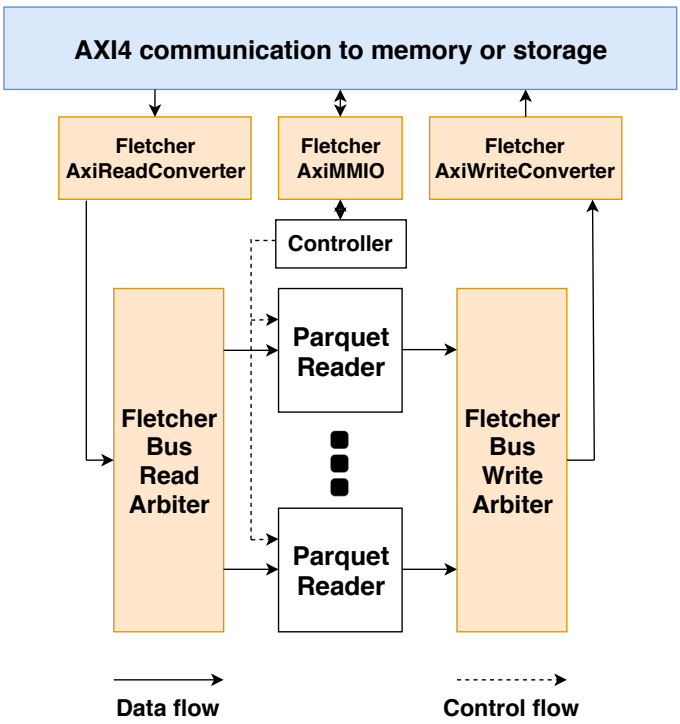


Figure 3.5: Using Fletcher's bus architecture (shown in orange) for a parallel configuration of ParquetReaders

3.2 Module design

Implementing a prototype of the Parquet reading architecture described in Section 3.1.2 that can read plain encoded Parquet files containing non-nullable fixed-width primitives requires the implementation of several modules: the Ingester, DataAligner and MetaDataInterpreter modules and a ValuesDecoder module that contains a simple decoder for the plain encoding that counts the number of primitives it passes to the Fletcher ArrayWriter. The design of these modules is discussed in this section.

For many of the modules (and sub-modules) discussed in this section, a schematic representation of the VHDL entity is included as a top-level view of the module. These figures follow the legend in Fig. 3.6.

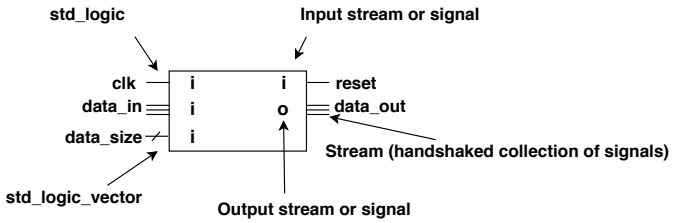


Figure 3.6: Legend for the top-level views included in this thesis

3.2.1 Ingester

The goal of the Ingester is to send AXI4 compliant read requests over the Fletcher bus in order to read the Parquet pages in the column chunk. The Ingester and DataAligner are designed to impose no alignment requirements, which means that the first byte of the column chunk can be at any address of the AOM in the Amazon EC2 F1 instances. This decision was made so future users of ParquetReader implementations would not have to think about memory alignment, improving the usability of the design.

Figure 3.7 shows the top-level view of the Ingester. The `start` and `stop` signals are directly connected to the controller for starting and stopping the reading from memory. Through `base_address` and `data_size`, the Ingester knows at what address to start reading and how much data it may read at most. Two streams are produced by Ingester: the data out stream and the producer alignment stream, which are both connected to the DataAligner. The data out stream simply streams the read data to the DataAligner, while the producer alignment stream supplies the DataAligner with an initial misalignment of the data caused by a read from an unaligned starting address. Figure 3.8 shows how AXI4 responds to a read request with an unaligned `base_address`, supplying invalid data between the closest aligned address and requested address. This initial misalignment will be corrected by the DataAligner, as shown in Fig. 3.8.

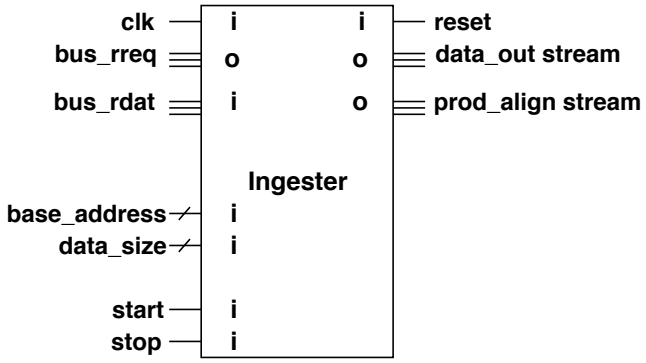


Figure 3.7: Top-level view of the Ingester module

The internal design of the Ingester is based on Fletcher's BufferReaders. AXI4 requires that a read burst does not cross 4 kB address boundaries [22]. To that end, the Ingester will request bursts with 1 transfer per burst until it reaches a 4 kB aligned address where it will start requesting full length (4 kB) bursts. The AXI4 interconnect to memory on the system used for testing has a bus width of 512 bits, making the Ingester able to read 64 bytes from memory every clock cycle.

A Fletcher BusReadBuffer is used to buffer the read requests and the data response. The BusReadBuffer will only allow a read request to go to the Fletcher bus if it can buffer the full response, thereby avoiding blocking the entire bus with internal backpressure in one ParquetReader module. Simulations showed that a small BusReadBuffer that allows for few outstanding read requests causes the ParquetReader to be unable to efficiently read from memory due to memory latency after each read request. Therefore, to ensure the Ingester would not limit the read bandwidth of the system, the developed prototype has the BusReadBuffer set to be large enough to

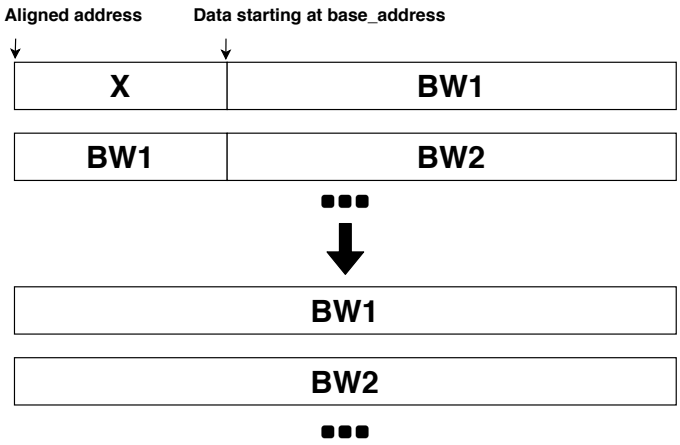


Figure 3.8: Realignment after unaligned memory access to two bus words containing only valid data

buffer 8 data responses to read requests. Simulations showed that this was more than enough to ensure full usage of the read bandwidth.

3.2.2 DataAligner

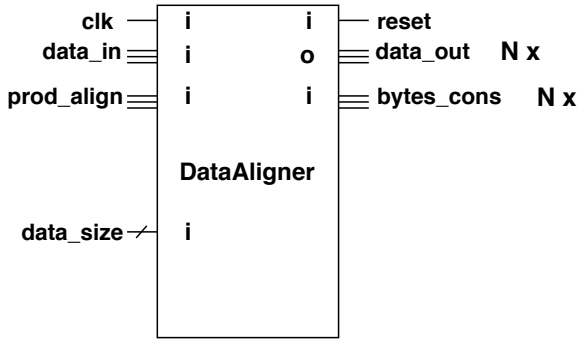


Figure 3.9: Top-level view of the DataAligner module, N is the number of consumers

Apart from the job of correcting misalignments resulting from unaligned reads, the DataAligner also needs to realign for each of the four blocks in the Parquet page. One bus word send to one of its consumers can contain data from both the page header and the page values for example, in which case a similar situation as in Fig. 3.8 occurs. Through the "bytes consumed" stream, each of the consuming modules report to the DataAligner how many bytes in the last received bus word they actually needed (also signaling their completion).

In order to correctly align the misaligned bus words, a shifter is needed to shift the bytes to their correct positions. Logical barrel shifters need a lot of multiplexers for shifting larger bit width words, taking up a lot of area in the FPGA [25]. Because the 512 bit bus words this shifter needs to shift are too large to shift combinatorially, a pipelined barrel shifter from vhlib's Stream package was used. The downside of this pipelining is that every bus word is shifted during multiple cycles, and during shifting it might turn out that a bus word in the pipeline needs to be shifted according to the alignment of the next block in the Parquet page. These misaligned bus words somehow need to have their alignment corrected. This problem was solved by including a HistoryBuffer in the DataAligner. This HistoryBuffer is essentially a FIFO that stores every bus word that enters the shifting pipeline until the DataAligner logic is sure that that bus word has been correctly aligned. If a bus word has been incorrectly aligned and requires realignment, the entries in the HistoryBuffer will form the input to the shifting pipeline until the situation has

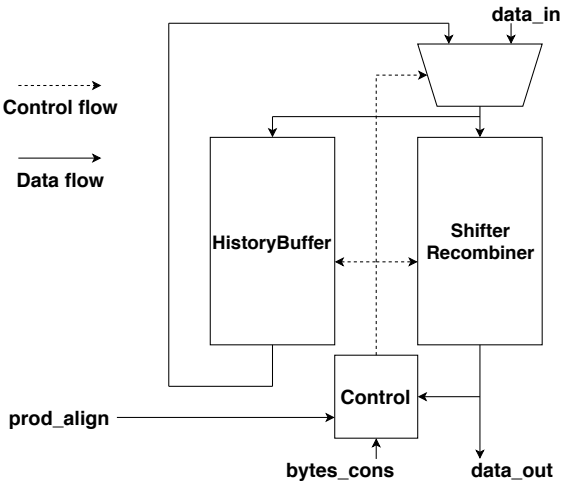


Figure 3.10: Schematic representation of DataAligner internals

been corrected. A schematic representation of this process can be seen in Fig. 3.10. A downside of this solution is that the pipeline is flushed every time a block in the Parquet page is completed, causing one cycle delay for each stage in the pipeline (6 by default in this design) and 3 cycles for the output buffer in the ShifterRecombiner.

Multiple alternative realignment solutions have been considered. The inclusion of the HistoryBuffer could be avoided (saving area) if the DataAligner would ask the Ingester to read back the column chunk from an earlier address when realignment is required. This would however have a severe negative impact on performance as both the Ingester and DataAligner would have to be flushed and the design would have to wait for memory latency every time realignment is required.

Another alternative solution would be to have the DataAligner predict which bus words would form the boundary between blocks based on the size of each block provided by the MetadataInterpreter. Such a solution would only work for Parquet v2.0 files as Parquet v1.0 files don't store the size of the repetition and definition levels in the page headers. For Parquet v1.0 files the repetition level decoder and definition level decoder have to determine the lengths of their respective blocks themselves. This solution was not chosen in order to allow the DataAligner to work in such scenarios where the size of the blocks is not known before starting processing on the blocks. This decision is briefly evaluated in the context of the rest of the ParquetReader design in Section 5.2.

If the DataAligner detects it has consumed the maximum allowed data from the Ingester as dictated by its `data_size` input, it will start flushing itself to ensure that no data gets stuck in the pipeline.

3.2.3 MetadataInterpreter

The MetadataInterpreter is tasked with extracting the required metadata from the Parquet page headers. As discussed in Section 2.2.2, the page headers are serialized using the Thrift compact protocol, a protocol that requires inspecting every byte to determine what the meaning of the next byte will be (see Section 3.1.1). This byte-wise parsing makes the MetadataInterpreter suitable for implementation as a large finite-state machine (FSM). In order to read the PageHeader structure as described by the Thrift definition in Fig. 2.2, four different states are needed. One state to that determines the state of the module (whether it is currently processing), one to determine which field of the PageHeader the module is processing, one state to determine which field of the internal DataPageHeaderV2 structure the module is processing, and a final state to distinguish between field headers and field data.

The fields in the DataPageHeaderV2 structure whose values are required by other modules in the ParquetReader are: the size in bytes of the repetition and definition levels, the uncompressed

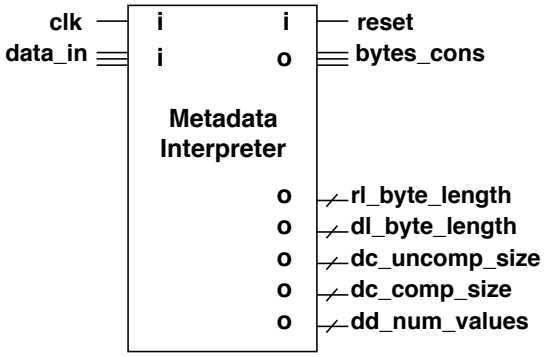


Figure 3.11: Top-level view of the MetadataInterpreter module

and compressed size in bytes of the values in the page, and finally the number of values in the page. All these fields are of the Thrift "i32" type. As discussed in Section 2.2.2, integers are encoded in Thrift using VarInt encoding, which is decoded with the VarIntDecoder whose inputs and outputs can be seen in Fig. 3.12.

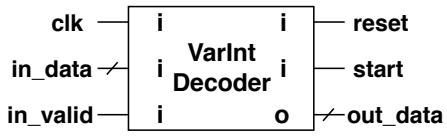


Figure 3.12: Top-level view of the VarIntDecoder module

Upon starting the VarIntDecoder it will take the least significant seven bits of every received byte and store them in the output register in the correct position. When a byte is encountered whose most significant bit is 0, the VarIntDecoder is done. The VarIntDecoder module was developed with an optional combinatorial circuit on the output that converts zigzag encoded integers to two's complement representation, a necessity for the integers in the Thrift compact protocol.

Because the MetadataInterpreter processes the page header byte by byte, the MetadataInterpreter needs a number of clock cycles equal to the number of bytes in te page header to finish processing the header. The length of the page header varies with the amount of data in a page, with larger pages having longer page headers. Generally the page headers for non-nullable, non-nested types are between 25 and 30 bytes. Thus, for the test cases in this thesis, the MetadataInterpreter is expected to take between 25 and 30 clock cycles to process each page header.

3.2.4 ValuesDecoder

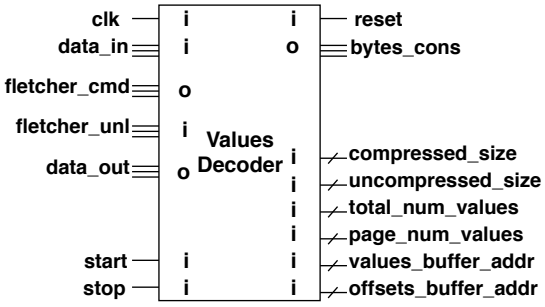
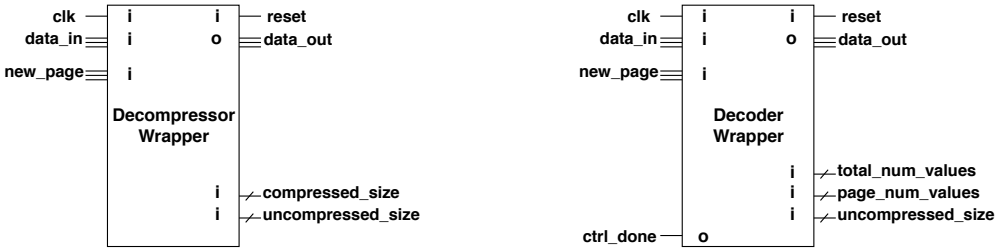


Figure 3.13: Top-level view of the ValuesDecoder module

The ValuesDecoders serves mostly as a wrapper for more interesting components, defining a port set-up for the expected inputs and outputs of the decompressors and decoders that can be slotted into the ValuesDecoder as seen in Fig. 3.14. The only actual logic the ValuesDecoder

itself is responsible for is the communication with the Fletcher ArrayWriter. Upon receiving a start signal from the host, the ValuesDecoder will stream a command to the Fletcher ArrayWriter containing the addresses in memory of the values buffer and the offets buffer. It will then await a response on the unlock stream that indicates the Fletcher ArrayWriter has written the full Arrow array to memory, after which the ValuesDecoder can send a done signal to the host.

Apart from a decompressor and a decoder, the ValuesDecoder also includes a buffer called the PreDecBuffer. This buffer is responsible for communication with the DataAligner on the "bytes consumed" stream, and handshakes both the decompressor and the decoder every time the DataAligner offers up a new page. A benefit of including a FIFO buffer upstream of the decompressor and the decoder is that it allows the MetadataInterpreter to read the page header of the next page while the decompressor and the decoder are still working on the previous one.



(a) Expected inputs and outputs for decompressors

(b) Expected inputs and outputs for decoders

Figure 3.14: Port set-up of decompressors and decoders in the ValuesDecoder

Together with the ValuesDecoder, a simple PlainDecoder was also developed to allow for reading Parquet columns containing floats, doubles, 32-bit integers or 64-bit integers encoded with the plain encoding. As discussed in Section 2.2.3, in the case of fixed-width primitives a plain encoded Parquet column is simply a contiguously stored list of the values in their standard binary representations. Therefore, the only thing the PlainDecoder has to do is to pass the values in the page to the Fletcher ArrayWriter and count them. If the PlainDecoder determines there are not enough values left in the page to fill an entire bus word it will notify the Fletcher ArrayWriter that the final bus word contains a limited number of values. The PlainDecoder also keeps track of the total number of values requested by the host so it can assert the `last` signal with the final transaction to the ArrayWriter.

3.3 Implementation & evaluation

In this section the implemented design will be evaluated in terms of area and performance. As discussed in Section 3.1.1, all testing was done on Amazon's EC2 F1 instances. The F1 instances come with a Xilinx XCVU9P FPGA and an Intel Xeon E5-2686 v4 processor of which 8 hardware threads are available which will be used to compare FPGA and CPU performance [26]. The design implemented in an XCVU9P meets all timing requirements at a clock frequency of 250 MHz, which is the maximum available clock speed for the interface between the custom logic and Amazon's mandatory included shell logic.

3.3.1 Area

In order to determine the area efficiency of the ParquetReader design, a ParquetReader module was configured for reading uncompressed, plain encoded 64-bit integers and implemented in a Xilinx XCVU9P FPGA using Xilinx Vivado 2018.3. Table 3.1 shows the area utilization hierarchy of this module reported by Vivado. With only 1.18% LUT usage, 1.27% register usage and 2.13% BRAM usage the ParquetReader uses just a small fraction of the available resources, leaving a lot of room for instantiating more ParquetReaders for parallel workloads. Fletcher's bus architecture and AXI interconnect take up another 0.29% of LUTs, 0.44% of registers and 0.00% of BRAM, while Amazon's mandatory shell requires 13.20% of LUTs, 9.48% of registers and 11.41%

Component	LUTs	Registers	BRAM tiles
ParquetReader	1.18% (13956)	1.27% (30074)	2.13% (46.0)
Ingester	0.11% (1318)	0.12% (2796)	0.35% (7.5)
DataAligner	0.31% (3710)	0.27% (6301)	0.69% (15.0)
HistoryBuffer	0.05% (537)	0.01% (16)	0.35% (7.5)
ShifterRecombiner	0.25% (2980)	0.26% (6202)	0.35% (7.5)
MetadataInterpreter	0.06% (662)	0.03% (640)	0.00% (0.0)
ValuesDecoder	0.10% (1173)	0.14% (3269)	0.35% (7.5)
PreDecBuffer	0.09% (1077)	0.11% (2625)	0.35% (7.5)
PlainDecoder	0.01% (93)	0.03% (640)	0.00% (0.0)
Fletcher ArrayWriter	0.60% (7049)	0.72% (17068)	0.74% (16.0)

Table 3.1: Area utilization as reported by Xilinx Vivado of a ParquetReader configured for reading uncompressed, plain encoded 64 bit integers in a XCVU9P FPGA

of BRAM. This means that on this particular system at most 41 ParquetReaders can be instantiated, limited by the amount of BRAM required in the design and the Amazon shell. Note that this amount is an upper bound. In practice the maximum number of ParquetReaders will most likely be smaller due to the Fletcher bus architecture increasing in size with more ParquetReaders, and place and route issues that will pop up when using a lot of area in the FPGA. On systems other than the F1 instances that do not require the proprietary Amazon shell, the freed CLBs make room for another 5 ParquetReaders at most, making for an upper bound of 46 ParquetReaders.

3.3.2 Performance

A Parquet reading utility that reads Parquet columns with plain encoded fixed-width primitives has to do very little to get the data in the column into Arrow format. For every page in the column chunk the page header needs to be read to find the size of the page, after which its contents can be directly copied to an Arrow buffer. Therefore, there is not really a "decompression bottleneck" to speak of when converting such files. Measuring the throughput of the ParquetReader in this configuration is however still important as it shows the ability of the design discussed in Section 3.1.2 to feed the decompressor and decoder that are included within the ParquetReader when converting more highly compressed Parquet files to Arrow format. The PlainDecoder used in the configuration benchmarked in this section can process a full 512 bit bus word per cycle, meaning it will never be the bottleneck in this design. As such, the measurements in these section will provide an upper bound to the throughput of the ParquetReader, which will come from either the design or the read/write bandwidth of the FPGA to memory. Latency of the design will not be used as a measure of performance because the design is expected to process large batches of data, the Parquet column chunks. The delay between the last data entering the hardware and the last results being written to memory is not as important as the ability of the hardware to process large amounts of data per unit of time.

A simple C++ runtime was written using Fletcher's API that allows the host to communicate with the FPGA. First the Parquet data is read from disk and loaded into host memory. Experimentation has shown that the write to memory has to be 4 kB aligned in order to make full use of the bandwidth from host memory to the AOM. Memory is allocated in both the AOM and the host memory in order to store the resulting Arrow array. Pointers to the accelerator-side memory regions that will contain the Parquet and Arrow data, the number of values to read and the column chunk size are supplied to the FPGA via MMIO. Once all this is done the runtime tells the FPGA it can start processing. It then continuously polls the FPGA for its completion. Afterwards, the resulting Arrow array is copied back to host memory and verified.

Because the C++ implementation of Parquet is the only one that can do automatic conversion between Arrow arrays and Parquet files, that implementation was used to verify the Arrow arrays. As mentioned in Section 2.2.4, the parquet-cpp library can not read or write Parquet v2.0 files. Therefore, a setup for generating and verifying test cases was created using both the Java (parquet-mr) and C++ implementations of Parquet. First, test data is generated using parquet-

cpp and written to a Parquet v1.0 file. This file is then converted to a Parquet v2.0 file using any page size, compression or encoding needed by parquet-mr. This file is used as input data for the ParquetReader while the Parquet v1.0 file is used by parquet-cpp to verify the result. For the benchmarks discussed in this section, seven Parquet v2.0 files were generated containing 125×10^6 64-bit integers with varying page sizes ranging from 800 to 10^9 bytes. All the data in these Parquet files is stored in one singular column chunk, that once fully read and converted results in an Arrow array of 1 GB.

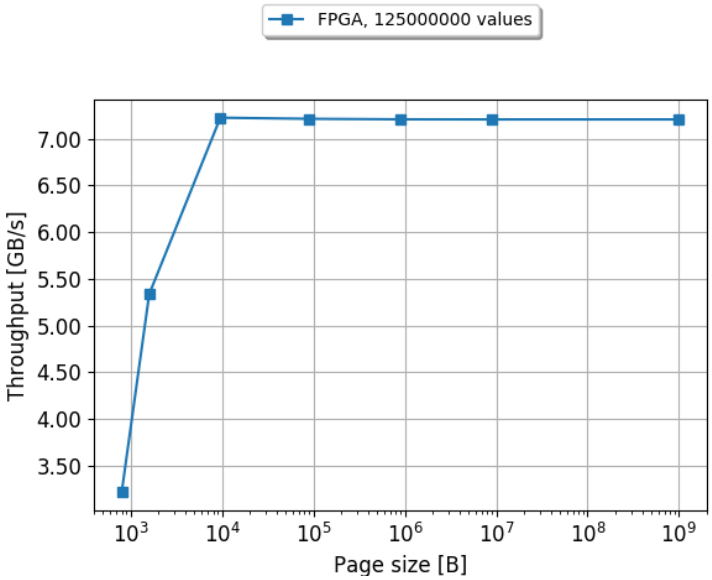


Figure 3.15: Throughput in GB/s for different Parquet page sizes in Parquet files containing $125 \cdot 10^6$ 64-bit integers

Figure 3.15 shows how the throughput of the ParquetReader hardware is influenced by the average size of a Parquet page. Throughput is calculated by dividing the size of the input Parquet column chunk by the processing time of the FPGA. The measurement for FPGA processing time starts after the Parquet file has been copied to the accelerator card once the start signal has been given to the FPGA, and stops once the FPGA has written the Arrow array to the AOM and it has signaled its completion. As expected, having small Parquet pages negatively impacts the throughput because the ParquetReader spends a lot of time reading the many page headers (byte by byte). In the test case with the largest pages the Parquet file consists of only one page which means the ParquetReader only has to read one page header, after which it can simply copy all the data to memory in one go.

The measurements show that the combined read and write bandwidth of one DDR controller is 7.2 GB/s. FPGA kernels measured in [8] have shown read bandwidths of up to 14.28 GB/s on an Amazon F1 instance, but those kernels did not have to write to the AOM at the same time. At a page size of 10 kB the ParquetReader is already fully utilizing the available bandwidth to memory. Considering that the Apache Parquet documentation recommends a data page size of 8 kB, Parquet files do not have to deviate far from the default recommendations in order to make optimal use of the hardware in this system.

Figure 3.16 shows that the throughput of the ParquetReader benefits from reading more values at once from the Parquet file. If only a small number of values is read the execution time suffers because of the time it takes to start the hardware, the memory latency, and the 100 µs polling intervals for checking the completion of the ParquetReader. This shows Parquet files with larger column chunks (allowing for larger sequential reads) can make the most optimal use of the hardware. For the two examples in Fig. 3.16, which are the measurements done with the largest and smallest pages of the test cases, maximum throughput is attained at different points. The test case with the largest page sizes reaches its maximum output once the resulting Arrow array reaches a size of around 100 MB (roughly equal to the consumed data from the Parquet

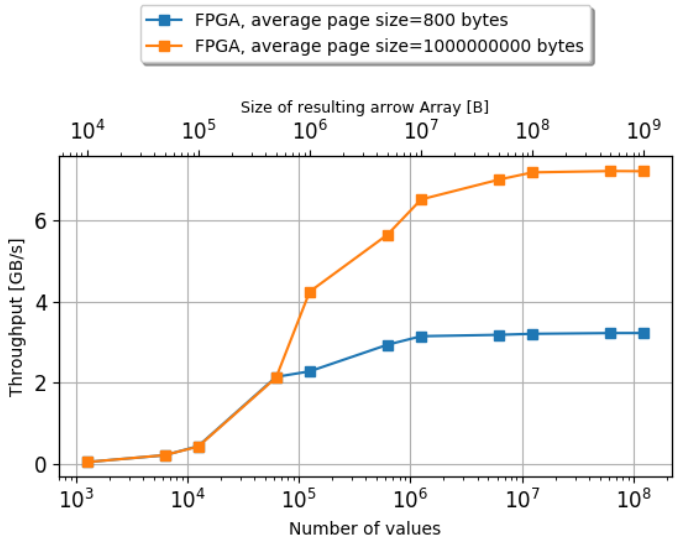


Figure 3.16: Throughput in GB/s with varying numbers of values read from the Parquet file

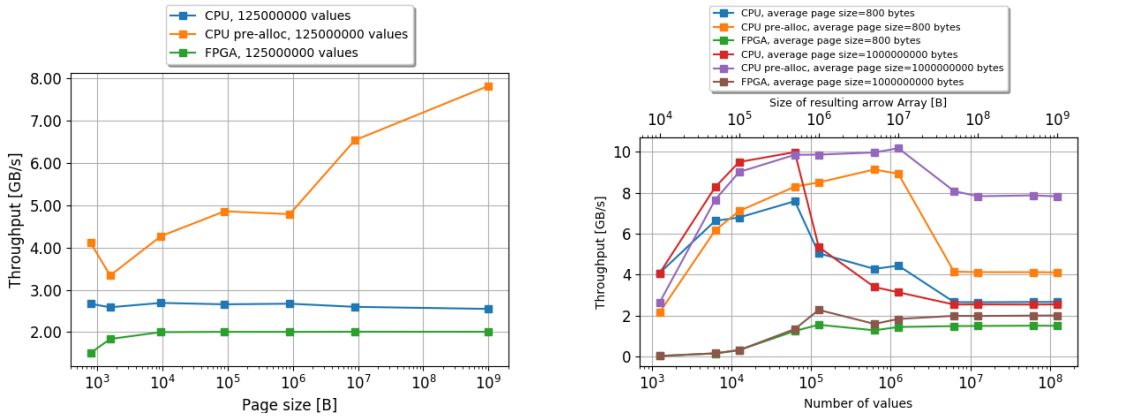
file). The test case with the smallest pages reaches this point at around 10 MB. The Parquet documentation recommends that users write Parquet files with large row groups (512 MB to 1 GB is mentioned as a possible range), which does not necessarily say anything about column chunk size as the number of column chunks in a row group is dependent on the number of columns in the Parquet file [27]. But for Parquet files with a limited number of columns (5 to 10), the results fit well with the default recommendations.

3.3.3 System considerations

As discussed in Section 3.1.1, the system used for testing is not ideal. The performance in Section 3.3.2 comes from measuring the time it takes the FPGA to read the Parquet data from the on-board memory on the accelerator and write it back as an Arrow array. If this system were to be used in a real world scenario the Parquet data would first have to be read from storage into host memory, then copied from host memory to the AOM, after which the result would have to be copied from the AOM back to the host memory. All this data movement significantly increases the total time required to convert a Parquet file to Arrow format. In this section the performance of the Parquet to Arrow converter will be investigated with the properties of the system used for testing taken into account, thus demonstrating the importance of system-level implementation decisions in real world applications using the Parquet to Arrow conversion hardware. All measurements in this section are done on a naive implementation of the Parquet to Arrow converter that does not make use of pipelining for concurrent copying and processing.

In order to put this performance into context a software-only Parquet to Arrow converter was developed in C++, whose performance was also measured when run on the Amazon F1 system. The development of a custom software-only Parquet to Arrow converter was needed because the standard Parquet implementation in Java does not include Arrow functionality yet and the standard Parquet implementation in C++ does not support Parquet v2.0 files [15][16]. In order to make a fair comparison, the software based Parquet to Arrow converter is allowed to make the same assumptions at compile time as the hardware based one. It will for example only read the fields from the page headers that give information about the size of the page and the number of values it contains. In this way it assumes the input Parquet data has the type and encoding as defined during compile time. The C++ `SWParquetReader` class reads page headers byte by byte like the hardware, and copies the data in the pages to the Arrow buffers using `memcpy()`. The software is compiled using GCC's `-Ofast` and `-march=native` optimization flags.

The execution time for the C++ implementation is measured in two ways, one where the resulting Arrow buffers are already allocated and touched using the Arrow library's allocation functions and `memset()`, and one where the time to allocate memory is included in the measurement. There will be a significant difference in execution times between these measurements because the second measurement will have a large address translation overhead due to virtual addresses of the Arrow buffers not being in the translation lookaside buffer (TLB) upon starting the measurement. This is not a problem the FPGA implementation has to deal with, as it does not use virtual addressing for the AOM. By doing two measurements a distinction can be made between the pure computing time needed by the CPU and the time realistically needed to do the Parquet to Arrow conversion work, the same as is being done for the FPGA in this section. Like with the measurements in the previous section, the execution time is measured from the start of the Parquet to Arrow conversion until completion, with the Parquet data already being loaded into memory beforehand.



(a) Throughput for varying page sizes

(b) Throughput for varying numbers of values read

Figure 3.17: Throughput in GB/s for Parquet to Arrow conversion with system initialization and the copy from AOM to host memory taken into account

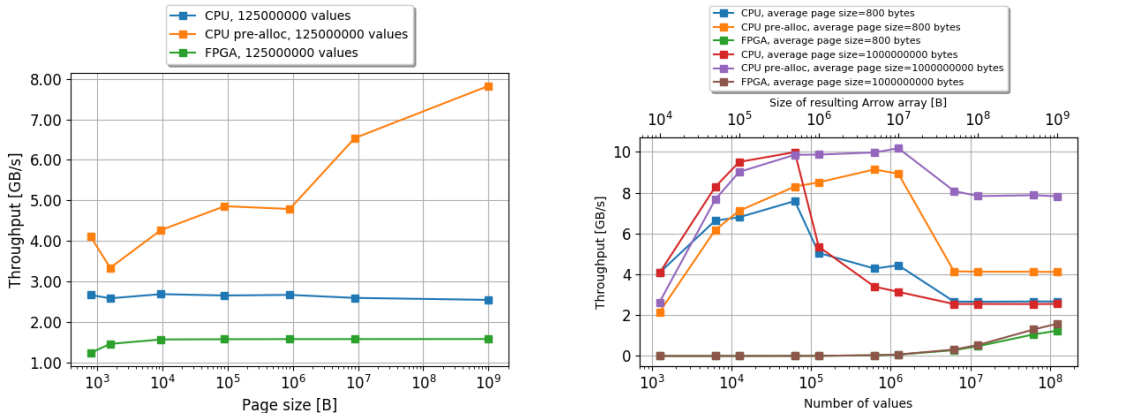
Figure 3.17 shows the throughput of the FPGA implementation when the time required to copy the resulting Arrow array from AOM to host memory is added to the execution time. This simulates a scenario where the FPGA has direct access to a storage medium with the Parquet data but does not operate in the same shared memory space as the host system. In the same graphs the throughput of the software based Parquet to Arrow converter is plotted.

As expected the pre-allocated version of the CPU implementation is significantly faster than the version that has to allocate the memory itself. Figure 3.17b shows that the pre-allocated version can reach a throughput of almost 8 GB/s for large pages, while the other reaches only 2.6 GB/s due to cache misses in the TLB. Curiously, the throughput is larger when reading small numbers of values. This might be due to the entirety of the data fitting in the 45 MiB level 3 processor cache.

Figure 3.17a shows that in this set-up the FPGA can not beat the CPU in performance, reaching a throughput of at most 2.0 GB/s. Note that when comparing FPGA performance and CPU performance in Fig. 3.17a, one should compare FPGA performance to the slower CPU implementation as the FPGA now also suffers from address translation overhead because of having to copy to host memory.

To complete the picture for the Amazon F1 system, Fig. 3.18 shows what the throughput for the FPGA looks like when the copy from host memory to AOM is also added to the execution time. The total execution time now includes two copies. This shows the performance of the total system when the time required to read from storage into memory (which would impact the CPU and FPGA performance equally) is still disregarded.

According to Fig. 3.18a, the maximum attainable throughput for the FPGA is now only 1.6 GB/s. Figure 3.18b now shows an especially low throughput up to 10^6 values, but this is a misleading



(a) Throughput for varying page sizes

(b) Throughput for varying numbers of values read

Figure 3.18: Throughput in GB/s for Parquet to Arrow conversion with system initialization and copies to and from the AOM taken into account

graph because the currently very simple C++ runtime for communication with the FPGA copies the entire column chunk to the AOM irrespective of the number of values that need to be read from it.

As mentioned before, it is very important to stress that the poor throughput seen in Figs. 3.17 and 3.18 does not mean the Amazon F1 instances can't be used as an efficient system for implementing hardware accelerated Parquet to Arrow conversion. The FPGA can do useful work while host-side processes are copying data to or from the AOM. The DMA drivers on the system even support concurrent accesses from processes or threads on the CPU, meaning the entire process can be fully pipelined. Implementing an end-to-end application that can make optimal use of this specific system is however out-of-scope for this thesis.

3.4 Summary

In this section a system for converting Apache Parquet files to Apache Arrow format was proposed. The different steps involved in the conversion process were analyzed and allocated to either the CPU or the FPGA, taking into the account the strengths of both processors. An overview was given of a system configuration that would allow for both processors to communicate with each other, non-volatile storage and memory with minimal overhead. Since the hardware of such a system was not available, an alternative system configuration to be used for testing was proposed.

With the tasks of the FPGA in the conversion process well defined, a hardware architecture was proposed that would serve as a blueprint for all necessary Parquet page reading functionality. The hardware architecture was designed to allow for a significant degree of modularity, a necessity for a dynamic format such as Parquet that supports many different encoding and compression schemes. This architecture would form the basis for the ParquetReader module, which functions as a single Parquet to Arrow conversion engine. One or more of these engines can be included in an FPGA configuration to allow for conversion of different types of Parquet columns by using Fletcher's bus architecture.

With the high-level view of the ParquetReader hardware completed, multiple VHDL modules had to be designed and developed in order to create a ParquetReader with basic Parquet reading functionality. Through the inclusion of the PlainDecoder module, the ParquetReader would gain the ability to convert non-nested, non-nullable Parquet columns containing fixed-width primitives to Arrow format.

The basic ParquetReader was implemented in an XCVU9P FPGA on an Amazon F1 instance. The ParquetReader module has shown great area efficiency, allowing many Parquet to Arrow

conversion engines to be instantiated in parallel, or allowing for the inclusion of custom hardware operating on the data produced by the ParquetReaders.

After completing development of a simple C++ runtime based on Fletcher's runtime, the completed ParquetReader could be tested on an actual FPGA for simple Parquet files. Performance measurements have shown that the ParquetReader is fastest for Parquet files with large page sizes, as predicted. If no computationally intensive decoding or decompression is required and the page size is large enough, the ParquetReader will be limited by the combined read and write bandwidth between FPGA and memory. This bandwidth was shown to be 7.2 GB/s on the system used for testing. By reaching this bandwidth for relatively small page sizes, it was shown that the basic design would be able to feed decoding or decompression hardware included with the ParquetReader at sufficient speed.

Finally, the limits of the Amazon F1 instance used for testing were explored. This allowed the previously performed measurements of the performance upper bounds of the ParquetReader to be put in the context of the larger system. Through the creation of a simple C++ based Parquet reading library for performance comparisons to the FPGA, the merits (or lack thereof) of using hardware acceleration on this particular system were investigated. It was determined that a naive implementation with sequential copying between host memory and AOM causes the hardware accelerated Parquet reading approach to perform badly when compared to a CPU-only implementation. If the copies to and from AOM and the FPGA processing time were to be pipelined, a significantly better performance would be found for the hardware accelerated system.

Chapter 4

Delta and delta length decoding

4.1 Motivation

In order to make the the Parquet to Arrow converter practically useful, it should have the ability to read Parquet columns that contain either fixed-width primitives or strings. With the ParquetReader module as described in Chapter 3, reading plain encoded fixed-width primitives is successfully supported. It might be tempting to enable support for string reading by making a decoder for plain encoded strings. However, this will most likely not result in an efficient design.

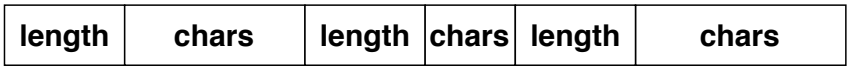


Figure 4.1: Data layout of plain encoded strings in a Parquet page

Figure 4.1 shows the data layout of plain encoded strings in a Parquet page. Lengths of strings are stored as 32-bit integers, after which the number of characters indicated by the length follows. This interleaved length and characters combination repeats throughout the page for every string. When this data is streaming into the decoder, the decoder can only know where in the data the length of the second string is if it has read the length of the first one. Once it has this information, the data needs to be shifted with a number of bytes equal to the length plus four to align the data for the next string. This way, only one string can be streamed to the ArrayWriter for each shift. Adding insult to injury is the fact that implementing single-cycle barrel shifters that can shift an arbitrary number of bytes can result in very significant area usage in the FPGA. This problem can be solved by reducing the bus width or the maximum number of bytes it can shift, but both these solutions will negatively impact the throughput of the design.

While certainly a difficult task, it may not be impossible to create a design that will result in a larger throughput than the strategy described above. The question that must now be asked is whether or not one should. In Parquet v2.0 plain encoding is not the string encoding that will result in the best compression ratio. The "DELTA_LENGTH_BYTE_ARRAY" described in Section 2.2.3 and the similar but slightly more complex "DELTA_BYTE_ARRAY" encoding (default for strings) will create significantly smaller Parquet files. A Parquet to Arrow converter that is able to read these encodings will be of much better use in an actual application. Because the lengths of the strings are stored contiguously (as bit-packed deltas) and separate from the characters, a decoder for delta length encoded strings does not suffer from the same limitations a decoder for plain encoded strings may suffer from. As an added benefit, developing a decoder for delta length encoded strings requires much of the same logic a delta decoder for integers requires. Thus, the development of a delta length decoder also results in a functional decoder for the standard encoding for integers in Parquet.

Such a project is not without its challenges. An efficient decoder for plain encoded strings is difficult to develop, but developing a working one is quite simple. For delta length decoding, the logic required for reading block headers, unpacking (variable width) bit-packed deltas, and adding the deltas, is complicated. However, once working it stands a much better chance at

outperforming a CPU implemented delta (length) decoder, which will serve in answering the research question. To that end, it was decided to implement both a delta decoder and a delta length decoder. "DELTA_BYTE_ARRAY" decoding will not be implemented in this thesis due to its increased complexity in comparison to delta length decoding. Such a decoder would however use much of the same hardware as a delta length decoder, so future work can implement it starting from the work done in this thesis.

4.2 Design overview

As mentioned in Section 2.2.3, a delta length encoded Parquet page contains a block of delta encoded string lengths, directly followed by a contiguous list of the characters in the strings. Therefore, a delta length decoder can be built from a delta decoder that decodes the integer string lengths and a module that streams the correct number of characters to the Fletcher ArrayWriter. Figure 4.2 shows a design that uses this idea to allow for a lot of reusable code. The white modules in the architecture will be made so that they can function together as a DeltaDecoder module. Some of these modules will get optional output streams that go unused in a normal DeltaDecoder module, but can connect to the yellow module in order to make the total design function as a DeltaLengthDecoder.

Data streaming into the decoder first needs to pass through the DeltaHeaderReader, which extracts the relevant values from the delta header and then aligns the output data to the first block header. "First value" is needed not only as a value that needs to be sent to the ArrayWriter, but also as a starting point for determining all the integers in the page from the deltas. "Total value count" is not needed as this has already been determined by the MetadataInterpreter from the page header. The "block size in values" and "number of miniblocks" are important for decoding the packed deltas in the miniblocks, but it was decided that they should be set at compile time. While theoretically these values could be changed to allow for slightly more efficient encoding of some types of datasets, parquet-mr sets these values as constants in the source code, with "block_size_in_values" being set to 128 and "number_of_miniblocks" being set to 4. Tweaking these parameters therefore requires changing the source. Even if someone decided to do this, they would most likely keep these encoding parameters constant throughout the column, as such a decision would likely be based on a very specific pattern in the dataset that is stored in this column. By keeping "block size in values" and "number of miniblocks" constant the hardware becomes less complex and will require less area due to the simplification in the logic that deals with counting the values in the miniblocks.

The StreamSerializer (provided by the vhlib library accompanying Fletcher), will split each bus word into multiple parts and send them downstream sequentially. This narrows the stream, allowing for smaller decoding logic downstream. Due to the bit-packed nature of the encoding, 64 bits could already contain many values depending on the bit-packing width. In the case of small deltas, a bit-packing width of 4 bits is possible for example, meaning those 64 bits would contain 16 values. Because determining the values in the column requires a prefix sum of the deltas, only a limited number of deltas can be unpacked each clock cycle or the module responsible for the prefix sum will encounter timing issues. Therefore, narrowing the stream is a good way to save area without losing performance. The serialization ratio can be selected through VHDL generics. The effects of different settings of this feature will be discussed in Section 4.4.

The BlockValuesAligner, the BitUnpacker and the DeltaAccumulator are responsible for the transformation of bit-packed deltas to actual integers. The BlockValuesAligner contains a BlockHeaderReader for reading the block headers and a BlockShiftControl that uses the information from the BlockHeaderReader to determine how many values can be extracted from the encoded data per cycle. The BlockShiftControl sends this information to the BitUnpacker, an array of shift and mask pipelines that transforms the bit-packed deltas into full width integers. The DeltaAccumulator adds the minimum delta received from the BlockHeaderReader to these unpacked deltas, and then computes the prefix sum. The results are then streamed to the Fletcher ArrayWriter. The DeltaAccumulator also keeps track of the total number of values decoded so far, and signals completion to the ArrayWriter when it has determined all required values in the column chunk have been read.

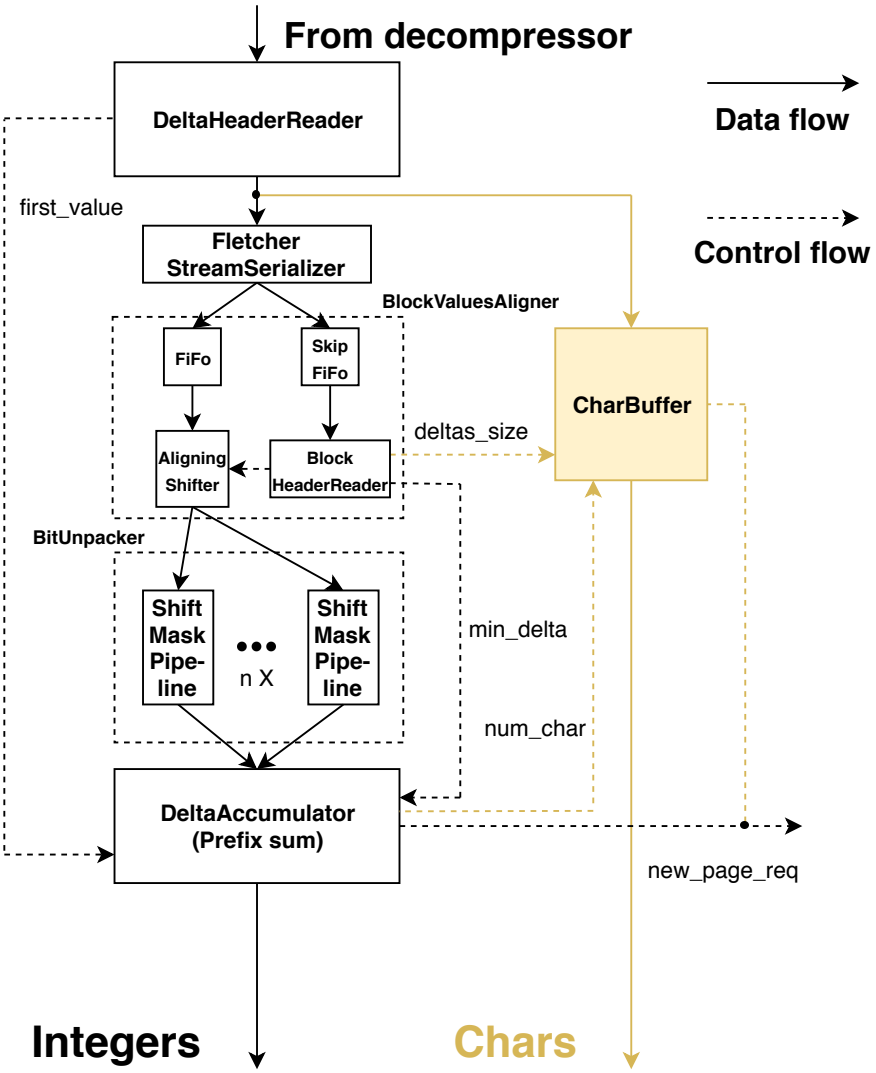


Figure 4.2: Hardware architecture of the DeltaDecoder with the optional CharBuffer for DeltaLengthDecoder functionality in yellow

A CharBuffer can be added to the aforementioned collection of modules to create a DeltaLengthDecoder. The CharBuffer will copy the data sent to the StreamSerializer and buffer it. The BlockHeaderReader will continuously send data to the CharBuffer that tells it how many bytes in the page are part of the encoded string lengths, allowing the CharBuffer to delete those bytes from its buffer. Once all string lengths have been calculated the DeltaAccumulator will stream the sum of all the string lengths to the CharBuffer so it knows how many characters it needs to stream to the ArrayWriter. If the current page happens to be the last one, the DeltaAccumulator will include that information in the transfer as well. Once the CharBuffer has received this value and a "done" signal from the BlockValuesAligner, it knows it does not have to delete bytes belonging to the string lengths anymore, meaning the rest of its buffered bytes are part of the string characters.

The CharBuffer and the DeltaAccumulator jointly determine when a page has been fully processed by counting characters or string lengths respectively. In the DeltaDecoder module no CharBuffer is needed, meaning the DeltaAccumulator decides by itself. Upon completion of the three-way handshake between the DeltaAccumulator, CharBuffer and the PreDecBuffer (who determines when a new page can be streamed to the decompressor and decoder as discussed in Section 3.2.4), all modules in the DeltaDecoder or DeltaLengthDecoder will be reset except the

CharBuffer and the DeltaAccumulator. The DeltaAccumulator needs to keep its state to track the total number of values read from the column chunk while the CharBuffer uses the handshake as a directive to proceed to its string length skipping state.

Instead of using the CharBuffer, a DeltaLengthDecoder could also have been implemented by using a DataAligner module as discussed in Section 3.2.2. The DataAligner would have two consumers, a DeltaDecoder and a module responsible for writing characters to the ArrayWriter. This second module would be similar to a PlainDecoder as described in Section 3.2.4. The DeltaDecoder would have to report the size in bytes of the delta encoded string lengths so the DataAligner could realign the incoming data for the string characters. This design could be quickly implemented because it uses previously developed and tested modules. It was decided not to go with this strategy because Section 3.3.1 showed the DataAligner to be the largest module in the ParquetReader. A custom CharBuffer might save resources.

Putting the modules described above together, a DeltaDecoder or DeltaLengthDecoder module is created with the following adjustable settings:

1. **BUS_DATA_WIDTH:** Number of bits transferred to the DeltaHeaderReader from the decompressor each clock cycle.
2. **DEC_DATA_WIDTH:** Number of bits transferred from the StreamSerializer to the BlockValuesAligner each cycle.
3. **LENGTHS_PER_CYCLE/ELEMENTS_PER_CYCLE:** Maximum number of integers calculated and transferred to the Fletcher ArrayWriter each cycle. Setting this too high may result in timing issues due to the large prefix sum required.
4. **CHARS_PER_CYCLE:** Maximum number of characters transferred to the Fletcher ArrayWriter each cycle (only for the DeltaLengthDecoder).
5. **PRIM_WIDTH:** Whether to decode 64-bit or 32-bit integers (only for the DeltaDecoder).

In order to reduce the complexity of the design, only powers of 2 are supported for all available settings.

The logic included within the DeltaDecoder and DeltaLengthDecoder VHDL modules themselves is limited. The only logic present is a process statement that counts the number of bytes received from the decompressor, while ensuring no more bytes than actually present in one page are consumed until the CharBuffer and the DeltaAccumulator are ready for a new page. Should the CharBuffer or DeltaAccumulator for some reason decide to signal the end of a page before all the bytes in it have been received by the decoder, this process will ensure that those bytes are consumed from the decompressor and then thrown away in order to allow the decompressor to finish its job and prepare it for the new page.

4.3 Module design

4.3.1 DeltaHeaderReader

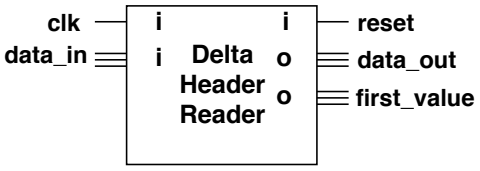


Figure 4.3: Top-level view of the DeltaHeaderReader module

The DeltaHeaderReader is build in a very comparable way to the MetadataInterpreter. It simply shifts through the input data byte by byte while using a VarIntDecoder to decode the VarInt

encoded integers. Once the entire delta header has been read in this way, a pipelined barrel shifter (supplied by vhlib) is used to align the data such that the block header starts at the most significant byte of the data word supplied to the downstream modules. In the current implementation of the DeltaDecoder, only the "first_value" field of the delta header is actually required, which the DeltaHeaderReader streams to the DeltaAccumulator. The DeltaHeaderReader needs to be reset in order for it to process a new delta header, which happens upon a handshake between the DeltaDecoder and the PreDecBuffer preceding a new page.

4.3.2 BlockValuesAligner

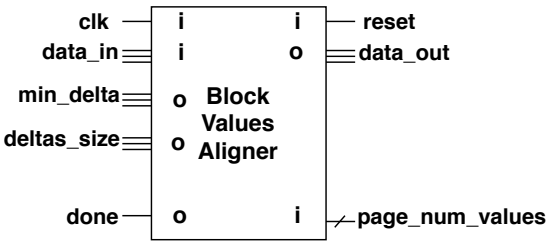


Figure 4.4: Top-level view of the BlockValuesAligner module

Figure 4.4 shows the top-level view of the BlockValuesAligner. The job of this module is to take the data from the StreamSerializer that contains interleaved block headers and miniblock data, and supply the BitUnpacker with data to unpack, accompanied by the width of the bit-packed deltas and the number of deltas it should unpack from the data.

The internals of the BlockValuesAligner can be found in Fig. 4.5. The main challenge this architecture is trying to solve is how to avoid stalling the pipeline every time a block header needs to be read. A naive implementation could read the block header, then proceed to use the information in the header to process the miniblocks, and then repeat. Reading the block header is a multi cycle process, since it first has to decode a VarInt (minimum delta) byte by byte and then read the bit widths.

A combinatorial VarIntDecoder could be made that can decode the VarInt in one cycle. A single-cycle shifter could then be included which can shift between 1 and the maximum number of bytes in the VarInt. This would reduce the reading of the block header to one cycle for the VarInt and one cycle for the bit widths, but it would result in a significant combinatorial path that could cause timing issues. Also, it would still require 2 clock cycles which can significantly impact the performance considering a block header is included in the page after every 128 values by default.

Therefore, the BlockValuesAligner was designed in a way that allows the block headers and miniblock data to be processed concurrently by two separate internal modules. A BlockHeaderReader for reading block headers and a BlockShiftControl for supplying data to the BitUnpacker. A module from vhlib called StreamSync is used to synchronize the input between the two internal modules to ensure they always consume the exact same data at the exact same time, essentially duplicating the input data stream. Both modules have FIFO buffers at their inputs, with the BlockHeaderReader's (advanceable) FIFO having custom functionality that allows the BlockHeaderReader to delete any number of data words from it at any point.

The BlockHeaderReader reads the minimum delta and the bit widths from the block header and passes the minimum delta to the DeltaAccumulator and the bit widths to the BlockShiftControl. Due to the number of miniblocks in a block (and therefore the number of bit widths in the block header) being set at synthesis time, the BlockHeaderReader knows the number of bytes in the block header immediately after reading the VarInt. This information is sent to the BlockShiftControl. After reading the bit widths, the BlockHeaderReader can calculate the number of bits in the miniblock data by multiplying the sum of the bit widths by the number of values in a miniblock. The BlockHeaderReader now deletes all the miniblock data in its AdvanceableFiFO buffer, allowing it to start reading the next block header while the BlockShiftControl is still processing the miniblock data. Every time the length of a block header or the size of the miniblocks

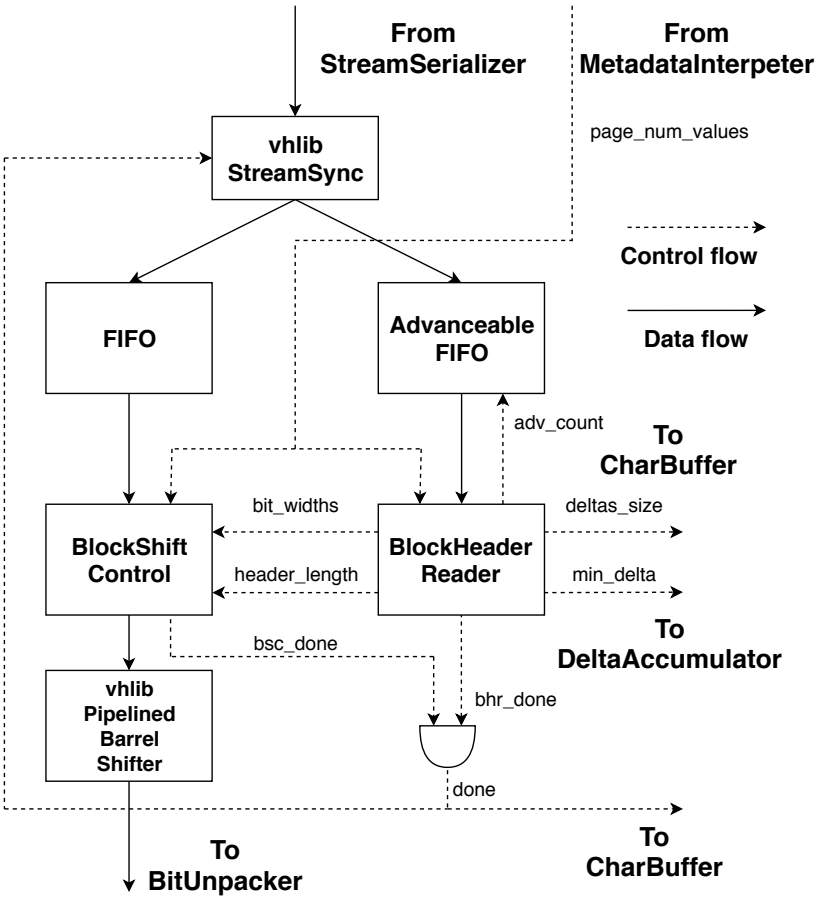


Figure 4.5: Hardware architecture of BlockValuesAligner

is determined, this information is also sent to the CharBuffer to allow it to skip through the string length data.

Every time the BlockShiftControl receives a block header length from the BlockHeaderReader, it skips the specified number of bytes and starts reading the miniblocks. Every data word received from upstream (whose width is specified by DEC_DATA_WIDTH) can contain a certain number of deltas, dependent on the bit-packing width in the miniblock the data word is a part of. The module contains two lookup tables, generated based on DEC_DATA_WIDTH and the LENGTHS_PER_CYCLE or ELEMENTS_PER_CYCLE settings that limit the maximum number of unpacked deltas per cycle. With the bit widths received from the BlockHeaderReader, the BlockShiftControl can use the lookup tables to determine how many bits to transfer to the BitUnpackers and the number of values the downstream module should unpack. The BlockShiftControl uses vhlib's pipelined barrel shifting logic to correctly align its output data for the BitUnpackers.

This parallel block reading configuration works because the BlockShiftControl does not always consume one data word per cycle. Instead it always consumes a set number of bit-packed values, which (depending on the bit-packing width) does not necessarily add up to an entire data word. This behavior causes backpressure in the decoder, which allows the FIFO buffers before the BlockHeaderReader and the BlockShiftControl to fill up, giving the BlockHeaderReader the opportunity to "look ahead" in the stream. Simulation has confirmed that the BlockShiftControl now only stalls its outputs for one cycle every block to shift out the block header, except in rare cases where the bit-packing width causes the BlockShiftControl to consume its input data at one data word per cycle. Whether or not this behavior occurs depends on the input data, DEC_DATA_WIDTH and the maximum number of deltas consumed per cycle. For example, if the bit-packing width in a certain part of the Parquet file is 16 bits while DEC_DATA_WIDTH is

64 bits, the BlockShiftControl can consume 4 bit-packed values per cycle for a total of 64 bits.

Both the BlockShiftControl and the BlockHeaderReader count how many values have been processed, and independently determine completion of a page by comparing this count to the total number of values in the page as determined by the MetadataInterpreter. Once they have both given their done signals, the CharBuffer knows it can start processing string characters, while the StreamSync in the BlockValuesAligner starts consuming data words from upstream unimpeded to avoid backpressuring the entire decoder.

4.3.3 BitUnpacker

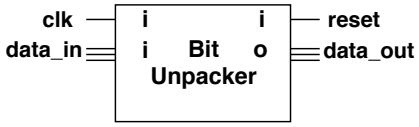


Figure 4.6: Top-level view of the BitUnpacker module

The BitUnpacker module takes data from the BlockValuesAligner, and based on the number of packed deltas and the bit-packing width supplied with the input data it unpacks the deltas and passes them to the DeltaAccumulator.

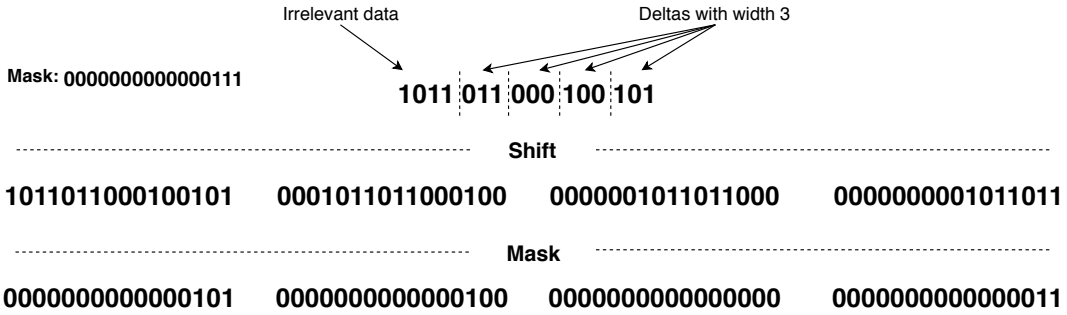


Figure 4.7: Unpacking example for DEC_DATA_WIDTH 16 with a bit-packing width of 3 bits and an unpacking count of 4

Figure 4.7 shows a bit-unpacking example in the case of a DEC_DATA_WIDTH set to 16 bits. In this particular example the BlockValuesAligner has supplied data that contains 4 deltas packed to 3 bits. The input data word is copied to 4 pipelined shifters that shift the data with an number of bits equal to the bit-packing width multiplied by their location in the array of shifters. This results in four data words that each have one of the four bit-packed values aligned to the right. A mask is then chosen based on the bit-packing width and through an AND operation with the shifted data the four values are retrieved. Note that the 16 bit results from the example in Fig. 4.7 can be sign extended to any desired width.

Figure 4.8 shows how this technique can be implemented in hardware. A number of BitUnpackerShifters is implemented equal to the maximum number of unpacked deltas per cycle. These BitUnpackerShifters perform the shifting operation as specified in the example. A FIFO is also included that stores the bit-packing width and value count for each data word entered into the BitUnpacker. The width is used to select a mask in the lookup table which can be used in the AND operation on the results of the BitUnpackerShifters. The results from these mask operations are sent (synchronized) to the DeltaAccumulator together with the count value stored in the FIFO, so the DeltaAccumulator knows how many values in the transferred data are valid.

4.3.4 DeltaAccumulator

The DeltaAccumulator module takes the unpacked deltas from the BitUnpacker, the first value from the DeltaHeaderReader, and the minimum delta from the BlockValuesAligner, and uses

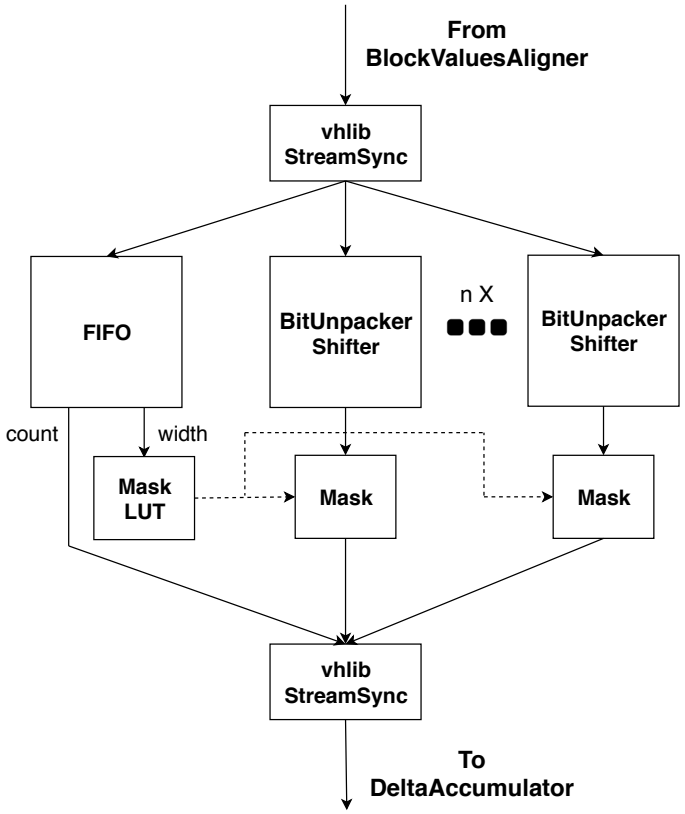


Figure 4.8: Hardware architecture of the BitUnpacker with "n" being the maximum number of deltas unpacked per clock cycle

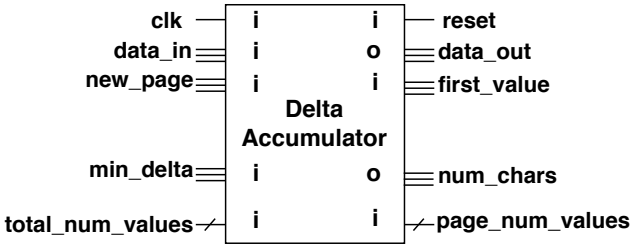


Figure 4.9: Top-level view of the DeltaAccumulator module

those values to calculate the integers stored in the Parquet page as the final decoding result. The structure of the DeltaAccumulator is not very complicated, as it is essentially only a few sum operations. Unfortunately, since a prefix sum is a sequential addition of values, only a finite number of values can be summed in the same clock cycle before time runs out. This means the DeltaAccumulator limits the maximum number of values that can be decoded in one cycle.

There are two stages in the DeltaAccumulator, with vhlib StreamSlices before, after, and in between to break up combinatorial paths as much as possible. The first stage adds the minimum delta received from the BlockValuesAligner to each of the unpacked values received from the BitUnpacker to create the final deltas. A counter is used to keep track of the number of values processed so the DeltaAccumulator can consume a new minimum delta from the BlockValuesAligner after processing every value in a block. The second stage calculates the actual prefix sum using the first value received from the DeltaHeaderReader as a starting point and sends the decoding results to the ArrayWriter. An extra responsibility of the second stage is to keep track of the total number of values processed in the column chunk, and notify the ArrayWriters once all the requested values have been decoded. This value count tracking is also used to detect page

completion.

Apart from the main data flow in the DeltaAccumulator, some extra logic is included in the case of delta length decoding. This logic inspects the decoded string lengths streaming out of the second stage and sums all their values. Once the page is completed, the total sum is sent to the CharBuffer where it is used to determine how many bytes need to be streamed to the ArrayWriter.

4.3.5 CharBuffer

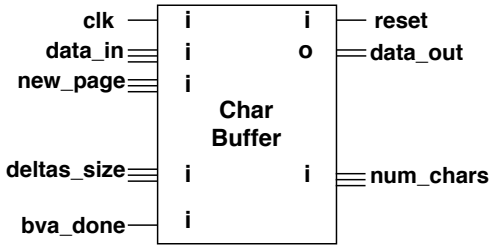


Figure 4.10: Top-level view of the CharBuffer module

The CharBuffer module is only included in the DeltaLengthDecoder and is used to extract the characters from the Parquet page and send them to the ArrayWriter. An AdvanceableFIFO as discussed in Section 4.3.2 is used as an input buffer that takes raw data from the DeltaHeaderReader. Since this raw data also includes all the string length data, the CharBuffer requires the BlockValuesAligner to tell it how much data it can delete from its input buffer until only character data is left. Once the BlockValuesAligner gives the signal, the CharBuffer requests the DeltaAccumulator to send it the sum of the string lengths. Once the number of characters streamed to the ArrayWriter equals the sum of the string lengths, the CharBuffer is done with the page and together with the DeltaAccumulator it now handshakes the PreDecBuffer to start the next page.

Setting the CHARS_PER_CYCLE setting in the DeltaLengthDecoder to be equal to the number of bytes in the input bus width (BUS_DATA_WIDTH/8) allows the CharBuffer to process a full input bus word per cycle. It is however possible to set the CHARS_PER_CYCLE to a value lower than that, in which case a vhlib StreamSerializer is used to narrow the output stream.

4.4 Implementation & evaluation

In this section the design will be evaluated in terms of area and performance when implemented on the same system used in Section 3.3: an Amazon F1 instance including a Xilinx XCVU9P FPGA and an Intel Xeon E5-2686 v4 processor of which 8 hardware threads are available.

As predicted in Section 4.2, large prefix sums in the DeltaAccumulator modules caused Vivado to be unable to meet timing constraints for the selected 250 MHz clock in the XCVU9P FPGA. Because this issue was predicted the design allowed for setting a limit to the number of values processed per cycle. The build only succeeds if the limit is set to 4 or fewer values per cycle. Setting the limit to 8 (to reduce complexity the design only allows for selecting powers of 2) results in a path through the DeltaAccumulator with a worst negative slack of -0.799 ns on a clock cycle of 4 ns.

Because limiting the number of values processed per cycle negatively impacts the throughput of the design, finding a solution to this problem would be beneficial to performance. Unfortunately, no custom VHDL solution was found that allowed Vivado synthesize a design that processes 8 values per cycle. Through the use of Vivado IP cores specifically optimized for calculating prefix sums, processing 8 values per cycle may be possible, but this would cause the design to stop being vendor agnostic. For both this reason and the extra development time required to implement such a solution it was decided to accept the 4 values per cycle limitation and only perform benchmarks and testing on implementations conforming with that limitation.

As discussed in Section 4.2, the number of bits processed in the BlockValuesAligner and the BitUnpacker per cycle can be set through the DEC_DATA_WIDTH setting (hereafter referred to

as the decoder width). An interesting consequence of the 4 values per cycle limitation is that it allows for narrow decoders when decoding Parquet columns with small deltas and narrow bit-packing widths. For example, if it is known for a Parquet column that all deltas are bit-packed to at most 4 bits, a decoder width of 16 bits would be enough as one word entering the decoder would then always contain the maximum number of deltas that can be processed in a cycle. For this reason, having a delta decoder for 32-bit integers with a decoder width larger than 128 bits would not make sense, as a 128-bit word can contain 4 deltas packed to the maximum bit-packing width of 32. The same is true for 64-bit integer delta decoders that have a decoder width larger than 256 bits.

4.4.1 Area

Tables 4.1 to 4.3 show the area utilization hierarchy of ParquetReaders containing a DeltaDecoder for 32-bit integers with decoder widths of either 64 bits or 128 bits. In these tables "Others" refers to the modules already discussed in Chapter 3. As can been seen in the tables, LUT, register and BRAM utilization for DeltaDecoders is very low. Instantiating ParquetReaders with large decoder widths seems to increase area utilization as expected. 128-bit width delta decoders use 24.4% more LUTs, 25.5% more registers and 62.3% more BRAM than their 64-bit counterparts. As expected, the growth comes from the BitUnpacker and BlockValuesAligner modules. These modules form the part of the decoder after the StreamSerializer that narrows the stream to the set decoder width, but before the DeltaAccumulator whose input port widths are affected by the number of values processed per cycle rather than the decoder width. While the growth in area utilization is definitely significant, it may still be worth it in performance returns that may occur when the dataset contains large deltas requiring larger bit-packing widths. In such cases a larger decoder width may allow more bit-packed deltas to fit in one decoder word, which increases the number of values that can be processed in a single cycle (with a maximum of 4). The effects on performance will be investigated in Section 4.4.2.

Component	**64-bit decoder**	**128-bit decoder**
ParquetReader	1.46% (17221)	1.55% (18282)
DeltaDecoder	0.45% (5326)	0.56% (6644)
DeltaHeaderReader	0.20% (2382)	0.22% (2657)
Fletcher StreamSerializer	0.00% (7)	0.00% (11)
BlockValuesAligner	0.07% (833)	0.12% (1410)
BitUnpacker	0.07% (874)	0.12% (1422)
DeltaAccumulator	0.08% (967)	0.09% (1078)
CharBuffer	0.00% (0)	0.00% (0)
Others	1.01% (11895)	0.99% (11638)

Table 4.1: LUT utilization for 32-bit integer delta decoders with varying decoder widths

Component	**64-bit decoder**	**128-bit decoder**
ParquetReader	1.50% (35348)	1.61% (38159)
DeltaDecoder	0.47% (11175)	0.59% (13986)
DeltaHeaderReader	0.22% (5211)	0.22% (5221)
Fletcher StreamSerializer	0.02% (516)	0.02% (515)
BlockValuesAligner	0.07% (1712)	0.13% (3147)
BitUnpacker	0.10% (2266)	0.15% (3631)
DeltaAccumulator	0.06% (1378)	0.06% (1380)
CharBuffer	0.00% (0)	0.00% (0)
Others	1.11% (27143)	1.03% (24173)

Table 4.2: Register utilization for 32-bit integer delta decoders with varying decoder widths

Component	64-bit decoder	128-bit decoder
ParquetReader	2.85% (61.5)	2.99% (64.5)
DeltaDecoder	0.53% (15.5)	0.86% (18.5)
DeltaHeaderReader	0.35% (7.5)	0.35% (7.5)
Fletcher StreamSerializer	0.00% (0.0)	0.00% (0.0)
BlockValuesAligner	0.19% (4.0)	0.32% (7.0)
BitUnpacker	0.00% (0.0)	0.19% (4.0)
DeltaAccumulator	0.00% (0.0)	0.00% (0.0)
CharBuffer	0.00% (0)	0.00% (0)
Others	2.13% (46.0)	2.13% (46.0)

Table 4.3: BRAM tile utilization for 32-bit integer delta decoders with varying decoder widths

In Section 3.3.1 it was determined that the upper bound of the number of ParquetReaders that can fit in a single XCVU9P FPGA on an Amazon F1 system was limited by the amount of BRAM used by the ParquetReaders and the static Amazon shell. The shell uses 11.41% of BRAM, leaving 88.59% for ParquetReaders. This is enough for 31 ParquetReaders with 64-bit decoders or 29 ParquetReaders with 128-bit decoders. On a system that does not require the Amazon shell these numbers would be 35 and 33 ParquetReaders respectively. As mentioned in Section 3.3.1, instantiating many ParquetReaders in a single FPGA will likely cause congestion issues making this number an upper bound for the maximum number of ParquetReaders, which in practice will most likely be a smaller amount. Still, the area efficiency of the ParquetReaders with DeltaDecoders allows for a significant amount of parallelization by instantiating multiple engines.

Component	64-bit decoder	128-bit decoder	256-bit decoder
ParquetReader	1.68% (19814)	1.76% (20829)	1.90% (22440)
DeltaDecoder	0.61% (7186)	0.68% (8050)	0.86% (10179)
DeltaHeaderReader	0.27% (3169)	0.25% (2953)	0.24% (2887)
Fletcher StreamSerializer	0.00% (12)	0.00% (5)	0.00% (8)
BlockValuesAligner	0.09% (1048)	0.14% (1645)	0.23% (2717)
BitUnpacker	0.10% (1230)	0.16% (1865)	0.25% (2920)
DeltaAccumulator	0.14% (1712)	0.13% (1581)	0.14% (1644)
CharBuffer	0.00% (0)	0.00% (0)	0.00% (0)
Others	1.08% (12628)	1.09% (12779)	1.05% (12261)

Table 4.4: LUT utilization for 64-bit integer delta decoders with varying decoder widths

Component	64-bit decoder	128-bit decoder	256-bit decoder
ParquetReader	1.64% (38781)	1.76% (41678)	1.99% (46956)
DeltaDecoder	0.54% (12777)	0.66% (15674)	0.89% (20954)
DeltaHeaderReader	0.22% (5252)	0.22% (5259)	0.22% (5256)
Fletcher StreamSerializer	0.02% (516)	0.02% (515)	0.12% (514)
BlockValuesAligner	0.07% (1754)	0.14% (3198)	0.26% (6253)
BitUnpacker	0.11% (2657)	0.17% (4107)	0.27% (6334)
DeltaAccumulator	0.11% (2506)	0.11% (2503)	0.11% (2505)
CharBuffer	0.00% (0)	0.00% (0)	0.00% (0)
Others	1.11% (26004)	1.11% (26004)	1.11% (26002)

Table 4.5: Register utilization for 64-bit integer delta decoders with varying decoder widths

For DeltaDecoders that can decode 64-bit integers the area utilization hierarchy can be found in Tables 4.4 to 4.6. Like their 32-bit counterparts, these DeltaDecoders again show great area efficiency. Although the larger number of bits in the 64-bit integers require DeltaDecoders with

Component	64-bit decoder	128-bit decoder	256-bit decoder
ParquetReader	2.66% (57.5)	2.99% (64.5)	3.24% (70.0)
DeltaDecoder	0.53% (11.5)	0.86% (18.5)	1.11% (24.0)
DeltaHeaderReader	0.35% (7.5)	0.35% (7.5)	0.35% (7.5)
Fletcher StreamSerializer	0.00% (0.0)	0.00% (0.0)	0.00% (0.0)
BlockValuesAligner	0.19% (4.0)	0.32% (7.0)	0.58% (12.5)
BitUnpacker	0.00% (0.0)	0.19% (4.0)	0.19% (4.0)
DeltaAccumulator	0.00% (0.0)	0.00% (0.0)	0.00% (0.0)
CharBuffer	0.00% (0)	0.00% (0)	0.00% (0)
Others	2.13% (46.0)	2.13% (46.0)	2.13% (46.0)

Table 4.6: BRAM tile utilization for 64-bit integer delta decoders with varying decoder widths

Component	LUTs	Registers	BRAM tiles
ParquetReader	2.79% (32959)	2.92% (68996)	4.47% (96.5)
DeltaLengthDecoder	0.91% (10732)	0.95% (22512)	1.55% (33.5)
DeltaHeaderReader	0.23% (2746)	0.22% (5212)	0.35% (7.5)
Fletcher StreamSerializer	0.00% (6)	0.02% (515)	0.00% (0.0)
BlockValuesAligner	0.12% (1395)	0.13% (3169)	0.32% (7.0)
BitUnpacker	0.14% (1633)	0.15% (3625)	0.19% (4.0)
DeltaAccumulator	0.09% (1044)	0.06% (1413)	0.00% (0.0)
CharBuffer	0.33% (3905)	0.36% (8484)	0.69% (15.0)
Others	1.89% (22227)	1.97% (46484)	2.92% (63.0)

Table 4.7: Area utilization for a delta length decoder with 128-bit decoder width

slightly more area, the general trends in area increases for larger decoder widths resemble those seen in Tables 4.1 to 4.3. BRAM is again the resource that limits the possible number of ParquetReaders in the FPGA. When the Amazon shell is taken into account the upper bound for the maximum number of ParquetReaders is 33 for the smallest decoder width and 27 for the largest decoder width. For systems without the Amazon Shell this is 37 and 30 ParquetReaders respectively.

A DeltaLengthDecoder was also implemented on the test system. As discussed in Section 4.1 and Section 4.2, a DeltaLengthDecoder is essentially a 32-bit delta decoder plus a CharBuffer module. Because this DeltaLengthDecoder was implemented with a 128-bit decoder width, the numbers in Table 4.7 resemble those found in for the 32-bit integer DeltaDecoder with 128-bit decoder width in Tables 4.1 to 4.3.

The resources used by the CharBuffer cause a significant increase in resource utilization when compared to the DeltaDecoders discussed in this section. Section 4.2 discussed how a DataAligner and a PlainDecoder module could be used instead of a CharBuffer to turn a DeltaDecoder into a DeltaLengthDecoder. The suspicion that such a design strategy would require more area has turned out to be incorrect; the CharBuffer is similar in size to a DataAligner. The CharBuffer-based DeltaLengthDecoder does still have the advantage that it saves a few cycles when compared to the DataAligner-based implementation due to the DataAligner requiring 9 cycles for a realignment.

Apart from the CharBuffer module that requires a significant amount of additional area the Fletcher ArrayWriter has also grown in size compared to the standard DeltaDecoders. This is because string writing ArrayWriters are more complex and require more logic than those that only have to write integers to memory. The upper bound of the total number of ParquetReaders supporting delta length decoding that can fit on an Amazon system is 19. Without the shell the upper bound would be 22. It should be noted that using a decoder with a width of 128 bits is probably unnecessary for practical applications when dealing with strings. String lengths are unlikely to be longer than a couple hundred characters depending on the dataset. This means the deltas stored in the Parquet file are in the order of hundreds as well, which means that they can be packed into a small number of bits. A DeltaLengthDecoder with a width of 64-bits can

perform at the maximum throughput of 4 values per cycle up until a bit-packing width of 16. A 16-bit value allows for deltas in the order of tens of thousands, which is an implausibly large number for string lengths. Narrowing the decoder width will result in area savings for the BlockValuesAligner and BitUnpacker modules.

4.4.2 Performance

In order to properly explore the performance impact of using different decoder widths in the DeltaDecoders, two datasets were generated with varying characteristics.

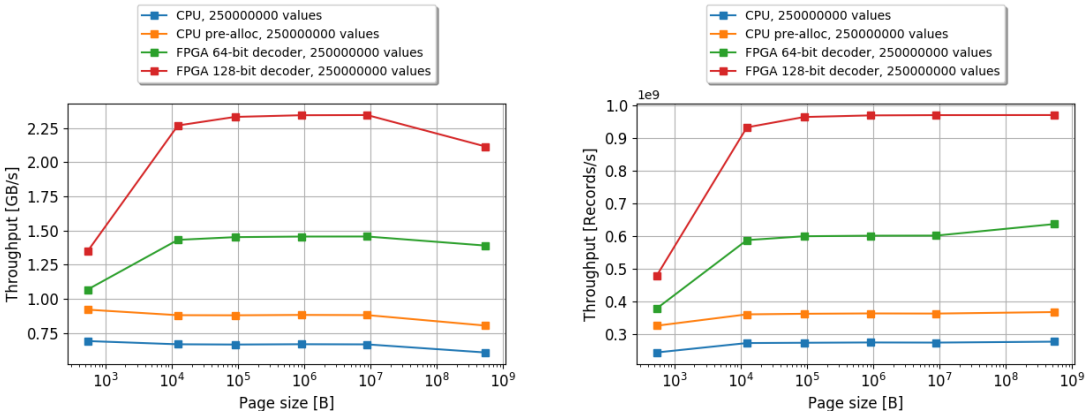
The *random* dataset simply contains integers that are uniformly distributed over the entire range of numbers that can be represented by either 32-bit or 64-bit two's complement representation. Because the number generation is fully random, it is possible that two adjacent number in the columns have a large delta between them. Because of this, many miniblocks in the Parquet column require the maximum 32-bit or 64-bit "bit-packing" width. This makes the dataset very inefficient to store using delta decoding. Larger decoder widths will profit from this as they can handle a larger number of values per cycle that are packed with a large bit-packing width.

The *delta varied* dataset is slightly more complex. A set of uniformly distributed integers is generated as with *random*, but after every 256th value it will select a new modulo 2^x where x is uniformly distributed with $0 \leq x \leq 31$ for the 32-bit integer dataset or $0 \leq x \leq 63$ for the 64-bit dataset. This modulo is then used to limit the randomly generated numbers to a range of $[0, 2^x)$. As a result of limiting the range, the required bit-packing width in the delta encoded Parquet column will change every 256 values. In the range $[0, 2^x)$ the minimum possible delta is $-2^x + 1$ and the maximum delta is $2^x - 1$. As discussed in Section 2.2.3, the minimum delta in a block is stored in the block header. The actual delta can be calculated by subtracting this number from the bit-packed value. Therefore, the difference between the maximum and the minimum possible delta is the largest value that needs to be stored in bit-packed form in a block. This difference is $2^{x+1} - 2$, which requires a maximum bit-packing width of $x + 1$. If every value in the block equals 0 ($x = 0$) no deltas have to be stored.

Note that the bit-packing width in a *delta varied* dataset is not uniformly distributed. If a block contains values from two different ranges the entire block will have the bit-packing width of the larger range due to the very negative minimum delta in the block header, causing the *delta varied* scheme to favor larger bit-packing widths. However, the *delta varied* dataset will still function as an interesting performance comparison to the worst case scenario of the *random* dataset. It is also great as a functional verification for the DeltaDecoder because it will have to be able to decode every possible bit-packing width in varying sequences.

Delta encoded Parquet files containing 250×10^6 32-bit integers were generated according to the previously described schemes, which after conversion will create an Arrow array of 1 GB. The Parquet files were generated with varying page sizes to test the performance of the decoders in multiple scenarios. In order to compare the Parquet to Arrow conversion performance of the FPGA to that of the CPU a C++ application was written that is allowed to make the same assumptions as the FPGA hardware at compile time (as previously seen in Section 3.3.3). As a basis for the fast bit-unpacking, code was used from Lemire [28]. This C++ code was taken and expanded to work for both 32-bits and 64-bits fixed-width primitives. For the same reasons discussed in Section 3.3.3, the performance of the C++ code was measured in two scenario's. One where the memory for the Arrow array was pre-allocated and set to 0 using `memset()`, and one where allocation time was included in the measurement. In both cases the Parquet data is available in host memory before the start of the measurement. The software is compiled using GCC's `-Ofast` and `-march=native` optimization flags.

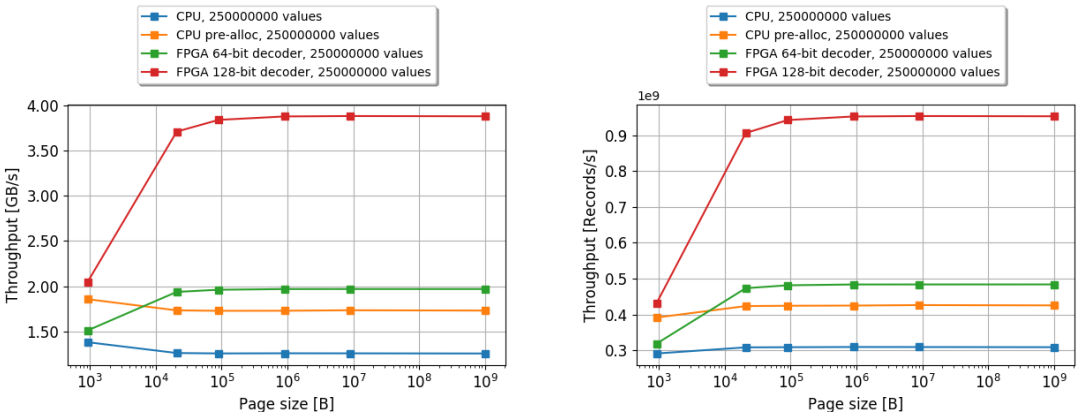
In Fig. 4.11 the results for the *delta varied* dataset can be found while the results for the *random* dataset are shown in Fig. 4.12. In the measurements, throughput is defined as the size of the input Parquet column chunk divided by the runtime of the either the FPGA or the C++ application. As previously done in Section 3.3.2, the measurement for FPGA processing time starts after the Parquet file has been copied to the accelerator card once the start signal has been given to the FPGA, and stops once the FPGA has written the Arrow array to the AOM and it has signaled its completion. The measurement results are shown in GB/s and in records/s. Showing the results only in GB/s would be misleading as the Parquet file containing the *random* dataset is



(a) Throughput in GB/s (b) Throughput in records/s

Figure 4.11: Throughput of ParquetReaders with DeltaDecoders for 32-bit integers decoding a Parquet column containing a *delta varied* dataset as a function of Parquet page size

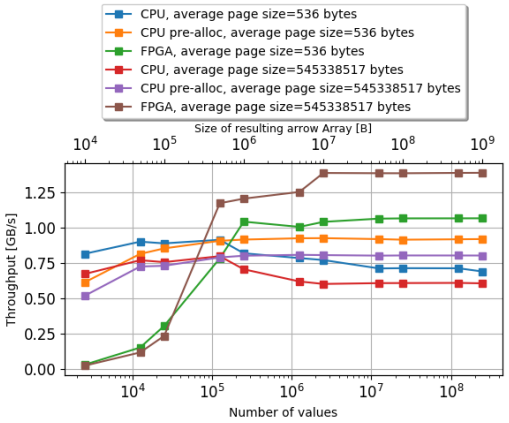
significantly larger than that of the *delta varied* dataset due to the inefficient bit-packing associated with the large deltas. This may cause the throughput in GB/s for the *random* dataset to be higher than that of the *delta varied* dataset, while the actual number of values decoded in a second would be smaller.



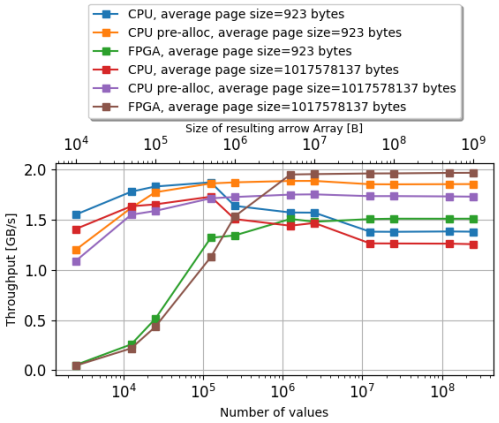
(a) Throughput in GB/s (b) Throughput in records/s

Figure 4.12: Throughput of ParquetReaders with DeltaDecoders for 32-bit integers decoding a Parquet column containing a *random* dataset as a function of Parquet page size

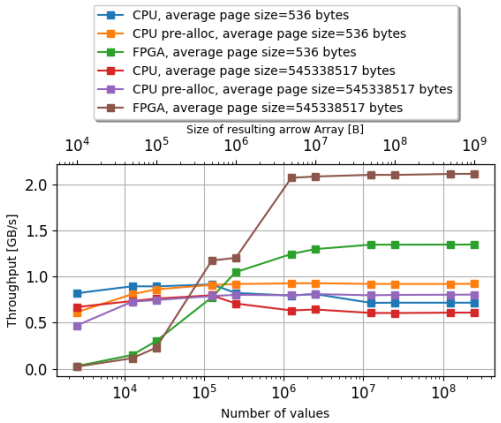
As predicted, the 128-bit decoder can handle both datasets at full speeds. Both the *random* and the *delta varied* dataset reached a throughput of 9.5×10^8 records/s. This number is very close to the expected 1×10^9 records/s that could theoretically be achieved by a perfectly efficient ParquetReader that processes 4 values per cycle on a 250 MHz clock. In this implementation this throughput is not reached because starting a new block in a delta encoded page costs a clock cycle (sometimes two). In terms of GB/s, the 128-bit decoder achieves 3.9 GB/s for the large Parquet file containing the *random* dataset, and at most 2.3 GB/s for the smaller Parquet file containing the *delta varied* dataset. This shows that the throughput in GB/s is very dependent on the compression ratio of the data after encoding. Note that the maximum throughput in GB/s for the *delta varied* dataset is not achieved for the Parquet file with the largest page size. This is because having only one page in the file resulted in a very efficiently encoded Parquet file that is significantly smaller than the other files, while still having to be decoded with the 4 values per cycle limit.



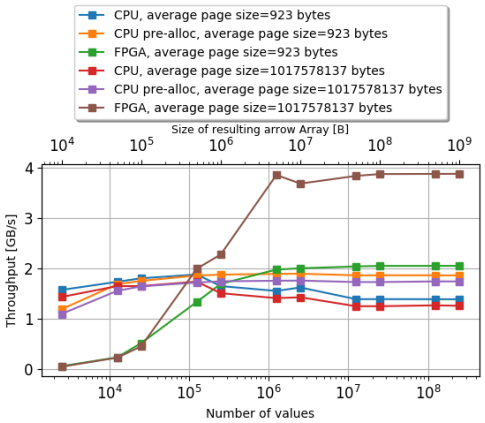
(a) Reading the *delta varied* dataset with 64-bit decoder width



(b) Reading the *random* dataset with 64-bit decoder width



(c) Reading the *delta varied* dataset with 128-bit decoder width



(d) Reading the *random* dataset with 128-bit decoder width

Figure 4.13: Throughput of ParquetReaders in GB/s with DeltaDecoders for 32-bit integers decoding a Parquet column as a function of number of values read

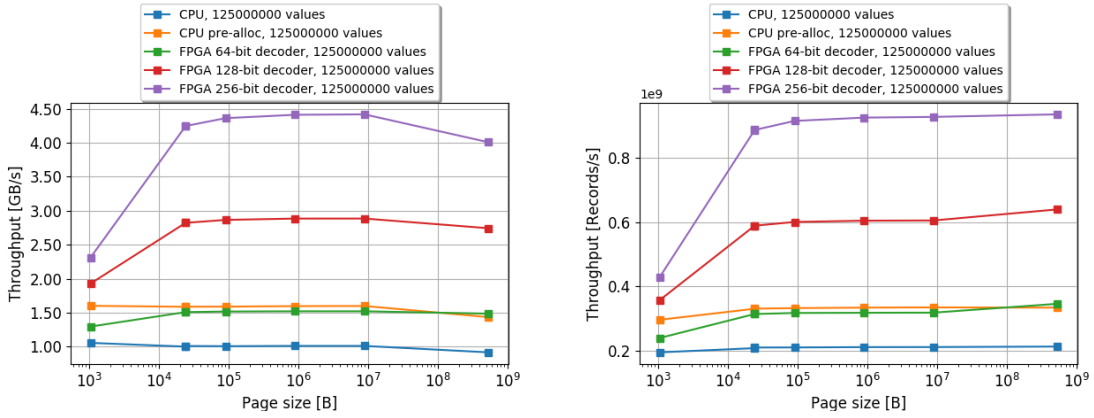
The 128-bit decoder is 1.52× faster than the 64-bit decoder for the *delta varied* dataset, a significant improvement. Especially for the *random* dataset the 64-bit decoder is hindered by the large bit-packing width of the delta encoded values, resulting in the 128-bit decoder being 1.97× faster. Section 4.4.1 showed that the 128-bit decoder required 62.3% more BRAM than the 64-bit decoder. When this growth is compared to the growth in performance, the extra cost associated with the 128-bit decoder may not be worth it in applications where area is at a premium. The decision to use either a 64-bit decoder or a 128-bit decoder should be based on both the nature of the data it will be decoding and the abundance of area on the targeted FPGA.

When the performance of the FPGA based decoders is compared to that of the CPU based program, the results are very promising. For the largest page size and the *delta varied* dataset the 128-bit decoder sees a speedup of 2.63× compared to the CPU pre-allocated implementation. The *random* dataset sees a slightly more modest 2.23× speedup because bit-unpacking is very easy for the CPU when every value is "bit-packed" to 32 bits.

Figure 4.13 shows that using FPGA based ParquetReaders is inefficient for reading small numbers of values from a Parquet column chunk due to the overhead of starting and stopping the hardware, and latency in the ParquetReader. For both the largest and smallest page sizes tested, the maximum throughput is reached around 10^6 values, equivalent to 4 MB of output data.

Figures 4.14 and 4.15 show the throughput of the ParquetReaders containing DeltaDecoders for 64-bit integers as a function of page size compared to the throughput possible on the CPU

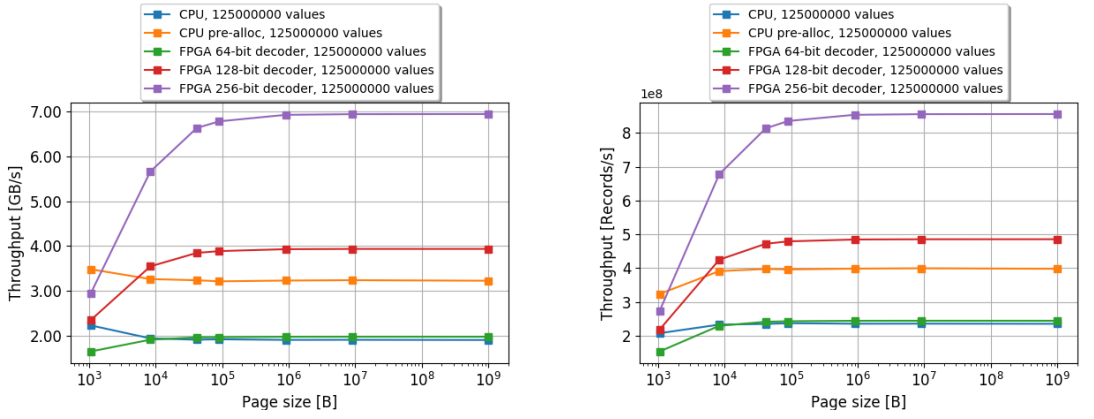
of the Amazon F1 instance. Each of the generated Parquet files contained 125×10^6 64-bit integers so the resulting Arrow array would be 1 GB in size. The same trends can be seen for 64-bit integer decoding as for 32-bit integer decoding: significant performance improvements for wider decoders (especially for the *random* dataset) and a wide decoder that approaches the theoretical maximum of 1×10^9 records/s for 4 values per cycle at 250 MHz. The difference in performance between the 256-bit decoder and the 64-bit decoder is very large. Compared to the 64-bit decoder, the 256-bit decoder is 2.70× faster for the *delta varied* dataset and 3.50× faster for the *random* dataset. Considering the BRAM utilization of the 256-bit decoder is only 2.09× larger than that of the 64-bit decoder, there is a good trade-off when choosing a wider decoder for datasets containing deltas with wide bit-packing.



(a) Throughput in GB/s

(b) Throughput in records/s

Figure 4.14: Throughput of ParquetReaders with DeltaDecoders for 64-bit integers decoding a Parquet column containing a *delta varied* dataset as a function of Parquet page size



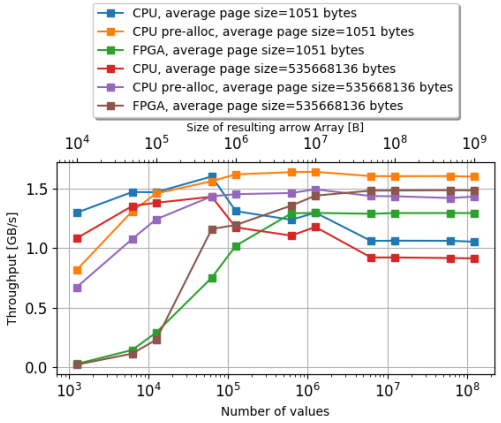
(a) Throughput in GB/s

(b) Throughput in records/s

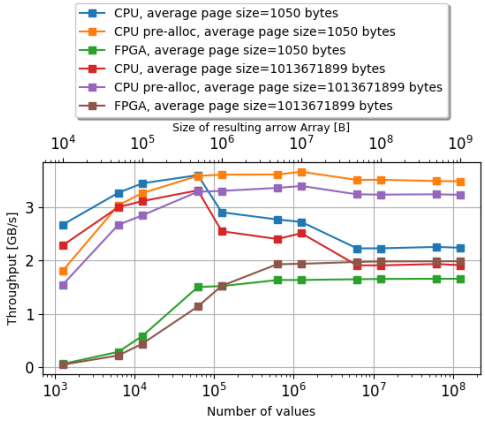
Figure 4.15: Throughput of ParquetReaders with DeltaDecoders for 64-bit integers decoding a Parquet column containing a *random* dataset as a function of Parquet page size

The FPGA performs significantly better than the CPU for 64-bit integer decoding like it did for the 32-bit integer decoding. The speedup attained by the 256-bit decoder is 2.14× for the *random* dataset and 2.79× for the *delta varied* dataset when compared to the CPU pre-allocated implementation. Notably, the 64-bit width decoder generally performs worse than the CPU implementation. Only for the *delta varied* dataset does it perform better when reading a Parquet file with a very large page size. The performance of the CPU for the *random* dataset is again better than that of the *delta varied* dataset because the Parquet file containing the *random* dataset needs

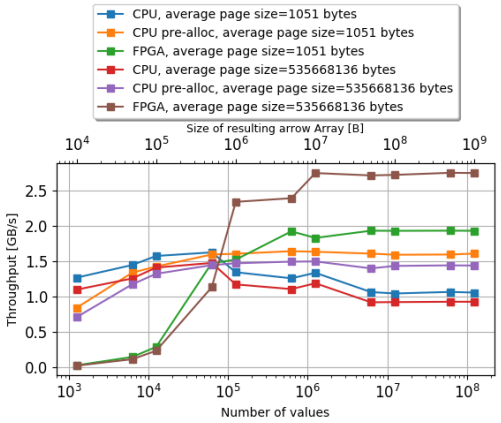
64 bits to store each delta, therefore not requiring any actual bit-unpacking. In terms of GB/s the highest throughput measured for the FPGA was 6.94 GB/s for *random* and 4.42 GB/s for *delta varied*, whereas the CPU only reached 3.38 GB/s and 1.66 GB/s respectively.



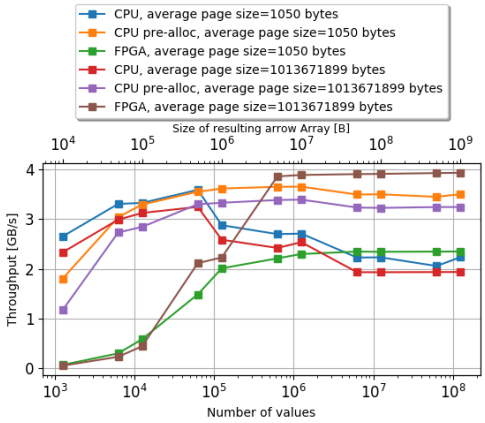
(a) Reading the *delta varied* dataset with 64-bit decoder width



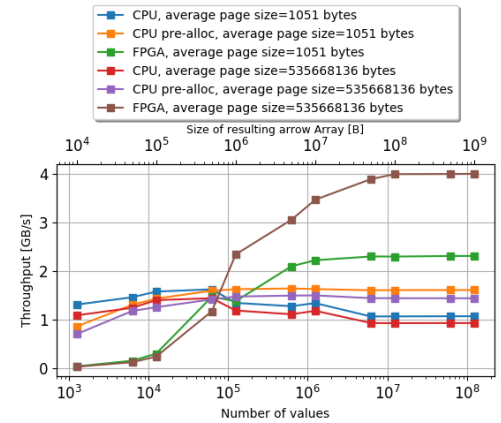
(b) Reading the *random* dataset with 64-bit decoder width



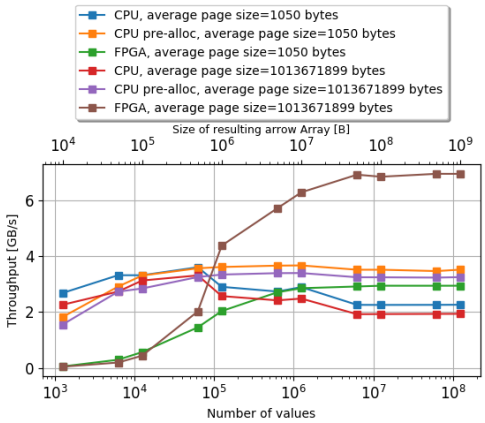
(c) Reading the *delta varied* dataset with 128-bit decoder width



(d) Reading the *random* dataset with 128-bit decoder width



(e) Throughput in GB/s for the *delta varied* dataset with with 256-bit decoder width

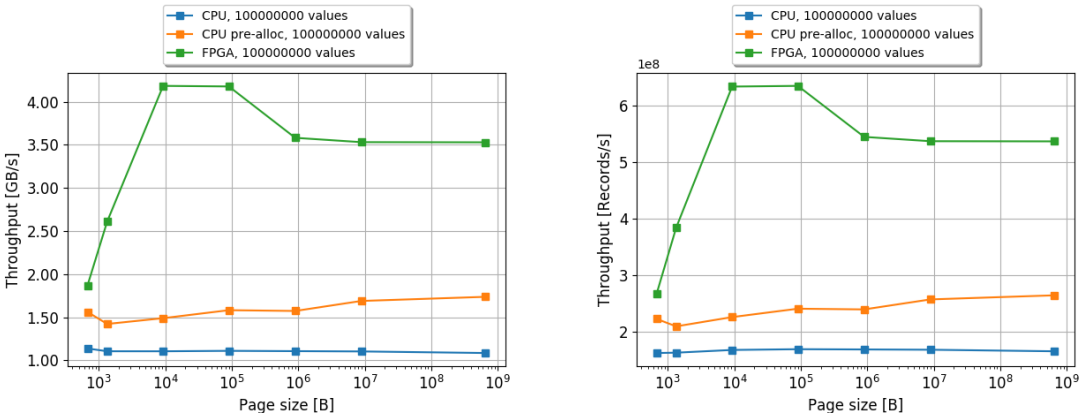


(f) Throughput in GB/s for the *random* dataset with 256-bit decoder width

Figure 4.16: Throughput of a DeltaDecoder for 64-bit integers as a function of number of values read

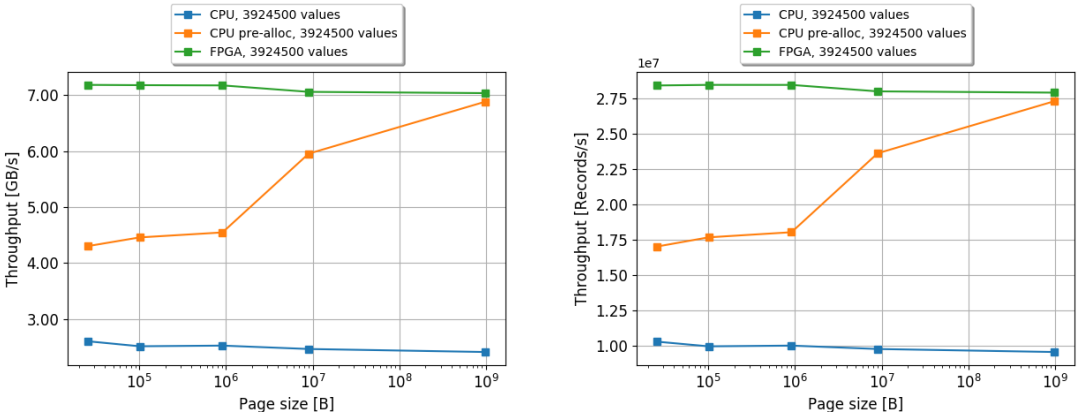
As expected, Fig. 4.16 shows the decoder for 64-bit integers is inefficient when used for reading small numbers of values. This behavior was also observed for the 32-bit integer DeltaDecoder in Fig. 4.13. The 256-bit width decoder reaches its maximum throughput between 10^6 and 10^8 values depending on the page size. The 64-bit decoder and 128-bit decoder reach their maximum throughput at smaller page sizes than the 256-bit decoder because their maximum throughput is lower.

As previous benchmarks showed that the performance of the DeltaDecoders is very dependent on the input data, it was decided to create two different datasets for testing the DeltaLengthDecoder for strings as well. The *small* dataset contains only small strings, each containing a number of characters in the range $[2, 10]$. The *large* dataset contains strings with lengths in the range $[2, 500]$. When decoding the *small* dataset the ability to read the delta encoded string lengths is important for performance because a comparatively large number of string lengths are stored in each page for all the characters. The *large* dataset has more stored characters for each string length in the page, making the ability to copy these to memory quickly an important factor in performance. Multiple Parquet files were generated with varying page sizes containing these data sets. The *small* dataset contains 100000000 strings while the *large* dataset contains 3924500 strings. These numbers were chosen so that the Arrow array after conversion is roughly 1 GB in size.



(a) Throughput in GB/s (b) Throughput in records/s

Figure 4.17: Throughput of a DeltaLengthDecoder for strings with a 128-bit width decoder as a function of page size, decoding a Parquet column containing the *small* dataset

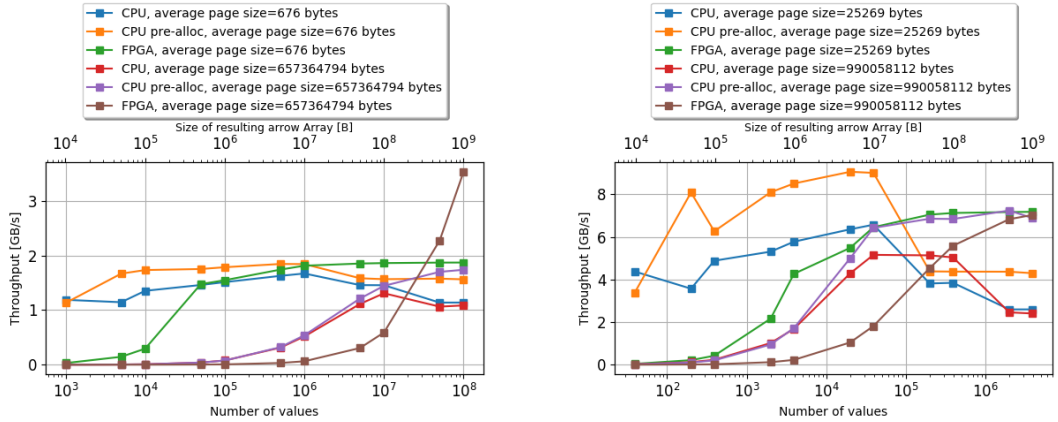


(a) Throughput in GB/s (b) Throughput in records/s

Figure 4.18: Throughput of a DeltaLengthDecoder for strings with a 128-bit width decoder as a function of page size, decoding a Parquet column containing the *large* dataset

The throughput attained for Parquet files with different page sizes containing the *small* and *large* datasets can be found in Figs. 4.17 and 4.18. The performance of the FPGA for the *small* dataset is very good when compared to the CPU (pre-allocated), attaining a speedup of 2.03× for the largest page sizes. Curiously, an even more impressive speedup of 2.81× is seen for smaller page sizes. For these page sizes a maximum throughput of 4.18 GB/s is attained by the FPGA as opposed to 1.49 GB/s by the CPU for the same page size. The decrease in performance of the FPGA for larger page sizes can be explained by the limits of the Amazon F1 instance used for testing. Large pages contain a very large number of contiguously stored characters. Once all the string lengths in the page have been processed all these (plain) characters need to be written to memory as quickly as possible. No further computations are necessary on these characters. During this copying phase the 7.2 GB/s limit for sustained reading and writing to the AOM is reached (Section 3.3), limiting the throughput of the hardware. If the pages in the file are smaller, a smaller number of characters need to be copied before the hardware needs to start processing string lengths again. The 9.3 kB pages contain roughly 8.3 kB of character data, which would form roughly 130 bus words in the hardware. This amount could realistically be buffered by the many buffers in the hardware, allowing the DeltaDecoder to immediately start processing string lengths again.

The performance of the hardware for *large* in Fig. 4.18 does show that the DeltaLengthDecoder does not shine on the Amazon F1 system when most of its time is spent copying characters. For large pages (where a lot of contiguous data is copied) the FPGA is no faster than the CPU. Memory bandwidth is now such an important factor in the Parquet string reading performance of the CPU that the behavior of the pre-allocated version resembles that seen in Fig. 3.17 where it only had to copy plain encoded integers to Arrow buffers in memory.



(a) Throughput in GB/s for the *small* dataset

(b) Throughput in GB/s for the *large* dataset

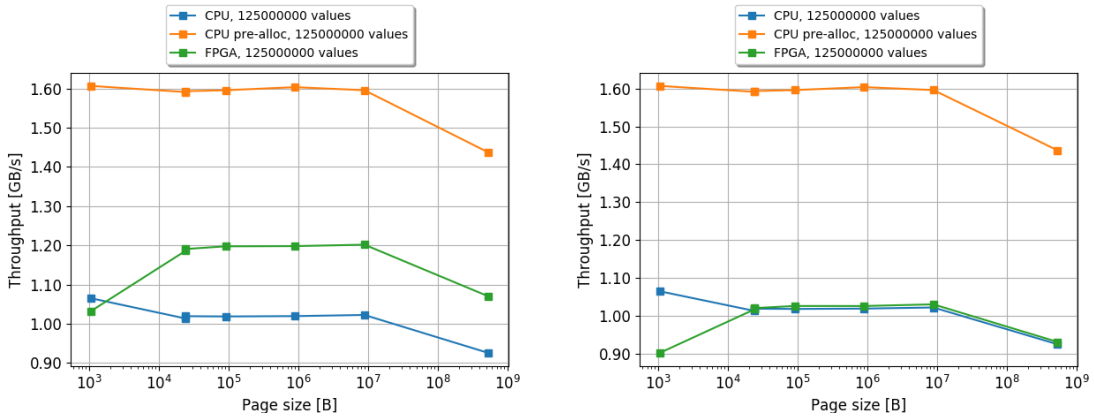
Figure 4.19: Throughput of a DeltaLengthDecoder for strings with a 128-bit width decoder as a function of number of values read

Figure 4.19 shows how the throughput of the DeltaLengthDecoder is impacted by reading fewer or more values from a Parquet column chunk. As with the 32-bit and 64-bit integer DeltaDecoders, the DeltaLengthDecoder is inefficient for reading small amounts of data. This is especially the case for large pages. The reason for this is that the decoder does not know where in the page the character data starts, it first has to read all the string lengths. Therefore, the decoder always has to process all the string lengths in a page, even if only one string is required. For large pages this causes a significant performance hit.

4.4.3 System considerations

As previously discussed in Section 3.3.3, the Amazon F1 instance used for testing is not ideal. It was shown that a naive implementation of the ParquetReader accelerators would result in bad performance as a lot of time was needed to copy Parquet data from host memory to the AOM, and Arrow data from the AOM to host memory. An example of this effect is seen in Fig. 4.20. All

measurement results shown in Section 4.4.2 avoid these issues by only measuring the processing time of the FPGA and the CPU. Because these testing system related problems were extensively discussed in Section 3.3.3, and the same conclusions hold true for the DeltaDecoders, they will not be discussed again in this section.



(a) Throughput in GB/s with the copy from AOM to host memory taken into account

(b) Throughput in GB/s with copies to and from AOM taken into account

Figure 4.20: Throughput of a DeltaDecoder for 64-bit integers with a 256-bit decoder width when copies between AOM and host memory are added to the execution time

It should however be noted that, unlike with the measurement results seen in Section 3.3.2, the limited bandwidth between the FPGA and the AOM was not always the limiting factor in performance. None of the benchmarks done for the DeltaDecoders for integers showed throughputs exceeding 7 GB/s, indicating the performance limiting factor was the 4 values per cycle limit imposed by critical path issues in the FPGA for the prefix sum in the DeltaAccumulator module. Only when a 256-bit decoder was used to decode 64-bit integers in the *random* dataset did the throughput come close to 7 GB/s. Figure 4.15b shows that the 256-bit decoder failed to reach a throughput of 9×10^8 records/s indicating it may have been limited by the bandwidth between the FPGA and the AOM.

While the effect of the low bandwidth between the AOM and the FPGA had only a limited effect on the measurements for the compute-bound DeltaDecoders, the DeltaLengthDecoder was I/O bound due to decreased importance of compute power with the characters being stored without any further encoding. Figure 4.18 showed the full memory bandwidth being reached for all page sizes, while the limited memory bandwidth is suspected to cause the decreased performance for larger page sizes in Fig. 4.17 (as discussed in the previous section).

4.5 Summary

In this chapter, the design for a DeltaDecoder and DeltaLengthDecoder to be included in a ParquetReader as described in Chapter 3 was discussed. Through the development of the DeltaDecoder, the ParquetReaders are now able to decode 32-bit and 64-bit integers encoded with the DELTA_BINARY_PACKED encoding, the standard encoding for integers in Parquet. The DeltaLengthDecoder allows for reading Parquet files containing strings, as long as they are encoded with the DELTA_LENGTH_BYTE_ARRAY encoding. The standard ParquetReader discussed in Chapter 3 did not support Parquet files containing strings for any encoding. Even though the DELTA_LENGTH_BYTE_ARRAY encoding is not the standard encoding for strings in Parquet, it still provides a significant compression ratio for strings. The development of the DeltaLengthDecoder also enables the development of a decoder for the DELTA_BYTE_ARRAY encoding in future work.

The design process for the DeltaDecoder and DeltaLengthDecoder modules was highly integrated through the development of modules that could be used for both decoders. A focus

was put on pipelining the different steps of the delta decoding process as much as possible. The limit of parallel execution in the delta decoding process in FPGA was found in the prefix sum required in the DeltaAccumulator module, limiting the number of values that can be calculated per second to 4. Because this was predicted beforehand, functionality was added to the VHDL design that allowed for easily setting this limit. The width of the decoder was also designed to be adjustable, which allows for an area and performance trade-off.

ParquetReaders including DeltaDecoders and DeltaLengthDecoders were implemented in the XCVU9P FPGA found on the Amazon F1 instances used for testing. Area utilization was shown to be very efficient with the most used resource being BRAM. The utilization of BRAM on DeltaDecoders and DeltaLengthDecoders ranged between 1.46% and 4.47% depending on type and decoder width, allowing for many delta decoding ParquetReaders to be instantiated in parallel in an FPGA. The remaining resources in the FPGA could also be used for implementing acceleration hardware operating on the decoded data.

Performance of the DeltaDecoders and DeltaLengthDecoders was shown to be very dependent on the input data and the selected decoder width. If the input data has many large deltas, a wide decoder will have a significant and positive impact on the performance of the ParquetReader. Should the nature of the data to be decoded be known before implementing a ParquetReader in the FPGA, the decoder width could be selected to provide an optimal area and performance trade-off.

Generally, the FPGA-based DeltaDecoder outperformed the CPU-based delta decoder. Because of the 4 value per cycle limit, a DeltaDecoder with a large enough decoder width has been shown to decode close to 1×10^9 records/s. The best results seen for the CPU only attained 4.25×10^8 records/s. If the DeltaDecoder had a sufficiently wide decoder, speedups between $2.14\times$ and $2.79\times$ were measured when compared to the CPU implementation.

Because DELTA_LENGTH_BYTE_ARRAY encoded Parquet column chunks contain many unencoded characters that only need to be copied to memory without any further computations, the measurements for the DeltaLengthDecoder suffered from the limited bandwidth between the AOM and the FPGA. Still, if reading a Parquet column chunk containing short strings, speedups were shown up to $2.81\times$ compared to the CPU implementation. If the DeltaLengthDecoder is used to decode a Parquet column chunk containing large strings, there is no speedup compared to the CPU. These results may be different on another system.

Chapter 5

Conclusions

5.1 Conclusion

The main research question of this thesis was formulated in Section 1.1.3 as "can FPGA-based hardware acceleration allow for better utilization of increasing I/O bandwidth when converting storage-focused data formats to in-memory data formats?". In order to answer this question the ParquetReader was designed and developed. The Parquet format was studied in order to find out which forms, variants and parts of the Parquet file format could benefit from hardware accelerated Parquet reading. This resulted in a ParquetReader design with out-of-the-box support for reading Parquet column chunks containing plain encoded floats or doubles, plain or delta encoded 32-bit integers or 64-bit integers and delta length encoded strings.

This thesis has shown that a well designed and implemented Parquet to Arrow conversion system using such FPGA-based hardware acceleration can indeed provide greater throughput than a CPU-only implementation, resulting in better utilization of the I/O bandwidth. A single ParquetReader engine in an FPGA has shown up to 2.81× better performance than a software based Parquet reader when decoding delta length encoded strings, or 2.79× when decoding delta encoded integers. An important lesson is that performance is not only heavily dependent on the encoding, page size and type of the Parquet column to be read, but also on the nature of the input data set. For the delta and delta length encodings studied in this thesis, the variance of the values stored in the Parquet column chunk significantly impacted the performance of both the FPGA-based and CPU-only implementations.

The low area footprint of the ParquetReader is one of the biggest strengths of the design. A single ParquetReader requires between 1.18% and 2.79% of LUTs, between 1.27% and 2.92% of registers and between 2.13% and 4.47% of BRAM. The resource utilization is dependent on the data type and encoding the ParquetReader is configured for. The DeltaDecoder and DeltaLengthDecoder designs have an easily configurable performance and area trade-off which can be tweaked based on the nature of the dataset the ParquetReader is expected to read, with datasets where successive values differ a lot benefiting from larger decoders.

Depending on the dataset in the Parquet column chunk, DeltaDecoders with sufficiently large decoder widths have achieved throughputs between 2.3 GB/s and 6.94 GB/s, while DeltaLengthDecoders have achieved throughputs between 4.18 GB/s and 7.2 GB/s. The throughput of these decoders is limited by either the 1×10^9 records/s limit inherent in the design at a clock frequency of 250 MHz, or the 7.2 GB/s maximum combined read and write bandwidth of the system used for testing. Especially for datasets that result in Parquet columns with narrow bit-packing widths, the throughput of a single engine may not saturate the read bandwidth of future NVMe SSDs. However, because of the small area footprint, many of these engines can be instantiated in a single FPGA, allowing for a very high total throughput.

The performed work has also shown that the integration of the hardware accelerator into the larger computing system warrants special attention. Many of the speedups and throughputs discussed in this work are based on the performance of the ParquetReader module itself. A naive implementation of a hardware accelerator containing these ParquetReader engines on the Amazon F1 instances used for testing would not provide such a performance when considered

from an end-to-end (storage to host memory) perspective. Creating efficient end-to-end system requires more than just fast hardware.

5.2 Discussion & recommendations

As discussed in Section 4.4, the biggest problem hampering the throughput of the DeltaDecoder and DeltaLengthDecoder is the limit on the number of values that can be processed in one cycle, imposed by the timing issues caused by having to perform a prefix sum. If more development time were to be spent on this issue, the performance of both decoders may improve significantly. It has been mentioned before in Section 4.4 that the use of an optimized IP core developed specifically for prefix sums might alleviate the critical path. If this would cause the limit on values per cycle to go from 4 to 8 (the next power of 2), the throughput of both decoders could double. Another angle from which to attack this problem could be in the BlockValuesAligner and BitUnpacker. Currently their logic only allows for the values per cycle limit to be a power of 2. If a more complicated component were to be developed that could deal with limits between 4 and 8, than the performance of the decoders could improve as a prefix sum of 6 (for example) could possibly be implemented on a 250 MHz clock. This more complicated logic would likely come at the cost of a larger area footprint.

Another avenue for performance improvement in the ParquetReader can be found in the DataAligner discussed in Section 3.2.2. When this module was originally designed, the intention of the ParquetReader was still to target both Parquet v1.0 and Parquet v2.0. Because Parquet v1.0 does not store the size of the repetition levels and definition levels in the page headers, the DataAligner needed the flexibility provided by the HistoryBuffer to realign the data in the pipelined shifter whenever one of its downstream modules signaled the completion of a block in the Parquet page. This flexibility comes at the cost of multiple clock cycles to realign the data every time one of the downstream modules of the DataAligner is done. Because the final design of the ParquetReader ended up only targeting Parquet v2.0 files, this flexibility is theoretically no longer needed as the sizes of the blocks are known beforehand, which means that the realignment boundaries can be predicted by the DataAligner. Especially once repetition and definition level decoders have been implemented a redesign of the DataAligner with Parquet v2.0 in mind would mitigate the performance penalty suffered when reading Parquet files with small page sizes that require frequent realignments for the page headers, repetition and definition levels, and values.

Future work may also be focused on implementing the ParquetReader hardware on different systems. The Amazon F1 instance used for testing imposed many limits on the project. The 7.2 GB/s bandwidth between FPGA and memory limited the extent to which the relationship between page size and throughput could be tested in Section 3.3, and the DeltaLengthDecoder hit this ceiling in the measurements of Section 4.4. Testing on a system with a larger bandwidth may provide new insights on the performance of the hardware. Putting more development time in integrating the ParquetReader hardware in the Amazon F1 instances may also improve the bandwidth, as the Amazon shell does support multiple DDR controllers. Other aspects of the project that could be further explored on other testing systems are the clock frequency (which is limited to 250 MHz on the interface between the Amazon shell and the custom logic), and measurements of power usage.

Finally, there are many features of Parquet that have not been implemented in the ParquetReader yet. The expandable and modular design allows for easy integration of any new decoding or decompression modules. A wrapper for an existing FPGA Snappy decompressor described in [29] has already been developed and integrated into the project. Unfortunately, some defects in the decompressor prevented successful implementation. Development of a definition level decoder is also an interesting avenue of exploration for the project, as it would enable nullable types. The development of such a decoder could also aid in the development of a repetition level decoder for support of nested types.

Bibliography

[1] Domo. (Jun. 2019). Data Never Sleeps 6.0, [Online]. Available: https://www.domo.com/solution/data-never-sleeps-6 (visited on 06/10/2019).

[2] M. Nanavati, M. Schwarzkopf, J. Wires, and A. Warfield, "Non-volatile Storage," *Queue*, vol. 13, no. 9, 20:33–20:56, Nov. 2015, ISSN: 1542-7730. DOI: 10.1145/2857274.2874238. [Online]. Available: http://doi.acm.org/10.1145/2857274.2874238.

[3] F. Kruger, "CPU Bandwidth The Worrisome 2020 Trend," Mar. 2016. [Online]. Available: https://blog.westerndial.com/cpu-bandwidth-the-worrisome-2020-trend/ (visited on 06/11/2019).

[4] U. Korn, "Efficient DataFrame Storage with Apache Parquet," Mar. 2017. [Online]. Available: https://tech.jda.com/efficient-dataframe-storage-with-apache-parquet/ (visited on 06/11/2019).

[5] W. McKinney, "Pandas: a Foundational Python Library for Data Analysis and Statistics," *Python for High Performance and Scientific Computing*, vol. 14, 2011.

[6] Apache Parquet. (Jun. 2019). Apache Parquet homepage, [Online]. Available: https://parquet.apache.org/ (visited on 06/11/2019).

[7] Apache Arrow. (Jun. 2019). Apache Arrow homepage, [Online]. Available: https://arrow.apache.org/ (visited on 06/13/2019).

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

[9] J. Le Dem, "The columnar roadmap: Apache Parquet and Apache Arrow," Strata Data Conference, Sep. 2017. [Online]. Available: https://conferences.oreilly.com/strata/strata-ny-2017/public/schedule/detail/61153.

[10] Apache Arrow. (Jun. 2019). Arrow Physical Memory Layout Specification, [Online]. Available: https://arrow.apache.org/docs/format/Layout.html (visited on 06/22/2019).

[11] L. T. J. van Leeuwen, "Fast-p2a," in. GitHub repository, 2019. [Online]. Available: https://github.com/Mythir/fast-p2a.

[12] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, "Database analytics acceleration using FPGAs," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 411–420.

[13] A. Trivedi, P. Stuedi, J. Pfefferle, A. Schuepbach, and B. Metzler, "Albis: High-Performance File Format for Big Data Systems," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA: USENIX Association, 2018, pp. 615–630, ISBN: 978-1-931971-44-7. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/trivedi.

[14] Apache Parquet, "Parquet-format," in. GitHub repository. [Online]. Available: https://github.com/apache/parquet-format (visited on 06/11/2019).

[15] Apache Parquet, "Parquet-mr," in. GitHub repository, 2019. [Online]. Available: https://github.com/apache/parquet-mr (visited on 06/11/2019).

[16] Apache Arrow, "Arrow," in. GitHub repository, 2019. [Online]. Available: https://github.com/apache/arrow.

[17] Apache Parquet, "Parquet-mr writerfactory selection," in. GitHub repository, parquet-mr fork, 2019. [Online]. Available: https://github.com/Mythir/parquet-mr/tree/select-writerfactory (visited on 08/13/2019).

[18] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive Analysis of Web-Scale Datasets," in *Proc. of the 36th Int'l Conf on Very Large Data Bases*, 2010, pp. 330–339. [Online]. Available: http://www.vldb2010.org/accept.htm.

[19] Apache Thrift, "Thrift," in. GitHub repository, 2019. [Online]. Available: https://github.com/apache/thrift.

[20] Accelerated Big Data Systems Group, "Fletcher," in. GitHub repository, 2019. [Online]. Available: https://github.com/abs-tudelft/fletcher.

[21] Accelerated Big Data Systems Group, "Vhlib," in. GitHub repository, 2019. [Online]. Available: https://github.com/abs-tudelft/vhlib.

[22] *AMBA AXI and ACE Protocol Specification*, ARM, Dec. 2017. [Online]. Available: https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications.

[23] J. Gaisler, "A structured vhdl design method," in *Fault-Tolerant Microprocessors For Space Applications*, 2011, pp. 41–50.

[24] Amazon Web Services. (Jun. 2019). Amazon EC2 F1 Instances, [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/ (visited on 06/28/2019).

[25] M. R. Pillmeier, U. Corporation, M. J. Schulte, E. George, and W. Iii, "Design Alternatives for Barrel Shifters," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 4791, Oct. 2002. DOI: 10.1117/12.452034.

[26] Amazon Web Services. (Jun. 2019). Amazon EC2 Instance Types, [Online]. Available: https://aws.amazon.com/ec2/instance-types/#instance-details (visited on 06/30/2019).

[27] Apache Parquet. (Jun. 2019). Apache Parquet documentation, [Online]. Available: https://parquet.apache.org/documentation/latest/ (visited on 07/01/2019).

[28] D. Lemire. (Mar. 2012). How fast is bit packing? [Online]. Available: https://lemire.me/blog/2012/03/06/how-fast-is-bit-packing/ (visited on 07/24/2019).

[29] J. Fang, J. Chen, Z. Al-Ars, P. Hofstee, and J. Hidders, "Work-in-Progress: A High-Bandwidth Snappy Decompressor in Reconfigurable Logic," in *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sep. 2018, pp. 1–2. DOI: 10.1109/CODESISSS.2018.8525953.