High-Performance COTS FPGA SoC for Parallel Hyperspectral Image Compression With CCSDS-123.0-B-1

Antonis Tsigkanos, *Member, IEEE*, Nektarios Kranitis, *Member, IEEE*, Dimitris Theodoropoulos, *Graduate Student Member, IEEE*, and Antonios Paschalis, *Member, IEEE*

***Abstract*— Nowadays, hyperspectral imaging is recognized as a cornerstone remote sensing technology. Next generation, high-speed airborne, and space-borne imagers have increased resolution, resulting in an explosive growth in data volume and instrument data rate in the range of gigapixel per second. This competes with limited on-board resources and bandwidth, making hyperspectral image compression a mission critical on-board processing task. At the same time, the "new space" trend is emerging, where launch costs decrease, and agile approaches are exploited building smallsats using commercial-off-the-shelf (COTS) parts. In this contribution, we introduce a high-performance parallel implementation of the CCSDS-123.0-B-1 hyperspectral compression algorithm targeting SRAM field-programmable gate array (FPGA) technology. The architecture exploits image segmentation to provide the robustness to data corruption and enables scalable throughput performance by leveraging segment-level parallelism. Furthermore, we exploit the capabilities of a COTS FPGA system-on-chip (SoC) device to optimize size, weight, power, and cost (SWaP-C). The architecture partitions a hyperspectral cube stored in a DRAM framebuffer into segments, compressing them in parallel using a flexible software scheduler hosted in the SoC CPU and several compressor accelerator cores in the FPGA fabric. A 5-core implementation demonstrated on a Zynq-7045 FPGA achieves a throughput performance of 1387 Msamples/s [22.2 Gb/s at 16 bits per pixel per band (bpppb)] and outperforms previous implementations in equivalent FPGA technology, allowing seamless integration with next-generation hyperspectral sensors.**

***Index Terms*— Consultative Committee for Space Data Systems (CCSDS), field-programmable gate array (FPGA), hyperspectral compression, on-board data processing, system-on-chip (SoC).**

I. Introduction

HYPERSPECTRAL imaging is a remote sensing technology used to record image data over many narrow contiguous spectral bands. The number of sampled wavelengths dictates the spectral resolution, which, from a few to several tens, hundreds or thousands of bands, gives rise to multispectral, hyperspectral or ultraspectral images.

Hyperspectral imaging is already consolidated as a key enabling technology in remote sensing applications, used in civilian applications such as smart agriculture, geology, environmental monitoring, disaster response, and recovery to name a few. However, the explosive growth of data volume from next-generation high-resolution and high-speed hyperspectral remote sensing systems compete with the limited on-board storage resources and bandwidth available for the transmission of data to ground stations, making hyperspectral image compression a mission critical and challenging on-board payload data processing task.

At the same time, a new trend has emerged in space developments, termed "new space" [1], where launch costs are rapidly decreasing, and agile approaches are being exploited in building small satellites using commercial-off-the-shelf (COTS) parts. CubeSats have enabled the use of low-cost technology, usually based on COTS components, in small satellite deployments. One of the benefits of this trend is that CubeSats can be used not only as the test beds for nonconventional developments but also as the members of large remote sensing constellations.

Low-cost COTS SRAM field-programmable gate arrays (FPGAs) can widen the scope of tradeoffs with larger, more power efficient FPGAs. In hyperspectral sensing systems, more capable FPGAs can better handle the increasing spatial and spectral resolution in data-acquisition that the latest hyperspectral sensors require. At the same time, the increased computing capability of large COTS FPGAs can more effectively compress data to improve the downlink bandwidth bottleneck.

Currently, besides the continued efforts to develop high-end hyperspectral missions [2], including Hyperspectral Infrared Imager (HyspIRI) [3], Environmental Mapping and Analysis Program (EnMAP) [4], PRecursore IperSpettrale della Missione Applicativa (PRISMA) [5], Copernicus Hyperspectral Imaging Mission for the Environment (CHIME) [6], and Traceable Radiometry Underpinning Terrestrial- and Helio-Studies (TRUTHS) [7] that are built and operated by space agencies, there is also a trend toward smaller and more agile hyperspectral missions. The Compact Hyperspectral Instrument Engineering Model (CHIEM) and its successor Compact

Manuscript received April 17, 2020; revised July 16, 2020; accepted August 12, 2020. Date of publication September 11, 2020; date of current version October 23, 2020. This work was supported in part by the Hellenic Foundation for Research and Innovation (HFRI) and in part by the General Secretariat for Research and Technology (GSRT) under the first call for HFRI Research Projects for the support of Post-doctoral Researchers under Grant 990. *(Corresponding author: Antonis Tsigkanos.)*

Antonis Tsigkanos, Dimitris Theodoropoulos, and Antonios Paschalis are with the Digital Systems and Computer Architecture Laboratory (DSCAL), Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, 15784 Athens, Greece (e-mail: antts@di.uoa.gr; theodimitris@di.uoa.gr; paschalis@di.uoa.gr).

Nektarios Kranitis is with the Digital Systems and Computer Architecture Laboratory (DSCAL), Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, 15784 Athens, Greece, and also with the Department of Digital Systems, University of the Peloponnese, 23100 Sparta, Greece (e-mail: nkran@di.uoa.gr).

Digital Object Identifier 10.1109/TVLSI.2020.3020164

Smartspectral Imager for Monitoring Bio-agricultural Areas (CSIMBA) [8], compatible with a 12U Cubesat integrate a 12-Mpixel CMOS 2-D detector array with read-out electronics (ROE) with a data rate up to 30 Gb/s.

As a result of these demanding requirements, the implementations of Consultative Committee for Space Data Systems (CCSDS) hyperspectral compressors on FPGAs for on-board use have been presented targeting various tradeoffs. A high performance by using a static configuration and fine-grain parallelism in [9], feature completeness and run-time configuration in a single core in [10], a segment-parallel approach integrated over PCIe for a specific sensor in [11], and targeting Band-SeQuential (BSQ) pixel order with modular redundancy features in a system-on-chip (SoC) [12]. However, none of the existing works takes a general approach to provide high-performance, reliability, run-time configuration, and SoC integration at the same time.

In the proposed work, we exploit modern COTS SoC FPGA devices to optimize size, weight, power, and cost (SWaP-C), along with segment-level parallelism to attain robustness to data corruption and very high performance in hyperspectral image compression. The implemented platform targets the Xilinx Zynq-7000 FPGA SoC device family. Many compression accelerator cores are tightly integrated with a flexible software scheduler that controls a set of compression cores working in parallel and can serve other payload system-level functionality (e.g., imager configuration, location integration, heater and motor control, and so on).

The rest of this contribution is organized as follows. Section II provides background on the CCSDS 123.0-B-1 algorithm as well as the tradeoffs in pixel order, segmentation, and considerations for COTS device usage. Section III introduces the proposed SoC architecture in detail including the parallelism scheme, hardware and software design, as well as considerations on system integration in a payload data chain. Finally, Section IV presents experimental results, including implementation results, technology limitations on scaling, and performance as well as detailed comparisons with existing parallel hyperspectral compressors from the literature. Section V concludes the contribution.

II. Background

A. CCSDS 123.0-B-1 Overview

The CCSDS 123.0-B-1 Recommended Standard [13], [14] is a formalization of the NASA fast lossless (FL) compressor [15], an adaptive predictive technique for lossless compression of multispectral and hyperspectral imagery. The standard achieves a combination of low complexity and compression effectiveness, far exceeding the best results from the literature [15]. CCSDS 123.0-B-1 is a prediction-based algorithm, structured in two functional parts, a predictor and an encoder. The predictor estimates the predicted sample value based on the values of nearby samples in a small 3-D neighborhood. The prediction residual (i.e., the difference between the predicted and actual sample values) is then mapped to an unsigned integer that can be represented using the same number of bits as the input data sample. These mapped prediction residuals

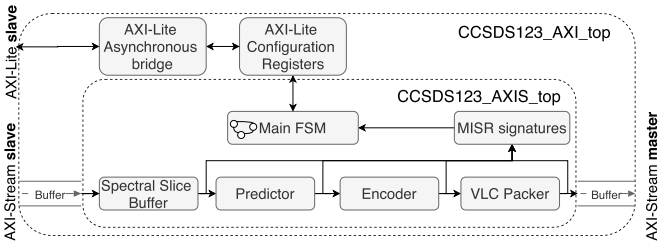

Fig. 1. CCSDS 123.0-B-1E IP core top level structure.

make up the predictor output. The predictor adaptively adjusts prediction weights for each spectral band using adaptive linear prediction. The encoder losslessly encodes the mapped prediction residuals using the sample adaptive encoder of NASA FL or as an alternative option, the block-adaptive encoder as specified in the CCSDS 121.0-B-2 standard for lossless data compression [16]. The sample-adaptive entropy encoder maps the prediction residuals using length-limited Golomb-Power-Of-2 codes, adaptively selected based on statistics that are updated after each sample is encoded. The standard has been recently updated with CCSDS 123.0-B-2, to include the capability for near-lossless compression, maintaining backward compatibility with the lossless version implemented in this work.

B. Image Pixel Order

The pixel order defines the organization of spatial and spectral pixels that comprise the hyperspectral data cube in the context of either sensor streaming or image data random access. Three orders are defined at a high level by the standard [13], band interleaved by pixel (BIP), band interleaved by line (BIL), and BSQ. The effect of pixel order on the algorithm's dependencies in computational loops and by extension on the achievable performance of an implementation is significant. The single-core accelerator used in this work uses BIP order; therefore, it may consume data (stored or streamed from a sensor) in BIP order. Any other order can be used after an intermediate conversion. The conversion cost from BSQ or BIL to BIP order is some amount of memory buffering and initial latency, dependent on the image cube dimensions. For an image sized N_x, N_y, N_z samples, a spectral frame buffer ($N_x * (N_z - 1)$ samples) is required for streaming BIL to BIP conversion or an image cube buffer [$(N_x * N_z - 1) * N_y$ samples] for BSQ to BIP conversion. If random access to this amount of memory is possible, rather than only streaming access, a DMA-based scheme may be used to perform the conversion. In any case, it is preferable for high-performance applications to avoid the conversion entirely, by either using a streaming BIP sensor or storing the data in BIP order.

C. Single-Core Accelerator

The segment-level parallel architecture proposed in this work utilizes the CCSDS 123.0-B-1E [17], [18] single-core implementation. The internal architecture of the CCSDS 123.0-B-1E IP core (Fig. 1) consists of four units at the highest level: the predictor and the encoder comprising the compression engine along with an on-chip spectral slice buffer that can

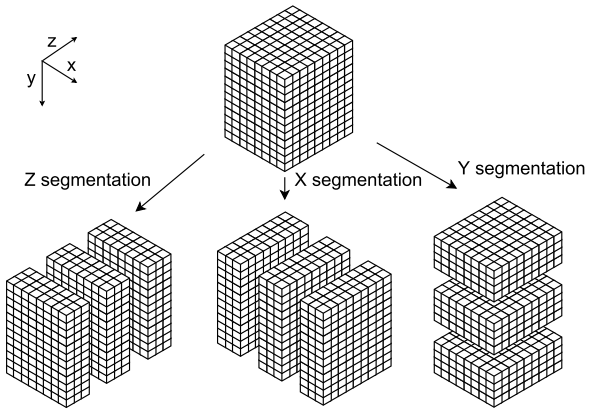

Fig. 2. Axes of image cube segmentation.

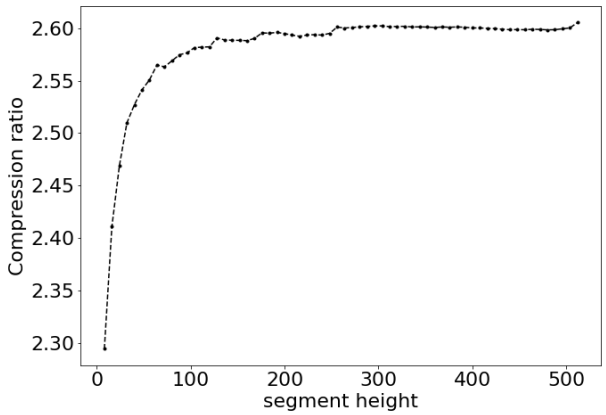

Fig. 3. Compression ratio for the values of segment height, AVIRIS image.

accommodate a spectral slice of typical hyperspectral images in on-chip block RAM (BRAM). The encoder variable length code outputs are collected by a variable length codeword packer into 64-bit output packets. The CCSDS 123.0-B-1E IP core presents two streaming interfaces based on AXI4-Stream and a configuration interface based on AXI4-Lite. Internally, commands written to configuration registers in the AXI4-Lite address space control the master controller of the compressor core, whereas various status and progress information is made available in status registers in the same address map.

To enable integration in the parallel SoC architecture for low-power and high-throughput performance, CCSDS 123.0-B-1E includes four asynchronous clock domains, negotiating internal domain crossings appropriately. Each streaming I/F is clocked separately, as well as the configuration interface and internal datapath. Other single-core CCSDS 123.0-B-1 compressors have been presented in the literature designed for various trade-offs, targeting specific sensors [19], feature completeness and run-time configuration [10], [20], or a fixed configuration of image and compression parameters [21].

D. Image Segmentation

The recommended standard [13] does not directly address image segmentation; however, the working group's informational report [14] suggests the use of segmentation to improve robustness to data corruption. Using segmentation, each image segment is compressed as a separate image, therefore limiting the impact of data corruption to the affected segment. In Fig. 2, segments are produced by partitioning an image along the X- and Y-axis spatial dimensions or the spectral Z-axis; however, combinations in a block (tile and strip)-based scheme are also possible. For sensors that produce data in BIP or BIL order, segmentation across the Y-axis is a straightforward approach, similarly for BSQ imagers across the Z-axis. According to [14], segmentation does not affect compression effectiveness when segments are large; greater than 50 line high segments are found to compress as effectively as without segmentation.

We validate this heuristic from [14] with an experiment: Fig. 3 shows the compression ratio for the different values of segment height (step of eight lines), for the benchmark AVIRIS image (scene 0) using Y-segmentation on the input BIP image. We compress using the same compression parameters, equal segments sizes, and default weight initialization. In this compression performance benchmark, each segment is compressed independently, then summing the compressed segments' sizes, and we compute the total compression ratio. The curve converges to the lossless compression ratio, with a knee at about 50 lines of segment height. Note that [14] suggests that using custom weight initialization to initialize each segments' prediction weight with the final weights of the previous segment can mitigate the compression performance loss due to segmentation. This, however, cannot be used in parallel compression because it introduces a dependence across the segments of the image.

E. SoC FPGA Devices in Space

The Xilinx Zynq-7000 class of heterogeneous SoC devices integrate single- or dual-core ARM (Cortex-A9) microprocessors with Xilinx Artix-7 or Kintex-7 FPGA fabrics into a single chip, promising opportunities for significant savings in SWaP-C. These devices comprise a different class compared to common FPGAs, by allowing entirely new levels of integration at the system level through software–hardware codesign.

In the prevailing spacecraft subsystems' architecture [22], [23], FPGA devices are combined with radiation-hard microprocessors which usually perform control tasks. The discrete FPGA devices perform computationally intense data processing tasks (such as data compression), whereas a controller processor such as the BAE Systems RAD750 performs command, control, and monitoring functions. A benchmarking study of high-performance computing in space, focusing on vision-based navigation [24] as an application scenario, finds that FPGAs are the orders of magnitude better in performance per Watt than typical [25] space-grade CPUs, whereas COTS CPUs and GPUs require prohibitive amounts of power. The study [24] concludes that it is preferable to use FPGA SoC devices for their SWaP-C and integration benefits.

Although there are currently no space-qualified heterogeneous SoC parts, they have successfully been used in the International Space Station (ISS) and multiple CubeSats in low earth orbit (LEO), where the exposure to radiation is limited. Multiple missions and payload platforms are in use or under

development [26], [27] featuring COTS SoC FPGAs centrally in their architecture.

Due to their aggressive circuit scaling and high reliance on SRAM to store the configuration data, COTS SRAM FPGAs such as those in the Zynq-7000 devices are vulnerable to transient effects, such as single event effects (SEEs), caused by ionizing radiation in space. Nevertheless, mitigations [28] can be designed, e.g., designers of Advanced Processor Core for Space Exploration SoC (APEX-SoC) [26] propose to combine certain functionality of the COTS Zynq such as watchdogs with mitigation techniques on the FPGA to increase the error resilience of the system as a whole. Neutron beam test results in [27] suggest that in specific configurations, the COTS Zynq part (e.g., disabling caches) can operate in a radiation environment, whereas no single-event functional interrupts (SEFIs) are observed in control circuits [29]. The consensus on system design around the Zynq COTS devices is that as long as appropriate techniques are employed for configuration memory scrubbing, external watchdog timers and certain device configurations are used, the parts are suitable for use in spacecraft payloads in LEO. In this spirit, the CHREC Space Processor (CSP) [27] features COTS devices to perform the critical data processing, and simpler radiation-hard devices to monitor and manage the COTS devices, augmenting them both with fault-tolerant computing techniques in software and hardware. This combination of COTS and rad-hard devices at the system level can maximize performance and reliability while minimizing energy consumption and cost.

This contribution targets a Zynq-7000 device to accelerate the compression of hyperspectral images, in an envisioned spacecraft payload data system. The required techniques to increase system reliability, this being a COTS part, are considered orthogonal to the efforts of the integration and architecture of the accelerator. Configuration memory scrubbing as well as watchdog timers are system level concerns that do not influence the accelerator design and, therefore, are not considered. The implemented benchmark platform targets the performance optimized tier of the Zynq-7000 family, the Kintex-7 dual ARM core xc7z045 part. However, the platform can be ported to other devices in the Zynq family, even to a Zynq UltraScale+ device. Due to FinFET technology, UltraScale+ devices are susceptible to single event latch-ups (SELs) [30] compared to planar devices; however, recently, the manufacturer has proposed a (yet untested in a mission) workaround to the SEL sensitivity [31].

III. Proposed SoC Architecture

A. Segmentation and Course Grain Parallelism

The proposed architecture takes the advantage of image segmentation to split the image into smaller parts compressed independently, allowing for scalable performance using many compression cores, operating on separate segments each. At the same time, segmentation limits the effects of potential data corruption on the reconstructed imagery to the affected image segment only.

In practice, implementing segmentation in the proposed architecture: the input image is in BIP order at rest in main memory when compression starts and the compressor accelerators receive segments' contents in BIP order. The architecture uses strip or Y-segmentation with a variable segment size. When an image is stored in BIP format, segmentation across the Y-axis is the least complex scheme, a matter of calculating offsets and segment sizes, fetched in a single memory transaction per segment. Assume a 16 bits per pixel per band (bpppb), image cube (N_x, N_y, N_z) evenly divisible in three segments with height $N_s = N_y/3$ each $N_s * N_x * N_z$ samples. The offset to the second segment is $1 * (N_s * N_x * N_z) * 2$ bytes and to the last segment $2 * (N_s * N_x * N_z) * 2$ bytes. When the image cannot be evenly divided into segments, nonequal segment heights are used, but no other special provisioning is needed. Each compression core compresses a segment independently and includes the segment size in the output bitstream headers, as the triplet (N_x, N_y, N_z) where N_y is implied to be N_s for that segment.

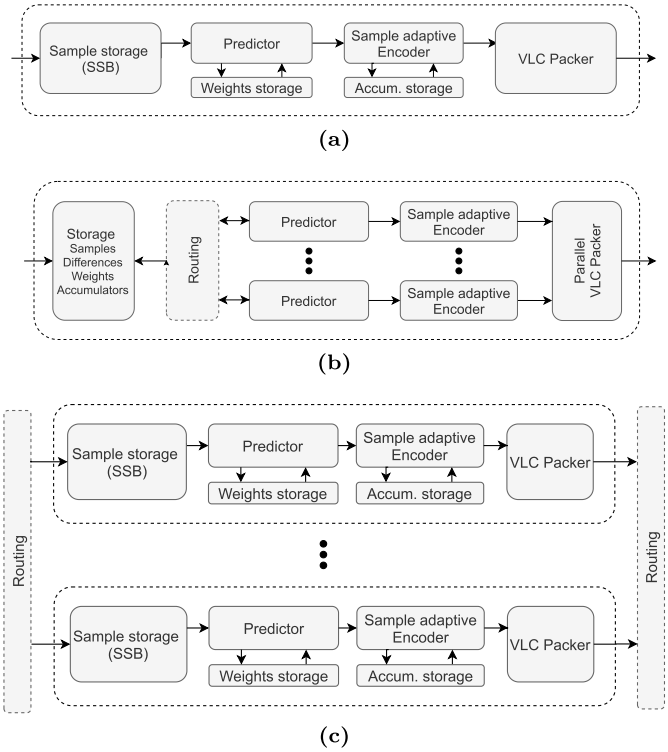

Fig. 4. (a) Serial, (b) fine-grain, and (c) coarse-grain parallel architectures.

To summarize, the proposed segmentation scheme uses the following:

1) *Variably sized segments*, although commonly as large as possible to ensure negligible effect on compression effectiveness (lossless compression ratio);
2) *Y-segmentation* on input BIP order images, compressing segment data in BIP order.

Depending on the structural level where parallelism is applied, coarse or fine-grain parallelism is possible in the compression operation. The basic blocks of the accelerator are shown in the single-core architecture in Fig. 4(a). The two possible parallel architectures are shown in Fig. 4(b) and (c): the fine-grain parallel architecture scales predictor/encoder pairs with a single storage component and parallel input

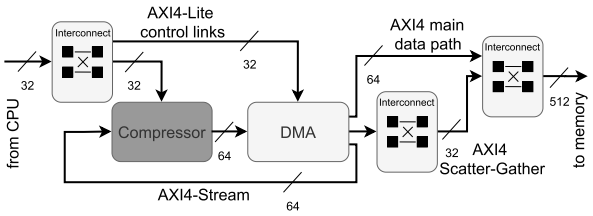

Fig. 5. Compressor scalable element.

TABLE I
PARALLEL COMPRESSOR SoC ADDRESS SPACE: $N = 5$

Master	Slaves	Range
ARM PS interconnect	PS DDRC	1G
ARM GP0 port	CCSDS123 {0..4} S_AXIL DMA {0..4} S_AXIL	8M
ARM GP1 port	MIG S_AXI	1G
DMA {0..4} M_AXI_SG	MIG S_AXI	1G
DMA {0..4} M_AXI	MIG S_AXI	1G

packer, whereas the coarse-grain parallel architecture scales an entire core to compress the image in segments.

The fine-grain parallel architecture applies parallelism efficiently at a low level but does not use segmentation. To take the advantage of error containment by segmentation, the proposed work implements the coarse-grain parallel or segment-parallel architecture, which is based on splitting the image cube into segments and then considering each as an independent image. The compression of segments is performed by independent compressor cores, either in parallel by many compressors or scheduled in time. The primary motivation for segmentation is to limit the effects of potential data corruption due to external factors during data downlink. Data loss is limited to the affected segment, whereas the rest of the image can be recovered, therefore by using smaller segments error containment is improved. However, on the other side of this trade-off, when segments are very small, the compression effectiveness suffers due to the loss of adaptation in the predictor.

As segments are independently compressed, no aspect of overall performance deteriorates when adding another core. Performance scales linearly with the number of compressors implemented, and the upper bound depends on technology limits (device resources and memory bandwidth). At system level, this architecture maps naturally to standard SoC design patterns which allows using standard buses and well-verified interconnects and memory controllers, as implemented in this work. Furthermore, each segment can be compressed independently in a standard compliant way, without modification to the compression algorithm.

However, segmentation imposes system level challenges, i.e., multiple compressed data packets are produced per image rather than one and multiple compressors must be configured and controlled. To tackle the overhead of managing multiple cores, the compressors are integrated in a SoC environment, where this complexity is managed by a software scheduler.

Similar segmentation schemes have been presented in the literature to exploit data parallelism and increase robustness to errors. Rodríguez *et al.* [12] use ARTICo^3 [32] to schedule a number of compressors on the FPGA, compressing the blocks of the image in parallel. The authors use a small square block partitioning of the image [across X- and Y-axes: blocks sized $(N_z, N_y, N_x) = (224, 8, 8)$], which leads to smaller memory footprint at the expense of some deterioration in compression ratio compared to large stripe segments as used in this work. In the NASA JPL implementation [11], the authors use a Y-segmentation scheme with fixed 32-line high segments, compressing an image in BIP order, for performance and error containment.

B. Hardware Design

The implementation targets the Xilinx ZC706 development board which among other peripherals and interfaces features 1 GB of DRAM connected to the processing system (PS), 1 GB of DRAM connected to the programmable logic (PL), as well as, an Ethernet port and flash memory.

The architecture is based on a scalable element which is replicated to increase the number of cores. It comprises a control AXI4-Lite interconnect, a CCSDS 123.0-B-1E compressor core, a Xilinx DMA controller core, a performance-optimized AXI4 interconnect, and a resource-optimized interconnect in cascade. The interconnect cascade on the sink side is a resource optimization detailed later in this section, and for functional purposes, it may be seen as a single component with all slave interfaces and a single master interface. The scalable element is shown in Fig. 5 at a high-level view. The following AXI4 links are shown in the following:

1) compressor core and DMA core control AXI4-Lite (32b) connected to the same interconnect;
2) AXI4-Stream (64b) links between the compressor core and DMA: Compressor source and sink connected to DMA sink and source respectively;
3) AXI4 (64 bit) link for the high speed data path from the DMA core toward main memory, through the data interconnect;
4) AXI4 (32b) link connected to the interconnect cascade toward main memory for reading scatter-gather lists.

The scalable element is replicated to attain parallelism; the simplified block diagram of the 5-core implementation for the benchmark platform is shown in Fig. 6. The control interconnect connects two AXI4-Lite interfaces per compressor core and is slaved to the ARM processor. The interconnect cascade connects the processor master, the scatter-gather, and data interfaces of the DMA cores, to the AXI4 slave (512 bit) interface of the PL DRAM controller with the appropriate buffering and bus resizing. This allows access to the PL DRAM address space, to both the processor and DMA cores.

Each compressor core is accessible from the ARM processor through the AXI4-Lite control interconnect and in turn its AXI4-Lite interface. The address space of each core includes control and status registers to allow programming, control, and progress reporting for each compressor. Each accelerator core is mapped to a base address in the global address space (Table I), which allows access to the processor through the

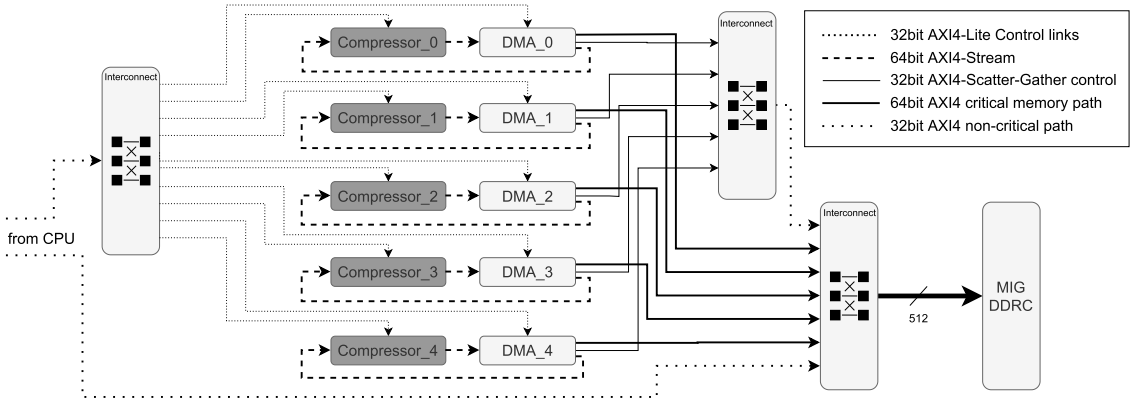

Fig. 6. Parallel compressor SoC architecture for 5-core implementation.

PS General Purpose AXI master port. Similarly, the processor accesses the control interface of the DMA core for each compressor-DMA pair, to program the receiving and transmiting transactions.

To feed each compressor with data from memory and store output compressed bitstreams, the DMA controllers connect to the memory interface generator (MIG) memory controller to access PL DDR memory. In the offline scenario, the CPU loads the image data to memory from the sensor, and conversely, in the real-time scenario, the data are continuously streamed in from the ROE through a double buffer. Each DMA controller is programed to stream a segment into the compressor from the appropriate offset in the input image buffer, using the memory map to stream (MM2S) AXI4-Stream interface. Concurrently, the same DMA controller is programed to stream data from a compressor core's output Stream-to-memory map (S2MM) interface toward a preallocated buffer for that segment,[1] in main memory. The calculation of offsets, allocation of buffers, programming, and response to events (interrupts and status polling) are controlled by the software scheduler. The DMA controller is a Xilinx IP core, configured for a performance level (burst size, bus widths, and so on) such that the compressor core is never starved of data or its output stalled. The DMA transaction length is a sensitive setting for this DMA core (it has a strong impact on $F_{\max}$); therefore, a scheme to minimize the allowed transaction length is used: input and output are packetized from the perspective of the DMA core, to allow multiple small transactions to transfer a segment data and bitstream. The software controller makes the appropriate arrangements with multiple small buffers to accommodate this.

Each DMA core has two master interfaces: one to read the scatter-gather (M_AXI_SG) transaction list and another for the stream to/from memory-map data movement proper. The M_AXI_SG interface is not performance critical, being only used twice per transaction to read a few bytes (offset, length, and so on) while traversing the transaction list. Therefore, multiplexing these five M_AXI_SG master interfaces should be optimized for resources, and the associated 5:1 interconnect does not need transaction buffering for bursts, or to allow multiple in-flight transactions. Similarly, the control AXI4-Lite interconnect is prime for resource optimization. The DMA master data interfaces, on the other hand, are performance critical, and they must be multiplexed unto a wider channel (MIG slave I/F is 512 bits) across a different clock domain, whereas multiple (narrow to wide), large burst transactions are in flight.[2] This interconnect, therefore, must be optimized for performance. To achieve this, we use custom-resource optimized RTL interconnects for the former and the Xilinx SmartConnect core for the latter. The cascading scheme used avoids the alternative 11:1 performance optimized interconnect which would waste resources to optimize paths (e.g., the scatter gather control paths) that we know from performance analysis and do not require high levels of performance or advanced AXI4 features.

C. Software Design

The proposed architecture uses a software scheduler running on Linux on the ARM processor to orchestrate the parallel compression accelerators. The scheduler is responsible for controlling data movements to and from the accelerators, monitoring progress and status as well as fetching raw image data from a sensor in the real-time scenario. Linux is configured to access the full SoC address space via the device tree and the appropriate drivers, including cores' register banks and both DDR memories. It should be noted that an alternative approach would be based on bare-metal software, i.e., lacking an operating system. In that approach, software can be easily written to be deterministic (hard real time) and have full access to physical memory, FPGA address space, and SoC interrupts. This would make programming the core system functions dramatically easier at design time but severely limit programmability, ease of debugging, introspection, and expanding the system in the long term.

The compressor cores programed into the FPGA and connected to the SoC bus are assigned addresses in the global address map of the SoC. As we use Linux, we must traverse the virtual address mapping to reach the physical address,

[1]The size of the compressed bitstream is not known before compression, and these buffers are sized for worst case uncompressible data.

[2]These requirements call for very large complexity in the interconnect logic, a greatly more logic intense design to a basic AXI4 interconnect.

which is controlled by the Linux kernel. There are three ways to provide to user space applications, the capability to access peripherals in Linux: by creating a driver for this specific peripheral, directly accessing `/dev/mem`, or via the User space IO (UIO) framework. In this work, we use the UIO approach, specifically the generic UIO driver which is well suited for simple devices such as the AXI4-Lite memory maps of the compressor cores.

The platform heavily relies on DMA operations to read/write data at speed to/from the compression accelerator IP Cores. To allow the controller application to configure and control the DMA cores, we use a stack of drivers: the Linux DMA API, the Xilinx DMA driver, and the axidma [33] driver. To avoid fragmentation, we reserve in the device tree a large region in the PL DDR memory and assign it to the DMA drivers to use via contiguous memory allocator (CMA), ensuring that large contiguous buffers are available for compression. The driver configuration aims to transfer the control of a DMA buffer from Linux user space to the DMA engine and to control the DMA engine. In the offline compression benchmark, for the downstream operation (MM2S), the buffer is filled by the user space application with sensor data, and then, the DMA engine is instructed to start. For the upstream operation (S2MM), the buffer is written by the DMA engine which then signals "finished" to the application. The kernel and the DMA driver stack is an intermediary during these operations. Note that no copy operations are performed to maintain high performance, and transfers refer to a notion of ownership.

The user-space scheduler application uses the APIs provided by the driver configuration for UIO and DMA. A single thread uses callbacks to be notified by the kernel for the completion of transactions on either transmit or receive channels. To command and check the status of the compression cores, the UIO drivers are used to `mmap()` into each compression core's address map and read/write registers at the appropriate offset.

The scheduler performs the following major tasks to compress one image in parallel.

1) Reads in the configuration file with compression parameters for the benchmark.
2) Allocates buffer pairs (transmit/receive) for each DMA channel.
3) Memory maps input data from the sensor to the DMA transmit buffer. In the benchmarking system, input data are mapped from a file, on a real-time scenario double buffered from the sensor.
4) Perform segmentation: by calculating the appropriate offsets (segment stride), the input image is split by the number of cores to map the correct segment into each DMA engine.
5) In debug mode, at this stage, sanity checks are performed by reading the compression cores' configuration registers, status, and multiple-input signature register (MISR) signatures.
6) The compression cores' status is checked to be in standby mode, and then, they are soft reset.
7) The compressors are configured according to the compression parameters specified in the configuration file.
8) They are then commanded to start compressing any incoming data. From this point on, actual data will reach the cores only when the DMA engines start.
9) Channel locks are initialized, and two callbacks are setup per DMA engine (transmit and receive).
10) Benchmark timing is started, the DMA transactions are started, and the main thread attempts to wait on the locks, once for each channel.
11) Eventually, compression finishes which invokes the callbacks, which release their lock once per channel. This finally unblocks the main thread when all callbacks have been executed.
12) Benchmark counters are stopped, the compressed data are written to files, and statistics are logged.
13) If operating in debug mode, the compressed files are compared to reference outputs and status, and MISR stream signatures are logged for all compression cores.

D. System-Level Integration Considerations

The implemented benchmarking system performs an offline, one-image-at-a-time compression sequence, which starts with an image at rest in an image buffer in DRAM and ends with the compressed data in a separate buffer in DRAM (comprising multiple adjacent, padded, compressed segment packets). The breadboard demonstrator platform implements the portion of the payload data chain that relates closely to the compressor. We make provisions by reserving bandwidth and buffers, for a real-time variant of the system that includes a typical continuous data path for a payload data-chain, from the sensor toward a ground station. This real-time scheme relies on two double buffers, for sensor readout and data downlink.

Commonly, in on-board payload data systems, logic to manage the imaging sensor exists in the context of ROE, which control sensor operation through interfaces such as SPI to configure acquisition parameters (e.g., synchronization, framerate, resolution, and so on). ROE electronics are nontrivial and sensor dependent; however, the pixel data channel for most sensors is a set of high-speed IOs connected to the payload FPGA. The protocol carrying the data varies and may be SpaceFibre, RapidIO, or other, but, in the common case, the final data are seen as a stream of pixels by the payload processor. For instance, the CHIEM [34] high-speed hyperspectral imager uses 64 high-speed LVDS links to stream pixel data.

On the other end, compressed data must be streamed out from DRAM toward the radio subsystem for downlink. Alternatively, compressed data may be streamed through a subsystem-local interface toward the spacecraft mass memory or payload storage subsystem (for later downlink depending on orbit phase). We refer to both operations as downlink for posterity, and from the perspective of the compression subsystem, they are both output streams.

The proposed scheme to integrate this compression architecture aims to perform the three operations in parallel, effectively pipelining sensor readout, compression and downlink. However, one cannot independently write from the sensor into the image buffer and read from the same buffer to compress.

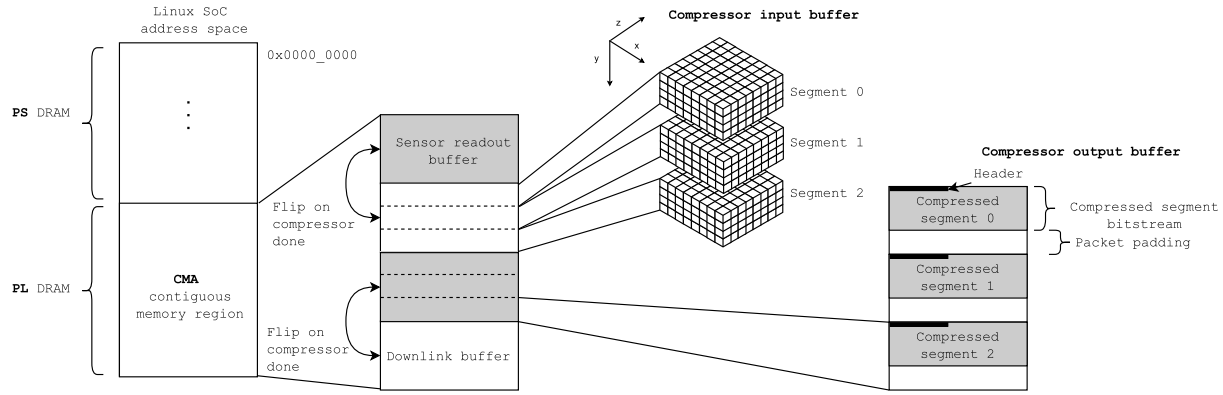

Fig. 7. Sensor and downlink double buffers' memory scheme.

Similarly, the compressed data buffer cannot be concurrently written by the compressor and read from the downlink component without some kind of synchronization. This situation is commonly solved using a double buffer (also known as ping-pong buffer) scheme, in this case one for image data and another for compressed data.

The real-time compression scheme pipelining sensor readout, compression with the parallel compressor, and downlink operations are shown in Fig. 7. A region of memory is reserved for the compression buffers shown on the left. This region is split further into four regions for two double buffers, incoming (readout) and outgoing (downlink). Operations start with the sensor stream writing into the upper readout buffer side (through S2MM DMA). When this first write operation is finished, the incoming double buffer flips, and the sensor stream of the second image is written to the other side of the incoming double buffer. Meanwhile, the compressors start reading segments from the previously finished first image. As compression of the first image segments progresses, the compressors write the compressed segments' bitstreams into the upper side of the outgoing double buffer. Similarly, when compression is finished, the outgoing double buffer flips, and downlink starts streaming out compressed data (first image), whereas the compressors continue writing segment bitstreams (second image) into the other side of the outgoing double buffer.

It should be noted that this scheme requires four times the compression throughput to be available as aggregate memory bandwidth (sensor write + compressors read + compressors write + downlink read) and four times the size of an image in DRAM. Typical hyperspectral images are in the low hundreds of MB, so this amount of DRAM is commonly available. Memory bandwidth, however, is at a premium, especially due to the high speed of the proposed compressor at scale; in Section IV-B that follows, an analysis evaluates the limits that can arise depending on the FPGA device and memory used.

IV. Experimental Results

A. Design Implementation

The architecture is implemented targeting the xc7z045-2 device on the Xilinx ZC706 development board, to verify operation and benchmark compression performance. For verification, we use the Siemens-Mentor Questa simulator and the VUnit Open Source unit testing framework, to simulate a high coverage test suite. Furthermore, in on-chip tests, we verify lossless operation by decompression and comparison to the originally compressed data. For synthesis and Place and Route (P&R), we use Vivado 2018.2, gearing optimizations for performance. The design is parametric at a high level with generics, in the number of compressor cores, as well as the ranges of various compression parameters which are tuned in the implementation results for an AVIRIS sensor scene with dimensions $(N_x, N_y, N_z) = (680, 512, 224)$, a common benchmark image. The image size is important due to BRAMs limiting the number of cores that can fit in a device. The lower limit of BRAMs required by the compression cores for N cores are

$$\text{BRAMs} \geqslant N * \left(6 * \left\lceil \frac{N_z}{1024} \right\rceil + \frac{(N_x * N_z)}{2048} + \left\lceil \frac{3 * \left\lceil \frac{N_z}{1024} \right\rceil}{2} \right\rceil \right).$$

The impact of image size and spatial–spectral aspect ratio to scaling and performance, besides BRAM utilization, is minor. If the segment size is too small, due to both very few bands (multispectral image) and a small number of columns (N_x), then performance may suffer. Specifically, if the segment size is smaller than the burst size used for DMA, incomplete bursts and smaller memory accesses will create overheads in data movement in the SoC. However, the burst size in samples (1024) is extremely small compared to hyperspectral image sizes, making it very unlikely that the segmentation of a real image would approach this limit.

The architecture shown in Fig. 6 does not show clocks, resets, and interrupts for clarity; there are multiple clock domains in the SoC: the PS clock at 666 MHz and PS DRAM at 533 MHz, the PL DRAM at 800 MHz and MIG controller at 200 MHz, the PL side SoC interconnect (AXI4-Lite and AXI4 I/Fs toward MIG) clocked at 200 MHz, and finally, the compressor cores at 285 MHz. Clock domain crossings are negotiated with synchronizers or dual-clock FIFOs, and each clock domain has an associated reset forming a reset tree, deasserting based on the PLL locking sequence.

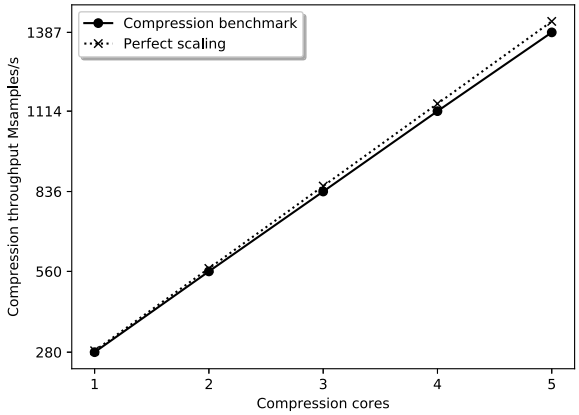

Fig. 8. Parallel compression scaling benchmark.

Table II shows implementation resources after P&R, for 1, to 5 compression cores for image size $(N_x, N_z) = (680, 224)$. Table III shows the utilization breakdown per SoC high level component family for LUTs. This highlights that the Xilinx IP cores used for the SoC data movement and interconnects have nontrivial resource requirements. Note that miscellaneous components, e.g., reset generators, are not shown separately but are included in the total counts. The throughput column shows benchmarked performance, where the test image was compressed on chip, timing wall time from command start of the first DMA core to the finished interrupt of the last DMA core.

Fig. 8 shows on-chip compression throughput benchmark results with the parallel compressor and also the perfect linear scaling line to highlight losses from ideal parallelism. Up to 5 cores, throughput losses are between 1.7% and 2.7% compared to perfect scaling; performance increases almost linearly by adding cores until the targeted FPGA is fully utilized (in BRAMs) at five compressor cores. Each compressor IP core operates on one 16-bit sample per clock cycle; therefore, throughput Msamples/s should converge to compressor clock megahertz given images large enough to amortize the cost of initial latency. Overall throughput in Msample/s is ideally clock MHz * N cores if the performance bottleneck is the compression operation, which is ensured with the appropriate clocking and bus width in the data path.

Although the throughput loss is too small to warrant investigation, the following sources of losses may impact performance in the system.

1) Each compressor initializes internal state after being commanded to start and before consuming samples. This latency is proportional to the number of bands in the input image.
2) Benchmarking is pessimistic and starts timing before the start command is sent: this adds delay, whereas the scheduler loops to command compressors and DMAs.
3) Packetization: input and output streams from the compressors are packetized. Each packet has a small latency to setup (in software) and perform (in hardware) the DMA transaction.
4) Other software is running on the ARM CPU sharing memory and compute. This may induce unknown delays in the scheduler operations.

TABLE II
POST P&R RESOURCES AND THROUGHPUT AT 285 MHz

Cores	LUTs	BRAMs	FFs	DSPs	Throughput (Msamples/s)	TOCP† (W)
1	36028	90	42492	6	280.08	5.33
2	49907	179	61639	12	559.66	6.31
3	64342	269	80731	18	836.14	7.21
4	78009	358	99605	24	1114.32	8.11
5	91845	448	118992	30	1386.97	8.82

† Total On-Chip Power, Vivado power estimator.

TABLE III
POST P&R LUT UTILIZATION PER COMPONENT

Cores	CCSDS 123.0-B-1E Cores	DMAs	Interconnects	Total
1	8505	2229	11911	36028
2	17008	4458	15069	49907
3	25489	6687	18785	64342
4	34001	8911	21720	78009
5	42564	11147	24760	91845

B. Performance Limit Analysis

This section discusses possible and realized performance bottlenecks at various levels of performance for the system as a whole. The limits shown are specific to the targeted device and board; different systems may have different DRAM configurations (bandwidth, frequency, and number of banks), speed grades ($F_{\max}$ for RTL components), FPGA size (higher resource limit), or other limitations that cannot be accounted for in the general case here (e.g., limited or abundant LVDS IOs to the sensor/downlink and so on).

Note that the AXI4 protocol has separate read and write address/data channels; therefore, the maximum bandwidth at bus width $\times$ frequency is independent for incoming and outgoing channels on an interface. On the other hand, transactions with memory are on a half-duplex channel; therefore, we consider aggregate transmit and receive throughput. In the case of compression, requirements are symmetric on receive and transmit[3], so aggregate throughput is appropriate. In other applications besides compression, a throughput for the two directions may be better considered separately.

The limits to consider for the targeted device are as follows.

1) PS DRAM memory connected to the hard memory controller in the ARM PS, bandwidth: 4264 MB/s.
2) PL DRAM memory connected to the transceivers of the FPGA fabric IOs, interfaced using a soft memory controller, bandwidth: 12 800 MB/s.
3) Four available HP ports, with a maximum width of 64 bits, at 200 MHz, aggregate bandwidth: 12 800 MB/s.
4) DMA stream side maximum aggregate throughput 64 bits at 200 MHz: 3200 MB/s per DMA core.
5) 218.6K LUTs and 19.2-Mb device BRAM.
6) Compression throughput/core at 285 MHz: ~570 MB/s.

[3]Symmetric only in the worst case, commonly the compressed data rate is a fraction of the raw data rate. For the purposes of system design, we assume the worst case of un-compressible data and in practice enjoy a generous safety margin.

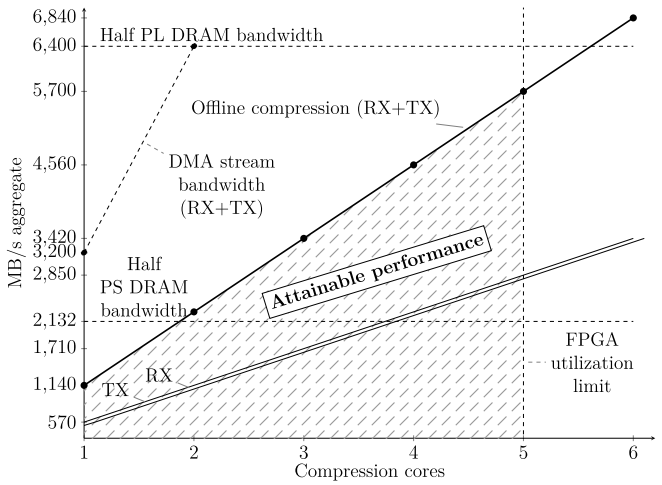

Fig. 9. Scaling bounds: *offline compression*.

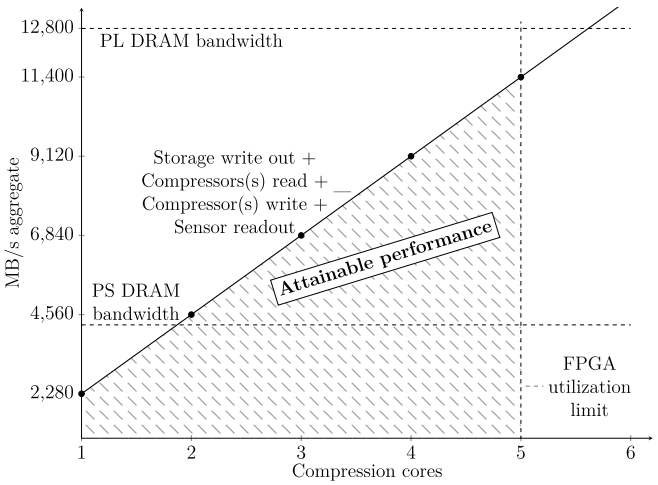

Fig. 10. Scaling bounds: *real-time compression*.

The scaling limits plots in Figs. 9 and 10 show the potential and actual system scaling limits related to the number of parallel compressor cores of the proposed architecture. Note that both figures show the same scenario; however, Fig. 9 shows the breadboard system as benchmarked in this work, i.e., assuming half of DRAM bandwidth is available. This is to allow provisioning for the real-time double buffer-based system as described in Section III-D. The detailed scaling limits in the double-buffer readout and downlink scenario are shown in Fig. 10 (N.B. DRAM bandwidth limit is the full bandwidth limit).

In both cases, targeting the xc7z045 device and a $(N_x, N_y, N_z) = (680, 512, 224)$ image, parallelism is limited by FPGA BRAMs. At 5 cores, the device is almost full (80% BRAM utilization) and cannot place a sixth core without severely deteriorating P&R quality of results due to congestion. However, even if via BRAM usage optimizations, a 6-core design was implemented (or targeting the larger xc7z100 device), another limit would be met before full compression performance was realized, the PL DRAM bandwidth limit.

In the platform as configured, the compressor cores can access either the PS or PL DRAM memories on the board via the HP ports and hard PS memory controller or the MIG memory controller on the FPGA, respectively. The architecture reserves the PL DRAM exclusively for the compression operation both in terms of memory space and available bandwidth. If that were not the case and PS memory was used for compression buffering, overall performance would be limited at about 2 cores, ∼570 Msamples/s.

The scaling plots as a design aid, besides making apparent limits in the system, shows potential optimizations as well. Consider that all other limits, besides the lowest performance limit that is actually met, can be exploited for resource and power saving improvements. The DMA stream limit and HP port limits are far from being close to constrain compression performance; therefore, the relevant clocks can be set much lower than their maximum for power savings. In the case of DMA streaming, this assumes that the compressor interfaces via clock domain crossing, asymmetric aspect ratio AXI4-Stream FIFOs. Similarly, some AXI4 interconnects can be configured to minimize buffering (BRAM resources) since they are not operating close to their maximum performance.

To summarize, the performance scaling limit analysis informs the following architectural decisions: 1) the system needs to use the PL DRAM for performance; 2) optimizing for BRAM resources to fit 6 cores is not needed as, beyond 5 cores' performance will be limited by PL DRAM bandwidth; 3) up to 2 cores could use PS DRAM via the HP ports avoiding the MIG and second DRAM altogether; and 4) SoC interconnect is not the performance limit, therefore, can be clocked lower, or optimized for resources.

C. Comparison With Prior Work

This section presents in Table IV, a comparison with implementations from the literature, which uses a parallel approach. In benchmarking on-chip, the proposed architecture reaches 1387 Msamples/s compression throughput performance with 5 cores on a Zynq-7045 device, an ∼1.85× increase over the previous state of the art. For the Kintex-7 technology used for benchmarking, performance is limited by the size of the chosen FPGA and the DRAM bandwidth of the development board to 5 cores. Although the proposed work implements a coarse-grain parallel architecture, applying fine-grain parallelism without an embedded processor is also possible. Other works have successfully used FPGA SoCs to compress hyperspectral data, Nascimento *et al.* [35] implement compressive sensing on a Zynq platform with very high energy efficiency.

The fine-grain parallelism approach has been implemented on FPGA first by Báscones *et al.* [36] and later refined and improved by Orlandić *et al.* [9]. Orlandić *et al.* [9] proposed a parallel implementation building up from their previous work in [21]. The CCSDS-123-B-1 single core is adapted for parallel processing, and the organization of several compression pipelines in parallel is implemented. The limitations of data routing between predictor–encoder pairs and the packing operation in [36] are successfully overcome,

TABLE IV
COMPARISON OF CCSDS 123.0-B-1/2 PARALLEL FPGA ACCELERATORS

	Orlandić et al. [9]	Rodríguez et al. [12]	Keymeulen et al. [11]	Proposed work
Compression mode	lossless	lossless	near-lossless	lossless
Image size (N_x, N_y, N_z)	512, 2000, 128	512, 512, 220	640, 480, 480	680, 512, 224
Pixel order	BIP	BSQ	BIP	BIP
Robustness measures	None	Segmentation & modular redundancy	Segmentation	Segmentation
LUTs	14709	51070‡	297807	91845
FFs	12830	26830‡	260750	118992
BRAMs	37	256‡	556	448
Target device (#part)	Kintex-7 (7Z035)	Kintex-7 (7Z100)	Virtex-7 (VX690T)	Kintex-7 (7Z045)
Speed grade	2	2	3	2
Power (W)	0.515	2.617	11.40	8.82
Parallelism	5	16	15	5
Throughput (MSamples/s)	**750***	**67.04**†	**106**†	**1386.97**†

* Static timing analysis estimate.
† Experimental FPGA-in-the-loop performance results.
‡ Authors note resources per compressor kernel, 2932 LUTs, 1529 FFs, 16 BRAMs.

increasing performance. Overall compression throughput is increased by reducing stall cycles with an improved scheduling of the predictor–encoder pipelines and also by improving variable length code packing with a high-performance parallel input packer. The work in [9] uses a fine-grain parallelism scheme, where multiple N_p predictor–encoder pairs, each with a sample-per-cycle microarchitecture, are scheduled to process N_p samples in parallel, feeding into a N_p-input packer component. On a Kintex-7 device, the implementation achieves 750 Msamples/s and is highly resource efficient by using RTL generics for image and compression parameters that are fixed at runtime.

Similar segmentation schemes as used in this work have been presented in the literature to exploit data parallelism and improve the error containment [11], [12]. Rodríguez *et al.* [12] used ARTICo^3 [32] to schedule a number of compressors on the FPGA, compressing blocks of the image in parallel. Due to limitations in the ARTICo^3 framework in the memory per accelerator, the authors use a square block partitioning of the image (across the X- and Y-axes) which leads to smaller memory footprint. The implementation targets BSQ pixel order and reaches 67.04 Msamples/s throughput on a Kintex-7 device using 16 compressor cores, a state-of-the-art performance for BSQ order hyperspectral compression. The authors report almost linear scalability up to 8 cores, proving that they operate in a computing-bounded region. However, using 16 accelerators only renders 9.7× speedup, which indicates that the system is then entering the memory-bounded region. Such implementation has properties favorable to system integrators, i.e., the ability to change at runtime the number of cores via dynamic partial reconfiguration or redundantly compress image blocks.

The NASA JPL implementation by Keymeulen *et al.* [11] as a part of a complete avionics system for data acquisition, cloud screening, and data compression implements compression with FLEX, an FPGA implementation of lossless and near-lossless compression according to CCSDS 123.0-B-2. Due to the complexity and data dependencies in near-lossless mode, FLEX does not achieve a high performance in a single core, as opposed to other lossless-only compressors. The authors implement segmentation across the Y-axis to increase throughput performance and present a variant of FLEX in an airborne demonstration using COTS parts. On an Alpha Data board with a Virtex-7 VX690T-3 FPGA, FLEX achieves 106 Msamples/s using 15 cores and 32-line high segments. A software component on a ruggedized COTS PC performs real-time data analysis and schedules data movements between sensor, FPGA, and CPU over PCIe. The authors also describe a FLEX variant in a SoC architecture targeting a Zynq Z7045Q for the 2024 EMIT mission.

All works in Table IV use the Xilinx tool flow. The implementation in [9] reports static timing analysis estimate extrapolating performance from $F_{\max}$, whereas this proposed work and [11] and [12] report experimental FPGA-in-the-loop performance benchmark results. The sample adaptive encoder option is used in all implementations, except [11] which uses the new hybrid encoder of the latest version of the compression standard.

On scalability, [9] reported a decrease in F_{max} when increasing parallelism whereas, in this work, the scaling overhead is

very limited; i.e., per-core throughput is marginally affected when increasing parallelism. On reliability, although all implementations target COTS parts, this work and the NASA JPL implementation [11] use segmentation taking the advantage of error containment, whereas [12] additionally to segmentation may use modular redundancy on the compression cores. The implementation in [9] describes only an FPGA IP core and not a full avionics or SoC system; therefore, resource and power comparisons are qualitative. However, [9] is significantly more resource and power efficient than the other implementations compared.

The proposed segment-parallel scheme requires more resources compared to [9], mostly due to the significant resource overhead of the SoC components used (interconnects, memory controllers, and so on) and due to being based on a larger single-core implementation (comparing LUTs, 3012/core in [21], and 8505/core in [17] and [18]). These increased logic resources for the SoC peripherals account for the increased power consumption. Memory utilization in the segment-parallel approach, when implemented across the Y-axis, has memory requirements linear to the parallelism factor. Each parallel compressor may require on-chip storage for a spectral frame ($N_x \cdot N_z$ samples) causing an increased BRAM utilization. However, the segment parallel approach allows linear performance scaling, bounded only by device size. Furthermore, the approach is not specific to the accelerator IP Core; in principle, [9] could be used in the parallel SoC designed in this work. In that way benefiting from both approaches, a higher performance baseline per core and system level scalability and reliability at the same time.

The proposed architecture sets a new state-of-the-art compression throughput performance at 1387 Msamples/s utilizing 91 845(42.02%) LUTs for a full SoC with 5 cores on the Zynq-7045 device.

V. Conclusion

In this work, we have introduced a high-performance parallel SoC implementation of the CCSDS 123.0-B-1 hyperspectral compression algorithm targeting COTS FPGA SoC technology to optimize SWaP-C. The proposed architecture exploits image segmentation to provide an increased robustness to data corruption and enable scalable throughput performance by leveraging segment-level parallelism. A flexible software scheduler hosted in the PS of the FPGA SoC controls the compression accelerator cores in the PL fabric, orchestrates DMA transactions over the on-chip bus interconnects, and can also serve other system-level payload data processing SoC functionality. The proposed parallel architecture is demonstrated on chip by implementation on a Zynq-7045 FPGA device with 5 compression cores and achieves a throughput performance of 1387 Msamples/s (22.2 Gb/s at 16 bpppb), which outperforms previous implementations in equivalent FPGA technology and allows seamless integration with next-generation hyperspectral sensors.

References

[1] P.-P. Mathieu *et al.*, "The ESA's earth observation open science program [space agencies]," *IEEE Geosci. Remote Sens. Mag.*, vol. 5, no. 2, pp. 86–96, Jun. 2017.

[2] J. Transon, R. d'Andrimont, A. Maugnard, and P. Defourny, "Survey of hyperspectral earth observation applications from space in the sentinel-2 context," *Remote Sens.*, vol. 10, no. 3, p. 157, Jan. 2018.

[3] C. M. Lee *et al.*, "An introduction to the NASA hyperspectral InfraRed imager (HyspIRI) mission and preparatory activities," *Remote Sens. Environ.*, vol. 167, pp. 6–19, Sep. 2015.

[4] L. Guanter *et al.*, "The EnMAP spaceborne imaging spectroscopy mission for earth observation," *Remote Sens.*, vol. 7, no. 7, pp. 8830–8857, Jul. 2015.

[5] S. Pignatti *et al.*, "The PRISMA hyperspectral mission: Science activities and opportunities for agriculture and land monitoring," in *Proc. IEEE Int. Geosci. Remote Sens. Symp. (IGARSS)*, Melbourne, VIC, Australia, Jul. 2013, pp. 4558–4561.

[6] J. Nieke and M. Rast, "Towards the Copernicus hyperspectral imaging mission for the environment (CHIME)," in *Proc. IEEE Int. Geosci. Remote Sens. Symp. (IGARSS)*, Valencia, Spain, Jul. 2018, pp. 157–159.

[7] N. Fox *et al.*, "Traceable radiometry underpinning terrestrial- and heliostudies (TRUTHS)," *Adv. Space Res.*, vol. 32, no. 11, pp. 2253–2261, Dec. 2003.

[8] J. Blommaert *et al.*, "CSIMBA: Towards a smart-spectral cubesat constellation," in *Proc. IEEE Int. Geosci. Remote Sens. Symp. (IGARSS)*, Yokohama, Japan, Jul. 2019, pp. 4614–4617.

[9] 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, Mar. 2019.

[10] L. Santos, A. Gomez, and R. Sarmiento, "Implementation of CCSDS standards for lossless multispectral and hyperspectral satellite image compression," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 56, no. 2, pp. 1120–1138, Apr. 2020.

[11] D. Keymeulen *et al.*, "High performance space data acquisition, clouds screening and data compression with modified COTS embedded system-on-chip instrument avionics for space-based next generation imaging spectrometers (NGIS)," in *Proc. 6th ESA Int. Workshop Board Payload Data Compress.*, Matera, Italy, Sep. 2018.

[12] A. Rodriguez, L. Santos, R. Sarmiento, and E. De La Torre, "Scalable hardware-based on-board processing for run-time adaptive lossless hyperspectral compression," *IEEE Access*, vol. 7, pp. 10644–10652, 2019.

[13] *CCSDS Recommended Standard for Lossless Data Compression*, Standard CCSDS 121.0-B-2, CCSDS Recommendation for Space Data System Standards, May 2012.

[14] *CCSDS Informational Report for Lossless Multispectral & Hyperspectral Image Compression*, Standard CCSDS 120.2-G-1 Green Book, CCSDS, 2015, no. 1.

[15] M. Klimesh, "Low-complexity adaptive lossless compression of hyperspectral imagery," *Proc. SPIE*, vol. 6300, Sep. 2006, Art. no. 63000N.

[16] *CCSDS Recommended Standard Lossless Multispectral & Hyperspectral Image Compression*, Standard CCSDS 123.0-B-1, CCSDS Recommendation for Space Data System Standards, May 2012.

[17] 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 Trans. Emerg. Topics Comput.*, early access, Jul. 12, 2018, doi: 10.1109/TETC.2018.2854412.

[18] A. Tsigkanos, N. Kranitis, and A. Paschalis, "CCSDS 123.0-B-1 multispectral & hyperspectral image compression implementation on a next-generation space-grade SRAM FPGA," in *Proc. 6th ESA Int. Workshop Board Payload Data Compress.*, Matera, Italy, Sep. 2018, pp. 21–22.

[19] D. Keymeulen, N. Aranki, A. Bakhshi, H. Luong, C. Sarture, and D. Dolman, "Airborne demonstration of FPGA implementation of fast lossless hyperspectral data compression system," in *Proc. NASA/ESA Conf. Adapt. Hardw. Syst. (AHS)*, Jul. 2014, pp. 278–284.

[20] Y. Barrios, A. J. Sanchez, 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. 54269–54287, 2020.

[21] J. Fjeldtvedt, M. Orlandic, and T. A. Johansen, "An efficient real-time FPGA implementation of the CCSDS-123 compression standard for hyperspectral images," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 11, no. 10, pp. 3841–3852, Oct. 2018.

[22] A. Hofmann, R. Wansch, R. Glein, and B. Kollmannthaler, "An FPGA based on-board processor platform for space application," in *Proc. NASA/ESA Conf. Adapt. Hardw. Syst. (AHS)*, Jun. 2012, pp. 17–22.

[23] M. Cohen, A. Larkins, P. L. Semedo, and G. Burbidge, "NovaSAR-S low cost spaceborne SAR payload design, development and deployment of a new benchmark in spaceborne radar," in *Proc. IEEE Radar Conf. (RadarConf)*, May 2017, pp. 0903–0907.

[24] G. Lentaris *et al.*, "High-performance embedded computing in space: Evaluation of platforms for vision-based navigation," *J. Aerosp. Inf. Syst.*, vol. 15, no. 4, pp. 178–192, Apr. 2018.

[25] T. M. Lovelly and A. D. George, "Comparative analysis of present and future space-grade processors with device metrics," *J. Aerosp. Inf. Sys.*, vol. 14, no. 3, pp. 184–197, 2017.

[26] X. Iturbe *et al.*, "An integrated SoC for science data processing in next-generation space flight instruments avionics," in *Proc. IFIP/IEEE Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Oct. 2015, pp. 134–141.

[27] D. Rudolph *et al.*, "CSP: A multifaceted hybrid architecture for space computing," in *Proc. 28th Annu. AIAA/USU Conf. Small Satell.*, 2014, pp. 1–7.

[28] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, "Mitigation of radiation effects in SRAM-based FPGAs for space applications," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 1–34, Jan. 2015.

[29] M. Amrbar, F. Irom, S. M. Guertin, and G. Allen, "Heavy ion single event effects measurements of Xilinx Zynq-7000 FPGA," in *Proc. IEEE Radiat. Effects Data Workshop (REDW)*, Jul. 2015, pp. 1–4.

[30] J. Karp, M. J. Hart, P. Maillard, G. Hellings, and D. Linten, "Single-event latch-up: Increased sensitivity from planar to FinFET," *IEEE Trans. Nucl. Sci.*, vol. 65, no. 1, pp. 217–222, Jan. 2018.

[31] *Using LVAUX Mode in XQ Ruggedized UltraScale+Devices for Airborne Systems Design Guide UG584*, Xilinx, San Jose, CA, USA, Dec. 2019.

[32] A. Rodríguez, J. Valverde, J. Portilla, A. Otero, T. Riesgo, and E. de la Torre, "FPGA-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: The ARTICo3 framework," *Sensors*, vol. 18, no. 6, p. 1877, Jun. 2018.

[33] B. Perez. *A Zero-Copy Linux Driver and a Userspace Interface Library for Xilinx's AXI DMA and VDMA IP Blocks*. Accessed: Nov. 20, 2018. [Online]. Available: https://github.com/bperez77/xilinx_axidma

[34] J. Blommaert *et al.*, "CHIEM: A new compact camera for hyperspectral imaging," in *Proc. Small Satell. Conf.*, Logan, UT, USA, Aug. 2017, pp. 1–14.

[35] J. M. P. Nascimento, M. P. Véstias, and G. Martín, "Hyperspectral compressive sensing with a system-on-chip FPGA," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 13, pp. 3701–3710, 2020.

[36] D. Báscones, C. González, and D. Mozos, "Parallel implementation of the CCSDS 1.2.3 standard for hyperspectral lossless compression," *Remote Sens.*, vol. 9, no. 10, p. 973, Sep. 2017.