A Low-Cost Hardware Accelerator for CCSDS 123 Lossless Hyperspectral Image Compression

Wesley Grignani, Douglas Santos, Luigi Dilillo, Felipe Viel, Douglas Melo

Title: A Low-Cost Hardware Accelerator for CCSDS 123 Lossless Hyperspectral Image Compression

Author(s): Wesley Grignani, Douglas A. Santos, Luigi Dilillo, Felipe Viel, and Douglas R. Melo

Document version: Post-print version (Final draft)

Please cite the original version:
W. Grignani *et al.*, "A Low-Cost Hardware Accelerator for CCSDS 123 Lossless Hyperspectral Image Compression," 2023 36th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2023.

A Low-Cost Hardware Accelerator for CCSDS 123 Lossless Hyperspectral Image Compression

Wesley Grignani*, Douglas A. Santos†, Luigi Dilillo†, Felipe Viel*‡, and Douglas R. Melo*
*LEDS, University of Vale do Itajaí, Itajaí, Brazil
†IES, University of Montpellier, CNRS, Montpellier, France
‡SpaceLab, Federal University of Santa Catarina, Florianópolis, Brazil
wesley.grignani@edu.univali.br, {douglas.santos, luigi.dilillo}@umontpellier.fr, {viel, drm}@univali.br

Abstract

Several remote sensing applications that collect specific data in the space environment use images capable of providing a large volume of information, known as hyperspectral images. Given the amount of data, one of the most critical issues in applications that use hyperspectral images is the demanded for compression, which also affects restrictions on the storage capacity and processing in space applications. This work aimed to implement hyperspectral image compressors, considering the standard by the CCSDS (Consultative Committee for Space Data Systems). The solutions were implemented using a High-Level Synthesis tool (HLS) and a manual description in a Hardware Description Language (HDL). Results show that, compared to the software solution, the HLS and the HDL implementation accelerated the application by 1.6× and 4×, respectively. For images up to 512 spectral bands, the HLS solution presented a throughput of 9.11 MSa/s, while the HDL solution can process 21.47 MSa/s, which meets the real-time requirements of the standard. The HDL solution uses about 3× fewer LUTs (Look-Up Tables) LUTs and 8× fewer FFs (Flip-Flops) than the HLS implementation. Due to the low cost observed in the results, we intend to harden the accelerator and integrate it into a future multi-core satellite system.

Index Terms

Systems-on-Chip, Hardware Accelerators, Image Processing, Hyperspectral Images, CCSDS 123.0-B-2.

I. Introduction

Space applications that collect data about the Earth use remote sensing techniques for this task. One technique is the hyperspectral images (HSIs), which can be used for Earth image collection, climate analysis, and monitoring forest environments [1]. HSIs are three-dimensional structures of pixels, where each pixel is represented in a (x, y, z) coordinate system. Each z-axis layer represents a single image at a given electromagnetic spectrum. The dimensions can sometimes be hundreds or thousands of bands [2].

Given the large volume of data in hyperspectral images, a reduction in their size becomes desirable, aiming to minimize impacts on performance in transmission and processing techniques. A well-known compression algorithm for this type of image is CCSDS 123, developed by the Consultative Committee for Spatial Data Systems [3]. Recently, a new version of the standard entitled CCSDS 123.0-B-2 was released. This version introduces some modifications compared to the previous version, such as changes in the predictor equations and changes in the header structure in the encoder. In addition, this standard introduces a near-lossless compression mode for use with the new hybrid entropy encoder.

The existence of many bands to compute HSIs turns it into a data-intensive processing structure for space systems. It affects the ability to store, process, and even transmit this imagery to a ground station. For example, the HyspIRI sensor developed by the National Aeronautics and Space Administration (NASA) can produce up to 5 Terabytes of data per day [4].

Using reconfigurable architectures, such as FPGAs, has seen an increasing adoption for creating high-performance systems for image processing applications. However, algorithm implementation for a hardware design is more complex than software implementation. Therefore, adopting High-Level Synthesis (HLS) tools has begun, enabling hardware development using programming languages [5].

High-level synthesis tools allow the creation of a hardware design from a high-level programming language. The hardware creation process starts from the code specification, and the HLS tool is responsible for creating and

This work was supported in part by the Foundation for Support of Research and Innovation, Santa Catarina (FAPESC-2021TR001907), the Brazilian National Council for Scientific and Technological Development (CNPq - processes 138179/2021-2, 140368/2021-3 and 350794/2023-5), and the Region d'Occitanie and the École Doctorale I2S from the University of Montpellier (contract 20007368/ALDOCT-000932).

allocating resources and generating the hardware description representing the design at the Register-Transfer Level (RTL) [6].

Some works have implemented hardware accelerators using HDL (Hardware Description Language) or HLS techniques for HSI compression. The works [7], [8] and [9] implemented a hardware accelerator in the CCSDS 123.0-B-1 standard, however, [8] implemented only the prediction step. The work [10] presents a versatile solution that implements both CCSDS 123.0-B-1 and CCSDS 121.0-B-2 standards. The works [11] and [12] implemented the complete compressor of the CCSDS 123.0-B-2 standard, in which [11] implemented using VHDL and [12] using an HLS tool.

In this work, we present the implementation of accelerators in HDL and HLS for the lossless compression of HSI compatible with the CCSDS 123.0-B-2 standard. The accelerators were designed to reduce development complexity and low resource utilization for applications in space systems, like the new nano/cube satellite segment that introduces stringent constraints in terms of power and resources.

II. CCSDS 123 Accelerator Design

In this work, we extend a previous project [13] by implementing the accelerator in HDL and optimizing the HLS version. We present a detailed characterization of the cost and performance and compare it with related works. The compression standard is composed of the predictor and the encoder blocks. Each block has several calculation steps, which can be seen as follows or in more detail in the specification [14]. Fig. 1 presents an overview of the prediction design.

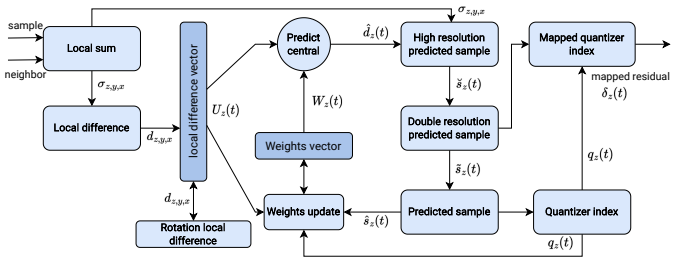

Fig. 1. Predictor component design.

A. Prediction

The prediction calculation that involves part of the lossless compression process is separated by sub-steps such as local sum, local difference, weights and local difference vector, central prediction, and mapped quantizer index.

1) Local sum: The first step of the compressor consists of a local sum between samples from the same spectral band. Each sample is represented by $s_{z,y,x}$ through the coordinate system or by the index t, defined as $t = yN_x + x$. There are three options for performing local sum, wide or narrow neighbor-oriented, and column-oriented.

2) Local difference: The local difference is performed considering the local sum calculated earlier. The local difference considers the current pixel and performs the local directional differences in N, NW, and W coordinates.

3) Weights and local difference vectors: The local difference vector is used to store P_z user-defined local differences, which are later used to calculate the predicted value of the sample. On the other hand, the weight vector stores the P_z values that will be used in the prediction by multiplying them by the vector of local differences. The P_z represents the number of preceding bands that predict a sample.

4) Predict: For the prediction calculation, the local difference prediction is performed first, being the multiplication between the weight vector and the local difference vector. Then, the high-resolution predicted sample, the double-resolution predicted sample, and the predicted sample are calculated. These calculations are necessary to obtain the mapped quantizer index, which represents the output of the predictor.

5) Mapped quantized index: After the previous steps, the output value of the predictor is obtained, called the mapped quantized index. For this, a quantizer index is used, calculated by considering the difference between the value of the current sample and the value of the predicted sample, called the prediction residual.

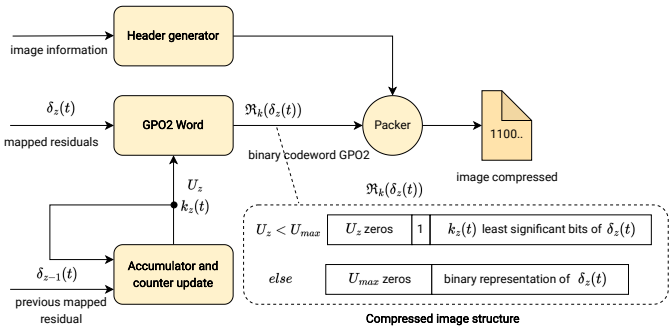

Fig. 2. Encoder component design.

B. Encoding

The design of the encoder (Fig. 2) follows the sample adaptive encoding model and receives the residual prediction from the previous component as the input value. The output form of the variable length words can change according to the calculations performed by the compressor.

The encoder is responsible for receiving the samples from the predictor component and encoding them using one of the methods presented in the specification. The encoder should generate a file with a header and body structure of variable sizes specified by the user.

The header contains information about the specification of the method used to perform the prediction and encoding, as well as information about the image that is being compressed. The body of the compressed image should contain the data generated by the entropy encoder used for the implementation.

This work implemented the sample-adaptive entropy encoder because it shows better compression results in the reduced and lossless prediction mode. Each mapped quantized index must be encoded using a variable-length binary codeword. The encoder has an accumulator and counter vector that stores information from each spectral band of the image.

III. Hardware Implementation

The lossless compressor implementation in this work is compatible with the new standard CCSDS 123.0-B-2. This CCSDS standard retains all features available in the previous B-1 standard, and the compressed image header of the recommended standard has been designed to be backward compatible with the old version. Thus compressed images produced by a compressor compliant with B-1 will also be compliant with the B-2 [15].

A. Design and Implementation Choices

We made some design choices to reduce the complexity of the implementation and have the lowest use of logic resources. Table I presents the design choices made between the allowed modes of implementation in the standard.

TABLE I
Compressor design and implementation choices.

Parameter	Specification	Implementation
Compression mode	Lossless, Near-lossless	Lossless
Prediction mode	Full, Reduced	Reduced
Local sum mode	Wide, Narrow, Column	Column
Processing order	BSQ, BIL, BIP	BIP
Entropy encoder	Block, Sample, Hybrid	Sample-Adaptative

As can be seen in Table I, several implementation options are allowed in the standard. All these options have an impact on the image compression ratio but also have a significant impact on implementation complexity and resource utilization.

1) Compression mode: The near-lossless model requires the additional implementation of the sample representatives $s''_{z,y,x}$ component and the fidelity control method $m_z(t)$, not needed for the lossless model. Setting $m_z(t) = 0$ we reduce the complexity because the quantization calculation is removed, which eliminates the integer division operations in quantization and mapped steps. In addition, we also reduce the hardware complexity by eliminating the sample representative calculation by setting the parameters to 0 for all bands.

2) Prediction mode: The reduced prediction mode does not require additional calculations of the directional local differences present in the full mode. Moreover, the storage of these values in the weights and local difference vectors increases in the full mode because of these directional differences. The reduced mode was selected for our implementation.

3) Local sum mode: The local sum considers neighboring pixels for the calculation. According to the order of processing, Band Sequential (BSQ), Band Interleaved by Line (BIL), or Band Interleaved by Pixel (BIP) implies the size of the buffers needed to store the samples to calculate the local sum. The column-oriented local sum presents the least dependence on neighboring pixels for the calculation compared to the other options. In this work, the compressor does not buffer the samples to do the local sum but receives the sample and neighbor as input.

4) Processing order: The processing order directly affects the size of the local differences to be stored. The prediction in the spectral band z depends specifically on the local differences, computed in bands $z-1, z-2, .., z-P_z$ and stored in the local differences vector. In the BIP method, the size of the local difference vector is only based on the number of previous bands considered for prediction, since this method goes through all the bands of a (x, y) position at a time. Fig. 3 illustrates the BIP storage process for $P_z = 3$ previous local differences.

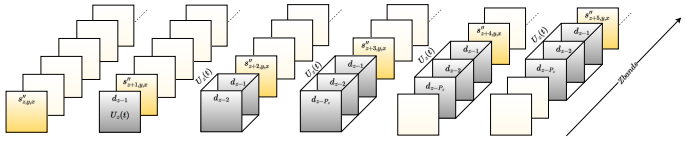

Fig. 3. Processing order and local difference vector storage process.

Running through the BSQ or BIL, the vector would need to store all the differences calculated in that band, so when it moves to the next spectral band, the local difference values in the previous bands for each pixel would be stored for the prediction. Thus, the size of the vector in BIP mode is implemented based only on the previous P_z bands and does not need to consider the N_x and N_y dimensions of the image, which are needed in BSQ or BIL methods [15].

These design choices made in the local sum step and processing order make the resource utilization of the compressor change only based on the Nz bands of the input image, and not on the Nx and Ny spatial resolution.

B. High-Level Synthesis

In the HLS implementation, each compression step was implemented in a function using the C-based HLS language. The HLS tool has optimization directives that can be used to inform the synthesis tool on generating the hardware in that specific part. The tool also has libraries for manipulating variables with a fixed bit size, which is crucial for optimized resource usage. The main directives that are widely used are:

- **Pipeline**: allows the hardware creation in parallel and with specific pipeline levels so that the startup interval of a new calculation is reduced, increasing the throughput.
- **Unroll**: enables the operations in a repetition loop to be executed in a single clock cycle, usually using more logic resources.
- **Array Partition**: forces the compiler to partition the memory vectors into smaller blocks and allocate them in registers, allowing simultaneous access to the data.

The HLS tool automatically uses the described directives to generate hardware with better performance. On the other hand, these directives can significantly increase resource usage. Each C-based function that implements a compression step is called sequentially. No optimization directive was manually added at first to check the behavior of the HLS tool in generating the hardware since it automatically applies the optimization directives previously mentioned. The implementation uses the *ap_fixed* library from the HLS tool, which allows us to specify the size of a variable in bits according to the standard specification. To reduce the complexity, calculations with powers of base 2 were done using shifters, avoiding the use of mathematical library functions.

C. Hardware Description Language

In the HDL solution, each compression step was implemented using VHDL. The prediction and encoding components were designed to work in parallel to obtain higher throughput. In addition, some steps have been parallelized to reduce the number of cycles to process a sample. It was observed that some steps did not have data dependency on previous steps or presented simple calculations that could be executed in parallel using a more sophisticated controller.

The local sum and local difference were allocated to execute in the same cycle. The predicted sample and the quantizer index were also organized to run in the same cycle after the double resolution. Besides that, we observed that the new local difference in the vector could be stored in any step after the central prediction, and it was allocated to run together with the weight update in the last cycle.

In addition, the local difference and weights vectors were created using registers to store the values. This way, all values can be read in a single cycle instead of reading all values from RAM sequentially. In the encoder block, there is no need to access all the accumulator and counter values simultaneously, enabling block RAMs to implement these vectors, which tends to reduce logical resource usage.

D. SoC Design

We used a Zedboard Zynq-7000 with an ARM Cortex-A9 to prototype the developed compressor. Fig. 4 illustrates the system integrated with the compressor via an AXI4-Lite communication interface. In addition, the compressor has an interrupt signal connected to the global interrupt controller of the processor to inform when a sample has been compressed.

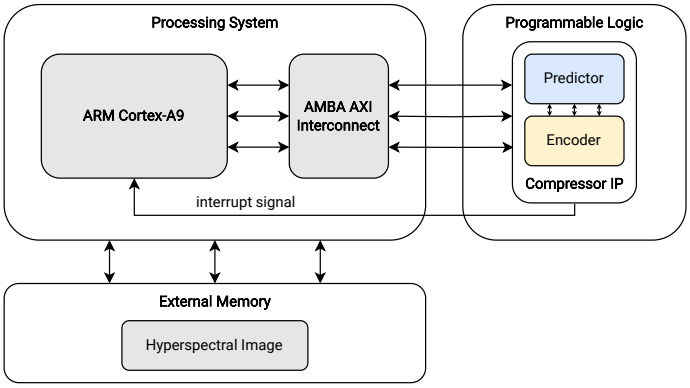

Fig. 4. SoC design overview.

The ARM processor reads the image in BIP mode from an SD card and buffers the input samples to correctly send the sample and neighbor values according to the column-oriented local sum mode. This buffering is done according to the size of the image and is not implemented internally in hardware. As the accelerator processes each image sample, the interrupt signal is active, telling the ARM processor that the compressed sample can be read and stored back in memory.

IV. RESULTS

A pattern test image from AVIRIS sensor, provided by CCSDS, was used for the compressor validation. The final compression result was compared with the reference implementation of the Empordà software, developed by [16]. The synthesis and performance results were collected using the Xilinx Vivado 2022.2 for the HDL solution and Xilinx Vitis HLS 2022.2 for the HLS solution.

Results for both solutions in terms of resource utilization, maximum frequency, and dissipated power were obtained for different values of bands for a hyperspectral image, considering the range for sensors up to 512 spectral bands, covering hyperspectral sensors like AVIRIS-NG (480 bands).

A. HLS Results

Table II presents the results obtained from the HLS solution for different Nz bands. The HLS tool generated a hardware solution that takes 11 cycles to process a sample. Thus, the throughput for different band values is approximately 9 MSa/s and varies a little due to the fact that the frequency also changes. The use of LUTs (Look-Up Tables) and FFs (Flip-Flops) has increased as the number of bands has also increased, but no Block Randomic Access Memorys (BRAMs) were allocated. This happened because the tool automatically inferred registers instead of BRAMs. In the prediction block, the use of local difference vectors and weights was synthesized as registers to have access to all its values in a single clock cycle. On the other hand, in the encoding block, only one value of the vector is accessed at a time, justifying the increase in resource utilization as the number of bands increases.

To get around this, the *bind_storage* directive can force the tool to allocate these vectors to use BRAMs since only one value of each vector is accessed at a time. The results in Table III show that by applying the directive we

TABLE II
HLS RESOURCE UTILIZATION.

N_z	LUTs	FFs	DSPs	BRAMs	Fmax (MHz)	Power (mW)	Troughtput (MSa/s)
2^5	4,139	3,832	4	0	101.01	159	9.18
2^6	5,813	5,058	4	0	101.73	164	9.25
2^7	9,842	6,825	4	0	103.23	170	9.38
2^8	16,123	12,780	4	0	101.78	200	9.25
2^9	30,272	22,843	4	0	101.75	262	9.25

can keep the same throughput achieved previously and make use of the BRAMs. When comparing the solutions for $Nz = 2^9$ bands, we observe a reduction of 8$\times$ in the use of LUTs and 7$\times$ in the use of FFs, besides a reduction in the power dissipated from 262 mW to 149 mW, making the solution more energy efficient. We can highlight that the force use of BRAMs instead of registers does not bring differences in terms of reliability, since they have similar sensitiveness to single events. On the other hand, the results have substantial outcomes in terms of resources and power overhead.

TABLE III
HLS RESOURCE UTILIZATION USING *bind_storage* DIRECTIVE

N_z	LUTs	FFs	DSPs	BRAMs	Fmax (MHz)	Power (mW)	Troughtput (MSa/s)
2^5	2,887	2,604	4	1	101.36	144	9.21
2^6	3,038	2,702	4	1	107.52	141	9.77
2^7	3,140	2,832	4	2	104.64	141	9.51
2^8	3,373	3,089	4	2	103.10	143	9.37
2^9	3,819	3,091	4	4	100.20	149	9.11

However, aiming to have a solution using fewer resources, we forced the tool not to apply the unroll and pipeline directives that are being applied. Table IV presents the results for $Nz = 2^9$ for the standard configuration in comparison with the previous optimized solution in Table III.

TABLE IV
HLS SOLUTIONS COMPARISON.

Solution	LUTs	FFs	DSPs	BRAMs	Fmax (MHz)	Power (mW)	Troughtput (MSa/s)
HLS opt.	3,819	3,091	4	4	100.20	149	9.11
HLS std.	2,577	2,428	15	4	121.34	145	3.37

When forcing the tool to not apply directives in the solution, we observe that there is a reduction in the use of resources in terms of LUTs and FFs. However, there is an increase in the number of DSPs used, and the solution has a throughput about 3$\times$ lower because the HLS standard solution takes 36 cycles to process a sample.

With these results, we can point out that even using unroll and pipeline directives, which tend to increase resource utilization, may not increase the entire resource utilization of the solution as seen in Table IV. We could observe an increase in LUTs and FFs but a reduction in the use of DSPs, besides getting a higher throughput, 9.11 against 3.37 MSa/s. This way, the application of optimization directives are good alternatives to be explored aiming at a trade-off between performance and resources.

B. HDL Results

Table V presents the resource utilization obtained from the HDL solution for different spectral band values. The HDL solution uses almost the same DSPs and BRAMs as the HLS. However, the use of LUTs and FFs does not increase so much as the number of bands also increases, in contrast to the HLS implementation. Moreover, the HDL solution considering the higher number of bands $Nz = 2^9$, has a throughput of 21.47 versus 9.11 MSa/s and uses about 3$\times$ fewer LUTs and 8$\times$ fewer FFs than the HLS solution.

TABLE V
HDL UTILIZATION RESOURCES FOR DIFFERENT NUMBER OF BANDS.

N_z	LUTs	FFs	DSPs	BRAMs	Fmax (MHz)	Power (mW)	Troughtput (MSa/s)
2^5	1,314	378	4	1	107.15	119	21.43
2^6	1,320	382	4	1.5	107.52	120	21.50
2^7	1,322	384	4	2	107.32	120	21.46
2^8	1,320	384	4	2.5	107.12	119	21.42
2^9	1,321	386	4	4	107.34	120	21.47

This reduction of resources, especially regarding FFs, represents a good result regarding reliability facing Single Event Upsets (SEUs) since FFs are one of the most sensitive parts of circuits in the space environment. Moreover, unlike Single Event Transients (SETs), in FFs, the bit-flips remain stored and potentially create problems up to the FF update [17].

The achieved throughput of 21.47 MSa/s was possible because the solution takes 5 cycles to process a sample. This is due to the HDL architecture design considering the parallelization of some steps that did not have a dependency and a control unit that made the predictor and encoder blocks run simultaneously.

C. SoC Results

The compressor was also tested on the Zynq-7000 SoC to measure the acceleration of the compression application, as shown in Fig. 4. The system was initially configured to perform image compression only by the ARM Cortex-A9 processor in a software solution. Then, the ARM processor is used only to send the samples to the accelerator, which controls the entire compression step.

The results in Table VI highlight that the compressor in HLS and HDL accelerated the HSI compression application by 1.6$\times$ and 4$\times$ compared to the ARM processor running at 667 MHz. The solution accelerates the application by moving the routines from software to dedicated hardware even when using the AXI4-Lite communication interface, which requires the processor to send and receive samples directly to the compressor over a single channel, which slows the communication.

TABLE VI
EXECUTION TIME TO COMPRESS AN AVIRIS-NG (30X50X7) IMAGE.

Implementation	Execution time (ms)	Acceleration
Software (ARM)	56.38	-
Hardware (HLS opt.)	34.86	1.62
Hardware (HDL)	13.86	4.06

Furthermore, while an HDL implementation performs better than an HLS implementation, the design time also differs. The HDL implementation took approximately 5$\times$ longer than the HLS implementation for this project, which could favor projects looking for fast verification and design iteration instead of optimal performance.

D. Comparison with Related Works

Table VII presents the results of the HLS and HDL implementations with the related works, where the implementations of this work considered those obtained with the higher number of spectral bands $Nz = 2^9$.

The HDL implementation has the lowest resource utilization of any other related works, even versions that implement lossless or near-lossless compression. The HDL solution of this work uses about 12$\times$ fewer LUTs and 40$\times$ fewer FFs than [11], and 13$\times$ fewer LUTs and 30$\times$ fewer FFs than [12]. On the other hand, these works have a higher throughput than this work and implement the near-lossless version of the standard.

When compared with [8] and [9], which also implement the same lossless version, we observe that the HDL solution has a resource utilization about 2$\times$ lower and higher throughput than [8], which implements only the prediction step of the compressor. Compared to [9], the HDL solution uses 11$\times$ fewer LUTs and 33$\times$ fewer FFs, but has a much lower throughput since [9] focuses on a highly parallelized implementation to obtain higher throughput.

Works [7] and [10] achieved a throughput about 10$\times$ and 7$\times$ higher than our HDL solution, since these works have a high data rate bus interface and an internal buffering architecture capable of processing one sample per cycle.

The implementations in this work show the use of resources lower than the related works but also a lower throughput. However, it is important to highlight that even though the related works have a better energy efficiency

TABLE VII
SYNTHESIS AND PERFORMANCE RESULTS IN COMPARISON WITH RELATED WORKS.

Work	Implementation	CCSDS	FPGA	LUTs	FFs	DSPs	BRAMs	Fmax (MHz)	Power (mW)	Troughtput (MSa/s)	Energy* (mJ)
[7]	VHDL	lossless	Virtex-5 FX130T	9,462	9,990	6	83	213	-	213.00	-
[8]	VHDL	lossless	Zynq-7000	2,244	630	3	-	142	106	20.40	5.22
[9]	VHDL	lossless	Zynq-7035	14,709	12,830	-	37	150	515	750.00	0.68
[10]	VHDL	lossless	Zynq-7035	4,619	2,765	8	74	151	-	151.00	-
[11]	VHDL	near-lossless	Virtex-7 VC709	16,458	15,707	30	187	250	732	250.00	2.93
[12]	HLS	near-lossless	Kintex XCKU40	17,185	11,915	63	85	125	500	12.50	40.00
HLS std.	HLS	lossless	Zynq-7000	2,577	2,428	15	4	121	145	3.37	43.15
HLS opt.	HLS	lossless	Zynq-7000	3,819	3,091	4	4	100	149	9.11	16.39
HDL	VHDL	lossless	Zynq-7000	1,321	386	4	4	107	120	21.47	5.56

* Maximum estimated energy consumed to process 1MSa.

due to their high throughput, the total dissipated power of the implementation is about 5 to 7 times higher than this work, which may not meet requirements for low power applications.

V. CONCLUSION

This paper presents lossless HSI compressors compatible with the CCSDS 123.0-B-2 standard described using HLS and manually in VHDL. The implementations were designed to reduce development complexity and low resource utilization for applications in space systems.

We compared the achieved performance with works developed in HDL and HLS. The HDL implementation presented a higher performance and lower resource utilization than the HLS solution. Directives applied to the HLS solution showed good results regarding resources and power overhead. The HDL solution shows good reliability when facing SEUs due to the low utilization of resources compared to the HLS solution and related works. In addition, both solutions accelerated the compression application, enabling the use of accelerators.

For future work, we plan to use an AXI4-Stream interface with a DMA system to speed up the compression application and integrate the compressor into a multi-core system. We also plan to apply fault tolerance techniques to the implemented compressor for further fault injection campaign evaluation.

REFERENCES

[1] L. Zhu, J. Suomalainen, J. Liu, J. Hyyppä, H. Kaartinen, H. Haggren *et al.*, "A review: Remote sensing sensors," *Multi-purposeful application of geospatial data*, pp. 19–42, 2018.

[2] G. Lopez, E. Napoli, and A. G. Strollo, "FPGA implementation of the ccsds-123.0-b-1 lossless hyperspectral image compression algorithm prediction stage," in *2015 IEEE 6th Latin American Symposium on Circuits & Systems (LASCAS)*. IEEE, 2015, pp. 1–4.

[3] CCSDS, "The consultative committee for space data systems," Available at: https://public.ccsds.org. Accessed: November 25, 2021, 2021.

[4] S. J. Hook, K. Turpie, S. Veraverbeke, R. Wright, M. Anderson, A. Prakash, J. Mars, D. Quattrochi *et al.*, "Nasa 2014 the hyperspectral infrared imager (hyspiri)," Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space, Tech. Rep., 2014.

[5] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1014–1029, 2020.

[6] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[7] A. Tsigkanos, N. Kranitis, G. 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 Transactions on Emerging Topics in Computing*, vol. 9, no. 1, pp. 90–103, 2018.

[8] L. M. Pereira, D. A. Santos, C. A. Zeferino, and D. R. Melo, "A low-cost hardware accelerator for ccsds 123 predictor in FPGA," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2019, pp. 1–5.

[9] M. Orlandić, J. Fjeldtvedt, and T. A. Johansen, "A parallel FPGA implementation of the ccsds-123 compression algorithm," *Remote Sensing*, vol. 11, no. 6, p. 673, 2019.

[10] Y. Barrios, A. J. Sánchez, L. Santos, and R. Sarmiento, "Shyloc 2.0: A versatile hardware solution for on-board data and hyperspectral image compression on future space missions," *Ieee Access*, vol. 8, pp. 54 269–54 287, 2020.

[11] D. Báscones, C. Gonzalez, and D. Mozos, "A real-time FPGA implementation of the ccsds 123.0-b-2 standard," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 60, pp. 1–13, 2022.

[12] Y. Barrios, A. Sánchez, R. Guerra, and R. Sarmiento, "Hardware implementation of the ccsds 123.0-b-2 near-lossless compression standard following an hls design methodology," *Remote Sensing*, vol. 13, no. 21, p. 4388, 2021.

[13] W. Grignani, G. Wisbecki, F. Viel, and D. R. Melo, "A high-level synthesis compressor of hyperspectral images based on ccsds 123.0-b-2," in *2022 LACW - 5th IAA Latin American CubeSat Workshop*. IAA, 2022, pp. 1–10.

[14] CCSDS, "Low-complexity lossless and near-lossless multispectral and hyperspectral image compression," Available at: https://public.ccsds.org/Pubs/123x0b2c3.pdf. Accessed: November 25, 2021, p. 102, 2019.

[15] ——, "Lossless multispectral and hyperspectral image compression informational report," Available at: https://public.ccsds.org/Pubs/120x2g2.pdf. Accessed: April 19, 2023, p. 147, 2022.

[16] GICi, "Gici emporda ccsds 123.0-b-1," Available at: http://gici.uab.cat/GiciApps/EmpordaManual.pdf. Accessed: August 19, 2022, p. 15, 2011.

[17] D. J. Sorin, *Fault Tolerant Computer Architecture*. Morgan and Claypool Publishers, 2009.