Verilog PNG Encoder

André Filipe Romão Merendeira
Electrical and Computer Engineering Department
Instituto Superior Técnico
Lisbon, Portugal
andre.merendeiraa@tecnico.ulisboa.pt

***Abstract*—In recent times the quality of images has greatly increased and so did the amount of memory needed to store them. The Portable Network Graphics (PNG) is a reliant and powerful image format that allows us to compress raw images to decrease their storage size while not losing any of their excellent quality. As such, this thesis objective is to develop a system that can achieve 30 Frames Per Second (FPS) for Video Graphics Array (VGA) in a System On Chip (SoC)-Field-Programmable Gate Array (FPGA) while sacrificing as little compression ratios as possible. To achieve this, an Intellectual Property (IP) hardware core that comprises the pre-compression process and the Lempel-Ziv Coding 1977 (LZ77) algorithm has been developed.**

This system was implemented in a Xilinx Kintex UltraScale FPGA board and the software runs in a Reduced Instruction Set Computer (RISC)-V Central Processing Unit (CPU) developed by the IObundle team. After the program was executed with no hardware acceleration, the time results showed that the IP would have to greatly decrease the process execution time.

Even though the system was able to decrease, on average, the total run time by 10 times, it still was not possible to achieve the final objective of 30 FPS but was able to achieve an equal compression ratio or better. From these results, it was concluded that the communication between the software and the hardware still needs further optimizations and that the IP needs to include more operations that are being performed by the software.

***Index Terms*—PNG, pre-compression, LZ77, RISC-V, FPS, FPGA.**

I. Introduction

With the new generations of digital cameras capable of delivering high-resolution quality images, the amount of raw image data has increased significantly in these devices. This means that storing and/or transmitting these images in their raw state will be very expensive in terms of resources. The solution is to compress the data, while keeping its quality unchanged.

The compression allows us to decrease the size of an image, allows for faster transmission when using the same rate as the uncompressed data. It also lowers the risk of congestion in the network. Moreover, saving and retrieving data can be done much faster, resulting in an overall speed up of the system.

The PNG [1] format is the best file format to use in this case, since its use has recently increased and is expected to replace the also very popular Graphics Interchange Format (GIF) [2] format. The reasons why PNG has been chosen for this work have to do with the fact that PNG is patent-free and has many attractive features, such as lossless compression, opacity control, and interlacing.

The chosen implementation is the open-source LodePNG software [3] since it provides an all in one encoder and decoder without any dependency on external libraries. Another advantage of using this program is that it offers several utilities, being the most important one the benchmark, which provides an easy to use set of tests that use locally generated patterns with customisable sizes.

These images pose some challenges to the software, since they take longer to encode and present a low compression ratio, and can be considered as a worst-case scenario. After acceptable results have been achieved for this benchmark, real images were tested.

All the previously mentioned images will use Red Green Blue (RGB) and Red Green Blue Alpha (RGBA) color models, with 640 by 480 dimensions.The objectives of the accelerator are to achieve an equal or better compression ratio than the one obtained with LodePNG, and a minimum performance of 30 frames per second FPS on VGA-size images.

The real-world images used in this works have been chosen taking into consideration the following pre-requisites, which pose difficulties to the original software but are quite common:

- Abrupt pattern change.
- Big color palette.
- Common everyday life pictures.

The compression ratio is defined as the number of bytes needed to store the original uncompressed data divided by the number of bytes needed to store the compressed data:

$$\text{Compression Ratio} = \frac{\text{image original size}}{\text{compressed image size}}. \quad (1)$$

The FPS rate will in turn be calculated by measuring the total time it takes for one image (frame) to be compressed and then inverting this result.

This work starts by explaining several important aspects about image formats and how a PNG image is formatted. Then, it details how the compression of a PNG image is performed and how it can be divided into separate parts. It details the program that performs the PNG encoding, explains how the software was tested and the time profiling and compression ratio results that originated from those tests are presented and discussed.

The hardware IP that was developed in order to accelerate the previously mentioned software is shown and its architecture is detailed. After that the FPGA board resources utilization is shown and the performance results obtained from the hardware acceleration are presented, discussed and compared with the ones obtained from the non-accelerated version. Finally the conclusions resulting from this work are shown and future work that can improve the system's performance is discussed.

II. Background

There are two ways of compressing data, either by *lossless* or *lossy* compression.

Lossless data compression allows the process to be reversed to obtain the original data. The downside of this type of compression is that the compression ratio has a maximum practical limit dictated by the algorithm used.

Lossy compression allows for a trade-off between quality and compression ratio, by using inexact approximations and partial data discarding. Lossy encoding allows for much higher compression ratios than lossless encoding but it irreversibly corrupts the original data.

Since this work is meant to support post-processing tasks, which rely on authentic data input, the desirable compression method is the lossless data encoding approach which is used in image file formats such as PNG and GIF.

GIF is a bitmap image format, meaning it is a dot matrix data structure that represents a rectangular grid of pixels, being each pixel's color specified by a number of bits. It was developed on June 15, 1987, by a team of online service providers from *Compuserve* [2], and is now widely used on the Internet. It offers the ability to create animated images and is portable among several operating systems and applications. It is a lossless method and provides good compression ratios.

Since the format became subject to licensing and other legal problems, a patent-free replacement for GIF was later created by an Internet committee: the PNG format addressed in this work. This format emerged from the necessity of an improved patent-free replacement for GIF and was created in 1996 by an Internet committee of computer graphics experts and enthusiasts [1]. Although PNG does not offer the ability to create animated images, PNG presents other improvements over GIF, such as:

- No licensing needed for software development.
- Alpha channel to support transparency and its degree.
- Palletes of 24-bit RGB or 32-bit RGBA colours (the extra byte is for the alpha channel).
- True colour support.
- Better compression ratios.

A. File Format

The PNG file is composed by a file header followed by a series of chunks that obey to strict rules.The file header is composed by 8 bytes and represents a signature of the file format. Any PNG file has this header. After the 8-byte header, there is a series of *chunks* that carry information about the image. This layered structure allows for PNG to be extended or improved without causing compatibility issues with older versions.

The chunks are divided into two classes: *critical* and *ancillary*. The critical chunks are necessary for the correct operation of the format, while the ancillary chunks may be ignored if the processing program does not need them. Each chunk, regardless of their class, is composed of four parts as presented in Table I.

TABLE I: Chunk parts and their description.

Part	**Size**	**Description**
Length	4 bytes	Unsigned integer that announces the size of the *Data* chunk
Chunk type	4 bytes	Four-letter case sensitive ASCII code that defines the type of the chunk
Data chunk	*Length* bytes	Contains the chunk's data
CRC	4 bytes	Provides the *checksum* computed for the *type* and *Data* chunks

There are four critical chunks that must be present in every PNG file and that are described in the chunk type in American Standard Code for Information Interchange (ASCII) encoding as follows:

- **IHDR**: This identifies the first chunk to be presented and contains (in order) the image's width (4 bytes), height (4 bytes), bit depth (1 byte), color type (1 byte), compression method (1 byte), filter method (1 byte), and interlace method (1 byte), in a total of 13 bytes.
- **PLTE**: Contains the colour palette;
- **IDAT**: Contains the data composing the image.
- **IEND**: Indicates the end of the image.

It must be pointed out that there can exist multiple IDAT chunks since the image may be split into various sub-images

A PNG file may also contain several other optional ancillary chunks that provide information about numerous image attributes such as textual metadata, background color, gamma values, alternative color palette, and transparency. Since these fields are not mandatory and may be ignored by the encoder, they are not in detail as the critical chunks are.

III. PNG Compression

Image data compression using the PNG format is done in two different stages: *pre-compression* [4], also known as filtering, and *compression* using DEFLATE, a non patented lossless data compression algorithm [5]. Although the pre-compression phase increases the file size (by adding a byte to specify the filter in each line) and demands more computing resources, it allows for a higher compression ratio after the DEFLATE algorithm is applied to the filtered data.

A. Pre-compression

The pre-compression process is done by using a simple prediction method: for each line of the image, a filter that predicts each pixel by looking at its neighbors and subtracting the predicted value from the actual value is applied. This filtering method is known as *method 0* since the filtered values will cluster around 0, especially in images where neighbour pixels are similar, which greatly increases the compression ratio provided by the DEFLATE algorithm.

The type of filter for making these predictions varies for each line and is chosen by the encoder, meaning that the used filter for a line may differ for different encoders. The pixels used to compute the value to subtract from the original value (pixel **0**) are presented in Figure 1.

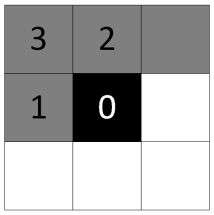

Fig. 1: Position of the pixels used by the filtering method.

There are 5 types of filters chosen according to the surrounding pixels used to make the prediction:

- **None** Filter: The predicted value is 0 meaning that the raw pixel suffers no change.
- **Sub** Filter: Uses pixel **1** as the predicted value and subtracts it from pixel **0**.
- **Up** Filter: Subtracts the value of pixel **2** from the current pixel.
- **Average** Filter: Computes the mean value between pixel **1** and pixel **2**, rounding it down and subtracts it from the original value.
- **Paeth** Filter: Computes $p = pixel\mathbf{1} + pixel\mathbf{2} - pixel\mathbf{3}$, and, from pixels **1**, **2** and **3**, chooses the pixel closest to p and subtracts it from pixel **0**.

In cases where one or more of the required pixels to make the prediction do not exists, i.e., at the image edges, the used value is zero. This means that the algorithm does not wrap around, that is, does not uses pixels from the other side of the image.

B. Compression

As previously mentioned, in PNG the compression phase is done by applying the DEFLATE algorithm that uses a combination of the LZ77 algorithm [6] and Huffman encoding [7]. This process was defined by *PKWARE* in 1991 as part of their *PKZIP* archiver [8] and is now widely used by the GNU ZIP (GZIP) [9] utility meaning that it has been severely tested and is one of the most commonly used file-compression algorithms.

1) LZ77 Algorithm: The LZ77 lossless data compression algorithm was first published in papers by Abraham Lempel and Jacob Ziv in 1977 [6], hence the name LZ77. It is based on the concept of a *sliding window* with a certain *width* positioned immediately before the position that is going to be processed and that englobes some of the previous data. The previous positions that are inside the window are then used to process the current position and when this is done, the window is moved to include this position meaning that the sliding is done when the current position is updated. The window can be perceived as a dictionary used to encode the subsequent data and therefore, the LZ77 algorithm is in practice a dictionary encoder.

In order to achieve compression, the LZ77 algorithm replaces repeated continuous parts of data with a *pointer* to one occurrence of it that has previously been found in the uncompressed data stream and is inside the sliding window. This pointer is comprised of a pair of numbers called *length-distance pair*, where *length* is the number of characters that composes the repeated string and the *distance*, mostly known as *offset*, is the number of characters one most travel in the uncompressed stream in order to find the beginning of the corresponding string.

To better understand how the LZ77 algorithm works, take into consideration the uncompressed data presented in Figure 2 where each square represents a byte of data and the letter inside it represents the byte's value.

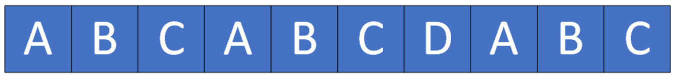

Fig. 2: Example input byte-stream.

Let us consider a window of width 4. The algorithm analyses the stream from left to right. Since the window is still empty, i.e. no byte has yet been processed, the first byte, with value **A**, is directly inserted into the output buffer as a *literal*. On the second and third byte, since their value is not present in the sliding window, they will also be directly inserted in the output.

When the fourth byte is processed, a match is found in the window (the first byte has the same value) and the algorithm moves on to the next position, saving the value of the best offset, in this case, as **3** and starts to count the length of the string that has consecutive matches. When the algorithm finally encounters the byte with value **D**, the length counter stops at 3 and the string "**ABC**" gets encoded into the output buffer as a length-distance pair of length **3** and distance **3**. Since **D** is not in the window then, it will be outputted as a literal.

The last "**ABC**" string is then encoded as a length-distance pair with length **3** and offset **4** since the closest occurrence of this string is the one that was previously encoded as a length-distance pair. This evolution of the sliding window and match finding can be perceived in Figure 3 and the final output will be as presented in Figure 4, where the length-distance pair is represented as **[L,D]**, being **L** the length of the match and **D** the offset.

2) Huffman coding: After the data-stream is compressed by the LZ77 method, it is then encoded a second time by means of *Huffman coding*, which was developed by David A. Huffman and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes" [7]. This algorithm produces optimal prefix codes which can be perceived as a variable-length code where one entire code-word is not part of the initial segment of any other code-word. The code corresponding to a certain symbol is computed based on its frequency of occurrence in the stream, where more frequent symbols get encoded with smaller code words, while less frequent ones are attributed to bigger words.

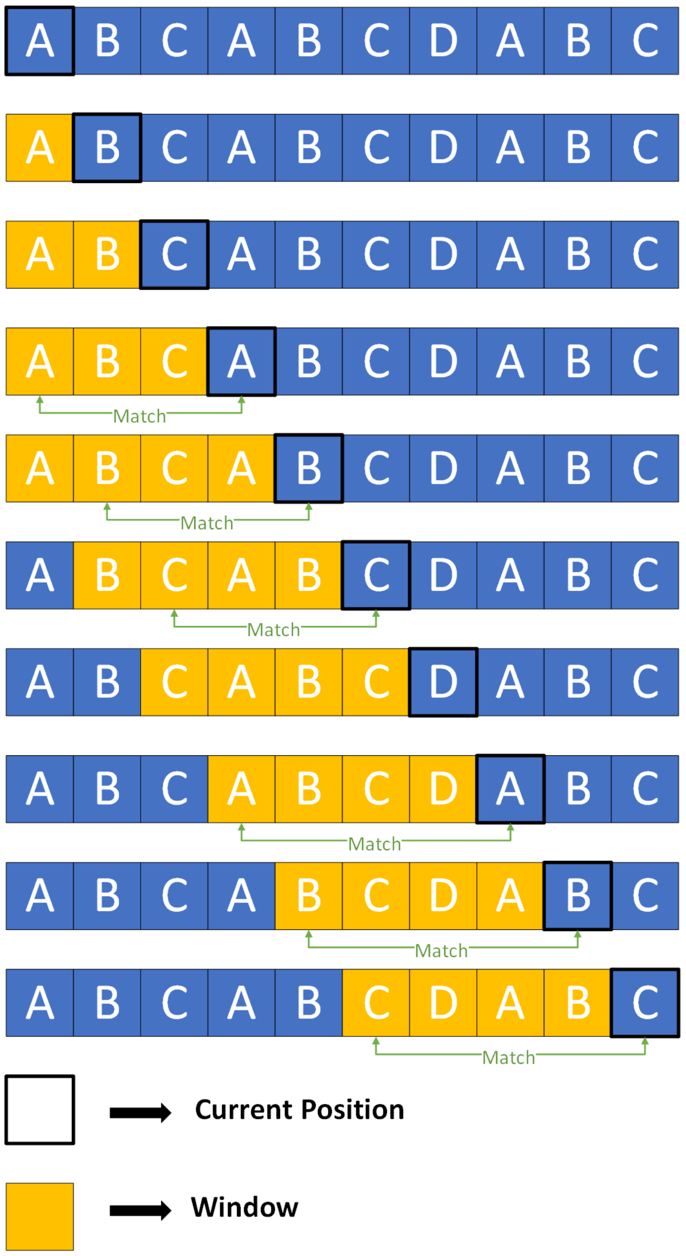

Fig. 3: Example of the LZ77 sliding window and matching behaviour.

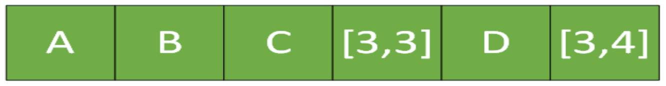

Fig. 4: LZ77 compressed stream .

This method of encoding symbols based on their probability is known as an entropy encoding method since it focuses on using smaller codes for more frequent symbols which translates in smaller output sizes. This is done using a binary tree to create the code, in a way that the most frequent values are placed closer to the tree's root, resulting in shorter binary codes for the most used symbols.

In DEFLATE, two variants of *Huffman coding* can be used, one where the codes have a fixed and predetermined length depending on what value they represent and another where the length of each code is computed based on the frequency of its symbol. Considering this work's main focus is to achieve high compression speeds, the fixed-length *Huffman coding* will be used and therefore the DEFLATE algorithm will be designated as *Fixed DEFLATE encoding*.

IV. LodePNG Program Implementation

The software that will serve as a starting point for this work is the LodePNG program [3], which is an open-source PNG encoder/decoder that has no dependencies on external libraries, meaning it can run in the targeted *RISC-V* processor [10] after making some small changes. The LodePNG encoder outline view is detailed in Figure 5.

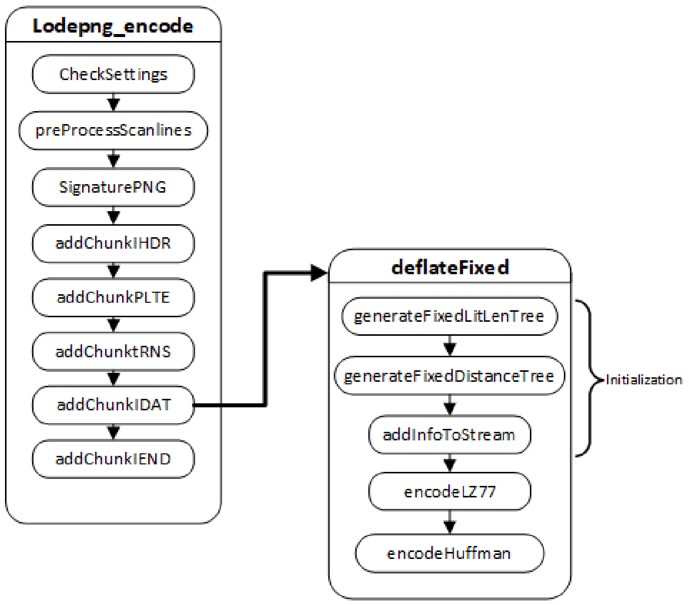

Fig. 5: LodePNG program overview.

First and foremost, it checks if the settings that were set can provide a valid image (`CheckSettings`), it then filters the image (`preProcessScanlines`) as described previously, providing a better compression ratio. After that, it adds the PNG signature chunk to the beginning of the output bit stream that will contain the compressed image (`SignaturePNG`) and it then inserts the IHDR, PLTE, and tRNS chunks after the signature and in the presented order (`addChunkIHDR`, `addChunkPLTE`, and `addChunktRNS`).

In order to create the IDAT chunk (`addChunkIDAT`), the Fixed DEFLATE algorithm is executed (`deflateFixed`), which is done by first initializing the trees that will generate the codes for the Huffman encoding (`generateFixedLitLenTree` and `generateFixedDistanceTree`), it then signals that the chunk is final and that it as been encoded using Fixed DEFLATE (`addInfoToStream`). The initialization of the trees and the adding of this information will be grouped as

`Initialization` for profiling purposes, since it runs at a constant time depending on the data raw size. After this, it then encodes the previously filtered data using the LZ77 algorithm (`encodeLZ77`) which is encoded again using Huffman coding (`encodeHuffman`) and the result is then added to the output data stream.

Now that the image as been compressed, the IEND chunk is added (`addChunkIEND`) to the final stream in order to signal the end of the image.

A. Program Profiling

The images that were used to test the LodePNG program can be divided into two types: five computer-generated patterns based on mathematical models that are provided as a benchmark by the program's developer and two images used in João Cardoso's thesis [11] profiling. All the images have 640 by 480 pixels, differing on which color palette they use (greyscale or RGB format), meaning that although they have the same number of pixels, their raw size is different since greyscale only uses one byte per pixel and RGB uses three bytes per pixel.

The generated images can be divided into *simple* patterns, where an obvious arrangement of pixels can be observed, and *complex* ones, where the pixel pattern appears to be random. These images offer the opportunity to study the effect of several variables that impact the program's performance, such as, the usage of different color palettes and compressing simple images versus complex ones.

The real-life photographs used for profiling the software allow to understand the impact that these type of images have on the algorithm. These pictures were chosen in order to stress the program since they offer a big variety of colors and no discerning pattern. They also allow the opportunity to compare the results obtained in this work with the ones collected in João Cardoso's thesis that uses the same algorithm, but that was implemented in an Advanced RISC Machine (ARM) CPU based system and uses dynamic Huffman encoding instead of a fixed implementation.

B. Profiling Results

The profiling of the algorithm was obtained by running the program for each of the previously shown images in the SoC RISC-V based processing system using the board's external memory that was developed by the IObundle team [12]. In order to account for how much each part contributes for the final runtime, a timing hardware IP, also developed by the same team [13], was used since it allows very precise timing results because it uses an internal clock cycle counter. The results that were obtained are summarized in Table II

From this outcome, it can be concluded that the *Filtering* and *IDAT* parts are the ones that contribute the most to the program's runtime (approximately 100%) and that are the ones that deserve closer inspection in order to achieve a higher FPS and all the other parts can be ignored. Since the *IDAT* chunk encoding process can be divided into three sub-parts (*Initialization*, *encodeLZ77* and *encodeHuffman*), some extra

TABLE II: LodePNG without IP time profiling.

		Part								
Image	**Value Type**	**Filtering**	**Signature**	**IHDR**	**PLTE**	**tRNS**	**IDAT**	**IEND**	**Total (μs)**	**FPS**
X	Absolute (μs)	3,055,110	10	40	10	10	2,595,810	10	5,651,000	0.18
	Relative	54.1%	0%	0%	0%	0%	45.9%	0%		
Y	Absolute (μs)	3,079,000	10	40	10	10	2,582,900	10	5,661,980	0.18
	Relative	54.4%	0%	0%	0%	0%	45.6%	0%		
Mandel	Absolute (μs)	3,091,220	10	40	10	10	9,425,070	10	12,516,370	0.08
	Relative	24.7%	0%	0%	0%	0%	75.3%	0%		
XOR	Absolute (μs)	9,411,390	10	40	10	10	16,544,170	10	25,955,640	0.04
	Relative	36.3%	0%	0%	0%	0%	63.7%	0%		
Sine	Absolute (μs)	9,400,470	10	40	10	10	37,446,870	10	46,847,420	0.02
	Relative	20.1%	0%	0%	0%	0%	79.9%	0%		
VGA-01	Absolute (μs)	9,476,120	10	40	10	10	31,975,350	10	41,451,550	0.02
	Relative	22.9%	0%	0%	0%	0%	77.1%	0%		
VGA-02	Absolute (μs)	9,452,340	10	40	10	10	34,465,140	10	43,917,560	0.02
	Relative	21.5%	0%	0%	0%	0%	78.5%	0%		

profiling was performed in order to understand which of these should be the main the focus since they are very complex processes.

TABLE III: IDAT chunk without IP time profiling.

		Part			
Image	**Time**	**Initialization**	**encodeLZ77**	**encodeHuffman**	**Total (μs)**
X	Absolute (μs)	237,700	2,133,210	49,830	2,420,740
	Relative	9.8%	88.1%	2.1%	
Y	Absolute (μs)	237,710	2,120,230	49,880	2,407,820
	Relative	9.9%	88.1%	2.1%	
Mandel	Absolute (μs)	237,710	5,917,150	2,733,890	8,888,750
	Relative	2.7%	66.6%	30.8%	
XOR	Absolute (μs)	237,700	15,677,520	189,380	16,104,600
	Relative	1.5%	97.3%	1.2%	
Sine	Absolute (μs)	237,700	31,830,720	4,379,350	36,447,770
	Relative	0.7%	87.3%	12.0%	
VGA-01	Absolute (μs)	237,700	19,225,680	10,664,330	30,127,710
	Relative	0.8%	63.8%	35.4%	
VGA-02	Absolute (μs)	237,710	23,071,790	9,470,460	32,779,960
	Relative	0.7%	70.4%	28.9%	

The results of this profile can be observed in Table III and from those, we can conclude that the LZ77 encoding algorithm has a bigger impact in the *IDAT* part (63.8%, in the best scenario) and is the best candidate for hardware acceleration.

Since the aim of this thesis is to achieve an equal or higher compression ratio than the one offered by the software non-accelerated implementation, the resulting compressed size and the deriving compression was also accounted for in Table IV.

TABLE IV: LodePNG without IP compression ratios.

Image	**Raw Size**	**Compressed Size**	**Reduction**	**Compression Ratio**
X	307,200	3,128	0.010	98.21
Y	307,200	3,131	0.010	98.12
Mandel	307,200	170,486	0.554	1.81
XOR	921,600	11,526	0.013	79.96
Sine	921,600	270,306	0.293	3.41
VGA-01	921,600	662,664	0.719	1.39
VGA-02	921,600	592,532	0.637	1.57

From this results, we can infer that images that have simpler patterns provide better compression ratios (in average, the final image is approximately 92 times smaller than the original one) while more complex ones offer worse compression ratios (in average, the final image is only 2 times smaller).

Taking these conclusions into account and that the *Filtering* part and *encodeLZ77* are sequential, a hardware IP accelerator was devised comprising these two parts. Since the PNG standard uses an Adler-32 checksum [14] to validate an encoded image that is calculated from the filtered data, an extra Adler-32 calculator was added to the accelerator, also providing a small acceleration that was not discriminated in the profiling and is present in the *Filtering* part.

Since João Cardoso's thesis also uses the LodePNG algorithm as an implementation basis, but is implemented in an ARM processor based system and uses dynamic Huffman encoding instead of a fixed implementation, it is relevant to compare the results obtained in both profiling for the VGA-01 and VGA-02 images that are used in both works. Since the Filtering part is the same in both implementations it can easily be observed that the RISC-V system is much slower. This is then escalated in the final runtime, meaning that this processor achieves much less FPS (around 50 times less). The final compression is also worse, this is due to the use of the fixed Huffman encoding that generates bigger codes for the symbols used while providing a faster compression.

V. IP Core Architecture

The overall architecture of the IP core designed to accelerate the compression process can be observed in Figure 6.

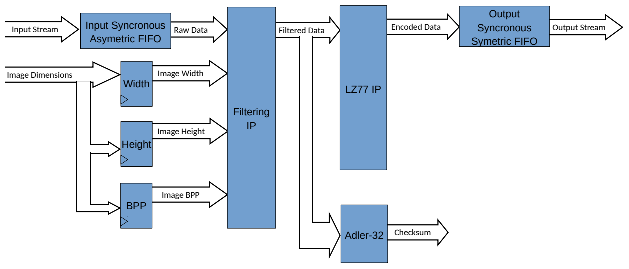

Fig. 6: Accelerator top architecture.

The core has two main data inputs, the *Image Dimensions* which sets the *Width*, *Height* and *Bytes Per Pixel (BPP)* registers, and the *Input Stream* which receives all the image raw data. First, the dimensions must be set, or the accelerator will not work correctly, and only then the *Input First-In First-Out (FIFO)* module can start to receive unprocessed data.

Once the FIFO has at least one byte of information, the *Filtering IP* starts to process data from it. After one line of the image has been filtered (hence the need for image dimensions), it signals the *LZ77 IP* that it has results ready to be compressed. When the compressor is free, it starts reading the filtered bytes until the filter block signals it that the result has ended. This process repeats itself until the entire filtered image has been passed.

At the same time that the *LZ77 IP* receives its input data, so does the *Adler-32* block which computes the image checksum, this value will only be valid after the entire image has been filtered and transmitted to the compressor.

When the *LZ77 IP* has a valid encoded result, it is written to the *Output FIFO*. After the image has been processed, the encoded data can be read through the *Output Stream* and the *Checksum* can be retrieved.

In order to communicate with the CPU, the ports presented in Table V were used.

TABLE V: Core ports.

Port	Type	Size (bits)	Description
`clk`	`input`	1	Receives the clock signal
`reset`	`input`	1	Resets the entire core
`address`	`input`	3	Addresses the internal registers
`wdata`	`input`	32	Receives the write data
`valid`	`input`	1	Validates the write data
`write`	`input`	1	Signals a write operation to the internal registers
`read`	`input`	1	Signals a read operation from the internal registers
`rdata`	`output`	32	Outputs the data from a read operation
`ready`	`otuput`	1	Signals that the core is free to perform an operation

A. Internal Registers

The core's internal registers that handle the communication between the encoder and the CPU are mapped as shown in Table VI .

TABLE VI: Encoder core internal 32-bit registers.

Name	R/W	Addr	Description
ENCODER_SOFT_RESET	W	0	Triggers the block's soft reset.
ENCODER_W	W	1	Data input to FIFO for transmission.
ENCODER_R	R	2	Data output from FIFO for reception.
ENCODER_FULL	R	3	Reads 1 when input FIFO is full, otherwise 0.
ENCODER_EMPTY	R	4	Reads 1 when output FIFO is empty, otherwise 0.
SET_IMG_PARAM	W	5	Sets the image parameters.
GET_ADLER32	R	6	Returns the Adler-32 checksum.
GET_OCUPANCY	R	7	Returns the output FIFO's occupancy.

B. Image Dimensions

The image dimensions are set by writing the desired values into the three registers shown in Figure 6. In order to reduce the transmission overhead, this writing is done in only one transaction by sending the three dimensions concatenated in a 32-bit value, which is then split as shown in Table VII and written into the respective register. The size used for each dimension was defined to accommodate large images (4095 by 4095 pixels) with RGB colors (three BPP).

TABLE VII: Image dimensions arrangement.

Dimension	Size (bits)	Input Bus Section
Width	12	[11:0]
Height	12	[23:12]
BPP	2	[25:24]

C. Input FIFO

The chosen buffer for the input data stream is a simple standard synchronous and asymmetric FIFO implemented using a Random-Access Memory (RAM) for the internal memory. Its write and read interfaces are detailed in Table VIII.

TABLE VIII: Input FIFO interfaces description.

Interface	Port	Type	Size (bits)	Description
Write	`data_in`	`input`	32	Receives data to be written
	`full`	`output`	1	Signals if the FIFO is full
	`write_en`	`input`	1	When `1`, if not full, performs a writing (1 clock cycle delay)
Read	`data_out`	`output`	8	Outputs the data read
	`empty`	`output`	1	Signals if the FIFO is empty
	`read_en`	`input`	1	When `1`, if not empty, performs a reading (1 clock cycle delay)

The *Write* interface is operated by the CPU and the *Read* interface is commanded by the *Filtering IP* and are the only ones that influence the FIFO.

D. Filtering IP

The architecture of the *Filtering IP* that performs the pre-compression of the image can be observed in Figure 7.

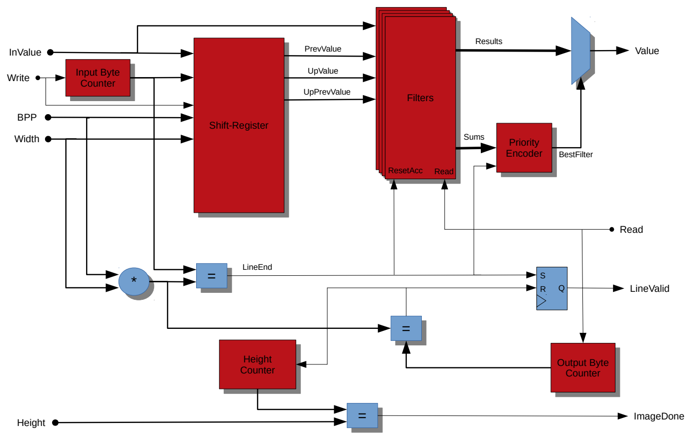

Fig. 7: Filtering IP architecture.

This block receives the image parameters (`Width`, `Height` and `BPP`) and the input value (`InValue`). In order to calculate how many bytes are in each line, the line width must be multiplied by the number of bytes per pixel.

When the input FIFO is not empty, it sends a stream of unfiltered values to this IP which are then inserted in the *Shift-Register* so that the values needed to filter the image (`PrevValue`, `UpValue` and `UpPrevValue`) can be retrieved by the *Filters* block. When a new value is read from the FIFO, the *Input Byte Counter* is incremented.

Once the line has been completely written to the Shift-Register, they have also all been filtered and are now available in the *Filters* internal buffers (the block has no delay). The sum accumulators are then reset and the `LineValid` is set to `1`, which signals the *LZ77 IP* that the data equivalent to a filtered line can be read from the multiplexer, which uses the `BestFilter` signal to select the result that corresponds to the smallest sum. One cycle after the line transfer has finished, the *Input Byte Counter* is reset.

The *Filters* block is composed of five different independent sub-blocks, that share the same input values, but have different outputs. Each sub-block has its sum accumulator and internal buffer and differ in which operation filter is applied to the input value

The *Priority Encoder* finds the smallest sum and outputs the correct filter type to the multiplexer. The chosen filter only changes when `LineEnd=1`.

When the *LZ77 IP* finishes reading a filtered line, the *Output Byte Counter* is reset and the *Height Counter* is incremented.

The whole process starts again and repeats itself until the *Height Counter* is the same as `Height`. When this happens, the `ImageDone` signal is set so that the *LZ77 IP* outputs its final results.

1) Filters: This block is composed of five different sub-blocks that are similar in their structure but differ in how they compute the filtered result. This structure is shown in Figure 8.

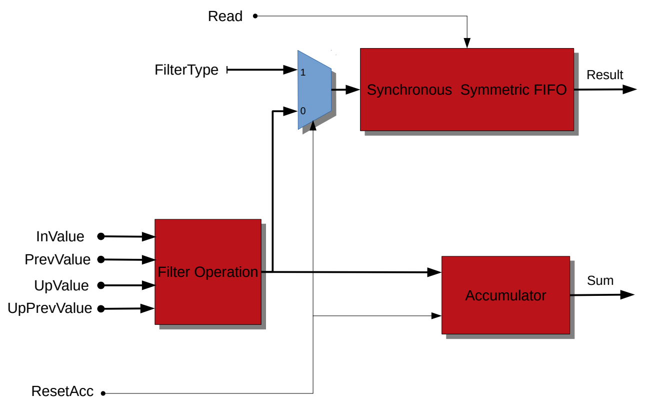

Fig. 8: Individual filter architecture.

The previously referred internal buffer is implemented with a synchronous symmetric FIFO with a width of height bits and enough depth to hold two RGB lines with 640 pixels ($640 \times 3 \times 2 = 3840$ positions).

`FilterType` is a predefined constant that is specific to each filter block and that will be inserted at the beginning of each line in the internal FIFO.

The *Filter Operation* depends on which filter is being implemented in the sub-block. After the value has been filtered by this block, it is written to the FIFO and accumulated in the *Accumulator*.

Whenever a line ends, the sum accumulator is reset and the filter type is inserted into the FIFO signaling that a new line filtering will begin soon.

E. Adler-32 IP

The *Adler-32 IP* is a simple block that is composed of two 32-bits sum accumulators and two serial divider blocks and its architecture is displayed in Figure 9.

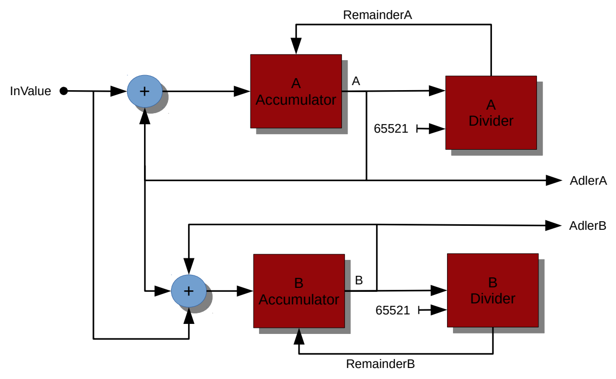

Fig. 9: Adler-32 architecture.

The sum accumulators, A and B, are initialized with 1 and 0, respectively, and accumulate the filtered values sent by the *Filtering IP*. The *Accumulator A* only sums the previous `A` value with the input value, while the *Accumulator B* sums the `B`, `A` and input values.

At the end of each filtered line (signaled by the filter) the accumulated values are divided by 65521 in the divisors and the remainders (*RemainderA* and *RemainderB*) are inserted into the corresponding accumulator. This process repeats itself until the entire image has been processed.

It is important to notice that even though the serial dividers have a significant delay, the delay between each line transfer is much bigger and so does not need to be taken into account.

The final `checksum` value is computed by retrieving the 16 less significant bits from each accumulator and concatenating them into a 32-bit result, being the `AdlerA` the 16 less significant bits and `AdlerB` the 16 most significant ones.

F. LZ77 IP

The *LZ77 IP* implements the LZ77 encoding part of the DEFLATE algorithm and its architecture is presented in Figure 10.

The IP reads one byte at a time (`Read=1`) from the *Filtering IP* if the output FIFO is not full (`FIFOFull`$\neq$`1`) and if there are valid values to be read (`LineValid=1`). This byte is then compared with the others present in the *Shift-Register*. In the next cycle, the byte is inserted into the *Shift-Register*, mimicking the sliding window mechanism.

For every byte in the *Shift-Register* that equals the input byte, the *Comparator Array* outputs which positions have corresponding matches. For example, if only the byte present in the second position of the *Shift-Register* equals the input value, then only the second less significant bit of the output will be "1".

The result of the comparison with the flagged positions is then fed to the *Matcher* block, which compares the previous matches with the current ones. If at least one position is set in both matches, then the match keeps going, otherwise, there are

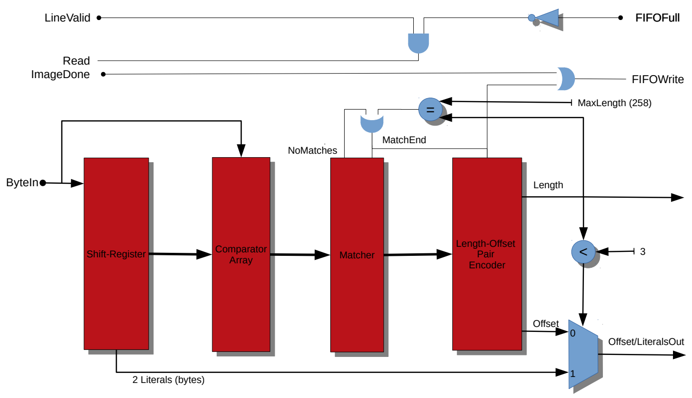

Fig. 10: LZ77 IP architecture.

no more matches and the sequence has ended. This process will be presented in more detail in subsection V-F1.

At last, the *Length-Offset Pair Encoder* computes the sequence's length and offset. The length part is calculated with the use of a counter that increments each time a new byte continues the sequence and is reset when there are no more matches. The offset is computed by feeding the result from the *Matcher* to a *Priority Encoder* that returns the first position that is set, which corresponds to the smallest offset.

A length-offset pair is written to the output FIFO (`FIFOWrite=1`) when a sequence ends, which happens when there are no more matches or when the maximum length has been achieved. When the image ends, a sequence end is mimicked by the `ImageDone` signal, or otherwise, the last sequence would not be written to the output since there are no more valid values that can signal a match end.

1) Matcher: The *Matcher* block is depicted in detail in Figure 11. It starts by receiving the comparisons computed in the *Comparator Array*, if the match has ended (`MatchEnd=1`), then the current matches are not compared with the previous ones and are outputted by the block.

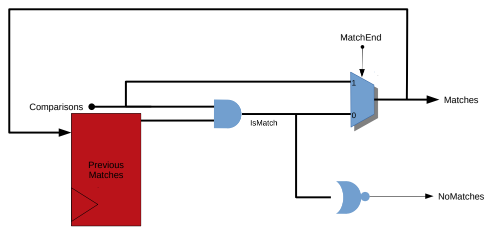

Fig. 11: Matcher block architecture.

If the sequence continues (`MatchEnd=0`), then the current matches are compared to the previous ones through a bitwise `AND` gate and the result of this operation is outputted. If no match continues the sequence (`NoMatches=1`), which is computed by the `NOR` logical gate, then the end of the match is signaled and the new comparisons are outputted instead.

The final result is always saved in the presented register, so it can be compared in the next computing cycle.

G. Output FIFO

The *Output FIFO* which is used as a buffer between the encoded results and the CPU is a synchronous symmetric FIFO, contrary to the asymmetric input one since it has a `data_in` and `data_out` with the same size of 32 bits. Its *Write* interface is operated by the *LZ77 IP* and the *Read* interface by the processor.

The CPU will read all the values present in this FIFO and encodes them in the final image according to their length, if it has a length smaller than two, then the two most significant bytes represent two literals, otherwise, the result is length-offset pair.

VI. Results

This work was developed on a Xilinx Kintex UltraScale FPGA board [15] [16] using the IOb-SoC developed by the IObundle team [12], which comprises a PicoRV32 CPU processor, an Static Random-Access Memory (SRAM) memory subsystem, a Universal Asynchronous Receiver-Transmitter (UART), and an Advanced eXtensible Interface (AXI)-4 connection to an external Double Data Rate (DDR) memory. The *Timer* component was added to the time profile of the algorithm.

The *LZ77* IP window width was set to 2048 bytes, which is the recommended size by the LodePNG program. This value can be changed to any power of two. The compression ratios directly depend on the window size, but an increase to the next available dimension (4096 bytes) has shown no significant increase in these ratios while doubling the resources needed to implement the IP.

A. Performance Results

In the same faction as the time profiling was done for the original LodePNG program, the same was done for the accelerated version, which is shown in Table IX. In order to account for the communication overhead between the CPU and the IP, the `SendRecv` part was added to the profiling and is responsible for timing how long the program spends sending, receiving, and processing the data to and from the CPU.

For the IP accelerated system a significant increase, over the non-accelerated version results (presented in Table II), in the FPS rate has been observed in all the images. In the worst-case scenario (*VGA-01*) the FPS has increased by 2.82 times and in the best case (*XOR*) is increased by 28.08 times.

In Table X the compressed sizes and equivalent compression ratios are shown.

After comparing these results with the ones presented in Table IV it was concluded that the case of the *XOR* image the

TABLE IX: LodePNG with IP time profiling.

		Part								
Image	**Time**	**SendRecv**	**Signature**	**IHDR**	**PLTE**	**tRNS**	**IDAT**	**IEND**	**Total (μs)**	**FPS**
X	Absolute (μs)	340,350	10	40	10	10	57,210	10	397,640	2.51
	Relative	85.6%	0%	0%	0%	0%	14.4%	0%		
Y	Absolute (μs)	340,390	10	40	10	10	57,400	10	397,870	2.51
	Relative	85.6%	0%	0%	0%	0%	14.4%	0%		
Mandel	Absolute (μs)	754,410	10	40	10	10	3,184,350	10	3,938,840	0.25
	Relative	19.2%	0%	0%	0%	0%	80.8%	0%		
XOR	Absolute (μs)	568,250	10	40	10	10	356,040	10	924,370	1.08
	Relative	61.5%	0%	0%	0%	0%	38.5%	0%		
Sine	Absolute (μs)	1,720,600	10	40	10	10	4,903,070	10	6,623,750	0.15
	Relative	26.0%	0%	0%	0%	0%	74.0%	0%		
VGA-01	Absolute (μs)	2,324,860	10	40	10	10	12,393,780	10	14,718,720	0.07
	Relative	15.8%	0%	0%	0%	0%	84.2%	0%		
VGA-02	Absolute (μs)	2,403,700	10	40	10	10	11,038,540	10	13,442,320	0.07
	Relative	17.9%	0%	0%	0%	0%	82.1%	0%		

TABLE X: LodePNG with IP compression ratios.

Image	Raw Size	Compressed Size	Reduction	Compression Ratio
X	307,200	2,491	0.008	123.32
Y	307,200	3,502	0.008	122,78
Mandel	307,200	170,486	0.556	1.80
XOR	921,600	18,608	0.020	49.53
Sine	921,600	262,423	0.285	3.51
VGA-01	921,600	665,408	0.719	1.39
VGA-02	921,600	587,517	0.637	1.57

compression ratio is much smaller, almost half, this is due to a special case of the Huffman algorithm where it encodes the values with a larger number of bits than the original program. In the *X* and *Y* patterns there is an increase in the compression ratio, which is due to the bigger number and size of the offset-length pairs that are found by the *LZ77 IP*. In the remaining images, the compression ratio has no significant changes.

Simply by looking at timing results, it can be observed that the overall execution times and IDAT chunks encoding times were greatly reduced by the accelerator IP. In the case of the total execution time, a significant decrease in the IDAT chunk processing can be observed. The *SendRecv* process now occupies a small portion of the program that was originally the `Filtering` and `encodeLZ77` processes.

In Table XI the calculated FPS acceleration and compression ratios increase are resumed for simpler final analysis. From these results, it can be observed that there is a considerable increase in the number of FPS, especially for the more simple patterns (*X*, *Y* and *XOR*). The obtained acceleration is satisfying since now the program's execution time greatly depends on the Huffman encoding algorithm.

TABLE XI: Speedup and compression ratio increase.

Image	Speedup	Compr. Ratio Increase
X	14.21	1.3
Y	14.23	1.3
Mandel	3.18	1.0
XOR	28.08	0.6
Sine	7.07	1.0
VGA-01	2.82	1.0
VGA-02	3.27	1.0

VII. Conclusions

The best option found for the PNG encoding software was the LodePNG program since it is open-source and does not rely on external libraries. To better understand the program's choke-holds, it was run for several patterns and real images, and its execution time was profiled. With these results, it was possible to understand that there were three main culprits for the resulting low FPS rates: the image *pre-compression* process, the *LZ77* algorithm, and the *Huffman* encoding. Since, of these three, the *LZ77* algorithm was the one that took the biggest time percentage, it became the best candidate for hardware acceleration. After it was implemented, the resulting FPS rates still were not satisfactory, so the *Filtering* process has also been accelerated.

A. Achievements

The open-source LodePNG software was used to compress several images and its main time bottlenecks were identified. From there, it was implemented an encoder IP that comprise the *Filtering* process, the *Adler-32* checksum calculation and the *LZ77* lossless data compression algorithm. After the resulting images were correctly compressed, the program was profiled again and the final results were recorded.

The final resource utilization was also reported and it was concluded that the IP uses a reasonable amount of resources in spite of the fact the encoder needs to process a big amount of data and performs some complex operations.

Although the 30 FPS objective was still not achieved for real-life VGA colored images, the results were still satisfactory, especially for simple images (such as the *X* and *Y* patterns), where an acceleration of more than 14 times was achieved. The final compression ratio results were very solid since they matched those of the original software and, in some cases, even improved.

These results showed that the system has increased acceleration on simple images with repeating patterns or that have a small color palette, which are more likely to use the accelerated parts. In less regular images, where the entropy coding portion is more important, no hardware acceleration has been provided and thus the performance is poor.

B. Future Work

To achieve a 30 FPS rate or higher, several work fronts can be explored. The first and simplest one is to use a faster processor and a a faster way for the IP to access memory, such as the use of a Direct Memory Access (DMA) unit, instead of the input and output FIFOs driven by the processor.

The second front that can be explored is the acceleration of the Huffman encoding algorithm (entropy coding) in hardware, and using its dynamic algorithm instead of the fixed version which is simpler but less accurate. Besides increasing the FPS figure, it would also improve the compression ratios. This is a very complex undertaking and may be the object of another thesis.

The final work front is to implement a system with several parallel encoders controlled by different CPU cores that encode separate images at the same time. This would not increase the FPS rate for each image stream but would improve the FPS over all image streams.

References

[1] et. al. G.Randers-Pehrson. Png (portable network graphics) specification, version 1.2, July 1999. [Online] Available at: http://www.libpng.org/pub/png/spec/1.2/png-1.2.pdf.

[2] Incorporated CompuServe. G i f (tm) graphics interchange format (tm) a standard defining a mechanism the storage and transmission of raster-based graphics information. [Online] Available at: https://www.w3.org/Graphics/GIF/spec-gif87.txt.

[3] Lode Vandevenne. Lodepng for c (iso c90) and c++. [Online] Available at: https://lodev.org/lodepng/.

[4] W3. Png filters. [Online] Available at: https://www.w3.org/TR/PNG-Filters.html.

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

[6] Abraham Lempel Jacob Ziv. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[7] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.

[8] Pkzip. [Online] Available at: https://www.pkware.com/pkzip.

[9] Gzip. [Online] Available at: http://www.gzip.org.

[10] CS Division, EECS Department, University of California, Berkeley. *The RISC-V Instruction Set Manual*, document version 2.2 edition, May 2017.

[11] João Gonçalo Esteves Freire Cardoso. Verilog png encoder.

[12] Iobundle soc. [Online] Available at: https://github.com/IObundle/iob-soc.

[13] Iobundle timer module. [Online] Available at: https://github.com/IObundle/iob-timer.

[14] P. Deutsch and J.-L. Gailly. Zlib compressed data format specification version 3.3. *Network Working Group Requestfor Comments (RFC)*, May 1996.

[15] Xilinx kintex ultrascale fpga board. [Online] Available at: https://www.avnet.com/shop/us/products/avnet-engineering-services/aes-ku040-db-g-3074457345635221623/.

[16] Xilinx. *UltraScale Architecture and Product Data Sheet: Overview*, document version 3.13 edition, July 2020.