A Self-aware Data Compression System on FPGA in Hadoop

Yubin Li[1], Yuliang Sun[1], Guohao Dai[1], Yuzhi Wang[1], Jiacai Ni[2], Yu Wang[1], Guoliang Li[2], Huazhong Yang[1]
[1]Dept. E.E., [2]Dept. C.S., Tsinghua National Laboratory for Information Science and Technology,
Tsinghua University, Beijing, China {yu-wang, liguoliang, yanghz}@tsinghua.edu.cn
{liyb13, sunyuliang13, dgh14, yz-wang12, njc10}@mails.tsinghua.edu.cn

***Abstract*—With the exponential growth of data size, data storage and analysis have been exposed to more challenges due to the lack of disk capacity and the limited network bandwidth. Data compression technique provides a good solution to mitigate these effects. In this paper, we propose a self-aware data compression system on FPGA for typical data warehousing, such as Hive, with column stored data and multi-threading requirements. The hardware accelerators can change the degree and hierarchy of parallelism depending on the data to be compressed (during the runtime). We test the system performance on a Xilinx VC707 FPGA board and the experimental results show that, up to 16 3-parallelism accelerators can be implemented and the throughput could be improved up to 432 MB/s. It is 6.25X speedup compared with the software solution under the same number of threads.**

I. Introduction

The fast development of information and communication technology has created an exponential growth in the quantity of data to be stored on hard disk and transmitted over the network. In the big data environment, Apache Hadoop [1] has been widely used in conjunction with Hive [2] to store and process extremely large data sets on commodity hardware. Hive supports deflate [3] compression in the procedure of data loading into Hadoop. Deflate consists of LZ77 [4] and Huffman [5] algorithms. Both LZ77 and Huffman are the representative lossless compression algorithms. LZ77 is an important dictionary based algorithm, which achieves compression by replacing recurring data with a reference to their previous occurrences, while Huffman encodes LZ77 results with variable-length code. Gzip [6] and Zlib [7] codecs are typically used in Hive for data compression, and both use the deflate algorithm.

In the deflate compression software solution, a large amount of CPU and memory resources are consumed. A lot of work have been done on the design of deflate compression hardware accelerators [8] [9] [10] [11] [12] [13]. These work improve data compression throughput in different extent by exploiting the parallelism potential of hardware. The compression throughput in the software solution (single thread) is bound to 15∼20MB/s, while the hardware accelerator achieves 200∼300MB/s (on average 16X speedup compared with software solution) [12].

This work was supported by Huawei Innovation Research Program, 973 project 2013CB329000, National Science and Technology Major Project (2011ZX03003-003-01), National Natural Science Foundation of China (No.61373026), and Tsinghua University Initiative Scientific Research Program.

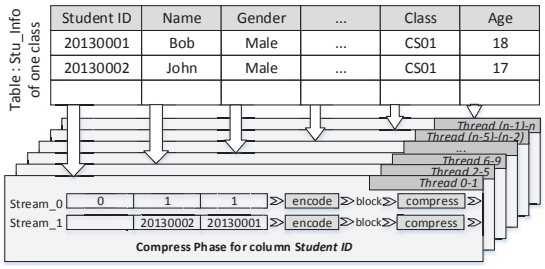

Fig. 1. Data compression in a column manner in Hive. Each column is processed by a thread.

Fig. 1 shows an example of the column-compression manner in Hive. One column of data generates different number of streams depending on the column data type. It is 2 for *int* and 4 for *string*. Data in each stream will be transformed into byte-string and be accumulated to generate one block if the size reaches 256 KB. Each block will be compressed by one compressor and results are written into Hive in the Optimized Row Columnar (ORC) File Format [14]. It is noted that each column data contributes to different software (hardware) compression performance, and Hive tends to use as much threads to compress multiple blocks simultaneously. However, most proposed hardware accelerators keep architecture unchanged to compress all the input data, and a lot of existing work can not afford sufficient parallelism required by Hive. Though previous work [9] [10] can achieve up to ∼3GB/s of the throughput, they could not afford more than 3 threads on one FPGA chip.

In this paper, we present a software-hardware co-designed deflate compression system, whose task-parallelism and data-parallelism can be adaptively adjusted according to different data intrinsic properties. The main contributions of this paper are as follows:

1) We design a parallelism-parameterized hardware accelerator that can be implemented on FPGA. We use the accelerator to evaluate the influence on compression performance brought by different contents and different hardware parallelism.

2) We propose a self-aware data compression system, whose task-parallelism and data-parallelism can be adaptively adjusted according to different input data. The proposed hardware accelerator consumes less resources, such that we can implement up to 16 accelerators on one Xilinx VC7VX485T

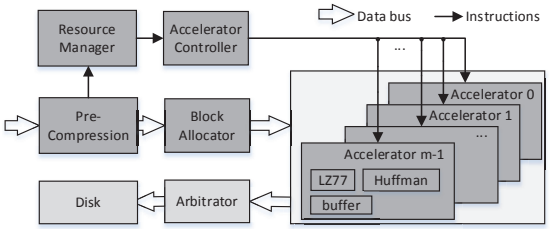

Fig. 2. The overall architecture of the system

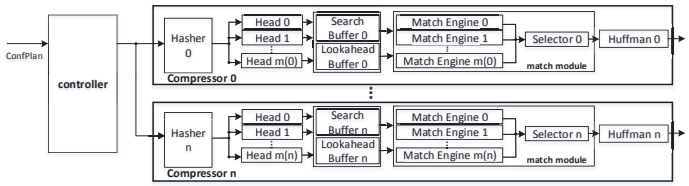

Fig. 3. The architecture of the hardware system on FPGA

chip.

3) We implement our hardware design on a Xilinx VC707 FPGA board, and achieve up to 6.25X and 1.7X speedup compared with existing software and hardware [12] solutions respectively.

II. Related Work

Most recent work [8] [9] [10] [11] [12] [13] use systolic arrays. The hash value is organized in a linked list, and the input sequence finds its candidate matches one by one. Systolic arrays is typically used in recent LZ77 implementations. Recent work focuses on the improvement of one thread's throughput. IBM researchers [9] report the best compression results on FPGA up to now, and Mohamed S. Abdelfattah [10] realizes this work using OpenCL. They make use of a heavily-pipelined custom hardware implementation to achieve a high throughput of 3 GB/s. Based on these work, Jeremy Fowers [13] achieves 5.6 GB/s by scaling up to 32 bytes/cycle in a single engine. In these work, the large resource consumption limits the task-parallelism.

Joo-Young Kim [12] proposes a scalable multi-engine Xpress9 compressor with asynchronous data transfer. They use various IP blocks to implement full Xpress9 compression algorithm. The system architecture supports up to 7 engines working simultaneously. They achieve the throughput of 200 $\sim$ 300 MB/s with 7 compressor engines.

Column-compression and multi-threading requirements in Hive lead to the inefficiency of these existing work. Firstly, these work use a same architecture despite the data sets are different. Results in [15] shows a compression throughput comparison between software (CPU) and hardware (FPGA). The speedup changes obviously under different data sets. When the CPU bandwidth decreases from 84.695 MB/s to 9.46 MB/s, the speedup of hardware increases from 30.7% to 547.8%. Secondly, Hive tends to use as much threads to compress multiple blocks, while the most recent work focuses on one thread's throughput improvement. They can not implement more than 3 accelerators on one FPGA chip [9] [10].

III. Self-aware Data Compression System

A. System Overview and Self-awareness

The overall architecture of the proposed self-aware data compression system is shown in Fig. 2. The whole system is composed of a master processor and multiple accelerators. Each accelerator accelerates one data block compression. The master processor is the software core in the host computer. It receives the input data blocks and mines data intrinsic property (software compression ratio and throughput) of each block through a window-sized (32KB) pre-compression, to predict expected hardware speedup depending on different data-parallelism hardware accelerators. Resource Manager generates one data compression plan (*ConfPlan*), which manipulates the task-parallelism (number of accelerators configured to run simultaneously) and data-parallelism of each accelerator. According to the plan, the required accelerators are configured and its data block is allocated.

We call the proposed approach self-aware, as the system could adjust its architecture according to different workload and resource available. At first, the task-parallelism is determined according to the number of input data blocks and available resources on FPGA. Then the data-parallelism of each accelerator under different workload is traversed to build prior knowledge of hardware speedup. After that, the master processor can predict the hardware speedup for each input data block under different hardware accelerator configurations. Finally, the *ConfPlan* is generated. As one data block compression is finished, the performance will feedback to the master processor and make the prior knowledge more accurate.

B. Hardware Dynamic Adjustment Realization

The parallelism-adaptive adjustment refers to changing task-parallelism and data-parallelism depending on the input data blocks for higher resource efficiency. Since LZ77 is more time and space consuming than the static huffman tree, the adjustment focuses on the LZ77 architecture. In Fig. 3, the task-parallelism is n and data-parallelism of *compressor i* is *m(i)*. The upper bound of n is fixed after the hardware design. Some modules are left unused when the number of accelerators needed is lower than the upper bound. Parameter *m(i)* could be adjusted as follows.

Decomposition and Reorganization Method. Several replicates of each function module described can be synthesized on FPGA. The controller manages the communication of each module to construct the required compressors. This method makes the design very difficult while it brings great flexibility in the parallelism adjustment. Every module can be a component of every compressor, this may lead to poor route performance, higher resource consumption and lower frequency. Therefore, we propose the limited decomposition and reorganization method.

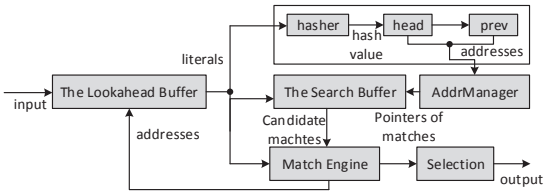

Fig. 4. LZ77 Architecture

Limited Decomposition and Reorganization Method. The system in Fig. 3 can be divided into two parts. The static part is LZ77 compressors with constant data-parallelism, while the dynamic part consists of several modules which can be connected with compressors in the static part to increase their data-parallelism. The dynamic part can be shared by all static compressors or some of them, depending on the design requirements. In this method, the lower bound of data-parallelism is depending on the static part, and the upper bound is limited by the available dynamic resources. It does not sacrifice great flexibility but guarantees good route performance and a high frequency.

C. Hardware Compressor Implementation

In this section, we present the hardware design of LZ77 and static huffman on FPGA. The hardware architecture of LZ77 is shown in Fig. 3.

1) LZ77 Implementation: Fig. 4 illustrates the LZ77 architecture. The *Hasher* computes the hash value *hv* of each input character, and inserts the results into hash tables, *head* and *prev*. Multiple hash values representing positions of candidate matches in the search buffer are returned if necessary. Then, *Match Engine* compares each candidate match with current sequence and *Selector* selects the longest match as the final match. A distance-length pointer pointing to the selected match will be returned.

Sliding Window. The sliding window consists of the lookahead buffer and the search buffer. The lookahead buffer stores the input data to be encoded currently. We make use of the on-chip registers to achieve a high bandwidth for data access. Whenever new data come, the oldest ones in the circular buffer are replaced. The search buffer stores the compressed data recently and returns the candidate matches for current encoding data. In order to acquire several candidate matches in one cycle, we implemented the search buffer in distributed and "interleave" way. There are *NumBuffer* distributed buffers. This approach can reduce the number of conflict accesses. Similar to the circular lookahead buffer, the new input data would replace the oldest ones. To further increase the access bandwidth and reduce conflict accesses, we load *ByteBatch* bytes at one position in the search buffer.

Hash Table. Hash function is used to locate candidate matches of input characters in the search buffer. We make a modification to the hash table to support returning several values in one cycle. In our design, we make several copies of *head* to generate a heads-prev linked list. The latest hash index is stored in the first *head*, while the seconde latest is stored in the second *head*, and so on. The final head is linked to the *prev*. When a new hash value *hv* is computed by *hasher*, its position is stored in the first *head* at the address of *hv*, while the hash index in the first *head* is loaded into the second *head*, the one in the second *head* is loaded into the third *head*, and so on.

AddrManager. Because of the properties of distribution and multiple bytes at one address in the designed search buffer, many continuous characters share the same address in the search buffer. However, the position recoded in the hash table is the position where the character appears in the input series. It is different from its address in the search buffer. We use *buffer_id* and *buffer_addr* to locate the candidate match in the search buffer. When one candidate position *addr* is returned by *hash table*, *buffer_id* and *buffer_addr* are computed as the following equation.

$$\begin{aligned} buffer\_id &= \frac{addr}{ByteBatch} \mod NumBuffer \\ buffer\_addr &= \lfloor \frac{addr}{ByteBatch} \div NumBuffer \rfloor \end{aligned} \tag{1}$$

Match Engine. Each candidate match from the search buffer is compared with the current sequence from the lookahead buffer in the match engine. And it computes the match length. After all candidate matches are compared, the distance-length pointer pointing to the longest match is returned. If the longest match is shorter than 3, the character in the head of current sequence is returned and the current sequence is processed form the next character.

2) Huffman Implementation: We implement the static huffman tree on FPGA. On one hand, static huffman tree need not to collect the frequencies of every symbol. The input stream can be encoded one by one sequentially. On the other hand, the static huffman tree need not to be included in the final compression results. This shrinks the benefits that may brought by dynamic huffman tree.

IV. Evaluation

A. Experimental Setup

In our experiments, we implement the hardware design on the Xilinx Virtex-7 XC7VX485T FPGA chip. The master processor is an Intel Core i7 with 2.4 GHz. We use Canterbury corpus [16] as the test suites. To achieve the dynamic adjustment of hardware architecture, we modify the parallelism parameter, and load different configure files into FPGA. Parameters *NumBuffer* and *ByteBatch* are 16 and 4 respectively in our experiment.

We compress the test suites on the master processor using software deflate algorithm. The deflate level is 6 as default. Table I shows the software compression performance, including compression ratio and throughput. Fig. 5(a) shows that, the data compression throughput and compression ratio have relative consistent trend.

B. Hardware Acceleration for One Workload

We implement the different data-parallelism accelerators on FPGA. Table II shows the resource consumption by different

TABLE I
THE COMPRESSION THROUGHPUT OF SOFTWARE SOLUTION AND HARDWARE ACCELERATORS

Text		alice29.txt	kennedy.xls	lcet10.txt	plrabn12.txt	ptt5	book1	book2	geo	obj2	pic	alphabet.txt	random.txt
Ratio		2.80	5.02	2.95	2.47	9.08	2.45	2.96	1.50	3.03	9.08	353.10	1.32
throughput (MB/s)	SW	4.64	24.39	15.44	7.84	17.90	7.02	8.65	2.86	5.48	27.84	19.53	6.10
	1-P	4.15	20.32	13.20	6.91	14.30	6.89	7.13	2.31	4.71	25.21	41.35	20.11
	2-P	7.71	37.35	24.71	12.98	25.86	12.77	13.95	4.53	9.15	42.13	41.35	36.46
	3-P	10.99	38.74	36.35	18.54	40.01	18.32	19.84	6.49	13.42	42.23	41.36	37.53
	4-P	15.03	38.85	37.41	25.78	40.01	24.00	27.01	8.29	17.39	42.24	41.36	38.00
	5-P	18.11	38.85	37.43	30.20	40.02	27.21	32.91	10.17	21.17	42.24	41.37	38.13
	6-P	23.25	38.84	37.43	31.31	40.02	27.23	32.92	11.99	24.00	42.25	41.37	38.14

TABLE II
THE RESOURCE CONSUMPTION OF HARDWARE ACCELERATORS

	1-P	2-P	3-P	4-P	5-P	6-P	Huffman
REGs	2623	2744	2876	3016	3164	3322	428
LUTs	8048	13061	18772	22877	27755	32863	1105

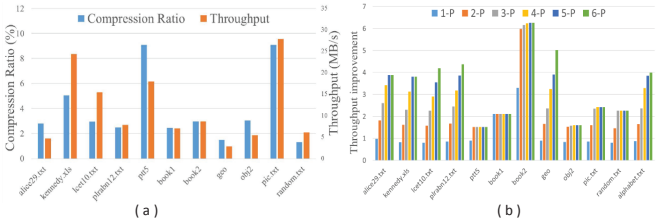

Fig. 5. (a)The relationship between software compression ratio and throughput.(b)The throughput improvement of different data-parallelism hardware accelerators compared to software.

data-parallelism hardware accelerators. Because the Huffman is independent with LZ77 accelerator, we separate the resource consumed by LZ77 and Huffman. Fig. 5(b) describes the throughput improvement of different data-parallelism hardware accelerators for different workload compared with the software solution. The growth rate of the speedup for different workloads varies dramatically as the data-parallelism increases. The throughput of *alice29.txt*, *kennedy.xls*, *lcet10.txt*, *plrabn12.txt* increases as the data-parallelism increases. For text *ptt5*, *book1*, *book2*, these is no obvious improvement on throughput when the data-parallelism is more than 3. These results show the strong correlation between compression performance and data property.

C. Hardware Acceleration for Multi-column Data Table

To evaluate the performance of the proposed system, we assume that a 16-column data table is required to be compressed and loaded into Hive. Each column is compressed by one accelerator. The limited resource in the VC707VX485T FPGA supports 16 3-parallelism accelerators. We compute the system throughput based on Table. I.

The average throughput for 3-parallelism accelerator is 27MB/s in Table. I, the whole system throughput can be achieved is 432MB/s. It should be noticed that this is not the highest throughput when compressing 16 blocks concurrently. It is 6.25X of the software solution with 16 threads and 1.7X of the average throughput achieved by work [12]. If we adjust the data-parallelism of each accelerator for different column, further improvement of the throughput can be obtained.

In practical applications, the data blocks are continuously compressed and loaded into Hive. We design the whole system dataflow into pipeline, i.e., the preprocessing for data blocks after the first batch blocks, could prepare the results before the accelerators become idle. Thus, the introduced software preprocessing (pre-compression and hardware resource management) methods bring little overhead to the system compression throughput.

V. CONCLUSION

In this paper we present a self-aware data compression system on FPGA in Hadoop. The system satisfies both column-compression and multi-threading requirements in Hive. Experimental results show that the accelerator performance has strong correlation with data property and accelerator architecture. Moreover, up to 16 3-parallelism accelerators can be implemented on one FPGA chip, and the throughput is up to 432 MB/s, 6.25X of the software solution with 16 threads and 1.7X of hardware solution [12].

REFERENCES

[1] "Apache hadoop," http://wiki.apache.org/hadoop.

[2] A. Thusoo *et al.*, "Hive-a petabyte scale data warehouse using hadoop," in *Data Engineering (ICDE), 2010*, pp. 996–1005.

[3] L. P. Deutsch, "Rfc 1951: Deflate compressed data format specification version 1.3, may 1996," *Status: INFORMATIONAL*.

[4] J. Ziv *et al.*, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory, 1997*, vol. 23, no. 3, pp. 337–343.

[5] D. A. Huffman *et al.*, "A method for the construction of minimum redundancy codes," *Proceedings of the IRE, 1952*, vol. 40, no. 9, pp. 1098–1101.

[6] L. P. Deutsch, "Gzip file format specification version 4.3," 1996.

[7] P. Deutsch and J.-L. Gailly, "Zlib compressed data format specification version 3.3," RFC 1950, May, Tech. Rep., 1996.

[8] E. Ghany *et al.*, "Design and implementation of fpga-based systolic array for lz data compression," in *Circuits and Systems, 2007*, pp. 3691–3695.

[9] A. Martin, D. Jamsek *et al.*, "Fpga-based application acceleration: Case study with gzip compression/decompression streaming engine," *ICCAD Special Session C, 2013*, vol. 7.

[10] M. S. Abdelfattah *et al.*, "Gzip on a chip: High performance lossless data compression on fpgas using opencl," in *Proceedings of the International Workshop on OpenCL 2013 & 2014*, p. 4.

[11] S. Rigler *et al.*, "Fpga-based lossless data compression using huffman and lz77 algorithms," in *CCECE 2007*, pp. 1235–1238.

[12] J. Y. Kim *et al.*, "A scalable multi-engine xpress9 compressor with asynchronous data transfer," in *FCCM 2014*, pp. 161–164.

[13] J. Fowers *et al.*, "A scalable high-bandwidth architecture for lossless compression on fpgas," in *FCCM 2015*.

[14] A. Floratou *et al.*, "Sql-on-hadoop: Full circle back to shared-nothing database architectures," *Proceedings of the VLDB Endowment, 2014*, vol. 7, no. 12, pp. 1295–1306.

[15] J. Ouyang *et al.*, "Fpga implementation of gzip compression and decompression for idc services," in *FPT, 2010*, pp. 265–268.

[16] "The canterbury corpus," http://corpus.canterbury.ac.nz/descriptions/, 2005.