2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-3577-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPSW52791.2021.00035

FPGA Acceleration of Zstd Compression Algorithm

[1,2]Jianyu Chen [2]Maurice Daverveldt [1]Zaid Al-Ars
[1]Accelerated Big Data Systems, Delft University of Technology, Delft, The Netherlands
[2]Optiver, Amsterdam, The Netherlands

***Abstract*—With the continued increase in the amount of big data generated and stored in various application domains, such as high-frequency trading, compression techniques are becoming ever more important to reduce the requirements on communication bandwidth and storage capacity. Zstandard (Zstd) is emerging as an important compression algorithm for big data sets capable of achieving a good compression ratio but with a higher speed than comparable algorithms. In this paper, we introduce the architecture of a new hardware compression kernel for Zstd that allows the algorithm to be used for real-time compression of big data streams. In addition, we optimize the proposed architecture for the specific use case of streaming high-frequency trading data. The optimized kernel is implemented on a Xilinx Alveo U200 board. Our optimized implementation allows us to fit ten kernel blocks on one board, which is able to achieve a compression throughput of about 8.6GB/s and compression ratio of about 23.6%. The hardware implementation is open source and publicly available at https://github.com/ChenJianyunp/Hardware-Zstd-Compression-Unit.**

I. Introduction

Many big data applications face the challenge of managing the immense amount data needed for their operations. Particularly for applications where large amounts of data need to be transferred in real-time to a remote storage location, bandwidth limitations represent a serious bottleneck. This challenge can be mitigated by compressing the data at the source location, thereby reducing the bandwidth requirements.

In some big data applications, such as high-frequency trading (HFT), it is more cost effective to increase the compute capacity of a single server than use multiple servers. HFT data is collected at international stock exchanges and transferred to remote locations across the globe. This introduces the additional limitation of the excessively high cost of co-locating compute services at the exchanges. Therefore, increasing the amount of compute capacity per co-located server provides a more cost-effective solution than increasing the number of servers. This makes using hardware accelerators such as FPGAs an interesting alternative for offloading the data compression task as well as other compute intensive tasks from the processor.

There has been much research investigating the acceleration of compression algorithms on FPGAs. [1]–[3] propose hardware implementations of the Gzip algorithm, which consists of LZ77 compression combined with Huffman coding. [4] introduces a hardware implementation of the LZW algorithm, which is based on dictionary compression rather than LZ77. However, these classical compression algorithms are not very suited for compressing big data sets. Some accelerated solutions targeting big data applications focus on decompression rather than compression of streaming data [5]–[7]. Some major technology providers also have their compression acceleration solutions. Microsoft, for example, has their own hardware compression architecture and implementation for Zlib and Gzip [8]. These solutions are proprietary and can only be accessed in the cloud. Xilinx also provides an open-source library for data compression and decompression for Zlib and Snappy algorithms. However, these kernels are relatively slow, where the compression speed of a single-engine kernel is limited to about 300MB/s. IBM [9] provides an embedded Zlib compression accelerator on-chip on the state-of-art Power9 processor rather than an external accelerator, which makes it only accessible for Power9 users.

These existing solutions are tuned for general purpose compression and do not implement more recent compression algorithms specifically designed for big data applications, such as Zstd [10]. Our research shows that Zstd is one of the most suitable compression algorithms for compression of big data sets, such as HFT data, and can benefit from FPGA acceleration to increase real-time throughput and increase the cost efficiency of the implementation.

This paper makes the following contribution:

1) An efficient HW architecture for the Zstd algorithm.
2) Implementing the Zstd compression format, with new features like FSE (finite state entropy) encoding on HW.
3) HW optimization for real-time compression of HFT data.

In this paper, we first introduce the Zstd format in Section II and discuss a number of new features it provides. Section III discusses the proposed hardware architecture of the algorithm along with various implemented innovations. Section IV optimizes the hardware implementation for the HFT data characteristics. In Section V, we compare the compression ratio and throughput of the hardware implementation with the software implementation of Snappy and Gzip. Finally, we conclude the paper in Section VI.

II. Zstd compression algorithm

Like most Deflate compression algorithms, Zstd consists of two stages, that can be executed independently. The first stage is the LZ77 compression stage and the second stage is the entropy encoding stage. Zstd also provides a special mode to use dictionary compression as the first stage for small data. This mode is not discussed in this paper since we focus on big data application domains. Particularly for HFT data, the amount of data generated is usually in the range of gigabytes per second. The compression output of the first stage is organized in two streams of data: literal stream and sequence stream. The literal stream is the data that cannot be matched to the sequences already identified and stored in the history buffer, while the sequence stream is the set of data consisting of matches found within the history buffer. Each matching result is represented using three numbers: offset of the match, length of the match, and the number of literal bytes preceding the match. Zstd uses a special method to encode the repeated offset values. In this method, if the offset of the sequence is the same as one of the last three sequences, the offset will be encoded into some special values which take fewer bits. The normal way to find matches is to use hash tables. Therefore the size of hash tables is related to the compression ratio. In the second stage, the literal stream and sequence stream are compressed using Huffman coding and FSE coding, respectively. However, the compression ratio of Huffman coding highly depends on the distribution of symbols, which means the size of the compressed data is not always smaller than the original. Therefore, the Zstd compression software first tries to compress the literal data, then checks whether the compressed data is smaller. If not, the literal bytes will be stored directly instead of Huffman streaming. Therefore, we only use FSE coding in our accelerator and do not implement the Huffman coding algorithm.

III. The proposed architecture

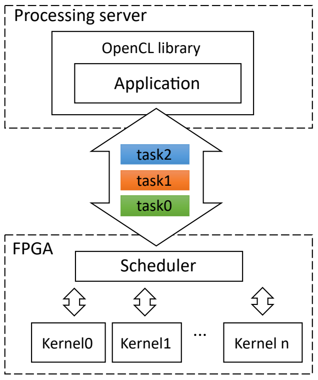

Fig. 1. System overview of the Zstd accelerator solution

Fig. 1 shows an overview of the system implementing the Zstd accelerator. The accelerator is implemented using the Xilinx Vitis platform running on the processing server. The host software is designed with the OpenCL library provided by Xilinx. During the compression, the application first passes the raw data and some other necessary compression information to the FPGA. To increase throughput, multiple compression tasks are executed at the same time. The scheduler on the FPGA distributes the tasks to multiple compression kernels. After the compression is performed, the compressed data is transferred back to the processing server. The application on the processing server packages the compressed data into the Zstd format. The communication protocol between server and FPGA is PCIe 3.0.

On the Vitis platform, the OpenCL library, the PCIe controller, the DMA and the scheduler are provided by Xilinx. This paper implements the software application and the hardware compression kernels. The application and the hardware kernels interact with the Vitis platform through the OpenCL API and the AXI interface, respectively.

In order to ensure a 4-byte per cycle throughput for a single kernel, we propose using 4 hash match engines in each compression kernel. Fig. 2 shows the general work distribution over the four hash match engines in each kernel. The insert operation inserts information of a string starting at 4 subsequent bytes to the hash tables. Each lookup operation searches for information of one string from each of the 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 in total, which means that the number of hash tables is quadratic with the number of concurrent input bytes.

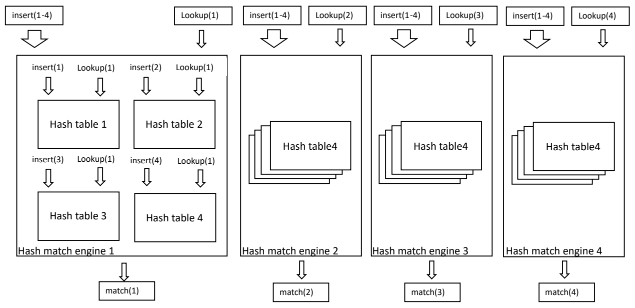

Fig. 2. Four hash match engines to process 4 insert and 4 lookup operations

IV. Optimization for HFT data

In this section, we optimize our implementation for HFT data. We use three example datasets data-1, data-2 and data-3 that represent data from 3 international stock exchanges. Each of these datasets is 400MB.

A. History size optimization

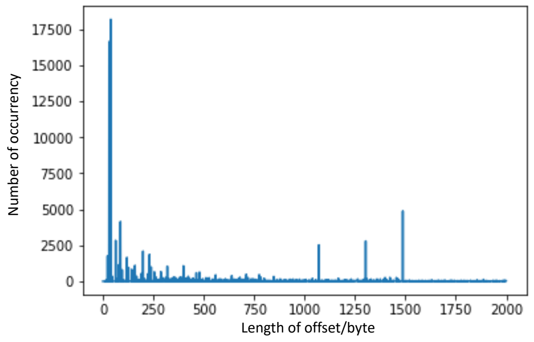

Fig. 3. Distribution of match offset for trading data

The size of buffered history is an important optimization parameter in Zstd software. A larger buffer is able to extend older matches, thus increase the compression ratio. But a larger buffer also consumes more RAM resources. To explore the suitable size of the history buffer, we analyze the statistics of the offset of matches. As presented in Fig. 3, most of the offsets are shorter than 750 bytes. With the increase of length, the occurrence of offsets decreases. There are three spikes identified at the length of about 1100, 1300 and 1550 bytes. Therefore, the size of buffer should be larger than 1550 bytes.

In this paper, we implement the history buffer using 2 Ultra RAMs (URAMs), the storage of which is 64KB.

B. Static FSE hash tables

	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 I
COMPRESSION RATIO USING DYNAMIC AND STATIC FSE TABLES

The generation of dynamic FSE tables consists of several steps: calculate the possibility of every symbol, normalize the possibility, and generate the transformation tables [10]. The computation in these steps is highly sequential in nature and thus is not suitable for FPGA implementation. To measure the overhead on the compression ratio of using static tables compared with using dynamic tables, we first add code to the software implementation to output the tables of all blocks when compressed. Then we use the tables of one block and force the software to use the pre-defined tables and record the compression ratio. As listed in Tab. I, the overhead of using static FSE tables is small. Therefore, we conclude that the method of using static FSE tables is more suitable for trading datasets targeted in this work.

C. Size of hash tables and hash functions

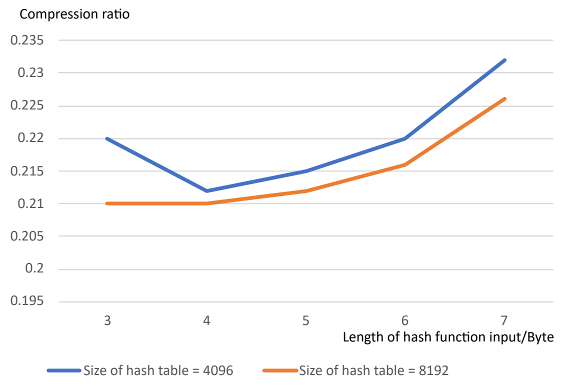

Fig. 4. Compression ratio when using different hash functions

In this work, we use the same hash functions as the Zstd official software. The Zstd software includes five different hash functions, of which the length of input data ranges from three bytes to seven bytes. Since the matches in Zstd should be at least three bytes, it is not suitable to use less than three bytes to calculate the hash value. Using different hash functions influences the compression ratio. On the other hand, the optimal hash function also varies when using hash tables of different sizes. In this work, we explore the compression ratio of different hash functions using two different sizes of hash tables. The results are presented in Fig. 4. The size of hash table shown in the figure indicates the number of entries in each hash table. The figure shows that when using 4096-entry hash tables, the 4-byte hash function achieves the best compression ratio. The compression ratio of 8192-entry hash tables outperform that of 4096-entry under every hash function input length. However, the improvement brought by 8192-entry is small, while the RAM resources used by 8192-entry hash tables is doubled. Therefore, we decide to use 4-byte hash function and 4096-entry hash tables.

V. EVALUATION

A. Experimental setup

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 hyper-threaded cores, clocked at 2.9 GHz, giving a total of 16 cores (32 threads) in the system. The FPGA used in this paper is an Alveo U200. The interface between the host machine and the FPGA is PCIe 3.0 x 16. The host-to-FPGA and FPGA-to-host throughput is 11.4GB/s and 12.1GB/s, respectively. The throughput mentioned in this section 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.

B. Resource utilization and throughput

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

TABLE II
RESOURCE UTILIZATION OF A SINGLE ZSTD HW KERNEL

The resource utilization of a single kernel is listed in Tab. II. The critical resource is the BRAM, which takes 4.77% of all BRAM resources. In this work, we place up to 10 kernels on the FPGA, with total resource utilization of about 47.7%. The total throughput of these 10 kernels can be up to 12GB/s in theory. However, the compression throughput is lower in practice. The throughput of the different kernels is shown in Fig. 5. 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. This is caused by device memory bandwidth bottlenecks. These 10 kernels use only 50% of the FPGA resources, leaving enough area to implement further data processing logic if needed.

C. Throughput comparison

Fig. 6 shows the throughput of Gzip, Zstd software, Snappy and Zstd hardware. In this work, the version of the compression software used is Zstd 1.4.2, Gzip 1.5, and Snappy 1.1.8. In Zstd and Gzip software, we can set the compression level in the

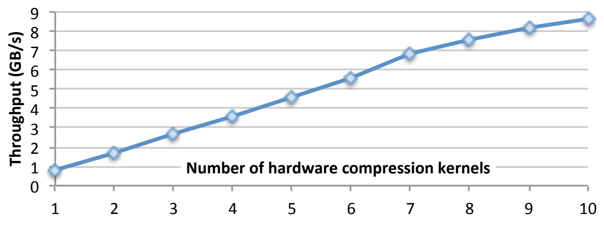

Fig. 5. Compression throughput vs number of hardware kernels

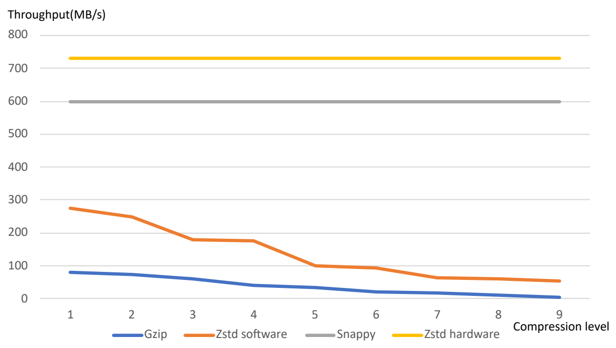

Fig. 6. Comparison of single-thread throughput of Gzip, Zstd software, Snappy and Zstd single-kernel hardware

application. The higher the compression level is, the smaller the compressed file becomes. The default levels are 6 for Gzip and 3 for Zstd. Snappy and the Zstd hardware have no compression level setting, thus the throughput of all compression levels is the same. The benchmark we use is a dataset of 400MB and the tests are single-threaded or single-kernel. The figure shows that a single Zstd hardware kernel achieves the highest throughput compared to a single software thread, which is about 4x larger than Zstd level-3 and 18x larger than Gzip level-6. Running the Zstd level-3 software fully multi-threaded on the server gives a throughput of about 3.4GB/s, which means fully utilizing the FPGA achieves about 2.5x acceleration.

D. Compression ratio

	data-1	data-2	data-3
Zstd software-L3	0.174	0.149	0.232
Zstd hardware	0.236	0.181	0.269
Snappy software	0.279	0.212	0.313

TABLE III
COMPRESSION RATIO AMONG ZSTD SOFTWARE, ZSTD HARDWARE AND SNAPPY SOFTWARE

The compression ratio of the 400MB data of all three types of trading datasets using Zstd software, Zstd hardware, and Snappy software are presented in Tab. III. The compression ratio of the hardware is higher than the Zstd software and lower than the Snappy software.

From our analysis, there are several reasons why Zstd hardware cannot achieve the same compression ratio as the software:

1) Smaller hash table and history buffer
2) Static FSE tables and no Huffman encoding
3) No lookups from the last offset

Zstd software uses a special mechanism to find matches: buffer the offset of the last match and try to look back at the same offset to find a match before lookup from the hash tables. This method is not implemented in hardware since we do not find an efficient way to implement it with low overhead. Fig. 4 shows that increasing the size of the hash tables improves the compression ratio only a little.

VI. Conclusion

In this paper, we propose a high throughput hardware architecture to accelerate the Zstd algorithm. This allows for real-time compression of data streaming in big data applications. The paper adapts various parts of the algorithm to allow for an efficient hardware design. The paper also discusses how to optimize the algorithm specifically for HFT applications. Experimental results show that our implementation is able to achieve a throughout of up to 8.6GB/s and a compression ratio of 23.6%, with only 50% FPGA utilization. The throughput of a single FPGA core is about 4x larger than single thread Zstd software, while 10 cores are about 2.5x larger than the fully multi-threaded Zstd software for comparable compression ratios. The hardware implementation is open source and publicly available.

References

[1] S. Rigler, W. Bishop, and A. 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.

[2] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng, "Fpga implementation of gzip compression and decompression for idc services," in *2010 International Conference on Field-Programmable Technology*. IEEE, 2010, pp. 265–268.

[3] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on fpgas using opencl," in *Proceedings of the international workshop on openCL 2013 & 2014*, 2014, pp. 1–9.

[4] W. Cui, "New lzw data compression algorithm and its fpga implementation," in *Proc. 26th Picture Coding Symposium (PCS 2007)*, 2007.

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

[6] J. Fang, J. Chen, J. Lee, Z. Al-Ars, and H. P. Hofstee, "A fine-grained parallel snappy decompressor for fpgas using a relaxed execution model," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 335–335.

[7] J. Fang, J. Chen, J. Lee, Z. Al-Ars, and H. P. Hofstee, "An efficient high-throughput lz77-based decompressor in reconfigurable logic," *Journal of Signal Processing Systems*, vol. 92, no. 9, pp. 931–947, Sep 2020. [Online]. Available: https://doi.org/10.1007/s11265-020-01547-w

[8] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "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.

[9] B. Abali, B. Blaner, J. Reilly, M. Klein, A. Mishra, C. B. Agricola, B. Sendir, A. Buyuktosunoglu, C. Jacobi, W. J. Starke *et al.*, "Data compression accelerator on ibm power9 and z15 processors."

[10] Y. Collet. Zstd github repository from facebook. [Online]. Available: https://github.com/facebook/zstd