A High-Bandwidth Snappy Decompressor in Reconfigurable Logic

Fang, Jian; Chen, Jianyu ; Al-Ars, Zaid; Hofstee, Peter; Hidders, Jan

DOI
10.1109/CODESISSS.2018.8525953

Publication date
2018

Document Version
Accepted author manuscript

Published in
2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)

Work-in-Progress: A High-Bandwidth Snappy Decompressor in Reconfigurable Logic

Jian Fang, Jianyu Chen, Zaid Al-Ars
Delft University of Technology
Delft, the Netherlands
{j.fang-1, z.al-ars}@tudelft.nl

Peter Hofstee
IBM Research & TU Delft
Austin, USA
h.p.hofstee@tudelft.nl

Jan Hidders
Vrije Universiteit Brussel
Brussels, Belgium
jan.hidders@vub.be

***Abstract*—While in-memory databases have largely removed I/O as a bottleneck for database operations, loading the data from storage into memory remains a significant limiter to end-to end performance. Snappy is a widely used compression algorithm in the Hadoop ecosystem and in database systems and is an option in often-used file formats such as Parquet and ORC. Compression reduces the amount of data that must be transferred from/to the storage saving both storage space and storage bandwidth. While it is easy for a CPU Snappy decompressor to keep up with the bandwidth of a hard disk drive, when moving to NVMe devices attached with high bandwidth connections such as PCIe Gen4 or OpenCAPI, the decompression speed in a CPU is insufficient. We propose an FPGA-based Snappy decompressor that can process multiple tokens in parallel and operates on each FPGA block ram independently. Read commands are recycled until the read data is valid dramatically reducing control complexity. One instance of our decompression engine takes 9% of the LUTs in the XCKU15P FPGA, and achieves up to 3GB/s (5GB/s) decompression rate from the input (output) side, about an order of magnitude faster than a CPU (single thread). Parquet allows for independent decompression of multiple pages and instantiating eight of these units on a XCKU15P FPGA can keep up with the highest performance interface bandwidths.**

***Index Terms*—Snappy, Decompression, FPGA**

I. Introduction

Due to the significant I/O bottleneck when database operations access data in storage, database systems have been moving to processing data in-memory. However, loading the data from persistent storage into main memory remains a bottleneck. Systems such as Netezza [4] have used compression algorithms for bandwidth amplification. In these kinds of systems, the raw data is compressed and stored in storage, and only when it needs to be loaded, is the data transferred and decompressed into memory. Although this architecture can save both storage space and storage bandwidth, prior designs do not keep up with current high-bandwidth connections such as PCIe Gen4 or OpenCAPI-attached NVMe [8].

Snappy [5] is an LZ77-based, byte-level (de)compression algorithm developed by Google and open-sourced in 2011. It has been widely used in big data platforms, especially in the Hadoop ecosystem, supporting big data formats such as Parquet [2] and ORC [1]. Similar to LZ77 [9], a Snappy compressed file consists of two kinds of tokens: literal tokens and copy tokens. A literal token contains data (uncompressed for Snappy) while a copy token specifies a copy of data from previously decompressed data. Snappy 64KB blocks are independently compressed.

Due to the dependency between adjacent tokens in Snappy [3], the general expectation is that there is little acceleration performance potential. Previous work [6] proposed a token-level parallel implementation on FPGAs that can process two tokens each cycle by doubling the token history for read conflict resolution and combining write operations for write conflict resolution. However, this method takes advantage of the parallelism of FPGA logic but not of the array structure in the FPGA, one of the key advantages of which is to allow parallel block ram (BRAM) access. In this paper, we propose an FPGA-based Snappy decompression accelerator that makes optimal use of BRAM-level parallelism. Our FPGA implementation translates a stream of tokens into independent read and write commands that access a single BRAM to gain maximum parallelism and efficiency within a single stream.

II. Decompressor Architecture

A. Architecture Overview

As shown in Fig. 1, the proposed decompressor consists of a slice parser (SP), an arbiter, multiple BRAM command parsers (BCPs), execution modules, recycle units, and the corresponding FIFOs and selector logic. The SP reads 16 bytes of data every cycle and parses it into a "slice" that contains the token information in these 16 bytes such as token start position, etc. After that, an arbiter is used to assign each slice to one of the second level parsers, the BCPs, in which a slice will be converted into one or multiple BRAM commands. There are two types of BRAM commands, write commands and copy commands. A write command indicates a data write operation on the BRAM while a copy command leads to a read operation followed by one or two write commands to place the data in the appropriate BRAM blocks. For each execution module, a read selector and a copy selector are used to select the BRAM commands from all BCPs. In the execution module, the BRAM commands are executed by performing BRAM read/write operations. As the BRAM can perform both a read and a write in the same cycle, each execution module can simultaneously process a write command and one copy (read) command at the same time. Since new write commands and new copy commands (if the read data is not ready) are generated after the copy command is executed, a recycle unit

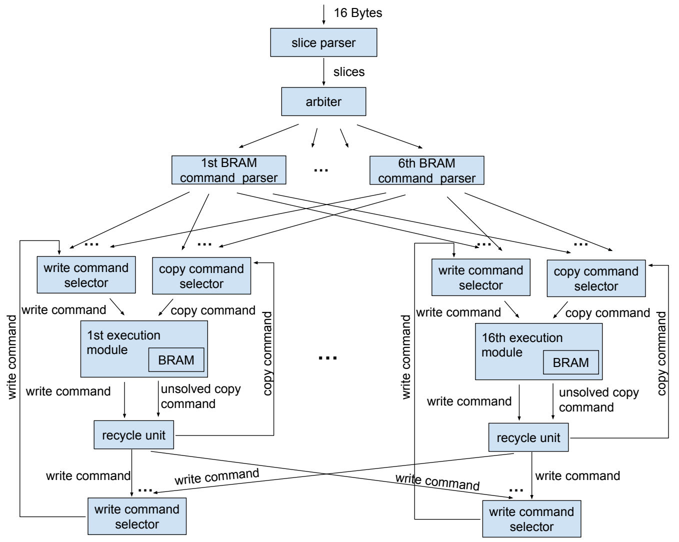

Fig. 1. Architecture Overview

is adopted to collect these commands and send them back for a new execution round. The approach of re-executing read commands that do not produce valid data is similar in spirit to the GPU-optimized algorithm in [7] and reduces control complexity.

B. Two-level Parser

We adopt a two-level parsing technique to split data into finer-grained commands to obtain high parallelism. In the first level parser, the slice parser reads 16 bytes of data and computes the number of tokens in these 16 byte data, as well as the token start position, token remainder counter, and the literal remainder flag, etc. Our proposed architecture adopts a second level parser to further split tokens into fine-grained BRAM commands to fully use the BRAM parallelism. This way, more BRAMs are active and more tokens can be processed in parallel.

C. Parallel BRAM Operations

The BCPs generate two kinds of BRAM commands, the write command and the copy command that write data to or copy data from only one BRAM. Since each BRAM has one independent read port and one independent write port, each BRAM can process one read command and one copy command each clock cycle. While the write command can always be processed successfully, the copy command can fail when the target data is not ready in the BRAM. So there should be a recycle mechanism for failed copy commands. After executing a copy command, there can be three kinds of results: 1) all the target data is ready (hit); 2) only part of the target data are ready (partial hit); 3) none of the target data is ready (miss). In the hit case and the partial hit case, one or two write commands are generated to write the copy results to one or two BRAMs. In the partial hit case and the miss case, a new copy command is generated and recycled, waiting for the next round of execution.

TABLE I
Implementation Results

Flip-Flop	LUTs	BRAMs	Frequency	Power
32K(3.32%)	46K(8.95%)	29(2.95%)	250MHz	2.9W

III. Experimental Results

We implemented a single instance of our proposed architecture in the XCKU15P FPGA. The implementation results are shown in Table I. A 250MHz implementation of the proposed architecture takes 3.32% of the Flip-Flops, 8.95% of the LUTs, and 2.95% of the BRAMs. The design uses the minimum number of BRAMs per block and is LUT-limited.

A preliminary experiment was run on decompressing the book "Alice's Adventures in Wonderland" [5], the compressed (uncompressed) size of which is 85KB(149KB). The proposed decompression accelerator achieves a processing rate about 13 Bytes/clock cycle, meaning an input (output) throughput of 3GB/s (5GB/s). Compared with a single-thread (similar resource) CPU implementation [5], our design is about an order of magnitude faster than a thread in Core i7 processor (500MB/s). Meanwhile, synthesis results indicate that eight instances only consume 13.6W of power, leading us to expect more than an order of magnitude power efficiency advantage at a chip level.

IV. Conclusions and Future Work

Based on our single-unit results we expect our design to have an order of magnitude better per-thread Snappy performance than a CPU, about an order of magnitude more chip-level throughput, and close to two orders of magnitude better power efficiency. Our future work targets on multiple instances with follow-up additional processing such as filtering that can keep up with the latest high-bandwidth interfaces. Our design will be made open source to enable further community improvement[1].

References

[1] Apache. Apache ORC. https://orc.apache.org/. Accessed: 2018-06-03.
[2] Apache. Apache Parquet. http://parquet.apache.org/. Accessed: 2018-06-03.
[3] J. Fang et al. Adopting opencapi for high bandwidth database accelerators. In *3rd International Workshop on Heterogeneous High-performance Reconfigurable Computing*, 2017.
[4] P. Francisco et al. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics. http://www.ibmbigdatahub.com/sites/default/files/document/redguide_2011.pdf, 2011. Accessed: 2018-06-03.
[5] Google. Snappy. https://github.com/google/snappy/. Accessed: 2018-06-03.
[6] Y. Qiao. An fpga-based snappy decompressor-filter. Master's thesis, Delft University of Technology, 2018.
[7] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross. Massively-parallel lossless data decompression. In *Proc. of the International Conference on Parallel Processing*, pages 242–247. IEEE, 2016.
[8] J. Stuecheli. A new standard for high performance memory, acceleration and networks. http://opencapi.org/2017/04/opencapi-new-standard-high-performance-memory-acceleration-networks/. Accessed: 2018-06-03.
[9] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on information theory*, 23(3):337–343, 1977.

[1]https://github.com/ChenJianyunp/FPGA-Snappy-Decompressor