remote sensing

Article

A Parallel FPGA Implementation of the CCSDS-123 Compression Algorithm

Milica Orlandić [1]*, Johan Fjeldtvedt [1] and Tor Arne Johansen [2]

[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 intellegent 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 Ω which is defined as bit precision of the weight elements. The scaled predicted sample value $\tilde{s}_z(t)$ 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 $e_z = 2s_z(t) - \tilde{s}_z(t)$ 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 $\tilde{s}_z(t)$ is re-normalized to the range of the input sample (D-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 $\frac{\delta_z(t)}{2^{k_z(t)}}$, 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.

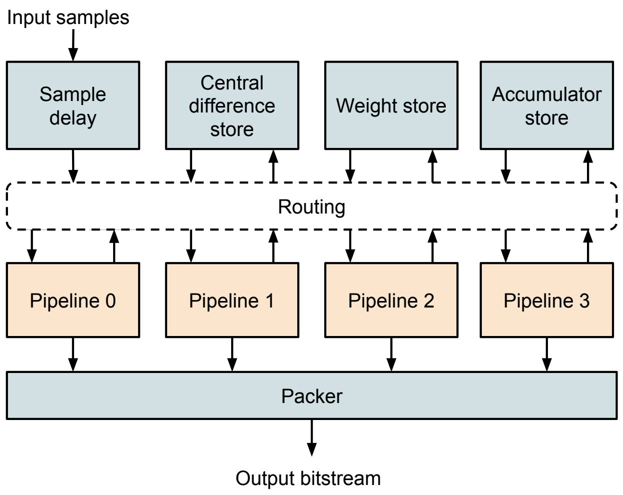


Figure 1. Overview of the parallel CCSDS-123 implementation for $N_p = 4$.

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.

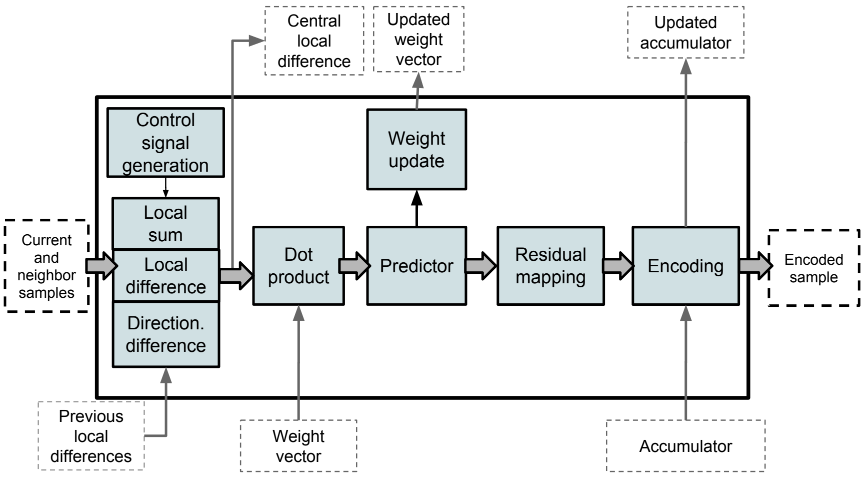


Figure 2. Overview of pipeline architecture.

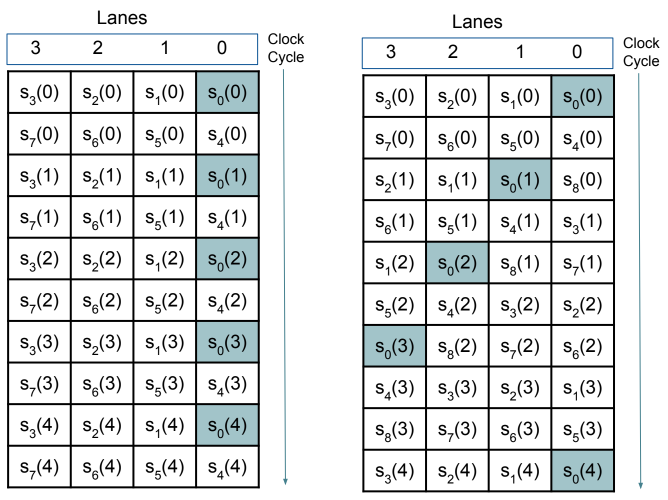
The data packages streamed into the CCSDS-123 core contain N_p samples. A lane is defined as a position of samples in the input package. Figure 3a,b show the sample placement grids for $N_p = 4$ lanes in the first 10 clock cycles for the number of bands is $N_z = 8$ and $N_z = 9$, respectively. The first 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:

$$\mathrm{shift}(i, \Delta t) = (i + N_z\Delta t) \mod N_p. \tag{1}$$

In Figure 3b, sample $s_0(5)$ is streamed in the lane which is computed as $\mathrm{shift}(3, 2) = 0$ 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:

$$delay(i, \Delta t) = \left\lfloor \frac{i + N_z\Delta t}{N_p} \right\rfloor. \tag{2}$$



(a)

3	2	1	0
$s_3(0)$	$s_2(0)$	$s_1(0)$	$s_0(0)$
$s_7(0)$	$s_6(0)$	$s_5(0)$	$s_4(0)$
$s_3(1)$	$s_2(1)$	$s_1(1)$	$s_0(1)$
$s_7(1)$	$s_6(1)$	$s_5(1)$	$s_4(1)$
$s_3(2)$	$s_2(2)$	$s_1(2)$	$s_0(2)$
$s_7(2)$	$s_6(2)$	$s_5(2)$	$s_4(2)$
$s_3(3)$	$s_2(3)$	$s_1(3)$	$s_0(3)$
$s_7(3)$	$s_6(3)$	$s_5(3)$	$s_4(3)$
$s_3(4)$	$s_2(4)$	$s_1(4)$	$s_0(4)$
$s_7(4)$	$s_6(4)$	$s_5(4)$	$s_4(4)$

(b)

3	2	1	0
$s_3(0)$	$s_2(0)$	$s_1(0)$	$s_0(0)$
$s_7(0)$	$s_6(0)$	$s_5(0)$	$s_4(0)$
$s_2(1)$	$s_1(1)$	$s_0(1)$	$s_8(0)$
$s_6(1)$	$s_5(1)$	$s_4(1)$	$s_3(1)$
$s_1(2)$	$s_0(2)$	$s_8(1)$	$s_7(1)$
$s_5(2)$	$s_4(2)$	$s_3(2)$	$s_2(2)$
$s_0(3)$	$s_8(2)$	$s_7(2)$	$s_6(2)$
$s_4(3)$	$s_3(3)$	$s_2(3)$	$s_1(3)$
$s_8(3)$	$s_7(3)$	$s_6(3)$	$s_5(3)$
$s_3(4)$	$s_2(4)$	$s_1(4)$	$s_0(4)$

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

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.

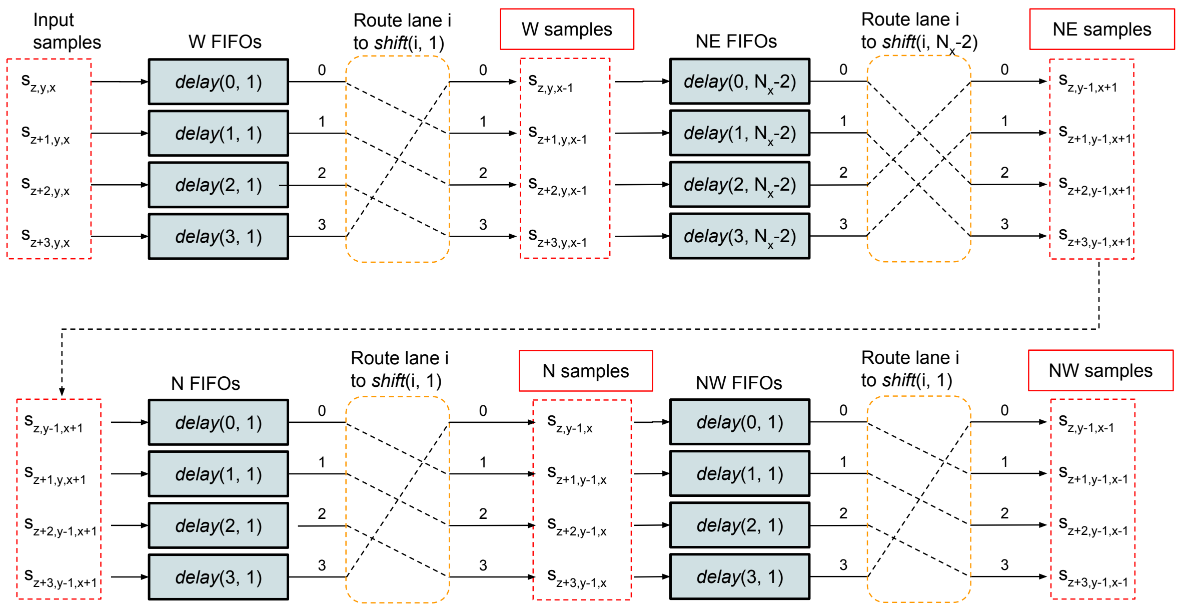


Figure 4. Sample delay processing chain described by delay$(i, \Delta t)$ and shift$(i, \Delta t)$ functions.

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.

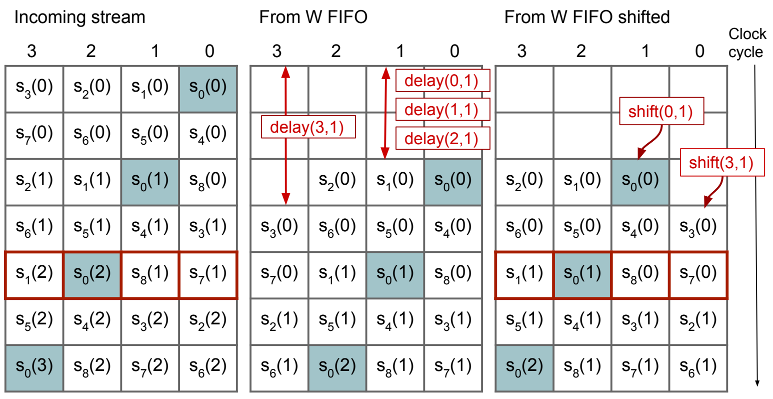
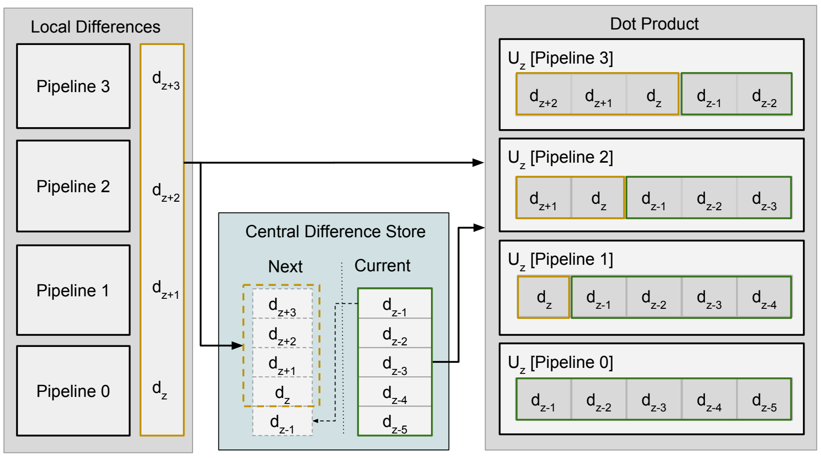


Figure 5. Sample delay operation for obtaining W neighbor samples for $N_p = 4$ and $N_z = 9$.

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 $[(z-1), (z-(P \bmod N_p))]$ are kept in the store to be used in the next clock cycle.



Local Differences
Pipeline 3
Pipeline 2
Pipeline 1
Pipeline 0
d_{z+3}
d_{z+2}
d_{z+1}
d_z
Central Difference Store
Next
Current
d_{z+3} d_{z+2} d_{z+1} d_z d_{z-1}
d_{z-1} d_{z-2} d_{z-3} d_{z-4} d_{z-5}
Dot Product
U_z [Pipeline 3]
d_{z+2} d_{z+1} d_z d_{z-1} d_{z-2}
U_z [Pipeline 2]
d_{z+1} d_z d_{z-1} d_{z-2} d_{z-3}
U_z [Pipeline 1]
d_z d_{z-1} d_{z-2} d_{z-3} d_{z-4}
U_z [Pipeline 0]
d_{z-1} d_{z-2} d_{z-3} d_{z-4} d_{z-5}

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 $z = N_z - 1$. 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 $wr\_cnt(i) = \text{delay}(0, 1)$. 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:

$$r\_addr(i) = \begin{cases} rd\_cnt, & i + N_z \mod N_p < N_p \\ (rd\_cnt - 1) \mod M, & i + N_z \mod N_p \geq N_p, \end{cases} \tag{3}$$

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

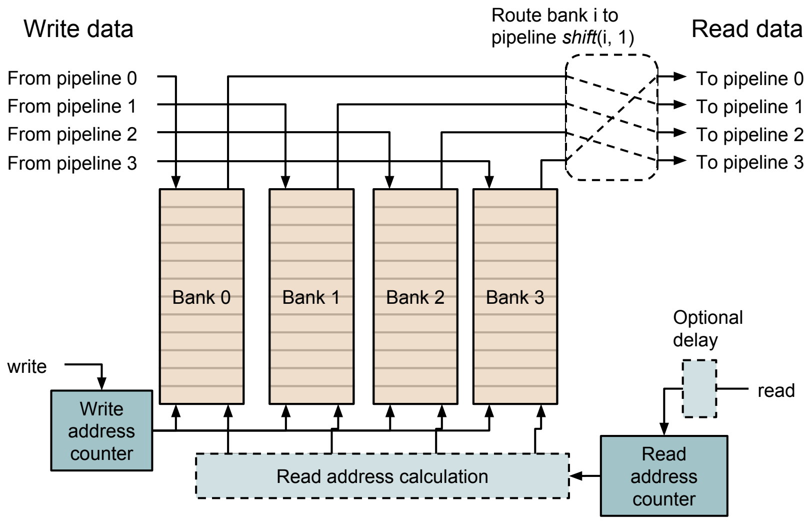


Figure 7. Implementation of the shared store module.

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 $[s_{60}(0), s_0(1), s_1(1), s_2(1)]$ 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.

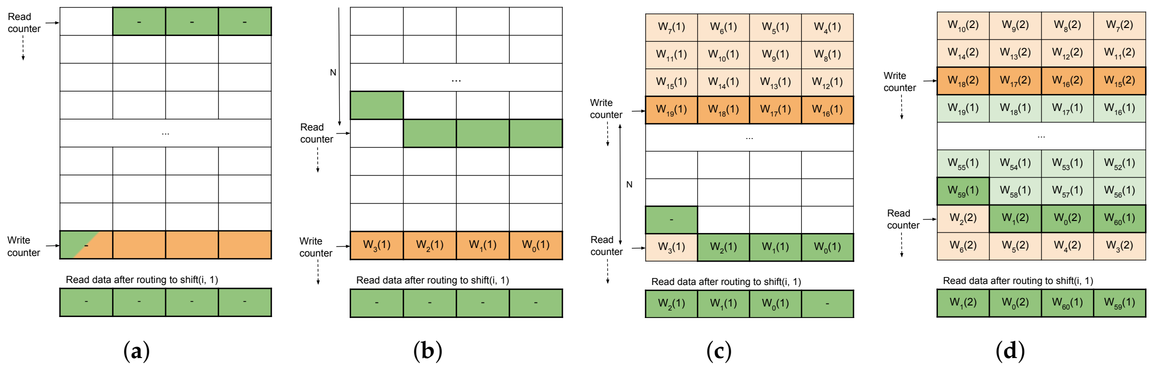

Read counter
Write counter
Read data after routing to shift(i, 1)
N
...
$W_3(1)$ $W_2(1)$ $W_1(1)$ $W_0(1)$
$W_7(1)$ $W_6(1)$ $W_5(1)$ $W_4(1)$
$W_{11}(1)$ $W_{10}(1)$ $W_9(1)$ $W_8(1)$
$W_{15}(1)$ $W_{14}(1)$ $W_{13}(1)$ $W_{12}(1)$
$W_{19}(1)$ $W_{18}(1)$ $W_{17}(1)$ $W_{16}(1)$
$W_2(1)$ $W_1(1)$ $W_0(1)$ -
$W_{10}(2)$ $W_9(2)$ $W_8(2)$ $W_7(2)$
$W_{14}(2)$ $W_{13}(2)$ $W_{12}(2)$ $W_{11}(2)$
$W_{18}(2)$ $W_{17}(2)$ $W_{16}(2)$ $W_{15}(2)$
$W_{19}(1)$ $W_{18}(1)$ $W_{17}(1)$ $W_{16}(1)$
$W_{55}(1)$ $W_{54}(1)$ $W_{53}(1)$ $W_{52}(1)$
$W_{59}(1)$ $W_{58}(1)$ $W_{57}(1)$ $W_{56}(1)$
$W_2(2)$ $W_1(2)$ $W_0(2)$ $W_{60}(1)$
$W_6(2)$ $W_5(2)$ $W_4(2)$ $W_3(2)$
$W_1(2)$ $W_0(2)$ $W_{60}(1)$ $W_{59}(1)$
(a)
(b)
(c)
(d)

Figure 8. States of weight shared store for $N_z = 61$ and $N_p = 4$ (**a**) after reset (**b**) during writing operation of the first weight samples for pixel 1 (**c**) during reading operation of the first stored weight samples for pixel 1 (**d**) after 15 cycles from the first reading operation for pixel 1.

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_{prev}. After that, word W_0 is concatenated to the bits remaining from the previous cycle, W_{prev}. In general, shifting of the word W_i by $L_{prev} + L_0 + \cdots + L_{i-1}$ positions is followed by concatenation of W_i to the chain $W_{prev}W_0 \ldots W_{i-1}$. It is observed that the number of shifts depends heavily on N_p and maximum length $U_{max} + D$ 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.

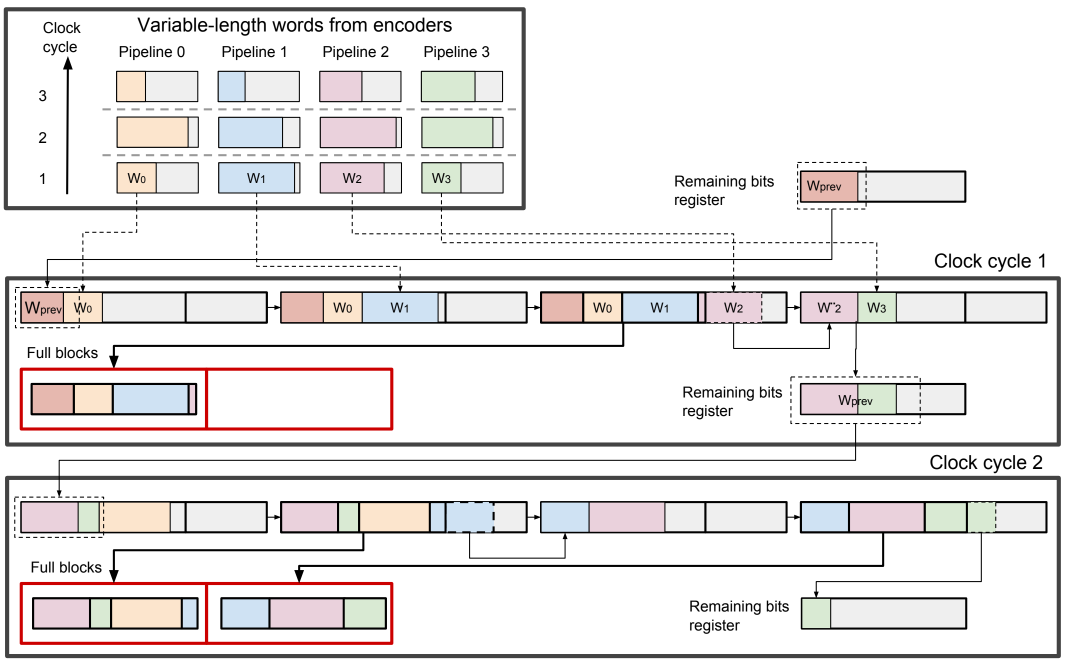


Figure 9. Operation of the variable length word packer.

The block extraction limits the maximum word chain length to $B - 1$ regardless of N_p or maximum world 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:

$$s_i = \Sigma L_i \mod B, \tag{4}$$

where

$$\Sigma L_i = L_{prev} + \sum_{j=0}^{i-1} L_j. \tag{5}$$

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

$$e_i = \left\lfloor \frac{\Sigma L_i}{B} \right\rfloor. \tag{6}$$

If e_i is non-zero, the number of accumulated bits $\sum L_i$ 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.

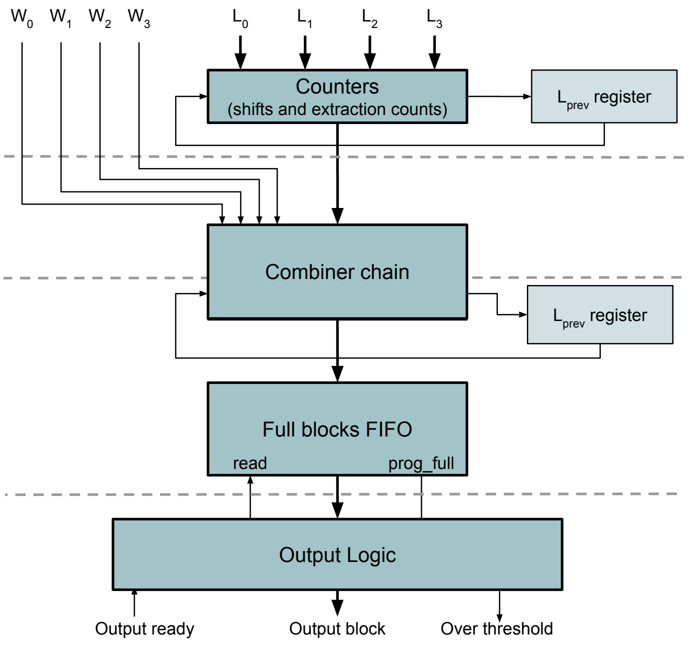


Figure 10. Implementation of variable length word packer.

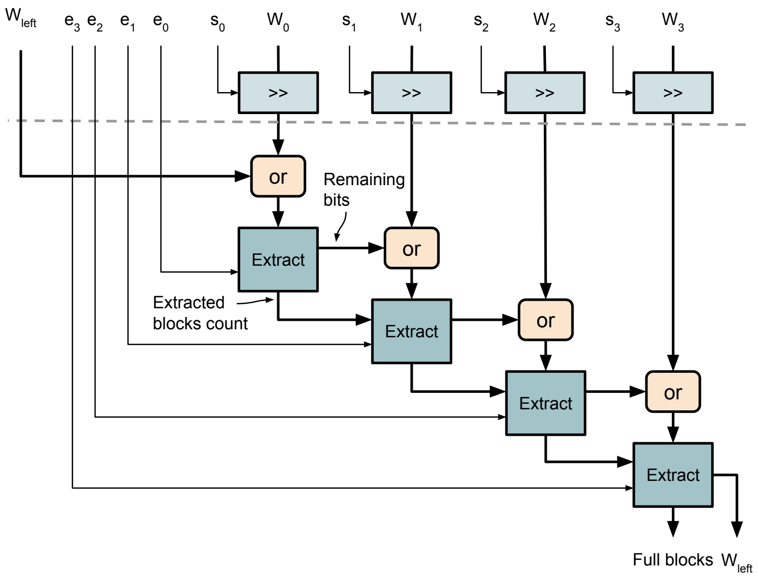


Figure 11. Implementation of combiner chain.

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:

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

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:

$$N_c = \lceil N_p / N_{per\_chain} \rceil. \tag{8}$$

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:

$$\Sigma\bar{L}_i = \sum_{j=0}^{i-1} L_j, \tag{9}$$

whereas the complete length $\Sigma L_i = \Sigma\bar{L}_i + L_{prev}$ 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.

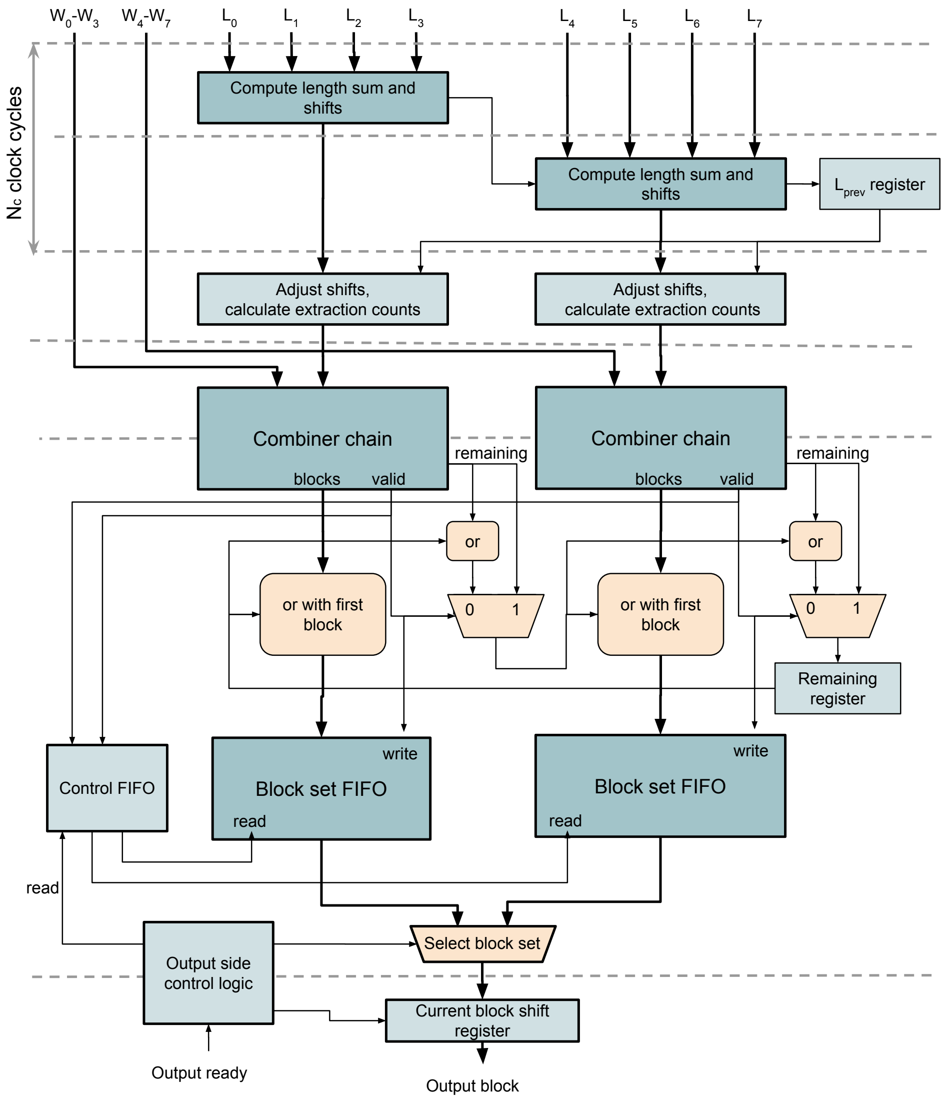


Figure 12. Implementation of improved variable length word packer.

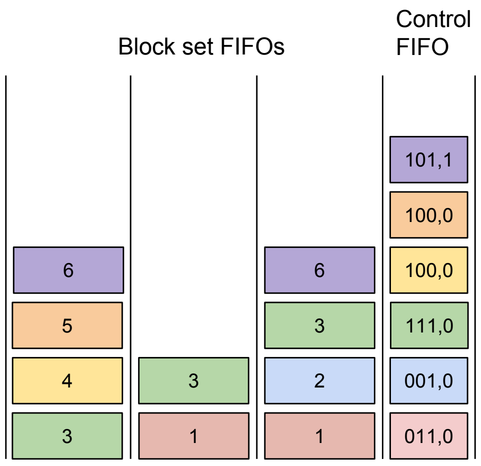


Figure 13. Memory organization for block set FIFOs.

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_{limit} is determined as follows:

$$N_{\text{limit}} = \text{FIFO capacity} - \left\lceil \frac{N_{\text{stages}}(U_{max} + D)}{\text{B}} \right\rceil, \tag{10}$$

based on the assumption that each pipeline stage has valid data and each data word has the maximum length of $U_{max} + D$. 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_{limit}.

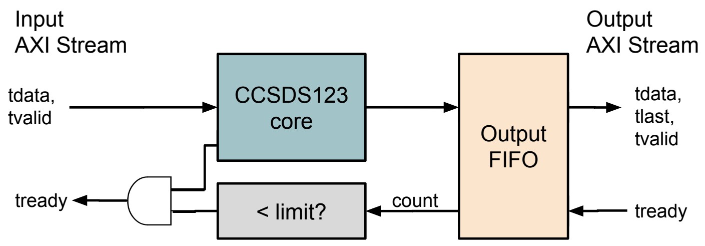


Figure 14. CCSDS-123 IP module.

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 32×1 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 $N_p = 4$.

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

Table 2. LUT use in various stages for different N_p.

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

Table 3. Memory element use in various stages for different N_p.

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

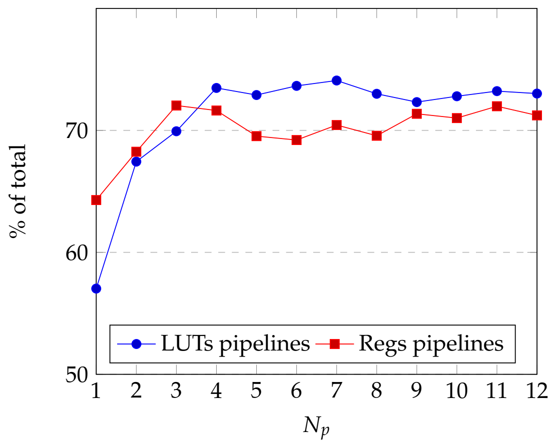


Figure 15. Resource use by pipeline logic with respect to the available resources.

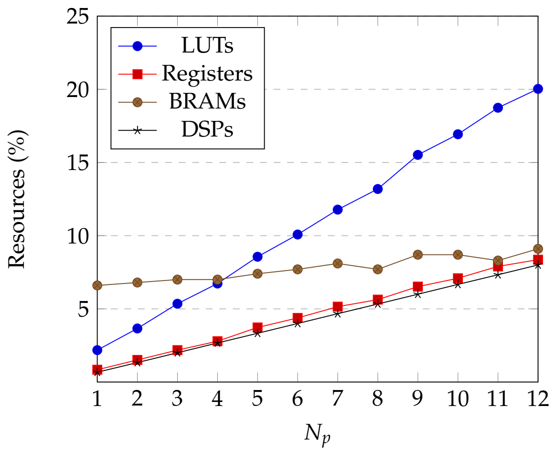


Figure 16. Resource use on Zynq Z-7035.

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.

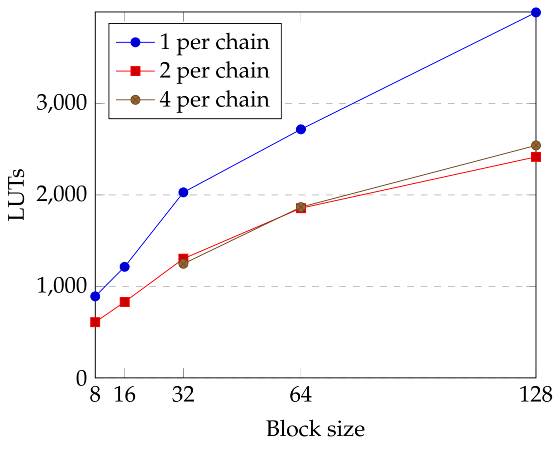


Figure 17. LUT use in packer module for various B and N_{per_chain}, $N_p = 4$.

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.

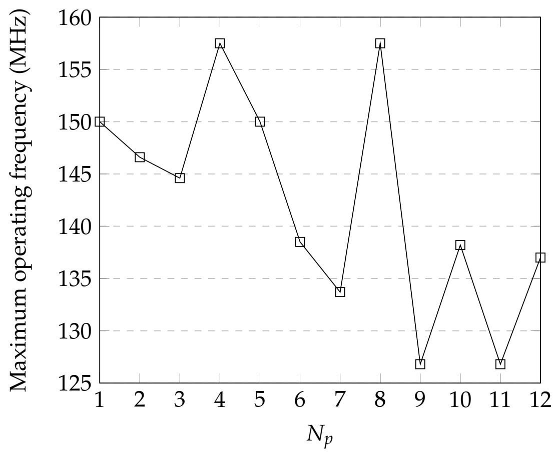


Figure 18. Maximum operating frequency for different number of pipelines.

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 $N_x N_z / N_p$, is a power of two when N_p is also a power of two.

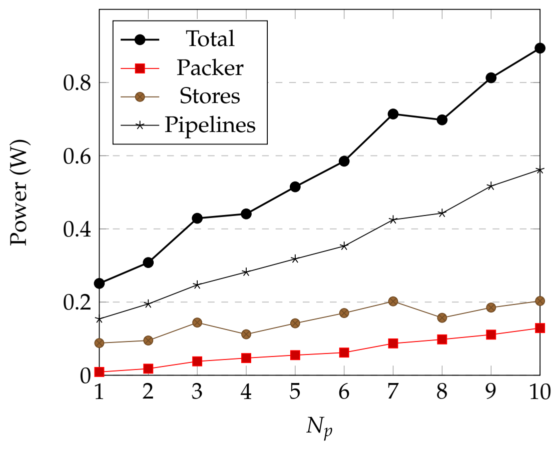


Figure 19. Power estimates for different N_p.

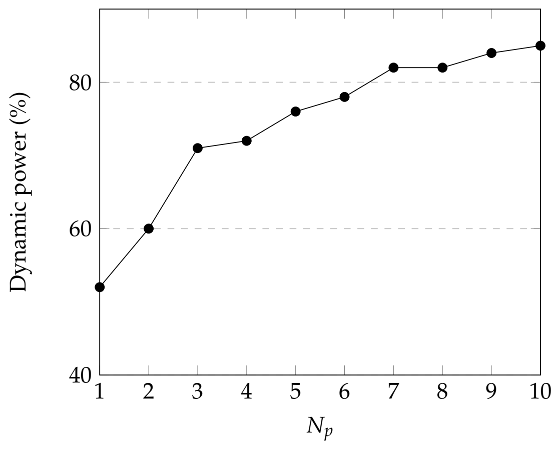


Figure 20. Dynamic power as percentage of total power usage.

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.

Implementation	**Order**	P	D	**Platform**	$f_{\max}$ **[MHz]**	**Throughput [MSa/s]**	**[Mb/s]**	**Power [mW]**
AVIRIS-NG [3]	-	-	14	Sensor Max.	-	30.72	430	-
HICO [5,24]	-	-	14	Sensor Max.	-	4.78	66.92	-
Keymeulen et al. [12]	BIP	3	13	Virtex-5 (FX130T)	40	40	520	-
HyLoC, Santos et al. [13]	BSQ	3	16	Virtex-5 (FX130T)	134	11.2	179	1488
Theodorou et al. [14]	BIP	3	16	Virtex-5 (FX130T)	110	110	1790	-
Bascones et al. [15]	BIP	0–15	16	Virtex-7	50	47.6	760	450
Bascones et al. [16]—$C = 7$	BIP	0–15	16	Virtex-5 (FX130T)	-	179.7		3040
Bascones et al. [16]—$C = 7$	BIP	0–15	16	Virtex-7	-	219.4	3510.4	5300
SHyLoC, Santos et al. [17]	All	0–15	16	Virtex-5 (FX130T)	140	140	2240	-
Tsigkanos et al. [18]	BIP	3	16	Virtex-5 (FX130T)	213	213	3300	4720
Fjeldtvedt et al. [19]	BIP	0–15	16	Zynq-7000	147	147	2350	295
Proposed work—$N_p = 4$	BIP	0–15	16	Zynq-7000	157	624	9984	440
Proposed work—$N_p = 5$	BIP	0–15	16	Zynq-7000	150	750	12,000	515

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

1. George, A.D.; Wilson, C.M. Onboard Processing With Hybrid and Reconfigurable Computing on Small Satellites. *Proc. IEEE* **2018**, *106*, 458–470. [CrossRef]
2. NASA. Moderate Resolution Imaging Spectroradiometer (MODIS). Available online: https://modis.gsfc.nasa.gov/ (accessed on 12 November 2018).
3. NASA. Airborne Visible InfraRed Imaging Spectrometer (AVIRIS). Available online: https://aviris.jpl.nasa.gov/ (accessed on 12 November 2018).
4. Aires, F.; Chédin, A.; Scott, N.A.; Rossow, W.B. A regularized neural net approach for retrieval of atmospheric and surface temperatures with the IASI instrument. *J. Appl. Meteorol.* **2002**, *41*, 144–159. [CrossRef]
5. Naval Research Laboratory. Hyperspectral Imager for the Coastal Ocean (HICO). Available online: http://hico.coas.oregonstate.edu/ (accessed on 12 November 2018).
6. Corson, M.R.; Korwan, D.R.; Lucke, R.L.; Snyder, W.A.; Davis, C.O. The hyperspectral imager for the coastal ocean (HICO) on the international space station. In Proceedings of the IGARSS 2008 IEEE International Geoscience and Remote Sensing Symposium, Boston, MA, USA, 7–11 July 2008; Volume 4.
7. Soukup, M.; Gailis, J.; Fantin, D.; Jochemsen, A.; Aas, C.; Baeck, P.; Benhadj, I.; Livens, S.; Delauré, B.; Menenti, M.; et al. HyperScout: Onboard Processing of Hyperspectral Imaging Data on a Nanosatellite. In Proceedings of the Small Satellites, System & Services Symposium (4S) Conference, Valletta, Malta, 30 May–3 June 2016.
8. Consultative Committee for Space Data Systems. Lossless Data Compression-CCSDS 121.0-B-2. In *Blue Book*; CCSDS Secretariat: Washington DC, USA, 2012.
9. Consultative Committee for Space Data Systems. Image Data Compression-CCSDS 122.0-B-1. In *Blue Book*; CCSDS Secretariat: Washington DC, USA, 2005.
10. Consultative Committee for Space Data Systems. Lossless Multispectral and Hyperspectral Image Compression-CCSDS 120.2-G-1. In *Green Book*; CCSDS Secretariat: Washington DC, USA, 2015.
11. Consultative Committee for Space Data Systems. Lossless Multispectral and Hyperspectral Image Compression-CCSDS 123.0-B-1. In *Blue Book*; CCSDS Secretariat: Washington DC, USA, 2012.
12. Keymeulen, D.; Aranki, N.; Bakhshi, A.; Luong, H.; Sarture, C.; Dolman, D. Airborne demonstration of FPGA implementation of Fast Lossless hyperspectral data compression system. In Proceedings of the 2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), Leicester, UK, 14–17 July 2014; pp. 278–284.
13. Santos, L.; Berrojo, L.; Moreno, J.; López, J.F.; Sarmiento, R. Multispectral and hyperspectral lossless compressor for space applications (HyLoC): A low-complexity FPGA implementation of the CCSDS 123 standard. *IEEE J. Sel. Top. Appl. Earth Observ. Remote Sens.* **2016**, *9*, 757–770. [CrossRef]
14. Theodorou, G.; Kranitis, N.; Tsigkanos, A.; Paschalis, A. High Performance CCSDS 123.0-B-1 Multispectral & Hyperspectral Image Compression Implementation on a Space-Grade SRAM FPGA. In Proceedings of the 5th International Workshop on On-Board Payload Data Compression, Frascati, Italy, 28–29 September 2016; pp. 28–29.
15. Báscones, D.; González, C.; Mozos, D. FPGA Implementation of the CCSDS 1.2.3 Standard for Real-Time Hyperspectral Lossless Compression. *IEEE J. Sel. Top. Appl. Earth Observ. Remote Sens.* **2017**, *11*, 1158–1165. [CrossRef]
16. Báscones, D.; González, C.; Mozos, D. Parallel Implementation of the CCSDS 1.2.3 Standard for Hyperspectral Lossless Compression. *Remote Sens.* **2017**, *9*, 973. [CrossRef]
17. University of Las Palmas de Gran Canaria, Institute for Applied Microelectronics (IUMA). SHyLoC IP Core. Available online: http://www.esa.int/Our_Activities/Space_Engineering_Technology/Microelectronics/SHyLoC_IP_Core (accessed on 12 November 2018).

18. Tsigkanos, A.; Kranitis, N.; Theodorou, G.A.; Paschalis, A. A 3.3 Gbps CCSDS 123.0-B-1 Multispectral & Hyperspectral Image Compression Hardware Accelerator on a Space-Grade SRAM FPGA. *IEEE Trans. Emerg. Top. Comput.* **2018**. [CrossRef]
19. Fjeldtvedt, J.; Orlandić, M.; Johansen, T.A. An Efficient Real-Time FPGA Implementation of the CCSDS-123 Compression Standard for Hyperspectral Images. *IEEE J. Sel. Top. Appl. Earth Observ. Remote Sens.* **2018**, *11*, 3841–3852. [CrossRef]
20. Augé, E.; Sánchez, J.E.; Kiely, A.B.; Blanes, I.; Serra-Sagristà, J. Performance impact of parameter tuning on the CCSDS-123 lossless multi-and hyperspectral image compression standard. *J. Appl. Remote Sens.* **2013**, *7*, 074594. [CrossRef]
21. GICI Group, Universitat Autonoma de Barcelona. Emporda Software. Available online: http://www.gici.uab.es (accessed on 12 November 2018).
22. ARM. *AMBA AXI and ACE Protocol Specification*; Technical Report; ARM, 2011. Avilable online: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0022d (accessed on 12 November 2018).
23. Xilinx. *7 Series FPGAs Configurable Logic Block User Guide*; Technical Report; Xilinx: San Jose, CA, USA, 2016.
24. Lewis, M.D.; Gould, R.; Arnone, R.; Lyon, P.; Martinolich, P.; Vaughan, R.; Lawson, A.; Scardino, T.; Hou, W.; Snyder, W.; et al. The Hyperspectral Imager for the Coastal Ocean (HICO): Sensor and data processing overview. In Proceedings of the OCEANS 2009, MTS/IEEE Biloxi-Marine Technology for Our Future: Global and Local Challenges, Biloxi, MS, USA, 26–29 October 2009; pp. 1–9.
25. Consultative Committee for Space Data Systems. Low-Complexity Lossless and Near-lossless Multispectral and Hyperspectral Image Compression-CCSDS 123.0-B-2. In *Blue Book*; CCSDS Secretariat: Washington DC, USA, 2019.
26. Kiely, A.; Klimesh, M.; Blanes, I.; Ligo, J.; Magli, E.; Aranki, N.; Burl, M.; Camarero, R.; Cheng, M.; Dolinar, S.; et al. The new CCSDS Standard for Low-Complexity Lossless and Near-Lossless Multispectral and Hyperspectral Image Compression. In Proceedings of the ESA On-Board Payload Data Compression Workshop (OBPDC), Matera, Italy, 20–21 September 2018.