Article

**A Parallel FPGA Implementation of the CCSDS-123 Compression Algorithm**

Milica Orlandić¹*, Johan Fjeldtvedt¹ and Tor Arne Johansen²

¹ Department of Electronic Systems, Norwegian University of Science and Technology, 7491 Trondheim, Norway; jaffe1@gmail.com
² Centre for Autonomous Marine Operations and Systems (NTNU-AMOS), Department of Engineering Cybernetics, Norwegian University of Science and Technology, 7491 Trondheim, Norway; tor.arne.johansen@ntnu.no

* Correspondence: milica.orlandic@ntnu.no

Received: 27 February 2019; Accepted: 16 March 2019; Published: 21 March 2019

**Abstract:** Satellite onboard processing for hyperspectral imaging applications is characterized by large data sets, limited processing resources and limited bandwidth of communication links. The CCSDS-123 algorithm is a specialized compression standard assembled for space-related applications. In this paper, a parallel FPGA implementation of CCSDS-123 compression algorithm is presented. The proposed design can compress any number of samples in parallel allowed by resource and I/O bandwidth constraints. The CCSDS-123 processing core has been placed on Zynq-7035 SoC and verified against the existing reference software. The estimated power use scales approximately linearly with the number of samples processed in parallel. Finally, the proposed implementation outperforms the state-of-the-art implementations in terms of both throughput and power.

**Keywords:** CCSDS-123 compression; parallel implementation; Field programmable gate arrays (FPGA); hyperspectral imaging; real-time processing

**1. Introduction**

In recent years, space development has moved towards small-satellite (SmallSat) missions which are characterized by capable low-cost platforms with introduced budget and schedule flexibility. Space-related applications, such as synthetic aperture radar (SAR), multispectral and hyperspectral imaging (HSI) require critical data processing to be performed onboard in order to preserve transmission bandwidth. In this respect, the compression algorithms are commonly used as a final step in onboard processing pipelines to reduce memory access and limit data transfer to Earth. To fulfill real-time data processing requirement, hybrid processing systems with reconfigurable hardware (FPGAs) have become the standard choice in small-satellite missions. The expansion of logic resources in the current FPGAs allows execution of complex algorithmic tasks in parallel and the trend for CubeSats and other SmallSat single-board computers is to use common SoC devices with commercial FPGAs due to their superior performance in terms of power, speed and resources compared to radiation-hardened FPGAs [1].

Hyperspectral and multispectral imaging have been both widely used in remote sensing Earth observation missions in recent decades. Unlike multispectral sensors, such as Landsat, MSG and MODIS [2], with a fairly limited number of discrete spectral bands, hyperspectral sensors record a very large number of narrow spectral bands. Airborne hyperspectral sensors such as Compact Airborne Spectrographic Imager (CASI), Airborne Visible/InfraRed Imaging Spectrometer (AVIRIS) [3], Infrared Atmospheric Sounding Interferometer (IASI) [4] and the Hyperspectral Imager for the Coastal Ocean (HICO) [5,6] have provided an expansion of hyperspectral research in the number of applications
such as environmental monitoring, coastal ecosystems, geology and land cover. A hyperspectral imager has recently been deployed on an intelligent nano-satellite [7], where a key feature is intensive onboard data processing including operations such as comparisons of images in subsequent orbits. However, a mission with HSI payload to fulfill its objectives, in addition to smart onboard processing, requires compression for downlink of the acquired data.

The Consultative Committee for Space Data Systems (CCSDS) has developed image compression algorithms [8–11] specifically designed for space data systems. In particular, the CCSDS-123 compression standard [10,11] is an efficient prediction-based algorithm characterized by low complexity and, thus, is suitable for real-time hardware implementation. In fact, in the recent years several FPGA implementations of the CCSDS-123 standard are presented in the literature [12–19]. Keymeulen et al. [12] propose an on-the-fly implementation in BIP sample ordering. In the implementation proposed by Santos et al. [13], the focus is on low complexity and low memory footprint. The chosen BSQ sample ordering requires only one weight vector and one accumulator to be stored. However, the repeated computations of local differences decrease the input bandwidth efficiency. This approach requires either the non-sequential memory access pattern with potentially reduction of streaming efficiency, or that the data is arranged in memory in the desired streaming order. The serial CCSDS-123 implementation with BIP ordering proposed by Theodorou et al. [14] relies on external memory to buffer samples coming from the image sensor such that the current, N and NE neighboring samples are streamed in parallel reducing greatly on-chip memory requirements. The downside is, however, lack of support for on-the-fly compression. Báscones et al. [15] propose an implementation with BIP sample ordering characterized by the ability to perform compression without relying on external memory. This is achieved by queuing incoming samples in internal FIFOs, resulting in linear dependence of memory usage with respect to the product of width and depth of a HSI cube. A parallel CCSDS-123 implementation proposed by Báscones et al. [16] consists of several instances of the CCSDS-123 core that share local differences. Other than sharing local differences, the cores operate independently by processing samples from a fixed subset of bands.

In this paper, an efficient parallel FPGA implementation of the CCSDS-123 compression algorithm is proposed. The high throughput is achieved by the use of several optimization techniques for data routing between parallel processing pipelines and for efficient parallel packing. In the proposed solution, parallel processing of several samples is only constrained by the logic resources of the chosen technology.

The paper is structured as follows: Section 2 presents an overview of the CCSDS-123 standard. The proposed parallel hardware implementation is described in Section 3. The influence of the number of pipelines and chosen architectural solutions on the logic use, timing and power are analyzed in Section 4. Finally, the conclusions are given in Section 5.

2. Background

The CCSDS-123 standard for lossless data compressors is applicable to 3D HSI data cubes produced by multispectral and hyperspectral imagers and sounders, where a 3D HSI data cube is a three-dimensional array \((N_x, N_y, N_z)\). A sample in the HSI cube is specified by coordinates \((x, y, z)\), whereas an HSI pixel is characterized by fixed \((x, y)\) coordinates and consists of \(N_z\) components in spectral domain. The standard supports Band Interleaved (BI) and Band Sequential (BSQ) orderings for scanning the HSI coordinates. Special cases of BI ordering are Band Interleaved by Pixel (BIP) and Band Interleaved by Line (BIL). BSQ ordering traverses the components band by band - \((z, y, x)\) order. In BIP ordering, each full pixel is accessed sequentially in \((y, x, z)\) order. In BIL ordering, traversing is performed frame by frame in \((z, x)\) order.

The integer samples of the HSI cube are labeled as \(s_{z,y,x}\) or \(s_z(t)\) where \(t = y \cdot N_x + x\). The sample \(s_{z,y,x}\) is predicted by computation of a local sum \(\sigma_{z,y,x}\) of nearby predicted samples \((s_{z,y,x-1}, s_{z,y-1,x-1}, s_{z,y-1,x}, s_{z,y-1,x+1})\) at positions \((W, NW, N, NE)\) with respect to sample \(s_{z,y,x}\). The reduced prediction mode computes central local differences \(d^k\) for previously processed bands \(k = 0, \ldots, P\) as
\[d^k = 4 \cdot (s_{z-k,y,x}) - \sigma_{z-k,y,x}\], whereas full prediction mode includes also directional differences \([d^W, d^{NW}, d^N] \) between neighbor samples \((4 \cdot s_{z,y,x-1}, 4 \cdot s_{z,y-1,x-1}, 4 \cdot s_{z,y-1,x})\) and local sum \(\sigma_{z,y,x}\), respectively. The created differences are then stored in the local difference vector \(U_z(t)\). Predictor parameters such as the number of prediction bands \(P\), the local sum type and the prediction mode impact significantly the overall performance of the CCSDS-123 standard, and suggested non-normative default values of these parameters provide a reasonable trade-off between performance and complexity \([10,20]\).

The computation of the rounded scaled predicted sample includes the dot product operation of weight vector \(W_z(t)\) and local difference vector \(U_z(t)\) and shifting operation of local sum \(\sigma_z(t)\) by parameter \(\Omega\) which is defined as bit precision of the weight elements. The scaled predicted sample value \(\hat{s}_z(t)\) is a version of the rounded scaled predicted sample in the range \([-2^D, 2^D]\) for signed integers, where \(D\) is a dynamic range of HSI samples. The weights are dynamically updated based on the prediction error \(e_z = 2s_z(t) - \hat{s}_z(t)\) by the weight update factor \(\Delta W_z(t)\) which depends on several user-defined parameters which control convergence speed of the learning rate at which the predictor adapts to the image statistics. The scaled predicted sample value \(\hat{s}_z(t)\) is re-normalized to the range of the input sample \((D\text{-bit quantity})\) resulting in \(\hat{s}_z(t)\). Finally, the residual mapping converts the signed predicted residual \(\Delta_z(t) = s_z(t) - \hat{s}_z(t)\) to a \(D\)-bit unsigned integer mapped prediction residual \(\delta_z(t)\).

In the sample-adaptive encoding, code words are generated based on the average value of the input residuals in each band. The encoder updates an accumulator \(\Sigma_z(t)\) by storing recent sample values and then divides the result by the counter \(\Gamma(t)\) which tracks the number of processed samples. A code word generator computes quotient and residual pair, \((u_z,r_z)\) from the division \(\frac{\delta_z(t)}{\Gamma_z(t)}\), where the parameter \(k_z(t)\) is defined as the largest non-negative integer satisfying an inequality expression depending on relations between the accumulator \(\Sigma_z(t)\) and the counter \(\Gamma(t)\).

### 3. Implementation

The proposed parallel implementation contains \(N_p\) pipelines for concurrent processing of multiple samples and shared resources for storing intermediate data. The block diagram of the proposed implementation for \(N_p = 4\) is shown in Figure 1.

![Figure 1. Overview of the parallel CCSDS-123 implementation for \(N_p = 4\).](image)

A number of data samples are streamed into the shared sample delay module in each clock cycle. The samples are rearranged and sent to pipelines where a pipeline contains a chain of modules performing the local sum and difference computation, prediction, residual mapping and sample adaptive encoding as illustrated in Figure 2. The predicted data computed in central local differences, the updated weight vector elements and accumulator values in sample adaptive encoding are routed
to the central difference store, the weight store and accumulator store modules, respectively, which are shared between the pipelines.

The data packages streamed into the CCSDS-123 core contain \( N_p \) samples. A lane is defined as a position of samples in the input package. Figure 3a,b show the sample placement grids for \( N_p = 4 \) lanes in the first 10 clock cycles for the number of bands is \( N_z = 8 \) and \( N_z = 9 \), respectively. The first sample in each pixel is highlighted. When the number of bands is divisible by the number of pipelines i.e., \( N_z \mod N_p = 0 \), lane \( i \) contains a fixed subset of bands for each pixel so that the sample from band \( z \) is always streamed in the lane \( i = z \mod N_p \). For \( N_z \) not divisible by \( N_p \), samples from the same band are no longer confined to a specific lane. Instead, samples shift between lanes. After streaming the last sample in a pixel, the input stream can be stalled so that the first sample of the next pixel is in lane \( i = 0 \). In this manner, a fixed subset of samples is processed by each pipeline similarly to the case when \( N_z \) is divisible by \( N_p \). The downside of the introduced stalling is reduced throughput and additional logic. To avoid stalling, an interleaved pipeline approach is proposed. In this approach, samples from the same bands are processed in different pipelines, requiring from pipelines to share additional information besides local differences. For instance, a sample arriving to the sample delay module in lane \( i = 0 \) is also sent to pipeline 2 as the neighbor of a sample arriving to lane \( i = 2 \). Furthermore, vector \( W_0(1) \) is produced by pipeline 0 when processing \( s_0(0) \), but it is then also used by pipeline 1 when processing \( s_0(1) \). The advantage of the interleaved approach is a generic implementation with maximized throughput and independent of the parameters \( N_z \) and \( N_p \).

In the proposed interleaved approach, the data shifting is introduced for moving data from different lanes to corresponding processing pipelines. If a current sample is in lane \( i \), then the sample with distance \( n \) from the current sample is in lane \( (i + n) \mod N_p \). The distance between neighboring samples \( s_z(t) \) in lane \( i \) and \( s_z(t + \Delta t) \) is given as \( N_z \Delta t \), where the lane of sample \( s_z(t + \Delta t) \) is computed as:

\[
\text{shift}(i, \Delta t) = (i + N_z \Delta t) \mod N_p.
\]  

In Figure 3b, sample \( s_0(5) \) is streamed in the lane which is computed as \( \text{shift}(3, 2) = 0 \) based on the distance from sample \( s_0(3) \) from lane \( i = 3 \). Due to the data shifting, the number of clock cycles between samples within the same band is not constant. The number of clock cycles between two samples is equivalent to the number of rows between them in the grid. In the edge case, when sample \( s_z(t) \) is in the left-most lane, sample \( s_{z+1}(t) \) is streamed in the right-most lane of the next row. The time delay between \( s_{z+1}(t) \) and \( s_z(t) \) in lane \( i \) is computed as follows:

\[
\text{delay}(i, \Delta t) = \left\lfloor \frac{i + N_z \Delta t}{N_p} \right\rfloor.
\]
3.1. Pipeline

A pipeline contains a chain of modules implemented as described in the previous work [19] on a sequential CCSDS-123 implementation. To accommodate parallel processing, adaptation of the sequential modules are required. This includes several modifications such as setting FIFO depths and RAM sizes to $z/N_z$ instead of $z$.

3.2. Sample Delay

The sample delay module delays incoming samples so that the current sample and the previously predicted neighboring samples are available at its output. The proposed parallel implementation of sample delay module is shown in Figure 4.

![Sample placement timing diagram](image)

**Figure 3.** Sample placement timing diagram, (a) $N_z = 8$, $N_P = 4$, (b) $N_z = 9$, $N_P = 4$.

![Sample delay processing chain](image)

**Figure 4.** Sample delay processing chain described by delay$(i, \Delta t)$ and shift$(i, \Delta t)$ functions.

For each lane $i$, there is a set of FIFOs with the depth determined by the delay$(i, \Delta t)$ function. The outputs of FIFOs are then shifted according to the shift$(i, \Delta t)$ function, so that the delayed samples are used as neighbors in $(W, NW, N, NE)$ positions with respect to the samples which
are currently processed by each pipeline. The performed sample delay operation described by the use of the streaming grid (lanes, clock cycles) is presented in Figure 5. In the example, W neighbors \((s_1(1), s_0(1), s_8(0), s_7(0))\) of samples \((s_1(2), s_0(2), s_8(1), s_7(1))\) are obtained by delay and shift operations.

![Figure 5](image)

**Figure 5.** Sample delay operation for obtaining W neighbor samples for \(N_p = 4\) and \(N_c = 9\).

### 3.3. Local Differences

The computed local differences are stored in the central difference store since there is a need to share differences between the pipelines. The local difference vectors \(U_z\) for each pipeline are assembled as a combination of local differences from lower indexed pipelines and from the central difference store. The pipeline with the lowest index contains \(P\) differences only from the central difference store. An example of local differences routing between pipelines and to/from the central difference store for \(N_p = 4\) and \(P = 5\) is illustrated in Figure 6. Pipelines 0 – 3 produce local differences \(d_z(t)\) to \(d_{z+3}(t)\) for input samples \(s_z(t)\) to \(s_{z+3}(t)\), respectively. Since each pipeline requires \(P\) previous local differences, pipeline 3 requires differences \([d_{z+2}(t), d_{z+1}(t), d_z(t), d_{z-1}(t), d_{z-2}(t)]\) where the differences \([d_{z+2}(t), d_{z+1}(t), d_z(t)]\) are produced by pipelines \([2-0]\) in the current clock cycle and the other two elements \([d_{z-1}(t), d_{z-2}(t)]\) are fetched from the central difference store. After using \(P\) differences from central difference store to create \(U_z\) vectors, the differences from the bands in the range \([z-1), (z-(P \mod N_p)]\) are kept in the store to be used in the next clock cycle.

![Figure 6](image)

**Figure 6.** Routing of central differences between pipelines for \(N_p = 4\) and \(P = 5\).

When \(z < P\), it is required to use only \(z\) previous local differences and no local differences remaining from the previous pixel. In the serial implementation, the contents of the difference store is set to zero when \(z = N_c - 1\). For the parallel one, since previous local differences are used directly
from the pipelines, the differences are masked based on the \( z \) coordinate. In this manner, the local differences with index \( i \leq z \) are included in the local difference vector and elements with index \( i > z \) are set to zero.

### 3.4. Weights and Accumulators

Weights and accumulators are stored in two instances of the same module, shared store, with different element sizes of stored vectors. Figure 7 shows the shared store implementation with \( N_p \) block RAMs of depth \( M = \lceil N_z / N_p \rceil \). A read counter \( rd_{cnt} \) and a write counter \( wr_{cnt} \) are used for computation of the read and write addresses in each bank. The counters are initialized as \( rd_{cnt}(i) = 0 \) and \( wr_{cnt}(i) = delay(0, 1) \). The write counter is used directly as the write address \( w_{addr}(i) \), whereas the read address \( r_{addr}(i) \) for bank \( i \) is computed as follows:

\[
    r_{addr}(i) = \begin{cases} 
    rd_{cnt}, & i + N_z \mod N_p < N_p \\
    (rd_{cnt} - 1) \mod M, & i + N_z \mod N_p \geq N_p,
    \end{cases}
\]

creating the initial distance between the read and write addresses equal to \( delay(i, 1) \).

![Figure 7. Implementation of the shared store module.](image-url)

The behaviour of the weight shared store for parameters \( N_z = 61, M = 16 \) and \( N_p = 4 \) is presented in Figure 8. The initial state after reset in Figure 8a shows that \( rd_{cnt} \) is initialized to 0 and \( wr_{cnt} \) is set to \( delay(0, 1) = 15 \). For lanes 0 – 2, the read addresses are equal to the counter value \( (rd_{cnt} = 0) \), whereas for lane 3 the read address is 15 based on the condition \( (3 + 61 \mod 4) \geq 4 \). However, the data read from weight shared store for the first pixel are not used since the standard defines no prediction for the first pixel. Figure 8b shows the write operation of the first weight samples of pixel 1 at the address of the weight store pointed to by \( wr_{cnt} \). At this time stamp, counter \( rd_{cnt} \) is \( N \) positions from its initial position, where the delay \( N \) corresponds to several pipeline stages from the weight reading operation to the end of the weight update operation. The delay \( N \) is equal to \( 8 + S \), where parameter \( S \) is the number of pipeline stages in the dot product. In Figure 8c, the read counter is set to position \( M - 1 \), the \( r_{addr} \) are computed as \( [M - 2, M - 1, M - 1, M - 1] \) and the first weights \([-, W_0(1), W_1(1), W_2(1)]\) are read simultaneously with samples \([s_{59}(0), s_0(1), s_1(1), s_2(1)]\) at the input of the compression core. Figure 8d shows the state of weight shared store after 15 cycles when samples \([s_{59}(1), s_{60}(1), s_0(2), s_1(2)]\) arrive at the input.
3.5. Packing of Variable Length Words

The last stage includes packing of the variable-length encoded words \( W_0, \ldots, W_{N_p-1} \) with respective lengths \( L_0, \ldots, L_{N_p-1} \) from \( N_p \) pipelines into fixed-size blocks. The packing operation for \( N_p = 4 \) is illustrated in Figure 9. The packing process starts by shifting the word \( W_0 \) from the first pipeline by the number of bits from the previous cycle, \( L_{\text{prev}} \). After that, word \( W_0 \) is concatenated to the bits remaining from the previous cycle, \( W_{\text{prev}} \). In general, shifting of the word \( W_i \) by \( L_{\text{prev}} + L_0 + \cdots + L_{i-1} \) positions is followed by concatenation of \( W_i \) to the chain \( W_{\text{prev}}W_0 \ldots W_{i-1} \). It is observed that the number of shifts depends heavily on \( N_p \) and maximum length \( L_{\text{max}} + D \) of each word. On the other side, the standard defines the fixed-size output blocks of size \( B \) which is extracted each time the sum of the words’ lengths exceeds \( B \).

![Figure 9. Operation of the variable length word packer.](image)

The block extraction limits the maximum word chain length to \( B - 1 \) regardless of \( N_p \) or maximum word length. Therefore, the number of bits left after block extraction and the number of extracted blocks are introduced. The number of bits left after block extraction \( s_i \) is computed as follows:

\[
s_i = \Sigma L_i \mod B,
\]
where

\[ \Sigma L_i = L_{prev} + \sum_{j=0}^{i-1} L_j. \] (5)

The extraction count \( e_i \), indicating the number of blocks to extract, is defined as:

\[ e_i = \left\lfloor \frac{\Sigma L_i}{B} \right\rfloor. \] (6)

If \( e_i \) is non-zero, the number of accumulated bits \( \Sigma L_i \) is greater than \( B \).

The implementation of packer module is presented in Figure 10. In the first stage, computation of \( s_i \) and \( e_i \) parameters is performed for input word \( W_i \). In the second and third stage, a combiner chain combines input words using computed \( s_i \) and \( e_i \) as shown in Figure 11.
The shifting operation for each word is performed in parallel by using $s_i$ to select among shifted versions of $W_i$ from a multiplexer. The last pipeline stage concatenates shifted words and extracts full blocks based on extraction count $e_i$. The produced full blocks are added to the chain of complete blocks, the count of full blocks is updated and the remaining bits $W_{prev}$ are stored into a register to be combined in the next cycle. Finally, the last flag is set when the remaining bits are output as a separate block. After combiner chain, a chain of full blocks, its length and the last flag are pushed into an output FIFO. To output blocks sequentially, there is a need to buffer the blocks sent from the combiner. The data word width of the FIFO is determined by the maximum number of blocks $N_{max}$ produced in one clock cycle given as:

$$N_{max} = \left\lfloor \frac{B - 1 + N_p(U_{max} + D)}{B} \right\rfloor + 1, \quad (7)$$

where $(B - 1)$ is the maximum number of leftover bits from the previous cycle, $N_p(U_{max} + D)$ is the maximum word length produced in one clock cycle and factor 1 accounts for the last block when the last flag is set. If the average bit rate of the encoded samples is higher than the output bus width, there is risk for FIFO to become full. Thus, it is required to stall the data streaming into the core before the overflow occurs by de-asserting ready signal at the input. This is done by setting a threshold $N_{th}$ to the number of data words in the FIFO. In this manner, it is ensured that all encoded samples, which are streamed in from the cycle when de-assertion of ready signal happens, are stored. The threshold $N_{th}$ is equal to $S + 15$ which corresponds to the total number of pipeline stages from the core input to the FIFO. In the on-the-fly processing, stalling of the input stream is not possible and the choice of FIFO depth is dependant on the image statistics and speed of predictor’s ability to adapt.

The proposed serial packing of incoming words in combinatorial logic is feasible for $N_p < 6$. For larger $N_p$ values, the critical path in the initial pipeline stage does not meet timing requirements due to the dependence of sum of word lengths $L_i$ on $N_p$. For this reason, a modified version of the packer module is presented in Figure 12. The modified packer distributes the incoming words across several combiner chains operating in parallel. Large critical paths are then avoided by displacing each combiner chain by one clock cycle. The generic parameter $N_{per\_chain}$ is introduced to define the number of words per combiner chain where $1 \leq N_{per\_chain} \leq N_p$. The number of combiner chains $N_c$ is then computed as:

$$N_c = \left\lceil \frac{N_p}{N_{per\_chain}} \right\rceil. \quad (8)$$

Parameters $s_i$ and $e_i$ are computed sequentially for each combiner chain across $N_c$ clock cycles as shown in Figure 12.

Since this operation takes more than one clock cycle, $L_{prev}$ is not available when computation starts. Therefore, the partial sum of word lengths are initially computed as follows:

$$\Sigma L_i = \sum_{j=0}^{i-1} L_j, \quad (9)$$

whereas the complete length $\Sigma L_i = \Sigma L_j + L_{prev}$ is computed in the $N_c$-th clock cycle. Large critical paths can be created due to existing data dependence between combiner chains. To avoid this, large delay registers for the left-most chains are used to keep the full blocks from each chain synchronized with the last chain $N_c - 1$. The proposed solution is that each combiner chain shifts its input words by $L_{prev}$ without concatenating it with the remaining bits. Instead, the concatenation is done at the output of each combiner chain. The outputs of each combiner chain are a block set and a length of the produced block set which is sent to a block set FIFO. The output logic controls the streaming of created blocks and tracks which FIFO contains the packed blocks for that particular set of words. In particular, the control FIFO monitors which block set FIFOs contain valid data. For each block set pushed to the block FIFOs, a new word is pushed to the control FIFO with a block set mask and the last flag, where the bits in
block set mask correspond to one of the combiner chains. In Figure 13, block set mask ‘101.1’ for $N_c = 3$ indicates that valid block sets are from combiner chains 0 and 2 and last flag is high.

Figure 12. Implementation of improved variable length word packer.

Figure 13. Memory organization for block set FIFOs.
4. Results

The proposed parallel architecture of the CCSDS-123 compression algorithm is described by the VHDL language, and the Vivado tool is used for synthesis, implementation, power estimation, testing and verification on a PicoZed board with a Zynq-7035 FPGA. The implementation supports BIP sample ordering and both on-the-fly and offline processing. In addition, the implementation is tested against the reference software Emporda [21] and it is fully compliant with the standard allowing user-defined parameter selection.

The proposed core implementation is tested as a part of a larger system supported by AXI bus [22]. Since the internal stalling of output stream is not supported by the core, it is necessary to buffer the output data in a FIFO as shown in Figure 14. Data streaming into the core is stopped when the number of words in FIFO is larger than a certain limit. The FIFO capacity limit $N_{\text{limit}}$ is determined as follows:

$$N_{\text{limit}} = \text{FIFO capacity} - \left\lceil \frac{N_{\text{stages}}(U_{\text{max}} + D)}{B} \right\rceil,$$

based on the assumption that each pipeline stage has valid data and each data word has the maximum length of $U_{\text{max}} + D$. The depth of the FIFO is a trade-off between area usage and frequency of output stalling. It is, however, required the depth to be larger than $N_{\text{limit}}$.

![Figure 14. CCSDS-123 IP module.](image)

4.1. Utilization Results

The resource use is affected by several parameters such as the number of bands used for prediction $P$ and sample bit resolution $D$. As reported in [19], both LUT and register use in the dot product, predictor and weight update modules in the pipeline scale linearly with $P$. However, in the proposed implementation throughput is not affected by the choice of parameters and remains $N_p$ samples per clock cycle for any chosen parameter configuration.

The proposed implementation of the CCSDS-123 algorithm supports the majority of the standard’s parameter settings, including full ranges of bit resolution $D$, number of previous bands for prediction $P$ and output word size $B$. The implementation supports both neighbor- and column-oriented local sums, full and reduced prediction modes. However, only the sample adaptive encoder is supported. In the following resource use analysis for the proposed implementation, the chosen parameter configuration is set to the configuration provided in [10] with parameters $D = 16$ and $P = 3$. Table 1 shows resource use for the proposed implementation for a variety of different hyperspectral and multispectral image sensors. The main factor affecting the area use results is frame size $N_x \cdot N_z$ which determines the amount of memory required for storing delayed samples, weights and accumulators.

The used resources in terms of LUTs, registers and block RAM have been elaborated in more details for core configuration set for processing available 16-bit L1b HICO data cubes [5]. Initially, the number of block RAMs varied considerably for different number of pipelines. In particular, weight store and sample delay block RAM use varied depending on $N_p$. This happens due to the synthesis tool which extends depth of an array to the closest power of 2, and by that increases significantly used block RAM resources. To avoid this, LUT elements are used as Distributed RAMs instead of block RAMs. Since one LUT element in 7-series FPGAs [23] can be configured as a $32 \times 1$ bit
dual port RAM, LUTs are used as RAM for storing weights, accumulators and in the one-pixel delay FIFOs, whereas block RAMs are used in NE FIFO module. The LUT use increases then linearly with $N_x \cdot N_z$. Regarding register use, each lane has its own memory element with read data registers and the register use in these modules scales linearly with $N_P$. The resources in terms of LUTs, registers and block RAMs used for the total design and the main components are presented in Tables 2 and 3, respectively. The packer module is the largest contributor in logic use among shared modules due to the fact that as $N_P$ grows, the size of combiner chains for $N_P \leq 4$ and the number of combiner chains for $N_P > 4$ also increase. With the larger number of combiner chains, the number of block sets to select in the output logic also increases, requiring larger multiplexers for selection. Figure 15 shows that the resource use for a pipeline chain for $N_P \geq 4$ stabilizes at 72% of total resources, whereas the ratio of used and available resources for the complete core grows linearly with $N_P$ as presented in Figure 16. Thus, the choice of $N_P$ for the selected set of compression parameters and image size are constrained by available LUT resources.

**Table 1.** Resource use for compressing HSI images from different sensors for $N_P = 4$.

<table>
<thead>
<tr>
<th>Model</th>
<th>$D$</th>
<th>$N_x$</th>
<th>$N_y$</th>
<th>$N_z$</th>
<th>LUTs</th>
<th>Regs</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFSI</td>
<td>12</td>
<td>496</td>
<td>140</td>
<td>240</td>
<td>9416</td>
<td>8730</td>
<td>46</td>
</tr>
<tr>
<td>MSG</td>
<td>10</td>
<td>3712</td>
<td>3712</td>
<td>11</td>
<td>7984</td>
<td>8133</td>
<td>16</td>
</tr>
<tr>
<td>MODIS</td>
<td>12</td>
<td>1354</td>
<td>2030</td>
<td>17</td>
<td>8859</td>
<td>8682</td>
<td>12</td>
</tr>
<tr>
<td>M3-Target</td>
<td>12</td>
<td>640</td>
<td>2843</td>
<td>260</td>
<td>10,824</td>
<td>8827</td>
<td>64</td>
</tr>
<tr>
<td>M3-Global</td>
<td>12</td>
<td>320</td>
<td>28,283</td>
<td>386</td>
<td>11,351</td>
<td>9086</td>
<td>48</td>
</tr>
<tr>
<td>Landsat</td>
<td>8</td>
<td>1024</td>
<td>1024</td>
<td>8</td>
<td>6583</td>
<td>7410</td>
<td>7</td>
</tr>
<tr>
<td>Hyperion</td>
<td>12</td>
<td>256</td>
<td>3242</td>
<td>242</td>
<td>9640</td>
<td>8888</td>
<td>28</td>
</tr>
<tr>
<td>Crism-FRT</td>
<td>12</td>
<td>640</td>
<td>510</td>
<td>545</td>
<td>12,882</td>
<td>9313</td>
<td>130</td>
</tr>
<tr>
<td>Crism-HRL</td>
<td>12</td>
<td>320</td>
<td>480</td>
<td>545</td>
<td>12,646</td>
<td>9130</td>
<td>68</td>
</tr>
<tr>
<td>Crism-MSP</td>
<td>12</td>
<td>64</td>
<td>2700</td>
<td>74</td>
<td>8803</td>
<td>8843</td>
<td>6</td>
</tr>
<tr>
<td>CASI</td>
<td>12</td>
<td>405</td>
<td>2852</td>
<td>72</td>
<td>8922</td>
<td>8960</td>
<td>16</td>
</tr>
<tr>
<td>AVIRIS</td>
<td>16</td>
<td>614</td>
<td>512</td>
<td>224</td>
<td>12,033</td>
<td>10,696</td>
<td>71</td>
</tr>
<tr>
<td>AIRS</td>
<td>14</td>
<td>90</td>
<td>135</td>
<td>1501</td>
<td>12,191</td>
<td>8569</td>
<td>68</td>
</tr>
<tr>
<td>IASI</td>
<td>12</td>
<td>66</td>
<td>60</td>
<td>8461</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HICO</td>
<td>16</td>
<td>512</td>
<td>2000</td>
<td>128</td>
<td>11,589</td>
<td>10,661</td>
<td>35</td>
</tr>
</tbody>
</table>

**Table 2.** LUT use in various stages for different $N_P$.

<table>
<thead>
<tr>
<th>$N_P$</th>
<th>Pipeline</th>
<th>Sample Store</th>
<th>Accum Store</th>
<th>Weight Store</th>
<th>Packer</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2137</td>
<td>468</td>
<td>112</td>
<td>504</td>
<td>526</td>
<td>3747</td>
</tr>
<tr>
<td>2</td>
<td>4247</td>
<td>672</td>
<td>128</td>
<td>366</td>
<td>884</td>
<td>6297</td>
</tr>
<tr>
<td>3</td>
<td>6435</td>
<td>866</td>
<td>196</td>
<td>566</td>
<td>1139</td>
<td>9202</td>
</tr>
<tr>
<td>4</td>
<td>8499</td>
<td>856</td>
<td>180</td>
<td>366</td>
<td>1665</td>
<td>11,566</td>
</tr>
<tr>
<td>5</td>
<td>10,723</td>
<td>1029</td>
<td>230</td>
<td>464</td>
<td>2263</td>
<td>14,709</td>
</tr>
<tr>
<td>6</td>
<td>12,765</td>
<td>1226</td>
<td>272</td>
<td>555</td>
<td>2513</td>
<td>17,331</td>
</tr>
<tr>
<td>7</td>
<td>15,005</td>
<td>1458</td>
<td>317</td>
<td>647</td>
<td>2826</td>
<td>20,253</td>
</tr>
<tr>
<td>8</td>
<td>16,550</td>
<td>1802</td>
<td>350</td>
<td>731</td>
<td>3238</td>
<td>22,671</td>
</tr>
<tr>
<td>9</td>
<td>19,297</td>
<td>2042</td>
<td>397</td>
<td>815</td>
<td>4131</td>
<td>26,682</td>
</tr>
<tr>
<td>10</td>
<td>21,191</td>
<td>1886</td>
<td>440</td>
<td>923</td>
<td>4668</td>
<td>29,108</td>
</tr>
<tr>
<td>11</td>
<td>23,584</td>
<td>2186</td>
<td>416</td>
<td>1014</td>
<td>5008</td>
<td>32,208</td>
</tr>
<tr>
<td>12</td>
<td>25,136</td>
<td>2268</td>
<td>454</td>
<td>1112</td>
<td>5455</td>
<td>34,425</td>
</tr>
</tbody>
</table>
Table 3. Memory element use in various stages for different $N_p$.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1856</td>
<td>156</td>
<td>36</td>
<td>152</td>
<td>687</td>
<td>2887</td>
<td>32</td>
<td>1</td>
<td>33</td>
</tr>
<tr>
<td>2</td>
<td>3532</td>
<td>238</td>
<td>56</td>
<td>280</td>
<td>1069</td>
<td>5175</td>
<td>32</td>
<td>2</td>
<td>34</td>
</tr>
<tr>
<td>3</td>
<td>5394</td>
<td>351</td>
<td>78</td>
<td>410</td>
<td>1255</td>
<td>7488</td>
<td>33</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>6869</td>
<td>440</td>
<td>98</td>
<td>546</td>
<td>1636</td>
<td>9589</td>
<td>32</td>
<td>3</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>8921</td>
<td>540</td>
<td>120</td>
<td>670</td>
<td>2579</td>
<td>12,830</td>
<td>32.5</td>
<td>4.5</td>
<td>37</td>
</tr>
<tr>
<td>6</td>
<td>10,424</td>
<td>648</td>
<td>142</td>
<td>814</td>
<td>3033</td>
<td>15,061</td>
<td>33</td>
<td>5.5</td>
<td>38.5</td>
</tr>
<tr>
<td>7</td>
<td>12,460</td>
<td>756</td>
<td>164</td>
<td>951</td>
<td>3358</td>
<td>17,689</td>
<td>35</td>
<td>5.5</td>
<td>40.5</td>
</tr>
<tr>
<td>8</td>
<td>13,455</td>
<td>808</td>
<td>184</td>
<td>1085</td>
<td>3810</td>
<td>19,342</td>
<td>32</td>
<td>6.5</td>
<td>38.5</td>
</tr>
<tr>
<td>9</td>
<td>15,994</td>
<td>909</td>
<td>206</td>
<td>1209</td>
<td>4094</td>
<td>22,412</td>
<td>36</td>
<td>7.5</td>
<td>43.5</td>
</tr>
<tr>
<td>10</td>
<td>17,311</td>
<td>1000</td>
<td>228</td>
<td>1332</td>
<td>4507</td>
<td>24,378</td>
<td>35</td>
<td>8.5</td>
<td>43.5</td>
</tr>
<tr>
<td>11</td>
<td>19,546</td>
<td>1100</td>
<td>250</td>
<td>1476</td>
<td>4784</td>
<td>27,156</td>
<td>33</td>
<td>8.5</td>
<td>41.5</td>
</tr>
<tr>
<td>12</td>
<td>20,479</td>
<td>1200</td>
<td>272</td>
<td>1611</td>
<td>5189</td>
<td>28,751</td>
<td>36</td>
<td>9.5</td>
<td>45.5</td>
</tr>
</tbody>
</table>

Figure 15. Resource use by pipeline logic with respect to the available resources.

Figure 16. Resource use on Zynq Z-7035.

The LUT use in the packer module is analyzed in terms of several words per chain $N_{per\_chain}$ and block sizes $B$ and results are presented in Figure 17. It is observed that regardless of $N_{per\_chain}$, the block size is the main factor which greatly affects area use.
4.2. Timing

The maximum operating frequency of the proposed implementation for different \( N_p \) is shown in Figure 18. It is shown that the operating frequency depends on \( N_p \) with a downward trend and varies in the range 126–157 MHz. The critical path is in the output logic which produces the last flag signal obtained as a logical sum of the last signals from the control modules in each of the pipelines.

4.3. Power Estimation

The power estimation has been performed in Xilinx Vivado on the post implementation design in combination with data in post-implementation functional simulation. Figure 19 shows that power usage increases linearly with \( N_p \) in all modules for \( 1 \leq N_p \leq 8 \). Static power consumption of 0.125 W is mainly due to leakage in the memories in the stores, whereas dynamic power grows with respect to \( N_p \) as presented in Figure 20. The estimates for stores refer to the power sum in the weight, accumulator, sample and local difference stores. The linear increase is due to the added logic for each pipeline and to increasing complexity of the packer module. Fluctuations appear in the power contribution of the stores when \( N_p \) is power of 2 since inference of block RAMs in the NE FIFO of the sample delay module is the most effective when the depth of FIFO is a power of two. For example, for HICO data set the depth of FIFO, computed as \( N_z N_z / N_p \), is a power of two when \( N_p \) is also a power of two.
4.4. Comparison with State-of-the-Art Implementations

The comparison of the proposed parallel implementation of CCSDS-123 algorithm with recent sequential [12–15,17,18] and parallel [16] FPGA implementations with regards to maximum frequency, the throughput performance and power is presented in Table 4. The majority of implementations target Virtex-5 FX130T FPGA which is commercial equivalent of radiation hardened Virtex-5QV. However, the detailed power and performance analysis on parallel implementation [16] is reported for powerful Virtex-7 FPGA device. The sensor maximum data rates for AVIRIS NG and HICO imagers, representing real-time sensor throughput requirements, are also given. The implementations with BIP ordering have a roughly similar architecture but with large performance differences. In the implementation proposed by Santos et al. [13], the chosen sample ordering requires that local differences are recomputed when needed. As a consequence, each sample is read \(2(P + 1)\) times and the input bandwidth efficiency is decreased. This approach requires either the non-sequential memory access pattern with potentially reduction of streaming efficiency, or that the arrangement of the data samples in memory follows the irregular streaming order, occupying \(2(P + 1)\) as much storage. The implementation SHyLoC [17] supports all three sample orderings, where a different architecture is suggested for each ordering. The implementation by Bascones et al. [15] achieves throughput lower than 50 Msamples/s on Virtex-7 suggesting less than one sample compressed per clock cycle.

In the parallel implementation proposed by Bascones et al. [16], the throughput of 3510 Mb/s is reported for \(C = 7\) compression cores employed in parallel. By fixing the subset of bands processed by each CCSDS-123 core, throughput degradation can be introduced when the number of bands is not divisible by number of cores \(C\) since this requires stalling of several cores when processing the last samples of each pixel. Another limitation is the serial nature of the final packing stage which creates a
significant throughput bottleneck for large number of parallel cores. The paper suggests, however, that the serial packing circuit can be clocked faster.

The proposed parallel implementation builds up on the processing chain implemented in the previous work [19] which is characterized by a throughput of 2350 Mb/s. The CCSDS-123 processing chain adaptation for parallel processing and the structuring of several CCSDS-123 compression chains in parallel are introduced. The limitations of data routing between processing chains (CCSDS-123 cores) and packing operation in the work proposed by Bascones et al. [16] are successfully overcome in the proposed implementation. In fact, the throughput is maximized by the proposed interleaved data routing between parallel processing chains which eliminates pipeline stalling. In addition, the proposed parallel packing provides linear scaling of the throughput when the number of pipelines is increased. In fact, the ability to achieve high throughput for the number of spectral bands $N_z$ which is not an integer multiple of $N_p$, and to pack any number of variable length words into fixed-size words in each clock cycle are the greatest improvements of the proposed implementation. In comparison with the state of the art, the proposed parallel implementation achieves superior performance in terms of processing speed such as data rates of 9984 Mb/s and 12,000 Mb/s for $N_p = 4$ and $N_p = 5$, respectively.

### Table 4. Performance comparison of CCSDS-123 implementations.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Order</th>
<th>$P$</th>
<th>$D$</th>
<th>Platform</th>
<th>$f_{\text{max}}$ [MHz]</th>
<th>Throughput [Mba/s]</th>
<th>Power [mW]</th>
</tr>
</thead>
<tbody>
<tr>
<td>HICO [5,24]</td>
<td>-</td>
<td>-</td>
<td>14</td>
<td>Sensor Max.</td>
<td>-</td>
<td>4.78</td>
<td>66.92</td>
</tr>
<tr>
<td>Keymeulen et al. [12]</td>
<td>BIP</td>
<td>3</td>
<td>13</td>
<td>Virtex-5 (FX130T)</td>
<td>40</td>
<td>40</td>
<td>520</td>
</tr>
<tr>
<td>HyLoC, Santos et al. [13]</td>
<td>BSQ</td>
<td>3</td>
<td>16</td>
<td>Virtex-5 (FX130T)</td>
<td>134</td>
<td>11.2</td>
<td>179</td>
</tr>
<tr>
<td>Theodorou et al. [14]</td>
<td>BIP</td>
<td>3</td>
<td>16</td>
<td>Virtex-5 (FX130T)</td>
<td>110</td>
<td>110</td>
<td>1790</td>
</tr>
<tr>
<td>Bascones et al. [15]</td>
<td>BIP</td>
<td>0–15</td>
<td>16</td>
<td>Virtex-7</td>
<td>50</td>
<td>47.6</td>
<td>760</td>
</tr>
<tr>
<td>Bascones et al. [16]</td>
<td>BIP</td>
<td>0–15</td>
<td>16</td>
<td>Virtex-5 (FX130T)</td>
<td>-</td>
<td>179.7</td>
<td>3040</td>
</tr>
<tr>
<td>Bascones et al. [16]</td>
<td>BIP</td>
<td>0–15</td>
<td>16</td>
<td>Virtex-7</td>
<td>-</td>
<td>219.4</td>
<td>3510.4</td>
</tr>
<tr>
<td>SHyLoC, Santos et al. [17]</td>
<td>All</td>
<td>0–15</td>
<td>16</td>
<td>Virtex-5 (FX130T)</td>
<td>140</td>
<td>140</td>
<td>2240</td>
</tr>
<tr>
<td>Tsigkanos et al. [18]</td>
<td>BIP</td>
<td>3</td>
<td>16</td>
<td>Virtex-5 (FX130T)</td>
<td>213</td>
<td>213</td>
<td>3300</td>
</tr>
<tr>
<td>Fjeldtvedt et al. [19]</td>
<td>BIP</td>
<td>0–15</td>
<td>16</td>
<td>Zynq-7000</td>
<td>147</td>
<td>147</td>
<td>2350</td>
</tr>
<tr>
<td>Proposed work—$N_p = 4$</td>
<td>BIP</td>
<td>0–15</td>
<td>16</td>
<td>Zynq-7000</td>
<td>157</td>
<td>624</td>
<td>9984</td>
</tr>
<tr>
<td>Proposed work—$N_p = 5$</td>
<td>BIP</td>
<td>0–15</td>
<td>16</td>
<td>Zynq-7000</td>
<td>150</td>
<td>750</td>
<td>12,000</td>
</tr>
</tbody>
</table>

Future work will include an hardware implementation of emerging Issue 2 of CCSDS-123 standard [25,26] which builds up on the current version (Issue 1) of CCSDS-123 compression standard [10]. The Issue 2 focuses on new features such as a closed-loop scalar quantizer to provide near-lossless compression, modified hybrid entropy coder for low entropy data and support for high-dynamic-range instruments with 32-bit signed and unsigned integer samples. The introduced data dependencies can affect the throughput and challenge parallel processing.

### 5. Conclusions

In this paper, a parallel FPGA implementation of CCSDS-123 compression algorithm is proposed. The full use of the pipelines is achieved by the proposed advance routing with shifting and delay operations. In addition, the packing operation of variable-length words is performed fully in parallel, providing throughput of user-defined $N_p$ samples per clock cycle. This implementation significantly outperforms the state-of-the-art implementations in terms of throughput and power. The estimated power use scales linearly with the number of input samples. In conclusion, the proposed core can compress any number of samples in parallel provided that resource and I/O bandwidth constraints are obeyed.
Author Contributions: Conceptualization, M.O. and J.F.; methodology, J.F.; validation, J.F., M.O. and T.A.J.; investigation, J.F.; writing—original draft preparation, M.O.; writing—review and editing, M.O., J.F. and T.A.J.; visualization, M.O.; supervision, M.O. and T.A.J.; project administration, T.A.J.; funding acquisition, T.A.J.

Funding: This work was funded by the Research Council of Norway (RCN) through the MASSIVE project, grant number 270959, and the AMOS project, grant number 223254.

Conflicts of Interest: The authors declare no conflict of interest.

References


© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).