Data Compression Algorithms in FPGAs

Gonçalo Ribeiro
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

***Abstract*—Data compression is increasingly important as more and more data is produced, transferred and stored on a daily basis. It allows to reduce storage costs, speed up data transfers over limited bandwidth, reduce network congestion and even improve energy efficiency in wireless devices.**

In this work we propose a stream-based LZ77 hardware accelerator that implements the most computationally intensive part of many compression algorithms. The stream-based interface of this IP makes it easy to integrate with other hardware or software components to implement different compression applications.

This IP was integrated into a hardware/software architecture that implements the widely used Deflate lossless compression algorithm. This architecture targets embedded computing systems using software and hardware components. The hardware component accelerates the LZ77 algorithm, while the software selects the best matches and codes them with static Huffman. Executing the software part on a generic processor allows for greater flexibility of the encoding part.

A prototype implementation targeting a Zynq device demonstrated that the hardware accelerator can process 123 MiB/s and can easily be scaled in order to enhance the compression ratio. Compared with a software-only implementation running on the Zynq ARM processor a speedup of 1.5 is achieved.

The proposed IP can also be used to implement a system that surpasses the throughput of Gzip running on a Intel Core i-5 computer by a factor greater than 2, for typical files.

I. Introduction

In the last decades data compression has received increasing attention as the rate at which data is produced continues to increase. In 2013 about 2 million terabytes were generated per day and it is estimated that by 2018 the global Internet traffic will be 50 TB/s.

Data compression can help transmitting and storing these large amounts of data. The most obvious application is reducing the size of stored data, which can benefit both big data centres and small devices such as mobile phones. Compressing the data allows for faster transmissions as it virtually increases the bandwidth of links, which can also reduce congestion. Wireless devices can also benefit in terms of battery life, as transmitting a single bit can use as much energy as one thousand cycles of an embedded processor [1]. Finally, compression allows to save and retrieve data faster from slow storage media, which allows for example an operating system kernel to load faster, resulting in improved boot times.

Unfortunately, compressing data is a computationally intensive task. Its CPU-bounded nature means that a system can spend most of its processor time with a single compression task, seriously degrading performance for other tasks. It also means that, on a certain system, compression is only as fast as the available CPU. For these reasons, using compression coprocessors is beneficial for systems where compression is a common task.

In this work we propose an architecture for a hardware accelerator implementing the LZ77 algorithm [2] and implement it using a Zynq FPGA. LZ77 is the base for many other compression algorithms, one of which is the widely used Deflate [3], which is found in formats such as Zip, Gzip, Zlib and PNG. Deflate compressed resources are prevalent on the web, where it is currently used by 70 % of the top ten million websites [4].

The following section briefly describes previous works implementing LZ77 accelerators. Section III provides a brief explanation of the LZ77 algorithm. Section IV describes the proposed hardware architecture, which is used in Section V to implement a full Gzip hardware/software compressor. Finally, Section VI shows performance results.

II. State of the Art

Multiple architectures for implementing LZ77 or Deflate on reconfigurable devices have been proposed in the last ten years. We categorise them into two types — hashless or hashful — according to whether they use hashing functions to aid the search for matches.

Hashless architectures compare all the data in the window with the one in the lookahead, trying to find matches. It has no a priori knowledge of where in the window the best matches for the current lookahead might be. Consequently it needs to compare the lookahead with all the dictionary's positions, making it impractical for "large" dictionaries. Architectures in this category are found in [5], [6], [7], [8]. The largest dictionary in this category — seen in [8] — is 4 kB in size, which is combined with a lookahead of 16 bytes. The maximum published throughput in this category is 1.1 GB/s.

Hashful architectures only compare the lookahead to portions of the window that are more likely to have a match. This is accomplished by applying some hash function to the lookahead and using a hash table to store (for each hash value) positions of the window whose hash corresponds to that value. We refer to the maximum number of window positions stored per hash value as the *depth* of the architecture. Works with this type of match discovery mechanism are found in [9], [10], [11], [12], [13]. In [9] and [10] the hash table is implemented using a linked list. This allows for memory area efficiency, but results in reduced throughput because the matches are processed sequentially. The maximum published throughput using linked lists is 46 MB/s.

On the other hand, the hashful architectures in [11], [12], [13], use a direct mapped hash table and can process multiple

matches per cycle. This results in increased throughput but puts the dictionary and hash table memories under stress due to the high number of required accesses. This is generally solved with memory duplication. The largest dictionary used in these works is 64 kB in size and a lookahead with 32 bytes is used. The highest published throughput is 5.6 GB/s.

Comparing the mentioned works reveals that hashful architectures are more area efficient and therefore support longer dictionaries and lookaheads than their hashless counterparts, which results in better compression ratios. In terms of throughput, both types of architectures can reach 1 GB/s. However, hashful architectures have more potential to achieve high throughputs while keeping the implementation area within reasonable values.

III. An LZ77 Brief

LZ77 is a dictionary-based compression algorithm. A *dictionary* refers to a list of sequences occurring frequently in the input. In the case of the LZ77 algorithm the dictionary is a sliding *window* that slides over the input as it is encoded. This *window* stores a portion of the most recently seen bytes of input. The encoding process consists in finding a portion of the current input sequence — called the *lookahead* — in the sliding window. When a match is found in the dictionary the LZ77 coder emits a distances and length which describe that match. The *distance* refers to where in the window the match starts, relative to the position of the current input byte. The *length* simply represents how many consecutive bytes from the matched sequence are equal to the current lookahead. If no match is found for the current lookahead a *literal* is emitted instead of the match.

IV. LZ77 Hardware IP

This section describes the architecture of the proposed LZ77 stream-based IP.

Figure 1 shows the topmost level view of the architecture. The core has four blocks: the lookahead buffer, a lookahead position register (`strstart`), a hasher and a LZ77 unit. The LZ77 unit is the most important block and contains almost all the complexity of the system. By replicating that unit the overall depth of the system can be increased.

The lookahead consists simply of a chain of 1-byte registers. The `strstart` block is a counter register which maintains the position in the window memory corresponding to the beginning of the lookahead. It allows to calculate the match distance. The hasher is a very simple block implementing the Gzip's hashing function, which hashes a trigram (s_i, s_{i+1}, s_{i+2}) from a sequence of symbols $(s_0, \ldots, s_{N-1})$

$$h(i) = \left[\left[\left(\left(2^5 s_i\right) \oplus s_{i+1}\right) 2^5\right] \oplus s_{i+2}\right] \bmod 2^{15}. \quad (1)$$

The multiplications by 2^5 amount to left shifts of 5 bits, while $x \bmod 2^{15}$ means x is truncated to its lowest 15 bits.

During the architecture's planning phase several important parameters for the compression ratio were studied, using a compression ratio simulator we developed for that purpose. The size of the dictionary memory, the maximum length of the matches and the number of bits of the hash values have been explored. These parameters were chosen such that a LZ77 IP with depth 3 combined with a static Huffman codes from Deflate would achieve a compression ratio greater than 2 for all the test corpora. Thus, the dictionary is 32 kB long, the maximum match length is 32 and the 13 least significant bits of the hash value given by Equation 1 were used as the hash function.

The LZ77 unit as well as its internal components are described next.

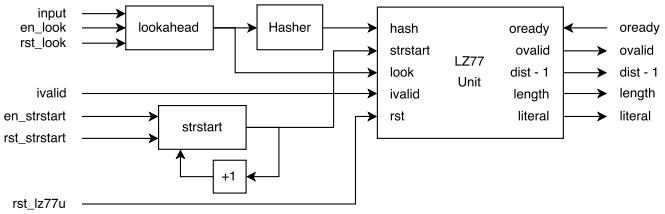

Fig. 1. Topmost level view of the LZ77 IP datapath

A. LZ77 Unit

The LZ77 unit receives the current lookahead, its hash and the current lookahead position and outputs a distance-length pair for matches, when a match is found, or a literal byte, when no match was found. The block diagram for the proposed LZ77 unit is shown in Figure 2.

The functionality of the block is as follows. The hash value coming from the hasher is used to index the hash table memory, which outputs a position in the window where a match for the current lookahead is believed to exist. This *candidate position* is then used to index the custom window memory, which outputs 32 bytes. These 32 bytes constitute the *candidate match* that will be compared to the lookahead in order to determine its length. Because in LZ77 the lookahead can match a portion of itself (see below), the output of the dictionary memory is fed into an overlapper block, which outputs a sequence of bytes which may consist only of bytes from the window or a combination of bytes from the window and the lookahead. Lastly, the overlapper output is compared to the lookahead in order to determine the length of the match, which can range from 0 (there is no match) to 32 (the maximum match length).

In parallel with the previously described process, the match distance is determined. First, one unit is subtracted from the `strstart` value. Then, the candidate position coming from the hash table is subtracted from `strstart-1` and the match distance of the match relative to the current lookahead position is obtained. The calculation of the match distance relies on the overflow of the subtraction to handle results that would otherwise be negative distances. Since a unit was subtracted from `strstart`, the distance results range from 0 to $2^{15}-1$, but represent values from 1 to 2^{15}. The subtraction also allows the overflow to give correct results for all distances — which would not happen otherwise, making an adjustment necessary for only one case.

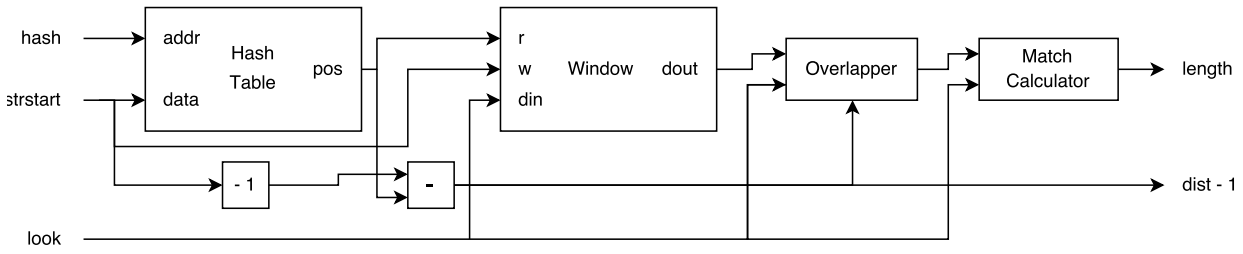

Fig. 2. Block diagram for the LZ77 unit

B. Window Memory

In order to achieve the highest possible throughput the architecture has to process at least one lookahead per cycle. To accomplish this, it is important to be able to make byte-aligned accesses to the window. Furthermore, in order to use a maximum match length of 32 the window must be able to output at least 32 bytes per cycle.

To satisfy these requirements a custom memory was designed for the window using BRAM primitives as building blocks. Taking into account the aspect ratios and the maximum number of accesses per cycle which the BRAMs in the target device support, the solution was to use 32 BRAMs in parallel.

The dictionary is stored sequentially in the BRAMs such that consecutive bytes of the memory are stored in different BRAMs. The address of the dictionary memory is split in two: the lowest 5 bits identify the alignment, while the upper bits are used to address the BRAMs. When an access aligned to 32 bytes is made, all the BRAMs are addressed with the same address. Otherwise, a subset of the BRAMs is addressed directly with the upper part of the original address and another subset is addressed with that addressed incremented by one unit.

The 32 bytes read from the BRAMs will be in an order not suitable to be used during the comparison with the lookahead. Hence, the output of the BRAMs is passed through a rotator circuit. There, 32 bytes are rotated such that the most significant byte of the output is the most distant byte of the window; and the least significant byte is the least distant. The output is now ready to be used by the subsequent blocks.

Regarding the writing of new bytes to the dictionary, the alignment bits are used to generate a write enable signal only for the BRAM that has the oldest byte of the window.

C. Hash Table Memory

The hash table memory is simple, as a direct mapped organisation for the hash table is used. Since 13-bit hash values are used and the window has 32k positions, the number of addresses of the memory will have to be 2^{13}, each position with a word width of at least 15 bits.

With the available BRAM configurations this memory can be implemented by using four RAMB36E1 primitives. The four primitives are stacked such that they form a continuous address range. Each memory is addressed with the 11 lowest bits of the hash value, while the two most significant bits are used to select the output of the appropriate BRAM. The least significant bits are also used to generate the write enable for the suitable memory.

D. Overlapper

For distances smaller or equal to the size of the lookahead, the candidate match extends into the lookahead. Therefore, for these cases the candidate match will be a combination of bytes from the window and the lookahead.

The overlapper consists of a 31-byte shifter and multiplexers. The match distance (calculated in the previous stages of the architecture) is used to select how many bytes should be shifted. The result of the shifter is fed into the multiplexers, whose select signals are also determined according to the match distance.

E. Match Calculator

This block compares the 32 bytes of the lookahead with those in the corresponding positions coming out of the overlapper. The result of these comparisons is an array of 32 bits, each of which is set to 1 if the corresponding bytes are equal. The number of consecutive set bits, starting from the most significant bit, are then counted in order to obtain the match length.

F. AXI Interface

The LZ77 IP was packaged into an AXI Stream interface such that it can be easily integrated into a complete compressor design. The data bus is 32-bit wide, with little endian byte order. The protocol uses input and output "valid" signals to indicate that the data on the buses is valid. The interface also has "ready" signals to specify if the bus is ready to read new data. When no valid data is found at the input of the IP bubbles will form on the pipeline; and when the output bus is not ready the IP will stall. A "last" signal also exists both on the slave and the master interfaces. This signal allows to delimit bursts by marking a word as the last of the current transmission.

V. Gzip Hardware/Software System

This section describes how the LZ77 IP was integrated into a hardware/software system to implement a full Gzip compressor. The platform used was a Zybo development board, which has a Zynq device accompanied by some peripherals.

A. HW/SW Partition

This section presents profiling results for the implementation of the Deflate algorithm in the Gzip software.

Gzip was run with its standard compression level (level 6) and for the 1 GB Enwik9 corpus — which should provide more accurate results than a smaller corpus — and profiled using `gprof`. The relative time of execution of each function can be seen in Table I.

The results show that 66 % of the execution time is spent finding the longest matches. The `longest_match()` function combined with `fill_window()` constitute Gzip's implementation of the LZ77 algorithm. As expected, LZ77 is the most computationally intensive part of Deflate and the use of a LZ77 hardware accelerator is a good solution. The remaining functions implement dynamic Huffman coding, lazy match selection and computing a checksum of the input and are implemented in software. This hardware/software partition is indicated in Table I.

TABLE I
PROFILING RESULTS AND HARDWARE/SOFTWARE PARTITION

Time (%)	Function name	Partition
66.87	`longest_match`	Hardware
13.66	`deflate`	Software
5.78	`updcrc`	Software
3.99	`fill_window`	Hardware
3.75	`compress_block`	Software
2.58	`send_bits`	Software
2.56	`ct_tally`	Software
0.29	`bi_init`	Software

B. HW/SW Implementation

The LZ77 IP was implemented in the programmable logic of the Zynq device, while the software components were implemented in the ARM processor. The DDR3 RAM was divided into four regions: a region for the input file, one for the compressed file and two regions for the results of the LZ77 IP.

The input to the LZ77 IP is forwarded from the RAM using an AXI DMA IP. The DMA is used so that the ARM is free to perform other tasks while the DMA manages the input and output of the IP. Using the DMA, the ARM only needs to initiate the transfers and verify if they were completed.

A diagram for the system is shown in Figure 3. The ARM starts by initialising the LZ77 IP accelerator, by requesting the IP to transfer 32 bytes to fill the lookahead. Then — still part of the system's initialisation — one request is sent to the DMA to start transferring the first chunk of the file into the IP; and another request instructs it to write the output of the IP into the first region of the RAM reserved for IP results. The ARM then waits for the IP to finish before proceeding to the main processing loop.

The core functionality of the main loop is to produce the final compressed output for a chunk of the file, using the results from the IP. The results for a match are read from the RAM and the ARM verifies whether it should encode a match or a literal. The encoding is done using the static Huffman coding described in the Deflate specification. In order to improve compression in about 5 % Gzip's lazy match evaluation was used, which only encodes a match after it has been verified that no better match can be found for the next position of the input.

The remaining part of the loop controls the DMA transfers for chunks of the file. In order for the ARM not to have to wait for the processing of the IP the chunks of the file that the ARM and the IP process in each moment are offset by one unit. For example when the IP is processing chunk n, the ARM is processing chunk $n + 1$. This is the reason why two memory regions for the results of the IP are used: one region is read by the ARM while the other is being written by the DMA with new results from the IP.

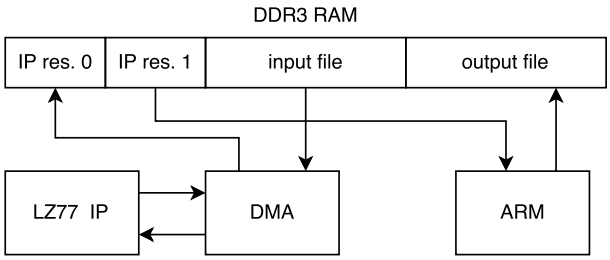

Fig. 3. Diagram for the hardware/software system

VI. RESULTS

This section starts by presenting the throughput and area results for the LZ77 IP. Then, the results for the hardware/software prototype implementing the full Deflate algorithm are shown. Finally, we discuss how the proposed LZ77 IP can be used to produce a high performance compression system.

A. LZ77 IP

The LZ77 accelerator was implemented with a frequency of 130 MHz on a Zynq Z-7010 device. It can receive an uninterrupted stream of bytes and consume one byte in each cycle. Consequently, its maximum throughput is 123.9 MiB/s.

Feeding the IP through the AXI DMA might limit the throughput of the IP. In order to measure this difference, a simple test was designed which consisted of transferring chunks of 16 kB to the IP through the DMA. The 16 kB correspond to the default maximum transfer size the DMA IP supports in a single transfer. The transfers were controlled by the Zynq's ARM and a total of 200 MB were transferred during the test.

The obtained throughput was 108.7 MiB/s, which is 88 % of the maximum value. This difference should be due to the relatively small size of each transfer compared to the total transferred size. Using the provided AXI DMA software library, each new transfer request involves multiple accesses to the DMA's AXI-Lite interface in order to verify that the DMA is free and set it up. On the other hand, it was verified — using

the Integrated Logic Analyzer (ILA) — that the AXI Stream transfers are not completely continuous. In particular it seems that the first few words in each transfer are transmitted isolated from the remaining words. Therefore, in order to improve the loss of throughput through the DMA, larger transfer sizes should be used and the operating frequency of the DMA could be increased.

The hardware utilisation results for the LZ77 accelerator after implementation are in Table II. The percentages on that table are relative to the total number of each resource available on the Zynq Z-7010 device. The relative utilisation reveals that the limiting factor for the accelerator's scalability is the amount of available block RAMs. The amount of used LUTs is only 12 %. The components using more LUTs are a rotator inside the custom dictionary memory and a shifter in the overlapper block.

Increasing the depth of the accelerator results in a linear increase of the resource utilisation. However when increasing the depth, the relative usage of BRAMs can be reduced by duplicating the frequency of the window memory: for a depth of 3 only 28 BRAMs would be needed, against 60 without the frequency increase.

TABLE II
GLOBAL AND PARTIAL HARDWARE UTILISATIONS FOR THE LZ77 IP WITH DEPTH 1. RELATIVE UTILISATIONS ARE FOR A ZYNQ Z-7010.

		Utilisation	
Component	Resource	Abs.	Rel. (%)
LZ77 IP	Slice LUTs	2171	12.3
	LUT as logic	2042	11.6
	LUT as memory	129	2.2
	Slice registers	1536	4.4
	Slices	671	15.3
	RAMB36E1	20	33.3
LZ77 Unit	Slice LUTs	2122	12.0
	LUT as logic	1994	11.3
	LUT as memory	128	2.1
	Slice registers	1189	3.3
	Slices	633	14.3
	RAMB36E1	20	33.3
Hash Table	Slice LUTs	350	1.9
	LUT as logic	350	1.9
	LUT as memory	0	0.0
	Slice registers	4	0.0
	Slices	135	3.0
	RAMB36E1	4	6.6
Dictionary	Slice LUTs	949	5.3
	LUT as logic	949	5.3
	LUT as memory	0	0.0
	Slice registers	269	0.7
	Slices	367	8.3
	RAMB36E1	16	26.6
Overlapper	Slice LUTs	490	2.7
	LUT as logic	490	2.7
	LUT as memory	0	0.0
	Slice registers	0	0.0
	Slices	211	4.8
	RAMB36E1	0	0.0

B. Hardware/Software System

The full hardware/software system was implemented in a Zynq Z-7010. Throughput and compression ratio were measured for the usual compression corpora, which range in size from 2.7 to 202 MB. These results are shown in Table III.

The table shows that throughputs from 8.2 to 10 MB/s were obtained, while the compression ratio varies from 1.8 to 2.4. The compression ratios are in accordance with what would be expected for a system using a 32 kB dictionary, 32 bytes of lookahead, depth 1 and static Huffman encoding.

However, the obtained throughputs are much lower than the measured throughput of the IP through the DMA — 108 MB/s. This is due to the fact that the ARM cannot produce the final encoding of the stream at the same rhythm as the LZ77 IP produces results.

TABLE III
THROUGHPUT AND COMPRESSION RATIO FOR THE GZIP HARDWARE/SOFTWARE SYSTEM

Corpus	Throughput (MB/s)	Comp. ratio
Calgary	8.77	1.98
Canterbury	10.08	2.43
Enwik7	8.19	1.81
Enwik8	8.26	1.83
Silesia	9.28	2.14

C. Maximising Performance

The previous results show that, in order to maximise throughput performance, the LZ77 IP should be integrated into hardware-only implementations. The IP's AXI Stream interface makes it easy to combine with other AXI IPs. As an example, it could be combined with Xilink's Ethernet Subsystem IP and a static Huffman IP in order to compress an Ethernet stream at a theoretical rate of 1 Gbps (the LZ77 IP can compress at 1.04 Gbps).

Table IV shows the throughputs of Gzip's fastest mode running on a Intel Core i5-2410M processor. It also shows the expected speedups of a hardware-only architecture, considering its maximum throughput (123 MB/s). The throughput of a hardware-only implementation of Gzip employing the proposed IP would be about 20 times faster than Gzip running on the Zynq's ARM, and from 1.7 to 2.4 times faster than on the Core i5.

TABLE IV
SPEEDUP OF A GZIP HARDWARE IMPLEMENTATION VERSUS GZIP'S FASTEST MODE, RUNNING ON THE ZYNQ'S ARM AND ON A CORE I5

	Throughput (MB/s)		Speedup	
Corpus	ARM	Core i5	ARM	Core i5
Calgary	5.91	50.11	20.97	2.47
Canterbury	6.99	62.57	17.73	1.98
Enwik7	5.45	44.98	22.72	2.75
Enwik8	5.49	45.76	22.55	2.71
Silesia	6.25	51.96	19.83	2.38

The scalability of the LZ77 IP can be seen from two perspectives: the depth of a single IP can be increased in order to improve the compression ratio; or multiple accelerators can be implemented in a single device in order to compress multiple streams simultaneously.

In the largest Zynq device, Z-7100, up to 37 LZ77 units can be implemented. Using a depth of 2 combined with static Huffman results in a compression ratio gain of about 7 %, compared with depth 1. However, as the compression ratio increases, the difficulty to further improve is exponential: using a depth of 37 results in 22–28 % better compression than for depth 1, depending on the corpus. The best way to further increase compression is to use dynamic Huffman, which would allow to reach gains from 35 to 50 %, instead of increasing the depth even more.

On the other hand, scaling the system by using multiple instances of the LZ77 IP could result in a global throughput of 4.4 GB/s, distributed equally by 37 streams. Depending on the application it might be useful to instantiate multiple accelerators with differing depths. This would allow to apply different compression ratios to multiple streams.

VII. Conclusion

In this paper a streaming hardware architecture implementing the LZ77 compression algorithm was proposed. The resulting IP can be used easily in multiple compression applications. The results show that a maximum throughput of 123.9 MiB/s can be achieved by the IP. This allows to use it in fast applications such as compressing Ethernet streams at 1 Gbps.

The functionality of the LZ77 IP was demonstrated by integrating it into a hardware/software system implementing Gzip. The results for this system showed that for this type of application the ARM processor constitutes a bottleneck for the performance of the system and thus, in order to maximise throughput, the proposed IP should be used in hardware-only implementations.

As future work a dynamic Huffman IP should be designed so that a full hardware Gzip accelerator can be realised with the best possible compression ratios.

References

[1] B. Tiwari and A. Kumar, "Aggregated deflate-rle compression technique for body sensor network," in *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on*. IEEE, 2012, pp. 1–6.

[2] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, Sep. 1977.

[3] L. P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951, May 1996. [Online]. Available: https://tools.ietf.org/html/rfc1951

[4] (2017, Mar.) Usage of Gzip Compression for websites. Q-Success. [Online]. Available: https://w3techs.com/technologies/details/ce-gzipcompression/all/all

[5] R. Mehboob, S. A. Khan, and Z. Ahmed, "High speed lossless data compression architecture," in *2006 IEEE International Multitopic Conference*. IEEE, 2006, pp. 84–88.

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

[7] M. A. A. Elghany, A. E. Salama, and A. H. Khalil, "Design and implementation of fpga-based systolic array for lz data compression," in *2007 IEEE International Symposium on Circuits and Systems*, May 2007, pp. 3691–3695.

[8] K. Papadopoulos and I. Papaefstathiou, "Titan-R: A Reconfigurable hardware implementation of a high-speed compressor," in *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*. IEEE, 2008, pp. 216–225.

[9] I. Shcherbakov, C. Weis, and N. Wehn, "A high-performance fpga-based implementation of the lzss compression algorithm," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 449–453.

[10] S. Rigler, "Fpga-based lossless data compression using gnu zip," Master's thesis, University of Waterloo, 2007.

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

[12] 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*. ACM, 2014, p. 4.

[13] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on fpgas," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 52–59.