Submicrosecond Latency Video Compression in a Low-End FPGA-based System-on-Chip

Tobías Alonso, Mario Ruiz, Ángel López García-Arias, Gustavo Sutter, and Jorge E. López de Vergara
High Performance Computing and Networking Research Group
Escuela Politécnica Superior, Universidad Autónoma de Madrid, Spain
{tobias.alonso, mario.ruiz, angel.lopezg, gustavo.sutter, jorge.lopez_vergara}@uam.es

***Abstract*—In this paper, we present an efficient hardware implementation of a video encoder optimized for ultra low-latency, using the Logarithmic Hop Encoding algorithm. This design provides the following features: (i) A maximum marginal output latency of 23 clock cycles, (ii) small area requirements, (iii) proven rate up to 95 Millions of pixels per second in a low-end FPGA (*i.e.* FHD video can be streamed), (iv) on-the-fly configuration, (v) scalable architecture. The proposed design has been tested in a real video transmission scenario, where the video transmitter prototype is implemented using a ZynqBerry board, leveraging all SoC capabilities.**

***Keywords*-Video Codec; Logarithmic Hop Encoding; SoC**

I. Introduction

Nowadays, computer vision applications for control systems are flourishing due to higher bandwidth links, better and cheaper camera sensors and more fcomputational power. When the camera system is part of a feedback loop the latency of the video system is critical, since increasing amounts of it decreases the stability of the system.

In general, commercial cameras capture 10 bit Bayer filtered images producing a very large data rate, which would need a great deal of effort to design bandwidth efficient and robust radio links to transmit it. However, using video compression would allow using standard links and reducing the width of the bands to accommodate at expense of added latency. The problem here is that state-of-the-art codecs tend to fail in achieving very low latency using low-end devices.

Nevertheless, a new encoding algorithm, the Logarithmic Hop Encoding (LHE) [1], has been created, achieving an $O(n)$ time complexity and low memory requirements. We leverage this algorithm to create a low footprint FPGA design that exploits hardware benefits, accomplishing an architecture with on-the-fly configurations, capable of encoding at line rate adding just minimum latency.

The rest of this paper is structured as follows: First of all, section II presents the related work. Section III provides a brief description of the LHE algorithm, and then, it states the system requirements and the methodology used to achieve them. Next, the encoder architecture details are given in section IV and, after that, section V dives into the testing methodology. Later, section VI presents the implementation results: footprint and latency. Finally, section VII summarizes our contributions, conclusions and future work.

II. Related work

Since LHE is a relatively new algorithm, there are not prior hardware developments related to it. However, several systems implementing video encoders, especially H.264, were designed for FPGA and/or ASICs, aiming at low latency.

For instance, H. Loukil *et al.* [2] proposed an FPGA implementation of H.264 intra-frame encoder clocked at 100 MHz, being able to process one Macroblock in 573 cycles. Encoding latency for a camera video stream is not computed in this work, however it should be around the transmission time of 16 lines (in the case that the video source is a camera) plus the processing time. This is accomplished by using 28.5K LUTs, 32 KB of memory and 535 DSP block of an ALTERA Stratix II board.

Another approach is that from Fraunhofer HHI [3], which offers an H.264 intra-frame encoder implementation with a latency below one macroblock line, which is at least 3 ms. It allows up to 1080p video with a system clock of 145.8 MHz, requiring 170K gates and a small on-chip memory.

SoC Technologies [4] also commercializes an H.264 encoder, able to reach a latency below 0.25 ms, supporting up to 4K @ 120 FPS. Although it is able to get a high compression rate since it supports P frames and high profile, this design is not suitable for current low-end FPGAs, since it needs 103K LUTs, 287 36Kb Block RAMs and 302 DSPs for the case of Xilinx devices.

A lower footprint solution, suitable for small devices, was designed by A2e technologies [5]. It runs at 115MHz in Zynq 7020-1, has a 1.5 clock/pixel processing rate, supports P frames and baseline profile with a latency below 1 ms for a 1080p @ 30 FPS video stream (system clock for latency measure is not stated).

Xilinx [6] also provides an integrated solution for H.264 and H.265 in its MPSoC devices. This Hard IP has a 16.6 ms latency and supports 3840x2160 @ 60 FPS with group of pictures. But currently, some projects cannot afford the price of these devices.

As we will see, when compared with the works presented above, our proposal achieves a latency that is several orders of magnitude lower than the current best approach, with an architecture suitable for low-end devices.

III. Compression algorithm and Design Methodology

A. LHE algorithm

LHE [1] is a lossy codec with an $O(n)$ time complexity that works directly in the space domain (meaning that no domain transformation is needed). There are several versions of the codec, choosing the LHE basic algorithm, since the presented work is the first proof of concept of a hardware implementation. It performs simple operations and relies only in the previous pixel and previous line, which makes it attractive to encode camera streams with low latency and low memory requirements.

To have a more robust transmission scheme, we divide each frame in blocks. The defined protocol does not set the size of these blocks, but it allows them to be configurable, being up to the system designer to manage the trade-offs. In this way, each block is encoded as a whole image according to LHE algorithm, and the encoded binary is composed by a global header, stating image and encoder parameters, followed by encoded blocks with their local headers.

B. Design Methodology

The design presented in this paper was implemented in the context of a larger project, which aimed to enable digital video transmission for FPV racing drones. In this scenario, not only a light, cheap and low power system is needed (each drone will be carrying one) but also an efficient processing has to be achieved to meet the latency requirement. The first milestone was to be able to process 640x480 px @30 FPS video stream, however achieving higher resolution (FHD) and FPS (60) was set as a middle-term goal.

We started writing a C/C++ version to validate the algorithm to be implemented in hardware. Since this is a codec in a development phase, our first approach was using high-level synthesis (HLS) [7] to implement the encoding pipeline, to be able to adapt faster to future versions. Besides, this lets us iterate through different architectures, trying to meet the system requirements.

However, with that approach, we were not able to get the desired area and performance. In part, because of the use of a low-end target. After an analysis, we spot resources that could be shared, and low-level optimizations that shortened the critical path. In addition, we saw that some long combinational paths could be turn into multi-cycle paths, greatly improving the system performance. For this reason, we proceeded to create an RTL design of the pipeline stages, utilizing Vivado HLS when it was considered appropriate, and using the C/C++ implementation to validate RTL simulation results.

To improve the reusability of the RTL modules, in case of changes in the encoding algorithm, simple tasks were assigned to each stage and a great level of decoupling was sought. Moreover, each module needed the minimum pre-synthesis parameters and configurations once implemented. This was accomplished with auto-adapt mechanisms and well defined interfaces and signals with simple and straightforward meaning and purpose.

IV. Architecture Overview

As it is shown in Fig. 1, the codec pipeline is integrated by 4 or 5 modules, depending if transmission to main memory is required (to be post processed by software). AXI4-Stream is used in order to stream the data, whereas, AXI4-Lite is used to configure the encoder parameters. The LHE encoder takes a stream of RGB pixels (one pixel at a time) with a Start of Frame (SoF) signal and either outputs N binary streams (where N is the maximum number of block columns) or interfaces to a DMA module.

First, the RGB pixels are mapped to YUV color space by the *RGB2YUV*. Then, the *FPS Handler* synchronizes its output so that the next modules do not receive fractions of frames. The *LHE quantizer* is the next module. Being the heart of the system, since it generates the hops from the YUV pixel stream. Finally, the *Entropy encoder* maps hops into variable length codes and aligns them in a configurable width output word. If the compressed stream is needed on a memory, the *Block Mover* module is used for this purpose.

Some parameters can be configured on the fly by using an AXI4-Lite interface, such as the FPS, switch between chrominance modes, block size and frame size (up to a limit configurable before synthesis). This information is stored in the *Configuration registers* when the synchronization signal is activated. In addition, this module formats the global header to be transferred to the memory. The system is able to support any image size up to a configurable maximum image width limit and block size limits are also configurable.

A. RGB to YUV

The function implemented by this module is a simple point operation consisting of a multiplication of a 3D vector and a 3x3 matrix. To improve performance and reduce area consumption, the integer operations [8] version were used instead of floating point ones, at the expense of a ± 1 error between the original RGB and the result of transforming to YUV and back to RGB space.

Due to the nature of this module functionality, it was written and tested in C++ to speed up the development time, using Vivado HLS to get the HDL code. This lets us to obtain a 5-stage pipeline module using dedicated DSPs in just one day, achieving a period of 5.1 ns (Vivado HLS estimation) with an II=1.

B. FPS Reduction & Sync

This module can be configured on the fly to divide the frame rate and stops the frame flow when the receiving system is not ready. Other important role of this module is to act as the configuration synchronizer, so all the codec modules see the same configuration despite of the moment configuration had changed, avoiding data corruption. Also is in charge of filtering frame fractions by synchronizing its output stream with the start of frame.

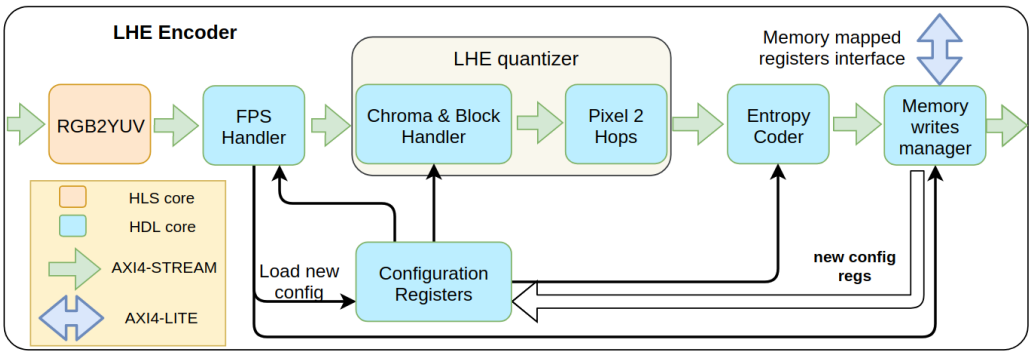

Fig. 1: LHE system.

C. LHE Quantizer

This component is subdivided in two submodules: *Chroma & Block Handler* and *Pixel2Hop*. The latest needs some pixel meta-data to operate, which is calculated by *Chroma & Block Handler* according to the chosen chroma sub-sampling scheme, frame and block size.

1) Chroma & Block Handler: The column block number, the number of bits required to code this number ($log_2(number\,of\,block\,columns)$) and the relative position of the pixel within the block are needed to compute the hop. To set well defined memory areas in the previous row buffer used by each image block (several blocks can be in the encoding process by one Pixel2Hop concurrently), the column block number and the number of bits to code it are used to divide the buffer and choose which address can be selected. Meanwhile, the relative position is needed to compute the prediction.

In addition, the *Chroma & Block Handler* chooses which YUV channels should be used at any moment according to the chroma-subsample scheme. The supported chroma sub-sampling schemes are YUV 422, YUV 420, and grayscale. YUV420 is supported even using 1-pixel height blocks.

2) Pixel2Hop: In this module is where the LHE main algorithm takes place, and its inner structure is shown in Fig. 2. When the input control accepts a new pixel, an output command is inserted in the FIFO. This command is read by the output control module, which will synchronize the output of the channels and composing the output data accordingly. The custom 3-port ROM memory, acting as the LHE cache, is implemented with a 2-port Block RAM, where one of its ports is shared by the chrominance channels using round robin scheduling.

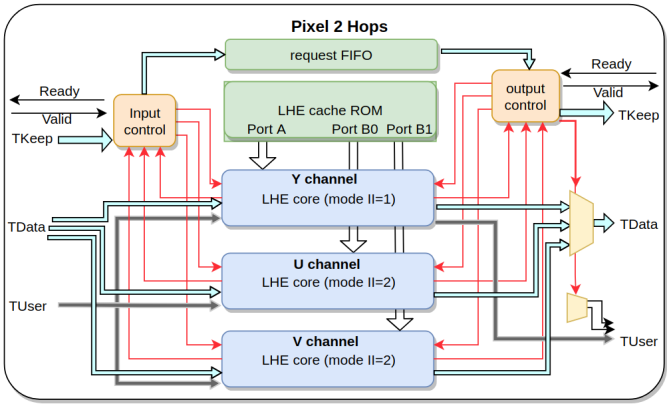

Fig. 2: Pixel2Hops Overview

Each LHE Quantizer Core takes one YUV channel and is itself a pipeline with a feedback between two of its stages due to the nature of the LHE algorithm. This feedback creates a bottleneck for the design clock frequency since data has to go through several logic levels.

To improve the performance of the design, we can take advantage of the fact that in the worst case the chrominance channels accept new data every other clock cycle. This is why the module was designed to operate in two modes, II=1 and II=2. The last one has the advantage that can share the LHE cache and it is able to operate at a much higher clock frequency, which is achieved by using multi-cycle paths. The use of the II=2 mode for the chrominance channels relaxes the time constrains for these modules, allowing a better placement and routing of the II=1 core used for the Y channel and, in this way, improving the overall performance.

Another key aspect is that there are several ways to choose the values in the LHE cache ROM. The prediction and the $\alpha(x)$ were selected as a compromise between ROM size and combinational path length.

D. Entropy Encoding

There is not a well-defined entropy encoding scheme for LHE hops. Thus, for this first prototype, a simple mapping of hops to variable length codes was chosen using a Huffman code. For a first approximation, this module is composed of N entropy encoder cores (to be able to interleave N blocks) and an AXI4-Switch [9] (acting as a demux), being able to process a stream of interleaved pixels of different blocks. Also, the cores have a pipeline architecture decoupling the symbol mapping from the word composing task, allowing to easily change the encoding scheme.

After the last word of each block is sent, a local header is generated to convey the information required to handle the block. In addition to this, every time a new frame arrives, the global header is transmitted.

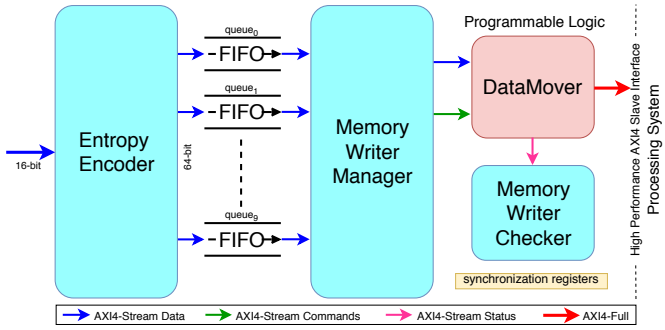

Fig. 3: Block Mover System.

E. Block Mover Subsystem

Once the compression has been done, those data have to be moved to the Processing System (PS) memory. We chose an AXI Data Mover alternative in order to manage the whole flow within the Programmable Logic (PL) [10], so, the consumer does not have to worry about programming DMA transactions. Each block has a reserved area of 1360 bytes in main memory —the worst scenario when no compression is attained— which is independent of the size and only depends on the relative position within the frame.

Fig. 3 shows a high-level overview of *Block Mover* system. Each block is being stored in the queue_n regarding its relative position in the image. Ten queues are implemented, but the maximum number of used queues depends on the user settings.

The FIFOs are read in a round robin fashion, when a complete block is consumed the following FIFO is read and so on. Communication between *Memory Writer Manager* and *Data Mover* is done using two AXI4-Stream ports, one for data and the other for commands. Commands tell the offset address where the data have to be written. A FSM was implemented to create the AXI4-Stream commands.

We implemented a robust synchronization mechanism between software and hardware through the *Memory Writer Checker*. This module receives the status of each programmed transaction and verifies if was done properly. Each successful transaction increments the hardware pointer and each consumed block increments the software pointer (yellow boxes in Fig. 3). If the software cannot keep up with the rate, data will be dropped to not overwrite the memory.

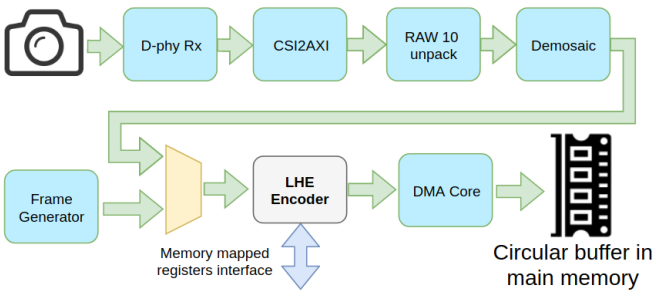

Fig. 4: Hardware testing.

V. System Testing

The verification of the designed system was made in two steps. First, a components wise and system HDL simulation, and second, the complete FPGA-based SoC using a camera as video source, Programmable Logic (PL) compressing data, and a user Linux program running in Processing System (PS) sending the data to the network.

A. Simulation Tests

The first tests were made through per module test-benches following a traditional development flow. Then, a more complex verification was done to the composed system in order to ensure all the interactions. This was accomplished by using a C++ software codec that feeds the system with provided image datasets, compiles the test-bench sources, launches the simulation and gathers the output. Golden data is generated for the datasets using a software version of LHE encoder and are compared with RTL simultaneously. When a difference is spotted, verification is halted, showing debug information and waveforms generated by the HDL simulation at the time wrong output is captured. This system reduced notably the developing time, since bugs were spotted automatically within minutes.

B. LHE compression and Storing In-Hardware Testing

For debugging purposes, an RGB frame pattern generator was created and connected as shown in Fig. 4. In this way, when it was selected as video source, upstream problems were decoupled from downstream ones. We compress the generated images with LHE modules and store them in main memory to be compared against golden data. The need of these step is because the patterns generated by the camera is modified by the Bayer demosaic algorithm making debugging troublesome.

C. Complete System Testing

An end-to-end system was deployed using the ZynqBerry board [11], a commercial low-end SoC, with a Raspberry Pi form factor. The video captured and coded, then is transmitted through the network and finally decoded and shown in a desktop workstation. In Fig. 4, we also show the system implemented in the Programmable Logic (PL) of the Zynq chip.

We use one of the i2c PS peripherals to configure the camera at startup of the system. Camera data is directly transferred to the FPGA using D-PHY and CSI-2 protocols. Then, data is unpacked to get the 10-bit pixels captured by the sensor, which are fed to the Bayer filter demosaic to get the RGB pixels. To accomplish this procedure, IPs developed by board vendor and Xilinx were used. Next, the LHE system compress the RGB pixel stream, placing the result in Processing System (PS) main memory.

The PS packetizes the encoded video blocks and transmits them through a network interface using a small C library written to provide the means to interface the compressed stream and configure the encoder. A software video player was developed to visualize the streaming video. Test pattern and images are exposed in front of the camera to validate the system.

TABLE I: Post implementation resource consumption and latency per module

Module	LUT	FF	BRAM	DSP48	Latency (cycles)
RGB2YUV	149	44	0	6	5
FPS Manager	65	36	0	0	1
Chroma and Block Handler	109	21	0	0	1
Pixel2Hop	653	100	9	0	5
Entropy Encoding	4113	2020	0	0	7-8*
Main Memory Writer	3034	1404	14	0	3
Total LHE	8134	3628	23	6	22-23*
Total LHE (%)	46.21	20.61	38.33	7.5	
Total system (%)	70.80	38.41	53.33	18.75	

* If last variable length codes overflow buffer, two transactions are needed.

VI. Implementation Results

As we discussed in section IV, this design is a 23 clock-cycle latency pipeline implementation of the LHE encoder algorithm, working at pixel rate, and clocked at 95 MHz in a Zynq XC7Z010-1 FPGA.

The footprint results are shown in Table I. The FPGA fabric resource consumption of the whole video transmission system is under 70.8%, leaving room for further logic without bringing problems and meeting timing constraints. The table also shows the resource consumption per module. The resource consumption of the AXI-Switch and ten entropy encoder cores is included in Entropy Encoding, as well as Memory Writer Checker is included in Main Memory Writer. Total value includes other logic needed to get an RGB frame. The latest column shows the latency (measured in cycles) for each module. The only non-deterministic module is the shared DDR memory, but we cannot do much about it.

VII. Conclusion and Discussion

In this paper, we have presented an FPGA implementation of a novel video encoder algorithm (LHE [1]). Table II summarizes our implementation against relevant related works, showing outstanding results. Despite the fact that this is the first hardware implementation of this algorithm, the results are very promising, both in terms of latency and complexity, paving the way for further research and commercial deployment. Although the achieved compression is not as high as the other codecs and that there is not bit rate control mechanism, developing versions of LHE are solving these problems.

This ultra-low latency makes this design suitable not only for human controlled systems, but also for high frequency automatic control systems. In such systems, this encoder can be integrated in the camera sensor chip, reducing effectively the required interface bandwidth, introducing almost no latency. The idea is extensible for the cases where the internal pixel generation rate of the sensor is higher than the interface rate. This is the case of cameras used in industrial control and automation, which can generate thousands of FPS.

Ongoing research is oriented to develop a more sophisticated entropy encoder with the capability of handling several blocks at a time to reduce logic and improve compression rate. Another future research line is the use of inter-frame encoding.

TABLE II: Encoders comparison

Implementation	Latency	Area (DSP/LUT)	Mbps [4]
Ours	242 ns [1]	6/7416	33.3
Ref. [2]	120 μs [2]	535/28.5K	14.5
Ref. [3]	>= 3 ms	– [3]	14.5
Ref. [4]	0.25 ms	302/103K	8.3
Ref. [5]	1 ms	8/9825	9.2
Ref. [6]	16.6 ms	– [3]	7.4

[1] clocked at 95 MHz; [2] our estimation; [3] hard IP; [4] Av. Mbps of compressed 4CIF ref. videos using supported features to get LHE output quality with YUV420

Acknowledgment

This work was partially supported by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund under the projects RACING DRONES (MINECO/FEDER RTC-2016-4744-7) and TRÁFICA (MINECO/FEDER TEC2015-69417-C2-1-R).

References

[1] J. J. García Aranda, M. González Casquete, M. Cao Cueto, J. Navarro Salmerón, and F. González Vidal, "Logarithmical hopping encoding: a low computational complexity algorithm for image compression," *IET Image Processing*, vol. 9, no. 8, pp. 643–651, 2015.

[2] H. Loukil, A. B. Atitallah, P. Kadionik, and N. Masmoudi, "Design implementation on fpga of h.264/avc intra decision frame," in *5th International Conference on Design Technology of Integrated Systems in Nanoscale Era*, March 2010, pp. 1–4.

[3] Fraunhofer Heinrich Hertz Institute, "Ultra Low LatencyVideo Codec," Tech. Rep., 2011. [Online]. Available: https://www.hhi.fraunhofer.de/en/departments/vca/technologies-and-solutions/hardware-solutions/ultra-low-latency-video-codec.html

[4] System-On-Chip (SOC) Technologies, "H.264 4k Video Encoder IP Core Datasheet," Tech. Rep., 2014. [Online]. Available: https://www.soctechnologies.com/technical_docs/ip_cores/h264/encoder/Datasheet%20-%20H264%204K%20Encoder%20IP%20Cores.pdf

[5] A2e Technologies. (2016) Micro footprint low latency Encoder core. [Online]. Available: http://www.a2etechnologies.com/products.html

[6] Xilinx Inc., "Xilinx Advanced Multimedia Solutions with Video Codec / Graphics Engines," Tech. Rep., 10 2017. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp497-multimedia.pdf

[7] Xilinx Inc. Vivado High-Level Synthesis. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[8] International Telecommunication Union. (2017) Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide Screen 16:9 Aspect Ratios (BT.601-7 (03/2011)). [Online]. Available: https://www.itu.int/rec/R-REC-BT.601/en

[9] Xilinx Inc., "AXI Interconnect v2.1," Tech. Rep., 12 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf

[10] Xilinx Inc., "Leveraging Data-Mover IPs for Data Movement in Zynq-7000 AP SoC Systems," Tech. Rep., 1 2015. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp459-data-mover-IP-zynq.pdf

[11] J. Kumann, "Trenz ZynqBerry: TE0726 Technical Reference Manual," Tech. Rep., 1 2018. [Online]. Available: https://wiki.trenz-electronic.de/display/PD/TE0726+TRM