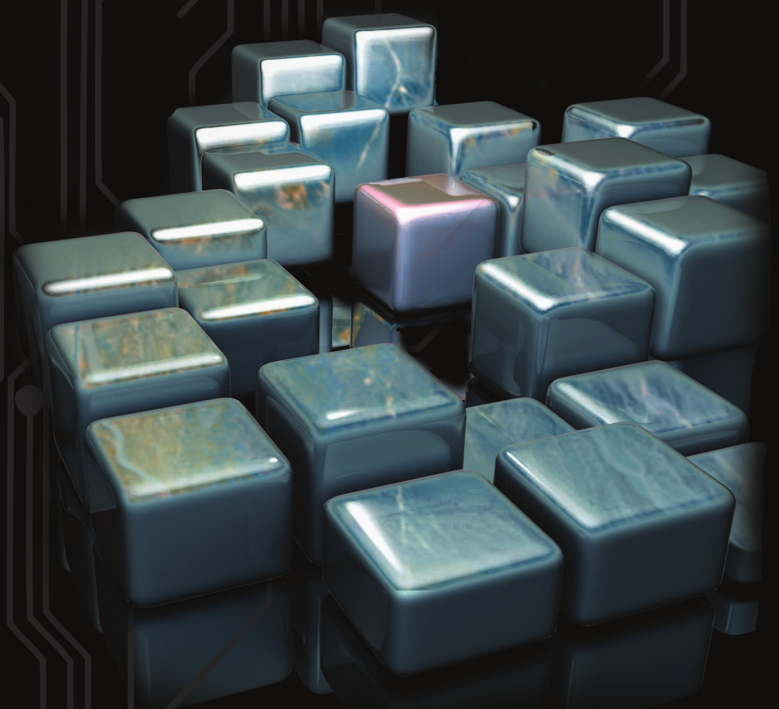
UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada

HYPERSPECTRAL IMAGE COMPRESSION ONBOARD NEXT-GENERATION SATELLITES: IMPLEMENTATION SOLUTIONS ON GPUs AND FPGAs



Tesis Doctoral

María Lucana Santos Falcón

Las Palmas de Gran Canaria, Julio 2014

D. PEDRO PÉREZ CARBALLO SECRETARIO DEL INSTITUTO UNIVERSITARIO DE MICROELECTRÓNICA APLICADA DE LA UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA,

CERTIFICA,

Que el Consejo de Doctores del Departamento en su sesión de fecha **seis de junio de 2014** tomó el acuerdo de dar el consentimiento para su tramitación, a la tesis doctoral titulada "***Hyperspectral image compression onboard next-generation satellites: implementation solutions on GPUs and FPGAs***" presentada por la doctoranda Dña. María Lucana Santos Falcón y dirigida por los Doctores D. José Francisco López Feliciano y D. Roberto Sarmiento Rodríguez.

Y para que así conste, y a efectos de lo previsto en el Artº 6 del Reglamento para la elaboración, defensa, tribunal y evaluación de tesis doctorales de la Universidad de Las Palmas de Gran Canaria, firmo la presente en Las Palmas de Gran Canaria, a **seis de junio de 2014**.

D. PEDRO PÉREZ CARBALLO SECRETARIO DEL INSTITUTO UNIVERSITARIO DE MICROELECTRÓNICA APLICADA DE LA UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA,

CERTIFICA,

Que el Consejo de Doctores del Departamento en su sesión de fecha **seis de junio de 2014** ha acordado que la tesis doctoral titulada "***Hyperspectral image compression onboard next-generation satellites: implementation solutions on GPUs and FPGAs***" presentada por la doctoranda Dña. María Lucana Santos Falcón y dirigida por los Doctores D. José Francisco López Feliciano y D. Roberto Sarmiento Rodríguez **reúne todos los requisitos para optar a la acreditación de DOCTORADO EUROPEO**.

Y para que así conste, firmo la presente en Las Palmas de Gran Canaria, a **seis de junio de 2014**.

P.P.

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada

Instituto: INSTITUTO UNIVERSITARIO DE MICROELECTRÓNICA APLICADA

Programa de doctorado: INGENIERÍA DE TELECOMUNICACIÓN AVANZADA

Título de la Tesis

HYPERSPECTRAL IMAGE COMPRESSION ONBOARD NEXT-GENERATION SATELLITES: IMPLEMENTATION SOLUTIONS ON GPU AND FPGAs

Tesis Doctoral presentada por Dña. MARÍA LUCANA SANTOS FALCÓN

Dirigida por el Dr. D. ROBERTO SARMIENTO RODRÍGUEZ

Codirigida por el Dr. D. JOSÉ FCO. LÓPEZ FELICIANO

El Director/a,	**El Codirector/a**	**la Doctoranda,**
(firma)	**(firma)**	**(firma)**

Las Palmas de Gran Canaria, a 6 de junio de 2014

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada

DIVISIÓN DE DISEÑO DE SISTEMAS INTEGRADOS

TESIS DOCTORAL

Hyperspectral image compression onboard next-generation satellites: implementation solutions on GPUs and FPGAs

María Lucana Santos Falcón

Abstract

The compression of multispectral and hyperspectral on-board satellites is at the same time a challenge and a necessity whose importance is currently growing as the resolution of the sensors tend to increase. The images are captured in several different wavelengths, which can range from tens to hundreds or thousands, and therefore represent a huge amount of data which has to be reduced in order to meet the available on-board storage and the transmission bandwidth limitations.

On-board multispectral and hyperspectral compression algorithms have to meet several requirements which are specific to the space environment such as low complexity and error resilience. The available processing power on a satellite is limited, and most usual data compression algorithms used on ground cannot be applied to space data systems. Therefore, besides the proposal of new compression techniques, it is also necessary to develop new physical implementations which are able to execute the proposed algorithms on devices suitable for operating on-board a satellite.

This Thesis proposes new technological solutions for the physical implementation and execution of multispectral and hyperspectral compression on-board satellites. In particular, we explore the implementation of a lossy compression algorithm named LCE on a GPU and an FPGA, and devise strategies in order to accelerate it by exploiting parallel processing features.

First, a parallelization strategy is designed for both the LCE compressor and the corresponding decompressor, and they are implemented on a GPU using Nvidia's CUDA parallel architecture. Experimental results on several hyperspectral images with different spatial and spectral dimensions show significant speedups of up to 15 times faster with respect to a single-threaded CPU implementation.

We present furthermore an FPGA implementation of the LCE algorithm. The results of the implementation on a Virtex 5VFX130 display effective performance in terms of area (maximum device utilization at 14%) and frequency (86 MHz). A comparison of the technologies utilized to implement the LCE is also provided, showing that, although the GPU is the one yielding the highest throughput, the FPGA offers the best tradeoff between performance, low power consumption and flexibility.

Finally, a low complexity FPGA implementation of the recent CCSDS-123 standard for multispectral and hyperspectral compression is presented. A hardware architecture is conceived and designed with the aim of achieving low hardware occupancy and high performance on a space-qualified FPGA from the Microsemi RTAX family. The resulting FPGA implementation is therefore suitable for on-board compression. The effect of the several CCSDS-123 configuration parameters on the compression efficiency and hardware complexity is taken into consideration to provide flexibility in such a way that the implementation can be adapted to different application scenarios. Results show an occupancy of 34% and a maximum frequency of 43 MHz on an RTAX1000S.

The benefits of the proposed implementations have been addressed and compared in terms of the computational performance, the cost of the solution and the flexibility of the implementation. Ultimately, this work is intended to make contributions to the future space missions, providing solutions which can yield implementations of hyperspectral compression algorithms with increased flexibility, high-performance and low power consumption.

Resumen

La compresión de imágenes multiespectrales e hiperespectrales en satélites supone al mismo tiempo un reto y una necesidad, cuya importancia se está haciendo cada vez mayor a medida que la resolución de los sensores tiende a aumentar. Las imágenes se capturan en un número de longitudes de onda que puede variar desde decenas hasta miles, conformando un cubo de datos cuyo tamaño debe ser reducido para cumplir con las limitaciones actuales en relación a la cantidad de almacenamiento disponible a bordo y los anchos de banda de las comunicaciones.

Los algoritmos de compresión de imágenes multiespectrales e hiperespectrales deben cumplir determinados requisitos específicos del entorno espacial, como son la baja complejidad y la tolerancia a errores, entre otros. La capacidad computacional de los equipos que operan en los satélites es limitada, y en consecuencia, la mayoría de los algoritmos para compresión de datos usados en el sector terreno no pueden ser utilizados en el espacio. Es por ello que, además de proponer nuevas técnicas de compresión, es necesario desarrollar las implementaciones físicas para ejecutar dichos algoritmos en dispositivos aptos para trabajar en el espacio.

En esta Tesis se proponen nuevas soluciones tecnológicas para la implementación física y ejecución de algoritmos de compresión de imágenes hiperespectrales a bordo de satélites. En concreto, se estudia la implementación de un algoritmo de compresión sin pérdidas denominado LCE en una GPU y en una FPGA, y se elaboran estrategias para acelerarlo mediante técnicas de procesamiento paralelo.

Se realiza, por tanto, una paralelización del compresor LCE y de su correspondiente descompresor, y se implementa en una GPU utilizando Nvidia

CUDA. Los resultados experimentales, realizados sobre imágenes hiperespectrales de diferentes tamaños en la dimensión espacial y espectral muestran que se ha obtenido una aceleración significativa. La implementación para GPU del LCE se ejecuta 15 veces más rápido que su equivalente sobre CPU.

Además, se presenta una implementación del LCE sobre una FPGA. Los resultados experimentales muestran una ocupación de área reducida (como máximo del 14 %) y una frecuencia de 80 MHz cuando se sintetiza en una Virtex 5 5VFX130. Finalmente, se realiza una comparación de las diferentes tecnologías utilizadas para implementar el LCE, en la que se muestra que, aunque la GPU tiene un rendimiento superior en cuanto a número de muestras procesadas por unidad de tiempo, la FPGA ofrece el mejor compromiso entre rendimiento, bajo consumo de potencia y flexibilidad.

Este trabajo de investigación se completa con una implementación de baja complejidad del recientemente publicado estándar CCSDS 123 para la compresión de imágenes multiespectrales e hiperespectrales sin pérdidas. Se realiza un diseño arquitectural con el objetivo de conseguir una baja ocupación de recursos hardware y alto rendimiento en una FPGA cualificada para el espacio, en concreto la RTAX1000S de la familia Microsemi. La implementación resultante es, por lo tanto, apta para ser utilizada a bordo de un satélite. Durante esta investigación se realiza un estudio del efecto de los distintos parámetros de configuración que admite el estándar CCSDS 123 en el ratio de compresión y en la complejidad de la implementacón resultante, de modo que sea flexible para adaptarse a distintos escenarios manteniendo una baja complejidad. Los resultados muestran una ocupación del 34% y una frecuencia máxima de 43 MHz en la citada RTAX1000S.

Las ventajas de las soluciones presentadas se ponen de manifiesto y se comparan en cuanto a su rendimiento, el coste de la solución y la flexibilidad de la implementación. En definitiva, este trabajo pretende realizar una contribución para las misiones espaciales futuras, de manera que cuenten con

soluciones capaces de ejecutar algoritmos de compresión de imágenes hiperespectrales con más flexibilidad, alto rendimiento y bajo consumo de potencia.

Acknowledgements

I would not have been able to complete this journey without the aid and support of countless people over the past four years. Foremost, I would like to express my gratitude to my supervisors, Prof. José López and Prof. Roberto Sarmiento, who have been greatly supportive and have guided me during this research and while writing this Thesis, offering constructive comments and warm encouragement. Over the years, I have received funding from several entities, which have supported me while I completed my PhD. I would like to thank Thales Alenia Space España S.A., the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HIPEAC), the Institute for Applied Microelectronics (IUMA) and Ayuntamiento de Las Palmas de Gran Canaria for their financial support.

I highly appreciate the feedback offered by Luis Berrojo and Javier Moreno, from Thales Alenia Space España S.A., who have made an important contribution to the technical quality of this Thesis. I also thank the European Space Agency for offering me the opportunity to make a fruitful research stay. In particular I would like to thank Raffaele Vitulli and Luca Fossati for their generous support and for sharing their immense knowledge. I am also grateful to the Group on Interactive Coding of Images (GICI) from Universitat Autònoma de Barcelona and to Enrico Magli, from Politecnico di Torino for kindly offering their help. I'd like to thank also my fellow labmates, for all the stimulating discussions, the fun we have had, the coffees and their patience. I could not have imagined a better work environment for pursuing the PhD.

Last, but not least, I am deeply grateful to my parents for generously offering me support and the education that has made it possible for me to get here. Thanks also to my sister, Isabel, the rest of my family and to Lola and Ricardo, for instilling me confidence and believing in me.

Contents

List of Figures

List of Tables

Abbreviations

2D	two-dimensional
3D	three-dimensional
ASIC	application-specific integrated circuit
BIP	band interleaved by pixel
BPE	bit plane encoder
CALIC	Context-based, Adaptive, Lossless Image Codec
CCD	charge-coupled device
CCSDS	Consultative Committee for Space Data Systems
CMOS	complementary metal-oxide semiconductor
CNES	Centre National d'Etudes Spatiales
COTS	commercial off-the-shelf
CPU	central processing unit
CUDA	Computer Unified Device Architecture

DCT	Discrete Cosine Transform
DPCM	differential pulse code modulation
DSP	digital signal processor
DWT	Discrete Wavelet Transform
EDAC	error detection and correction
EEPROM	electrically erasable programmable read only memory
EPROM	erasable programmable read only memory
ESA	European Space Agency
FAPEC	Fully Adaptive Prediction Error Coder
FET	field effect transistor
FL	Fast-Lossless
FLOPS	floating-point operations per second
FP7	The Seventh Framework Programme
FPGA	field-programmable gate array
FSM	finite-state machine
GDDR	Graphics Double Data Rate
GPU	graphics processing unit
HLS	high-level synthesis
ITI	Innovation Triangle Initiative

JPL Jet Propulsion Laboratory

KLT Kahrunen-Loève Transform

LCE Lossy Compression for Exomars

LUT lookup-table

LWIR long wavelength infrared

MAE maximum absolute error

MSE mean-squared error

MSRE mean square-root error

MWIR medium wavelength infrared

NASA National Aeronautics and Space Administration

NIR near infrared

NOAA National Oceanic and Atmospheric Administration

OpenMP Open Multi-Processing

PC principal component

PCA Principal Component Analysis

PLB Processor Local Bus

PSNR peak signal-to-noise ratio

RAM random access memory

RD rate-distortion

RD	rate-distortion
RTL	register-transfer level
SDRAM	synchronous dynamic random access memory
SEE	single event effects
SEU	single event upset
SLSQ	Spectrum-oriented Least Squares
SM	streaming multiprocessor
SMID	single instruction multiple data
SoC	System-on-Chip
SPECK	set partitioned embedded block
SPIHT	set partitioning in hierarchical trees
SRAM	static random access memory
SWIR	short wavelength infrared
TD	Tucker Decomposition
TDP	thermal design power
TID	total ionizing dose
TIR	thermal infrared
TMR	triple modular redundancy
UTQ	uniform-threshold quantizer

VNIR visible and near infrared

VSWIR visible shortwave infrared

Symbols

x	row index
y	line index
z	band index
n	row index within a block
m	line index within a block
s	image sample
$\hat{s}$	predicted sample
$\tilde{s}$	reconstructed sample
α	least-square estimator
μ	average value
α'	quantized value of α
μ'	quantized value of μ
$e_{z,y,x}$	prediction error
k	Golomb-code parameter
Nx	number of rows
Ny	number of lines
Nz	number of bands
N	number of pixels in a squared spatial block

hb	horizontal block index
vb	vertical block index
$B_{z,hb,vb}$	$N \times N$ block with spatial coordinates (hb, vb) in band z
Nhb	number of horizontal blocks
Nvb	number of vertical blocks
NB	total number of blocks
i	block index $i = hb + vb \times Nhb$
j	sample index within a block of $N \times N$ samples
	$j = m + n \times N$
q_j	number of bits taken by the j-th codeword of a block
Q_j	bit position of the j-th codeword in the compressed block
l_i	bits left unused in the last 32-bits word of the
	i-th compressed block
L_i	prefix-sum of l_i
p_i	position of the last codeword within block i
P_i	word position where compressed block i starts in the
	compressed stream
CH	chunk
NC	number of chunks
K	first element of the last chunk
$\lceil x \rceil$	ceil x to the nearest integer
$\lfloor x \rfloor$	floor x to the nearest integer
sh_left_i	number of bits that block i has to be shifted to the left
sh_right_i	number of bits that block i has to be shifted to the right

Dedicado a Isabel, Lucana, Juan y Ricardo

Chapter 1

Introduction

The main objective of this Chapter is to present the motivation for this Thesis, whose major contributions are to the field of hardware platforms for on-board multispectral and hyperspectral image compression. The basic concepts of this work are introduced by briefly explaining the notion of multispectral and hyperspectral images, how they are collected and their applications. Furthermore, the necessity for performing on-board multispectral and hyperspectral image compression is justified and the grounds of data compression are explained. Finally, the research goals and methodology followed throughout this Thesis work are presented.

1.1 Outline

Hyperspectral and multispectral imaging system are considered nowadays the most powerful tools in the field of remote sensing. These systems are able to provide images in which single pixels have spectral information of the scene under observation.

Airborne multispectral sensors have recorded spectral information since the mid 1950s, but it was since the early 1970s that a large number of spaceborne multispectral sensors have been launched on-board satellites, like for instance the LANDSAT [1], SPOT [2] or the Indian Remote Sensing (IRS) [3] satellite series. The first sensor considered hyperspectral and hence capable of acquiring data in continuous narrow bands simultaneously, was the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) [4], proposed by the National Aeronautics and Space Administration (NASA) in 1983. AVIRIS is nowadays still considered the best hyperspectral remote sensing sensor ever manufactured, and represents a significant precursor and driving force for the development of remote sensing technologies as a whole [5]. Airborne and spaceborne remote sensors were conceived to satisfy the scientific and research data requirements, nevertheless these data are used today in many commercial applications too.

Current remote sensors cover a large area of the Earth surface with unprecedent spatial and spectral resolution. This yields accurate imagery which enables a variety of applications, e.g. identification of materials or estimation of physical parameters. New sensors are expected to increase their resolution and will therefore be able to capture even more data. This, together with the development of new software tools for multispectral and hyperspectral image analysis, has positioned hyperspectral imagery to become one of the most common research, and fastest growing technologies in the field of remote sensing.

The high dimensionality of multispectral and hyperspectral data is highly advantageous from the point of view of image analysis for scientific purposes. However, a challenge appears when the images are acquired on a satellite, where the amount of storage and downlink bandwidth are limited. Data compression can alleviate this problem, by reducing the data volume prior to transmission to a ground station. It has become a popular research field in the past years, and many different algorithms have been proposed for satellite data compression. These algorithms have the particular feature of meeting specific requirements, as a consequence of the limited on-board computational power available on a satellite and the special operating conditions of the hardware in the space environment.

Developing a physical implementation of a multispectral and/or hyperspectral data compression algorithm which can efficiently operate on a satellite is an important and exciting challenge. Hardware technologies running on-board a satellite have to feature high-performance, and at the same time they must be small in size and weight, have low power consumption and be tolerant to radiation. This Thesis work provides a discussion and proposes several technological solutions for hardware implementations for on-board hyperspectral image compression.

1.2 Preliminary concepts

The following sections introduce the reader to the main topics that will be repeatedly mentioned throughout this document, including a definition of multispectral and hyperspectral images, a brief explanation about how they are acquired and the basics of remote sensing data compression.

1.2.1 Multispectral and hyperspectral images

Remote imagers are designed to measure the reflectance of areas on the Earth's surface. Reflectance is defined as the percentage of the light hitting a material that is then reflected by that material, and can be represented across a range of wavelengths, showing a pattern or spectrum, which can uniquely identify certain materials. In imaging spectroscopy, a sensor acquires a spectral vector in hundreds or thousand different wavelengths of every pixel in a given scene. This results in a three-dimensional image known as hyperspectral or multispectral image. Hyperspectral and multispectral imagery is typically depicted as a data cube with the spatial information represented in the X-Y plane, and spectral information represented in the Z-direction, as it is shown in Figure 1.1, where Ny, Nx and Nz are the number of lines, columns and bands respectively.

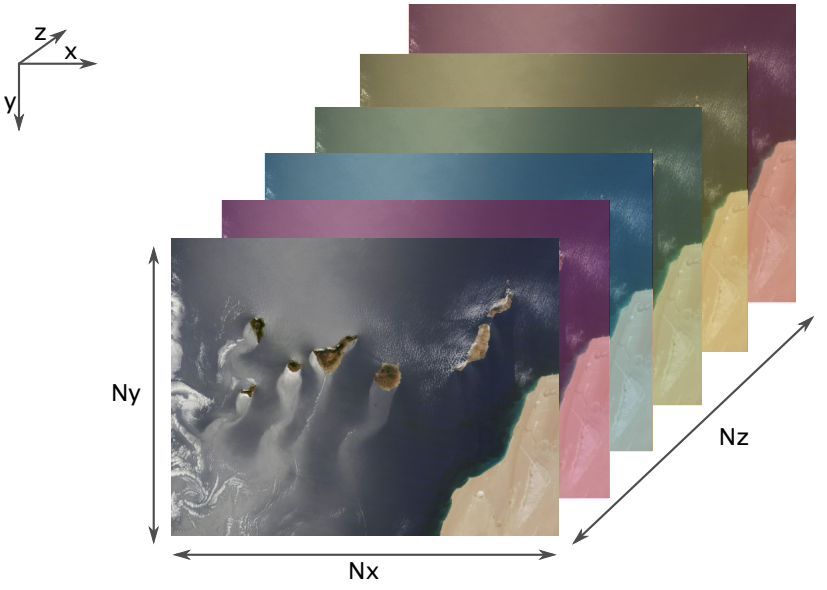


FIGURE 1.1: Hyperspectral data cube.

There is no agreed criterion on how to make the difference between hyperspectral and multispectral images. The literature includes definitions in terms of the number of bands, their narrowness or if the bands are contiguous or there are gaps between them. However, analysing the available documents, we find that it is reasonable to define hyperspectral imaging systems as those which collect at least 100 spectral bands of 10-20 nm width. Multispectral sensors, on the other hand, can be defined as those collecting less than 20 non contiguous bands [6, 7]. Hyperspectral imagers produce a detailed, seemingly continuous spectrum that can provide much more information than a multispectral pixel spectrum. Hence, hyperpsectral systems have a very wide capability of spectral discrimination, while multispectral systems provide bands that detect information in specific combinations of desirable regions of the spectrum. Most hyperspectral and multispectral imaging systems work in a wavelength range from the visible to the infrared, the latter commonly divided in bands called near infrared (NIR), λ: 0.7 - 1.1 μm; short wavelength infrared (SWIR), λ: 1.1 - 3.0 μm; medium wavelength infrared (MWIR), λ: 3.0 - 5.5 μm; and long wavelength infrared (LWIR), λ: 7.7 - 14 μm. Figure 1.2 shows the electromagnetic spectrum.

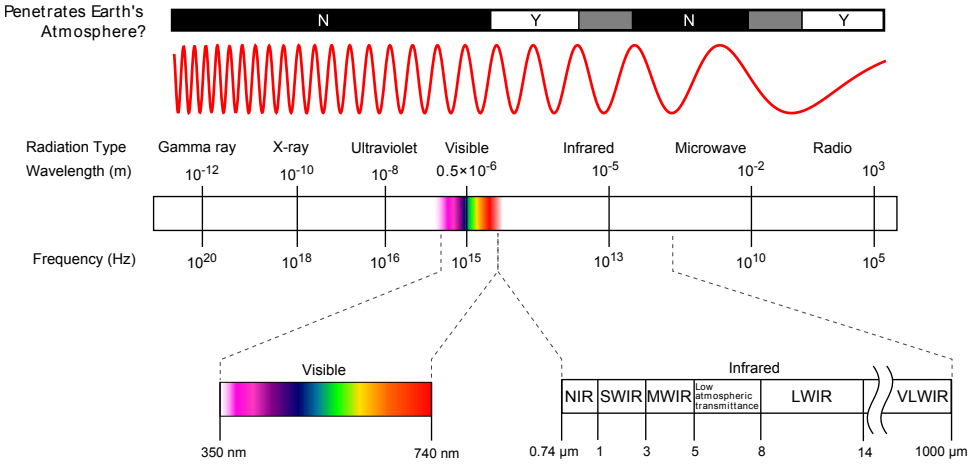


FIGURE 1.2: Electromagnetic spectrum.

Although multispectral and hyperspectral images have different features, for the sake of simplicity, in the rest of this document we will use the term "hyperspectral" to refer to both.

1.2.2 Instruments and sensors for hyperspectral data collection

The data collection in hyperspectral systems is a four dimensional problem, consisting of two spatial, one spectral and one time dimension, i.e. the collection of a hyperspectral cube requires scanning, in a time interval, in the spectral or spatial domain. In a remote sensing instrument, the light reflected by objects is captured by photodectector arrays, in which each element records one pixel of the image.

Remote sensors, as opposed to photographic, acquire data using scanning systems that sweep over the terrain to build up and produce a two-dimensional image of the surface. The hyperspectral scanners are of two main types: whiskbroom and pushbroom, as depicted in Figure 1.3. Whiskbroom scanners or across track scanners reflect light into a single dectector using a mirror which moves back and forth to collect measurements from one pixel in the image at a time. All LANDSAT sensors prior to LANDSAT 8 use the whiskbroom design [1]. Other examples of whiskbroom imagers are the ones from the National Oceanic and Atmospheric Administration (NOAA): the Advanced Very High Resolution Radiometer (AVHRR) and the Geostationary Operational Environmental Satellite (GOES). On the other hand, pushbroom scanners, or along track scanners, use a line of detectors arranged perpendicular to the flight direction of the spacecraft. The image is collected one line at the time as the spacecraft flies forward. Instruments that use the pushbroom design include the ones on-board LANDSAT 8 [1], SPOT [2], IRS [3], PLEIADES [8] and PROBA [9]. The selection of the type of scanner depends mainly on the purpose and specific requirements of the

remote sensing mission. Whiskbroom scanners are mechanically more complex but simpler from the optical point of view, while pushbroom imagers consist of a simpler mechanical system but more complex optics. The swath width is longer for whiskbroom imagers, when compared with pushbroom. Moreover pushbroom sensors must have the detectors perfectly calibrated to avoid stripes in the data caused by the varying sensitivity in the along track direction.

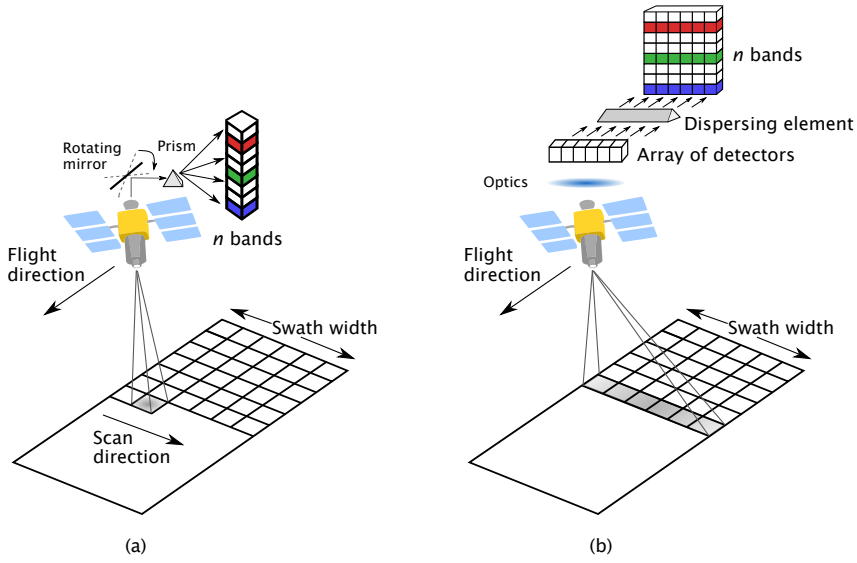


FIGURE 1.3: Remote sensing scanners. a) Whiskbroom b) Pushbroom

As it was already mentioned, hyperspectral imagers have to collect information of each pixel at different wavelengths. There are several types of devices for the spectral division or selection, which can be divided in three main classes: dispersive spectrometers; Fourier Transform interferometers; and narrow band tunable filters. Dispersive spectrometers use grating or a prism coupled with a two-dimensional array of detectors, in such a way that a spectral image is produced with the spatial information along one axis and

the spectral information along the other. Fourier Transform interferometers split the radiation into two beams, introducing a controlled phase shift, and recombining them. The wavefronts of the beams on recombination interfere by the principle of superposition and the combined beam is focused on a detector. Finally, the narrow band tunable filters pass radiation through a very narrow bandpass or spectral bin, which can be spectrally tuned over different wavelengths, usually in a very short time.

Although most past and current hyperspectral sensors have been airborne, many new space-based hyperspectral sensors have been proposed recently. These sensors have become increasingly important, because they are able to achieve near global coverage repeated at regular intervals of time, providing more imagery than their airborne counterpart.

Once the image samples are detected they are converted into digital values, which can be arranged in different forms, commonly known as band-sequential or band-interleaved formats. In band sequential order, the samples are stored in raster order, band by band, i.e. all the samples in a specific band are stored before the storing of the samples in the next band start. On the other hand, in band interleaved format, the information is stored in such a way that all the spectral information of a specific pixel or line of pixels is stored before continuing with the next pixel or line of pixels. This is better illustrated in Figure 1.4.

1.2.3 Applications of hyperspectral images

Hyperspectral technologies were developed for science and research purposes, but have progressed to allow the development of a diversity of commercial applications. The captured three-dimensional data cubes are processed after they are collected, so that the most relevant information can be extracted from them [10]. This information is extremely useful in many diverse fields,

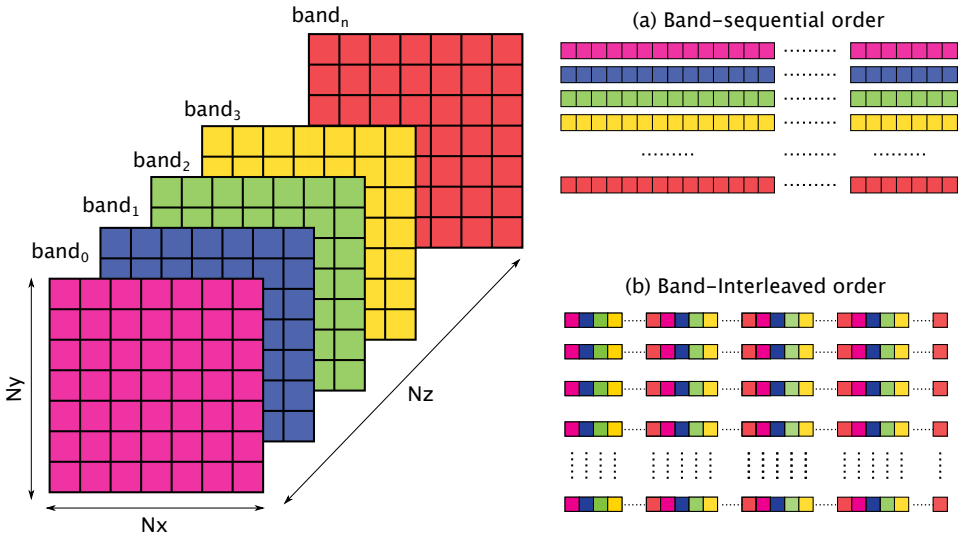


FIGURE 1.4: (a) Band-sequential order (b) band-interleaved order

including mineralogy, Earth monitoring, surveillance and medicine. Some of the applications of hyperspectral data analysis are listed next.

- Atmospheric characterization and climate research
- Geologic mapping
- Environmental monitoring
- Vegetation analysis, food safety
- Monitoring of coastal environment
- Urban grow analysis
- Biological and chemical detection
- Non invasive diagnosis in cancer detection
- Surveillance

- Detection of weapons of mass destruction
- Detection of landmines

Some of the main techniques utilized to extract useful information from hyperspectral data are summarized next.

- *Dimensionality reduction.* The dimensionality of the input hyperspectral image is reduced, in order to facilitate the subsequent processing of the scene [11].
- *Hyperspectral unmixing.* The signal captured by a hyperspectral sensor at a given band and from a given pixel is a mixture of the reflectances of the different materials located in the respective pixel area. When mixing occurs, it is not any more possible to determine which materials are present in the pixels directly from the measured spectral vectors. The ability to discriminate materials can be recovered by applying hyperspectral unmixing techniques. An extensive amount of research work has been devoted to hyperspectral unmixing [12, 13].
- *Classification.* Hyperspectral image classification has been a very active area of research in recent years [14]. Given a set of observations, the goal of classification is to assign a unique label to each pixel vector, so that it is well-defined by a given class.
- *Hyperspectral target detection.* Hyperspectral imagery has been used in reconnaissance and surveillance applications where targets of interest are detected and identified [15]. In the process of detecting a target, first the anomalies in the image are extracted [16, 17]. Then, the targets can be identified by their spectral signature, by comparing it with the data available in a spectral library [18] or from a set of training data.

1.2.4 Hyperspectral image compression

Hyperspectral images acquired by aircrafts or satellites represent a high amount of data. For instance, a single AVIRIS image occupies 134 Mbytes of data storage. In the specific case of remote sensors placed on satellites, the computational power, storage and downlink bandwidth are limited, therefore applying compression techniques have become an effective and cheap solution in order to cope with these restrictions and still allow the hyperspectral imagers to produce images at a high data rate.

Compression of hyperspectral images is effective because the pixel values of neighbouring locations and wavelengths are highly correlated. Removing the correlation allows for reducing the data volume. Hyperspectral images have similar characteristics to natural photographic images or video, and consequently their size can be reduced with compression tools which were developed for image or video [19, 20].

The fundamentals of data compression are based on representing the necessary information with the smallest possible amount of bits. In general, compression can be lossless or lossy. Lossless techniques make it possible to recover all the original information after decompression, while lossy methods permit to recover the original data with some losses of information, enabling higher compression ratios.

Compression techniques take advantage of several facts. The spatial or spectral redundancy in the images makes it possible to deduct a pixel value by using information of neighbouring pixels. Decorrelation methods like prediction o transformation can be applied, in such a way that it is no longer necessary to transmit the whole pixel information, but just the necessary information for the decoder to be able to calculate - or approximate if the technique is lossy - the value of the original pixel. In addition, compression methods exploit the statistical redundancy in the data, exploring the

probability of the symbols in such a way that long codewords are used to represent symbols with low probability and short codewords are utilized to represent the most frequent symbols. These codes try to reduce the redundancy present in a source of information, and represent it with fewer bits which carry more information, minimizing the average length of the messages according to a particular assumed probability model, which is known as entropy encoding. Among the most common entropy encoding techniques we can find:

- *Huffman codes.* In Huffman codes [21], the source is encoded using a table of codes of variable length, which has been derived based on the estimated probability of occurrence for each possible value of the source symbol.

- *Arithmetic codes.* These type of codes convert strings of data into single floating point numbers between 0 and 1. They establish a model of the entire data set and find the occurrences of sequences of symbols that can be expressed in the form of a single number with high precision.

- *Universal codes.* Universal codes are prefix codes which map positive integers into binary codewords, ensuring that the length of the resulting codeword is within a constant factor of the expected lengths that an optimal code would have assigned.

- *Golomb codes.* These codes are prefix codes, used when the distribution of the source data is geometric. Golomb codes [22] first find the quotient and the remainder of the division of the source and a tunable parameter. The codeword consists of the quotient of the division expressed in unary notation, followed by a stop bit and the remainder in truncated binary notation, as showed in Figure 1.5. These codes are known as Rice codes [23] when the tunable parameter (the divisor) is a power of two. Golomb and Rice codes are widely used for lossless

image and video compression, as well as for satellite data compression. In fact, it is employed in the Consultative Committee for Space Data Systems (CCSDS) standard for universal lossless satellite data compression, CCSDS 121 [24]; and in the standard for multispectral and hyperspectral image compression CCSDS 123 [25].

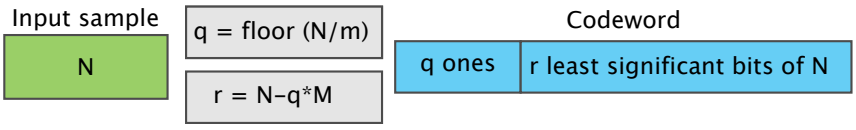


FIGURE 1.5: Example of the generation of the codeword with Golomb codes

Regardless the selected type of entropy encoding, there is a limit which determines the smallest possible expected number of bits needed to encode an event, known as the Shannon limit [26]. Shannon limit establishes that, given a set of mutually distinct events $e_1, e_2, e_3 ... e_n$, and the probability distribution P of the events, the smallest possible expected number of bits needed to encode an event is the *entropy* of P, denoted by:

$$H(P) = \sum_{k=1}^{n} -p\{e_k\} \log_2 p\{e_k\} \tag{1.1}$$

In the former equation, $p\{e_k\}$ is the probability that event $\{e_k\}$ occurs. An optimal code outputs $-\log_2 p$ bits to encode an event whose probability of occurrence is p.

It is important to assess how well a specific compression method performs when it is employed, since the compression efficiency will depend on the compression technique as well as on the distribution of the image data. The

compression ratio is calculated and expressed commonly in two different ways: as the relationship between the size in bits of the raw image and the compressed image; or as the number of bits necessary to represent a pixel after compression, also known as bits per pixel. The formulas that can be employed to estimate how much a compression method reduces the data volume are shown next:

- *Compression ratio*

$$CR = \frac{Size\ of\ original\ image(bits)}{Size\ of\ compressed\ image\ (bits)} \tag{1.2}$$

- *Bits per pixel per band*

$$bpppb = \frac{Size\ of\ compressed\ image\ (bits)}{Lines \times Columns \times Bands} \tag{1.3}$$

1.2.4.1 Lossless versus lossy compression

As it was mentioned, compression can be lossless or lossy. Lossless compression allows the source data to be reconstructed perfectly, therefore it is acceptable to be used to compress any kind of scientific data without sacrificing data quality. On the contrary, lossy compression removes some information in order to achieve higher compression ratios. The removed information cannot be recovered when the data are decompressed.

The compression ratios for lossless techniques are typically limited to values around 2 or 3, except for data with low information density such as a black sky. Another limitation of lossless compression is the fact that the compression ratios cannot be predicted in advance, what makes it difficult to estimate the amount of data that will be sent, i.e. the necessary downlink capability. Furthermore, it is necessary to introduce error containment

strategies, since when the data are corrupted, it can no longer be perfectly reconstructed and errors can propagate.

Lossy compression allows to find a trade-off between source fidelity and compression ratio and achieves significantly higher compression ratios, although it shares some of the limitations of the lossless techniques, like the possibility of error propagation. There is no upper limit for the ratio that can be achieved with lossy compression techniques, which is established depending on the downlink constraint and the amount of original data measured.

Furthermore, progressive compression provides a bridge between lossless and lossy compression methods. It partitions the data into ordered hierarchical segments. Each compressed segment, when combined with the previous ones, allows for the reconstruction of successively higher fidelity versions of the data. The initial version of the reconstruction is very lossy, while the final reconstruction can in principle be lossless or nearly lossless.

1.3 Motivation of research

While the resolution of the remote sensors, and consequently the data rates, continue to increase, the available downlink bandwidth is comparatively stable, as has been observed by NASA [27] and European Space Agency (ESA) [28]. The solution offered is to apply data compression, hence payload data processors on-board satellites have to be able to accomplish this task. In particular, Earth Observation missions have the highest performance needs for data processing, data reduction and compression, and future missions and applications will require more powerful on-board processing platforms.

1.3.1 Importance of on-board hyperspectral image compression

The challenge and importance of data compression on-board satellites can be illustrated with a few example missions, which are briefly described next.

The Euclid space science mission was selected by ESA to be launched in 2019 in the Cosmic Vision Program [29]. Its goal is to understand the origin of the accelerating expansion of the Universe by mapping the geometry of the dark universe. Euclid is not an Earth Observation mission, it was conceived to observe the Universe. It carries probes able to measure the shape and spectra of several hundreds of millions of galaxies over more than 15000 square decrees of extragalactic sky, both in the visible and in the near infrared. The focal plane is composed by 6×6 full frame charge-coupled devices (CCDs) of 4096×4096 and a data rate of around 1.08 Terabits per day, with a single frame taking more than 10 Gbits. However, the rate should not exceed the value of 520 Gbits per day, which will be achieved by applying lossless compression with a ratio greater than 2.8 [30].

Other examples which demonstrate that on-board compression is becoming increasingly important, are the SPOT and Pleiades series of satellites developed at the Centre National d'Etudes Spatiales (CNES). While the first satellites launched by CNES had a spatial resolution of 10m, SPOT-5, launched in 2002 had a resolution of 2.5m to 5m; and future missions are expected to reach resolutions below 50cm. The enhancement in resolution cannot be compensated with the evolution in telemetry equipments. For example, SPOT-4 only used one channel of 50Mbps, whereas two channels with the same capacity are used for SPOT-5. Pleiades-HR was designed with three channels of 155Mbps each. In the future, it is expected to reach an overall capacity of 2Gbps by improving coding, modulation and transmission efficiency. If we take SPOT-4 as an example, we observe that the improve of

TABLE 1.1: Imaging spectrometers on-board HyspIRI

Instrument	**Type**	**Bands**	**Spectral range**	**Spatial resolution**	**Swath**	**Repeat cycle**
VSWIR	PB	214	380 nm - 2510 nm	60 m	145 km	19 days
TIR	WB	8	7.3 μm - 12.1 μm	60 m	600 km	5 days

downlink capacity can be estimated of a factor of 40, while the pixel density will have grown by a factor of around 1000. Therefore, efficient compression techniques will become an essential element of on-board processing units.

The HyspIRI mission was scheduled by NASA to be launched in the timeframe 2013-2016 [31]. Its goal is to detect ecosystem responses to climate change and human land management. Two imaging spectometers are carried by the mission, the visible shortwave infrared (VSWIR) hyperspectral imaging spectrometer and the thermal infrared (TIR) multispectral imager. The details of the imagers can be seen in Table 1.1.

The HyspIRI mission is considered unique due to the amount of data being collected, stored and distributed, with a data rate of the order of hundreds of gigabytes per day and a maximum sample rate of approximately 70 Mega-samples per second. If we take into consideration that each sample is represented with 14 bits, the resulting data stream is considerable. The downlink bandwidth and on-board processing capabilities (including data compression) will determine the amount of data that can be collected and analysed by the HyspIRI mission. The mission will collect approximately five terabits per day, which will make nearly two petabytes over the three year mission life. The spacecraft is also limited by the on-board storage, which must be able to store hundreds of gigabytes until the next ground station is in range. The on-board data storage of HyspIRI is limited to 1.6 terabits, i.e. 200 gigabytes. As a consequence, the mission utilizes hardware implementations of data processing and compression methods in order to reduce the

data stream coming from the instruments. HyspIRI performs lossless data compression, which must be able to compress up to 70 Msamples/second in real time. The compression algorithm selected for the HysPIRI mission is the Fast-Lossless (FL) algorithm, developed at the Jet Propulsion Laboratory (JPL), which achieves a compression ratio of 3.1 and was implemented on an field-programmable gate array (FPGA) with a resulting throughput of 33 Msamples/second at a system clock rate of 33 MHz [32]. The reduction and compression techniques will reduce the data volume from 6.43 petabytes to 1.67 petabytes over the 3-year mission and from 6455 gigabytes to 637 gigabytes per day.

1.3.2 Limitations and difficulties of the on-board hardware

Besides the selection of an appropriate algorithm for on-board hyperspectral data compression, it is crucial to choose adequately a hardware technology to physically implement the selected algorithm. Compression is required to provide a high reduction of the data volume with minimum consumption of the on-board resources, i.e. low hardware occupancy and low power; and it has to be robust enough in order to minimize the propagation of errors in the data.

Data compression can be implemented in hardware or software. The implementation platform is selected based on a trade-off of several factors, which include: compression performance; lossless versus lossy compression; hardware resources requirements; impact of system complexity; impact on reliability and implementation cost. It has to be noted that there is always a trade-off between the effectiveness of the algorithm and the complexity of its implementation, i.e. the more compression is achieved, the more the resources needed by the hardware implementation. The algorithm must have low computational and memory requirements, with a number of operations

per sample of the order of 10 and a reduced amount of multiplications, which take a long number of clock cycles to be executed.

Hyperspectral on-board compression algorithms are usually implemented in hardware rather than in software. The motivation for this is speed, since hardware compression may perform as much as five times faster than software compression for data of this type. Hardware implementations of on-board compression algorithms are particularly challenging, due to the existing limitations of the on-board hardware, which has to meet specific requirements like tolerance to solar radiation and low power consumption. All this makes the on-board processing devices have a lower performance and a higher cost than the available commercial solutions which can be found on the ground segment.

Normally, on-board compression algorithms are implemented on FPGAs, application-specific integrated circuits (ASICs) or digital signal processors (DSPs). Among them, FPGAs stand out because they present multiple advantages such as the ability to apply parallel processing to increase throughput, provide flexibility to adapt the designs to successive upgrades of compression algorithms as well as scalability and data integrity features. One of the most important features of FPGAs when used for on-board processing is the fact that they can be reconfigured, what will make them become even more important for space missions in the near future. As science goals change or as spacecraft capabilities are limited, the ability of an FPGA to be reprogrammed from Earth allows for the functional evolution of hardware through the life of the mission. A new compression algorithm with higher performance could be implemented on the satellite in the future with relative ease.

Besides the aforementioned hardware platforms, graphics processing units (GPUs) have become popular recently, and widely used for data-intensive

TABLE 1.2: Missions implementing on-board data compression

Instrument	**Satellite**	**Launched**	**Status**	**Compression algorithm**	**Payload data processing**
COIS	NEMO	Planned 2000	Never flown	ORASIS (Compression ratio > 10)	Array of DSP (2.5 GFLOPS)
HERO	NDA	NDA	Planned	Near-lossless vector quantization	Xilinx Virtex FPGA
Hyperion	EO-1	2001	Running	Rice coding	RISC processor 12 MHz Mongoose 5
HRS	SPOT-5	2002	Running	DCT-based	Marconi MDC31750
OMEGA	Mars Express	2003	Running	Wavelet-based	TSC12020
HySI-T	IMS-1TWsat	2008	Running	JPEG2000	NDA
HiRI	Pleiades-HR	2012	Running	Wavelet-based (Compression ratio $4-7$)	NDA
OLI	Landsat-8	2013	Running	Optional lossless	ASIC
HysPIRI	NDA	Planned 2013	Mission concept	Fast Lossless	Xilinx Virtex FPGA

computation, making it possible to achieve a substantial acceleration of algorithms taking advantage of their parallel multiprocessors. These devices are at the moment not qualified to operate in space, however they represent a promising alternative. Many hyperspectral compression algorithms allow for a fair amount of parallelism, therefore it is possible to obtain significantly short processing times when parallelized and executed on GPUs.

Table 1.2 shows some of the current and planned missions which implement multispectral or hyperspectral on-board data compression.

NDA stands for 'No Data Available'.

1.4 Research goals

As it has been explained, future remote sensing missions will have to implement on-board compression to allow capturing data with a high data rate and meet the storage and downlink bandwidth limitations at the same time. Efficient compression relies on both, compression algorithms and the on-board physical implementation of the algorithms. The main goal of this Thesis is to propose new technological solutions for the physical implementation and execution of hyperspectral compression algorithms on-board satellites. The results of this research work are expected to improve the state-of-the-art benchmarks of the current available hardware implementations, as well as set the direction for future research work on this topic, which will lead to further developments. The principal objectives of this research work are summarized next.

- New solutions for accelerating algorithms on several technologies, namely GPUs and FPGAs are proposed. Two different algorithms of the state-of-the-art are utilized as subjects of study, due to their particular interest for future space missions: the lossy compression algorithm for Exomars (LCE), developed under a collaboration between ESA and Politecnico di Torino [33]; and the standard for lossless multispectral and hyperspectral compression, recently issued by the CCSDS [25].
- A characterization of the aforementioned algorithms is provided, as well as an analysis of the architectural options for their implementation.
- The improvements obtained when accelerating the compression algorithms on GPUs and FPGAs are assessed and compared in terms of the computational performance, the cost of the solution and the flexibility of the obtained implementation to adapt to future changes or improvements.

- The validity of the proposed solutions is demonstrated by showing the correctness of the algorithm's execution on the different platforms, comparing the results with those obtained with golden reference software implementations.

Further goals are achieved part of the progress of the work plan. These are detailed next.

- Different digital hardware design methodologies are utilized and compared, including high-level synthesis solutions.
- Accelerating algorithms by means of parallelization implies solving bottlenecks and data dependencies which can reduce the parallelization capabilities. The common difficulties which appear when trying to accelerate hyperspectral image compression algorithms are identified.
- Solutions are given in order to solve data dependencies and allow for more parallelization of the algorithms. When possible, the parallelization strategies are given in a general way, so that they can be applied for other algorithms of the same kind.

All these contributions are expected to be useful to reduce the cost and improve the performance of future satellite missions in which hyperspectral on-board data compression will play a critical role.

1.5 Organization of this document

The present document is structured in six chapters, including this introductory one, which is dedicated to present the main motivations and goals of this Thesis work. The rest of the chapters are briefly described next.

Chapter 2: On-board hyperspectral image compression algorithms and hardware implementations

This chapter presents a review of the state-of-the-art in the field of algorithms for hyperspectral data compression on-board satellites and their hardware implementations. The main requirements of the algorithms and the on-board hardware are described and the different solutions proposed in the literature are analysed, with the objective of contextualizing this research work and setting quantitative goals.

Chapter 3: Implementation of a lossy compression algorithm for hyperspectral images on a GPU

This chapter describes the GPU implementation of a lossy compression algorithm for hyperspectral images, showing the acceleration that can be potentially obtained.

Chapter 4: Implementation of a lossy compression algorithm for hyperspectral images on an FPGA

This chapter shows the implementation on a FPGA of a lossy compression algorithm for hyperspectral images. The resulting experimental results are useful to evaluate how well the algorithm performs on an FPGA and furthermore make it possible to perform a comparison with the GPU implementation of the same algorithm developed in Chapter 3.

Chapter 5: Implementation of the CCSDS standard for lossless hyperspectral image compression on a space-qualified FPGA

An implementation of the CCSDS standard for lossless hyperspectral image compression is performed on a space-qualified FPGA, which is currently being used in space-applications. Several architectural options are explored for the implementation of the algorithm, selecting the one that provides the best trade-off between complexity, power consumption and flexibility of the

implementation. Experimental results are provided and a comparison with the other implementations studied in this Thesis as well as implementations of the state-of-the-art is given.

Chapter 6: Conclusions

Finally, the collection of contributions of this Thesis are summarized and further research works are proposed.

Chapter 2

On-board hyperspectral image compression algorithms and hardware implementations

This Chapter summarizes the main characteristics of the algorithms for satellite data compression, focusing on those specifically designed for on-board hyperspectral image compression. An overview of the particular requirements of on-board compression is provided, as well as a detailed description of the most relevant algorithms of the state-of-the-art, based on their theoretical basis. Furthermore, the physical implementations and performance figures of several hyperspectral compression algorithms on different hardware technologies, including space-qualified devices, are presented.

2.1 Outline

An efficient compression of hyperspectral images on-board satellites is mandatory in order to save bandwidth and storage space. Hence, it has become a very popular research topic for academia and the space industry. Reducing the data volume in a harsh environment like space, where the computational power is limited, is also a challenge which has been faced with a twofold approach: to propose new algorithms, specifically designed to take advantage of the nature of hyperspectral images; and to present the technologies and strategies to execute the compression in the hardware available on a satellite, minimizing the complexity, and consequently the resource usage and power consumption. In order to make it easier for the space industry to implement on-board compression on their satellites, an effort has been made to develop standard algorithms. In this sense, the Consultative Committee for Space Data Systems (CCSDS), a consortium of the major space agencies in the world, has issued three recommended standards for space data compression: a universal lossless compression solution [24], a lossless to lossy two-dimensional (2D) image compressor [34] and a lossless compression algorithm for multispectral and hyperspectral images [25].

Both, lossless and lossy techniques for hyperspectral image compression can be found in the literature. As any other state-of-the-art compression algorithm, they utilize the redundancies in the image samples to reduce the data volume. Two different approaches are feasible in the case of hyperspectral image compression, either 2D coding, which only takes advantage of the spatial redundancies among neighbouring pixels, or three-dimensional (3D) coding, which also exploits the existing redundancies between bands. Among the 2D approaches it is possible to find algorithms like LOCO-I [35] or 2D-CALIC [36]. However, since most remote-sensing images have a large number of spectral components (hundreds of bands in the case of hyperspectral images), taking into consideration the third spectral dimension has been proven

to increase the compression performance of the algorithms, achieving higher compression ratios. Examples of these are the 3D extensions of the aforementioned algorithms, LCL-3D [37] and 3D-CALIC [38]; and the algorithms presented in Section 2.2.

Compression algorithms are inherently computationally demanding, and those designed for hyperspectral images are not an exception. When they are to be executed on a satellite, the specific requirements of on-board processing have to be considered. The available processing power is limited, and most usual data compression technological solutions used on ground cannot be applied to space data systems. Therefore, together with the algorithms, different hardware implementations have been proposed for the compression to be executed on-board a satellite. The most usual approach is to implement a demonstrator of the algorithms in software, which is executed on a general-purpose single-threaded central processing unit (CPU). However, the processors available for on-board usage are not powerful enough to accomplish the compression in an efficient way. Other solutions, more suited for on-board compression, include highly-customized ASICs and implementations on reconfigurable FPGAs. Lately, GPUs have been shown as a promising alternative, although they cannot be used for on-board compression nowadays. A discussion about the current state-of-the-art hardware implementations for hyperspectral image compression is presented in Section 2.3.

2.2 Algorithms for on-board hyperspectral image compression

Typically, a hyperspectral compression algorithm consists of a spatial and/or spectral decorrelator, a quantization stage and an entropy coder, which explores the probability of the symbols to assign short codewords to the most

probable symbols and long codewords to the less probable ones. The decorrelator can be transform-based or prediction-based. In the former approach, a transform like the Discrete Wavelet Transform (DWT), Kahrunen-Loève Transform (KLT), or Principal Component Analysis (PCA) is utilized to decorrelate the data, whereas in the latter, the samples are predicted from neighbouring (in the spectral or spatial direction) samples, and the prediction errors are encoded.

Lossless algorithms have been traditionally preferred, to preserve all the information present in the hyperspectral cube for scientific purposes. However, the performance in terms of compression ratio of the lossless techniques is limited, usually showing a data volume reduction from half up to one third the size of the original image, at most. The necessity for obtaining higher compression ratios will become more critical in the near future, as the data-rate of the next-generation sensors is expected to increase. Therefore, lossy techniques have been also proposed. Whenever lossy techniques are employed, it is necessary to evaluate the impact of the losses in the reconstructed data. Usually the relationship between the peak signal-to-noise ratio (PSNR) and the compression ratio, commonly named rate-distortion (RD) ratio, is used as a metric, together with the maximum absolute error (MAE) and the mean square-root error (MSRE). The aforementioned metrics are defined in the following equations, where $s_{z,y,x}$ represents a sample located in coordinates (x, y) and band z and $\hat{s}_{z,y,x}$ is the corresponding reconstructed sample. Ny, Nx and Nz are the number of lines, columns and bands respectively.

- Compression ratio in bits per pixel per band

$$bpppb = \frac{Size\ of\ compressed\ image\ (bits)}{Nz \times Ny \times Nz} \tag{2.1}$$

- Maximum absolute error (MAE)

$$\text{MAE} = \max\left(|s_{z,y,x} - \hat{s}_{z,y,x}|\right) \tag{2.2}$$

- Mean-squared error (MSE)

$$\text{MSE} = \frac{\sum |s_{z,y,x} - \hat{s}_{z,y,x}|}{Nz \times Ny \times Nx} \tag{2.3}$$

- Peak signal to noise ratio (PSNR)

$$\text{PSNR} = \frac{10 \log(2^{15} - 1)^2}{MSE} \tag{2.4}$$

Nevertheless, it has been observed that a high PSNR does not necessarily yield higher quality in the reconstructed hyperspectral images when they are used in specific applications [39]. Hence, other application-oriented assessments of the impact of the losses have been reported in the literature, where the experiments aim at demonstrating how useful the reconstructed hyperspectral images are at the post-processing stage for particular purposes, e.g. classification, endmember extraction or anomaly detection [40–43]. It is observed that the compression techniques might introduce artifacts which have a little impact in the PSNR but can significantly bias the analysis results of the decompressed image. Lossy compression can also produce a low-pass filtering of the image, which might ease the extraction of the endmembers from the reconstructed data. For instance, in [20] the H.264/AVC video coding standard is utilized to compress hyperspectral images, taking advantage of the fact that both, video and hyperspectral images, constitute 3D data. Despite the high compression ratios achieved, it was demonstrated that a very accurate endmember extraction from the decompressed data was still possible.

2.2.1 Requirements and limitations of an on-board hyperspectral image compression algorithm

The hyperspectral image compression algorithms need to meet several requirements to be amenable to operate on the hardware available on a satellite, specifically:

- The complexity of the compression algorithm has to be low. Regardless of the technology where the algorithm is finally implemented, the computational power of the space-qualified hardware used nowadays on satellites is much lower than that of any personal computer or workstation used on ground. Although the radiation-hardened space-qualified components have the same functionality of an equivalent standard processor, they are designed to be insensitive to ionisation and hence are more expensive to design and manufacture. As a result, the available devices do not have the state-of-the-art computational capabilities. Moreover, it is advantageous that the algorithm can be parallelized in order to speed up the compression process for high data-rate sensors; and must use the available resources effectively, possibly not needing an external memory.

- It is desirable that the algorithm is resilient to errors. Errors can take place during on-board compression because of the effects of radiation, e.g. bit flips in the on-board memory, and while transmitting the data to the ground station. A corrupted packet will prevent the decoding of other packets that depend on it, causing significant error propagation. Traditional compression algorithms cannot recover from a single bit error, causing a wrong decoding of the remainder of the compressed file after the error. Error-resilience aims at limiting error propagation at the cost of losing compression performance, by using error-resilient

entropy codes or by partitioning the data in units that are coded independently, in such a way that an error in one unit will not prevent from decoding other units.

- The compression algorithm has to be able to handle raw data. Nevertheless, most of the times the algorithms are only tested on calibrated data. The significance of these results is bounded by the fact that raw data generated on-board are known to have quite different characteristics than calibrated data. For instance, an algorithm for hyperspectral image compression based on look-up tables known as LUT [44] showed remarkable high compression ratios when applied to old calibrated AVIRIS data, because it exploited artificial regularities introduced in the conversion of raw data values to radiance units. These methods did not work that well on raw or newer calibrated data.

2.2.2 Transform-based compression algorithms for hyperspectral images

A transform-based compression method applied to 2D data consists of transforming the spatial information to another domain, in such a way that the data are decorrelated. The most popular transforms used in compression are the DWT and the Discrete Cosine Transform (DCT). The transform is followed by the quantization and encoding of the resulting coefficients. Wavelet-based compression techniques are particularly interesting and have shown excellent rate-distortion performance for traditional 2D imagery. Although transform-based methods are mostly popular for lossy compression, reversible transforms allow lossless compression likewise. Examples of popular transform-based compression methods are the JPEG2000 standard [45], and set partitioning methods, such as set partitioning in hierarchical trees (SPIHT) and its 2D and 3D variations (SPIHT-2D, SPIHT-3D, SPECK).

The algorithms for hyperspectral image compression which are based on transforms are usually extensions of 2D compression techniques. 3D compression involves coupling a spatial transform with a transform in the spectral direction. A popular approach is to apply a one-dimensional spectral decorrelator, such as KLT [46, 47] or the DWT [48] followed by JPEG2000. The latter serves as spatial decorrelator, rate allocator and entropy coder.

Several 3D transform coding techniques for lossy hyperspectral image compression are compared in [39], showing increased rate-distortion performance when a spectral KLT is employed, especially in the low bit-rate region. However, the complexity of the KLT is rather high, due to the need to estimate covariances matrices, solve eigenvector problems and computing matrix-vector products. Hence, a low-complexity version of the KLT is also presented, [39], showing a minor performance loss with respect to the full KLT and a 20 to 100 times less complexity. In [49] a fault-tolerant implementation with reduced complexity is presented, introducing a error detection and correction (EDAC) method for the matrix factorization operation of the integer KLT transform.

In [50], PCA is used as a spectral decorrelator before JPEG2000 to perform lossy compression of hyperspectral images. PCA transforms a set of correlated data into linearly uncorrelated variables, known as principal components (PCs). The performance of the proposed approach, named PCA+JPEG2000 is addressed in terms of the usefulness of the decompressed data for detection and classification and rate-distortion (RD) performance. The results are compared with an approach in which the DWT is employed before JPEG2000, showing increased performance for PCA+JPEG2000. Furthermore, PCA produces dimensionality reduction in the spectral domain, and it is equivalent to the KLT when all PCs are retained. Remarkably, in [50] it is observed that the best rate-distortion performance occurs when

significantly less PCs than the total number of spectral components are retained, because the minor PCs are mostly noise. The main drawbacks of using PCA for compression is that computing the transform is computationally intensive, together with the fact that the transform matrix has to be communicated to the decoder, what produces an overhead which can affect the compression performance at low bit-rates. A method to reduce this overhead is presented in [51].

Another application of transform-based methods for hyperspectral image compression is presented in [52], where 2D-DWT is employed in each spectral band, followed by Tucker Decomposition (TD) of the transform coefficients to increase the compression ratio. The results exhibit better rate-distortion performance when compared with PCA+JPEG2000, especially at bit-rates lower than 0.1 bpppb, and better classification accuracy.

ICER [53] is a wavelet-based 2D image compressor which is currently being used on-board the Mars Exploration Rovers for compression of a large majority of the images returned [54]. The 3D extension of ICER, ICER-3D uses a 3D wavelet decomposition to provide decorrelation in the spectral dimension as well as both spatial dimensions. ICER-3D is progressive, i.e. the compressed information is organized so that as more of the compressed data stream is received, reconstructions with successively higher overall image quality can be reproduced. ICER-3D uses reversible wavelet transforms in such a way that it can provide lossless or lossy compression. To limit the effects of data loss during transmission, the wavelet-transformed data are partitioned into a user-selectable number of segments which are compressed independently. The compression ratio results show that ICER-3D gives effective compression, but is outperformed by the much simpler Fast-Lossless (FL) compressor based on prediction [55], which is described with more details in Section 2.2.3.

2.2.3 Prediction-based compression algorithms for hyperspectral images

Although transform-based methods provide an efficient compression, they are better suited for the compression of 2D images than for hyperspectral cubes, since extending the methods to perform an additional transform in the third dimension requires significant computational resources, which are not available on-board.

The predictive coding paradigm represents an alternative that achieves a good balance between performance and complexity. It consists of predicting the value of a pixel from past data, generally neighbouring pixels in the spatial (intra-band prediction) and/or spectral (inter-band prediction) dimension, subtract the predicted value from the actual sample, quantize the resulting prediction error and entropy code it. This is a form of differential pulse code modulation (DPCM). Figure 2.1 shows the principle of a prediction scheme where the previously processed sample in the spatial dimension is used for prediction.

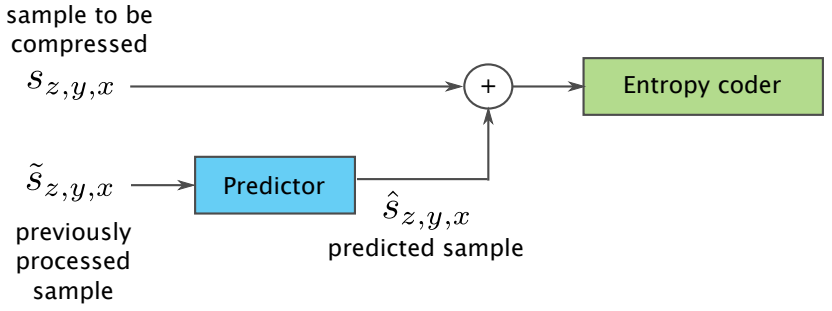


FIGURE 2.1: Prediction-based compression scheme.

While lossy compression is more efficiently performed by transform-based methods at the cost of increased complexity; prediction is mostly preferred

for lossless compression. Nevertheless, near-lossless or lossy compression can be achieved by a predictive coding algorithm by means of selecting an appropriate quantization method [33, 56]. The prediction plus entropy coding paradigm is amenable to low complexity. Traditionally, on-board compression algorithms avoid using arithmetic coding, as it is considered a relatively complex coding scheme. Instead, Golomb-power-of-two codes [57] are the preferred choice, because they achieve a good balance between performance and complexity.

Several methods based on prediction can be found in the literature. Among the lossless techniques, the FL algorithm [55] uses low-complexity adaptive filtering for predictive compression of hyperspectral images, using the sign algorithm [58]. The samples are predicted using only causal information, i.e. the part of the image which has been already processed. In particular, it utilizes the samples in the current band ans well as in the three previous bands and adapts the predictor coefficients using recursive estimation. This algorithm is particularly interesting, since it was selected by the CCSDS to be standardized and become the CCSDS 123 recommendation for multispectral and hyperspectral compression [25].

A method based in Context-based, Adaptive, Lossless Image Codec (CALIC), named 3D-CALIC, is presented in [38]. 3D-CALIC switches between intra-band and inter-band compression mode depending on the strength of the correlation between two consecutive bands. In [59], a modification of 3D-CALIC is presented, in which only inter-band prediction is utilized, specifically, the prediction is performed using two samples in the previous band in the same spatial position of the current sample.

Furthermore, clustered DPCM [60] partitions the spectral vectors into clusters and then applies a separate least-squares optimized linear predictor to each cluster of each band. An adaptive least squares optimized prediction technique called Spectrum-oriented Least Squares (SLSQ) can be found in

[61]. The predictor is optimized for each sample and each band in a causal neighbourhood of the current samples.

Other predictive algorithms, are based on lookup-tables (LUTs) [44], searching the previous band for a sample of equal value to the sample co-located to the one to be coded. The sample in the same position as the obtained sample is used as the predictor. LAIS-QLUT performs also quantization of the co-located samples. A generalization of the LUT method to multiband and multi-LUT is proposed where the prediction of the current band relies on N previous bands. It was demonstrated that LUT-based methods exploit artifacts that are sometimes introduced by the calibration process, making them less appealing for on-board use, where those artifacts are not likely to occur [44].

Prediction-based algorithms can achieve an improved compression performance when band reordering is applied, as has been demonstrated in the literature [62–64]. In the latter, the spectral channels of the image are reordered to maximize the correlation of adjacent bands. However, searching for the most correlated band given a specific one is computationally demanding, and would significantly increase the hardware complexity if it is performed on-board.

In order to allow parallelization and reduce the impact of errors, some algorithms partition the image in squared independent blocks and compress one block at a time. For instance, the BH algorithm [65] employs a simple block-based predictor. It first predicts the block from the corresponding block in the previous band, then selects a predesigned code based on the prediction errors and finally it encodes the predictor coefficient and errors.

In [56] a very simple lossless to near-lossless compression algorithm is described, which is based on block-by-block prediction and predictive Golomb coding. The proposed algorithm can exploit optimal band reordering, and

can be extended to near-lossless compression by means of a uniform scalar quantizer. In order to avoid the complex operations that band reordering involves, the authors of [56] propose to perform the band reordering at the ground station based on sample data and then upload it to the satellite for the compression of the captured images. The motivation for that is tha the optimal ordering depends on both the sensor and the scene, with the former potentially dominating the ordering. Experiments on AVIRIS data show better compression than LUT, almost as good as FL and similar to 3D-CALIC. Band reordering shows a small improvement of 1% in the compression ratio. When near-lossless compression is performed, it is demonstrated that the algorithm, although being block-based, does not produce blocking artifacts in the reconstructed images.

As it has been already stated, prediction methods are preferred for lossless compression. However, they can be also successfully applied for low-complexity lossy compression. In [33], a block-based lossy compression scheme is proposed, based on prediction, uniform-threshold quantization and rate-distortion optimization. The experimental results demonstrate a performance competitive with the state-of-the-art transform coding techniques, but with significantly lower complexity. Since it is block-based, it is able to limit the scope of errors and it is amenable for a parallel implementation. All this makes it a good candidate for on-board compression at high throughputs. This algorithm was developed under an ESA project in the framework of the Exomars mission [66].

2.2.4 Recent research on hyperspectral image compression algorithms

Recent work has applied ideas from distributed source coding to construct extremely simple and error-resilient algorithms [67]. Distributed source coding techniques consider a situation in which two or more statistically dependent information sources must be encoded by separate encoders which do not share any information. The theory proves that, under certain conditions, separate coding is optimal, provided that the sources are decoded jointly. When applied to hyperspectral images, the previous band is used for the prediction of the current band. The first band is transmitted uncompressed, while for all others, the prediction parameters are not sent to the decoder. Instead, the decoder reconstructs the pixels by guessing them, and computing a cyclic redundancy check (CRC). Once the CRC matches the one included in the compressed file, the process terminates. This approach provides furthermore error resilience, since an error in the transmitted compressed data does not necessarily yield an erroneous reconstruction. The proposed algorithm shows a competitive complexity when compared with the state-of-the-art, adding the advantage of error resilience features. However, an appropriate trade-off is yet to be found between robustness, complexity and compression performance. The algorithms based on distributed source coding can achieve a compression performance higher than the state-of-the-art 2D prediction algorithms and slightly lower than other 3D prediction-based algorithms, which is the price to be paid for error resilience.

Other recent studies are focused on compressed sensing techniques [68], which suggest that a signal, supposed to be sparse, can be perfectly reconstructed from a limited, i.e. fewer than Shannon, number of incoherent measurements. These techniques could indeed simplify the process of hyperspectral image acquisition [69], providing a reduced number of measurements directly produced by the sensor, saving an important amount of resources.

In fact, preliminary results have demonstrated that the amount of measurements needed to represent a hyperspectral image can be reduced by a factor of up to 10 [70]. Nevertheless, the design of a sensor able to produce these measurements is difficult, and a lot of technological developments are still needed in order to leverage the full potential of this know-how for hyperspectral imaging [71].

2.2.5 CCSDS Standard algorithms for satellite data compression

The importance of an efficient data compression in space missions is further evidenced by the fact that the CCSDS has issued several standards which facilitate for the different the space agencies and industries to exploit the benefits of compression, by making high quality documentation available and helping to establish a broad user community.

The first released compression standard for space applications is known as CCSDS 121 [24] and is a universal lossless data compressor consisting of a preprocessor and an entropy coder based on Rice coding [72]. The objective of the preprocessor is to change the statistics of the data by applying a reversible function, hence reducing the entropy. The recommendation does not strictly specify the preprocessing stage, which can be determined by the final user according the specific characteristics of the target data. The subsequent stage consists of a mapper followed by an entropy coder which operates on blocks of J samples. It incorporates multiple coding options, based on Golomb power-of-two codes, which are applied concurrently to a J-samples block. Furthermore, it includes a zero-block and a no compression option, as well as a low entropy option known as second extension. The algorithm option that yields the shortest encoded length is selected for transmission.

The CCSDS 121 features very low complexity, however its performance decreases significantly with the presence of outliers or when the data do not follow any well-defined statistics. With the motivation of overcoming this difficulties, some alternatives to the CCSDS 121 can be found in the literature. In [73] a Fully Adaptive Prediction Error Coder (FAPEC) is presented, together with its software and hardware implementations. FAPEC shows increased compression ratios when compared with CCSDS 121, and a complexity that is amenable for an on-board implementation. More details about the hardware implementation of FAPEC can be found in Section 2.3.

The Image Data Compression recommendation [34], CCSDS 122, describes a compression technique which can be used to produce both lossy and lossless compression of 2D satellite images. It consists of a DWT module that performs decorrelation and a bit plane encoder (BPE). The DWT module employs a three-level 2D-DWT, by repeatedly applying a one-dimensional DWT. It is possible to choose between a float DWT or an integer approximation to this transform. The output coefficients are converted to integer values before applying the BPE, which represents each value with a binary word consisting of a single sign bit along with several magnitude bits depending on the bit width of the input image data. The CCSDS 122 is similar to JPEG2000, however it has a a reduced performance which allows for lower complexity and hence low-power hardware implementations.

Finally, the CCSDS 123 recommendation [25] defines a payload lossless data compressor that can be applied to multispectral and hyperspectral imagers and sounders. The compressor consists of a predictor and an entropy coder and is based on the FL algorithm [55]. The predictor uses an adaptive linear prediction method to predict the value of each image sample based on the values of nearby samples in a small three dimensional neighbourhood. The residual of the prediction is mapped to an unsigned integer value and encoded with an entropy coder whose parameters are adaptively adjusted to adapt to

changes in the statistics of the mapped prediction residuals. The standard offers the alternative of using the block-based entropy coder defined in the lossless data compression standard, CCSDS 121. Experimental results in terms of compression ratio for a real hyperspectral and multispectral image corpus show that the CCSDS 123 standard is competitive with other state-of-the-art algorithms, providing the best trade-off between coding performance and computational complexity [74].

FIGURE 2.2: CCSDS 123 Recommendation for lossless multi- and hyperspectral image compression.

2.3 Physical implementations for on-board compression of hyperspectral images

Several algorithms with different complexity and performance features for on-board compression have been proposed in the literature. Nevertheless, it

is also necessary to provide physical implementations which serve to demonstrate that the algorithms are suited for on-board compression, and that their performance will be maintained when implemented in the on-board hardware.

Obtaining a physical implementation is a difficult and time-consuming task, therefore not all the studies about hyperspectral image compression algorithms include them. Many times only the theoretical basis is explained, and the experimental results are obtained with high-level software tools, such as MATLAB. In general, the different implementations found in the literature can be classified based on their technology as follows:

- *Software implementations:*
  - Implementations on general-purpose CPUs. Usually a high-level programming language is used to produce a software which can be executed on any general-purpose CPU. These kinds of implementations are really flexible and can be obtained in a relatively short time at a low cost. However, the low throughput presented by the on-board processors when performing hyperspectral image compression makes them inadequate to perform hyperspectral image compression on-board a satellite, as it will be further explained in Section 3.3.
  - Implementations on DSPs. These devices are specifically designed to perform signal processing in an efficient way and are therefore an interesting option for hyperspectral image compression, due to the amount of mathematical operations demanded by the algorithms, mainly in those based on transforms.
  - Implementations on GPUs. These devices have recently become popular for general-purpose computing, employing massively parallel processing to achieve high throughput. Nevertheless, they are

not space-qualified and their usage on-board satellites is subtle to future technical developments which would reduce their power consumption and make them insensitive to solar radiation.

- *Hardware implementations:*
  - Implementations on ASICs. These highly-customized implementations achieve high throughputs and low power at the cost of relatively high design times, more expensive manufacturing and lack of flexibility.
  - Implementations on FPGAs. Represent a trade-off between customization and cost, making it possible to obtain high throughputs and low power consumption.

The remaining of this Chapter gives more details about the aforementioned technologies and the most significant hardware implementations of several of the algorithms described in Section 2.2. It has to be noted that not all the presented implementations correspond to 3D hyperspectral compression algorithms. FPGA implementations of universal satellite data compression algorithms or 2D image compressors are also presented, since they are considered relevant for the state-of-the-art and they can anyhow be employed for hyperspectral image compression with a reduced compression efficiency, as it was shown in Section 2.2.

2.3.1 On-board hardware technology requirements

In general, any hardware implementation, regardless of the technology, is desired to be small in area, have low power consumption and achieve a high throughput. In this sense, the implementation of algorithms for on-board hyperspectral image compression are no exception. However, the specific

characteristics of the space environment makes it mandatory for the hardware operating on a satellite to meet additional requirements, mainly tolerance to solar radiation and low power consumption, among others. All this difficulties the on-board device fabrication, increases its cost and ultimately reduces its performance. The most relevant requirements of the on-board hardware are summarized next.

Tolerance to high energy radiation

Hardware operating in space has to be integrated by components and manufactured with materials which can tolerate high energy radiation. The biggest threat to their operation are high energy particles like galactic cosmic rays, solar winds, solar events and radiation belts. High energy particles can cause upsets in complementary metal-oxide semiconductor (CMOS) and field effect transistor (FET) technologies, which are used to manufacture most of nowadays microelectronic devices. A single event upset (SEU) is a change of state caused by ions in a microelectronic device, producing an error in the device output or even a permanent damage which can destroy the device.

The relevant effects of the space radiation environment on microelectronics can be divided into two categories: total ionizing dose (TID) effects and single event effects (SEE). The SEEs can be further subdivided into effects that lead to permanent damage (latch-up) and recoverable effects.

Radiation effects that accumulate over time are referred to as total ionizing dose (TID). In normal operating conditions, a voltage applied on te gate of a MOSFET transistor creates an electric field, which reaches into the semiconductor below the gate oxide. This electric field causes the formation of a conducting channel between source and drain. When high energy particles impact on the gate oxide of a transistor, they cause ionisation. Electrons get swept out, leaving behind immobile holes, as it is shown in Figure 2.3. These

positively charged holes in the gate oxide decrease the threshold voltage required for the creation of a conductive channel. If the threshold becomes too low the device is in a permanent "on" state. Space-qualified microelectronic materials are tested for a specific total ionising dose, to ensure they operate as expected in space.

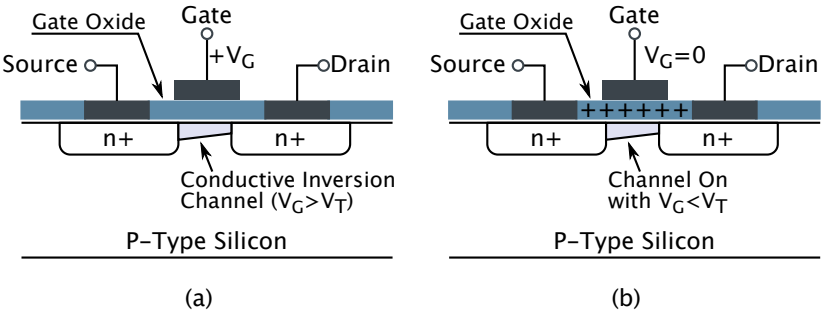


FIGURE 2.3: Radiation effects on a MOSFET transistor.(a) Normal operation. (b) Post irradiation.

A latch-up is the occurrence of a path of low resistivity between the voltage supply and ground connection, which can destroy the device.

On the other hand, protons can origin nuclear reaction with silicon atoms, what produces short range ionisation and can upset a memory cell. When a heavy ion impacts on material, it loses energy which causes ionisation of the atoms in the proximity of the impact trace. When ionisation occurs in a pn-junction of a microelectronic device the charges created are separated due to the pn-junction's electric field. This process creates a charge in the electronic device, what can cause an erroneous transient or interact with the charge of a memory cell. If the charge created exceeds a critical value, the state of the memory cell can be upset.

When microelectronic devices for space are manufactured with a technology susceptible to SEUs, mitigation techniques have to be employed in order to

reduce the system error rate. The typical approach is to apply triple modular redundancy (TMR), which uses three replicas of the same circuit and applies a majority voting strategy to select the correct output among the replicas. The direct consequence of utilizing TMR is an increase in the design area.

Some commercial off-the-shelf (COTS) solutions have been tried in the past to implement on-board compression algorithms. In particular, in [75] an implementation study of JPEG2000 standard is performed with radiation hardened components, exhibiting disappointing results because of the complexity of the algorithm for its implementation in such a hardware. Subsequently, in the same study, a commercial hardware platform implementing JPEG2000 is used to perform a performance analysis and a study of tolerance to the spatial environment. The circuit revealed a high sensitivity to radiation, demonstrated by the fact that not a single image was compressed successfully under heavy ions beams. Although the COTS solution is really efficient to implement JPEG2000, it is really inappropriate for on-board usage. Examples like these illustrate the need for hardware specifically designed to operate on-board a satellite.

Power efficiency

Besides the device being radiation-tolerant, it has to be considered that the available power on-board a satellite is limited and therefore the constraints in terms of power consumption for on-board technologies are much more restrictive than for commercial applications.

High throughput

In the specific case of hyperspectral image compression, taking into account the amount of data to deal with, a high throughput is desired, especially when real-time compression is wanted. Although compression is expected to become a necessity in the future space missions, there is still a lack of viable on-board platforms to perform significant image processing and compression.

2.3.2 Software implementations

A hyperspectral image compression algorithm can be described by a set of instructions suitable to be executed in a computer's processor. Among the existing software implementations, it is possible to find implementations on general-purpose processors, digital signal processors (DSPs), and graphics processing units (GPUs). The main contributions that can be found in the literature are summarized in the following sections. Moreover the advantages and disadvantages of each approach are explained, from the point of view of the on-board hyperspectral image compression needs.

2.3.2.1 Implementations on general-purpose CPUs

Most of the algorithms proposed for hyperspectral image compression have been implemented in software for general-purpose processors. The programming languages utilized vary, being C/C++ and Java the most popular. These software implementations can be compiled and executed on any processor, and most of them are open source. Their purpose is usually to demonstrate how efficient the algorithms are in terms of compression ratio. They are also sometimes utilized to address the complexity of the algorithms, by comparing for instance the execution times of two different algorithms. Although it can be in general inferred that the faster the execution of the algorithm, the lower its complexity, the execution times are also dependent on how the software was programmed -e.g. to make the source code understandable or to maximize the throughput- or the compiler options. Hence, in order to address the complexity of an algorithm other metrics besides the software execution times shall be used, for instance the number of operations, the bit width of the different variables, the precision of the operations or if they are integer or floating-point operations.

Some examples of software implementations of algorithms for hyperspectral image compression are summarized next. In [74] a software implementation of the CCSDS 123 standard is presented. The implementation is open source, developed in Java programming language and utilized to present a review of the state-of-the-art, providing an experimental comparison of the coding performance, and extensive results over the vast corpus of test images from the CCSDS working group. It serves as reference implementation. Recently, ESA made an open source implementation in C language of the same algorithm [76].

Whitedwarf [77] is an application developed by ESA that supports the evaluation of compression algorithms, by enabling the compression and decompression of files and optimization of the algorithm choice and compression parameters. It supports the CCSDS 121 and the CCSDS 122 standards. Additional compression algorithms are expected to be added by ESA in the future, once the related standardization process are completed.

Although software implementations are highly flexible and can be developed at low cost and in a short time, they have limited throughput performance and are power-hungry. Hence, they are usually inefficient for the particular case of on-board hyperspectral image compression. Satellites are generally equipped with an on-board computer, which is able to run software and consists essentially of a microprocessor, non-volatile and volatile memories and interconnection buses. The major microprocessor currently used in most European space applications is the LEON2. The LEON2-FT design is an extension of the basic LEON2 model including advanced fault-tolerance features at design level, in order to to withstand arbitrary SEU errors without loss of data. There are two newer versions of LEON, namely LEON3 (and its fault-tolerant version LEON3-FT) and LEON4, which were designed by Aeroflex Gaisler. LEON3-FT was licensed in 2007 for new space missions

in Europe [78]. A LEON3-FT spacecraft controller chip manufactured by Astrium is shown in Figure 2.4.

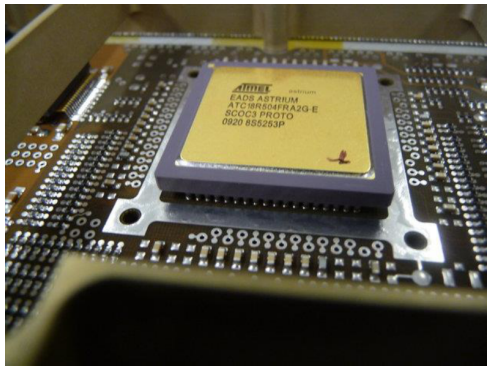


FIGURE 2.4: LEON3 spacecraft controller on a chip.

On-board computers do not provide enough throughput to compress hyperspectral images and are generally loaded with other tasks, such as altitude and orbit control, telecommands execution or dispatching, housekeeping telemetry gathering and formatting, on-board time synchronisation and distribution, failure detection, isolation and recovery, etc. Dedicated hardware solutions are highly desirable, taking off load of the main processor, while providing power efficient solutions at the same time.

2.3.2.2 Implementations on DSPs

Alternatively to a general-purpose processor, it is possible to utilize DSPs to implement hyperspectral compression algorithms. These processors are specialized and optimized for the operational needs of digital signal processing and hence have been commercially used for digital image processing, including image and video compression.

An example of a space-qualified DSP, is manufactured in Europe by Atmel and is known as TSC21020F. It is radiation-tolerant with a typical

performance of 40 Mega-floating-point operations per second (FLOPS) (60 MFLOPS peak) and has already become obsolete. For this reason and with the motivation of reducing the dependency on critical technologies from outside Europe, research has been performed towards the development of a new space-qualified DSP under The Seventh Framework Programme (FP7) project of the European Commission called DSPACE [79]. The need of this DSP has been also stressed by ESA [28] and the developed device is expected to feature a performance superior to ≥ 1 GFLOPS, radiation harness ≥ 100 KRad TID, EDAC memory protection, support for standard interfaces, high reliability and low power consumption.

Several algorithms for 2D image compression on satellites were implemented on DSPs, however few research studies exist where DSPs are used for hyperspectral image compression. An example of the latter is the study presented in [49] where an integer approximation with reduced complexity of the KLT is used for hyperspectral image compression. The proposed algorithm includes an EDAC method which introduces fault-tolerance. An implementation of this algorithm on a multi-core DSP manufactured by Texas Instruments is presented and implemented on an evaluation board TDMSEVM6678L, which includes 8 DSP cores operating at 1.0 GHz and 512 DDR3 memory. It supports Open Multi-Processing (OpenMP), a simple and flexible interface for developing parallel applications on shared memory multiprocessing platforms. For this implementation, the KLT is performed in clusters, i.e. z bands are decorrelated by the transform instead of the total number of bands, Z, being $z < Z$. The total number of clusters is $c = Z/z$. In the multi-core DSP implementation the clusters are executed concurrently, each of the 8 cores encoding an individual cluster. Hyperspectral images from the AVIRIS and HYPERION datasets are used for compression on the platform with several number of clusters. Results show a throughput of 53.4 Mbps for the AVIRIS image at the optimal clustering level with error detection enabled.

Despite being more specialised for digital data processing, and having the flexibility inherent to any other software implementation, DSP implementations suffer from the same problems of CPU implementations when used for on-board hyperspectral image compression. Although the DSPs achieve a higher throughput, it is far from what can be obtained with more customized hardware like ASICs or FPGAs. Moreover, the power consumption for space applications should be lower. In the aforementioned study, the multi-core DSP implementation shows an average power consumption higher than 15 Watts.

2.3.2.3 Implementations on GPUs

GPUs became popular thanks to the video games industry, but they have developed fast, and have allowed their usage for general-purpose computation. They are able to dramatically increase the computational speed of applications my means of massive parallelism and have evolved much faster than CPUs in terms of GFLOPS. GPUs consist of a set of multiprocessors, each composed of a set of simple processing elements working in single instruction multiple data (SMID) mode. Unlike CPUs, where most of the transistors are devoted to control and memory, GPUs devote them on many arrays of small execution units, dispatches, small volumes of shared memory and memory controllers, as shown in Figure 2.5. All of these does not accelerate the execution of separate streams, but allows a GPU to process several thousands of execution threads. Therefore, GPU achieve high speedups in applications which are computationally intensive, rather than control-flow intensive.

Hyperspectral image compression is both computationally and control-flow intensive. Many of the typical processes involved in the compression, e.g. the entropy coder, are mainly sequential and there are strong data dependencies. Nevertheless, hyperspectral image compression involves handling a massive amount of data, and some algorithms allow for a fair amount of data

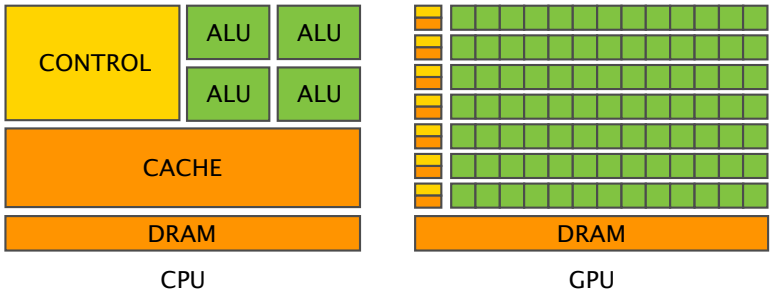


FIGURE 2.5: Simplified CPU and GPU architecture comparison.

parallelism. These algorithms can benefit from GPU acceleration, yielding high throughput at a relatively low-cost. However, space-qualified GPUs do not exist yet, and their power consumption is usually too high for space applications, ranging from around 10 W to above 200 W depending on the particular GPU model and the demands of the application running on it [80].

A parallel implementation of the CCSDS 123 standard for lossless multi- and hyperspectral image compression is presented in [81]. The study provides a fast parallel software implementation which is described and subsequently executed using hardware acceleration on GPUs and multicore processors. The performance of the proposed implementation exceeds that of previous hardware and software versions of the same algorithm. Although the operations performed by the CCSDS 123 algorithm are highly sequential, parallelism is gained by image segmentation coupled with an improved data-flow. The final implementation is executed on a Nvidia GeForce 560M GTX with 1.5 GB of random access memory (RAM) memory. The specifications show a power consumption of 75 W (thermal design power (TDP)). The performance results show a speedup of 10.38× and a throughput of 297.15 Msamples/second, when the algorithm is executed on a system with one GPU. There is a slight, but not significant, improvement when the same algorithm is executed in a system with two GPUs, what demonstrates that the parallel portion of

the algorithm is so fast, that the run time is dominated by the inevitable sequential operations, like file accesses or bus transfers. On the other hand, the multicore implementation shows a maximum speedup of $\times 7.89$ when 4 cores (Intel i7-2760QM at 2.4 GHz and 16 GB of RAM) are utilized. The power consumption of the Intel i7-2760QM processor is 45 W (TDP).

GPUs exhibit promising results when utilized to accelerate hyperspectral image compression algorithms. Nevertheless, their usage on-board satellites is not feasible yet, due to their high power consumption and lack of tolerance to radiation. Research is currently being performed towards the development of low power GPUs, mainly for commercial small technology devices, like phones or tablets. Evidence of this is the EU-funded project named "Low-power GPU" (LPGPU), which aims at enabling a next-generation of advanced graphics technologies for power-efficient devices [82].

2.3.3 Hardware implementations

As an alternative to software, which is executed on general-purpose processors, physical hardware devices can be designed and manufactured to perform the specific task of hyperspectral image compression. In the following, the main characteristics of the hardware design flow are explained. On-board hardware implementations of data compression algorithms can be found in the form of ASICs and programmable FPGAs. Some examples of both approaches are also provided in this Section, with special emphasis in those designed for hyperspectral image compression.

2.3.3.1 Hardware design flow

Digital hardware design involves describing a digital circuit for one particular application. At the first step of the design flow, the circuit specifications are

determined. Afterwards, the behaviour of the circuit is described. This description can be performed at different levels of abstraction. The most commonly utilized is register-transfer level (RTL). At RTL, a synchronous design is modelled in terms of logical operations and the data path between hardware registers. Hardware description languages like Verilog or VHDL are utilized in order to create a high-level representation of a circuit, from which a lower level representation and actual wiring (physical design) can be derived, performing the logic synthesis, placement and routing. Finally, the design is prepared for manufacturing depending on the final implementation technology.

The RTL design flow is rather long, and it involves careful planning and a designer experienced with hardware description languages. Verification is necessary in order to match the behaviour of the design with the specification at various stages of the design flow. The process can be shortened with the help of high-level synthesis tools, which are capable of porting high-level source codes written in programming languages like C/C++ or Matlab to RTL. Examples of these are CatapultC, Impulse C or C-to-Silicon [83]. These tools are useful for rapid prototyping, and present several advantages, e.g. the amount of code to be written by the designers is highly reduced and the tools open up opportunities for extensive design space exploration. However, if a highly optimized implementation is desired, it is mandatory to write the RTL description from scratch.

2.3.3.2 Implementations on ASICs

ASICs are tailored to be optimum for a particular application and therefore can achieve high throughputs and low power when specifically designed for on-board hyperspectral image compression. Most of the 2D compression algorithms which are currently executed on satellites are implemented on ASICs [84].

An example of an ASIC implementation for space is the CWICOM chip, see Figure 2.6, developed by Astrium in the frame of an ESA contract. CWICOM is a high speed 2D image compression ASIC which implements the CCSDS 122 standard for image data compression [34]. The CCSDS 122, as it was explained in Section 2.2.5, consists of a DWT followed by a BPE and can perform lossless as well as lossy compression.

FIGURE 2.6: CWICOM compression ASIC.

CWICOM features high data rate of 60 Msamples/sec, what results in 960 Mbps if the sample values are represented with 16 bits. The power consumption is < 100 mW/Msample/sec. This means that for the maximum throughput of 60 Msamples/sec the consumption would be approximately 3 W. It supports a TID of 100 KRad and is tolerant to SEU thanks to its internal EDAC. One of the main challenges present during the development of a hardware implementation of the CCSDS 122 algorithm is the high amount of memory which is needed to store the DWT coefficients during the processing. An external memory can address this problem, but would decrease the performance. Hence, the CWICOM includes a high amount internal memory cells, and a very efficient internal embedded memory organization, making it possible to compress the images without the need of an external memory.

While ASICs are power and area efficient, they also present some weaknesses. In many occasions, space applications require more flexibility and

scalability for post-launch modifications and repair. ASICs, once manufactured, cannot be configured to efficiently match subsequent mission needs and requirements.

2.3.3.3 Implementations on FPGAs

Alternatively to ASICs, FPGAs offer solutions at a lower cost and with increased flexibility. They consist of arrays of blocks of generic logic cells which can be interconnected in a general way. Both the logic blocks and the interconnection structure are programmable by means of switches which can be set as open or short circuit. FPGAs are manufactured by a number of companies like Microsemi, Altera or Xilinx, among others. The basic scheme and elements of an FPGA are shown in Figure 2.7.

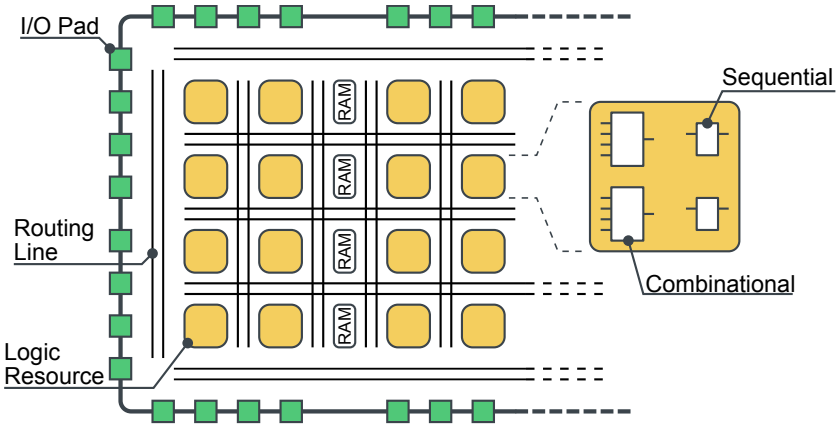


FIGURE 2.7: Basic elements of an FPGA.

FPGAs present several advantages for on board hyperspectral image compression. They are able to apply parallel processing to increase throughput and provide some flexibility at the same time. Moreover, FPGAs offer the

possibility of adapting the designs to successive upgrades of compression algorithms and electronic components over a long term. Moreover, FPGAs provide scalability and data integrity features.

The switches which allow the FPGA programming can be constructed in several ways, including: pass-transistors constrolled by RAM, anti-fuses, EPROM transistors or EEPROM transistors (flash-based). Both static random access memory (SRAM)-based and anti-fuse FPGA are used nowadays in satellites.

Traditionally, anti-fuse technology has been preferred by the space industry, due to its increased robustness against radiation. An anti-fuse normally resides in a high-impedance stage, but can be "fused" into a low-impedance state when programmed by a high voltage. An example of radiation-tolerant anti-fuse FPGAs is the RTAX family from Microsemi. However, anti-fuse FPGAs present an important disadvantage: they are one-time programmable.

On the other hand, in SRAM-based FPGAs the programmable connections are made using pass-transistors, transmission gates or multiplexers that are all controlled by SRAM cells. The major advantage of this technology is that it provides an FPGA that can be reprogrammed many times and very quickly, and can be produced using standard CMOS technology. However, one of their disadvantages it that they have to be programmed again every time the system is powered up, due to the volatile nature of SRAM. SRAM-based FPGAs are becoming increasingly important for new space missions. The lifetimes of the satellites expand far beyond 10 years, which is much longer than the validity of telecom standards. Hence, reprogrammability becomes an important requirement.

SRAM-based FPGA are susceptible to SEUs, which are induced by high-energy particles in the harsh environment of space. This problem is addressed through the use of radiation-tolerant and radiation-hardened technologies as well as SEU mitigation techniques. Unlike ASICs, where only memory elements have to be protected against SEU, SRAM-based FPGAs must implement full triplication of all the design elements due to the high sensitivity of its configuration memory to radiation effects. A single error affecting a TMR circuit is masked and tolerable, however, when multiple independent SEUs hit a TMR circuit, it might occur that the majority voter in the TMR scheme votes for the wrong answer. To mitigate his effect, the configuration bit-stream has to be written periodically back to the FPGA, whith a scrub cycle that must be selected according to the frequency of SEUs occurrence. During this reconfiguration time, which can be of the order of milliseconds, the device is offline, what can be intolerant for systems with hard real-time constraints.

The FPGA manufacturer Xilinx has developed several radiation-hardened models as part of the Virtex IV and Virtex 5 families. The radiation-hardened Xilinx FPGA XQR40662XL on-board FedSat, is the first demonstration of hardware reconfiguration in space [85].

Finally, flash-based FPGAs are similar to SRAM-based FPGAs, but they present the advantage that the configuration memory is non-volatile and therefore they can be programmed off-line. The downside is that flash-based FPGAs tend to have higher static power consumption and longer programming times. An example of radiation-tolerant flash-based FPGAs are the ProASIC3 family, manufactured by Microsemi. This model, however, is not used in any current space mission.

The most relevant contributions to the field of FPGA implementations of satellite data compression algorithms are summarized next.

The FAPEC universal lossless data compressor was implemented on a radiation-hardened RTAX anti-fuse FPGA, from a RTL description in VHDL language [73]. Several considerations are taken into account in order to lower the complexity of FAPEC in such a way that its FPGA implementation is more efficient. For instance, floating-point operations are avoided and the data block size is reduced. These changes are reported to have little or no effect in the performance of the algorithm. The designed compressor is finally implemented in a ProASIC3L developement board which includes a M1A3P1000L FPGA, whose number of equivalent gates is equal to that of the space-qualified anti-fuse RTAX1000S. The implementation shows a critical path of 18.32 ns (maximum frequency 55 MHz), with a throughput of 2 Msamples/sec (32 Mbps if the input samples are 16 bits wide). The hardware occupancy is 12% of combinational logic and 15% of sequential logic and the power consumption is estimated at 35 mW.

Another universal compression algorithm and the corresponding hardware architecture are presented in [86]. The algorithm performs context-based statistical lossless compression of multiple types of data. It takes advantage of FPGAs that support partial and dynamic reconfiguration, which consists of changing a portion of the reconfigurable hardware while the rest is still operating. Their method is based on a dynamically reconfigurable modelling stage followed by statistically configured probability estimation and an arithmetic coding. The dynamic modelling is specialized to each data type and uses a combination of context modelling, predictive coding and motion estimation depending on the data type being processed. The throughput performance of the proposed system is 100 Mbps on a Xilinx Virtex4 SX35 FPGA. Partial dynamic reconfiguration has become a popular research topic in the space industry, since it can enhance space applications with run-time adaptive functionality, enabling mission specific adaptability. However, technical developments are still needed for this technology to become reliable and fault tolerant for its usage on-board satellites [87].

TABLE 2.1: Results of implementing CCSDS 122 on an RTAX2000S FPGA

Resource	**Proba-V**	**EnMap**
Combinational C-Cells	8455 (39%)	9772 (45%)
Sequential R-Cells	7125 (66%)	7620 (71%)
Total Cells	15580 (48%)	17392 (54%)
Block RAMs	54 of 64	58 of 64

An FPGA implementation of the CCSDS 122 standard for 2D satellite image compression can be found in [88]. The study presents the designed hardware architecture and its implementation on a space-qualified FPGAs, specifically on an anti-fuse RTAX2000S. However, the ProASIC3E flash-based FPGA is utilized for prototyping. As it was explained in Section 2.2.5, the CCSDS 122 standard algorithm consists basically of a Discrete Wavelet Transform (DWT) plus a bit plane encoder. The intermediate coefficients of the DWT have to be stored and rearranged during the compression process, what requires more memory than available in the RTAX2000S FPGA. Hence, an external SDRAM memory is used, with an appropriate memory organization to reduce the access overhead. The hardware occupancy and frequency of the implementation depends on the selected configuration parameters of the CCSDS 122 algorithm. The results of two different configurations for two specific space missions, namely ESA Proba-V and the German EnMap are shown in Table 5.23. The maximum frequency is 64 MHz for the Proba-V mission and 50 MHz for the EnMap mission. The throughput is 90 Mbps for Proba-V and 130 Mbps for EnMap.

As it was already mentioned, transform-based methods for hyperspectral image compression require a fair amount of processing memory, are computationally intensive, and not amenable to parallelization due to the sequential nature of the transforms. Hence they are not popular candidates for an FPGA implementation. However, efforts have been made to accelerate

transforms in order to be able to implement them efficiently on FPGAs, as shown in [89], where the KLT is investigated. A comprehensive analysis of the computations needed to calculate the KLT is performed to inspect the feasibility of different acceleration techniques, such as parallelism. The proposed designs are implemented in a FPGA-based System-on-Chip (SoC), which incorporates a flash-based FPGA, a 32-bit ARM Cortex M-3 microcontroller subsystem and an analogue computing engine. The hyperspectral images are divided in clusters of 32 bands to reduce the processing time and memory requirements. The results show an improvement of more than 54 % of the execution time for the proposed architecture and a power consumption of 225 mW for a cluster of 32 bands. These results demonstrate that it is possible to reduce the complexity of the KLT transform to target space applications.

The implementation of the ICER-3D algorithm [90] is another interesting case in which a transform-based hyperspectral compression method is implemented on an FPGA. Specifically, it was implemented on a Xilinx Virtex2 Pro (XC2VCP70) with an embedded PowerPC processor and on-chip bus architecture. The implementation features efficient utilization of off-chip memory through internal buffering to minimize intensive input/output operations. The results display a maximum frequency of 50 MHz and a throughput of 4.5 Msamples/sec. The power consumption of the prototype is less than 6.5 W and the hardware occupancy is around 60%.

Among the FPGA implementations of prediction-based algorithms for hyperspectral image compression, the one of the FL algorithm is particularly remarkable [55]. It has to be noted that the FL algorithm was selected by the CCSDS for standardization and it represents, with little modifications, the base of the current CCSDS 123 standard for lossless multi- and hyperspectral image compression. The FL algorithm was implemented on a Xilinx Virtex4 LX160 SRAM-based FPGA [32]. The implementation has a critical path of

TABLE 2.2: Virtex IV LX160 device utilization of the FL algorithm

Device	**Virtex IV LX160**
Resource	Used
Slice	67584 (5%)
FIFO/RAMB16	9 (3%)
DSP48	6 (6%)

TABLE 2.3: Virtex4 and Virtex2 device utilization of a lossless and near-lossless hyperspectral compression algorithm

Device	**XQR4VLX200**	**XQ2V3000**
Used LUT	10306 (5%)	10248 (35%)
RAMB16S	21 of 336 (5%)	21 of 96 (22%)
Mult18×18	-	9 of 96 (9%)
DSP48	9 of 96 (9%)	-

29.5 nsec (maximum frequency 33 MHz) and compresses one sample every clock cycle, which results in a throughput of 33 MSamples/sec. It processes the samples in band interleaved by pixel (BIP) order. It is 33 times faster than a software implementation running on a Pentium IV machine. The implementation has a rather low hardware occupancy, as shown in Table 2.2, and its power consumption is estimated at 1.27 W.

Another example of a prediction-based algorithm is the lossless to near-lossless hyperspectral compression algorithm presented in [56]. The algorithm implemented on a Xilinx Virtex2 V3000 and on a Xilinx Virtex4 LX200 FPGA, producing the hardware occupancy results presented in Table 2.3, with a maximum frequency of 81 MHz and a throughput of 70 Msamples/sec for the Virtex4 FPGA and 79 MHz and 69 Msamples/sec for the Virtex2.

Table 2.4 summarizes the most relevant figures in terms of throughput, frequency and power of some of the FPGA implementations of satellite data compression algorithms presented in this Section. Since the algorithms were

implemented on different technologies, the hardware occupancy data could mislead the reader to imprecise comparisons and therefore are not included in Table 2.4. Nevertheless, the on-board compression algorithms implemented on FPGAs that can be found in the literature have a complexity which is low enough to allow their implementation on most of the currently available space-qualified FPGAs.

TABLE 2.4: FPGA implementations of on-board data compression algorithms. NDA stands for No Data Available.

Algorithm	**Type**	**Basis**	**FPGA Technology**	**Troughput (Msamples/ sec)**	**Troughput (Mbps)**	**Max. Freq. (MHz)**	**Power (mW)**
FAPEC	1D	Prediction	Antifuse (prototype in flash)	2	NDA	55	35
CCSDS 122 for PROBA-V	2D	Transform	Antifuse (prototype in flash)	NDA	90	64	NDA
CCSDS 122 for EnMap	2D	Transform	Antifuse (prototype in flash)	NDA	130	50	NDA
ICER-3D	3D	Transform	SRAM	4.5	NDA	50	< 6500
FL	3D	Prediction	SRAM	33	NDA	33	1700
Lossless to near-lossless	3D	Prediction	SRAM	80	NDA	81	NDA

Chapter 3

Implementation of a lossy compression algorithm for hyperspectral images on a GPU

This Chapter presents the implementation of an algorithm for lossy compression of hyperspectral images on a GPU. The compression algorithm is described, showing that it allows a fair amount of parallelization, and hence is a good candidate for a GPU implementation. The main strategies utilized to boost the performance of the algorithm on a GPU are presented, as well as experimental results demonstrating the achieved speedup.

3.1 Outline

The Lossy Compression for Exomars (LCE) algorithm [33] was specifically designed for the compression of hyperspectral images on-board satellites. Hence, it meets several important requirements which facilitate its implementation on the available on-board hardware and exhibits high compression efficiency at the same time. It is inherently data- and task-parallel what makes it a good candidate for its implementation in technologies which can exploit parallel processing. This is the case of both, GPUs and FPGAs.

In this Chapter we assess how well the LCE algorithm performs in each of these technologies, demonstrating the features of the algorithm in terms of low-complexity and hardware-friendliness. Furthermore, we present strategies in order to exploit parallelization as much as possible, by removing data dependencies and finding solutions for reducing the number of iterations and simplifying computations along the algorithm's flow.

The suitability of both implementation options, GPUs and FPGAs, for implementing the LCE algorithm is proven, and the performance of the implementations is evaluated in terms of throughput, area occupancy (in the case of the FPGA) and power consumption.

Whenever hyperspectral compression algorithms are designed for on-board applications, it is mandatory to determine how well they will perform when they are actually implemented in hardware. However, most of the literature analysed limit their implementations to software which is executed on general-purpose processors, which might not be the best candidates for hyperspectral image compression, because of the nature of the algorithms and their characteristic complexity. This study aims at exploring other implementation options, which can possibly exhibit very good performance when utilized for hyperspectral image compression, exploiting parallelism and a more customized implementation design.

3.2 LCE algorithm description

The Lossy Compression for Exomars (LCE) algorithm [33] was developed in Politecnico di Torino and was conceived with the aim of fulfilling the data compression needs of the Exomars mission [66]. It derives from a lossless compression algorithm based on distributed source coding principles, which was the outcome of an ESA project in the framework of its Innovation Triangle Initiative (ITI) program, whose main objective is to promote the introduction of innovations and technologies in the space environment. The LCE algorithm presents an improved version of that algorithm with significant differences. LCE is purely lossy and not based on distributed source coding, but on predictive coding coupled with quantization and RD optimization. These last steps, quantization and RD optimization, make the algorithm truly lossy, and not near-lossless; and allow the algorithm to perform well at low bit-rates, where near-lossless compression typically shows poor performance. Another important characteristic of the LCE, which motivates its inclusion in this Thesis work, is that it had been designed specifically taking into consideration that it would be used in space missions, and therefore it was expected to operate on-board. Consequently, the algorithm was designed to achieve a high compression efficiency, and meet the requirements for on-board compression at the same time, specifically:

- *Low-complexity.* The algorithm applies a scheme based on prediction plus Golomb power-of-two entropy coding of the prediction residuals. This scheme exhibits lower complexity than others based on transforms or arithmetic coding. The low-complexity of the algorithm facilitates its physical implementation, which is expected to require a low amount of hardware resources; and increases the throughput (number of samples compressed per unit of time) and the possibility of obtaining an implementation which would operate in real-time.

- *Error-resilience.* Errors can be caused by radiation effects or occur during the transmission of the data. A single bit error in the compressed bit stream can cause a significant loss of data, because it prevents the decompression of the rest of the image after the error. The LCE algorithm operates on independent blocks of $N \times N$ samples with all bands, providing some resilience to errors, since the error propagation will be confined to the block where it was produced. The rest of the blocks in the hyperspectral image can be decoded successfully despite the error.

- *Hardware-friendliness.* The LCE algorithm is a good candidate for an ASIC or an FPGA implementation, due to its low-complexity and the fact that it can operate using only integer arithmetic. Moreover, the fact that it operates on independent blocks of data makes it possible to parallelize it in order to speed up the compression. The amount of data parallelization in the algorithm makes it also suitable for an implementation on a GPU.

As it was mentioned, the LCE algorithm compresses independent non-overlapping blocks of spatial size $N \times N$ samples with all bands. The fact that each block is processed separately entails a small performance penalty, since the entropy coding operations have to be reset at the end of each block. However, it offers two important advantages: it allows parallelization and provides error resilience. Throughout this study, we consider that the size of the block is set to $N = 16$, since it yields the best trade-off between compression performance and resilience to errors [33].

The aforementioned features make the LCE algorithm the best candidate at the moment to be included in this Thesis as the implementation target in the hardware technologies that this work is focused on: FPGAs and GPUs. In the following sections, the main stages of the LCE algorithm are described.

3.2.1 Prediction

In this first stage, the LCE algorithm performs the prediction of the current sample to be compressed. In the following we use indices (z, y, x) to locate a sample in the whole hyperspectral cube and indices (z, m, n) to locate a sample within a block of 16×16 pixels with all bands. Therefore, $s_{z,y,x}$ is used to denote a sample in the z-th band, y-th line and x-th column; and $s_{z,m,n}$ a sample located in the z-th band, m-th line and y-th column within a specific block, as shown in Figure 3.1.

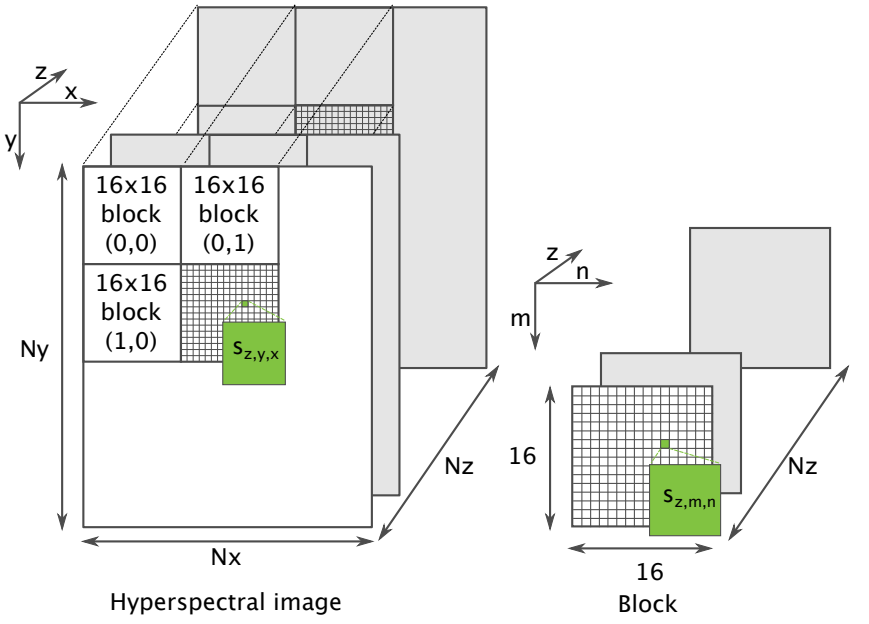


FIGURE 3.1: Division of the hyperspectral cube into blocks and notation.

The prediction neighbourhood used by the LCE algorithm is illustrated in Figure 3.2. For the first band (z=0), 2D compression is performed without any information of other bands (INTRA-mode) using a predictor defined as:

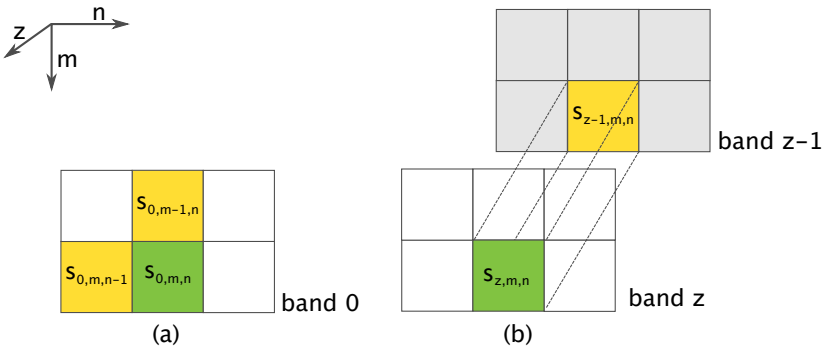


FIGURE 3.2: Prediction neighbourhood of the LCE algorithm: a) first band b) all other bands.

$$\hat{s}_{0,m,n} = (\tilde{s}_{0,m-1,n} + \tilde{s}_{0,m,n-1}) \gg 1 \tag{3.1}$$

In the previous equation, $\hat{s}$ denotes the predictor, $\tilde{s}$ the decoded value and $\gg$ stands for right shift.

For all the other bands, the samples $s_{z,m,n}$ are predicted from the decoded samples $\tilde{s}_{z-1,m,n}$ in the co-located block in the previous band. A least-square estimator (LMS) is computed over the block as

$$\alpha = \alpha_N / \alpha_D \tag{3.2}$$

where

$$\alpha_N = \sum_{m,n} [(\tilde{s}_{z-1,m,n} - \mu_{z-1})(s_{z,m,n} - \mu_z)] \tag{3.3}$$

and

$$\alpha_D = \sum_{m,n} [(\tilde{s}_{z-1,m,n} - \mu_{z-1})(\tilde{s}_{z-1,m,n} - \mu_{z-1})] \tag{3.4}$$

where μ_z and μ_{z-1} are the average values of the co-located decoded blocks in bands z and $z-1$. Quantized versions of α and μ_z, denoted α' and μ'_z, are generated using a scalar quantizer. Finally, the predicted values are computed for all samples in a block as:

$$\hat{s}_{z,m,n} = \mu'_z + \alpha'(\tilde{s}_{z-1,m,n} - \mu_{z-1}) \tag{3.5}$$

and the prediction error is calculated as:

$$e_{z,m,n} = s_{z,m,n} - \hat{s}_{z,m,n} \tag{3.6}$$

3.2.2 Rate-distortion optimization

Before proceeding with the quantization of the prediction error samples, the algorithm checks if the prediction is so close to the actual pixel values that it makes sense to skip the encoding of the prediction error samples and, instead, rise a one-bit-flag indicating that the current block contains all-zero prediction error samples (this is denoted as *zero_block* condition). To make this decision, the energy of the predictor error is computed:

$$D_0 = \frac{1}{N \times N} \sum_{m,n} e^2_{z,m,n} \tag{3.7}$$

If $D_0 < D_T$, with D_T a chosen threshold, then the *zero_block* condition is triggered and the decoded values are calculated as:

$$\tilde{s}_{z,m,n} = \hat{s}_{z,m,n} \tag{3.8}$$

3.2.3 Quantization and mapping

For the non-zero blocks, the prediction error samples are quantized to integer values $e'_{z,m,n}$, and dequantized to reconstructed values $e''_{z,m,n}$. The decoded pixel values can be then obtained by adding the dequantized prediction error to the predicted value:

$$\tilde{s}_{z,m,n} = \hat{s}_{z,m,n} + e''_{z,m,n} \tag{3.9}$$

For the first band, the quantization process is performed pixel by pixel using a scalar uniform quantizer. For the other bands, it is possible to choose between a scalar uniform quantizer and the uniform-threshold quantizer (UTQ) described in [91]. The UTQ quantizer exhibits superior reconstruction performance, however, this quantizer, as described in the LCE algorithm, cannot be implemented with integer arithmetic only and involves a non-linear formula. This has to be taken into account when implementing the algorithm in hardware. As an alternative, the user can opt for the scalar quantizer, which utilizes only integer arithmetic.

Finally, the quantized prediction errors are mapped to non-negative values and entropy encoded, with one exception: in the first band, the first sample of each block is neither mapped nor encoded. It is written in binary format using 16 bits.

3.2.4 Entropy coding

The 16×16 residuals of a block are encoded in raster order using a Golomb code whose parameter is constrained to a power of two, except for the first sample of each block, which is encoded using an exponential Golomb code of order zero.

Exponential Golomb (exp-Golomb) codes were first proposed in [57]. The codewords consist of a prefix and a suffix. The prefix part of the exp-Golomb code of parameter k for a nonnegative integer τ consists of a unary code corresponding to the value $u = \left\lfloor \log_2 \left(\frac{\tau}{2^k} + 1 \right) \right\rfloor$. The suffix is computed as the binary representation of $\tau + 2^k(1 - 2^u)$ using $k + l$ significant bits. In the particular case of the LCE, the k value for the exp-Golomb code is always fixed to 0.

The rest of the samples are encoded using a Golomb code. The parameter k_j for the j-th sample of the block is computed from a running count R_j of the sum of the magnitude of the last 32 unmapped residuals of the block; for samples with index less than 32 only the available values are used.

Similarly to a Huffman code, Golomb codes are unable to produce codewords shorter than 1 bit. This means that, without the rate-distortion (RD) optimization stage described in Section 3.2.2, it would be impossible to obtain bit-rates lower than 1 bpp. However, with the proposed scheme, the minimum rate for each block is still 1 bpp, except for the blocks that are skipped, what makes it possible for the average rate to go as low as desired.

3.2.5 File format

The compressed file is a concatenation of coded blocks which are read spatially in raster order, and each block is coded with all bands. The first band is conformed by the quantized first sample represented with 16 bits followed by the codewords of the remaining mapped prediction residuals of the block. For all other bands, since the predictor is not causal, it is necessary to write the parameters α' and μ_z for each block. 10 bits are used to store α' and 16 bits are reserved for μ_z. These parameters are followed by the *zero_block* decision bit. For the non-zero blocks, the exp-Golomb and Golomb encoded

codewords are appended. Figure 3.3 illustrates the format of the compressed file.

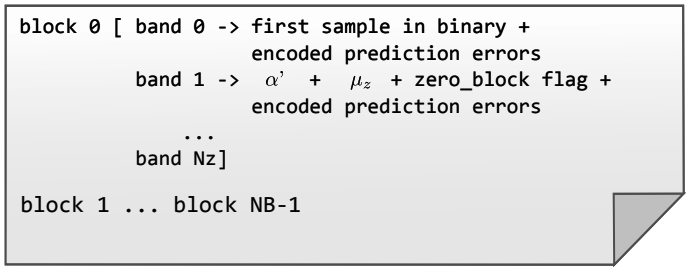


FIGURE 3.3: File format of the LCE compressed data.

3.2.6 LCE compression efficiency

The compression efficiency of the LCE algorithm was compared in terms of RD in [33], showing the relationship between the achieved compression bit-rate and the PSNR of the reconstructed images. The results demonstrate that the LCE algorithm shows equal or better rate-distortion (RD) than JPEG2000 with a spectral wavelet transform, but with a significantly lower complexity and memory requirements. The RD curves obtained for an hyperspectral image taken by AVIRIS are depicted in Figure 3.4. The Figure is shown exactly as it appeared in [33].

3.3 Software implementation of the LCE algorithm

The LCE algorithm was originally implemented in software using C programming language, to be executed on a single threaded CPU. It operates independently on every $N \times N$ block with all its bands. For the specific case of this study, we consider the block size $N = 16$, which as stated in

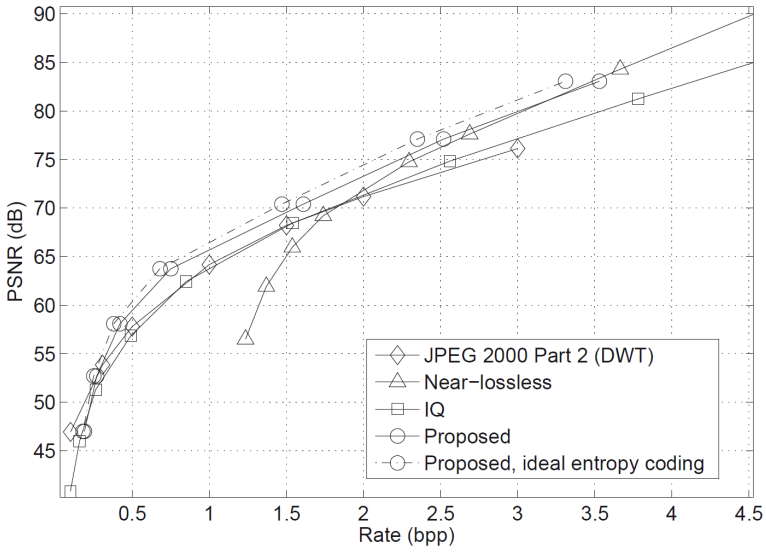


FIGURE 3.4: Rate-distortion curves for AVIRIS when compressed with LCE and other algorithms of the state-of-the-art [33]

[33] typically optimizes the algorithm performance. Each block in the image is identified with two coordinates in the vertical and horizontal spatial directions, namely vb and hb. $B_{z,vb,hb}$ represents a 16×16 pixel block in coordinates (vb, hb) and band z. A single sample in $B_{z,vb,hb}$ is symbolized as $s_{z,m,n}$. Additionally, Nvb and Nhb are the total number of 16×16 blocks that can be found in the image, in vertical and horizontal spatial dimensions.

The LCE compressor consists essentially of a chained loop, which iterates to cover all horizontal and vertical blocks in the image, and all bands in a block. The innermost loop reads a 16×16 block $B_{z,vb,hb}$ and performs the different stages of the LCE algorithm presented in Section 3.2 for every single pixel $s_{z,m,n}$, namely the prediction, rate-distortion optimization, quantization, entropy coding and bit packing to create the compressed file. The pseudo-code

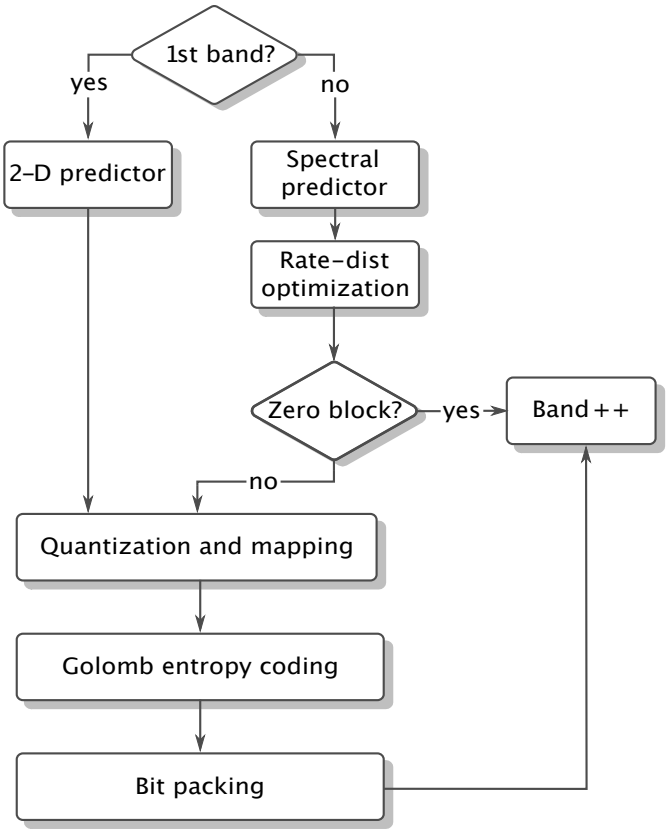


FIGURE 3.5: Flowchart of the LCE algorithm

showing the chained loop and the different compression stages is shown in Figure 3.6.

3.3.1 Generation of the compressed file

The result of the previously described chained loop is a bit stream which represents the compressed image. The codewords generated at the entropy coding stage and the parameters α' and μ'_z are written by the software in a single file, in the same order they are obtained. Since Golomb codes produce

```
void lce_compressor {
    for (vb = 0; vb ≤ Nvb; vb++) {
        for (hb = 0; hb ≤ Nhb; hb++) {
            for (z = 0; z ≤ Nz; z++){
                if (z == 0) {
                    2D-prediction;
                    quantization and mapping;
                    entropy coding; }
                else{
                    spectral prediction;
                    rd-optimization;
                    quantization and mapping;
                    entropy coding; }
            }
        }
    }
}
```

Additional for loops are needed to cover all the pixels $s_{z,m,n}$ in a block

FIGURE 3.6: Pseudo-code of the main function of the LCE algorithm implementation in C language.

codewords of variable length, they are buffered in a bit-by-bit fashion in a 32-bit variable. When the buffer is full, it is written to the compressed file, generating the bit stream.

3.3.2 Configuration parameters

The software implementation allows the user to configure the LCE algorithm by selecting several parameters to set the functionality mode (baseline or advanced) and the quality of the resulting compressed image. The baseline algorithm performs compression using a scalar quantizer, fully implemented with integer arithmetic; and an advanced algorithm replaces the scalar quantizer with the uniform-threshold quantizer (UTQ) [91], which uses floating-point arithmetic. The parameter UTQ can be set by the user to select the functionality mode. Setting UTQ = 1 enables the UTQ quantizer and setting UTQ = 0 uses the baseline functionality mode.

The quality of the resulting compressed image can be likewise set by the user by assigning values to a parameter named *delta*, making it possible to find a trade-off between quality and bit rate. *Delta* sets the quantization step size of the quantizer, therefore, increasing *delta* yields to higher compression ratios, but lower quality of the reconstructed image. *Delta* has to be an integer greater than 1, with $delta = 1$ providing the maximum possible quality.

Taking into account the specifications of the LCE algorithm, we can conclude that it can benefit from parallel execution schemes, due to the amount of operations that can be performed independently. As it was explained before, the algorithm compresses independent blocks of data, and having a closer look at the stages of the algorithm, we observe that there are operations that can be performed on every sample independently. Given the amount of data that can be processed in parallel, we first explore how much the algorithm can be accelerated when executed in a GPU. In the following, we present a brief description of the GPUs architecture and programming environment. Afterwards, we show the main strategies utilized to optimize the LCE implementation in order to achieve the maximum possible speedup with the GPU. Finally, we show the achieved acceleration with experimental results running on real hyperspectral images.

3.4 GPU architecture and NVidia CUDA

GPUs are structured as a set of multiprocessors, each composed of a simple processing element working in single-instruction multiple data mode. They offer the possibility to dramatically increase the computation speed in applications where a huge amount of data can be processed in parallel. At the time this document is written, the biggest GPU manufacturers are Intel,

Nvidia and AMD. GPUs were initially designed to accelerate graphic computations, but recently they have become popular also for general-purpose computing, i.e. they can be programmed to compute any computable value. However, they are very restrictive in operations and programming, and are only effective for problems which can be solved using stream processing - processors that can operate in parallel by running one kernel on many records in a stream at once; and the hardware can only be used in certain ways. The currently dominant open general-purpose GPU programming language is OpenCL [92], but other proprietary frameworks exist. The GPU utilized for this Thesis work is the Tesla C2075 from Nvidia (see 3.4.3), and consequently, it was decided to program it using Nvidia's platform, known as CUDA.

The Computer Unified Device Architecture (CUDA) [93] [94] [95] is a parallel computer architecture developed by Nvidia, which provides a scalable programming model and a software environment for parallel computing. It is an extension of C language, which offers an Application Programming Interface (API), allowing programmers to use a GPU as a massively multi-threaded general purpose co-processor. The APIs make it possible to manage devices and memories, but hide the real hardware from the developers, making it unnecessary for the programmers to explicitly manage threads. The CUDA abstractions guide the programmer to partition the problem into coarse subproblems that can be solved independently in parallel and then into finer pieces that can be solved cooperatively in parallel.

3.4.1 CUDA abstractions

CUDA names the CPU and its memory the *host* and the GPU and its memory, the *device*. The code executed in parallel using the GPU is typically called a *kernel*. A kernel is launched from the host and can have parameters, like any other C language functions (see Figure 3.7).

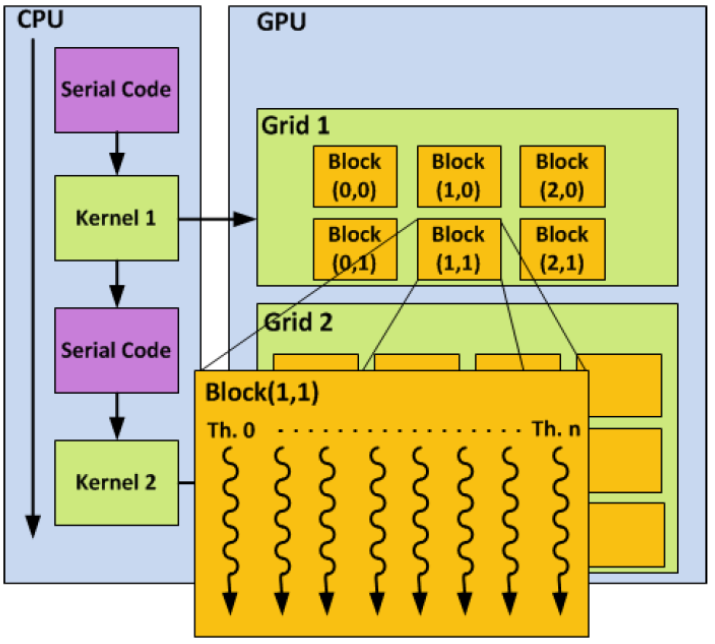


FIGURE 3.7: CUDA abstractions: threads, blocks and grid.

A CUDA kernel executes in parallel across a set of parallel *threads*. The programmer organizes these threads into a hierarchy of *grids* of *blocks*. A CUDA block is a set of concurrent threads that can cooperate among themselves. A grid is a set of blocks that may be executed independently and may thus execute in parallel. The dimensions of the grid and blocks is set by the programmer in the kernel call, however it must be noted that there is a maximum number of blocks and threads that can be launched in a single kernel invocation. This number depends on the specific model of GPU in use.

Each thread is given a unique thread identification number (*threadIdx*) within its block, and each block is given a unique block identification number (*blockIdx*) within its grid. These identifiers can be used by the programmer to index data allocated in the GPU memory.

3.4.2 CUDA memory spaces

Threads may access data from multiple memory spaces during their execution. Each single thread has a private local memory, not visible for other threads. Shared memory is a fast access memory which is visible to all threads in a block. Finally all threads in all blocks have access to the same global memory. The CUDA memory abstractions are depicted in Figure 3.8.

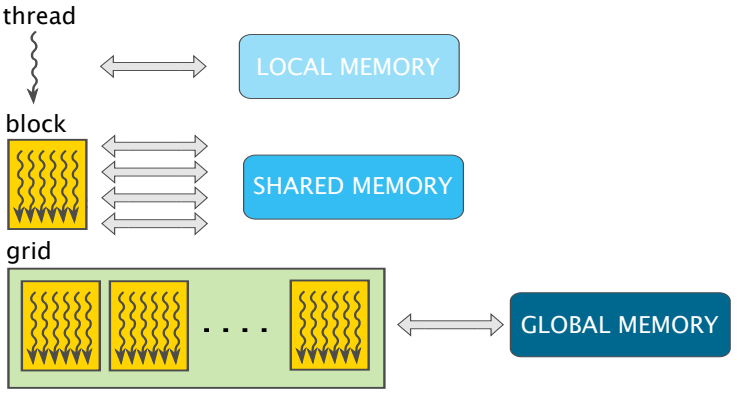


FIGURE 3.8: CUDA memory spaces

3.4.3 Nvidia TESLA C2075 GPU

Before parallelizing an algorithm using the CUDA abstractions, the programmer has to keep in mind the hardware architecture, specifications and

TABLE 3.1: Main specifications of the Tesla C2075 GPU.

Description	**Tesla C2075**
Nvida CUDA Cores	448
GPU memory	6144 MB
Maximum memory bandwidth	144 GB/sec
Peak double precision floating point performance	515 Gflops
Frequency of CUDA cores	1.15 GHz
Memory speed	1.50 GHz
Power consumption	225 W TDP

limitations of the target GPU. Specifically, these limitations refer to the amount of dedicated memory storage, computation capability and maximum number of CUDA blocks and threads that can be launched by a single kernel. In this Thesis work, the LCE algorithm is implemented in a Nvidia Tesla C2075 GPU, which has 6 GBytes of dedicated off-chip Graphics Double Data Rate (GDDR), Version 5 memory and restricts the number of CUDA blocks and threads to a maximum of 65535 and 1024 respectively.

The most important features of the Nvidia Tesla C2075 are summarized in Table 3.1. It is based on the Nvidia GPU architecture code-named Fermi [96]. The first Fermi based GPU, implemented with 3.0 billion transistors, features up to 512 CUDA cores. A CUDA core executes a floating point or integer instruction per clock for a thread. The 512 CUDA cores are organized in 16 streaming multiprocessors (SMs) of 32 cores each. A Fermi GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-Express. The GigaThread global scheduler distributes thread blocks to SM thread schedulers. A representation of the Fermi architecture can be seen in Figure 3.9.

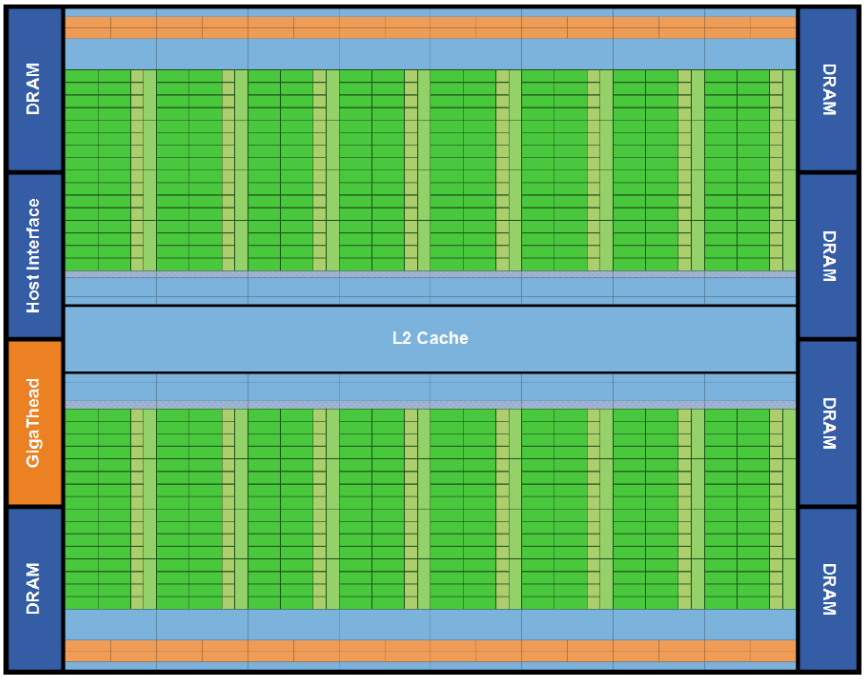


FIGURE 3.9: Fermi architecture. Streaming multiprocessors (SM) are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contains an orange portion (scheduler and dispatch), a green portion (execution unit) and light blue portions (register file and L1 cache)

The Fermi architecture has a configurable 64KB on-chip private first-level (L1) cache with every streaming multiprocessor and a 768 KB second-level (L2) cache shared by all multiprocessors. The L1 cache can be configured as 48 KB of shared memory with 16 KB of L1 cache or as 16 KB of shared memory with 48 KB of L1 cache. The configuration is selected by the user at compiler time, in order to optimize the performance of a specific application. L1 cache and shared memory help to hide the memory latency of the off-chip GDDR5.

The Fermi memory architecture is related with the CUDA memory spaces abstraction as is shown in Figure 3.10 and is explained next. Global memory

resides off-chip, and it is cached in L1 and L2. CUDA shared memory resides on-chip, and the user should ensure that the configuration of the on-chip memory makes it possible to allocate all the necessary shared data. CUDA local memory also resides in the GDDR5 memory and is chached in L1. It must be noted that accessing local memory can be faster than accessing the global memory, since the local memory addresses are resolved by the compiler.

The first step when executing a parallel kernel on a GPU is to send the necessary data from the host to the device global memory, in such a way that it is available for all the threads. Transactions between CPU and GPU might represent a bottleneck for performance, which is carefully studied in the particular case of hyperspectral image compression, since the amount of data to be sent to the GPU is considerable.

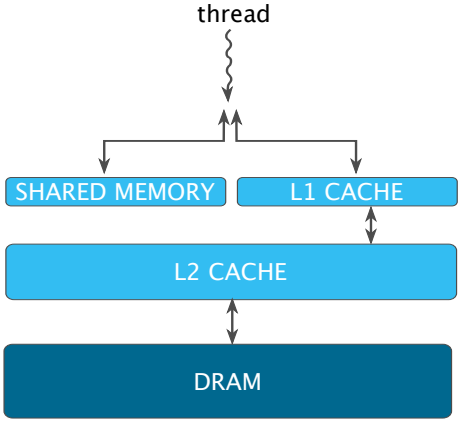


FIGURE 3.10: Fermi memory hierarchy

3.5 Parallelization of the LCE compressor with CUDA

In order to parallelize the LCE compressor for its execution in a GPU, several facts are considered, in particular:

- The compression of every $B_{vb,hb}$ block with all its bands can be performed independently.
- The prediction of all samples $s_{z,m,n}$ in a block $B_{z,vb,hb}$ in a specific band can be performed independently, except for the first band ($z = 0$), where 2D prediction is employed.
- There is a dependency between bands for the prediction, rate-distortion optimization and quantization phases. This means that these operations have to be finished for band z before they can be started for next band $z + 1$.
- The entropy coding operations can be performed in every spatial 16×16 block $B_{z,vb,hb}$ independently, as there is no dependency between bands for this operation. However, there is a data dependency between samples at the entropy coding stage, specifically when calculating the Golomb parameter. A strategy is devised (see Section 3.5.3), in order to make it feasible to compute the Golomb parameter and create the codewords in parallel for all the samples in a block.

The CUDA implementation of the compressor consists of various kernels, instead of a single one which performs all stages. By doing so, we permit to optimize the dimension of the kernels for the different levels of parallelism that can be achieved at the several stages of the algorithm.

Both, the parallel LCE compressor and decompressor, are designed in such a way that they are useful for any hyperspectral image, regardless its spatial

or spectral dimension, achieving high performance at the same time. Hence, the number of CUDA threads and blocks called in each kernel is established in such a way that the CUDA application is suitable to compress any hyperspectral image regardless its spatial or spectral size in short compression times. Nevertheless, the kernel dimensionality might be further optimized for a specific image, taking into consideration its size and adjusting the number of threads and blocks accordingly.

3.5.1 Allocation of the image data in the GPU

The necessary image data has to be sent and stored in the GPU dedicated memory in order to make it possible to perform operations on them. It is decided to copy the whole hyperspectral cube in the GPU before executing any of the kernels, in order to minimize the number of transactions between the CPU and GPU. Once the necessary data are stored in the GPU, an efficient use of the different memory spaces is necessary to hide latency. When data are copied from the CPU to the GPU, they are initially stored in global memory. However, CUDA local or shared memory spaces are used whenever it is convenient in order to accelerate memory accesses.

The hyperspectral data are read in the same format which was used in the C implementation, i.e. band-sequential (BSQ). Once allocated in the GPU, the different 16×16 pixel blocks are read and copied to the shared memory to hide the global memory latency, since the original pixels are accessed repeatedly and shall be visible for all threads in a block.

3.5.2 Prediction, quantization and mapping

The main difficulty found when parallelizing these stages of the LCE algorithm is that it is impossible to avoid having a sequential loop to cover

all bands, because the processing in band z cannot be started until the processing of samples in band $z-1$ is finished. Acceleration by means of parallelization is obtained by taking advantage of the fact that each 16×16 image block with all its bands can be processed independently.

The parallelization strategy, as shown in Figure 3.11, is based on mapping the problem to the CUDA abstractions in such a way that a CUDA block is launched by the kernel for each spatial 16×16 block $B_{vb,hb}$ with all its bands. Each CUDA block is set up to have 256 (16×16) CUDA threads, each responsible for performing operations on a sample $s_{z,m,n}$ in parallel. Therefore, the number of CUDA blocks which are launched can be calculated for a specific hyperspectral cube as:

$$CUDA\ blocks = \frac{Ny \times Nx}{16 \times 16} \tag{3.10}$$

where Ny is the number of lines and Nx represents the number of columns in a specific band of a hyperspectral image.

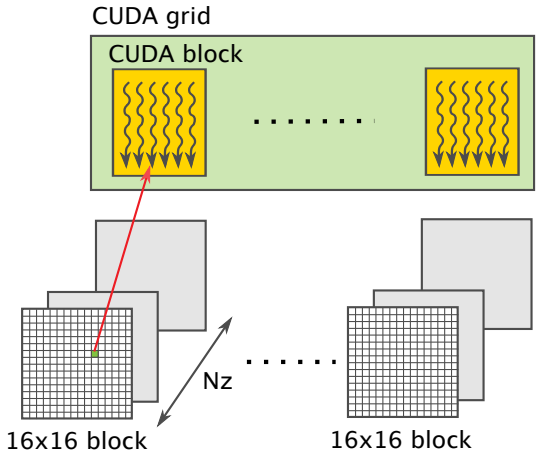


FIGURE 3.11: CUDA abstractions for the parallel execution of the LCE prediction, quantization and mapping stages.

The hyperspectral image samples are stored in the CUDA global memory. As the samples are going to be accessed many times by the CUDA threads, it is decided to copy them to the CUDA local memory. Specifically, every iteration of the loop that covers all bands, a spatial 16×16 block is copied to the CUDA local memory, indexing it with the CUDA blocks and threads identifiers. This way, threads have fast access to the sample they are entitled to process.

Local memory can only be accessed by a single thread, and it is invisible for the other threads in a CUDA block. Nevertheless, not all the operations on a sample $s_{z,m,n}$ can be performed independently, as it is the case of computing parameters α and μ, for which a summation has to be performed. However, these operations can be accelerated by making threads cooperate and share information by means of CUDA shared memory. Specifically, in order to reduce the number of iterations of the loops when computing summations, the so-called parallel reduction algorithm is employed [97]. The strategy presented in [98] is also utilized to reduce the number of iterations when prefix-sums have to be performed.

Once the prediction errors are obtained, they are copied again to the CUDA global memory, to make them accessible for the next stages of the LCE.

3.5.3 Entropy coding

Unlike the prediction stage previously described, the entropy coding operations can be performed on every 16×16 block of a specific band in parallel, without any information from neighbouring bands, what makes it possible to process more data in parallel.

A kernel is designed to perform the entropy coding of the mapped prediction residuals, which processes each 16×16 block of prediction residuals in parallel, as shown in Figure 3.12. For optimization purposes, it is established

that each kernel launches the maximum number of possible threads allowed by the Tesla C2075 GPUs, which is 1024. Therefore, the number of CUDA blocks to be launched can be calculated as:

$$CUDA\ blocks = \frac{Nz \times Ny \times Nx}{16 \times 16} \tag{3.11}$$

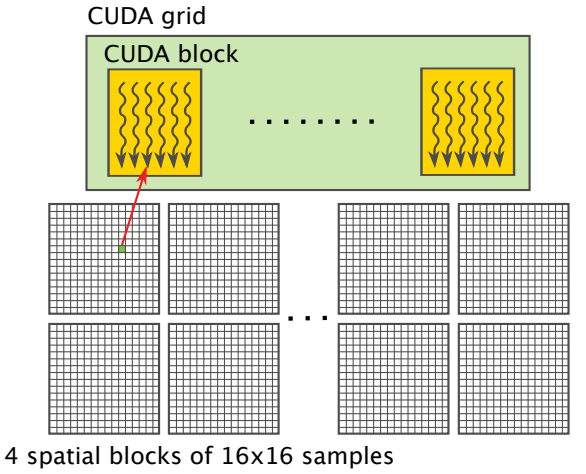


FIGURE 3.12: CUDA abstractions for the parallel execution of the LCE entropy coding stage.

It has to be considered that the maximum number of CUDA blocks is limited to 65535. If the number of CUDA blocks calculated in the previous equation is higher than this maximum, then the kernel has to be called more than once, affecting the performance negatively. Setting the number of threads to the maximum is likewise a way to minimize the number of necessary CUDA blocks, and therefore reduce the impact of having to invoke the kernel repeatedly.

Several facts are considered when designing the GPU implementation of the entropy coding stage:

- The codewords shall be computed in parallel for each mapped prediction error sample.
- It is not possible to directly write on a file from the GPU, consequently, the codewords have to be saved to variables in the CUDA memory spaces.
- The codewords have variable lengths, which can be greater than 32 bits.

The strategy followed to generate the codewords and pack them to a bitstream must be different to the one followed in the CPU implementation, which is based on the sequential ordered generation of the codewords, as it was explained in Section 3.3. For the GPU implementation, the approach is to pre-process the Golomb parameters of all mapped prediction error samples in a 16×16 block, and afterwards compute the codewords for every sample in parallel.

Computing the codewords

A strategy is designed to compute the Golomb parameter of every j-th sample of the block, k_j, in parallel. This parameter is computed from a running count R_j of the sum of the magnitude of the last 32 unmapped prediction errors of the block, e_j; for samples with index less than 32, only the available values are used. This implies that the Golomb parameter of a specific prediction error in a block depends on the accumulated sum of the previous unmapped prediction errors. The running count is calculated for every sample as:

$$R_j = R_{j-1} - |e_{j-33}| + |e_{j-1}| \tag{3.12}$$

In order to be able to obtain R_j in parallel, the prefix-sum of all the unmapped values is computed as:

$$E_j = \begin{cases} 0, & \text{if } j = 0 \\ \sum_{j=1}^{254} e_j, & \text{if } j > 0 \end{cases} \tag{3.13}$$

This is implemented in CUDA following the scheme proposed in [98], as it is explained in the following. We note that this strategy requires the cooperation of threads, therefore the necessary data has to be copied to the CUDA shared memory.

The array of unmapped prediction errors in a block of 16×16 is represented as: $[e_0, e_1, ...e_{255}]$

The prefix-sum gives the result:

$$E = \begin{bmatrix} E_0 \\ E_1 \\ E_2 \\ ... \\ E_j \\ ... \\ E_{255} \end{bmatrix} = \begin{bmatrix} 0 \\ |e_0| \\ |e_0| + |e_1| \\ ... \\ |e_0| + |e_1| + ... + |e_{j-1}| \\ ... \\ |e_0| + |e_1| + ... + |e_{254}| \end{bmatrix} \tag{3.14}$$

After the prefix-sum, R_j is calculated by subtracting element E_{j-33} to each E_j with $j > 32$ in parallel, resulting in the desired R_j:

$$R_j = \begin{cases} E_j, & \text{if } j \leq 32 \\ E_j - E_{j-33}, & \text{if } j > 32 \end{cases} \tag{3.15}$$

Once k_j is known for every prediction error sample, the codewords can be created in parallel by the CUDA threads. Each thread computes and saves a codeword in its local memory. The codewords are of variable length and each of them is saved in a 32-bits unsigned integer variable. The Golomb codes can produce codewords of any length, i.e. it might happen that a codeword is longer than 32-bits. For those specific unusual cases, an auxiliary array is created to save the codeword in more than one 32-bits variable. Utilizing this auxiliary array can cause a performance penalty. As part of a future work, this problem can be solved by setting a maximum codeword size so that, in case the codeword is longer than the maximum, the data are not encoded but saved in binary. The size in bits of the codewords is also saved by every thread, in order to be able to create the encoded buffer, as it is explained in the following.

Generating a compressed bit stream for each 16×16 block

The encoded prediction residuals have to be saved in raster order to produce the final encoded 16×16 block, which contains the ordered sequence of codewords, without leaving any bits unused between them. The strategy presented in [99] is followed to write every codeword of a 16×16 block in a single output buffer, as shown in Figure 3.13.

First, the final position of a codeword in the output buffer is calculated. This final position is given by two coordinates: the word position where the codeword is saved, *word_position*; and the bit position in that word where the codeword starts, *starting_bit* of every codeword. These coordinates can be calculated in parallel for every codeword. The number of bits taken by each codeword is known as a result of the entropy coding stage. In the following,

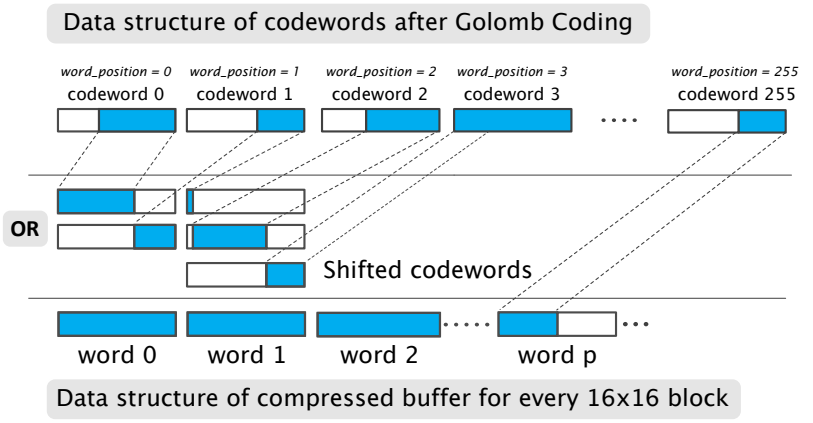


FIGURE 3.13: Parallel generation of a compressed 16×16 block.

we denote q_j the number of bits taken by each codeword j. A prefix-sum of these data yields the bit position of a codeword in the output buffer, Q_j.

$$Q = \text{prefix-sum}(q_j) = [0, (q_0 + q_1), (q_0 + q_1, q_2), ..., \sum_{j=0}^{254} q_j] \tag{3.16}$$

Dividing these results by 32 and obtaining the remainder yields the desired *word_position* and *starting_bit*. Once the two coordinates are calculated, the codewords are shifted in such a way that they start in the corresponding *starting_bit*. Afterwards, a logical OR is performed between the codewords which share the same *word_position*, using CUDA atomic operations, which make it possible for a thread to perform an operation without interference from any other threads, to avoid having threads with the same *word_position* accessing the output encoded buffer at the same time.

3.5.4 Bit packing

Once the encoded blocks corresponding to a 16×16 pixel portion of the image are obtained, they have to be written to a single bit stream which represents the compressed image. As each block has been processed independently in the entropy coding kernel, the resulting encoded blocks have been written to a specific position of the global memory. To construct the final output bit stream, the encoded blocks have to be saved in sequential order and, as it happened with the codewords, in such a way that no bits are left unused between them.

A similar strategy to the one used by the entropy coder is followed: the word position and starting bit is calculated for every compressed block. Afterwards, the compressed block is shifted and and atomic OR is performed. However, this time it is necessary to perform the operations on complete encoded blocks (conformed by more than one 32-bits variable). The *word_position* and *starting_bit* are calculated now for every encoded block. The blocks are copied from global to shared memory where they are shifted according to the *starting_bit* in parallel. Finally the atomic operations are used to perform the logic OR and create the compressed bit stream, see Figure 3.14.

The strategy to perform all these operations in parallel is explained in more detail next.

Calculating the final position of a block in the output buffer

Let p_i be the position of the last codeword written in block i, and let l_i be the number of bits left unused in the 32-bits variable where the codeword in p_i is stored. Let NB be the total number of blocks in the image, computed as:

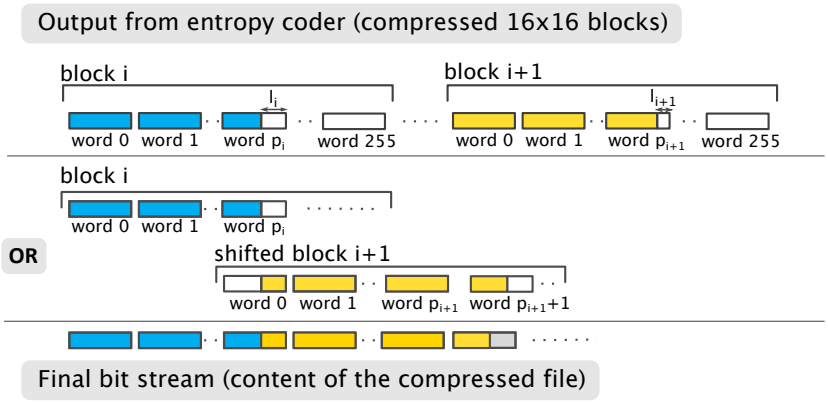


FIGURE 3.14: Bit packing strategy

$$NB = \left\lceil \frac{Nz \times Ny \times Nx}{16 \times 16} \right\rceil \tag{3.17}$$

In order to calculate the number of bits that a block i has to be shifted to the left, a prefix-sum of l_i is performed, obtaining for each block:

$$L_i = \begin{cases} 0 & \text{if } i = 0 \\ \sum_{i=1}^{i-1} l_{i-1} & \text{if } i > 0 \end{cases} \tag{3.18}$$

The number of bits that a block has to be shifted to the left, sh_left_i is then calculated as follows:

$$sh\_left_i = \begin{cases} L_i, & \text{if } L_i < 32 \\ L_i \bmod 32, & \text{if } L_i \geq 32 \end{cases} \tag{3.19}$$

where operator *mod* stands for modulo. The prefix-sum performed to calculate L_i is computed using the same strategy explained in Section 3.5.3.

However, in this case, the number of elements to sum, i.e. the number of blocks NB is, for most hyperspectral images greater than the number of possible CUDA threads allowed by the GPU, which in our case is 1024. Nevertheless, it is possible to perform a prefix-sum of vectors greater than the number of available CUDA threads by following the strategy shown in [100]. This strategy was adapted to our problem as it is explained in next and illustrated in Figure 3.15.

First, the array, whose elements will be prefix-summed, is divided in chunks of 1024 elements. Hence, each chunk has as many elements as the maximum possible number of CUDA threads and is processed by a CUDA block. A kernel performs the prefix-sum of each chunk in parallel. Besides, the summation of all the elements in each chunk is computed and saved to an auxiliary array. Afterwards, the prefix-sum of this auxiliary array is also performed, and the resulting values are added to each element of the prefix-summed chunks.

Let $L = [l_0, l_1, l_2, ..., l_{1024}, ...l_{NB}]$, be the array where the prefix-sum has to be performed, with $NB > 1024$. We divide the array into chunks (CH), resulting in:

$$\begin{aligned} CH_0 &= [l_0, l_1, l_2, ..., l_{1023}] \\ CH_1 &= [l_{1024}, l_{1025}, l_{1026}, ..., l_{2047}] \\ &... \\ CH_{NC-1} &= [l_K, l_{K+1}, l_{K+2}, ..., l_{NB-1}] \end{aligned} \tag{3.20}$$

In the previous equation, NC is the number of chunks, calculated as $NC = \lceil NB/1024 \rceil$. We note that for a better understanding, the indices used in the elements of the chunks are the ones in the original array, L. Symbol

K represents the index of the first element in the last chunk, CH_{NC-1}, calculated as $K = (NC - 1) \times 1024$.

A CUDA kernel of NC blocks and 1024 threads computes the prefix-sum of each chunk in parallel, as in Section 3.5.3. However, in this occasion, the summation of all the elements in a chunk is also saved to an auxiliary array.

CHUNKS AFTER PREFIX-SUM	AUXILIARY ARRAY
$CH_0 = [0, l_0, (l_0 + l_1), ..., \sum_{i=0}^{1022} l_i]$	$\sum_{i=0}^{1023} l_i$
$CH_1 = [0, l_{1024}, (l_{1024} + l_{1025}), ..., \sum_{i=1024}^{2046} l_i]$	$\sum_{i=1024}^{2047} l_i$
...	...
$CH_{NC-1} = [0, l_K, (l_K + l_{K+1}), ..., \sum_{i=K}^{NB-2} l_i]$	$\sum_{i=K}^{NB-1} l_i$

The prefix-sum of the auxiliary array is also performed, resulting in:

AUXILIARY ARRAY AFTER PREFIX SUM
$[0, \sum_{i=0}^{1023} l_i, \sum_{i=0}^{1023} l_i + \sum_{i=1024}^{2047} l_i, ..., \sum_{i=0}^{K-1} l_i]$

Finally, the elements of the prefix-summed auxiliary array are added as scalars to every element of the corresponding chunk, as:

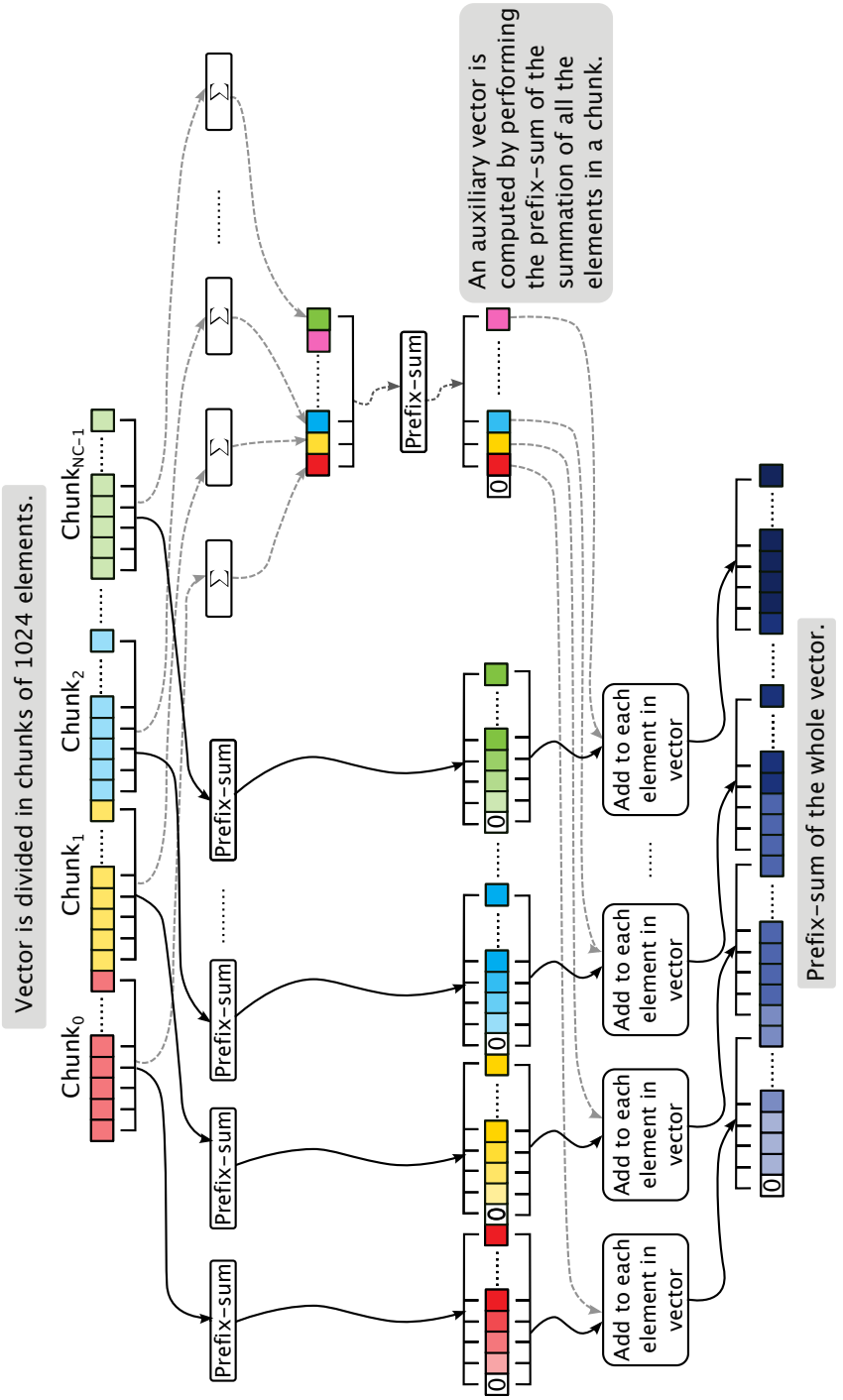


FIGURE 3.15: Prefix-sum of vector with more than 1024 elements.

CHUNKS AFTER PREFIX-SUM		AUXILIARY ARRAY AFTER PREFIX-SUM
$CH_0 = [0, l_0, (l_0 + l_1), ..., \sum_{i=0}^{1022} l_i]$	+	0
$CH_1 = [0, l_{1024}, (l_{1024} + l_{1025}), ..., \sum_{i=1024}^{2046} l_i]$	+	$\sum_{i=0}^{1023} l_i$
...	...	
$CH_{NC-1} = [0, l_K, (l_K + l_{K+1}), ..., \sum_{i=K}^{NB-2} l_i]$	+	$\sum_{i=0}^{K-1} l_i$

This yields the desired prefix-sum of L, and from each element in L, L_i, it is possible to obtain the number of bits a block has to be shifted, sh_left_i.

The final word position in which the block i has to be saved in the compressed output stream is given by $P_i = \sum i = 1^{i-1} p_{i-1}$, where p_i is the position of the last codeword written in block i, as it was already mentioned. Hence, P_i can be obtained also with a prefix-sum. It is important to note that p_i must be updated before performing the prefix-sum, because in some cases, shifting the compressed blocks will make them take one more word, as it happens to block $n+1$ in Figure 3.14. This can be inferred from the resulting sh_left_i and the number of bits left unused by the last codeword of each block, l_i.

$$P_n = \begin{cases} p_i + 1, & \text{if } sh\_left_i = 0 \text{ and } 32 - l_i = 0 \\ p_i + 1, & \text{if } sh\_left_i \neq 0 \text{ and } sh\_left_i \leq 32 - l_i \\ p_i, & \text{otherwise} \end{cases} \tag{3.21}$$

Shifting a block in parallel and saving the final compressed stream

Each of the compressed blocks has to be shifted sh_left_i bits to the left in order to obtain the final compressed stream. A parallel CUDA kernel is

designed to perform these operations. The strategy followed in this kernel is shown in Figure 3.16.

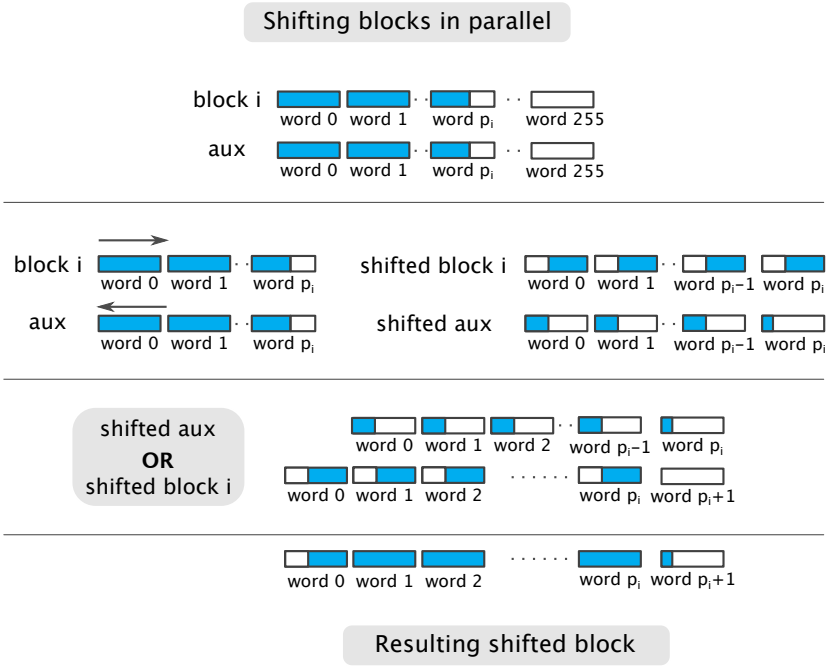


FIGURE 3.16: Shifting compressed blocks in parallel.

First, each CUDA thread creates an auxiliary variable in which an element of the block is copied. Then, each element in the block is shifted in parallel to the right $sh\_right_i = 32 - sh\_left_i$ bits. The auxiliary variables are also shifted the amount of bits established by sh_left_i. Finally, a logical OR is performed between the auxiliary variables and the next element in the block.

After shifting the compressed blocks, the only step remaining is to copy the compressed shifted block to its final position in the compressed stream. The final location is determined by the word position, stored in $P_i = \sum_{i=1}^{i-1} p_i$, therefore the CUDA threads can perform an atomic OR operation in parallel for each element in a block. The final compressed stream is stored in CUDA

global memory, and therefore the atomic OR operations are performed directly to this memory space. Finally, the compressed data are copied to the CPU memory and stored in a file.

3.6 Parallelization of the LCE decompressor

After finishing the CUDA implementation of the LCE compressor, we perform also the implementation of the decompression unit, which has basically the same stages of the compressor, but in inverse order, also processing independent 16×16 blocks with all bands. Having a hyperspectral image decompressor accelerated on a GPU presents several advantages which can be exploited nowadays in real applications since, unlike the compression, the decompression is not performed on-board but on the ground station, where any commercial GPU can be utilized.

The main function in the decompressor software consists of a nested loop, with the same structure of the nested loop encountered in the compressor. First, the compressed data are read in order from the compressed file, decoded and inverse mapped to obtain the quantized prediction error samples. For the first band, an inverse quantizer is employed and afterwards, inverse 2D prediction is applied. For all other bands, a least-mean-squares predictor for the current band is computed from parameters α' and m, which are also read from the compressed file, using the previous band as reference.

The prediction error residuals are then dequantized to obtain the prediction error samples, which are added to the predictor values, yielding the reconstructed samples. The output file contains the reconstructed image in exactly the same format as the input to the compressor.

3.6.1 Preliminary considerations

The main drawback encountered when trying to parallelize the LCE decompressor algorithm lies in the fact that the codewords have been buffered to the compressed file one after the other, in a bit-by-bit fashion and it is impossible to know when a compressed block starts. Therefore, in principle, although the blocks can be decoded independently, the compressed file must be read sequentially to obtain the codewords for every 16×16 block.

There are several reasons why it is not feasible to find out where the compressed blocks start in the compressed file:

- The codewords are of variable length, therefore the compressed 16×16 blocks have different sizes in bits.
- A single block contains codewords obtained in different ways:
  - α, μ_z and the first sample of the first band are not encoded.
  - The first sample of blocks in band $z > 0$ is encoded using an exponential Golomb code.
  - The rest of the samples are encoded using a Golomb code.

These facts make it impossible to split the compressed file in order to obtain the different decompressed blocks in parallel without additional information about the location of the blocks in the compressed bit stream. In order to solve this issue, it is proposed to add a header to the compressed file, which contains information about the size in bits of each compressed block. The downside of adding a header is that it increases the size of the compressed file, and therefore reduces the compression ratio. Consequently, the header must be designed in such a way that it allows achieving more parallelization, without affecting the compression ratio considerably. The amount of acceleration that can be obtained by adding the header can be calculated taking

into account the number of blocks that will be processed in parallel and the number of iterations needed to decode each block.

3.6.2 Header design

Two different options are explored before adding a header to the compressed file. For simplicity and as a worst case scenario, it is considered that the header is saved at the beginning of the compressed file, storing each value in a 32-bits word.

The impact of adding a header to the compressed image is evaluated in terms of:

- The increment (I) in bits of the size of the compressed file, which can be calculated as:
$$I = Size\ of\ header \times 32 \tag{3.22}$$
- The number of elements that can be processed in parallel if the header is added (P_{BLOCKS}), which gives an idea of the amount of acceleration that will be obtained.
- The number of sequential iterations (SI) that are necessary to decode each parallel element.

Taking into account these facts, a figure of merit (FM) is defined:

$$FM = \frac{P_{BLOCKS}}{I \times SI} \tag{3.23}$$

The figure of merit is evaluated for each of the proposed options, in order to find out which of them is able to potentially produce a higher speedup with the lowest possible impact on the compression ratio.

Option1

In the first option explored, the header contains the size in bits of every 16×16 spatial block with all its bands, as shown in Figure 3.17. For this option, the size of the compressed file increases proportionally to the spatial dimension of the hyperspectral image.

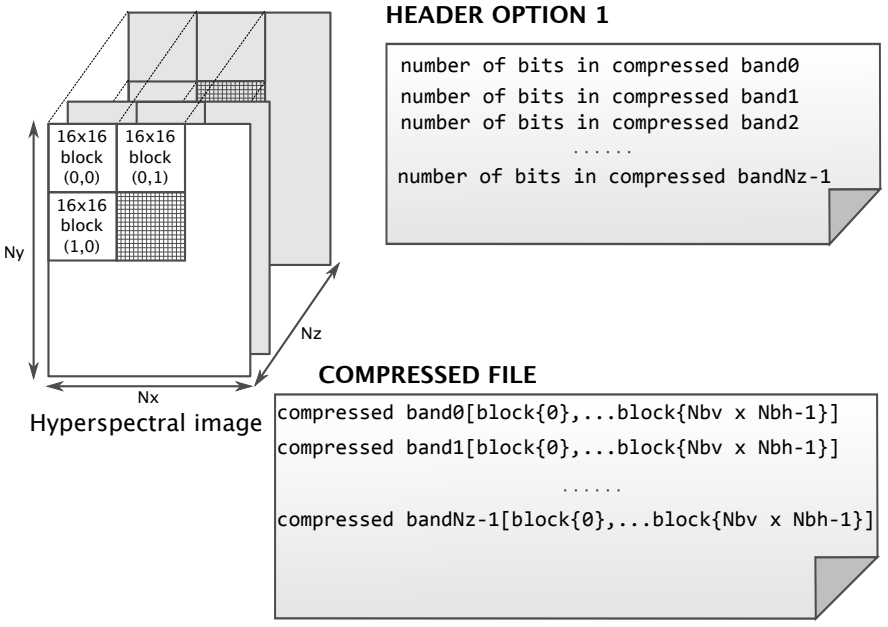


FIGURE 3.17: Format of header and compressed file for Option1.

The increment of the compressed file in this case is calculated as:

$$I = Size\ of\ header \times 32 = \frac{Ny \times Nx}{16 \times 16} \times 32 \tag{3.24}$$

where N_y and N_x denote the number of lines and columns in the hyperspectral cube respectively.

The number of elements which can be processed in parallel corresponds to the number of 16×16 spatial blocks with all their bands present in the hyperspectral image.

$$P_{BLOCKS} = \frac{Ny \times Nx}{16 \times 16} \tag{3.25}$$

The number of sequential iterations which are necessary to decode the compressed buffer is the number of bands in the image, Nz:

$$SI = Nz \tag{3.26}$$

Finally, the figure of merit (FM) is calculated, yielding the result:

$$FM_{OPT1} = \frac{1}{32 \times Nz} \tag{3.27}$$

Option2

In the second option, it is proposed that the header contains the size in bits of all 16×16 spatial blocks in a specific band, resulting in a header with the format shown in Figure 3.18. With this header, the spatial blocks in a band can be decoded in parallel. The header contains a value for every band in the image, each saved in a 32-bits variable.

Adding the described header involves making additional changes to the encoder, since the order in which the codewords are saved to the compressed file must be altered. Specifically, as it is shown in Figure 3.18, it must contain the codewords which represent a compressed band with all its spatial blocks in order. The changes must be made to the compressor's main nested loop, which must be modified so that all horizontal and vertical 16×16 spatial blocks in a band are processed before the compression of the next band begins. Additional changes might be needed in order to guarantee

that the necessary information from the previous band is available for the compression of a specific band.

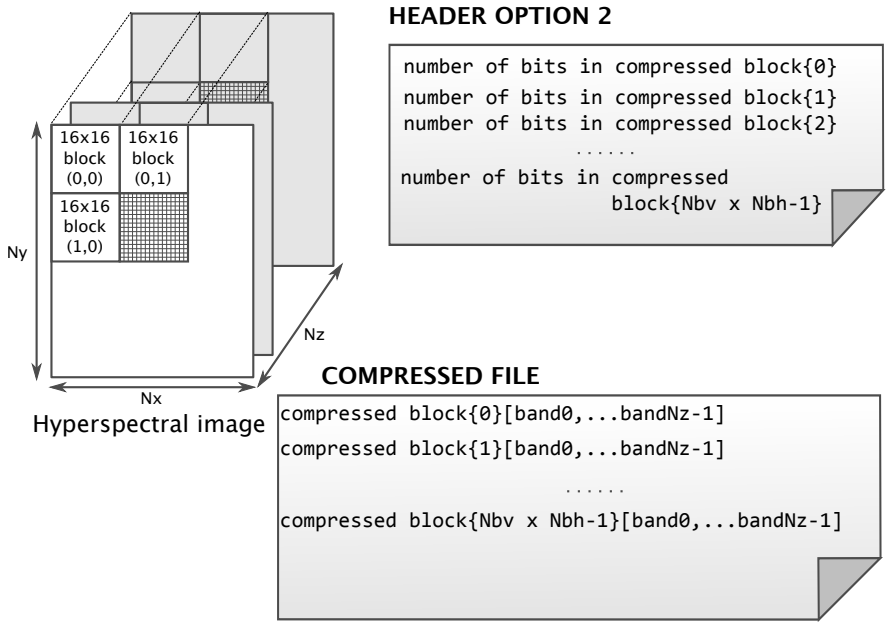


FIGURE 3.18: Format of header and compressed file for Option2.

The increase of size (I) in the compressed file and the number of parallel elements (P_BLOCKS) for this case are:

$$I = Size\ of\ header \times 32 = Nz \times 32 \tag{3.28}$$

$$P_{BLOCKS} = Nz \tag{3.29}$$

The number of sequential iterations (SI) which have to be performed by the decoder for every parallel element corresponds to the number of spatial 16×16 blocks, given by:

$$SI = \frac{Ny \times Nx}{16 \times 16} \tag{3.30}$$

Yielding a figure of merit of:

$$FM_{OPT2} = \frac{8}{Ny \times Nx} \tag{3.31}$$

Comparing the figures of merit resulting from both options, it can be observed that the best trade-off between the increase of the compression ratio and the potential speedup achieved depends basically on the image dimensions. For images which are spatially big and do not comprise a very high number of bands, which is the case of multispectral and hyperspectral images, Option1 is more convenient. However, for ultraspectral images, which can contain thousands of bands and are usually smaller in the spatial dimensions, Option2 should be considered.

It is decided to adopt Option1 for the reasons stated below:

- Most of the sensors are multispectral or hyperspectral, therefore performance is expected to be better with Option1.
- Other than adding the header to the compressed file, no changes need to be made to the original LCE compressor.

3.6.3 Decoding the blocks

The first stage of the LCE parallel decompressor is to read and decode the compressed file. The header containing the size in bits of each compressed 16×16 block with all its bands is attached to the compressed bit stream to make it possible to parallelize this stage.

With the information in the header, the exact location of the first codeword of each 16×16 compressed block can be calculated performing a summation. A kernel is created to read and decode every compressed block in parallel in such a way that once a block is read from the compressed buffer, the kernel iterates sequentially to decode all the codewords. These iterations have to cover all bands and all samples in a 16×16 block. As it will be demonstrated in the experimental results (Section 3.7) this loop represents the main performance weakness of the parallel LCE decompressor.

After the execution of this kernel, parameters $\hat{\alpha}$, μ_z and all the decoded prediction errors are saved to the CUDA global memory so that the inverse predictor can be applied to them.

3.6.4 Inverse quantization and prediction

The kernel designed to perform the inverse quantization and prediction is almost the same as the one designed for the LCE CUDA compressor, however, it performs the inverse operations in the inverse order. As for the compressor, the inverse quantization and prediction kernel is designed in such a way that a CUDA block is launched for every block in the image and every CUDA block launches 256 (16×16) threads.

3.7 Experimental results

The CUDA implementation of the LCE compressor and decompressor are executed on an Nvidia Tesla C2075 GPU. The experiments are designed in such a way that it is possible to assess several aspects: the accuracy of the GPU implementation; the acceleration achieved when compared with the execution of the algorithm in a single threaded CPU; the relationship

TABLE 3.2: Hyperspectral images under compression

Sensor	**Area**	**Nz**	**Ny**	**Nx**	**bpppb**
MODIS	-	17	1984	1344	12
AVIRIS	Indian Pines	220	1952	608	16
AIRS	Granule	1501	128	80	14

between the achieved acceleration and the hyperspectral image dimensions and the configuration parameters of the LCE algorithm.

Three different hyperspectral scenes, acquired by different sensors and with different spatial and spectral sizes, are used as targets for compression. Table 5.11 summarizes the main characteristics of the images: the sensor which acquired them; the number of lines (Ny), columns (Nx) and bands (Nz); and the number of bits utilized to represent the raw samples (bpppb).

All the experiments are performed in a worskstation with a 3.19 GHz Intel Xeon W5580 processor, running on a 64-bits operating system with 12 GBytes of RAM memory. The LCE algorithm implementation operates with integer variables only, except for the quantization stage, where it is necessary to utilize double precision floating point variables when the uniform-threshold quantizer (UTQ) is enabled.

3.7.1 Validation

Before any experiment is performed, it is mandatory to verify that the GPU implementations of both the compressor and decompressor produce the same results as their CPU version counterpart. For this purpose, the images are compressed and decompressed with the GPU and the CPU implementations. The resulting compressed and reconstructed files are then compared bit by bit, demonstrating that the results of both implementations are identical.

3.7.2 Impact of adding a header

Once it is verified that the GPU versions of the LCE algorithm produce the correct results, we assess the impact in the compression ratio of adding a header to the compressed file, in order to demonstrate the validity of the decisions adopted in Section 3.6.2.

The compression strength of the LCE algorithm can be configured by the user by setting a parameter *delta*. Low values of *delta* yield low compression while a high *delta* value increases the compression ratio at the cost of producing more losses of information. The impact of adding a header is more significant if the compression ratio is high, i.e. the compressed file is smaller, hence, each hyperspectral image under test is compressed with two different values of *delta*, namely 1 and 60 in order to perform a fair evaluation.

The compression ratio (CR) in bits per pixel per band (bpppb) is calculated as:

$$CR\ (bpppb) = \frac{Size\ of\ compressed\ file\ (bits)}{Nz \times Ny \times Nx} \tag{3.32}$$

Moreover, the percentage of increment in the compression ratio ($CR\ increment$) is calculated as:

$$CR\ increment(\%) = \frac{Size\ of\ header\ (bits)}{Size\ of\ compressed\ file\ (bits)} \times 100 \tag{3.33}$$

It is observed in Table 3.3 that, although adding a header to the compressed file increases the compression ratio, the impact produced is not considerable, with an average of 0.06% and a maximum of 0.20% for all the images under test.

TABLE 3.3: Impact of adding a header to the compressed file

Sensor	**Size** ($Nz \times Ny \times Nx$)	**delta**	**CR without header**	**CR with header**	**Increment (%)**
MODIS	$17 \times 1984 \times 1344$	1	7.3352	7.3426	0.10
		60	3.9538	3.9612	0.19
AVIRIS	$220 \times 1952 \times 608$	1	5.8970	5.8976	0.01
		60	2.3604	2.3610	0.02
AIRS	$1501 \times 125 \times 80$	1	4.3474	4.3475	0.00
		60	0.8704	0.8704	0.01

3.7.3 Profiling

The GPU implementation of the LCE compressor and decompressor are profiled with the software tools supplied by Nvidia. For this purpose a 512×512 subimage of the AVIRIS scene, which comprises 220 bands, is employed. This hyperspectral cube is compressed with the GPU implementations and profiled with the tools supplied by Nvidia, in order to detect which of the kernels is the most time-consuming and where are the main bottlenecks of the implementation.

GPU compressor profiling

Figure 3.19 shows the profiling of the LCE compressor. The total time to perform the compression is 394 ms, which is almost a quarter of the time achieved with the GPU implementation of JPEG2000 presented in [19]. The predictor takes a 23.29%, what shows that, although the predictor has to loop to cover all bands in the hyperspectral cube, this fact is not a bottleneck for global performance. The most time-consuming operations for both GPUs are the entropy coding (44.24%), and the memory transactions between the CPU and the GPU. The strategy designed for the bit packing, which is used to create the final bit stream shows very good results, taking only 4.73% of the total compression time.

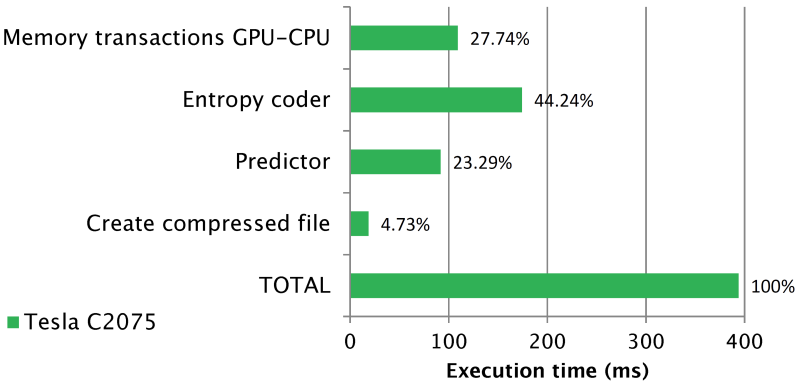


FIGURE 3.19: Profiling of the CUDA implementation of the LCE compressor.

Afterwards, the execution times of the prediction, entropy coding and bit packing stages of the GPU implementation are compared with the corresponding stages in the CPU implementation. The results in Figure 3.20 show the high acceleration obtained with the GPU implementation in both stages. The strategy designed to parallelize the entropy coding and bit packing stages exhibits very good results. This stage is executed more than ten times faster in the GPU than in the CPU.

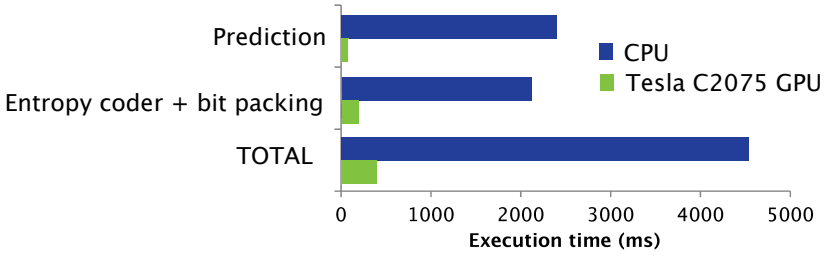


FIGURE 3.20: Comparison between the CPU profiling and the GPU profiling

GPU decompressor profiling

Similar results are shown in Figure 3.21 for the GPU decompressor. Nevertheless, the total time necessary to perform the decompression is higher than the GPU compression, mainly because of the time spent by the entropy decoding kernel, which is 864.20 ms, a 78.62% of the total time, and approximately 5 times the time spent by the entropy coding kernel in the GPU compressor. It is observed that the difficulties existing when parallelizing the entropy decoding stage, which were partially solved with the addition of a header to the compressed file, cause the GPU decompressor to achieve a smaller speedup than the one obtained by the compressor.

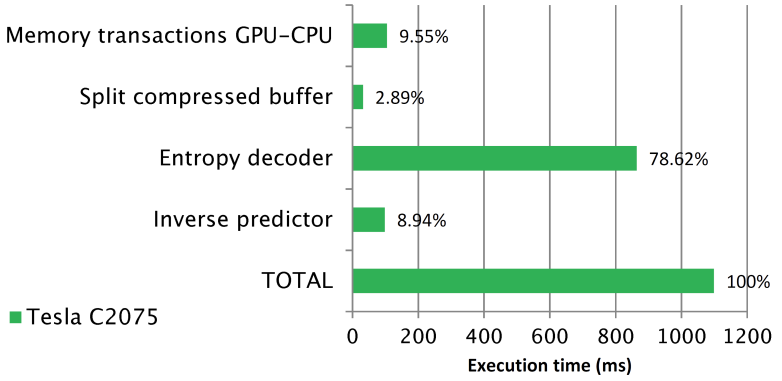


FIGURE 3.21: GPU decompressor profiling

3.7.4 Speedup

After profiling the GPU implementation of the LCE compressor and decompressor, their performance is evaluated in terms of the speedup achieved with respect to the CPU implementation. The compression time of the CPU implementation is measured with the CPU timers, whereas the compression

time of the GPU implementation is obtained with the timers provided by the CUDA environment.

Several subimages of the available hyperspectral cubes are created with different spatial sizes, ranging from 64×64 to their maximum original size. Each hyperspectral image is compressed and decompressed with the CPU implementation of the LCE algorithm and with the GPU implementation. Afterwards, the speedup is calculated for both, compressor and decompressor as:

$$Speedup = \frac{Total\ CPU\ time}{Total\ GPU\ time} \tag{3.34}$$

Table 3.4 show the range of widths and lengths selected for generating the subsets of hyperspectral images, and the average speedup achieved. It can be observed, that high speedup is achieved for the GPU compressor, of up to an average of 15.41. Nevertheless, although speedup is also obtained for the GPU decompressor, it is much lower than that of the GPU compressor. The reason for this lies in the limitations found when parallelizing the entropy decoder of the decompressor. Although adding a header helps to solve this issue, a lot of data still has to be read from the compressed bit stream sequentially. The worse speedup results of the decompressor are obtained for AIRS, which is an expected result, taking into consideration the fact that the image from this sensor is ultraspectral and hence has a high number of bands but is spatially smaller than the others. Therefore, the header added to this particular image does not permit to have as many parallel elements and a lot of iterations are needed to perform the entropy decoding.

TABLE 3.4: GPU compressor and decompressor speedup

Sensor	Range		Nz	Average Speedup	
	Ny	Nx		Compressor	Decompressor
MODIS	64-1984	64-1344	17	12.50	2.95
AVIRIS	64-1952	64-608	220	15.41	5.82
AIRS	64-128	64-80	1501	13.35	1.32

3.7.5 Throughput

The performance is now evaluated in terms of throughput, defined as the number of samples computed per second for all the hyperspectral images under study.

$$\begin{matrix} Throughput \\ (MSamples/sec) \end{matrix} = 10^{-6} \times \frac{Nz \times Ny \times Nx}{total\ execution\ time(sec.)} \tag{3.35}$$

Figures 3.22, 3.23 and 3.24 show the number of samples computed per second, which are achieved by the GPU and CPU for the different images under test, against the total number of samples. The worst performance results are obtained when the number of elements in the image is low, because this implies that the number of blocks which can be computed in parallel is also small, and therefore the utilization of the multiprocessors of the GPU is poor. However, as the number of samples to be compressed increases, the performance of the GPU gets better until it becomes almost stable. Although the performance of the kernel increases with the number of samples, the time necessary to transfer the image from the CPU to the GPU is still proportional to the amount of samples, what stabilizes the performance.

The performance of the compressor is in general better than that of the decompressor. This is caused by the fact that there are more operations

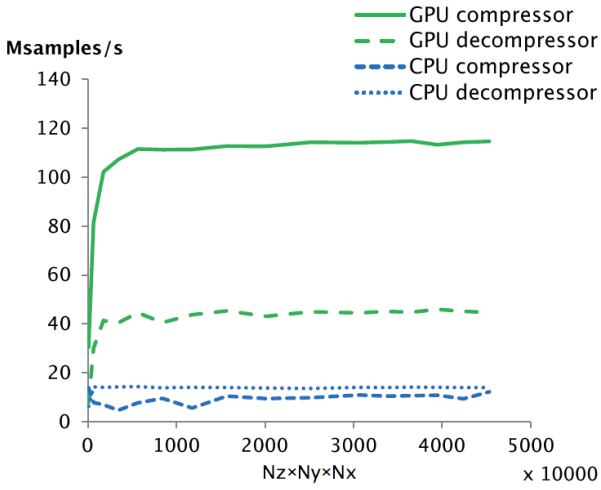


FIGURE 3.22: Number of samples computed per second against number of samples for the MODIS hyperspectral image

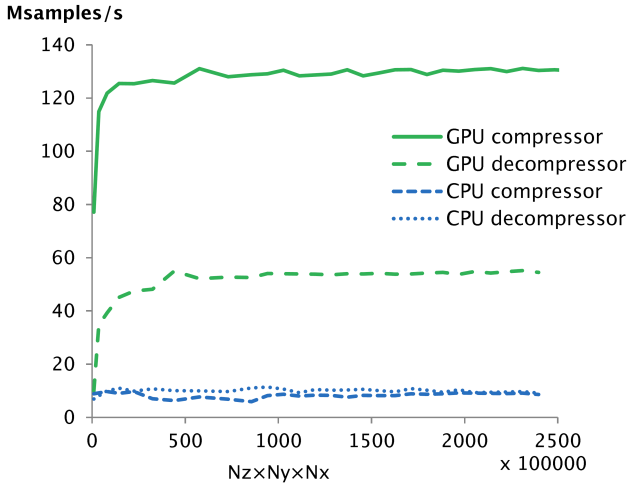


FIGURE 3.23: Number of samples computed per second against number of samples for the AVIRIS hyperspectral image

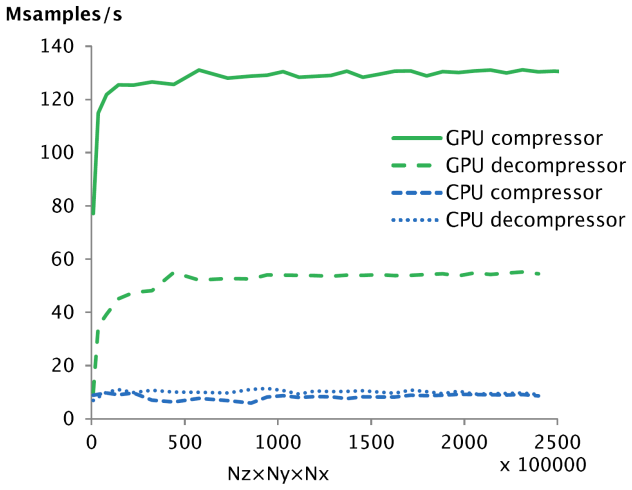


FIGURE 3.24: Number of samples computed per second against number of samples for the AIRS hyperspectral image

which can be programmed to be executed in parallel in the GPU compressor, as the parallelization of the decompressor is limited by the bit unpacking and the entropy decoding stages.

Specifically, for the AIRS image it is observed that the performance of the GPU decompressor is much worse than that of the other images under test; showing even higher compression times than the CPU for the smallest image of 64×64 pixels. Calculating the figures of merit for the different header options shown in Section 3.6.2, it is observed that the use of Option 2 for the header represents a better option for this image, what should be taken into account in case the GPU decompressor is developed for an image with similar spatial and spectral dimensions.

3.7.6 Effect of the configuration parameters in the performance of the GPU implementation of the LCE compressor

Finally, the effect of the configuration parameters of the LCE compressor in the performance of both implementations is presented. The parameters UTQ and *delta* can have an impact in the compression time of both, the CPU and the GPU implementations of the algorithm. In the following, it is analysed how the variation of these parameters affects the performance of the GPU implementation, with respect to the CPU implementation.

To evaluate the effect of varying UTQ, the set of images shown in Table 3.4 is compressed with the CPU and GPU implementation with UTQ = 1 and then with UTQ = 0. Besides, two possible configurations of *delta* are set, namely *delta*=1, which provides the best possible results in terms of quality and the lowest compression ratio; and *delta* = 60, which decreases the quality and increases the compression ratio.

In order to better asses the effect of the parameters, the performance is evaluated in terms of the number of samples computed per second, calculated with Equation 3.35.

This value is calculated for all the sub-images, and the average is computed. The results are summarized in Figure 3.25.

Both, the GPU and CPU show better results when UTQ is disabled, since, with UTQ enabled, double precision operations have to be performed in the quantization stage of the LCE compressor. On the other hand, for parameter *delta*, it is observed that when *delta* is high, the performance improves, and this improvement is more noticeable in the CPU implementation, particularly for the AIRS image. This is explained by the fact that, when *delta* is increased, the number of blocks which skip quantization is also high, and

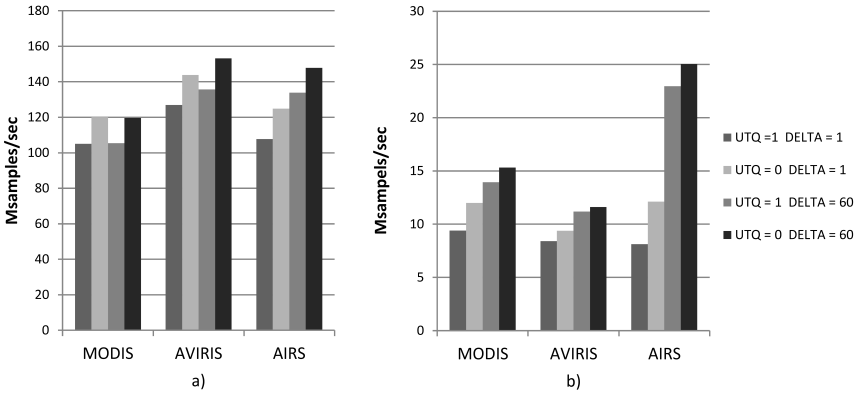


FIGURE 3.25: Effect of the configuration parameters of the LCE algorithm in the performance of the GPU compressor implementation (a) and the CPU implementation (b)

therefore the CPU implementation can avoid part of the operations and complete the processing of the image in a shorter time. Although the number of skipped block is the same for the GPU implementation, since the data are processed in parallel, the fact that some of the blocks can be skipped does not make a significant difference in the processing time of the whole hyperspectral cube.

To better depict this fact, in Figure 3.26, the 512×512 AVIRIS image with 220 bands is compressed with the CPU and the GPU implementation of the LCE algorithm for increasing values of *delta*, starting with $delta = 1$ up to $delta = 100$. It can be observed that for both implementations, shorter compression times are achieved when *delta* is higher and that the number of samples computed per second for the GPU implementation is always around 10 times greater than the number of samples per second achieved by the CPU. It can be observed that the variation of the number of samples computed per second is more noticeable for the CPU implementation. Nonetheless, even for higher values of *delta*, the performance of the GPU is anyhow significantly better than that of the CPU.

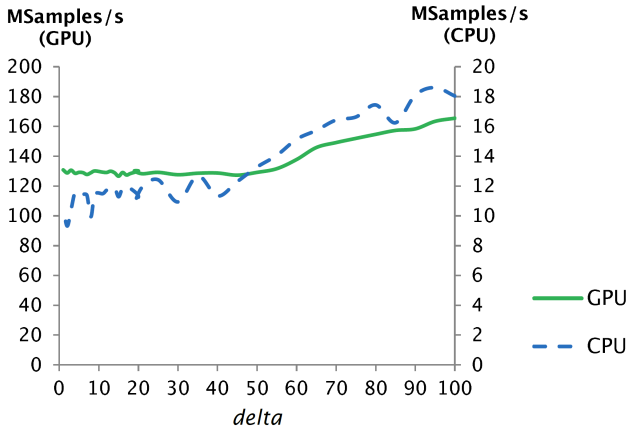


FIGURE 3.26: Effect of varying *delta* in the performance of the GPU and the CPU for the AVIRIS image)

Chapter 4

Implementation of a lossy compression algorithm for hyperspectral images on an FPGA

In this Chapter, we obtain an FPGA implementation of the lossy compression algorithm for hyperspectral images known as LCE, whose GPU implementation was presented in Chapter 3. The RTL description of the algorithm is obtained with high-level synthesis tools. The experimental results presented demonstrate the suitability of the LCE algorithm for an FPGA implementation and furthermore make it possible to assess and compare the potential features of both, GPU and FPGA technologies for on-board data compression.

4.1 Outline

In Chapter 3, the implementation of the LCE compressor and its decompression counterpart have been implemented on a GPU, showing a fair amount of paralellization that yields a high acceleration. GPUs are, to the date, not suited to operate on-board a satellite. To demonstrate the benefits of the LCE algorithm for on-board compression, an implementation in a different technology is needed. We opt to implement the algorithm on an FPGA for several reasons: to benefit from parallelization inherent to the LCE algorithm; FPGAs provide faster developments and less expensive designs than ASICs; and there are space-qualified options (unlike GPUs). Moreover, FPGAs they offer flexibility through reconfiguration.

We present in what follows an implementation of the LCE algorithm on an FPGA. Although FPGAs are very attractive for on-board compression, obtaining an efficient implementation usually involves following a rather long design flow. Furthermore, most implementations are usually optimized for a specific technology, what forces the developers to spend a long time adapting the original design whenever changes or improvements are necessary. In this sense, high-level synthesis (HLS) tools represent an interesting option they make it possible to reduce the times necessary for obtaining an efficient hardware description of a digital design suitable for an FPGA or ASIC implementation. This is the case of the popular tool known as CatapultC [101], which offers the possibility to generate RTL implementations from C or C++ specifications with little restrictions. Moreover, it increases the flexibility of the implementations, enabling the exploration of different optimizations or target technologies without the need for the developer to perform changes in the original C or C++ source code. CatapultC will also generate a testbench together with the description of the user's design, so that it is possible to run simulations with ModelSim and verify the correct behaviour of the generated RTL.

The LCE algorithm which is our target of study in this Chapter (see Section 3.2) is a good candidate for an FPGA implementation, because of its low complexity, the fact that it can operate on integer variables only and its parallelization capabilities. Likewise, it is a good example for obtaining an RTL description with HLS tools, since a working and verified software implementation in C language was available prior to this study. During our work, several modifications are made to the C source code of the LCE compressor software in order to make it possible for CatapultC to generate an efficient implementation of the algorithm. Although these alterations are specifically performed for the LCE algorithm, most of the ideas and strategies devised can be applied to other prediction-based adaptive compression algorithms. Moreover, general recommendations regarding the HLS design flow can be inferred from this work, since the whole experience serves as a case of use demonstrator.

In the next sections of this Chapter, an FPGA implementation of the LCE algorithm is obtained with CatapultC, to demonstrate its features for on-board compression, providing a quantitative idea of its low-complexity and suitability for an FPGA implementation in terms of area, latency and throughput; and at the same time showing how CatapultC can be used to generate efficient hardware implementations and shorten the design flow. HLS has been proven to be useful for obtaining synthesizable hardware blocks in a short time [102] [103]. As it was mentioned in Section 3.2, the LCE algorithm consists of several differentiated stages with their corresponding data dependencies. The C source code utilized as input for CatapultC contains the description of all the compressor stages. This serves also to assess how well CatapultC performs for a rather complex system as it is the case of a compression algorithm. Most of the examples given in the CatapultC documentation are rather small designs which comprise a few lines of C language code. The LCE compressor is a good example to evaluate the suitability of HLS for these kind of problems.

4.2 CatapultC design flow

CatapultC [101] is a synthesis tool which provides implementations from C or C++ working specifications, generating RTL descriptions, netlists (Verilog, VHDL and SystemC), simulation scripts, schematics and reports. The output generated by CatapultC can be synthesized, mapped and placed and routed on an FPGA using synthesis tools such as Mentor Graphics Precision or Synplify. This study is focused on obtaining a synthesizable description in VHDL from a source code which is written in C language.

The user is guided through the CatapultC design methodology by following a sequence of specific steps, which are shown in Figure 4.1 and are briefly explained next.

Step 1: Writing and testing the C/C++ source code

The way in which the C language source code of the algorithm is written affects the final result more than any other step in the CatapultC design flow. CatapultC is able to generate a hardware implementation from almost any algorithm written in C or C++ with little restrictions, e.g. dynamic memory allocation cannot be utilized. However, in order to optimize the results or meet specific requirements in terms of area, throughput or latency, it is necessary to carefully design the C code and understand how CatapultC interprets it. In the specific case of the LCE algorithm, several minor and major changes had to be done to the original C language code in order to obtain an optimized implementation with CatapultC. These changes are detailed in Section 4.3.

Step 2: Setting global hardware constraints

The global hardware constraints are mainly the clock frequency, reset and enable behaviour of the design, and the target technology, which must be determined by the user. The user has to identify, likewise, the main function

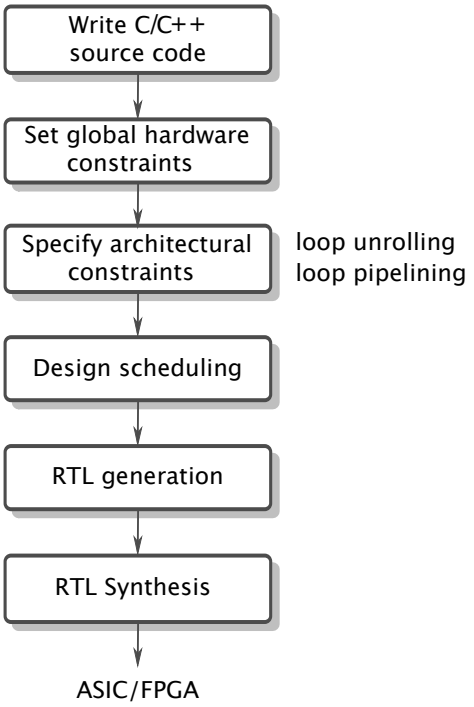


FIGURE 4.1: CatapultC design flow

of the C language code, which will be the top module of the generated design. All functions called from inside the main function are considered part of the hardware design, while those functions outside the top design module are considered part of the testbench and no VHDL code is generated for them.

Step 3: Specifying architectural constraints

Subsequently, the user selects the architectural constraints, i.e. maps arrays to memory resources; decides how memories are mapped; optimizes loops; identifies inputs, outputs and global resources or controls the input and output interfaces. The most remarkable feature at this step is the loop optimization by means of loop unrolling and loop pipelining.

CatapultC allows partial or full unrolling of loops, making it possible for the user to decide how many times a loop is replicated, as shown in Figure 4.2. If there are no data dependencies between iterations, i.e. the individual iterations can be executed in parallel, loop unrolling can reduce the latency and increase the throughput of the resulting implementation at the cost of increased area. It has to be noted that CatapultC always respects the data dependencies. If the user asks the tool to unroll a loop where data dependencies exist between the iterations, CatapultC makes an effort to unroll it anyway, as long as it has enough hardware resources to replicate the iterations. The result in this case is a repetition of hardware without any parallelism and therefore no benefits in terms of latency and throughput are achieved.

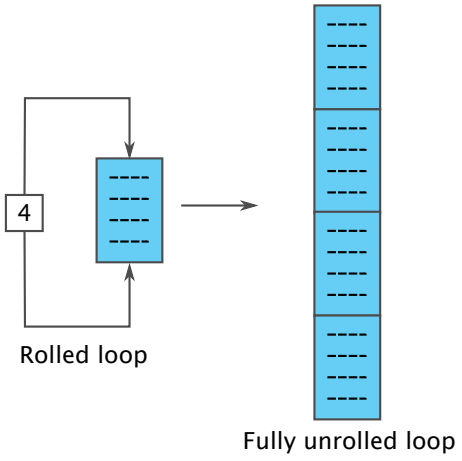


FIGURE 4.2: Setting architectural constraints: loop unrolling.

On the other hand, when a loop is pipelined, see Figure 4.3, the second iteration starts before the first one has finished. CatapultC provides a parameter called *Initiation Interval (II)*, which sets the number of cycles the second iteration waits before it starts. To get the maximum possible throughput,

II should be set to 1. Loop pipelining can fail for the selected *II* for two reasons: there are not enough hardware resources in the target technology to implement the pipelined loop or data dependencies cannot be guaranteed. In order to solve this problem, the user can try to increase the *II* .

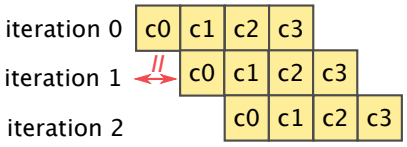


FIGURE 4.3: Setting architectural constraints: loop pipelining.

Pipelining can be combined with partial unrolling, in such a way that the body of a loop is replicated several times and the remaining iterations in each of the replicated loops are pipelined.

Furthermore, the number of iterations in the loop must be known to get optimal implementations and an accurate estimation of the latency of the design. If the number of iterations is variable, it must be set to the maximum possible value.

Step 4: Scheduling the design The design is scheduled by CatapultC according to the given constraints. A Gantt chart showing also the data dependencies is generated, what gives the user valuable information in order to further optimize the loops and options set in the previous steps.

Step 5: Generate RTL After the scheduling, the RTL files are generated, which can be used to perform the synthesis on the ASIC or FPGA chosen by the user. CatapultC provides a direct link to the Mentor Graphics Precision tool, which can be used to perform the synthesis, mapping and placement and routing.

4.3 Adapting the C language source code of the LCE algorithm for CatapultC

Several changes are made to the original C language source code of the LCE algorithm in order to obtain an optimized hardware implementation of the design with CatapultC. The original C implementation is used as golden reference, to ensure the correctness of the C code modified for CatapultC.

First, the data types are changed to Algorithmic-C bit-accurate data types, which allow the user to create variables of any bit width and also determine if they are signed or unsigned. When these data types are used, CatapultC has more accurate information about the hardware resources necessary to perform the different operations in the design, and is able to find better optimizations.

Furthermore, the size of all the data arrays in the design is determined and set to fixed values, i.e. all the dynamic memory allocation sentences are eliminated from the source code, as they cannot be interpreted by CatapultC. The present global variables in the source code are removed and transformed to parameters to the different functions.

The code is also analysed to make sure there are no redundant sentences or lines of code written for debug purposes or to make the code more understandable. The variables whose value does not change during the execution time are turned into constants, to make it easier for CatapultC to interpret them.

4.3.1 Identification of the top function and inputs and outputs of the design

The original C implementation of the LCE algorithm reads the whole hyperspectral image and then compresses it block by block. As the 16×16 blocks with all bands are independent, it is decided to create a hardware implementation which would perform the compression of a 16×16 block of pixels in a specific band. This way, the user can have the flexibility to use the designed core for compressing the complete images by using several instances according to the specific throughput or area needs.

Consequently, the top design module of the LCE hardware implementation compresses a 16×16 block of pixels in a specific band. The top function in the C implementation modified for CatapultC has to perform exactly those operations, as it is interpreted as the top module. A function called *pred1block()* is created in the C source code to be the top function of the implementation. The pseudo-code of the top function is shown in Figure 4.4.

```
void pred1block (int cur_block[256], int ref_block[256], int* filecount,
                 int* pp, int* m, int* ym) {
    if (z == 0) {
        2D-prediction;
        quantization and mapping;
        entropy coding; }
    else {
        spectral prediction;
        rd-optimization;
        quantization and mapping;
        entropy coding;}
}
```

FIGURE 4.4: Pseudo-code of the C source code containing the top function for the hardware implementation of the LCE algorithm

The parameters of the function *pred1block()* are interpreted by CatapultC as the input and output ports of the top design. CatapultC decides if a parameter should become input (I), output (O) or input/output (I/O) depending on

how they are utilized inside the function. The parameter *cur_block*[256] contains the input pixels in a 16×16 block; *ref_block*[256] contains the decoded pixels, which are necessary perform the spectral prediction in the next band. It is decided to map these arrays to RAM memory. The variable *ym* contains the mean value of the samples in the current band, which will be also used for the spectral prediction in the next band. The parameter *block_out*[256] is also mapped to RAM and stores the output results after the entropy coding phase.

The top design module *pred1block* performs the prediction, quantization and mapping of the pixels in *cur_block*[256]. In the entropy coding phase, the codewords are generated. These codewords represent the compressed pixels and have to be saved in raster order in a bit by bit fashion, not leaving any bits unused between codewords. For this purpose, and considering that the generated codewords have variable length, parameters *pp* and *m* are utilized. The parameter *pp* is a 32-bit buffer where the codewords are saved and *pp* indicates the number of unused bits in *pp*. Only when *pp* is full, it is saved to the output memory. After the compression of one block, *pp* might be still partially full, and the first codeword generated in the compression of the next block must be also buffered in it. The parameter *filecount* is used to indicate how many 32-bit words are written in the output memory. It must be noticed, that, although the compression of the 16×16 blocks with all bands can be performed independently, the core *pred1block* must have finished writing in *pp* and *m* and updating *filecount* before the first codeword of the next block of 16×16 pixels is generated. The resulting top design with its inputs and outputs is shown in Figure 4.5. A summary of the inputs and outputs of the design is presented in Table 4.1, showing the size and a description of each port, and if it is mapped to a RAM memory.

TABLE 4.1: Input and output ports of the top module

Port	Direction	Size(bits)	Mapped to	Description
curr_block	I	256 × 16	Dual port RAM	16 × 16 samples of an uncompressed block.
ref_block	I/O	256 × 16	Dual port RAM	16 × 16 decoded values of the previously compressed band.
pp	I/O	32		Buffer to store the generated codewords.
m	I/O	6		Number of bits used in the buffer (pp).
ym	I/O	16		Mean of the samples a band.
filecount	I/O	32		Number of 32-bits words written in the output memory.
block_out	O	256 × 32	Dual port RAM	Compressed data.

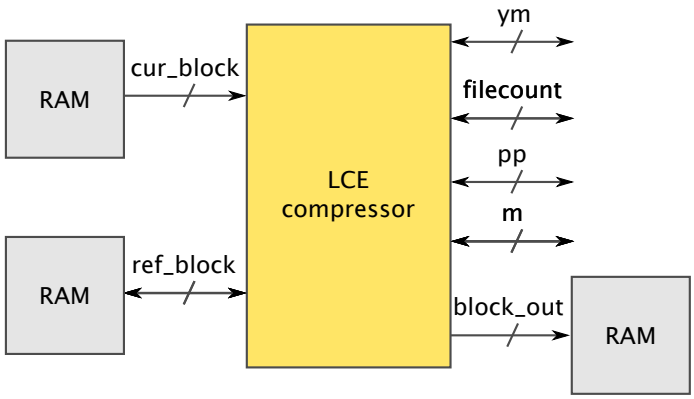


FIGURE 4.5: Top design with its inputs and outputs

TABLE 4.2: Constants of the FPGA implementation of the LCE algorithm

Constant	**Description**
NB	Number of bands in the image.
RI	Number of lines.
CI	Number of columns.
signed	Input data are represented with signed integers (1) or unsigned (0).
UTQ	Choose whether to use or not the UTQ quantizer.
delta	Strength of the compression.
DB	Number of samples in a spatial block. Set to 256.

4.3.2 Configuration parameters

As it was mentioned in the previous Sections, the LCE algorithm allows the user to set several parameters. It is necessary for the algorithm that the user provides the number of samples and dimensions of the hyperspectral cube, as well as how the raw data are represented. Furthermore, the user can set two configuration parameters of the algorithm: the type of quantization, by enabling or disabling the UTQ; and the strength of the compression, by setting parameter *delta*. These parameters do not change their value during the execution of the algorithm. However, in the C language implementation they are represented as global variables. From the point of view of the hardware design, it does not make sense to have these parameters as I/O ports of the design, because they will always have the same value. Therefore, these variables are eliminated from the C language code and turned into #define clauses, in such a way that CatapultC treats them as constants that the user can change before synthesizing the design. A summary of the variables turned into constants is shown in Table 4.2

4.3.3 Reducing the complexity of the mathematical operations to calculate the gain factor α

It is mandatory to study all the mathematical operations in the C language source code of the algorithm, in order to obtain an optimized implementation with CatapultC. For instance, we found that many multiplications could be replaced by shift operations. Setting the correct bit width of the variables is also important, to provide Catapultc with the necessary information to appropriately allocate resources for the operations. A particularly difficult and interesting case was the optimization of the mathematical operations to calculate, quantize and dequantize the gain factor α, necessary to perform the prediction (see Equations 3.2, 3.3, 3.4 in Section 3.2.1). It is observed that computing α takes the division of two integer variables which can be up to 47 bits wide. The division of two variables is costly to implement in hardware in terms of area and latency. With the aim of avoiding this division, a strategy is conceived to replace it with a dichotomic search.

As it was shown in Section 3.2.1, α is computed over the block as $\alpha = \alpha_N/\alpha_D$. Once it is calculated, it is quantized using a uniform scalar quantizer with 256 levels in the range [0,2). The quantization yields $\alpha' = \text{floor}(128\alpha)$ and α' is then clipped between 0 and 255. The dequantized gain factor is obtained as $\alpha'' = \alpha'/128$ and used to compute the predicted values within the block.

The original C source code implemented this operations as it is explained next. In order to avoid performing a fixed point division, and use only integer operations, α is computed as $\alpha = \text{round}(100\alpha_N/\alpha_D)$. Scaling α_N by a factor of 100 makes it possible to calculate it with more precision. After computing α, it is quantized, yielding $\alpha' = \text{round}(128\alpha, 100)$ and the dequantized value is obtained as $\alpha'' = \text{round}(100\alpha', 128)$.

The alternative approach which is proposed for this Thesis work, aims at finding a different way to calculate the quantized α' and the dequantized α'',

avoiding the division between two variable integers. Consequently, instead of calculating the actual value of α, the quantized value is obtained directly as $\alpha'' = \text{floor}(\alpha) = \text{floor}(128\alpha_N/\alpha_D)$. Taking into account that the quantized value is represented with 8 bits at most, because it has 256 levels, and hence it is then clipped between 0 and 255; it is also possible to avoid dividing α_N/α_D by searching between the possible 256 values which one minimizes the subtraction $|\alpha'\alpha_D - 128\alpha_N|$. This can be performed with a dichotomic search which involves a loop of 7 iterations. As it will be demonstrated in this Section, the cost in hardware of this approach is in any case lower than the one obtained with the original C implementation.

After computing α', it is necessary to calculate also its dequantized counterpart, α'', which is used to obtain $\widetilde{s}_{z,y,x}$. With the presented approach, we found out that it is more convenient to use an alternative way to dequantize α' when it is computed with the proposed dichotomic search. In particular, instead of dequantizing α' as $\alpha'' = \text{round}(100\alpha', 128)$, we compute $\alpha'' = \text{ceil}(100\alpha', 128)$, except when $\alpha' = 0$, whose dequantized value is set to $\alpha'' = 0$.

The effect of making these changes in the computation of the quantized and dequantized α is addressed next. In order to evaluate the possible impact in terms of compression ratio or quality of the reconstructed image, the same three hyperspectral images introduced in Table 5.11 are compressed with the original C code for computing α' and α'', referred as ALPHA_ORIGINAL and with the proposed approach, referred as ALPHA_CATAPULT.

Several simulations are run, compressing the images with both options for different values of *delta* ranging from 1 to 100. The compression ratio obtained with ALPHA_ORIGINAL and ALPHA_CATAPULT is calculated in bits per pixel per band (bpppb).

TABLE 4.3: Accuracy of the results obtained with the proposed implementation of the alpha quantizer for CatapultC (MODIS)

	MODIS IMAGE							
	ALPHA_CATAPULT				**ALPHA_ORIGINAL**			
DELTA	bpppb	MAE	MSE	PSNR	bpppb	MAE	MSE	PSNR
1	7.335	1	0.513	93.209	7.335	1	0.513	93.209
20	5.290	2	2.004	87.290	5.290	2	2.004	87.290
40	4.463	4	6.668	82.069	4.463	4	6.669	82.068
60	3.954	6	14.121	78.810	3.954	6	14.124	78.809
100	3.317	46	36.450	74.692	3.317	46	36.464	74.690

After compressing the images, they are decompressed, and the quality of the reconstructed image obtained for ALPHA_ORIGINAL and ALPHA_CATAPULT is also evaluated in terms of PSNR, MAE and mean-squared error (MSE).

It has to be noted that the decompressor for the images compressed with ALPHA_CATAPULT is also modified so that the compressed image can be reconstructed correctly.

In Tables 4.3, 4.5 and 4.4 it is observed that for all the images, the results in terms of compression ratio and fidelity of the decompressed image are almost the same, the differences, if any, are negligible. Therefore, it is possible to conclude that the ALPHA_CATAPULT implementation is valid and can be used instead of the ALPHA_ORIGINAL to save hardware resources.

In order to address the benefits of using ALPHA_CATAPULT in terms of the hardware occupancy, number of clock cycles to complete and critical path of the FPGA implementation, both ALPHA_CATAPULT and ALPHA_ORIGINAL are isolated from the rest of the source code and their RTL descriptions are generated using CatapultC. The target technology and clock frequency are set to the same values which are later used to obtain

TABLE 4.4: Accuracy of the results obtained with the proposed implementation of the alpha quantizer for CatapultC (AIRS)

	AIRS IMAGE							
	ALPHA_CATAPULT				**ALPHA_ORIGINAL**			
DELTA	bpppb	MAE	MSE	PSNR	bpppb	MAE	MSE	PSNR
1	4.338	1	0.588	92.616	4.333	1	0.588	92.612
20	2.559	11	2.051	87.188	2.554	11	2.053	87.184
40	1.637	36	8.782	80.873	1.628	33	8.838	80.845
60	0.948	58	22.470	76.793	0.942	59	22.554	76.777
100	0.447	86	54.061	72.980	0.445	82	54.051	72.981

TABLE 4.5: Accuracy of the results obtained with the proposed implementation of the alpha quantizer for CatapultC (AVIRIS)

	AVIRIS IMAGE							
	ALPHA_CATAPULT				**ALPHA_ORIGINAL**			
DELTA	bpppb	MAE	MSE	PSNR	bpppb	MAE	MSE	PSNR
1	5.898	1	0.536	93.019	5.897	1	0.536	93.018
20	3.946	2	2.000	87.298	3.945	2	2.000	87.297
40	3.130	4	6.670	82.068	3.130	4	6.669	82.068
60	2.580	128	49.346	73.376	2.580	129	49.345	73.376
100	1.650	128	49.346	73.376	1.649	129	49.345	73.376

the implementation of the whole LCE compressor (see Section 4.4), specifically: Virtex 5VFX130 and 80 MHz. CatapultC estimations of cycles to complete, area and slack for both options are shown in Table 4.6, where it is observed that despite ALPHA_CATAPULT taking 4 more cycles to complete, the savings in area and critical path are evident, and therefore, for the targeted technology, it is the best option.

TABLE 4.6: Estimation of area, cycles and slack for ALPHA_ORIGINAL and ALPHA_CATAPULT

Virtex 5VFX130 80 MHz			
Option	**Area (gates)**	**Cycles to complete**	**Slack (ns)**
ALPHA_ORIGINAL	6254.22	13	0.08
ALPHA_CATAPULT	3388.08	17	2.94

4.3.4 Loop optimization

The way CatapultC schedules and implements the loops of the design can be controlled by the user by means of loop unrolling and loop pipelining. All the loops in the original C source code of the LCE algorithm are carefully analysed, in such a way that it is ensured that the number of iterations in the loop is known, which is mandatory for CatapultC to be able to schedule it correctly. In those cases where it is impossible to determine the exact number of iterations, the number of iterations is set to the maximum possible value. In general, the strategy followed to optimize loops aims at unrolling them whenever it is possible and, in those cases where the data dependencies make unrolling impossible, use pipelining with $II = 1$, maximizing the throughput with the goal of obtaining one compressed sample per clock cycle.

We present next the strategy followed to optimize three of the loops in the entropy coding stage of the LCE algorithm, namely: the loop which computes the Golomb parameter, the loop for calculating the exponential Golomb codeword; and the loop for calculating the Golomb codewords. Our primary motivation for describing these procedures is that we consider they are good example of code optimization for CatapultC, for those cases where there are loops in the source code where the maximum number of iterations is unknown a priori. Although the strategies shown are designed and applied specifically to the LCE algorithm, the main ideas might be employed for

the HLS hardware implementation of other sample-adaptive entropy coders, which are widely used for data compression.

Loop for calculating the Golomb parameter

The Golomb parameter is calculated using the one-liner operator:

$$\text{for } (k = 0;\ (J \ll k) <= R_j;\ k{+}{+})$$

where R_j is a running count of the sum of the magnitude of the last 32 unmapped residuals of the block. J is the number of values used for computing R_j and k is the desired Golomb parameter.

Taking into account the maximum possible bit width of the unmapped prediction residuals, the maximum number of iterations of this loop is set to 32. It is also decided to fully unroll the loop with CatapultC, what is equivalent to obtaining the 32 possible values of (J≪k) and detect which is the smallest, which is greater than R_j.

Loop for calculating the exponential Golomb codeword

The exponential Golomb codeword is obtained originally with the loop:

$$\text{while } (((1 \ll n) - 1) < M1)\ \ n{+}{+};$$

The obtained codeword consists of $n-1$ zeros concatenated with the n least significant bits of $M1$. The loop has an unknown number of iterations and hence some changes are applied. We observe that iterating can be avoided by obtaining n as $n = \log_2 M1$. To compute the logarithm, a function which finds the leading one in $M1$ with a dichotomic search is created. The dichotomic search can be fully unrolled with CatapultC, and the loop virtually disappears from the design.

Loop for calculating the Golomb codewords

The Golomb codeword is a concatenation of a specific number of ones, a zero and a remainder. It is computed in the original C source code as:

- Unary coding:

for $(j = 0;\ j < (M \gg k);\ j{+}{+})$ write_file(1);

- Zero:

write_file(0);

- Remainder:

write_file(M &$((1 \ll k) - 1)$ using k bits);

where k is the Golomb parameter and M the value to be encoded. In the original C implementation, the number of iterations of the loop is unknown and the values are saved directly to a file, which cannot be implemented in hardware.

The Golomb parameter is adaptive according to the data entropy. Performing a worst-case analysis, it is observed that the maximum possible number of iterations of the unary coding loop can become really high. It is not advisable to set constraints in CatapultC so that it generates a design for such a high number of iterations, since that would mean adding hardware resources for a worst-case scenario which would take place only in a few occasions during the compression of a real image.

We present next a strategy to solve this issue and modify the C source code to avoid this loop.

The number of ones to be written can be easily calculated as $(M \gg k)$. However, the resulting codeword has to be saved to the 32 bits buffer *pp* and

then saved to memory, and the result of $(M \gg k)$ can be much greater than 32 for some codewords.

It is proposed to initialize the output memory, which stores the compressed block, with ones. Therefore, when $(M \gg k)$ is greater than 32, it is not necessary to write the ones to memory. It is enough to save the value of the remainder to the bit packing buffer *pp*. A variable is used to indicate the memory position where the next word has to be written to memory. This variable can be easily updated taking into account the total number of ones in the codeword.

Figures 4.6 and 4.7 show how the codewords are saved in a memory when it is not initialized and how it is saved in an initialized memory, avoiding having to loop an unknown number of iterations.

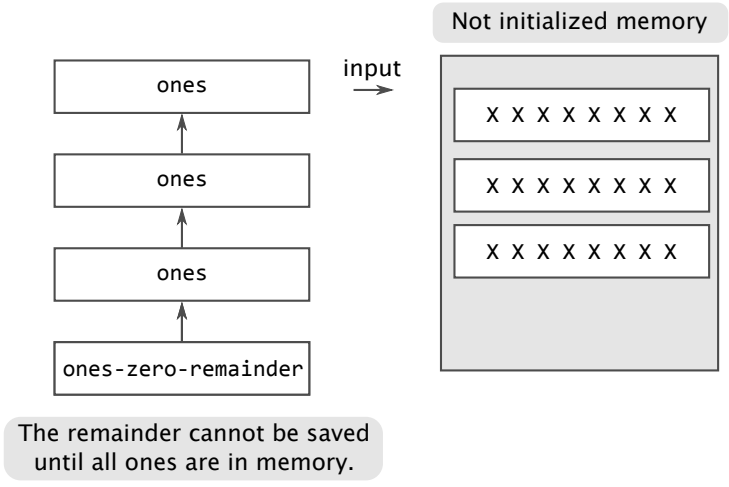


FIGURE 4.6: Saving the codewords to a memory which is not initialized

We note that initializing the output memory implies introducing an additional loop in the design. The number of values to initialize is set to 128× 32 bits. With a dual port memory, 32 bits wide, the initialization can be done in

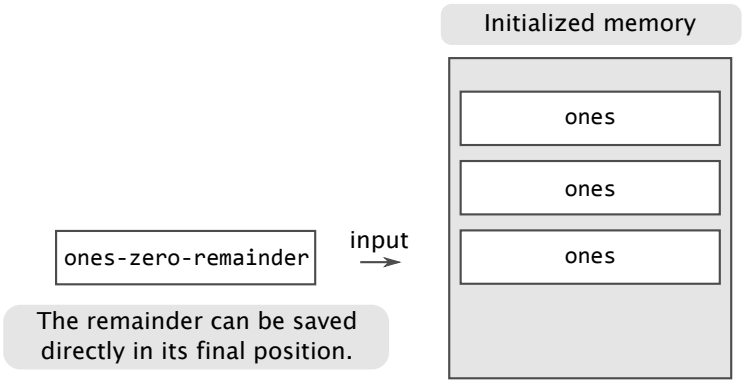


FIGURE 4.7: Saving the codewords to a memory which has been initialized with all ones

64 cycles, increasing the time necessary for the first sample to be compressed, but producing a negligible effect in the throughput, as it is explained later in Section 4.4.1.

Since we will reference the loops during the rest of this Chapter, the most significant are described in Table 4.7. We also reference which loops solve the equations of the LCE presented in Section 3.2.

4.4 Results of the FPGA implementation of the LCE algorithm with CatapultC

After optimizing the C language source code of the LCE algorithm for a CatapultC implementation, the modified code is verified by utilizing the original LCE software implementation as golden reference. Afterwards, the modified C code is introduced in CatapultC University Version Release 2010a

TABLE 4.7: Description of the loops in the design

Band	**Loop**	**No. of iterations**	**Description**	**Equations in LCE**
0	INIT	128	Initializes the output memory with ones.	
	PRED2D	256	Performs the 2D prediction, mean value of the pixels in the current block and the entropy coding for the first and.	3.1, 3.6
> 0	INIT	128	Initializes the output memory with ones.	
	MEAN	64	Computes the mean value of the pixels in the current block for bands other than the first.	
	EST	256	Estimate α_N and α_D	3.3, 3.4
	LS	7	Compute the quantized gain parameter α and dequantize it.	3.2
	CALCPERR	256	Calculate predictor and prediction error. Calculate distortion incurred to decide if the *zero_block* option is raised.	3.5, 3.6, 3.7
	ZEROB	256	Calculate *ref_block* in case of *zero_block* decision.	3.8
	MID_ENTR	256	Calculate *ref_block* when *zero_block* is not triggered. Entropy coding of the prediction errors.	3.9

to obtain a hardware implementation on an FPGA. It has to be noted that the University Version limits part of the possible optimizations of the design.

The source code is first compiled in CatapultC and the top design and the global hardware constraints are established. The target technology is set to a Virtex 5VFX130, since it has an equivalent radiation-tolerant chip, and it is particularly interesting for space applications. We establish the clock frequency at 80 MHz and select the function *pred1block()* to be the top design function, as it was stated in Section 4.3.

After setting the global constraints, the architectural hardware constraints are specified. At this step, we confirm that CatapultC has correctly identified

the top design, and the inputs and outputs of the top function are interpreted by the tool as expected.

Furthermore the loop optimization constraints are set, i.e. it is specified if loops are unrolled and pipelined. A summary of the constraints utilized to optimize each loop is shown in Table 4.8. The parameters U and II show how many times the loop is unrolled and the selected initiation interval for pipelining, respectively. The number of cycles necessary to complete each loop is also presented as well as the latency, defined as the number of cycles before the first sample is processed; and the throughput, defined as the number of samples processed per cycle after the latency cycles. We consider this particularly useful to understand the final results provided by CatapultC and to explore further optimizations.

CatapultC, performs the scheduling of the design and generates the RTL. Afterwards, Mentor Precision Synthesis 2009a is employed in order to synthesize the design in the target FPGA. Once the synthesis is finished, the placement and routing (P&R) is performed. The results after P&R are summarized in Table 4.9 and Table 4.10.

As a result of the scheduling process, CatapultC provides information in terms of latency and throughput of the implemented design, which can be used, together with the P&R results to create Table 4.11, where we present the number of latency cycles and the amount of samples per clock cycle and samples per second obtained after the latency. We also indicate the number of cycles it takes to process the 16×16 samples in a block in a specific band and finally we calculate the average number of samples/second as:

$$MegaSamples/sec = 10^{-6} \times \frac{Number\ of\ samples\ in\ a\ block}{total\ cycles \times (clock\ frequency)^{-1}} \quad (4.1)$$

TABLE 4.8: Optimization of the loops in the design

Band	**Loop**	**No. of iterations**	**Unroll Pipeline**	**Area (gates)**	**Latency (cycles)**	**Throughput (samples/cycle)**	**Cycles to complete**	**Slack (ns)**
0	INIT	128	U = 2 II = 1	57.17	1	2/1	65	9.67
	PRED2D	256	U = 0 II = 7	3985.29	5	1/7	1797	0.14
> 0	INIT	128	U = 2 II = 1	53.17	1	2/1	65	9.67
	MEAN	64	U = 0 II = 1	283.70	2	1/1	66	5.25
	EST	256	U = 0 II = 1	187.52	1	1/1	257	7.32
	LS	7	U = 0 II = 2	3388.08	3	1/17	17	2.94
	CALCPERR	256	U = 2 II = 1	3060.93	6	2/1	134	2.69
	ZEROB	256	U = 0 II = 1	91.98	1	1/1	257	7.49
	MID_ENTR	256	U = 0 II = 1	3956.66	13	1/1	269	2.12

TABLE 4.9: Utilization results after P&R

Virtex 5VFX130		
Resources	***Total***	%
BUFGs	1 out of 32	3
DSP48Es	17 out of 320	5
Number of RAMB18X2s	4 out of 298	1
Number of slices	2951 out of 20480	14
Number of Slice Registers	4208 out of 81920	5
Number of Slice LUTS	7836 out of 81920	9
Number of Slice LUT-Flip Flop pairs	8886 out of 64000	10

TABLE 4.10: Timing results after P&R

Virtex 5VFX130	
Minimum period	***Maximum frequency***
12 ns	80 MHz

TABLE 4.11: Number of samples processed per second for each band

Band	**Latency cycles**	**Throughput samples/cycles**	**MSamples/sec (After latency)**	**Total cycles**	**MSamples/sec (Average)**
0	72	1/7	11.5	1862	11.0
> 0	808	1/1	80.2	1064	19.2

Analysing these results, we observe that CatapultC, it did not schedule the execution of different loops in parallel when the data dependencies make it possible. Therefore, there is still room for optimization in order to improve the design throughput.

4.4.1 Manual scheduling of the design

In order to explore how much the aforementioned CatapultC results can be improved, a study of the data dependencies between the loops and the possible scheduling of the compression of the different bands is performed, to

TABLE 4.12: Number of samples processed per second for each band with a manual scheduling of the design

Band	Latency cycles	Throughput samples/cycles	Total cycles	Mega samples/sec (Average)
0	72	1/7	1862	11.0
1	421	1/1	677	30.3
> 1	164	1/1	420	48.8

have an idea of the amount of parallelism which can be gained. The results are shown in Figure 4.8, where the arrows indicate data dependencies and the loops which are drawn in the same column can be executed in parallel.

It can be observed that the INIT loop is parallelized with other operations for bands greater than the first, hiding most of the latency introduced by adding it to the design. The average throughput with a manual scheduling of the design is presented in Table 4.12.

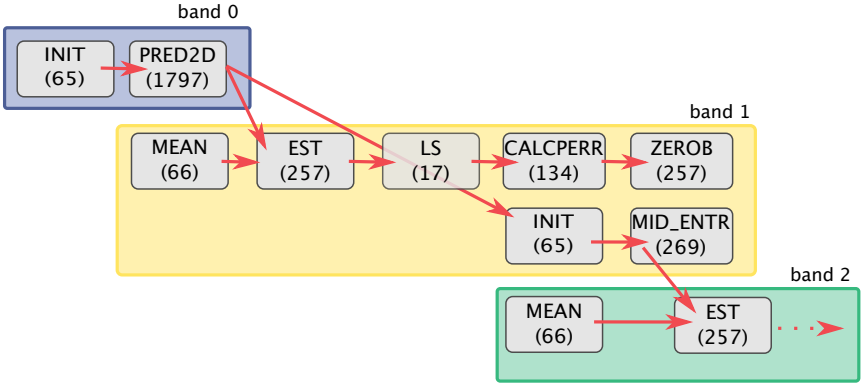


FIGURE 4.8: Manual scheduling of the loops in the LCE compressor

To obtain this potential improvement, a different design methodology is proposed, namely, a modular approach consisting of dividing the C source code

TABLE 4.13: Functional units identified in the modular approach

Unit Name	**Description**
init_output	Output memory initialization.
mean	Computation of the mean of the samples in the block.
pred_2D	2D prediction and entropy coding for the first band.
estimation_ls	Computation, quantization and dequantization of the gain factor (α)
calc_perr	Prediction error computation; rate-distortion relationship and *zero_block* decision.
zero_block	Computation of the reference samples when the *zero_block* flag is triggered
ad_golomb_code_perr	Computation of the reference samples, when block is not zero, and entropy coding of the prediction errors.

in different processing units, obtaining the VHDL description with CatapultC and manually scheduling the units to obtain a schedule similar to the one shown in Figure 4.8. The downside of this approach is that it is mandatory to manually write VHDL code. This strategy would make it possible to hide latency for bands other than the first one, yielding a gain in the average throughput. The results with this approach are presented in Section 4.12.

4.4.2 Implementation of the LCE algorithm using a modular approach

In this approach, the different stages of the LCE algorithm are identified in the C language source code and split in independent files. An RTL description of each of them is obtained with CatapultC, and a finite-state machine (FSM) is then manually written in VHDL to control the behaviour of the different modules. Seven different modules are identified, as can be seen in Table 4.13

TABLE 4.14: Occupancy modular approach against non-modular approach

Virtex 5VFX130		**Modular approach**		**Non-modular approach**	
Resources	***Available***	***Used***	%	***Used***	%
DSP48Es	320	17	5	25	8
Number of RAMB18X2s	298	4	1	4	1
Number of slices	20480	2951	14	1935	10
Number of Slice Registers	81920	4208	5	5995	7
Number of Slice LUTS	81920	7836	9	7738	10

TABLE 4.15: Throughput of the modular approach against the non-modular approach

	Virtex 5VFX130	
	Modular approach	**Non-modular approach**
Max.Frequency (MHz)	86	80
Throughput (Msamples /sec)	27.7	16.7

Table 4.14 and 4.15 show the results of the aforementioned approach in comparison with the non-modular approach.

4.4.3 Comparison with the FPGA implementation of a near-lossless algorithm

Finally, we compare our results with those shown in [56] for the FPGA implementation of a lossless to near-lossless image compression algorithm. Like the LCE algorithm, it is based on a prediction plus adaptive entropy coding scheme. For the sake of comparison, we provide an implementation with CatapultC targeting an FPGA from the Virtex IV family, namely 4VLX200. We select the same constraints and settings used for the Virtex 5 implementation with CatapultC. The results of the LCE and the lossless to near-lossless implementation after P&R are shown in Table 4.16.

TABLE 4.16: Implementation comparison

Virtex IV 4VLX200	**LCE CatpultC**	**Lossless to near-lossless**
Used LUT	9283 (5%)	10306 (5%)
RAM 16s	4 (1%)	21 (6%)
DSP48	25 (26%)	9 (9%)
Max. Frequency	75 MHz	81 MHz

We observe that the LCE algorithm shows better results than the lossless to near-lossless algorithm in terms of frequency and LUT and memory resources requirements. However, the LCE uses more DSPs, which is reasonable because of the mathematical operations involved in a lossy algorithm.

4.5 Performance comparison: FPGA, GPU, CPU

With the results obtained in this Chapter, it is possible to make a performance comparison between the throughput that can be obtained with the different technologies utilized to implement the LCE compressor: FPGA, CPU, and GPU. We calculate the number of samples compressed every second for the AVIRIS, MODIS and AIRS hyperspectral images. In the case of the CPU and GPU, we calculate the throughput taking into account the dimensions of the images and the execution times as:

$$\begin{matrix} Throughput\ GPU\ and\ CPU \\ (MSamples/sec) \end{matrix} = 10^{-6} \times \frac{Nz \times Ny \times Nx}{total\ execution\ time(sec.)} \tag{4.2}$$

In the case of the FPGA implementation, we calculate the throughput of every image taking into account the dimensions of the image, the maximum

TABLE 4.17: Throughput of the FPGA implementation of the LCE for the hyperspectral images under evaluation

Image	**Throughput modular approach (Msamples /sec)**
MODIS	27.9
AVIRIS	27.7
AIRS	24.6

frequency and the number of clock cycles necessary to perform the compression with the non-modular implementation obtained with CatapultC, using the formula:

$$\begin{matrix} Throughput\ FPGA \\ (MSamples/sec) \end{matrix} = 10^{-6} \times \frac{Nz \times Ny \times Nx}{total\ cycles \times (clock\ frequency)^{-1}} \tag{4.3}$$

The obtained results can be seen in Table 4.17. To calculate these results, we consider that we only implement one instance of the compressor in the FPGA. Nevertheless, taking into account the low occupancy and the fact that the compression of blocks are independent, we note that it is possible to multiply this throughput by implementing more than one LCE compressor core per FPGA.

The comparative results of the performance of the three implementations of the LCE compression algorithm: GPU, CPU, and FPGA is shown in Figures 4.9, 4.10, and 4.11.

The GPU and FPGA implementations of the LCE algorithm yield better performance than the CPU. Comparing GPU and FPGA, the GPU provides the best performance with about 5-6 times more samples computed per second, when compared with an FPGA implementation of a LCE compression module for one image block of 16×16 samples. Nevertheless, if we take into

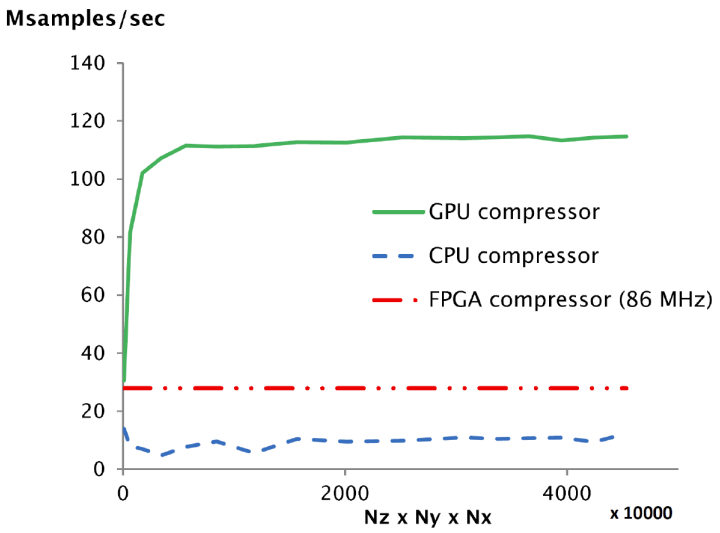


FIGURE 4.9: Comparison of the throughput of the GPU, CPU and FPGA implementations of the LCE algorithm for the MODIS image

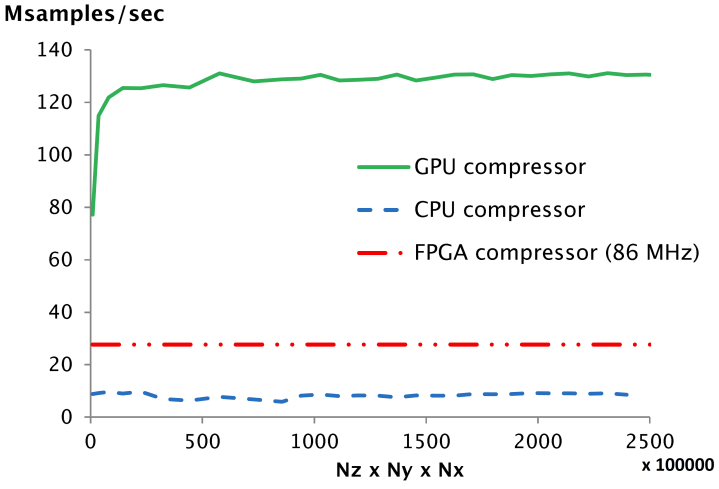


FIGURE 4.10: Comparison of the throughput of the GPU, CPU and FPGA implementations of the LCE algorithm for the AVIRIS image

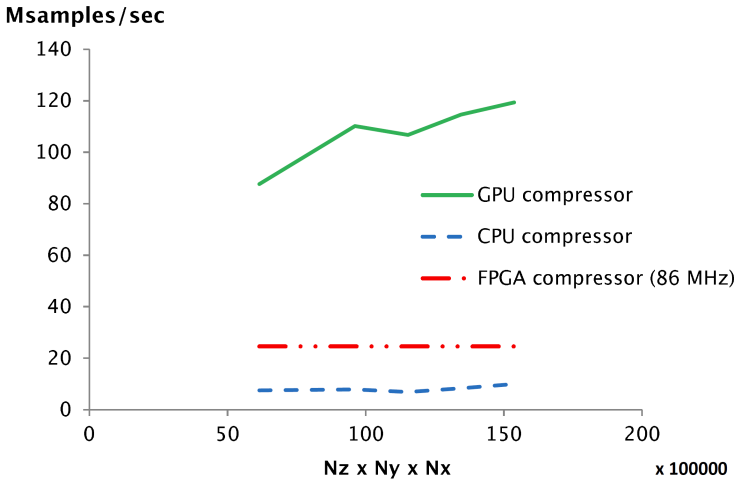


FIGURE 4.11: Comparison of the throughput of the GPU, CPU and FPGA implementations of the LCE algorithm for the AIRS image

account that the compression of the image blocks is independent, we can observe that it is possible for the FPGA implementation to reach the throughput of the GPU by instantiating more than one compression module, which is perfectly feasible, due to the low occupancy of the algorithm in the FPGA (around 14% of a Virtex 5VFX130). We note that, the power consumption of the GPU is dramatically higher than that of an FPGA implementation, and should also be considered when comparing both technologies. The Tesla C2075 used in the experiments of this Chapter has a power consumption of up to 225 W TDP, whereas the FPGA implementation generated with CatapultC shows a consumption of to 2.679 W.

Although GPUs achieve a very high throughput, they are not suited to operate on-board. They are not radiation-tolerant and their power consumption is too high for on-board usage. FPGAs are more amenable to be used on-board, exhibiting a similar throughput but a much lower power consumption.

Nevertheless, GPUs provide very flexible implementations, and the design flow is much shorter than that of the FPGA and requires less resources. From all this facts, we conclude that it makes sense to continue investigating on GPUs for space, and the biggest efforts should be made towards reducing their power consumption.

Chapter 5

Implementation of the CCSDS standard for lossless hyperspectral image compression on a space-qualified FPGA

This Chapter describes the implementation of the recent CCSDS 123 standard for multispectral and hyperspectral compression on a space-qualified FPGA from the Microsemi RTAX family. The main objective is to design a hardware architecture with low resource occupancy and high performance, which is suitable for on-board use in current and future space missions.

Part of this work has been funded by Thales Alenia Space España S.A. (TASE) under a collaboration agreement with the Institute For Applied Microelectronics (IUMA) and are subject to copyright (TASE/IUMA-12.016).

5.1 Outline

As it has been exposed in the previous Chapters of this Thesis, the compression of hyperspectral images on-board satellites is at the same time a challenge and a necessity whose importance is currently growing as the resolution of the sensors tend to increase. The images are captured in several different wavelengths, which can range from tens to hundreds, and therefore represent a huge amount of data which has to be reduced in order to meet the available on-board storage and transmission bandwidth requirements. Consequently, hyperspectral image compression has become a very popular research topic, which has motivated the proposal of different compression algorithms with diverse compression efficiency and complexity. On-board compression algorithms have to meet additional requirements which are specific to the space environment such as low complexity and error resilience. The available processing power on a satellite is limited, and most usual data compression algorithms used on ground cannot be applied to space data systems. Nevertheless, to actually benefit from on-board compression, not only efficient algorithms are needed; it is also essential to provide physical implementations of those algorithms, which can operate on-board.

Recently, the CCSDS, which represents the major space agencies in the world, has issued a recommendation for lossless multispectral and hyperspectral data compression, the CCSDS 123 [25]. The CCSDS 123 compressor is based on the FL algorithm [55] and consists of a predictor and an entropy coder. The predictor uses adaptive linear prediction based on values of nearby samples in a small three-dimensional neighbourhood. Afterwards, the residuals of the prediction are mapped and entropy coded. Experimental results have demonstrated that the CCSDS standard is competitive with other state-of-the-art algorithms, providing the best trade-off between coding performance and computational complexity [74].

The CCSDS 123 algorithm has been already implemented in software for its execution on CPU [74, 76] and GPU [81]. However, none of these implementations are suited for on-board use. This Chapter presents a low-complexity hardware architecture of the CCSDS 123 algorithm which can be implemented on a space-qualified FPGA with low hardware occupancy. The architecture is carefully designed taking into consideration the impact in terms of compression efficiency and implementation complexity of the different user-defined parameters allowed by the CCSDS 123 standard.

The resulting architecture is the basis of an IP core named HyLoC, which is described at RTL level using VHDL, and then implemented on the space-qualified RTAX1000S FPGA. HyLoC is fully compliant with the CCSDS 123 standard, allows the adjustment of the user-defined parameters and is technology independent. Finally, the HyLoC VHDL description is validated on a Xilinx prototyping board with a Virtex 5 FPGA, which provides further evidences of the benefits of the proposed implementation.

We note that an FPGA implementation of the FL algorithm was already presented in [32]. Although the FL algorithm has a lot in common with the CCSDS 123, the architecture presented here is drawn up with a different approach, making it fully compliant with the standard and considering the different combinations of user-defined parameters, which are not part of the FL algorithm.

5.2 The CCSDS 123 standard for lossless multispectral and hyperspectral image compression overview

The CCSDS 123 algorithm performs lossless compression of multispectral and hyperspectral images [25]. It utilizes a scheme based on prediction and

entropy coding of the prediction residuals. Let $s_{z,y,x}$ be a sample located in spatial coordinates (y, x) and band z. The predicted sample $\hat{s}_{z,y,x}$ is computed using the previously processed neighbouring samples of $s_{z,y,x}$ in the current band as well as in P previous bands. Figure 5.1 illustrates the typical neighbourhood of samples used for prediction; this neighbourhood is suitably truncated for the samples located in the image edges. The number of previous bands used for prediction, P, is a user-defined parameter which can range from 0, for which no information from previous bands is utilized, to 15. The flowchart which describes the several steps of the algorithm is shown in Figure 5.2.

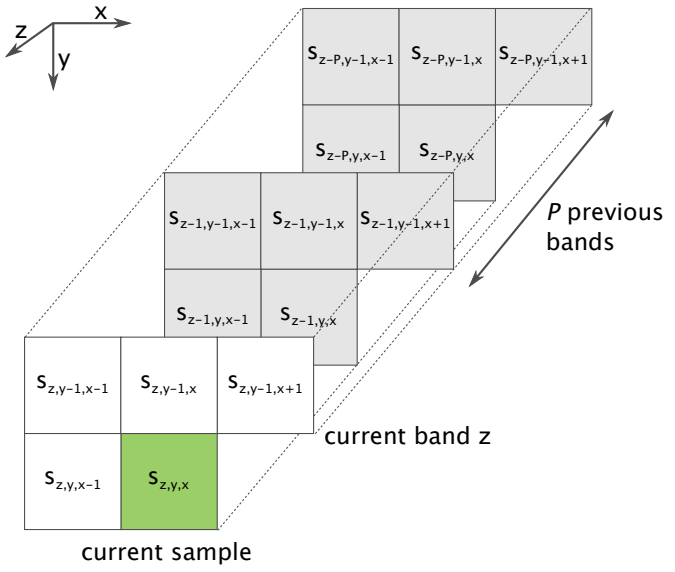


FIGURE 5.1: Current sample and neighbours used for computing the local sums and local differences

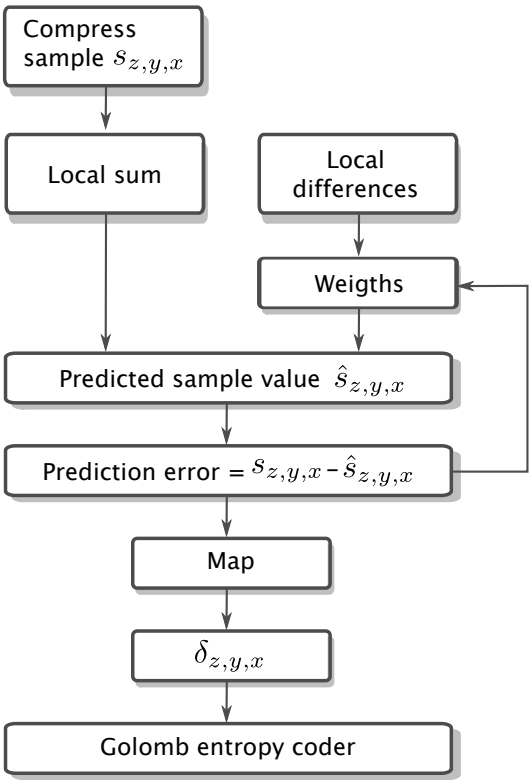


FIGURE 5.2: Flowchart of the CCSDS 123 algorithm

5.2.1 Prediction

First, a *local sum*, $\sigma_{z,y,x}$, of the neighbouring samples of $s_{z,y,x}$ in the current band is computed. A user-defined parameter is employed to select between two possible configurations for the local sum computation: column-oriented and neighbour-oriented. The neighbours utilized by each configuration are shown in Figure 5.3.

The column-oriented local sum is computed using the neighbour on top of the current sample, $s_{z,y-1,x}$; on the other hand, the neighbour-oriented local sum is calculated using 4 neighbouring samples, namely: $s_{z,y-1,x-1}$, $s_{z,y-1,x}$, $s_{z,y-1,x+1}$ and $s_{z,y,x-1}$. The equations for computing the local sum are shown in Table 5.1.

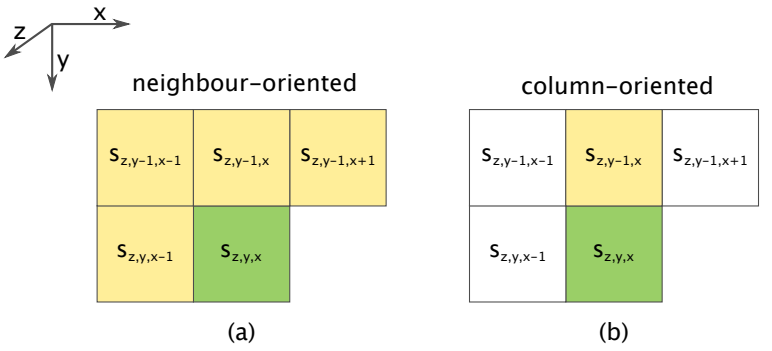


FIGURE 5.3: Current sample and neighbours used for computing the directional local sum

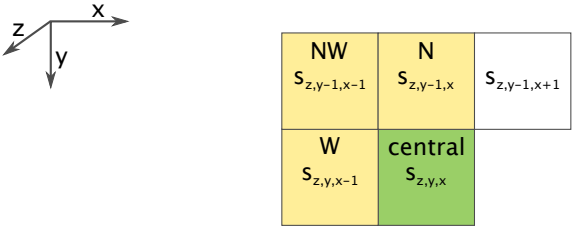


FIGURE 5.4: Current sample and neighbours used for computing the directional local differences

The local sums are used to calculate the *central local differences* values $d_{z,y,x}$ and the *directional local differences*: $d^N_{z,y,x}$, $d^W_{z,y,x}$ and $d^{NW}_{z,y,x}$ (see Figure 5.4). The user can choose to perform prediction in *full* or *reduced* mode by selecting the appropriate parameter. Under reduced mode, the prediction is computed from a weighted sum of the central local differences calculated in

TABLE 5.1: Equations for calculating the local sum

Local sum $\sigma_{z,y,x}$ of sample $s_{z,y,x}$	
Neighbour oriented	**Column oriented**
$s_{z,y-1,x-1} + s_{z,y-1,x} + s_{z,y-1,x+1}$	$4s_{z,y-1,x}$

P preceding bands. The directional local differences are not utilized under reduced mode and therefore do not need to be calculated. On the other hand, under full prediction mode, the prediction depends not only on the central local differences in the P previous bands, but also on the weighted sum of the directional local differences in the current band. The weight values are updated adaptively according to the resulting prediction residual.

The central local differences in the P previous bands together with the 3 directional local differences conform a vector, $U_{x,y,z}$, whose elements are computed according to the selected configuration as it is shown in Table 5.2.

The prediction is calculated by computing the inner product, $\hat{d}$, between the local difference vector $U_{z,y,x}$ and the weight vector $W_{z,y,x}$:

$$\hat{d} = W_{z,y,x}^T \cdot U_{z,y,x} \tag{5.1}$$

The weight vector has the same amount of components of the local differences vector, and measures the effectiveness of each component of the local differences vector in predicting the sample being coded. The *scaled predicted sample value*, $\tilde{s}_{z,y,x}$, is computed from $\hat{d}$ and used to calculate the *scaled prediction error* as:

$$e_{z,y,x} = 2s_{z,y,x} - \tilde{s}_{z,y,x} \tag{5.2}$$

The components of the weight vector, $W_{z,y,x}$, are adaptively updated based on the sign of $e_{z,y,x}$. The rate at which the weight vector adapts changes over time, following a schedule which is determined by three user-defined

TABLE 5.2: Equations for calculating the elements of the local differences vector $U_{z,y,x}$

		Local differences vector $U_{z,y,x}$	
FULL PREDICTION	DIRECTIONAL LOCAL DIFF	$d^N_{z,y,x} = 4s_{z,y-1,x} - \sigma_{z,y,x}$	
		$d^W_{z,y,x} = 4s_{z,y,x-1} - \sigma_{z,y,x}$	
		$d^N W_{z,y,x} = 4s_{z,y-1,x-1} - \sigma_{z,y,x}$	
	CENTRAL LOCAL DIFF	$d_{z-1,y,x} = 4s_{z-1,y,x} - \sigma_{z-1,y,x}$	**REDUCED PREDICTION**
		$d_{z-2,y,x} = 4s_{z-1,y,x} - \sigma_{z-1,y,x}$	
		...	
		$d_{z-P,y,x} = 4s_{z-P,y,x} - \sigma_{z-P,y,x}$	

parameters: the weight update scaling exponent initial parameter (ν_{min}), the weight update scaling exponent final parameter (ν_{max}) and the weight update scaling exponent change interval (t_{inc}). The larger the weight increments, the faster the adaptation to the source statistics, but the worse the steady-state prediction.

The *predicted sample value*, $\hat{s}_{z,y,x}$, is computed as:

$$\hat{s}_{z,y,x} = \lfloor \frac{\tilde{s}_{z,y,x}}{2} \rfloor \tag{5.3}$$

and it is used to compute the *prediction residual*, $\Delta_{z,y,x} = s_{z,y,x} - \hat{s}_{z,y,x}$, which is then mapped to positive integer values, $\delta_{z,y,x}$.

Subsequently, the mapped prediction residuals, $\delta_{z,y,x}$, are sequentially encoded in the order selected by the user: band-sequential (BSQ) or band-interleaved (BI). This encoding order specifies likewise the order in which the encoded samples are arranged in the compressed file.

5.2.2 Entropy coding

The CCSDS 123 standard allows the selection between a sample-adaptive entropy coder and a block-adaptive entropy coder. For the implementation presented in this Thesis work, only the sample-adaptive entropy coder option is considered. Under this approach, the mapped prediction residuals $\delta_{z,y,x}$ are encoded using a Golomb power-of-two variable-length binary codeword. The codes are adaptively selected based on statistics which consist of an accumulator and a counter which are updated after each sample is encoded, and reset periodically according to an interval set by the user-defined parameters. The final codeword is composed by U zeros followed by a 1 and the k most significant bits of $\delta_{z,y,x}$, as shown in Figure 5.5.

5.3 Design methodology

Unlike the lossy hyperspectral compression algorithm FPGA implementation described in Chapter 4, we consider that high-level synthesis (HLS) is not the most appropriate methodology for obtaining an RTL description of the CCSDS 123 algorithm which is suitable for an implementation on a space-qualified FPGA and which would meet our objectives of low-occupancy at the same time. Using HLS has been proven to shorten the design times by generating RTL descriptions from C-language working implementations.

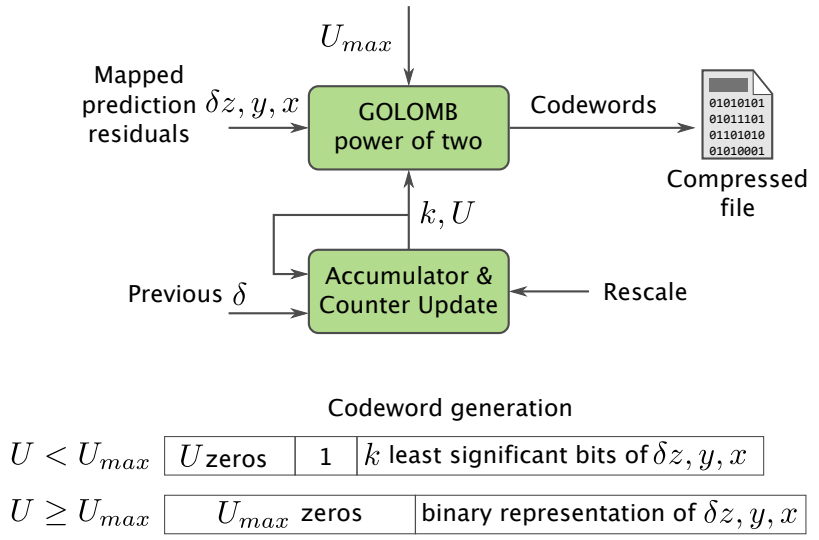


FIGURE 5.5: Sample-adaptive codeword generation.

However, in the case of the CCSDS 123 algorithm, we did not have a working and verified C-language implementation of the algorithm that we could use as input for the HLS tool. Besides, we wanted to design the FPGA implementation in such a way that we could ensure low-complexity for any combination of the algorithm's user-defined parameters, keeping in mind that the final target would be a space-qualified FPGA. The Microsemi space-qualified FPGA for which we design the implementation of the CCSDS 123 algorithm cannot be set as a target for implementation in our HLS tool, CatapultC. This fact does not prevent the tool from generating the VHDL code, however, it cannot guarantee that the design will fit in the desired FPGA or meet the timing constraints. In order to accomplish our goals of achieving low-complexity and the resulting implementation being able to operate on-board, it was decided to write the VHDL code from scratch, making it possible to design the hardware architecture carefully, and be able to control the behaviour of each single module in the compressor.

We present next the architectural description and FPGA implementation of the CCSDS 123 algorithm, which is designed according to the following methodology. We first perform simulations and study the results in [104], in order to establish which of the user-defined parameters have a greater impact in the compression ratio. Afterwards, the impact of these parameters in the complexity of the resulting hardware implementation is studied and several architectural options are proposed and compared, with the goal of selecting the one that provides low area utilization, low power and high throughput.

5.4 Impact of the user-defined parameters in the compression efficiency

The CCSDS 123 standard allows the user to set several parameters which permit to control the performance and behaviour of the compressor. Before designing the hardware architecture of the compressor, we study the effect of these parameters, in order to establish which of them have a greater impact in the compression ratio. To accomplish this, we perform the compression of several multispectral and hyperspectral images with the CCSDS 123 standard, with a software called Empordá, developed in Universitat Autònoma de Barcelona (UAB) under an ESA contract. The referred software is open-source and can be downloaded from [105]. The hyperspectral images used as a target for compression are summarized in Table 5.3, where Nx represents the number of image samples in the spatial direction x; Ny represents the number of samples in direction y and Nz is the number of bands

The 3MI-VNIR(9) and 3MI-VNIR(21) images are created to be representative of a hyperspectral image acquired by the visible and near infrared (VNIR) sensor of the 3MI instrument, which is planned to be part of the Metop-SG satellites (Eumetsat Polar System Second Generation missions) [106]. The images are created from the uncalibrated AVIRIS Yellowstone

TABLE 5.3: Hyperspectral images used to assess the effect of the user-defined parameters of the CCSDS 123 standard

Sensor	**Area**	Nz	Ny	Nx	**Image type**	**Pixel bit width**	**Name in legend**
AVIRIS	Indian Pines (subset)	220	256	256	calibrated	16	INDIAN PINES
MODIS	-	17	2300	1354	raw	12	MODIS
AIRS	-	1504	135	90	raw	14	AIRS
AVIRIS	Yellowstone Scn0	224	512	677	raw	16	RAW YEL-LOWS
AVIRIS	Yellowstone Scn0	224	512	680	calibrated	16	CAL YEL-LOWS
3MI-VNIR(9)	Yellowstone Scn0	9	512	677	raw	16	3MI-VNIR(9)
3MI-VNIR(21)	Yellowstone Scn0	21	512	677	raw	16	3MI-VNIR(21)

Scene0, by selecting the wavelengths which would be acquired by the aforementioned sensor.

In order to characterize the CCSDS 123 compressor based on its user-defined parameters, the Empordá compression software is initially configured with the default parameters. The selection of the default settings for Empordá is based on the developer's experience [74, 104]. Each parameter is then varied in its allowed range, while the rest are kept in their default values. The parameters studied, their range and their default values and description are summarized in Tables 5.4 and 5.5.

We calculate then the compression achieved by the different configurations by computing the number of bits per pixel of the compressed images as:

$$\text{Bit rate (bpppb)} = \frac{Size\ of\ compressed\ file\ (bits)}{Nz \times Ny \times Nx} \tag{5.4}$$

TABLE 5.4: Parameters studied for the prediction characterization

Parameter	**Range**	**Default in Empordá**	**Description**
Number of bands for prediction (P)	$[0-15]$	15	Number of previous bands used to perform the prediction.
Prediction mode	[full, reduced]	full	Indicates if the directional local differences are used in the prediction calculation.
Local sum mode	[neighbour, column]	neighbour	Defines the neighbourhood used to compute the local sums.
Register size	$[32-64]$	32	Size of the register used in the prediction calculation.
Weight component resolution (Ω)	$[4-19]$	13	Determines the number of bits used to represent the weight vector components, which is calculated as $\Omega+3$.
Weight update scaling exponent change interval (t_{inc})	$[4-11]$	6	Sets the interval at which the weight update scaling exponent increments.
Weight update scaling exponent initial parameter (ν_{min})	$[-6-\nu_{max}]$	-1	Determines the initial rate at which the prediction adapts the weight vector to the input samples.
Weight update scaling exponent final parameter (ν_{max})	$[\nu_{min}-9]$	3	Determines the final rate at which the prediction adapts the weight vector to the input samples.

TABLE 5.5: Parameters studied for the entropy coder characterization

Parameter	**Range**	**Default in Empordá**	**Description**
Initial count exponent (γ_0)	$[1-8]$	1	Sets the initial counter value.
Accumulator initialization constant (k'_z)	$[4-9]$	6	Sets the initial accumulator value.
Rescaling counter size (γ^*)	$[4-9]$	6	Determines the interval between the rescaling of the counter and accumulator.
Unary length limit (U_{max})	$[8-32]$	16	Limits the maximum length of any encoded sample.

Afterwards, Figures like 5.6 are obtained for each of the described parameters. In this document, we only show the results of the influence of the number of bands used for prediction and the weight component resolution because of their particular relevance. We observe that initially the compression improves as P becomes higher, but it becomes almost stable for $P > 3$. Hence, we can conclude that setting $P > 3$ does not yield a significant improvement in the compression ratios. Furthermore, it can be seen that the weight component resolution, Ω, also affects the compression in a significant way, producing smaller compressed images for higher values of Ω. Regarding the local sums and prediction modes, we note that reduced mode in combination with column-oriented local sums yields higher compression for raw (uncalibrated) input samples from pushbroom imagers that exhibit significant along-track streaking artifacts. On the other hand, full mode in combination with neighbour-oriented local sums is better suited for whiskbroom, frame imagers and calibrated imagers.

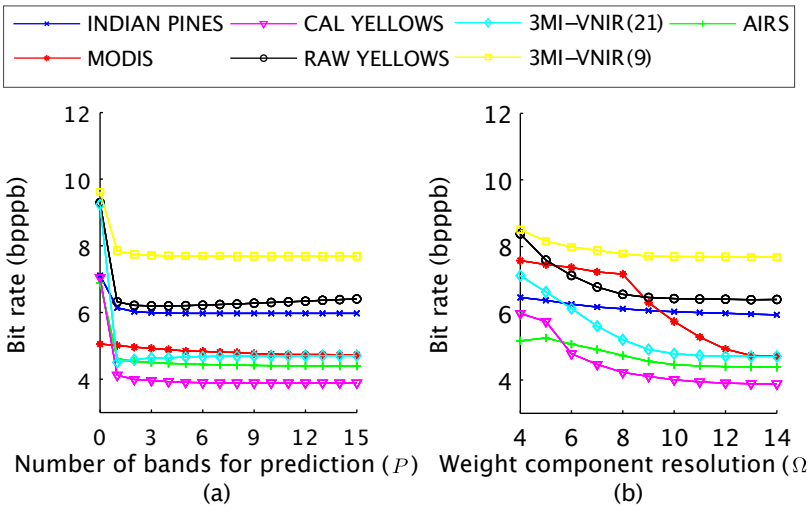


FIGURE 5.6: Influence of the user-defined parameters in the compression ratio. (a) Number of bands used for prediction (P); (b) Weight component resolution (Ω)

Tables 5.6 and 5.7 summarize the main conclusions about the effect of all the user-defined parameters of the CCSDS 123 algorithm in the compression ratio. We note that a similar study was performed in UAB, which corroborates our conclusions [104].

5.5 Architectural design considerations

As we have described in the previous Section, the CCSDS 123 standard allows a fair amount of user-defined parameters which can be set by a potential user to optimize the compression ratio according to his needs or the characteristics of the image sensor. When a hardware architecture is to be conceived, the effect of these parameters in the complexity of the design must be additionally taken into consideration.

TABLE 5.6: Influence of the user-defined parameters of the predictor in the compression ratio

Parameter	**Default in Empordá**	**Conclusions**
Number of bands for prediction (P)	15	Higher P yields better compression, however setting $P > 3$ does not show improvement.
Prediction mode	full	Raw images → Reduced+Column; Calibrated images → Full+Neighbour
Local sum mode	neighbour	
Register size	32	Doest not have a significant impact. It has to be large enough to prevent overflow.
Weight component resolution (Ω)	13	Has a noticeable impact. A large Ω yields more compression.
Weight update scaling exponent change interval (t_{inc})	6	Does not have a significant impact.
Weight update scaling exponent initial parameter (ν_{min})	-1	Does not have a significant impact.
Weight update scaling exponent final parameter (ν_{max})	3	Has a moderate impact. In general, better compression is achieved for higher ν_{max}.

TABLE 5.7: Influence of the user-defined parameters of the entropy coder in the compression ratio

Parameter	**Default in Empordá**	**Conclusions**
Initial count exponent (γ_0)	1	Does not have a significant impact.
Accumulator initialization constant (k'_z)	6	Does not have a significant impact.
Rescaling counter size (γ^*)	6	Does not have a significant impact.
Unary length limit (U_{max})	16	Does not have a significant impact.

In this research work, our main goal is to design a low-complexity architecture which is flexible enough to allow the setting of most of the parameters included in the standard, without significant changes in the hardware occupancy or performance. It is worth mentioning that the selected parameters are expected to be chosen and maintained for a specific application, and thus the potential user is expected to set them before the design synthesis.

It is observed in Section 5.4 as well as in [104] that not all the user-defined parameters in the CCSDS 123 standard have an important impact in its compression performance. For instance, most of the parameters of the entropy coder do not have a significant effect in the compression ratio, nor in the hardware complexity. However, the setting of some of the parameters in the predictor, in particular the number of bands used for prediction P, the prediction mode and the local sum mode, are important to optimize the performance in terms of compression efficiency and final hardware implementation complexity. Next, the design considerations taken into account regarding these parameters are summarized. The encoding order will only affect the compression ratio when the block-adaptive entropy coder is utilized. However, in this research work, we only consider the implementation of the sample-adaptive coder. Despite the encoding order not having a real effect in the compression ratio, it is critical when a hardware architecture is to be designed. Therefore, its influence is also considered.

5.5.1 Encoding order

As it was already mentioned, the samples can be encoded in band-sequential (BSQ) or band-interleaved (BI) order. In BSQ order, all the samples in a particular band are compressed in raster order before the compression of the following band. On the other hand, in BI order, a particular sample is compressed in all spectral channels (or a subframe of channels) before the compression of the next sample in raster order.

Although the encoding order and the prediction order can be different according to the CCSDS 123 standard, the samples should be predicted in the same order as they are encoded, if low-complexity of the implementation is desired. The encoding order defines how the input samples are read and therefore it has an important impact in the architectural design. We assume that the raw samples are stored in an external mass memory which features burst reading and we design 5 architectures with different encoding orders and study their complexity, as it is shown in Section 5.5.4.

5.5.2 Local sum mode and prediction mode

As it was aforementioned, the local sum can be obtained from one previously processed neighbour (column-oriented) or from four neighbours (neighbour-oriented). Furthermore, the user can also select between full or reduced prediction modes.

Regarding the compression ratio, we saw that the use of reduced mode in combination with column-oriented local sums tends to yield smaller compressed image data volumes for raw (uncalibrated) input samples, while the use of full mode in combination with neighbour-oriented local sums tends to yield smaller compressed image data volumes for calibrated imagery.

With respect to the hardware complexity, we observe that for any combination of these parameters the previously processed neighbouring samples of $s_{z,y,x}$ are necessary for its compression. These previously processed neighbours can be stored in a set of FIFOs at the compressor's input, in order to reduce the number of accesses to the external input memory. The size of these FIFOs can be reduced if for every sample to be compressed, not only $s_{z,y,x}$ is read, but also the top right neighbour $s_{z,y-1,x+1}$ is read, as it is illustrated with more detail in Section 5.5.4.

5.5.3 Number of bands for prediction

The number of bands used for prediction P is one of the parameters with the greatest impact in the compression ratio as well as in the computational complexity.

It is observed in Section 5.4 and in [104] that the higher P, the higher the compression efficiency. Nonetheless, it is also illustrated that $P > 3$ does not yield a significant improvement in the compression ratio. The presented architectural design is optimized taking this fact into consideration. In order to study the impact of P in the computational complexity of the compressor, we consider how it affects the necessary internal memory storage and the number of operations in the compression process.

Parameter P and the prediction mode define the size of the local differences vector $U_{z,y,x}$. We define C_z as the size of the local differences vector. $C_z = P$ if reduced prediction mode is selected and $C_z = P + 3$ under full prediction mode. As it was explained in Section 5.2, the weighted sum of vector $U_{z,y,x}$ is necessary to perform the prediction of the current sample, therefore C_z is proportional to the number of multiplications and accumulations needed to perform the prediction. The weights conform a vector, $W_{z,y,x}$, which has the same length as $U_{z,y,x}$ and is updated after the prediction of every sample $s_{z,y,x}$.

Furthermore, if we consider BSQ encoding order, we note that when sample $s_{z,y,x}$ is going to be compressed, the local differences vector of sample $s_{z-1,y,x}$ has to be available. In order to meet this requirement, two approaches are possible: one is to store all the local differences vectors of all the samples in the current band, so that they are available when processing next band (Option STORE). The alternative is to re-calculate the local differences vector for all the co-located samples in the P previous bands (Option CALCULATE). The complexity of both options is compared in Table 5.13, in terms

TABLE 5.8: Effect of parameter P in the computational complexity

	Option STORE	**Option CALCULATE**
Elements stored for local differences vectors.	$P \times (N_x \times N_y - 1) + C_z$	C_z
Elements stored for the weight vector.	C_z	C_z
Number of operations per sample to compute predictor.	C_z (×) + C_z (+)	C_z (×) + C_z (+)
Number of operations to compute the local differences vector.	4 (+)	$C_z \times 4$ (+)

of the hyperspectral image dimensions, Nz, Ny and Nz. Symbols (+) and (×) stand for addition and multiplication operators respectively.

As it is observed, Option STORE requires a fair amount of memory storage, which is proportional to the spatial size of the image. As it will be demonstrated in Section 5.5.4, the amount of storage can be easily higher than the internal storage available on an FPGA. On the other hand, re-calculating the local differences as suggested in Option CALCULATE avoids the storage but increases the number of operations needed per compressed sample proportionally to Cz.

5.5.4 Hardware complexity estimation

Finally, 5 different architectures combining the aforementioned options are designed and compared. For all the options it is considered that the uncompressed samples are stored in an external input memory. Table 5.9 summarizes the different architectural options and determines for each of them the selected encoding order, if the local differences vector is stored or re-calculated and if the top right neighbour $s_{z,y-1,x+1}$ is read together with the current sample $s_{z,y,x}$.

TABLE 5.9: Summmary of the proposed architectural options

Option	Encoding order	Local differences vector	Samples read
1	BSQ	STORE	Current
2	BI	STORE	Current and top right
3	BSQ	CALCULATE IN PARALLEL	Current
4	BSQ	CALCULATE IN PARALLEL	Current and top right
5	BSQ	CALCULATE SERIALLY	Current and top right

We assess the complexity of each option in terms of the user-defined parameters, in particular those which affect the most, which are the number of bands used for prediction (P) and the weight component resolution (Ω). With respect to the type of local sums and prediction mode, we assume for all the following estimations that neighbour-oriented local sums and full prediction are utilized, since this configuration is relatively more complex. If a different configuration is selected for the local sum and prediction mode, the user will just experience a slightly less complex implementation, as will be demonstrated during the experimental results (Section 5.7).

Option 1

In this architectural option, the samples are read in BSQ order. A FIFO is used to store the $Nx + 1$ previously processed samples in an image line, so that the necessary neighbours to perform the prediction of the samples in the following line can be easily obtained. The P elements of the local differences vector are stored for every sample in a specific band. For each compressed sample, all the elements of the local differences vector have to be computed.

Option 2

We now consider that the samples are read in BI order. When BI order is utilized, only one element of the local differences vector, $U_{z,y,x}$, has to be updated after the compression of a sample. As in Option 1, a FIFO

is used at the input to store the previously processed samples. In order to reduce the size of this input FIFO, it is decided to read the top right neighbour, $s_{z,y-1,x+1}$ together with the current sample to be compressed, $s_{z,y,x}$. The read samples are appropriately arranged in the input FIFOs, in such a way that they become the necessary neighbour for the compression of the subsequent samples. This implies that, for neighbour-oriented local sums, three neighbours have to be saved in the FIFOs for each band.

Option 3

The samples are read in BSQ order, as in Option 1. However, in this case, instead of storing the local differences vector, all the elements of the vector are re-calculated in parallel for each compressed sample. As a consequence, the input FIFOs will store not only the $Nx+1$ previously processed samples, but also the co-located samples in the P previous bands. This way, the necessary neighbours for computing the local sum and the local differences in the current and the previous bands, which will be used to calculate $U_{z,y,x}$, can be easily obtained.

Option 4

The samples are read in BSQ order, as in Option 3, but we now focus on reducing the size of the input FIFOs. To accomplish this, the current sample $s_{z,y,x}$ is read from the input memory together with the top right neighbour $s_{z,y-1,x+1}$ in the current band and in the P previous ones. The local differences vector, as in Option 3, is re-calculated in parallel for every sample.

Option 5

The samples are read likewise in BSQ order and the current sample $s_{z,y,x}$ is read together with the top right neighbour $s_{z,y-1,x+1}$ for the current and the P previous bands. In order to reduce the amount of resources needed in

Option 4, the elements of the local differences vector are calculated serially instead of in parallel.

5.5.4.1 Comparison and complexity estimation

A comparison of the necessary internal memory storage, hardware resources and number of accesses to the external memory is performed for each of the architectural options.

To compute the amount of memory needed we take into account the size of the input FIFOs as well as the storage of any other data which is necessary to perform the compression of a sample, such as the local differences vector or the weight vector. Table 5.10 shows an estimation of the memory storage needed by every option, expressed as a function of the following factors:

- The number of bands, Nz, lines, Ny and columns, Nx in the image.
- The bit-width of the input samples, D.
- The bit-width of the weight components, $WCR = \Omega + 3$, which is a function of the weight component resolution.
- The number of bands used for prediction, P.
- The number of elements in the local differences and weight vectors, Cz
- The bit-width of the entropy coder accumulator and counter, $AccR$

Afterwards, a subset of 3D remote sensing images compiled by the CCSDS is utilized as examples of possible compression targets, since as we have seen, the size of the image is also associated with the final hardware complexity. Table 5.11 summarizes the multispectral and hyperspectral images used for evaluation as well as their size and the bit width of the pixel samples. This

TABLE 5.10: Estimation of the memory storage (bits) needed by the proposed architectural options.

Option	**Estimated memory storage (bits)**
1	$(Nx{+}1){\cdot}D{+}Nx{\cdot}Ny{\cdot}P{\cdot}(D{+}4){+}3{\cdot}(D{+}4){+}Cz{\cdot}WCR{+}2{\cdot}AccR$
2	$3 \cdot Nz \cdot D + Cz \cdot (D+4) + Nz \cdot Cz \cdot WCR + (Nz+1) \cdot AccR$
3	$(P+1) \cdot Nx \cdot D + Cz \cdot (D+4) + Cz \cdot WCR + 2 \cdot AccR$
4	$5 \cdot (P+1) \cdot D + Cz \cdot (D+4) + Cz \cdot WCR + 2 \cdot AccR$
5	$3 \cdot (P+1) \cdot D + 2 \cdot D + Cz \cdot (D+4) + Cz \cdot WCR + 2 \cdot AccR$

TABLE 5.11: Target images used to evaluate the complexity of the proposed architecture

Image	N_z	N_x	N_y	**Pixel bit width**
SPOT5	3	1024	1024	8
PLEIADES	4	224	2456	12
LANDSAT	6	1024	1024	8
MSG	11	3712	3712	10
MODIS1	14	1354	2030	12
CASI	72	405	2852	12
AVIRIS	224	680	512	16
SFSI	240	496	140	12
HYPERION	242	256	1024	12
AIRS	1501	90	135	14
IASI	8461	66	60	12

information is extracted from the data in [74]. We assume the most complex configuration of local sum and prediction mode, i.e. neighbour oriented local sum and full prediction. As it was aforementioned, $P > 3$ does not yield a significant improvement in the compression ratio, therefore the number of previous bands used for prediction is set to $P = 3$, as an example of a typical use, and to $P = 15$ as a worst-case scenario in terms of complexity.

TABLE 5.12: Internal memory storage (Kbits) needed by the proposed architectures

Image	**Option 1**		**Option 2**		**Option 3**		**Option 4**		**Option 5**	
	$P = 3$	$P = 15$	$P = 3$	$P = 15$	$P = 3$	$P = 15$	$P = 3$	$P = 15$	$P = 3$	$P = 15$
SPOT5	37758	37758	1	1	33	33	1	1	1	1
PLEIADES	26410	35213	1	1	12	14	1	1	1	1
LANDSAT	37758	75506	2	2	33	58	1	1	1	1
MSG	578753	2121995	2	4	149	446	1	2	1	1
MODIS1	131951	615708	3	6	66	245	1	2	1	2
CASI	55448	277220	12	26	20	79	1	2	1	2
AVIRIS	20901	104460	40	83	44	175	1	2	1	2
SFSI	3340	16672	40	86	25	96	1	2	1	2
HYPERION	12587	62919	40	87	13	50	1	2	1	2
AIRS	658	3283	256	544	6	21	1	2	1	2
IASI	192	952	1388	3013	4	14	1	2	1	2

TABLE 5.13: Differences in terms of hardware resources needed by the proposed architectures

	Options 1, 2 and 5		**Options 3 and 4**	
Operation	**#Adders**	**# Multipliers**	**# Adders**	**# Multipliers**
Computing local sum	3	0	$3 \times P$	0
Computing local differences	4	0	$4 \times Cz$	0
Computing predictor	1	1	Cz	Cz

The results in terms of internal memory storage are shown in Table 5.12. We observe that Option 1 takes significantly more storage resources than the rest, which might be higher than what is available in many FPGA models. This fact would force the addition of an external data memory to store the local differences values. Options 4 and 5 take significantly less resources.

Table 5.13 shows the main differences of the proposed architectures in terms of computational complexity. It is observed that Options 3 and 4 have a higher complexity, which is proportional to the number of bands used for prediction P. As it was already mentioned, $C_z = P$ if reduced prediction mode is selected and $C_z = P + 3$ under full prediction mode.

Finally, we consider the number of memory accesses per compressed sample for the different architectural options. The results are summarized in Table 5.14. As it was mentioned, for all the options, we assume that the uncompressed samples are stored in an input memory. However, for the particular case of Option 1 it is considered that the P local differences are stored in an additional external data memory, considering the data displayed in Table 5.12.

Taking into account the aforementioned results and the initial goal of achieving low complexity, we find that the best trade-off between hardware utilization and external memory accesses is found in Option 5, since it makes it

TABLE 5.14: External memory accesses per compressed sample for the different architectures

Option	**Accesses per compressed sample**
1	$1 + P$
2	2
3	1
4	$2 \times (P + 1)$
5	$2 \times (P + 1)$

possible to compress any of the images without the need of an additional external data memory to store the local differences values, and features also the lowest complexity in terms of the necessary logic to compute the mathematical operations at the prediction stage. The fact that it requires more accesses to the input memory per sample than the alternative options can yield an increased latency for high values of P. However, considering that it is demonstrated that there are no benefits in terms of compression ratio if $P > 3$, we find that it is possible to mask any possible reading latency if the external input memory features burst reading and the sensed samples are appropriately allocated in it, in such a way that the necessary samples are obtained in a single read operation. The latency can be further avoided if obtaining new samples from the external input memory is overlapped with the compression operations.

5.6 HyLoC Hardware architecture description

Once Option 5 has been selected as the most convenient architectural option (see Section 5.5.4.1), the schematic of the HyLoC compressor is created, as shown in Figure 5.7 and each module is then described in VHDL. The behaviour of the compressor is explained with more detail next.

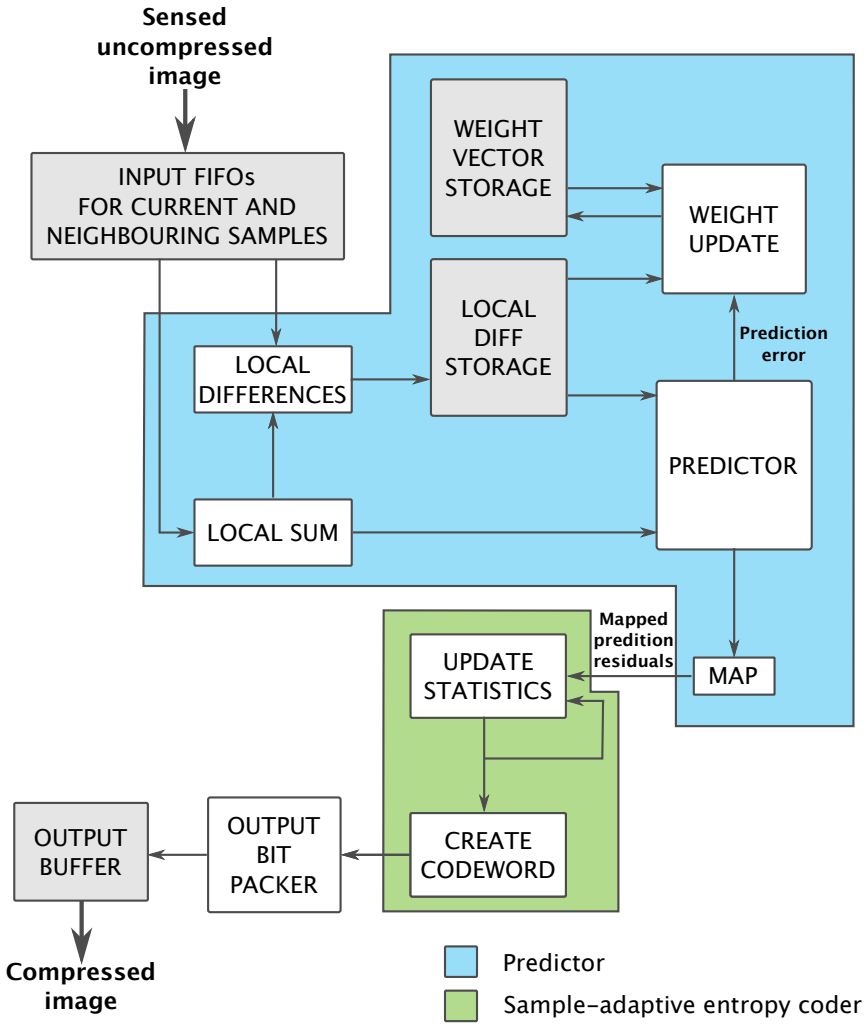


FIGURE 5.7: HyLoC schematic

According to what it was established in the selected architectural option, the samples are compressed in BSQ order, as it is specified by the following set of nested loops:

1: $for(z = 0;\ z < Nz;\ z++)$
2: $\quad for(y = 0;\ y < Ny;\ y++)$
3: $\quad\quad for\ (x = 0;\ x < Nx;\ x++)$
4: $\quad\quad\quad compress\ (s_{z,y,x})$

First, the necessary raw samples are read from the input memory. For the compression of each sample $s_{z,y,x}$, the following data have to be obtained:

- The current sample to be compressed, $s_{z,y,x}$.
- The co-located samples in P previous bands, $s_{z-1,y,x}$ ... $s_{z-P,y,x}$.
- The top right neighbour of the current sample, $s_{z,y-1,x+1}$.
- The top right neighbours of the current sample in P previous bands, $s_{z-1,y-1,x+1}$ ... $s_{z-P,y-1,x+1}$.

The read samples are stored in the input FIFOs and arranged in such a way that they can be easily accessed for the compression of the following samples, with the scheme shown in Figure 5.8.

Once the neighbouring and current samples are available, the local sum and local differences in the current and previous bands are calculated and utilized to compute the prediction. The weights are then updated according to the prediction results. Finally, the prediction errors are mapped and encoded, as specified by the CCSDS 123 standard.

The described architecture is implemented at RTL level. The resulting IP core is referred to as HyLoC and described in VHDL for its implementation

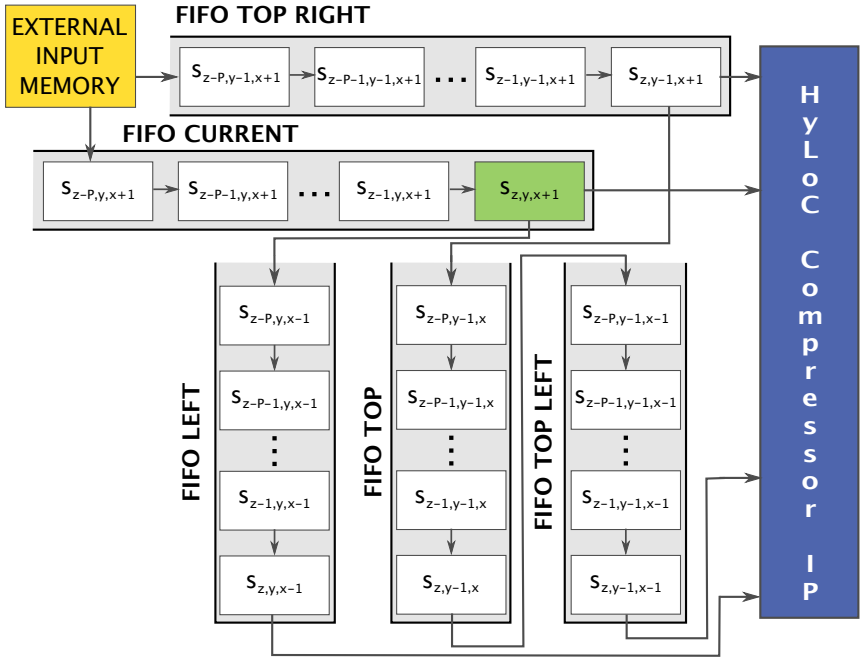


FIGURE 5.8: HyLoC input buffers to arrange current samples and neighbours

on a space-qualified FPGA. The obtained VHDL source code is completely independent from the implementation technology, and therefore can be utilized to implement the design in any FPGA or as an ASIC. The top module of the HyLoC compressor is shown in Figure 5.9.

As it has been explained, HyLoC accepts uncompressed data and generates the compressed bit stream according to the CCSDS 123 standard algorithm. Two FIFOs are used to interface with the external input memory (FIFO TOP RIGHT and FIFO CURRENT in Figure 5.8). These FIFOs store the read samples (the current sample to be compressed and its top right neighbour) and make them available for the compressor. A simple handshake protocol is utilized to indicate the compressor that the necessary data are available

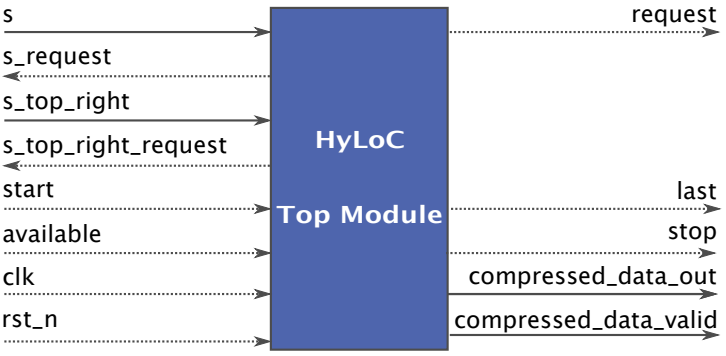


FIGURE 5.9: HyLoC top module

in the FIFOs. Finally, a flag is utilized to validate the output of the HyLoC compressor.

5.6.1 HyLoC verification and validation

The HyLoC IP core is simulated and extensively verified using ModelSim targeting several real calibrated and uncalibrated hyperspectral images and a wide combination of configuration parameters. A testbench is designed, following the schematic from Figure 5.10. The original uncompressed samples are sent to the HyLoC compressor, and afterwards, the resulting compressed bit streams are compared bit by bit with those obtained for the same hyperspectral images utilizing the Empordá as golden reference software implementation of the CCSDS 123 algorithm [105].

Finally, a demonstrator of the HyLoC IP core is created in order to validate the correct behaviour of the compressor when it is running on a real FPGA. We do not consider creating a prototype by using a the space-qualified antifuse FPGAs of the RTAX family, since they are one-time programmable and expensive. Instead, a commercial development board is selected, namely the

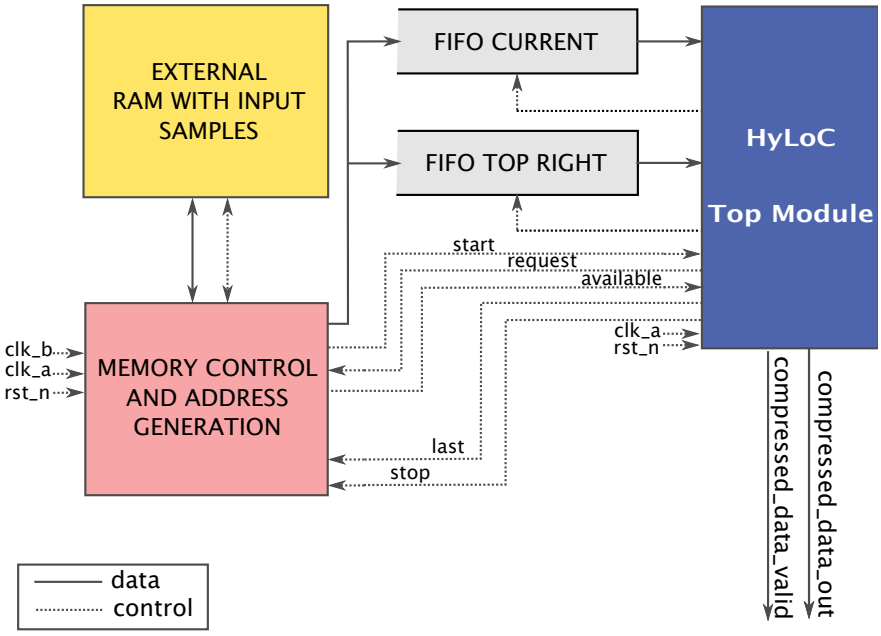


FIGURE 5.10: HyLoC testbench schematic

ML507 from Xilinx. We opted for this prototyping board because it was the one available at the moment this demonstrator was designed and because it gives the possibility of having a working implementation on an FPGA in a relatively short time. The ML507 includes a Virtex 5 FPGA, with an embedded PowerPC. This makes it possible to simplify the execution of some of the operations, in particular the interfaces with the external input memory, by performing them in software. The downside of this decision is a potential loss in performance, because the input and output operations are expected to be slower than the compression itself. However, the purpose of the demonstrator is to validate the HyLoC compressor at work, and verify that, once implemented on an FPGA, it exhibits the same behaviour which had been observed in the simulations.

The development board and its several components are shown in Figures 5.11 and 5.12. The demonstrator is designed as shown in the schematic of Figure 5.13. All the components communicate through the Processor Local Bus (PLB). The raw hyperspectral image samples and the corresponding compressed bit streams generated with the golden reference software Empordá are stored in the CompactFlash card. The PowerPC processor takes care of copying the images to the SDRAM, from which they are read and sent to the HyLoC compressor. The resulting compressed stream is sent to the SDRAM via the PLB bus and from there read by the PowerPC and compared with the reference compressed bit stream.

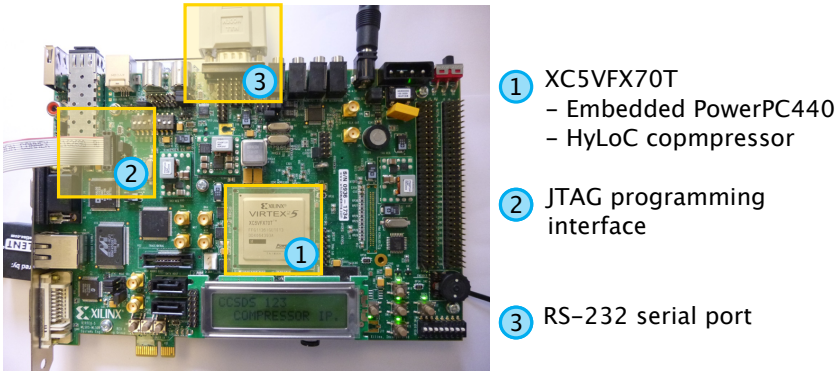


FIGURE 5.11: Development board used in the HyLoC demonstrator (I)

Three different images, shown in Figure 5.14 were successfully compressed in the HyLoC demonstrator, with two different configuration options, a baseline configuration and a less complex one. The number of bands for prediction, local sum mode and prediction mode were tuned as follows:

- Baseline configuration: $P = 3$, neighbour-oriented local sum, full prediction.

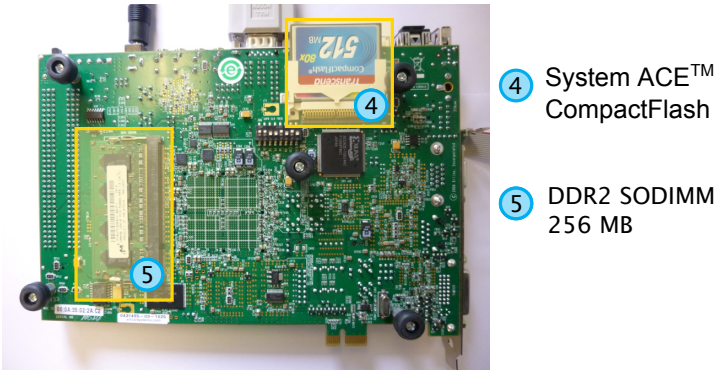


FIGURE 5.12: Development board used in the HyLoC demonstrator (II)

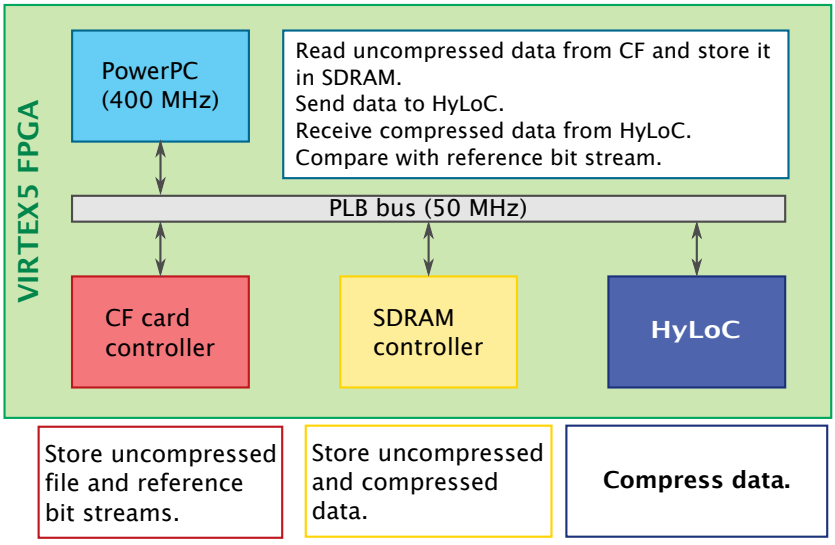


FIGURE 5.13: Schematic of the HyLoC demonstrator

- Less complex configuration: $P = 1$, column-oriented local sum, reduced prediction.

FIGURE 5.14: Hyperspectral images used in the HyLoC demonstrator. The dimensions are given in $Nz \times Ny \times Nx$

5.7 Experimental results

The designed hyperspectral compressor IP is finally synthesized on a space-qualified RTAX1000S FPGA from Microsemi.

HyLoC is configured to compress a hyperspectral image from the AVIRIS sensor, specifically the Yellowstone Uncalibrated Scene 0 whose samples are represented with 16 bits unsigned integers. It comprises 224 bands and contains 512 lines and 680 pixels per line. In order to explore how the different user-defined parameters affect the hardware complexity, we synthesize 4 different configurations for the local sum and prediction mode, particularly all the possible combinations between column oriented and neighbour oriented local sums and full and reduced prediction. We set the number of previous bands used for prediction to $P = 3$. The synthesis results in terms of hardware occupancy and maximum frequency are shown in Table 5.15.

TABLE 5.15: HyLoC synthesis results on an RTAX1000S

	Number of bands used for prediction $P = 3$			
	Reduced prediction		**Full prediction**	
	Neighbour	**Column**	**Neighbour**	**Column**
Combinational cells	4354 (36%)	4166 (34%)	3971 (34%)	3713 (34%)
Sequential cells	1490 (36%)	1485 (34%)	1344 (34%)	1233 (34%)
I/O cells	75	75	75	75
Max. Frequency	43.4 MHz	43.9 MHz	43.0 MHz	43.4 MHz

TABLE 5.16: HyLoC synthesis results on an RTAX1000S for the most complex conguration

	Number of bands used for prediction $P = 15$
	Neighbour oriented and full prediction
Combinational cells	5162 (43%)
Sequential cells	2688 (44%)
I/O cells	75
Max. Frequency	43.3 MHz

We explore next a worst-case scenario by setting the configuration parameters to those which would result in the highest hardware complexity, i.e. neighbour oriented local sum, full prediction and $P = 15$. Table 5.16 illustrates the resulting hardware occupancy and maximum frequency. We observe the hardware occupancy increases significantly, from 34% to 44%.

The maximum throughput depends likewise on the selected configuration parameters. In particular, since the local differences are calculated serially, the number of cycles necessary to compress a sample will increase with P. The rest of the parameters only affect the maximum frequency of the design, as Tables 5.15 and 5.17 show. For the most complex configuration and $P = 15$, the throughput is 28 Mbits/sec, which is around half of the throughput

TABLE 5.17: Maximum throughput for different configurations of HyLoC

	Number of bands used for prediction $P = 3$			
	Reduced prediction		**Full prediction**	
	Neighbour	**Column**	**Neighbour**	**Column**
Max. Throughput (Mbits/sec)	69	70	57	58

achieved with $P = 3$. These results reinforce even more the rationale for keeping $P \leq 3$, because the opposite increases the hardware complexity and lowers the throughput without a significant gain in compression ratio.

We note finally, that the RTAX1000S implementation of HyLoC exhibits very low power consumption, estimated at 93 mW.

5.8 Comparison of hardware technologies for the implementation of hyperspectral image compression algorithms

As part of this research work, described in Chapters 3, 4 and the present one, several technological options have been studied for the implementation of a lossy and a lossless prediction-based algorithm for hyperspectral image compression, namely the LCE and the CCSDS 123 standard. We provide a comparison of the results obtained with the technologies explored, in particular GPU and FPGA, with references to relevant work of the state-of-the-art found in the available literature (see Chapter 2).

In all the results presented in what follows, we consider that the target for compression is the AVIRIS Scn0 raw image, which comprises 512 lines, 680 columns and 224 bands. For the HyLoC compressor, which implements the CCSDS 123 standard algorithm, we set the configuration for all the presented results to the baseline configuration settings: full prediction and neighbour

oriented local sums; number of bands for prediction, P, is set to 3; the rest of the parameters are set to the default values established in Section 5.4.

The implementations presented in this Section are compared in terms of the available metrics, depending on the specific technology considered including: hardware occupancy, power consumption and throughput. The latter is calculated depending on the technology as:

$$\begin{matrix} Throughput\ FPGA \\ (MSamples/sec) \end{matrix} = 10^{-6} \times \frac{Nz \times Ny \times Nx}{total\ cycles \times (clock\ frequency)^{-1}} \tag{5.5}$$

$$\begin{matrix} Throughput\ GPU\ and\ CPU \\ (MSamples/sec) \end{matrix} = 10^{-6} \times \frac{Nz \times Ny \times Nx}{total\ execution\ time(sec.)} \tag{5.6}$$

The throughput is also given in Mega-bits per second (Mbps), and is obtained by just multiplying the number of Mega-samples per second by the number of bits in an image sample. In all the tables that follow NDA stands for "No Data Available".

5.8.1 Implementations on GPUs

An additional assessment of the results obtained with the algorithms which are subject of study for this Thesis, namely the LCE and the CCSDS 123 algorithm, is presented next. We consider the acceleration which can be obtained with the hardware technologies considered for this work: GPUs and FPGA. The comparison is performed in terms of throughput, considering the best results that have been reported in the literature or obtained as part of this research work. For the particular case of the GPU implementation of

TABLE 5.18: Hardware technologies with the best reported throughput for the LCE and CCSDS 123 algorithms

Technology	**LCE**	**CCSDS 123**
CPU	Intel Xeon W5580 3.19 GHz	Intel Core i7 2.4 GHz
GPU	Nvidia Tesla C2075	NVidia GeForce 560M GTX
FPGA	Virtex 5VFX130	Virtex 5VFX130

the CCSDS 123 algorithm, we consider the throughput results obtained in [81]. For the sake of comparison, we also include the throughput obtained when the algorithms are executed on single-threaded CPUs.

The technologies where the best results have been obtained in terms of throughput for each algorithm are summarized in Table 5.18.

Finally, the comparison of the best achieved throughput figures are shown in Figure 5.15. This, together with the results presented in Chapters 3 and 4, shows that GPUs exhibit a superior performance in terms of throughput at the expense of a significantly higher power consumption.

5.8.2 Implementations on FPGA

We implement the HyLoC IP core on a Virtex 5 (5VFX130) in order to make a comparison with the FPGA implementation of the lossy compression algorithm for hyperspectral images LCE presented in Chapter 4. The aforementioned algorithm is also prediction-based, but introduces losses by quantization and rate-distortion optimization. As part of this research work, it was implemented on an FPGA using high-level synthesis (HLS) tools.

The comparative results in terms of hardware occupancy can be seen in Table 5.19. We explore as well the throughput and the maximum frequency

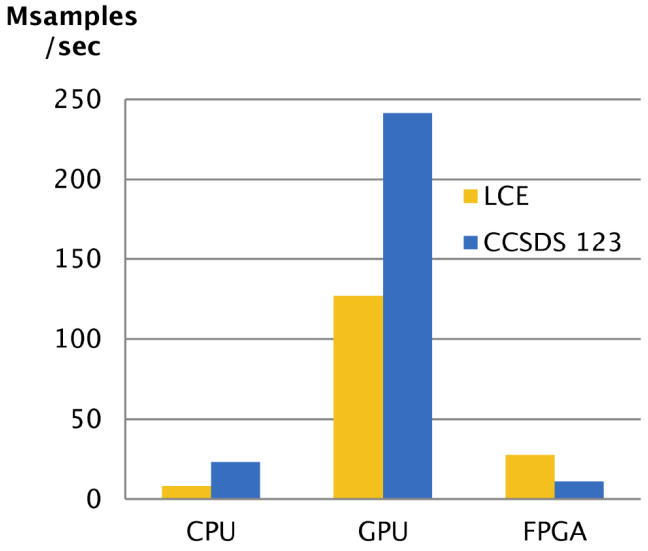


FIGURE 5.15: Throughput of the different technologies. Best achievable cases for the LCE and the CCSDS 123 algorithm on CPU, GPU and FPGA

TABLE 5.19: Occupancy lossy LCE and lossless HyLoC (CCSDS 123) on a Virtex 5

Virtex 5VFX130		**LCE**		**HyLoC**	
Resources	***Available***	***Used***	%	***Used***	%
BUFGs	32	1	3	1	3
DSP48Es	320	25	8	1	1
Number of RAMB18X2s	298	4	1	0	0
Number of slices	20480	1935	10	842	4
Number of Slice Registers	81920	5995	7	1535	1
Number of Slice LUTS	81920	7738	10	2342	2

and power consumption for the FPGA implementations of the LCE and the HyLoC compressor on the aforementioned Virtex 5VFX130. The results can be observed in Table 5.20.

TABLE 5.20: Throughput of the lossy LCE and lossless HyLoC (CCSDS 123) on a Virtex 5

	Virtex 5VFX130	
	LCE	**HyLoC**
Max.Frequency (MHz)	86	134
Throughput MSamples /sec	27.7	11.30
Throughput Mbps	443	180
Total Power (mW)	2679	2354

We observe in Table 5.19 that the hardware occupancy of the LCE algorithm is higher than that of HyLoC. This is due to the mathematical operations involved in the quantization and rate-optimization stages of the LCE algorithm. The prediction in the LCE algorithm is simpler than that in HyLoC, in the sense that a smaller amount of previous samples are necessary. This makes it possible for the LCE to achieve a higher throughput due to the presence of less data dependencies. Besides, the LCE algorithm operates on independent blocks, what means that the throughput can be further increased by synthesizing more than one core. This is perfectly feasible, considering the presented hardware occupancy results. The HyLoC core, on the other hand, is meant to compress the entire image without dividing it into blocks. Nevertheless, the CCSDS 123 standard permits to partition the image also in independent blocks for compression. The effects of partitioning the image in the compression efficiency are not discussed in the standard, nor as part of this research work. The power consumption of HyLoC is around 15 % less than the LCE algorithm.

5.8.2.1 Comparison with state-of-the-art FPGA implementations of hyperspectral compression algorithms

The FL algorithm, presented in [55], is the basis of the CCSDS 123 standard. Both algorithms perform the same operations, however they have some differences. The CCSDS 123 allows more parameters and the use of the block-predictor entropy coder, which is not considered by the FL algorithm. An FPGA implementation of it is described in [55]. In particular, the implementation target is a commercial Virtex IV. The FL algorithm utilizes 3 previous bands for prediction, which is the same number of bands set in the selected HyLoC configuration. However, the described FPGA implementation of the FL algorithm considers that the samples are read and compressed in BI order, while HyLoC compresses the image in BSQ order. As it was observed in the architectural discussion presented in Section 5.5.4, the use of BI has the potential to require more hardware resources in terms of storage and at the same time yield a higher throughput.

The comparison in terms of resource usage, frequency and throughput is shown in Table 5.21. In order to make a fair comparison, the RTL descriptions of HyLoC is synthesized for the same FPGA utilized for the FL implementation in [55].

The FL algorithm achieves more throughput than HyLoC, partly due to the use of BI compression order, for which less data dependencies are present. Regarding the resource usage we observe that, HyLoC requires considerably less than the implementation of the FL. Power consumption is likewise slightly lower for HyLoC.

We provide next a comparison of with the near-lossless prediction-based algorithm presented in [56]. This algorithm is similar to the LCE algorithm, and performs basically the same operations, except for the rate-optimization

TABLE 5.21: Virtex IV LX160 device utilization of the FL algorithm and HyLoC

Virtex IV LX160		
Resource	**FL**	**HyLoC**
Slice	67584 (5%)	1788 (2%)
FIFO/RAMB16	9 (3%)	0 (0 %)
DSP48	6 (6%)	0 (0 %)
Max. Frequency (MHz)	33	133
Throughput (Msamples/sec)	33	11.2
Throughput (Mbps)	NDA	179
Power (mW)	1700	1488

TABLE 5.22: Virtex IV LX200 comparison lossless and near-lossless, LCE, HyLoC

Virtex IV LX200			
Resource	**Lossless to near-lossless**	**LCE**	**HyLoC**
Used LUT	10306 (5%)	9283 (5%)	2907 (1%)
RAMB16S	21 of 336 (6%)	4 of 336 (1%)	0 (0%
DSP48	9 of 96 (9%)	25 (26%)	1 (1%)
Max. Frequency (MHz)	81	75	116
Throughput (Msamples/sec)	70	23.8	9.7
Throughput (Mbps)	NDA	380	115
Power (mW)	NDA	NDA	1806

phase, which is included in the LCE algorithm, but is not part of the lossless to near-lossless approach.

The results can be observed in in Table 5.22, where we also include the those obtained for the HyLoC IP core which implements the CCSDS 123 algorithm. All algorithms are implemented on the same Virtex IV model, in order to make a fair evaluation.

The HyLoC implementation takes considerably less hardware resources, however it also yields a considerably smaller throughput. The lossless to near-lossless approach and the LCE display similar results, with a slightly higher occupancy for the LCE algorithm due to the increased amount of mathematical operations involved.

5.8.3 Implementations on space-qualified FPGAs

We compare now the implementation of the LCE algorithm and HyLoC (CCSDS 123 standard) on space-qualified FPGAs. Although the results presented for HyLoC in 5.7 are those obtained for an RTAX1000S from Microsemi, we present now results on an RTAX2000S for the sake of comparison and due to its increased amount of hardware resources, which are needed since the LCE algorithm does not fit in the RTAX1000S.

The comparative results in terms of occupancy can be seen in Table 5.23. We include also the results of the FPGA implementation of the CCSDS 122 standard for satellite image compression presented in [88].

The CCSDS 122 algorithm was also conceived for satellite data compression and, unlike the LCE algorithm and the CCSDS 123 standard, is transform-based and targets only bi-dimensional images. The work presented in [88] shows an implementation of this algorithm on an RTAX2000S space-qualified FPGA for two different configurations of the algorithm, which meet the requirements of the EnMap and PROBA-V missions.

In the presented results we can observe that the LCE has a significantly higher occupancy than the others, which suggests that a more careful design of the RTL description of the quantization and rate-optimization stages might be still necessary. HyLoC requires less hardware resources than the

TABLE 5.23: Implementation on an RTAX2000S FPGA: CCSDS 122, LCE and HyLoC (CCSDS 123)

Resource	**CCSDS 122**		**LCE**	**HyLoC**
	Proba-V	**EnMap**		
Combinational C-Cells	8455 (39%)	9772 (45%)	18101(84%)	4282 (13%)
Sequential R-Cells	7125 (66%)	7620 (71%)	10752 (53 %)	1438 (18%)
Total Cells	15580 (48%)	17392 (54%)	23834 (73%)	5720 (18%)
Block RAMs	54 of 64	58 of 64	7 (25 %)	0 (0%)
Max. Frequency (MHz)	50	90	18	41
Throughput (Msamples/sec)	NDA	NDA	5.7	3.5
Throughput (Mbps)	90	130	91	56
Power (mW)	NDA	NDA	1171	1124

CCSDS 122 algorithm, which was expected taking into account that transform-based algorithms are in general more complex than the ones based on prediction. The CCSDS 122 algorithm presents moreover a higher throughput than HyLoC. This suggests that it is interesting to explore the possibility of implementing two cores of HyLoC which would operate at the same time, in such a way that the throughput is potentially doubled. Despite the hardware occupancy being also doubled, it would still be lower than that of the implementations of the CCSDS 122 algorithm. The most important challenges that appear if more than one HyLoC core are expected to be used at the same time are, on one hand, the possible loss in the compression efficiency caused by compressing parts of the same image independently; and on the other hand, the difficulty of creating a single bit-stream representative of the complete compressed image, what would require the inclusion of side-information for the decompressor, as well the definition of additional bit packing operations.

From the presented comparison, we can drive the conclusion that, as expected, both GPUs and FPGAs yield higher throughput than the CPU counterparts. This confirms that compression algorithms can be more efficiently executed in technologies that allow more data parallelization. The GPU provides significantly higher performance results, nevertheless, FPGAs offer the possibility to scale and replicate the cores in order to achieve higher throughput, what makes them potentially as efficient as GPUs at a much lower cost in terms of power consumption. Therefore efforts should be put in making GPU radiation-hardened and reduce their power consumption without significantly reducing their performance. With respect to the FPGAs we note the important advantages in performance and flexibility of the SRAM-based FPGA over the anti-fuse ones. Technological developments in the near-future should promote the use of SRAM-based FPGAs in space, by making them more reliable, which would significantly reduce the cost of missions, providing flexibility, high-performance and low power consumption.

Chapter 6

Conclusions

This Chapter summarizes the main contributions of the Thesis, with special emphasis in their relevance to the topic it covers: hardware technologies for on-board multispectral and hyperspectral image compression. Furthermore it proposes further research topics, which are expected to complement and enhance the future developments of this work.

On-board multispectral and hyperspectral compression is necessary in order to meet the bandwidth and storage limitations in the current space missions. At the same time, it is an important challenge, due to the difficulty of handling the high amount of data captured by the sensors in the hardware technologies available. The computational power of the hardware operating on-board satellites is limited, because it has to meet several requirements to be able to operate in space, including low power consumption and tolerance to sun radiation, among others. This has motivated industries and research centres in the space sector to dedicate efforts in order to design new compression algorithms that are able to reduce the data volume substantially, and at the same time are of low-complexity, so that they can efficiently operate on-board satellites.

As part of the Thesis work, an extensive study of the available literature has been performed, characterizing the current trends in the development of new algorithms for hyperspectral image compression, and the most usual hardware technologies utilized for their implementation. Besides, we have identified the main requirements of both, algorithms and their physical implementations on-board satellites. It was observed that there is a fair amount of lossless and lossy compression algorithms for hyperspectral image compression specifically designed to operate in space, with different degrees of complexity and compression performance figures. However, it is hardly ever demonstrated how well the algorithms perform in the available on-board hardware, nor it is assessed which technology suits the algorithm better or which strategies are necessary in order to implement the algorithm in such a way that it can achieve a high throughput. This Thesis aims at filling the aforementioned gaps, contributing to the development of hardware technologies for on-board hyperspectral image compression. In particular, two hardware technologies are considered, namely graphics processing units (GPUs) and field-programmable gate arrays (FPGAs), and the performance of several compression algorithms on them is evaluated.

Several implementation solutions have been explored for on-board lossy compression compression of hyperspectral images, in particular the Lossy Compression for Exomars (LCE) algorithm [33] has been implemented on a GPU and an FPGA. The main difficulties found when parallelizing the LCE algorithm have been addressed, and several solutions have been proposed and implemented in order to overcome them, including strategies to achieve parallelization of typically sequential operations, like the ones involved in the entropy coding stage of the algorithm. An evaluation of the suitability of GPUs for hyperspectral image compression has been performed, by creating and executing a GPU implementation the LCE algorithm, using Nvidia's parallel architecture (CUDA). The designed implementation yields very high speedups, of up to 15.41 when compared with a single-threaded central processing unit (CPU) execution. The performance in terms of throughput is likewise high, being able to compress between 100 and 140 MegaSamples per second for the hyperspectral images under evaluation.

We perform also the implementation of the decompressor on a GPU. The performance gains obtained with the GPU decompressor are lower than that of the compressor, because of the format of the compressed file, which makes it impossible to identify the independent elements that can be processed in parallel. In this exploration, this issue is solved by introducing additional information in the compressed file, specifically in the form of a header.

Nowadays it is not possible to employ GPUs for on-board satellite image compression, due to their lack of tolerance to sun radiation and intolerable power consumption for space missions. However, the research work accomplished in this Thesis demonstrates that high performance and very flexible implementations can be obtained when GPUs are used for satellite data compression, and contribute to justify and motivate further research in this field, making efforts towards the development of space qualified GPUs.

After implementing the LCE compressor and decompressor on a GPU, the potential benefits and challenges of FPGAs for on-board hyperspectral image compression have been assessed by implementing the LCE compressor on an FPGA. This work enables to compare the performance of both FPGAs and GPUs in the execution of the same algorithm. The FPGA implementation of the LCE has been obtained with the high-level synthesis tool CatapultC. As a result, the low-complexity of the LCE algorithm has been demonstrated, providing an FPGA implementation with low area requirements. This, together with the fact that the algorithm compresses independent blocks of data, offers the possibility of dramatically increasing the throughput by configuring more than one instance of the developed core on the same FPGA.

A comparison between three different physical implementations options, namely CPU, GPU and FPGA, of the LCE algorithm has been carried out, showing that the GPU yields the highest throughput. However, it is also the technology with the highest power consumption, which is 225 W thermal design power (TDP) for the Tesla C2075 GPU used in our experiments. The FPGA implementation can potentially achieve a similar throughput to that of the GPU by instantiating more than one compression core, at a much lower cost in terms of power. In fact, the power consumption of the FPGA implementation of the LCE algorithm is lower than 3 W, which is a much more reasonable figure for space applications.

The work accomplished for the implementation of the LCE algorithm on a GPU and an FPGA makes it furthermore possible to compare the design methodologies followed to obtain the presented implementations. We have observed that it is possible to program a GPU in a relatively short time, with the flexibility of any software implementation and with much more ease than that of an FPGA development. With respect to the FPGA implementation, we have utilized high-level synthesis (HLS) with CatapultC in order to shorten the design times. We can conclude that CatapultC is useful

for obtaining a synthesizable VHDL description of a complex system as it is the case of the LCE compression a algorithm. However, in order to port the C code to register-transfer level (RTL) code in a highly optimized way, many substantial changes had to be performed in the original C source code, with the consequent impact in the development times.

We further demonstrate the benefits of FPGA for on-board hyperspectral image compression by implementing the Consultative Committee for Space Data Systems (CCSDS) standard for lossless multispectral and hyperspectral image compression (CCSDS 123) [25] on a space-qualified FPGA. The presented implementation serves also as a case of use demonstrator, which could be employed nowadays in a real space mission. An architectural study of the effect of the different configuration parameters of the CCSDS 123 standard has been performed, in such a way that we were able to identify which parameters have a greater effect in the compression ratio and the hardware complexity. We have proposed 5 different architectural option, and selected the one which exhibits the best trade-off between flexibility to choose between the desired parameters, and low-complexity. The RTL description of the selected option is created by writing VHDL code from scratch, constituting an IP core named HyLoC. HyLoC was implemented on a space qualified FPGA, particularly on an RTAX1000S, and it was demonstrated that the goal of of low-complexity has been achieved, with a hardware occupancy of 34% for a typical compressor configuration. The presented implementation exhibits also the advantage of having very low power consumption, estimated at 93 mW. The architectural study presented is an additional contribution of this work, which is useful for future developments and improvements HyLoC, or to adapt it to the specific needs of future space missions. To the date, there are no other implementations of the CCSDS 123 in the state-of-the-art which achieve lower occupancy or power consumption than the presented implementation of HyLoC.

This Thesis provides finally a comparison of the results obtained with the explored hardware technologies, i.e. GPUs and FPGAs, with references and comparisons to the most relevant work of the state-of-the-art available in the literature.

We can conclude that both the CCSDS 123 and LCE algorithms good candidates for on-board compression on FPGAs, showing lower or similar hardware occupancy and comparable throughput to other algorithms of the state-of-the-art. Despite the LCE algorithm exhibiting a higher FPGA hardware occupancy, due to the mathematical operations involved in the inclusion of losses in the algorithm, it is perfectly feasible to implement it on a space-qualified FPGAs, in particular we implemented it on the RTAX2000S from Microsemi.

Regarding the throughput of the different hardware technologies which have been used to implement the LCE and the CCSDS 123 algorithms, we observed that, as expected, both GPUs and FPGAs yield higher throughput than the CPU counterparts, demonstrating that the technologies that allow data parallelization are better suited for the implementation of hyperspectral image compression algorithms. While GPUs exhibit significantly better throughput results, their high power consumption and the fact that they are not radiation-tolerant, makes it infeasible to use them in space missions at the moment. On the other hand, FPGAs offer the possibility of scaling and replicate the compression cores in order to achieve higher throughput, what makes them potentially as efficient as GPUs at a much lower cost in terms of power consumption.

With the accomplished research work, the main objectives of this Thesis have been achieved. New solutions for accelerating hyperspectral compression algorithms on GPUs and FPGAs have been proposed, and the improvements obtained by the mentioned technologies have been evaluated and compared

in terms of computational performance, the cost of the solution and the flexibility of the implementation. Moreover, the algorithms have been deeply studied and characterized, and the validity of the proposed implementation solutions have been demonstrated. During the implementation of the algorithms we have been able to compare different design methodologies, including high-level synthesis solutions. Finally, we have identified the common difficulties which appear when trying to parallelize hyperspectral compression algorithms, and provided solutions to solve data dependencies which can be generalized to other algorithms of the same kind. All these contributions are expected to be useful to reduce the cost and improve the performance of future satellite missions in which multispectral and hyperspectral on-board data compression will play a critical role.

6.1 Further research work

Several topics are proposed which might be part of future research work in order to achieve more efficient implementations of hyperspectral compression algorithms for on-board use.

Regarding the parallelization of the LCE algorithm on a GPU, it is still possible to improve the speedup achieved by both the compressor and the decompressor, by applying some optimizations. The efforts should be put in reducing the latency of the GPU - CPU data transactions, by parallelizing these transactions with the kernel executions. On the decompressor side, specifically regarding the attachment of a header, other strategies should be studied in the future in order to optimize the trade-off between the amount of parallelization obtained and the impact in the compression ratio of adding the necessary side information.

On the FPGA implementation side, further improvements are being carried out at this moment, working towards latency reduction, by fully exploiting

the parallelization features of the LCE algorithm with an optimized scheduling of the design, implementing the data control in a highly parallel way. Further optimizations of the quantization and rate-distortion optimization stages can reduce the hardware occupancy of the design, which is relevant if it is desired to implement it on a space-qualified anti-fuse FPGA.

From the point of view of the LCE algorithm itself, we have observed that changing the entropy coder can help to speedup the execution on both the GPU and the FPGA. As it was explained, the Golomb entropy coder used by the LCE algorithm adapts to the statistics of the image. However, in the presence of outliers, the entropy coder adapts badly, and might produce codewords which are represented with more bits than the original data without encoding. Hence, it would be convenient to impose limit on the size of the generated Golomb codewords, what would potentially reduce the size of the compressed file and also reduce the number of iterations needed to generate the codewords, with the consequent positive impact in the performance of the algorithm on the GPU as well as on the FPGA.

Although the implementation of CCSDS 123 algorithm on a space-qualified FPGA requires few hardware resources and power, it would be worth to explore strategies which can yield a higher throughput without a significant impact in the hardware occupancy. A first approach can be to adapt the architecture to operate in band-interleaved order, what would eliminate some data dependencies and hence allow more data parallelization and reduce the amount of clock cycles necessary to compress an image sample. The possibility of dividing the hyperspectral image in blocks and compressing each block independently with the CCSDS 123 algorithm should be also explored, in order to evaluate how the partitioning affects the compression ratio; and allow the possibility of increasing the throughput by implementing more than one core of the CCSDS 123 on the same FPGA.

We have demonstrated that the CCSDS 123 algorithm provides efficient compression at low complexity. However, it is a lossless compression, and in the future it is expected for lossy compression to become more popular, since space missions are expected to require even more reduction of the data volume on-board. For this purpose, it would be useful to count also on a standard for lossy on-board hyperspectral image compression. Two main lines of action are possible: to introduce losses in the CCSDS 123 or to adapt the CCSDS 122 2D image compression standard to operate on three-dimensional data.

Regarding the GPU as a technological option for on-board hyperspectral image compression, it is evident that technological developments are mandatory in order to make them radiation-tolerant and qualify them for space. The biggest challenge in this sense is to reduce their power consumption, while keeping their high performance capabilities. Of all the technologies studied, GPUs yield the highest throughput, however their power consumption is prohibitive for space-missions.

With respect to the FPGAs, we have shown the important advantages in performance and flexibility of the SRAM-based FPGA over the anti-fuse ones. Technological developments in the near-future should promote the use of SRAM-based FPGAs in space, by making them more reliable, which would significantly reduce the cost of missions, providing flexibility, high-performance and low power consumption.

Appendix A

Sinopsis en español

Se presenta en este Capítulo una visión general del trabajo de investigación realizado en esta Tesis Doctoral, poniendo especial interés en las contribuciones en el campo de las implementaciones sobre FPGAs y GPUs de algoritmos para compresión de imágenes hiperespectrales a bordo de satélites.

A.1 Introducción

Las imágenes multiespectrales e hiperespectrales se forman captando la energía reflejada o emitida por objetos terrestres en un gran número de longitudes de onda, dando como resultado un cubo de datos que contiene cientos de bandas. Son capturadas a bordo de un satélite o avión y transmitidas posteriormente a una estación terrena para ser procesadas. Actualmente, los sensores hiperespectrales cuentan con una gran resolución, de manera que el volumen de los datos capturados es muy alto. Sin embargo, la capacidad de almacenamiento y los anchos de banda de transmisión son limitados, lo que hace que en numerosas ocasiones sea indispensable la compresión de la imagen hiperespectral antes de ser enviada a la estación terrena.

Tradicionalmente se han aplicado técnicas de compresión sin pérdidas para reducir la cantidad de información de la imagen hiperespectral. Estas técnicas eliminan ciertas redundancias innecesarias de la imagen, pero permiten reconstruir los datos originales de manera exacta a partir de la imagen comprimida. Por otro lado, también existen las técnicas de compresión con pérdidas, las cuales permiten obtener mayor compresión de la imagen a costa de eliminar definitivamente determinada información. En consecuencia, la imagen recuperada no es idéntica a la original, pero conserva calidad suficiente para poder ser procesada posteriormente. Cuando las imágenes hiperespectrales son captadas por un satélite, la compresión debe realizarse con los equipos y recursos disponibles a bordo, que son limitados en términos de capacidad de computación y potencia, en comparación con los equipos que acostumbramos a utilizar en el sector terreno. Con el fin de cumplir con estos requisitos técnicos, existe un creciente interés en el desarrollo de nuevas técnicas específicas para la compresión de imágenes hiperespectrales. Estas técnicas se han de caracterizar por ser sencillas y consumir pocos recursos, ya que los algoritmos y estándares de propósito general actuales tienden a ser demasiado complejos para ser utilizados a bordo de un satélite.

Los algoritmos de compresión constan generalmente de tres etapas principales: eliminación de redundancias, o decorrelación; cuantificación y codificación entrópica. En el caso de imágenes hiperespectrales es posible encontrar redundancias tanto en el domino espacial como en el dominio espectral. La cuantificación consiste en asignar los valores de entrada a una serie de valores discretos, produciendo consecuentemente pérdidas de información. La codificación entrópica consiste en explotar la probabilidad de los símbolos, asignando palabras de código más cortas a los símbolos más probables y palabras más largas a los menos probables. En cuanto a la etapa de decorrelación, actualmente, los algoritmos de compresión de imágenes hiperespectrales están basados principalmente en técnicas de transformada o en técnicas predictivas. Las técnicas de transformada consisten en aplicar una modificación a los datos para eliminar la correlación entre ellos y poder así representarlos con una cantidad menor de símbolos. Por otro lado, las técnicas predictivas se fundamentan en predecir el valor de la muestra comprimida actual a través de los valores de muestras vecinas que han sido procesadas previamente. El error de predicción, es decir, la diferencia entre la muestra predicha y la actual, se codifica finalmente con un codificador entrópico.

Tal es la importancia de la compresión a bordo de satélites, que el Comité Consultivo para los Sistemas de Datos Espaciales (en inglés, Consultative Committee for Space Data Systems (CCSDS)), que está formado por los representantes más relevantes de las agencias y la industria espacial, ha desarrollado tres estándares para impulsar y facilitar el uso de algoritmos para reducir el volumen de datos en las misiones espaciales: un compresor universal sin pérdidas, conocido como CCSDS-121 [24]; un estándar de compresión con pérdidas o sin pérdidas para imágenes bidimensionales, el CCSDS-122 [34]; y un estándar de compresión sin pérdidas para imágenes multiespectrales o hiperespectrales, el CCSDS-123 [25]. Todos estos estándares cumplen con los requisitos impuestos a los algoritmos que han de a ser ejecutados en el

hardware disponible a bordo de los satélites: alta eficiencia en la compresión y baja complejidad.

Existen asimismo otros algoritmos, específicamente diseñados para la compresión de imágenes hiperespectrales a bordo de satélites, en la literatura científica disponible actualmente. Entre ellos, cabe destacar por su relevancia en el trabajo presentado en esta Tesis Doctoral, el algoritmo para compresión con pérdidas conocido como Lossy Compression for Exomars (LCE). Este algoritmo está basado en técnicas de predición, e introduce pérdidas de información combinando la técnica de cuantificación y una técnica que ajusta el ratio de compresión. Este ajuste permite que la distorsión en la imagen reconstruida no exceda un determinado umbral seleccionado por el usuario.

No solamente es importante desarrollar nuevos algoritmos para la compresión de imágenes hiperespectrales a bordo de satélites. También es necesario considerar el soporte físico (hardware) en que se van a ejecutar, ya que debe cumplir con ciertos requisitos, entre los que destacan la tolerancia a la radiación solar y un bajo consumo de potencia. Generalmente, los algoritmos se implementan en procesadores de a bordo (on-board processor), procesadores digitales de señal (digital signal processor, DSP), circuitos integrados de aplicación específica (application-specific integrated circuit, ASIC), o field programable gate array (FPGA). Una FPGA es un dispositivo que contiene bloques de lógica programable cuya interconexión y funcionalidad puede ser configurada. Recientemente, las FPGAs han alcanzado gran popularidad para la implementación de aplicaciones en el sector aeroespacial, principalmente debido a su alto rendimiento, bajo consumo y la posibilidad de ser reprogramadas. Además, existen versiones en el mercado tolerantes a la radiación y, por lo tanto, aptas para operar en un satélite.

Aparte de los soportes físicos previamente enumerados, en el ámbito de la computación de altas prestaciones se ha popularizado recientemente la

programación de propósito general en tarjetas gráficas (graphics processing unit (GPU)), conocidas como GPU por sus siglas en inglés. Inicialmente, las GPUs se hicieron populares en la industria de los videojuegos por su alta capacidad para el procesamiento masivo de datos en paralelo. En los últimos años se ha ido extendiendo su uso hacia la computación de propósito general, gracias en parte a la aparición de entornos y directivas para la programación de estos dispositivos, como es el caso de Computer Unified Device Architecture (CUDA), desarrollado para la programación de tarjetas gráficas de NVidia. Una GPU consiste en un conjunto de multiprocesadores que trabajan en paralelo en modo SIMD (del inglés Single Instruction, Multiple Data, en español: "una instrucción, múltiples datos"). A diferencia de los procesadores de próposito general o CPUs, las GPUs dedican la mayor parte de sus componentes al cómputo en vez de al control y la memoria, por lo que consiguen acelerar sustancialmente los algoritmos caracterizados por su alto paralelismo a nivel de datos. Tal es el caso de los algoritmos de compresión de imágenes hiperespectrales, que bien podrían utilizar esta tecnología para reducir sus tiempos de cómputo. En este sentido, uno de los objetivos de este trabajo de Tesis será estudiar cómo de convenientes son las GPUs para acelerar los algoritmos de compresión de imágenes hiperespectrales. A pesar de que actualmente las GPUs no están cualificadas para trabajar en el espacio, debido a su elevado consumo de potencia y por no ser tolerantes a la radiación solar, tienen un elevado potencial. Por lo tanto, no se descarta que en un futuro se lleven a cabo los desarrollos tecnológicos necesarios para habilitarlas para ser utilizadas para la compresión de datos a bordo de satélites en misiones espaciales.

A.2 Objetivos y metodolgía de trabajo

Se espera que los sensores hiperespectrales en el futuro capturen cada vez más volumen de datos, mientras que los anchos de banda de transmisión y la

cantidad de almacenamiento disponible se mantendrán relativamente estables. Como resultado, la compresión de datos a bordo se hará indispensable. En este sentido, una compresión eficiente dependerá, tanto del algoritmo utilizado, como de la tecnología física en que se ejecute dicho algoritmo. La meta principal de este trabajo de Tesis es aportar nuevas soluciones para la implementación física y ejecución de algoritmos de imágenes hiperespectrales a bordo de satélites. Este trabajo de investigación ayudará a mejorar los actuales resultados de las implemetaciones hardware existentes dentro del estado del arte, estableciendo además directrices para los futuros trabajos de investigación en este ámbito. A continuación se enumeran los objetivos de esta Tesis:

- Caracterizar los algoritmos de compresión de imágenes hiperespectrales.
- Proponer soluciones para acelerar dichos algoritmos en distintas tecnologías hardware, en concreto GPUs y FPGAs.
- Comparar las soluciones en ambas tecnologías en cuanto a su rendimiento, coste de la solución, flexibilidad de la implementación y consumo de potencia.
- Validar las soluciones propuestas, mostrando el correcto funcionamiento de las implementaciones resultantes.

Para la consecución de estos objetivos se realiza la implementación de dos algoritmos, diseñados específicamente para la compresión de imágenes hiperespectrales a bordo de satélites, en dos tecnologías hardware: FPGAs y GPUs. Ambas tecnologías son capaces de acelerar algoritmos que permiten el procesado de datos en paralelo, pero presentan múltiples diferencias en cuanto a sus metodologías de diseño, consumo de potencia y tolerancia a la radiación solar. En concreto, se diseña la implementación en GPU y FPGA

de un algoritmo de compresión sin pérdidas de imágenes hiperespectrales, lo que permite hacer una comparativa en términos de rendimiento entre ambas alternativas tecnológicas. Finalmente, se realiza la implementación del algoritmo estándar, el CCSDS para compresión sin pérdidas de imágenes hiperespectrales, en una FPGA cualificada para el espacio. Este trabajo sirve para evaluar la viabilidad de la utilización del estándar CCSDS 123 en misiones espaciales actuales, y es a su vez, sirve como ejemplo de caso de uso. Se realizará una comparativa entre los dos algoritmos estudiados y las tecnologías hardware consideradas, así como con otras soluciones del estado del arte.

A.3 Compresión con pérdidas en GPU y FPGA

Con el objetivo de estudiar las posibles implementaciones hardware para la compresión de imágenes hiperespectrales, a bordo de satélites, se proponen en este trabajo diferentes soluciones. En concreto, se realiza la implementación del algoritmo LCE en una GPU y una FPGA.

El algoritmo LCE es particularmente relevante, ya que fue diseñado específicamente para ser ejecutado a bordo de satélites. Por lo tanto, cumple con determinados requisitos que son cruciales para poder realizar la compresión a bordo de un satélite, como son: baja complejidad, alta eficiencia, tolerancia a fallos y sencillez de cara a la implementación en dispositivos hardware. El algoritmo comprime la imagen por bloques independientes de 16×16 píxeles con todas sus bandas. Consta de una primera etapa de predicción en la que se obtienen los errores de predicción, los cuales se codifican mediante un codificador entrópico. Las palabras de código obtenidas se empaquetan en un fichero que representa la imagen comprimida. La técnica de compresión por bloques hace que el LCE sea tolerante a fallos, ya que un error en la transmisión de un bloque no impide la decodificación del resto de

bloques, una vez la imagen sea recibida. La división de la imagen en bloques se hace como se muestra en la Figura A.1.

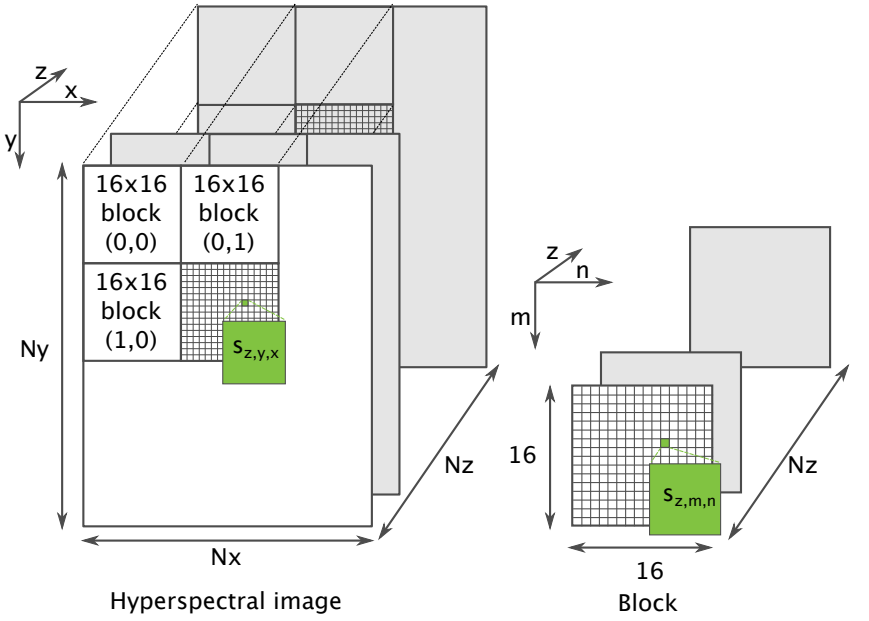


FIGURE A.1: División de la imagen hiperespectral en bloques independientes

Al inicio de este trabajo, se contaba con una implementación software del algoritmo LCE descrito en lenguaje de programación C, que es posible ejecutar en cualquier PC o estación de trabajo. Este software sirve como base para desarrollar tanto la implementació en GPU como la implementación en FPGA, y además se utiliza como referencia para validar los resultados obtenidos con ambas tecnologías.

A.3.1 Implementación del algoritmo LCE en una GPU

Se realiza una evaluación de la viabilidad de las GPUs para ejecutar la compresión de imágenes hiperespectrales, a través de una paralelización y ejecución del algoritmo LCE en una GPU de Nvidia utilizando CUDA . Durante la paralelización, se adapta el algoritmo a las abstracciones de CUDA, diseñando un kernel paralelo para la ejecución de cada una de las etapas del algoritmo. Esto implica aplicar diferentes estrategias para identificar qué datos pueden procesarse independientemente, y qué dependencias de datos es posible eliminar. Además, se utilizan estrategias para la redución de iteraciones en los bucles en que las dependencias de datos son inevitables. Todo esto se realiza teniendo en cuenta la arquitectura hardware de la GPU seleccionada, de modo que pueda maximizarse la utilización de los multiprocesadores presentes en ella.

La implementación diseñada muestra una alta aceleración, hasta 15.41 veces más rápida que la ejecución del mismo algoritmo en una CPU. El rendimiento, definido como el número de muestras procesadas por unidad de tiempo, es igualmente muy alto. El algoritmo LCE es capaz de comprimir entre 100 y 140 muestras cada segundo cuando se ejecuta en una GPU.

Las dificultades y cuellos de botella más importantes encontrados a la hora de realizar la paralelización del LCE se han identificado como parte de este trabajo. Destaca principalmente la gran cantidad de tiempo empleado en el envío de la imagen hiperespectral sin comprimir a la GPU para que pueda ser procesada. Se proponen líneas de trabajo futuras para superar este problema y optimizar el rendimiento de la implementación.

Teniendo en cuenta las similitudes entre el compresor LCE y el correspondiente decompresor, se considera conveniente relizar también una implementación en GPU de dicho decompresor. La paralelización del decompresor

supone dificultades adicionales, que hacen que el rendimiento obtenido por la GPU sea más bajo que el que se obtuvo para el compresor. En concreto, el principal problema con que se encuentra el decompresor es el formato de la imagen comprimida, que impide la identificación de los elementos independientes que es necesario extraer para poder realizar las tareas de decompresión en paralelo. En este trabajo de Tesis se propone una solución parcial a este problema, que consiste en añadir una cabecera al archivo comprimido con información relativa a la ubicación de los elementos independientes en el archivo comprimido. La inclusión de esta cabecera permite procesar más datos en paralelo, pero a su vez aumenta el tamaño del archivo comprimido, reduciendo por tanto el ratio de compresión. Se estudian distintos formatos de cabecera, eligiendo aquel que permite obtener la mejor relación entre la cantidad de elementos que se descomprimen en paralelo y el tamaño de la cabecera.

Como ya se ha comentado anteriormente, hoy en día no es posible utilizar las GPU para la compresión a bordo de satélites, ya que no son tolerantes a la radiación y presentan consumos de potencia demasiado elevados. Sin embargo, este trabajo de investigación es útil pra demonstrar que es posible obtener implementaciones eficientes y flexibles cuando se utilizan las GPUs para la compresión de datos de imágenes hiperspectrales, y contribuye a justificar y motivar la investigación en este campo, de manera que se hagan esfuerzos para desarrollar GPUs cualificadas para el espacio en un futuro.

A.3.2 Implementación del algoritmo LCE en una FPGA

Se realiza además una implementación del algoritmo LCE para compresión de imágenes hiperespectrales con pérdidas a bordo de satélites sobre una FPGA. Para realizar dicha implementación se utiliza la herramienta de

síntesis automática CatapultC, que genera los ficheros de descripción hardware a partir de código escrito en lenguaje de programación C. Esto hace posible obtener implementaciones en un tiempo reducido.

La metodología de trabajo en CatapultC consta de varias etapas. La primera de ellas es la optimización y verificación del código en lenguaje C. Seguidamente se establecen las restricciones globales del hardware, indicando la frecuencia de reloj, el comportamiento de la señal de reset y la tecnología en que se va a realizar la implementación final. Se debe indicar en este paso qué función dentro del código C representa el bloque más alto de la jerarquía. A continuación se establecen restricciones concretas de la arquitectura, realizándose, entre otras tareas, la optimización de bucles, que pueden ser desenrrollados (unrolling) o segmentados (pipeline). Finalmente, CatapultC realiza la planificación temporal del diseño y genera la descripción en VHDL del código. Ésta es válidada antes de ser ser introducida en herramientras de síntesis, como Mentor Graphis Precision o Synplify. Estas herramientas generan los archivos necesarios para programar la FPGA con el diseño final.

Las modificaciones realizadas al código C original del compresor LCE fueron tanto sintácticas como funcionales. Las modificaciones sintácticas se centran en eliminar aquellos elementos del lenguaje C no reconocibles por la herramienta CatapultC, ya que no son realizables en hardware; como puede ser el alojamiento de memoria de forma dinámica. También se realizan modificaciones funcionales, cambiando el algoritmo para evitar realizar computaciones que en hardware requieren un alto consumo de recursos, como puede ser una división de dos variables enteras. Una vez realizadas las modificaciones en el código C, se procede a optimizar la configuración en la herramienta CatapultC para obtener el resultado deseado en términos de área y latencia.

El módulo implementado realiza la compresión de un bloque de 16×16 muestras. Sus entradas y salidas se representan en la Figura A.2. Además

TABLE A.1: Ocupación en la FPGA del LCE

Virtex 5VFX130		**Modular approach**		**Non-modular approach**	
Resources	***Available***	***Used***	%	***Used***	%
DSP48Es	320	17	5	25	8
Number of RAMB18X2s	298	4	1	4	1
Number of slices	20480	2951	14	1935	10
Number of Slice Registers	81920	4208	5	5995	7
Number of Slice LUTS	81920	7836	9	7738	10

de la implementación de este módulo con CatapultC, se realiza una implementación modular del mismo algoritmo LCE, generando la descripción VHDL de los distintos bloques funcionales con CatapultC y diseñando el control y flujo de datos entre los módulos manualmente con el objetivo de mejorar el rendimiento del diseño.

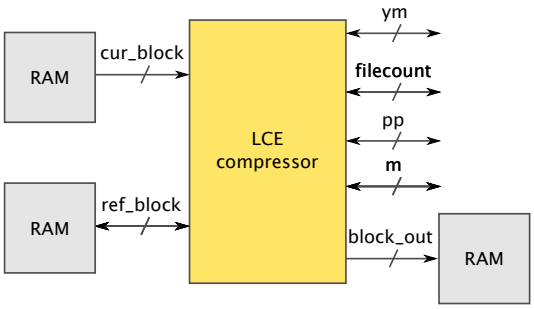


FIGURE A.2: Módulo de compresión y sus interfaces de entrada/salida

La ocupación de recursos, frecuencia máxima y rendimiento de la implementación del LCE en la FPGA, utilizado las mencionadas estrategias, se muestran en las Tablas A.1 y A.2

Table A.2: Muestras comprimidas por segundo de la implementación sobre FPGA del LCE

	Virtex 5VFX130	
	Diseño modular	**Diseño no modular**
Max.Frecuencia (MHz)	86	80
Mega-Muestras/seg	27.7	16.7

Los resultados demuestran la validez del algoritmo LCE para compresión a bordo de satélites, y prueban que es posible implementarlo con una baja utilización de recursos y baja potencia en una FPGA. Asimismo, puede observarse que, la implementación modular ha conseguido mejorar el rendimiento del diseño, manteniendo resultados similares de ocupación de la FPGA.

Se realiza finalmente una comparativa de las implementaciones realizadas del compresor LCE en términos de rendmiento (número de muestras procesadas por unidad de tiempo). Los resultados correspondientes a la compresión de una imagen del sensor AVIRIS de 220 bandas mediante el algoritmo LCE sobre CPU, GPU y FPGA se muestran en la Figura A.3. En la citada figura se muestran el número de muestras procesadas por segundo en función del tamaño espacial de la imágen (número de líneas por número de columnas). Aunque la GPU es capaz de procesar mayor número de muestras por unidad de tiempo, actualmente no es posible utilizarlas en el espacio y, por lo tanto, de momento no pueden considerarse una alternativa viable a las FPGA. Además, es posible acelerar aún más la implementación presentada en la FPGA, realizando la implementación de más de un módulo compresor en el mismo dispositivo, y haciendo que trabajen en paralelo. De este modo, podría conseguirse un rendimiento comparable al obtenido en la GPU, mientras que el consumo de potencia sería notablemente inferior en la FPGA.

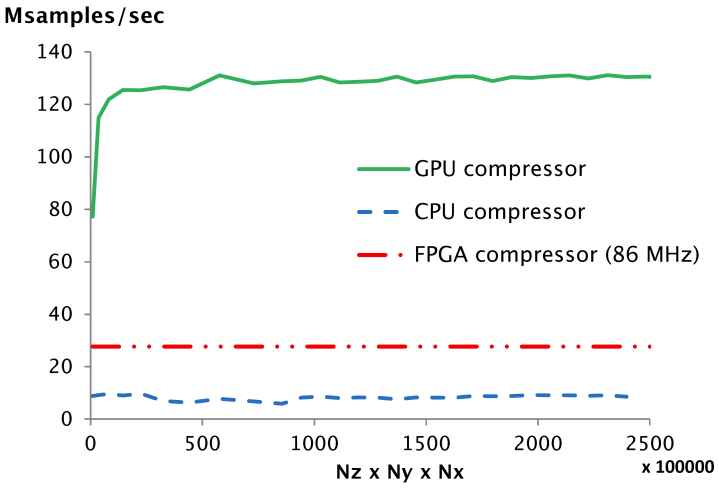


FIGURE A.3: Comparasión del rendimiento de la implementación del LCE en CPU, GPU and FPGA cuando se comprime una imagen de AVIRIS.

A.4 Compresión sin pérdidas en FPGA

Recientemente, el CCSDS, que representa las más importantes agencias espaciales del mundo, ha publicado un estándar para la compresión de imágenes hiperespectrales sin pérdidas, conocido como
CCSDS 123 [25]. Este algoritmo consiste en una etapa de predición, seguida de un codificador entrópico. La predicción se hace de forma adaptativa teniendo en cuenta los valores de las muestras vecinas en la banda actual y en las bandas anteriores. Los errores de predicción se codifican posteriormente con un codificador entrópico.

El CCSDS 123 había sido implementado, previamente a este trabajo de Tesis, para su ejecución en una CPU [74, 76] y GPU [81]. Sin embargo, ninguna de estas implementaciones puede utilizarse a bordo de un satélite. Se presenta a continuación una implementación del CCSDS sobre una FPGA cualificada para el espacio, que presenta además una baja ocupación de los recursos

hardware. La arquitectura del compresor se diseña teniendo en cuenta el impacto de los diferentes parámetros de configuración del CCSDS 123 en la eficiencia de la compresión y en la complejiidad de la implementación resultante.

A.4.1 Algoritmo CCSDS 123

El algoritmo CCSDS 123 realiza la compresión sin pérdidas de imágenes multiespectrales e hiperespectrales. Utiliza un esquema basado en la predicción seguida de un codificador entrópico.

Dentro de cada banda espectral, el predictor computa una suma local de las muestras de valores vecinos. Cada suma local es empleada para calcular la diferencia local. Para el cómputo de la suma local el usuario puede elegir una configuración orientada a columnas u orientada a vecinos. Para la diferencia local de cada banda espectral, el usuario puede escoger entre el modo reducido y el modo completo. El valor de la muestra predicha se calcula empleando la suma local de la banda espectral actual y una suma ponderada de la diferencia local de la banda actual y las P bandas espectrales anteriores. Los pesos empleados son actualizados de manera adaptativa. Cada residuo predicho, es decir, la diferencia entre una muestra dada y su correspondiente valor predicho se mapea en un entero sin signo que posteriormente se codifica mediante un código de Golomb.

El número de bandas previas usadas para la predicción puede tener valores entre 0 y 15. Este parámetro, junto con el tipo de suma local, el modo de cálculo de las diferencias locales y el número de bits utilizado para representar los pesos, representan los parámetros que más impacto tienen en el ratio de compresión obtenido y la complejidad de la implementación hardware del algortimo CCSDS 123.

A.4.2 Implementación del CCSDS 123 sobre una FPGA cualificada para el espacio

Se realiza la implementación del algortimo CCSDS 123 sobre una FPGA cualificada para el espacio siguiendo la siguiente metodología. En primer lugar, se realiza un estudio de cómo los parámetros de configuración del CCSDS 123 afectan al ratio de compresión. Seguidamente, se identifica cuáles de esos parámetros tienen un impacto en la complejidad de la implementación hardware. Como resultado, se observa que los parámetros más relevantes son: el número de bandas previas utilizadas para la predicción P, que puede variar entr 0 y 15; el tipo de suma local, orientada a columnas u orientada a vecinos; el modo de cálculo de las diferencias locales, reducido o completo; y el número de bits utilizado para representar los pesos. Una observación importante en este punto es que, si bien aumentar número de bandas previas utilizadas en la predicción favorece la compresión, ésta no mejora significativamente cuando el número de bandas utilizadas es superior a tres ($P = 3$).

Una vez identificados estos parámetros se proponen cinco opciones arquitecturales distintas, que se comparan en función de su complejidad, requisitos de memoria y número de accesos a la memoria externa. Se selecciona finalmente la opción arquitectural que permite obtener el mejor compromiso entre flexibilidad para variar la configuración del algortimo, y menor complejidad de implementación. Se realiza la descripción de la arquitectura elegida en código VHDL, creándose así un módulo IP al que se denomina HyLoC. HyLoC se implementa en una FPGA cualificada para el espacio, en concreto la RTAX1000S de Microsemi. Los resultados de la implementación para distintas configuraciones del compresor en cuanto a la ocupación de recursos y el rendimiento se resumen en las Tablas A.3 y A.4

TABLE A.3: Resultados de la síntesis de HyLoC en una RTAX1000S para distintas configuraciones

	Número de bandas utilizadas para la predicción $P = 3$			
	Predicción reducida		**Predicción completa**	
	Vecinos	**Columnas**	**Vecinos**	**Columnas**
Combinational cells	4354 (36%)	4166 (34%)	3971 (34%)	3713 (34%)
Sequential cells	1490 (36%)	1485 (34%)	1344 (34%)	1233 (34%)
I/O cells	75	75	75	75
Max. Frecuencia	43.4 MHz	43.9 MHz	43.0 MHz	43.4 MHz

TABLE A.4: Rendimiento de las distintas configuraciones de HyLoC

	Número de bandas utilizadas para la predicción $P = 3$			
	Predicción reducida		**Predicción completa**	
	Vecinos	**Columnas**	**Vecinos**	**Columnas**
Rendimiento (Mbits)	69	70	57	58

Puede observarse que se ha conseguido el objetivo de baja complejidad, con una ocupación hardware en torno al 34% para una configuración típica del compresor. La implementación presentada tiene además un muy bajo consumo de potencia, que se estima en torno a 93 mW. Además, el estudio arquitectural presentado como parte de este trabajo puede ser útil como base para futuros desarrollos y mejoras de HyLoC, o para adaptarlo a requisitos específicos de futuras misiones espaciales.

A.5 Conclusiones

Tras la finalización del trabajo previamente descrito, se han estudiado diferentes opciones tecnológicas para la implementación de algortimos de compresión de imágenes hiperespectrales con pérdidas y sin pérdidas. En concreto, se consideran las GPUs y FPGAs como dispositivos hardware para

la implementación de los algoritmos LCE y el estándar CCSDS 123. Los resultados obtenidos se comparan también con implementaciones similares que han sido publicadas en la literatura científica reciente.

Es posible concluir que los dos algoritmos estudiados, LCE y CCSDS 123 son buenos candidatos para la realización de compresión de imágenes hiperespectrales a bordo de satélites. La ocupación de recursos hardware cuando se implementan sobre FPGAs es muy reducida, similar o inferior a las encontradas para otros algoritmos comparables del estado del arte.

En cuanto al rendimiento, expresado como número de muestras comprimidas por unidad de tiempo, se observa que tanto las GPUs como las FPGAs mejoran el rendimiento de las implementaciones en procesadores de propósito general, CPUs, demonstrándose así que las tecnologías que permiten la paralelización de datos son más idóneas para la implementación de los algoritmos de compresión de imágenes hiperespectrales.

Las GPUs ofrecen de rendimiento significativamente superiores, sin embargo, su elevado consumo de potencia junto con el hecho de que no son tolerantes a la radiación solar, hacen que no puedan ser utilizadas hoy en día en misiones espaciales. Por otro lado, las FPGAs ofrecen la posibilidad de escalar y replicar los módulos de compresión para obtener un mayor rendimiento, lo que las hace potencialmente igual de eficientes que la GPUs, con un consumo de potencia significativamente inferior. El consumo de potencia de las implementaciones en FPGA presentadas en este trabajo es siempre inferior a 3 W, mientras en que las especificaciones de la GPU utilizada se estima su consumo en 225 W.

Appendix B

Publications

B.1 Journals

[1] Santos, L., Lopez, S., Callico, G. M., López, J. F., and Sarmiento, R. (2012). Performance Evaluation of the H.264/AVC Video Coding Standard for Lossy Hyperspectral Image Compression. *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of*, *5*(2), 451–461.

[2] Santos, L., Magli, E., Vitulli, R., López, J. F., and Sarmiento, R. (2013). Highly-Parallel GPU Architecture for Lossy Hyperspectral Image Compression. *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of*, *6*(2), 670–681.

[3] Santos, L., Magli, E., Vitulli, R., Núñez, A., López, J. F., and Sarmiento, R. (2013). Lossy hyperspectral image compression on a graphics processing unit: parallelization strategy and performance evaluation. *Journal of Applied Remote Sensing*, 7(1), 074599-074599.

B.2 International Conferences

[1] Santos, L., López, S., Callicó, G. M., López, J. F., and Sarmiento, R. (2011). Hyperspectral image compression with a H.264/AVC baseline encoder. In *Conference on Design of Circuits and Integrated Systems (DCIS 2011)*.

[2] Santos, L., López, S., Callicó, G. M., López, J. F., and Sarmiento, R. (2011). Lossy hyperspectral image compression with state-of-the-art video encoder. In *Proceedings of SPIE* (Vol. 8183, 81830).

[3] Santos, L., Vitulli, R., López, J. F., and Sarmiento, R. (2012). GPU implementarion of a lossy compression algorithm for hyperspectral images. In *IEEE Workshop on Hyperspectral Image and Signal Processing – Evolution in Remote Sensing (WHISPERS) 2012.*

[4] Santos, L., Vitulli, R., López, J. F., and Sarmiento, R. (2012). CUDA based GPU implemenmtation of a parallelized algorithm for lossy hyperspectral image compression. In *On Board Payload Data Compression Workshop – OBPDC 2012.*

[5] Santos, L., Vitulli, R., López, S., Marrero, G. M., López, J. F., and Sarmiento, R. (2012). Accelerating lossy hyperspectral image compression on a GPU. In *Conference on Design of Circuits and Integrated Systems (DCIS 2012).*

[6] Santos, L., Lopez, J. F., Sarmiento, R., and Vitulli, R. (2013). FPGA implementation of a lossy compression algorithm for hyperspectral images with a high-level synthesis tool. In *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on* (pp. 107–114).

[7] García, A., Santos, L., López, S., Callicó, G.M., López, J.F., Sarmiento, R. (2013) High level modular implementation of a lossy hyperspectral image compression algorithm on a FPGA. In *IEEE Workshop on Hyperspectral Image and Signal Processing Evolution in Remote Sensing (WHISPERS), 2013*

[8] García, A., Santos, L., López,, S., Callicó, G. M., Lopez, J. F., and Sarmiento, R. (2014). Efficient lossy compression implementations of hyperspectral images: tools, hardware platforms, and comparisons. In *SPIE Sensing Technology+ Applications* (pp. 912408-912408)

References

[1] National Aeronautics and Space Administration (NASA). Landsat science, 2014. URL http://landsat.gsfc.nasa.gov/.

[2] Centre National d'Etudes Spatiales (CNES). Spot earth observation mission. eyes in space., 2013. URL http://smsc.cnes.fr/SPOT/.

[3] Indian Space Research Organisation (ISRO). Indian remote sensing (irs) satellite system, 2008. URL http://www.isro.org/scripts/currentprogrammein.aspx#IRS.

[4] National Aeronautics and Space Administration (NASA). Airborne visible/infrared imaging spectometer (aviris), 2014. URL http://aviris.jpl.nasa.gov/.

[5] Alexander FH Goetz. Three decades of hyperspectral remote sensing of the earth: A personal view. *Remote Sensing of Environment*, 113: S5–S16, 2009.

[6] Peg Shippert. Introduction to hyperspectral image analysis. *Online Journal of Space Communication*, 3, 2003. URL http://spacejournal.ohio.edu/pdf/shippert.pdf.

[7] William F Belokon, Spectral Imagery Training Center, and Logicon Geodynamics. *Multispectral imagery reference guide*. Logicon Geodynamics, 1997.

[8] Centre National d'Etudes Spatiales (CNES). URL http://smsc.cnes.fr/PLEIADES/.

[9] European Space Agency (ESA). Proba missions, 2014. URL http://www.esa.int/Our_Activities/Technology/Proba_Missions.

[10] J Bioucas-Dias, Antonio Plaza, Gustavo Camps-Valls, Paul Scheunders, N Nasrabadi, and JOCELYN Chanussot. Hyperspectral remote sensing data analysis and future challenges. *Geoscience and Remote Sensing Magazine, IEEE*, 1(2):6–36, 2013.

[11] Antonio Plaza, Jon Atli Benediktsson, Joseph W Boardman, Jason Brazile, Lorenzo Bruzzone, Gustavo Camps-Valls, Jocelyn Chanussot, Mathieu Fauvel, Paolo Gamba, Anthony Gualtieri, et al. Recent advances in techniques for hyperspectral image processing. *Remote Sensing of Environment*, 113:S110–S122, 2009.

[12] J.M. Bioucas-Dias, A. Plaza, N. Dobigeon, M. Parente, Qian Du, P. Gader, and J. Chanussot. Hyperspectral unmixing overview: Geometrical, statistical, and sparse regression-based approaches. *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of*, 5(2):354–379, April 2012. ISSN 1939-1404. doi: 10.1109/JSTARS.2012.2194696.

[13] N. Keshava and J.F. Mustard. Spectral unmixing. *Signal Processing Magazine, IEEE*, 19(1):44–57, Jan 2002. ISSN 1053-5888. doi: 10.1109/79.974727.

[14] David A Landgrebe. *Signal Theory Methods in Multispectral Remote Sensing*. Wiley, Newark, NJ, 2005.

[15] G. Shaw and D. Manolakis. Signal processing for hyperspectral image exploitation. *Signal Processing Magazine, IEEE*, 19(1):12–16, Jan 2002. ISSN 1053-5888. doi: 10.1109/79.974715.

[16] Stefania Matteoli, Marco Diani, and Giovanni Corsini. A tutorial overview of anomaly detection in hyperspectral images. *Aerospace and Electronic Systems Magazine, IEEE*, 25(7):5–28, 2010.

[17] David WJ Stein, Scott G Beaven, Lawrence E Hoff, Edwin M Winter, Alan P Schaum, and Alan D Stocker. Anomaly detection from hyperspectral imagery. *Signal Processing Magazine, IEEE*, 19(1):58–69, 2002.

[18] Dimitris Manolakis and Gary Shaw. Detection algorithms for hyperspectral imaging applications. *Signal Processing Magazine, IEEE*, 19 (1):29–43, 2002.

[19] A. Plaza M. Ciznicki, K. Kurowski. Gpu implementation of jpeg2000 for hyperspectral image compression. In *SPIE - The International Society for Optical Engineering*, volume 8183, 2011.

[20] L. Santos, S. Lopez, G.M. Callico, J.F. Lopez, and R. Sarmiento. Performance evaluation of the h.264/avc video coding standard for lossy hyperspectral image compression. *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of*, 5(2):451 –461, april 2012. ISSN 1939-1404. doi: 10.1109/JSTARS.2011.2173906.

[21] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952. ISSN 0096-8390. doi: 10.1109/JRPROC.1952.273898.

[22] S. Golomb. Run-length encodings (corresp.). *Information Theory, IEEE Transactions on*, 12(3):399–401, Jul 1966. ISSN 0018-9448. doi: 10.1109/TIT.1966.1053907.

[23] R. Rice and J. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *Communication Technology, IEEE Transactions on*, 19(6):889–897, December 1971. ISSN 0018-9332. doi: 10.1109/TCOM.1971.1090789.

[24] *Lossless data compression recommended standard CCSDS 121.0-B-2.* The Consultative Committee for Space Data Systems, 2012.

[25] *Lossless multispectral and hyperspectral image compression recommmended standard CCSDS 123.0-B-1.* The Consultative Committee for Space Data Systems, 2011.

[26] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, January 2001. ISSN 1559-1662. doi: 10.1145/584091.584093. URL http://doi.acm.org/10.1145/584091.584093.

[27] W Schober, F Lansing, K Wilson, and E Webb. High data rate instrument study. *JPL publication*, pages 99–4, 1999.

[28] R. Trautner. Esa's roadmap for next generation payload data processors. In *Proceedings of DASIA 2011 Conference*, 2011.

[29] European Space Agency (ESA). Euclid mission, 2014. URL http://sci.esa.int/euclid/.

[30] A.M. Di Giorgio, S.J. Liu, G. Giusi, and G. Palamara. Euclid visible imager on-board lossless data compression: Performance assessment trade-off activities. In *Proceedings of 2012 ESA workshop on Onboard Payload Data Compression (OBPDC)*, 2012.

[31] C.M. Hartzell, L.C. Graham, T.S. Tao, H.R. Goldberg, J. Carpena-Nunez, D.M. Racek, C.E. Taylor, and C.D. Norton. Data system design for a hyperspectral imaging mission concept. In *Aerospace conference, 2009 IEEE*, pages 1–21, March 2009. doi: 10.1109/AERO.2009.4839507.

[32] N. Aranki, A. Bakhshi, D. Keymeulen, and M. Klimesh. Fast and adaptive lossless on-board hyperspectral data compression system for

space applications. In *Aerospace conference, 2009 IEEE*, pages 1–8, 2009. doi: 10.1109/AERO.2009.4839534.

[33] A. Abrardo, M. Barni, and E. Magli. Low-complexity predictive lossy compression of hyperspectral and ultraspectral images. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 797 –800, may 2011. doi: 10.1109/ICASSP.2011.5946524.

[34] *Image data compression recommended standard CCSDS 122.0-B-1.* The Consultative Committee for Space Data Systems, 2005.

[35] M.J. Weinberger, G. Seroussi, and G. Sapiro. The loco-i lossless image compression algorithm: principles and standardization into jpeg-ls. *Image Processing, IEEE Transactions on*, 9(8):1309–1324, 2000. ISSN 1057-7149. doi: 10.1109/83.855427.

[36] Xiaolin Wu and N. Memon. Context-based, adaptive, lossless image coding. *Communications, IEEE Transactions on*, 45(4):437–444, 1997. ISSN 0090-6778. doi: 10.1109/26.585919.

[37] S. Hunt and L.S. Rodriguez. Fast piecewise linear predictors for lossless compression of hyperspectral imagery. In *Geoscience and Remote Sensing Symposium, 2004. IGARSS '04. Proceedings. 2004 IEEE International*, volume 1, pages –312, 2004. doi: 10.1109/IGARSS.2004.1369023.

[38] Xiaolin Wu and N. Memon. Context-based lossless interband compression-extending calic. *Image Processing, IEEE Transactions on*, 9(6):994–1001, 2000. ISSN 1057-7149. doi: 10.1109/83.846242.

[39] B. Penna, T. Tillo, E. Magli, and G. Olmo. Transform coding techniques for lossy hyperspectral data compression. *Geoscience and Remote Sensing, IEEE Transactions on*, 45(5):1408 –1421, may 2007. ISSN 0196-2892. doi: 10.1109/TGRS.2007.894565.

[40] Bruno Aiazzi, Luciano Alparone, and Stefano Baronti. Quality issues for compression of hyperspectral imagery through spectrally adaptive dpcm. In Bormin Huang, editor, *Satellite Data Compression*, pages 115–147. Springer New York, 2011. ISBN 978-1-4614-1182-6. doi: 10.1007/978-1-4614-1183-3_6. URL http://dx.doi.org/10.1007/978-1-4614-1183-3_6.

[41] Chulhee Lee, Sangwook Lee, and Jonghwa Lee. Effects of lossy compression on hyperspectral classification. In Bormin Huang, editor, *Satellite Data Compression*, pages 269–285. Springer New York, 2011. ISBN 978-1-4614-1182-6. doi: 10.1007/978-1-4614-1183-3_13. URL http://dx.doi.org/10.1007/978-1-4614-1183-3_13.

[42] F. Garcia-Vilchez, J. Muñoz-Mari, M. Zortea, I. Blanes, V. Gonzalez-Ruiz, G. Camps-Valls, A. Plaza, and J. Serra-Sagrista. On the impact of lossy compression on hyperspectral image classification and unmixing. *Geoscience and Remote Sensing Letters, IEEE*, 8(2):253–257, 2011. ISSN 1545-598X. doi: 10.1109/LGRS.2010.2062484.

[43] Q. Du, N. Ly, and J.E. Fowler. An operational approach to pca+jpeg2000 compression of hyperspectral imagery. *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of*, PP(99):1–9, 2013. ISSN 1939-1404. doi: 10.1109/JSTARS.2013.2274527.

[44] Jarno Mielikainen. Lookup-table based hyperspectral data compression. In Bormin Huang, editor, *Satellite Data Compression*, pages 169–184. Springer New York, 2011. ISBN 978-1-4614-1182-6. doi: 10.1007/978-1-4614-1183-3_8. URL http://dx.doi.org/10.1007/978-1-4614-1183-3_8.

[45] D.S. Taubman and M.W. Marcellin. *JPEG2000: Image Compression Fundamentals, Standards, and Practice*. Kluwer, 2001.

[46] L. Chang, Ching-Min Cheng, and Ting-Chung Chen. An efficient adaptive klt for multispectral image compression. In *Image Analysis and Interpretation, 2000. Proceedings. 4th IEEE Southwest Symposium*, pages 252–255, 2000. doi: 10.1109/IAI.2000.839610.

[47] Pengwei Hao and Q. Shi. Reversible integer klt for progressive-to-lossless compression of multiple component images. In *Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on*, volume 1, pages I–633–6 vol.1, 2003. doi: 10.1109/ICIP.2003.1247041.

[48] B. Penna, T. Tillo, E. Magli, and G. Olmo. Progressive 3-d coding of hyperspectral images based on jpeg 2000. *Geoscience and Remote Sensing Letters, IEEE*, 3(1):125–129, 2006. ISSN 1545-598X. doi: 10.1109/LGRS.2005.859942.

[49] N.R.M. Noor and T. Vladimirova. Parallelised fault-tolerant integer klt implementation for lossless hyperspectral image compression on board satellites. In *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, pages 115–122, 2013. doi: 10.1109/AHS.2013.6604234.

[50] Qian Du and J.E. Fowler. Hyperspectral image compression using jpeg2000 and principal component analysis. *Geoscience and Remote Sensing Letters, IEEE*, 4(2):201 –205, april 2007. ISSN 1545-598X. doi: 10.1109/LGRS.2006.888109.

[51] Wei Zhu, Qian Du, and JamesE. Fowler. Hyperspectral image compression using segmented principal component analysis. In Bormin Huang, editor, *Satellite Data Compression*, pages 233–251. Springer New York, 2011. ISBN 978-1-4614-1182-6. doi: 10.1007/978-1-4614-1183-3_11. URL http://dx.doi.org/10.1007/978-1-4614-1183-3_11.

[52] A. Karami, M. Yazdi, and G. Mercier. Compression of hyperspectral images using discerete wavelet transform and tucker decomposition. *Selected Topics in Applied Earth Observations and Remote Sensing,*

IEEE Journal of, 5(2):444–450, 2012. ISSN 1939-1404. doi: 10.1109/JSTARS.2012.2189200.

[53] Kiely A. and Klimesh M. The icer progressive wavelet image compressor. In *Interplanetary Network Progress Report 42-155*, pages 1–46, 2003.

[54] Kiely A. and Klimesh M. Preliminary image compression results from the mars exploration rovers. In *Interplanetary Network Progress Report 42-156*, pages 1–8, 2004.

[55] M. Klimesh. Low-complexity lossless compression of hyperspectral imagery via adaptive filtering. *The Interplanetary Network Progress Report*, 2005.

[56] Andrea Abrardo, Mauro Barni, Andrea Bertoli, Raoul Grimoldi, Enrico Magli, and Raffaele Vitulli. Low-complexity approaches for lossless and near-lossless hyperspectral image compression. In Bormin Huang, editor, *Satellite Data Compression*, pages 47–65. Springer New York, 2011. ISBN 978-1-4614-1182-6. doi: 10.1007/978-1-4614-1183-3_3. URL http://dx.doi.org/10.1007/978-1-4614-1183-3_3.

[57] J. Teuhola. A compression method for clustered bit-vectors. *Information Processing Letters*, 7(6):308–311, 1978. URL http://www.scopus.com/inward/record.url?eid=2-s2.0-0018020596&partnerID=40&md5=8cc387e5be31644a5a6b9d049dee69f8.

[58] A. Gersho. Adaptive filtering with binary reinforcement. *Information Theory, IEEE Transactions on*, 30(2):191–199, 1984. ISSN 0018-9448. doi: 10.1109/TIT.1984.1056890.

[59] E. Magli, G. Olmo, and E. Quacchio. Optimized onboard lossless and near-lossless compression of hyperspectral data using calic. *Geoscience*

and Remote Sensing Letters, IEEE, 1(1):21–25, 2004. ISSN 1545-598X. doi: 10.1109/LGRS.2003.822312.

[60] J. Mielikainen and P. Toivanen. Clustered dpcm for the lossless compression of hyperspectral images. *Geoscience and Remote Sensing, IEEE Transactions on*, 41(12):2943–2946, 2003. ISSN 0196-2892. doi: 10.1109/TGRS.2003.820885.

[61] F. Rizzo, B. Carpentieri, G. Motta, and J.A. Storer. Low-complexity lossless compression of hyperspectral imagery via linear prediction. *Signal Processing Letters, IEEE*, 12(2):138–141, 2005. ISSN 1070-9908. doi: 10.1109/LSP.2004.840907(410)12.

[62] S.R. Tate. Band ordering in lossless compression of multispectral images. In *Data Compression Conference, 1994. DCC '94. Proceedings*, pages 311–320, 1994. doi: 10.1109/DCC.1994.305939.

[63] P. Toivanen, O. Kubasova, and J. Mielikainen. Correlation-based band-ordering heuristic for lossless compression of hyperspectral sounder data. *Geoscience and Remote Sensing Letters, IEEE*, 2(1):50–54, 2005. ISSN 1545-598X. doi: 10.1109/LGRS.2004.838410.

[64] Jing Zhang and Guizhong Liu. An efficient reordering prediction-based lossless compression algorithm for hyperspectral images. *Geoscience and Remote Sensing Letters, IEEE*, 4(2):283–287, 2007. ISSN 1545-598X. doi: 10.1109/LGRS.2007.890546.

[65] M. Slyz and D. Zhang. A block-based inter-band lossless hyperspectral image compressor. In *Data Compression Conference, 2005. Proceedings. DCC 2005*, pages 427–436, 2005. doi: 10.1109/DCC.2005.1.

[66] Euroepan Space Agency. Robotic exploration of mars, 2013. URL http://exploration.esa.int/mars/.

[67] A. Abrardo, M. Barni, E. Magli, and F. Nencini. Error-resilient and low-complexity onboard lossless compression of hyperspectral images by means of distributed source coding. *Geoscience and Remote Sensing, IEEE Transactions on*, 48(4):1892–1904, 2010. ISSN 0196-2892. doi: 10.1109/TGRS.2009.2033470.

[68] E.J. Candes and M.B. Wakin. An introduction to compressive sampling. *Signal Processing Magazine, IEEE*, 25(2):21–30, 2008. ISSN 1053-5888. doi: 10.1109/MSP.2007.914731.

[69] G. Coluccia, S.K. Kuiteing, A. Abrardo, M. Barni, and E. Magli. Progressive compressed sensing and reconstruction of multidimensional signals using hybrid transform/prediction sparsity model. *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, 2(3): 340–352, 2012. ISSN 2156-3357. doi: 10.1109/JETCAS.2012.2214891.

[70] A. Abardo, Barni M., Carretti C.M., Magli E., Kuiteing Kamdem S., and Vitulli R. Compressed sensing techniques for hyperspectral image recovery. In *Proceedings of 2010 ESA workshop on Onboard Payload Data Compression (OBPDC)*, 2010.

[71] Magli E., Barni M., Barducci A., Guzzi D., and Pippi I. Technological issues in compressive sensing. In *Proceedings of 2012 ESA workshop on Onboard Payload Data Compression (OBPDC)*, 2012.

[72] Rice Robert F., Yeh Pen-Shu, and Miller Warner H. Algorithms for high speed universal noiseless coding. In *Proceedings of AIAA Computing in Aerospace, IX*, 1993. doi: 10.2514/6.1993-4541.

[73] A.G. Villafranca, S. Mignot, J. Portell, and E. Garcia-Berro. Hardware implementation of the fapec lossless data compressor for space. In *Adaptive Hardware and Systems (AHS), 2010 NASA/ESA Conference on*, pages 164–170, 2010. doi: 10.1109/AHS.2010.5546264.

[74] J.E. Sanchez, E. Auge, J. Santalo, I. Blanes, J. Serra-Sagrista, and A. Kiely. Review and implementation of the emerging ccsds recommended standard for multispectral and hyperspectral lossless image coding. In *Data Compression, Communications and Processing (CCP), 2011 First International Conference on*, pages 222–228, 2011. doi: 10.1109/CCP.2011.17.

[75] Carole Thiebaut and Roberto Camarero. Cnes studies for on-board compression of high-resolution satellite images. In Bormin Huang, editor, *Satellite Data Compression*, pages 29–46. Springer New York, 2011. ISBN 978-1-4614-1182-6. doi: 10.1007/978-1-4614-1183-3_2. URL http://dx.doi.org/10.1007/978-1-4614-1183-3_2.

[76] European Space Agency (ESA). Ccsds 123.0-b-1 multispectral and hyperspectral lossless data compression, 2013. URL http://www.esa.int/TEC/OBDP/SEM069KOXDG_2.html.

[77] European Space Agency (ESA). Whitedwarf data compression evaluation tool, 2013. URL http://www.esa.int/TEC/OBDP/SEM069KOXDG_2.html.

[78] European Space Agency (ESA). Esa onboard computing and data handling - microprocessors, 2012. URL http://www.esa.int/TEC/OBCDH/SEM1IZEURTG_0.html.

[79] Nvidia Corporation. Dspace project - new digital signal processor for space application, 2013. URL http://www.dspace-project.eu/.

[80] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink. How a single chip causes massive power bills gpusimpow: A gpgpu power simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 97–106, 2013. doi: 10.1109/ISPASS.2013.6557150.

[81] B. Hopson, K. Benkrid, D. Keymeulen, and N. Aranki. Real-time ccsds lossless adaptive hyperspectral image compression on parallel gpgpu amp; multicore processor systems. In *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, pages 107–114, 2012. doi: 10.1109/AHS.2012.6268637.

[82] LPGPU Consortium. Low-power gpu (lpgpu), 2013. URL http://lpgpu.org/.

[83] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012. ISSN 0929-5585. doi: 10.1007/s10617-012-9096-8. URL http://dx.doi.org/10.1007/s10617-012-9096-8.

[84] Guoxia Yu, Tanya Vladimirova, and Martin N. Sweeting. Image compression systems on board satellites. *Acta Astronautica*, 64 (9–10):988 – 1005, 2009. ISSN 0094-5765. doi: http://dx.doi.org/10.1016/j.actaastro.2008.12.006. URL http://www.sciencedirect.com/science/article/pii/S0094576508004062.

[85] Xilinx. Xilinx chips enable world's first "on-the-fly" reconfigurable satellite, 2003. URL http://www.xilinx.com/prs_rls/design_win/0317satellite.htm.

[86] J.L. Nunez-Yanez, Xiaolin Chen, N. Canagarajah, and Raffaele Vitulli. Statistical lossless compression of space imagery and general data in a reconfigurable architecture. In *Adaptive Hardware and Systems, 2008. AHS '08. NASA/ESA Conference on*, pages 172–177, 2008. doi: 10.1109/AHS.2008.9.

[87] B. Osterloh, H. Michalik, S.A. Habinc, and B. Fiethe. Dynamic partial reconfiguration in space applications. In *Adaptive Hardware and*

Systems, 2009. AHS 2009. NASA/ESA Conference on, pages 336–343, 2009. doi: 10.1109/AHS.2009.13.

[88] L. Li, B. Fiethe, H. Michalik, and O. Björn. Efficient implementation of the ccsds 122.0-b-1 standard on space-qualified fpgas. In *Proceedings of 2012 ESA workshop on Onboard Payload Data Compression (OBPDC)*, 2012.

[89] C. Egho, Tanya Vladimirova, and M.N. Sweeting. Acceleration of karhunen-loeve transform for system-on-chip platforms. In *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, pages 272–279, 2012. doi: 10.1109/AHS.2012.6268662.

[90] Kiely A., Klimesh M., Xie H., and Aranki N. Icer-3d: A progressive wavelet-based compressor for hyperspectral images. In *Interplanetary Network Progress Report 42-164*, pages 1–21, 2006. URL http://tmo.jpl.nasa.gov/progress_report/42-164/164A.pdf.

[91] G.J. Sullivan. On embedded scalar quantization. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, volume 4, pages iv–605 – iv–608 vol.4, may 2004. doi: 10.1109/ICASSP.2004.1326899.

[92] Khronos Group. The open standard for parallel programming of heterogeneous systems, 2014. URL http://www.khronos.org/opencl.

[93] Nvidia Corporation. Cuda technology, 2006. URL http://www.nvidia.com/CUDA.

[94] E. Kandrot J. Sanders. *CUDA by example*. Addison Wesley, 2011.

[95] Nvidia Corporation. Cuda documents, 2012. URL http://http://docs.nvidia.com/cuda/index.html.

[96] Nvidia Corporation. Nvidia's next generation cuda compute architecture fermi, 2009. URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[97] M. Harris. Optimizing cuda. *Supercomputing conference, Reno, NV*, 2007.

[98] M. Harris and M. Garland. Optimizing parallel prefix operations for the fermi architecture, 2011.

[99] A. Balevic. Parallel variable-length encoding on gpgpus. In *Proceedings of Euro-Par'09 Int. Conf. on Parallel Processing*, pages 26–35, 2009.

[100] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[101] *Catapult® C Synthesis User's and Reference Manual - University Version - Release 2010a.* Mentor Graphics Corporation, 2010.

[102] Sangchul Kim, Hyunjin Kim, Taeil Chung, and Jin-Gyeong Kim. Design of h.264 video encoder with c to rtl design tool. In *SoC Design Conference (ISOCC), 2012 International*, pages 171–174, 2012. doi: 10.1109/ISOCC.2012.6407067.

[103] T. Damak, I. Werda, N. Masmoudi, and S. Bilavarn. Fast prototyping h.264 deblocking filter using esl tools. In *Systems, Signals and Devices (SSD), 2011 8th International Multi-Conference on*, pages 1–4, 2011. doi: 10.1109/SSD.2011.5767375.

[104] J.E. Sanchez, Augé E., Kiely A., Blanes I., and J. Serra-Sagristà. Performance impact of parameter tuning on the ccsds-123 lossless multi- and hyperspectral image compression standard. In *Proceedings of 2012 ESA workshop on Onboard Payload Data Compression (OBPDC)*, 2012.

[105] Group on Interactive Coding of Images Universitat Autonoma de Barcelona. Emporda software (a ccsds-123 implementation), 2013. URL http://gici.uab.es/GiciWebPage/emporda.php.

[106] Eumetsat. Eumetsat polar system - second generation, 2014. URL http://www.eumetsat.int/website/home/Satellites/FutureSatellites/EUMETSATPolarSystemSecondGeneration/index.html.