# An Extremely Pipelined FPGA Implementation of a Lossy Hyperspectral Image Compression Algorithm

Daniel Báscones, Carlos González, and Daniel Mozos

***Abstract*— Segmented and pipelined execution has been a staple of computing for the past decades. Operations over different values can be carried out at the same time speeding up computations. Hyperspectral image compression sequentially processes samples, exploiting local redundancies to generate a predictable data stream that can be compressed. In this article, we take advantage of a low complexity predictive lossy compression algorithm which can be executed over an extremely long pipeline of hundreds of stages. We can avoid most stalls and maintain throughput close to the theoretical maximum. The different steps operate over integers with simple arithmetic operations, so they are especially well-suited for our FPGA implementation. Results on a Virtex-7 show a maximum frequency of over 300 MHz for a throughput of over 290 MB/s, with a space-qualified Virtex-5 reaching 258 MHz, being five times as fast as the previous FPGA designs. This shows that a modular pipelined approach is beneficial for these kinds of compression algorithms.**

***Index Terms*— Compression, FPGA, hyperspectral image, pipeline.**

## I. Introduction

HYPERSPECTRAL imaging is a technique in which every pixel in an image has hundreds of samples corresponding to different wavelengths. For the past decades, this technique has enabled remote and precise studies of soils [1], [2], can be used for cancer detection [3], [4] or food quality control [5] and is useful in many more areas [6], [7]. The great results obtained in the analysis of these images are possible thanks to the large amounts of information they contain. A single image totaling gigabytes is taken in seconds [8], so storage can easily fill up or communication links be saturated in embedded or restricted systems.

Compression is a technique which reduces data size so that handling the data is easier. This means that storage fills up slower, and transmission is faster and cheaper thanks to not having to send as much information. Compression comes in many different flavors [9], [10]: from lossless algorithms which recover the original data, to lossy versions that approximate the original data but achieve much higher compression ratios. Since hyperspectral imaging deals with great amounts of data, many different algorithms have been designed and adapted to compress them. In particular, simple algorithms that can be accelerated to meet real-time requirements [11]–[13], which for the latest sensors is 31 MS/s or 62 MB/s [8], [14].

FPGAs are devices that contain reprogrammable logic and interconnections. A custom circuit with a specific function can be designed in a hardware description language (HDL) and then ported to the FPGA. The reprogrammable elements will emulate the circuit behavior at the hardware level, offering great speed and reduced power consumption [15]. Hyperspectral images are mainly found in remote sensing, with sensors mounted on planes or satellites. These platforms usually have restrictions in resources such as memory, power, or bandwidth. Hyperspectral image compression can help make better use of memory and bandwidth. FPGAs are capable of housing the algorithms needed, and their reprogrammability allows for updates and changes unlike ASICs. These facts have made them a good candidate for hyperspectral image compression [16], [17].

Many algorithms have been designed for this kind of application, with some designs aimed toward FPGA implementation. In this article, we focus on the low complexity predictive lossy compression (LCPLC) algorithm [18], [19]. The implementation is highly flexible, being able to be configured on-the-fly for different image sizes and compression parameters. A highly pipelined architecture is designed which allows for real-time throughput while keeping FPGA usage low, making the algorithm usable in space-qualified FPGAs such as the Virtex-5 [20]. From a purely theoretical standpoint, we found that LCPLC is competitive with both lossy and lossless algorithms, achieving results on par with or better than the state of the art. When looking at the implementation level, we found resource requirements to be closer to those of lossless algorithms in comparison with the much more demanding lossy algorithms. Near-lossless algorithms thus offer the advantages of both worlds while having little disadvantages.

The rest of this article is organized as follows. Section II starts by giving a brief overview of hyperspectral compression algorithms and LCPLC implementations. Section III describes the inner workings of LCPLC, explaining, from a mathematical point of view, how all steps work together. Section IV

Manuscript received January 15, 2020; revised February 25, 2020; accepted March 16, 2020. Date of publication April 6, 2020; date of current version September 25, 2020. This work was supported by the Spanish Government under Project TIN2017-87237-P. *(Corresponding author: Daniel Báscones.)*

Daniel Báscones and Carlos González are with the Department of Computer Architecture and Automatics, Universidad Complutense de Madrid, 28040 Madrid, Spain (e-mail: danibasc@ucm.es).

Daniel Mozos is with the Department of Computer Architecture and Automatics, Universidad Complutense de Madrid, 28040 Madrid, Spain, and also with the Instituto de tecnología del conocimiento, Universidad Complutense de Madrid, 28040 Madrid, Spain.

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TGRS.2020.2982586

explains how we took the algorithm and transformed it into hardware, delving into each individual module, designed at the HDL level. Section V shows LCPLC's performance from both the compression point of view (distortion ratio) and the hardware point of view (compression speed). Results are further discussed and compared in Section VI, with conclusions drawn in Section VII.

## II. Previous Work

Many algorithms have been designed to compress hyperspectral images. For real-time, low-power applications, lossless algorithms are usually more fit thanks to their low complexity in both data dependences and data operations [16], [21]. However, lossy algorithms provide much better compression ratios which can be advantageous when storage is limited [22], but are much more complex.

In the middle ground, near-lossless algorithms usually offer the simplicity of lossless algorithms, while being able to perform lossy compression. The very recent CCSDS 123 revision [23], which already has seen implementations [24], [25], and the LCPLC algorithm [18], [19] are two great examples of them. They offer very good lossless compression performance and are able to also perform lossy compression with distortion-ratio performances close to those of pure lossy algorithms.

When looking at FPGA implementations, lossless algorithms are usually implemented for a fixed image size [26], [27]. For lossy compression, a lot of work has been aimed toward 2-D image compression in the form of accelerators for JPEG and JPEG2000 [28], that can be integrated in some hyperspectral compression systems. But the best lossy algorithms, that fully take advantage of the 3-D structure of hyperspectral images, have seen less FPGA implementations [29] due mainly to their complexity. There are even implementations of CCSDS 123 [30] and its latest revision [14] available for spaceborne systems.

We believe LCPLC can be competitive at the distortion-ratio level with lossy algorithms, while keeping hardware complexity similar to less demanding lossless algorithms. Other researchers have tackled this algorithm before, achieving good results on an FPGA [31]–[33], but not reaching the real-time performance [8] that is essential for continuous image sensing. An implementation on a GPU [34] does reach real-time, but requires a power budget of over 200 W, which would most likely be unavailable in embedded, space or airborne platforms where capture devices are usually mounted.

In this article, we aim for the fastest and smallest design we can make, by using an extremely pipelined dataflow architecture in which every step is highly segmented. Instead of using high-level synthesis (HLS) tools such as the ones used in previous implementations, we implement everything in pure very high speed integrated circuit HDL (VHDL) with the goal of tailoring the design toward raw performance.

Our results show that this approach is indeed very effective for this algorithm, reaching speeds well above real-time requirements and above GPU implementations, while maintaining low board occupancy and power consumption. Distortion-ratio performance is also competitive with both lossy and lossless algorithms, making this LCPLC implementation on FPGA a good option for hyperspectral image compression.

## III. LCPLC Algorithm

The LCPLC Algorithm, presented in [18] and [19] has a few yet powerful steps to compress an image.

### A. Blocking

First, the image is divided into nonoverlapping blocks of size $N \times M \times B$ (usually $N = M$) made up of samples of bit-depth $P$. $N$ and $M$ are spatial dimensions, whereas $B$ is the spectral dimension. All bands, $B$, of the image are retained for each block, although it would also be valid to further split a block into $n$ nonoverlapping subblocks of size $N \times M \times B_i$, where $B = \sum_{i=1}^{n} B_i$.

### B. Prediction

For each block, let $x_{m,n,i}$ be the sample at the $m$th line, $n$th sample, and $i$th band within the block. A block is coded in a slice-by-slice manner (a slice being the 2-D matrix that contains all samples of a certain band b). Each slice is coded in a raster order, coding the lines in sequential order. Two values are defined for this purpose: $\hat{x}_{m,n,i}$ which is the decoded value at a given position, and $\tilde{x}_{m,n,i}$ the prediction at the same place.

For the first slice, a simple 2-D predictor is used, which follows the following equation:

$$\tilde{x}_{m,n,0} = \frac{\hat{x}_{m-1,n,0} + \hat{x}_{m,n-1,0}}{2}. \tag{1}$$

A prediction error $e_{m,n,i}$ is calculated as $e_{m,n,i} = x_{m,n,i} - \tilde{x}_{m,n,i}$, then mapped before coding to nonnegative values, following:

$$m^{e}_{m,n,i} = \begin{cases} 2\left|e_{m,n,i}\right| - 1, & e_{m,n,i} > 0 \\ 2\left|e_{m,n,i}\right|, & e_{m,n,i} \le 0. \end{cases} \tag{2}$$

For subsequent slices, the previous slice is used for prediction. To that end, an estimator $\alpha_i$ is obtained from $\mu$ and $\hat{\mu}$ which are, respectively, the average values of the raw and decoded values in a slice

$$\begin{aligned} d_{m,n,i} &= x_{m,n,i} - \mu_i \\ \hat{d}_{m,n,i} &= \hat{x}_{m,n,i} - \hat{\mu}_i \\ \alpha_i^N &= \sum_{m,n} \left(\hat{d}_{m,n,i-1} \cdot d_{m,n,i}\right) \\ \alpha_i^D &= \sum_{m,n} \left(\hat{d}_{m,n,i-1}\right)^2 \\ \alpha_i &= \alpha_i^D / \alpha_i^N. \end{aligned} \tag{3}$$

Before coding, $\alpha_i$ is quantized with 10 bits [19] in the range $[0, 2)$ as $\hat{\alpha}_i$, and the same is done with $\mu_i$ in the range $[0, 2^P - 1)$, yielding $\hat{\mu}_i$ with $P$ bits. Then, the predicted value is obtained as $\tilde{x}_{m,n,i} = \hat{\mu}_i + \hat{\alpha}_i(\hat{x}_{m,n,i-1} - \mu_{i-1})$ and the error and mapped error are obtained in the same fashion as with the first slice.

Additionally, more compression can be achieved by quantizing the error before mapping. This is done with a uniform threshold quantizer of parameter $Q$ that works with powers of two (for faster hardware implementation). In this way, we can obtain $\hat{e}_{m,n,i} = \text{sign}(e_{m,n,i})((|e_{m,n,i}| + \lfloor 2^{Q-1} \rfloor)/2^Q)$ and then dequantize it to obtain an approximation of the original error $e'_{m,n,i} = \hat{e}_{m,n,i} * 2^Q$, and use it instead of the original error in our calculations for $\hat{x}_{m,n,i}$.

### C. Slice Skipping

For each slice, the distortion $D$ can be calculated as $D = 1/\text{NM} \sum_{i=0}^{\text{NM}} e^2_{m,n,i}$. A threshold can be established below which a slice can be skipped, thus saving bits in the output. Block skipping is indicated by 1 bit (either 0 or 1 if it is skipped) before coding the slice. This bit is referred to as $d = D \leq D^{\text{thresh}}$.

The threshold equation is given by $D^{\text{thresh}} = (\gamma 2^{Q+1}/3)$, where higher $\gamma$ values make skipping slices more common. A value of $\gamma = 3$ usually offers a good compromise between compression ratio and quality [19].

### D. Coding

Two different coders are used to generate the outputs.

1) *An Exp-Golomb [35] Code of Order 0:* For a value $v$, its magnitude $m = \lceil \log[2] v \rceil$ is first coded in unary, followed by the binary value of $v$ (using $m$ bits).
2) *A Golomb Coder [36] With Power-of-Two Parameters:* A power of two $2^p$ is used to split the value $v$ into the integer quotient $q = v/(2^p)$ and remainder $r = v \mod (2^p)$. The result is $q$ zeros followed by a one, followed by the remainder coded with $p$ bits.

Coding is again done slightly different depending on if the first slice, or others, is being coded.

1) For the first slice, the raw $x_{0,0,0}$ is exp-Golomb coded (after going through the quantization, dequantization, and mapping steps) and then all mapped errors $m^e_{m,n,i}$ are golomb coded.

   Let $l = m + nM$ be the sample index within a slice, and $a_w$ a user-defined parameter which is the number of past sample errors used for coding. Define the values

$$R_{m,n,i} = R_{l,i} = \sum_{i=\max(0,l-a_w)}^{l-1} e_{l,i} \tag{4}$$

$$J_{m,n,i} = J_{l,i} = \min(l, a_w). \tag{5}$$

   Then we have

$$k_{m,n,i} = \left\lceil \log[2] \frac{R_{m,n,i}}{2^{\lceil \log[2] J_{m,n,i} \rceil}} \right\rceil + 1 \tag{6}$$

   where $k_{m,n,i}$ is the parameter used for the golomb coder, obtained from an accumulator which holds the sum of the last $a_w$ errors for the current slice. Note that the denominator in the equation is just the value $J_{m,n,i}$ in [19]. We found no change in compression efficiency with our change, which greatly simplifies hardware implementation by replacing a divider with a shifter.
2) For subsequent slices, both $\alpha_i$ and $\mu_i$ are coded raw, followed by a "slice skipping" bit. Then, if the slice is not skipped, all mapped errors follow, coded in the same fashion as the ones from the first slice.

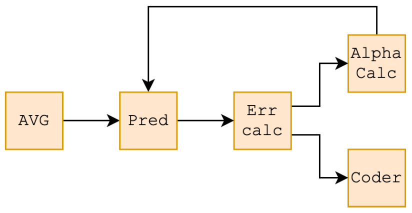

Fig. 1. High level diagram of the different modules.

The final compressed image's structure is the sequence of its compressed blocks, the individual structure of which is as follows.

1) For the first slice, the following holds.
   a) Quantize, map, and exp-zero golomb code $x_{0,0,0}$.
   b) Golomb code $m^e_{m,n,0} \forall m, n$ with parameter $k_{m,n,i}$.
2) For the rest of slices, the following holds.
   a) $\hat{\alpha}$, followed by $\hat{\mu}$, both coded raw.
   b) A bit, indicating if the block is skipped or not.
   c) If the block is not skipped, $\forall m, n \quad m^e_{m,n,i}$ golomb coded with parameter $k_{m,n,i}$.

### E. Decoding

Decoding is done following the same flow as coding, but inverting the operations. The result can be lossless if block skipping and quantization are disabled. Otherwise, losses of information are possible (but not certain). Compression will be higher with more quantization and higher distortion thresholds.

## IV. Implementation

There are two layers to the implementation.

1) The *core* works at a dataflow level, using AXI4-Stream [37] links to transmit data.
2) The *controller* adds the control logic on top, having AXI4-Lite [38] controlled registers for configuration, as well as an AXI4 bus to read from/write to memory.

Modules have been implemented in VHDL and Verilog to improve performance. But that alone is not enough. A modular dataflow design is necessary to easily pipeline the design, increasing latency but at the same time increasing throughput.

Since a lot of data operations are performed multiple times, we created basic blocks with AXI4-Stream inputs and outputs. These can be inserted in the middle of a AXI4-Stream link to manipulate the data in an easy way. The different modules can be seen in Table I.

### A. LCPLC Core

The core contains all necessary logic to perform block compression. Raw data are received via an AXI4-Stream link and compressed information is sent via another. Internally, all communications are also done via AXI4-Stream links. A high level diagram of the core design can be seen in Fig. 1.

For each slice, the average is first calculated. Then it is used, along with the alpha values from the previous slice, to make the predictions and obtain the errors. The errors feedback into the alpha calculation and are also coded to produce the output.

TABLE I
BASIC MODULES USED IN THE ARCHITECTURE

| | | | | | |
|---|---|---|---|---|---|
| **Sync** Y | Synchronizes multiple input streams, outputting once for every input tuple. | **Merge** Y | Alternatively redirects one of two input streams to the output. | **FIFO** →\|\|\| | Queue that acts as a buffer within a stream. |
| **Split** Y | Replicates the contents of an input stream, in all its output streams. | **Divert** Y | Alternatingly redirects the input to one of two output streams. | **Filter** ▽ | Allows or blocks a stream of data based on flags. |
| **Sum** Σ | Calculates the sum of an incoming data flow. A flag controls when the output is ready. | **Avg** $\bar{\Sigma}$ | Calculates the average of an incoming data flow. A flag controls when the output is ready. | **Repeat** ⇄ | Repeats a value in the input stream until a flag is set. Then the next value is repeated again and so on. |
| **Minus** − | For every pair of inputs $a$ and $b$, send an output $c = a - b$. | **Plus** + | For every pair of inputs $a$ and $b$, send an output $c = a + b$. | **Times** × | For every pair of inputs $a$ and $b$, send an output $c = ab$. |
| **Divide** / | For every pair of inputs $a$ and $b$, send an output $c = a/b$. | **Comp** > | For every pair of inputs $a$ and $b$, send an output $c = a > b?$. | **Clamp** ⊢⊣ | Clamp the input stream to a predefined range and output it. |

A more detailed diagram showing the inner modules can be seen in Fig. 2.

Although a core could be designed with a specific image size in mind, our goal was to generate a generic core that can process images and blocks of varying sizes. We allow the designer to choose a maximum desired size ($N$, $M$, and $B$), and the inner queues and buffers will be sized accordingly. The size of the inner buffers is proportional only to the slice size, with queues having to store two slices on the fly (the one being processed and the next). Image size does not have an impact on queue size since it is split into blocks and slices before being processed. Any image equal or smaller in size can then be processed by the core. If the image size is not a multiple of the block size, smaller blocks will be taken from the sides and processed as such.

Data operations depend on where in the image we are. Samples that are on the corners or edges of the block have fewer neighbors than central ones and are treated differently. Since input size is not fixed, we determine sample position with different flags. Thus, the input AXI4-Stream contains, aside from the ready and valid control signals, five different lanes.

1) *Data:* Contains the raw $x_{m,n,i}$ values.
2) *Last "r" ($l_x^r$):* Set to 1 when $m = M - 1$ (sample is the last of its row).
3) *Last "s" ($l_x^s$):* Set to 1 when $m = M - 1, n = N - 1$ (sample is the last of its slice).
4) *Last "b" ($l_x^b$):* Set to 1 when $m = M-1, n = N-1, i = B - 1$ (sample is the last of its block).
5) *Last "i" ($l_x^i$):* Set to 1 when the sample is the last of its block, and we want to force a flush of the inner buffers. This indicates the sample is the last of its image.

The general idea is as follows. First, the input data and flags are split in three streams.

1) One goes to the error calculation module (Section IV-A3) where the error will be calculated using the prediction values.
2) A second stream goes to the predictors after passing through a diverter, which selects which predictor to use [one for the first band (Section IV-A1), and a different one for the rest (Section IV-A2)]. For bands other than the first, the average is calculated beforehand. To avoid samples from having to be accessed multiple times in memory, buffers of the slice size are used to retain the values until the average is ready.
3) The last stream retains only the last block flag $l_x^b$ to be merged with the $d$ flag after the error calculation. This saves latches along the error calculation path.

Predictions from the two predictors are merged in an alternating fashion. First, all predictions from the first band predictor are sent through, and the merger switches to the $n$th band predictor after the $l_x^s$ flag is raised. Then, the same is done for the $n$th band predictor until the $l_x^b$ flag is raised. The cycle then repeats. The $l_x^b$ flag is sent through a FIFO to again save on latches on the main path.

After prediction, the error is calculated by comparing the original values with the predicted ones. Residuals and coding parameters are funneled directly to the coder. The decoded values will depend on the error flag $d$, and can be $\hat{x}$ or $\tilde{x}$ if the slice was skipped or not, respectively. We precalculate averages of both options and then select the one we are interested in as soon as the $d$ flag is ready. This way, we avoid doing the average calculation after the $d$ flag is obtained.

The decoded values of one slice are then combined with the raw values from the next to generate the $\alpha$ value (Section IV-A5), which is needed for prediction in all slices other than the first. This closes the prediction $\rightarrow$ error calculation $\rightarrow$ alpha calculation loop.

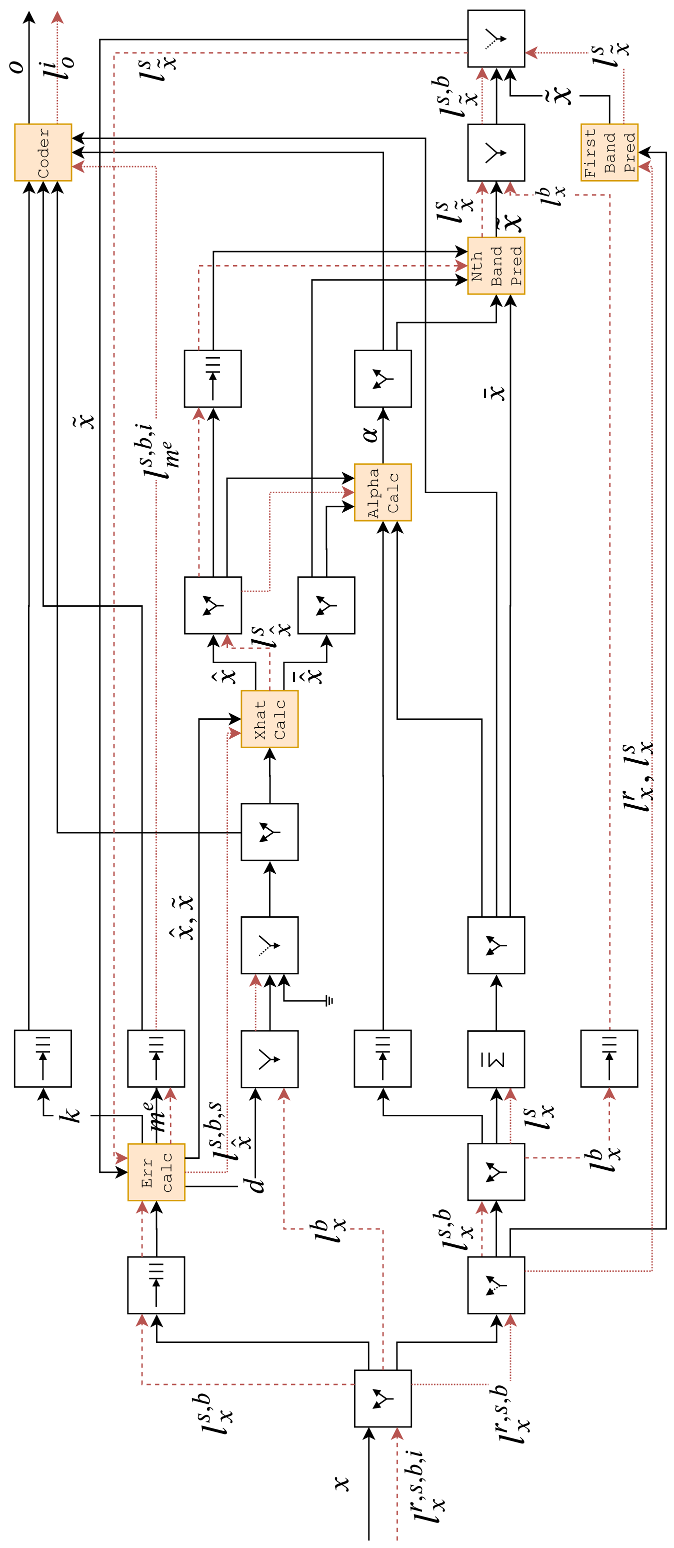

Fig. 2. Diagram showing the internal modules of the LCPLC core. In orange, composite modules (further explained later). Simple white modules control the flow of data. Black solid lines: flow of data. Red dotted lines: flow of control flags. Red dashed lines: flow of flags with no immediate effect on control.

The coder (Section IV-A6) feeds from different points in the loop, getting the mapped errors, coding parameters $\alpha$ and $\mu$, and $d$ flag, which will decide whether or not the mapped errors are encoded.

The LCPLC core contains the following modules.

*1) First Band Prediction:* The first band predictor is one of the simplest modules. Since prediction requires the previous sample in both spatial directions, an FIFO keeps the last few samples to have them available. Thus, at any point we have access to $x_{m,n,0}$ $x_{m-1,n,0}$ $x_{m,n-1,0}$. We use flags $l_x^r$ and $l_x^s$ to track the sample's position and make one of four predictions

$$\hat{x}_{m,n,0} = \begin{cases} 0, & m,n=0 \\ x_{m,n-1,0}, & m=0 \\ x_{m-1,n,0}, & n=0 \\ \left(x_{m-1,n,0}+x_{m,n-1,0}\right) \gg 1, & m,n \neq 0. \end{cases} \tag{7}$$

Error quantization can also be used to reduce size, so we include it in the feedback loop before making the prediction, as seen in Fig. 3.

*2) Nth Band Prediction:* For the prediction of subsequent bands, we get the already decoded values from the previous band $\hat{x}_{m,n,i-1}$. We also get $\alpha$, $\mu_i$, and $\hat{\mu}_{i-1}$. The operations are straightforward, and two pipelined multipliers ensure that the throughput reaches one prediction per cycle. FIFOs are used to buffer data for the last stages while the pipeline is filling up. Fig. 4 shows this process.

*3) Error Calculation:* The error calculation module receives both the raw and predicted values. Its main task is to calculate the error and error coding parameter (which the coder then processes). Since this module also knows what the decoded values will be, it also pipes them out to be used in the prediction of subsequent slices to calculate $\alpha$ (Section IV-A5). Fig. 5 shows this module's diagram.

The input prediction is split into three streams: The first is clamped and will join the decoded value generated by the error module, avoid prediction runoff. The other two are used in the error calculation itself.

The raw input value is subtracted from one of the prediction streams to create the raw error value, which has two uses. First, it is accumulated for the whole slice and then the distortion flag $d$ is calculated. Second, it goes through the quantization–dequantization stages to obtain the mapped error and coding parameter.

Quantization of level $Q$ for a value $e_{m,n,i}$ is done in a two-stage segmented module

$$\begin{cases} a_{m,n,i}^e = e_{m,n,i} + \lfloor 2^{Q-1} \rfloor \\ \hat{e}_{m,n,i} = a_{m,n,i}^e \gg Q. \end{cases} \tag{8}$$

The quantized error is then mapped to a positive value before being sent to the coder

$$m_{m,n,i}^{\hat{e}} = \begin{cases} (\left|\hat{e}_{m,n,i}\right| \gg 1) - 1, & \hat{e}_{m,n,i} > 0 \\ \left|\hat{e}_{m,n,i}\right| \gg 1, & \hat{e}_{m,n,i} \leq 0. \end{cases} \tag{9}$$

Dequantization reproduces what will happen at the decoder when information from the previous decoded slice is available. It is done in two stages by reversing (8)

$$d_{m,n,i}^{\hat{e}} = \hat{e}_{m,n,i} \ll Q. \tag{10}$$

The dequantized error is added to the predicted value to generate the decoded value that will be present at the decoder.

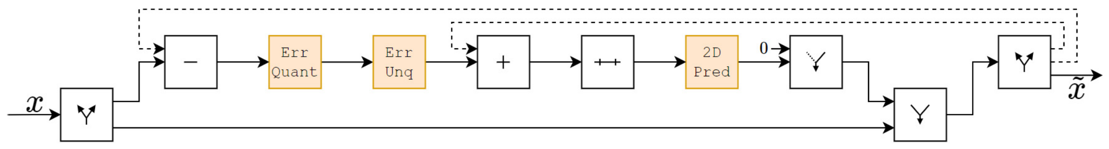

Fig. 3. Diagram of the first band predictor.

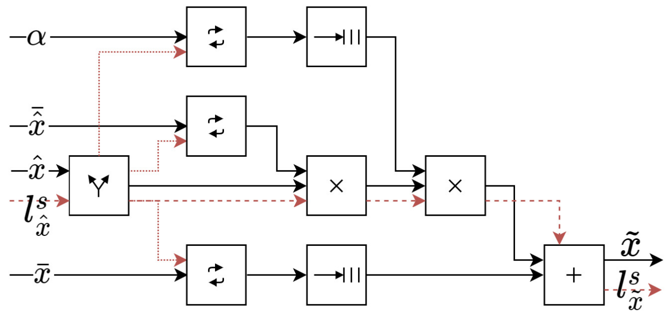

Fig. 4. Diagram of the $n$th band predictor.

This will later be used for prediction of the next band. The dequantized error also goes on to an sliding accumulator to calculate the coding parameter $k_{m,n,i}$.

The sliding accumulator keeps the sum of the latest dequantized errors. It outputs both the accumulator and the number of samples accumulated as defined in (4) and (5).

The coding parameter is then obtained, in a pipelined fashion, as

$$\begin{aligned} b^J_{m,n,i} &= \lceil \log[2] J_{m,n,i} \rceil \\ R^s_{m,n,i} &= R_{m,n,i} \gg b^J_{m,n,i} \\ k_{m,n,i} &= \lceil \log[2] R^s_{m,n,i} \rceil + 1. \end{aligned} \tag{11}$$

*4) $\hat{x}$ Precalc:* $\hat{x}$ can either be the raw prediction or the prediction adjusted with the mapped error residuals. This depends on the flag $d$ indicating a block skip. Instead of waiting for the $d$ flag and then calculating $\bar{\hat{x}}$, we calculate the average for both options (Fig. 6) and then wait for the $d$ flag before sending one or another. This saves around $MN$ cycles on each slice, since the $d$ flag becomes available with the last sample of the slice.

*5) Alpha Calculation:* The alpha module takes the current values and slice average, along with the predicted values and slice average for the previous band. It then calculates a least squares estimator $\alpha$ by multiplying the deviation to the mean in colocated positions of the current and previous slices. These squared errors are then added together and divided to get the estimator, as shown in (3). The diagram for this module is shown in Fig. 7.

*6) Coder:* Finally, the coder takes the different outputs from the previous modules in order to output a bitstream that contains the required information for decoding. To that end, the mapped error residuals $m^e_{m,n,i}$, coding parameters $k_{m,n,i}$, $d$ flag, and values $\hat{\mu}$ and $\alpha$ are all sent for every slice. Fig. 8 shows the coder diagram.

The input mapped error comes with the necessary flags to know where in the block we are currently, so the first thing is to split the flags from the data to control the output. The data are then diverted: first sample goes through the exp-zero golomb coder, and the rest of the samples from a slice go to the golomb coder.

The exp-zero golomb coder takes an input value $\upsilon$, and outputs a code $c^{\text{ez}}$ and code length $l^{\text{ez}}_c$, where

$$\begin{aligned} c^{\text{ez}}(\upsilon) &= \upsilon + 1 \\ l^{\text{ez}}_c(\upsilon) &= \lceil \log[2] c^{\text{ez}}(\upsilon) \rceil \ll 1 + 1. \end{aligned} \tag{12}$$

Then, the output stream must contain the $l_c(\upsilon)$ least significant bits of $c(\upsilon)$.

Similarly, the golomb coder takes an input value $\upsilon$ and a parameter $k$, outputting in the same manner codes $c^g$ and $l^g_c$

$$\begin{aligned} r(\upsilon) &= \upsilon \mod k \\ q(\upsilon) &= \upsilon / k \\ c^g(\upsilon) &= ((1 \ll q(\upsilon) - 1) \ll (k+1)) \parallel r(\upsilon) \\ l^g_c(\upsilon) &= k + q(\upsilon) + 1. \end{aligned} \tag{13}$$

The buses have a width limit, so if $l^g_c$ exceeds that threshold then the code is partitioned and sent as multiple subcodes, from most to least significant.

The control unit then assembles the output stream as a sequence of coded slices:

1) For the first slice, the following holds.
   a) The output from the exp-zero golomb coder, which corresponds to the first sample.
   b) The output from the golomb coder, corresponding to the rest of samples, and ending when the $l^s$ flag is raised.
2) For the next slices, the following holds.
   a) The $d$ flag indicating if the slice is skipped or not.
   b) The values $\hat{\mu}$ and $\hat{\alpha}$.
   c) If the slice is not skipped, the following holds.
      i) The exp-zero golomb coder output.
      ii) The output from the golomb coder up to the $l^s$ flag.

The output has a fixed bus width, however, samples do not always generate outputs that are multiples of that width. To force flushing the inner buffers, the last sample in a block can raise the $l^i$ flag. In this case, the buffers are dumped by filling the least significant bits with zeros, and the core is ready to process the next batch of samples. If the flag is not raised, the new coded data will be appended to the buffered data.

## B. Core Pipelining

There are two levels of pipelining in this architecture. The first refers to the pipelining of the different modules. From a high-level view, the core can be represented as seen in Fig. 1. Some modules process samples independently and their

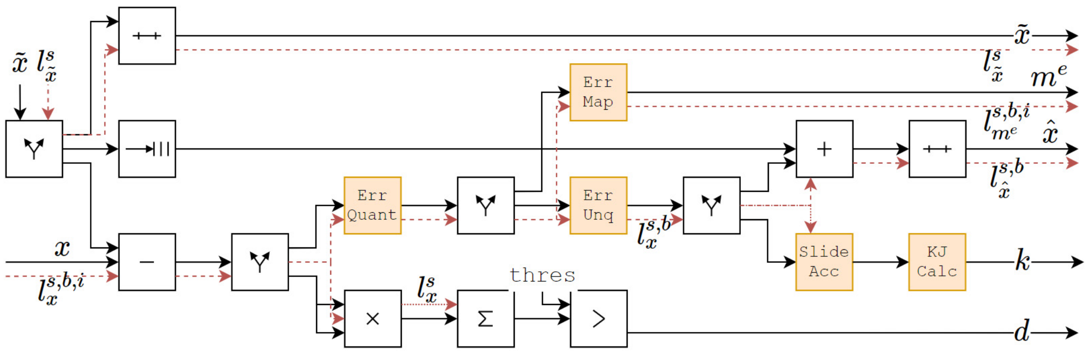

Fig. 5. Diagram of the error module.

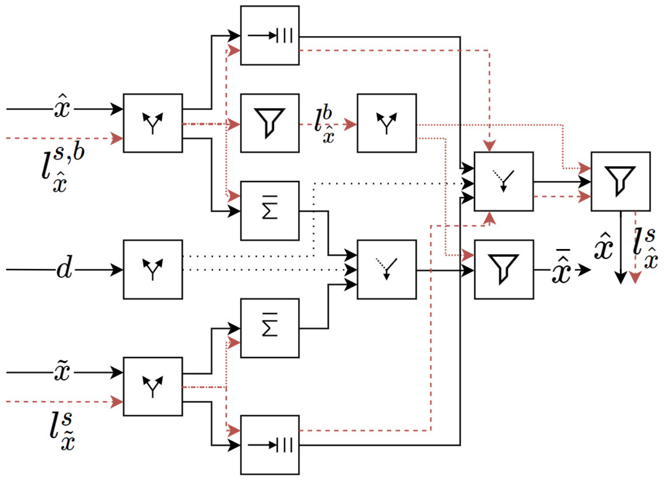

Fig. 6. Diagram of the $\hat{x}$ precalculation module.

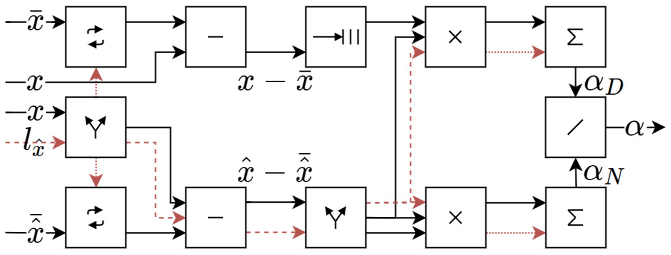

Fig. 7. Diagram of the alpha module.

operations can be chained together, while others need the previous module to have finished before starting processing samples. This happens at the slice level and can be seen in Fig. 9.

The second level of pipelining exists within the individual modules. They process individual samples one by one, performing the same transformations on each one (see Section III). All basic modules have an efficiency of one sample processed per clock cycle, so a chain of them has the same efficiency. The penalty for filling the pipeline in between slices is 77 cycles. Thus, the bigger the slices, the better this cost is amortized. This effect is later shown in Table IV.

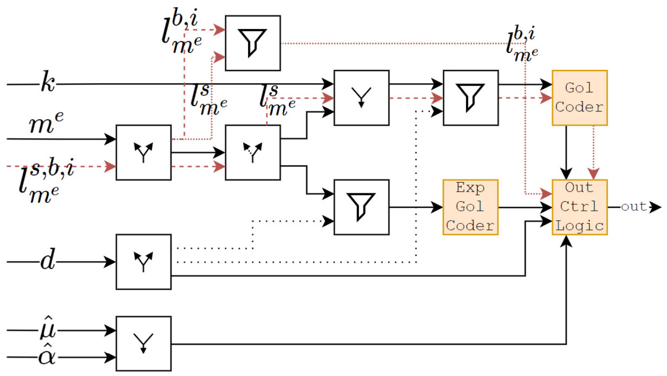

Fig. 8. Diagram of the coder.

The advantage of this design strategy is that synchronization logic between pipeline stages is not needed, as data availability dictates when each stage can be executed. Thus, we only need to ensure that the FIFOs connecting different modules have enough space to store data that is waiting on other data. In this case, this can happen only at the slice size, so FIFOs are sized to store up to a full slice of samples.

### *C. LCPLC Controller*

So far, interaction with the LCPLC module has to be done through the AXI4-Stream input and output buses. These include the control signals $l_x^r$, $l_x^s$, $l_x^b$, and $l_x^i$. The module also has ports for the threshold and quantization configurations. These inputs make it not trivial to drive, so a controller has been designed, as a wrapper, to make that task easier. The main diagram can be seen in Fig. 10.

An AXI4-Lite control bus is used to write to configuration registers (and can also be used to retrieve it). The main control module reads these registers and then controls the data path.

The data path starts by requesting memory transactions to the AXI4 bus it is connected to, at the address specified by the control. Data are then sent to a queue. Transactions are made as long as the queue is not full, mitigating latency in requests. The data then go through a flagging stage, where the $l_x$ flags are added depending on the image size (set through configuration registers). It is then processed in the core and sent to an output FIFO queue. When enough data are ready,

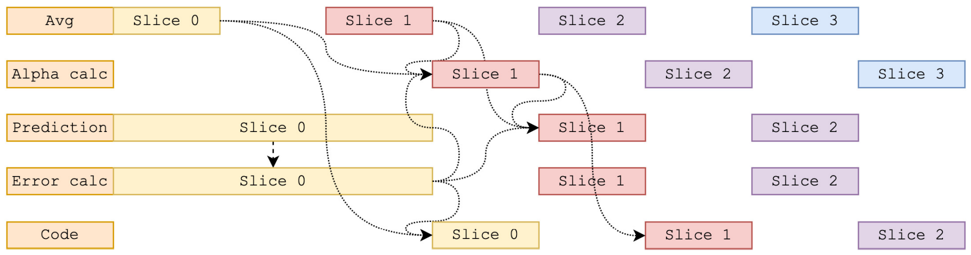

Fig. 9. Data flow diagram of the high-level modules. Dotted lines: data flow. Calculations have to be fully finished before the next module can start processing.

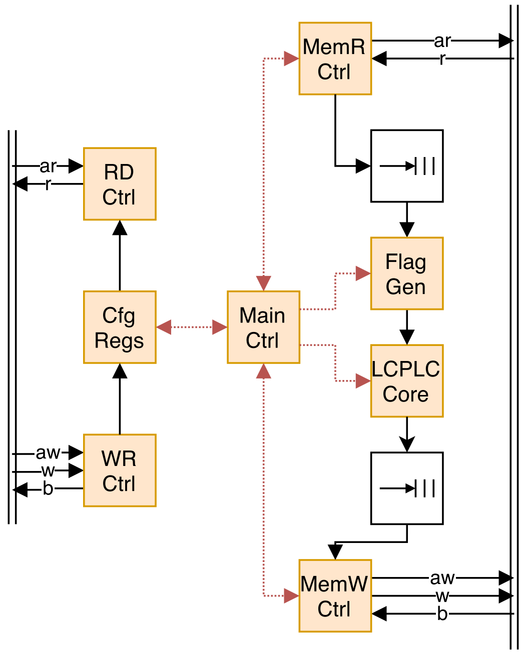

Fig. 10. Diagram of the LCPLC controller. Black solid lines: data flow. Red dotted lines: control flow. Control bus is AXI4-Lite. Memory bus is AXI4.

an output module initiates an AXI4 transaction to send output data in bursts.

The LCPLC controller is responsible for the following.

*1) Memory R/W Control:* These modules, along with the input and output FIFOs, ensure that memory transactions are done in bursts equal to the maximum AXI4 size. This way, we avoid the overhead of single-data transactions, getting a much faster I/O throughput.

The modules also ensure that every transaction is completed, to avoid entering a locked state where one end of the transaction is left waiting.

*2) Flag Generation:* Flag generation is set via the configuration registers, where image and block size are set. The input stream is flagged with the appropriate values by keeping track of where we are within the image and block. Flags indicate if a sample is the last in a pixel, row, band, or image.

*3) LCPLC Core:* The core has two configuration ports available, allowing setting the threshold for slice skipping, as well as the quantization shift to reduce precision (and increase compression).

*4) Main Control:* The main control is tasked with sending control and data across clock domains (one for the control bus, one for the memory bus, and one for the core).

Startup and stop sequences can be sent through the configuration registers to control the compression process, and information such as bytes processed or cycles elapsed is reported through read-only configuration registers.

## V. Experimental Results

Experiments were carried out in both software and hardware, to, respectively, test the effect of LCPLC parameters in compression efficiency and in hardware occupancy.

### A. Software

Images from two libraries were used to test compression efficiency. Three from the Spectir [39] library and ten from the CCSDS 123 test data set [40]. All had bit-depths $P$ of either 12 or 16.

1) **SUW:** Aerial view of the Suwanee natural reserve. (Size $360 \times 320 \times 1200$)
2) **BEL:** Beltsville crop fields. MD, USA. $360 \times 320 \times 600$.
3) **REN:** Satellite view of Reno, NV, USA. $356 \times 320 \times 600$.
4) **CUP:** An image of the Cuprite Hills. $188 \times 350 \times 350$.
5) **HAW:** View of Hawaii, USA. $224 \times 512 \times 614$ ($P = 12$).
6) **MAI:** Image from Maine, USA. $224 \times 512 \times 680$ ($P = 12$).
7) **YEL:** Seven images from Yellowstone National Park. $224 \times 512 \times 677$ and $224 \times 512 \times 680$.

To test different LCPLC configurations (and find out which ones would yield the best results), different images were compressed by LCPLC with slice sizes of $4 \times 4$, $8 \times 8$, $16 \times 16$, $32 \times 32$, and $64 \times 64$, quantization steps of 0, $2^0$, $2^1$, $2^2$, $2^4$, and $2^8$, and $\gamma$ values of 0, 0.1, 0.25, 0.5, 1, 3, and 5.

Distortion-ratio results (showing how peak signal-to-noise ratio (PSNR) fares against compression ratio) are shown in Fig. 11. Points that are higher in the graph indicate higher quality, while being toward the left indicates a better compression ratio. A point $p = (x_1, y_1)$ is objectively better than a point $q = (x_2, y_2)$ when $x_1 < x_2$ and $y_1 > y_2$, indicating that the compression configuration used for $p$ yields better results than the one used for $q$.

It can be clearly seen that the most important parameter affecting the distortion-ratio performance is slice size. Sizes of $4 \times 4$ and $8 \times 8$ underperform the bigger $\geq 16 \times 16$ counterparts. Some outliers also occur when setting the quantization step to $2^8$ and $\gamma$ to 0, with very poor PSNR performance at around the $(0.2, 0.3)$ range.

But, disregarding these outliers, all points follow a smooth curve where higher distortion-ratio performance means

TABLE II
COMPRESSION RESULTS FOR DIFFERENT ALGORITHMS AND AVIRIS IMAGES (bpppb) RESULTS FOR ALGORITHMS OTHER THAN LCPLC ARE TAKEN FROM [12]

| Image | CCSDS 123 | LCPLC16 | LCPLC32 | JPEGLS | JPEG2000 | CCSDS 122-0-B | ESA | LUT |
|---|---|---|---|---|---|---|---|---|
| **HAW** | 2.62 | 3.00 | 2.92 | 3.27 | 3.99 | 3.29 | 2.94 | 3.38 |
| **MAI** | 2.73 | 3.14 | 3.11 | 3.36 | 3.09 | 3.36 | 3.07 | 3.51 |
| **YEL00U** | 6.19 | 6.44 | 6.41 | 6.95 | 6.65 | 6.7 | 6.45 | 7.15 |
| **YEL03U** | 6.06 | 6.29 | 6.28 | 6.83 | 6.47 | 6.54 | 6.3 | 6.93 |
| **YEL00U** | 3.96 | 4.13 | 4.06 | 4.73 | 4.46 | 4.63 | 4.59 | 4.85 |
| **YEL03U** | 3.83 | 3.94 | 3.83 | 4.63 | 4.31 | 4.5 | 4.46 | 4.65 |
| **YEL10U** | 3.37 | 3.28 | 3.12 | 4.01 | 3.68 | 3.94 | 3.8 | 3.93 |
| **YEL11U** | 3.64 | 3.70 | 3.60 | 4.28 | 4.1 | 4.31 | 4.18 | 4.43 |
| **YEL18U** | 3.91 | 4.04 | 3.96 | 4.68 | 4.51 | 4.68 | 4.56 | 4.86 |

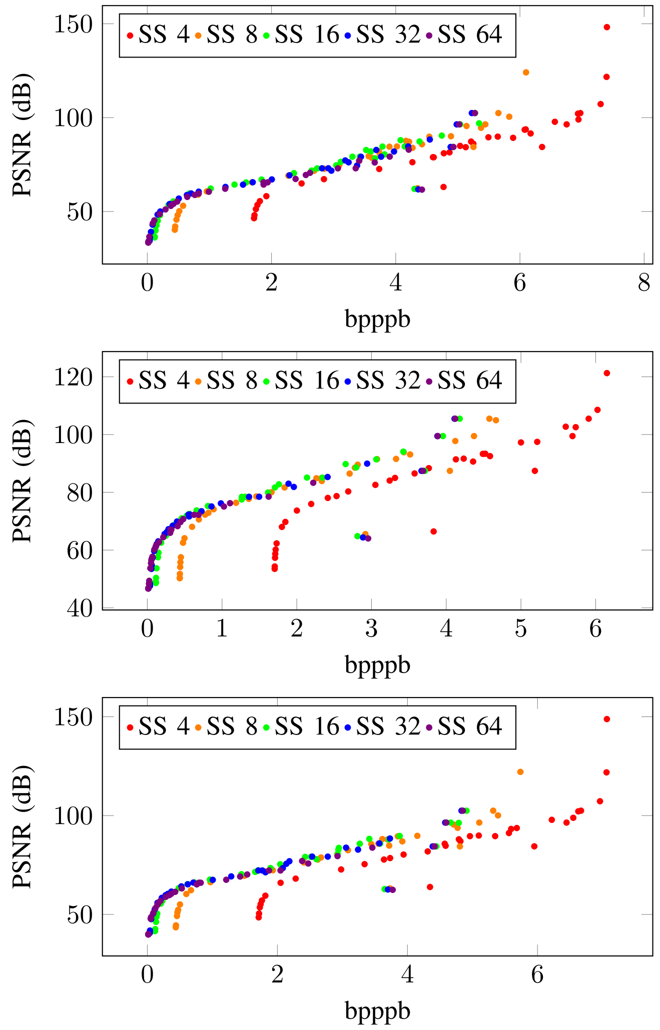

Fig. 11. Distortion-ratio for the **REN**, **SUW**, and **BEL** images, respectively. We can see similar distortion-ratio curves across all images.

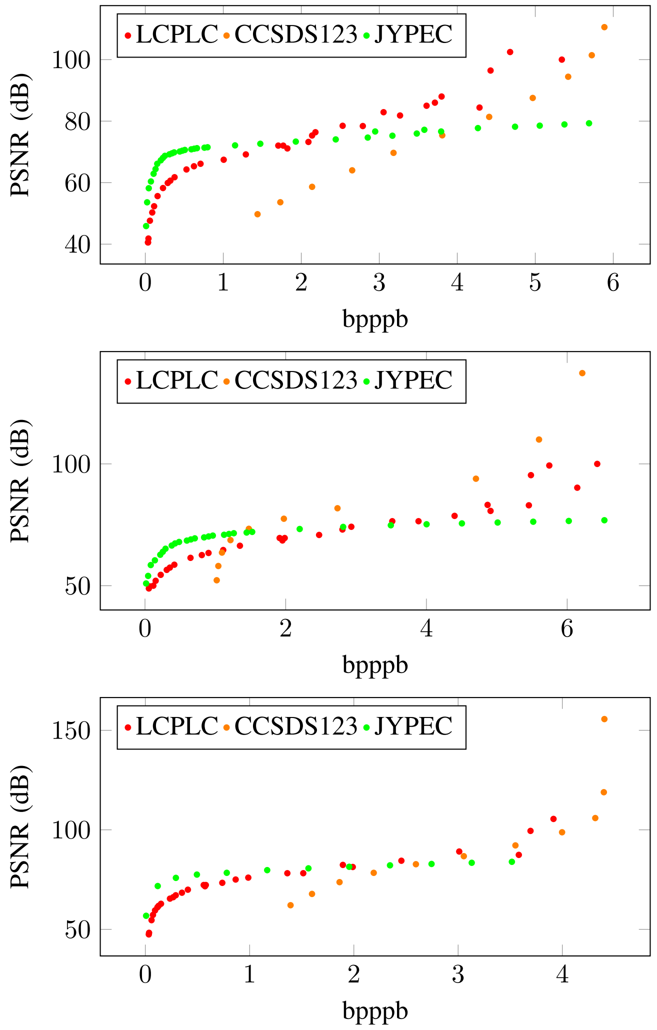

Fig. 12. Different lossy algorithms run on the **BEL**, **CUP**, and **SUW** images, respectively.

lower quantization, lower $\gamma$ or higher slice size. Lower distortion-ratio is achieved with the opposite parameters.

Comparisons with both lossless and lossy algorithms were also carried out. On the lossy side, we compared LCPLC (slice size 32) against the new near-lossless CCSDS 123 revision [23] and the more aggressive JYPEC [41] algorithm mixing JPEG2000 and dimensionality reduction.

For LCPLC, we varied the quantization steps and $\gamma$ values as before. On CCSDS 123, we tested different absolute and relative error values (from $2^3$ to $2^{13}$ in powers of two). In JYPEC, we tested with the parameters that yielded the best distortion-ratio curve. Results can be seen in Fig. 12. For low bit-rates, the JPEG2000 algorithm outperforms LCPLC, which in turn outperforms CCSDS 123. For higher bit rates (and higher quality), both near-lossless algorithms outperform JPEG2000, with one being better than the other depending on image characteristics.

Lossless results are shown in Table II. The $16 \times 16$ and $32 \times 32$ slice size versions are compared with results from multiple algorithms [16], including the CCSDS 123

TABLE III
LCPLC Core Occupancy Results for Different Maximum Block Sizes on the 7vx690tffg1761-2 and xc5vfx130t-2tt1738 FPGAs

| Config | Slice size | Image Size | LUT | Regs | DSPs | BRAMs | Occupancy | Power | Frequency |
|---|---|---|---|---|---|---|---|---|---|
| **V7_2_12** | $\leq 2^4$ | $\leq 2^{36}$ | 5544 | 5950 | 5 | 1 | 1.27% | $0.629W$ | $342.58MHz$ |
| **V7_3_12** | $\leq 2^6$ | $\leq 2^{36}$ | 5598 | 5965 | 5 | 3 | 1.29% | $0.632W$ | $342.81MHz$ |
| **V7_4_12** | $\leq 2^8$ | $\leq 2^{36}$ | 5880 | 6062 | 5 | 3.5 | 1.35% | $0.639W$ | $342.23MHz$ |
| **V7_5_12** | $\leq 2^{10}$ | $\leq 2^{36}$ | 6371 | 6172 | 5 | 6 | 1.47% | $0.672W$ | $341.99MHz$ |
| **V7_6_12** | $\leq 2^{12}$ | $\leq 2^{36}$ | 6523 | 6245 | 5 | 19.5 | 1.50% | $0.714W$ | $321.85MHz$ |
| **V5_2_12** | $\leq 2^4$ | $\leq 2^{36}$ | 6276 | 5958 | 5 | 2 | 7.66% | $2.672W$ | $258.74MHz$ |
| **V5_3_12** | $\leq 2^6$ | $\leq 2^{36}$ | 6552 | 6034 | 5 | 3 | 7.99% | $2.700W$ | $252.69MHz$ |
| **V5_4_12** | $\leq 2^8$ | $\leq 2^{36}$ | 6679 | 6057 | 5 | 8 | 8.15% | $2.712W$ | $258.68MHz$ |
| **V5_5_12** | $\leq 2^{10}$ | $\leq 2^{36}$ | 6837 | 6157 | 5 | 10 | 8.34% | $2.732W$ | $247.35MHz$ |
| **V5_6_12** | $\leq 2^{12}$ | $\leq 2^{36}$ | 7155 | 6271 | 5 | 22 | 8.73% | $2.853W$ | $240.66MHz$ |

lossless standard. Its performance is measured across a test bench of AVIRIS images, reaching bpppb (bits per pixel per band) values close to CCSDS 123 (the best of all) beating all other candidates.

## B. Hardware

Board occupancy is a very important aspect when considering a new design for an algorithm such as the one implemented here, especially when taking into account its original goal of being implemented in a satellite mission in Mars [19]. Less occupancy will allow implementation on smaller FPGAs, or leave extra room on bigger ones. In our case, occupancy is affected by maximum slice size.

To see its impact on hardware, five different syntheses were made targeting the Virtex-7 VC709 [15] and Virtex-5QV130FX [20] boards, with different maximum slice sizes. The latter is space-qualified and can be used on satellites. Maximum image size allowed did not have significant impact ($<1\%$ difference) on occupancy, so it was set at maximum of $2^{36}$ pixels. Results are shown in Table III. It is important to remark that any slice size smaller than the maximum allowed size can also be compressed.

It can be observed that all configurations have a very similar occupancy. The only resource notably affected is the number of BRAMs used, which is increased with slice size. Power consumption is low throughout the board (estimated with Xilinx's Xpower tool), though configurations with a higher maximum slice size are slightly more power hungry.

Note that, aside from these resources, external memory is needed to store the image (in our case DDR3). The module is equipped with an AXI4 bus to request memory transactions (which in our tests were made with an AXI4 to DDR3 bridge). The advantage of this setup is the ability of the module to compress all image sizes below its limit (instead of having a fixed image size like many CCSDS123 algorithms [17], [26], [27]) just by changing compression parameters via the AXI4-Lite configuration port.

As for algorithm speed, the only parameter affecting it is slice size. The data path requires a full slice to be predicted and its error calculated, before being able to extract the $\alpha$ value needed for the next slice. Since all the predicted samples need to be accumulated before starting the $\alpha$ calculation, the ideal one sample per cycle throughput cannot be reached.

TABLE IV
HW Speed Results for a 360 Band Image. Throughput Is Calculated at 300 MHz. The Theoretical Maximum With This Design Would Be 300 MB/s.
SPC: Samples per Cycle

| Config | **2_XX** | **3_XX** | **4_XX** | **5_XX** | **6_XX** |
|---|---|---|---|---|---|
| Test slice size | $2^4$ | $2^6$ | $2^8$ | $2^{10}$ | $2^{12}$ |
| Block size (bands) | 360 | 360 | 360 | 360 | 360 |
| First slice cycles | 243 | 918 | 3624 | 14387 | 57562 |
| First slice spc | 0.0658 | 0.0697 | 0.0706 | 0.0712 | 0.0712 |
| $N^{th}$ slice cycles | 137 | 333 | 1101 | 4173 | 16460 |
| $N^{th}$ slice spc | 0.1168 | 0.1922 | 0.2325 | 0.2454 | 0.2488 |
| Full block cycles | 30045 | 64797 | 203991 | 760797 | 2987997 |
| Full block spc | 0.1917 | 0.3556 | 0.4518 | 0.4845 | 0.4935 |
| Throughput ($MS/s$) | 57.513 | 106.671 | 135.529 | 145.363 | 148.048 |
| Throughput ($MB/s$) | 115.027 | 213.343 | 271.059 | 290.726 | 296.096 |

Instead, only a theoretical $1/2$ sample per cycle can be obtained. First, all the samples have to be processed to calculate the prediction accumulation, and then again all the predictions have to be used in the $\alpha$ calculation. Filling and emptying this pipeline has a cost (in our case 77 cycles regardless of slice size) that is amortized by the high throughput of a filled pipeline. Bigger slice sizes will result in less penalties for this process. Table IV shows this effect.

The first band, having a feedback loop between predictions of one sample and the next, requires around three times as many cycles as the rest. Subsequent bands are compressed at a rate of close to 0.25 samples per cycle for the bigger slice sizes. In all cases, the overall samples per cycle is closer to 0.5 thanks to parallel execution within the data pipeline. These results are only possible if the memory containing the image has a bandwidth greater or equal than the throughput of the core. We found no restrictions when using DDR3 memory, which in our setup had a clock of 233 MHz, capable of delivering bandwidths of gigabytes per second.

## C. Test Platform

First, the algorithm was implemented in Java, with the operations closely resembling those on the hardware design. This way, stimuli for the different modules could be obtained. Stimuli obtained from the test image suite were used to verify the individual modules. After that, the on-board experiments were carried out to test full image compression. To that end, the module is controlled via a Microblaze, and the data

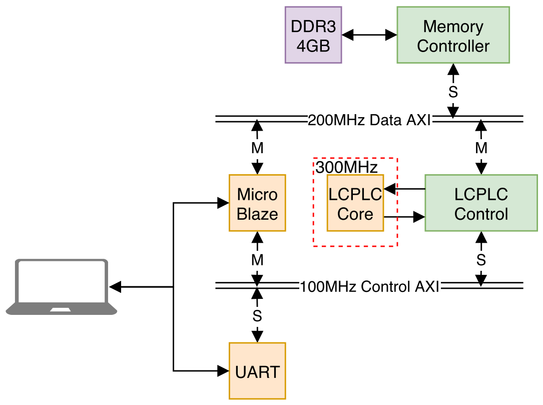

Fig. 13. Test bench designed around the module to test its functionality on-board.

(assumed to be in BSQ order) stored in a DDR3 RAM. The diagram can be seen in Fig. 13.

The microblaze controls both the LCPLC controller and a UART module to report results back to the computer. The computer can modify and read the memory via the Microblaze debug connection. A small script runs on the Microblaze, and a driver written in C encapsulates the configuration to the specific registers within the module for ease of use.

Three clock domains are present. One is used for the control bus and logic, and does not need to be fast since it only controls the compression configuration (done once per image). The second one is the memory clock, which was set by the Vivado program and was found to be fast enough. The core was placed on a third clock domain since at the same clock, the core was the bottleneck. This way it can be brought up to its maximum frequency without affecting other components.

Images were loaded in the FPGA DDR3 memory, and then compressed with the core at 300 MHz, verifying the simulation speeds obtained in Table IV. This was achieved thanks to two queues placed at the I/O of the core controller, ensuring enough space for maximum burst length (256) AXI transactions to take place between core and memory. This meant the transaction overhead of sending addresses and responses was reduced to just two cycles per every 256 samples ($\approx 1\%$). The input queue was never empty and the output queue never full throughout the experiments, meaning the core was used to its full capacity. All images tested produced the same output streams in both the FPGA design and the reference software. The code is available on Github [42].

## VI. Discussion

Two main aspects of the design have to be taken into account: distortion-ratio performance and hardware efficiency.

### A. Distortion-Ratio Performance

By analyzing Fig. 11 we see a clear improvement when going from $4 \times 4$ to $8 \times 8$ and $16 \times 16$, where for the same quality, smaller compression ratios are observed. This can be explained by slice overhead that is introduced by saving $\alpha$, $\mu$ and the skip bit. For the smallest $4 \times 4$ slice this amounts to over a bit of overhead per sample, in contrast to the mere 0.0066 extra bits per sample with the big $64 \times 64$ slice.

So even when the predictions should be better (smaller slices adapt better to local changes), the overhead is too much even when skipping and the compression ratio is worse.

The distortion-ratio performance levels off at above the $16 \times 16$ slice size, with the big $64 \times 64$ being worse when a minimum quality is required. If quality is not an issue, the bigger sizes are better since the slice overhead is practically nonexistent.

With the ideal $32 \times 32$ size, the algorithm was competitive in the lossless domain (Table II), being on par with the CCSDS 123 algorithm and beating all others. On the lossy side, it had a very interesting distortion ratio curve (Fig. 12), being slightly better than the CCSDS 123 algorithm in high bit rates while being able to go down to very low bit rates. Against more complex algorithms, it was not as effective on the low bit rate part but was able to reach higher qualities at high bit rates. In conclusion, the algorithm achieves good performance throughout all possible compression scenarios.

### B. Hardware Efficiency

More conclusions can be drawn from the hardware results:

It is obvious, by looking at Table III that the maximum image size should be kept at a high enough value to allow for compression of any image size that could arise, since it has no impact over LUTs, memory, or power.

Regarding maximum slice size, both the $16 \times 16$ and $32 \times 32$ sizes are similar in all aspects (including distortion-ratio performance), so the $32 \times 32$ version seems more appealing since it can also deal with the smaller size. The $64 \times 64$ size is no longer practical since it has increased occupancy, power consumption, and at that large slice size distortion-ratio performance starts degrading.

Concerning speed, the results from Table IV further support our previous ideas: small slices are not only decreasing distortion–ratio performance, but also decreasing speed because of the frequent emptying and filling of the internal pipelines. Even though the biggest $64 \times 64$ size is the fastest, the lower distortion–ratio achieved by that option leaves the $32 \times 32$ as the best candidate, reaching almost the same throughput.

To further cement these results, tests on board have empirically demonstrated that the simulation results are matched in a real scenario.

### C. Other Implementations

LCPLC has seen more implementations since its inception. Three different FPGA implementations are shown in [31]–[33].

All three FPGA designs are based on a modular architecture where different operations of the algorithm are done in different modules. Modules are then connected to RAM blocks where the data exchange takes place. HLS is used to convert C code to synthesizable VHDL for each module, and custom VHDL is written to control intermodule communication.

TABLE V
COMPARISON BETWEEN DIFFERENT IMPLEMENTATIONS. FREQUENCY IS MEASURED IN MHz, SPEED IN MS/s. REAL-TIME IS 30.72 MS/s [8]

| | [31] | [31] | [32] | [32] | [33] | [34] | This | This |
|---|---|---|---|---|---|---|---|---|
| Platform | 5VFX130 | 4VLX200 | 5VFX100 | 4VLX200 | RTAX2000S | T-C2075 | VC709 | 5VFX130 |
| Frequency | 80.212 | 77 | 86.964 | 75.844 | 18.649 | 1150 | 341.99 | 247.35 |
| Speed (MS/s) | 30.25 | 29.04 | 27.9 | 24.33 | 6.05 | 130 | 162.3 | 119.96 |
| Throughput (MB/s) | 60.5 | 58.08 | 55.8 | 48.66 | 12.1 | 260 | 324.6 | 239.92 |
| Efficiency | 0.377 | 0.377 | 0.321 | 0.321 | 0.321 | 0.113 | 0.485 | 0.485 |
| LUTs | 7836 | 10015 | 7746 | 9283 | 18101 | n/a | 6371 | 6837 |
| BRAMs | 4 | 4 | 4 | 4 | 7 | n/a | 6 | 10 |
| DSPs | 17 | 20 | 25 | 25 | n/a | n/a | 5 | 5 |
| Power draw | n/a | n/a | n/a | n/a | 0.378W | 225W | 0.714W | 2.732W |

Given the parallelizable nature of LCPLC (at the image block level), a GPU implementation is presented in [34], where multiple blocks can be compressed in parallel achieving great compression speeds.

A summary of all implementations (including the one presented here) is shown in Table V.

Our implementation achieves the fastest frequency, due to the modular design and deep pipelining which is given by the AXI stream links. It is four times as fast as the previous fastest FPGA design.

Not only that but it also has the greatest efficiency (samples processed per cycle). Since there is no global control needed to transfer and synchronize modules, data are processed as soon as it is available, almost reaching the theoretical 0.5 maximum efficiency.

Resourcewise, the footprint of our implementation is also the smallest out of all FPGA implementations due to the custom HDL design versus the semiautomated one of other implementations. Power draw is comparable (considering frequency difference) and is more than two orders of magnitude smaller than the fastest implementation (done in GPU).

## VII. CONCLUSION

We have analyzed how LCPLC works and adapted it to hardware execution to get as much performance as possible. With the current dependences, the theoretical limit of 0.5 samples processed per cycle is almost achieved, meaning further improvements to the design would have to come mainly as a reduction of the critical path.

Regarding distortion-ratio, we found that we should push the block size up to $32 \times 32$ instead of the default $16 \times 16$, since it offers increased quality at the same compression ratio, and reduces the number of times that the pipeline has to fill up, resulting in improved performance.

As for hardware requirements, we keep in line with the logic demands of previous designs but manage to drastically reduce DSP usage at the expense of a slight memory increase due to our intense FIFO usage. All of this is done while ensuring that the larger $32 \times 32$ block size can be compressed. Maximum frequency is kept well above 300 MHz thanks to the modular data flow design. We also demonstrate that implementation on a space-qualified FPGA is possible, bringing the capabilities of this algorithm to remote sensing applications.

Comparing with previous LCPLC implementations, we improve on FPGA designs by increasing throughput by a factor of over 5. This is mainly achieved thanks to the fully custom HDL code developed for it along with the highly pipelined data path versus the HLS approach that had been taken so far.

All these results are further cemented by the fact that LCPLC is competitive against other compression algorithms. It reaches the lossless performance of the state-of-the-art CCSDS 123 beating many other algorithms. On the lossy side, it obtains unmatched quality at high bit rates, while keeping decent quality at low bit rates, better than other near-lossless algorithms.

In conclusion, we presented a very fast, power-efficient implementation of a flexible algorithm, allowing for real-time lossless-to-lossy hyperspectral image compression over a wide range of target qualities, apt for space-qualified FPGAs.

## REFERENCES

[1] J. Dozier, "Spectral signature of alpine snow cover from the landsat thematic mapper," *Remote Sens. Environ.*, vol. 28, pp. 9–22, Apr. 1989.

[2] G. Camps-Valls, D. Tuia, L. Bruzzone, and J. A. Benediktsson, "Advances in hyperspectral image classification: Earth monitoring with statistical learning methods," *IEEE Signal Process. Mag.*, vol. 31, no. 1, pp. 45–54, Jan. 2014.

[3] H. Akbari *et al.*, "Hyperspectral imaging and quantitative analysis for prostate cancer detection," *J. Biomed. Opt.*, vol. 17, no. 7, Jul. 2012, Art. no. 0760051.

[4] H. Fabelo *et al.*, "A novel use of hyperspectral images for human brain cancer detection using *in-vivo* samples," in *Proc. 9th Int. Joint Conf. Biomed. Eng. Syst. Technol.*, 2016, pp. 311–320.

[5] D. W. Sun, Ed., *Hyperspectral Imaging For Food Quality Analysis and Control*. Amsterdam, The Netherlands: Elsevier, 2010.

[6] C.-I. Chang, *Hyperspectral Data Exploitation*, C.-I. Chang, Ed. Baltimore, MD, USA: Wiley, 2007.

[7] P. Ghamisi *et al.*, "Advances in hyperspectral image and signal processing: A comprehensive overview of the state of the art," *IEEE Geosci. Remote Sens. Mag.*, vol. 5, no. 4, pp. 37–78, Dec. 2017.

[8] Jet Propulsion Laboratory. *Airborne Visible/Infrared Imaging Spectrometer*. Accessed: Jun. 18, 2019. [Online]. Available: https://aviris-ng.jpl.nasa.gov/

[9] D. Salomon and G. Motta, *Handbook of Data Compression*. Springer, 2010.

[10] N. Abramson, *Information Theory and Coding*. New York, NY, USA: McGraw-Hill, 1963.

[11] H. Wang, S. D. Babacan, and K. Sayood, "Lossless hyperspectral-image compression using context-based conditional average," *IEEE Trans. Geosci. Remote Sens.*, vol. 45, no. 12, pp. 4187–4193, Dec. 2007.

[12] CCSDS Secretariat, National Aeronautics and Space Administration, "Lossless multispectral and hyperspectral image compression," CCSDS, Washington, DC, USA, Tech. Rep. 120.2-G-1, Dec. 2015.

[13] R. Guerra, Y. Barrios, M. Díaz, L. Santos, S. López, and R. Sarmiento, "A new algorithm for the on-board compression of hyperspectral images," *Remote Sens.*, vol. 10, no. 3, p. 428, 2018. [Online]. Available: http://www.mdpi.com/2072-4292/10/3/428

[14] D. Keymeulen *et al.*, "High performance space computing with system-on-chip instrument avionics for space-based next generation imaging spectrometers (NGIS)," in *Proc. NASA/ESA Conf. Adapt. Hardw. Syst. (AHS)*, Aug. 2018, pp. 1–20.

[15] Xilinx. *Xilinx Virtex-7 FPGA VC709 Connectivity Kit*. Accessed: Jan. 15, 2018. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html
[16] CCSDS Secretariat, Space Communications and Navigation Office, "Lossless multispectral & hyperspectral image compression," CCSDS, Washington, DC, USA, Tech. Rep. CCSDS 123.0-B-1, May 2012. [Online]. Available: http://public.ccsds.org/publications/archive/123x0b1ec1.pdf
[17] D. Bascones, C. Gonzalez, and D. Mozos, "FPGA implementation of the CCSDS 1.2.3 standard for real-time hyperspectral lossless compression," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 11, no. 4, pp. 1158–1165, Apr. 2018.
[18] A. Abrardo, M. Barni, and E. Magli, "Low-complexity lossy compression of hyperspectral images via informed quantization," in *Proc. IEEE Int. Conf. Image Process.*, Sep. 2010, pp. 505–508.
[19] A. Abrardo, M. Barni, and E. Magli, "Low-complexity predictive lossy compression of hyperspectral and ultraspectral images," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2011, pp. 797–800.
[20] Xilinx. *Space-Grade Virtex-5QV FPGA*. Accessed: Jan. 29, 2018. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-5qv.html
[21] J. Mielikainen, "Lossless compression of hyperspectral images using lookup tables," *IEEE Signal Process. Lett.*, vol. 13, no. 3, pp. 157–160, Mar. 2006.
[22] B. Penna, T. Tillo, E. Magli, and G. Olmo, "Progressive 3-D coding of hyperspectral images based on JPEG 2000," *IEEE Geosci. Remote Sens. Lett.*, vol. 3, no. 1, pp. 125–129, Jan. 2006.
[23] CCSDS Secretariat, National Aeronautics and Space Administration, "Low-complexity lossless and near-lossless multispectral and hyperspectral image compression," CCSDS, Washington, DC, USA, Tech. Rep. CCSDS 123.0-B-2, Feb. 2019. [Online]. Available: https://public.ccsds.org/Pubs/123x0b2c1.pdf
[24] *Hyperspectral Imaging and Compression Solutions*, ADATA, Taipei, Taiwan, Sep. 2018.
[25] L. Santos, A. Gomez, and R. Sarmiento, "Implementation of CCSDS standards for lossless multispectral and hyperspectral satellite image compression," *IEEE Trans. Aerosp. Electron. Syst.*, early access, Jul. 23, 2019, doi: 10.1109/TAES.2019.2929971.
[26] L. Santos, L. Berrojo, J. Moreno, J. F. Lopez, and R. Sarmiento, "Multispectral and hyperspectral lossless compressor for space applications (HyLoC): A low-complexity FPGA implementation of the CCSDS 123 standard," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 9, no. 2, pp. 757–770, Feb. 2016.
[27] M. Orlandić, J. Fjeldtvedt, and T. Johansen, "A parallel FPGA implementation of the CCSDS-123 compression algorithm," *Remote Sens.*, vol. 11, no. 6, p. 673, 2019.
[28] S. D. Jayavathi and A. Shenbagavalli, "FPGA implementation of MQ coder in JPEG 2000 standard—A review," *Int. J. Innov. Sci. Res.*, vol. 28, no. 1, pp. 76–83, 2016.
[29] S. Sanjith and R. Ganesan, "A review on hyperspectral image compression," in *Proc. Int. Conf. Control, Instrum., Commun. Comput. Technol. (ICCICCT)*, Jul. 2014, pp. 1159–1163.
[30] A. Tsigkanos, N. Kranitis, G. A. Theodorou, and A. Paschalis, "A 3.3 Gbps CCSDS 123.0-B-1 multispectral & hyperspectral image compression hardware accelerator on a space-grade SRAM FPGA," *IEEE Trans. Emerg. Topics Comput.*, early acccess, Jul. 12. 2018, doi: 10.1109/TETC.2018.2854412.
[31] L. Santos, J. F. Lopez, R. Sarmiento, and R. Vitulli, "FPGA implementation of a lossy compression algorithm for hyperspectral images with a high-level synthesis tool," in *Proc. NASA/ESA Conf. Adapt. Hardw. Syst. (AHS)*, Jun. 2013, pp. 107–114.
[32] A. Garcia, L. Santos, S. Lopez, G. Marrero, J. F. Lopez, and R. Sarmiento, "High level modular implementation of a lossy hyperspectral image compression algorithm on a FPGA," in *Proc. 5th Workshop Hyperspectral Image Signal Process., Evol. Remote Sens. (WHISPERS)*, Jun. 2013, pp. 1–4.
[33] A. García, L. Santos, S. López, G. M. Callicó, J. F. López, and R. Sarmiento, "FPGA implementation of the hyperspectral lossy compression for exomars (LCE) algorithm," in *Proc. High-Perform. Comput. Remote Sens. IV*, vol. 9247, Nov. 2014, Art. no. 924705.
[34] L. Santos, E. Magli, R. Vitulli, J. F. Lopez, and R. Sarmiento, "Highly-parallel GPU architecture for lossy hyperspectral image compression," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 6, no. 2, pp. 670–681, Apr. 2013.
[35] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Trans. Inf. Theory*, vol. 21, no. 2, pp. 194–203, Mar. 1975.
[36] S. Golomb, "Run-length encodings," *IEEE Trans. Inf. Theory*, vol. 12, no. 3, pp. 399–401, Jul. 1966.
[37] *AMBA 4 AXI4-Stream Protocol Specification V1.0*, document ARM IHI 0051A ID030510, ARM, 2010.
[38] Arm Limited. (2017). *AMBA AXI and ACE Protocol Specification AXI3, AXI4, AXI5, ACE and ACE5*. [Online]. Available: http://www.arm.com/company/policies/trademarks
[39] Spectir. *Free Data Samples*. Accessed: Jan. 29, 2018. [Online]. Available: https://www.spectir.com/free-data-samples/
[40] CCSDS. *Collaborative Work Environment*. Accessed: Nov. 22, 2019. [Online]. Available: https://cwe.ccsds.org/sls/default.aspx
[41] D. Báscones, C. González, and D. Mozos, "Hyperspectral image compression using vector quantization, PCA and JPEG2000," *Remote Sens.*, vol. 10, no. 6, p. 907, 2018. [Online]. Available: http://www.mdpi.com/2072-4292/10/6/907
[42] D. Bascones. (2019). *Lcplc*. Accessed: Aug. 2, 2019. [Online]. Available: https://github.com/Daniel-BG/Lcplc

**Daniel Báscones** received the bachelor's degree in both mathematics and computer science and the M.Sc. degree in computer science from the Complutense University of Madrid, Madrid, Spain, in 2016 and 2018, respectively, where he is currently pursuing the Ph.D. degree in hyperspectral image compression.

He was a Research Associate during the time of his M.Sc. with the Department of Computer Architecture and Automatics, Complutense University of Madrid. His main interests include hyperspectral image compression on field-programmable gate arrays, dealing with fast lossless algorithms that aid with data transmission and more complex lossy algorithms for long-term storage.

Mr. Báscones received the FPI Scholarship in 2018.

**Carlos González** received the M.S. and Ph.D. degrees in computer engineering from the Complutense University of Madrid, Madrid, Spain, in 2008 and 2011, respectively.

He is currently a Teaching Assistant with the Department of Computer Architecture and Automation, Complutense University of Madrid. As a Research Member of the GHADIR Group, he mainly focuses on applying runtime reconfiguration in aerospace applications. His research interests include remotely sensed hyperspectral imaging, signal and image processing, and efficient implementation of large-scale scientific problems on reconfigurable hardware. He is also interested in the acceleration of artificial intelligence algorithms applied to games.

Dr. Gonzalez attended the Design Competition of the 2009 and 2010 IEEE International Conferences on Field Programmable Technology and obtained the second prize in 2012. He received the Best Paper Award of an Engineer under 35 years old at the 2011 International Conference on Space Technology.

**Daniel Mozos** received the B.S. degree in physics and the Ph.D. degree in computer science from the Complutense University of Madrid, Madrid, Spain.

He is currently a Full Professor with the Computer Architecture and Automation Department, Universidad Complutense de Madrid, where he leads the GHADIR Research Group on dynamically reconfigurable architectures. He was the Dean of the Computer Science Faculty, Universidad Complutense de Madrid, from 2010 to 2018. His research interests include design automation, computer architecture, and reconfigurable computing.