LZ4 Compression Algorithm on FPGA

Matěj Bartík
CTU FIT & CESNET
matej.bartik@fit.cvut.cz

Sven Ubik
CESNET
ubik@cesnet.cz

Pavel Kubalík
CTU FIT
pavel.kubalik@fit.cvut.cz

***Abstract*—This paper describes analysis and implementation of a LZ4 compression algorithm. LZ4 is derived from a standard LZ77 compression algorithm and is focused on the compression and decompression speed. The LZ4 lossless compression algorithm was analyzed regarding its suitability for hardware implementation. The first step of this research is based on software implementation of LZ4 with regard to the future hardware implementation. As a second step, a simple hardware implementation of LZ4 is evaluated for bottlenecks in the original LZ4 code. Xilinx Virtex–6 and 7–Series FPGAs are used to obtain experimental results. These results are compared to the industry competitor.**

I. Introduction & Motivation

Fast lossless compression algorithms become more important than before, even though they do not reach compression ratios of their predecessors. The main usage of these algorithms is in reducing bandwidth reqirements, typically in multimedia applications, where bandwidth of a multimedia interface is slighly higher than of a transmission line. For example [1], it allows transmition of 12G-SDI (4K60p uncompressed video) over 10 Gbit Ethernet or Full HD video can be fit into a standard metalic gigabit ethernet. The following parameters are important for such a kind of an application:

- Universal compression for every data type (video, audio, service informations),
- High throughput for video,
- Low–Latency for real–time interaction,
- Little overhead for uncompressible data,
- Compression ratio is the least important parameter.

There are two algorithms which satisfy these requirements. The LZ4 compression algorithm, which is mentioned above and the LZO [2]. The LZ4 better fits our requirements [3]. Our research is focused on the LZ4 compression algorithm. LZ4 is based on LZ77 (Lempel – Ziv) [4] like other fast compression algorithms, because LZ77 is one of the few one–pass compression algorithm [5].

It has been shown that LZ77 can be used in an application, where throughput up to 9.17 Gbps is required [6], [7]. The authors also said, that their design can operate on higher frequencies, when pipelining or loop unrolling will be used. This will enable to reach 10G+ throughput.

However, this architecture is using extremely small search and look–ahead buffers (only a few bits wide), that make the architecture very inappropriate for a practically useful compression application.

A. Other Examples of LZ4 Usage

- Fast (de)compression of GNU/Linux kernel [8],
- A new use can be (de)compressing data between Solid–State Drive (SSD) for increasing throughput. There is a high probability, that SSD's vendors are prefering high throuhput/low latency algorithms based on LZ77.
- The LZ77 is also used for (de)compressing FPGA bitstreams [9].
- And also for the IP packet compression [10].

II. Teoretical background of the LZ4

It the following sections we describe main features of the LZ4 lossless compression algorithm [11], [12] when compared to the LZ77. The biggest advantages of the LZ4 are a hash based match search and the support of match overlaping.

A. LZ4 Lossless (De)compression Algorithm

LZ4 itself is not an algorithm in the original meaning. LZ4 only defines an output data format (like LZ77 [13]). This allows to create various derivates of compression methods (with different speeds and compression ratios) and also allows to decompress the LZ4 file format by a single tool, no matter what compression algorithm was used.

However, the author of LZ4 created a reference code in the C programming language and this code has been ported to many other programming languages. The LZ4 code contain various optimizations for different processor architectures to achieve maximum performance.

LZ4 as well as LZ77 is an asymmetric compression method, where decompression is much simplier (and faster) than compression. The decompression process is very similar to LZ77. It is based on copying literals from the decoded part.

B. Pseudocode of LZ4

A very simplified reference code expect the following parameters (byte oriented):

- I : An input data buffer
- O : An output data buffer
- Isize : Size of input buffer

```
pointer ip = 0; // address to I
pointer op = 0; // address to O
hash_table HT; // Zeroed
```

978-1-5090-0246-7/15/$31.00 ©2015 IEEE

```
while (ip < Isize-5) {
  h_adr = read U32 *ip, calculate hash;
  read possible match address HT(h_adr);
  store current address HT(h_adr)=ip;

  if !(match found) ||
     !(distance < offset_limit) ip++;
  else {
    if (ip > Isize-12) break;

    // writing to O buffer
    encode Token;
    encode Literals length;
    copy literals;
    encode Offset;
    encode Match length;

    increase input and output pointers;
  }
}

encode last literals;
return output pointer (data size);
```

III. LZ4 Analysis from the Hardware Designer View

In this section we discuss difficulties and advantages when implementing LZ4 compression algorithm in FPGA. The LZ4 implementation on the standard processor architecture benefits from high frequencies of CPU (Central Processing Unit) or RAM (Random Access Memory), instruction and data caches and software based optimization (software pipelining, loop unroling, usage of compiler/processor specific instructions) [14]. For this evaluation, revision r127 is used for analysis.

A. Hash Table and Hashing Algorithm

The first improvement of LZ4 againts LZ77 is the use of a hash table for storing reference addresses. LZ77 uses a searching algorithm for match detection with linear complexity. Due to performance reasons, LZ4 uses the least complex method for storing addresses, overwriting the previous address in the hash table. The hash table size is designed to fit the L1 Data Cache in the standard processor architecture.

One of possible further improvements is to implement the hash table as a regular cache with the limited degree of associativity extended with the LRU (Least Recently Used) algorithm. This may improve the compression ratio without significant performace loss.

The hash calculation algorithm is based on the Fibonacci hashing principle [15]. Read value is multiplacated with a constant 2654435761. This number is the closest Prime to $\frac{2^{32}}{\phi} = 2654435769$, where ϕ is the value of the Golden ratio. This 32-bit constant can be easily multiplicated by four DSP48 slices (and three only after place & route phase) available since Xilinx Virtex–5 chip generation in 6 clock periods. Generated IP core is pipelined. The result of multiplication is trimmed to the highest bits used for hash table address.

The biggest weakness of the LZ4 hashing mechanism is zeroing of the hash table. A larger RAM implemented by the on-chip BlockRAM or a distributed RAM does not have a feature for clearing the entire RAM content by reset. For the first run, we can benefit from the bitstream loading process, because the RAM content is part of the bitstream. For next runs it is necessary to clear the RAM content, for example by linear passage and clearing memory cell, one by one.

B. Match Search and Memory Access for Reference Addresses

The process of match search starts from reading a reference address (read from hash table). A 32-bit data are read from the reference address and from the input pointer address and then, these values are compared. When the difference between these two addresses is less than offset limit, a match is found. Otherwise, the input pointer is increased by one.

However, buffers are byte oriented and LZ4 also supports 64-bit computing. The memory subsystem of the LZ4 algorithm should support 8-bit, 32-bit and 64-bit (optional for maximum memory performance) access. This will result into a complex memory subsystem. Alternatively the memory subsystem can be 8-bit only and support of the two remainding modes can be solves by reading portions of 8-bit values. This also solves problems with unaligned memory access (caused by increment of input pointer by a one).

The biggest disadvantage of a simple 8-bit memory subsystem is a massive loss of performance. There is also no mechanism, that prevents match searching from reference addresses, that are equal to zeroes. Zeroed address means no write to a hash table cell and may be skipped.

C. LZ4 Sequence Encoding

The encoding of an LZ4 sequence requires linear passage through the input buffer byte after byte. However, the literals can be copied in up to 64-bit data blocks between the input and output buffer. There are same problems as we discussed in the previous section.

D. Size of Input and Output Buffers

The LZ4 algorithm adds only a tiny overhead for the uncompressible data. It is necessary to generate one block with all literals and with literal match. That is not a problem to alocate the output buffer slightly bigger (necessary size can be calculated with the formula $o_{size} = i_{size} + \frac{i_{size}}{255} + 16$) than the input one. For example the 16kB input buffer will require 273 byte long overhead (0.4% relative). But the FPGA has hard RAM blocks (BlockRAM) with the limited sizes and limited bus widths. The most important thing is that all BlockRAMs have the same size. There are same several ways to solve this:

- The input and the output buffer will have the same size and pretend that the problem does not exist (buffers are bigger than limits).
- Limit the size of input buffer.

- Combine BlockRAM and Distributed RAM to provide additional space with significant complexity increase of the address decoder.

E. Other Sources of Inefficiency

A standard computer architecture works with memories in a simple way with one read/write in one clock cycle. However, FPGA's BlockRAM has the possibility of using a Dual-Port BlockRAM to increase memory throughput by reducing the time to read longer data, for example. That means further analysis of the LZ4 reference code to create an optimalized code for the Dual-Port BlockRAM and its memory controller.

IV. Implementation of LZ4 Compression Algorithm

A. Principles of Implementation Platform

CESNETs MVTP–4K (Modular Video Transmision Platform – 4K) is an FPGA based system, that provides up to 8 input–output pairs of SDI (Serial Digital Interface) interfaces for transmitting 4K video content with audio. MVTP systems are communicating over optical version of 10G Ethernet. Uncompressed 4K transmission takes approximately 5 Gbps and a Full HD channel takes approximately 1.1 Gbps. The 4K video content is split into four quadrants and transmitted over four HD-SDI interface separately with synchronization. Each quadrant frame consist lines with video, audio and service data. The lengths of a line in each quadrant is less than the maximum size of an Ethernet jumbo frame payload, so one packet is created from each SDI line. This way of transmission allows for easy compensation of lost lines by duplicating the previous line. This property can be maintained with line-by-line compression. On the contrary, with inter-frame compression, the multimedia stream may disintegrate to the next synchronization frame. Next reason for lossless compression is to allow raw multimedia stream distribution between servers without loss of quality.

B. Assumptions for Implementation

- SDI signal is transmitted line by line. The length of one SDI line is slightly less than the maximum size of an IP packet payload in a jumbo Ethernet frame.
- We are looking for a block compression for IP packets, that is up to 9216 bytes. The closest power of 2 greater then this limit is 16kB. Therefore the size of input and output buffer will be 16kB.
- We reduce the hash table size from 4096 records (16 kB) to 1024 records (N = 10) and we reduce the size of buffer pointers (from 32–bit to 13–bit).
- The current system for video transmission is based on Xilinx Virtex-6 (XC6VLX240T-2FF1156), therefore design will be optimized for Virtex–6, but it can be easily transfered to Xilinx 7–Series FPGAs chips.

C. Implementation and Results

First, we created a simple design without any advanced features. The design is writen and tested in VHDL language. The goal was to evaluate how LZ4 code exactly works and to provide information about bottleneckes. The LZ4 code was divided into a datapath and its control. The control is realized as a Moore finite state machine, that represents lines of the original C code. The FPGA resource utilization includes buffering all input and output signals. The current resource utilization for multiple Xilinx FPGAs are in Table I.

The required frequency of MVTP system is 156.25 MHz, so the compression core meets requirements of the MVTP. We didn't measure the precise throughput/latency of the implemented LZ4 core yet, because it was designed for evaluation only (based on the reference LZ4 source code).

The implemented datapath (Fig. 1) contains all important blocks, but finite state machine used for control is very complex and slow. The datapath design is also reduced to support only 8–bit operations, therefore operations with wider operands has to be split. Even when the design was simplified, it was sufficient for evaluation and critical parts of LZ4 compression algorithm for hardware implementation were found. The critical path is a memory controller, where the address bus is created from cascaded multiplexers (multiple pointers to the input memory are required by the LZ4 algorithm) and combined with an adder (for offset support) in the last stage.

D. Comparison with Competitor

The Helion LZRW3 core [16] has been selected for a very simple comparison. Both algorithms (LZRW and LZ4) are derived from the LZ77. The Helion LZRW core is a highly optimised state of the art industry solution. But the current version of LZ4 (even unoptimised) is comparable to the LZRW core in resource utilization (usually slightly more LUTs are taken because of two additional bits in memory subsystem and we are using three DSP48 blocks). Even that, Virtex–6 implementation take less resources.

The maximum frequency of the LZ4 core is quite higher (with same speedgrades and synthesis technology, with exception of Artix–7 might be caused by limited routing capabilities in low–cost FPGAs) then LZRW and there is still a huge space for optimizations (software based pipelining of the reference source code caused duplication of all memory pointer registers).

TABLE I
Comparison with Helion LZRW3 Core [17]

Virtex–6 (XC6VLX75T-2FF784) Resource Utilization						
Solution	**Slices**	**LUTs**	**FFs**	**BRAMs[1]**	**DSP48s**	**Frequency**
LZ4	216	729	404	16 + 1[2]	3	224 MHz
Helion	225	770	N/A	4	0	198 MHz
Kintex–7 (XC7K70T-2FBG676) Resource Utilization						
Solution	**Slices**	**LUTs**	**FFs**	**BRAMs[1]**	**DSP48s**	**Frequency**
LZ4	266	891	441	16 + 1[2]	3	241 MHz
Helion	227	789	N/A	4	0	210 MHz
Artix–7 (XC7A100T-2FGG676) Resource Utilization						
Solution	**Slices**	**LUTs**	**FFs**	**BRAMs[1]**	**DSP48s**	**Frequency**
LZ4	243	764	375	16 + 1[2]	3	146 MHz
Helion	226	789	N/A	4	0	148 MHz

Note 1: LZ4 is using two 16 kB buffers for I/O, Helion is using 2 kB size.
Note 2: One BRAM is dedicated for the hash table for LZ4 design.

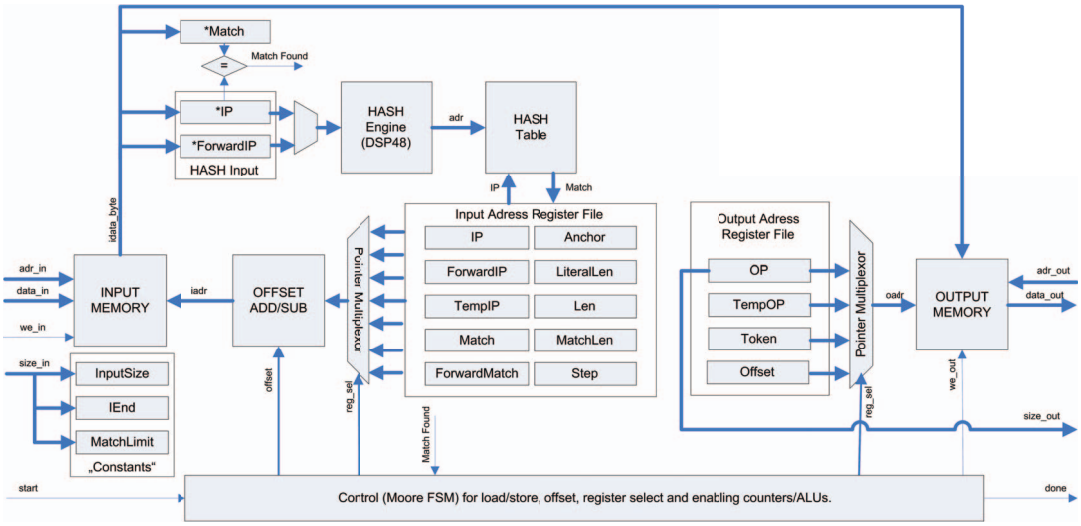

Fig. 1. Architecture of the LZ4 Lossless Compression Algorithm Design.

The LZRW core was synthetizied with 25% buffers size, so we can be expected 20% deterioration of all results, as mentioned in the documentation of the LZRW core. Parameter comparison is summarized in Table I.

V. Future Work

We plan to improve the LZ4 lossless compression algorithm speed in hardware. Possible improvements may be pipelining or an advanced memory subsystem. The impact of these improvements will be measured. We expect a significant improvement by porting 64–bit CPU specific optimizations into an FPGA. This should exceed a theoretical (throughput) limit of current hardware implementations. This limit can be calculated by multiplying a frequency (200 MHz) and an operand–width (8–bits), that results into 1.6 Gbps peak performance [18] per compression core. We also plan to create and compare HLS (High Level Synthesis) version of the LZ4 reference C source code.

VI. Conclusion

We have implemented The LZ4 lossless compression algorithm on FPGA in VHDL (and compared to the industry solution) from a reference C code (r127). Limitations and bottlenecks hashing table zeroing, software pipelining overhead, differences between architectures, etc.) of the LZ4 lossless compression algorithm have been found during the process of analysis and implementation.

Acknowledgment

This research has been partially supported by the project SGS15/020/OHK3/1T/18.

References

[1] Hafi, L. ”Mapping SDI with a Light-Weight Compression for High Frame Rates and Ultra–HD 4K Transport over SMPTE 2022–5/6” [Online]. Available: tinyurl.com/o3lxgcl

[2] Oberhumer, M.:LZO real-time data compression library [Online]. Available: http://www.oberhumer.com/opensource/lzo/

[3] Ruan Delgado Gomes, Yuri Gonzaga Gonalves da Costa, Lucenildo Lins Aquino Jnior, Manoel Gomes da Silva Neto, Alexandre Nbrega Duarte, and Guido Lemos de Souza Filho. 2013. A solution for transmitting and displaying UHD 3D raw videos using lossless compression. In Proceedings of the 19th Brazilian symposium on Multimedia and the web (WebMedia ’13). ACM, New York, NY, USA, 173-176.

[4] Rigler, S.; Bishop, W.; Kennings, A., ”FPGA-Based Lossless Data Compression using Huffman and LZ77 Algorithms,” Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on , vol., no., pp.1235,1238, 22-26 April 2007

[5] J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, vol. 23, no. 3, pp. 337343, 1977.

[6] Mehboob, R.; Khan, S.A.; Ahmed, Z.; Jamal, H.; Shahbaz, M., ”Multigig lossless data compression device,” Consumer Electronics, IEEE Transactions on , vol.56, no.3, pp.1927,1932, Aug. 2010

[7] El Ghany, M.A.A.; Salama, A.E.; Khalil, A.H., ”Design and Implementation of FPGA-based Systolic Array for LZ Data Compression,” Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on , vol., no., pp.3691,3695, 27-30 May 2007

[8] Lee, K. ”LZ4 Compression and Improving Boot Time” [Online]. Available: tinyurl.com/q3p4v2l

[9] Khu, A.: Xilinx FPGA Configuration Data Compression and Decompression. [Online]. Available: tinyurl.com/nhhgqbj

[10] Munteanu, D.; Williamson, C.; An FPGA-based Network Processor for IP Packet Compression. [Online]. Available: http://pages.cpsc.ucalgary.ca/ carey/papers/2005/FPGA.pdf

[11] Collet, Y.: RealTime Data Compression: Development blog on compression algorithms. [Online]. Available: tinyurl.com/qc9yve4

[12] Fiedler, O.: LZ-Family Data Compression Methods, Bachelor Thesis, 2014. [Online]. Available: http://tinyurl.com/opmd5sc

[13] Solomon, D.: Data Compression: The Complete Reference (Fourth ed.). 20007, Springer. ISBN 9781846286032.

[14] Kane, J.; Yang Q., ”Compression Speed Enhancements to LZO for Multi-core Systems,” Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on , vol., no., pp.108,115, 24-26 Oct. 2012

[15] Knuth, D. E.: The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998, ISBN 0-201-89685-0.

[16] Helion Technology Limited, LZRW Compression cores. [Online]. Available: http://www.heliontech.com/comp_lzrw.htm

[17] Xilinx Inc., LZRW1 & LZRW3 Lossless Data Compression (Helion) [Online]. Available: tinyurl.com/ngmtx25

[18] Naqvi, S.; Naqvi, R.; Riaz, R; Siddiqui, F. Optimized RTL design and implementation of LZW algorithm for high bandwidth applications [Online]. Available: http://pe.org.pl/articles/2011/4/68.pdf