FPGA accelerated trading data compression

Jianyu Chen

FPGA accelerated trading data compression

by

Jianyu Chen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday June 26, 2020 at 13:00 PM.

Student number:	4668952	
Project duration:	September 1, 2019 – June 26, 2020	
Thesis committee:	Dr. Zaid Al-Ars,	TU Delft, supervisor
	Dr. Fabio Sebastiano,	TU Delft
	Mr. Maurice Daverveldt,	Optiver
	Prof. Dr. Peter Hofstee,	TU Delft
Thesis number:	Q&CE-CE-MS-2020-06	

TU Delft

Contents

1 Introduction

Optiver is a leading global electronic market maker in Amsterdam. It trades on multiple exchanges all over the world and its trading applications are deployed on the co-location servers in the data centers of the exchanges. At the exchanges, Optiver captures all the market data during the trading time. The amount of data is large and keeps rising. For example, the servers in an exchange in the USA captures around 3TB per day of data in Chicago currently. The traders and developers in Amsterdam need this data to optimize strategies, run trading analysis. Therefore, the data produced in the co-locations needs to be sent back to Amsterdam.

1.1. Context of the project

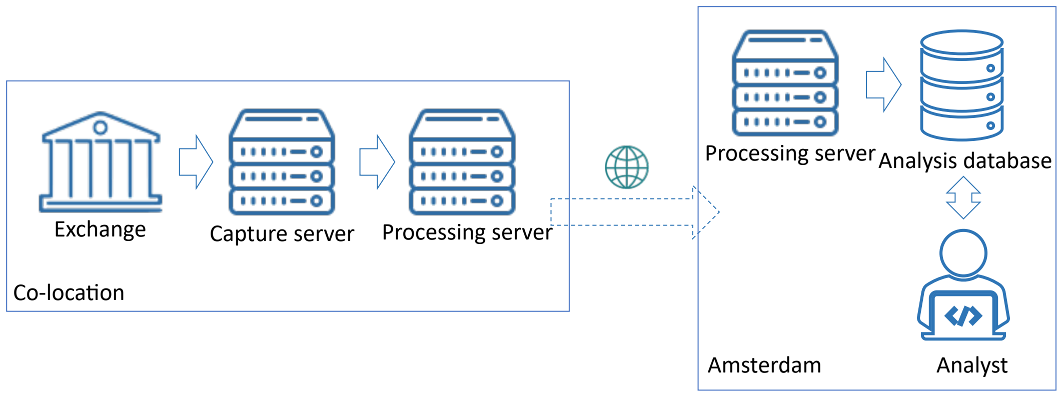


Figure 1.1: Structure of the data transmission system

The structure of the current system is presented in Fig 1.1. In the current system, the data from the exchanges will first go into capture server in co-location, then a processing server will fetch data from the capture server and send it to another processing server in Amsterdam. Finally the data will be stored in the analysis database for people to do analysis.

The bottleneck of the current system is in the bandwidth between co-location and Amsterdam. The data to be transferred is packages that are fetched from several routers and the packages are duplicated for several times. Inside each package, not all of the information needs to be sent back. Now the data is first deduplicated, then only part of the information is extracted and transferred back to Amsterdam when the data is generated and some other parts of the data sent back overnight after trading. This method reduces the amount of data by one order of magnitude. However, this solution cannot fully satisfy the need of Optiver, people in Optiver want to have all the data in Amsterdam immediately once it is captured.

We can solve this problem by simply buying more international lines and transferring all the raw data. However, a reliable international Ethernet line is very expensive. On the other hand, the volume of existing lines is limited so that we cannot always buy more bandwidth. Especially for some exchanges on other continents, sending all the data through the Atlantic directly is generally not feasible. Furthermore, since the volume of data received from exchanges keeps rising, the current solution can hardly follow the growth.

To solve this problem, we proposed a new method: compress the data before sending, and do the decompression in Amsterdam when needed. However, the processing system needs to do some other necessary tasks with higher priority, and thus does not have enough redundant resources to compress data. In this case, we want to offload the data compression tasks to an external device.

1.2. Problem definition and research questions

There are multiple solutions for external compression devices. The most trivial way to do the data compression is to have another server running software compression algorithms. In this scenario, the processing server is connected to another specific compression server via Ethernet cables.

However, the rent of a slot to deploy a server in co-location is very expensive. In addition, having an extra server also increases the maintenance burden. Therefore, we come up with another idea to use Field Programmable Gate Array (FPGA) as an accelerator.

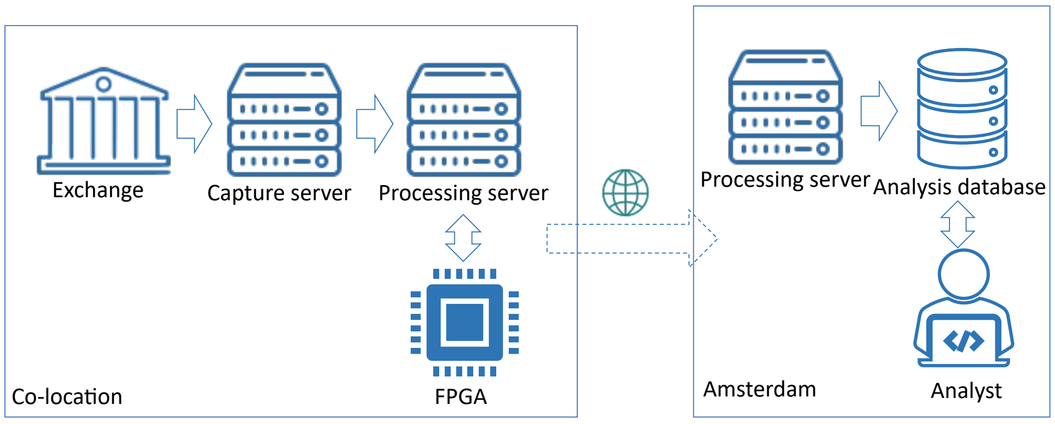


Figure 1.2: Structure of the data transmission system with FPGA

Nowadays, using FPGAs to accelerate compute intensive tasks is becoming increasingly popular in both academia [21, 9] and industry [12, 10, 11]. For example, [6] shows that FPGAs are able to do fast data decompression, [2] introduces a new method to do matrix multiplication for Posit numbers. In order to reduce the difficulty of accelerated kernel design, [20, 22] proposes a framework which enables user-defined kernels to read the data from a standard data format called Apache Arrow easily. In this project, the architecture with an FPGA accelerator is presented in Fig 1.2, which shows that the processing server sends data to the FPGA via peripheral component interconnect express (PCIe), and the FPGA returns the compressed data. The processing server will format the compressed data and will send it to Amsterdam. In Amsterdam, the data will be stored in the analysis database and be decompressed by software when needed.

At this moment, developers in Optiver do not have an existing FPGA compression system and do not know what performance the system can achieve. The goal of this thesis project is to explore the performance of doing trading data compression on FPGAs, which communicates with the process server through PCIe. The throughput needs to be up to 10GB/s to keep up with the speed of capture. The extra latency brought by the FPGA should be lower than one second. We want to make the compression ratio as low as possible to reduce the requirements of bandwidth from the co-location to Amsterdam. Scalability is also an important factor. Once the amount of data becomes too large in the future, there should be a simple way to scale up the capacity of data compression system. In addition, we need to take into account the single-file compression speed, since sometimes we do not have multiple files to compress in parallel.

The main research questions of this project are listed below:

- Which compression algorithm is most suitable for our use case? We will test and compare the compression ratio and throughput of commonly used compression algorithms for trading data.
- How can we implement an FPGA solution to accelerate the compression algorithm on hardware? We will explore how to fit the compression algorithm into FPGA to achieve a high throughput and low compression ratio for trading data.

Compression ratio is an important metric for a compression algorithm. This concept has a couple of definitions of the literature. In some documents researchers define it as *compression ratio = compressed size*

/ original size , while in other documents *compression ratio = original size / compressed size*. In this report we use the former definition.

In this project, we need to design a solution to compress data from three different exchanges. In this report, we refer to these exchanges as: exchange-1, exchange-2, and exchange-3, while the data from these 3 exchanges are referred to as: data-1, data-2 and data-3 in this report, respectively. The exchange-1 is located in another continent, and the data transmission is expensive and difficult. Therefore the main focus of this project is on the data from exchange-1, namely data-1.

1.3. Thesis outline

In Chapter 2, we first describe the format of trading files, which represents the format of the file we need to compress. Then we compare the performance and format of different compression algorithms. Then we have a more detailed introduction of Zstandard (Zstd) and Vitis, which are the compression algorithms we chose to implement on FPGA and the hardware development platform we plan to use, respectively.

In Chapter 3, we analyze the official software implementation of Zstd. The first part is the introduction of several algorithms used in Zstd. Then we demonstrate the performance of Zstd on trading data, including throughput and compression ratio. Finally, we explain the features of trading data and analyze the reasons that make Zstd suitable for trading data.

Chapter 4 is the introduction of the hardware architecture we designed and how we do the its verification. We start with the overall architecture and dive into the implementation detail of each module.

Chapter 5 demonstrates the result of the compressor, including single-kernel throughput, multi-kernel throughput, compression ratio, resource utilization and latency.

In Chapter 6, we draw the conclusion for this project, and introduce some possible directions for improvement.

2 Background

In this chapter, we begin with an introduction of the targeted files that need to be compressed in this project. Then we describe the details of some commonly used compression formats and algorithms. The final part is the detail of the hardware, the development platform, and the development tools.

2.1. Input files and Extensible Record Format (ERF) packages

In Optiver, the data fetched from the exchanges are Ethernet packets containing User Datagram Protocol (UDP) packets, Transmission Control Protocol (TCP) packets, and some other formats. The fetched data contains the trading information from the exchanges, such as the update of prices or bills. These packets are stored in the Extensible Record Format (ERF) format with some extra information, for example, the timestamps, which indicates when the packets are received. The ERF files can be opened and analyzed by an open-source software called Wireshark [17].

Normally, each packet in the ERF files is duplicated several times, because the capture server reads the same data from different routers. The data inside the packages, like IP addresses, price, volumes, are also usually the same or similar. Therefore, it should be possible to compress these ERF files to a small size.

Here we use the format of packets from exchange-1 as an example. The timestamp is an 8-byte unsigned integer format indicating the time when the packet is received. Generally the data is stored in the sequence of reception. Therefore, the most significant bytes of the timestamp of adjacent packets are normally the same. Source and destination are the IP address of the sender and receiver of a packet, respectively. Both of these are 4-byte numbers. From our observations, most of the packets are from 6 specific IP addresses and are heading to the same 6 addresses [14].

Apart from the duplication of items across the packets, there is also a packet-level duplication. The data received from the exchanges is in the form of UDP packets. Thus it is possible for some packets to be lost. The solution is that we connect to multiple routers at the same time and receive the same packets multiple times. In this scenario, most of packets in the data are duplicated.

From our perspective, there should be a way to compress the data-1 to a smaller file by replacing the repeated bytes with copy tokens. So our research for compression algorithm is mainly to this direction.

2.2. LZ77 compression

LZ77 compression algorithm

LZ77 is a lossless data compression format proposed by Abraham Lempel and Jacob Ziv in 1977 [30]. LZ77 forms the basis of many other compression algorithms like LZSS [19]. The main idea behind LZ77 is to use the previous data as a dictionary to find repeated occurrences of data in a file [30]. For example, the string "ABCDEFGABCD1234" can be re-written in this format: "ABCDEFG(7,4)1234". The first number in the parentheses is the offset, which represents the number of bytes we need to trace back when doing decompression. The second number means the number of bytes we need to copy from the previous data. Therefore, (7,4) means copy 4 bytes from 7 bytes ago, which is literally "ABCD".

Due to the limit on the size of memory, it is inefficient to buffer all the history during both compression and decompression for large files, thus LZ77 uses sliding windows strategy. Under this strategy, the algorithm

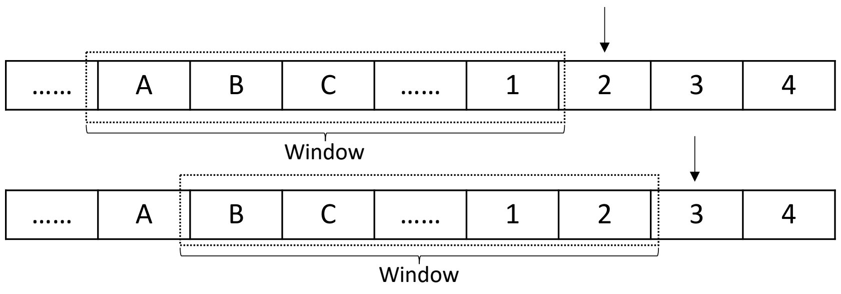


Figure 2.1: Sliding window

will only try to find a match in a window with a fixed size, and this window is sliding during compression [30]. For example, as we can see in Fig 2.1, when we look for a match for the string starting from the character "1", we only search in the window from "A" to the last character "1". When we switch to process the next character, the window also slides for 1 byte.

In LZ77 format, the size of windows is fixed in each file and can be configured by users. When the size of the window increases, the compressed result will be smaller at the cost of more memory usage in both compression and decompression [30].

Huffman coding and Finish state (FSE) entropy coding

Symbol	Frequency	Code
space	7	111
e	4	000
a	4	010
f	3	1101
n	2	0010
t	2	0110
m	2	0111
i	2	1000
h	2	1010
s	2	1011
o	1	00110
u	1	00111
x	1	10010
p	1	10011
r	1	11000
l	1	11001

Figure 2.2: An example of Huffman table

Huffman coding is a form of lossless compression to replace a symbol, for example a byte, to a prefix code [13]. Prefix code is a coding system in which the lengths of codes can be different. An important property is that no code is the prefix of another code. With this property, we can decode the prefix code from beginning to end without without extra information to indicate the starting point of each code. The Huffman coding algorithm is developed by David A. Huffman in 1952. First, the algorithm uses statistics of the occurrence of a symbol and generates a probability table of all symbols. Then the probability is transformed into a prefix code table, in which each symbol is related to a unique code. Finally, each symbol will be replaced by the corresponding prefix code. Fig 2.2 shows an example of Huffman table, each symbol in this table is a character. As you can see from this table, the symbol with a higher frequency is encoded with less bytes and vice versa [13].

However, Huffman coding has a limit: the number of bits in each prefix code can only be an integer number [28]. For example, we cannot encode a symbol to 3.3 bits. In practice, the theoretically optimal length

Magic number	Frame header	Data block 1	Data block 2-n (Optional)	Content checksum
4 Bytes	2-14 Bytes	>1 Bytes	>1 Bytes	0-4 Bytes

Table 2.1: Structure of a frame

Frame header descriptor	Windows descriptor (optional)	Dictionary ID (optional)	Frame content size (optional)
1 Byte	0-1 Byte	0-4 Byte	0-8 Bytes

Table 2.2: Structure of frame header

of prefix code is normally a fraction. To approach the optimal compression ratio, researchers proposed the idea of asymmetric numeral systems (ANS). ANS is a combination of arithmetic coding and Huffman coding. It is normally implemented by a finite state machine, such as finite state entropy (FSE) coding proposed by Facebook [4].

Deflate compression algorithms

Deflate originally means a lossless compression format which consists of two stages: LZSS and Huffman coding [18]. It is invented by Phil Katz in 1996. Later on, the definition of Deflate was extended to all the lossless compression formats that use a similar pattern [27].

The most widely used format in the Deflate family is Gzip, which is used in Linux. Gzip makes a balance between the compression ratio and throughput. However, since it is invented decades ago, it is not optimized for modern processors [24]. Snappy is invented and used within Google as a high-throughput compression algorithm. The original file is first split into 64KB blocks except for the last block. Each block is compressed independently, which means it is not allowed to find a match across the boundary of blocks. In this scenario, the length of the dictionary cannot be larger than the size of a block, which is 64KB [23]. Instead of generating a Huffman table dynamically, Snappy uses a static pre-trained table. This method saves time for compression and saves extra space to store the table.

Another popular compression algorithm is Zstandard (Zstd), which targets real-time compression with a decent compression ratio. To achieve a better compression ratio, Zstd uses Huffman coding and Finite state entropy (FSE) coding together. In Zstd, there are a lot of parameters for configuration, for example, the size of the dictionary, the size of the Huffman table and the size of the FSE table. These configurations provide large room to make a trade-off among memory usage, compression throughput, and compression ratio.

Format of the Zstd algorithm

Like common Deflate compression formats, Zstd consists of two stages, such that these the two stages can be done independently. The first stage is the LZSS compression, which is the same as Gzip. The compression result of the first stage is two streams of data: literal and sequence. Literal is the data that cannot be matched in the history, while a sequence is a set of data consisting of a matching result. The matching result is represented using three numbers: offset of the match, length of the match and the number of literal bytes in front of the match. The literal streaming is compressed by Huffman coding and sequence FSE coding. However, the compression ratio of Huffman coding highly depends on the distribution of symbols, which means the size of compressed data is not always smaller than the original. Therefore, the Zstd compression software will first try to compress the literal data in a block. Then check whether the compressed streaming is smaller. If not, the literal bytes will be stored directly instead of Huffman streaming.

A Zstd file consists of one or more frames, each frame can be decompressed without any information from other frames. The original file should be the concatenated result of the decompressed result of all frames. Within each frame, there are one or more blocks. Each block is compressed separately in the second stage, and the decompressed size of each block cannot be larger than 128KB.

Zstd also supports some special mode, for example skippable frame to store user-defined metadata, and dictionary mode to compress small-size of data. These special modes are not taken into account in this project. Each Zstd always started with a 4-byte magic number, which is only used to indicate the starting of a valid compressed frame.

The structure of a frame is shown in Tab 2.1. Each frame is started with a magic number, which is a four-byte number 0xFD2FB528 in little-endian. The frame header is the metadata of a frame, which contains some

Bit number	Field name
7-6	Frame content size flag
5	Single segment flag
4	Unused bit
3	Reserved bit
2	Content checksum flag
1-0	Dictionary ID flag

Table 2.3: Content of the frame descriptor

Bit number	Field name
7-3	Exponent
2-0	Mantissa

Table 2.4: Content of the window descriptor

information for decompression. The content of the frame header is presented in Tab 2.2. The header consists of four part: *frame header descriptor*, *window descriptor*, *dictionary ID*, and *frame content size*. The first byte, which is the *frame header descriptor*, indicates the size of the next 3 fields.

The bit 7 and bit 6 indicates the size of *frame content size*. Its meaning is presented in Tab 2.5. As we can see, the field size can be 0 or 1 when the flag value is 0.

The Tab 2.3 lists all fields of *frame header descriptor*. The bit 7-6 indicates the size of *frame content size*. The second field is *window descriptor*, which indicates the minimum size of buffered data during the decompression. As we can see from Tab 2.4, the highest 5 bits and lowest 3 bits are called exponent and mantissa, respectively. The size of the required buffer can be calculated by the formula below:

$$windows\ size\ (Byte) = 1 << (10 + exponent) + (1 << (10 + exponent))/8 * mantissa.$$

From the formula above, we can deduce that the minimum size of the buffer is 1KB and the maximum size is about 3.75TB.

Dictionary ID is the size of dictionary in the frame. The relation of dictionary ID flag value and size of dictionary is presented in Tab 2.6. Dictionary compression is not used in this project, therefore we do not give further discussion about it.

Single segment flag presents whether the compression uses all the previous data as windows. If this flag is set to high, there is no *window descriptor* because the size of the window is the same as the size of the original file.

Unused bit and *reserved bit* are not used in the official zstd format. The *reserved bit* can be used as user defined flag while the *unused bit* must always be set to 0.

During the first stage, all the data in this frame is compressed from beginning to the end, thus generate two streams of data: literal and sequence. Then in the second stage, the literal stream and sequence stream are sliced into one or more blocks and compressed by Huffman and FSE, respectively.

In the second stage of Zstd, the literal bytes and sequences in each block are compressed from the end to beginning, which means the last literal byte and the last sequence are compressed first. In this scenario, the decompression starts from the end of streaming.

Algorithms to find the match used in Zstd

Most of the innovation in Deflate is in how to find a good match in the first stage. The most trivial solution is to iterate every place in the history to find all the possible matches, but this method takes too much time,

Flag value	Field size
0	0 or 1
1	2
2	4
3	8

Table 2.5: Size of frame content size

Flag value	Field size
0	0
1	1
2	2
3	4

Table 2.6: Size of dictionary size

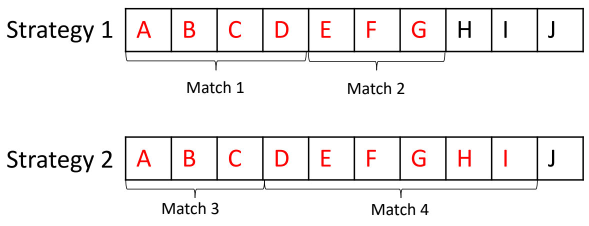


Figure 2.3: Longer match is not always better

thus unrealistic. On the other hand, how to determine the optimal combination of matches when we have several matches is also difficult. The longest match can be regarded as local optimal in most of case, but usually not the global optimal one. For example, if we look for matches for string “ABCDEFGHIJ” and find four matches: “ABC”, “ABCD”, ”EFG” and “EFGHI”. As we can see in Fig 2.3, we have two strategies to decide the combination of matches, and strategy 1 is always use longer match. Since the compression goes from left to right, the strategy 1 should obviously choose “ABCD” rather than “ABC”. However, the total length of the matched string for strategy 1 is even shorter. If we want to find the global optimal combination of matches for a file, the only way is to compare every possible combination in all of the file, which is an NP-hard problem.

Therefore, Zstd uses several heuristic methods to find matches and decide the combination of matches. The simplest and fastest way is a using a single hash table. The algorithms are introduced and analyzed in Chapter 3.

2.3. FPGA acceleration

In this section, we will first introduce the FPGA board used in this project: Alveo U200. After that, we present the detail of the Vitis platform, which can greatly reduce the development time on U200. Finally, we list some other possible solutions which are considered by the developers in the company.

Alveo U200 FPGA board

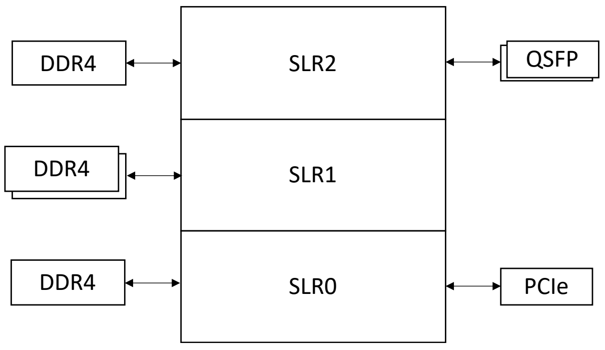


Figure 2.4: Structure of Alveo U200

Alveo U200 is a data center accelerated card with PCI Express Gen3x16. It is designed as a computation acceleration card attached to computer. The chip on the board is Xilinx XCU200 FPGA, which is specific for this board. The chip comprises of 3 subsections called super logic region (SLR), but the 3 SLRs are not the same in the topology. As you can see in the Fig 2.4, the SLR1 have access to 2 DDR4 banks while other 2 SLR only one. The SLR0 has unique access to PCIe and only SLR2 can access Quad Small Form-factor Pluggable

(QSFP) directly. On the FPGA, the logic can be placed across SLRs [29].

On the Alveo U200 board, there are three kinds of memory: Block RAM, Ultra RAM, and DDR4. Block RAM and Ultra RAM are static RAMs dedicated within the FPGA chips. On the XCU200, the storage of a BRAM and URAM is 9 kbit and 36 kbit, respectively, and read/write bus width of both kinds of RAM are 36 bits. Compared with URAM, BRAM is faster and more flexible. Each BRAM can be configured as two half-BRAM with half of the storage, and each half-BRAM can work independently with independent input and output ports. Also the BRAM can be initialized by user-defined data, while the data in the URAM are always zero when the chips is powered on. The reading and writing latency of URAM and BRAM are deterministic and can be 1 clock cycle. However, the output of the RAMs are usually buffered for one or more cycles to get a high working frequency.

DDR4 is a type of synchronous dynamic random-access memory released in 2014. Compared with its predecessor, DDR3, DDR4 allows each DIMM to contain up to 64GB storage. The DDR4 memory used on the board is manufactured by Micro. The capacity of each DIMM is 16GB with ECC error detection and correction [26]. The reading and writing latency of DDR memory is not deterministic thus we cannot expect the outcome within a certain time.

Vitis acceleration platform

Vitis unified software platform is a new tool introduced by Xilinx in 2019. This tool combined several previous framework and tools like Vivado high level synthesis(HLS), SDAceel and Xilinx Software Development Kit (SDK) [29]. Vitis provides an acceleration platform to connect X86 machine heterogenous computing like FPGA, Zynq SDSoC and Versal ACAPs by PCI-E. The targeted devices of Vitis currently contains Alveo data center accelerator cards, including Alveo 50, Alveo200 and Alveo250 [29].

The Vitis platform consists of DMA and some AXI control logic on the FPGA and the software OpenCL libraries. Users can design of computation kernels and connect them to the platform and host software based on the OpenCL library, which largely simplify the development. The user kernel can be generated by HLS or coded in RTL language directly, and the kernels communicate with the platform through AXI interfaces. All the hardware details of DDR memory and PCI-E are hidden behind the interfaces [29].

Currently, the Vitis platform supports several cards, including Alveo U200. When using U200 board on Vitis platform, part of the FPGA becomes unreconfigurable for users, we call it static area.

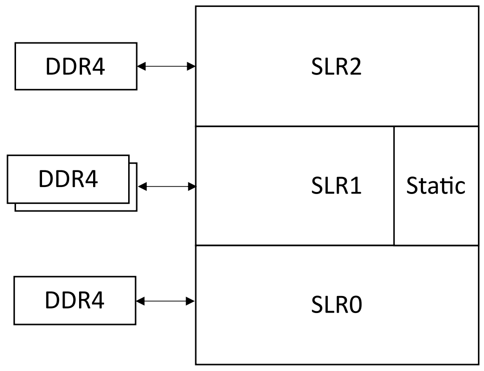


Figure 2.5: Structure of U200 board on Vitis platform [29]

Fig 2.5 shows the hardware architecture of the system, the FPGA on the Alveo U200 board contains a static region in SLR1. The static region is used to implement PCIe controller, Direct Memory Access (DMA), clock/reset module and some others card management modules. The logic in the static region is provided by Xilinx and cannot be changed if using the Vitis platform. The resources in the dynamic region are used for user-designed kernels and some AXI control modules generated by Vitis. As you can see from Fig 2.5, the host machine cannot access the on-board memory directly, but via the static region of FPGA only [29].

On the hardware of the platform, user-defined logic cannot control PCIe, but can read and write data from and to the DDR4 memory. The QSFPs are not supported in the current platform.

Acceleration model of Vitis

Xilinx package the hardware architecture into a simpler acceleration model, which is presented in Fig 2.6. In this module, the host software can communicate with the global memory, which is essentially the on-board DDR memory, directly by calling OpenCL functions. The software can also pass some variables to the user

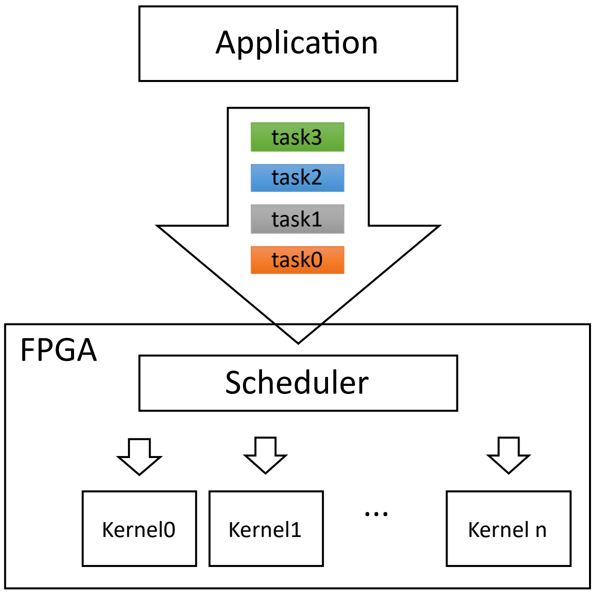


Figure 2.6: Vitis acceleration platform [29]

kernels. The variables can be integers, floats, doubles, pointers or some other information. But this kind of variable passing is unidirectional, the user kernels can only send a finish signal to the software to indicate the end of computation. The other data can only be stored in the global memory for the software to fetch [29].

The user kernels exchange data with the global memory through AXI interfaces, the number of AXI interfaces and the data width of each interface can be configured by users. The data width of the Xilinx DDR controller is 512-bit, which means only AXI interfaces of 512-bit can fully use the throughput of the DDR bank. Therefore, all the AXI interfaces with different data widths will be transformed into 512-bit by Vitis at the price of extra resource utilization [29].

Profiling and debugging tools of Vitis

Debugging is always a time-consuming work on hardware development, especially when it involves the communication between host software and hardware. Developers usually need to build a testbench to simulate the behaviors of software themselves, and this kind of simulation are usually quite different from the real one. To solve this problem, Vitis provides a function called hardware emulation. By using this function, developers can run the host software and hardware simulation together as if on the real hardware and get the waveforms inside the kernels.

Due to the indeterministic behavior of the DDR memory, the real behavior of the AXI interface is thus hard to simulate. Therefore, some bugs may not be triggered on the simulation and will thus make the result on hardware wrong or even make the machine down.

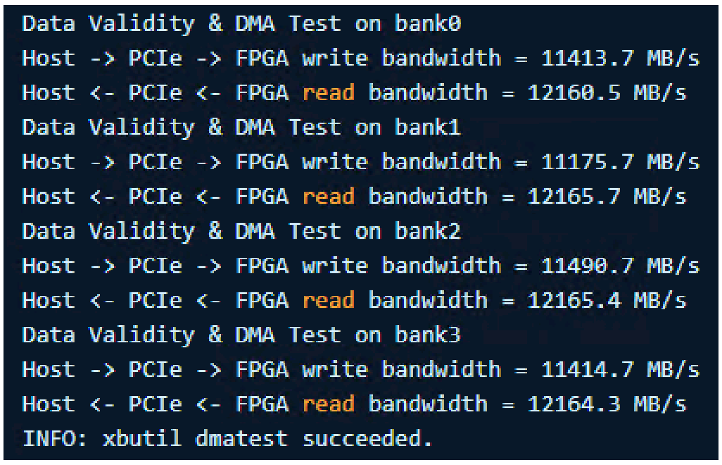
Profiling is normally necessary when developers want to get a better performance and an efficient scheduling of data I/O can reduce a lot of running time. Xilinx provides Vitis Analyzer tool to help us on it. On Vitis Analyzer, it shows the stating time and ending time for each OpenCL function and data transfer operation.

Benchmark of Alveo U200 using the validation tool

In theory, the throughput of each lane in PCIe Gen3 can be 1GB/s, thus PCIe Gen3x16 can read or write data at 16GB/s [1]. But this number can never be achieved in practice. The easiest way to get the real throughput is to test it on the real machine. Xilinx provides a validation tool for the Alveo serial cards to do the throughput profiling, the running result is presented in Fig 2.7. As you can see, the reading speed and writing speed for all four DDR banks can be up to 12GB/s and 11GB/s respectively.

Packet from Ethernet

Most of Ethernet data received from exchanges are in UDP (User Datagram Protocol) and TCP (Transmission Control Protocol) format. The purpose of TCP packets is to enable a computer to communicate with the internet and guarantees that the data is delivered successfully. UDP is also for communication between



```
Data Validity & DMA Test on bank0
Host -> PCIe -> FPGA write bandwidth = 11413.7 MB/s
Host <- PCIe <- FPGA read bandwidth = 12160.5 MB/s
Data Validity & DMA Test on bank1
Host -> PCIe -> FPGA write bandwidth = 11175.7 MB/s
Host <- PCIe <- FPGA read bandwidth = 12165.7 MB/s
Data Validity & DMA Test on bank2
Host -> PCIe -> FPGA write bandwidth = 11490.7 MB/s
Host <- PCIe <- FPGA read bandwidth = 12165.4 MB/s
Data Validity & DMA Test on bank3
Host -> PCIe -> FPGA write bandwidth = 11414.7 MB/s
Host <- PCIe <- FPGA read bandwidth = 12164.3 MB/s
INFO: xbutil dmatest succeeded.
```

Figure 2.7: Result of throughput benchmark

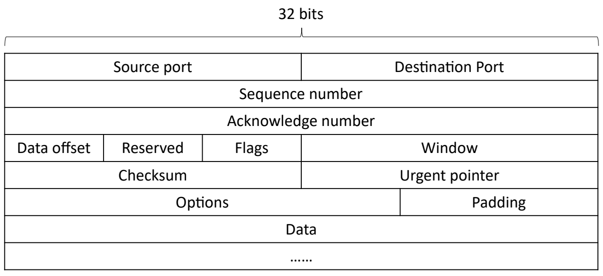


Figure 2.8: Format of TCP packet

computer and internet while does not do error-checking and does not guarantee reception.

Fig 2.8 shows the format of TCP packet. For the packets fetched from the same exchange, several fields will be similar or the same like the sequence number. The content of data section is defined by the exchange

There are already some existing hardware solutions for hardware compression. For example, Xilinx provides open-source library for data compression and decompression for Zlib and Snappy. The compression speed for a single engine is only about 300MB/s, and the compression algorithm cannot achieve a good compression ratio.

Microsoft and IBM also have their own hardware compression architecture and implementation for Zlib and Gzip. Their main concern is to have high throughput and low compression ratio, thus consume more resource. On the other hand, all these solutions target the compression for general usage, not specific to the trading data from exchanges [7].

From our research, Xilinx provides Gzip hardware IP core for commercial usage, which achieves a maximum 1.8 GB/s throughput for single kernel at the cost of a lower compression ratio of only 1.7. Xilinx also proposed a general LZ-family compression framework with a comparable compression ratio to software and only 250MB/s throughput. The performance of this architecture is too low for us and it also requires some preprocessing and package work on CPU, which increases the workload of process servers. And from the previous research inside the company, Gzip is not an ideal solution due to its low compression ratio and low software compression speed.

What is more, Gzip is developed for general data compression and provide a small range for users to make tradeoff among different metrics, but this project only focuses on high frequency trading data, which may show different features than the normal one.

2.4. Related work

Before this project, there are already hardware implementations for data compression. [25] proposed a method to generate Huffman tree on FPGA. With this method, we can compute the Huffman tables on the chip and compress.

[5] introduced a new algorithm for LZW compression algorithm and the corresponding FPGA architec-

ture. It is a light-weight compression kernel, which achieves a high compression ratio and low throughput. Its compression input throughput is only about 200MB/s. The main advantage is that the resource utilization is comparably small. We believe that the main reason is that it is designed more than a decade ago, and the resource of FPGA was much less.

Microsoft also designed their own hardware implementation of compression, which is called Project Zipline [16]. This project is implemented and deployed on Microsoft Azure for data center acceleration. They designed their own compression algorithm aiming to achieve a balance between compression ratio and throughput.

3
Analysis of Zstd software

3.1. Comparison between Zstd, Snappy and Gzip

Before the analysis of compression algorithms, we first need to explore the performance of existing software compression solutions. Zstd, Snappy and Gzip are commonly-used compression format in database. In this project, we compress the trading data into all three compression formats and see the difference in compression ratio and throughput.

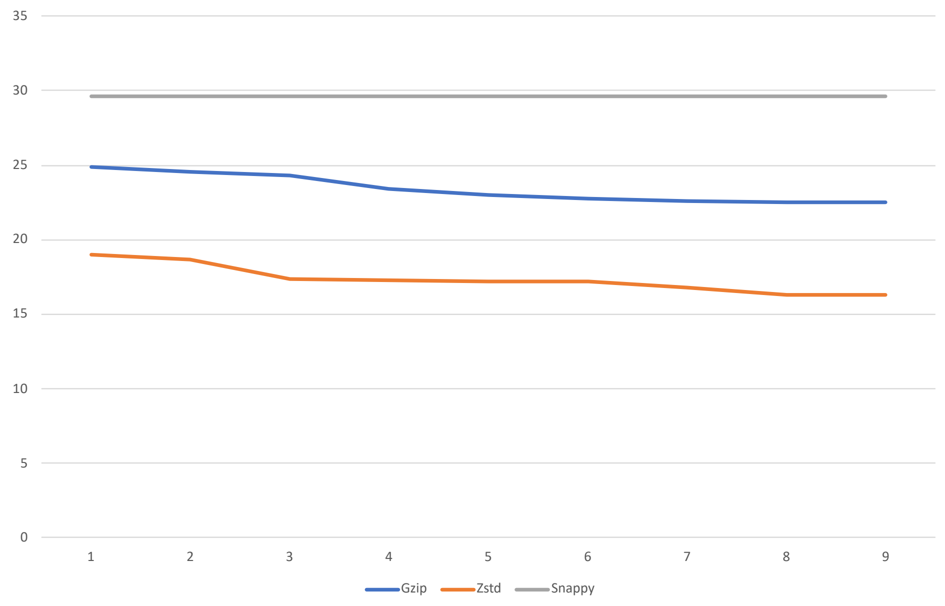


Figure 3.1: Compression ratio(%) VS compression level

We download the official implementation of Zstd [3] and Snappy [8] and install them on CentOS. The Gzip implementation used is the generic one in the CentOS. The benchmark is 200MB of data-1. The result is presented in Fig 3.1. In Zstd and Gzip, we can set the compression level when start compression. The higher the compression level is, the smaller the compressed file is. Even though the compression level can higher than 20, we only test the level from 1 to 9, since the higher levels are too slow to be used in practice. The official Snappy implementation cannot be configured, therefore the compression ratios presented are the same. Even if we compare the level-1 Zstd with Snappy and level-9 Gzip, we still see that the compression level of Zstd is the best one. The compression ratio of Snappy is the highest one, since it is designed for high-throughput compression and decompression [8].

We also profile the compression throughputs of Gzip, Zstd and Snappy under different compression levels.

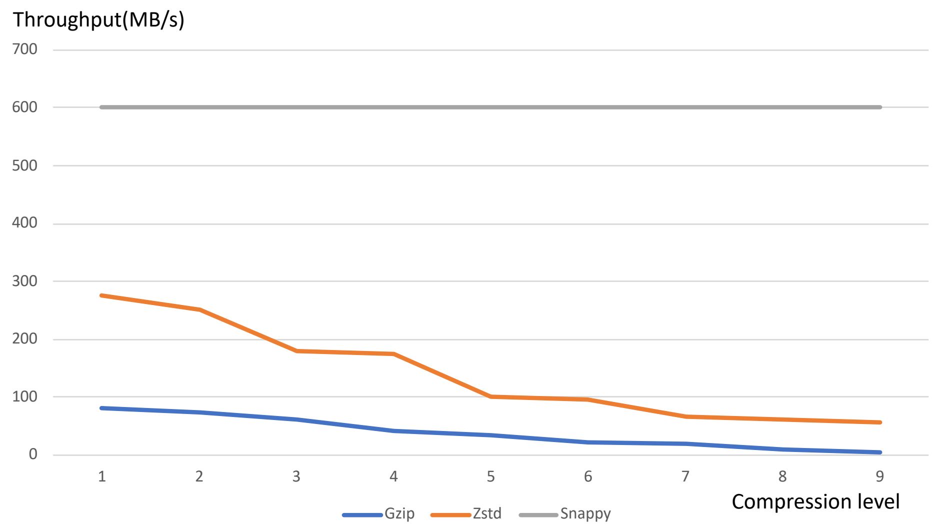


Figure 3.2: Throughput of software VS compression level

The test is done on HP-G9 machine. The CPU used in the machine is Intel Xeon E5-2697 and the size of memory of the machine is 32GB.

As presented in the Fig 3.2, the throughput of Zstd is higher than Gzip in every compression level. Therefore, we conclude that Zstd is better than Gzip in both compression level and compression throughput. The throughput of Snappy is about 600MB/s, which is about 3 times as large as level-3 Zstd.

Even though the compression of Snappy is much faster than Zstd, we decide to use Zstd format as our solution since compression level is also taken into account.

3.2. Discussion of different software compression algorithm

As we mentioned above, the Zstd compression software proposed several new ideas to find the repeated strings in the first stage. The simplest way is to use a single hash table.

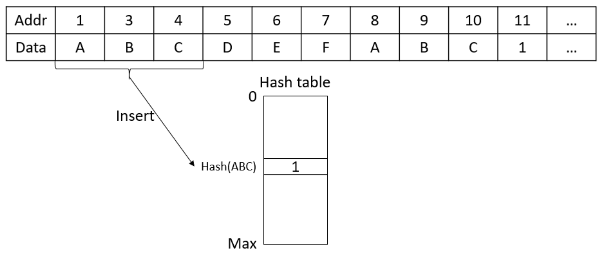


Figure 3.3: Insert data to hash table

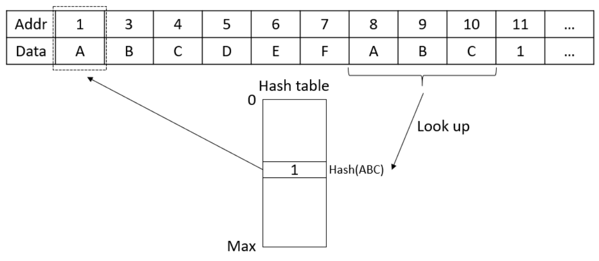


Figure 3.4: Look up the hash table to find repeated string

For every byte of data, we can use this byte and the next 2 bytes to calculate a hash value. Then we insert the address of the byte into the hash table, and the address of the hash table is the hash value. Fig 3.3 is an example of inserting. The address of the first character "A", which is 1, is inserted into the table in the address hash(ABC). We do these steps for every byte from left to right. After that, we try to look for a matched string from the previous data. As shown in Fig 3.4, we calculate the hash value in the same way and use this hash value as the address to read the hash table. The reading result is the address of a possible match. Then we can go back to the buffer and do the comparison byte by byte.

The process mentioned above is the most straightforward algorithm, which is also the algorithm used in Snappy. Zstd provides further improvements for it to get a better compression ratio. The first improvement is to change the input of the hash function from 3 bytes to 6 or 7 bytes, which means using 6 or 7 bytes data to calculate the hash value. The next one is to buffer the offset of the last match and look back at the same offset to find a match.

The single hash implementation cannot achieve a low compression ratio due to two reasons. First, when we use 6 bytes to calculate the hash value, the hash table cannot find a match that is shorter than 6 bytes. Second, the old data in the hash table is always replaced with the new one when a hash conflict happens. Thus we are not able to have multiple matches and select the best of them.

Therefore, the author of Zstd comes up with two variants of the single hash algorithm. One of them is to add an extra small hash table. In this approach, the algorithm maintains two independent hash tables with different sizes. The larger one uses a 7-byte hash function and the smaller one 3-byte. For each byte of data, the algorithm inserts twice to two tables, respectively. During the lookup, the algorithm will first look for a match by using the larger hash table, then use the smaller one if the larger one fails.

Another variant is normally called hash chain algorithm. The algorithm maintains a hash chain on a single hash table. In the inserting process, the algorithm will first check the position *hash(X)*, while x is the 3-byte data. If *hash(X)* is already occupied by old data, the data will be moved to the position *hash(X+1)*. While before moving to *hash(X+1)*, it will first check whether other old data occupy it. The other old data will be moved to *hash(X+2)* if there is. The algorithm iterates this process until *hash(X+k)*, while k means the maximum length of the hash chain. The data in this hash chain are naturally sorted by the time of inserting, and the oldest one will be discarded if the chain is full when inserting. In this scenario, there can be multiple matches found for each input hash value.

The hash chain algorithm and its variants are not the only way to look for repeated string. The other method is to use binary trees. In Zstd official implementation, if we set the compression level higher than 8, the algorithm will use binary trees to store the history information for looking up. However, the binary tree algorithm family is not suitable for FPGA implementation since the algorithm consists of many I/O accesses and they cannot be conducted in parallel to improve the performance of FPGA. Therefore, these algorithms are not discussed in this report as this project is FPGA-targeted.

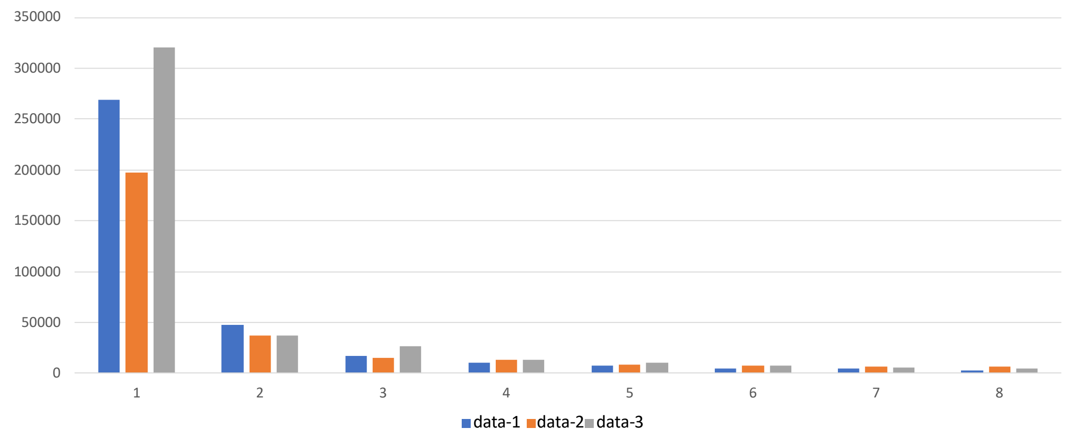


Figure 3.5: The positional of optimal matches in the hash chain

Finding the optimal combination of matches is an NP-hard problem. Therefore we need heuristic ways to reduce the computation while keeping a decent compression ratio. One way we can do is to analyze the relation between optimal matches and their offsets. In Zstd compression level 7, the algorithm maintains hash chains of a maximum of eight nodes, and it is capable of getting more than one match[3]. Even though it cannot always choose the optimal matches, but we assume that it is close to due to the low compression ratio. Fig 3.5 shows where we can find the best matches in the hash chains. The x-axis is the position of the chain, the smallest number 1 is the newest node, which means the offset is the smallest, and vice versa. We can easily conclude from this graph that if we have multiple matches matching from the same byte, the newer one has a higher possibility to be optimal.

The official Zstd compression algorithm provides 23 different compression levels from 1 to 23, in which level 1 indicates the worst compression ratio and fastest throughput, and level 23 is the other way around. Each level defines a specific compression algorithm and a set of configurations in the algorithm. In this project, we only benchmark the compression level of 1 to 9, since the algorithms used on level 10 and above are not feasible on FPGA.

3.3. Single thread benchmark

To analyze the single-core throughput and compression ratio of the Zstd, we compile the Zstd official software and do the benchmark on HP G8 machine. The HP G8 machine contains 128GB memory and Xeon E5-2631 server. The benchmark data is 200MB trading data in ERF format extracted from three different exchanges.

Compression level	1	2	3	4	5	6	7	8	9
data-1(%)	19.0	18.7	17.4	17.3	17.2	17.2	16.8	16.3	16.3
data-2(%)	15.8	15.6	14.9	14.8	13.8	13.5	12.8	12.7	12.7
data-3(%)	24	22.9	22.2	21.9	21.6	21.4	20.7	20.5	20.6

Table 3.1: Compression ratio of Zstd

Tab 3.1 demonstrates the compression ratio for three different kinds of data under different compression levels. As we can see from the table, compression ratio changes for a small amount after level 3. The compression ratio of data-2 is the highest and data-3 lowest.

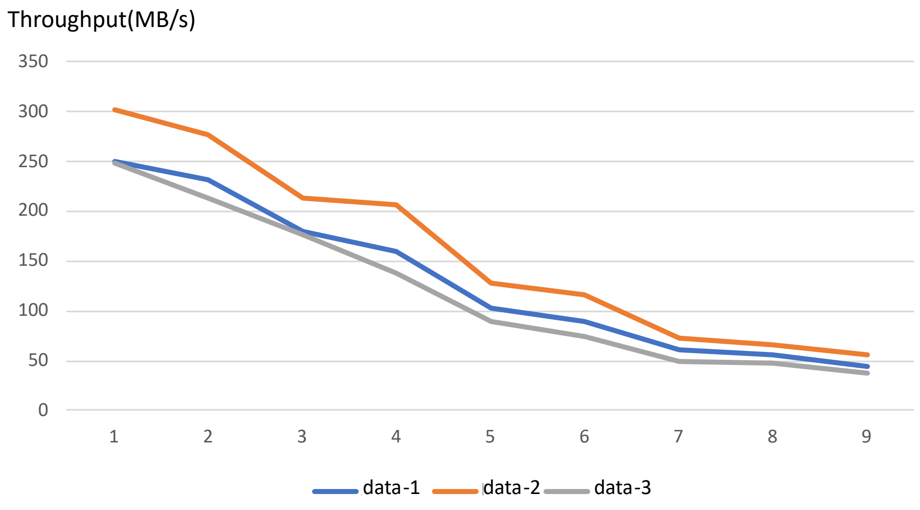


Figure 3.6: Throughput of Zstd software VS different compression level

Fig 3.6 shows the time consumed and throughput of Zstd software on different compression levels. As we can see, the throughput does not vary a lot for the three different kinds of data. The throughput we test is memory-to-memory throughput, which means the input data is from memory, and the compressed data is transferred to the memory.

3.4. Multi-thread benchmark and the analysis of bottleneck

As we know, a modern server CPU is usually comprised of dozens of physical cores, and each of them works independently. The G8 server used in the benchmark contains 28 cores, and the single-thread software only makes use of one of them. To test the full performance of the CPU, we implement a multi-thread version by bash shell script. In this version, the script creates 96 threads, while each thread compresses a 200MB slice of data-1. Since we want to know the relation between performance and the number of cores, we limit the number of cores to different numbers from 2 to 28. The testing result is presented in Fig 5.1. It is obvious that the performance is almost proportional to the number of cores. The memory utilization is not listed here because it is less than 1% of all the memory.

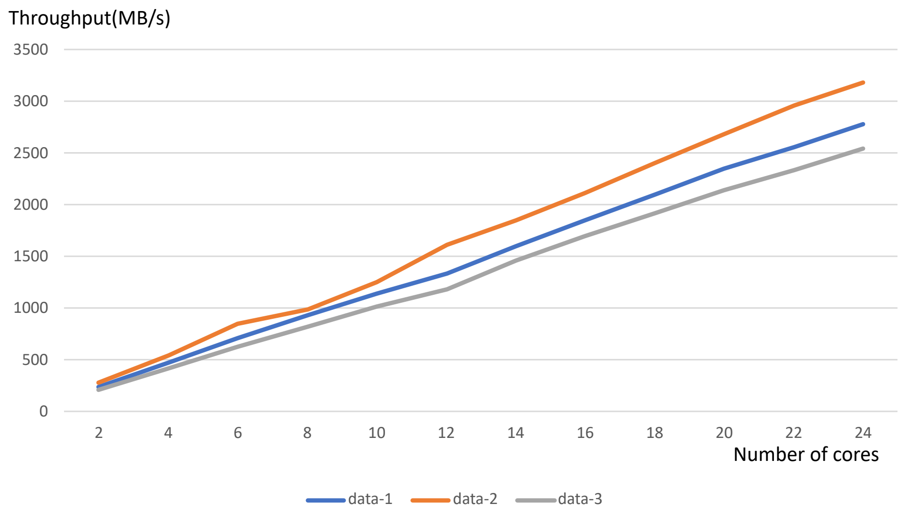


Figure 3.7: Throughput of Zstd software when using different number of cores under compression level 3

Fig 3.7 demonstrates the throughput when using different number of cores. As we can see, the throughput grows linearly in terms of the number of cores. Since we do not see a roofline in the figure as we expected, we believe that the bottleneck of the software performance should not be the bandwidth of the memory but inside the core. To further explore the performance, we use the CPU profiling tool to profile the performance of the CPU. From the profiling of CPU, we observe that the cache miss rate is only 17%, and the CPU utilization is 98%. Finally, we conclude that the bottleneck is in the computation resource of CPU.

3.5. Statistics of match length and match offset

The algorithms used in the software are not suitable to be implemented directly on the FPGA. Thus we need to modify it based on the features of FPGA and the trading data. Therefore, it is necessary for us to know the feature of Zstd when compression Zstd data.

Since the distribution of match length and match offset varies when we use different compression levels, we choose the level 7 for analysis. The reason is that the algorithm behind the level 7 is similar to the algorithm in hardware. To do the statistics for the match length and match offset, we modify the software to output the parameters of every match to a log file, and then we use Python to parse the log file.

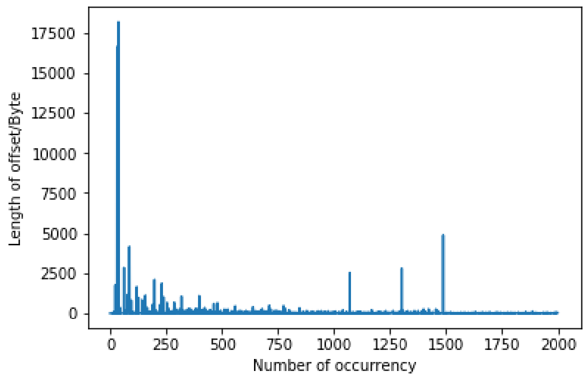


Figure 3.8: Distribution of offset for data-1

The size of the buffered history is an important factor for the compression ratio. If the size of the buffer

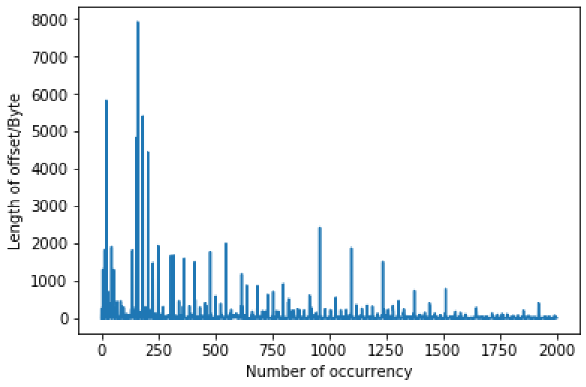


Figure 3.9: Distribution of offset for data-2

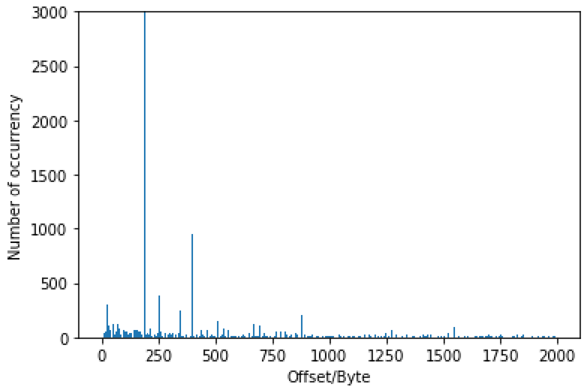


Figure 3.10: Distribution of offset for data-3

is too small, we will miss many matches. On the other hand, allocating a large buffer on FPGA consumes resources. Therefore, we need to analyze the offset of the match to make a balance between compression ratio and resource utilization. Fig 3.8, 3.9 and 3.10 shows the distribution of data-1, data-2 and data-3 respectively. We can see from the graphs that the number of occurrence goes down with the growth of offset while there are some spikes amid it. The offset of most matches are smaller than 2000. From our analysis, the spikes come from the features of the packages of Ethernet. For example, a large number of packages are the updating of price, the size of which is a fixed number. In this case, there will be a lot of matches with the same offset, which is the same of the size of the packages.

The length of matches is another important factor. Instead of doing comparison one byte by one byte, FPGA can read and compare multiple bytes from the buffer with the cost of more resources. Fig 3.12, Fig 3.13 and Fig 3.11 demonstrate the distribution of match length for data-1, data-2 and data-3 respectively. As we can see, the peak value of all three data sets is in the position of fewer than 8 bytes, and most of the matches are shorter than 40 bytes.

3.6. Repetition of offset

Zstd does an extra step to encode the offset of matches before the second stage. The algorithm buffers offsets of the 3 newest sets, and we call them repeated_offset1, repeated_offset2 and repeated_offset3 respectively. Then it replaces the offset value by a specific offset code defined in the Tab 3.2. As we can see, there is spe-

	offset=offset_1	offset=offset_2	offset=offset_3	offset=offset_1-1	Others
literal length=0	0	1	2	3	offset+3
others	1	2	3	offset+3	offset+3

Table 3.2: Repeated offset encoding

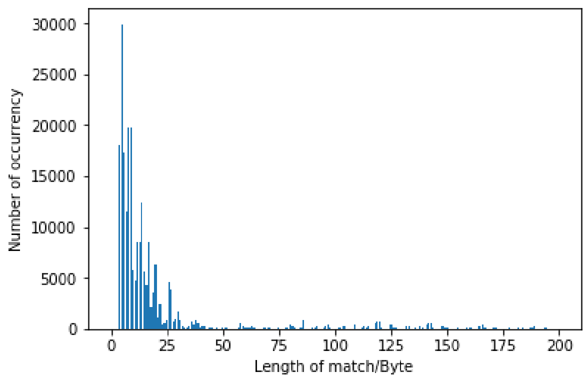


Figure 3.11: Distribution of match length for data-3

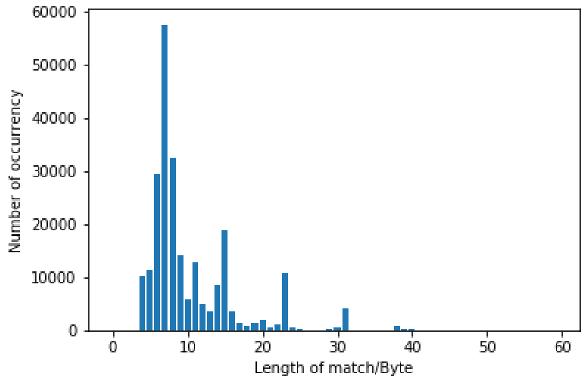


Figure 3.12: Distribution of match length for data-1

cial coding when offset equals to four special cases: repeated_offset1, repeated_offset2, repeated_offset3 and (repeated_offset1-1), and the special offset code can be compressed with fewer bits in the second stage compared with the regular code.

We found that in data-1 data, a large percent of the offset value, about 30%, can be encoded in the special case offset=offset_1. We believe that this is one of the reasons why Zstd can outperform Gzip and Snappy in the compression ratio.

3.7. Bad performance of Huffman coding

As we mentioned in Section 2.2, the compression ratio of Huffman coding depends on the distribution of the symbols, and it cannot guarantee that the compressed size is smaller than the original one. Therefore, Zstd will compare the compressed size of the original one for each block and only use the compressed block if it is smaller than the original.

To explore the gain of Huffman coding for the compression ratio, we add an extra counter to count the reduction of compressed size for each block in Zstd level 3. When calculating the benefit of the Huffman coding, we also need to take into account the extra space to store the Huffman tables, which is necessary for decompression. The benchmark is 100MB of all three kinds of trading data, and the result is listed in Tab 3.3. We can conclude that the benefit we can get from Huffman coding is comparably small. Thus we need to

Compression level	Reduced size(Byte)	Reduced compression ratio(%)
data-1	485300	0.46
data-2	631430	0.62
data-3	148487	0.14

Table 3.3: Gain of Huffman coding

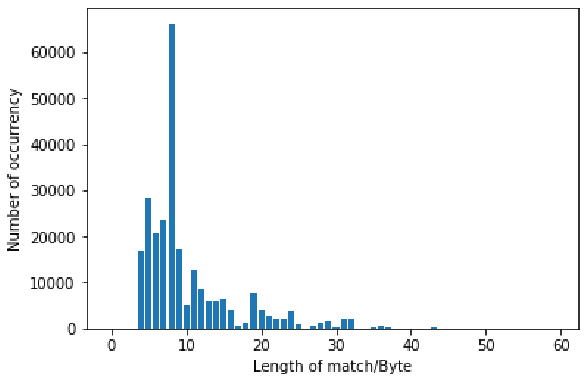


Figure 3.13: Distribution of match length for data-2

consider whether we should use the Huffman coding in our hardware design.

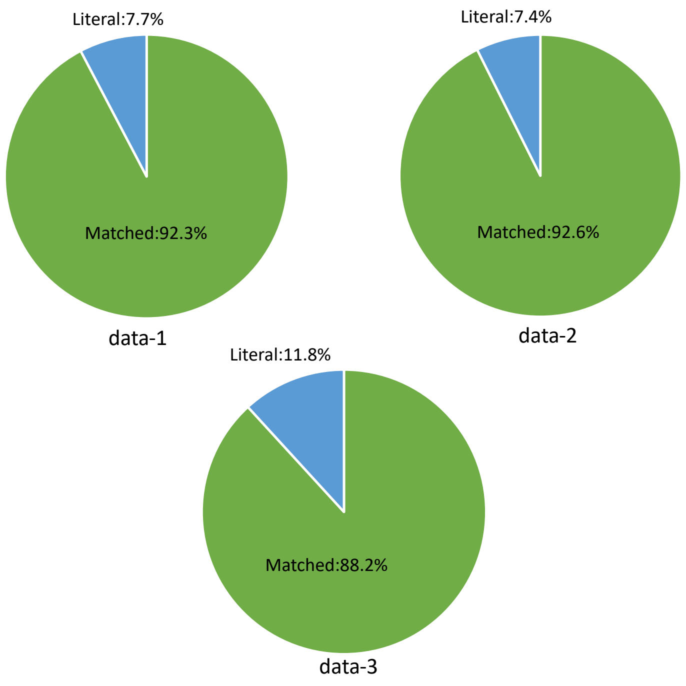


Figure 3.14: Percentage of data replaced by matches for data-1(left), data-2(right), and data-3(bottom)

From our analysis, there are two reasons why Huffman coding does not do well for trading data. The first is that the distribution of literal characters is comparably even, and the idea of Huffman is to replace characters with a higher possibility with fewer bits. The other reason is that only a small part of the data cannot be replaced with matches. As we can see from Fig 3.14, only about 10% of data is left uncompressed in the first stage and can be compressed by Huffman.

4 Hardware architecture

4.1. Overview of the architecture

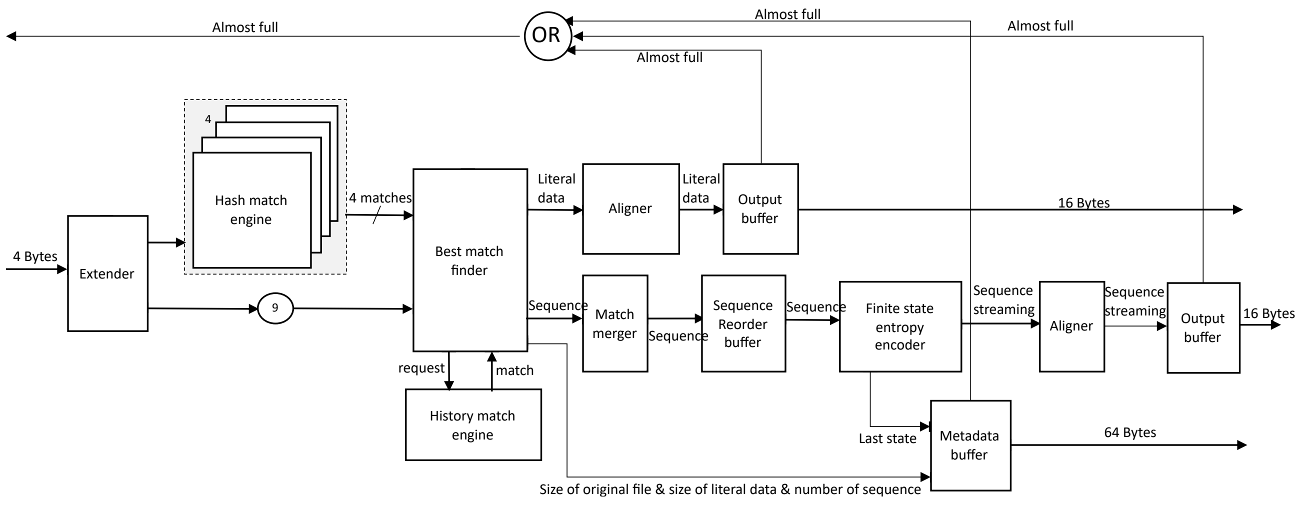


Figure 4.1: Architecture of compressor

Fig 4.1 presents the overview of the proposed architecture. Generally speaking, the compressor consists of two stages: pattern match stage and entropy stage.

The input throughput of the first stage is 4 bytes of data per cycle. For every 4-byte input, the compressor inspects the history to find matching strings and picks one of the matches when multiple matches are found. This stage consists of four modules: extender, hash match engine, best match finder, history match engine, and match merger. The outputs of the first stage are literal bytes and information of sequences. Literal bytes are data for which we cannot find a match in the history buffer. Each sequence consists of 3 information: the number of literal bytes before the match, offset of the match, and the length of the match.

In the second stage, the literal bytes and the sequences are processed independently. The literal bytes are aligned to make sure 16 bytes go into the output buffer in each valid clock cycle except the last one. The sequences are first compressed using a static FSE encoding table into bits. Then the bits are aligned for output.

Following the format of the Ztsd, we also need to know the number of sequences, the number of original bytes, and the last state of FSE encoder for each block. Therefore, we have a metadata buffer to fetch this information from the best match finder and finite state entropy encoder (FSE encoder). As we mentioned before, we can configure multiple output interfaces of different bandwidth in the Vitis platform, so we do not need to schedule the output of 3 output streamings ourselves.

4.2. Hardware algorithm of the pattern match

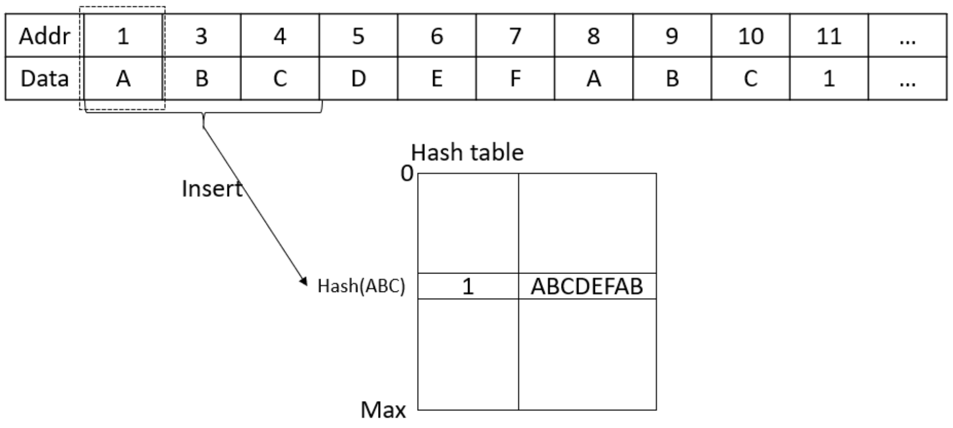


Figure 4.2: Insert data in hash table in hardware

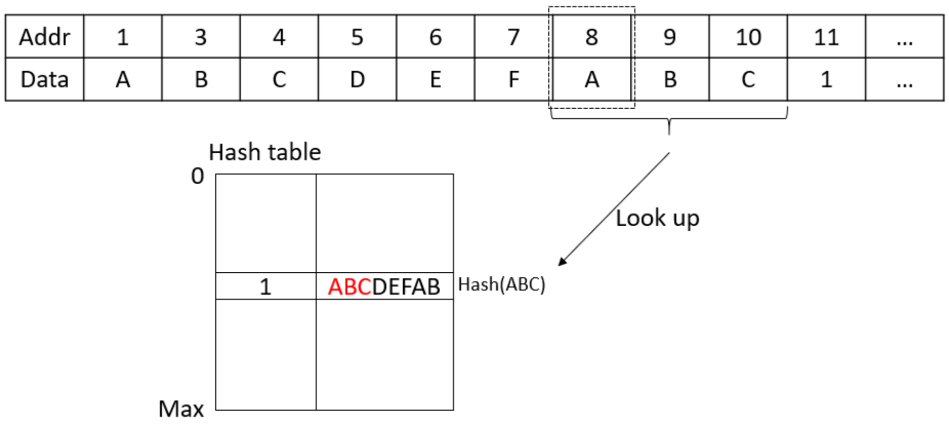


Figure 4.3: Look up from hash table in hardware

Different from the software, FPGA can read and compare multiple bytes at the same time. Therefore, we propose a new structure which achieves higher efficiency by exploiting these features. As presented in Fig 4.2, we insert the address, as well as 8 bytes of data into the hash table together. The process of lookup is presented in Fig 4.3. As we can see from the figure, when we look up the history buffer, we can get the address and an 8-byte string of the history at the same time. Therefore, we can find a match of a maximum of 8 bytes by accessing the memory only once. In this scenario, a hash table can process one byte of input data in each cycle.

By using the architecture described above, we can achieve an input throughput of one byte per cycle. In order to get better throughput, we want to process multiple input bytes at the same time, which means we need to do multiple insert operations and multiple lookup operations at the same time. However, the on-chip memory on FPGA has only one reading port and one writing port and can only do one inserting and one lookup each cycle. The solution is trivial: put multiple hash tables working together.

The architecture we proposed can process 4-byte input for each cycle so that there are four hash match engine modules, and each of them is in charge of looking for a match starting from the corresponding byte. As we can see from Fig 4.4, each hash match engine consists of four hash tables, and four hash tables handle four inserting respectively. Each hash match engine can process only one lookup operation, which means all four hash tables inside a hash match engine look for matches for the string starting from the same byte.

Fig 4.5 shows the general work distribution of four hash match engines. Four hash match engines process the same four insert operations and one different lookup operation. While inside each engine, four hash tables perform the same lookup and different inserts. Four engines contain 16 hash tables totally, and we can easily conclude that the number of hash tables is n^2 in terms of the input bandwidth or specifically $number\ of\ hash\ tables = number\ of\ input\ bytes^2$.

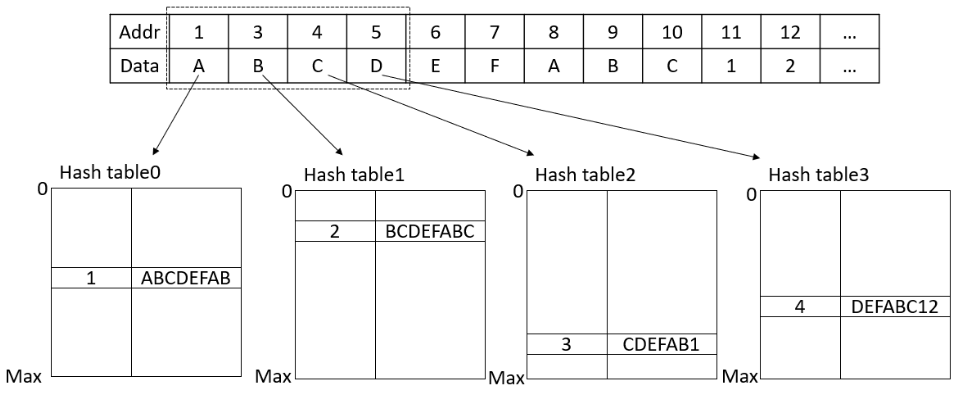


Figure 4.4: Structure of hash match engine

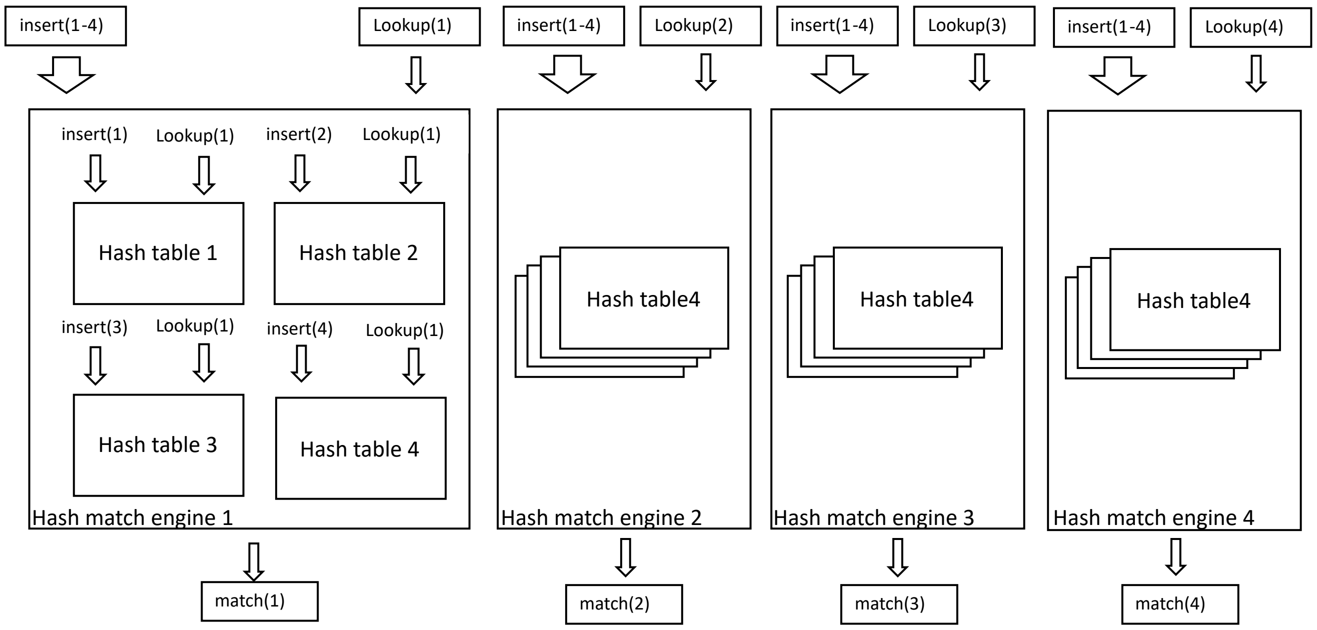


Figure 4.5: Four hash match engines to process 4 insert and 4 look-up operations

Since each hash table within a hash match engine is possible to find a match, we need to have a strategy to choose one from them as the output of the hash match engine when multiple matches are found. In this project, we explore three different strategies based on the data-1 and list the result in Tab 4.1. As presented in Tab 4.1, the strategy 1 achieves the highest compression ration, so we decide to use it. We do not compare the resource utilization of three strategies because the difference is negligible for a modern FPGA.

The hash table is implemented by on-chip memory on FPGA. Even though the FPGA chip on the Alveo board also has access to the large-capacity on-board DDR4 memory, the high and indeterministic reading and writing latency of DDR4 makes it unsuitable to be used to implement hash tables. Each hash table is comprised of two parts: address data and history data. The storage of the history data is implemented by URAM since its width is 8-byte, the same as the length of a string stored in the hash table. The address is a 32-bit unsigned integer number, and we will waste half of the storage if we use URAM to store it. Thus we implement the address storage by BRAMs since its widths can be configured to 4-byte.

As you can see from Fig 4.1, there are four hash match engines working together producing up to four matches each cycle. The produced matches go into the best match finder module. Since the range for these four matches are overlapped, the first thing the best match finder module needs to do is to choose one match from them. In this project, we propose and test two strategies and the result is shown in Tab 4.2. The earliest engine means the engine to look from the bytes with smallest address. For example, if the hash match engine 2, 3 and 4 in Fig 4.5 find matches in the same cycle, the engine 2 is the earliest. Different from our initial

	Strategy 1	Strategy 2	Strategy 3
Description	Choose the longest match. If multiple match have the same longest length, choose newest and longest one.	Always choose the newest match	Choose longest match. If multiple matches have the same longest length, choose a random one.
Compression ratio (Compare with Strategy 1)	100%	103%	115%

Table 4.1: 3 different strategies to choose a match within hash match engine module

	Strategy 1	Strategy 2
Description	Always choose match from the earliest engine	Choose longest match. If multiple matches have the same longest length, choose the match from earliest engine.
Compression size (Compare with Strategy 1)	100%	107%

Table 4.2: 2 different strategies to choose a match among hash match engines

hypothesis, the strategy 1 performs better in the compression ratio and we do further study on the reason of it. From our observation, the strategy 1 reduces the number of literal bytes at the cost of shorter average match length, and the extra literal byte surpasses the overhead of shorter match length. However, the benchmark of the test is only the trading files in this project and it is possible that the strategy 2 outperform the strategy 1 for some other data sets.

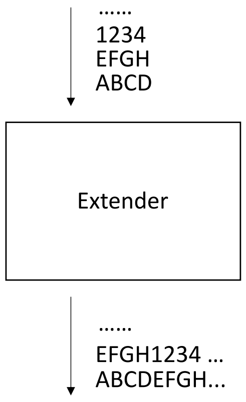


Figure 4.6: Extender module

The input bandwidth of the compressor is 4 bytes, while the data stored in the history buffer is 8 bytes. To foresee the data of the next cycles, we implement the extender module. The function of the extender is to buffer the input data and append the "future" data at the end. Fig 4.6 shows an example of input/output pattern. The module does not generate any new information.

By using the algorithm mentioned above, the compressor can handle 4-byte input data per cycle. However, the algorithm can only find a match that is less than 8 bytes since the string buffered in the hash table is only 8-byte long. Enlarging the buffer of the hash table is a trivial solution, but the larger the buffer is, the more resource it costs. On the other hand, as the Fig 3.11, Fig 3.12 and Fig 3.13 show, the history buffer should be about 35 bytes to cover most of the matches, which is not realistic if we want to achieve 10G/s on-chip throughput.

Therefore, we propose the history match engine module. As we can see from Fig 4.1, the history match engine is attached to the best match finder module. The function of the best match finder is to buffer the input data. When the best match finder module selects a match that covers to the end of the buffer in the match engine, the best match finder requests the history match engine to compare the data after the match.

The buffer in the history match engine is implemented by URAM which takes a longer time for reading and writing compared with BRAM. So that the latency of the history match engine module is two cycles to

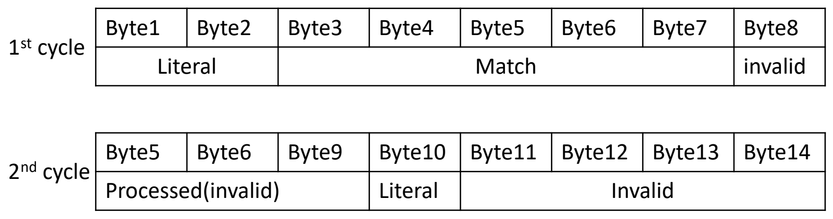


Figure 4.7: Some starting bytes are processed in the last cycle

keep a high working frequency. Even though the input data is extended to more than four bytes per cycle, only the starting four bytes can be regarded as valid bytes. The best match finder only outputs literal data starting from the valid bytes and the literal bytes from the valid bytes, and the extended bytes are only used to determine the length of the matches. The starting four bytes are not always regarded as valid bytes since the match in the last cycle can cover part of them. Fig 4.7 shows an example of this case. The match covers from byte3 to byte7, thus the second cycle sees only one valid byte and thus can only output only a literal byte. As we mentioned, each match must be at least 3 bytes following the definition of Zstd, but the matches found by the history match engine can be less than 3 bytes. Since the matches from the history match engine are essentially not an independent match, but an extension of the last match.

The size of the history buffer is 65536 bytes, which are the size of two URAMs, and the history match engine cannot extend the match whose offset is longer than the size of the buffer. On the other hand, the data in hash tables are only replaced when hash conflict so that it can store history at any offset in theory. Therefore, the best match finder checks the offset before using requesting the history match engine. A special case is that the match in current cycle covers all four starting bytes for the next cycle so that there is no output for the next cycle. When a history match is requested, it is always in this special case, and there is no valid output in the next cycle.

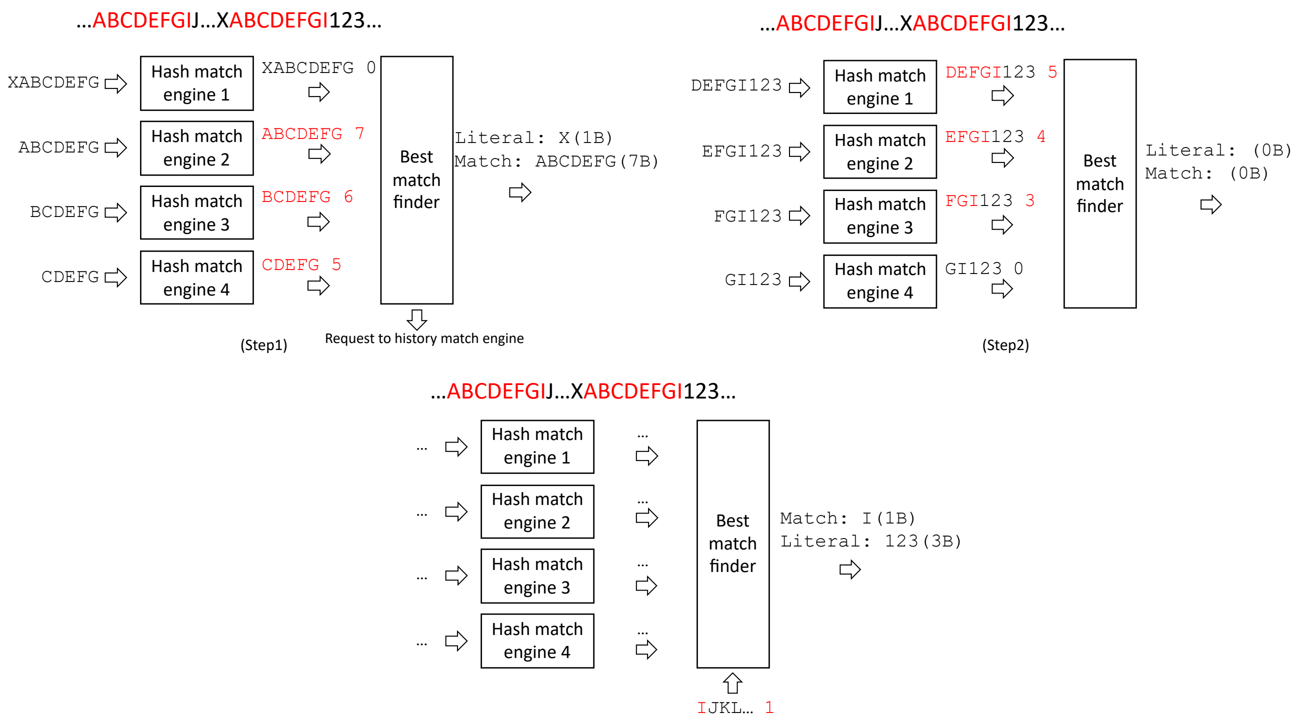


Figure 4.8: Steps to request history match engine

The selection of matches relies on the decision of the last cycle so the selection in the current cycle must be completed before starting the next selection. However, in our architecture, the history match takes two cycles to look for a match from history. It seems that the best match finder needs to stall for one cycle to wait for the history match engine, but in practice, the best match finder is always in the special case mentioned in the last paragraph when the history match is requested. This feature ensures that the history match engine gets an extra cycle to find a match. Fig 4.8 shows an example of how the best match finder and the history match engine cooperate. In the first cycle, hash matches engine 2, 3 and 4 find matches. Then the best match finder picks the match from engine 2 since engine 2 is the earliest among the engines finding matches. Since the match extends to the end of the history buffer inside the hash engine, the best match finder sends a request to the history match engine at the same time. In the second step, since the starting 4 bytes are already processed in the first cycle, no valid bye and no output. Finally, in the third cycle, the history match engine returns a match of 1 byte. The best match finder takes the 1-byte match and outputs a 1-byte match and three literal bytes.

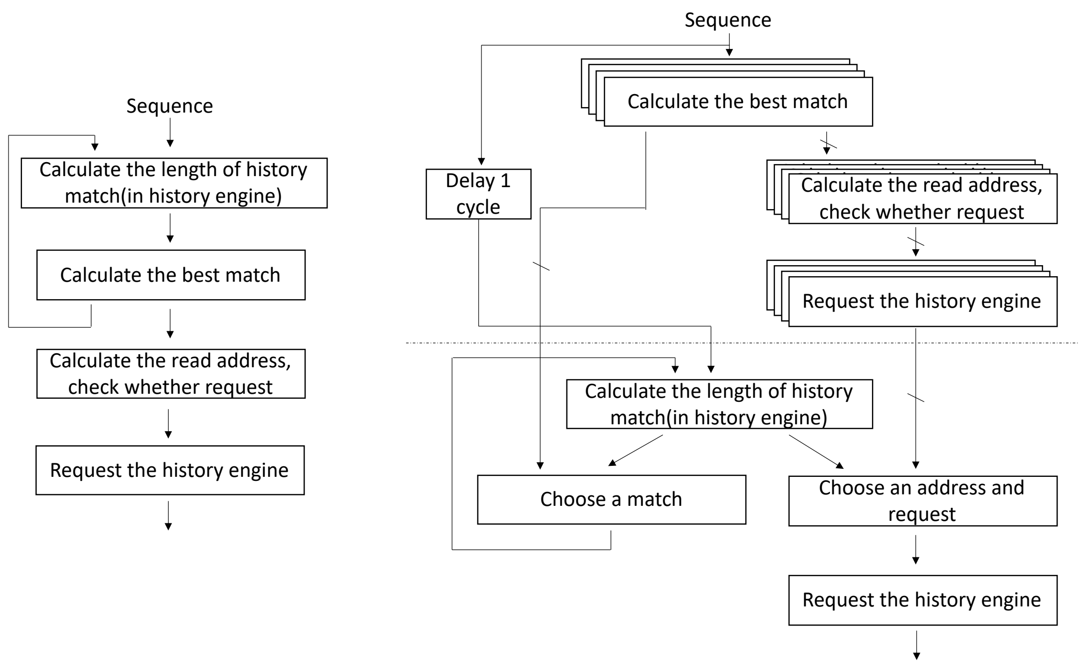

Sequence
Calculate the length of history match(in history engine)
Calculate the best match
Calculate the read address, check whether request
Request the history engine
Sequence
Calculate the best match
Delay 1 cycle
Calculate the read address, check whether request
Request the history engine
Calculate the length of history match(in history engine)
Choose a match
Choose an address and request
Request the history engine

Figure 4.9: Best match selecting logic with(right) and without(left) pre-calculation

For a fully pipelined architecture, we can quickly improve the working frequency by increasing the number of cycles and reduce the amount of computation for the critical cycles. However, one stage in the best match finder module cannot be further separated, because the module needs to know the decision of the best match of the last cycle to make the decision of this cycle in one cycle. The trivial solution cannot meet our desired 300MHz. Therefore we propose an optimization by pre-calculation. The initial and improved workflows are presented in Fig 4.9. In the initial workflow, the first step is to check the length of the history match, which is implemented by combinational logic in the history match engine. Secondly, the best match finder uses the length of the history match engine to decide which hash match should be used. The third step is to check whether we need to request the history match engine again and the address of requesting. Finally, the best match finder requests the history engine if needed. The result of the first step, which is the length of history match, is a number from 0 to 8. From our analysis, the value from 4 to 8 causes the same result of the next step: no hash match will be used. Based on this fact, we classify the length of history into 5 cases: 0, 1, 2, 3, and larger than 3, and duplicate the step 2 and 3 to compute the results for each case. As presented in Fig 4.9, the step 2 and 3 are duplicated for four times to do the computation of cases when the length equals to 0, 1, 2 and 3 in the previous clock cycles. The case of larger than 3 does not need any computation because no hash match should be used. Then in the critical cycle, the best match finder uses the length of history match to make the selection directly without any extra computation. Even though we spend more resources to do the redundant computation, we improve the working frequency of the whole compressor.

The output of the best match finder comes from three sources: match from hash match engines, match from history match engine, and literal. The matches from hash match engines must be at least 3 bytes, and the matches from history match engine must start from the first byte of input data of each cycle. All the possible combinations of output are listed in Tab 4.3. There is a special pattern [match from hash][match from hash], of which the first match is 3-byte. Since each hash match is inferred to a sequence by adding the literal byte

Possible patterns	Comment
[Match from history] [Literal] [Match from hash]	
[Match from history] [Match from hash]	
[Match from history]	
[Match from history] [Literal]	
[Match from hash] [Literal]	
[Match from hash]	
[Match from hash] [Match from hash]	Infeasible
[Literal] [Match from hash]	
[Literal]	

Table 4.3: All possible patterns of the output of best match finder

in front of it, the special pattern will generate two sequences in the next stage. In this case, we need to have much more extra logic to handle it if we want to keep our architecture a pipeline. Therefore we decide to eliminate this pattern by changing the second match into literal and $[match\ from\ hash][match\ from\ hash]$ becomes $[match\ from\ hash][Literal]$. It is not easy to process all nine kinds of feasible patterns directly, and there can also be some invalid bytes in the beginning of each 4-byte slice, so we designed a general pattern to present all the feasible patterns. The format of a general pattern is

$$[Match\ from\ history/Invalid\ starting\ bytes][Literal\ 1][Match\ from\ hash][Literal\ 2],$$

and there are six constraints on the general pattern:

- The length of each element can be zero
- The lengths of a match from history and literal 1 cannot be non-zero together.
- The length of Literal 2 can only be 0 and 1
- When the length of Literal 2 is 1, the lengths of a match from history and literal 1 must be 0.
- There cannot be a match from history and invalid starting bytes at the same time.
- There cannot be Literal 1 and Literal 2 at the same time.

The constrains of the general pattern are used in the simulation to check whether the output of the best match finder is correct.

Once the best match finder module decides the pattern, it can pick the data which cannot be replaced by matches and generate the literal byte streaming. Literal 1 and Literal 2 are processed differently. The Literal 1 will be picked and sent to the output, while the Literal 2 is buffered and appended to the beginning of the literal output of the next cycle. The maximum length of Literal 1 is four bytes and Literal 2 one byte, so that the maximum literal bytes generated each cycle is 5 bytes.

4.3. Match merger

The output of the best match finder consists of two streamings. The first one is a literal bytes streaming for data which cannot be replaced by matches. The second one is a sequence streaming in the general pattern mention above. The pattern stored in Zstd should comply the pattern $[Literal][Match]$, therefore we propose the match merger module to convert the general-pattern sequence into Zstd-pattern sequence.

The function of match merger module is to merge multiple components cross sequences into one component following the rules listed in the Tab 4.4, and the rules can be applied multiple times on one components. The lengths of all components in the table is non-zero. For example, the pattern

$$[Match\ from\ hash][Match\ from\ history][Match\ from\ history]$$

can be converted into $[Match\ from\ hash]$ by using the second rule twice. The rule 3 can only be used when two matches have the same offset. And this pattern only occurs when the offset is out of the range of history match engine, otherwise the second match from hash should be found by history match engine.

Before merge	After merge	Comment
[Literal 1/2][Literal 1]	[Literal 1]	
[Match from hash][Match from history]	[Match from hash]	
[Match from hash][Match from hash]	[Match from hash]	Only if the offsets of two matches are the same

Table 4.4: The merge rules (the lengths of all components are non-zero)

After merging, all the matches from hash will be eliminated and a literal will not be followed by another literal. The next step is to output sequences in the Zstd-pattern $[Literal\ 1/2][Match\ from\ history]$, of which the length of the Literal can be zero and the Match cannot. Since the input goes in one by one, we need a mechanism to check whether the merging of the last Zstd-pattern sequence are finished. From the observation, we can easily find that the Literal and the Match can never be merged, therefore the incoming a a new $[Literal\ 1/2]$ indicates the end of the last sequence.

Another function of the best match finder is to split the sequences and literal data into blocks. We decide to implement the splitting work in the best match finder module since this is the last module two streamings are synchronized. After going out of the best match finder, the two streamings will be processed independently with different latency. Following the definition, the original uncompressed size of each block contains no more than 128KB. However, the on-chip memory on FPGA is the critical resource, so we do not want to buffer a large block. Therefore, we count the number of sequences to add an extra limit on the number of sequences below 256 for each block.

On the other hand, we want to make a balance on the resource utilization of URAM and BRAM because the capacity of on-chip memory is likely to be the bottleneck from our experience. The FPGA on Alveo U200 contains 1914 BRAMs and 960 URAMs. In other words, the number of BRAMs is about twice as much as URAM.

4.4. Hardware algorithm of the entropy encoding

As we mentioned in Section 2.2, the literal streaming and sequence streaming are compressed in reverse order if the Huffman coding and FSE coding are applied. At the same time, we generate the literal bytes and sequences from the beginning to the end. Thus we need to buffer each block to change the order. As we mentioned in Section 3.7, the Huffman coding introduces almost no improvements in the compression ratio but requires extra resources to do the reverse buffering. Therefore we decide not to use Huffman coding. As mentioned in Section 4.2, the number of the output of literal bytes varies in each cycle and can be up to 5 bytes. Therefore we need to align the data before going into the output buffer, and this is the function of the aligner module. After the alignment, the literal streaming contains 16 bytes in each valid cycle except the last one.

	Dynamic FSE tables	Static FSE tables
data-1	17.4%	17.6%
data-2	14.9%	15.0%
data-3	22.2%	22.3%

Table 4.5: Compression ratio when using dynamic FSE tables and static FSE tables

The generation of FSE tables consists of several steps: calculate the possibility of every symbol, normalize the possibility, and generate the transformation tables [3]. The computation in these steps is highly serial and thus is not suitable for FPGA implementation. To measure the overhead on the compression ratio of using static tables is small compared with using dynamical tables, we first add code to the software implementation to output the tables of all blocks when compressed. Then we add tables of one block and force the software to use the pre-defined tables and record the compression ratio. As listed in Tab 4.5, the overhead of using static FSE tables is small.Therefore, we conclude that the method of using static FSE tables is more suitable for the trading data in this project.

The first step to process the sequence streaming is to reverse the order of sequences in each block. Fig 4.10 presents the architecture of the reorder buffer. The reorder buffer contains two stacks that work as first-in-last-out buffers. In the beginning, the input data of recorder buffer goes into the stack 1. When the stack 1 becomes full, the recorder buffer outputs the data in stack 1 in a reverse way store the new input in the

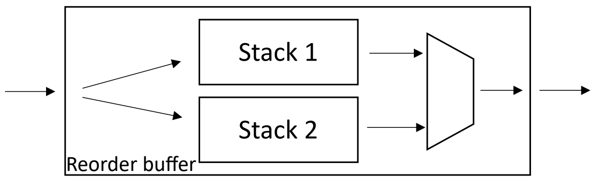


Figure 4.10: Reorder buffer

stack 2, and the work of two stacks switch back when the stack 2 becomes full. The reorder buffer sends a sequence out a sequence in every cycle once there is at least one sequence ready for output. This mechanism guarantees that the speed of output will never be slower than the input. Therefore, when one stack becomes full and ready to output, the other stack will always be empty. Since we decide to make every block contains no more than 256 sequences, the depth of each stack is 256, and each stack can be implemented by one BRAM.

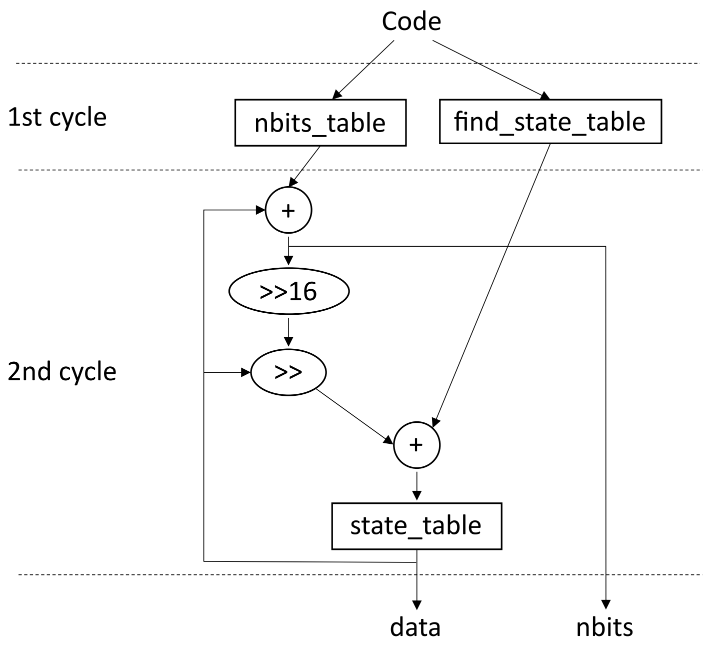


Figure 4.11: Architecture of FSE encoding

After changing the order, sequences are sent into the finite state entropy (FSE) encoder module to compress. As we mentioned in Section 2.2, the offset, match length of literal length are first encoded into codes variant length of bits independently. The code is compressed by FSE and the bits are appended at the end of the compressed code. The architecture to compress the code is presented in Fig 4.11. As we can see from the Figure, the compression takes two cycles and two outputs are generated: data and nbits. Data is the compressed bits and nbits indicates the number of valid bits in the data. Finally, all the valid bits of offset, match length and literal length are merged together.

Since the length of every elements in a compressed sequence is variant, the lengths of all each compressed sequences are also not the same. Therefore, we also need to have an aligner module after the FSE encoder to make the width of output the same in every cycle.

As presented in Fig 4.1, the streamings of literal data and sequence data are buffered separately before going into the Vitis platform. Since the whole compressor does not have any stall mechanism, we need to make sure the output buffer modules and the metadata buffer modules will not be overflowed. Each of the buffer can generate an almost_full signal to stop the input data from going in when the numbers of elements in the buffers reach certain thresholds.

Since the pipeline architecture buffers data in the pipeline naturally, each of the thresholds depends on the size of buffer and the number of data which can be buffered in the corresponding data path. Tab 4.6 listed

Module	Latency(cycles)	Comment
Extender	4	
Hash match engine	8	
Best match finder	7	
History match engine	2	
Aligner_literal	4	Buffer the data of only 1 cycle output
Match merger	4	
Sequence reorder buffer	257	
Finite state entropy encoder	3	Buffer the data of only 1 cycle output
Aligner_sequence	2	Buffer the data of only 1 cycle output

Table 4.6: Latency of modules

the latency of every module. The latency of two aligner modules are larger than two, but the data buffered is only for one-cycle output. The data routine of the first stage is extender->hash match engine->best match finder and thus the total latency is 19 cycles. The history match engine works with best match finder and does not add extra latency. The data buffered in the second stage for literal streaming is one, so the total amount of data is 20 cycles and the almost_full signal must set to high when the number of empty entries of the literal output buffer equals to or smaller than 20. The routine of the sequence in the second stage is match merger->sequence reorder buffer->finite state entropy encoder->aligner, so that we can conclude that the data buffered for the sequence can be up to 274. The data path for the metadata is different. Since only one set of metadata is generated for each block and each block consists of no more than 256 sequences, we can easily find that there cannot be more than 2 blocks stored in the data path and thus the redundant place for metadata buffer is 2.

4.5. Verification of single kernel

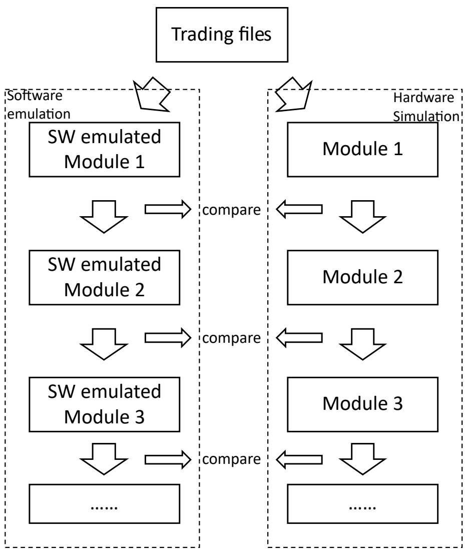


Figure 4.12: Test workflow for single testbench

Verification is an important step to make sure the hardware works correctly. In this project, we design own verification workflow. First, we use C++ to generate a software emulation of every module of the hardware compressor. The software emulation is not cycle-accurate to the hardware implementation but generates the same outputs. On top level of the hardware implementation, we also connect the output of every module to the output of the compressor for debugging purpose. The idea of the testbench is presented in Fig 4.12. We compress the file by software emulation and hardware simulation at the same time, and compare the output of every module to see whether they are the same.

The testing files include the trading files from all three exchanges. However, as we mentioned, trading files from three exchanges show similar features, and the coverage of the testbench is low if we only use the trading files. Therefore, we add other testing files from Silesia corpus. Silesia corpus[15] is a set of files specifically used as benchmark of data compression.

4.6. Multi-kernel accelerator

To fully utilize the bandwidth of PCIe, we implement multiple compressor units on the same FPGA. Each compression task consists of four steps:
1. The host machine sends data to the on-board memory
2. Compressor compresses the data
3. Host machine fetches compressed data from FPGA
4. Host machine formats the compressed data into Zstd format

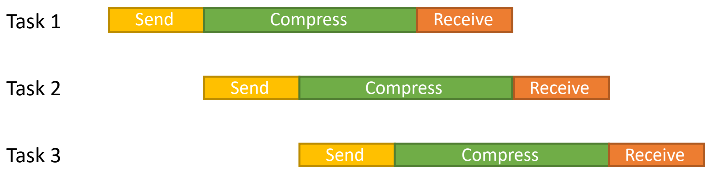


Figure 4.13: An example of task scheduling

Step 4 is totally on the host machine thus the task scheduling only takes into account Step 1 to 3. The trivial way is to do the 3 steps for every tasks one by one. But in this scenario, only one compressor is running at any time and we cannot get any improvements on the performance. From our analysis, the system can do sending, compressing, and fetching at the same time. PCIe is full duplex so the sending and fetching should not influence each other in theory. Based on these observations, we implement a scheduling policy presented in Fig 4.13. All the files are sent to the on-board memory one-by-one. Once the sending of a file is completed, the corresponding compressor starts compressing and the sending of the next file starts. The fetch will only start when the compression work of a file finishes. Even though multiple sending and fetching can start at the same time, the bandwidth of PCIe is limited and we decide to do them one by one.

5 Experimental result

5.1. Experimental setup

In this chapter, we discuss the performance measurements and experimental results of the FPGA implementation of the HW compression algorithm to show its advantages in practice. First we start by describing the experimental setup we used to do the measurements. The host server used for the measurements is an HP G8 machine. The HP G8 machine contains 128GB memory and two Xeon E5-2690 server-class CPUs. Each CPU consists of 8 single-threaded cores, which are working at 2.9 GHz, giving a total of 16 cores in the system. The FPGA used in this project is an Alveo U200. The interface between the host machine and the FPGA is PCIe 3.0 x 16. As mentioned in Section 2.3, the host-to-FPGA and FPGA-to-host throughputs are 11.4GB/s and 12.1GB/s, respectively. All the throughput mentioned in this chapter is memory-to-memory throughput, which means the original data is read from host memory and written back to host memory. This limits the maximum throughput to 11.4GB/s. Since the compressed data is stored in the host memory after the compression, the bandwidths of disk and Ethernet are not taken into account.

In this chapter, the Zstd software used is the official software implementation from Facebook. We used the Zstd version 1.4.2. The Zstd model is a C++ model we built to simulate the behavior of the hardware. Since the hardware compilation on FPGA takes hours, we use the model to explore different configurations and see the compression ratio. The benchmark data used in this chapter is the data used throughout this thesis, represented by data-1, data-2 and data-3, which are trading data fetched from exchange-1, exchange-2 and exchange-3, respectively.

5.2. Compression ratio

In the architecture proposed in the last chapter, there are several parameters we can use to improve compression ratio, for example the size of hash tables, the size of the history buffer and the length of history hash function. In this chapter, we evaluate the influence of the 3 configurations on compression ratio in this section.

Since the compression consists of two stages, the compression ratio depends on the performance of both stages. However, when we change the parameters in the first stage, the statistics of the values in the sequences are also changed. Thus, we cannot use a single set of static tables to evaluate all configurations. In this project, we use an easy way to decouple the influence between two stages. As we already mentioned in Chapter 4, we built a model in C++ to emulate the compression ratio of the hardware. We change the static tables FSE compression in the second stage to the dynamic method, which is the same as the official software implementation. In this scenario, the first stage of the compression architecture is the hardware one and the literal data are left uncompressed as hardware. While the software calculate the dynamic tables and compress the sequences dynamically, thus we eliminate the influence of improper static tables. We use the model to compress 50MB slice of three kinds of trading data, and the compression results are listed in Tab 5.1, Tab 5.2, and Tab 5.3, respectively. The length of hash function indicates the number of bytes used as the input of hash function. The size of hash table listed in the tables indicates the number of entries in each hash table. We test two different modes: single-table and double-table. In the single-table mode, each hash match engine contains 4 hash tables and in the double mode, each hash match engine contains 8. In the double-table mode, all 8 hash tables are of the same size, which is the size of hash table listed in the tables. Four of the hash table

Single or double table	Length of hash function	Compression ratio	Size of hash table
Single	3	0.220	4096
Single	4	0.212	4096
Single	5	0.215	4096
Single	6	0.220	4096
Single	7	0.232	4096
Single	3	0.210	8192
Single	4	0.212	8192
Single	5	0.212	8192
Single	6	0.216	8192
Single	7	0.226	8192
Double	3	-	4096
Double	4	0.210	4096
Double	5	0.209	4096
Double	6	0.207	4096
Double	7	0.207	4096
Double	3	-	8192
Double	4	0.209	8192
Double	5	0.208	8192
Double	6	0.207	8192
Double	7	0.207	8192

Table 5.1: Compression ratio of data-1 under different configurations

use the hash function of 3-byte, while the other four use the hash function of which the length is listed in the tables. In the double-table mode, the usage of on-chip memory doubles.

As we can see from the table, when we use single-table mode, we can observe a clear improvement if we increase the size of hash table from 4096 to 8192. However, the RAM resource utilization of double-table mode with 4096 is the same as single table with 8192 entries, but the compression ratio is better. On the other hand, we can also conclude from Tab 5.1, Tab 5.2, and Tab 5.3 that increasing size of hash table from 4096 to 8192 can hardly improve the performance when the mode is single-table.

From our analysis above, we decide to make the size of hash table 4096 and put the compressor in single-table mode. In this configuration, the 4-byte hash function is the most suitable one.

Another parameter we want to explore is the size of history buffer. From our analysis in Section 3.5, the offsets of most matches are smaller than 2000. Therefore, we estimate that the increase on the size of history buffer will not bring significant increase on the compression ratio. We do a test to verify our hypothesis, the result is presented in Tab 5.4. As we can see from the table, When we increase the size of hash table from 8192 to 16384, compression ratio was improved only a little at the cost of 2 times more RAM resource utilization on the table.

5.3. Throughput

In theory, the throughput of a single compression kernel can be up to 1.2GB/s. However, it takes time to send the raw file to FPGA and fetch compressed data from FPGA. We test the throughput by compressing the 400MB data-1 (repeated 4 times) and the result is listed in Tab 5.5. As the table shows, the single kernel throughput is about 740MB/s. The sending time is the time consumed for transferring data from host to FPGA. The compression time is the time used to run the on-chip compression algorithm. Receiving time is the time consumed by the host machine to read all the data from the FPGA on-board memory. In our compression kernel, the type of data has no effect on the sending time and compression time. The host machine takes 3 steps to fetch data from the FPGA on-board memory:

1. Fetch meta data
2. Parse the meta data to get the lengths of the literal streaming and the sequence streaming
3. Fetch the literal streaming and the sequence streaming

Single or double table	Length of hash function	Compression ratio	Size of hash table
Single	3	0.160	4096
Single	4	0.157	4096
Single	5	0.157	4096
Single	6	0.171	4096
Single	7	0.181	4096
Single	3	0.160	8192
Single	4	0.158	8192
Single	5	0.156	8192
Single	6	0.169	8192
Single	7	0.179	8192
Double	3	-	4096
Double	4	0.157	4096
Double	5	0.155	4096
Double	6	0.156	4096
Double	7	0.155	4096
Double	3	-	8192
Double	4	0.156	8192
Double	5	0.155	8192
Double	6	0.155	8192
Double	7	0.155	8192

Table 5.2: Compression ratio of data-2 under different configurations

The receiving time is the time spent on all three steps combined.

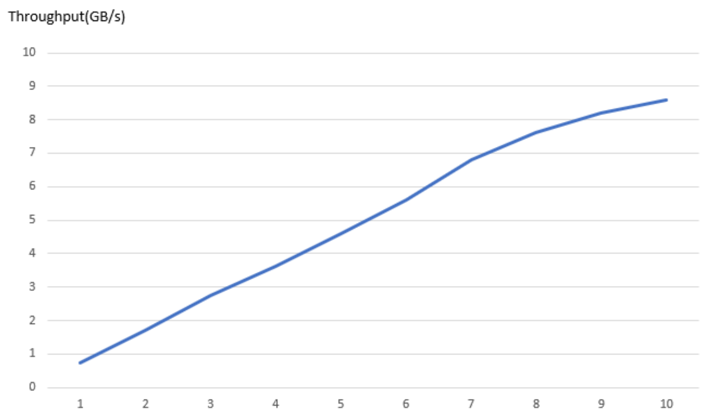


Figure 5.1: Compression speed VS number of hardware kernels

To increase the compression throughput, we place multiple kernels on the same FPGA and compress multiple files at the same time. We first implement 10 kernels on the FPGA, and enable part of them to do compression. The throughput of the different kernels is listed in Fig 5.1. The x-axis in the figure indicates the number of kernels and y-axis is the throughput. As the figure shows, when we increase the number of cores, the throughput first grows linearly to about 7GB/s. Then the growth of throughput slows down and finally reaches about 8.6GB/s when using 10 kernels.

The resource utilization of a single kernel is listed in Tab 5.6. The critical resource is the BRAM, which takes 4.77% of all the resource. In this project we place 10 kernels on the FPGA, total resource utilization is about 47.7%. And the total throughput of 10 kernels can be up to 12GB/s in theory.

5.4. Analysis of compression ratio

In the last chapter, we list the compression ratio between Zstd software, Zstd model, Zstd hardware, and Snappy software. In this chapter, we explore the reasons for the difference in compression ratios.

Single or double table	Length of hash function	Compression ratio	Size of hash table
Single	3	0.248	4096
Single	4	0.246	4096
Single	5	0.252	4096
Single	6	0.256	4096
Single	7	0.267	4096
Single	3	0.246	8192
Single	4	0.244	8192
Single	5	0.243	8192
Single	6	0.242	8192
Single	7	0.242	8192
Double	3	-	4096
Double	4	0.243	4096
Double	5	0.241	4096
Double	6	0.240	4096
Double	7	0.240	4096
Double	3	-	8192
Double	4	0.242	8192
Double	5	0.241	8192
Double	6	0.240	8192
Double	7	0.239	8192

Table 5.3: Compression ratio of data-3 under different configurations

Testbench	Size of history buffer(Byte)	Compression ratio
data-1	8192	0.210
data-1	16384	0.208
data-2	8192	0.160
data-2	16384	0.159
data-3	8192	0.246
data-3	16384	0.243

Table 5.4: Compression ratio under different size of history buffer

To compare explore the difference in compression ratio, we list the different items of compression algorithms in Tab 5.7. The sequence encoding and literal encoding indicate the type of tables used to do the encoding. Size of hash tables means the total number of entries in the hash table. The tables of three Zstd methods are generated following the type of data, and the Snappy table is fixed in the format of Snappy. The Zstd software and the Snappy software use one large hash table, while the Zstd model and the Zstd hardware use four small hash tables in each hash match engine. Sliding windows not used in Snappy. As mentioned in Section 2.2, Snappy slices data into 64KB blocks, and only matches within the block are allowed.

The compression ratio of 400MB slices of all three types of trading data using Zstd software, Zstd hardware, and Snappy software are presented in Tab 5.8. The compression ratio of the hardware is different from the software model and the official software implementation. We start from the analysis between hardware implementation and software model.

The first stages of the hardware and the model are the same. However, the second stages are different. The model uses dynamically-generated tables, which is the same as the official implementation, to encode the sequences, and the hardware uses static ones. Compared with the dynamic ones, the performance of static tables are slightly worse but close, which is the same as we expected.

We observe that the compression ratio of our hardware compressor is higher compared with the official software implementation with compression level 3. We believe that there are three main reasons for that:

1. Smaller hash table and history buffer
2. Static FSE tables and no Huffman encoding

	Sending time(s)	Compression time(s)	Receiving time(s)	Total time(s)	Compression speed(MB/s)
1	0.126	0.332	0.089	0.547	731
2	0.143	0.321	0.082	0.546	732
3	0.148	0.323	0.083	0.554	722
4	0.132	0.327	0.098	0.557	718

Table 5.5: Single kernel throughput

	LUT	Flip-flop	BRAM	URAM	DSP
Single kernel	8883(0.89%)	8670(0.41%)	87(4.77%)	22(2.29%)	16(0.23%)

Table 5.6: Resource utilization of a single kernel

	Zstd software (Level 3)	Zstd model	Zstd Hardware	Snappy software
Sequences encoding	Dynamic	Dynamic	Static	Static
Literal encoding	Dynamic	No	No	No
Size of hash table	131072	16384	16384	4096
Size of history buffer	512KB	64KB	64KB	64KB
Sliding windows	Yes	Yes	Yes	No
Match from the last offset	Yes	No	No	No

Table 5.7: Algorithm comparison among Zstd software, Zstd model, Zstd hardware and Snappy software

	data-1	data-2	data-3
Zstd software	0.174	0.149	0.232
Zstd hardware	0.236	0.181	0.269
Snappy software	0.279	0.212	0.313

Table 5.8: Compression ratio among Zstd software, Zstd hardware and Snappy software

3. Do not look up from the last offset

From the previous tests, we already found that the first and second points do not have a considerable influence on the compression ratio. Therefore, we believe that the main difference is from the third point. From our analysis, if we implement this feature in hardware, the throughput will be significantly reduced from 20% to 30%. Since after deciding the best match, we need to look back at the history and possibly delete all the intermediate results in the pipeline. In this scenario, the compressor cannot process 4-byte data input in each cycle, but every three or more cycles. Therefore, we decided to give up this feature.

When we compare the algorithms of Zstd hardware and Snappy software in Tab 5.8, we notice that the Zstd hardware implementation is the same or better than the Snappy software in every aspect. Thus the compression ratio is better. On the other hand, the Zstd hardware also does better on throughput.

The memory-to-memory compression latency of a single file depends on the size of the file and the compression ratio of the file. For 400MB blocks, as you can see from Tab 5.5, the latency is about 0.55 second. In practice, it should be able to lower the latency by reducing the size of each block. However, the throughput will also become smaller for small blocks.

6 Conclusions and future work

Throughput of transferring data from exchanges on another continent to Amsterdam is gradually becoming a bottleneck of the trading system in Optiver, due to the continued increase in the amount of trading data in exchanges. Using FPGA to compress the data in these exchanges is a possible solution, but there is no hardware compressor designed for trading data currently.

In this project, we benchmark and compare the compression ratio and compression throughput of commonly used compression algorithms for trading data, and find out the most suitable one.

Then we explore the details of the compression algorithm, fit it into FPGA and achieve a tradeoff between high throughput and low compression ratio for trading data. In this chapter, we will summarize the contribution and research result of this thesis project. Recommendations to future work of this project is also given.

6.1. Contribution and result

Addressing Research Question 1

In this project, we research three different compression algorithms, which are Gzip, Zstd and Snappy. We benchmark all three algorithms for the trading data. Snappy achieves the highest single-core throughput of about 600MB/s, which is more than two times faster than Zstd and more than 6 times faster than Gzip. However, the compression ratio of Snappy is the worst. Data-1 compressed by Snappy is about 50% larger than Zstd and 20% larger than Gzip. From our test, Zstd performs better than Gzip in both metrics on our used datasets. Based on the comparison between Snappy and Zstd, we believe that Zstd makes a better tradeoff between compression ratio and throughput in this project. Therefore, we decide to implement Zstd in this project.

Addressing Research Question 2

Based on the decision to use Zstd, a study on the format and compression algorithms of Zstd is performed. From our analysis, a good compression ratio is from both the format and the algorithm. In the algorithm of Zstd, there is a special mechanism to find a match: try to look up the history with the same offset of the last match. This mechanism fits the trading data well and helps the software to find matches with low cost. The Zstd algorithm also proposes a special coding to encode the repeated offsets to less bits.

For the second stage, which is entropy encoding, Zstd encodes the matches by FSE coding rather than Huffman coding. In Huffman coding, each byte can be encoded into an integer number of bits. However, the optimal number of bits is usually a fraction if we want to minimize the entropy. Therefore, developers from Facebook use FSE in Zstd.

In order to increase the throughput of the algorithm without overloading the CPU, we implement the Zstd algorithm in hardware to be executed in an FPGA accelerator. The hash table algorithms in the software cannot be efficiently implemented in the hardware and the hardware cannot buffer megabytes of data like software. So we propose a variant of the hash table algorithm. In this variant, not only the address, but also 8-byte history, is stored in the hash table so that the lookup takes only one memory access and multiple bytes can be compared at the same time.

In order to increase the single hardware kernel throughput, the compressor contains four hash match engines, which can do four lookup tasks in parallel. In this scenario, the compressor can process a 4-byte input data in each cycle. Since the working frequency of the hardware kernel is 300MHz, the maximum throughput of each kernel is 1.2GB/s.

In the software, the transformation tables for the FSE encoding are dynamically generated. In this project, the profiling of the influence of dynamically generated FSE tables shows that the dynamical tables have almost no effect on the compression ratio. Static tables generated specifically for each type of trading data can almost achieve the same compression ratio, therefore we decide to generated the static tables in the software and use them directly in hardware.

Each kernel proposed in this project can compress up to 1.2GB of data in one second. In theory, ten kernels can be placed on the FPGA and thus the on-chip compression throughput can be 12GB/s. However, moving data between host memory and FPGA on-board memory is the bottleneck. The max throughput of the PCIe between host and FPGA can be only about 11GB/s and the scheduling of data introduces overhead and results in a reduction of the throughput.

Finally, the compression throughput can be close to 9 GB/s, close to the full speed of a state-of-art Xeon CPU. And it is close to but lower than the required 10GB/s. The latency of our implementation is about 0.55 second, which is five times smaller than the software compression solution since the hardware can outperform the software implementation in single-file compression and is within the one-second requirement. Even though the hardware algorithm is more straightforward than the software one and takes less memory, the compression ratio is still comparable to that of the software implementation due to the features of trading data. The hardware compression in this project is built on the Vitis platform from Xilinx. In this platform, we can add more kernels by simply changing the parameters on the Vitis software. We can also move the solution to an FPGA card with faster interface without changing the hardware code if the FPGA card is supported by the Vitis platform.

6.2. Future work

As mentioned in Section 4.2, the method to pick a match when multiple matches are found is fixed. However, from the analysis of the offset of optimal matches, we observe that some specific values' occurrence is higher than the other ones. As presented in Fig 3.8, the number of occurrence goes down when the length of the offset grows generally. However, there are some spikes at about 1050, 1300, and 1500, which means the match of specific offset has a higher possibility to be the optimal one. The other two kinds of trading data show similar features. Therefore, we believe it should be possible to design a smarter match-picking method by exploiting the offset feature.

In this project, we use the hash functions presented in the Zstd standard hardware, which involves the multiplicand of very large prime numbers. The implementation of these functions on FPGA requires using DSPs, which consume more power during running. Therefore, it should be possible to design a hardware-optimized hash function by using shifting and mapping, which can be implemented at a low cost on FPGA.

As introduced in Section 2.2, we can optionally append a checksum at the end of a compressed file to make sure the decompressed data is the same as the original one. From our understanding, implementing a hardware checksum generator on FPGA should bring almost no influence on the throughput and takes only a little resource. Since it is difficult to eliminate all bugs on hardware, having a hardware checksum can avoid the catastrophic consequences of potential compression mistakes.

As mentioned in Section 5.4, one of the reasons why the hardware's compression ratio is worse than the software is that we do not implement a mechanism to look up the last offset. It is maybe possible to create some heuristics to implement such a look up algorithm in hardware at a low cost.

The throughput of PCIe from host to FPGA is about 11GB/s according to our tests. Since the on-chip compression throughput is 12GB/s, it should be possible to schedule the data transfer and fully exploit the throughput of PCIe.

Currently, the bottleneck of the throughput is at the speed of PCIe. One way to further increase the throughput is to use the Ethernet port on the FPGA and place more compression kernels on the FPGA.

Bibliography

[1] Jasmin Ajanovic. “PCI express 3.0 overview”. In: *Proceedings of Hot Chip: A Symposium on High Performance Chips*. Vol. 69. 2009, p. 143.

[2] Jianyu Chen, Zaid Al-Ars, and H Peter Hofstee. “A matrix-multiply unit for posits in reconfigurable logic leveraging (open) CAPI”. In: *Proceedings of the Conference for Next Generation Arithmetic*. 2018, pp. 1–5.

[3] Yann Collet. *facebook/zstd*. URL: `https://github.com/facebook/zstd`. (accessed: 28.05.2020).

[4] Yann Collet. *Finite State Entropy - A new breed of entropy coder*. URL: `http://fastcompression.blogspot.com/2013/12/finite-state-entropy-new-breed-of.html`. (accessed: 28.05.2020).

[5] Wei Cui. “New LZW data compression algorithm and its FPGA implementation”. In: *Proc. 26th Picture Coding Symposium (PCS 2007)*. 2007.

[6] Jian Fang et al. “Refine and Recycle: A Method to Increase Decompression Parallelism”. In: *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Vol. 2160. IEEE. 2019, pp. 272–280.

[7] Jeremy Fowers et al. “A scalable high-bandwidth architecture for lossless compression on fpgas”. In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2015, pp. 52–59.

[8] Google. *google/snappy*. URL: `https://github.com/google/snappy`. (accessed: 9.06.2020).

[9] Laiq Hasan and Zaid Al-Ars. “An Overview of Hardware-based Acceleration of Biological Sequence Alignment”. In: *Computational Biology and Applied Bioinformatics*. InTech, 2011, pp. 187–202.

[10] J. Hoozemans et al. “VLIW-Based FPGA Computation Fabric with Streaming Memory Hierarchy for Medical Imaging Applications”. In: *Applied Reconfigurable Computing (ARC), Lecture Notes in Computer Science*. Springer, 2017. DOI: `https://doi.org/10.1007/978-3-319-56258-2_4`.

[11] Ernst Joachim Houtgast et al. “Hardware acceleration of BWA-MEM genomic short read mapping for longer read lengths”. In: *Computational Biology and Chemistry* 75 (2018), pp. 54–64. ISSN: 1476-9271. DOI: `https://doi.org/10.1016/j.compbiolchem.2018.03.024`. URL: `http://www.sciencedirect.com/science/article/pii/S1476927118301555`.

[12] Ernst Houtgast et al. “An FPGA-based systolic array to accelerate the BWA-MEM genomic mapping algorithm”. In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. Samos, Greece, 2015, pp. 221–227. DOI: `10.1109/SAMOS.2015.7363679`. URL: `https://ieeexplore.ieee.org/abstract/document/7363679`.

[13] David A Huffman. “A method for the construction of minimum-redundancy codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[14] Santosh Kumar and Sonam Rai. “Survey on Transport Layer Protocols: TCP & UDP”. In: *International Journal of Computer Applications* 46.7 (2012), pp. 20–25.

[15] Matt Mahoney. *Silesia Open Source Compression Benchmark*. URL: `http://mattmahoney.net/dc/silesia.html`. (accessed: 10.06.2020).

[16] Microsoft. *Hardware innovation for data growth challenges at cloud-scale*. URL: `https://azure.microsoft.com/en-us/blog/hardware-innovation-for-data-growth-challenges-at-cloud-scale/`. (accessed: 29.05.2020).

[17] Angela Orebaugh, Gilbert Ramirez, and Jay Beale. *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006.

[18] Savan Oswal, Anjali Singh, and Kirthi Kumari. “Deflate compression algorithm”. In: *International Journal of Engineering Research and General Science* 4.1 (2016), pp. 430–436.

[19] Himali Patel et al. “Survey of lossless data compression algorithms”. In: *International Journal of Engineering Research and Technology* 4.04 (2015).

[20] Johan Peltenburg et al. “Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow”. In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2019, pp. 270–277.

[21] Johan Peltenburg et al. “Maximizing systolic array efficiency to accelerate the PairHMM Forward Algorithm”. In: *IEEE International Conference on Bioinformatics and Biomedicine*. Shenzhen, China, 2016, pp. 758–762. DOI: `10.1109/BIBM.2016.7822616`. URL: `https://ieeexplore.ieee.org/abstract/document/7822616`.

[22] Johan Peltenburg et al. “Supporting Columnar In-memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow”. In: *Applied Reconfigurable Computing (ARC), Lecture Notes in Computer Science*. Springer, 2019. DOI: `https://doi.org/10.1007/978-3-030-17227-5_3`.

[23] Yang Qiao. “An fpga-based snappy decompressor-filter”. In: *Master’s thesis* (2018).

[24] Priyanka Raichand. “A short survey of data compression techniques for column oriented databases”. In: *Journal of Global Research in Computer Science* 4.7 (2013), pp. 43–46.

[25] Suzanne Rigler, William Bishop, and Andrew Kennings. “FPGA-based lossless data compression using Huffman and LZ77 algorithms”. In: *2007 Canadian conference on electrical and computer engineering*. IEEE. 2007, pp. 1235–1238.

[26] Kyomin Sohn et al. “A 1.2 V 30 nm 3.2 Gb/s/pin 4 Gb DDR4 SDRAM with dual-error detection and PVT-tolerant data-fetch scheme”. In: *IEEE journal of solid-state circuits* 48.1 (2012), pp. 168–177.

[27] James A Storer and Thomas G Szymanski. “Data compression via textual substitution”. In: *Journal of the ACM (JACM)* 29.4 (1982), pp. 928–951.

[28] Ian H Witten, Radford M Neal, and John G Cleary. “Arithmetic coding for data compression”. In: *Communications of the ACM* 30.6 (1987), pp. 520–540.

[29] Xilinx. *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*. URL: `https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf`. (accessed: 28.05.2020).

[30] Jacob Ziv and Abraham Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343.