DOI: ADD DOINUMBER HERE

Implementation of CCSDS Hyperspectral Image Compression Algorithm on FPGA on board a nanosatellite

Neelanchal Joshi[⋆†] *and Parth Kalgaonkar*[⋆†]
[⋆]*Team Anant, BITS Pilani*
Address
neelanchaljoshi@gmail.com · parthkalgaonkar@gmail.com
[†]Corresponding author

Abstract

Team Anant is building a nanosatellite whose objective is to perform hyperspectral imaging of the Indian Ocean. Due to low bandwidth availability and large image size, the image must be compressed by the On-Board Computer. The CCSDS 123.0-B-1 algorithm can be broken down into a predictor and an encoder. This paper demonstrates a pipelined implementation of the algorithm on a Xilinx 7-Series Field Programmable Gate Array. This paper also discusses the ways the compression hardware can be interfaced to the Processor for control by Software and elaborates the different entropy encoders following the prediction stage.

1. Introduction

Team Anant is a group of passionate undergraduate students from BITS Pilani, whose objective is to build a 3U CubeSat using commercially available off-the-shelf components. Founded in 2013 by three undergraduate students, the team now consists of forty students from various disciplines. The CubeSat houses a hyperspectral imager as its primary payload, for the categorization of various phytoplankton in the ocean. The bandwidth provided for such satellites for communication is very low. Hyperspectral cameras pose two challenges, the first being large image cube size and the second being the power consumed by it. The former is a big concern because of the limited downlinking capabilities possessed by a nanosatellite, and to counter this, we implement a compression algorithm on board. To speed up the compression process and minimize the energy spent (higher power but smaller duration), computationally intensive compression algorithm are implemented using FPGAs. To effectively downlink the image, the recommended standard was chosen, called the CCSDS 123.0.B.1. A Xilinx 7 Series Field Programmable Gate Array (FPGA) is being used to implement the compression algorithm.[2]

1.1 System Description

The team is divided into six subsystems: 1. On-Board Computer, 2. Electrical Power, 3. Attitude Determination and Control, 4. Structural and Thermal, 5. Telemetry, Tracking and Command and 6. Payload. Such a division of labour was created to ensure efficiency and accuracy while developing the components for the satellite.

The On-Board Computer of the satellite acts as the primary source of all the commands and monitors the status of the various subsystems. The OBC architecture also performs compression of the hyperspectral image before the data get downlinked. Our satellite follows an amalgamation of centralized and distributed architecture. On-Board computer subsystem makes use of a Zynq-7000 SoC which comprises of an ARM-cortex A9 based processing system (PS) and Kintex-7 based programmable logic (PL).

1.2 Functions of the On-Board Computer

The primary functions of the OBC are as follows:

1.Acquiring housekeeping data.
2.Storage and downlinking of payload and housekeeping data.
3.Executing attitude determination and control algorithms.

4.Determination of the mode of operation and flow of control.
5.Processing of the Telecommand Data.
6.Acquisition of the image and execution of the compression algorithm.

1.3 Software and Hardware Specifications

Petalinux, a Linux based operating system has been chosen for this satellite because of ease of development coupled with the availability of many client drivers, bus controller drivers and fairly good documentation even for kernel level programming. One of the most important reasons for using Petalinux was the availability of documentation and its popularity with the Xilinx community. For software development, C is chosen as it is by far the most powerful and versatile language that provides just the right amount of abstraction for development of highly specific applications with constraints on running time and determinism. The implementation of the compression algorithm on the FPGA is done using the hardware description language, Verilog. The simulations and packaging of the IPs are done with the help of the software, Vivado. The development and synthesis of the algorithm were done on the ZedBoard, a development board by Digilent.

2. Overview

In order to increase the downlink data rate, computationally intensive compression algorithms need to be implemented on-board in a power efficient manner. Since the hyperspectral image is needed in its truest form for in-depth analysis, the compression algorithm cannot be lossy. Hence, there were three challenges- heavy processing and computation requirements, power constraints and image quality retention. In order to counter the first two, the use of field-programmable gate arrays (FPGAs) to perform image processing is suggested, to enable hardware based parallel processing. FPGAs are a matrix of many configurable logic blocks (CLBs) that allow parallel algorithms to be implemented on them efficiently.

Optimized datapaths in FPGAs allow improved performance of computationally intensive tasks and reduced power and energy consumption as compared to general purpose processors (GPPs). In comparison with GPPs, FPGAs have been observed to improve performance by more than 15 times at 50% of the energy consumption. FPGAs are more versatile and configurable when compared to Application Specific Integrated Circuits (ASIC).

Typically, hyperspectral sensors image data over a few 100 spectral ranges. This results in each image actually being closely spaced layers of individual images, best imagined as a cube as shown in the picture below. In fact hyperspectral images are often called hypercubes.

Figure 1: Graphic Representation of Hyperspectral Data[3]

In modern day imagers, resolutions are quite high[4] and hence many pixels cover smaller areas, leading them to have largely similar coloring. Image compression often tries to take advantage of this correlation. In fact, a higher spatial resolution often implies better compression ratios.

The problem of compressing hyperspectral images is usually solved by two methods- by applying 2-D compression algorithms taking advantage of spatial redundancy between neighboring pixels, or by taking advantage of the spectral redundancy of the same pixel across a certain number of spectra as well as spatial redundancy within the spectrum.

The recent standard recommended by the CCSDS (Consultative Committee for Space Data Systems) for multispectral and hyperspectral image compression i.e. CCSDS 123.0-B-1 has been chosen for implementation. This standard was built keeping in mind the power and memory constraints of small satellites and providing lossless compression.

It has two primary components - a predictor and an entropy encoder. The predictor uses linear prediction techniques, and using values from previous spectra and neighboring pixels predicts the value of the current pixel. The standard advises the use of static codes like Golomb and Rice for entropy encoding.

3. Implementation of the algorithm

The implementation of the algorithm is divided into two parts:

1. Datapath
2. Control Path
3. Interfacing

The following sections describe how each component is oraganised.

4. Datapath

The datapath consists of various hardware IPs fabricated on the FPGA. The schematic of the datapath is shown below.

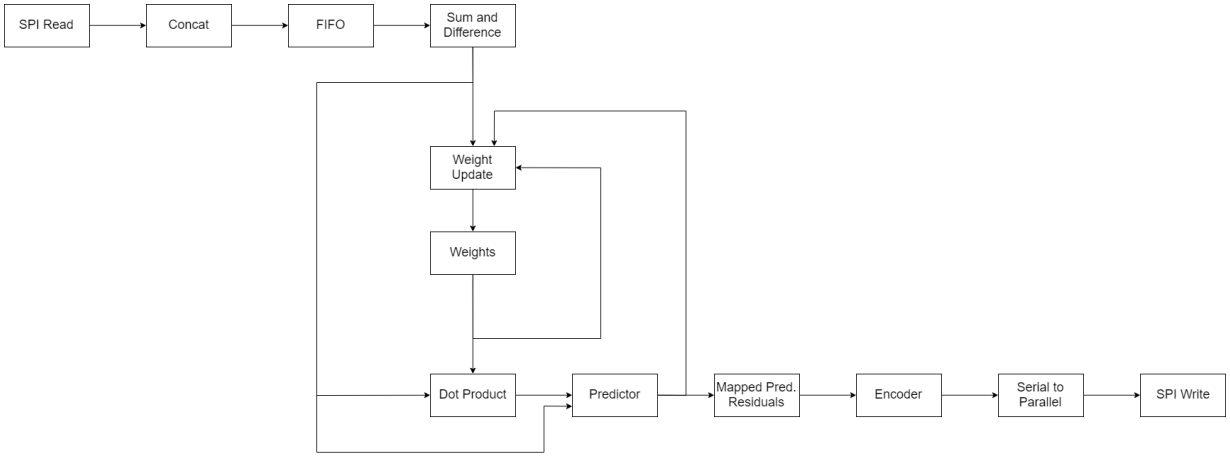

Figure 2: CCSDS Datapath

4.1 Concat

The payload of the satellite is the Ximea hyperspectral imager. The size of each pixel of the output image is 10 bits. But the memory is byte-organized (8 bits). The camera is interfaced to the Processing System via a USB 3.0 interface.

This IP concatenates the bits from two different memory locations to complete the 10 bits of a pixel. The 10-bit output of this IP is sent to the next IP, FIFO.

4.2 First In First Out (FIFO) IP

To process the compression of any pixel, the pixels near the current pixel are required (See 4.3). As the name suggests, this IP makes a queue of the output data from the previous IP. At a particular instant, the FIFO contains the data of all the pixels from the current band as well as P previous bands. Existing research shows that the values of $P > 3$ do not affect the compression ratio considerably. Therefore, we have chosen $P = 3$ in our system.

Assume that pixels are read-in in a Band Sequential Order, if the pixels till P previous bands are stored, each consecutive pixel just needs that pixel to be read in and the Last pixel in the Pipeline can be removed as that would no longer be required. That is why memory on the Fabric has been organised as a First In - First Out Queue.

The size of this FIFO would have to be equal to $N_x * N_y * P + 1$ to be able to store all the required data. For any reasonable system[4] this FIFO would be too large to be implemented as a completely Distributed RAM resource on an FPGA. The resources required for routing, address decoding, and multiplexing are too high for most commercially available FPGAs. This necessitates the need to use the Block RAM resources available. Our test system is a Xilinx Zynq 7000 series SOC that has 140 36Kb Block RAMs.[115] But Block RAM resources have the constraint of having not more than 2 read ports. As such the FIFO design can be done in two ways-

1. The FIFO can be made multi-cycle. All the read/write operations can be performed in such a case. The obvious drawback is the delay caused by such a multi-cycle approach. For every pixel to be compressed, $5 * P$ reads would have to be done from the FIFO. In this case the whole advantage of using a FIFO is lost and we are simply left with an On-Chip Cache for the image rather than having to read from a peripheral memory. The power constraints in nano-satellites allow only a small window for image compression when the power available is optimum. The delay leads to higher compression times, which in turn lead to higher power consumption.

2. The FIFO can be divided into multiple smaller FIFOs. For example, one FIFO could be used for each direction (NW, N, NE, W, Center). This could reduce latency to P reads per cycle. However the cost would be in terms of FPGA area used. In case of Full prediction, this would require 5 times more resources on the chip as compared to the optimum.

This is a big challenge in implementing the algorithm for a nano-satellite environment. This paper proposes a Mixed RAM approach to build the On-Chip Memory.

By observation, the pixels that are actually required for compression are only the ones directly below and surrounding the current pixel. The rest are stored simply to reduce the number of peripheral accesses required. As such the pixels that are required can be stored in really small pieces of Distributed RAM (Not more than 3 Pixel wide). These small slices can be interleaved with Large BRAM based FIFOs to create a FIFO of any required size. Such a FIFO would not have any read latency and would have very low overhead in terms of resources required for implementation. As a trade-off, such FIFO would be difficult to implement if BRAM based FIFOs are not already available and the design scales poorly. Additionally, to change the resolution of images, the designer would have to modify the configuration completely and this process would be very inefficient. But most missions have a fixed spatial and spectral resolution of images and can thus do with a fixed implementation.

The output of this IP is a **pixel vector** containing the pixels that surround the current pixel of compression in both spatial and spectral dimension. This **pixel vector** would be continuously updated as the algorithm progresses through the image. In boundary cases, some of the elements in this vector would be undefined.

4.3 Sum and Difference IP

This IP calculates the local sum and differences for the current pixel. The FIFO outputs a pixel vector containing the pixels shown in the figure below. The logic to compute the local sums and differences can be purely combinational. The symbols carry the usual meaning.[2]

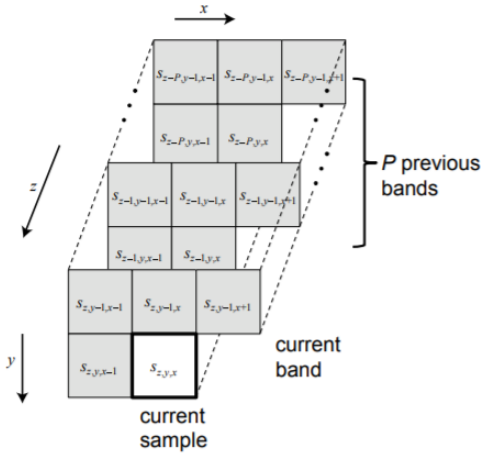

Figure 3: Prediction Neighbourhood[2]

The local sum for a pixel is given by:

$$\sigma_{z,y,x} = \begin{cases} s_{z,y,x-1} + s_{z,y-1,x-1} + s_{z,y-1,x} + s_{z,y-1,x+1}, & \text{if } y > 0, 0 < x < N_x - 1 \\ 4s_{z,y,x-1}, & \text{if } y = 0, x > 0 \\ 2\Big(s_{z,y-1,x} + s_{z,y-1,x+1}\Big), & \text{if } y > 0, x = 0 \\ s_{z,y,x-1} + s_{z,y-1,x-1} + 2s_{z,y-1,x}, & \text{if } y > 0, x = N_x - 1 \end{cases} \tag{1}$$

The central local difference of a band is given by:

$$d_{z,y,x} = 4s_{z,y,x} - \sigma_{z,y,x} \tag{2}$$

The directional local differences are given by:

$$d^N_{z,y,x} = \begin{cases} 4s_{z,y-1,x} - \sigma_{z,y,x}, & \text{if } y > 0 \\ 0, & \text{if } y = 0 \end{cases} \tag{3}$$

$$d^W_{z,y,x} = \begin{cases} 4s_{z,y,x-1} - \sigma_{z,y,x}, & \text{if } x > 0, y > 0 \\ 4s_{z,y-1,x} - \sigma_{z,y,x}, & \text{if } x = 0, y > 0 \\ 0, & \text{if } y = 0 \end{cases} \tag{4}$$

$$d^{NW}_{z,y,x} = \begin{cases} 4s_{z,y-1,x-1} - \sigma_{z,y,x}, & \text{if } \ x > 0, y > 0 \\ 4s_{z,y-1,x} - \sigma_{z,y,x},, & \text{if } x = 0, y > 0 \\ 0, & \text{if } y = 0 \end{cases} \tag{5}$$

The output of this IP is the local sum and a difference vector containing the directional local differences and central local differences from P previous bands.

4.4 Weights

The result of prediction is a function of a weighted sum of all local differences, directional as well as central (central only in case of reduced prediction mode). These weights are updated for every pixel. All the symbols carry usual meaning.

$$W_z(t+1) = clip(W_z(t) * \lfloor \frac{1}{2}(sgn^+(e_z(t))\dot{2}^{-\rho(t)}\dot{U}_z(t) + 1) \rfloor, (\omega_{min}, \omega_{max}) \tag{6}$$

$\rho(t)$ is the weight update scaling exponent. Larger values of $\rho(t)$ produce smaller increments resulting in slower adaptation to image data statistics but better steady state performance.

The weights are stored as in a slice of Distributed RAM. The weight update scaling exponent is implemented as a counter that increments at fixed intervals.

4.5 Predictor IP

The scaled predicted sample value of the pixel is given by:[8]

$$\tilde{s_z}(t) = \begin{cases} clip\Bigg(\Big\lfloor \Big[\hat{d_z}(t) + 2^{\Omega} + 2s_{mid} + 1\Big] \Big\rfloor\Bigg) & \text{if } t > 0 \\ 2s_{z-1}(t) & \text{if } t = 0, P > 0, z > 0 \\ 2s_{mid}(t) & \text{if } t = 0 \text{ and } (P = 0 \text{ or } z = 0) \end{cases} \tag{7}$$

In the above calculation:
For $t > 0$, the predicted central local difference $\hat{d_z}(t)$ is the inner product of the difference vector and the weight vector. Thus the predicted sample value is

$$\hat{s_z}(t) = \left\lfloor \frac{\tilde{s_z}(t)}{2} \right\rfloor \tag{8}$$

Most multiplications throughout the algorithm are with powers of 2. As such, they can be implemented using simple shifting hardware. However the vector multiplication here, between weight vector and difference vector necessitates the use of dedicated multiplication hardware. Combinational multipliers would be too hardware intensive while

Booth Multipliers introduce undesirable delay. To tackle this, DSP slices available in the FPGA were used to generate the scalar product. The available DSP resources on the Zedboard support 18x25 bit multiplication.[5] Thus these can be used to compute the scalar products. Local differences would be 13 bits wide for 10 bit unsigned samples and the weight resolution can be adjusted to be the 7 to 22 bits wide signed samples.[2]

4.6 Mapped Prediction Residual IP

This IP calculates the mapped prediction residuals based on the scaled predicted sample values. Since the entropy encoder requires non-negative values, we need to define another quantity.

$$\delta_z(t) = \begin{cases} |\Delta_z(t)| + \theta_z(t) & |\Delta_z(t)| > \theta_z(t) \\ 2|\Delta_z(t)| & 0 \leq (-1)^{\bar{s}_z(t)}|\Delta_z(t)| \leq \theta_z(t) \\ 2|\Delta_z(t)| - 1 & otherwise \end{cases} \tag{9}$$

where

$\delta_z(t)$**:** Mapped prediction residual. This value is now encoded to form the compressed image.

$\Delta_z(t)$**:** Difference between predicted and actual sample values.

$\theta_z(t)$**:** Minimum of $\hat{s}_z(t) - s_{min}$ and $s_{max} - \hat{s}_z(t)$

The prediction process ensures that the number of times a value occurs is decreased as most of the residuals are clumped at the centre of the double-sided Gaussian curve.

4.7 Encoder IP

Entropy encoding or simply encoding encompasses various methods, preferably in various situations, all of which are used in an attempt to try to minimize the average length of the overall source This section specifies the encoding stage of the compressor and the format of a compressed image. A compressed image consists of a header followed by a body. The variable-length header, encodes image and compression parameters. The body, consists of encoded mapped prediction residuals from the predictor. The mapped prediction residuals are sequentially encoded in the order selected by the user and indicated in the header. This encoding order need not correspond to the order in which samples are output from the imaging instrument or processed by the predictor. To encode the mapped prediction residuals for an image, a user may choose to use the sample adaptive entropy coding approach or the block-adaptive approach. Our system uses the sample adaptive approach for accuracy and better compression ratio. The sample-adaptive entropy coder typically yields smaller compressed images than the block adaptive entropy coder.[10]

Various encoding schemes can be used. Given below are some encoding schemes that can be used for the algorithm:

4.7.1 Unary Encoding

This is a very simple technique and results in a prefix and self-synchronizing code which provides ease of decodability and error protection. The following are the codes for say, 10 symbols:

Table 1: Unary Code Example

Digit	**Unary Code**
0	0
1	01
2	001
3	0001
4	00001
5	000001
6	0000001
7	00000001
8	000000001
9	0000000001

4.7.2 Truncated Binary Encoding

This encoding technique also results in prefix codes. Let us assume that n symbols need to be encoded and $2k \leq n < 2k+1$, let $u = 2k + 1 - n$. Let, the codes written using i bits be written in ascending order be called the $list_i$. Now, the first u symbols are written using the first u symbols of the $list_k$ and the last $n - u$ symbols are written using the last $n - u$ symbols of $list - k + 1$. This explanation will become clearer, with the example below:
Let the symbols be 0, 1, 2, 3, 4, 5. Hence, $n = 6$, $2.2 \leq 6 < 2.3 + 1 \Rightarrow k = 2$ and $u = 2$.

Table 2: Lists for Truncated Binary Code

List 2	**List 3**
00	000
01	001
10	010
11	011
	100
	101
	110
	111

Hence, we encode as:

Table 3: Truncated Binary Code Example

List 2	**List 3**
0	00
1	01
2	100
3	101
4	110

4.7.3 Golomb Rice Encoding

Mapped prediction residuals are encoded using either the sample-adaptive entropy coding approach. Under the sample-adaptive entropy coding option, each mapped prediction residual $\delta_z(t)$ shall be encoded using a variable-length binary codeword. The selection of the code used to encode $\delta_z(t)$ is based on the values of the adaptive code selection statistics specified below.

Adaptive Code selection statistics: The adaptive code selection statistics consist of an accumulator $\Sigma_z(t)$ and a counter $\Gamma(t)$ that are adaptively updated during the encoding process. The initial values for $\Gamma(t)$ and $\Sigma_z(t)$ are as follows:

$$\Gamma(1) = 2^{\gamma_0} \tag{10}$$

where γ_0 is the initial count exponent ($1 \leq \gamma_0 \leq 8$) and

$$\Sigma_z(1) = \left\lfloor \frac{1}{2^7}(3.2^{K+6} - 49)\Gamma(1) \right\rfloor \tag{11}$$

where K is the Accumulator Initialization constant ($0 \leq K \leq D - 2$, where D is the fixed-size dynamic range of the data samples) For $t > 1$, the value for accumulator and counter are:

$$\Sigma_z(t) = \begin{cases} \Sigma_z(t-1) + \delta_z(t-1) & , \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \left\lfloor \frac{\Sigma_z(t-1)+\delta_z(t-1)}{2} \right\rfloor & , \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \tag{12}$$

$$\Gamma(t) = \begin{cases} \Gamma(t-1) + 1 & , \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \left\lfloor \frac{\Gamma(t-1)+1}{2} \right\rfloor & , \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \tag{13}$$

where γ^* is the rescaling counter size parameter.

Encoding: For $t > 0$, the codeword for the mapped prediction residual $\delta_z(t)$ depends on the values of $k_z(t)$ and $u_z(t)$, where $k_z(t) = 0$ if $2\Gamma(t) > \Sigma_z(t) + \lfloor \frac{49}{2^7}\Gamma(t) \rfloor$ otherwise $k_z(t)$ is the largest positive integer $k_z(t) \leq D - 2$ such that

$$\Gamma(t).2^{k_z(t)} \leq \Sigma_z(t) + \left\lfloor \frac{49}{2^7}\Gamma(t) \right\rfloor \tag{14}$$

and $u_z(t)$ is calculated as

$$u_z(t) = \left\lfloor \delta_z(t)/2^{k_z(t)} \right\rfloor \tag{15}$$

For $t > 0$, the codeword for $\delta_z(t)$ shall be determined as follows:
a) If $u_z(t) < U_{max}$ then the codeword for $\delta_z(t)$ shall consist of $u_z(t)$ zeros, followed by a one, followed by the $k_z(t)$ least significant bits of the binary representation of $\delta_z(t)$.
b) Otherwise, the codeword for $\delta_z(t)$ shall consist of U_{max} zeros, followed by the D-bit binary representation of $\delta_z(t)$.

4.7.4 Conclusion on Encoders

Golomb encoding gives best results for sources which have a probability distribution similar to a Gaussian distribution. While the body consists of the entropy encoded codewords, the header consists of the image metadata (12 bytes), predictor and encoder metadata, in formats as defined in the recommended standard. The image metadata consists of fields giving information about things like the dimensions of the image, the sample type (signed or unsigned), the encoding order, etc. The predictor metadata gives information about the predictor parameters like the number of previous bands taken for prediction, the type of prediction, weight initialization method, etc. Finally, the encoder metadata has parameters like the unary length parameter (U_{max}), initial count exponent (γ_0), accumulator initialization constant, etc.

4.8 Serial to Parallel IP

This IP, as the name suggests, divides the encoded value into packets of eight bits each. Since the memory is byte organised, it is easier for packets of one byte each to be stored into memory. Such organisation also allows easy retrieval of data from memory.

5. Interfacing

Interfacing the Compression Hardware with the rest of the system has three major components:

1. Getting Image data to and from the hardware.
2. Control of Hardware by the main Processor and User processes.
3. Recovery Mechanisms in case of any Emergency.

One of the major concerns in designing an efficient interface is to maintain a high CPU utilization. At the time of compression, CPU may be needed to perform other intensive tasks such as attitude control, telemetry, housekeeping etc. So there should be a way that the image compression hardware can access the image data without CPU intervention.

5.1 Getting Image data to and from the hardware

Here we assume that the Image data is initially stored on a peripheral memory since Hyperspectral Images are too large to be completely stored on the On-Chip Memory. So, this image must be provided to the compression hardware in parts. For storing the image, SD and SPI based flash memories are proposed due to high data transfer rates (compared to other peripheral buses such as I^2C and CAN).[6] Additionally, flash memories guarantee both the nonvolatility in case of power loss and a highest storage density as well as they are shock-resistant and power economic.[1]

In systems where there is no contention for use of the peripheral bus that connects the memory, the FPGA fabric can be used to generate a purely Hardware based peripheral controller. However, in most systems, these peripheral buses would be used by other components such as sensors and actuators. Therefore, the CPU must perform arbitration of the peripheral bus. Additionally, if the peripheral memory is used as a block device, block management would be hidden from the hardware.

To achieve communication between the compression hardware and peripheral memory, BRAM modules on the FPGA should be used as buffer holding spaces for both uncompressed and compressed image. Software can periodically load image data from the peripheral memory and store it into these buffers. When state of these buffers are low, interrupts can be sent to the Processing system to transfer more data.

5.2 Control of Hardware by main processor and User Processes

The flight plan of the satellite would be implemented as a user level process.[9] This flight-plan should be able to control the compression hardware and also receive feedback as asynchronous interrupts. For this GPIO based device drivers need to be created. The driver creates an abstraction of the hardware features in the form of device files and IOCTL calls.

5.3 Recovery in case of emergencies

Due to various external factors such as critically low power, sudden temperature changes, etc, the On-Board Computer might need to temporarily shut down the compression hardware. In such a case the data compressed till now would have to be compressed again because the statistics are adaptive. The satellite would loose a lot of time and energy in compressing it again. Due to this, there needs to be some recovery mechanism to recover from such crashes. This can be implemented in software for better control. One possible solution could be to save the "state" at regular intervals. For eg, whenever the input and output buffers are flushed to the peripheral memory, Software could save statistics such as current weights etc, to a non-volatile memory and effectively create a restore point from which compression can be resumed by simply refilling the FIFO queue and setting the values of the statistics to saved ones.

6. Control Path

The Datapath creates a few pipelining hazards when implemented. To overcome this, the controller of the Datapath is connected to each IP and sends control signals to them. The IPs are enabled by the controller at specific instants to overcome the bottlenecks that creep in due to multiple feedback loops in the algorithm and also due to the fact that a some IPs are sequential and some are combinational.

Let us denote the current time by t. The following table shows the appropriate instances when the IPs are enabled to avoid hazards and efficiently compress the image.

Table 4: Enable Signals to IPs

Sum and Diff	**Weight Update**	**Weights**	**Dot Product**	**Predictor**
$t-2$	*	$t-2$		
*	$t-1$	*	$t-2$	$t-2$
$t-1$	*	$t-1$	*	*
$\hookrightarrow$	t	*	$t-1$	$t-1$
		t		

7. Future Work

Future work includes:

1. Development of a co-processor implementing the CCSDS algorithm alongside a RISC processor.
2. A testbench will be created to test the synthesised algorithm. This testbench will monitor the power constraints when particular parameters are changed in the algorithm.
3. The Image Compression mode of the satellite will be tested alongside all the other modes in which the satellite will operate.[7]
4. Software control for the hardware has to be developed, so that the OBC can monitor the compression on board the satellite.

References

[1] Maurizio Caramia, Stefano Di Carlo, Michele Fabiano, and Paolo Prinetto. Exploring design dimensions in flash-based mass-memory devices. Oct 2010.

[2] The consultative Committee for Space Data Systems. *Lossless Multispectral and Hyperspectral Image Compression, Recommended Standard CCSDS 123.0-B-1*. May 2012.

[3] Sr. Dr. Nicholas M. Short. Graphic representation of hyperspectral data, 2007.

[4] Jet Proportional Laboratory. Aviris & hyperion hyperspectral images. Accessed: 2019-06-19.

[5] M.A. Enderwitz L.H. Crockett, R.A. Elliot and R.W. Stewart. *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media, first edition, 2014.

[6] Elyas Razzaghi. *Design and Qualification of On-Board Computer for Aalto-1 CubeSat*. PhD thesis, Luleå University of Technology, Department of Computer Science, Electrical and Space Engineering, 2012. Validerat; 20120828 (anonymous).

[7] Kushagra Aggarwal Tushar Goyal Abhinav Sundhar Ujjwal Anand Nishad Sahu Joy Parikh Rutwik Jain, Shubham Sharma. Modes of operation for a 3u cubesat with hyperspectral imaging payload. In *69th International Astronautical Congress(IAC)*, 2018.

[8] Lucana Santos, Luis Berrojo, Javier Moreno, José Francisco López, and Roberto Sarmiento. Multispectral and hyperspectral lossless compressor for space applications (hyloc): A low-complexity fpga implementation of the ccsds 123 standard. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 9:757–770, 2016.

[9] Shubham Sharma Kushagra Aggarwal Dhananjay Mantri Tanuj Kumar Saurabh M. Raje, Abhishek Goel. Development of on board computer for a nanosatellite. In *68th International Astronautical Congress(IAC)*, 2017.

[10] Dharam Shah, Kuhelika Bera, and Sanjay Joshi. Software implementation of ccsds recommended hyperspectral lossless image compression. *International Journal of Image, Graphics and Signal Processing*, 7:35–41, 03 2015.

[11] Xilinx. *Zedboard Hardware User's Guide*. 2014.