Verilog PNG Encoder

João Freire Cardoso
joao.freire.cardoso@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2019

Abstract

The objective of this work is to develop an Intellectual Property (IP) subsystem for Portable Network Graphics (PNG) encoders, capable of operating at 30 frames per second (FPS) with the Video Graphics Array (VGA) standard when embedded system on chip hardware. The developed IP subsystem comprises an ARM Cortex A9 CPU running at 667 MHz, coupled with a hardware IP module that implements the LZ77 algorithm, running at 100 MHz. The LZ77 compression step is the most time consuming of the tasks involved in the encoder when done in software, and the LZ77 hardware module accelerates its execution by a factor of 40 when compared with the software-only implementation running on the ARM system. The operation of the LZ77 IP is similar to that of a content-addressable memory, which is a well known technique to implement dictionaries in hardware. The proposed solution is implemented on an FPGA board, using a Xilinx Zynq 7000 device. Although the LZ77 IP has achieved 82.2 fps in VGA, thus exceeding the objective set for this work (30 fps) by roughly two times, the whole PNG encoder is only capable of achieving 3.5 fps, which is three times thrice as fast as its software-only implementation. For the encoder to be capable of operating at 30 fps, other functions need hardware acceleration as well, such as the input image filter, the entropy coder, and the bit stream packer. Additionally, the whole software base needs significant optimizations for that to be possible.

Keywords: PNG, Hardware, LZ77, compression, accelerator

Introduction

In recent years, the advances in optical integration technologies have resulted in rapid proliferation of devices — such as cameras, displays, tablets, etc. — that can capture and display pictures, some of them with using resolution standards such as High Definition (HD) or Ultra High Definition (UHD). With the proliferation of these devices, there are growing expectations about the next-generation services for the distribution of HD contents via broadcasting and network delivery. However, the use of higher definition and resolution implies more data, and bigger files. Even though memory space has become cheaper in the last years, it is always better to make the most efficient use of the available space. This can be achieved with encoders, which compress the original file data into smaller files.

The last wave of technological advances, e.g., neural networks, artificial intelligence, image processing[6], and the Internet of Things (IoT), among others, have increased between different systems, the need to transfer more and more image data between different systems, and at higher rates. With it, there is an accompanying need for faster image compression, so as to keep up with the growing demand for image transmission. this trend is clearly apparent in the latest research works, in which use of image processing is used, to correct errors and detect changes or patterns for such different purposes as health monitoring, security, control, etc.

The Portable Network Graphics (PNG) format [8] uses data compression schemes without information loss, (commonly referred to as "lossless data compression"). It is an extensible image storage format. Furthermore, it supports black and white (binary mask), grayscale, indexed-colours, and truecolour images.

Hardware acceleration is the use of hardware to perform some function faster and more efficiently than would be possible in software running on a general-purpose central processing unit (CPU). The hardware that performs the acceleration, when separated from the CPU, is referred as the "hardware accelerator".

Traditional processors are sequential (instructions are executed one by one), and are designed to run generic algorithm. Morden CPUs can, in

fact, achieve some degree of parallelism via Instruction Level Parallelism, Data Level Parallelism or Thread Level Parallelism, but they remain essentially sequential.

Hardware accelerators improve the execution of a specific algorithms by allowing greater concurrency, providing specific data-paths, and possibly reducing the overhead of instruction control. Modern processors are multi-core and often feature parallel Single Instruction Multiple Data units, but hardware acceleration still yields considerable benefits. Hardware acceleration is suitable for any repetitive intensive key algorithm, and it can vary from small to large logic blocks. Hardware acceleration has more benefits, like reducing the power needed to perform the task. This is crucial, since as modern systems become more and more power hungry, it becomes crucial to minimize power consumption and, in some cases, extend battery duration.

This paper will first summarize the PNG encoder algorithms and establish the necessary background. Since and an open source PNG encoder software (LodePNG) was adopted as a starting point for the hardware implementation presented here, an analysis of the execution time of its main functions is then presented, in order to choose the best candidate functions to be accelerated in hardware. A new hardware design for the LZ77 function is then presented, to replace the software/based version of the algorithm, thus leading to the desired IP. As will be shown, the results show that the proposed hardware IP accelerated the LZ77 algorithm by a factor of approximately 40 when used with VGA-sized images. The overall performance of the PNG encoder is then discussed.

Background

The PNG format is essentially the combined result of a first filtering step(prediction), followed by compression step, which uses the Deflate algorithm. The filtering action in the first stage and is, in fact, a pre-compression method that tries to make the raw image data more compressible, by transforming the data before compression so as to extract the maximum efficiency from the compression algorithm (Deflate).In the second stage, a non-patented lossless data compression algorithm based on a combination of LZ77 and Huffman coding (the Deflate algorithm), will compress the processed data. The parameters and specifications for each step of the PNG encoder have only been specified for Method 0, the only method currently specified in the PNG specification [8], which describes how to encode a file into the PNG format

Filtering

As mentioned above, before the compressing stage, the scanlines (image's lines) are filtered, to increase the Deflate compression ratio. This 'filtering action', results in a sequence of bytes of the same size as the incoming sequence plus one byte. It does not reduce or compress the scanlines, but changes the representation of the data sequence, and precedes it with a byte indicating the filter type. It is important to note that **PNG filters are lossless**, that is, no information is lost [8].

The scanlines can use different filter types. The used filters are specified by the filter type byte, at the start of each scanline. Even though this byte is not part of the image data, it is included in the datastream sent to the compression step.

The choice of which filter to apply is left to the encoder, which means that different encoders may use different filters for the same image [8]. Currently, the PNG standard contains only the specification of Method 0, which defines five basic filter types. If ever extended, a different number will be used to define the new set. As has been said, there is no golden rule dictating which filter to apply to each scanline; the decision is left to the encoders.

The filtering algorithms are applied byte-by-byte, independently of the image pixel depth or the colour types. The existence of the alpha channel does not change the way in which the image is filtered. There are no other divisions inside each image line, except for bytes. The filtering algorithm sees a byte datastream without any kind of types division (RGBA or pixels).

The Deflate Algorithm

The use of the Deflate compression algorithm is specified in the PNG compression Method 0, and is based on a sliding window of 32,768 bytes. The deflate compression is derived from the LZ77 algorithm, adding Huffman coding to improve compression results. The data streams after Deflate compression, in PNG format, are stored in the "ZLIB" format. The ZLIB compression method code must specify method code 8 ("deflate" compression) and a window size of not more than 32,768 bytes (constraint specified in Method 0) for LZ77. Note that ZLIB compression method number and PNG compression method number are not the same and the additional flags must not specify a pre-set dictionary. A PNG decoder by definition must be able to decompress any valid ZLIB data stream that satisfies these additional constrains.

If the data to be compressed contains 16,384 bytes or fewer, the encoder can set the window size by rounding it up to a power of 2, being 256 the minimum value possible. This does not affect the compression ratio and decreases the memory required for both encoding and decoding.

A compressed data set consists of a series of blocks, each block is compressed using a combina-

tion of the LZ77 algorithm and Huffman coding. The Huffman trees for each block are independent of those from the previous or subsequent blocks; the LZ77 algorithm may use a reference to a duplicated string occurring in a previous block, up to 32kB before [5].

The division of the data into different blocks allows to keep the compression efficiency, by using fresh trees for each block. And avoid the compressor buffer from overflowing in case the input datastream been bigger than what the compressor was designed for.

Each block consists in two parts: a pair of Huffman code trees that describe the representation of the compressed data, and the compressed data. The compressed data consist of a series of elements of two types: literal bytes, and pointers to duplicated strings. A pointer is represented as a pair <length, backward distance>. Each type of value (literals, distance, and lengths) in the compressed data is represented using a Huffman code, using one code for literals and lengths and a separate code tree for distances. The code trees for each block appear in compact form just before the compressed data for that block. The Huffman trees themselves are compressed using Huffman encoding[5].

LZ77 Algorithm

This algorithm introduced the concept of a 'sliding window' which brought about significant improvements in compression ratio over more primitive algorithms [1]. The base concept of this algorithm is a dictionary based on pointers to represent repeated strings in a byte sequence. The pointer is formed by the following three parts:

1. An *offset* that points out how far, from the start of the file, a given string is at;
2. A run *length*, that tells how many characters past the offset are part of the string;
3. The deviating character, that is, an indication that a new string was found;

The string is equal to the string from *offset* to *offset+length* plus the deviating character.

The dictionary used changes dynamically based on the sliding window as the file is parsed for. The larger the sliding window, the larger the entries in the dictionary will be.

The LZ77 algorithm method has the following logic: it searches, in the sliding windows, for the same value as the one on its current position (Figure 1 blue area). This search is backward (Figure 1 "value search direction" arrow), it starts from the values closer to the current position. When a value is found it checks the following bytes (Figure 1 "match search direction" arrow), comparing them

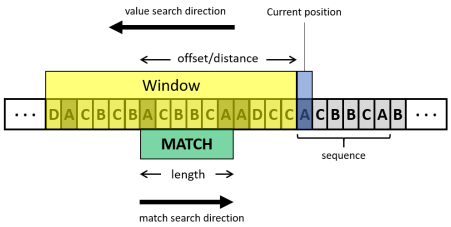

Figure 1: LZ77 Algorithm - Sequence Search

to the bytes following the current position (Figure 1 blue area), registering only the match with the bigger length.

The algorithm after checking all previous bytes for a sequence, writes the pointer for the largest sequence found, and moves for the next byte after the sequence. This next byte on the example of the figure 1 is the third byte 'B' outside, after the current byte and outside the sequence. If the search does not finds a sequence it writes the current byte and moves to the following byte.

In the Deflate specification, the pointer is made up by the distance (offset) and the length only. The encoded data is made up by literals (the literals refers to the representation of symbols on the LZ77 output, as it will be explained further ahead) and a pair of offset&length.

The smaller division that LZ77 uses for the symbols is the byte, meaning the symbol value is a decimal between 0 and 255. The LZ77 algorithm uses values between 0 to 285 to represent the pointers, and the literals.

The output of LZ77 is a mix of pointers and literals. The literal represents literally the symbol in the output datastream, having the values from 0 to 255, meaning no change has been made to the symbols binary form,(that is why it is called literal).

The remaining values 257 to 285 are used to represent the offset and the length. For these representations extra bits are needed, to save the pointer correctly, since the length can have values between 3 to 258 and the distance can be between 1 to 32,768.

The value 256 is reserved. This value is used to mark the end of the block, it means the end of the LZ77's datastream.

Huffman Coding

Huffman coding creates an unprefixed tree of non-overlapping intervals, where the length of each sequence is proportionally inversed to the probability of the symbol needing to be encoded. The more likely a symbol is to be encoded, the shorter its bit sequence will be [12]. The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this

table from the estimated probability or frequency of occurrence (weight), for each possible value of the source symbol. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted.

A deflate compression can use fixed Huffman codes or dynamic Huffman codes. These two options dictate where the algorithm gets the code table for the symbols. The fixed Huffman codes uses a pre-made table for the Huffman Coding, decreasing the processing time necessary for the execution of Huffman Coding. The Dynamic application will use the current datastream to construct the table, ensuring the best compression since the most frequent symbols (in the current datastream) will have a shorter code.

These two options have different impacts on the compression and execution time of the deflate.

Software-only PNG Encoder Profile

The profiling was performed on a Digilent ZedBoard Zynq-700 ARM/FPGA Soc Development Board, containing a Xilinx Zynq 7020 device. Its objective was to obtain real time data concerning the LodePNG [2] encoding speed and verifying the sections where the hardware acceleration will be implemented to improve the encoder to the required specifications. The profile obtained during experiment has been done with non-orthodox methods since it was not possible to use the profile option in Xilinxs tool. The method used to count the algorithm time is as follows: using specific functions to count the actual CPU cycle, and register these values before (start_time) and after (end_time) calling a function, or at the start (start_time) and end (end_time) of the function. The subtraction of this value gives the number of cycles that the selected part took. Printing the obtained times values to a log file or to another output.

This method has to be implemented carefully, when the counter section includes another section with counters the result will be incorrect. The obtained time values will be higher than reality, inducing errors on the profiling.

The program profile used 3 Benchmarks. The first two benchmark (Paint and Forest) are two different images types in different resolutions. These benchmark were used to gain more knowledge about the PNG encoder. The third benchmark is composed by photographs with the original VGA standard resolution (640x480 pixels). This benchmark is used to evaluate the speed up obtained after applying hardware acceleration.

Table 1 present the duration of the encoder and its two phases (Filtering and Deflate) for the 3 benchmarks. This table allow to evaluate which phase spends most time.

Table 1: CPU Time (ms)

Picture	Encoder	Filtering	Deflate
Forest144	70.63	11.23	53.47
Forest240	197.92	31.36	152.08
Forest360	452.86	69.84	350.96
Forest720	1,796.45	278.62	1,388.13
Forest1080	4,145.07	616.98	3,241.07
Paint144	39.26	7.82	27.35
Paint240	90.98	21.47	59.65
Paint360	175.60	47.12	107.47
Paint760	653.77	187.86	372.99
Paint1080	1,350.95	414.06	739.70
VGA-01	772.92	84.56	643.06
VGA-02	881.67	87.91	754.56
VGA-03	676.88	85.97	550.74
VGA-04	811.14	86.11	685.24
VGA-05	708.81	84.03	586.99
VGA-06	990.40	85.67	867.52

The comparison between the Forest and Paint benchmark allows to conclude that the Deflate duration is dependent of the image data. The Forest type image has a more diverse colour pattern than Paint type. Increasing colour diversity prolongs Deflate duration.

Table 2 summarizes the Deflate main operations (LZ77 encoding, Huffman Tree creation, Deflate writing) execution time.

In Deflate the most time consuming algorithm is the LZ77 as it can be observed in Table 2. The VGA benchmark corroborates the conclusion taken from the first two benchmarks. We can conclude that the LZ77 should be the first one to be accelerated.

Related Work

This section summarizes the characteristics of the implementations and works related to the PNG format algorithms.

During the research phase, it was difficult to find related works and available IP implementations for a PNG encoder or its algorithms. The only hardware PNG Encoder found is a commercial product from Visegni that works with a rate of 1 pixel per clock cycle [13]. However, there are 3 solutions for a Deflate Hardware implementation: One commercial product from CAST-Inc, which achieves a throughput of 100 Gbps [4]; and two open source implementations. The LZ77 Algorithm has a variety of works about increasing its speed, despite none of these implementations being available.

Table 3 indicates the implemented algorithm and if it includes the LZ77. The frequency, the through-

Table 2: CPU Time - Deflate (ms)

Picture	LZ77	Huffman Trees	Write Data
Forest144	32.40	1.45	17.46
Forest240	93.29	3.92	48.36
Forest360	225.87	6.83	109.36
Forest720	918.60	10.79	419.88
Forest1080	2,198.62	23.67	931.40
Paint144	22.31	1.80	2.76
Paint240	49.42	4.39	5.10
Paint360	91.12	6.71	8.47
Paint720	342.27	8.28	19.62
Paint1080	687.73	18.28	29.17
VGA-01	502.34	7.65	127.10
VGA-02	632.76	6.39	108.63
VGA-03	399.47	7.32	135.08
VGA-04	552.02	6.95	119.48
VGA-05	445.85	6.73	120.86
VGA-06	761.24	6.91	93.88

put achieved in each approach and its implementation size is presented next. The window size used is indicated in the last column, when the approach involves the LZ77,

The Hash approach has good working frequency and throughput, however the hash system may reduce the compression rate, failing to detect older sequences inside the window.

In the Table 3 is possible to observe that from the available implementations, most use small search depth (window size) and have a slower throughput. The most recent CAM implementation achieve better results than a systolic array with the same frequency, making it a good example to follow and to try to improve it further.

Implementation

The LZ77 IP is designed to accelerate the deflate algorithm, removing this workload from the processor. It should be able to produce an index (distance) to the repeated word sequence very fast. The PNG profile shows that most of the execution time is spent in the search of the best match. Even with software optimizations, this process is still very time consuming. The LZ77 algorithm can be divided in three parts:

- the search for the start of the match (first three bytes must match);
- the matching of the rest of the sequence;
- the writing of the result;

The first two parts repeat until the best sequence is found – these two parts are the most time consuming. The last part is a simple write of one or four integers containing the information of the LZ77 compression. This part is a single step and can not of course be improved. In conclusion the first two parts must be improved, aiming at reducing the time spend on the sequence search and matching (length calculation), using pipeline and parallelization.

The solution is to use a CAM-like memory allowing to find the start of the input sequences in parallel, reducing the LZ77 search time. With this methodology, the compression time equals the stream transmission time. The spatial complexity of the circuit grows proportionally to the window size used.

The datapath consist of three blocks: “Pointer Search”, “Length/Distance Coding” and “Output Buffer”. These blocks are serially connected, and have pipeline implemented to reduce the critical path. This datapath is very simple, therefore the hardware system does not need a control unit for the stream management or for its configuration. The datastream management is implemented by a simple combinatorial circuit. It freezes the system whenever the input is not valid or the output can not be received by the next system. This control method is only possible due to the simplicity of the LZ77 algorithm.

The LZ77 compression produces a datastream composed by coded literals and coded pointers. These codes in the LodePNG LZ77 are saved in integers (32-bit). Keeping the hardware datastream in the same format prevents extra work in the software. The software is spared from formatting datastream.

The LZ77 IP inputs a string of bytes and outputs a 128-bit word consisting of 4x32-bit LZ77 codes, containing literals and/or pointers expressed according to the LZ77 data structure.

Shift Register

The shift-register is the system memory. It implements in hardware the ’sliding window’ concept introduced by the LZ77 algorithm.

Pointer Search

The ”Pointer Search Block”, shown in Figure 2, searches in the window for the longest matching sequence. It is in the ”Pointer Search Block” block that the repeated sequences are found and the best is selected and coded as a length/distance pair. Its input connected to the Shift Register and the output connected to the Length/Distance Coding block. It reads all the data saved in the shift-register, and compares each byte with the current byte, allowing all matches to be found in one clock

Table 3: Related Works Characteristics

Approach	Algorithm	LZ77	Frequency	Size	Troughput	window size
VISENGI PNG Encoder	Deflate	yes	NA	NA	1 Byte/clk	32 KiB
CAST ZipAccel-C	Deflate	yes	NA	20K-100K Gates	100 Gbps	32 KiB
HDL-Deflate	Deflate	yes	100 MHz	9823 LUT	31.79 MiB/s	32 B
HT-Deflate-FPGA	Deflate	yes	NA	NA	NA	NA
Novel Adaptive Version	Huffman	no	56 MHz	3,250 LUT	NA	-
VLSI Design	LZ77	yes	40 MHz	18,397 transistors	12.7 MiB/s	NA
FPGA-based Systolic Array	LZ77	yes	105 MHz	NA	15.5 MiB/s	1 KiB
Shiftable CAM device	LZ77	yes	12.5 MHz	1,590 Gates	12.5 B/s	16 B
CAM Approach	LZ77	yes	101.309 MHz	7870 Slices	96.62 MiB/s	64 B
Hash Architecture	LZ77	yes	130 MHz	2171 Slices	108.7 MiB/s	NA

cycle.

The block can be divided in two different sub-blocks: "Word Search" that performs the search; "Selector" that selects the longest sequence;

The "Word Search" is composed of as many "Char Matcher" units as positions in the Shift Register, which find the sequences. The block size is proportional to the window size used. The size of the remaining datapath is independent of the window size value.

The "Char Matcher" is shown in Figure 3. Each block searches a sequence with a distance/offset value pair. Essentially, each byte in the window will be associated with a "Char Matcher", allowing the simultaneous search and discovery of any possible sequence in as many cycles as the length of the sequence.

A very simplified version of "Char Matcher" logic is presented oin this figure with the block divided into three pipeline levels. In the first level the comparisons between different registers from the shift-register are performed. The second level guarantees that only matches with the minimum size are valid sequences. The third level marks the state of the "Char Matcher" and starts/ends the matching process in the correct timing.

The "Char Matcher" only starts the matching process, when no other "Char Matcher" is running. When a "Char Matcher" is working, it inhibits the others from starting. This condition is critical to avoid the corruption of the detected sequences. When all the "Char Matchers" are idle, an idle signal is generated by the "idle" block. The "Char Matcher" uses this signal to unlock the matching, switching to working state.

The "Char Matcher" units start to work after they are idle-reset and find an initial matching of 3 equal bytes, using the current byte and the "preview" registers, which unlock the matching process. For this initial match of three bytes, it is required to have already in the shift-register the next two bytes. These next two bytes are saved in the "Preview" registers. The "Char Matcher" only compares three bytes in the first match, after this initial match the comparisons will be of one byte per cycle.

After all the "Char Matcher" units stop working, the "Word Search" block switches to idle. When the "word search" is idle the longest sequence is selected as the block output. Since all "Char Matcher" units start working at same time, the longest sequence will be given by the "Char Matcher" with the longest execution. When no sequence is found the literal is the selected output. When a sequence is found the pointer is the selected output. This block outputs six signals: three flags (valid, last, pointer) and three different values (literal, offset, length). The "last" control signal is not shown on the picture.

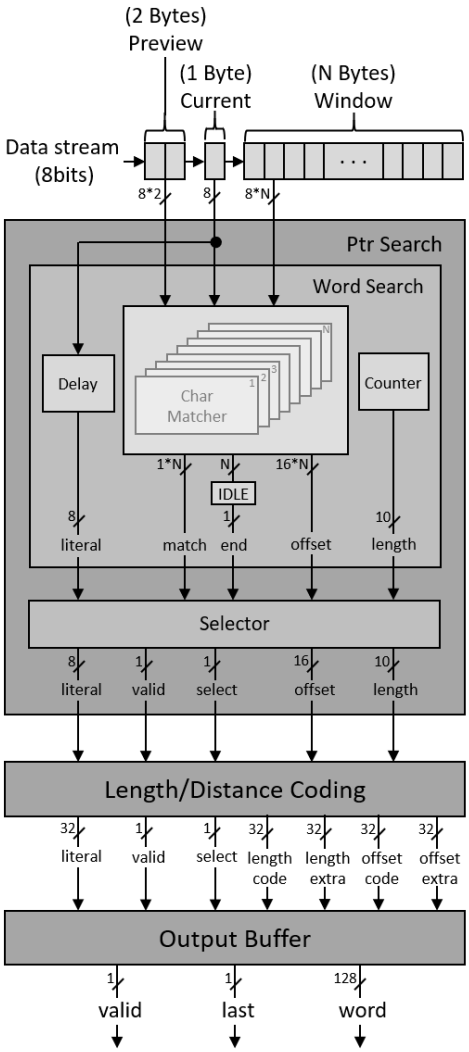

Figure 2: Pointer Search Block Diagram

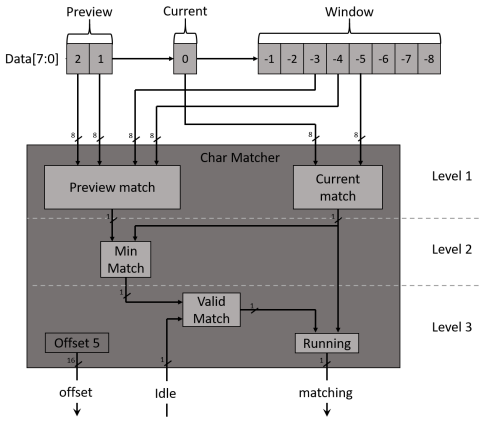

Figure 3: Char Matcher Block Diagram - 3 pipeline levels

The "Word Search" has also implements additional logic that guarantees that the length for a sequence is restrained to the maximum allowed value. When the length reaches the maximum value, a flag called "maxlen" restarts the matching, forcing the block "Word Search" into idle state. This forced change to idle state ends the current matching, and the next block ("Selector") selects the sequence with the lowest offset. The "Word Search" starts a search for a new sequence, immediately after the flag "maxlen" is activated.

Length/Distance Coding
The "Length/Distance Coding" (Figure 2) codifies the information from the "Pointer Search" block output, that can be one of different types:

- The literal;
- The pointer;

The code for any literal value, as previously mentioned, is the value itself. The pointer describes the sequence found in the "Pointer Search" block. The pointer is a structure (normally denominated "length distance pair"), constituted by two pairs, the length and the offset values, which are routed together in the logic. In each pair, the first member is the "base code" of the value (length or offset), and the second member is the "extra bits", which distinguish different values sharing the same base code. (more details in [3])

Output Buffer
Since literal and pointer data have different sizes, they require an output buffer, that groups the output into a constant data size, sending it on an AXI Stream interface. (more details in [3])

The Output Buffer Block is essentially a Ping Pong Buffer that groups different size outputs to produce one unique datastream.

Experimental Results
This section presents the execution time results for the VGA sized images. The tests consist in encoding the images using the prototype and recording their execution times. The execution times (Table 4) are broken down in their two main steps: filtering and deflate.

Table 4: Execution Time (ms)

Picture	Filtering	Deflate	Total Encoder
VGA-01	85.33	159.02	289.85
VGA-02	88.80	137.64	268.22
VGA-03	86.68	170.05	303.66
VGA-04	87.37	150.97	284.62
VGA-05	84.65	154.75	283.30
VGA-06	86.79	126.83	253.80

It is possible to observe a decrease in the Encoder and Deflate duration. The encoder execution time is reduced in half and the Deflate duration is one third of its previous value. It is also possible to note that the execution times for the different VGA images have a low variation variation between them. This is because the execution time depends only on the image size.

The Deflate execution time is expanded in Table 5 to show the time spent in its most important components. It is possible to observe in that the LZ77 algorithm time is reduced immensely when the IP is used. For an uninterrupted datastream transfer, for VGA images, the expected LZ77 IP execution time is 9,220.8 μs. However, the experimental execution time of the prototype for the LZ77 IP is about 12ms. The difference can explained by the DMA interruptions and other communication and processing overheads.

Table 5: Execution Time - Deflate (ms)

Picture	LZ77 (ARM)	LZ77 (IP)	Huffman Trees	Write Data
VGA-01	502.34	13.52	7.64	125.25
VGA-02	632.76	13.50	6.38	106.03
VGA-03	399.47	13.51	7.33	134.49
VGA-04	552.02	13.51	6.97	116.64
VGA-05	445.85	13.51	7.09	120.92
VGA-06	761.24	13.50	6.93	93.28

Compression Ratio

Table 6 compares the datastream size before and after the LZ77 compression, for both the prototype and the original encoder. The column named Relative is computed using the following equation:

$$R = C/U$$

,

where: RS is the relative size; C is the compressed data size; U is the uncompressed data size.

This table shows that the designed LZ77 IP also increases the compression rates in addition to providing speedup. The reason is the fact that the IP finds all sequences in parallel and chooses the best one while the LZ77 software searches sequentially and stops after a 128-long sequence is found.

It is important to note that images used in these tests do not favour the algorithm. They are the most challenging, since big repetitive sequences are hard to find in this kind of pictures. The images used are good to show the best possible acceleration obtained for the worst case. Other types of images if with the same resolution will have the same execution time. As explained before, the LZ77 throughput is constant and its execution time is only dependent on the data size.

Table 6: LZ77 Compression with/without Hardware acceleration (unit)

Picture	LZ77 Compression	
	Software	IP
	Relative (%)	Relative (%)
VGA-01	87.24	82.34
VGA-02	85.83	80.38
VGA-03	97.73	89.29
VGA-04	87.34	81.73
VGA-05	99.66	88.92
VGA-06	79.37	75.84

Speedup

Table 7 presents the speedups achieved by the prototype over the base software encoder. The results are given individually for the LZ77 algorithm, for the Deflate algorithm (which includes LZ77), and finally for the whole encoder.

Table 7: Encoder Speedup

Picture	LZ77	Deflate	Encoder
VGA-01	37.15	4.04	2.67
VGA-02	46.87	5.48	3.29
VGA-03	29.56	3.24	2.23
VGA-04	40.85	4.54	2.85
VGA-05	33.00	3.79	2.50
VGA-06	56.39	6.84	3.90
average	40.64	4.66	2.91

It is clear that the prototype is faster than the original version. The LZ77 IP is on average 40x faster than the software-only encoder while running at a frequency 6.667x lower. Unfortunately, this acceleration does not decrease the overall Deflate speed by the same factor, only reducing its execution time by 77.19%. The full encoder execution time is only reduced in half, which means that to reach the initial goal of this work – a performance of 30 FPS – much still needs to be done.

Throughput

The LZ77 IP is designed to have the same execution time for datastreams of the same size. The small variations are caused by the communication between the processing system and the IP. The throughput for the IP at a frequency of 100 MHz is 95.37 MiB/s (100 MB/s), since it can consume one byte per clock cycle. The IP image throughput is measured in FPS and it is presented on Table 8. This table compares the expected FPS with the

value in the presence of the communication overheads.

Table 8: LZ77 IP Frames per Second

Image Resolutions	FPS	
	Theoretical	Experimental
VGA	108.50	82.19
HD	36.17	32.52
FHD	16.08	14.55
QHD	9.04	8.189
UHD/4K	4.02	3.65
8K	1.00	0.91

Table 8 shows that the IP achieves the 30 FPS goal for the first two resolutions. It is important to note that these results are for a single IP that processes image blocks serially. The PNG encoder algorithm divides the image data into different blocks which could be compressed independently by multiple IP blocks. This possible parallelization provides a solution to further improve Deflate with multiple LZ77 IPs.

Discussion

Even though the developed IP did, in fact, surpass the acceleration objectives, the underlying objective of reaching an overall encoding speed of 30 FPS (which has been defined in order to address some motion picture scenarios) would require further interventions in other aspects of the encoder other the LZ77 compression, as has been discussed. In particular, and besides the need for a general software optimization, three main bottlenecks have been identified in the LodePNG encoder, which should be addressed in future work, so that the improved PNG encoder may reach the 30 FPS goal. These three bottlenecks, and the approaches proposed to address each one of them in future work are the following:

Deflate Writing Acceleration

It has been observed that long execution times are required when writing the Deate output. This results from the need to build a data stream out of the variable length codes produced by the Huffman coding step, an operation that is not efficiently done by regular processors. By designing specific hardware for this task, it will be possible to render this execution time negligible, which will have in fact eliminate this bottleneck.

Filtering Acceleration

Another bottleneck lies in the filtering step, which applies a sequence of filters to each line of the image entering the processor. This task can be parallelized in hardware, by applying all filters simultaneously to the scanline. This parallelization will reduce the filtering time by 80% of its original duration. This will eliminate the second identified bottleneck.

LZ77 IP Improvements

Finally, it must be noted that the presented LZ77 IP still has room for improvement. The main improvements that should be worked on are reducing the required logic and increasing its frequency of operation. The design of the search unit can be improved, so as to maintain the efficiency while reducing the required implementation logic. An increased working frequency will allow a higher performance in terms of the number of processed FPS, further accelerating the encoder.

Conclusion and Future Work

In this paper, a hardware accelerated PNG encoder was proposed and implemented, in an attempt to allow the use of such encoders in small and low energy devices, which cannot afford to have a powerful processor to run these algorithms entirely in software, and for which there are surprisingly few solutions in the market, open source repositories, or even in the literature. Even though the overall objective of encoding at 30 fps has not been achieved, important results have been obtained concerning the encoders inner workings, and a major limitation to its encoding speed has been eliminated: the software-based LZ77 implementation. Additionally, several interesting directions for future work have been discovered. The three bottlenecks identified above should be addressed in future work. Eliminating them will hopefully be sufficient to attain the desired objective of having the PNG encoder achieve an overall encoding speed of 30 FPS.

References

[1] History of lossless data compression algorithms.

[2] Lodepng - open source png encoder/decoder.

[3] J. F. Cardoso. Verilog png encoder, November 2019.

[4] CAST. Zipaccel-c.

[5] L. P. Deutsch. Deflate compressed data format specification version 1.3. 1996.

[6] M. Egmont-Petersen, D. de Ridder, and H. Handels. Image processing with neural networksa review. *Pattern recognition*, 35(10):2279–2301, 2002.

[7] V. Z. Grajeda, C. F. Uribe, and R. C. Parra. Parallel hardware/software architecture for the bwt and lz77 lossless data compression algorithms. *Computación y Sistemas*, 10(2):172–188, 2006.

[8] e. a. G.Randers-Pehrson. Png (portable network graphics) specification, version 1.2, July 1999.

[9] S. Jones. 100 mbit/s adaptive data compressor design using selectively shiftable content-addressable memory. *IEE Proceedings G (Circuits, Devices and Systems)*, 139(4):498–502, 1992.

[10] N. Ranganathan and S. Henriques. High-speed vlsi designs for lempel-ziv-based data compression. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 40(2):96–106, 1993.

[11] A. E. Salama, A. H. Khalil, et al. Design and implementation of fpga-based systolic array for lz data compression. In *2007 IEEE International Symposium on Circuits and Systems*, pages 3691–3695. IEEE, 2007.

[12] M. Sharma. Compression using huffman coding. *IJCSNS International Journal of Computer Science and Network Security*, 10(5):133–141, 2010.

[13] VISENGI. Png encoder.