UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Hyperspectral image compression techniques on reconfigurable hardware

Técnicas de compresión de imágenes hiperespectrales sobre hardware reconfigurable

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Daniel Báscones García

Directores

Carlos González Calvo
Daniel Mozos Muñoz

Madrid

Hyperspectral Image Compression Techniques on Reconfigurable Hardware

Técnicas de Compresión de Imágenes Hiperespectrales sobre Hardware Reconfigurable

TESIS DOCTORAL

Daniel Báscones García

Directores:
Carlos González Calvo
Daniel Mozos Muñoz

Facultad de Informática
Universidad Complutense de Madrid

Octubre de 2020

Hyperspectral Image Compression Techniques on Reconfigurable Hardware

Ph.D. Thesis Dissertation

Dissertation presented to obtain the degree of
Ph.D. in Computer Science by:
Daniel Báscones García

Supervised by:
Carlos González Calvo
Daniel Mozos Muñoz

Facultad de Informática
Universidad Complutense de Madrid

Madrid, October 2020

Técnicas de Compresión de Imágenes Hiperespectrales sobre Hardware Reconfigurable

Tesis Doctoral

Memoria presentada para obtener el grado de doctor en Ingeniería Informática por:
Daniel Báscones García

Dirigida por los profesores:
Carlos González Calvo
Daniel Mozos Muñoz

Facultad de Informática
Universidad Complutense de Madrid

Madrid, Octubre 2020

Financial Support. This work was funded partially by the by the MINECO projects TIN2013-40968-P and TIN2017-87237-P, by HIPEAC collaboration grants (2018 and 2019) at ECSPEC, and by the pre-doctoral UCM CT17/17-CT18/17 grant.

To those that will skim through this text, understanding next to nothing, but still feeling an aura of mystery. Those that will appreciate it not because of what it contains, but because of what it means.

To you.

Para aquellos que ojearán este texto entendiendo más bien poco, pero sintiendo el misterio que rodea a la ciencia. Aquellos que lo apreciarán no por lo que contiene, sino por lo que significa.

Para ti.

Acknowledgments

The end. That's how most books close their last page. They are self-contained with a beginning, some story-telling in the middle, and a clear and distinct end. When they do not end, it is because a second part is in the works. But this approach limits creativity. It limits imagination. Why does the whole world need be contained within the book cover and back?

And that's how most thesis are, aren't they. A culmination. The mental climax of years of effort that just... stops. Why does it have to be a farewell? Each day of someone's life does not have to be the end of anything. It is the beginning of the rest. The path might change, the objective might be different, but nothing is finished. Every milestone along the way is just polishing the one that came before.

That polishing compound that shapes our lives is people. From the moment someone is born, their life has already depended on others, has already been influenced by others. This thesis marks just a milestone and, as is tradition, all the polishers will be acknowledged without naming to avoid the shame from forgetting someone's name.

Personal computers were expanding like wildfire among the tech-savvy population. For all I cared, the computer at home was just a gaming machine. But not just any kind of gaming machine: a *serious* game one. The term might not have been as popular at that point in time as *educational* games, but that is what it was. It was still fun, but it was not only a fun experience but a *learning* experience. Even under that premise, playing time was limited to avoid the omnipresence of eye damage that today plagues society.

But once introduced to the world of "good" addictions, it is not possible to take it back. Games were ending. New programs had to be installed, The Internet had to be set up.

```
>ipconfig
```

I still remember that simple line written over a black canvas. Press enter, and a bunch of white mono-spaced characters cascaded through the screen conveying some mysterious information. Unknown information appeared on the screen. What that was is sometimes still today a mystery, but it was *cool*. There is something about it. The mystery and astonishment when looking at something that just *works*. Like a well-oiled bearing supporting the wheels powered by a V-8, like the blades of a turbine spinning in place when synchronized to the shutter speed, like a crane precisely expanding its arm to a previously unreachable place.

A transistor. It's got something magical, ain't it?. Even though you cannot *see* them, you can *feel* them as any other engineering marvel.

Years went by and transistors were forgotten. A football was much more appealing and frankly, much more easy to deal with than a keyboard and mouse. But the interest for things that just... work, was still there.

It was a robot that brought it back. Ironically, the ones that will eventually take our jobs, either relegating us to slavery or perpetual artistic freedom. A simple interface where boxes and

lines formed execution graphs that the robot would follow. From a line follower to a calculator or an automated shooting turret, it was blessed by Turing and could do *anything*.

But interest faded away after a while. Time was becoming less available, and video games more prevalent. But even fun games can teach us things. A predecessor to the widely-known Minecraft, Roblox was a game where Lego-like characters moved around user-created levels built with Lego blocks. As it turns out, blocks were *programmable*. On sandbox mode, a weird-looking screen, reminiscent of that of `ipconfig`, showed up when clicking on blocks. It said things, in a very structured language, that could be understood.

They say some brains are better structured for certain tasks. A simple line quickly became a game of seeing what effects changing the different words and values had on the world. Building a level was relatively easy. Just copy things that others have done[1] with the functionality that you need and chain them together. Maybe spice it up by changing it a little bit so it does not do the *exact same thing* as the source.

That was programming. That was the seed that started polishing what had been dormant for long.

First it was a simple calculator where the button color palette was more important than the button functionality. Then it moved to tools that helped in games, purely motivated by increasing the post count by one in the forums. That's when some people would say a revelation happened. A moment that would steer things one way forever. But that had been already brewing for long.

From there, the *official* path begun. Alongside a squad of the best people one could possible desire, supported by the best people one could possibly have behind, and directed by some of the best guides one could hope for when lost in a forest.

At first it was a simple pocket knife cutting through the tall grass, then a machete slashing lianas, an ax cutting off tree branches and finally a bulldozer uprooting whole trees. The power built after iterations of the educational system, topped off with the best real world experience one could ask for, gives people the freedom to do what they want.

And that was going back to basics, going back to that feeling of things that just... work. The feeling of information boxes sliding through a data factory filling up containers along the way. The feeling of doing something that can impress oneself.

This work is not the summary of the last 3 years, is the summary of a lifetime that hasn't concluded. The polished stone resulting from all those that have contributed, which could even get smoother. Through support, revisions, suggestions, corrections, ideas, jokes, entertainment and love. Because life is a tangled net of interactions, and like a hurricane, it wouldn't be fair not to thank the butterfly that started it.

[1] Sponsored by StackOverflow

Agradecimientos

Fin. Así cierran los libros su última página. Autocontenidos con un comienzo, nudo, y desenlace diferenciados. Y, si no terminan, es porque esperan una secuela que cierre los hilos abiertos. Una aproximación tradicional que limita la creatividad. La imaginación. ¿Por qué debe estar contenido un mundo limitado entre la portada y contraportada?

También está presente esta filosofía en las tesis. Son una culminación. El clímax mental de años de esfuerzo que simplemente... termina. ¿Por qué tiene que ser una despedida? Cada día, momento de la vida no tiene por qué marcar el final de nada. Es el comienzo del resto. El camino puede cambiar, incluso el destino, pero nada tiene por qué terminar. Los hitos del camino no se dejan atrás sino que se van puliendo con el acarreo.

Y lo pulen las personas. Desde el momento que nacemos la vida depende de otros, es influenciada por otros. Esta tesis marca solo uno de esos hitos y, siguiendo la tradición, todos los pulidores serán agradecidos sin nombrar, evitando la vergüenza de olvidar alguno.

Los ordenadores se expandían a velocidad de vértigo entre la gente aficionada a la tecnología. Para mí, el ordenador de casa era solo una máquina para jugar jueguecitos. Pero no juegos cualesquiera, juegos *serios*. El término probablemente no era tan popular entonces como sí lo era *educativo*. Juegos que mantenían la diversión, pero que además enseñaban. Pero incluso bajo esa premisa, el tiempo estaba limitado para evitar, o posponer, los omnipresentes problemas visuales de hoy en día en la sociedad.

Y una vez integrado en el mundo de las "buenas" adicciones, no es posible salir. Los juegos se acababan, se instalaban nuevos programas, e internet tenía que configurarse para seguir adelante.

```
>ipconfig
```

Aún recuerdo esa simple línea con un cursor parpadeando sobre un lienzo negro. Presionabas intro, y un montón de caracteres blancos monoespaciados caían como una cascada, portando algún tipo de información desconocida en el momento. Algunas de las palabras que aparecían son aún hoy un misterio, pero un misterio *molón*. Había, y hay, algo detrás. El misterio y perplejidad al mirar a algo que simplemente funciona. Como un rodamiento bien engrasado que dirige ruedas alimentadas por ocho cilindros, como las afiladas hojas de una turbina que giran estáticas al sincronizarse a la velocidad del obturador, como una grúa extendiendo su brazo hasta un lugar inaccesible.

Y un transistor. Tiene algo mágico. Incluso sin poder verlo, se puede sentir su poderío como con cualquier otra maravilla de la ingeniería.

Los años pasaron y los transistores olvidados. Un balón era mucho más atractivo y, siendo sinceros, más fácil de domar que un teclado y ratón. Pero el interés por las cosas que simplemente... funcionaban, aún estaba ahí.

Fue un robot quien lo trajo de vuelta. Irónicamente, esos seres que finalmente acabarán con nuestros trabajos, ya sea haciéndonos esclavos o permitiendo un estado perpetuo de libertad

artística. Una simple interfaz con cajitas y líneas conseguía formar grafos de ejecución que el robot seguía. Desde un simple seguidor de líneas a una calculadora o torreta de defensa de la habitación, el robot había sido bendecido por Turing y podía hacer *cualquier* cosa.

El interés volvió a desaparecer tras un tiempo. El tiempo se hace escaso, y los videojuegos más prevalentes. Pero incluso juegos divertidos pueden enseñar. Roblox, previo al conocido Minecraft, tenía unos personajes similares a los de lego que se movían por niveles creados por los usuarios. Y los bloques eran *programables*. Con una combinación especial de teclas, una extraña pantalla, que recordaba a aquella de `ipconfig`, se mostraba al hacer click en los bloques. Decía cosas que, en un lenguaje muy estructurado, se podían entender.

Dicen que determinados cerebros están estructurados para ciertas tareas. Una simple línea de código se convirtió rápidamente en observar qué efectos tenía en el mundo cambiar las palabras. Construir un nivel era sencillo, solo bastaba copiar lo que habían hecho otros[2] con las funcionalidades que interesaban para concatenarlas todas y crear algo a medida. Quizá cambiándolo un poco para que no hiciera exactamente lo mismo que la fuente.

Eso era programar, eso era la semilla que comenzó a pulir aquello que había estado latente tanto tiempo.

Primero fue una simple calculadora donde la paleta de colores era más importante que la funcionalidad de los botones. Posteriormente fueron herramientas que ayudaban en juegos, puramente motivadas por incrementar el contador de mensajes de los foros. Es este cúmulo de cosas lo que algunos llamarían revelación, pero era un brebaje que había estado fermentado ya mucho tiempo.

Desde ahí, comenzó el camino *oficial*. Junto un grupo de la mejor gente que uno pudiera desear, apoyado por la mejor gente que uno pudiera tener, y dirigido por los mejores guías que uno pudiera esperar al perderse en un bosque oscuro.

Al principio era una simple navaja de bolsillo cortando césped, luego un machete sajando lianas, un hacha podando ramas y finalmente una excavadora levantando árboles enteros. El poder adquirido tras iteraciones del sistema educativo, completado con la mejor experiencia en el mundo real que uno pudiera pedir, es lo que proporciona la libertad de elegir lo que uno quiere.

Y era volver a lo sencillo, a ese sentimiento de cosas que simplemente... funcionan. El sentimiento de cajas llenas de componentes que se mueven por una fábrica de datos llenando contenedores por el camino. El sentimiento de impresionarse.

Este trabajo no es el resumen de los últimos 3 años, sino el resumen de una vida que no ha terminado. La piedra pulida por todos aquellos que han contribuido, y que aún puede tornarse más brillante. Con apoyo, revisiones, sugerencias, correcciones, ideas, bromas, entretenimiento y amor. Porque la vida es una confusa red de interacciones y, como un huracán, no sería justo dejar sin agradecer a la mariposa que empezó todo.

[2] Patrocinado por StackOverflow

Abstract

Sensors are nowadays in all aspects of human life. When possible, sensors are used remotely. This is less intrusive, avoids interferces in the measuring process, and more convenient for the scientist. One of the most recurrent concerns in the last decades has been sustainability of the planet, and how the changes it is facing can be monitored. Remote sensing of the earth has seen an explosion in activity, with satellites now being launched on a weekly basis to perform remote analysis of the earth, and planes surveying vast areas for closer analysis.

One of the most interesting sensors aboard these platforms is the hyperspectral image sensor. It extends the concept of humanly visible images, instead capturing the intensity of the whole electromagnetic spectrum at evenly spaced intervals. Hundreds of samples are present per pixel, providing a much more detailed profile of the image contents. Analysis of these images is of great importance for all kinds of studies about geology, hydrology, agriculture, vegetation and many more. However, a problem arises when transmitting or storing the data.

Hyperspectral images are big. A single one can go in the GB range, limiting what on-board processing can achieve with them. Transmission to ground stations, or on-board storage is a necessity. To do so efficiently, scientists have been using a technology available since the inception of computers: compression. Memory and bandwidth have always been limiting factors for data processing, and reducing the data flow between and into drives helps mitigate this issue.

Hyperspectral compression algorithms have been designed as adaptations of existing techniques, or as newly developed algorithms specific to this kind of data. Their computing requirements are usually higher than those of other data compression algorithms due to the inherent complexity of hyperspectral data. Needing to be executed on air and spaceborne systems, another constraint comes in: power is limited (both electrically and computationally). Custom ASICs could be developed, but from inception to having an actual usable product, the technology is already obsolete in a fast-moving world, and too expensive for one-time uses.

Luckily, a powerful platform that is also power efficient, flexible, and fairly fast to develop for exists: FPGAs. Reprogrammable hardware that can internally change to mimic the behavior of a custom circuit with few penalties other than limited resources. Furthermore, they are able to reach real-time constraints where other technologies fall short, being able to sit as a part of a capture-compression-storage/transmission pipeline where the limit is now the sensor's throughput.

In this thesis, compression of hyperspectral images on FPGAs is studied. The different types of algorithms available, as well as their portability to FPGAs, and which and how can benefit more from this technologies. Also, from the algorithmic point of view, how optimizations prior to FPGA development can lead to extensive gains even before implementation, avoiding more costly optimizations late in the process.

First, the CCSDS 123.0-B-1 algorithm is implemented, a standard by the CCSDS. It is designed to work by processing raw sensor data, compressing the image in raster order. A design is made that is parametrized with every algorithm option, producing an optimized core with each synthesis. While already quite fast, parallelization techniques bring it to a performance

of over $300MS/s$ on a Virtex-7 for the processing core, and over $140MS/s$ on a space-qualified Virtex-4 board, well above the $30.72MS/s$ real-time threshold. These speeds even surpass GPU implementations while drawing less than $\approx 100\times$ the power.

Secondly, a custom lossy hyperspectral image compression algorithm is designed based on a PCA+JPEG2000 approach. Extensive tests are done to add more compression techniques to the compression pipeline, as well as optimizing parameter selection over the range of compression ratios to get the best quality possible. A very efficient algorithm at low bit-rates is achieved, however its complexity is too high for a full FPGA implementation. Real-time is not achieved on a general purpose processor, sitting at half the required performance. An FPGA accelerator is designed for the most time-consuming part: the JPEG2000 encoder. With over 70% of time devoted to it, a $\approx 100\times$ acceleration accomplishes the real-time constraints.

Finally, a near-lossless algorithm, sitting in between the first two, is also analyzed and an FPGA core developed for it. Software analysis shows it is competitive in both the lossy and lossless domain. And with a highly optimized pipeline, hardware results show that it reaches a single-core performance above that of the parallelized CCSDS core at $322.5MS/s$, while taking a fraction of the resources as the CCSDS core, with $< 10\%$ occupancy on a Virtex-5 board. And it can even be parallelized for further improvements.

It is demonstrated that real-time FPGA compression is possible for the lossless and near-lossless algorithms with room to spare in case sensors improve over time. Software analysis before implementing the algorithms proves to be a very useful tool in estimating final performance, and the selection of the appropriate algorithm is crucial in being able to quickly adapt it to an FPGA.

All implementations have been compared, and their advantages and disadvantages explained in detail. As it is nearly always the case, there is not a perfect algorithm or technique that solves all problems. While FPGAs consistently outperform other platforms, the specific core design has to be carefully crafted to get the maximum performance out of it according to the algorithm specifications.

In the following chapters, all of these concepts will be expanded upon. A more detailed introduction will be presented in Chapter 1, further explaining the concepts of Hyperspectral images, compression and FPGAs. Different compression techniques will be explained in Chapter 2 as an introduction for the hyperspectral compression algorithms. Al three selected algorithms will be presented in Chapter 3 from an algorithmic point of view, explaining their mathematical basis as well. Their implementations for FPGA devices will be explained after in Chapter 4, with emphasis in existing and improved techniques. Results from both the algorithmic and implementation points of view will be shown in Chapter 5 and put into perspective in Chapter 6. Conclusions will close this work in Chapter 7 with the most important findings.

Keywords: Hyperspectral image, compression, FPGA, real-time.

Resumen

Los sensores aparecen hoy en día en todos los aspectos de nuestra vida. Cuando es posible, de manera remota. Esto es menos intrusivo, evita interferencias en el proceso de medida, y además facilita el trabajo científico. Una de las preocupaciones recurrentes en las últimas décadas ha sido la sostenibilidad del planeta, y cómo monitorizar los cambios a los que se enfrenta. Los estudios remotos de la tierra han visto un gran crecimiento, con satélites lanzados semanalmente para analizar la superficie, y aviones sobrevolando grandes áreas para análisis más precisos.

Uno de los sensores que vuela en estas plataformas es el de imágenes hiperespectrales. Extiende el concepto de imágenes visibles por humanos, capturando la intensidad del espectro electromagnético completo en intervalos equiespaciados. Cientos de muestras se recogen por cada punto, dando mucha más información sobre los contenidos de la imagen. El análisis de estas imágenes es de gran importancia para multitud de estudios sobre geología, hidrología, agricultura, vegetación y muchos más. Pese a ello, surge un problema al tratar los datos.

Las imágenes hiperespectrales son grandes. Una sola puede llegar a ocupar varios GB, limitando lo que se puede hacer con procesamiento a bordo. Por tanto, gran capacidad de almacenamiento o transmisión son esenciales. Para hacerlo de manera eficiente, los científicos han utilizado una tecnología disponible desde la aparición de los primeros ordenadores: la compresión. La memoria y ancho de banda siempre han limitado el procesamiento de datos, y reducir el flujo de los mismos ayuda a reducir este problema.

Los algoritmos de compresión hiperespectral se han diseñado adaptando técnicas existentes, o desarrollando nuevas específicas para este tipo de datos. Sus requisitos de computación son algo superiores a otros debido a la inherente complejidad de los datos. Siendo necesaria la ejecución en plataformas en vuelo, aparece además la restricción del consumo eléctrico y computacional. Se podrían desarrollar ASICs a medida, pero desde la idea hasta tener un producto útil, la tecnología queda obsoleta en este rápido mundo, siendo muy cara además para usos puntuales.

Por suerte, existe una plataforma potente, eficiente energéticamente, flexible, y para la que el desarrollo es rápido: las FPGA. Hardware reprogramable que puede cambiarse para imitar el comportamiento de un circuito a medida con pocas restricciones. Son además capaces de conseguir rendimientos en tiempo real donde otras tecnologías se quedan cortas, siendo capaces de colocarse en la cadena de captura-compresión-almacenamiento/envío, donde ahora el límite es el ancho de banda del sensor.

En esta tesis, se ha analizado la compresión de imágenes hiperespectrales en FPGAs, los diferentes tipos de algoritmos disponibles, y también su portabilidad a FPGAs y cómo se benefician de la tecnología. Desde el punto de vista algorítmico, también se analiza cómo las optimizaciones previas al desarrollo en FPGA pueden acarrear grandes beneficios incluso antes de la implementación, ahorrando esfuerzos posteriores en el proceso.

En primer lugar se implementa un estándar de la CCSDS, el 123.0-B-1. Está diseñado para procesar los datos en crudo del sensor, comprimiendo la imagen en orden de escaneo. Se ha realizado un diseño parametrizado con todas las opciones del algoritmo, produciendo un núcleo optimizado con cada síntesis. Siendo ya suficientemente rápido, las técnicas de paralelización

lo llevan a rendimientos de $300MS/s$ en una Virtex-7, y de más de $140Ms/s$ en una placa Virtex-4 apta para satélites, muy por encima del umbral de $30.72MS/s$ de tiempo real. Estas implementaciones incluso superan las de GPUs, consumiendo aproximadamente cien veces menos energía.

En segundo lugar se ha diseñado un algoritmo con pérdida basado en una aproximación que incluye PCA y JPEG2000. Se ha realizado una gran batería de tests para probar diferentes técnicas de compresión concatenadas, optimizando también los parámetros para dar con un algoritmo altamente eficiente. Se consigue mejorar la literatura, pero una implementación entera en FPGA resulta demasiado compleja. Por tanto, se hace un análisis extenso de las diferentes partes para alcanzar el tiempo real, que se queda lejos en un procesador de propósito general. Finalmente, y acelerando el codificador de JPEG2000, que ocupa el 70% del tiempo, se consigue superar el tiempo real para este complejo algoritmo.

Finalmente se experimenta con un algoritmo de pérdida limitada que se sitúa entre los dos anteriores. Un exhaustivo análisis software muestra que es competitivo tanto entre los algoritmos sin pérdida con entre los algoritmos con pérdida. Se genera una segmentación altamente optimizada, con resultados mononúcleo que superan al CCSDS con $322.5MS/S$ en rendimiento, mientras que en recursos se consume una fracción de aquellos del CCSDS. En total, se consiguen ocupaciones de $< 10\%$ en una Virtex-5 apta para satélites, pudiendo además paralelizarse para mejorar aún más el rendimiento.

Con todo esto, se demuestra que la compresión en tiempo real es posible en FPGAs tanto para los algoritmos con y sin pérdida, con suficientes recursos de sobra en caso de que mejoren los sensores. El análisis software previo de los algoritmos queda demostrado como un paso clave para estimar el rendimiento final, siendo la selección del algoritmo apropiado un paso crucial en una correcta y rápida implementación para FPGA.

Todas las implementaciones han sido comparadas, con sus ventajas e inconvenientes explicados en detalle. Como suele ser costumbre, no hay una solución perfecta para todos los problemas. Mientras que las FPGAs son mejores que otras plataformas de manera consistente, el diseño concreto tiene que ser meticulosamente realizado para conseguir el mejor rendimiento siguiendo al dedillo las especificaciones.

En los siguientes capítulos, todos estos conceptos se tratan en mucho mayor detalle. Una introducción más extensa y detallada se presenta en el Capítulo 1, explicando los conceptos de imagen hiperespectral, compresión y FPGAs. Las diferentes técnicas existentes de compresión se explican en el Capítulo 2 como introducción a los algoritmos de compresión hiperespectral. Los tres algoritmos seleccionados son explicados en el Capítulo 3 desde el punto de vista algorítmico, explicando también la base matemática. Su implementación en FPGA es explicada a continuación en el Capítulo 4, con énfasis en las técnicas ya existentes en la literatura, y las mejoradas. Resultados, tanto desde el punto de vista algorítmico como de implementación, se muestran en el Capítulo 5, y se ponen en perspectiva en el Capítulo 6. Las conclusiones cierran la tesis en el Capítulo 7 con los hallazgos más importantes, y futuras ideas de continuación.

Palabras clave: Imagen hiperespectral, compresión, FPGA, tiempo real.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We have always been, and will always be, fascinated and intrigued by our surroundings. What once was a feeling of mystery and fear for the unknown is now a pleasure to dive in. Long gone are the times when we needed to worry about farming, hunting, foraging... in a nutshell, surviving. When time was a scarce resource that was invested for future generations and not ourselves.

Thanks to the efforts of those that came before us, and their motivation to leave a better world than they found, we were able to progressively be freed from responsibilities. The question became not what we need but what we want. The answer? To know everything.

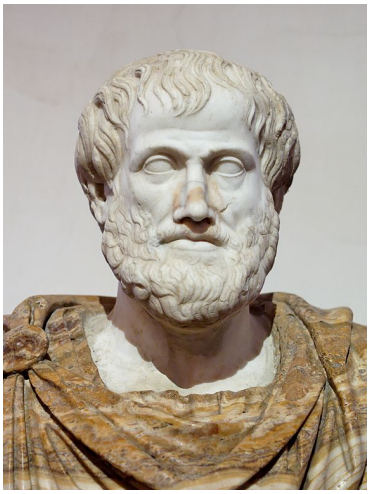

Figure 1.1: Aristotle was one of the first to help develop the scientific method aiming to answer our questions. [102].

Material needs have given way to existential questions and philosophy. We've transcended the physical world slowly rising towards a world where thoughts are as important, if not more, than just being alive.

But such a journey is not one that we start alone. Because the beauty of discovery is lost in the loneliness of solitude. We have the need to share our experiences. Not because we want others to be jealous, but because we want to be able to live experiences that our limited time doesn't allow us to.

Sharing is not easy. We are a mixture of emotions, irrational signals that interfere with the objectiveness of logic. Small might be enormous, what is fast might take eons to complete, and something bright might be a shadow of something brighter.

And that's why, as rational individuals, we establish the need for a unified system that allows us to share by comparing against a reference. From economical activities such as trading, engineering, architecture, landscaping to expressing what we have, or have seen, to others. The human error is still there, but now as an error of measure, not of expression.

The first units of measure defined distance. With our own bodies as a reference, we were able to measure small distances in inches (a thumb's width), medium lengths in feet (length of a foot as one would imagine), and long travels in miles (two thousand steps). Mass for example took reference in a single plant grain for small objects. Time was based on celestial body interaction, with earth's rotation marking days and earth's translation defining years.

These units allowed a way of objectively communicating things as long as the same reference was used. But being based in non-constant references, all of them meant different things depending on time and place. Distance could be halved depending of if a child or an adult was

measuring. A slight genetic variation in a plant could mean that a pound (7000 grains) would feed a family of 4 or 6. Far from the equator, four hour's work could mean forging a single sword in winter or three in the summer.

However, locally (both in time and space), those systems made sense and were convenient. It was when we had time to experiment and explore that units had to be standardized. Because if mountains on opposite sides of the globe needed to be measured, a consistent ruler was needed since both couldn't be seen at once. If a chemical reaction needed to be reproduced at two different points in time, the exact ingredients needed to be measured against a consistent weight. And if people were to travel across barren lands, supplies had to be taken to last a specific amount of time.

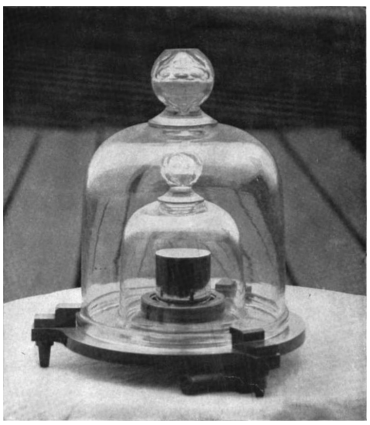

Figure 1.2: One of many standard kilogram prototypes, this one residing in the National Institute of Standards and Technology, USA. A platinum-iridium cylinder that differs just 0.0000075% in mass with the reference kilogram. [209].

Step by step, references were taken from unchangeable quantities that would remain consistent across time and space. The earth as a reference for distance was proposed in the XVII century. Water freezing and boiling points were common references for temperature that were established as standards in the XVIII century.

Units that could be defined based on others were lost for simplicity. Distance defined volume, which in turn defined mass. Velocity could be defined as a function of distance and time, and the magnetic field could be expressed as a function of distance, mass and time.

It was not until 1875 that the International Bureau of Weights and Measures was established. An international organization seeking to standardize units of measurement. Global prototypes (Figure 1.2) for different unit of measurement were made to distribute and reproduce across the world, with slight variations being acceptable in everyday use. Still, units still depended on those prototypes, without which an exact measurement was not possible.

Over time, increasing scientific knowledge drove change in what the reference point should be for certain magnitudes. Constants and laws were identified in the universe, and based on those, units were redefined. A meter was no longer 1/10000000 of a meridian but the distance light travels in vacuum in 1/299792458 of a second, which in turn is defined as 9192631770 oscillations of the element Cesium between two defined energy levels. Following the same procedure, all base units were redefined to be reproducible.

Meanwhile, curiosity was pushing scientists to make new discoveries. New lands, new elements, new materials, new techniques... We wanted to know what was in every single thing we could see or feel. Our eyes were always the best tool to analyze things. But how could something we see be measured? We could measure *what* we saw, not *how* we saw it. Our need to share the most visceral feelings and structured analysis was stuck in our inability to share something *we* were the measuring tool for.

Many attempts had been made at capturing the world as we see it through our eyes. The Shroud of Turin, dating from the XIV century, is believed by some to have been produced using some ancient type of photography since it so closely resembles a human face. Light sensitive materials had been documented as early as in the XIII century, and were a subject of interest in the XVI and XVII centuries. The first ideas for capturing light were documented in the XVIII century, with science fiction again forecasting the future with a fictional image capturing process described in Giphantie [167] in 1760.

Different techniques were tried in the first years of the XIX century, with the Dagerreotype (Figure 1.3) being the first published process describing how to take a photograph:

A silver plate had to be polished to a mirror finish. To avoid tarnishing, iodine and/or bromine fumes were used to create silver halides. The now light-sensitive plate was placed inside of a *camera obscura*, where a small opening projected the target image onto the plate. Fixing the image took from minutes to hours, depending on the brightness of the scene and the halogen used for sensitization. The exposed plate was then developed by means of exposure to mercury vapors, enhancing the image. Then, the light-sensitive material was no longer needed, and its removal required of a hot bath in a saturated solution of common salt. An improvement to this method included a bath in gold chloride to increase the image's stability.

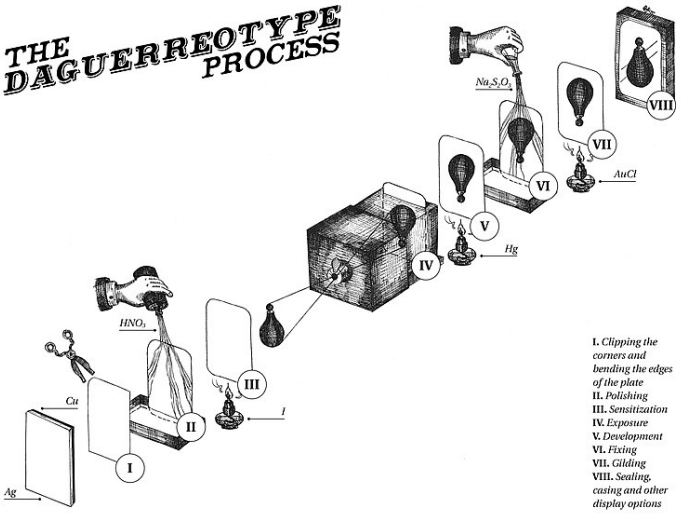

Figure 1.3: The different steps involved in the daguerreotype process for taking a photograph. [35].

Over time, this process was simplified. New substances were discovered that allowed for lower exposure times, and developing processes no longer needed toxic fumes to form the images. Color was introduced allowing for a more realistic representation of what our eyes could see. Cameras became smaller thanks to the use of lenses, and even portable. And with the advent of film, a single camera could take multiple photographs instead of having to replace a plate after each one. However the basic idea was still there: a light sensitive substance coating a surface which was exposed to light, and had to be later developed with chemicals.

But this was to change with computers taking over every task humans had been doing for the past millennia. Calculators replaced accountants, production lines were automated, even precise tasks such as watch making were eventually taken over by computer controlled robots. Data was being collected and then analyzed. We still wanted to know more, but know we were not the ones skimming through endless rows of information.

Figure 1.4: One of the first digital images, a scan of a photograph of a child. [118].

The first task to involve imaging was scanning. Documents were being transferred to a digital format for ease of use. At first, only characters were scanned, but this changed when, in 1957, a scan of a photograph was made to digitalize a picture of a child (Figure 1.4). This was a very limited and primitive form of digital "photography", since it was limited to objects stuck to the scanner's glass.

A remote sensor was developed in 1969 by Willard Boyle and George Smith called CCD (Charge-coupled device). A matrix of metal-oxide-semiconductor field-effect transistors (MOSFET or MOS) is exposed to light, accumulating charges proportional to the light exposure over time. Then, charges are shifted along the matrix across both directions so that, one by one, they exit the matrix and can be amplified, then converted from the analog to the digital domain. After reading the values from all pixels, the image is formed and transfered to memory.

These sensors allowed for the collection (and then study) of images from microscopes, telescopes and, of course, cameras. From bacterias to galaxies, we were able to compare analysis

across time and space, to fix in the digital domain what once was impossible to retain. From entertainment, sports, education, science... nearly all aspects of life have been impacted by imaging technology, and now every one of us has at least a couple of cameras at all times.

But measuring *visible* light was not enough. What we are able to see is just a fraction of the electromagnetic spectrum. At first, we wanted to just measure what we saw to be able to share it. But it was soon realized that, at least for science, there was much more to see in the (to the naked eye) invisible.

While photography was developing as a technology, infrared detectors were also invented. In the XIX century infrared radiation was discovered and soon remote thermometers were created. At the start of the XX century, infrared radiation could be detected at half a kilometer away with a precision of thousandths of a degree. The first application was iceberg detection, unfortunately just after (or maybe due to) the sinking of the titanic. Scientific applications include astronomy, chemical imaging or heat analysis (Figure 1.5).

Figure 1.5: An infrared image of a steam locomotive. [101].

Infrared has a longer wavelength than visible light, and ultraviolet has a shorter one. These high-energy waves were also interesting for scientific applications, being more reflected by certain materials. This has for example revealed hidden characteristics of the skin in forensic or medical analysis. The same sensors for traditional photography, with small modifications, have been used for this purpose since they are also ionized by the higher-energy waves. A filter is usually placed to block lower-wavelength light allowing only the high frequencies into the sensor.

So far, electromagnetic radiation has been measured and retained in digital form to be later studied. But it can only be measured at specific spots across the spectrum. Materials reflect light at all frequencies with different intensities, giving them a unique *spectral signature*. A spectral signature holds much more information than a single point in the spectrum, and so effort was put into developing sensors that could capture them. Spectrometers were developed to capture and retain this information, which could be used to even identify the elements present in a substance.

The usefulness of analyzing the full spectral signature motivated the creation of multispectral and hyperspectral images. These are not limited to visible light, expanding on both sides of the spectrum to infrared and ultraviolet. This meant enhanced analysis with a myriad of applications in astronomy, agriculture, medicine, surveillance... Each pixel, composed of tens to hundreds of samples of different wavelengths, has enough information to be compared with the spectral signatures of target elements or materials for comparison. Those can be identified for each pixel and more in-depth knowledge can be gained from the image than with traditional imaging techniques.

The challenge now is to properly manage these images. With increasing spatial and spectral dimensions, their size is digitally enormous even by today's standards, with memory or bandwidth being a limiting factor for some applications, as well as processing power for others.

With processors reaching the end of Moore's law, scalability is trending towards horizontal parallelism. Programming languages such as CUDA or OpenCL have emerged that work on a completely different paradigm than traditional imperative languages by simultaneously executing the same code over different sets of data. Processors themselves have been expanding to multicore architectures for a while, moving the bottleneck of processing power to intercommunication. Each core can operate over vectors of data to speed up further the computations. The use of GPUs with thousands of cores has skyrocketed with data growing in volume and new processing

algorithms adapting to the new paradigm of joining the forces of multiple smaller cores instead of using few very powerful ones.

When those approaches are not enough, custom hardware emerges that is specifically designed to operate on certain flows of data and algorithms. However, these specialized systems can only be used for the task they are designed to perform. More generic options such as multi-core CPUs or GPUs lack the performance needed for certain tasks, also drawing more power than simpler options.

When focusing towards hyperspectral applications, the problem resides in being able to quickly move the immense amount of data with minimal cost, as well as processing it, since many times the limit is not the capture ability, but the bottleneck that arises when processing cannot be done fast enough and the storage or communication link fills up. While custom ASICs can be designed, this is too costly and application specific, since often a single sensor or mission is targeted.

Field Programmable Gate Arrays (FPGAs) are a completely different option, and allow for the creation of custom hardware over reprogrammable silicon. A matrix of reprogrammable gates, interconnected by reprogrammable buses, that can mimic custom logic. A circuit that is designed in a Hardware Description Language (HDL) can be implemented, using automated tools, in an FPGA.

Nowadays, FPGAs offer not only reprogrammable resources but also include fixed hardware such as mathematical accelerators, processors, communication modules, transcoders... These options enable the creation of hardware that is almost as fast as an ASIC, but in a fraction of the time and cost, especially for devices that are produced in very small scales. This is the case with hyperspectral imagery and why FPGAs are very interesting to explore. Additionally, they can host multiple circuits at the same time, making them very appealing in places where adding improvements or changing functionality is of interest, such as research applications.

Thus, to explore the problem of processing hyperspectral data, and specifically reducing its size in real time with FPGAs to enable more data to be captured, this thesis was born: "Compression techniques for hyperspectral images on reconfigurable hardware".

1.1 Hyperspectral images

An image is a collection of equally spaced data points across a physical area, indicating the intensity of light at each point. The most basic image that can be considered is monochrome. A single sample in each data point (or pixel), indicating the intensity of light that was perceived at the moment of capture. In other words, the amount of electromagnetic radiation perceived in a certain part of the spectrum during the time that the sensor was active.

For black and white photographs, sensors are able to detect radiation in the visible spectrum (approximately $400 - 700nm$). The brightness can be determined, but not the color, since all wavelengths affect the final result: the sensor is not able to differentiate if the intensity was coming from an orange or purple source.

For color images, three different sensors are used, detecting blue, red and green light. This is inspired by the cone cells in human eyes, which come in three types, and are sensitive to those three wavelengths. In practice, three different monochrome images are captured, each for a different part of the spectrum. These are called bands. When the image is presented back to the observer, each band will translate to a different color on a viewing device, reconstructing the original image.

Human perception can only sense these three basic colors. But there is more outside of the visible spectrum. Radiation types depend on wavelength, and that can range from ultra short,

high frequency and highly energetic gamma rays to long, low frequency and low energy radio waves. Certain information is only present on those types of waves that lie outside of the visible spectrum. For example flowers are known to appear different under UV light depending on their maturity, attracting insects which can detect those short waves. Thermal information is also visible in the infrared part of the spectrum, remotely giving information about temperature.

Specialized sensors are available for monochrome images in these wavelengths, but more interesting are sensors which are able to capture multiple bands of the spectrum. When the number of bands is small, and the bands are not evenly spaced, these sensors produce *multispectral* images. They capture a subject of interest in the visible spectrum, adding certain characteristics with other bands (such as thermal or UV).

To consider an image *hyperspectral* (Figure 1.6), it must contain multiple *bands*, and those must be evenly spaced between the sensor's lowest and highest wavelength. Thus, discrete points are placed in that interval that approximate the continuous reality that lies between the extremes. The higher the number of bands, the closer the real spectrum can be represented. And similar as to how humans are able to tell apart colors by going from 1 to 3 bands, with hyperspectral images materials and substances can be distinguished even when they are the same color, since other characteristics of their spectrum do change.

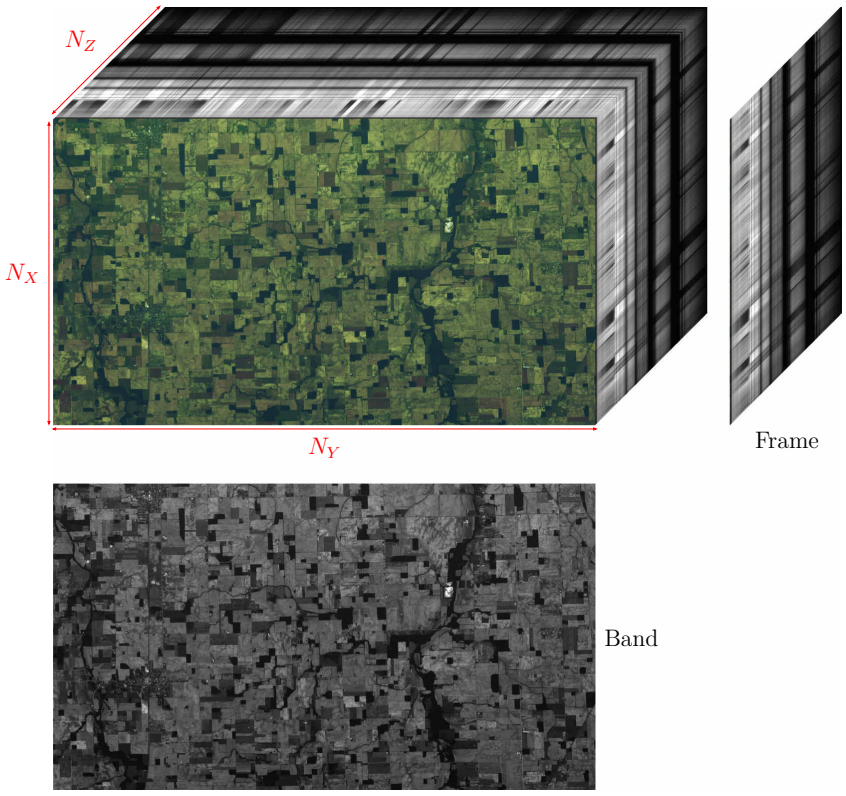

Figure 1.6: A typical hyperspectral image representation. On top, the projection of the Red, Green and Blue channels forms the visible-light image. Under it, hundreds of bands capturing intensity information on different wavelengths. Dimensions and notable sub-structures are highlighted.

An image has N_Z *bands* in the *spectral* dimension, and N_Y *frames* and N_X *pixels* as the *spatial* dimensions. The term line comes from the fact that sensors, to avoid complexity, generally capture frames (in the spatial dimension) one at a time, like a document scanner. Each frame will, in turn, have a number of *pixels*, which is equal the width of the sensor (when frame based). A full line of pixels is called a *frame*. *Samples* will refer to the individual values in an image (for each pixel, the value for each wavelength captured).

1.1.1 Sensors

There are plenty of hyperspectral sensors designed to capture hyperspectral images (Figure 1.7). They vary in how many bands they can capture, the wavelength range that they are able to sense, ground resolution, time to cover specific areas, and others. These parameters affect what kinds of studies can be done (depending on wavelength range) and how precise they can be (depending on the number of bands and the spacing between them). The different bands are

evenly spaced throughout the wavelength range of the sensor. Sometimes, a certain part of that range is of more interest (e.g: infrared radiation for fire monitoring) and more bands are captured in specific sub-ranges. Sensors can be used on the ground, but are generally either air or spaceborne since that is where most applications take place.

Another characteristic of sensors is how they capture the image. The amount of samples per pixel makes it impractical to build a sensor that captures all pixels at once. To solve this issue, different scanning techniques are used. Mainly a sensor can be of the push-broom (PB) type or the whisk-broom type (WB). A push-broom sensor captures whole frames by having a matrix that is able to capture a full line in all wavelengths at the same time. The image is progressively composed by capturing successive frames as the sensor moves over the target. A whisk-broom sensor captures pixels one by one. Commonly, the whole image is formed by having a set of rotating mirrors that selectively reflect light from different parts of the target in a raster-scan fashion.

One of the most popular and well-known sensors is the Airborne Visible and InfraRed Imaging Spectrometer (AVIRIS) [107], from which most images used in this thesis are taken from. It was developed by the Jet Propulsion Laboratory in California, and its main purpose has been airborne terrain exploration. With 224 bands, it offers a $10nm$ resolution across its $> 2000nm$ wide spectrum, which is precise enough to identify spectral signatures coming from thousands of different materials [193]. Different aircraft have flown it on-board, mostly capturing images from the USA. A second version called AVIRIS-NG [108] was developed afterwards, increasing resolution, and thus improving identification and monitoring of the Earth's surface and atmosphere. It maintains spatial resolution bRut doubles the spectral, giving much smoother curves for the obtained spectra. Its detectors are more precise and it increments data volume from approximately $100GB$ in gen 1 to $1000GB$. Both are aimed towards applications in ecology, geology, agriculture, environment, atmosphere and water analysis, among others.

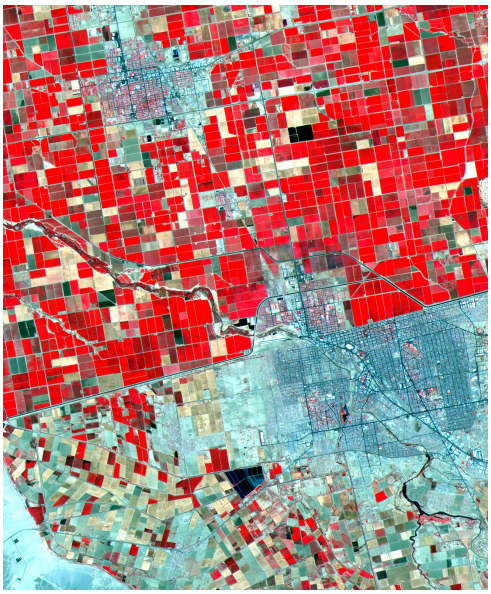

Figure 1.7: Crops near Mexicali clearly highlighted over background and cities thanks to the ASTER sensor. [44].

Hyperion [156], similar to AVIRIS, takes images with great spectral resolution (220 bands from infrared to ultraviolet) in a push-broom manner at a depth of 12 bits. But instead of being airborne, it is designed to be operated on orbit. Radiation hardening is part of the process of making a sensor, and calibration is important since once launched it is not possible to repair it in space. HYDICE [164] also works in a similar manner. Having a fine ground resolution of $3m$, it was used for mineral mapping.

Other very popular sensor is the Landsat Thematic Mapper (TM) [150], since it is the longest lasting mission to supply satellite imagery. Although not hyperspectral (with 7 bands it can only be considered multispectral), it has a very interesting feature: six of its bands are between $450 - 2000nm$, and one is at $11000nm$. This band is capable of capturing temperature information, very useful in forest fire monitoring or nighttime images. The last version, Landsat-8 [148] extends its range to 11 bands while maintaining its $15m$ panchromatic resolution.

SPOT (Satellite Pour l'Observation de la Terre) [78], launched two years later than TM, improved its ground resolution from 30 to 10 meters. The first SPOT version was a multispectral sensor with three bands: Red, Green, and near Infrared. A Panchromatic option yielded monochrome images at higher resolution. Subsequent versions added a fourth band (short wave

infrared), which was then changed by a blue band, targeting shorter wavelengths. Resolution has also improved over the years to an impressive $1.5m$ ground pixel size.

For higher infrared and thermal resolution, ASTER [45] has multiple sensors detecting four visible near infrared bands ($520-860nm$), six short infrared ($1600-2430nm$) and five thermal infrared ($8125-11650nm$). It is designed to map the entire globe every 16 days, providing close to "real time" images for applications such as mineral mapping or water management.

A faster sensor is MODIS [149], capturing the whole globe every day by sacrificing resolution ($250-1000m$ ground pixel size compared to $15-60m$ of ASTER). Large scale events can be seen even at this coarse grain resolutions (Figure 1.8), and a faster response can be given to understand and analyze terrestrial, atmospheric, and ocean phenomenology. The sensor contains precise on-board calibrators to extend its lifespan.

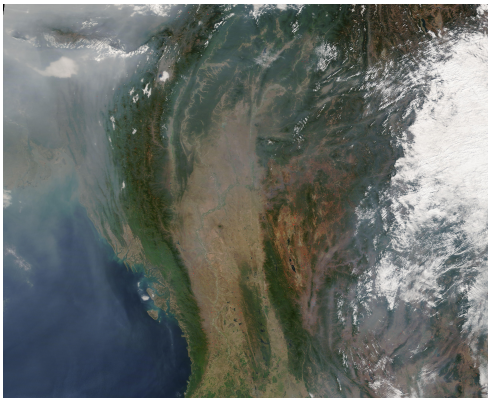

Figure 1.8: Image of forests of Myanmar captured by the MODIS sensor, used for fire detection based on thermal data [147].

Sensors with specific functionality include the Japanese Fuyo-1 [64], dedicated to geological, coastal and vegetation analysis. Its eight bands detect vegetation, chlorophyll, biomass, moisture and hydrothermal characteristics. The Warfighter-1 project [43] included a hyperspectral sensor among its payload to prove the viability of military applications of hyperspectral data, with ground resolutions of $8m$. A multispectral sensor with ground resolution of $4m$, and a panchromatic $1m$ resolution sensor were also included to complement the information recorded.

Newer satellites such as Enmap [63] are gaining spatial resolution for each spectral band, aiming for large swath widths ($290km$ for the Sentinel-2A sensor[46] which maps all of Europe and Africa ever 15 days), or focusing on extremely detailed resolution of the panchromatic bands [47].

There are hundreds [53, 62] of different sensors. Wavelengths, spectral and spatial resolution, ground pixel size, bit depth, air or spaceborne... All are different variables that prepare the sensor for different applications. From general observations with 1-day mappings of the whole earth, to precise images that can resolve people on the street in multiple spectral bands, what all sensors have in common is the huge image size, ranging from tens of MB to multiple GB (Table 1.1). Note that, while some of these are not hyperspectral, the same algorithms and techniques can be applied to them.

1.1.2 Data ordering

Hyperspectral data might be processed in different orders. This can depend on how the data is stored, but mainly comes from how the sensor captures the data.

Whisk-broom sensors make a full spectral depth scan over the spatial dimensions. They output whole pixels in sequential raster order, and samples within the pixel are output sorted by wavelength. This is called Band Interleaved by Pixel (BIP) ordering. Since the spectral dimension is usually the smallest, this offers the best correlation between neighboring samples in the output stream.

Push-broom sensors take full frames that will be stored one after another. Each frame itself is traversed in raster order, either first in the spectral direction (again BIP ordering) or in the spatial direction within the frame. This yields the Band Interleaved by Line (BIL) ordering. This ordering offers a optimal memory pattern access for retrieving full bands at the cost of worse pixel access, though it is not often used.

Sensor		Image					Wavelength		Platform
Name	Type	N_Z	N_X	N_Y*	Bits	Size**	Min	Max	
Aviris	WB	224	677	512	12	$116MB$	380	2500	Plane
Aviris-ng	WB	480	600	512	14	$258MB$	380	2510	Plane
Hyperion	PB	220	250	1000	12	$82MB$	400	2500	Satellite
HYDICE	PB	210	320	320	12	$32MB$	400	2500	Plane
Landsat TM	WB	7	6166	5733	8	$250MB$	450	12500	Satellite
Landsat 8	PB	11	2k-12k	2k-12k	12	$594MB$	441	12510	Satellite
SPOT-1	PB	4	3k/6k	3k/6k	8	$144MB$	500	890	Satellite
SPOT-7	PB	4	10k-40k	10k/40k	12	$6.4GB$	450	890	Satellite
ASTER	Both	14	1k-4k	1k-4k	8-12	$14MB$	520	11650	Satellite
MODIS	PB	36	2k-9k	10-40	12	$20MB$	405	14385	Satellite
Fuyo-1	PB	8	4096	2048	6	$50MB$	520	2400	Satellite
Enmap	PB	228	1000	1000	14	$423MB$	420	2450	Satellite
Sentinel-2A	PB	12	7k-29k	7k-29k	12	$3.8GB$	443	2190	Satellite
Worldview-3	PB	29	9k-35k	9K-35K	11-14	$10GB$	400	2245	Satellite

Table 1.1: Comparison between different sensors. PB: Pushbroom, WB: Whiskbroom. *When not available, the number of frames captured has been assumed to be equal to the number of pixels per frame, though most sensors capture variable-length runs of frames. **Sizes are approximate since different bands have different resolutions and bit depths, but offer an idea of how big these images are.

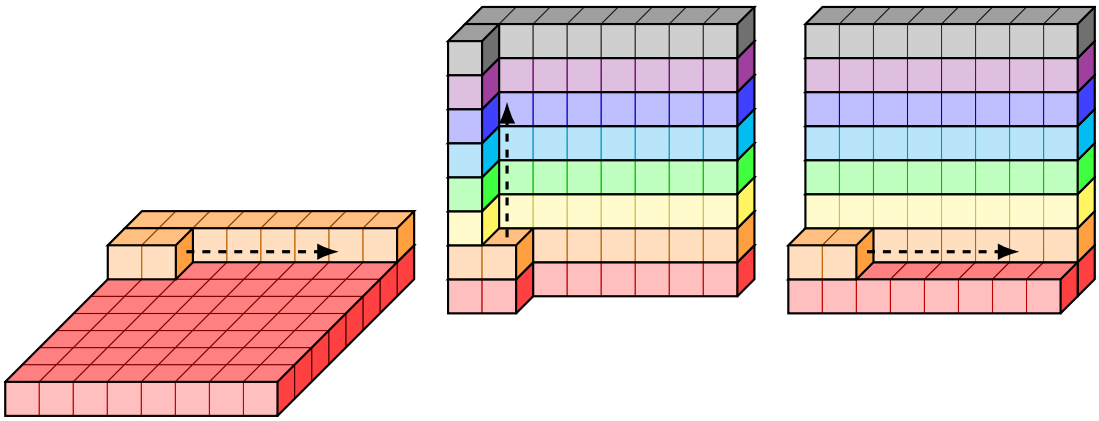

Figure 1.9: BSQ, BIP and BIL data orderings respectively.

Lastly, data might be ordered in a Band Sequential (BSQ) mode. When analyzing each band separately, this mode offers the best memory layout for quick access, since bands appear sequentially in memory. Hyperspectral sensors do not usually output directly in this mode, and it is the result of reordering afterwards to facilitate scientific analysis.

All three types are shown schematically in Figure 1.9. BSQ is oriented towards ease of full band access, with pixel access being slow since samples of the same pixel are very far apart. BIP is aimed towards full pixel retrieval, with very costly band access. BIL sits in between with good frame retrieval, decent band retrieval due to access patterns, and bad pixel retrieval. For compression algorithms, it will usually be the case that pixels are the basic units for processing, so BIP will be the preferred mode, also taking advantage of the great spectral correlation present more easily. It will also offer straightforward data accesses since most images are already captured this way, incurring in less memory overhead in an FPGA.

1.1.3 Applications

One of the main motivations for the development of hyperspectral images is, as with many technologies, the **military applications** [13, 30, 127, 204] that come with it. From training to war to humanitarian missions, hyperspectral images reveal hidden information about the terrain that is about to be explored, helping and guiding military personnel to whichever objective they have.

One of the most useful military applications is target detection and identification [30, 204]. Targets that are camouflaged in a certain wavelength can be detected in others, military vehicles can be identified and differentiated from civilian ones, and different man-made materials can also be identified to unveil bases, weapons, aircraft or other subjects of interest. Decoys that fool traditional detection techniques can also be identified with hyperspectral image processing. For this, two approaches are commonly used: anomaly detection algorithms detect even unknown targets by comparing against the background truth, while signature-based detection compares against a known spectral signature to detect a specific target. Multiple algorithms can even be used at the same time by using classification fusion [127], improving result accuracy.

Terrain analysis is another motivator for military research, since soil and vegetation can be characterized ahead of time to allow for more precise ground exploration afterwards. Information about soil, hydrology, vegetation or topography can be combined with already known information from maps such as roads and land use to create vehicle trafficability models [190], allowing for better movement of ground missions. The HYperspetral iMage EXploitation (HYMEX) program [13], supported by the Canadian Forces, studied different terrains such as forests and grasslands. Along with the Universities of Lethbridge, York, and Alberta, terrain analysis was performed to improve and complement maps developed by The Canadian Forces Mapping and Charting Establishment (MCE).

Water mapping [13] is especially of interest in near-shore applications. For this, interesting data includes characterization of the seabed in shallow waters, identifying algae and other tidal vegetation. Bathymetry, indicating water depth, can guide ships where there are dangers of getting stranded. Beach characterization is also important specially when taking into account time, defining patterns in coastal changes that are mainly derived from currents and tides.

For **vegetation control** and analysis, hyperspectral imaging helps identify certain characteristics and properties of leaves without the need for an in-situ analysis. Indices [202] are inferred through analysis of hyperspectral data and ground truth. These are formulas that predict moisture, element concentration, biochemical stresses and others. As an example, chlorophyll can be predicted by the index given in Eq. (1.1), where R_x is the reflectance for light with a wavelength of x nanometers:

$$\frac{(R_{700} - R_{670}) - 0.2\,(R_{700} - R_{550})}{R_{700}/R_{670}} \tag{1.1}$$

When growing crops, potassium is a key element in fertilizers. Different predictors have been proven good indicators of potassium content in leaves [137] by analyzing certain spectral bands in which this element is most reflective. Different fertilization treatments for corn, soybean and wheat have also been successfully monitored with these techniques [87].

For certain crop fields, knowing if hazardous materials are present is important in order to assess the quality of the final product, since these materials are absorbed by the plants. Heavy metals are of special interest since they are dangerous for human health. Arsenic has been successfully identified [187] in rice fields thanks to different predictors that exploit its reflectance in the 716, 568 and $552nm$ wavelengths. Field sampling, followed by laboratory analysis can be replaced by hyperspectral imaging to make this assessment process faster.

Crop biophysical variables can also be characterized through hyperspectral images. Wet biomass for estimating plant size, leaf area index and plant height to gauge development, and

grain yield are amongst the variables that are identified [203]. Chlorophyll content in leaves, which is a good indicator of plant activity, mutations or nutritional state is also of interest. Hyperspectral images estimate chlorophyll with high precision [218] so as to improve precision agriculture amongst others.

In **geology**, materials can be remotely identified with hyperspectral images. Ore and hydrocarbon prospecting, hydrothermal alteration and surface mineral mapping were amongst the first applications for the first generations of sensors [141]. These were later improved when moving from multi to hyperspectral imaging, allowing for applications such as vertical mine face mineral detection [145], where a mountain range can be tagged with the most abundant materials on each zone.

In mining quarries, the process of identifying ores or rocks that are of interest [125] is sped up, since overhead images can differentiate different materials that appear the same to the naked eye. Minerals such as carbonates, silica, muscovite, dolomite or limestone can be remotely detected from airborne sensors [121]. Even if the maps are noisy, they are a good reference since they have been found to be coherent with the known geology [50]. Identification starts being difficult though for spaceborne sensors since their spatial resolution is often too coarse for fine-grain detection.

Food safety also benefits from hyperspectral imaging because of its non-intrusiveness. Near infrared spectroscopy is being replaced with hyperspectral analysis [81] since the growing computing power of current devices is able to process hyperspectral data from a processing line in real time. Usually an analysis is done beforehand to identify the critical spectral bands that are of interest, to simplify data processing. Meat, fish, fruits, vegetables, mushrooms and cereals are amongst the most studied products [199], detecting unsafe food before it gets to the consumer.

Animal carcasses can be separated into different classes automatically, identifying septicemic and tumorous bodies [138]. Spectra were found to be different amongst all different types, detecting not only unwholesome carcasses but also the different problems they might exhibit. This techniques can also be applied to vegetables, identifying fungal contamination in fruits [117] thanks to a dual-illumination system where visible light was complemented with an ultraviolet lamp that exposed the problematic items.

Specific characteristics of food have also been identified: Strawberry moisture and acidity can be identified [61] to grade the product's quality. Potato water content and weight can also be detected through neural networks [109], allowing for faster sorting than human visual inspection. Even internal features can be detected. Pickling cucumbers need to be in perfect condition to avoid degradation in the maturing process. While external injuries are easily identified, internal ones are difficult to detect. Spectra of the defective cucumbers are similar in shape but of higher magnitude in injured ones, allowing for detection [55].

Medicine uses close-up hyperspectral images for disease diagnosis and image-guided surgery. Propagation of light through biological tissues is studied beforehand so that images can be interpreted to give an accurate result of the target property that is looked for. Cancer detection is a very popular application [136], especially in skin since it is the most visible organ.

Tongue analysis to diagnose human ailments, traditionally done by an expert with years of expertise and later on by RGB cameras, has been improved due to the introduction of hyperspectral data [241] and segmentation techniques [133] that identify areas of interest that can be used to generate accurate predictions based on tongue images.

Cancer regrowth is one of the problems when dealing with complex tumors that might not be completely removed after surgery. Biological markers in cancerous tissue can be detected thanks to hyperspectral images [155, 237] in real time, aiding cleanup after or during surgery *in vivo*. Sensitivity and specificity reach levels comparable with human inspection.

1.1.4 Algorithms

All of these applications require of powerful algorithms that are able to work with this high-dimensionality data in a correct, fast and efficient way.

One of the most recurrent problems is being able to identify the different spectral signatures within an image. Pixels can be of two types: Pure and mixed. Pure pixels contain information about only one basic element (e.g: a certain type of rock), while mixed pixels contain information of two or more basic elements (e.g: water and vegetation in coastal images).

Hyperspectral applications require identification [139] of the constituents of the different pixels. A general assumption is to assume that an image is sufficiently big to contain pure samples (*endmembers*) of all basic elements that constitute the rest of mixed pixels. This way, all pure elements will be vertices of an n-dimensional surface that contains all the mixed pixels inside. Algorithms usually take into account that outliers might exist due to sensor malfunctions, transmission errors or plain noise affecting the capture process.

One of the most popular algorithms is N-FINDR [216]. It is based on the assumption that the volume contained within a simplex of the purest vertices (pixels) is greater than that of any other pixel combination. A random selection of candidate endmembers is iteratively tested to see if other pixels might increase the volume when replacing any of the selection. The algorithm eventually converges to a simplex of maximum volume (not necessarily a global optimum but it is a local optimum according to this iterative process). N-FINDR requires an initial estimation of the amount of endmembers in the image, and cannot determine that number on its own. Improvements have been done [159] to initialize the algorithm with the appropriate number of endmembers, increasing its efficiency and reducing iterations until convergence.

The Pixel Purity Index (PPI) [29] is another algorithm that unmixes hyperspectral data based on creating a convex hull over the set of pixels, in which the vertices are the pure endmembers. As with N-FINDR, it is iterative and good results are only achieved with long runtimes. There are however variants [37] of the algorithm which decrease execution time.

Faster algorithms have also been developed under the linear mixing model. VCA [151] does progressive projections over the current subset of endmembers, adding the projection extreme to the subset until all endmembers are exhausted. Its simplicity achieves higher classification performance than N-FINDR or PPI while being much less computationally demanding. MVES [36] uses linear programming to improve VCA for higher accuracy.

All of these algorithms work under the assumption that the image is of certain quality, and does not present noise or any kind of spectral or spatial artifacting. This is not always true.

Atmospheric correction is one of the first steps in preparing an image. Heat, humidity and cloud cover are amongst the variables that greatly affect the end result of imaging a specific area. Water vapor, oxygen, carbon dioxide and aerosol scattering greatly affect the perceived signals by the hyperspectral sensors [74]. For example, images of open water bodies usually need small corrections due to cleaner air, while coastal and turbid water images [73] require a more aggressive filtering to remove atmospheric absorption and scattering effects, yielding an image that only contains water-leaving radiances for pure surface analysis.

At first, purely experimental models were used. Heavy assumptions about the available samples were made, such as a relatively flat (spectrally speaking) portion of the image being used as a baseline to estimate the atmospheric error and correct it in the rest of the image [166]. Later, corrections became more complex by using radiate transfer models. Variables such as sun angle, gas concentration, or aerosol presence are measured in order to estimate the expected errors introduced by the atmosphere in the path the light takes from the sun to the sensor after bouncing from the surface. Even the information conveyed by some bands can be used to estimate these variables on a per-pixel basis to precisely adjust the image [72].

Even then, anomalies might be present due to errors in the sensing, coding and transmitting process, as well as unexpected elements or contaminants being placed in between the sensor and target. In this case, anomaly detection algorithms detect these unexpected anomalies without the need of reference data [195]. They range from simple linear algorithms, that assume Gaussian distributions and mixtures of pure pixels, to more complex non-linear approaches [126] that better differentiate anomalous pixels. Further analysis might classify these anomalies as errors, or as objects of interest due to their particular characteristics.

Target detection works in a similar way, but instead finding out objects that stand out [140] within areas where pixels are not necessarily anomalous. This has been one of the main motivators for military use of hyperspectral data.

Even with corrected images, hyperspectral analysis might be hindered by data availability. While very high in spectral resolution, spatial resolution might be lacking in some cases. Hyperspectral sensors already have thousands of detectors that, if made smaller to fit a bigger resolution, are not as precise. Spatial resolution of the hyperspectral sensor can be increased in software in a process called super-resolution. At first [206] the information available in different bands was used to increase the resolution. Hardware solutions were partially developed by also including a higher-resolution multispectral or panchromatic imaging devices along with the hyperspectral sensor. Techniques were developed [8] that fused both high spatial and high spectral resolution data sources into a single high spatial resolution hyperspectral image. Bayesian approaches [9], neural networks [236] and coupled hyperspectral unmixing [128] are among the popular approaches.

Hyperspectral images, despite containing plenty of information themselves, are sometimes combined with other types of data in a process called Data fusion. This is not a technique new to hyperspectral images, and has been used in the past to combine information from multiple sensors into one coherent source of data [88, 211].

Hyperspectral data has been fused with Light Detection and Ranging (LIDAR) in order to identify both tree species as well as estimating tree canopy height and diameter [176], resulting in an increase of precision on both domains with respect to just analyzing hyperspectral data for the former, and LIDAR for the latter.

It has also been mixed with both Synthetic Aperture Radar (SAR) and High Resolution Imaging (HRI) [94]. The former reduces false detections for the radar images, confirming target detection for military purposes. The latter combines the hyperspectral cube with the image plane to create a combined spectral-spatial analysis that allows for a more accurate material detection process.

These are some examples of algorithms that work with hyperspectral images at a general level. Of course, each application (Section 1.1.3) will derive its own algorithm suited for its use case, based on the data enhancement and preprocessing algorithms presented here.

Another main type of generic hyperspectral image algorithms are compression algorithms. They take the immense amount of data available, reducing its size so that storage and transmission can be more efficient. Generally a part of a more complex processing pipeline, they ease complex tasks by minimizing data transfer. These are the ones this thesis focuses on.

1.2 Compression

We perceive the world around us as a continuous stream of stimuli. Human senses detect those and transmit them to the brain, which is able to interpret those signals, creating our own reality. Those experiences are stored in it, and then can be recovered via memories that can be shared with others. A more detailed or brief explanation can be given of different experiences to others,

sharing information that others can use to understand different perceptions of reality. In a naive way, how much information is transmitted by humans can be measured by the number of words. Assuming an efficient description, the more words that are transmitted, the better the listener will be able to reconstruct the original thoughts.

For the digital world, these continuous analog signals are brought into the discrete digital domain. Sensors measure the signal's intensity and then the interval of possible values is divided into sub-intervals. The digital signal's value will be that of the sub-interval in which the analog measure falls. Once a discrete number is obtained, it is represented in bits, or a sum of powers of two. The shorter the sub-intervals, the lower the measurement error at the cost of more bits per sample, and vice-versa. Normally, a signal is sampled multiple times across time and/or space. The set of all samples is the data d.

Because storing data takes physical space, the highest possible precision is desired while using the lowest possible amount of bits. By default, if a signal is measured m times with p bits of precision, data d is produced of size $s(d) = m * p$ bits. A question arises then: is it possible to store that data in $n < m * p$ bits?

The short answer is yes. Signals almost always have a certain degree of predictability, that is, they are not fully random. As with words, when something is redundant, it can be omitted, either referring to previous information or leaving it out of the speech for the listener to infer based on context. The techniques are abundant, and allow for *conveying* all of the information without *transmitting* it all.

The art of reducing data size is what is called compression. When compressing data, it can be done in a *lossless* way or a *lossy* way:

- Lossless compression is a process in which the original data, d, gets transformed by some algorithm a in some new data $d' = a(d)$. The algorithm or function a is invertible, meaning the original data can be recovered via $a^{-1}(d') = d$.

  Of course there are functions that satisfy this property: the identity function is a trivial example. But if *compressing* is the objective, at least for some data d of size n it must be true that $s(a(d)) < n$. The problem is that, if there is some d_1 for which that property holds, then there must exist some d_2 for which $s(a(d_2)) > n$. Otherwise, by the pigeonhole principle, two different sets of data collapse to the same result, and the inverse is impossible to obtain.

  So for a compression algorithm to be lossless, and useful, it must reduce the size of some data combinations while expanding the size of others. At first glance, this seems absurd since, on average, an algorithm applied over a random set of data will get results that on average are the same size as the starting data. The key in designing good compression algorithms is that the data they operate over is not random.

  The signals that are compressed are continuous in the analog domain, and even by taking discrete samples (both at the resolution level and sampling level), their values will be close together. Values that are similar are easily encoded (for example the differences between adjacent values can be stored instead of the full values). The original interval can be clamped to the maximum and minimum values and use less bits for each sample in this restricted space. Predictive models can be used that, if correct, completely skip the need for coding certain parts of the signal.

  All of these methods will be able to compress most data sets with great efficacy, and those that are expanded in size will be so rare that they will not pose any problems. This is the key: compress the common, since the uncommon will not make a difference.

  This property of being able to compress any data set gives lossless algorithms a nice property: they work over any kind of data. Even if they are specialized for images, audio

or medical data, any can compress data from the others. They are generic even if they might not be efficient with all types of data.

- When analyzing data, most results do not depend on exact values but on trends or patterns. For example, to know if temperatures have been higher than in previous years, exact values are not needed, but an average or trend line showing where anomalies might have occurred.

  Lossy compression aims at reducing data size by getting rid of the data that, even when not present, does not affect those properties that are useful. Noise and outliers caused by sensor errors will be removed, and signals smoothed out to make them more predictable.

  While the effectiveness of a lossless algorithm is measured solely on how much it reduces some reference set of data, lossy algorithms are measured on two parameters: size reduction and quality loss. Of course a lossy algorithm can be perfect in size reduction by ignoring input data and restoring a blank signal, but the reconstructed signal would be useless. It can also preserve the signal intact yielding perfect quality, but lossy algorithms underperform lossless ones in that regard. Generally, a set of parameters will be selected to achieve a compression of a certain approximate quality or ratio.

  The mathematics and computing behind lossy compression are usually more complex than the lossless counterpart. A transform such as Fourier, cosine or wavelet is the central part of most lossy algorithms. They map the input signal to a space where it is defined as a combination of more common and repetitive signals. Generally for a signal s of n samples, a transform t will map it as:

$$t(s) = \sum_{i=1}^{n} c_i s'_i \tag{1.2}$$

  Where the original signal s is expressed as some combination of signals s'_i. The n coefficients c_i are of the same size as the n samples of s, so the transform can be inverted and doesn't expand the size of the original data. The signals s'_i usually represent common patterns that arise in continuous signals, and can mathematically create any other. s'_1 will be the most common pattern, and s'_n the most uncommon. This way, when compression is needed, the least significant $m < n$ coefficients will be set to zero. The reconstructed signal will be similar to the original one but at a fraction of the size.

  Contrary to what happens with lossless algorithms, lossy ones are very specific to the kind of data they compress. Audio, image or video compression all have different requirements, and using one compressor for the wrong kind of data yields abysmal results.

1.2.1 History of compression

It was Claude Shannon in 1948 [184] who defined, mathematically, what is now known as information theory. With the definition of entropy, a theoretical limit was established that dictated how much information could be sent over a link of certain properties. If certain properties are known beforehand (such as what symbols are transmitted, as well as their relative frequencies) it is possible to send messages using less data than if a random source of symbols was assumed.

This is possible thanks to the use of codes, which are dictionaries that map the symbols that are to be sent to the symbols that the communication channel supports. A simple code might map the alphabet to 5-bit fixed-length binary numbers. That way a message of n characters is transmitted with $n * 5$ bits. But, if variable length codes are used, where the most common characters use shorter codes, the same message might be transmitted with e.g: $n * 4$ bits, saving 20% of bandwidth.

Different codes were explored at the beginnings of information theory, exploiting different mathematical distributions of the input alphabet. Golomb [79] and Huffman [96] codes were amongst the most popular. The first offered great compression for unbounded collections of

symbols with known probability, while the latter created the most efficient codes within a fixed set of symbols. More complex codes were developed afterwards, with arithmetic coding [3] reaching the limits of what was possible based on information theory.

But of course, as time went on scientists asked themselves whether this limit was really the limit. A simple string of alternating zeros and ones could not be compressed at all with any of these methods, since both symbols had exactly a 50% chance of occurring. Nonetheless, such string is extremely redundant, since the moment symbols are grouped in pairs, suddenly the same symbol is repeating 100% of the time.

This simple example shows that, even if information theory limits hold for the general case of an unknown source of *random* data, some skewed data might be compressible to limits beyond theoretical. This motivated the creation of the next generation of compression algorithms, based on dictionaries that evolved over time. Lempel and Ziv were the ones that popularized such methods with their LZ family of algorithms. In both LZ77 [243] and LZ78 [242] they exploit the fact that data sequences (in this case mainly text) contain many repeated subsequences. LZ77 maintains a window of previous symbols which it can reference for repetitions. LZ78 builds up a dictionary in which symbols reference previous entries within the dictionary, creating strings that can be referenced with the same amount of bits as individual symbols, saving space. Both algorithms dynamically exploit redundancies and can encode long strings within the space of a symbol, improving compression over one that just works on individual symbols.

The first improvements for these algorithms were bijective transforms that could be applied to the data in order to make it more redundant. The Burrows Wheeler transform [31] creates strings where symbol redundancy is locally higher by applying sorting transformations. A simpler approach is to use delta coding: code the difference between neighboring values, which for redundant data will be lower in mangitude (and more redundant due to the pidgeonhole principle) than the originals.

These transforms are still in use today in many algorithms, but that didn't stop improvements from being made for the LZ algorithms. LZSS [196] and LZW [215] are two examples that built up on the base LZ77 and LZ78 algorithms by adding new techniques. But the most popular is probably the DEFLATE [112] algorithm, that Phil Katz developed for the `.zip` file format which most of us still use on a daily basis. It builds up on LZSS by chunking the data and applying different compression techniques over each chunk, adapting to characteristics that are optimized for different variations of the same algorithm.

So far, these were all lossless techniques, since data storage was not an issue. But that was about to change with the digitalization of audio and images. A vinyl disk stores analog audio within its grooves the same way film is able to capture light in an analog fashion with light-sensitive materials. The first digital audios and images were mostly experimental, but soon it was realized that the main thing the digital world was bringing was data. Tremendous amount of data since the capture media could be reused. The first lossless algorithms fell short.

A CD, at $700MB$, could store just one hour of uncompressed music, enough for an album. However, there was too many useless information in those $700MB$. Due to the way the human ear and brain works, some sounds are virtually imperceptible to us despite being recorded by a digital microphone. Sounds that are too dim compared to the surrounding effects are not perceived by our senses, and certain peaks can be completely discarded while still retaining a high fidelity sound.

Once the inaudible parts are filtered out, Fourier transforms are applied converting the data from the time domain into the frequency domain. Very high and very low frequencies are discarded without affecting sound quality. The result is that, in high quality `.mp3` format, 7 hours of sound can be stored in a CD. For specific sounds such as speech, with a very limited frequency domain, up to hundreds of hours can be stored in the same media.

Images extend the audio concept of one dimensional waves to two dimensional ones. Line by line, the same concepts can be applied, but image characteristics are more efficiently exploited when using two dimensional transforms. In the same way that a Fourier transform is used over subsequent windows of sound samples, images use transforms over small blocks of fixed size. JPEG [210], one of the most popular compression algorithms, uses the discrete cosine transform. Operating over 8×8 windows, it transforms the block from the spatial domain into the frequency domain, seeing each block as a combination of two-dimensional waves. In this case, very high frequencies are discarded or its precision reduced, while low frequencies, that are more important in the visual domain are kept. This process smooths out the images even eliminating noise, but can sometimes create sharp edges between adjacent blocks.

To solve this, wavelet transforms were later introduced. They extend the concept of transforming to the frequency domain but over the full image. Blocks are of unlimited size, completely eliminating edges. But operations are computationally more expensive, which is the reason that these techniques didn't emerge until later standards such as JPEG2000 [189] appeared. For consumer use they are still not being widely used since JPEG usually proves to be sufficient.

Moving onto hyperspectral, a new dimension is introduced. Audio was made up of one-dimensional waves, images are two-dimensional, and hyperspectral images are three-dimensional. Two of its dimensions are *spatial* while one is *spectral*. While there is still correlation between adjacent samples, it is generally the result of two distinct effects. Spatially, samples that are close together tend to be similar due to the smooth nature of images. Spectrally, they are similar due to the continuous nature of each pixel's spectrum.

Algorithms have been developed that extend the concepts present in 2D image compression to 3D. Hierarchical trees (that split the image in blocks which are expected to have similar samples) give good results [40], while wavelets in three dimensions have been proved to achieve higher compression ratios [69, 157]. In these cases, both correlations are exploited at once. Computationally the algorithms are demanding, but obtain good results at compressing the images, achieving decent ratios with great quality.

A second type of hyperspectral compression is based on dictionaries [104]. Going back to the origins of compression, if a dictionary is created based on the frequencies of the symbols, less bits can be used for the most common ones. Since symbols in hyperspectral images are pixels of hundreds of samples (reaching kilobits), a dictionary can store thousands or millions of different symbols before being less effective than raw coding. Complex unmixing algorithms are used in this case, but the results are comparable to others. The benefit here is that, apart from compressing, each pixel's material is identified at the same time.

Lastly, methods that separately process both correlations have been proved to outperform the 3D-wavelet and dictionary based ones. The process is usually a spectral dimensionality reduction followed by some traditional image processing technique. Principal component analysis [57], combined with JPEG2000 has been shown to get excellent results at very low bit rates. When also aided by vector quantization, splitting the input pixel space into different categories, the results improve even further [24]. This is proof that with complex data, complex algorithms that exploit all possible ways of redundancy do achieve higher compression ratios than simpler ones.

1.3 Reconfigurable Hardware

The first devices to have any kind of intelligence were mechanical. Clocks and watches kept track of time by means of precisely engineered gears. Complex models of everyday life looked almost real thanks to all kinds of inventions that synchronized moving parts. Eventually, complexity evolved to mechanical calculators, which were able to take arbitrary inputs and convert them

to the appropriate result. In a sense, this generation of hardware was the *hardest*. Once built, functionality was completely fixed. If a watch kept track of seconds, it would always keep track of seconds and not hours. If a calculator could add but not multiply, then multiplication would always be out of the equation.

It was this idea of creating artificial calculators or brains, that was able to move us past the *hard* hardware barrier. Vacuum tubes enabled the creation of the flip-flop, a circuit able to store two different states which still is the base of computing.

The first vacuum tube computers were not programmable, being only able to solve specific problems such as linear equations. In a sense, these were still *hard*. But, as with many technologies, war was what really drove it forward. WWII brought the need to decipher encrypted communications, and with it came the first truly programmable computers. They used switches and plugs, with which programs could be changed to do different functions over the input data. Hardware no longer had the functionality within the circuit, the functionality was given by software, which worked on *generic* hardware.

Improvements were quickly made over the years, increasing tube quality and count, but the true revolution came with the transistor. In the 1950s, the so called "second generation" of computers appeared. Vacuum tubes, which were quickly reaching their limits in size and efficiency, were replaced by transistors and diodes. The first generations were less reliable than tubes, but they were already thousands of times more efficient in terms of power.

Soon enough, vacuum tubes were completely obsolete, and it was claimed that transistors were far away from their limits. Moore was the one to realize this, stating that transistor count would double every few years. This statement has remained true to this day, even though improvements are slowing down. But now, the end is near due to physical constraint, with transistors being just a few atoms wide, heat dissipation nearing its limits, and yield die limiting the silicon area that a chip can have.

Over time, this generic hardware improved in speed and functionality, but the core idea remained the same: create circuits that can solve some basic functions that can be combined to create complex behavior. That combination is programmed, and so a circuit can be reused for multiple purposes.

The only issue with this kind of hardware is that, due to its versatility, it has more functionality than its needed for any specific application. And that unused functionality translates in unused transistors. This means that operations will take longer to perform and power consumption will be greater.

The solution seems simple: create custom hardware for applications that have specific constraints that a multi-purpose circuit cannot cover. But this is very costly. Development of a chip can cost millions, and for small production runs this is infeasible. Some market niches do have specific hardware, such as GPUs, video encoding/decoding or encryption. Those chips can be programmed with software and deliver great performance for their applications. However, their specificity makes them inviable for general purpose processing, where they are inefficient.

But what if custom hardware could be created for any kind of application? That is exactly what engineers asked themselves in the 1980s. At first, *programmable hardware* consisted on a matrix of connections and logic gates, and was called Programmable Logic Array (PLA). The input to the logic gates can be programmed, meaning they feed off the user-controlled input or a programmer-defined value, changing functionality. Complex logical functions can be made, and these reprogrammable units were used as controllers, where the circuit's state could define the different component's actions in order to control the circuit. In theory, any function could be programmed if enough resources were available. The problem was that for certain functions, the size of the gate matrix needed to program them was too big to be practical, with many resources unused.

Other technologies also emerged at the same time, such as PALs (Programmable Array Logic) or GALs (Generic Array Logic), both improving PLAs by having more reprogrammability options. Still, scalability was an issue, and complex functions could only be made with newer products that had bigger gate matrices.

Engineers had to go back to basics. When any design is created, it is done in a modular way. The final circuit has some inputs and generates some outputs based on those. Inside, blocks of simpler functionality are connected together. This process goes down to the logic gate level, where simple logic gates are connected to build up functionality all the way to the top.

So that is exactly what they built, an array of simple blocks that could be selectively connected with others. Any circuit could now be mapped to the internal resources by reprogramming each block's functionality and the interconnections with others. Certain blocks had connections to the outside and would be used as I/O. This was called Field Programmable Gate Array, or FPGA.

1.3.1 FPGAs

Xilinx was the first company to produce FPGAs [230] along with watch manufacturer Seiko in 1985. The first FPGA wafers were full of short-circuits. Out of the initial 25 die run, only one was partially working. By applying enough current to the chip, the shorts blew like fuses and the first bitstream was able to be loaded: a simple inverter. Soon the full chip was able to be configured, and the manufacturing processes were improved removing the shorts. FPGAs were moving ahead in uncharted territory.

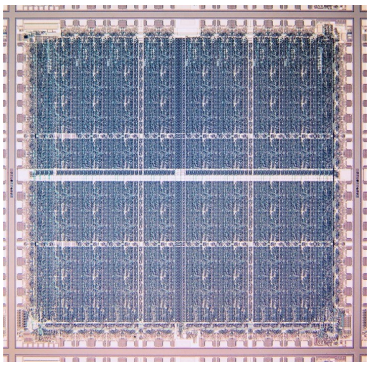

Figure 1.10: The first FPGA: The XC2064 [194]

At a cost of $55 and with a process of $2.5\mu m$, the XC2064 [169] offered 800 gates arranged into 64 configurable logic blocks (CLB). Each of those had a three input look-up table (LUT) that could be configured into any 3-input function. Even at that low CLB count, transistors went unused for many functions. At a time when every transistor was valuable, this seemed like a waste of resources and money. But Xilinx's founder Ross Freeman trusted Moore's law and transistors becoming so cheap that it would soon not be a problem. And he was right.

Xilinx's invention [231] was so powerful that it soon took over PLAs, PALs and GALs. It could perform their functions and more, replacing many ICs that previously had to be sourced independently. In just a couple of years, price went down to $15 with production ramping up to tens of thousands of units. The software though was a bit more expensive.

FPGAs, while very powerful, had one big limitation. While ICs normally come with simple instructions of how to handle inputs and outputs, FPGAs also have the added complexity of building the design that is programmed into them. Xilinx realized this and soon enough was selling software to automate this process. Logic functions could be defined, and a tool would automatically translate them to configuration files for the FPGA. This relieved engineers from having to manually configure the internal logic, which would have defeated the purpose of the FPGA in the first place. wo FPGAs continued to grow [205] in the 90s. Aside from pure logic, FPGAs now had integrated memory, and as such could contain much complex circuits than before. In 1992 the XC4010 surpassed the 10000 gate barrier, and just seven years later, the XCV1000 would reach a million. Such growth was due to Moore's law, and it applied everywhere in the industry. Processors, memories and network devices were growing in size and in types. While this first generation of FPGAs was useful, providing only reprogrammable logic was not enough.

With the expansion of the Internet, the need to process information from multiple sources, multiple protocols and at the same time was crucial for interconnecting different devices and networks. Custom ICs were available, but having one for each situation was too costly. FPGAs found a niche to grow thanks to their quick reprogrammability, and the enormous need for networking hardware accelerated their growth.

By the start of the 2000s, FPGAs were present in many digital systems. Companies that used to develop ASICs were moving to FPGAs since their designs were now able to fit in the reprogrammable logic, saving resources and time when developing new ideas. DSPs (Digital Signal Processing blocks) provided pre-built modules that were faster than synthesized ones, providing huge performance boosts. But having such huge FPGAs for small ASICs was a problem since power consumption and cost would be too prohibitive for some applications. Thus, FPGA vendors started launching multiple devices of varying capacity to address the needs of low and high complexity designs (Table 1.2).

But sometimes FPGAs fell short when doing certain tasks. Enough logic resources were now available for almost any function, but sometimes it just wasn't enough to meet timing or power constraints. Thus, resources started being used to integrate "hardwired" functionality. Microprocessors, multipliers, Ethernet and PCI express interfaces or floating point arithmetic are just some of the functions that are now pre-built on FPGAs. With no reconfigurability, they operate just as fast as an ASIC. For any added functionality, the programmable fabric is still there and can interact with all these modules.

Today, FPGAs have so many resources that they are being used to accelerate all kinds of tasks. High level synthesis has allowed pure declarative code to be converted into circuits, saving the pain of learning how to use FPGA-specific programming environments. However, the highest performance is still only reached by hand-optimized code. Any algorithm can now benefit from their custom capabilities since they are present all the way from cloud services to edge devices. Their present certainly seems bright, and their future looks brighter.

1.3.2 FPGA structure

FPGAs have been through an evolution that goes beyond exponential growth. But it is not only a growth in resources, it is also a growth in the type of resources available. Instead of just increasing CLBs, those have increased as well in size and functionality (Figure 1.11). New pre-built blocks have been added to the FPGA fabric to accelerate common functionality beyond what's possible with reprogrammable logic. I/O capacity has increased to accommodate new types of memory, ports and interconnections, allowing for more and more functions to be built every day.

Model	CLBs	RAM	I/O	FF	DSPs	Year	Adds
XC2064	64		58	122		1985	Gates, registers, routing
XC3195A	484		176	1320		1988	Three state bus
XC4085XL	3136	100kb	448	7168		1991	Carry logic, memory
XCV3200E	16224	852kb	804	34860		1999	Dynamic reconfiguration
XC2VP125	14416	10Mb	1200	111k		2002	Transceivers, processors
XC4VFX140	16128	10Mb	896	126k	192	2005	DSPs, Ethernet, PCIe
XC5VLX330	25920	10Mb	1200	207k	192	2006	
XC6VLX760	59280	25Mb	1200	948k	864	2009	System monitor
XC7VX1140T	89000	67Mb	1100	1424k	3360	2010	ADC
VU13P	216000	454Mb	832	3456k	12288	2016	

Table 1.2: Xilinx's FPGA models through time

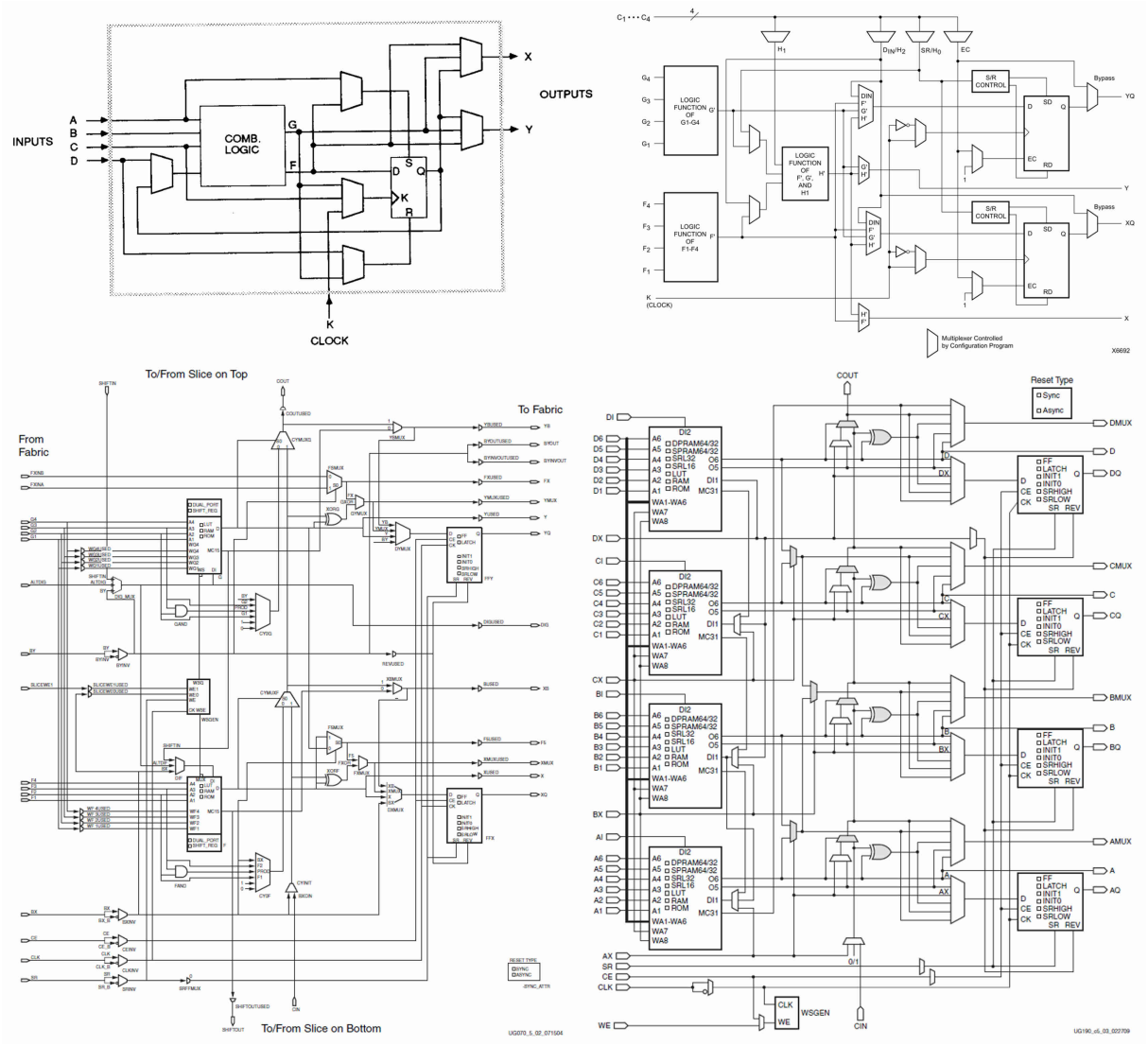

Figure 1.11: CLB diagrams for different FPGA models. In reading order: XC2000 series [227], XC4000 series [229], Virtex 4 series [224] and Virtex 5 series [225].

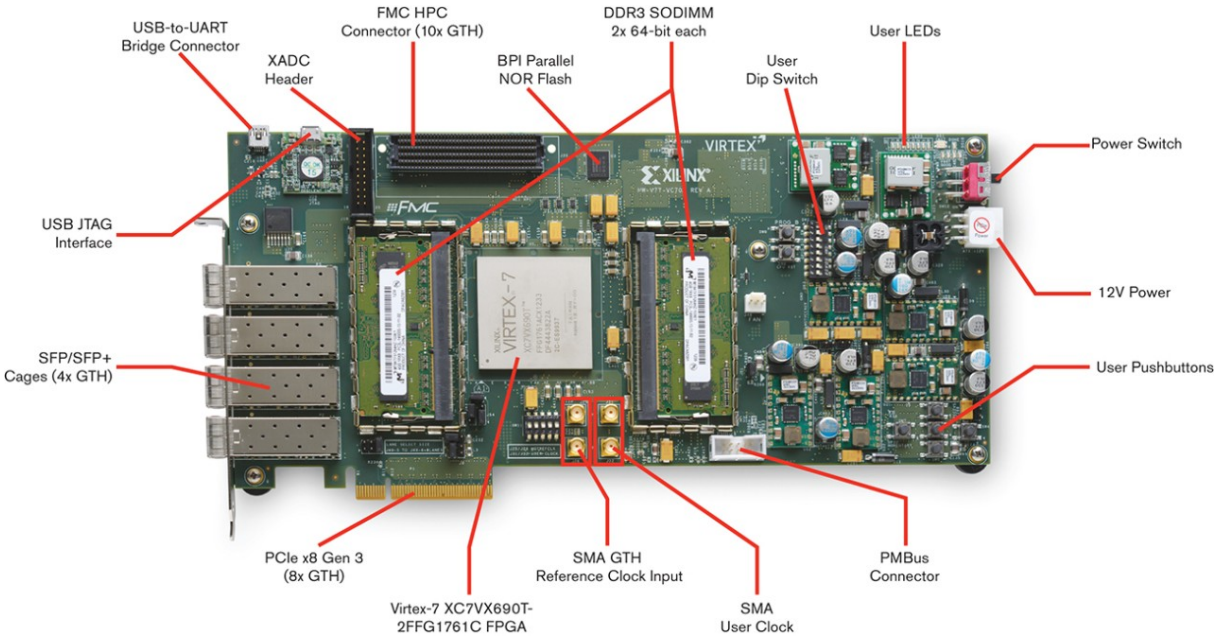

Figure 1.12: Annotated picture of a VC709 connectivity board for the Virtex-7 [232].

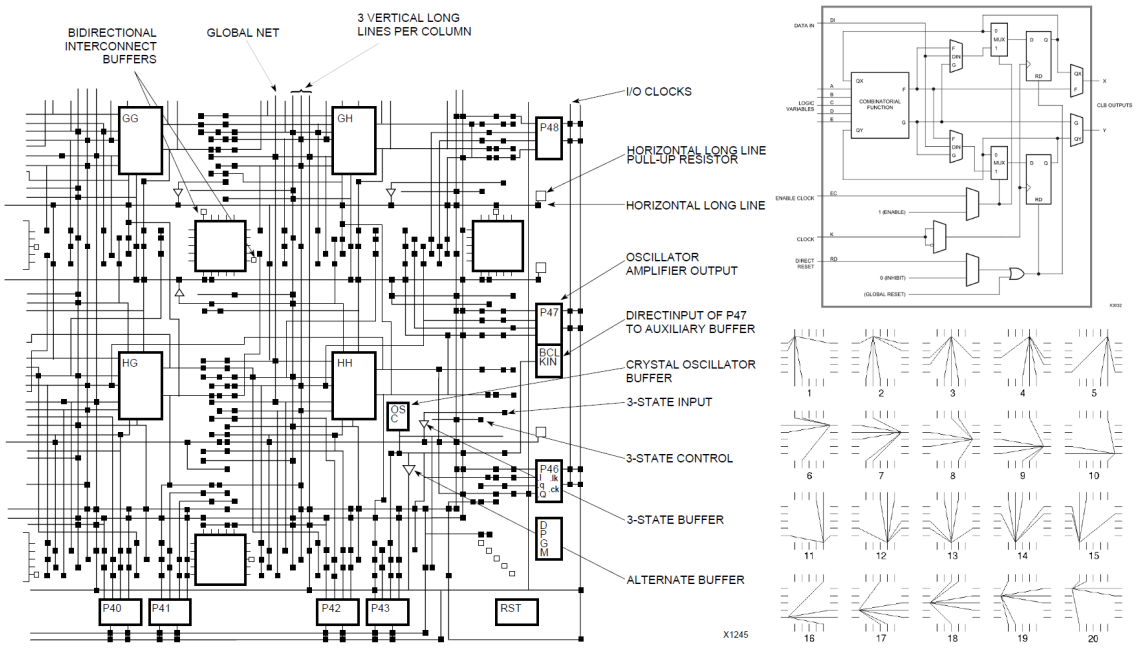

Figure 1.13: Detail of the interconnections present in a XC3032 [228]. Left: fabric interconnections. Top right: CLB structure. Bottom right: Switch matrix pin configurations.

A modern FPGA will be shown as an example: The Virtex-7. This is chosen since it's one of the models used to test the designs in this work. It is packaged in a test board for ease of use, as seen in Figure 1.12

These cards connect to the computer either via USB or the PCI express port. As seen, they have multiple connections to allow for a wide range of applications to take advantage of the FPGA. In this case, it is accompanied by 8GB of Ram, in two separate 4GB banks. Applications will usually load the memory with the data to be processed, accelerate that processing on the FPGA itself, and then transfer the data back to the computer. This concept of FPGAs as general purpose accelerators has been in use for cloud services [10] such as AWS for a couple of years.

Aside from connections or applications, the interesting part about a modern FPGA is the inside of the chip itself. For that, it is necessary to go back to an old FPGA, the XC3032 [228], where configurable blocks were simple enough to fit in one page. Figure 1.13 shows the internal structure of the FPGA. CLBs are the blocks named by two letters. Every black dot is a configurable switch that can connect or disconnect the lines it touches. CLB functionality can be configured for each CLB. Switch matrices can be configured per-pin. So, even for a small 64-CLB FPGA such as this one, the number of reprogrammable bits is enormous. Direct lines allow for fast connection between elements, while the switching matrices have greater flexibility but are slower.

Nowadays, each CLB has thousands of configurable bits, while switching matrices have hundreds of pins. Furthermore, FPGAs contain specific blocks such as DSPs that connect to the fabric as well, providing built-in functions like multiplication or division. Those have less flexibility but are reprogrammable as well, usually allowing trade offs between clock frequency and latency.

1.3.3 FPGA applications

FPGAs have seen a number of useful applications [197, 205]. Their reprogrammability has made FPGAs the perfect vehicle to carry circuits or accelerators that are be too costly or timely to

implement as ASICs. They allow for faster testing, turnout and provide high performance all at the same time. At first, that was the target market segment. With the growing size of ASIC design, small production runs became too costly to develop and were replaced by FPGA designs, which generally perform sufficiently fast.

Due to their capabilities, and the quick expansion of the networking market, that was at one point the biggest selling target for FPGAs, with reprogrammable switches [134] being cheap, fast, and ideal for new emerging protocols or updates that could be done at the hardware level. Instead of having to update the software of an ASIC or CPU which potentially would not be as fast for the new requirements, the custom implementation of the protocol could be loaded into the FPGA. Many I/O improvements were brought to FPGAs, which were catering the telecommunications segment but still remain today.

Specific control systems, where simplicity and reliability are key, have also been implemented on FPGAs due to the prohibitive cost of ASIC design. Small robots for civilian search and rescue [39], or semi-automated medical surgery equipment [200] have trusted FPGAs as their brain power. Control systems based on fuzzy logic [198] have also been targeted instead of using a specific ASIC because production runs would be very small, and processors require great amounts of power for the little computing capacity that is needed. FPGAs fit perfectly allowing for faster responses in those systems.

Over time, ASICs recovered part of their market share by expanding to system-on-chip (SoC) devices. Generic processors were combined with application specific blocks that would target specific functionalities. Programmable SoCs [49] were developed as an answer, including processors, networking processing blocks, Analog to Digital Converters, or multipliers and FP arithmetic for precise mathematical applications.

FPGAs are capable of performing any function, including running full-fledged operating systems [192] making FPGA resources available at the kernel level, improving performance and allowing seamless integration of the FPGA fabric for all processes. Despite these capabilities, complexity is still high, and FPGAs today are often used for increased performance where custom hardware is not available.

In neural networks [85] they can replicate the actual network in hardware adapting to their changes, being much faster than CPU or GPU [131] based systems. In an FPGA, the whole network is working simultaneously. However in a CPU or GPU different iterations are executed one after another, slowing down processing. Common applications based on neural networks on FPGAs [153] include speech recognition, feature extraction, image association, robot vision, face recognition, fingerprint matching, color quantization, and many more.

Video encoding performs a sequence of costly operations over successive frames in order to estimate motion and save bandwidth by using previously seen blocks as references. These complex operations have been accelerated in FPGAs [59, 122]. Decoding procedures have also been fully developed on FPGAs for the latest and more complex 4K encoding standard HEVC [2] in real time.

Cloud computing servers have seen the introduction of FPGAs as accelerators [65], which are far more efficient power-wise than multi-core CPUs or GPUs, and can also adapt to changing algorithms. Even some older FPGA characteristics such as bitstream encryption have been used recently [234] to ensure data privacy in FPGA cloud applications.

Low power applications have also been of particular interest [208] since FPGAs are able to preserve battery life longer, enabling off-the-grid applications or data collection systems that last longer than if using common SoCs. Even specific improvements have been researched [77, 172] to further decrease FPGA power draw by using different voltage domains or clock gating.

A place where FPGAs are of great use is satellite applications. It makes sense since satellites cannot be accessed for repairs. If a custom ASIC failed, or a processor had a cosmic ray alter

some hardware, that failure would be there indefinitely. It might be avoided by software in some cases, but repairs are not possible. If instead FPGAs are used, any defect on the synthesized logic, either from programming or from cosmic events, can be fixed by rerouting resources. New optimizations can be incorporated. Functionality can adapt to different needs, accelerating different processing flows if needed by reconfiguring when necessary. Communication links [90] and controllers [89] are two of the main applications that benefit from FPGA use, along with hyperspectral image compression.

1.3.3.1 Compression of hyperspectral images on FPGAs

FPGAs offer great advantages when flexibility is important. Space is one place where being able to reprogram a circuit can be extremely useful since repairs are impossible. Many hyperspectral sensors are space-borne, and hyperspectral data has sizes that quickly add up to fill up a satellite's memory. Compression is a must in that scenario, not only for storage but also for being able to send data back to ground stations in a more efficient manner. The cycle closes by realizing that compression, due to having many but simple operations, can be efficiently carried out on FPGAs [161, Ch.7][175].

FPGAs have already been used to implement hyperspectral compression algorithms. Lossless algorithms such as CCSDS 123.0-B-1 [106, 154, 180, 207] and even its predecessor the "fast lossless" coder [12, 115] have seen many implementations. Integration on real-time reconfigurable platforms [168] has also been explored, allowing for multi-core processing. Lossy ones such as the wavelet-based SPIHT [70, 95] have already been around for a while, as well as other methods such as vector quantization [161]. Other algorithms like JPEG2000 have also seen many implementations [84] that can also apply in the context of hyperspectral lossy compression. On the near-lossless side, the newer CCSDS 123.0-B-2 [52] already has an implementation, as well as the low complexity coder for ExoMars [75, 76].

The topic of compressing hyperspectral data on FPGAs is of interest, specially in satellites which most of these works apply to. But despite the plethora of options, there has been no effort to unify and experiment with all types of algorithms. Which one is suited for what application? Can all kinds of algorithms be space-borne? Is there a better compression algorithm under certain metrics? What are the trade-offs between all types?

These, as well as more questions, are the ones that this thesis tries to answer.

1.4 Objectives

The general objective of this thesis is to study hyperspectral image compression algorithms using reconfigurable hardware. The usual problems that arise in this context, like high data dimensionality, data quantity, processing and transmission times are to be taken into account. The approach to solving this problem is to study the Field Programmable Gate Array (FPGA) implementation of some of the most significant hyperspectral compression algorithms in the literature.

First, extensive research in compression is required to understand the main types of approaches to general data compression. Their adaptations to hyperspectral images are also of interest, and specially algorithms developed exclusively for this kind of data. Existing implementations of these algorithms on FPGAs will also provide with the required knowledge in novel techniques that can be used for this thesis.

After this first analysis of hyperspectral image compression in the context of FPGAs, three algorithms have been selected to be implemented and analyzed. Their differences are to be studied, understanding what the advantages of the different types of techniques are, and to

derive when and where it is necessary to apply each technique. Each implementation aims to achieve real-time performance as a baseline, to be able to compare each algorithm in an applicable scenario where images are compressed after capture in a possible real-world scenario.

- An in-depth analysis of the lossless CCSDS 123.0-B-1 standard [32] will be performed in order to understand the mathematics behind it. Being an international standard, it has gathered attention, and a review of existing FPGA implementations will bring ideas to the table and present their shortcomings in order to develop a novel implementation that solves them.

  Being highly configurable, an extensive review of occupancy impact is to be done, determining the ideal options to set in order to achieve the desired performance results.

- The JPEG2000 image compression algorithm [163] will be studied next. Its applicability is well-known for digital images, but the focus will be on hyperspectral adaptation. In this context, the combination with other algorithms will also be of interest to fully exploit the redundancies present in hyperspectral data for compression. This method will target an aggressive lossy compression, to see the feasibility of an FPGA implementation.

  An extensive software analysis will be made beforehand in order to look for the best possible pipeline that, including JPEG2000, is able to compress hyperspectral image. The pipeline will be optimized at all steps in order to bring this complex array of algorithms to real-time constraints.

- The low complexity predictive lossy compression (LCPLC) algorithm [5] is the final algorithm to be analyzed. It offers the ability of performing near-lossless compression, providing a quality threshold that is maintained in the output image. It offers less advantages than the two other approaches in their respective domains, while at the same time being competitive in both.

  Hardware optimizations will again be targeted to analyze the algorithm under tight timing and resource constraints, seeing as well how it compares in both compression performance and processing speed against the other two.

In the following chapters a more in-depth introduction to compression is seen, which is the main theoretical background that the rest builds upon. After studying compression techniques, they will be applied to three algorithms in the lossy, lossless and near-lossless category. These three are algorithms that have already been tested, and modifications are proposed that improve their performance in certain key aspects relative to prior implementations. Afterwards, the FPGA implementations will be presented, as well as the key aspects that this work focused on during development.

Results, showing compression ratios, quality, algorithm speed as well as FPGA occupancy are shown afterwards, also comparing with existing implementations of similar algorithms. Finally, conclusions will be drawn as to which algorithms are better suited for certain situations. The thesis will end with future work ideas, hinting at what might be possible to achieve based on the experienced gained while writing it. The final objective is to provide the reader with extensive data and experimental results to be able to decide which algorithm and implementation techniques are best suited for each application.

Chapter 2

Compression

It was 1948 when Shannon [184] set the mathematical background for information theory. In the first paragraph, he wrote:

"The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point".

That phrase alone establishes that what's key for transmitting information is the reproducibility of the message. It doesn't mention how the information is transmitted, just how it must arrive at its destination. Any kind of digital system that relies on communication with others usually has its bottlenecks at the communication level. Internally, operations are orders of magnitude faster: memory is available nearby and connections are extremely short compared to the long distances information has to travel between two separate systems.

Being the bottleneck, speeding it up is key to enable faster operation. There is a physical constraint on any communication link: signals cannot travel faster than light, and the bandwidth available depends on technology. So, with links that have certain imposed limitations, how can those parameters be improved? The answer is compression.

In this chapter, a broad look is taken at different compression techniques, that are applicable to many kinds of data, including hyperspectral. Special focus is put into the ones that are used or influence the algorithms implemented in this thesis. An in-depth look at those is provided in Chapter 3.

2.1 Basic concepts

Compression is the process through which some piece of information gets processed in order to reduce the amount of bits it takes to represent it, while still retaining the original meaning. With the first algorithms, the original data had to be able to be perfectly retrieved. The size was reduced but not the amount of information. This is lossless compression.

But not all types of data require exact reconstruction. Some signals can be compressed with a loss in quality while still retaining the important features. This is lossy compression. Going back to Shannon's quote, the message is reproduced either *exactly* or *approximately*.

When compressing losslessly, there will be a plethora of techniques to achieve the same result: reduce bit size. To establish which one is preferable, sizes of compressed data are compared. An algorithm which compresses down to less bits is more desirable than one that uses more bits to represent the compressed information. To measure this effect two different concepts are used:

- The **compression ratio** r. For a dataset D, compressed with an algorithm a, define r as:

$$r(a, D) = \frac{s(D)}{s(a(D))} \tag{2.1}$$

  where $s(D)$ is the size in bits of the dataset. For example, a compression ratio of 2 means the compressed dataset is twice as small as the original. A bigger ratio means compression is higher.

  The **inverse compression ratio**, which indicate the fraction of the original size that the compressed data takes, is defined as $ir = 1/r$.

- Compression can also be measured in **bits per pixel per band** (*bpppb*). This is specific to hyperspectral images. For an image I with bit-depth $b(I)$ (i.e: number of bits per sample) that is compressed with an algorithm a:

$$bpppb(a, I) = \frac{b(I)}{r(a, I)} = b(I)ir(a, I) \tag{2.2}$$

  In this case, lower *bpppb* values indicate higher compression. *bpppb* is inversely proportional to compression ratio. For this thesis, the factor $b(I)$ is assumed to be 16 unless stated otherwise.

For lossless algorithms, these concepts are enough, since comparing just output size will determine which algorithm achieves higher compression. (Though other factors such as processing time might be of interest as well). However for lossy algorithms it is of interest to also know the quality of the result. Metrics for determining the quality of compression are explained in Section 2.3.6. To compare lossy algorithms, both compression ratio and compression quality will be compared at the same time.

When fixing a ratio, the algorithm which yields a higher quality is considered to be better. And when fixing a quality, the algorithm which achieves it at a higher compression ratio is preferred. The curves that arise from plotting these values are called **distortion-ratio** curves. An algorithm a is better than a' if a presents higher quality then a' at the same ratio or higher ratio at the same quality.

Curves can cross each other, and sometimes an algorithm will outperform others for only some portions of the curve. Figure 2.1 shows this effect. The *bpppb* representation is more understandable since quicker growth means higher quality at a higher ratio. The *cr* representation gives an idea of how much compression can be achieved at a given quality.

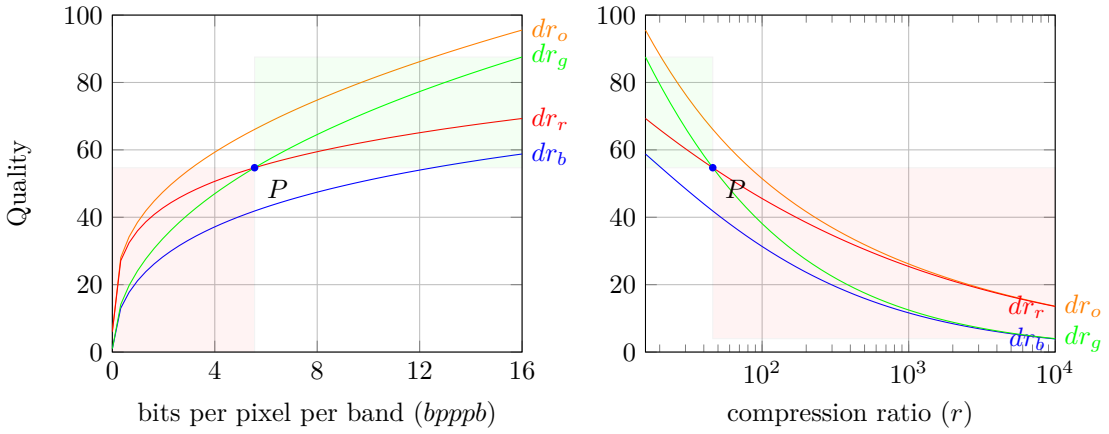

Figure 2.1: Distortion-ratio curves with both *cr* and *bpppb* shown. dr_o is the curve with highest distortion-ratio performance, while dr_b is the one with the lowest. dr_g and dr_r cross each other, and are respectively better in their corresponding shaded areas.

2.2 Lossless compression

Lossless compression is used when it is important to preserve original data exactly as it was obtained, with no approximations. Lossless compression is usually also faster to perform than lossy, so certain applications take advantage of this speedup to meet real-time constraints. Furthermore, lossy algorithms generally contain lossless coding after the lossy transformations.

The mathematical background comes from Shannon's Information Theory [184]. Back then, messages consisted of symbols transmitted over the communication link. If there is previous knowledge about the symbol source (how many symbols as well as their frequencies), messages can be sent using less data than if a random source was assumed.

A symbol is any object that carries information (such as a letter or digit). Symbols usually combine to form higher order objects (letters become words and digits become numbers). Those objects can also be symbols themselves. The set of symbols that are used is called the alphabet. Alphabets can be composed out of few simple symbols or out of many complex symbols. There will be trade offs between number of symbols in an alphabet and the number of bits that are needed to represent their symbols.

To transmit symbols over a communication channel, a mapping to an alphabet that the channel supports is needed. The channels are binary and transmit bits, that is, zeros and ones. Combinations of those will be used to map the symbols:

Definition 1 *A **code** is a mapping from symbols of an alphabet A to symbols of an alphabet B, that is the one supported by the communication channel. (In this case, binary words). As an example, the Latin alphabet can be mapped to five-bit symbols as follows:*

$$C = \{a \to \texttt{00000}, b \to \texttt{00001}, \ldots, z \to \texttt{11000}\} \tag{2.3}$$

In this case, the code has a **length** of five. For fixed length codes, each symbol begins at a boundary that is a multiple of the code length. But codes can be of *variable length* if every symbol can be distinguished by its prefix.

Consider a random variable X that generates symbols $x_i, i \in [1, n]$ from an alphabet A. Those symbols can be coded with a fixed length of $L(X) = \lceil \log_2 (|X|) \rceil$, having at least as many possible combinations as symbols there are in A.

Sometimes, the probabilities of appearance of certain symbols $P(x)$ will be skewed. For example, if A is the Latin alphabet and a book is being coded, vowels will be much more present than, for example, the letters q, j or z. In those cases, using a variable length code that assigns shorter lengths to more common symbols is advantageous, since less bits will be used overall to transmit the same message. The average length with which symbols of X can be transmitted has a theoretical limit known as Shannon's entropy $H(X)$:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_2 (P(x_i)) \tag{2.4}$$

where the logarithm in base 2 indicates that the alphabet that codes the symbols uses binary digits to do so.

$H(X) \leq L(X)$ always holds, so a trivial constant length mapping can usually be improved with a variable-length coding scheme. A simple example can be found in Shannon's work where symbols A, B, C, D are assigned probabilities of $1/2, 1/4, 1/8, 1/8$ respectively. In this case $L(X)$ is 2, however:

$$H(X) = -\left(\frac{1}{2}\log_2\left(\frac{1}{2}\right) + \frac{1}{4}\log_2\left(\frac{1}{4}\right) + 2\frac{1}{8}\log_2\left(\frac{1}{8}\right)\right) = \frac{7}{4} \tag{2.5}$$

So if a two bit per symbol code is used, part of the sent bits that are redundant are wasted. Using the following variable-length code: $A \rightarrow \texttt{0}$, $B =\!\!\rightarrow \texttt{10}$, $C =\!\!\rightarrow \texttt{110}$ and $D =\!\!\rightarrow \texttt{111}$, the theoretical limit is reached since, for N symbols sent:

$$bits(N) = N\left(\frac{1}{2}\cdot 1 + \frac{1}{4}\cdot 2 + \frac{2}{8}\cdot 3\right) = \frac{7}{4}\cdot N \tag{2.6}$$

This theoretical limit cannot always be reached. A simple example can be found in two-symbol alphabets. The best code is to assign **0** to one symbol and **1** to the other. So the same amount of bits and symbols will always be used, but the entropy can be arbitrarily close to zero when the probability of one of the symbols $p = P(x_1)$ tends to 1:

$$\lim_{p\to 1}\left(H(X)\right) = \lim_{p\to 1} -\left(p\log_2\left(p\right) + \left(1-p\right)\log_2\left(1-p\right)\right) = 0 \tag{2.7}$$

In any case, the code that is assigned needs to be decoded in a unique way, otherwise the message is completely lost in transmission. To solve this issue **prefix codes** are used, in which no symbol's representation is a prefix of a different symbol's representation, so that when decoding variable length codes, symbol's boundaries can be uniquely determined.

2.2.1 Huffman coding

Shannon had established a limit, known as entropy, below which messages could not be further compressed. That limit was not always reachable.

Huffman was one of the first to create an algorithm [96] that created the optimal mapping between the input alphabet and the output alphabet with the least average bits per symbol, according to Shannon's theory.

Symbols are sorted by descending probability of appearance. The two rarest symbols are grouped together and their probabilities added. They form a new symbol that gets added to the original pool of symbols. This process continues iteratively creating a tree of symbols where low probability symbols appear at very deep leaves, while high probability symbols are near the root.

To assign codes, the tree is traversed and each branch assigned either **0** or **1**. The code for each symbol is that of grouping together the bits that appear on the branches leading to them. Figure 2.2

2.2.2 Golomb coding

Huffman coding works on a finite set of symbols with known probabilities. But what if the set of symbols to be coded was infinite? For this, symbols need to be generated on-the-fly that can be decoded later.

Golomb coding [79] was originally designed for coding run-lengths of random symbols a and b such that the probability of one is much higher than the other. Runs $a^n b$ were coded, where a has the higher probability. This is the same as saying that numbers $n_i \in [0, \ldots, \infty)$, that follow a geometric distribution, are coded.

If p is the probability of symbol a, select m such that $p^m = 1/2$. If $p^m = 1/2$, then a run length of n symbols is double as likely to appear as a run of $n+m$ symbols. It makes sense then for the second run to take up 1 more bit.

By this principle, Golomb codes are created. Let k be the smallest integer satisfying $2^k \geq 2m$. Specifically, focus on the case where the equality is met (Golomb power of two coding). The

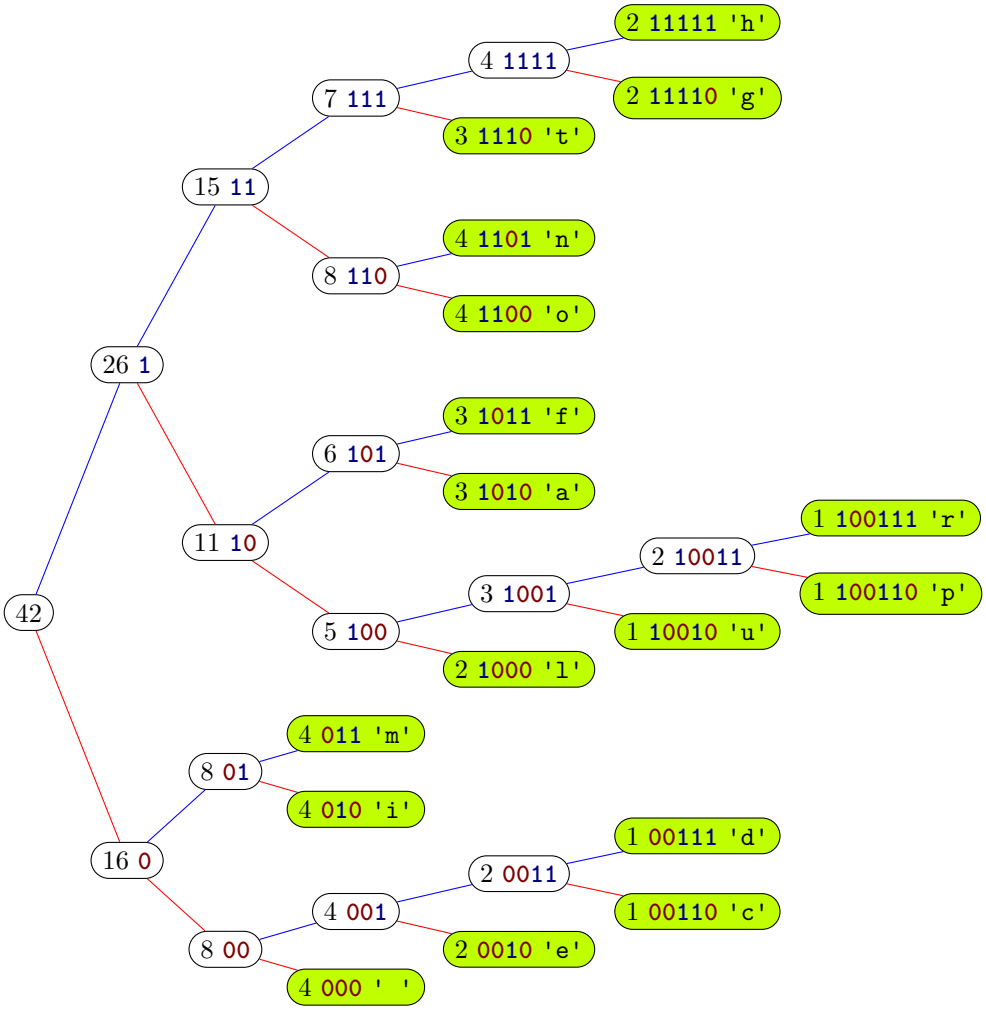

Figure 2.2: Huffman tree for the phrase "Implementation of Huffman Coding algorithm". Leaves indicate the codes used for each symbol. Numbers on each node indicate the aggregate of appearances of all the symbols that branch from them.

corresponding code then has m words of every length $\geq k$, which are bins. On each bin, the probability of appearance is halved. If value n is to be coded, it is first expressed as:

$$n = q * m + r \tag{2.8}$$

where q and r are the quotient and remainder of the division by m. The resulting code will be q "ones", followed by a zero, followed by the remainder r in $m - 1$-bit format. An example is shown in Table 2.1.

This way of coding is not only useful for runs of symbols, but also for numbers that follow a geometrical distribution. In that case, this is the best way of coding without knowing beforehand what the maximum value will be.

Number	0	1	2	3	4	5	6	7
$m = 1$	0	10	110	1110	11110	111110	1111110	11111110
$m = 2$	00	01	100	101	1100	1101	11100	11101
$m = 3$	000	001	010	011	1000	1001	1010	1011

Table 2.1: Golomb coding. A tradeoff is made between shorter codes at the beginning with longer codes following, or longer codes at the beginning which do not grow as fast.

Number	0	1	2	3	4	5	6	7
$k=0$	0	010	011	00100	00101	00110	00111	0001000
$k=1$	10	11	0100	0101	0110	0111	001000	001001
$k=2$	100	101	110	111	01000	01001	01010	01011

Table 2.2: Exponential Golomb coding.

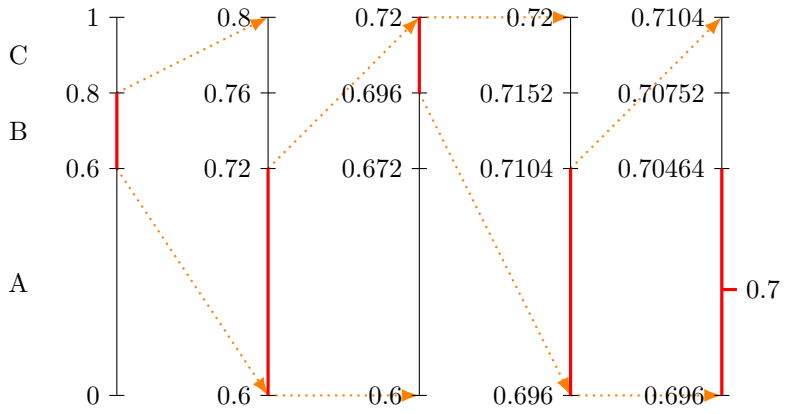

Figure 2.3: Arithmetic coding of BACAA assuming probabilities $P(A) = 0.6$; $P(B) = 0.2$; $P(C) = 0.2$. The result is the number of subdivisions (5) along with a representative (e.g: 0.7).

2.2.2.1 Exponential Golomb coding

In the same way that a geometric distribution is used for Golomb-coding, when the data presents an exponential behavior (with lower values being exponentially more likely to appear), exponential Golomb coding [60] can be used.

To code a value v, it is first normalized to a nonzero value by adding 1 $v' = v + 1$. Then, its magnitude $m = \lceil \log_2(v') \rceil$ is obtained with $m - 1$ zeros coded, followed by the binary value (in m bits) of v'. This is the zero-order exponential Golomb coder. For higher order-coders, where smaller numbers use more bits but bigger numbers are more compressed, a different technique is used. First, $\lfloor x/2^k \rfloor$ is coded using the zero-order exponential Golomb coder, and then $x \bmod 2^k$ is coded in k bits in binary. This is a kth order exp-Golomb coder. An example is seen in Table 2.2

2.2.3 Arithmetic coding

There were other codings different than Huffman's or Golomb's, but assigning codes to each symbol always resulted in some loss of efficiency since no symbol could be represented with less than 1 bit, steering away from Shannon's entropy limit.

A technique was developed [3, 26] that is be able to reach that limit by assigning non-integer lengths to different symbols.

The way of doing this is not trivial. How can a symbol be transmitted by using, for example, 4.32 bits? What arithmetic coding does is different than the previous approach. Based on the symbols it has to send, arithmetic coding will create a single "word" containing all of the symbols together.

In arithmetic coding, a message is represented by an interval $[c_n, c_n + a_n) \subseteq [0, 1)$. From the interval, a representative $p \in [c_n, c_n + a_n)$ is taken, which codes the message. The longer the message, the shorter the interval will be, and the more bits will be needed to code the representative p. The symbols in the message successively reduce the interval's length (Figure 2.3)

based on their probabilities. Symbols that are more likely to appear reduce the interval slightly, while less likely symbols generate more drastic reductions.

To update the interval, a probability function is used that indicates the likelihood of a symbol x_i appearing in the message $f_X(x_i)$. A cumulative function $F_X(x_i) = \sum_{j=0}^{i-1} f_X(x_j)$ is also used, indicating the sum of probabilities of all previous symbols. If $X = \{x_0, \ldots, x_n\}$, then $F_X(X) = 1$.

Updating the interval is thus done as:

$$a_{n+1} \leftarrow a_n f_X(x_n) \tag{2.9}$$
$$c_{n+1} \leftarrow c_n + a_n F_X(x_n) \tag{2.10}$$

Any value $p \in [c_n, c_n + a_n)$ is valid as a representative of the coded message. To decode, subdivisions are followed based on the probability functions described before, placing the value p down the subdivisions. A slight overhead is introduced to store the number of subdivisions expected, since otherwise the process could create an infinite message. Even with this slight overhead, as messages grow in length, the number of bits needed to represent them gets arbitrarily close to Shannon's limit.

The limitation was that, to update the intervals, arbitrary precision floating point numbers were needed. Theoretically arithmetic coding was a good idea, just not practicable. This was until a full implementation [217] was published that could perform arithmetic coding with no need of arbitrary precision:

- The implementation deals with integers to simplify the subdivision process, so $f_X(x_i)$ is now approximated as:

$$f_X(x_i) \approx p'_{x_i} = \lfloor 2^P p_{x_i} \rfloor \tag{2.11}$$

  where care is taken not to round anything to zero.

- Registers for a and c must be of finite length, so their size is limited to N and $N + P$ bits respectively.

The output bitstream can be decoded even if only a partial fraction is available at any time. N and P need not be high for an acceptable result, with $N + P < 32$ giving results close to theoretical limits that are achieved with arbitrary precision [201, p. 2.3.1].

2.2.4 Binary arithmetic coding

The idea of an arithmetic coder can be further simplified by demonstrating that any coder is equivalent to a binary one.

Let A_X be the input alphabet ($x_i \in A_X$) with cardinality $|A_X| = 2^K, K \in \mathbb{N}$. Each element can be represented as a K bit integer.

X is a random variable that generates 2^K possible outputs, but can also be interpreted as a vector of K binary random variables $B_0, \ldots, B_{K-1}$, where each B is a binary digit of X.

In the general case each symbol x_i is coded based on its probability p_{x_i}, now K bits are coded instead:

- For the first bit there is only one probability distribution, either the symbol starts by `0` or `1`.

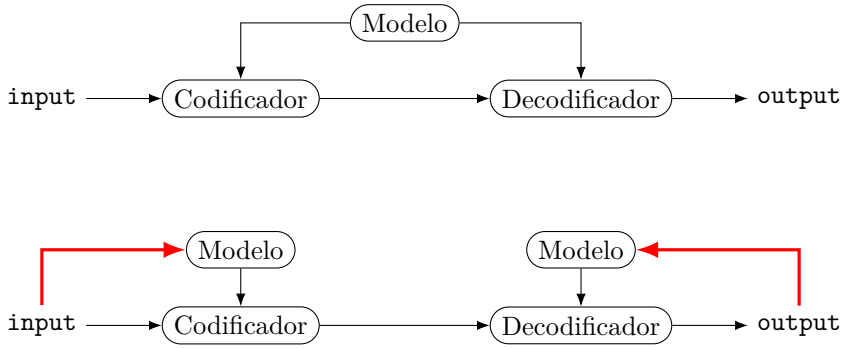

Figure 2.4: First diagram shows a non-adaptive coder. The model is static and shared. The second diagram shows an adaptive coder in which the model is updated from the stream of symbols. This ensures the model is synchronized.

- For the second bit there are two probability distributions, one gives the probability of a `0` or `1` when the first bit was `0` and the other gives the probability when the first bit is `1`.
- So for bit i there are 2^{K-1} distributions. Adding all together a total of $2^K - 1$ distributions arise. Since they are binary, each is defined by a single probability, for a total of $2^K - 1$ probabilities, the same that were needed when keeping track of the original 2^K symbols.

An advantage of binary arithmetic coders is that usually the distributions referring to the lower bits are mostly uniform, so p can be set to $p = 0.5$ for them. Doing this will provide a similar performance as the classic arithmetic coder, having less information to keep track of.

2.2.5 Adaptive entropy coder

Both Huffman and arithmetic coding are entropy coding algorithms. They use the (previously known) distribution probability of the input symbols with the aim of coding them with results as close as possible to Shannon's theoretical limits.

Sometimes the distribution changes over time, or can be adapted for local statistics. As an example, `000001111110` has an entropy of 1 bit because both symbols (`0` and `1`) appear the same number of times. However, splitting it in two yields `000001` and `111110`, both with entropy 0.65 that can be coded at higher efficiencies.

To build an adaptive coder the probability model is changed to update with the stream of symbols instead of being static (see Figure 2.4). When coding a symbol x_j, the model will only depend on symbols $x_i, i < j$. As an example, in Huffman coding, the codes for symbols could change every so often to adapt to local statistics. The decoder will also know the state at that point and will be able to adapt.

This allows us to break the entropy limit, since that was set for static codes. Information can be compressed even further in a lossless way.

2.2.6 Entropy reduction

All of the coding techniques so far base their efficiency in the entropy (either local or global) of the input data. Based on mathematical formulas, their efficiency is limited and cannot go over certain thresholds.

However, certain symbol sequences hold great correlation between consecutive symbols. In those cases, even if the entropy when looking at individual symbols is high, decorrelating them can lead to a great reduction of entropy.

2.2.6.1 Differential coding

Differential coding works best in temporal series of numbers. These series are usually fairly smooth, and consecutive samples are usually close together in value. A simple series $x = t, t \in [0, 99] \subset \mathbb{N}$ will have the maximum entropy possible since all values in the series are different.

However each value is close to their neighbors. If instead of coding the values $0, \ldots, 99$ their *differences* are coded, things will be much easier. In this case, the first value and then the differences after it are stored, using the vector $0, 1, 1, \ldots, 1$. That distribution has an extremely low entropy and an arithmetic coder is able to code it efficiently.

In general, for any kind of temporal series, the values obtained from the differences of subsequent values will be much lower than the original values. If the same number of samples is present but with a lower possible range of values, more repetitions will be present and thus the entropy will be lower.

2.2.6.2 Predictive models

A more generalized idea of differential coding comes from the concept of predictive models.

At any point in coding, when a sample is processed, a prediction is made based on the previously seen values. For smooth and predictable data, this value will be close to the actual value. For example, in the previous case where $x = t$, if the model discovers the generative function, the predictions will always match the actual values.

The differences between the actual and predicted values will be coded. The model approximates the data, yielding differences with the real values that are smaller than the values themselves, and can be more efficiently coded using differential coding.

The good thing about predictive models is that they can adapt to any kind of data, not only temporal. For example, images have great spatial correlation, which is found in two different directions at once: horizontal and vertical. When processing an image in raster scan order, the already processed neighborhood can be used to create a prediction.

Any kind of entropy coder will be able to feed off the predictions and outperform itself if it was given the raw data, since good predictions will have lower entropy than raw data.

2.2.7 Run-length coding

Run-length coding aims to reduce the size of long repetitive sequences of symbols. Run-length coding starts with a sequence of symbols. To compress it, runs of consecutive symbols are replaced by a pair of a number and a symbol. The number indicates the amount of times that the symbol is repeated. As an example:

$$aaaaaabbccccaaaa \rightarrow 6a2b4c4a \tag{2.12}$$

Sequences with many repetitions will be greatly reduced, while more random sequences will probably be expanded by this method, due to the overhead of including the number of repetitions along with each symbol.

Some algorithms use run-length coding to perform compression when they detect fairly uniform subsequences within the input data. Those regions will be compressed in a run-length way while other regions may use different techniques.

2.2.8 Dictionary coding

Dictionary coding exploits the fact that sometimes symbols group together in common constructs. Symbols in text are characters, however they are always grouped in words. If, instead of coding characters, words are coded, coding efficiency will be higher.

Continuing with the previous example, there are 26 letters in the English alphabet. On average, 5 bits are approximately neded to code each one. This can be improved by using Huffman coding, for example, with the average frequencies of letters in English.

In this case, letters can be grouped into words. The average word length in English is 4.7 characters. That means that, with no consideration for correlation between letters, each word has an entropy of approximately $\log_2(26) \cdot 4.7 \approx 22.09$ bits. However, with 22 bits, $2^{22} \approx 4$ million different combinations exist.

But there are much less words in English. Shakespeare, a reference in writing, used approximately 31534 different words in his books. Those can all be represented by using just 16 bits. So why use 22 bits per word when just 16 suffice? This is exactly what dictionary coding does. In the case of words, a dictionary maps them to the string of symbols (characters) they originate from. Binary codes are assigned to the words that appear on the dictionary, saving combinations.

This technique is useful when the number of words in the dictionary is significantly lower than the number of combinations that can arise from the combination of the original symbols (in this case characters). For natural language, a dictionary approach will almost always result in savings when compressing.

2.3 Lossy compression

In contrast to lossless compression, lossy compression allows some sort of distortion to be introduced in the decompressed data. This will often be applied over continuous and fairly smooth data (audio, images, video...) which, even if slightly approximated, retains the original meaning.

Lossy compression is important because lossless techniques have certain limitations. Even if Shannon's entropy barrier can be broken with certain techniques, these apply in specific scenarios and for certain sets of data they will not be efficient. If there are certain restrictions about data size, transmission speed or others, lossless compression ratios might not be low enough.

Generally, lossy compression will only be applied when its effects do not significantly affect data analysis afterwards. If data is rendered unusable, then compression is useless in the first place. Different techniques will apply to different scenarios, and often a mix of them will be used to properly manipulate the data.

2.3.1 Quantization

The first and easiest technique that can be applied is that of quantization. Quantization maps symbols from a large set onto a smaller set that can use less bits for representation. For numeric data, precision is reduced maintaining acceptable quality levels while gaining extra bits in the process. Any kind of analog-to-digital conversion requires quantization for example, since the arbitrary precision of the analog world gets lost when translated into a fixed-precision world.

Quantization is done in two different scenarios:

- For floating point data, it usually consists on rounding to the nearest integer, possibly after multiplying by a factor if the numbers are too small. Integers are easier to deal with

since operations between them are simpler to carry out in hardware. If the conversion is done properly, the inverse can recover the original numbers with great precision. A simpler quantizer can be as follows:

$$Q(x) = \Delta \cdot k, \quad k = \left\lfloor \frac{x}{\Delta} + \frac{1}{2} \right\rfloor \tag{2.13}$$

k is the quantized value which is an integer, and to recover the approximation of the original, a multiplication by the step-size Δ is performed. The smaller Δ is, the closer the reconstruction is, but the higher k will be, requiring more bits for storage.

The mean squared error (MSE) when reconstructing can be found by calculating the MSE over any interval of length Δ, since all values within the interval collapse to its center when quantized. Thus, integrating the difference, we find that the MSE is:

$$\frac{\int_{-\Delta/2}^{\Delta/2} x^2 dx}{\Delta} = \frac{\left[x^3/3\right]_{-\Delta/2}^{\Delta/2}}{\Delta} = \frac{\frac{\Delta^3}{2^3 \cdot 3} - \frac{-\Delta^3}{2^3 \cdot 3}}{\Delta} = \frac{\Delta^2}{12} \tag{2.14}$$

By adding or removing one bit from the quantizer, the value of Δ is halved or duplicated. This means the MSE goes down or up by a factor of $1/4$. Equivalently, the signal quality changes by $10 \log_{10}(1/4) \approx 5.756 dB$.

- For integer data, quantization consists on removing the lower bits of the numbers (just shifting). This saves as many bits as are removed per sample, while keeping decent quality if values are sufficiently sparse. If they are close together, they can be centered around the mean and then the upper bits removed instead.

For signal compression, a dead zone (quantization interval wider than the rest) in the quantizer output might be useful as a noise gate that collapses low values to zero to aid in compression. For this purpose, dead-zone quantizers are used where:

$$k = sign(x) \cdot \max\left\{0, \left\lfloor \frac{|x| - w/2}{\Delta} + 1 \right\rfloor\right\} \tag{2.15}$$

where w is the width of the dead zone, centered around zero in this case, which will collapse to zero. This type of quantizer is also useful when output values need be in sign-magnitude form, which is useful for some encoding schemes.

2.3.1.1 Down-scaling

Down-scaling is a special type of quantization where the signal's precision is reduced by a certain factor. For example, for floating-point numbers, it might increase their separation ϵ to $c\epsilon$, where c is the down-scale factor. Note that precision is worse the more separated samples are. An example can be seen in Figure 2.5.

For integers, down-scaling is usually done with simple shifting. Thus, the down-scale factor will always be $c = 2^n$ for some n. Doing this for a signal with k bits effectively reduces the total size by a factor of n/k. Cost-wise, it is the cheapest compression to do, since it only requires shifting of the input data. Recovery is done again with shifting, though the lower bits are lost, as with any kind of quantization.

2.3.2 Down-sampling

Another simple and fast way of reducing the amount of data is to down-sample the information, taking away every nth sample. More aggressive down-sampling is called decimation, where instead only every nth sample is kept.

Figure 2.5: Original image, down-sampled with cubic interpolation, down-sampled with no interpolation, and down-scaled to 9 bits.

Down-sampling might create artifacts for some signals, so usually filters are used to, instead of just removing samples, interpolate them to smooth out irregularities. For inverting the process, the missing values can be interpolated back to get an approximation of the original ones.

Down-sampling by a factor d effectively reduces data size by the same factor. When interpolation is not used for down-sampling, this method is even faster than down-scaling for size reduction since it does not require any processing at all, just discarding certain samples. Complex interpolation methods might increase processing times, but increase the quality of the result, as seen in Figure 2.5.

2.3.3 Domain changes

Data can be represented in multiple ways. Whenever a signal is sampled, it is done in a certain space: Audio is sampled in time, images are sampled in space and in different colors, video is sampled in both time and space...

This is due to the nature of the sensors. Certain physical properties are being captured, that require specific kinds of sensors for them. But there is a correspondence between different domains in which the data can be represented. There are different formats for audio, video and still images, all representing the same data in different domains.

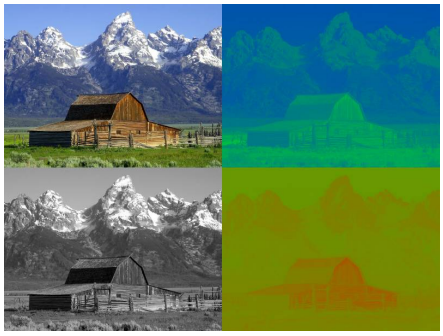

Figure 2.6: Separation [15] of the Y (luminance), Cb (blue-difference chroma) y Cr (red-difference chroma).

Sometimes this transformations make it easier to analyze or, in this case, compress data. Certain domains present more redundancies than others. For example, for images (see Figure 2.6), the YCbCr color space is often used for storage, since it separates the original RGB (red, green and blue) data into three channels, two of which can be down-scaled (the chroma components) without perceptible loss of information for the eye.

These domain changes are called transforms when referring to pure compression purposes.

2.3.4 Transforms

As data increases in redundancy, the potential compression ratio that can be achieved also increases. Transforms are a way of increasing redundancy in the data with little to no cost.

A transform can be invertible or non-invertible. In the first case, the transformation process can be reversed and the original data recovered. In the second case the original data will be lost, with only an approximation available. Transforms work by converting data in a time or space domain into data that lives in the frequency domain. This is based on Fourier series.

Joseph Fourier discovered that, given any function $f(t)$ in an interval, it can be decomposed in a summation of infinite sinusoidal functions as follows:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos\left(\frac{2n\pi}{T}t\right) + b_n \sin\left(\frac{2n\pi}{T}t\right) \right] = \sum_{n=-\infty}^{\infty} c_n e^{i\frac{2\pi nt}{T}} \tag{2.16}$$

where a_i and b_i are coefficients that can be simplified by using complex values to c_i. The interesting part here is that, by the Riemann–Lebesgue lemma, all coefficients tend to zero with infinity. In practice, they tend to zero fairly fast, and a few tens of coefficients suffice for approximating fairly well any simple function. Informally, this is equivalent to saying that lower-index coefficients are more "important" and carry more information than higher-index ones. This is what is called the energy compaction property.

And this is the key for transforms. Since lower-index coefficients are more important than higher-index ones, those are kept while the rest are discarded. The more that are ommited, the more distorted the original data will be, with more compression ratio achieved. Note that for this to work, the function needs to be fairly smooth, as it usually happens with functions that represent a real signal source. Of course arbitrary functions can be created that violate this property up to arbitrarily large indices, but those do not tend to appear naturally.

2.3.4.1 Discrete Fourier Transform (DFT)

The problem with transforms is that they deal with infinite coefficients and continuous functions. In a practica scenario, sampled data from an analog source is not continuous nor infinite. It will be composed of multiple discrete samples evenly separated over time, space or any other dimension. To transform that into the frequency domain, Discrete Transforms are used (in this case Fourier), that covert the input samples $x_n, n = 1, \ldots, N$ into the same number of output samples $X_k, k = 1, \ldots, N$ as follows:

$$X_k = \sum_{n=1}^{N} x_n \cdot e^{-\frac{i2\pi}{N}kn} = \sum_{n=1}^{N} x_n \cdot \left[\cos\left(\frac{2\pi}{N}kn\right) - i \sin\left(\frac{2\pi}{N}kn\right) \right] \tag{2.17}$$

And, to reverse it:

$$x_n = \frac{1}{N} \sum_{n=1}^{N} X_k \cdot e^{i2\pi kn/N} \tag{2.18}$$

The same property as with the non-discrete Fourier transform holds: Coefficients X_k for $k \approx N$ will bear less importance than those with $k \approx 1$, and so they can be removed with little impact on reconstruction. Here, the impact of transforms in compression is seen. By removing half of the original x_n values, half of the information is lost. However, by doing it with the X_k instead, information is removed after decorrelating in the frequency domain, losing only the less important data components.

2.3.4.2 Discrete cosine transform (DCT)

The DFT works over complex values, but it is very rare that the data will consist of complex numbers. The DCT [146] solves this issue by adapting the DFT to real-valued numbers. The most common formula for the DCT is as follows:

$$X_k = \sum_{n=1}^{N} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right] \quad k = 1, \ldots, N \tag{2.19}$$

which can be inverted with:

$$x_n = \frac{X_1}{N} + \frac{2}{N}\sum_{k=2}^{N} X_k \cos\left[\frac{\pi}{N}k\left(n+\frac{1}{2}\right)\right] \quad n = 1, \ldots, N \tag{2.20}$$

Despite being simpler to use than the DFT because of working in the real number domain, DCT offers a few more advantages that have popularized it:

- The DCT performs higher energy compaction than the DFT. With less coefficients, the inverse DCT can reconstruct the original signal better than the inverse DFT.
- It can be performed very fast computationally, with different acceleration techniques when working over small sets of data.
- It has a clear interpretation in the frequency domain of the obtained coefficients, same as the DFT, which helps to visually understand the process.

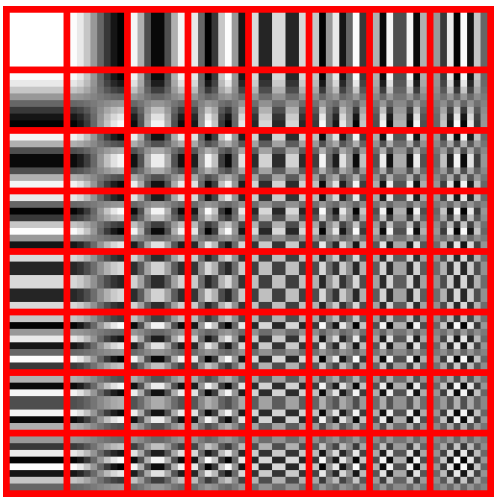

Figure 2.7: The DCT represents each 8×8 block as a combination of these 64 basic patterns. Each coefficient indicates the amount of each in the original [54].

The DCT is also easy to adapt to higher dimensions, by just successively applying DCTs along each dimension. This makes it perfect for images, which is what it was originally designed for. As an example, consider the following transformation:

$$\begin{bmatrix} 52 & 66 & 70 & 73 \\ 63 & 122 & 154 & 69 \\ 67 & 104 & 126 & 70 \\ 87 & 68 & 65 & 94 \end{bmatrix} \overset{\text{centering}}{\rightarrow} \begin{bmatrix} -76 & -62 & -58 & -55 \\ -65 & -6 & 26 & -59 \\ -61 & -24 & -2 & -58 \\ -41 & -60 & -63 & -34 \end{bmatrix} \overset{\text{dct}}{\rightarrow} \begin{bmatrix} -415 & 27 & 56 & 0 \\ -49 & -15 & -10 & 2 \\ 12 & -4 & -2 & 3 \\ 0 & -4 & -1 & 2 \end{bmatrix} \tag{2.21}$$

As clearly seen, after the transformation, the coefficients that are closer to the upper left corner are higher in value, while the bottom right ones are lower. Those can be safely ignored while (usually diagonal lines will be fully discarded) while keeping the high-valued ones. That is where the DCT gets compression from.

There is one problem however with this kind of technique: the complexity of calculations grows quadratically with the input size, since all transformed values require of all input values to be obtained. For small data sets this is not a problem, but it can quickly become a bottleneck. To solve this issue, data is usually split into small blocks. As an example, for images, they are tiled into small blocks which are independently processed by the DCT. Each coefficient in the transformed space indicates the amount of a certain pattern that is present within the block, as seen in Figure 2.7.

2.3.4.3 Wavelet Transforms

While the problem of complexity for DFT or DCT gets addressed with the tiling of the data, this can generate artifacts in the signal. Boundaries between tiles will not necessarily match, and discontinuities will be evident when aggressive compression is performed.

To solve that problem, wavelet transforms [160] are used. The principle is the same as with DFT or DCT: create a 1 to 1 mapping where the output set presents more redundancies than the input set, and where certain samples can be quantized or eliminated. However it does it in a different way:

Instead of using every input to calculate every output ($\mathcal{O}(n^2)$), output i is now a function of input i and its neighborhood. The neighborhood will be symmetric (e.g: using the three previous and three following samples). This way, cost is bounded by $\mathcal{O}(Kn)$, where K is a constant that depends on the neighborhood's (or window) size. For multi-dimensional transforms, this is applied along each direction, as was done with DFT or DCT. This merely means the cost scales linearly with the number of dimensions, not a problem in practice.

There is an additional difference: instead of a single function or "filter", now there are two. Each is a 1 to 1 mapping of the input space to the output space. At first glance, this doubles the number of samples. However just half of the samples in the output spaces will be needed for reconstruction. So how does this work?

Definition 2 *A **kernel** is a function $k : \mathbb{R}^n \to \mathbb{R}$ in which the output generally represents some property of the input vector. Kernels are usually represented by matrices such as:*

$$k: \begin{bmatrix} -\frac{1}{2} & 1 & -\frac{1}{2} \end{bmatrix} \tag{2.22}$$

These work as follows:

$$k\left(\{v_{-1}, v_0, v_1\}\right) = -\frac{1}{2}v_{i-1} + v_i - \frac{1}{2}v_{i+1} = w_i \tag{2.23}$$

Normally, kernels are applied as windows (Figure 2.8) that move over longer vectors $V \in \mathbb{R}^m, m \gg n$ following: $K : V = \{v_1, \ldots, v_n\} \to W = \{w_1, \ldots, w_n\}$ where:

$$w_i = -\frac{1}{2}v_{i-1} + v_i - \frac{1}{2}v_{i+1} \tag{2.24}$$

where around the edges, samples that go outside V's boundaries are usually mirrored or assumed to be zero.

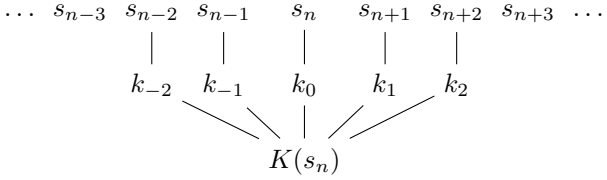

Figure 2.8: Kernel operating over a sequence. When computing, a window of samples slides over the kernel, which generates an output value for each input.

For wavelet transforms, two kernels K_l and K_h are used, which work as a lowpass and highpass filter respectively. The lowpass filter keeps the slowly-varying characteristics of the vector, and is a smoothed out version of the original. The highpass filter keeps the rapidly-varying characteristics.

The result of filtering is a mapping of the input data to two different images (low-passed and high-passed) with which it can be reconstructed. A particular property is of interest here: filters are prepared in such a way that the lowpass filter removes all frequencies above half the maximum (where maximum refers to half the sampling rate or Nyquist Frequency), and the highpass filter removes all frequencies below half the maximum [160, fig 4.1]. Therefore, if the original sampling rate for the discrete signal was sufficient to reconstruct the (original) continuous signal, then with half the frequency range, half the samples can be discarded safely in the lowpassed signal

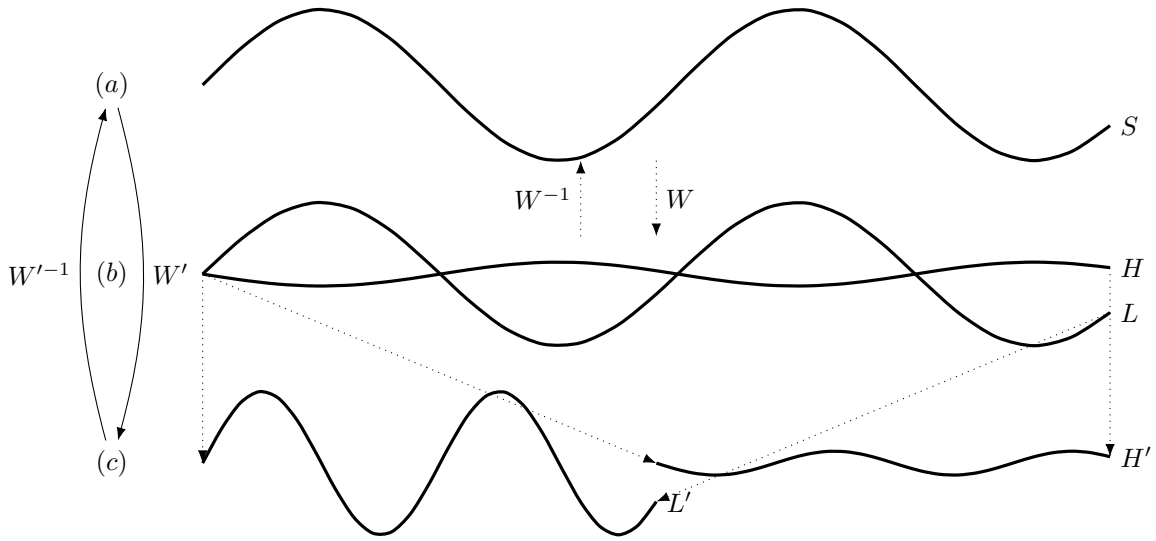

Figure 2.9: (a) is the original series, (b) shows the high and lowpass filtering, (c) shows (b) after subsampling.

while allowing for reconstruction. The same can be applied to the highpassed values, removing every other sample as well, and achieving a 1 to 1 invertible mapping where redundancies on the output signal are much higher.

Definition 3 *Given samples* $S = \{s_i\}_{i=1}^n$*, a* ***wavelet transform*** *is an invertible function* $W = (K_l, K_h) : \mathbb{R}^n \to (\mathbb{R}^2)^{n/2}$*.*

W consists of a tuple of kernels (K_l, K_h) that, when applied on S, act as low and highpass filters, generating sequences L and H respectively:

$$W(S) = (K_l(S), K_h(S)) = (L, H) = (\{l_i\}_{i=1}^n, \{h_i\}_{i=1}^n) \tag{2.25}$$

where both L and H are down-scaled to half their sizes and instead $L' = \{l_i\}_{i\in 2\mathbb{N}}$ $H' = \{h_i\}_{i\in 2\mathbb{N}}$ are used, since the rest of the samples are redundant. Finally, the output is $W'(S) = S' = \{s'_i\}_{i=1}^n$:

$$s'_i = \begin{cases} l'_{i/2} & i \in 2\mathbb{N} \\ h'_{(i-1)/2} & otherwise \end{cases} \tag{2.26}$$

The inverse wavelet W'^{-1} is a tuple of kernels (K'_l, K'_h) which reconstruct S after application. Figure 2.9 shows how this process works.

By looking at Figure 2.9 it is seen that, while the first part of the transformed signal is similar to the original, the second is skewed towards zero. This redundancy will be key for compression. Not only that, wavelets can be recursively applied over the lowpassed part for increased redundancy. High frequencies are usually very short lived within the signal, so the highpassed signals are usually random. However, low frequencies last for longer within the signal, and more redundancy can be extracted by recursively applying the filters to that portion of the signal.

When dealing with two-dimensional signals, such as images, these wavelets are applied along both dimensions. Figure 2.10 shows this process, where it can be seen that the lowpassed part is a small version of the original, while the highpassed parts are dull containing little information.

These four sectors (called subbands) that are formed when applying wavelet transforms to two-dimensional data present different types of redundancy, shown in Table 2.3.

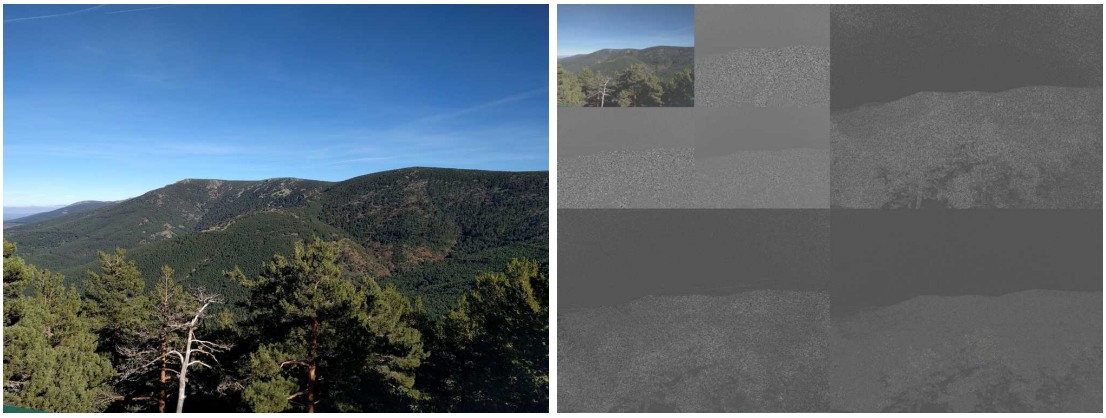

Figure 2.10: Original image on the left, transformed on the right after two wavelet passes. Information is concentrated on the top-left corner.

	Pass		
Subband	Horizontal	Vertical	Details
LL	Low	Low	Contains most of the information. Difficult to compress.
LH	Low	High	Presents horizontal redundancy.
HL	High	Low	Presents Vertical redundancy.
HH	High	High	Contains little information. Diagonal redundancy.

Table 2.3: Types of subbands resulting from a wavelet transform.

Wavelets are often applied recursively over the LL subband. This is because it still contains redundancy of higher frequencies that can be exploited with another wavelet pass. This breaks it into new subbands, which will be marked with a subindex that denotes how many times a wavelet transform has been applied on it (e.g: LH_2 is the LH subband of the first LL subband).

Depending on the selected kernels, many types of these transforms exist. Usually they differ on kernel size, and will detect different periods in the input data with their low and high pass functions. Normally these work on floating point numbers, but that can obviously lead to rounding errors when reconstructing. For that, integer transforms also exist that, while doing worse energy compaction, ensure perfect reconstruction.

2.3.5 Dimensionality reduction

Another way to reduce size is to directly reduce information. Normally, information is spread out within the data, and that is why transforms are useful to condense it into known places. More resources and time can be dedicated to properly compress these parts of the data.

A different approach is to use mathematical models that directly determine what is considered relevant information, discarding the rest. Dimensionality reduction methods achieve a fixed degree of compression by removing a specified amount of information according to those models.

As an example, consider facial recognition [235]. Images have millions of pixels, and doing recognition based on those can be near-impossible. Instead, the focus is put on features such as shape, color, length... With a much smaller set of features than pixels, it is possible to identify what is shown in the image. The same concept applies to data, where by just analyzing interesting features the full set can be understood.

Mathematically, the samples live in $\mathbb{R}^n$, and their dimension is reduced to $R^m, m < n$. The input vectors of dimension n are projected onto the output space of dimension m. Analysis and storage are then easier to carry out in the image, since less information is dealt with.

When doing these transforms information will be inevitably lost. It is expected to lose only redundant information that the algorithms will be able to recover, but some useful data might be lost. How far this reduction process is taken will depend on what trade-off is made between reduction and fidelity.

A dimensionality reduction is a function $f : \mathbb{R}^n \to \mathbb{R}^m$ that transforms a vector $\boldsymbol{x} \in \mathbb{R}^n$ in another $\boldsymbol{t} \in \mathbb{R}^m$. To go back to the original, a function is usually generated as $g : f(\mathbb{R}^n) \subset \mathbb{R}^m \to \mathbb{R}^n$. Ideally $g \equiv f^{-1}$, but it will not always be possible since f is not injective over its image, so an approximation $g(t) = \boldsymbol{x}' \approx \boldsymbol{x}$ is the best approach. That approximation will be closer the lower $m - n$ is, and will incur in more loss of information when $m \ll n$. The original data is represented as a matrix:

$$X = [\boldsymbol{x_1}, \ldots, \boldsymbol{x_p}] \in \mathcal{M}_{n \times p} \tag{2.27}$$

with p being the number of samples. The transformed data is thus:

$$T = f(X) = [f(\boldsymbol{x_1}), \ldots, f(\boldsymbol{x_p})] = [\boldsymbol{t_1}, \ldots, \boldsymbol{t_p}] \in \mathcal{M}_{m \times p} \tag{2.28}$$

For compression, both T as well as the function g used to undo the transformation are stored. g could be generic for multiple images and even be reused, avoiding having to pack it in the output stream. However, for improved results, it will usually be modified specifically for the set T, needing to preserve it as well.

2.3.5.1 Principal Component Analysis (PCA)

Principal Component Analysis is a dimensionality reduction technique that is broadly used in machine learning [27] due to its simplicity. Data visualization or feature extraction [1] are amongst its uses.

PCA is defined [92] as the orthogonal projection of n-dimensional data onto a space of lower dimension m, known as the principal subspace, where the projection's variance is maximized.

PCA keeps the maximum amount of variance present in the input data set, assumming it will keep the most amount of information in the process. An example of PCA can be seen in Figure 2.11

To do the projection, vectors $\{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m\}, \boldsymbol{w}_i \in \mathbb{R}_n, \boldsymbol{w}_i \perp \boldsymbol{w}_j \forall i \neq j$ are needed. Those form the matrix $W \in \mathcal{M}_{n \times m}$, such that $f(\boldsymbol{x_i}) = \boldsymbol{x_i}W$, and $g(\boldsymbol{t_i}) = \boldsymbol{t_i}W^T$ define the reduction function $f : \mathbb{R}^n \to \mathbb{R}^m$ and recovery funciton $g : \mathbb{R}^m \to \mathbb{R}^n$.

When $W^{-1} = W^T$, then $f \equiv g^{-1}$, but that is only possible when:

- W is orthogonal, which by definition it is.
- $n = m$, which for the purposes of reducing dimensionality will never be true.

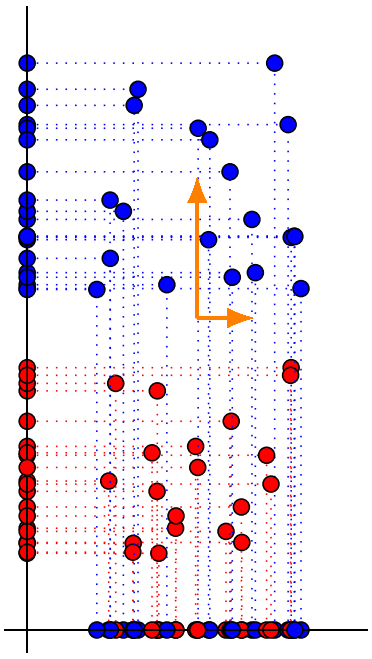

Figure 2.11: An example of PCA. Eigenvectors are shown in orange. The greatest variance exists along the direction of the eigenvector of greatest eigenvalue.

In practice, since the domain of f is a subset of $\mathbb{R}^n$ of implicit dimension that is not necessarily n, it is possible that g is the inverse of f over its image even when $n < m$.

PCA works best with data centered around zero, and so usually the set $Z = X - \bar{\boldsymbol{w}}$ will be used, where:

$$\bar{\boldsymbol{w}} = \frac{\sum_{i=1}^{p} x_i}{p} \tag{2.29}$$

To obtain the matrix W, first the covariance matrix $S \in \mathcal{M}_{n\times n}$ of the input data $X \in \mathcal{M}_{n\times p}$ is calculated, with p the number of samples in the original space. It is obtained as:

$$S = \frac{(X - \bar{\boldsymbol{x}})(X - \bar{\boldsymbol{x}})^T}{p-1} = \frac{ZZ^T}{p-1} \tag{2.30}$$

Let $W^r = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n\}$ be the eigenvectors and $\Lambda^r = \{\lambda_1, \ldots, \lambda_n\}$ the associated eigenvalues of S. Sort the eigenvalues in descending order:

$$\Lambda = [\lambda_{i_1}, \ldots, \lambda_{i_n}] \shortparallel \forall j, k \in \mathbb{N}^n \ \lambda_{i_j} >= \lambda_{i_k} \iff j > k \tag{2.31}$$

Generate the projection matrix $W \in \mathcal{M}_{n\times m}$ as:

$$W = [\boldsymbol{w}_{i_1}, \ldots, \boldsymbol{w}_{i_m}] \tag{2.32}$$

The reconstruction matrix $\bar{W}$ is just W^T, since $WW^T \approx I_n$. With this technique, it is guaranteed that the projection of the input data set maintains maximum variance [27, Ch.12].

2.3.5.2 Singular Value Decomposition (SVD)

SVD is a method similar to PCA, which mainly obtains the same result except for certain multiplicative factors [188]. SVD looks for the following decomposition:

$$X^T = U\Sigma V^T, \quad U \in \mathcal{M}_{p\times p}, \quad \Sigma \in \mathcal{M}_{p\times n}, \quad V \in \mathcal{M}_{n\times n} \tag{2.33}$$

where U and V are orthogonal matrices, and Σ is a diagonal matrix with the singular values σ_i in its diagonal.

The covariance matrix S can be diagonalized:

$$S = W\Lambda W^t \tag{2.34}$$

where W contains the eigenvectors as columns, and Λ the eigenvalues λ_i of S on its diagonal.

Substituting for Eq. (2.30), assuming $\bar{\boldsymbol{x}} = 0$ (since values can be centered around the mean with little computational cost):

$$S = \frac{(U\Sigma V^T)^T U\Sigma V^T}{p-1} = \frac{V\Sigma U^T U\Sigma V^T}{p-1} = V\frac{\Sigma^2}{p-1}V^T \tag{2.35}$$

V gives vectors proportional to W scaled by $\sigma_i/\sqrt{p-1}$, following $\lambda_i = \sigma_i^2/(p-1)$. This is only true when x_i are centered around zero, which is when $S = X^tX/(p-1)$ holds.

By sorting the values in the diagonal of Σ in Equation (2.33), in descending order, the so-called "compact" SVD is obtained, where $U \in \mathcal{M}_{n\times r}$, $\Sigma \in \mathcal{M}_{r\times r}$ and $V \in \mathcal{M}_{r\times m}$. Here r is the rank of X. The difference $p - r$ is filled with zeros and thus can be safely removed for the compact representation, simplifying the matrices. For the hyperspectral matrices used (that can be considered random), $r = m$ with a probability[1] of 1.

[1] A simple proof involves the fact that, given m random vectors, each spans a set of linearly dependent vectors of measure zero when embedded in m dimensions. Since m is finite, the likelihood of a random vector falling in one of those m zero-measure subspaces is zero, thus all vectors are expected to be independent.

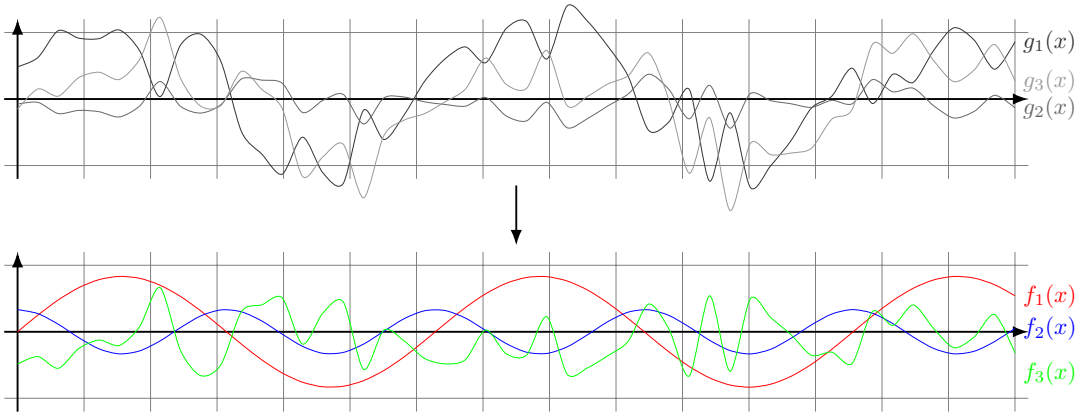

Figure 2.12: Data is generated by linear combination of $f_1(x) = \sin x, f_2(x) = \cos 2x$ and $f_3(x)$ which is just noise. ICA is capable of unmixing the functions g_1, g_2, g_3 extracting the original components.

So, when represented in the compact form, the transformed data is $T = U\Sigma$. Since $V^{-1} = V^T$, Equation (2.33) is rewritten as:

$$XV = U\Sigma \implies (XV)V^T = U\Sigma V^T = X \tag{2.36}$$

The projection matrix will be $W = V$, while the reconstruction matrix will be $\bar{W} = V^T$. If the inequality $r < m$ is forced (by removing the lowest $m - r$ values from the diagonal of Σ) compression is forced to take place.

When $\bar{\boldsymbol{x}} \neq 0$, both PCA and SVD lose performance since variance is calculated around zero. If clustering happens elsewhere, eigenvectors might not be as efficient to separate the set via variance. It is safe to assume however that data will be centered around zero (if not it is easy to transform), and the method that presents better numerical properties (in the sense of rounding and cumulative errors) will be used. Either solving directly for S, or calculating the decomposition $U\Sigma V^T$.

2.3.5.3 Independent Component Analysis (ICA)

This method looks for a representation of a data set of dimension n in another dimension $m < n$ such that the transformed components are as independent among them as possible.

To do so, it minimizes the mutual information [48] of the transformed components. The advantage over other methods is that ICA is invariant under component scaling (PCA is not). This is an advantage in the case of the magnitudes of latent components being very different between them. It can also be a problem in the presence of noise, since it can give the same importance to noise than to a strong signal.

Different approaches calculate mutual information, which adapt to different kinds of data and characteristics. Due to the iterative nature of the algorithm, usually functions with low complexity are used by assuming certain properties on input data [97].

ICA assumes there are m latent independent components in the original $\mathbb{R}^n$ space. The algorithm works best the closest m is to the actual value of latent components (which is unknown but can be estimated). In the case of exactly matching the number of components, performance is excellent as shown in Figure 2.12 for $n = m = p = 3$.

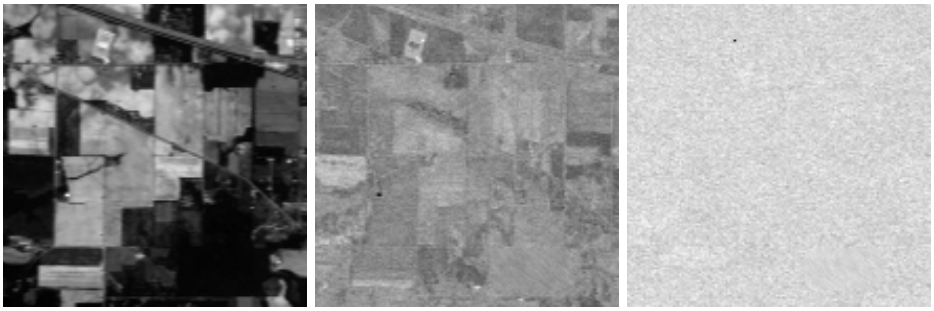

Figure 2.13: The first, fifth and twentieth components on a sample MNF transform of a hyperspectral image. Noise is lower for the first components.

2.3.5.4 Minimum Noise Fraction (MNF)

The minimum noise fraction [82] is a dimensionality reduction method similar to PCA, but that, instead of ordering the principal components based on variance, does so based on the amount of noise they contain. For that it uses not only the covariance matrix S, but also the noise covariance matrix $S_{\mathcal{N}} \in \mathcal{M}_{n\times n}$.

$S_{\mathcal{N}}$ is not often known beforehand, but can be estimated [80] if spatial correlation is assumed between samples x_i that are close together. This is reasonable in *natural* data such as pictures, where pixels that are close together have similar values. For that, the noise matrix $\mathcal{N} = [n_1, \ldots, n_p] \in p \times n$ is created:

$$n_i = \frac{1}{2}(x_i - x_{i+1}) \tag{2.37}$$

Then $\mathcal{N}$ is:

$$S_{\mathcal{N}} = \mathcal{N}\mathcal{N}^T \in \mathcal{M}_{n\times n} \tag{2.38}$$

Solving the generalized eigenvalue problem:

$$S_{\mathcal{N}}\boldsymbol{w} = \lambda S\boldsymbol{w} \tag{2.39}$$

Then $W = [\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n] \in \mathcal{M}_{n\times n}$.

The set T of data transformed with MNF can be expressed [28] then as $T = W^T Z$, where Z is the set X centered around 0 by subtracting the average value $\bar{\boldsymbol{x}}$ (Eq. (2.29)).

Here $T \in \mathcal{M}_{n\times p}$. Each t_i has successively more noise when i grows towards n, as seen in Figure 2.13. The most noisy $m - n$ components are eliminated to reduce T's dimensionality.

2.3.5.5 Vertex Component Analysis (VCA)

VCA was designed [151] as a spectral unmixing algorithm: Starting from a hyperspectral image, it is capable of extracting the spectral signatures of the pure pixels of the image, representing the rest of the samples as linear combinations of the pure pixels.

VCA starts by performing SVD of the data X. The data is projected as $X = XV^{svd}$ (note X is reused for simplicity). The mean value of the projected data $\bar{\boldsymbol{x}}$ is obtained, and a projective projection of X is done on that vector:

$$Y = \frac{X}{X^T\bar{\boldsymbol{x}}} \tag{2.40}$$

Now let matrix $A \in \mathcal{M}_{p\times p}$, where p is the target dimension, be:

$$A = [\boldsymbol{e}_u, \boldsymbol{0}, \ldots, \boldsymbol{0}] \shortparallel \boldsymbol{e}_u = [0, \ldots, 0, 1]^T \tag{2.41}$$

Now, iteratively, do the following p times with i as an iteration variable:

- Generate $\boldsymbol{w}$ as a random zero-mean Gaussian vector of covariance I_p. This vector is statistically expected to be independent than any others already present in A.
- Generate an orthonormal vector f to the subspace spanned by the current A matrix as follows:
$$\boldsymbol{f} = \frac{(I - AA^{\#})\boldsymbol{w}}{||(I - AA^{\#})\boldsymbol{w}||} \tag{2.42}$$
Where $A^{\#}$ is the pseudoinverse of A.
- Let $\boldsymbol{v} = \boldsymbol{f}^T Y$, and k the index of v which has the maximum value (the projection extreme).
- The ith row of A becomes the kth row of Y, and index k is stored in a list as k_i

The projection matrix W is that formed with the vectors from the SVD decomposition X of indices $k_i, i = 1, \ldots, p$.

Intuitively, this algorithm finds extreme points in the input set, and defines the rest as a linear combination of those. The reconstruction matrix $\bar{W}$ is in this case the pseudo-inverse of W obtained as $\bar{W} = pinv(W) = W^T(WW^T)^{-1}$.

2.3.5.6 Vector Quantization PCA (VQPCA)

VQPCA is a simple extension of PCA to adapt to nonlinear reductions. Linear projections (such as the methods previously seen) work fairly well when all of the data points are linear combinations of a certain set. Otherwise, information losses can be amplified.

VQPCA separates the input data set by doing Vector Quantization over it. Similar vectors end up clustered together. Then, linear dimensionality reduction (in this case PCA) is applied over each cluster. Thus, a piecewise linear reduction is obtained, which presents an improvement over plain linear methods [111], while still being fast enough for practical use.

VQPCA receives as an input both X and the number c of subsets that are to be formed. VQ divides X into c subsets of related vectors, in this case by applying the nearest neighbor algorithm to cluster them. Indices $C = \{c_1, \ldots, c_p\}$ are generated, indicating to which subset X_j each sample belongs to:

$$X_j = \{x_i \shortmid\shortmid c_i = j\}, \quad j \in \{1, \ldots, c\} \tag{2.43}$$

PCA is afterwards performed per subset, generating each transformed set T_j and the recovery functions $g_j : T_j \to X_j$. To retrieve X, the following values are needed:

- C, to separate T into its subsets, since $T_j = \{t_i \shortmid\shortmid c_i = j\}$.
- The transformed set $T = \cup_{j=1}^{c} T_j$
- $g_j, j \in \{1, \ldots, c\}$ to perform inverse PCA over each subset.

Saving T is mandatory for all methods, but VQPCA needs considerably more information than other methods. C requires an index per sample, and g_j are big matrices which also need to be stored. Both grow in size with c.

Both T and $g_j, j \in \{1, \ldots, c\}$ use high-precision floats that are not efficiently compressed in a lossless way. They are thus preprocessed with transforms or quantization incurring in some losses. However C needs to be kept intact because the exact integers are required for

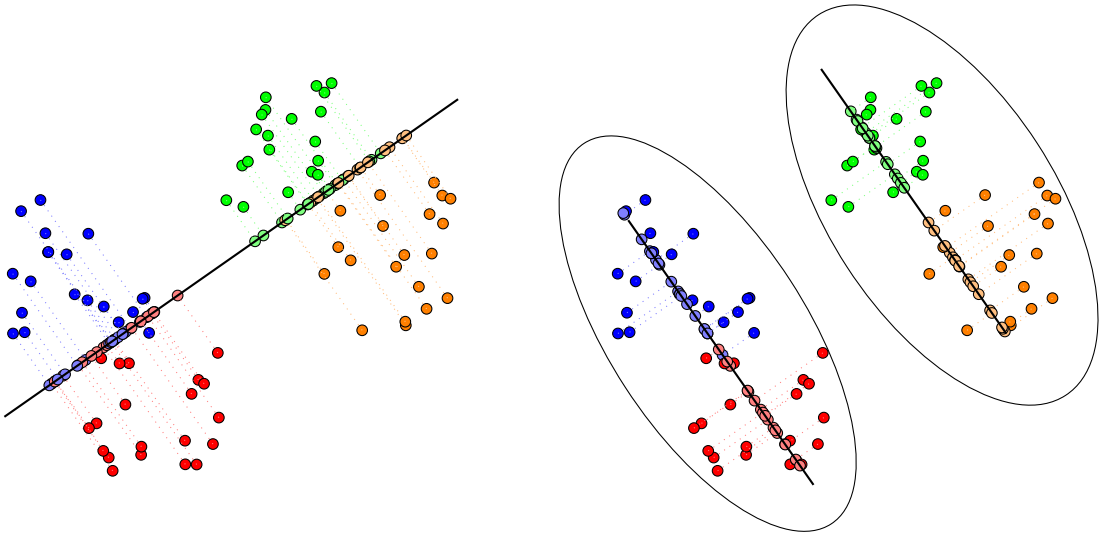

Figure 2.14: On the left, PCA. On the right VQPCA. VQPCA is able to separate both clusters, increasing the performance of PCA afterwards.

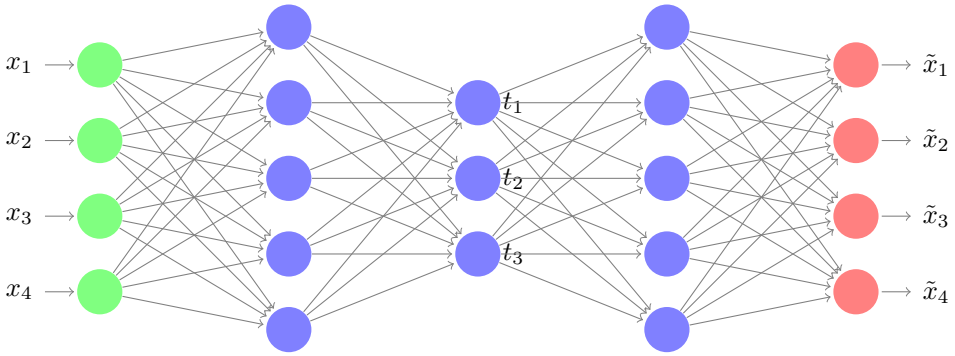

Figure 2.15: The input and output layers are the same size. The hidden layers perform compression by reducing the amount of neurons.

decompression. Since it presents great redundancy (samples that are close together tend to be in the same cluster), an adaptive entropy coder is used which will reduce its size considerably.

The advantages of VQPCA are seen in Figure 2.14. After doing PCA on each cluster obtained by VQ, separation is more clear than with plain PCA directly.

2.3.5.7 Auto-encoders

Auto-encoders take advantage of neural networks to perform compression. A network is built with an input and output layer of size n. In the middle, a funnel-like shape reduces the layer size to the target dimension m. This can be seen in Figure 2.15.

Compression relies on the hidden layers, of which at least one is of the target size $m < n$. A vector $\boldsymbol{x} = \{x_1, \ldots, x_n\}$ is fed into the n input neurons, and a new vector $\tilde{\boldsymbol{x}} = \{\tilde{x}_1, \ldots, \tilde{x}_n\}$ is obtained at the output. The network is trained so that $\tilde{\boldsymbol{x}} \approx \boldsymbol{x}$. This way, the network will be able to reconstruct the original values as close as possible. For compression, the internal values of the network at the hidden layer of size m are stored, as well as the layers after that that reconstruct the information.

This approach can perform linear reductions with just one fully connected layer, and can go beyond linearity as the number of layers increase. The problem is that the time required

to train the network is much higher than that of calculating the transform matrices for linear or piecewise linear methods. For a fixed data set, the network is ideally over-fitted since no new data will be processed by it. And that might take many iterations until fully completed. While this is a good method if computing power is available, it will not adapt well to real-time constraints.

2.3.6 Quality metrics

An important part of lossy compression is knowing how much information is lost. The reconstructed data (after compression) is compared against the original and a value is obtained. Its meaning depends on the metric used, and it will indicate how close the reconstructed data is to the original (i.e: its quality). Quality measures give results over the whole data set, meaning the resulting value indicates the average quality.

Definition 4 *The **maximum square error** (maxSE) is defined as the maximum value of the squares of the differences between each sample and its reconstructed value. Denoting a data set by $D = \{d_1, \ldots, d_n\}$:*

$$maxSE(D^a, D^b) = \max_{j=1}^{n}(d_j^a - d_j^b)^2 \tag{2.44}$$

This metric is useful to detect the maximum distortion in a data set. If the maxSE is small, the general distortion will be small as well. However, small anomalies can increase maxSE without being significant for the overall distortion, so other metrics are often used.

Definition 5 *The **mean square error** (MSE) is defined as the mean of the squares of the differences between samples and its reconstructed values:*

$$MSE(D^a, D^b) = \frac{\sum_{j=1}^{n}(d_j^a - d_j^b)^2}{n} \tag{2.45}$$

This measure gives a general idea of the absolute quantity that reconstructed samples deviate from the originals, but it can be misleading: Imagine all samples are of very low value (e.g: 1) and the reconstructed samples are either 0 or 2. In this case, the MSE is 1, but relative to the sample value it is a tremendous error. The reverse is true as well: for samples with high values, a small error (percent wise) results in a high MSE. Thus it is important to know the magnitude of the samples before getting conclusions out of this metric.

Definition 6 *The **peak signal to noise ratio** (PSNR) is defined as a function of the MSE and the maximum value that a sample could store $r_{\max}(D)$:*

$$PSNR(D^a, D^b) = 10\log_{10}\left(\frac{r_{\max}(D^a)^2}{MSE(D^a, D^b)}\right) \tag{2.46}$$

Here, the MSE is adjusted with the range of the samples, dampening the relative effects that could arise in MSE. This metric is best suited for when samples are evenly spaced throughout the sample space. If samples are all clustered towards the range limits, PSNR might give optimistic results. However, it is reliable for determining distorted sets of data. If PSNR is bad, then the reconstructed data is of low quality.

Definition 7 *The **normalized peak signal to noise ratio** (NPSNR) follows the idea of PSNR, but instead of taking the maximum theoretical value, it takes the actual range of the samples as reference:*

$$NPSNR(D^a, D^b) = 10\log_{10}\left(\frac{(\max(D^a) - \min(D^a))^2}{MSE(D^a, D^b)}\right) \tag{2.47}$$

The advantage of this definition is that now, the values obtained are representative of the actual distortion of the data, instead of just referring to the distortion over the maximum range, even if values did not appear in that range. NPSNR is invariant under dynamic range changes in the data. Even if sample precision is increased, if the actual values do not change, the measure will remain stable. The following holds:

$$NPSNR(D^a, D^b) \leq PSNR(D^a, D^b) \tag{2.48}$$

Definition 8 *The **power normalized signal to noise ratio** (POWSNR) is defined as:*

$$POWSNR(D^a, D^b) = 10 \log_{10} \left(\frac{pow(D^a)}{MSE(D^a, D^b)} \right) \tag{2.49}$$

where $pow(D)$ is defined as:

$$pow(D) = \frac{\sum_{j=1}^{n} (d_j)^2}{n} \tag{2.50}$$

This metric is less sensitive to extreme values than NPSNR, and more reliable under noisy data. The following holds:

$$POWSNR(D^a, D^b) \leq NPSNR(D^a, D^b) \tag{2.51}$$

Definition 9 *The **signal to noise ratio** (SNR) is defined as:*

$$SNR(D^a, D^b) = 10 \log_{10} \left(\frac{\sigma(D^a)^2}{MSE(D^a, D^b)} \right) \tag{2.52}$$

where $\sigma(D)^2$ is the variance of D with mean value $\mu(D)$:

$$\sigma(D)^2 = \frac{\sum_{j=1}^{n} (d_j - \mu(D))^2}{n} \tag{2.53}$$

Introducing the variance in the equation allows the metric to be more permissive on data sets that are less smooth, and more strict on narrow data sets with little variation where a small change can introduce more severe distortions. This is one of the most popular metrics and, as such, it is used to judge image quality. A SNR of 32 is considered [99] excellent quality, with acceptable quality being retained at a SNR of 20. The following holds for SNR:

$$SNR(D^a, D^b) \leq POWSNR(D^a, D^b) \tag{2.54}$$

Definition 10 *The **mean to standard deviation ratio** (MSR) is defined as the quotient of the mean $\mu(D)$ of a reference data set over the standard deviation between D and the reconstructed dataset:*

$$MSR(D^a, D^b) = \frac{\mu_(D^a)}{\sqrt{MSE(D^a, D^b)}} \tag{2.55}$$

This is a good approximation of the amount of error expected in a sample. For a sample of value x, an MSR of y indicates that the reconstructed value is most likely within the range $[x - x/y, x + x/y]$.

Definition 11 *The **structural similitude index** [213] (SSIM) is defined as:*

$$SSIM(D^a, D^b) = \frac{\left(2\mu(D^a)\mu(D^b) + c_1\right)(2\sigma + c_2)}{(\mu(D^a)^2 + \mu(D^b)^2 + c_1)(\sigma(D^a)^2 + \sigma(D^b)^2 + c_2)} \tag{2.56}$$

where σ is the covariance between data sets D^a and D^b, and c_1 and c_2 are correction factors, defined as:

$$c_1 = k_1^2 r_{\max}^2 \qquad c_2 = k_2^2 r_{\max}^2 \tag{2.57}$$

with k_1 and k_2 usually set respectively at 0.01 and 0.03. This metric comes directly from the world of image quality, and those are the values that have been found to work best.

This metric obtains a value between 0 and 1, which indicates higher similitude to the original the closer it is to 1. This metric is based on how humans perceive images, but it can still give a good idea of how much visually important information is lost in the case of images.

2.4 Near-lossless compression

Quality metrics can assess a compression algorithm's quality after it has been executed. For lossy algorithms, this means that knowing the resulting quality beforehand is not possible. Generally, the same settings for an algorithm will produce the same quality results for similar images, so configuration based on experience can usually guide distortion ratio. However, the output quality is not ensured by these methods.

Near-lossless algorithms ensure that the compressed data is above a certain threshold for a given metric. When decompressing, the data does not deviate from the original more than what the compression settings decided. There are different ways (see Fig. 2.16) of accomplishing it:

- **Predict and correct**: Prediction methods are typical of lossless algorithms. They create a model based on already seen data, predict the following sample, and code the difference with the actual values. Since differences are small when a good model is used, compression can be performed easier than on the original samples. When the difference is coded, both the actual and the predicted value are known. At that point, the difference is coded to reconstruct the original value afterwards. If it is skipped, errors are propagated in reconstruction. But those errors are already known at the time of coding.

  This method of compression takes advantage of that fact. It measures the error between the prediction and the original value and, if it is low enough, it just skips coding the difference altogether, letting the predictor do the work. A threshold is set for when to skip a sample, and depending on its permissibility it will skip more or less differences, trading compression for quality.

- **Iterative compression**: A different approach is to use a lossy algorithm, configure it to compress the data, and then decompress and see the result. If the quality matches the expectations, the algorithm stops. If not, it iteratively repeats the process with compression parameters that ensure higher quality until it meets the requirements.

  This can be applied with any lossy algorithm. Data can also be partitioned in smaller blocks to make this process faster, benefiting from some areas having more redundancy than others and thus being more compressible by staying separate.

- **Selective coding**: This method is an in-between of the two previous ones. Predicting and coding is blazingly fast, but can only adapt to local characteristics. Iterative compression can achieve good results but at the cost of repeating the same algorithm over and over, which can be costly.

  Selective coding divides the data into smaller subsets. Compression is applied on each subset. Then, if the quality for that particular subset meets the threshold, the compressed data is coded. If the quality does not meet requirements, then the original data is coded instead in a lossless manner.

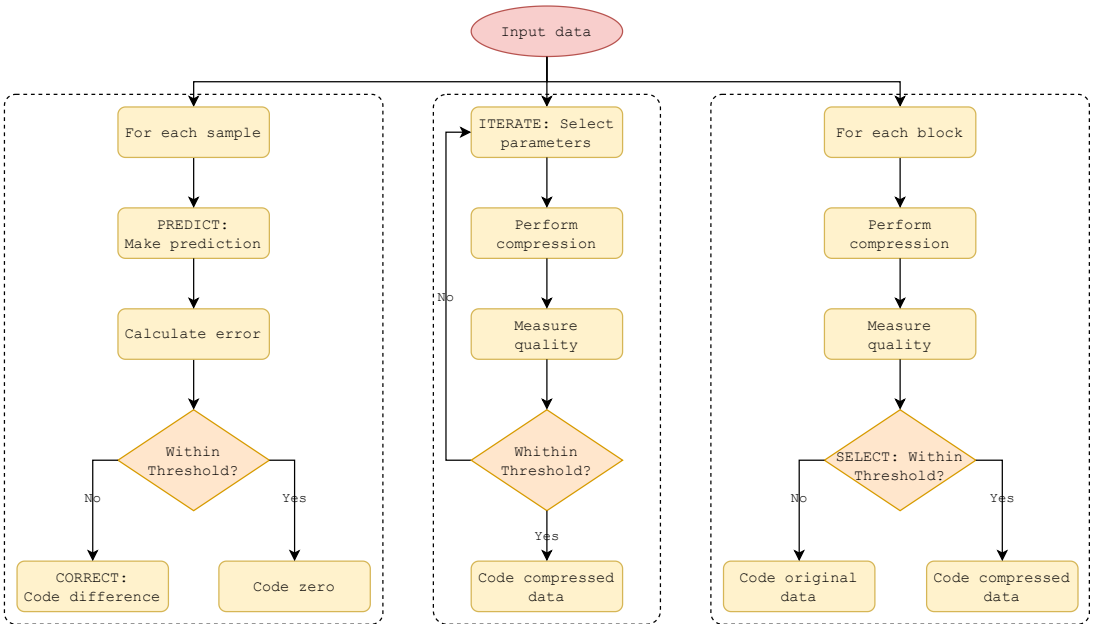

Figure 2.16: The three main near-lossless methods. From left to right: predict and correct, iterative compression, and selective coding.

This creates a sequence of blocks which are losslessly and lossyly compressed. How many blocks are lossless or lossy depends on the quality setting, which in ensuring a certain threshold will force certain blocks to be lossless. The higher the quality setting, the more blocks will need to be lossless, sacrificing size. The lower the quality, the more lossy blocks will be allowed to exist, reducing size further.

Near-lossless algorithms combine the simplicity of lossless algorithms, with the great compression ratio of lossy ones. They can range from lossless to lossy, sacrificing little while retaining good properties from both.

2.5 Summary

Compression techniques are plentiful. Generic ones are aimed to any kind of data. However the lack of compression efficiency for very specific types of data forces the use of specific algorithms. Each has their own trade-offs that need to be addressed for each application.

Different techniques can be combined for increased efficiency. Lossy algorithms often use lossless ones as coders for their transformed data. Near-lossless algorithms often decide between lossy and lossless methods depending on quality metrics.

The three methods offer different advantages, as seen in Table 2.4, so this thesis will focus on one of each to see how they fare in an FPGA. Lossless methods are generally the fastest, offering perfect reconstruction. Lossy methods are much slower but reduce image size by a much higher factor. Near-lossless methods sit in between, not being generally as fast, but offering a wide range of compression ratios.

Method	ratio	Quality at low ratio	Quality at high ratio	complexity	speed
Lossless	low	perfect	n/a	low	fast
Lossy	very high	moderate	excellent	very high	slow
Near-lossless	moderate	decent	moderate	moderate	fast

Table 2.4: Some key aspects about the three main compression techniques, when applied on hyperspectral images. Other applications might see different behaviors.

Chapter 3

Algorithms

A lot of effort has been put towards hyperspectral image compression. Many existing compression techniques have been adapted from the (non-hyperspectral) image compression domain. New, completely custom approaches have also been created.

This has produced an ecosystem of algorithms that share techniques such as prediction, coding, quantization, transforms, dimensionality reduction...

In this chapter, an introduction to existing hyperspectral compression algorithms is first made, showing the diversity that exists even when targeting such specific data. Three algorithms will be selected that broadly cover the whole spectrum, in order to explore the general applicability of FPGA implementations to this domain.

All three are explored in detail, delving into the specifics of each to give a good look before looking at the FPGA implementations in Chapter 4.

3.1 Preliminaries

3.1.1 State of the art

Compression algorithms for any kind of data are often developed as soon as the data is produced. The data needs to be stored to be studied, and compression allows for more information to be available within the same space. Generally, the most common methods are adapted first, with complex and custom variations coming afterwards.

Predictive methods have proven to be useful over and over again for any kinds of compression. Optimizations for the hyperspectral domain [144, Ch.2] have been done to find the ideal predictors for different image traversal modes, after which adaptive context-based encoders are used that take into account hyperspectral characteristics.

Band reordering algorithms [144, Ch.3] have expanded on this idea by enhancing the spectral properties of images, realizing that for some predictors that use many bands this might be unnecessary.

Context-based approaches [212] have also been used to avoid the overhead of band reordering. Lossless algorithms have been refined to the point of developing international standards such as CCSDS 123.0-B-1 [32], which combine the best techniques and are highly configurable.

Vector Quantization (VQ) has also been extensively used to exploit pixels similarities [171] that result from pixels being a mixture of pure spectral samples present within the image. Improvements to this technique include locally optimized quantization [144, Ch.5, 6] that achieves less distortion than globally calculated vectors. Other ideas included the use of VQ after perform-

ing DCTs on small image blocks [144, Ch.9][158], realizing that redundancies in the frequency domain were also exploitable.

Look-up table (LUT) approaches work in a similar but more local way. Patterns are looked for in the previous band, and used as predictors when they match the behavior of the current sample's neighborhood [95, Ch.8]. Working in blocks is also found to improve performance since similarities are more correlated, and the search space smaller. Multi-band LUTs [7] have also been explored in the lossless domain increasing complexity but improving compression. Dictionary approaches based on reference spectral signatures [104] further extend this concept improving performance when spectral signature presence is known or estimated beforehand.

Block-based approaches [95, Ch.3][5] have been fruitful in the near-lossless domain, where multiple samples are predicted at once, saving calculations of doing individual predictions. Coding is still done in the same way after differences are calculated. Near-lossless compression can be achieved at the block or sample level, with band reordering increasing slightly performance even when done per-sensor and not per-image. Even per-block parameter selection [191] has been done to improve this technique, all while providing random read access at the decompressor. Sample-level approaches have found its way to the new CCSDS 123.0-B-2 standard [34], with higher computational complexity but also higher distortion-ratio efficiency than block-based approaches. Rate control techniques are also included [42] to fulfill specific output constraints.

Wavelet techniques have also been of particular interest. 3D adaptations have been made [144, Ch.10][69] that extend concepts such as JPEG2000 compression by applying adaptive arithmetic coding at the wavelet output.

Anisotropic (different in each direction) decomposition [40] proved to improve slightly coding efficiency by better decorrelating the images. Resolution progressive decoding [157] is also possible with these techniques. Pure JPEG2000 with its multicomponent coding approach has also been used, but found [238] to underperform against specifically developed 3D approaches.

Combinations with other transforms such as the Karhunen–Loève Transform (KLT) have been found to yield higher compression ratios since the spectral correlation is dealt with separately [95, Ch.9]. Spatial wavelet transforms, combined with spectral KLT and 3D coding [56] has shown some of the best lossy distortion-ratio results. Dimensionality reduction techniques such as PCA [144, Ch.11][57] have been applied before the spatial wavelet decorrelation. Resulting coefficients are coded with 2D techniques (assuming spectral decorrelation is gone) or 3D adaptations of those techniques. The high computational cost of PCA has also prompted the development of segmented PCA techniques, working on subsets of bands [95, Ch.10] reducing computational cost and improving compression performance by working on more correlated data. Tensor decomposition [239] has also been tried as a 3D approach, yielding results with slightly higher distortion-ratio performance than PCA, but at a higher computational burden.

2D techniques such as EBCOT from JPEG2000 [162] have been used as 2D encoders. 3D adaptations include Embedded Zerotree Wavelet (EZW) [185] where full embedded streams are achieved by progressively coding the most important coefficients in a wavelet-transformed image. Coefficients are assumed to be zero, and in the event of finding ones, those are then coded. Coding starts in the block with most wavelet passes, and the rest depend spatially on it, referencing previous values for improved coding efficiency. SPIHT (Set Partitioning In Hierarchical Trees) [173] is an improved technique which smartly traverses the transformed coefficients to improve the embedded quality reconstruction, and HW-friendly modifications [95, Ch.4] have also been developed that require almost no memory.

Compression techniques have been found to affect analysis of the decompressed data. Efforts have been made [144, Ch.7] to improve VQ that still allow for near-perfect classification. More generally, outliers can be the cause of classification errors, and their independent coding [95, Ch.5][219] has also been explored to avoid the overhead that results from coding infrequent

values. Region of interest (ROI) coding has also been used in the lossy domain to improve quality [186] of the most interesting parts of images.

3.1.2 Selection of algorithms

With all of these options to choose from, the selection of algorithms is based on variety and preliminary results given in their respective papers. As for variety, a lossless, near-lossless, and lossy algorithms were selected. Result-wise, the lossless algorithm is a robust international standard, the near-lossless one has a simple but very efficient flow aimed at high-speed implementations, and the lossy algorithm is a custom design based on the best results from the literature that separately deal with spectral and spatial correlations.

The CCSDS 123.0-B-1 [32] gives a good starting point to analyze the impact of multiple parameters at the hardware level on FPGAs. Both on resource use, and on achievable speed, since many times the techniques used for an algorithm are interchangeable with others.

Next, a custom PCA+JPEG2000 based-algorithm is developed exploiting the great results in [57], in order to explore what techniques might be of interest to further increase distortion-ratio. At the core, a JPEG2000 Tier 1 Coder is developed for FPGAs to accelerate an otherwise CPU-affine algorithm.

Finally, the simplicity of [5] is explored for near-lossless compression, serving as a link to fill the gap between lossless and lossy compression.

3.1.3 Terminology

Certain terminology will be used specifically for compression of hyperspectral images, which will be based on that used by the CCSDS 123.0-B-1 standard [32].

First off, when referring to any kind of hyperspectral image, its samples (that is, information about each wavelength at each pixel) are being compressed. To identify a sample, three indices z, y, x are used, where y, x indicate the spatial coordinates, and z the spectral one. Thus, a sample $s_{z,y,x}$ is a sample located at coordinates (z, y, x). All three indices move within the following ranges:

$$z \in [0, N_Z - 1] \quad y \in [0, N_Y - 1] \quad x \in [0, N_X - 1] \tag{3.1}$$

where N_X, N_Y, N_Z are the image dimensions in number of pixels per frame, frames, and bands respectively. Sometimes, to ease notation, only one index t will be used to refer to the spatial coordinates $t = x + y \cdot N_X$, referring to a sample as s_t^z. Samples can also be referred to as $s_{y,x}^z$ or $s_z(t)$ for typesetting purposes. A sample's range is defined by quantities $s_{\min}$, $s_{\max}$ and s_{mid}, which indicate the minimum, maximum and intermediate value that a sample can take. For unsigned values, $s_{\min} = 0, s_{\max} = 2^D - 1, s_{\mathrm{mid}} = 2^{D-1}$, where D is the bit depth of the sample. For signed values, $s_{\min}$ is added to every sample to bring them up to unsigned values, so the same operations and algorithm can be used on both after normalization.

A different terminology for sub-indices is also used for consistency with existing literature, where instead of variables z, y, x, variables i, n, m are used in reverse order (e.g: $s_{m,n,i} = s_{z,y,x}$).

Common operations that take place in compression algorithms are:

- $\mathrm{mod}_R^*[x] = \left(\left(x + 2^{R-1}\right) \mod 2^R\right) - 2^{R-1}$, i.e: the result of storing an integer x in an R bit register in two's compliment, modulo overflow.
- $clip\left(x, \{x_{\min}, x_{\max}\}\right)$ is the result of clipping x to the interval given by $[x_{\min}, x_{\max}]$.
- $\mathrm{sgn}^+(x)$ is equal to 1 for $x \geq 0$ and -1 otherwise. It is used to normalize signed values to unsigned ones via multiplication. (i.e: $x \cdot \mathrm{sgn}^+(x) \geq 0 \quad \forall x$).

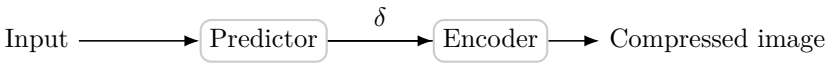

Figure 3.1: Compressor overview

3.2 CCSDS 123.0-B-1

The Consultative Committee for Space Data Systems (CCSDS) was founded in 1982 by the major spacial agencies in the world, to serve as a forum to discuss problems in development and operation of space systems. Over a hundred members are registered, including the most important space agencies in the world such as NASA (USA), ESA (Europe), JAXA (Japan), RSFA (Russia) and many others.

The CCSDS has been actively developing standard recommendations since its inception. The objective is to promote collaboration and interoperability between different agencies and their systems, both in planned and contingency scenarios that might be critical in space.

CCSDS 123.0-B-1 [32] is one of many standards that have been developed by CCSDS. It targets lossless compression of hyperspectral images, to compress images before transmission. The benefits include reducing bandwidth of down-links, reducing storage and buffering requirements on-board, and reducing transmission time for a given bandwidth. [1]

It is based on the FL (Fast Lossless) hyperspectral compression algorithm [119], using only integer arithmetic. This helps reduce computational load, simplifying the algorithm for implementation on constrained systems such as satellites.

3.2.1 Overview

Input data arrives at the compressor in a sequential manner. The exact ordering of input data is not important, but it has to satisfy one property: whenever a sample is to be processed, there is a neighborhood which has to be available in order to make a prediction. All of the samples in that neighborhood need to be already processed in order to have neighborhood availability. All sensors deliver data in a way that's compatible with this ordering, so usually this requirement is fulfilled automatically since the compressor is set right at the output of the sensor.

After prediction based on linear models, differential coding is performed on the differences between predicted and real values. These differences are losslessly coded with an adaptive entropy coder. This predictive compression (Figure 3.1) is a variant of the differential code pulse modulation (DCPM) in [51].

3.2.2 Predictor

The algorithm will be first described for real-valued samples. An integer version is later defined and used due to its reduced complexity while retaining numeric similarity.

Prediction is done on a single pass over the input data. The pass has to only ensure that previous samples for a neighborhood are already processed. This requirement forces each band to be traversed in raster-scan order. Since statistics are kept per-band, this means that with any traversal order, both the differences and entropy coder statistics will always be the same (even though their order might be different). Thus, compression *size* will not depend on scan order, even if the compressed *files* do change depending on it.

[1] A new version is available named CCSDS 123.0-B-2 [34]. It brings the possibility of lossy compression while keeping the original lossless flow. Since the algorithm is used solely for lossless compression, all references will be made to the first version [32].

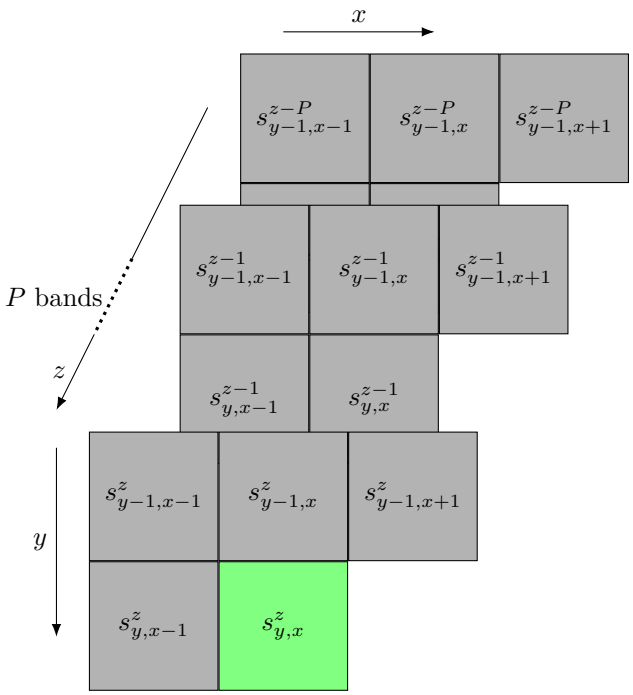

Figure 3.2: Sample neighborhood used for predictions

The neighborhood shown in Figure 3.2 is used for prediction. Based on neighboring samples of $s_{z,y,x}$ from P previous bands (P can be configured), the predicted sample $\hat{s}_{z,y,x}$ will be obtained, as well as the prediction residual $\delta_{z,y,x}$.

NOTE: All formulas assume that certain neighboring values are present (previous band, line or frame). For some samples this is not true (e.g: the first sample of the image has no neighborhood). Formulas are adjusted in those cases. To avoid writing all of them down, the general case is used here. For information about how they are adapted to boundaries, check [32].

First, a local mean $\mu_{z,y,x}$ is calculated to give a first estimation of what the actual value might be. There are two ways of calculating this value, either using the more expensive but precise neighbor-oriented way in Equation (3.2):

$$\mu_{z,y,x} = \frac{1}{4}\left(s_{z,y,x-1} + s_{z,y-1,x-1} + s_{z,y-1,x} + s_{z,y-1,x+1}\right) \tag{3.2}$$

or the cheaper but less precise column-oriented way in Equation (3.3):

$$\mu_{z,y,x} = s_{z,y-1,x} \tag{3.3}$$

Figure 3.3 shows both neighborhoods.

Using the generated $\mu_{z,y,x}$ value and the neighborhood, a difference vector is generated:

$$\Psi_{z,y,x} = \begin{pmatrix} s_{z,y-1,x} - \mu_{z,y,x} \\ s_{z,y,x-1} - \mu_{z,y,x} \\ s_{z,y-1,x-1} - \mu_{z,y,x} \\ s_{z-1,y,x} - \mu_{z,y,x} \\ s_{z-2,y,x} - \mu_{z,y,x} \\ \vdots \\ s_{z-P^*_z,y,x} - \mu_{P^*_z,y,x} \end{pmatrix} \tag{3.4}$$

where $P^*_z = min\{P, z\}$, adjusting to boundaries where previous bands are not available. Note that, Equation (3.4) gives the vector for *full prediction* mode. Under *reduced mode*, the first three

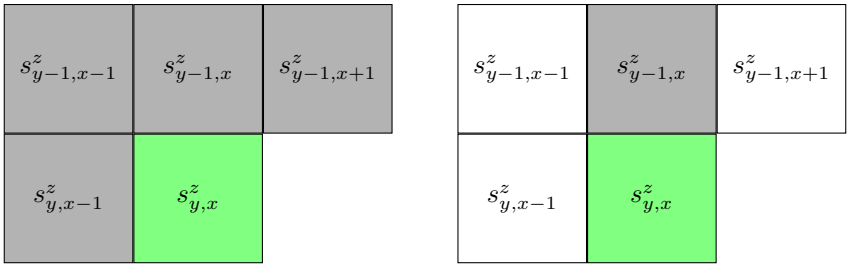

Figure 3.3: Sample neighborhood for $\mu_{z,y,x}$ calculation

entries, corresponding to the *north*, *west*, and *northwest* differences are not present. Reduced mode is used for not only simplifying operation but also eliminating some dependencies in the critical paths.

The predicted value $\hat{s}^*_z(t)$ is equal to the local mean plus a weighted sum of the local differences:

$$\hat{s}^*_z(t) = \mu_z(t) + \boldsymbol{V}^T_z(t)\boldsymbol{\Psi}_z(t) \tag{3.5}$$

where $\boldsymbol{V}^T_z(t)$ is a weight vector of the same size as $\boldsymbol{\Psi}_z(t)$. A vector is kept per band. These vectors are updated after each $\hat{s}^*_z(t)$ using the following formula:

$$\boldsymbol{V}^T_z(t+1) = \boldsymbol{V}^T_z(t) + \mathrm{sgn}^+(\epsilon_z(t)) \cdot 2^{-\alpha(t)} \cdot \boldsymbol{\Psi}_z(t) \tag{3.6}$$

where $\epsilon_z(t) = s_z(t) - \hat{s}^*_z(t)$ is the prediction error.

If the predicted value is higher than the actual one, the sign will be negative and the weights will decrease. Otherwise, weights will increase. The speed at which the weights change is controlled by $\alpha(t)$. It starts at a small value and increases up to a user-set limit. This way the weights adapt faster at the beginning when statistics are not yet stable. This is a variant of the least mean square (LMS) algorithm, which finds the gradient of the mean square error and descends through it to minimize it.

3.2.2.1 Mathematical background

The LMS algorithm uses the gradient descent method to find weights that minimize the objective cost function $C_z(t)$, in this case the expected $E\{\cdot\}$ square of the error:

$$C_z(t) = E\left\{|\epsilon_z(t)|^2\right\} \tag{3.7}$$

However, if this cost function is used, then Equation (3.6) becomes:

$$\boldsymbol{V}^T_z(t+1) = \boldsymbol{V}^T_z(t) + \epsilon_z(t) \cdot 2^{-\alpha(t)} \cdot \boldsymbol{\Psi}_z(t) \tag{3.8}$$

which despite having faster convergence, has two products, which in hardware are expensive for real-time applications. So the cost function is simplified for:

$$C_z(t) = E\{|\epsilon_z(t)|\} \tag{3.9}$$

Applying the gradient descent method by taking partial derivatives of the weight vector entries:

$$\nabla_{\boldsymbol{V}^T_z} C_z(t) = \nabla_{\boldsymbol{V}^T_z} E\{|\epsilon_z(t)|\} = E\left\{\nabla_{\boldsymbol{V}^T_z}(\epsilon_z(t)) \cdot \mathrm{sgn}^+(\epsilon_z(t))\right\} \tag{3.10}$$

$$\nabla_{\boldsymbol{V}^T_z}(\epsilon_z(t)) = \nabla_{\boldsymbol{V}^T_z}(s_z(t) - \hat{s}^*_z(t)) \tag{3.11}$$

$$= \nabla_{\boldsymbol{V}^T_z}(s_z(t) - \mu_z(t) - \boldsymbol{V}^T_z(t)\boldsymbol{\Psi}_z(t)) = -\boldsymbol{\Psi}_z(t) \tag{3.12}$$

the following is obained:

$$\nabla C_z(t) = -E\left\{\boldsymbol{\Psi}_z(t) \cdot \text{sgn}^+(\epsilon_z(t))\right\} \tag{3.13}$$

where $\nabla C_z(t)$ is a vector pointing in the direction of the greatest descent at point (z, y, x). To find the minimum, it is subtracted from the current value:

$$\boldsymbol{V}_z^T(t+1) = \boldsymbol{V}_z^T(t) - \alpha(t) \cdot \nabla C_z(t) = \boldsymbol{V}_z^T(t) + \alpha(t) \cdot E\left\{\boldsymbol{\Psi}_z(t) \cdot \text{sgn}^+(\epsilon_z(t))\right\} \tag{3.14}$$

where $\alpha(t)$ is the step, chosen by the user. For this, the value $E\left\{\boldsymbol{\Psi}_z(t) \cdot \text{sgn}^+(\epsilon_z(t))\right\}$ is required. Instead, the following unbiased estimator is used:

$$\hat{E}\left\{\boldsymbol{\Psi}_z(t) \cdot \text{sgn}^+(\epsilon_z(t))\right\} = \frac{1}{N}\sum_{i=0}^{N-1} \boldsymbol{\Psi}_z(t-i) \cdot \text{sgn}^+(\epsilon_z(t-i)) \tag{3.15}$$

Equation (3.6) is derived by taking $N = 1$ in Equation (3.15) and substituting in Equation (3.14).

3.2.2.2 Adapting to integers

The algorithm described works on real numbers. Real number arithmetic is complex and costly when implemented on hardware: numbers take more bits to represent, and precision can be lost over successive operations. Also, samples received from sensors are quantized to integers, so it makes sense to stay in the same domain. To adapt to integers [33] the following steps are taken along the way:

The *local sum* $\sigma_{z,y,x}$ is taken instead of $\mu_{z,y,x}$, since σ will always be an integer. Care is taken to allocate two more bits than the maximum sample bit size to calculate it:

$$\sigma_{z,y,x} = 4\mu_{z,y,x} \tag{3.16}$$

The differences are also scaled by a factor of 4 to produce integers, creating the vector:

$$\boldsymbol{U}_{z,y,x} = 4\boldsymbol{\Psi}_{z,y,x} \tag{3.17}$$

The parameter Ω is the resolution (in bits) of the weight vector when dealing with integers. As such, the weight vector is scaled by 2^Ω and adjusted to $\Omega + 3$ bits. This is equivalent to restricting weights to the interval $[-4, 4]$ with a precision of $\Omega + 3$ bits. So instead of $\boldsymbol{V}_z(t)$ the following is used:

$$\boldsymbol{W}_z(t) \approx 2^\Omega \boldsymbol{V}_z(t) \tag{3.18}$$

The predicted sample value $\hat{s}_z^*(t)$ is calculated now with the integer-valued scaled predicted sample value $\tilde{s}_z(t)$:

$$\begin{aligned}
\tilde{s}'_z(t) &= \left\lfloor \frac{\text{mod}_R^*\left[\hat{d}_z(t) + 2^\Omega\left(\sigma_z(t) - 4s_{\text{mid}}\right)\right]}{2^{\Omega+1}} \right\rfloor + 2s_{\text{mid}} + 1 \\
\tilde{s}_z(t) &= \begin{cases} \text{clip}\left(\tilde{s}'_z(t), \{2s_{\min}, 2s_{\max}+1\}\right) & t > 0 \\ 2s_{z-1}(t) & t = 0, P > 0, z > 0 \\ 2s_{mid} & t = 0 \wedge (P = 0 \vee z = 0) \end{cases}
\end{aligned} \tag{3.19}$$

where s_{mid} is the middle point between the minimum and maximum values that a sample $s_{z,y,x}$ can take. The operation mod_R^* could potentially imply a loss of information for small values of R. This does not mean compression will be lossy, but means that compression might lose

efficiency. To avoid it, the minimum value R^* ([33, p. 4.2.5]) that ensures no overflow will happen is used:

$$R^* = \Omega + 2 + \left\lceil \log_2 \left((2^D - 1)(8P + \kappa) + 1 \right) \right\rceil$$
$$\kappa = \begin{cases} 1 & \text{reduced mode} \\ 19 & \text{full mode} \end{cases} \tag{3.20}$$

Both the predicted sample value, and prediction error are now integers and given by the following equations:

$$\hat{s}_z(t) = \left\lfloor \frac{\tilde{s}_z(t)}{2} \right\rfloor \tag{3.21}$$

$$e_z(t) = 2s_z(t) - \tilde{s}_z(t) \tag{3.22}$$

Weights are now updated following:

$$\boldsymbol{W}_z(t+1) = \text{clip}\left(\boldsymbol{W}_z(t) + \left\lfloor \frac{1}{2} \text{sgn}^+ \left[e_z(t) \right] \cdot 2^{-\rho(t)} \cdot \boldsymbol{U}_z(t) + 1 \right\rfloor, \{\omega_{\min}, \omega_{\max}\} \right) \tag{3.23}$$

where $\rho(t)$ substitutes $\alpha(t)$ as an integer-valued exponent:

$$\rho(t) = \text{clip}\left(v_{\min} + \left\lfloor \frac{t - N_X}{t_{\text{inc}}} \right\rfloor, \{v_{\min}, v_{\max}\} \right) + D - \Omega \tag{3.24}$$

being $\rho(t)$ bounded by:

$$-24 \leq v_{\min} + D_{\min} - \Omega_{\max} \leq \rho(t) \leq v_{\max} + D_{\max} - \Omega_{\min} \leq 21 \tag{3.25}$$

where $D_{\min}, D_{\max}, \Omega_{\min}, \Omega_{\max}$ are the minimum and maximum values that D and Ω can take. $\rho(t)$ can be stored in just a 6-bit register, while the register size that avoids overflow in Equation (3.23) is:

$$\max\left(\Omega + 3, D + 3 + \min\left(1, v_{min} + D - \Omega\right)\right) + 1 \tag{3.26}$$

After calculation, the prediction residual $\Delta_z(t) = s_z(t) - \hat{s}_z(t)$ has to be fed to the encoder. But the coder works only on unsigned integers, so an invertible mapping $\delta : \left[-2^{D-1}, 2^{D-1} - 1\right] \rightarrow \left[0, 2^D - 1\right]$ is performed beforehand. Any mapping works, but ideally, low values at the output are preferred. The following is used:

$$\delta_z(t) = \begin{cases} |\Delta_z(t)| + \theta_z(t) & |\Delta_z(t)| > \theta_z(t) \\ 2\,|\Delta_z(t)| & 0 \leq (-1)^{\tilde{s}_z(t)} \Delta_z(t) \leq \theta_z(t) \\ 2\,|\Delta_z(t)| - 1 & otherwise \end{cases} \tag{3.27}$$

where:

$$\Delta_z(t) = s_z(t) - \hat{s}_z(t)$$
$$\theta_z(t) = \min\left(\hat{s}_z(t) - s_{\min}, s_{\max} - \hat{s}_z(t)\right) \tag{3.28}$$

3.2.3 Encoder

All of the processes performed by the predictor are reversible, meaning that a stream of prediction residuals can be used to reconstruct the original image. The next step in the compression process is to losslessly encode those residuals.

For that, a variant of the Low Complexity Lossless Coder (LOCO) [214] is used. Each residual is coded using Golomb power-of-two (Section 2.2.2) codes since they follow a geometrical distribution. The golomb parameter m is selected on-the-fly based on statistics from the preceding residuals. Statistics are kept separate for each spectral band since usually they present different characteristics due to the way sensors are built, with different detectors for each band.

The length of each coded data is bounded by a maximum user-defined value U_{max}, that allows an easier hardware implementation since very low probability values will have a bounded maximum size. This is not true in the general case of encoder, where theoretically arbitrarily large codes might be generated.

To code a positive integer δ with this method, the following formula is used:

$$\delta = u \cdot 2^k + r \tag{3.29}$$

where u and r are the quotient and remainder, respectively, when dividing by 2^k. The resulting code will be the unary representation of u, followed by a zero, followed by r in binary format. If $u > U_{max}$, then U_{max} zeros are coded followed by δ in binary format.

The encoder has a counter $\Gamma(t)$ and an accumulator $\Sigma_z(t)$ which is different for each band. The quotient $\Sigma_z(t)/\Gamma(t)$ gives an estimation of $\delta_z(t)$, and is used to select the parameter k by using the largest value satisfying:

$$2^k \leq \frac{\Sigma_z(t)}{\Gamma(t)} + \frac{49}{128} \tag{3.30}$$

This formula is not random, and has been experimentally found to give the values for k that bring the best compression ratios [116].

3.2.3.1 Mathematical background

The counter $\Gamma(t)$ will always store the number of samples that have been accumulated in $\Sigma_z(t)$, which in turn contains an approximation of the summation of those samples, weighting the newer samples exponentially higher than the older ones (this exponential weighting is why golomb exponential codes are used). This method ensures that, while older samples are not taken out of prediction, they have less impact in what the following samples will be predicted as.

First, equations for both the counter and accumulator are defined:

$$\Sigma_z(t) = \begin{cases} \Sigma_z(t-1) + \delta_z(t-1) & \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \left\lceil \dfrac{\Sigma_z(t-1) + \delta_z(t-1)}{2} \right\rceil & \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \tag{3.31}$$

$$\Gamma(t) = \begin{cases} \Gamma(t-1) + 1 & \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \left\lceil \dfrac{\Gamma(t-1)}{2} \right\rceil & \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \tag{3.32}$$

where it can be clearly seen that the counter resets periodically (renormalizes) the accumulator statistics by halving its value.

Observe that counter values are the following (after $t = 1$):

$$2^{\gamma_0}, \ldots, 2^{\gamma^*} - 1, 2^{\gamma^*-1}, \ldots, 2^{\gamma^*} - 1, 2^{\gamma^*-1}, \ldots, 2^{\gamma^*} - 1, \ldots \tag{3.33}$$

A direct formula can be derived for the counter:

$$\begin{array}{rcl} \Gamma_0 & = & 2^{\gamma^*} - 2^{\gamma_0} \\ \Gamma_a(t) & = & t - (2^{\gamma^*} - 2^{\gamma_0} + 1) \bmod 2^{\gamma^*-1} \\ \Gamma(t) & = & \begin{cases} 2^{\gamma_0} - 1 + t & t \leq \Gamma_0 \\ 2^{\gamma^*-1} + \Gamma_a(t) & t > \Gamma_0 \end{cases} \end{array} \tag{3.34}$$

The amount of times $\eta(t)$ the accumulator and counter have been renormalized is given by:

$$\eta(t) = \begin{cases} 0 & t < 2^{\gamma^*} - 2^{\gamma_0} + 1 \\ \left\lfloor \dfrac{t - \left(2^{\gamma^*} - 2^{\gamma_0} + 1\right)}{2^{\gamma^*-1}} \right\rfloor & otherwise \end{cases} \tag{3.35}$$

Let $\ell = 2^{\gamma^*} - 2^{\gamma_0}$, then the following can be derived:

$$\begin{aligned}
\Sigma_z(t) &= \frac{\Sigma_z^Z(t)}{2^{\eta(t)}} + \sum_{i=0}^{\eta(t)-2} \frac{\Sigma_z^i(t)}{2^{\eta(t)-i-1}} + \Sigma_z^N(t) \\
\Sigma_z^Z(t) &= \sum_{i=1}^{\min(t,\ell)} \delta_z(i) \\
\Sigma_z^i(t) &= \sum_{j=1}^{2^{\gamma^*-1}} \delta_z(i \cdot 2^{\gamma^*-1} + j + \ell) \\
\Sigma_z^N(t) &= \sum_{i=0}^{(t-\ell-1) \bmod 2^{\gamma^*-1}} \delta_z(i + 1 + \ell + (\eta(t) - 1) \cdot 2^{\gamma^*-1})
\end{aligned} \tag{3.36}$$

At any point t, the set $\Sigma_z^Z(t)$ has been renormalized $\eta(t)$ times, the second set $\Sigma_z^0(t)$ has been renormalized $\eta(t)-1$ times and so on. Thus, these sets of values added between renormalizations, have each half the weight than the next one. The current one $\Sigma_z^N(t)$ has as much weight as all the previous ones (1 versus $\sum_{i=1}^{\eta(t)}(1/2)^i$), giving them an exponentially higher weight than older ones . The reason as to why this readjusting method is appropriate are found in [214, pp. 3.3.2–3.4].

3.2.4 Summary

Samples $s_{z,y,x}$ are fed sequentially to the compressor, with the constraint that samples in the same band (i.e: same value of z) are fed in raster order, and samples in the same pixel (i.e: same values of (x, y) are fed with increasing values of z). A sample s_{z_1,y_1,x_1} has to enter the compressor before a sample s_{z_2,y_2,x_2} if:

$$\begin{gathered}
z_1 < z_2 \; and \; x_1 = x_2, y_1 = y_2 \\
z_1 = z_2 \; and \; y_1 \cdot N_x + x_1 < y_2 \cdot N_x + x_2
\end{gathered} \tag{3.37}$$

ensuring that the neighborhoods are present for processing. A prediction is made based on that neighborhood, that is weighted dynamically adapting to changing statistics.

The differentials between the real and predicted values are then encoded using an adaptive entropy golomb power of two coder. Based on a expected geometrical distribution of the predictor's output, parameters are selected to optimally encode the differentials, generating the final output stream.

The whole process ensures that local similarities, both spatially and spectrally are exploited for sample prediction, creating models for the expected differences, which are used to approximate the predicted values as close to the real ones as possible.

Decoding is done in the same way. The process ensures that predictions can be made with already decoded samples, and instead of using the new raw value to generate the coded stream, the inverse is done and the decoded stream used to re-create the original values.

3.3 Jypec

When performing lossy compression on any kind of data, a prior analysis of its redundancy is of great interest. Different kinds of data contain different patterns and redundancies. Predictors used for compression can be tailored to this particular set of behaviors. Furthermore, when approximating data, these characteristics can be exploited to produce compressed results that differ very little from the originals.

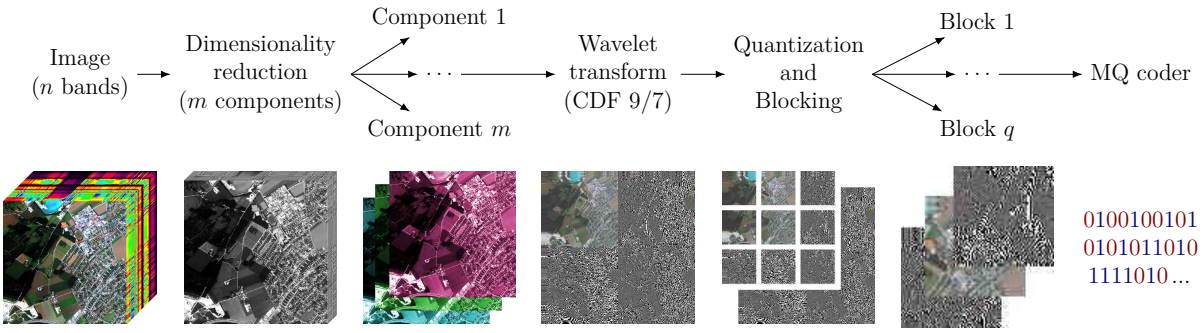

Figure 3.4: Flow of the algorithm. The image is progressively fragmented by removing correlations until small blocks can be coded and compressed.

Applying traditional image compression algorithms to each band of a hyperspectral image is a quick and straightforward way of compressing it [170]. However, advantage is only taken of the spatial correlation present in the image, while the spectral correlation is ignored. These two-dimensional algorithms have seen extensions to accommodate the spectral dimension [40, 157] with noticeable improvements in distortion-ratio performances. However, the spectral correlation is assumed to be similar to the spatial in this case, not taking full advantage of hyperspectral characteristics. Hybrid algorithms [57] that separately decorrelate both characteristics have proven to outperform the former distortion-ratio wise, while being simpler computationally.

Jypec is born as a lossy algorithm which aims to reach the highest possible compression ratios while still retaining visual fidelity. First, a dimensionality reduction algorithm is applied, followed by JPEG2000 compression on each band of the reduced image. Additionally, vector quantization can be done before the dimensionality reduction to create groups of pixels. These groups present similar characteristics and less information will be lost if reductions are performed individually on each of them. This is a similar idea to that presented in [57], but with a few modifications. Dimensionality reduction is extended beyond PCA, adding the possibility of a vector quantization step. Variable bit-depths are used for the different components, emphasizing the information of the ones with the most variance. Lastly, a custom JPEG2000 implementation is used that avoids headers and markers, compressing the image even further.

The process is seen in Figure 3.4. Information is lost on the dimensionality reduction step and on the wavelet transform (when quantizing the result). Coding is lossless. Parameters in both the reduction and wavelet steps will control quality as well as compression ratio. Coding configuration will allow for higher compression ratios.

A few considerations before defining the different steps are that:

- The stream in JPEG2000 is progressively decodeable thanks to careful ordering of the output blocks. This is deemed unnecessary for hyperspectral images since full-image compression is being targeted for bulk storage and transmission. Some markers are saved this way, increasing compression ratio.
- The JPEG2000 standard does provide a dimensionality reduction method for images with more than three components per pixel [100]. However, specific techniques will be used that have been found to perform well with hyperspectral data.
- Both the wavelet transform and arithmetic coders are those of JPEG2000.

3.3.1 Dimensionality reduction

Dimensionality reduction has already been explored for hyperspectral image compression, but few methods have been tried. By including multiple options, the aim is to verify the validity

of the current ones being used, and to find ones that outperform the others. The following dimensionality reduction algorithms have been included in Jypec's flow: **ICA**, **MNF**, **PCA**, **SVD**, **VCA** y **VQPCA** (see Section 2.3.5).

All methods generate projection and reconstruction matrices. The projection one is used for compression, and the reconstruction matrix is used for decompression. The latter has to be stored in the compressed stream as well as the compressed data. VQPCA uses not one but many matrices corresponding to each of the quantized clusters. The C vector, as well as the matrices, is needed to assign a reconstruction matrix to each of the samples.

The general flow is as follows:

- A **preprocessing** step is first performed to select a subset of pixels of the input image. These are the ones used for creating the matrix. The size of this subset determines the time it takes to create the projection matrix. A bigger subset will imply longer calculation times but will more closely resemble the original. This step is done just to reduce the algorithm time. The parameter which controls the portion of the pixels used for preprocessing is called t, and will usually be around 0.01 (1% of total).
- A **training** step comes right after, and uses the selected data to create the projection matrix. Additionally, it centers the data around its mean value, improving the performance of the dimensionality reduction methods.
- The **reduction** step is the last one in the pipeline. It centers the raw data around the average vector and then performs the projection. The inverse will be done when reconstructing: the data will be projected back with the reconstruction matrix, and then decentered. Training and preprocessing are only done when compressing, so decompression skips them.

First, the data X is subsampled according to the parameter t. Let $s_X = |X|$. A random list of unique indices $I_X = \{i_1, \ldots, i_{s_X}\}$ is generated, and a subsampled set X_s of size $s_{X_s} = |X_s|$ is generated:

$$X_s = x_i \in X \shortparallel i \in I_X \tag{3.38}$$

Its mean value is obtained as:

$$\bar{\boldsymbol{x_s}} = \frac{\sum_{x_s \in X_s} x_s}{s_{X_s}} \approx \frac{\sum_{x \in X} x}{s_X} = \bar{\boldsymbol{x}} \tag{3.39}$$

The original input data is then centered following:

$$X_c = \{x - \bar{\boldsymbol{x_s}}\}_{x \in X} \tag{3.40}$$

After obtaining $X_c \in \mathcal{M}_{s_X \times n}$, it is processed with one of the algorithms described in Section 2.3.5, obtaining a transformed matrix $T \in \mathcal{M}_{s_X \times m}$ of transformed data. The value of m selected will directly impact compression ratio in this step. If the original dataset X had $s_X \times n$ total samples, T has just $s_X \times m$ samples. So a compression ratio of n/m is achieved at this step. Further steps will of course affect this value.

The output from all algorithms is a transformation matrix W and recovery matrix $\bar{W}$. W is directly used to transform the data into T, while $\bar{W}$ gets stored in the compressed stream to later be used to decompress the data. VQPCA works differently since it outputs many W matrices (one per cluster) as well as the list C of what cluster each sample x belongs to. Within clusters, the algorithm performs the projections just as in the rest of algorithms, and saves the $\bar{W}$ matrices for reconstruction.

Now, the data from X has been spectrally decorrelated in T. T is then separated into bands $T_i = [T_{x,y,i} \forall x, z]$. Each of these bands will be processed by the JPEG2000 algorithm.

3.3.2 Outlier detection

Every dimensionality reduction algorithm does a transformation that globally achieves the best possible results. This is done based on the general characteristics of the input samples (i.e: pixels). Outliers in the input set will probably still be outliers in the output set.

These outliers might dissapear in the JPEG2000 flow, since the wavelet transform and quantization afterwards smooth out values that are too different from their neighbors. Still, those outliers might be critical for image processing, were they are precisely the spots that, for example, anomaly detection algorithms look after.

The dimensionality reduction algorithm will ideally have a target number of dimensions that matches closely the real underlying number of dimensions in the original data. That is where it will reach the best distortion-ratio performance. Each transformed band can be assumed mostly independent from each other, and so the detection of outliers is done per band.

To keep those values that otherwise are lost in the JPEG2000 compression flow, a threshold $p_o \in [0, 1]$ is set. It indicates the percentage of pixels that is to be saved losslessly (losslessly with respect to JPEG2000, since the dimensionality reduction step has already induced some loss). To that end, the average value of a band is obtained b_{avg}. All samples $b(t)$ in that band are sorted in descending order according to the difference $|b(t) - b_{avg}|$. For a number $N_X \times N_Y$ of samples in a band, the first $p_o \cdot (N_X \times N_Y)$ are saved losslessly by storing their value and coordinates in raw form.

Afterwards, the minimum and maximum value of the non outliers are calculated, and all samples in that band are clamped to that interval. Compression proceeds as usual. Note that if $p_o = 0$, no outlier detection is performed.

Outlier detection is specially useful since a normalization step occurs for the JPEG2000 wavelet transform to the interval $(-0.5, 0.5)$. If some samples deviate excessively from the rest, the rest might be clumped together and thus when quantizing they might fall in the same bin, losing information that is otherwise kept. This is avoided by storing the outliers in this manner, which improves separation after the normalization step.

3.3.3 JPEG2000

JPEG is a simple algorithm that took advantage of the mathematical properties of the DCT. It splits the image in 8×8 blocks, applying the DCT to each one individually. High-frequency components were removed, retaining as much low-frequency ones as required for a specific quality. This was (and still is) the selected approach for many situations were simplicity is preferred over newer methods with higher distortion-ratio performances.

JPEG2000 was born as a successor to JPEG in the Internet era. One of the main motivators was that JPEG2000 could be progressively decoded, meaning prefixes of the compressed stream could be used to approximate the whole compressed image. This was due to the use of wavelet transforms that applied to the whole image instead of the DCT that only targeted blocks. By cleverly arranging the wavelet coefficients in the output stream, the whole image can be progressively reconstructed. Another advantage is that the encoder works over longer runs of data, adapting more precisely to statistics.

These two main differences resulted in higher quality at the same bit rates (see Figure 3.5), however the processing power required grew considerably, which has kept JPEG2000 under the radar for a while. With computing power growing, it has again found its way to consumers and specially scientific applications, where the blockiness of JPEG can be detrimental to experiments.

JPEG2000 has three main steps. A color space transform (done by the dimensionality reduction step), followed by a wavelet transform on each color channel, and finally the encoding of the transformed data via the tier 1 and 2 coders.

The tier 1 coder consists on the bit plane coder and the arithmetic MQ-coder, which will later be seen. It losslessly compresses small blocks of data that have been processed by the wavelet transform and are full of redundancies. The output of the tier 1 coder has a special property: any prefix can be decoded, and gives an approximation of the final decoded block. This is useful for progressive reconstruction of the image, however if blocks are coded sequentially, while the property still holds locally per block, it cannot be applied to the whole image.

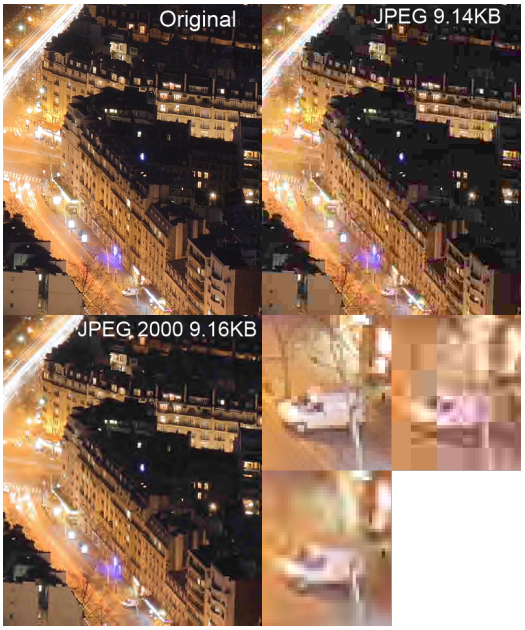

Figure 3.5: The difference [105] between JPEG and JPEG2000 compression can be seen, as the blockiness at high compression ratios is clear.

Thus, the tier 2 coder works on top of the tier 1 coder streams, intertwining them to create a progressive stream over the full image. This is interesting in a streaming scenario such as the Internet, however it adds unnecessary complexity for this use case, where the full image is always expected to be decompressed for analysis purposes. Thus, only the tier 1 coder is implemented.

In the following sections, the different parts of the algorithm: wavelet transform, quantization, and tier 1 coding (including block coding and MQ-coding) are explained, forming the JPEG2000 part of the JYPEC algorithm.

3.3.3.1 Wavelet transform

Recall that data from the dimensionality reduction step is stored in matrix T:

$$T = [\boldsymbol{t}_1, \ldots, \boldsymbol{t}_p], \quad \boldsymbol{t}_i \in \mathbb{R}^m \tag{3.41}$$

The wavelet transform is applied to each transformed band, so T can be divided in bands $B^k, k \in 1, \ldots, m$ as follows:

$$B^k = \left[\boldsymbol{t}_1^k, \ldots, \boldsymbol{t}_p^k\right] = [b_1, \ldots, b_p] \tag{3.42}$$

Even though for dimensionality reduction, data (X, T, B) were represented as vectors, it is useful to also see those as matrices. So, if the hyperspectral image size is $N_X \times N_Y \times N_Z$, where $N_X \times N_Y = N_P$, and $N_Z = n$ for X, $N_Z = m$ for T. Define:

$$B^k = \left[\boldsymbol{t}_{i,j}^k\right] = [b_{i,j}], \quad i \in \{1, \ldots, N_X\}, j \in \{1, \ldots, N_Y\} \quad b_{i,j} = b_{i*w+j} \tag{3.43}$$

The wavelet transform used is applied in both directions (horizontal and vertical). For that, two kernels (Definition 2) are applied:

$$K_h = \begin{bmatrix} 0.026 & -0.016 & -0.078 & 0.266 & 0.602 & 0.266 & -0.078 & -0.016 & 0.026 \end{bmatrix} \tag{3.44}$$

$$K_l = \begin{bmatrix} 0.091 & -0.057 & -0.591 & 1.115 & -0.591 & -0.057 & 0.091 \end{bmatrix} \tag{3.45}$$

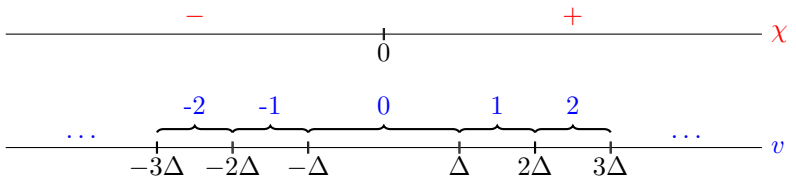

Figure 3.6: Uniform dead-zone quantizer at work. A sign-magnitude (χ, v) representation is obtained that generates a bin double the size of the rest around zero. That is the dead-zone.

The wavelet represented by these kernels is called CDF 9/7 [41], and was designed originally to operate on infinite signals. In this case, input vectors $\boldsymbol{v} = [v_1, \ldots, v_n] \in \mathbb{R}^n$ are extended at the borders as follows:

$$v_i = \begin{cases} v_i & 1 \leq i \leq n \\ v_{2-i} & i < 1 \\ v_{n-(i-n)} & i > n \end{cases} \tag{3.46}$$

From applying both lowpass and highpass kernels, the following is obtained:

$$K_h(\boldsymbol{v}) = \boldsymbol{v}^h \quad K_l(\boldsymbol{v}) = \boldsymbol{v}^l \tag{3.47}$$

which in turn create the transformed vector:

$$\boldsymbol{v}' = \left\{ \left\{ \boldsymbol{v}_i^l \right\}_{i \in 2\mathbb{N}}, \left\{ \boldsymbol{v}_i^h \right\}_{i \in 2\mathbb{N}} \right\} \tag{3.48}$$

This process is repeated for each line and each column of the input matrix. It can be then recursively applied over the LL sub-band if wanted.

3.3.3.2 Quantization

The CDF 9/7 used is a lossy transform that works on real numbers. Quantization is necessary to bring data to the integer domain. Results from the wavelet transform lie on the $(-1/2, 1/2)$ domain. The output interval after quantization will lie in the range $[-2^{n_b} + 1, 2^{n_b} - 1]$ (Note that the lower bound is not -2^{n_b} due to the deadzone quantizer used). Higher values of n will increase quality while at the same time lowering compression ratio.

To quantize the values, the following simple quantizer is used:

$$q_z(s_{z,y,x}) = \operatorname{sgn}^+(s_{z,y,x}) \left\lfloor \frac{|s_{z,y,x}|}{\Delta_b} \right\rfloor \tag{3.49}$$

This is the so-called *uniform dead-zone quantizer* with quantization step $\Delta_b = 2^{-n_b}$ (See Figure 3.6. Note that index b indicates that each band can have a different quantization step. Since the wavelet output lies in the $((-1/2, 1/2))$ interval, the number of possible quantized values will be $2^{n+1} - 1$, for which bits will be allocated. A finer quantization will inevitably lead to more bits used.

Functions can be applied prior to quantization to improve separation of samples, avoiding the collapse of similar samples in the same quantization bins at coarser quantization levels. This aids in lowering the reconstruction error. To pre-quantize a set of samples X, some pre-quantization functions are:

$$f_{log}(x) = \begin{cases} \log(x - \bar{x} + 1) & x > \bar{x} \\ \log(\bar{x} - x + 1) & x < \bar{x} \\ 0 & x = 0 \end{cases} \quad f_{sqrt}(x) = \begin{cases} \sqrt{x - \bar{x}} & x > \bar{x} \\ \sqrt{\bar{x} - x} & x < \bar{x} \\ 0 & x = 0 \end{cases} \quad f_{lin}^r(x) = \frac{x}{r} \tag{3.50}$$

where $\bar{x}$ is the mean value of the set X.

3.3.3.3 Block coding

Following quantization, each band is then coded. Blocks of 64×64 are taken following the standard maximum size (with size lowered at the edges if necessary) to lower memory loads.

Each block is coded in a way that allows it to be progressively decoded. That way, the progressive nature of the wavelet is also brought into the coding phase. For each block, its bit planes (the sets of bits of the same significance for all samples) are coded from high to low significance. Three pases will be done over each plane, in which bits are prioritized and those that are predicted to be more important are coded first.

A few variables are introduced to help when describing the algorithm:

j Position within a block. It is an abstraction of a two-dimensional coordinate to simplify notation.

$y[j]$ Sample at position j.

$v^{(p)}[j]$ pth bit of sample $y[j]$.

$\chi[j]$ sign of sample $y[j]$.

A predictive model that works on individual bits is at the core of the coding phase. Instead of working on full samples, like CCSDS 123.0-B-1, predictions will be made on individual bits and later an arithmetic coder will be used to encode them.

Definition 12 *$\sigma[t]$ is the **significance** of a sample $y[t]$, and can take the three following values: Insignificant, positive significant and negative significant.*

*As long as every bit $v^{(p)}[j]$ up to a certain point p that has already been coded are **0**, a sample is insignificant. As soon as a **1** is coded for a sample, it then becomes significant of the same sign as the sample, given by $\chi[j]$.*

Since bits are coded from more to less significant, the significance of a sample indicates, at any point, if that sample's contribution is of interest so far.

This concept of significance is what the three passes (Figure 3.7) of the algorithm are based on. Within each bit plane, some bits (depending on neighboring conditions as seen later) will have a higher chance of being "interesting" than others. These skewed probabilities will be taken advantage of when coding, and will aid in creating a progressively decodeable stream.

Significance propagation pass: Samples that are insignificant but are believed to turn significant this pass are coded as Algorithm 1 shows. If a sample's neighborhood has many significant samples, the actual sample will be included in this pass. The sign is coded in this pass, and when doing so a special XOR bit, associated to the context, is used to improve compression.

Refinement pass: When a sample is already significant, the behavior of the bits that are not yet coded is considered mostly random. These bits are coded in a special pass that does not have a model as skewed as the other two. Special care is taken for the first refinement pass compared to the others, since that still contains a fair bit of predictability. This is seen in Algorithm 2.

Cleanup: What was not coded in previous passes is coded here. Generally, that means zones that are mostly zeros. Thus, this pass will include a run-length coder to deal with such behavior in the most efficient manner, shown in Algorithm 3.

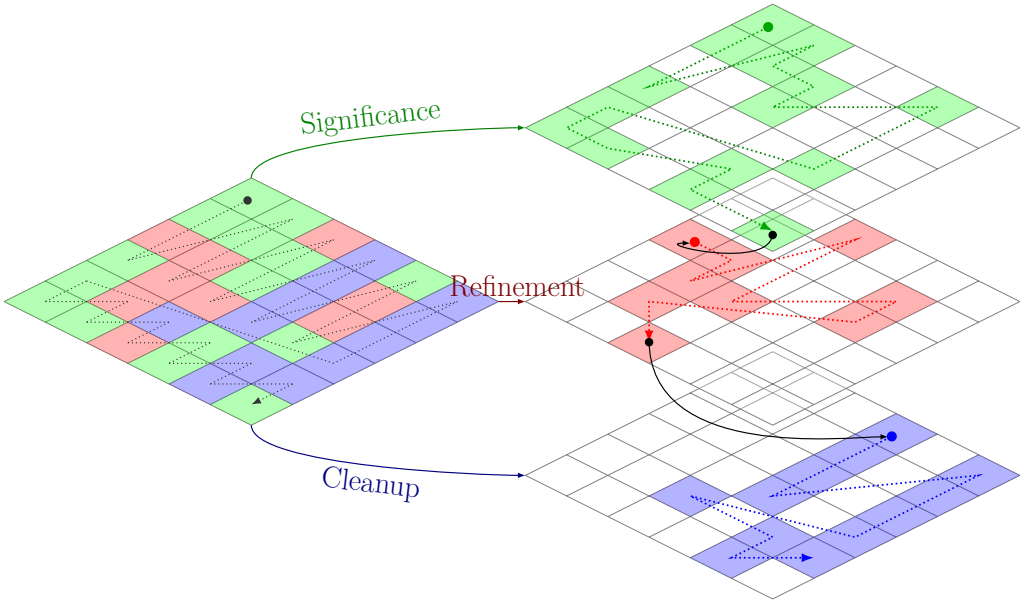

Figure 3.7: Each bit plane is coded in three passes. A zig-zag pattern of height 4 is followed to traverse it. This ensures areas that are physically close are visited in quick succession to maximize predictability.

Algorithm 1: EBCoder.codeSignificance

```
1  for each bit b in the zig-zag traversal of plane do
2      context ← significance context of b;
3      if b is not significant, and context ≠ ContextZERO then
4          mqCoder.code(b, context, bitStream);
5          if b is 1 then
6              b_s ← b's associated sign bit;
7              context ← b_s's associated sign context;
8              x_s ← context's associated XOR bit;
9              mqCoder.code(b_s ⊕ x_s, context, bitStream);
10         end
11         Set b as already coded in plane;
12     end
13 end
```

Algorithm 2: EBCoder.codeRefinement

```
1  for each bit b in plane's zig-zag traversal do
2      if b has not been coded and is significant then
3          context ← b's refinement context;
4          mqCoder.code(b, context, bitStream);
5          Set b as already coded in plane;
6      end
7  end
```

Algorithm 3: EBCoder.codeCleanup

```
1  for Each bit b in the zig-zag traversal of plane do
2      if b is the first of its column, the whole column is uncoded, and every context is
        ContextZERO then
3          if Every bit in the column is zero then
4              mqCoder.code(0, ContextRUNLENGTH, bitStream);
5          else
6              mqCoder.code(1, ContextRUNLENGTH, bitStream);
7              j ← index of the first non-zero bit within the column;
8              (b_0^j, b_1^j) ← two-bit representation of j;
9              mqCoder.code(b_0^j, ContextUNIFORM, bitStream);
10             mqCoder.code(b_1^j, ContextUNIFORM, bitStream);
11             b_s ← sign bit of the sample at position j;
12             context ← b_s's sign context;
13             x_s ← context's associated XOR bit;
14             mqCoder.code(b_s ⊕ x_s, context, bitStream);
15             Skip to bit in position j + 1 and go to 2;
16         end
17     else
18         Code b as if it was in the significance pass (Algorithm 1);
19     end
20 end
```

When processing a block, the first plane is only processed with a cleanup pass (since it is expected to have many zeros). After that, all three passes are used for all planes in the order they've been described. While at first the cleanup pass is the one that codes most bits, for the last bit planes, significance and then refinement are the ones that process the most bits. This is seen in Algorithm 4.

Algorithm 4: EBCoder.code

```
   input  : A block with n_p bit planes
1  for i ← 0 to n_p − 1 do
2      plane ← block.getPlane(i);
3      if i ≠ 0 then
4          significancePass(plane);                    /* Algorithm 1 */
5          refinementPass(plane);                      /* Algorithm 2 */
6      end
7      cleanupPass(plane);                             /* Algorithm 3 */
8  end
```

There is a special plane that is not processed in these passes: the sign plane. Recall that samples are in sign-magnitude format. All magnitude planes are coded with the aforementioned passes, while the sign is only coded whenever a sample turns significant, since it does not bear any information before that point. Thus, sign bits intertwine with the others.

On magnitude planes, a zig-zag pattern is followed for traversal. Rows are grouped in sets of four, which are traversed column-wise (Figure 3.8). This ensures neighboring bits follow each other for coding, which will exploit its similarities. Bits are marked after each pass, ensuring they are coded only once.

These similarities between neighboring bits are what allows predictive models to perform well. The significance and cleanup passes are the ones that are more redundant and can take the most

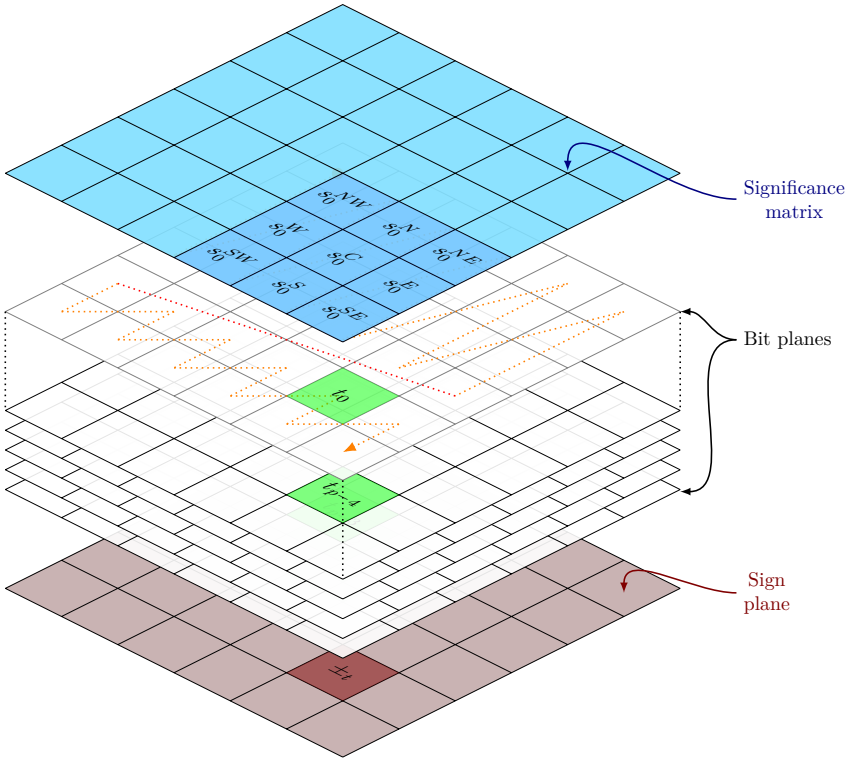

Figure 3.8: Block coding scheme. The zig-zag pattern is seen. Bit planes are coded from more to less significant (D to 0 assuming a depth of $D+1$). All while using contexts given by the significance matrix, where 3×3 neighborhoods are checked.

advantage for this, though the refinement pass will also have a smaller degree of predictability. There are many predictive models working at once for coding a block, these are based on the so-called contexts.

Definition 13 *A **context** represents a pattern in the data that is to be coded. Contexts are generated [201] based on the significance state of neighboring samples (Figure 3.8). The sub-band that is being coded (LL, LH, HL, HH) is also taken into account to improve predictability. 3^8 different contexts might exist, but for simplicity they are collapsed to just 16 different ones (Figure 3.9). Different contexts are used for the different passes.*

Bits with a certain context are thus expected to always have the same behavior. That is, to have the same probability of being either `0` or `1`. So, for a given context, its bits are expected to be heavily skewed towards either zero or one. And what is ideal for coding a heavily skewed binary distribution? An arithmetic coder.

3.3.3.4 The MQ arithmetic coder

Bit-context pairs are generated by the block coder, but despite its name it does not perform the coding itself. The block coder is just an entropy reduction step, while the arithmetic MQ-coder

Figure 3.9: Example of different neighborhoods that generate the same context. Significant samples are shown in red, insignificant in black. This context is used when a certain vertical and/or diagonal pattern is found.

[201, p. 12.1] does the bit reduction. A general diagram of this module is shown in Figure 3.10, and the full process in Algorithm 5.

Algorithm 5: MQCoder.code

input: A bit-context pair (b context)
input: A table mapping contexts to states
inout: A bitStream where coded data is dumped

```
1  state ← table [context ];
2  pred ← prediction associated to state;
3  prob ← probability from the probability table associated to state;
4  A ← A − prob;
5  if prob falls outside of the new subinterval (A < prob) then
6      Change pred to the opposite bit, so the wide subinterval is selected in case of success;
7  end
8  if adjusted prediction was successful then
9      C ← C + prob stay in the wide subinterval;
10 else
11     A ← prob stay in the narrow subinterval;
12 end
13 if renormalization is required then
14     if original unadjusted prediction was successful then
15         Updated state using the most probable symbol table;
16     else
17         Update state using the least probable symbol table, applying the XOR bit if
            required;
18     end
19 end
20 while renormalization is required do
21     C ← C ∗ 2;
22     A ← A ∗ 2;
23     t̄ ← t̄ − 1;
24     if t̄ = 0 then
25         Create a new byte with C's upper 8 bits;
26         if reserved sequence can be formed on next byte output then
27             t̄ = 7, forcing zero at the beginning of the next byte, avoiding 0xff;
28         else
29             t̄ = 8, waiting for 8 bits for a new byte;
30         end
31     end
32 end
```

This binary arithmetic coder takes the bit-context pairs. It exploits the fact that the input distribution is skewed, but the skew is not known beforehand, so it is of the *adaptive* type. Not only that, it will adapt to every input distribution (one per context) at the same time. How does it do it?

For each context κ, two values are maintained: A symbol $s_\kappa \in \{\texttt{0}, \texttt{1}\}$ (the current prediction), and an integer $\Sigma_\kappa \in \{0, \ldots, 46\}$, indicating the **state** of the predictor for that context. Whenever the prediction is correct, the state changes to one with more skew towards the predicted bit.

To determine the probability of the prediction s_κ being correct for a certain state, a table associates to each state a probability $\bar{p} \in \left\{0, \ldots, 2^{16} - 1\right\}$ that is mapped to the $[0, 1)$ interval by dividing by 2^{16}.

Since probabilities are fixed in the table, state change is made by using transition tables. These indicate which state to transition to, depending on if the prediction is right Σ_{mps} (*most probable symbol*) or wrong Σ_{lps} (*least probable symbol*).

To "double" the number of states, a table X_s (*switch* table) indicates for a state Σ_κ if prediction s_κ needs to be inverted since the skew on the input distribution has gone below 0.5. That way probabilities are always above 0.5, only the symbol they refer to is changed.

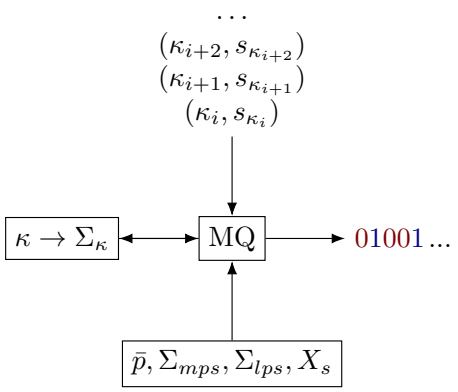

Figure 3.10: Diagram showing data flow within the MQ-coder. Bit-context pairs arrive. The context is used to determine the current state, which along with probability tables generates the output and updates the state for the following input.

All of these elements make up the predictive part of the arithmetic coder (the models). Recall from Section 2.2.3 that an arithmetic coder generated a fraction in the $[0, 1)$ interval. This coder keeps just a part of the fraction active, having 16 and 28 bits respectively for registers A and C that define the interval $[c, c + a)$. Every so often, the part of the interval C that is no longer going to change is shifted out and A is normalized, freeing up space to keep both registers within the designed limits. This is done via a variable $\bar{t}$ that keeps track of the number of fixed bits in C.

The only restriction is that a coder cannot emit symbols in the `0xff90-0xffff` range since those are reserved, so special care is taken if those are to be shifted out of the C register. To that end, a buffer T keeps the last emitted byte to check for reserved sequence formation.

3.4 LCPLC

The Low Complexity Predictive Lossy Compression (LCPLC) presented in [4, 5] is the last of the algorithms studied here. Despite its name, it allows not only for lossy but also for lossless compression. Distortion is dynamically measured when compressing and, based on a threshold, a decision is made whether to code the compressed (lossy) values or the uncompressed (lossless) ones. If the threshold is set to zero distortion, compression will be lossless. Anything allowing distortion will incur in compression with progressively more ratio and less quality as the threshold extends.

The interest of simple algorithms that can work in this near-lossless manner has been evidenced by the recent update to the CCSDS 123 standard (B-2 revision [34]). However the CCSDS revision works on individual sample skipping, instead of the full block skipping present in LCPLC. Memory requirements are lower, but hardware complexity higher since more operations are done per sample on CCSDS.

LCPLC operates on full hyperspectral images, and is designed to locally take advantage of both spatial and spectral similarities within adjacent image samples. For that, the image is divided into non-overlapping blocks of size $N \times M \times B$ (usually $N = M$ making the blocks square in the spatial direction). B is the number of bands, and any block always spans the full spectrum of the image. Each sample within a block has a bit-depth of D bits, usually 16. It is also valid to split blocks further in the spectral direction, though this is not practical since it makes compression both slower and less efficient distortion-ratio wise as it will be later seen.

A predictive model (Section 2.2.6.2) is used to predict, for each block, the values in a band based on the values from the previous band. This is done by minimizing the expected mean square error [5, p. 2.1]. If the prediction is good enough, it is used instead of the raw values going forward. Otherwise, differential coding (Section 2.2.6.1) is used to process the differences be-

tween raw and predicted values. Finally, golomb and exponential golomb coding (Section 2.2.2) create the final bitstream.

3.4.1 Prediction

After blocking the image, the predictor comes into play. For each block, consider $x_{m,n,i}$ the sample at spatial position (m, n) and spectral band i. Blocks are coded in slices, where a slice comprises all of the samples of a given band i within the block. The first slice to be coded is slice 0, and then the rest follow in incremental order. Within a slice, a raster scan is followed for the coding of each sample. Two values are required to define how the algorithm works: $\hat{x}_{m,n,i}$ is the decoded value at the given position, while $\tilde{x}_{m,n,i}$ is the prediction at that same position.

A simple two-dimensional predictor is used for slice 0, which is given by:

$$\tilde{x}_{m,n,0} = \frac{\hat{x}_{m-1,n,0} + \hat{x}_{m,n-1,0}}{2} \tag{3.51}$$

From it, the prediction error $e_{m,n,i}$ is calculated as $e_{m,n,i} = x_{m,n,i} - \tilde{x}_{m,n,i}$. Since the error can be negative, it is mapped to positive values. A simpler version than the one used by CCSDS (defined in Equation (3.27)) is used here:

$$m^e_{m,n,i} = \begin{cases} 2\left|e_{m,n,i}\right| - 1 & e_{m,n,i} > 0 \\ 2\left|e_{m,n,i}\right| & e_{m,n,i} \leq 0 \end{cases} \tag{3.52}$$

The rest of the slices are coded by looking at the previous slice for prediction. $\hat{x}_{m,n,i-1}$ is made similar to $x_{m,n,i}$ by means of a least squares estimator α_i, which minimizes the expected MSE by using it in Equation (3.54). It is obtained from μ and $\hat{\mu}$, which are the average values of the raw and decoded values in a slice respectively. They might also be referred as $\bar{x}$ and $\bar{\hat{x}}$ for readability. α is obtained as follows:

$$\begin{aligned} d_{m,n,i} &= x_{m,n,i} - \mu_i \\ \hat{d}_{m,n,i} &= \hat{x}_{m,n,i} - \hat{\mu}_i \\ \alpha_i^N &= \textstyle\sum_{m,n} \left(\hat{d}_{m,n,i-1} \cdot d_{m,n,i}\right) \\ \alpha_i^D &= \textstyle\sum_{m,n} \left(\hat{d}_{m,n,i-1}\right)^2 \\ \alpha_i &= \alpha_i^D / \alpha_i^N \end{aligned} \tag{3.53}$$

A diagram of how the prediction dependencies can be seen in Figure 3.11. Experimental results from [5] show that α_i can be quantized to 10 bits in the $[0, 2)$ range as $\hat{\alpha}_i$. The same is done for μ_i in the range $[0, 2^P - 1)$, yielding $\hat{\mu}_i$ with D bits. A prediction is made then as:

$$\tilde{x}_{m,n,i} = \hat{\mu}_i + \hat{\alpha}_i \left(\hat{x}_{m,n,i-1} - \mu_{i-1}\right) \tag{3.54}$$

while the error follows the same equations as with the first slice.

If higher compression is required, the error can be quantized using a uniform threshold quantizer of parameter Q using powers of two for faster implementation afterwards. So instead of the prediction error $e_{m,n,i} = x_{m,n,i} - \tilde{x}_{m,n,i}$, $\hat{e}_{m,n,i} = \mathrm{sgn}^+\left(e_{m,n,i}\right)\left(\left(\left|e_{m,n,i}\right| + \left\lfloor 2^{Q-1} \right\rfloor\right)/2^Q\right)$ is used. An approximation of the original error is $e'_{m,n,i} = \hat{e}_{m,n,i} * 2^Q$ when performing calculations for $\hat{x}_{m,n,i}$.

3.4.2 Slice skipping

The lossy part of the algorithm comes from the slice skipping process. Any slice other than the first has a set of predictions that are based on the previous slice. If predictions are good enough,

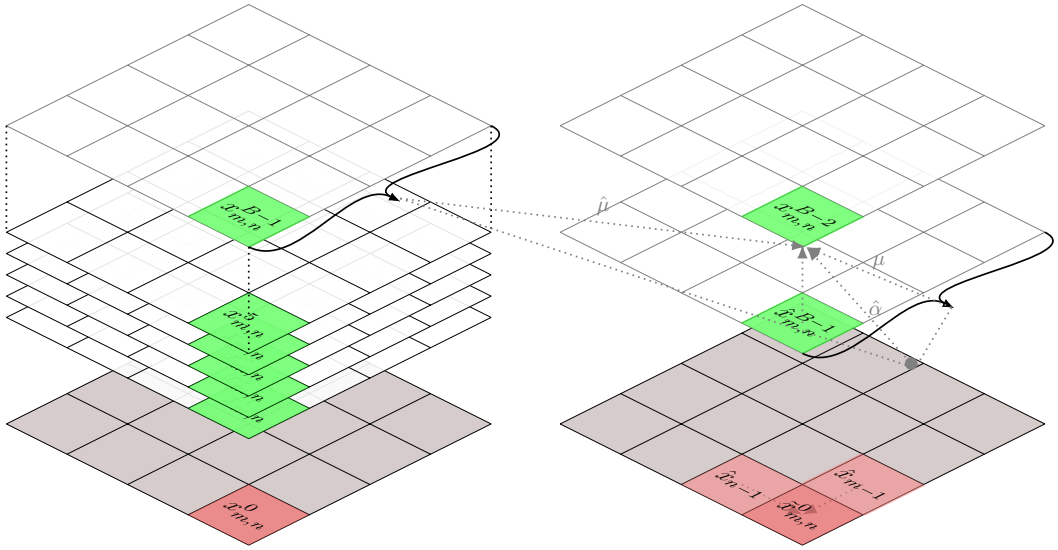

Figure 3.11: A diagram of LCPLC dependencies for prediction within a block. The first slice only depends on itself in a small neighborhood of the current sample (just using the top and left neighbors). For slice i, aggregates from the original i slice, as well as values from the decoded slice $i-1$, are combined with each sample of decoded slice $i-1$ to create the corresponding samples from slice i.

coding can be skipped altogether. The distortion D of a slice is calculated as:

$$D = \frac{1}{NM} \sum_{i=0}^{NM} e_{m,n,i}^2 \tag{3.55}$$

A threshold is set above which a slice needs to be coded since the incurred loss of using the predictions is unacceptable. However if the error is below the threshold, the slice coding is skipped and predictions used instead. The threshold equation is given by $D^{thresh} = \frac{\gamma 2^{Q+1}}{3}$. Experimentally [5], a value of $\gamma = 3$ has been found to be effective in yielding good qualities at decent compression ratios. Higher values provide better ratio and worse quality, and vice-versa.

3.4.3 Coding

Two different coders are used in LCPLC: an exponential Golomb coder of order zero, and a power-of-two Golomb coder (see Section 2.2.2).

For the first slice, the raw $x_{0,0,0}$ is exp-Golomb coded after being quantized, dequantized and mapped. Then, all mapped errors $m^e_{m,n,i}$ are golomb coded. Now, for simplicity, let $l = m+nM$, and define a_w as the number of past sample errors used for coding. Let $R_{m,n,i}$ and $J_{m,n,i}$ be:

$$R_{m,n,i} = R_{l,i} = \sum_{i=\max(0,l-a_w)}^{l-1} e_{l,i} \tag{3.56}$$

$$J_{m,n,i} = J_{l,i} = \min(l, a_w) \tag{3.57}$$

Define the parameter for the Golomb coder $k_{m,n,i}$ as:

$$k_{m,n,i} = \left\lceil \log_2 \left(\frac{R_{m,n,i}}{2^{\lceil \log_2 (J_{m,n,i}) \rceil}} \right) \right\rceil + 1 \tag{3.58}$$

The Golomb parameter is thus obtained from the accumulator that holds the sum of the last a_w errors for the current slice. Note that in [5], the denominator for Equation (3.58) is just

$J_{m,n,i}$, but here the first power of two greater or equal than it is instead used. This simplifies hardware implementation down the line and no difference in compression efficiency was found.

For slices other than the first, both α_i and μ_i are coded raw (in their 10 and D bit form), followed by a single bit indicating if the threshold was met. In the first case, coding for the slice is finished. In the latter, all mapped errors follow coded in the same fashion as with the first slice.

3.4.4 Summary

So, to sum the algorithm up, the following is done when coding each block.

- Do the following for the first slice:
  - $x_{0,0,0}$ is quantized, mapped and exp-zero Golomb coded.
  - $\forall m, n$, Golomb code $m^e_{m,n,0}$ with parameter $k_{m,n,0}$.
- Then, for every slice after the first one in ascending order:
  - Code $\hat{\alpha}$ in 10-bits followed by $\hat{\mu}$ in D bits.
  - Code a bit indicating if the block is skipped.
  - If it is not, then $\forall m, n$, Golomb code $m^e_{m,n,0}$ with parameter $k_{m,n,i}$.

Since each block is independently coded, they can be arranged into any order when assembled into the full compressed image. For simplicity, a raster order will be followed in compression.

If both block skipping and quantization are disabled, the result will be lossless. As quantization and the threshold are increased, the result will be progressively more compressed, though it might be the case that certain configurations still produce lossless results if the image is predictable enough.

Chapter 4

Implementation

Implementing algorithms involves as a first step deciding which hardware to target. General purpose processors are a flexible option, but lack the performance that other platforms offer. GPUs are good candidates if the algorithm is parallelizable at the data level, performing the same repetitive operations over different sets of data. A custom ASIC is the best option when time and money are available, creating the specific circuit that optimally executes the algorithm.

FPGAs offer a reconfigurable fabric of logic elements that can mimic a custom ASIC, and by slightly reducing performance are able to support implementations that can be developed almost as fast as CPU or GPU ones, while being far more power-efficient and fast.

When implementing different algorithms on an FPGA, there is not a single approach. The main goals are to increase throughput, reduce power consumption, and use less resources. Usually, to improve one of the three implies some loss on the other two.

Throughput is attained through parallelization, pipelining and clock frequency increase. The first two increase resource use and, in turn, power use. Higher frequency also results in more power being used proportionally, due to the increased voltage required which is quadratically proportional to power consumption.

To reduce this undesirable power consumption, the straightforward way is to lower the clock frequency and voltage. Care must be taken then to not lower throughput under the specified limit. Resources will be the same, but usually a bit of parallelization will be added to make up for the throughput loss, again increasing resource use.

And if the circuit is made to fit on a small FPGA, throughput will be sacrificed to spare the necessary gates to fit. This generally also results in reduced power, so as long as the speed requirements are met the circuit will be sufficiently good.

Of course, some algorithms are prone to parallelization, while others need strong optimizations in the critical path. Others might just reach a certain limit given their data dependencies, and so mathematical optimizations might be useful in those cases, changing the algorithm itself.

All in all, the ideal scenario is to meet throughput demand, and optimize area and power while keeping it above threshold. There is not a single best way of doing it, so for all three algorithms it will become clear that different paths were followed. There was a common methodology though: first implement the algorithm in software, so that the algorithm structure is made clearer for a hardware implementation, and then implement the hardware itself.

FPGAs can be targeted with two main tools: HLS synthesis and custom HDL synthesis. The first approach starts with a constrained high-level language code (such as `C++`) where some constructs are forbidden due to FPGA limitations. Without those, automatic tools are able to transform the high-level code into low-level HDL code. The second (and more traditional) approach is to directly write the HDL code (normally VHDL or Verilog).

High level synthesizers are becoming powerful tools capable of creating hardware that is just as fast as a custom design for easily parallelizable algorithms. But the technology is still limited for complex constructs or data dependencies, where low-level code is still advantageous. Certain coding styles that favor hardware implementation are needed, with data flows within high-level languages having to mimic hardware to achieve the best results in an automated way. Some constructs still need to be hand-optimized.

In the same way that compilers made obsolete hand-written assembly code, HDL synthesizers will replace most hand-written HDL. However, for this thesis, the focus is put on the great performance achieved by the latter, which are yet to be surpassed by automatic generation. Thus, implementations are custom-made in both VHDL and Verilog, and so far have outperformed HLS alternatives.

In this chapter, the implementation of the algorithms described in Chapter 3 is presented. First for the lossless CCSDS 123.0-B-1, followed by the JPEG2000's tier 1 coder in JYPEC, and finally for LCPLC. Detailed diagrams will be shown when necessary to shed light into the different optimizations that took place thanks to the custom HDL design.

All three algorithms have been implemented with performance in mind. The different parameters (image size, quantization values, register size...) are tunable. The goal is to, given certain algorithmic constraints, automatically synthesize the most optimized hardware possible, instead of having a lower-performance generic core. This is because the ultimate target are radiation-hardened FPGAs, which are ready to fly on satellites, but often need a more optimized circuit than a normal FPGA to overcome their resource and performance constraints.

4.1 CCSDS 123.0-B-1

Being an international standard, CCSDS 123.B-1 has received plenty of attention regarding implementations. One of the first golden standards software-wise was Empordá [83], providing an easy way of testing other coder/decoders by cross-validation.

A custom implementation [18] was developed for this work, allowing trace generation of the different values that were generated by the algorithm. Not only the outputs, but also intermediate operations were traceable. Correctness was ensured by cross-validating it against Empordá, and by means of applying the compression/decompression cycle and checking for perfect reconstruction (given that it is a *lossless* standard).

As for the more interesting part, hardware, a further look was had to existing implementations, to check for different techniques and to look for what was missing in a pool of already available designs.

4.1.1 Previous work

Being an international standard, different FPGA implementations of the CCSDS 123.0-B-1 exist that take advantage of different properties of the algorithm, mainly focusing on the different types of traversal through the image.

One of the first [180] already mentions the fact that different traversals through the image need different resources. The amount of memory can be made dependent on the number of bands or image slices, while the amount of logic resources is usually bound by the parameters selected for the algorithm, which trade off complexity for compression performance. In the end, a decision is made to have a $2 \times P + 1$ port memory to avoid any overhead of memory in the compressor at all. Improvements in speed are done by pipelining the different stages (mainly prediction and encoding), noting that memory is a bottleneck.

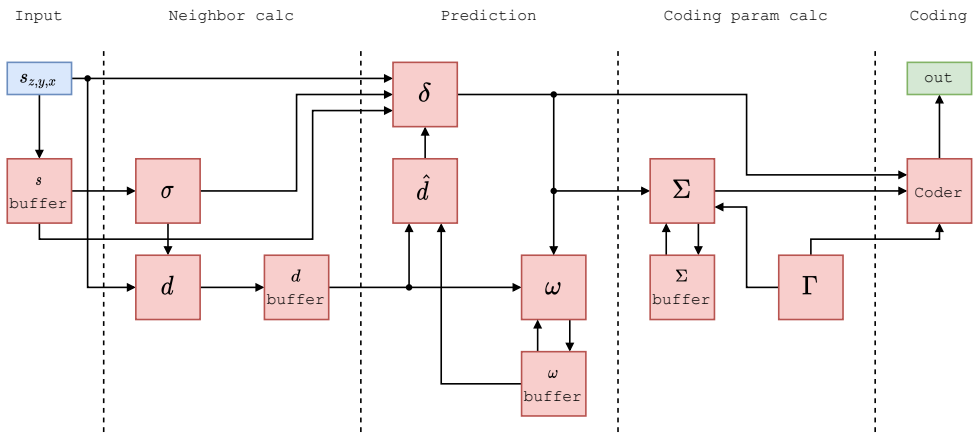

Figure 4.1: Full diagram of the CCSDS compression module. Dashed lines indicate potential for pipelining.

Moving a little bit of overhead to the on-board memory proved to be beneficial in [113]. BIP ordering is used and a full spectral slice is stored. But this is usually not a problem since, for the common image sizes, a full slice is under a Mb in size. Pipelining is not used and so even at low frequencies, a full sample per cycle is compressed, increasing the performance of [180].

Going further, in [68, 207] it is realized that, for BIP mode, pipelining of the main feedback loop (which updates the weights used for the prediction) is possible. This requires the image to have more bands than pipeline stages, since otherwise stalling occurs. This is not a problem in practice since, being hyperspectral, the number of bands is always high enough. Modifications to the pipeline depth are proposed [207] to take care of it in extreme cases. A full slice still needs to be stored, and resource use is higher than in [113], but the speed is increased by a factor of almost 5 thanks to the pipelining of the dot product stage.

Even more resources are used in [154] to generate data-level parallelism, again in BIP mode. Instead of having just one pipeline, a number n of pipelines are instantiated at once, each processing every nth sample. This again increases performance and, for four parallel pipelines, speed is increased by almost a factor of four with respect to [68], where the main limitation is the synchronization of the output values to make sure standard ordering is preserved.

BIP mode brings the most possibilities on the table, given that pipelining, as well as data-level parallelism are possible. However, this limits the algorithm to just one type of ordering, requiring additional memory to reorder the image in case the sensor does not capture it in that format, and also using more resources for pipeline management. For this implementation, it was decided to not use external memory, and to have the smallest core capable of dealing with any input ordering in real time.

4.1.2 Hardware implementation

Every step of the algorithm is performed in its own module for simplicity and correspondence to the mathematical definition. Figure 4.1 shows the overview of the different modules present within the compressor.

Samples come in a specified order (BSQ, BIL or BIP) through a single port. This is to avoid the need of external memory, but comes at the cost of using internal RAM blocks to store frames or bands.

First, samples s go into a buffer to allow sufficient values to be ready when calculating neighborhoods. The buffer is a set of FIFO queues with length tailored to a specific image size so that the head of the FIFO is ready the same cycle it fills up. Afterwards it will emit and

accept one sample per cycle. This limits flexibility to an exact sensor size, but at the same time uses the least memory possible.

The input samples, along with any buffered go through to the next stage, where local sums σ and differences d are calculated. The differences go through another buffer to have the appropriate ones available at any time.

A third stage takes the local values and transforms them into predictions δ using the weights ω. For that, the expensive $\hat{d}$ is calculated using a dot product between the weight and difference vector.

Coding comes afterwards, where δ is used to update the accumulator, which also uses a buffer since each band has its own. Along with the counter, they go on to the final stage to be coded. There, values are used to generate the final codewords and lengths, which a serializer will take and use to output the final bitstream.

Despite the clear-looking pipeline potential, the cost of the feedback loop in the prediction stage is so great that it is only useful to pipeline the stage between the prediction and coding parameter calculation. Any other pipeline stage added does not reduce operation time but does use additional resources, so it is avoided.

A more detailed diagram and description of the different modules is provided in the following pages, where the following notation is used for simplicity in the diagrams:

- x_l, y_l, t_l and z_l indicate that the respective coordinate is equal to zero. On the other hand, x_h, y_h, t_h, and z_h indicate that coordinate is the last within image boundaries. Note that sometimes these flags will be inverted (e.g: $\bar{x_l}$).

- Blue rectangles are inputs for the module depicted in a diagram, while green rectangles are outputs. Grey rectangles are constants that will bring optimizations to the circuitry instead of using variables.

4.1.2.1 Local sums

Local sums are calculated depending on the coordinates of the current sample being predicted. All possibilities are calculated, and muxes select the appropriate one to move forward.

Two different calculations can be done, either the neighbor-oriented from Equation (3.2) or the column-oriented from Equation (3.3) one, which simplifies the circuit. Both are seen in Figure 4.2.

$\sigma_{0,0,0}$ is not used within the implementation, but it will be generated anyways by hardware and discarded by the modules afterwards.

4.1.2.2 Sample storage

To be able to calculate $\sigma_{z,y,x}$, not only the sample for the current position is necessary but also previously seen samples. This issue could be solved by having multiple inputs to the modules with all necessary samples. This however requires multiple accesses to memory at once.

Here, the target is to take information in raster-order from the sensor itself and directly feed it to the module. Thus, a FIFO system is built for storing neighboring samples and then retrieving them from within the FPGA. Their design depends on the order in which samples are fed into the compressor. Three different designs are required, respectively, for neighbor-oriented and column-oriented sums (Shown in Figure 4.3).

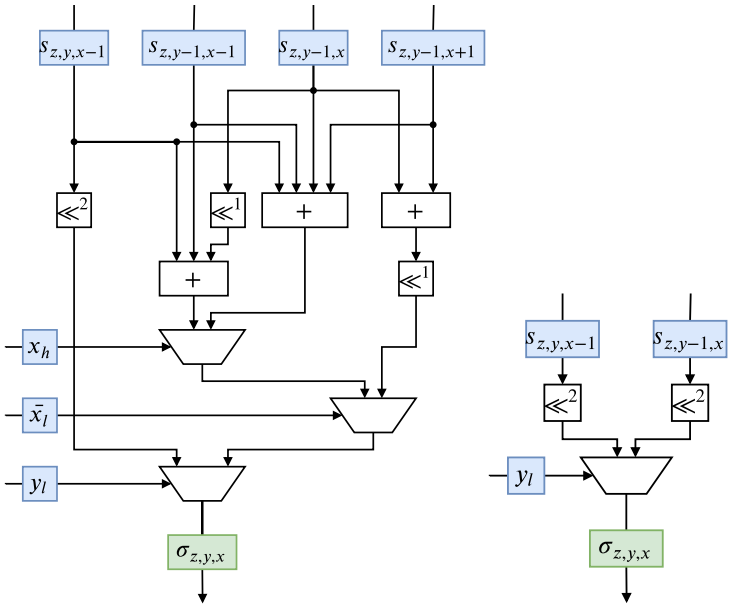

Figure 4.2: Neighbor-oriented and column-oriented local sum

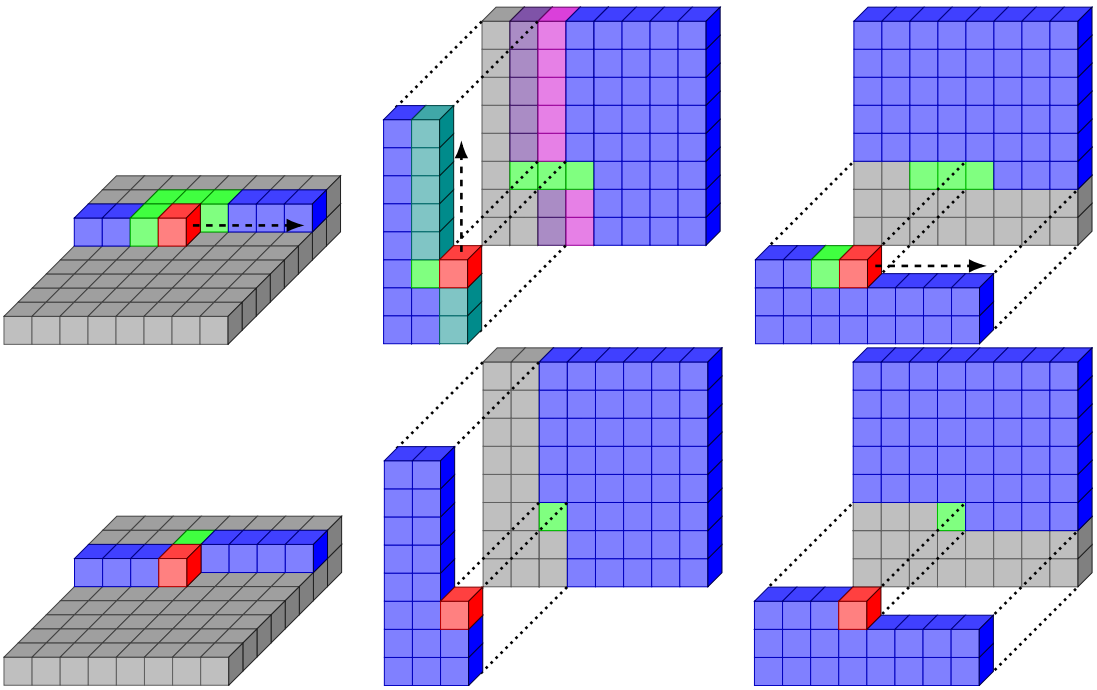

Figure 4.3: FIFO structure for different configurations. Top are for neighbor oriented sums, bottom for column-oriented sums. From left to right, BSQ, BIP and BIL orderings.

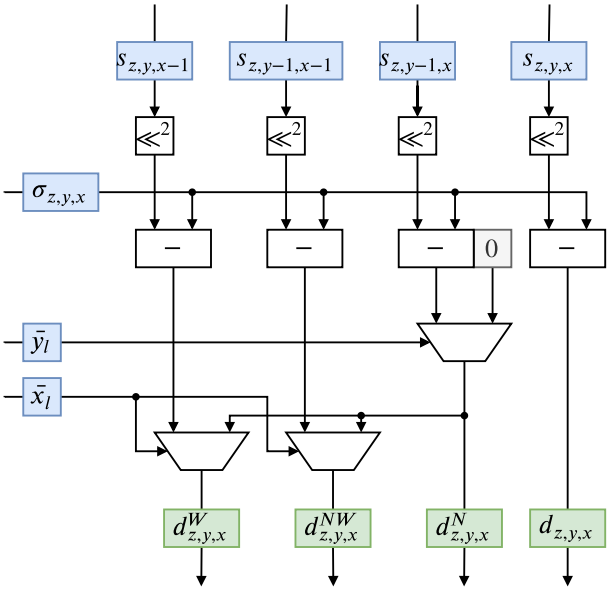

Figure 4.4: Difference calculation. North, West and Northwest differences are not used under reduced prediction mode.

In blue, stored samples are shown. In green, neighboring samples that are used. In red, the current sample. As seen, the type of prediction has little impact on the size of the FIFOs. However, the type of ordering for the image has great impact. For BSQ, FIFO size is just N_X, while for BIP and BIL is $N_X N_Y$. This is unavoidable if only one read-port is desired.

4.1.2.3 Difference calculation

Differences are calculated based on the local sum. The central difference $d_{z,y,x}$ is always present, while north, west and northwest directional differences are only present under full prediction mode. They all form the vector from Equation (3.4). Figure 4.4 shows both types.

For central differences, calculating them for every sample is time and resource consuming. Instead, a storage system is devised to buffer the values from previous bands so that they can be used for calculation.

4.1.2.4 Difference storage

Differences, in the same way samples need to, require to be stored for the difference vector assembly. Directional differences can and are calculated on-the-fly. But central differences will come from a FIFO structure with P read ports, one for each difference used. Here, the cost is $N_X N_Y P$ for BSQ ordering, $N_X P$ for BIL ordering, and just P for BSQ ordering.

BSQ requires excessive resources, while BIL is ideal in this case. For BSQ, the FIFO structure is designed with P memory units of size $N_X N_Y$. For BIL, the same number of memories is used, but requiring each to be only of size N_X. For BSQ, a simple shift register P samples is used. This is seen in Figure 4.5.

4.1.2.5 Weighted difference

The dot product from Equation (3.5) between the difference vector and the weight vector is done by a tree-reduction sum of the individual products as seen in Figure 4.6.

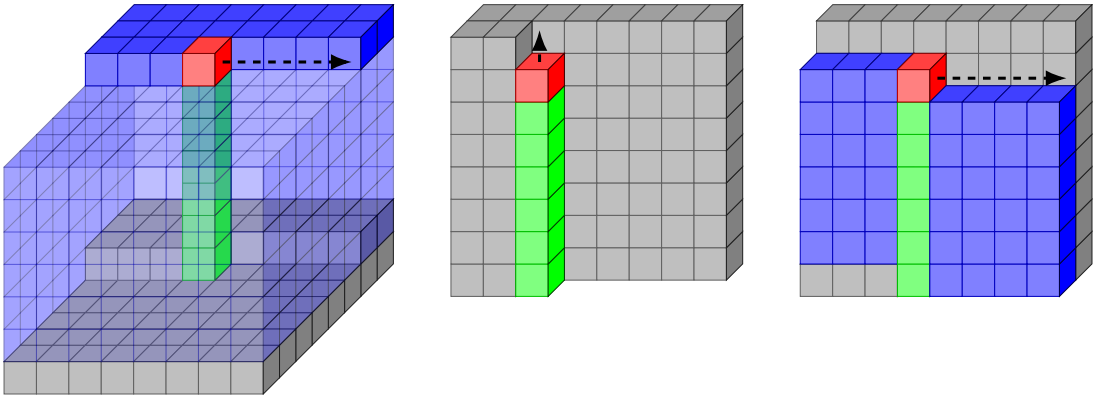

Figure 4.5: FIFO structure for differences. BSQ, BIP and BIL orderings respectively shown.

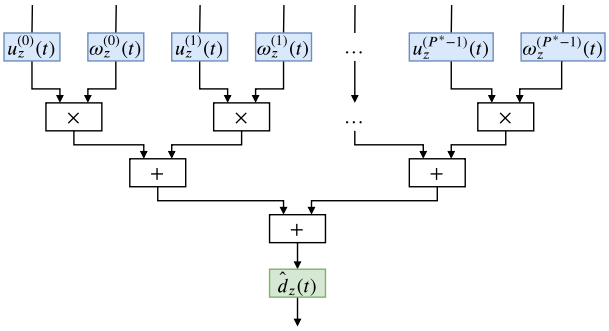

Figure 4.6: Central local predicted difference.

4.1.2.6 Weight vector storage

Weight vectors are initialized to default values when the reset signal is raised. Since vectors are shared for all samples in a band, for band-interleaved methods such as BIP and BIL, all vectors need to be stored at the same time, requiring a memory of size $N_Z P^*$. This however is not a problem since in any ordering, either the same or difference memories are going to be dominating the space requirements.

If on BSQ mode, the weight vector is a simple register which gets reset at t_l high. For BIL and BIL, N_Z vectors are saved in a circular FIFO, and on each band change, the top is shifted to the bottom. The new top register is the active weight vector until a new band change is issued.

4.1.2.7 Error calculation

The prediction error from Equation (3.22) is obtained in Figure 4.7. The standard allows for loss of information along the way, assuming the error won't be as precise. This is to avoid using long registers for $\tilde{s}_z(t)$ from Equation (3.19). Here, the smallest register size that ensures no overflow (see Equation (3.20)) is used, so no care is taken in case of overflows since they cannot mathematically happen.

4.1.2.8 Weight vector update

After each sample is processed, weights are updated. This is where the feedback look of the algorithm is: Before processing the next sample in a band, the weights must be updated. Simple operations allow for the calculation of $\rho(t)$ (Equation (3.24)), which can be calculated with just 6 bits (Equation (3.25)).

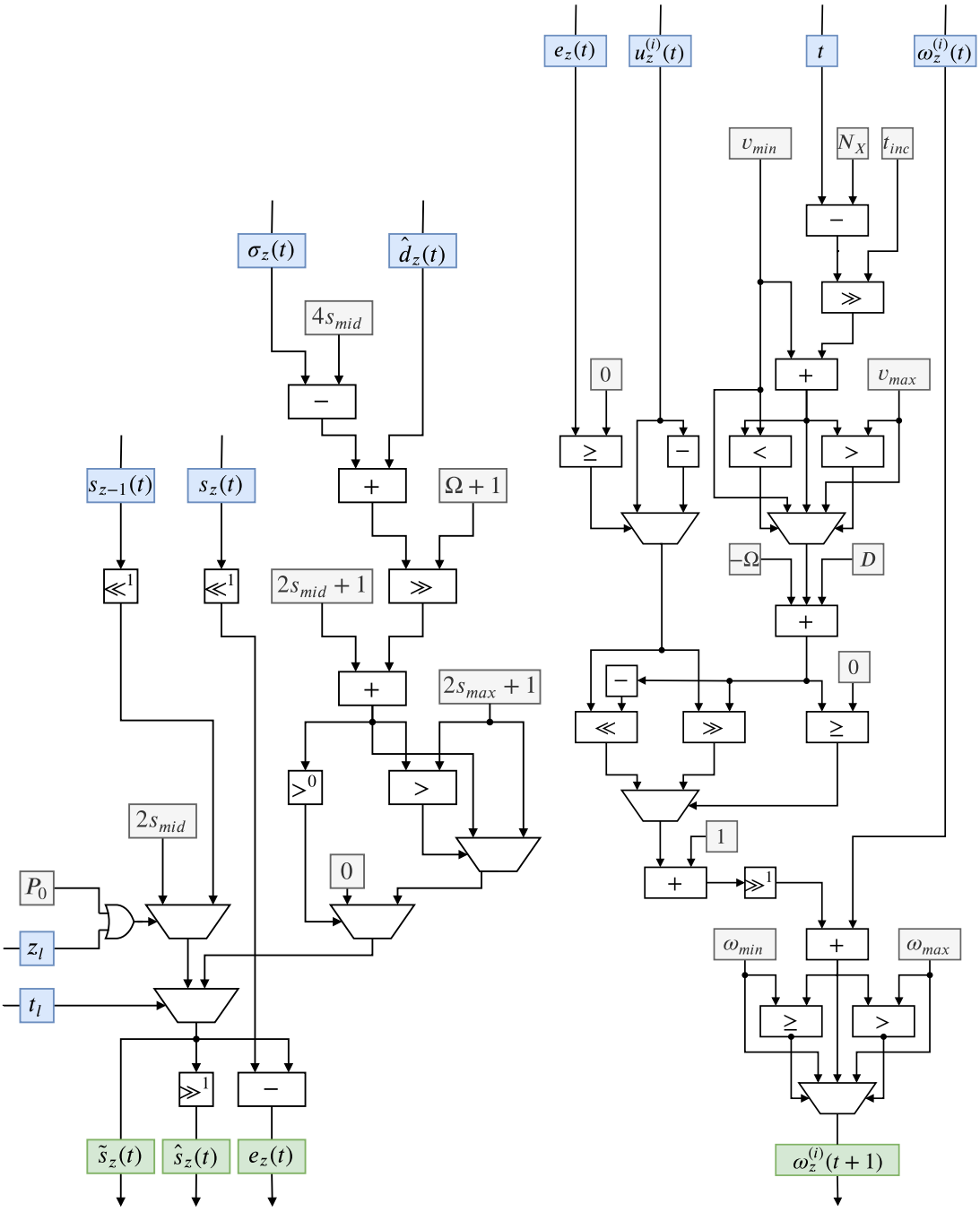

Figure 4.7: Error calculation module (left) and weight update module (right)

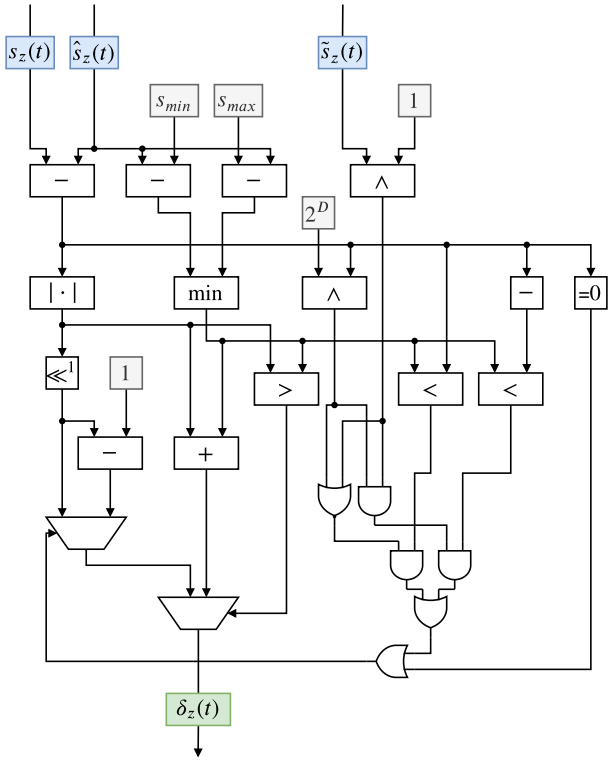

Figure 4.8: Prediction residual calculation.

So, for weight updates, the minimum register size to not incur in overflows, is:

$$\max\{\Omega+3, D+3+\min\{1, v_{min}+D-\Omega\}\}+1 \tag{4.1}$$

$D+3$ is the size of the elements in $\boldsymbol{U}_z(t)$. A look is then taken at ρ's lower limit $v_{min}+D-\Omega$ since it is the one generating a left shift. So assuming maximum value in both the vector's entries and ρ, the length of both is added, and an additional bit reserved in case of carry-outs. For the sign used in Equation (3.23), a simple mask is done with the upper bit. This all is seen in Figure 4.7.

4.1.2.9 Mapped prediction residual

For the mapped prediction residual, Equations (3.27) and (3.28) are used. The only trick here is that, for the mux selector, the conditions are all calculated based only on bit checks and comparators, instead of using any multipliers. This greatly simplifies hardware and is mathematically equivalent. The lowest bit of $\tilde{s}_z(t)$ is checked for parity, while the upper bit of $\Delta_z(t)$ is checked for sign. Care is taken when $\Delta_z(t)$ is zero to also take that into account despite the parity of $\tilde{s}_z(t)$. This is seen in Figure 4.8.

4.1.2.10 Encoder

For the coder, the mapped prediction residuals $\delta_z(t)$ are used along with an internal accumulator and counter. The counter is combinationally obtained via Equation (3.34) as seen in Figure 4.9.

For the accumulator, it is restarted when t_l is high, and otherwise just keeps adding the $\delta_z(t)$ values until a renormalization barrier is encountered (see Equation (3.31)), when it shifts by one effectively halving its value. The counter also halves its value, so the average $\Sigma_z(t)/\Gamma_z(t)$ is maintained.

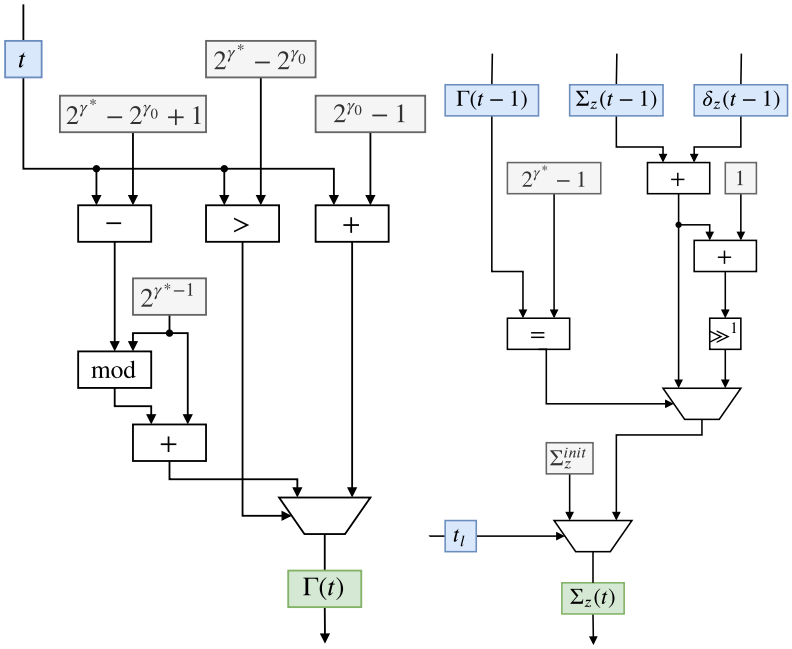

Figure 4.9: Counter and accumulator calculation.

Values for $u_z(t)$ and $k_z(t)$ from Equation (3.29) are obtained in Figure 4.10. For $k_z(t)$, a trick is used to avoid a sequential calculation that is required with divisions. Since its value is bounded by $D-3$, instead $D-2$ shifts are performed, and a priority encoder selects the one of lowest value since the first k that satisfies Equation (3.30) is the one looked after. $u_z(t)$ is just the original value shifted by $k_z(t)$.

For the output, three values are generated that will then feed a serial converter to generate a bitstream over a wire. The code will be made up of a certain amount of zeros $Z_z(t)$, and of $B_z(t)$ bits of the code $C_z(t)$. $Z_z(t)$ gets the value of $u_z(t)$ unless above the threshold, in which case it is u_{max}. It is zero for the first (uncoded) sample of each band, as seen in Figure 4.11.

The code is either $d_z(t)$ when on the first sample of a band, or the lowest $k_z(t)$ bits of $d_z(t)$ preceded by a **1**.

The amount of bits of the code used will be D for uncoded samples (the first in a band and if $u_z(t) > u_{max}$), and otherwise will be $k_z(t)+1$ to send the bits in $C_z(t)$ up to, and including, the inserted **1**.

4.1.2.11 Serial converter

A serial converter is also designed to output bits instead of the $(Z_z(t), C_z(t), B_z(t))$ triplet. It stores the triplet in a buffer and then outputs its information.

First, a counter is used to emit as many zeros as $Z_z(t)$ indicates, at one per cycle. Then, as many bits as $B_z(t)$ indicates are emitted from $C_z(t)$ from most to less significant. At the end of the process, $Z_z(t) + B_z(t)$ bits have been emitted in the same amount of cycles.

To make it work properly, a FIFO stores the triplets in the input. The first element is taken and processed accordingly, as soon as it's finished, the next element enters the queue. This goes on until the compression process has finished. A simple bitstream has enough speed to output the results from a CCSDS core. In the following section, the core is parallelized and throughput of a bitstream is not enough, a *byte*stream will take its place.

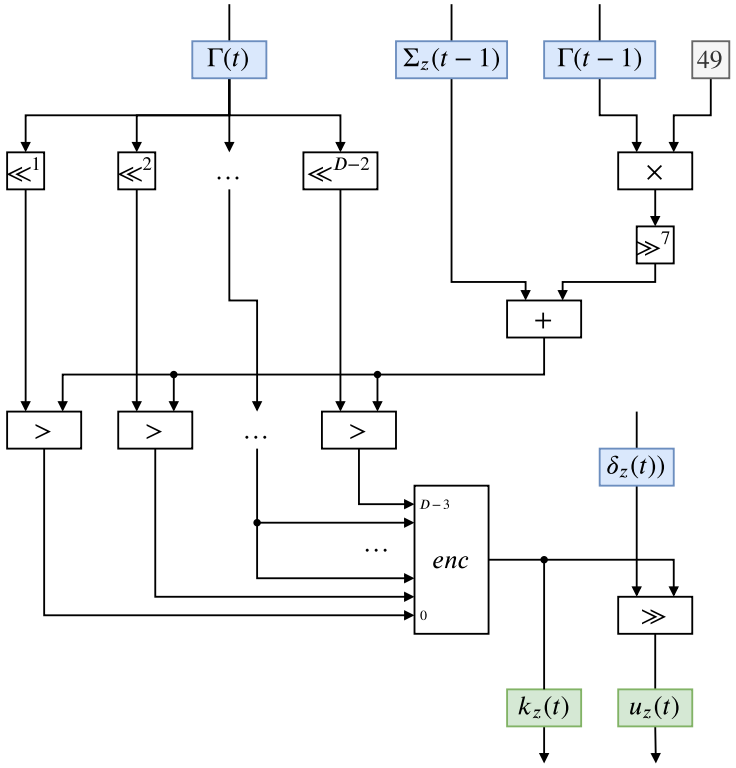

Figure 4.10: Coder parameter calculation.

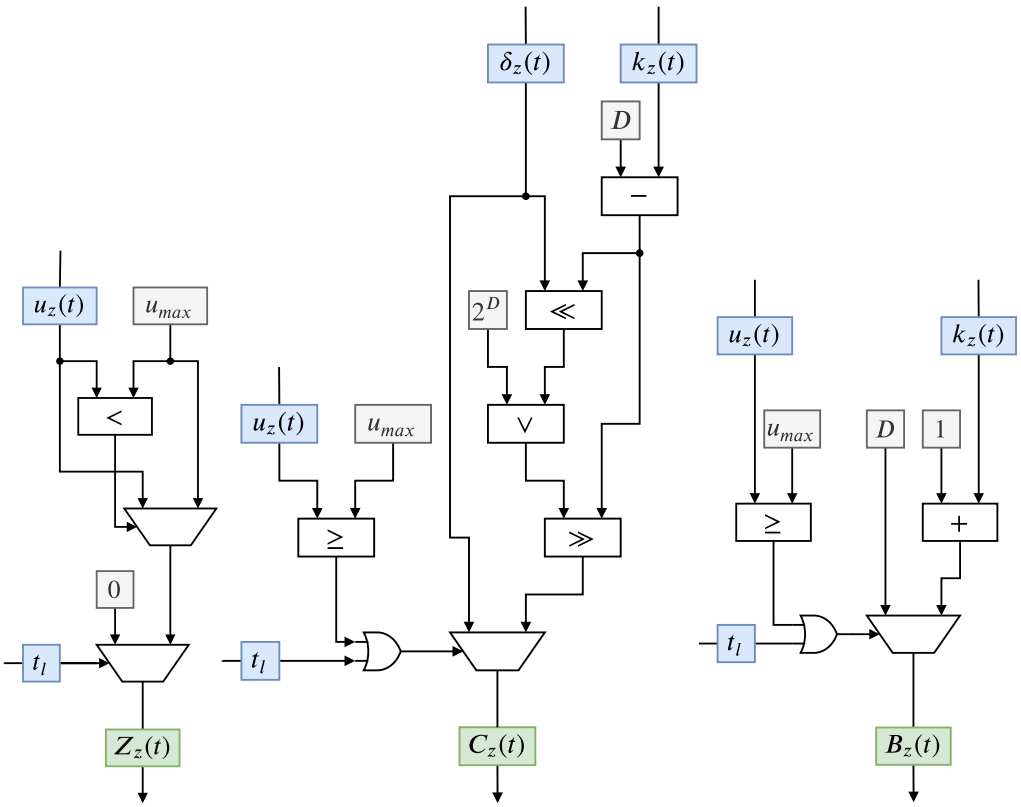

Figure 4.11: Respectively, the number of zeros to output, codeword, and valid bits from within the codeword.

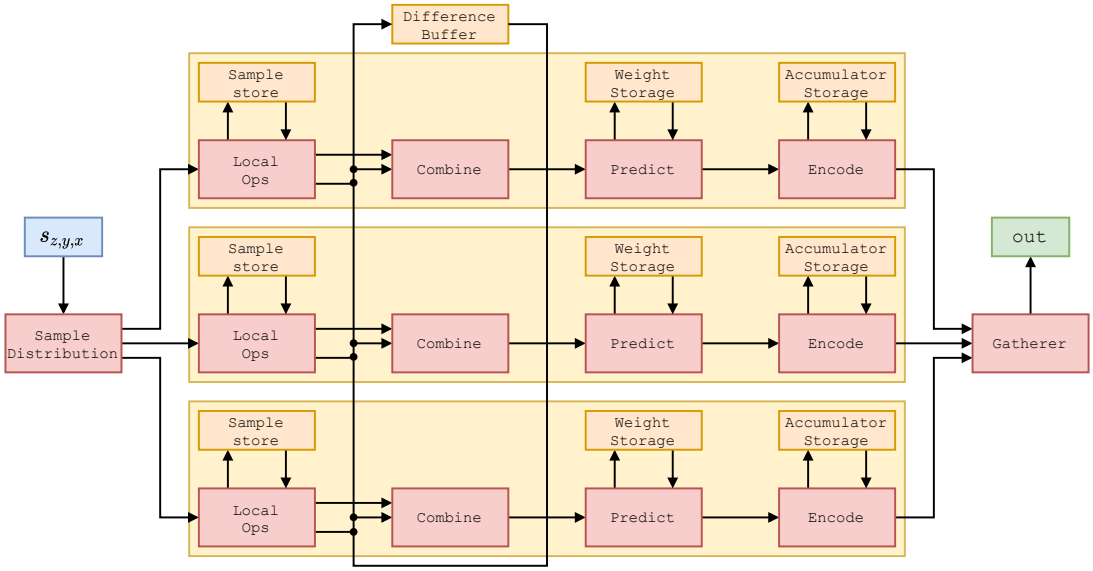

Figure 4.12: Parallel design of the CCSDS 123.0-B-1 algorithm. Multiple copies run in parallel, sharing the differences. Ordering must be BIP to avoid cascading dependencies on the weights.

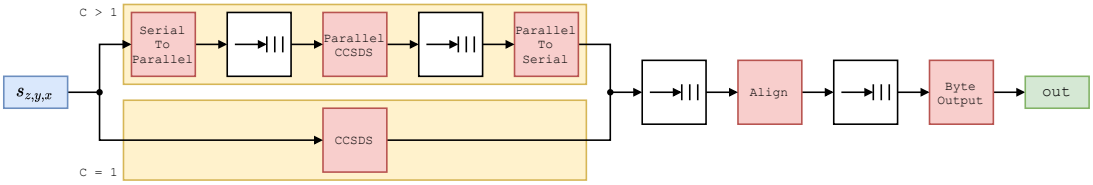

Figure 4.13: The parallel or serial CCSDS option is based on the value of concurrency C. Depending on that setting, different modules will be synthesized.

4.1.3 Parallelization and output

A good thing about CCSDS 123.0-B-1 is it can be parallelized at the pixel level. If running on BIP mode, multiple consecutive samples from the same pixel can be calculated in parallel, forwarding only the differences from one unit to the next. Since these only depend on the image samples, and weights are independent per band, cascading effects do not take place and full parallelization is possible with no penalty brought by the number of concurrent units. Parallelization is thus possible following a design such as the one shown in Figure 4.12.

One thing to note here is that storages are individual per algorithm instance since they do not share information. This is possible because the number of instances is always a whole divisor of the number of bands, and thus each instance will process all of the samples pertaining to a specific band. Values that are used per-band can then be safely stored without interaction between instances. The only exception are differences, which are shared between all since, for the prediction, a dot product is done with differences from previous bands.

Each instance works exactly as a normal CCSDS 123.0-B-1 core with the only change being the difference vector arriving from either the difference buffer or the previous core instead of a difference storage module. Aside from that, the sample distributor and results gatherer are the other main modules that are worth mentioning. They are simplified in Figure 4.12, and the full diagram is shown in Figure 4.13.

Samples always enter through the same channel, however the core will either be parallel or serial. The serial case is straightforward, so the parallel one is explained in more detail. Since it is made up of many serial CCSDS cores, the interface is the same for each one (input is s and output is $(Z_z(t), C_z(t), B_z(t))$).

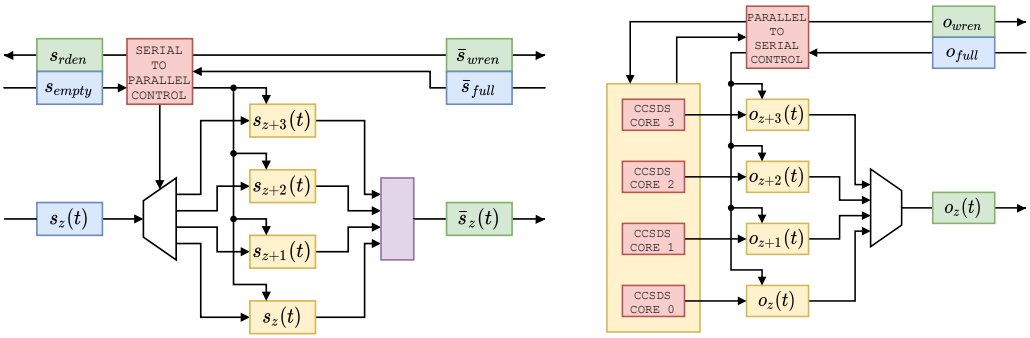

Figure 4.14: Serial to parallel and parallel to serial modules for CCSDS. A simple array of registers takes care of storing samples until a parallel vector is sent, and keeping a vector until it has been serialized respectively. They are only synthesized if $C > 1$.

The serial to parallel module feeds of a single sample stream, and outputs C sample streams, each containing samples $s_{x,y,z_i}, i \in \{1, \ldots, C\}$ where $z_i \bmod C = i$. It simply buffers the inputs for each of the output stream and emits them all at the same time once they are ready, since all the CCSDS cores are synchronized within the parallel module.

Output is quite similar. All outputs from the C modules are received simultaneously, and a buffer keeps them all in place while the parallel to serial converter emits them in order. Once they are all emitted, the next parallel value is ready to be received. Both serial-to-parallel and parallel-to-serial modules are shown in Figure 4.14.

After that, either with the parallel or serial version, a stream of triplets $(Z_z(t), C_z(t), B_z(t))$ enters the aligner shown in Figure 4.15. It creates a code by pre-pending $Z_z(t)$ followed by the least significant $B_z(t)$ bits of the code $C_z(t)$. All of the bytes that are fully completed exit the module, and any uncompleted bytes stay in a buffer to be pre-pended to the next triplet. The number of bytes emitted is limited by the maximum amount of bits that can be input, plus the maximum number of bits that might be buffered, for a total of $7 + U_{\max} + D + 1$, since up to $U_{\max}$ zeros might be output, and $C_z(t)$ can be up to $D + 1$ bits long.

The byte output module shown in Figure 4.16 receives thus up to $(7 + U_{\max} + D + 1)/8$ bytes, and emits them sequentially in the same way that the parallel to serial converter processed the triplets. At this point, and depending on the local characteristics of the image, many or very few bytes might be emitted. The latter case is not a problem, but the former produces stalls. This is why every point connecting modules within the pipeline is fitted with FIFOs, to be able to dampen the effect of the last module stalling in the case of going through a low compression rate part of the image.

4.2 JYPEC

While CCSDS has a simple software and hardware implementation, JYPEC is quite more complex. Not only does it have more steps, but those steps are also more complicated. Implementing the full algorithm in hardware is a monumental task, so a robust software implementation was first due to make an analysis of what parts could benefit from acceleration.

CCSDS could directly interface with raw image data by just knowing the bit-depth and pixel ordering, since the implementation is prepared for tailored synthesis targeting a specific sensor. JYPEC is more ambitious and deals with whole image compression for different image data types, sensor configurations, and algorithm settings. Given this variability, image and compression header information is also processed and compressed to deal with the wide range of options.

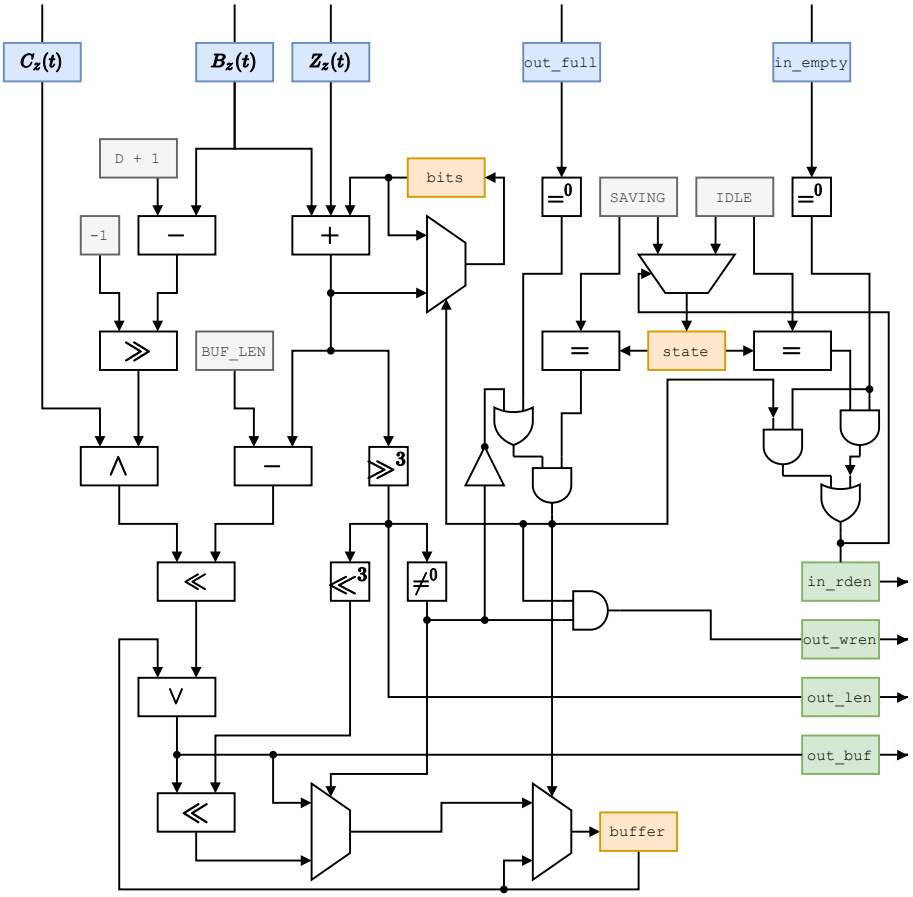

Figure 4.15: CCSDS module that aligns the serialized output of the CCSDS core/s to byte boundaries.

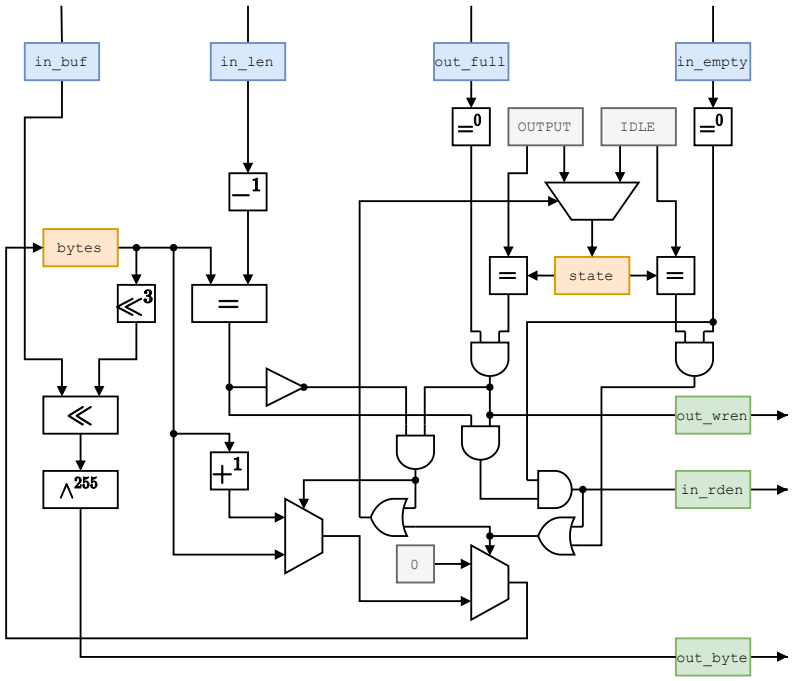

Figure 4.16: CCSDS module tasked with serializing the output byte arrays.

Also, since now the algorithm deals with lossy compression, distortion measures are needed to check its performance, since a simple check for equality between the decompressed and original values is futile.

4.2.1 Software

A good backbone was needed for the software implementation. This includes both a system for hyperspectral Image I/O, and a system for image quality assessment. Along with these modules, the core was also developed. Dimensionality reduction algorithms and the JPEG2000 compressor were included, with the latter being comprised of wavelet transforms, quantizers, and the JPEG2000 coder as a final step. The full program is found in [19] (GitHub).

4.2.1.1 Data

Data management is done though hyperspectral Image objects. These contain both the header and the data, both of them pointing to the raw memory contents. Wrappers ensure that data is accessed properly in any ordering present, giving a consistent interface across formats.

Compressed images are stored in a special format, containing a header that indicates the compression parameters, as well as the original image header (in compressed format) and the compressed stream.

4.2.1.2 Core

The core applies the algorithm step by step. First, a generic interface performs the dimensionality reduction over the image. It contains methods for reducing size. Underneath, any of the dimensionality algorithms might be at work (including no reduction at all). Then, for each band, the next steps are done:

First, outliers are saved (if requested) and their values clamped to the non-outliers limits. This improves quality since further steps work over a tighter range of values, decreasing rounding error loss.

Then, the band is prepared for the wavelet transform by normalizing it to the $(-0.5, 0.5)$ interval, after which the CDF 9/7 transform is performed. Results are pre-quantized and then quantized, using again generic interfaces that allow for different back-ends.

Finally, the image is split in blocks, which are then processed by the JPEG tier 1 coder and placed in the output stream one after another. Using the tier 2 coder means reordering these sub-streams and introducing relocation markers. This increases image size. Since the objective does not include progressive decoding (which is why the reordering is done), the tier 2 coder is ignored.

In hardware only the tier 1 coder is implemented, leaving the wavelet and dimensionality transforms for the CPU to perform. Others [71, 130] have already noted this fact, seeing that the Tier 1 Coder takes up to 70% of compression time for JPEG2000. Later in Section 5.4.1.10 it is seen that timing-wise, this is justified here too.

4.2.1.3 Quality assessment

Finally, the different quality metrics from Section 2.3.6 are also implemented. When compressing an image, the uncompressed result can be optionally compared with the original to obtain

the distortion. This gives a brief summary of compression quality. Compression ratio is also reported, giving both metrics the overall distortion-ratio performance.

4.2.1.4 Options

To optimize the configuration for the algorithm, different options are present in the code that will be explored in Section 5.4.1. Multiple reduction algorithms are available, in which the number of target dimensions can be set, as well as the fraction of input pixels to use for training. A percent of the resulting points can be configured as outliers for raw coding of its values. Finally, the number of wavelet passes can be configured, and the quantization function and bit depth can be set to process the coefficients.

All of these options will modify the base algorithm, obtaining a process that gives preference to a different set of characteristics. An exhaustive search from the distortion-ratio point of view will be carried out in the next chapter.

4.2.2 The JPEG2000 tier 1 coder

The JPEG2000 tier 1 coder works over small blocks (of usually 64×64 samples) of the image. It does a bit-by-bit scan of the block, coding each one at a time. In contrast to other methods such as CCSDS or LCPLC that work over samples, working over bits means that performance drops unless parallelization can be applied. Luckily, as each block is individually compressed, multiple instances of the tier 1 coder can run in parallel increasing performance by the same factor.

Both the bit plane coder (BPC) and the MQ-coder are carefully designed to work in unison as fast as possible. Different techniques have been found throughout time in both parts, but usually the efficiency has been assessed separately. The aim here is to look at both the BPC, the MQ-coder, and their combination to see what works best.

4.2.2.1 Previous work on the BPC

Many BPC implementations have been proposed for the context-data (CxD) pair generator. In [11] a simple BPC is designed that goes over the full block following a zig-zag pattern (recall Figure 3.8). It produces at most one CxD pair per cycle. However since it has to do three passes per bit plane, the actual throughput is expected at around 1/3 CxD pair per cycle.

Improving on that, [130] introduces the concept of skipping. Full 4-bit columns are loaded one by one, marking positions with flags when they have already been processed by a pass. This way the BPC can skip samples in the refinement and cleanup passes. Flags are included even for full columns and full passes, which for the first bit-planes usually allows to skip big chunks of bits. All of these skipping techniques result in around a 60% saving of clock cycles.

A different approach is taken in [86]. Instead of marking and skipping samples, a parallel approach is taken. Full columns of 4 bits are processed at once, emitting up to 10 CxD pairs per cycle. Dependencies within the same column are resolved by cascading operations, but still this is faster than pipelining within the column. Throughput is doubled with respect to the sample-skipping technique, and the extra memory for marking samples is removed.

[66] goes even further by simultaneously coding multiple planes using non-default options. It introduces a small loss in compression efficiency, but is able to increase performance by a factor in the order of the number of bit planes (normally 16 for hyperspectral images). This is specially useful in real-time video transmission. However, having this great throughput at the BPC level means that now the bottleneck in the MQ-coder is also 16 times higher, and so multiple coders

are needed to deal with it, further deviating from standard settings. In this thesis the focus is put on the standard approach since it is sufficient for real-time compression and provides a slightly higher compression efficiency.

4.2.2.2 Previous work on the MQ-coder

More work has been done towards the MQ-coder [103] than the BPC since it has always been the bottleneck within the tier 1 coder. CxD pairs from the BPC are received, and a compressed bitstream is generated which can be further processed by the tier 2 coder. Under default settings, the CxD pairs are processed serially, so little parallelization is possible at this stage. Two main approaches have been devised to accelerate its execution:

- **Pipelining**: Pipelining is a traditional way of improving performance. Despite the feedback loops, distinct stages have been identified that can be separated and pipelined. Namely the update of the A and C registers, as well as packing the output bitstream.
- **Dual symbol processing**: An unrolling of the main loop by a factor of 2 is able to improve performance since stronger optimizations are able to take place. This has motivated the design of MQ-coders with the capability of processing samples in pairs. Two cascading processing units are incorporated for these designs.

Both approaches have been known for a while, with a two decade old dual-symbol two-stage pipeline [38] design present. The first stage updates the A interval while the second performs renormalization and byte output at the same time. All of this is done for two symbols at once, doubling throughput.

In [183] a pipelined MQ-coder with three stages is proposed. Arithmetic operations are performed in a first stage, the A and C registers are updated after, and lastly bytes are emitted. The drawback is that the second stage can stall the first if the amount of shifts to be done in the C register is above one. This is because they use sequential instead of barrel shifters. However this turns out not to be a problem since 1) a faster clock domain is used for this second stage and 2) stalling only occurs around 1% of the time according to experimental results.

In [174] a simpler implementation with no acceleration techniques is presented. They note that the arithmetic coder is the bottleneck with the BPC being 5 times faster. Speed is thus increased by sharing the BPC among multiple coders working in parallel, since their simple implementation allows for replication with low overhead.

In [165], two different pipelining techniques are used: First, "traced pipelining" creates a pipeline for the most likely cases, where unlikely events create longer stalls at the benefit of likely events going through the pipeline without stalling. Also, cascading shifts are eliminated by looking ahead at the number of necessary shifts and performing them all at once.

Going further up the optimization ladder, [132] uses both pipelining and dual symbol processing improving on the already complex design by [58]. Dual processing is solved by having four different units calculating in parallel the four different scenarios resulting from using the LPS or MPS when subdividing the intervals. Pipelining is used to separate the A register update, C register update and byte output procedures.

A different pipelined approach comes from [6], where the three stages mentioned in other approaches are kept, but two more are added at the beginning by using two memory modules. First, context information (state and predicted symbol associated) is stored, with a second ROM outputting state change information. Whenever two consecutive contexts are equal, the second memory will be read with the updated state that is sent to the first one. This splits reading into a two-step process that accelerates the pipeline. Optimizations are also made in the shifting

step which turns out to be the critical path. A maximum of 7 shifts is allowed, stalling when the number is higher. This however occurs only in a marginal number of cases, so the limit is justified.

4.2.3 BPC implementation

For the BPC, the implementation presented here starts from the idea in [86] of processing columns of four at once, following the zig-zag pattern that the coder of JPEG2000 uses.

The general diagram of the coder is seen in Figure 4.17. Only the logic is shown, with memory modules hidden for visibility. The inputs are:

- The magnitude bits m for each of the four bits in the column, as well as the sign bits sg.
- A flag ic indicating, for each bit in the column, if it's been coded already by a previous pass.
- A flag fr indicating if this is the first refinement for each of the four samples which bits are in m and sg.
- A neighborhood of significance values s. Their sub-indices indicate which memory they belong to (p for previous, c for current, n for next) and the offset relative to the central output value.
- Flags $pass$ indicating which pass is being done (c for cleanup, s for significance, r for refinement).

It works as follows: The significance values are all fed into context generators GC, which are going to output the context for the bits m and s that need to be coded. A dependency exists for context generation. For that, two predictors for the cleanup CUP and significance SGN are used that speed up what otherwise would be a cascading of CGs. Information from these predictors, along with the $pass$ flags is used to emit the output context vector, which can contain up to 11 different context-bit pairs (of which up to 10 are valid in cleanup mode, 8 in significance and 4 in refinement).

The magnitude bits are used to generate flags abz (all bits zero) and fnz (first non zero) that are used for those predictions. The abz flag turns on when the full column is zero (this enables run-length mode under the cleanup pass). The fnz flag is used also for cleanup when the run-length is broken by an unexpected bit turning significant.

On the top-right of the diagram, the magnitude bits are combined with the ic flag and results from the predictors to generate the o_v (output valid) flags, that indicate which of the 11 CxD pairs in the output vector are valid. Just under it, the new significance values for the current column s_c are output from the predictors. Following down the diagram, the f_r and i_c flags are updated with the predictor information to store the refinement and coded status for the next pass. Finally, all of the information from the context generators is unified and output in the output bit o_b and output context o_c vectors, which contain all of the CxD pairs for the current pass, that will later be processed by a serializer.

Context generation can be seen in Figure 4.18. Neighboring significance values are checked, and three main outputs can be seen. On the top right, the magnitude refinement context mrc which selects between two contexts depending on if every neighbor is insignificant or not, and invokes a third in the case that the sample is not on its first refinement. (This is interpreted as unpredictable).

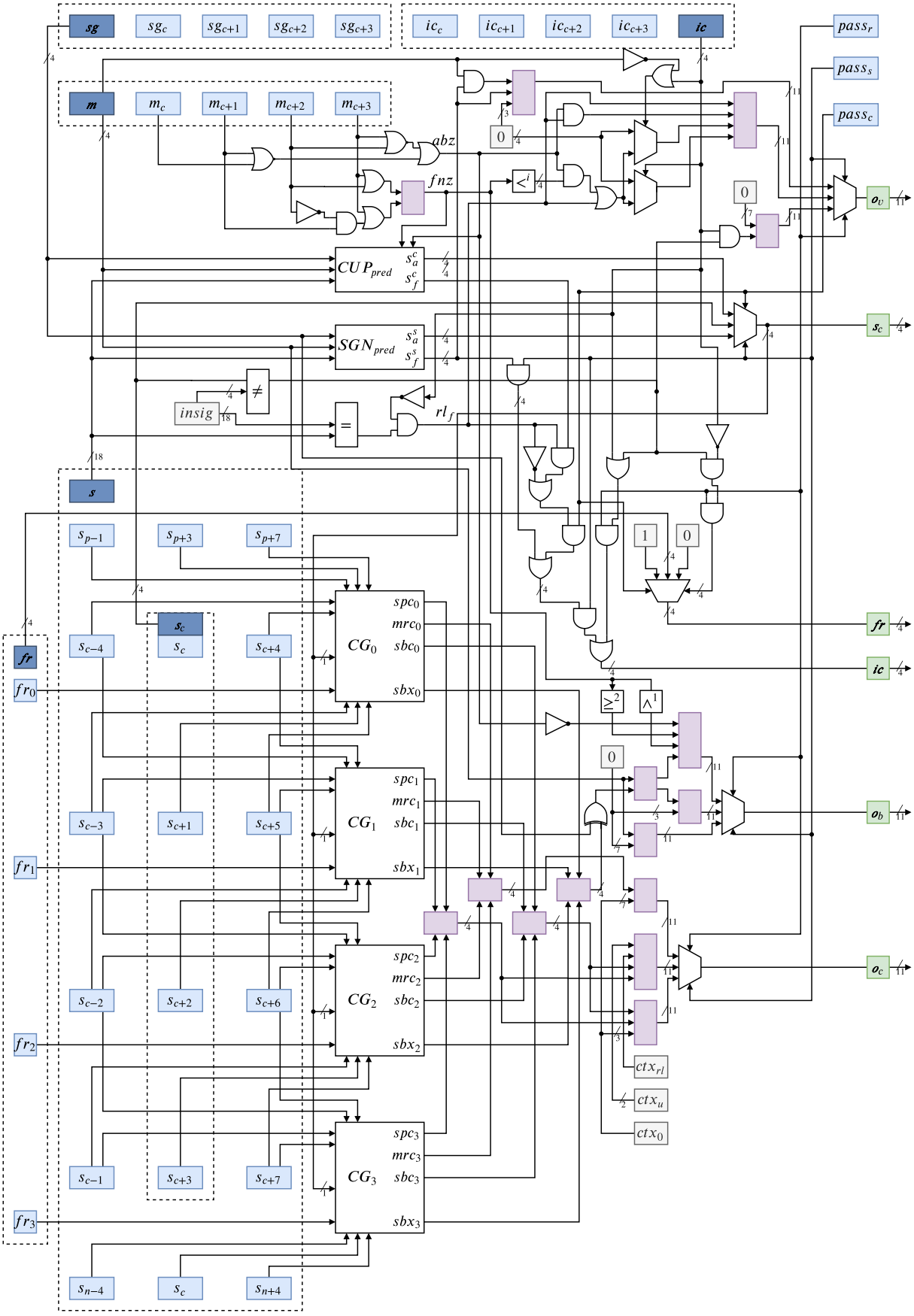

Figure 4.17: Core of the VYPEC compressor.

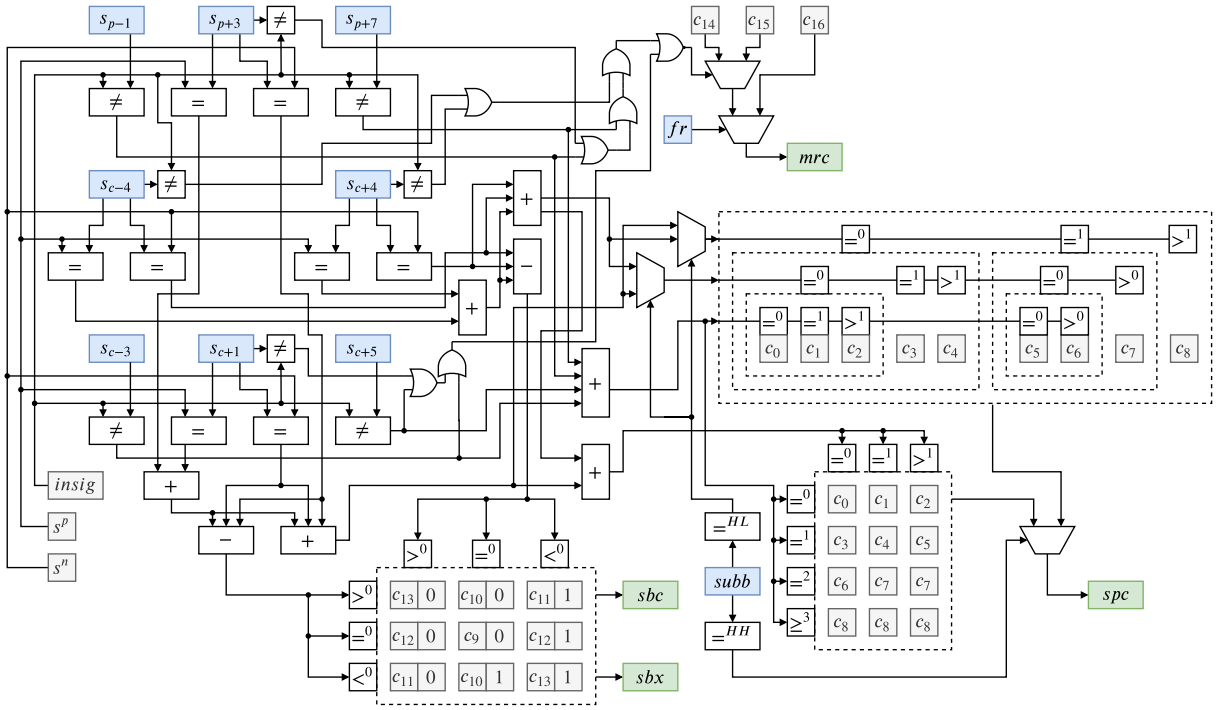

Figure 4.18: Context generation module outputting all contexts at once.

The sign bit context *sbc* and sign bit xor *sbx* bit are calculated on the bottom central part. They take into account the number of significant positive and negative neighbors, detecting patterns in horizontal and vertical neighbors.

Finally, the significance propagation context *spc* is the most complex of all. Two methods of obtaining it are run in parallel. They detect different kinds of patterns in the neighbors's significance values, and depending on if the block belongs to a HL, LL, LH or HH pass of the wavelet transform, use one or the other.

For cleanup, prediction is quite straightforward as seen in Figure 4.19. For the flags indicating if there is a significance change, the magnitude bit is checked along with the abz and fnz flags to see whether the bit is zero, or is the first in the column that is different than it. This, along with the sign, determines whether the new significance value will be preserved, or instead will be changed by that of the sign associated to the sample.

Things are more complex for the significance propagation prediction that is shown in Figure 4.20. Here, the inevitable cascading effect is seen as an orange path. But instead of feeding back into the CG modules, a simple path is used in the predictor module, greatly speeding up operation.

The new significance is either the one previously held or the one acquired by the sign bit associated to the sample being processed. Whether this new significance is valid or not depends on the neighboring samples being insignificant, and on the cascading feedback loop not converting any of the above samples in significant.

4.2.3.1 Module memory

Lots of different inputs are fed into the module that come from memories. First, the samples to be coded are stored in a four-word-wide memory that reads full columns of samples at once. From there, a mux selects, depending on the plane being coded, which is the magnitude bit m and which is the sign bit s for each of the four samples, feeding those to the module.

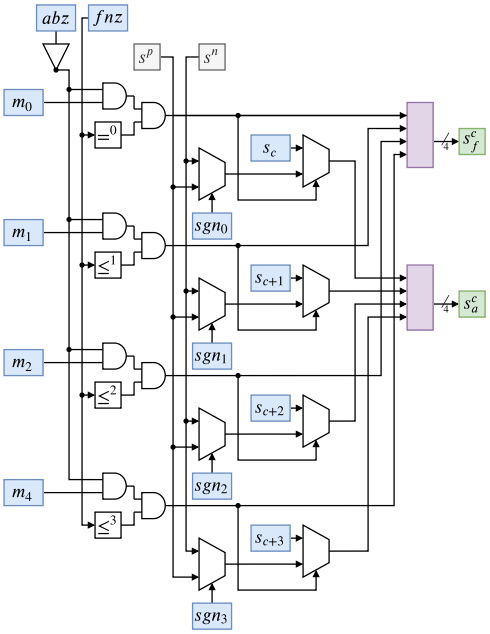

Figure 4.19: Prediction of the significance value after the cleanup pass.

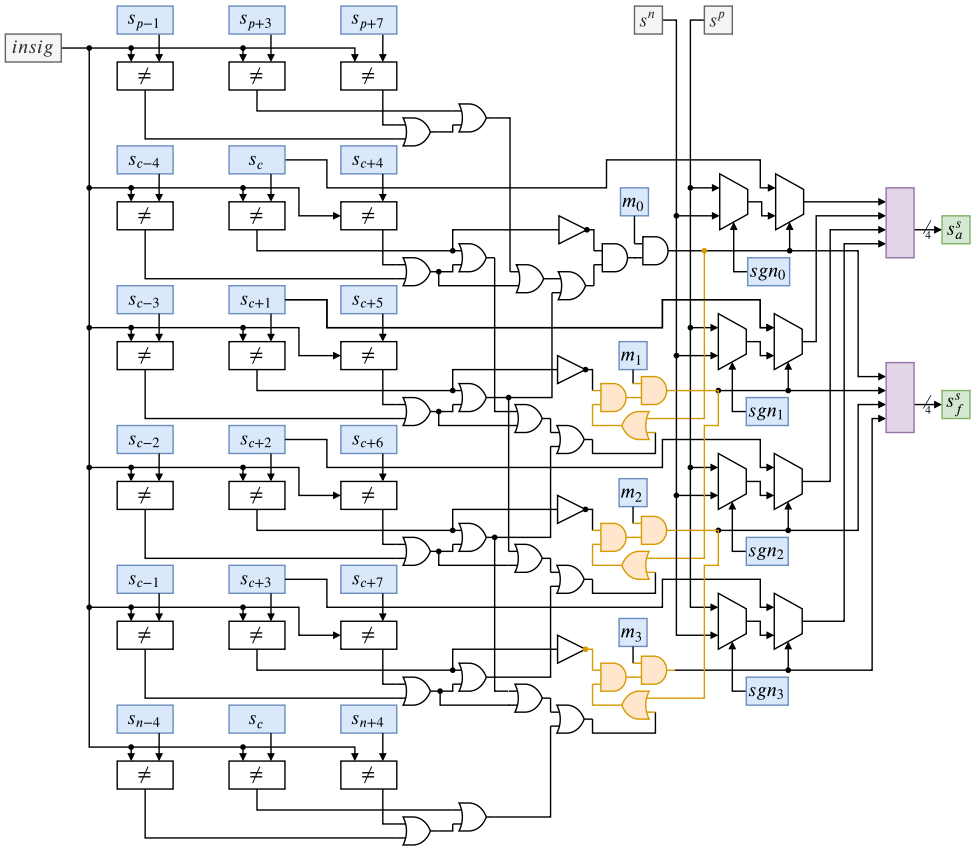

Figure 4.20: Prediction of the significance value after the significance propagation pass.

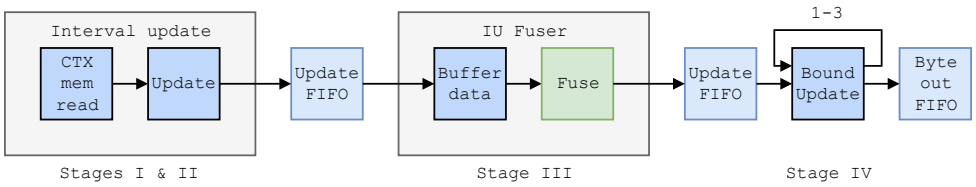

Figure 4.21: MQ-coder diagram.

The ic and fr flags are stored in the same manner, but this time in a memory with a bit depth of one. The main difference is that these flags will be updated by the coding process itself. To do that, and since they are initialized by the first pass, a FIFO memory is used that can more easily and faster address all the data.

The significance values are, in the same way, stored in a FIFO. But since the current column, as well as fourteen other neighboring samples are needed, the solution is to have three different FIFOs chained together. One corresponds to the current c row of columns, and the other two to the previous p and next n rows of columns. From each FIFO, three columns are output at once, the current column, the one to the left and the one to the right. A total of 36 values from which 18 are used. Four are updated and fed back into the FIFO chain.

4.2.4 MQ-coder implementation

Out of the two approaches seen in Section 4.2.2.2, pipelining and dual symbol processing, pipelining has been chosen here as the accelerating technique. Usually, dual symbol processing is done when traversal through the block is bit by bit, and at most two CxD pairs are generated at each clock cycle. This makes dual symbol processing useful to keep up with the output from the BPC. Here, pipelining is preferred since it results in a simpler design, and the problem of keeping up with CxD generation (which for this design can be up to 10 pairs per second) is solved by a CxD FIFO buffer between the BPC and MQ-coder.

The MQ-coder consists of three main parts separated by queues as seen in Figure 4.21.

First an interval update module keeps track of the value of the A register (the width of the interval). The probability change, whether it was a success or not and the amount of shift resulting from the probability change are passed to the next stage. These values are used by the bound update module to update the C register, outputting bytes as necessary. However, these three values passed can be merged together in some cases to feed less inputs to the last module (which might stall every so often). To help stalling be less prevalent, a fuser module combines pairs of the outputs from the interval update module by adding the probabilities and number of shifts together if the hit flag is either up or down for both outputs.

The first stage, or interval update stage, is seen in Figure 4.22. A clear separation is seen between both pipeline stages, highlighted by the orange squares that represent registers and memory modules.

First, the context memory is accessed. Seven outputs are produced, with all of the information related to that context. The probability estimate, the state to change if the most probably or least probable symbols are hit, the xor bit for the prediction change when probability goes below 0.5, the already shifted probability value (to avoid a shifter down the pipeline), the shift value itself, and the prediction.

The memory gets written from the outputs of the state ROM. It contains the same information for each of the different states of the predictor. Each context has a state associated to it, but accessing first the context and then the state information would take 2 cycles. Saving the information from the state ROM along with the context brings this down to 1 cycle. When

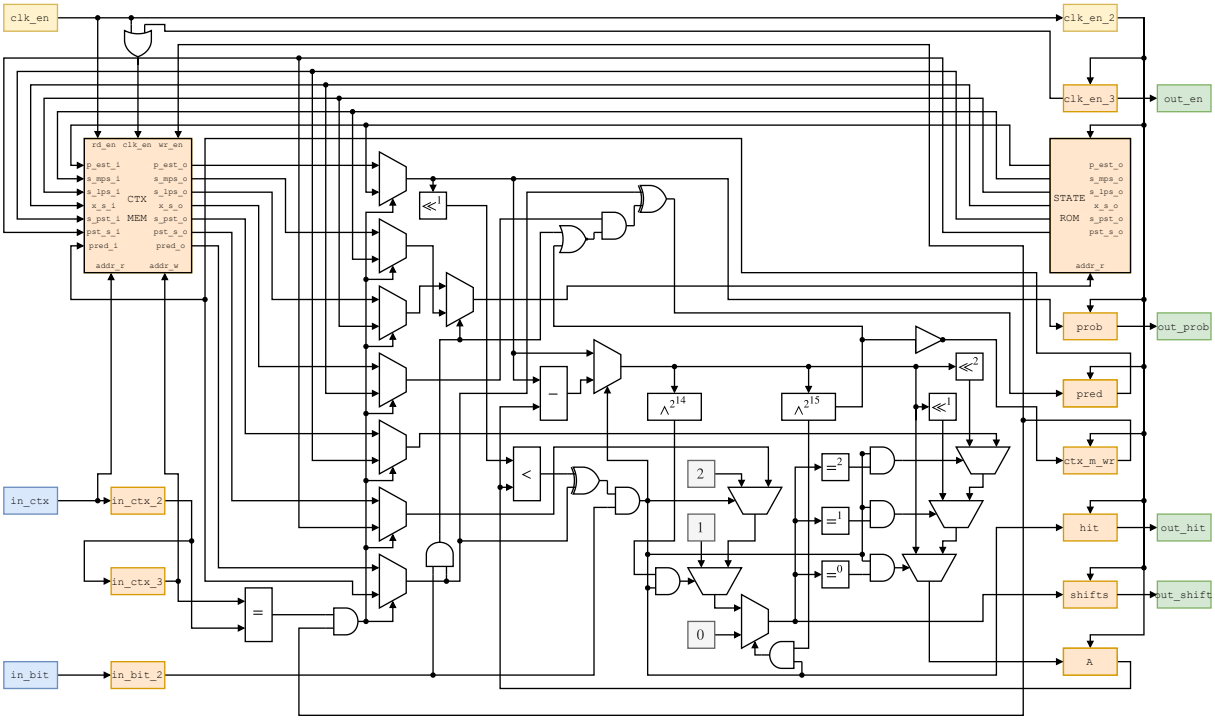

Figure 4.22: MQ-coder interval update module.

the same context appears twice consecutively, forwarding muxes get activated since the context memory is not yet updated. The values from the state ROM are read using the MPS and LPS state transitions, which are used if new symbol corresponds to the most probable or least probable according to the current state of the predictor.

Using the current context information, the prediction is adjusted, updating the A register. The key to speeding this up comes from the realization that A might be updated in one of four ways: Either it is not shifted, shifted once, twice, or the contents come from memory. In the last case, the number of shifts is already known and comes from memory. This results in time savings since a shifter is not needed, and values that otherwise would enter a barrel shifter are pre-calculated. The number of shifts, correspondingly, is either 0, 1, 2 or read from memory.

The number of shifts performed, probability value to add to the register C, and hit flag (indicating C needs to be updated with the probability) are sent to the next stage.

Lastly, the C register gets updated as seen in Figure 4.23. The input probability gets added, and shifting is performed.

A countdown timer keeps track of the number of bits already used up, and emits them when the leftmost byte of the C register gets filled. Depending on the value of that byte, different paths are taken, since the codes `0xff80` and up are reserved. If it is `0xff`, the following byte gets a `0` prepended, and if it is `0xfe`, care is taken not to overflow it when adding the next probability (since this would create a `0xff`).

The bytes are buffered to check for these patterns, and when no longer in used they are output along with a write flag to the output queue. Note that enabling of the circuitry depends on the queue not being full, so this ensures space is always available.

A finite state machine (Figure 4.24) keeps track of the different registers and controls the circuitry depending on the state. It also controls the input queue for data, and outputs flags to indicate the module's status.

The idle state is the one defaulted to after resetting. Whenever new data is available on the FIFO, the state turns to values read, and a shift is made to the C register. If more shifts are

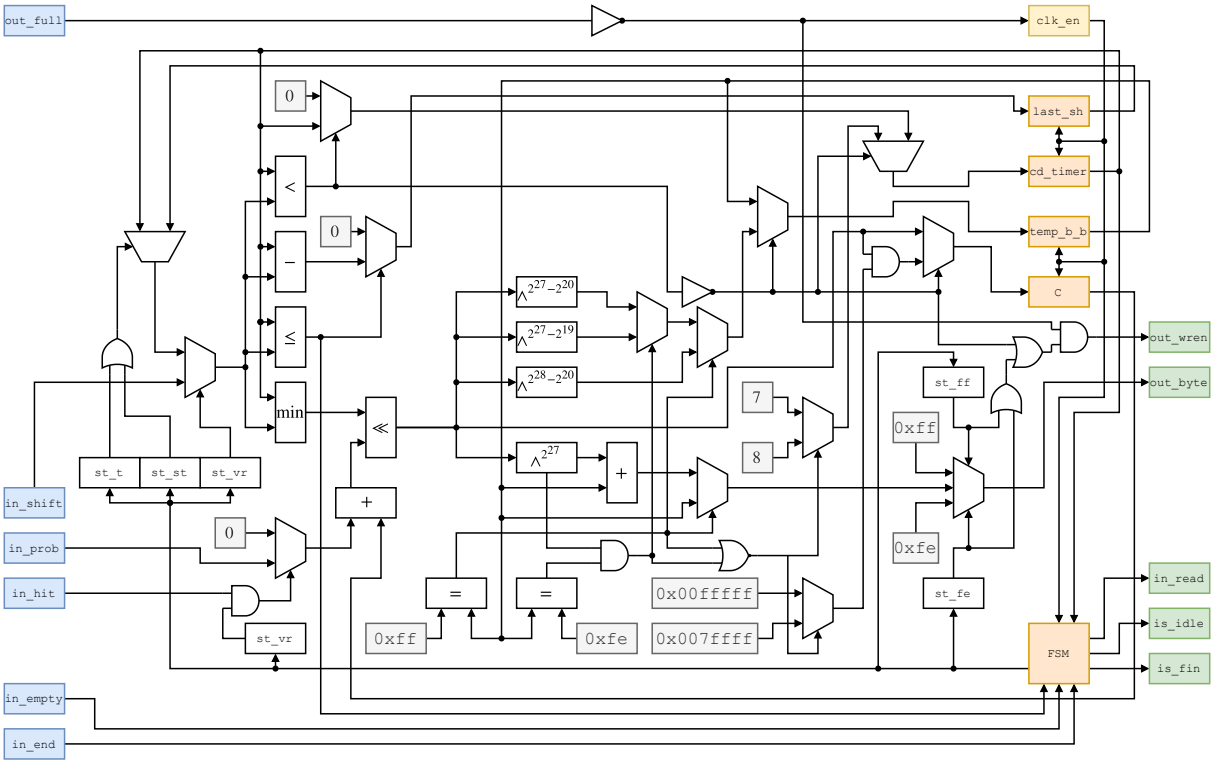

Figure 4.23: MQ-coder bound update module.

needed, the pipeline state is entered. From those two states, either the pipeline is maintained if necessary, or a new value is read, or the idle state is entered if more data is not available. Once finished, the termination sequence is entered. All remaining bytes will be dumped, after which the termination code **`0xfffe`** will be dumped to the output stream.

Each update, C might be shifted anywhere from 0 to 23 times, resulting in the output of up to three bytes. Instead of performing shifts bit by bit, full bytes are always shifted. This can result in stalling since only 1 byte is output per cycle. This could be solved at the expense of increasing the critical path by outputting all three bytes at once. However stalling has been found [165] to only happen around 1% of the time. As the critical path increases by over 1% in length if parallelized, stalling is preferred. In any case, the fuser module placed before will lower the amount of inputs to the bound update module to around 50%, so even when taking into account the stalling, the bound update keeps up with the interval update fast enough. In any case, for the extreme case where it did go slower, FIFOs ensure stalling happens with no data loss.

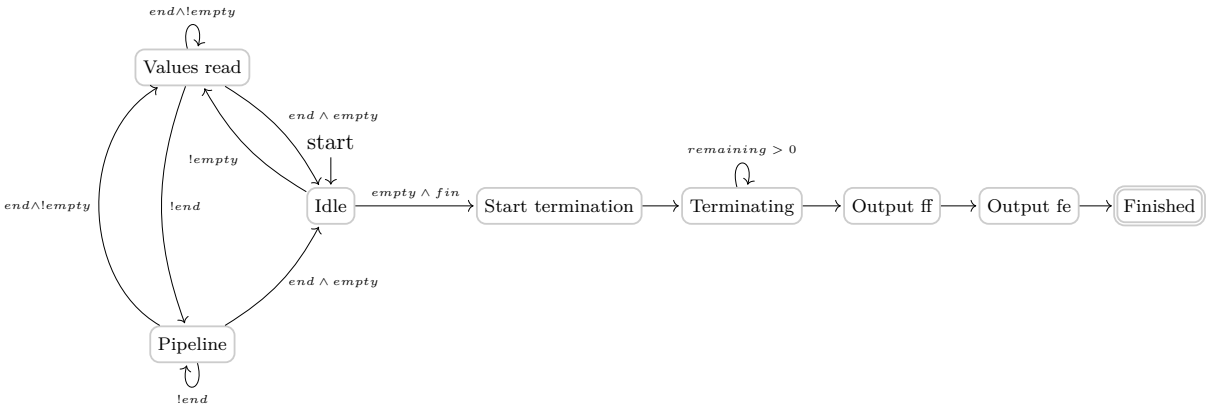

Figure 4.24: FSM of the bound update module within the MQ-coder.

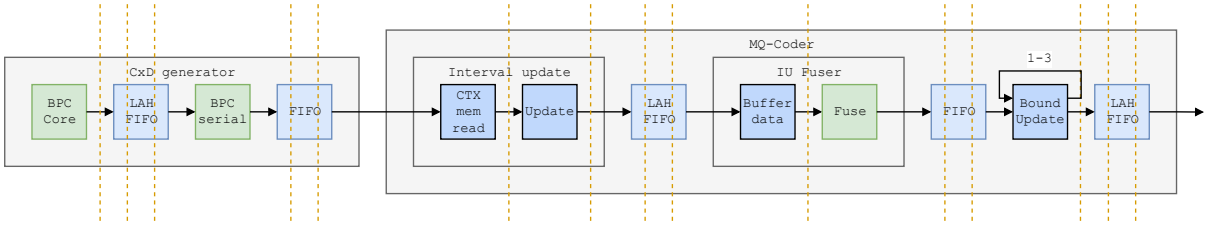

Figure 4.25: Tier 1 coder detailed pipelining.

4.2.5 Pipelined approach

Both the BPC and the MQ-coder are chained together to create the full tier 1 coder within the JPEG2000 compression flow. The diagram for this implementation is seen in Figure 4.25.

At first, the BPC creates the CxD pairs. Given the design, these are created at a variable rate in a parallel fashion, outputting anywhere from 0 to 10 pairs per cycle. To make the CxD flow uniform, a serializer is placed afterwards that takes the parallel output and selects only the valid pairs which are then fed into a single serial queue in the order specified by the standard. From here, CxD pairs flow to the MQ-coder.

It consists on the interval update module that deals with the value of the A register, followed by the fuser module, and lastly the bound update module which updates C and emits the consolidated bytes.

The FIFOs in between stages help mitigate the stalls from the bound update stage, and mitigate the droughts or excess of CxD pairs by having a big buffer in between the BPC and MQ-coder.

The full pipeline has a total of fifteen stages. Despite the deep pipeline, this has negligible impact on speed, since coding of a full 64×64 block of 16 bits per sample takes a minimum of $1024 \cdot 3 \cdot 14 + 1024 = 44032$ cycles, so filling the pipeline takes at most $15/44032 = 0.03\%$ of the total cycles.

4.3 LCPLC

Lastly, LCPLC is a more novel algorithm that targets near-lossless compression, being able to compress losslessly and to introduce a slight loss if necessary. In this aspect, it is similar to the newly released CCSDS 123.0-B-2[34], which is the second revision of CCSDS 123.0-B-1[32].

4.3.1 Software

Software wise, LCPLC does not have the number of implementations that CCSDS has. It was designed to be used in the Exomars mission. Unfortunately, being a specific algorithm, public implementations do not exist to the best of the author's knowledge.

In any case, designing the algorithm in software was as straightforward as following the equations in the original work [5]. As with the designs for CCSDS 123.B-1 and JYPEC, the important part was instrumenting the code to be able to trace any of the intermediate values in the data dependency chain. Specifically for this algorithm, and since its implementation was going to be highly pipelined, each hardware module has an equivalent software module that emulates it. This way, simulation values can be easily compared, and errors quickly detected.

The code can be accessed in [20]. It reuses the I/O and distortion measures from JYPEC to simplify development.

4.3.2 Previous work

A different approach is taken in LCPLC than in the other two algorithms. Since backwards dependencies are very few, a highly pipelined design was thought to be the best option. Few implementations of this algorithm exist, taking advantage of data parallelism [179] or coarse-grained task parallelism [76, 178] through HLS.

Given the simple nature of LCPLC's operations, a novel and simple approach is proposed: create simple modules that operate over streams of data, targeting fine-grained parallelism. Modules are chained together to create the complex functionality of the whole algorithm. A common protocol is mandatory to synchronize data transfers, and in this case it is the AXI-stream [14] protocol.

Many streams will be working in parallel at the different stages of the algorithm. Streams will split to parallelize execution and re-joined later for further processing. Latencies on each stream are different, so care is taken to provide proper synchronization. Furthermore, FIFOs are carefully allocated to avoid deadlocks due to data streams waiting to be synchronized with data that hasn't been yet produced.

4.3.3 The AXI-Stream protocol

The AXI-Stream protocol is a quite simple way of interconnecting modules that lie close together. A transfer can be done each clock cycle, since both the emitter and receiver of the data are clock synchronized. This makes it extremely efficient since it maintains the maximum efficiency of 1 clock/data transfer of a tightly coupled system while allowing the flexibility of a generic bus specification.

The AXI-Stream protocol connects just one emitter with one receiver, so it is a point-to-point connection. Its basic four lanes[1], seen in Figure 4.26, are:

- **Valid** flag. This flag is raised by the emitter, indicating that the data (also set by the emitter), as well as any information that accompanies it, is valid *in this clock cycle*.
- **Ready** flag. This flag, along with the valid flag, takes care of the control of the bus. When raised (by the receiver), it indicates that the receiver can read information presented on the bus *in this clock cycle*.
- **Data** lane. A variable-width data lane containing the necessary information for a transfer.
- **Last** flag. An optional flag that signals when a burst of data is finished (e.g. if processing packets of some kind, it can indicate when the packet is finished).

If the emitter sees the ready flag go up while it has the valid flag up, at the next clock cycle it can initiate a new transfer, and can safely assume that the receiver has read the data since it was ready.

Likewise, if the receiver sees the valid flag, it must operate on the input data since next cycle the emitter could be initiating the next transfer.

The receiver will usually have some kind of buffer which, when empty, raises the ready flag and waits for a valid signal before storing the contents of the data lane.

Whenever a transfer is made and the last flag is raised, this indicates that the burst is completed. A burst might just contain one transfer, and as such one can be initiated each clock cycle.

[1] More optional lanes exist, but are basically extensions of the data lane since they add no functionality protocol wise.

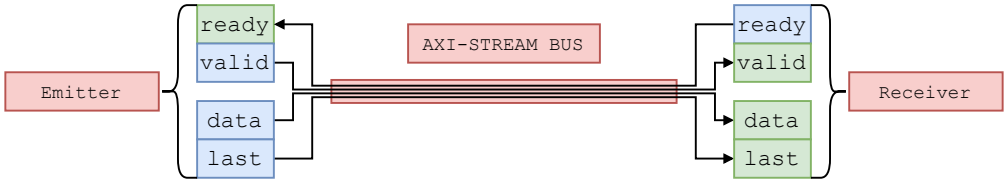

Figure 4.26: Simple diagram of an AXI bus. Valid and ready control synchronization. Data and last control data flow.

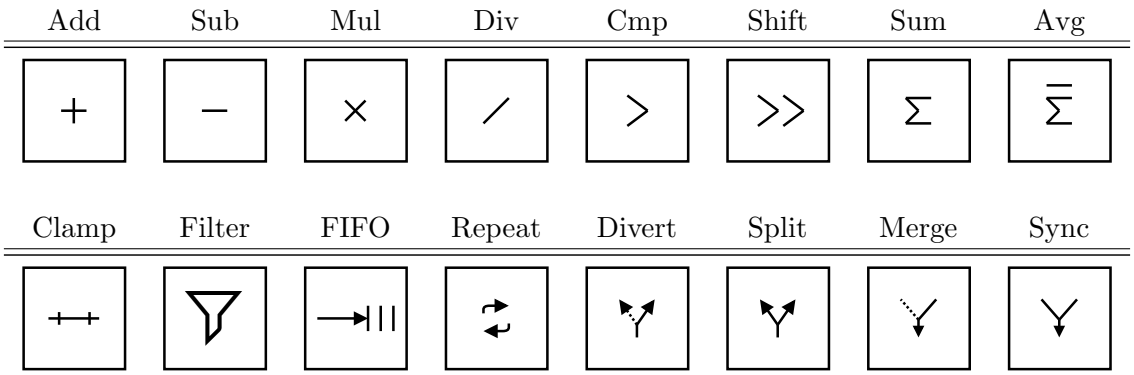

Table 4.1: The basic building blocks for the LCPLC algorithm. All follow the AXI4-Stream protocol on inputs and outputs.

4.3.3.1 Adapting the protocol

For the purpose of the LCPLC algorithm, the basic lanes have been used, but having four bits for the "last" lane. This allows to indicate whenever a row, slice, block or image is finished, giving precise information about where in the image the compressor is working. This, which implies a communication overhead, greatly reduces the otherwise even bigger overhead of having a counter to keep track of the current position. It also makes the core more flexible, since a counter needs to be configured each time with its limits, while a flag-based system can transition seamlessly between different data sizes.

All the following diagrams will contain interconnected modules, where arrows indicate AXI-Stream buses. The direction of the arrow indicates the direction of the data. However the ready flag will always be moving in the opposite direction.

Lower level logic is not considered interesting (aside from the occasional FSM diagram) since even simple adders have been encapsulated in AXI-Stream modules. The complexity of the design lies in how these modules interconnect, and not in the modules themselves. Nonetheless, care is taken to explain in detail the inner workings of the slightly more complex modules.

If that is still not enough detail, the reader is referred to [20] for concrete information about the implementation by looking at the code itself.

4.3.4 Basic modules

Before delving into LCPLC's implementation, first a few remarks are given about the simple modules that are listed in Table 4.1.

- The **add**, **sub**, **mul** and **div** modules all work in the same fashion: They take two input AXI-Streams and output a single combined output Stream. First, the inputs are synchronized so that a sample is read from both inputs at the same time. Then, both samples go through the operation pipeline. This takes a variable number of cycles. For additions

and substractions, 1 cycle is enough to perform the operation. For multiplications under 18×18 bits, three cycles are used, and four for multiplications under 25×25 bits (this is due to using 1 and 2 DSPs respectively). Divisions are done with a simple shift-and-substract algorithm, which takes up to n cycles where n is the difference in bits between the dividend and divisor.

Inputs can be configured signed or unsigned, with variable bit-widths which might impact latency as previously seen. All modules except the divisor are internally pipelined, and can process 1 sample/cycle once the pipeline is full. The divisor is not pipelined since divisions naturally do not overlap when executing the algorithm, and so the same registers can be reused for the division, saving resources.

- The **cmp** module performs a variety of comparisons between two input streams and puts the result in an output stream. The inputs are again synchronized like in the arithmetic modules.
- The **shift** module again syncs two input streams of data and shift amount, and outputs the data shifted by its corresponding amount. The maximum amount of shifts that are done per cycle can be configured (to gain speed) by the user. The stages are pipelined, and so despite the latency gain, throughput once the pipeline is full is maintained at 1 shift/cycle.
- The **sum** module works a bit differently. Instead of producing the same amount of outputs than of inputs, it keeps an internal accumulator growing until the "last" flag is passed along with a piece of data. It is at that moment when it outputs the current accumulator, and resets it for the next sample. It thus adapts to any block size, since counters and limits are not needed. The only thing that needs to be configured is the maximum amount of inputs that will be accumulated, to make room for the accumulator itself. If the input stream brings more samples than that number before the "last" flag is raised, then the output might have overflown. This is not a problem in practice since the number of samples before the "last" flag goes up is always bounded (less or equal than the block size).

  The **average** module adds a counter to the accumulation process. When outputting, the accumulator is divided by the counter with a division module. If the counter is a power of two, the division process is skipped and instead a shift is performed for an increase in performance.
- The **clamp** module takes an input value and, if outside a pre-configured interval, rounds it to the nearest value that lies within that interval. This is done in two (pipelined) stages for increased performance.
- The **filter** module feeds off two different streams, one of which is a flag stream which indicates, for each sample in the data stream, whether or not to keep it. The output will contain only those samples that are kept.
- The **FIFO** module serves as a buffer when two different streams split and join later down the line, but one has more stages than the other. In that case, failing to buffer data might result in the bigger pipeline stalling because of lack of input data, which would be waiting for the smaller pipeline to empty. If a feedback loop is present, this might even result in a deadlock. As a result, FIFOs are important to keep the machinery running.
- The **repeat** module receives two streams, a data and a flag stream. The data is repeated until the flag is brought up, at which point the next value in the data stream starts to get repeated. This is useful for values that need to be used many times for the same calculation as this doesn't consume them right away.
- The **divert** module alternatively redirects a stream between two output ports, depending on the value of an input flag that is synchronized with the input data stream.

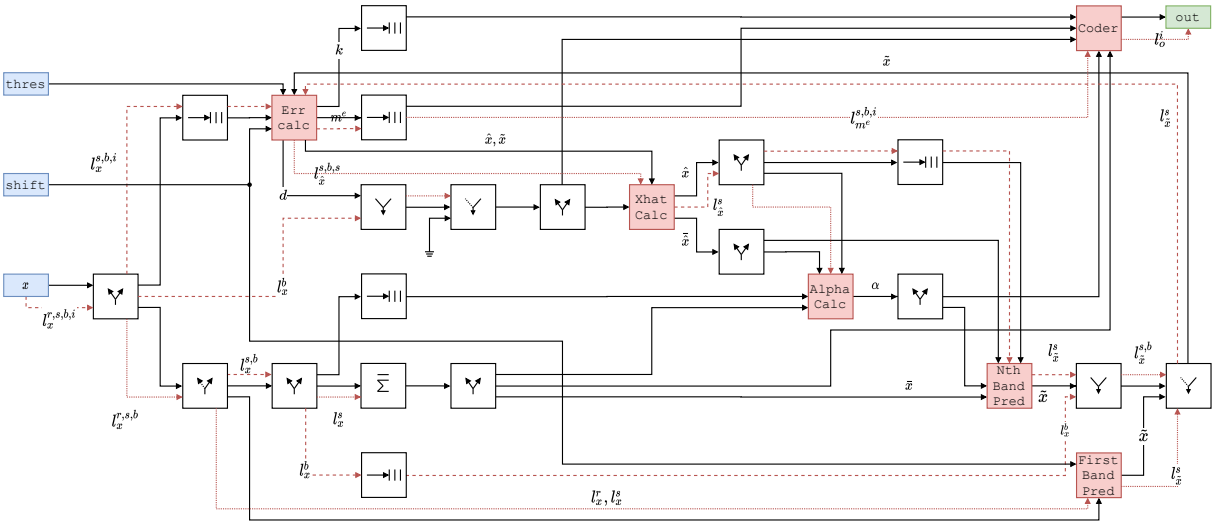

Figure 4.27: The full diagram for the LCPLC compressor. First band pred: Figure 4.28. Nth band pred: Figure 4.30. Err calc: Figure 4.31. Xhat calc: Figure 4.32. Alpha calc: Figure 4.33. Coder: Figure 4.34

- The **split** module, instead of sending the data to only one of its outputs, sends it to every output. It makes sure that all the output streams have read the output data before putting up the next value, ensuring no output stream loses samples. For this, a double buffer is used where first the input sample is stored, and then a buffer is placed for each output from which the output streams read individually. Once this second buffer is emptied, the first is dumped there and the process begins again.

- The **merge** module feeds off two data streams and are alternatively output to the output data stream. A flag comes with both input streams, and whenever the flag is raised in the "active" stream, it is switched to the other until its flag is raised and the process repeated.

- The **sync** module works also on two input streams, but this time the output contains the combined data of the two inputs. The input streams are first synchronized, and whenever samples from both of them are available, a single fused value is output.

4.3.5 The core

The main view for the LCPLC core is seen in Figure 4.27. There is just one Axi-Stream input and one Axi-Stream output. The input is the values x corresponding to the different samples of the image. Each sample comes accompanied with a set of flags l^r, l^s, l^b, l^i which respectively indicate if that sample is the last of its row, slice, block or image. These inputs alone control all the circuitry that comes after, since bound detection is made using them and not counters. Thus, the image size is unbounded. However, internal queues are sized according to a (settable) maximum image size, so the user must decide what the maximum input sizes ($N_X \times N_Y \times N_Z$) will be beforehand. Anything below that size can be compressed. Anything above might stall the circuit.

The output is a sequence of bytes, of which the last one will be marked. This mark indicates that the stream coming from input x has been successfully compressed up to the sample marked with the l^i flag, and that further bytes will correspond to the next run of samples starting after.

Each slice is processed independently, with cumulative results form one slice used to process the next. First, data is split into two streams: one will be used to calculate the error by comparing the raw values against the predictions, and the other will be used to calculate the

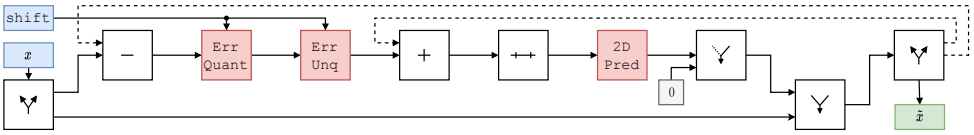

Figure 4.28: Module for the prediction of the first band.

predictions themselves. The prediction stream is also diverted into two, where the first slice will feed to the first band predictor, and the rest will feed to the nth band predictor.

Predictions from both predictors are merged and go on to the error calculation module, where they combine with the previously split raw input stream. There, the distortion is calculated as the mean squared error, and depending on the distortion threshold, the final value going forward will either be the raw or predicted values.

After deciding which one to use, the values will feed to the alpha module, where along with the next slice they will create the alpha value required for prediction of the nth slices. Then, the cycle keeps repeating until the end of the block is reached, where the input again switches to the first slice predictor.

4.3.5.1 First band prediction

Prediction for the first band is quite simple as shown in Figures 4.28 and 4.29. First, the samples go through the quantization process (in case lossy compression is desired), and then continue on to the predictor itself. The feedback loop is primed with a 0 for the first sample prediction, and afterwards it keeps emitting predictions until the slice is finished.

For quantization, samples are rounded to the nearest value multiple of the chosen power of two 2^n, by adding 2^{n-1} then shifting right n times. This quantization method is preferred to truncating since it minimizes the error according to the distortion measure used for this algorithm (Equation (3.55)). Dequantization is just a left shift of n.

To keep track of the position within the slice, a FSM is used. Simply enough, it starts at the first sample, for which the prediction is zero. Afterwards, it moves to the "first row" prediction mode where the prediction is equal to the previous sample. Finally, when the end of the row is reached, a third mode is entered where the sample above the current one (same position but in the previous row) is also used. If the sample is the first within the current row, the prediction is equal to the above sample. Otherwise, the above and previous samples are added together then shifted by one to get the current prediction.

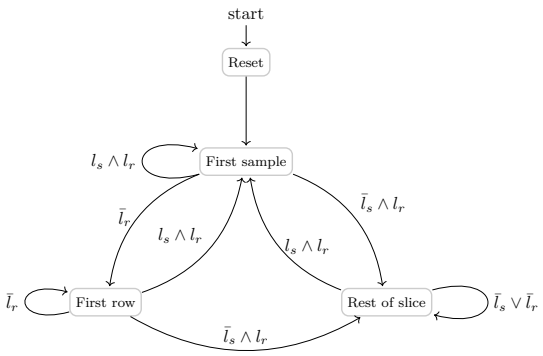

Figure 4.29: The FSM for the 2D predictor used in LCPLC for the first slice of each block.

4.3.5.2 Nth band prediction

Also simple is the prediction of any slice after the first one, seen in Figure 4.30. The formula from Equation (3.54) is taken into circuit form by chaining a subtraction, multiplication, and addition modules. Since α and μ values are generated once per slice but here are needed as many times as input samples are, they are repeated until the input samples run dry.

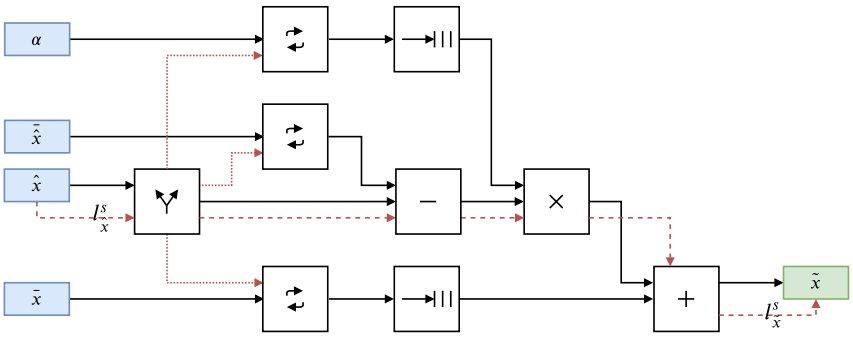

Figure 4.30: LCPLC module for the prediction of bands after the first.

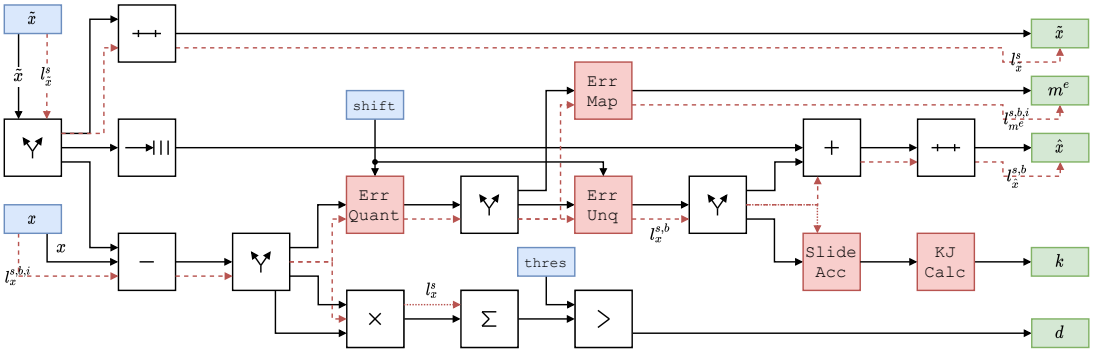

Figure 4.31: The error calculation module.

4.3.5.3 Error calculation

Both raw and predicted values are fed into the error module (Figure 4.31). Two main functions lie within: generate the outputs for the coder, and generate the values for the feedback loop.

Regarding the former, both raw and predicted values are subtracted and then the distortion is calculated and compared with the threshold. The d flag carries this information, indicating with either 0 or 1 if the distortion met the threshold or not. This decides what is to be coded, and what values to use for the prediction of the next band.

The same error that goes onto the d flag calculation is also quantized and mapped to produce the mapped residuals that are to be coded if under threshold. They get dequantized back to output the decoded values, as well as to generate the parameter k for the output coder.

Error mapping is done following a simple formula that maps all values to positive integers for the coder to process:

$$m^{\hat{e}}_{m,n,i} = \begin{cases} (|\hat{e}_{m,n,i} \ll 1|) - 1 & \hat{e}_{m,n,i} > 0 \\ |\hat{e}_{m,n,i}| \ll 1 & \hat{e}_{m,n,i} \leq 0 \end{cases} \tag{4.2}$$

The sliding accumulator module keeps a running sum of the 32 latest dequantized errors. A FIFO queue stores values for 32 cycles and then an add-subtract combo updates the accumulator value by adding the latest and subtracting the queue's output. Since the accumulator won't have 32 samples stored when starting, the actual number is also output along with the accumulated value. That way, the KJ calculation module can obtain the k parameter for coding based on an estimated average of the most recent errors seen.

To do so, a pipeline obtains the coding parameter k as follows, using R and J from Equations (3.56) and (3.57), following Equation (3.58):

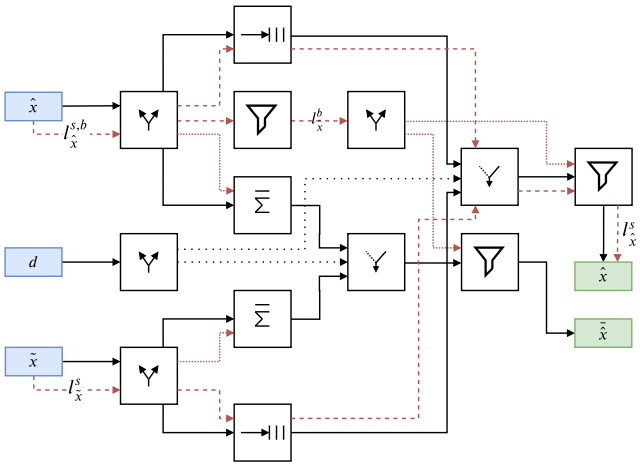

Figure 4.32: Module for the selection of the final value to be coded, depending on distortion.

$$\begin{aligned} b^J_{m,n,i} &= \lceil log[2] J_{m,n,i} \rceil \\ R^S_{m,n,i} &= R_{m,n,i} \gg b^J_{m,n,i} \\ k_{m,n,i} &= \left\lceil \log_2 \left(R^s_{m,n,i} \right) \right\rceil + 1 \end{aligned} \tag{4.3}$$

Note that this is different from the original design where k is the smallest value such that $J \ll k \leq R$. There, the straightforward implementation would be to shift right R for every possible value of k, checking against J the first inequality $J \leq R \gg k$ that holds. This would imply as many comparators as values k might hold (32) as well as the decision muxes, which is too expensive.

Instead, J is assumed to be of the form $J = 2^j$, then R is shifted right by j, which can only range from 1 to 5. then, k is equal to the position of the leftmost nonzero bit of $k \gg j$. This is much faster and simpler than the original version, causing no observable difference in compression efficiency.

4.3.5.4 Coded value selector

Depending on the distortion flag d, values $\hat{x}$ can come from two different streams, as well as their average. Since that is used in the feedback loop, the first thing that is done is to pre-calculate average values from both streams to avoid losing time after deciding which stream to use. This introduces the overhead of an additional average module, but reduces time by as many cycles as samples in a slice.

In any case, samples from both streams, as well as averages, wait on FIFO queues until the flag is ready. Then, they are filtered and only the ones from the correct stream used, as seen in Figure 4.32.

4.3.5.5 Alpha calculation

Figure 4.33 shows the process of obtaining the α value used in prediction.

Raw values, as well as decoded values are fed into two different lines, which generate the differences with respect to their respective average values. Those differences get multiplied with each other and then accumulated. The accumulated values are then divided to generate the α value, according to Equation (3.53).

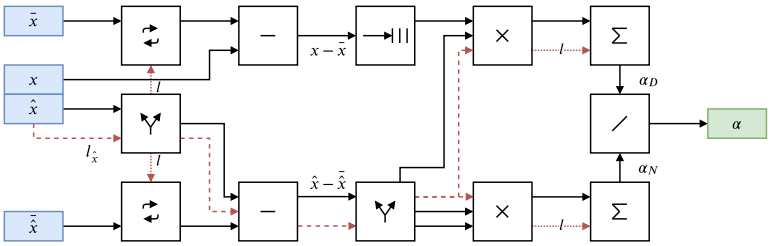

Figure 4.33: Module for the calculation of the α value.

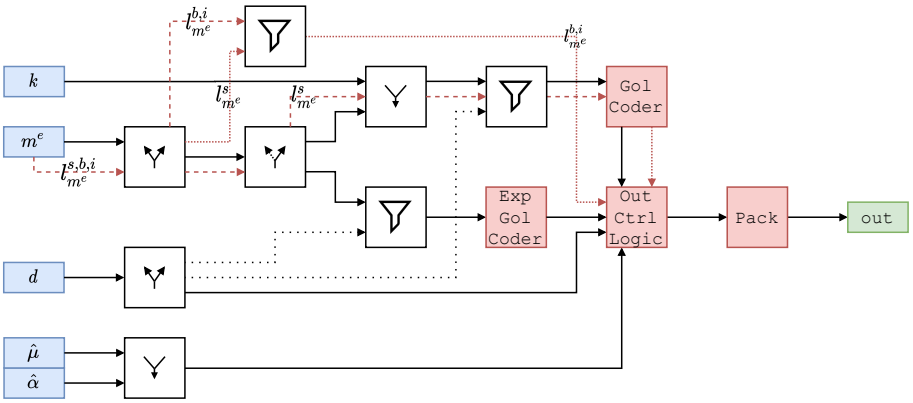

Figure 4.34: Module for the coding of the outputs in LCPLC.

4.3.5.6 Coder

Finally, the values that exit the main prediction loop get coded. $\hat{\mu}$ and $\hat{\alpha}$ are first coded. Then, the d flag decides whether or not to code the mapped residuals and k values based on the prediction accuracy. Thus, either they are not coded, which will imply having the prediction in the reconstructed image (lossy), or they are coded and the original data will be reconstructed (lossless). The coder is shown in Figure 4.34.

Care is taken to send the first instance of each run of mapped errors to the exp-golomb coder, while the rest go through the normal golomb coder along with parameter k.

For the exp-golomb coder, a code and code length are output for each input value v:

$$\begin{aligned} c^{ez}(v) &= v+1 \\ l_c^{ez}(v) &= \lceil \log_2 (c^{ez}(v)) \rceil \ll 1+1 \end{aligned} \tag{4.4}$$

Thus, the output stream will contain as many zeros as the length of the code, followed by the code itself (i.e: a value v of n bits of magnitude will take up $2n$ bits in the output).

The golomb coder uses an additional parameter k for computing the code and code length:

$$\begin{aligned} r(v) &= v \wedge \left(2^k - 1\right) \\ q(v) &= v \gg k \\ c^g(v) &= ((1 \ll q(v) - 1) \ll (k+1)) \vee r(v) \\ l_c^g(v) &= k + q(v) + 1 \end{aligned} \tag{4.5}$$

There is a rare case where the code might be too long, and is split across multiple output values.

Finally, a control logic module selects the order in which all of this data gets output (Figure 4.35).

At first, the d flag for the first slice is discarded since it is always losslessly coded. Then, the

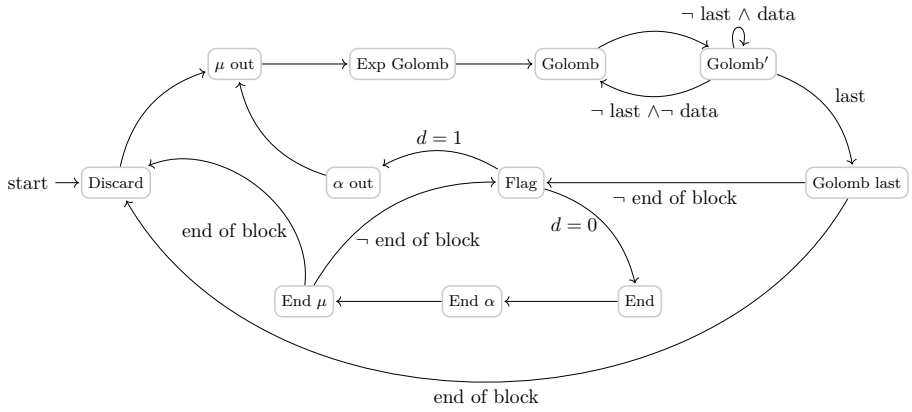

Figure 4.35: Finite state machine for the control logic of the coder in LCPLC. Note that all transitions occur when flags for data synchronization are up. This is not depicted for simplicity.

average value of the first slice is output. Note how the alpha value for the first slice is never output since it doesn't exist.

Afterwards, golomb coding comes into play. First, the exp-golomb coder will output the sample it codes, followed by the normal golomb coder. Three states are used to increase speed. A default golomb state that waits until data is ready, a *primed* state that both reads inputs values and outputs coded ones, and a last state that just outputs the last value. After that, the end of block flag is checked. If up, the cycle starts again on the first slice of the next block.

If the block hasn't ended, that means more slices are present. The flag to indicate if they are coded or not is output. If it indicates a skip, a path is taken to output just α and μ, and then check again if the block is finished. Otherwise, the golomb-coded samples are also output.

On each state, the FSM sends the packer the compressed it needs to emit. This is done via pairs of values which indicate both the code itself and its length. For example, a code might be `00010001` with length 5, indicating that the final output must contain just `10001`. The packer emits byte by byte, so it keeps a buffer with the bits that do not amount to a byte yet. Once a new value comes in, it is placed on the left of the buffer and, if 8 bits are present, a byte is emitted and a shift of 8 units done. Since incoming codes can be quite long, the resulting buffer might contain more than 8 bits after shifting. If that is the case, the packer stops accepting new inputs until the buffer is below 8 bits. When the last signal is received, what is left in the buffer is output, padded with zeros.

4.3.6 Doubling the algorithm's speed

With the main algorithm already explained, the dependencies can be analyzed. The main stages can be divided in five: average calculation, alpha calculation, prediction, error calculation and coding. They are slightly different for the first block slice, which doesn't include the alpha calculation, but otherwise are the same for the rest of slices (which amount to hundreds in most cases).

Average calculation has to complete before alpha calculation, but that is not a problem since all averages could be calculated beforehand. Then, the alpha value is used for prediction. The problem is that a full slice needs to go through the alpha module before outputting the α value. So those operations cannot overlap. Error calculation however can be done overlapping prediction, and its output goes to the coder which doesn't feed back into the system so for pipelining purposes can be ignored.

The only problematic slice-to-slice dependency comes because the error calculation of a slice

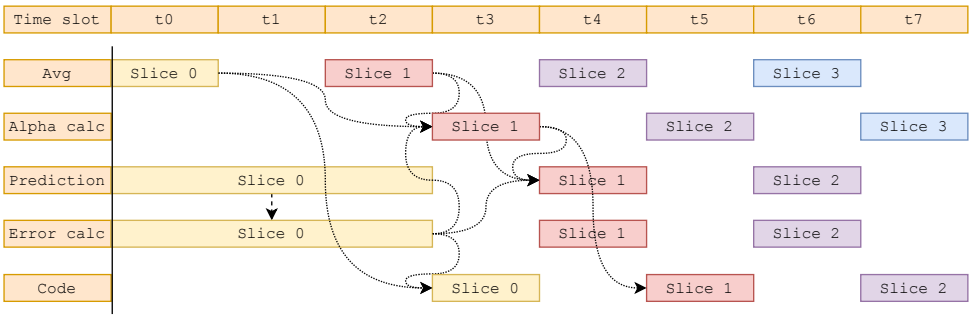

Figure 4.36: Timing diagram for the first approach to the LCPLC compressor.

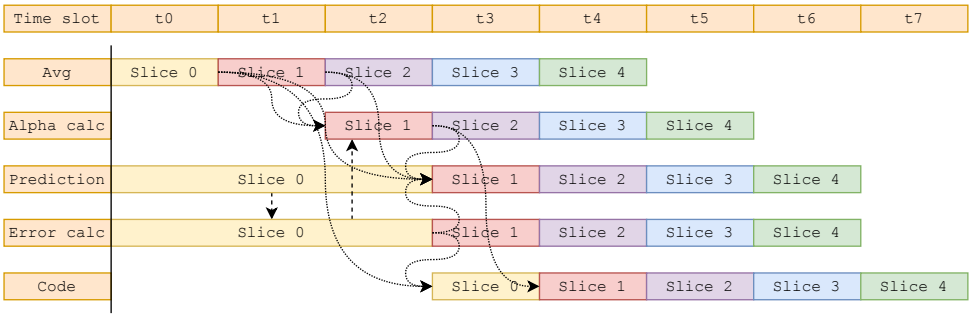

Figure 4.37: Timing diagram for the second approach to the LCPLC compressor. By forwarding data, stalls in between slots are eliminated, and perfect pipelining is achieved.

s_i needs to fully complete before the alpha calculation of the next slice commences, since the average value used for α calculation depends on the decoded value, which in turn depends on the distortion: if it's above threshold, the real average value will be used, and otherwise the average of the decoded values will be used instead.

So, when pipelining slices, not all stages can overlap, producing a stall of the order of the slice size, as seen in Figure 4.36.

However this problem can be avoided if the problematic dependency is removed. This is done via a slight modification of the algorithm. Instead of using the *decoded* average value, the *encoded* average value will be used. Both differ, but over a full slice, it is to be expected that, even in the lossy case, the average value of a full slice of decoded values will be close to the raw average value. Indeed, no difference was found in compression ratio when using this shortcut technique.

To fully achieve continuous operation, another trick has to be used: outputs $\hat{x}$ and $\tilde{x}$ from the error calculation module, which would have to wait until the d flag is ready for passing on to the α calculation, are fed into two different modules. That way, both possible α values are calculated and ready when the d flag is output. In that moment, all slice values for $\hat{x}$ and $\tilde{x}$, which were waiting behind a selector, go through, being selected based on d value. As soon as the prediction of one slice is finished, the alpha value can directly enter the nth band predictor since it was pre-calculated due to using the *encoded* mean. The next slice can thus start its cycle immediately. The timing diagram for this is seen in Figure 4.37.

All in all, the resulting hardware is slightly more complex, as seen in Figure 4.38. Queues are a bit longer to ensure no deadlocks occur (the diagram assumes a 16×16 slice, with values over the FIFOs indicating their depth), and two alpha modules instead of one are present. The input control for the alpha calculation modules is also a bit more complex. Other than that, the rest of the circuit is the same. The total overhead in size is studied in Section 5.5.2, but it is negligible compared to a gain of double the throughput as is the case.

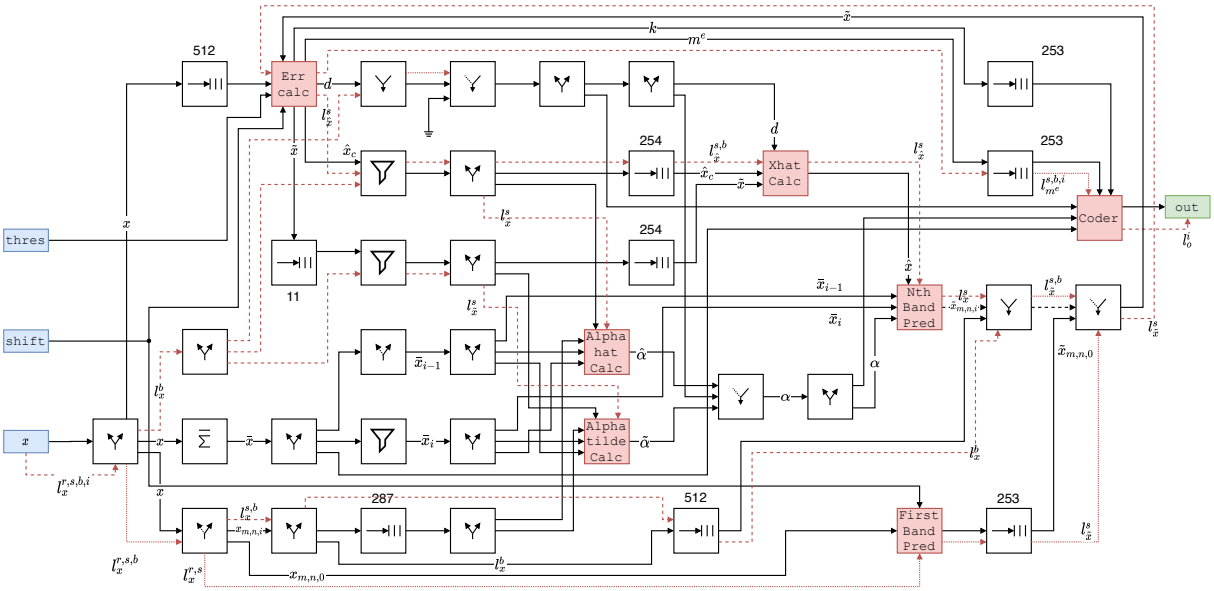

Figure 4.38: The full diagram for the advanced LCPLC compressor.

Chapter 5

Results

In this chapter, results for all three algorithms will be shown. Both from the algorithmic point of view (distortion-ratio performance) and from the implementation point of view (power consumption, frequency, throughput, resources, etc). It is important to make this distinction, since algorithms might have great algorithmic performance but lack on implementability, and vice-versa. This approach will give a great overview of the characteristics of each one, presented in Chapters 3 and 4.

Regarding implementation, three different FPGAs (Table 5.1) have been targeted throughout the experiments: Two are radiation-hardened and ready for spaceborne missions, while the third has greater processing capabilities but is not protected against radiation. This one will give a good idea of the maximum achievable performance for the implementations, while the former two will give an idea of the usability of these implementations on a more realistic scenario. Functionality tests have been performed in the former, while syntheses have targeted all three.

To test the algorithm, non-FPGA runs of the algorithms have also been done on a computer. This is a DELL XPS 13 9360 running Windows 10 as the OS. It has an i7-7500U processor [98] running at $2.7GHz$ with 4MB of cache, two physical cores running two threads each for a total of four, and a TDP of 15 Watts. Alongside it, 8GB of RAM at $1866MHz$ and $256GB$ of PCIe SSD.

Since on-board processing is one of the main targets of this approach, it is interesting that all algorithms work in real time. The throughput of the most popular sensors has been studied, and the threshold for real-time has been set as the throughput of AVIRIS-ng (Table 5.2) given it is fast, already operational, and their images widely used for experiments. A new sensor with a launch targeted at 2020 is also included [63] for reference in what comes in the future throughput-wise.

Different images of different sizes have been used to test the algorithms. Their performance depends on the image characteristics, so using multiple of them serves as a way of smoothing out these points where performance might be under or over stated. The image characteristics are seen in Table 5.3 and a preview in Figure 5.1.

Model	Logic Cells	Slices	RAM blocks	RAM(Kb)	DSP	I/O
XQR4VSX55 [223]	55296	24576	320	5760	512	640
XQR5VFX130T [222]	81920	20480	596	10728	320	836
XC7VC690T [220]	693120	108300	2940	52920	3600	1000

Table 5.1: Different FPGA models used for synthesis and testing. The first two models are radiation-hardened.

Sensor	N_Z	N_X	Scan rate	S. period	Throughput	Bits	Throughput
AVIRIS [107]	224	677	12 Hz	549.52ns	1819776S/s	12	2729664B/s
AVIRIS-ng [108]	480	640	100 Hz	32.552ns	30720000S/s	14	53760000B/s
EnMAP [63]	228	1000	230 Hz	19.069ns	52440000S/s	14	91770000B/s

Table 5.2: Throughputs of the AVIRIS, AVIRIS-ng and EnMAP sensors. S/s is samples per second. Note that the reported scan rates are the maximum, and the sensors can also work in lower throughput modes.

Image	N_X	N_Y	N_Z	Depth	Description
JR	512	614	224	16	Jasper Ridge natural reserve
WTC	512	614	224	16	The World Trace Center
CUP	350	350	188	16	Cuprite valley in Nevada
SUW	320	1200	360	16	Lower Suwannee natural reserve
DHO	320	1260	360	16	Deepwater Horizon oil spill
BEL	320	600	360	16	Crop fields in Belstville
REN	320	600	356	16	Urban and rural mixed area
CRW	614	512	224	16	Cuprite valley full image
HAW	614	512	224	12	Aerial view of Hawaii
MAI	680	512	224	12	Image from Maine, USA
YELXXC	680	512	224	16	Images from Yellowstone National Park

Table 5.3: Images used for testing.

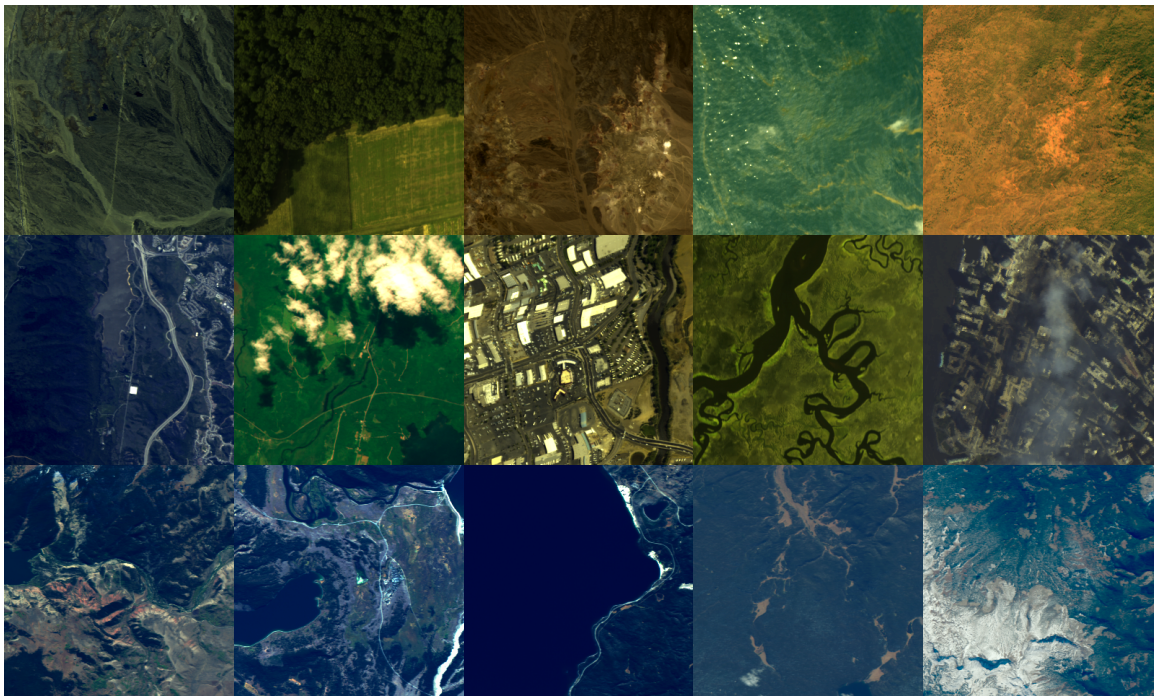

Figure 5.1: Small cutouts of the images from Table 5.3. In reading order: CRW, BEL, CUP, DHO, HAW, JR, MAI, REN, SUW, WTC, YEL00C, YEL03C, YEL10C, YEL11C, YEL18C.

5.1 Summary

Different approaches have been used to study the algorithms. Consequently, different kinds of results and data are available. This small section aims to provide a summary of all the experiments carried out.

For CCSDS, the analyses focus on how the different parameters impact the characteristics of the compression module. Compression ratio, occupancy, power draw and timing are analyzed, looking to find the best compromise between parameter selection and those four metrics.

For Jypec, a dual approach is used. First the software is analyzed in great detail, fine-tuning and optimizing for execution time, compression ratio and quality. To that end, distortion-ratio curves are analyzed as a function of different parameters. The effect of these parameters is profiled, and a multi-step optimization procedure carried out in which default settings are provided for all parameters. A set of configurations is obtained that optimize distortion-ratio performace at all the different possible levels, producing from high-quality moderately-sized images, to low-quality aggressively-compressed images.

Following the first round of analyses, profiling is done of a moderately aggressive configuration (that produces decent-quality highly-compressed images), and the most time-consuming part identified and accelerated. The throughputs are analyzed according to bitdepth (the only configurable parameter in this case). The final accelerated times are compared with the original ones, to see if FPGAs make a significant difference, by looking at the speedups obtained.

Regarding the LCPLC core, both the hardware aspect (occupancy and speed) are analyzed, as well as the algorithmic performance (compression and quality). Also, an in-depth analysis of the pipeline is performed to better understand the benefits of this approach.

The idea with the results that are obtained is to be able to put them into perspective with the literature (by comparing, in their respective fields, against the characteristics that are more often analyzed) as well as being able to compare the implementations of this thesis with each other. To that end, Chapter 6 offers conclusions as to which algorithm is better suited for different situations by comparing the different metrics obtained in this chapter.

5.2 Methodology

The results in the following sections correspond to both software simulations, hardware simulations, and on-board testing. The general procedure followed was to first test the compression/decompression cycle in software, then replace the *compression* part of the cycle for a hardware simulation, and then verify it with on-board testing. Decompression has been done in software.

All of the results for resource use and clock frequency given are for synthesis estimates from either Xilinx ISE design suite 14.7 [221] for the Virtex-4 and 5 boards, or from Vivado design suite 2018.3 [226] for the Virtex-7 board. These estimates are usually conservative, and improve with implementation, so they are representative within a reasonable margin that errs on the positive side of performance.

Power draw reports are post-implementation for the Virtex-4 and 5 boards, obtained with the Xpower tool [233] included in [221], and are pre-implementation for the Virtex-7 obtained with [226]. These are approximate results based on the maximum clock frequency attainable by the design, and a default estimate of gate toggling and memory accesses, and should only be looked at as broad approximations.

For on-board testing, the implementations from Chapter 4 were tested in order to verify correctness at their reported speeds. Given the unavailability of a hyperspectral sensor, two approaches were used:

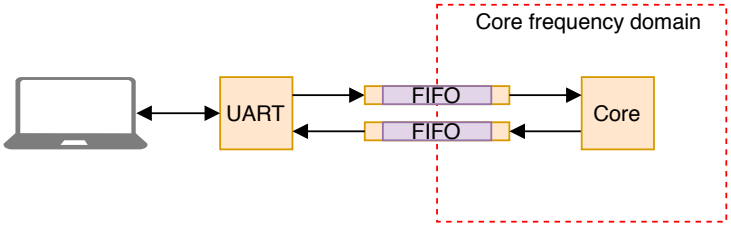

Figure 5.2: Testing setup for the first two algorithms. A UART link was set up to send/receive data to/from the FPGA, which performed the processing.

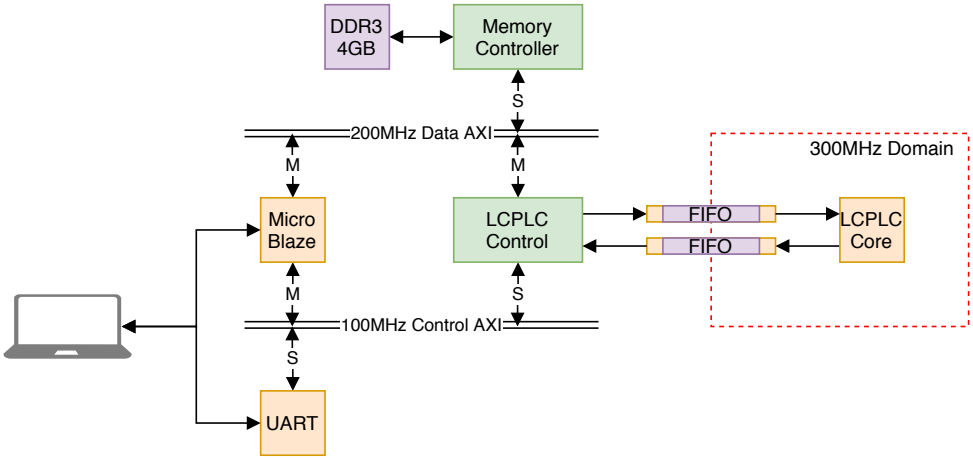

Figure 5.3: Testing setup for LCPLC. USB is used for data transmission, UART is used for control and status reports.

For CCSDS and JYPEC, their respective cores work on streams of data. In CCSDS, this stream comes directly from the sensor, and for JYPEC, it is the result of the wavelet transform after dimensionality reduction and bit shaving. In any case, the hardware that provides the raw stream was not present. A UART link was set up with the board to transmit the information. Even though the information flow and link are slow, the core run at the reported frequencies producing correct results.

A diagram of the process is shown in Figure 5.2. A RS232 link is set up with the FPGA through which data is streamed into the FPGA and received at the computer end. Tests lasted for a few hours given the slow speeds (approximately $230kb/s$ of the UART link). At the end, the data received from the FPGA was matched against the reference software result, errors did not usually arise at this stage and were caught before in simulation runs of small image sections.

For LCPLC, a different approach was used. Since the image had to be in memory for block access, a memory controller was developed, and the algorithm worked over data stored in RAM, not coming from a link such as UART. The images were loaded via USB in the FPGA board's RAM modules (quite faster than the previous UART), and once loaded the algorithm was run. Different clock domains were used for memory, status registers and control, and the core itself as seen in Figure 5.3. A MicroBlaze processor was used to control DMA access as well as the LCPLC core via the control AXI bus. It run a small script that started data transfer to and from the LCPLC core, configuring it according to the algorithm's parameters, and reporting back the results to the computer via the UART link.

Timing results were taken on-board to verify the algorithm was processing samples at the reported speed. The exact number of cycles taken to process each slice and block was stored by the algorithm itself, then compared with simulations. Compression results were again compared against the software golden standards to verify the implementation was working exactly as the reference.

$N_X = 512$	$N_Y = 614$	$N_Z = 224$
$D = 16$	$\Omega = 19$	$P = 3$
$v_{min} = -1$	$v_{max} = 3$	$t_{inc} = 6$
$\gamma_0 = 1$	$\gamma^* = 6$	$u_{max} = 16$
	$k_z = 5$	

Table 5.4: Default parameters for CCSDS

Experiments were carried out physically for the Virtex-4 and 7 FPGAs, and also synthesized for the Virtex-5. Generally, errors arose only in simulation, and hardware testing was only used to verify that the algorithm actually worked after having successful virtual tests.

The following sections show the results for each of the algorithms studied in this work.

5.3 CCSDS 123.0-B-1

CCSDS 123.0-B-1 is an algorithm designed for targeting a wide range of possible configurations. That is why a lot of parameters are tunable to the specific characteristics of the sensor, image, compression platform, etc. The parameters from Table 5.4 will be used by default, unless otherwise stated, in the results going forward.

Full prediction mode, as well as neighbor oriented sums are also assumed by default. Results are presented for the Virtex-7 FPGA.

At first, the effect of compression parameters P and Ω is analyzed in Figure 5.4.

Higher values of P and Ω increase the compression. P is special in the sense that, while the increase in performance is evident at first, it degrades later, worsening the ratio. Regarding resources, the effect of P can be seen in Figure 5.5.

Higher P values require more LUTs (given the operations that take place mainly in the weight update), but the memory resources skyrocket for BSQ traversal mode, while they remain stable for the frame-based traversal modes. The fact that P is ideal at a value of 3, along with sensors usually outputting in frame-based modes, makes BSQ impractical for real use.

Regarding speed and power consumption, Figure 5.6 is revealing: power grows in BSQ mode to enable all the BRAM modules, while clock cycle is similar for all three traversal modes, even for high values of P. This further supports the idea of $P = 3$ avoiding BSQ mode.

Ω, on the other hand, doesn't have as much impact in resources as P. Critical path and power consumption stayed relatively constant even changing it, and LUT usage went up linearly from

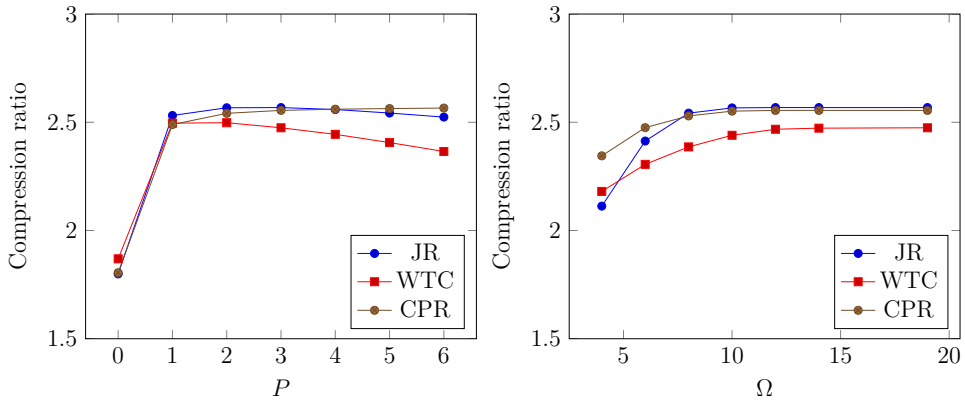

Figure 5.4: Compression ratio as a function of P and Ω. Scan order is irrelevant in this case.

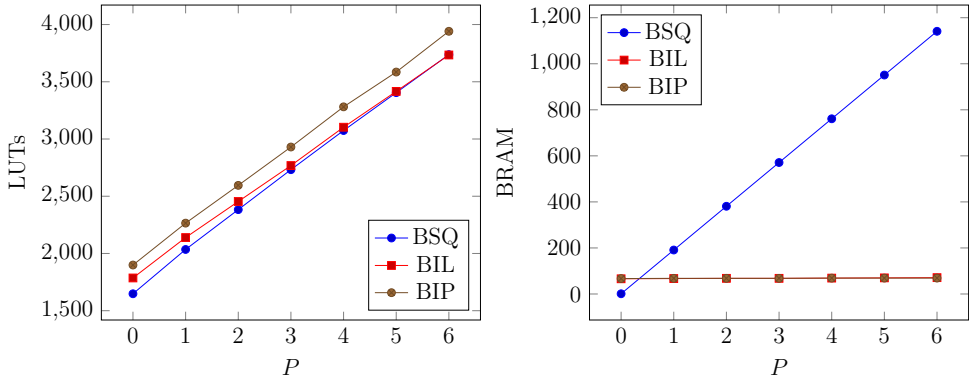

Figure 5.5: Resource use as a function of P and scan order.

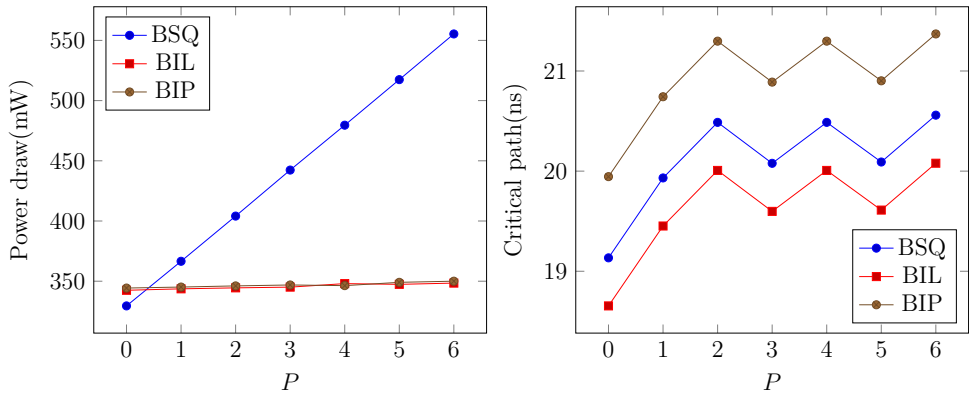

Figure 5.6: Power consumption and critical path timing as a function of P and scan order.

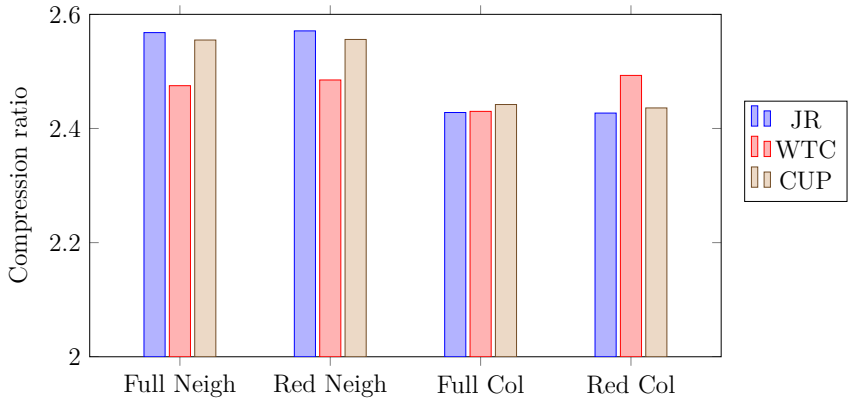

Figure 5.7: Compression of different images in different modes. Full (full) and reduced (red) prediction modes, and neighbor-oriented (neigh) and column oriented (col) modes.

2000 LUTs at $\Omega = 5$ to 2700 at $\Omega = 19$. The increase in compression ratio though is enough to justify using the maximum value.

Next, the different prediction modes are studied. Figure 5.7 shows how neighbor-oriented sums provide a good way of improving compression ratio, yet full prediction mode is not as powerful, sometimes even yielding poorer results.

When it comes to resources, Figure 5.8 shows a clear increase under full prediction mode, which coincidentally is the mode that doesn't provide much gain when it comes to compression ratio. However, neighbor oriented sums do not increase resources as much yet provide approximately a 4% gain in compression ratio. The ideal combination is thus reduced neighbor, while full neighbor can be enabled for slightly higher compression ratio if resources are available.

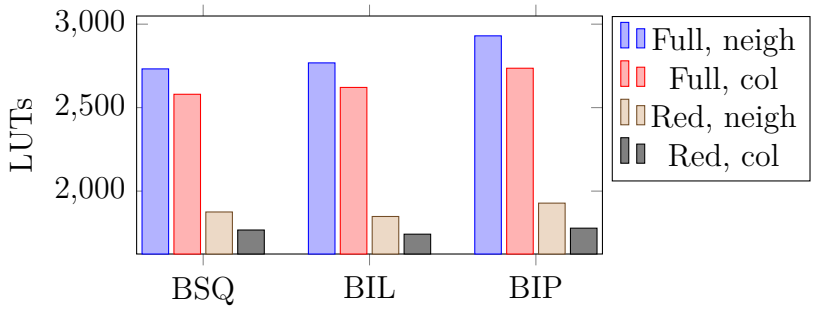

Figure 5.8: LUTs used for the different prediction modes under different traversal modes.

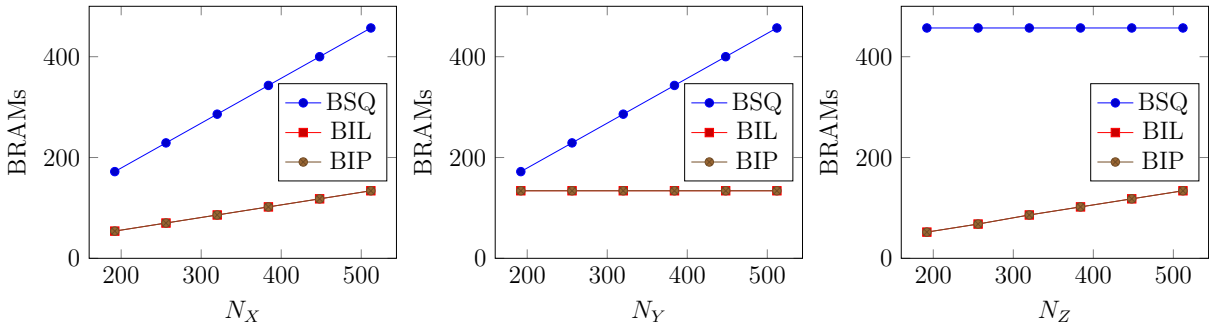

Figure 5.9: Number of BRAM used depending on varying image dimensions. Base size for these graphs is $512 \times 512 \times 512$, and the specified coordinate is varied.

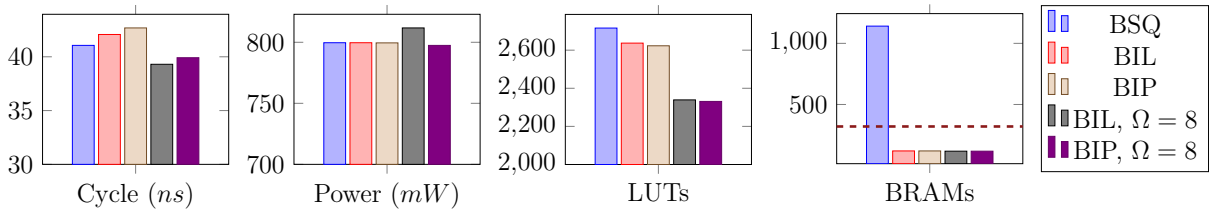

Figure 5.10: Resources used by the Virtex-4 FPGA. The threshold for BRAM is exceeded for BSQ mode.

As for segmentation, a $4ns$ gain from 20 to $16ns$ in the critical path is achieved by pipelining both sides of the prediction using the stages from Figure 4.1. The total cost for the pipeline is approximately 250 registers, low enough ($< 10\%$ increase in LUTs) to be beneficial to include considering the 20% increase in performance.

The amount of memory, which is the most limiting factor, is shown in Figure 5.9. BSQ is the ordering requiring more memory resources, since P full bands need to be stored for differences. Both BIP and BIL need the exact same amount of resources, which grows with the frame size $N_X \times N_Z$. Ideally, the preferred method is either BIL or BIP, with images that are long in the N_Y axis.

5.3.1 Space-grade FPGA

It was of interest to bring the compression core to a spaceborne system. In this case, the Virtex-4 FPGA is targeted, achieving the results shown in Figure 5.10. Critical path length doubles due to the Virtex-4 having older technology, as well as power consumption. BRAM usage also goes up (approximately 120 in all cases except BSQ), but LUTs stay at the same count.

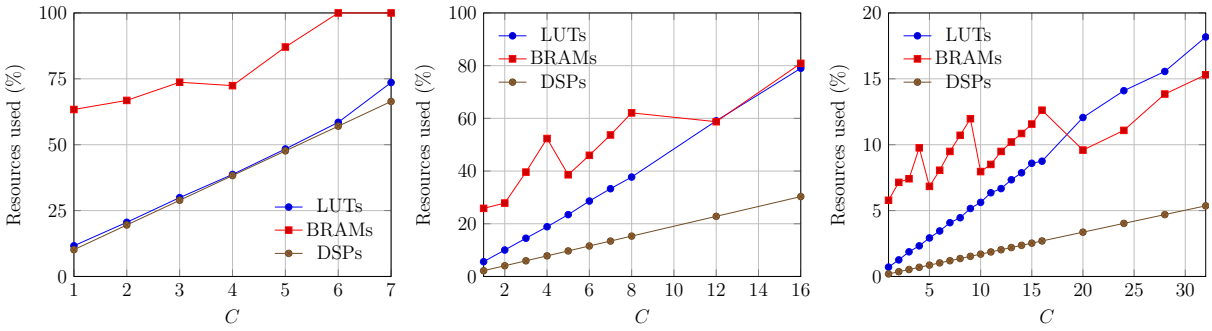

Figure 5.11: Parallel CCSDS occupancy results for the Virtex-4, 5 and 7 respectively. Note the scale of the Virtex-7 graph maxes out at 20 and not 100.

5.3.2 Parallelization

Results almost reach real-time in the Virtex 4, but do not quite match the speed (Table 5.2) of the AVIRIS-ng sensor. That's why the parallelization seen in Section 4.1.3 is important. Occupancy results are given in Figure 5.11, and performance in Figure 5.12.

The Virtex-4 maxes out at $C = 7$, while the 5 is still at 80% occupancy at $C = 16$, and the 7 is below 20% at $C = 32$. This highlights the capabilities of each FPGA. In any case, $C > 2$ is enough for real-time compression on the Virtex 4, while with $C = 1$ it already reaches real-time in both the Virtex 5 and 7, so anything higher than that is only useful when compressing a pre-recorded library of images.

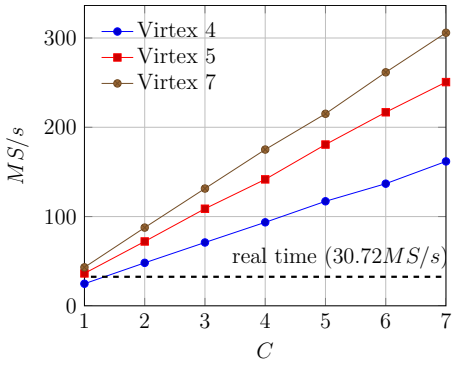

Figure 5.12: Throughput for different C values. CCSDS parallel implementation.

It is worth noting that the speed reported is that of only the core, assuming that it can receive C samples per cycle and emit C compressed samples per cycle. However, due to the serial nature of the I/O, a bottleneck arises at those input and output points. In this case, the limiting factor is at the output former, with $116MS/s$ on the Virtex-4, $179.7MS/s$ on the Virtex-5, and $219.5MS/s$ on the Virtex-7, which gives us a maximum practical C value of $C = 5$ on all platforms (when the bottleneck of the serializer arises).

5.4 JYPEC

5.4.1 Software analysis

JYPEC is an algorithm with many options (Section 4.2.1.4) that influence the output quality and ratio. As such, the effect of the different parameters will be studied in order to determine the selection that yields the best distortion-ratio curve. But first, let's see how it fares with what will be a good selection of parameters in Table 5.5.

One of the first things that pops out is the inconsistency of the results. Using the same parameters, compression ratios go from 120 to almost 2000, while SNR values vary between $20dB$ and $30dB$. Also, the image that has the worst ratio has also the worst SNR, while others with higher ratios are also compressed more.

Lossless algorithms already are unpredictable when it comes to compression ratio, depending on the specific image being compressed for the ratio itself. Lossy algorithms such as JYPEC introduce this variance also in the quality metric. So, while a general sense of the expected ratio

Image	Compression time	Compression ratio	SNR	SSIM
SUW	7.581s	486.84	30.04	0.99984
DHO	7.513s	1914.37	25.39	0.99968
BEL	7.654s	348.88	30.85	0.99966
REN	7.391s	399.37	27.28	0.99910
CRW	9.209s	121.25	21.28	0.99611
CUP	4.333s	185.81	23.19	0.99921

Table 5.5: Results for the compression of different images under three wavelet passes, ten bits of quantization, and a PCA of four dimensions trained on 1% of the total samples.

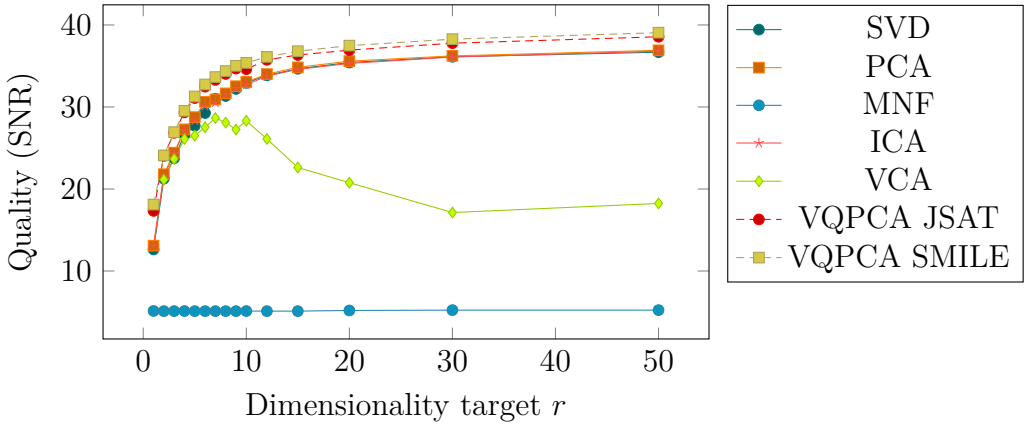

Figure 5.13: Distortion-ratio results for the different algorithms (REN image). 5 clusters were used for VQPCA, with $t = 1$.

and quality is obtained given a set of parameters, exact values are not possible to obtain, which is something to take into account when using any kind of lossy algorithm.

5.4.1.1 Dimensionality reduction algorithms

The first thing to study is how the dimensionality reduction algorithm affects the results. Figure 5.13 shows the curves for all dimensionality reduction algorithms, including two different libraries for vector quantization.

Two things are noteworthy in this graph. First, MNF falls way behind the others. This is due to not having the noise matrix available. Normally, it works by using the original data matrix and a noise matrix, which in this case is estimated from the data matrix. This estimation is done by assuming a uniform noise distribution across the image. However, for hyperspectral images, noise is often present as anomalies that happen across the width or length of the image differently at different bands. This creates extreme noise estimations in places where a sensor error might have occurred, giving them an importance that dims the rest of the image comparatively. Thus, it is not a good algorithm for this application, at least without prior knowledge of the sensor's noise.

Another interesting effect is how VCA fails to properly reduce dimensionality after $r = 7$. VCA works best when the target dimensionality is equal to the actual dimensionality of the embedded subspace. In this case, that value is probably between the $6 - 9$ range, thus lowering the performance after that point.

The algorithms with consistent curves are the best, following the same logarithmic curve. Both versions of the vector quantization are above the rest in quality, with PCA being the best among linear reduction algorithms.

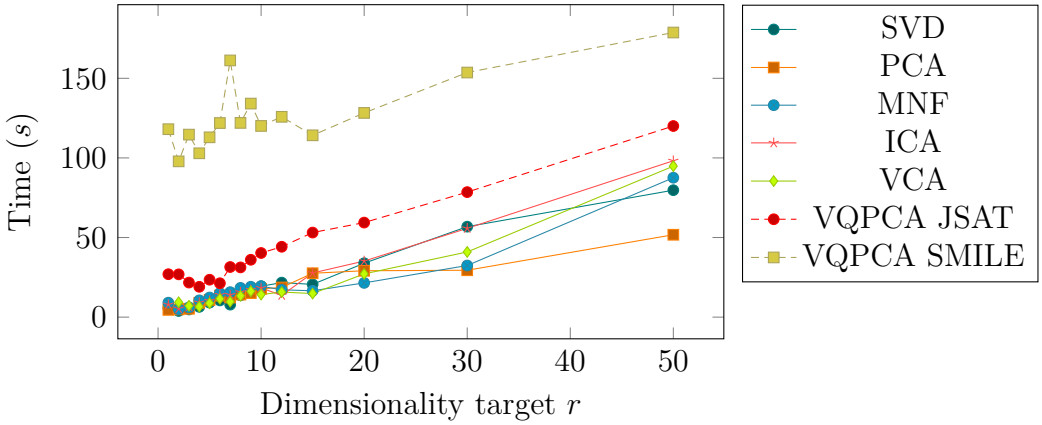

Figure 5.14: Time taken to compress for the different algorithms. 5 clusters were used for VQPCA, with $t = 1$. (REN image)

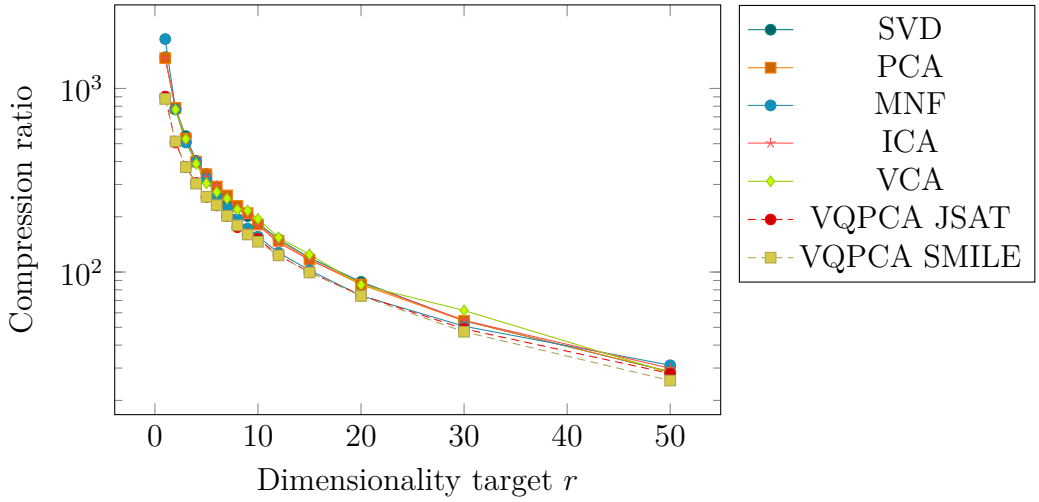

Figure 5.15: Compression ratio obtained by applying the different dimensionality reduction algorithms for JYPEC. 5 clusters were used for VQPCA, with $t = 1$. (REN image)

Time-wise, Figure 5.14 shows how the most costly algorithms are those using vector quantization, while PCA is, by far, the fastest. This fact, along with those three being the best in quality, will imply a trade-off between speed and quality.

Lastly, looking at the compression ratio in Figure 5.15, the least compression comes from the vector quantization algorithms, while the others are compressing much more (notice the logarithmic y axis). This is because the additional complexity requires additional resources to save all the values needed for decompression.

With the more complex VQPCA versions having greater quality but lower compression, the true performance is seen in Figure 5.16 with the distortion-ratio curves.

The linear algorithms SVD, PCA and ICA go hand in hand, while the nonlinear VQPCA-based algorithms also present the same curves. While at low compression, nonlinear algorithms present higher quality, at high compression, the linear dimensionality reduction algorithms outperform the former, as the need to store additional information is lower. With PCA being the best linear reductor, a direct comparison with VQPCA is due, based on the number of clusters. Figure 5.17 shows this comparison, realizing that PCA is just VQPCA with one cluster.

With more clusters, the SNR is higher at low compression. With less clusters, quality is higher at high compression. So, ideally, at each moment, the number of clusters should be the one that maximizes the distortion-ratio curve. This will be optimized in Section 5.4.1.6.

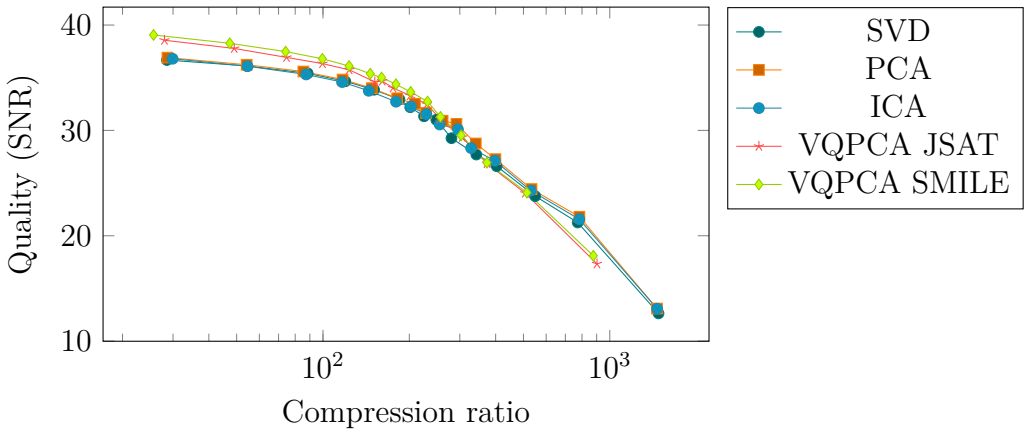

Figure 5.16: Distortion-ratio curves obtained by applying the different dimensionality reduction algorithms for JYPEC. 5 clusters were used for VQPCA, with $t = 1$. (REN image)

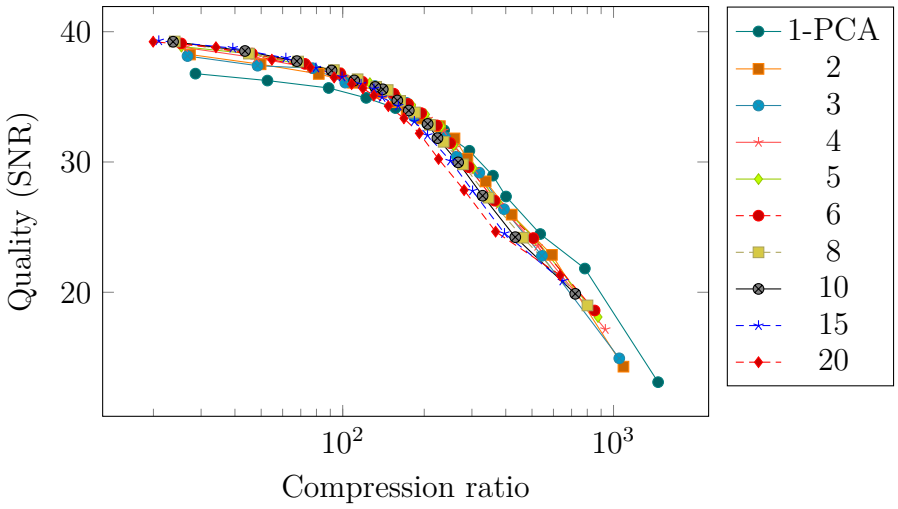

Figure 5.17: Distortion-ratio curves obtained by using VQPCA with different number of clusters. $t = 1$. (REN image)

5.4.1.2 Training sub-sampling

The training process can be sped up by sub-sampling the training samples using the parameter t, which indicates the fraction of input pixels used for training the dimensionality reduction algorithm. Table 5.6 clearly shows that higher values of t imply longer compression times, in that case by a factor of up to 7 times longer.

This is due to images having a great deal of redundancy. Pixels that are close together are very similar, and having more of them doesn't change how the algorithm behaves after training. A good compromise is found at $t = 0.01$ where quality is very slightly affected, but time has been reduced almost to the lower limit.

t	0.001	0.005	0.01	0.03	0.05	0.1	0.3	0.5	0.75	1
Ratio	353.6	350.1	348.9	343.8	349.2	350.8	345.3	344.4	344.7	344.6
SNR	30.67	30.78	30.84	30.88	30.85	30.86	30.88	30.88	30.89	30.89
Time (s)	5.402	5.673	5.918	6.353	6.891	8.338	14.22	20.08	27.62	37.63

Table 5.6: Results for the compression of the BEL image using different values for t. Time is the only significant parameter changing, while keeping a stable distortion-ratio performance.

b		2	3	4	5	6	8	10	12	16
SNR	BEL	6.333	9.560	12.33	15.59	19.80	28.03	30.84	31.00	31.01
	SUW	-0.29	4.304	10.35	14.15	18.08	25.93	30.03	30.39	30.41
	CUP	-3.87	-0.77	6.881	11.20	13.39	19.82	23.19	23.45	23.46
Ratio	BEL	11458	11087	10191	7168	3472	796.0	348.8	225.1	135.2
	SUW	21371	20109	18642	13077	6268	1361	486.8	276.9	150.8
	CUP	3363	3314	3200	2975	2237	462.8	185.8	118.7	71.05
Time	BEL	03.26	02.53	02.62	02.78	02.80	03.82	05.52	07.16	10.19
	SUW	05.75	04.72	05.38	05.54	05.67	07.54	09.58	12.27	18.37
	CUP	01.43	01.14	00.91	00.96	01.17	01.62	02.84	03.87	05.83

Table 5.7: Results of compressing various images with varying values of bit depth b used for quantization after the dimensionality reduction.

5.4.1.3 Quantization bit depth

The effect of the quantization bit depth b is also of interest, since it is the resolution of each sample that goes into the JPEG2000 coder and directly affects compression ratio. Figure 5.18 shows that, for values under $b = 4$, the noise level in the reconstructed image is sometimes even higher than the signal level ($SNR < 0$). In any case, quality is not acceptable until the $b = 8$ levels where the SNR reaches $20dB$.

Above 10 bits, no improvement is seen, suggesting that anything above that is only going to increase size with no positive impact in image quality. Also, compression time increases with b as seen in Table 5.7, further supporting the idea that the maximum values are not the ideal ones.

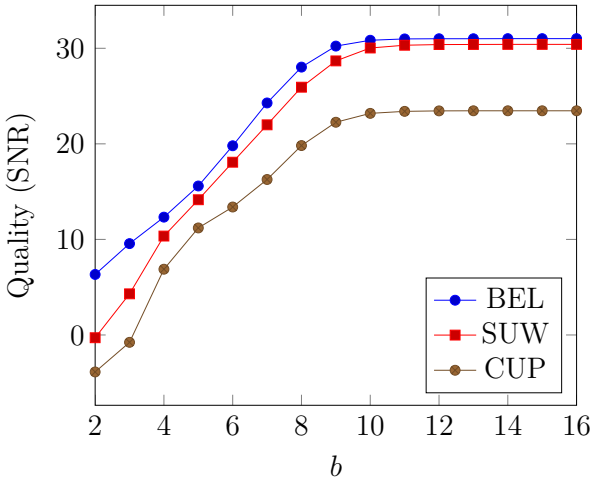

Figure 5.18: Image quality with respect to b

5.4.1.4 Target dimension

When applying the dimensionality reduction algorithms, a target dimension r is set. This target directly impacts time, compression ratio and SNR, since it will take longer to process the image with more dimensions in the reduced space, increasing quality and worsening the ratio. These results can be observed in Table 5.8.

Quality starts being acceptable above levels of $r > 4$, and increases significantly until $r = 15$, after which improvement is made slowly, compared to the worse ratio and longer compression time.

5.4.1.5 Optimizing bit-depth and dimensionality

b and r are the two parameters that affect compression ratio and quality the most. But varying one without varying the other can mean a much worse result than finding the optimal point where to set both to achieve a certain distortion-ratio performance. To do so, images SUW, DHO, BEL, REN, CPR and CUP were all compressed by varying b and r across their range,

r		1	2	3	4	5	8	15	30	100	200
SNR	BEL	16.54	24.25	28.72	30.84	33.15	36.96	39.36	40.73	43.52	44.10
	SUW	18.51	20.91	27.95	30.03	31.19	33.90	36.45	37.63	39.38	39.99
	CUP	14.13	16.65	21.45	23.19	25.24	28.53	31.04	33.02	34.66	n/a
Ratio	BEL	1157	684.8	464.8	348.8	272.7	162.4	79.60	35.97	9.733	5.100
	SUW	1616	858.9	666.3	486.8	336.8	199.4	84.85	39.44	10.89	5.542
	CUP	737.3	373.1	255.4	185.8	130.2	83.25	41.06	18.82	5.032	n/a
Time	BEL	04.14	03.98	04.61	05.80	07.12	11.64	17.40	36.77	2'08	4'09
	SUW	06.38	07.35	13.46	09.82	12.10	16.90	33.47	1'12	4'04	7'32
	CUP	02.71	02.02	02.45	02.99	03.64	05.22	10.06	20.98	23.30	n/a

Table 5.8: Results in the compression of multiple images when changing the value r of target dimensions.

$b\backslash r$	1	2	3	4	6	8	10	12	14	16
16	16.55	24.28	28.84	31.01	35.04	37.89	39.35	40.16	40.74	41.26
15	16.55	24.28	28.84	31.01	35.04	37.89	39.35	40.16	40.74	41.26
14	16.55	24.28	28.84	31.01	35.04	37.89	39.35	40.16	40.73	41.25
13	16.55	24.28	28.84	31	35.04	37.88	39.34	40.14	40.71	41.23
12	16.54	24.28	28.84	31	35.01	37.84	39.29	40.07	40.64	41.15
11	16.54	24.28	28.82	30.98	34.92	37.65	39.06	39.86	40.31	40.79
10	16.54	24.25	28.72	30.84	34.5	36.96	38.09	38.79	39.19	39.52
9	16.53	24.13	28.28	30.23	33.18	34.67	35.33	35.67	35.85	36.01
8	16.46	23.65	26.92	28.03	29.62	30.25	30.46	30.56	30.61	30.65
7	16.17	22.03	23.73	24.28	24.75	24.89	24.93	24.95	24.96	24.97
6	15.46	19.14	19.6	19.8	19.93	19.99	20	20.01	20.01	20.01
5	13.61	15.57	15.53	15.59	15.61	15.63	15.63	15.63	15.63	15.63
4	11.59	12.35	12.32	12.33	12.34	12.34	12.34	12.34	12.34	12.34
3	9.69	9.59	9.58	9.56	9.56	9.53	9.52	9.52	9.52	9.52
2	8.07	6.39	6.37	6.33	6.31	6.21	6.21	6.21	6.21	6.21

Table 5.9: Results for quality (SNR) after compressing BEL with varying b and r values.

b\r	1	2	3	4	6	8	10	12	14	16
16	504.41	268.61	180.27	135.26	87.66	65.98	52.15	42.97	36.63	31.79
15	555.41	297.79	200	150.06	96.95	72.99	57.62	47.43	40.4	35.04
14	618.04	334.25	224.7	168.59	108.48	81.71	64.4	52.92	45.05	39.03
13	697.18	381.35	256.66	192.56	123.24	92.89	73.05	59.91	50.95	44.08
12	800.76	445.16	300.07	225.11	142.99	107.84	84.54	69.14	58.72	50.7
11	944.24	537.75	363.3	272.52	171.07	129.14	100.75	82.05	69.53	59.86
10	1157.8	684.8	464.84	348.88	214.78	162.47	125.74	101.68	85.88	73.59
9	1510.9	946.25	650.47	489.97	291.81	222.08	169.56	135.45	113.76	96.69
8	2170.1	1467.9	1041	796.09	450.04	349.12	261.21	204.46	170.31	142.74
7	3546.5	2610.4	1969.7	1562.5	835.95	676.57	498.6	380.78	315.71	259.06
6	6601.7	5146.8	4220.1	3472.2	1960.4	1655.1	1247.8	964.17	814.14	662.3
5	12899	10040	8650.2	7168.2	5284.8	4409	3457.3	2924.1	2583.49	2245.9
4	17803	14120	12033	10191	8027.8	6627.9	5576.4	4810	4252.75	3816
3	18104	15169	12848	11087	8746	7173.8	6124.9	5336.8	4738.79	4258.7
2	19383	15709	13251	11458	9018.7	7435.4	6325	5503.1	4870.34	4368

Table 5.10: Results for compression ratio after compressing BEL with varying b and r values.

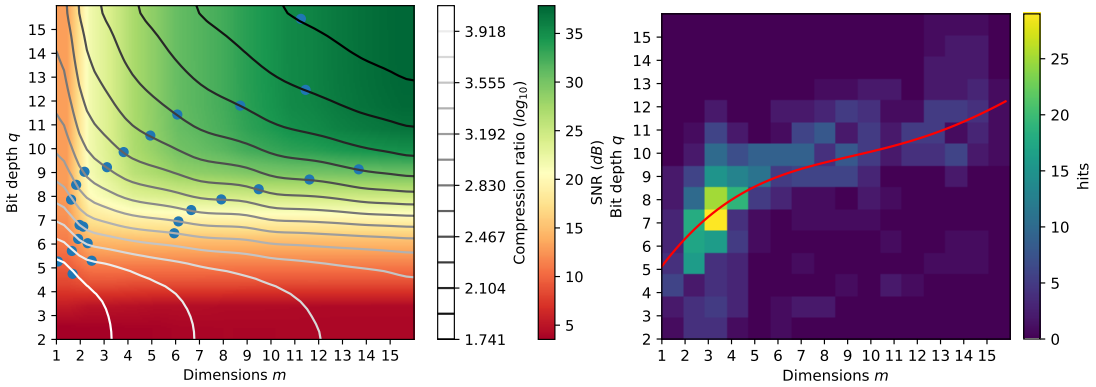

Figure 5.19: Optimization process for parameters b and r. On the left, the points where ratio is maximal given a specific SNR for the REN image. On the right, the aggregate of all those points, along with a fitted polynomial showing the general trend.

obtaining tables such as Tables 5.9 and 5.10. (Only the ones for BEL are shown for typesetting purposes).

It becomes clear that r has no impact on quality for very low values of b, however the ratio does change. This suggests that, for low qualities, it is preferable to lower r instead of b. Different conclusions can be drawn at different distortion-ratio performances, but to get the whole picture, a different approach is taken.

For each image, these tables are obtained, and interpolated to get heat-maps. Contour lines from the SNR image are over-imposed on the compression ratio image. On each contour, the maximum points are marked. This process is repeated for all test images, aggregating the results for them, as seen in Figure 5.19.

A trend clearly emerges, indicating that some (r, b) combinations are clearly superior to others. The polynomial fit gives us a path that, from highest to lowest compression ratio, gives the optimal configuration to get the maximum SNR at that point. Values from this regression will be referred as QR value, which gives the optimal values for b and r for a given index $j > 0$:

$$QR(j) = \{b(j), r(j)\} = \left\{ \begin{array}{ll} 5 + j & j \leq 3 \\ 9 + \left\lfloor \frac{j-4}{3} \right\rfloor & j > 3 \end{array} , \; j + 1 \right\} \quad (5.1)$$

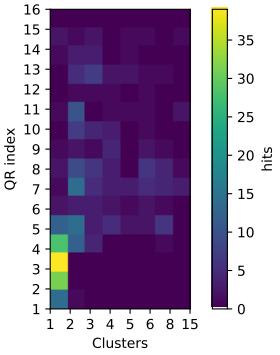

Figure 5.20: Optimization process for the number of clusters and $QR(j)$ index.

5.4.1.6 Optimizing the number of clusters

The exact same optimization process was repeated by matching the QR index against the number of clusters for VQPCA, obtaining the result given in Figure 5.20. The result is not as clear as with the (b, r) optimization, but still gives enough information to create a QRC index including also the number of clusters c to use to optimize distortion-ratio performance:

$$QRC(j) = \{QR(j), c(j)\} = \{QR(j), \lfloor j/6 \rfloor\} \quad (5.2)$$

5.4.1.7 Optimizing the number of wavelet passes

How many times the wavelet is recursively applied has an obvious impact in both speed (more passes means slower compression/decompression) and distortion-ratio quality. More passes will mean more optimal energy concentration, but also the range of the transformed samples doubles every pass, potentially overflowing the maximum quantization range in the latter stages,

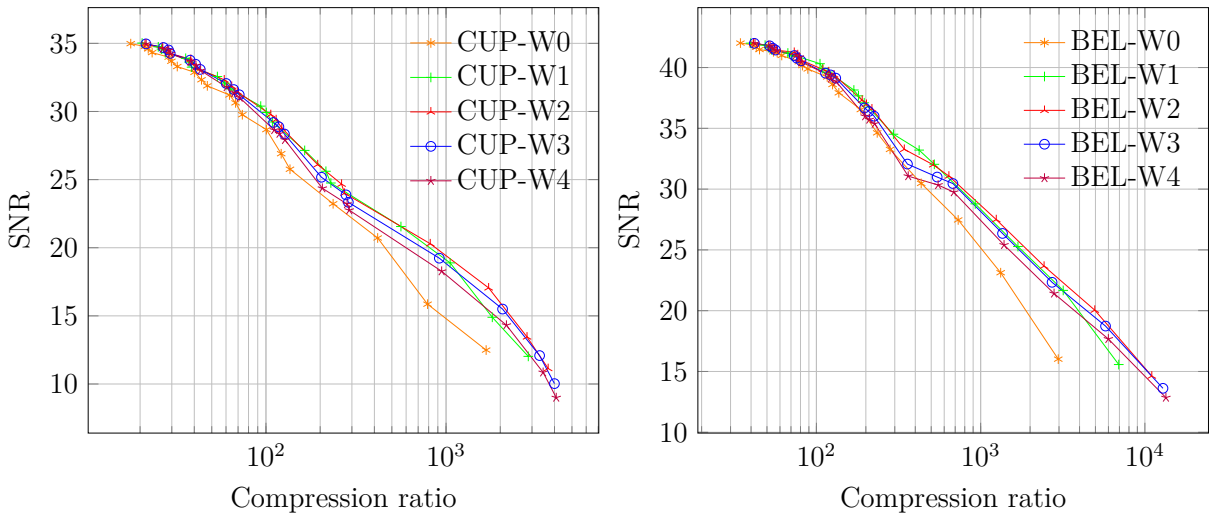

Figure 5.21: Comparison of the distortion-ratio curves when compressing the CUP and BEL images. Behavior was consistent across the whole image test set. Wx indicates x wavelet passes were performed.

worsening the ratio again. The general trend of changing the number of wavelet passes w is seen in Figure 5.21.

As clearly seen, doing no wavelet passes worsens the quality. Doing one wavelet pass works well at low qualities, but falls short when good quality is required. Between $w = 2$ and $w = 3$, results are similar. We use a value of $w = 3$, which works well [40, 157].

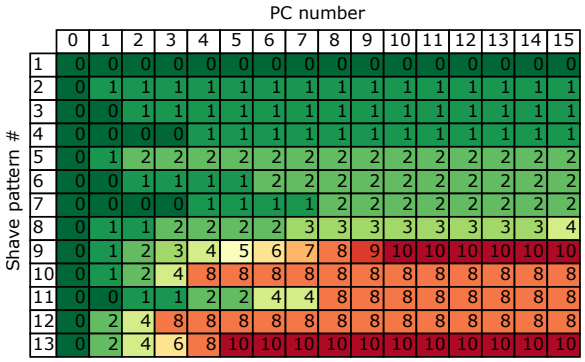

Figure 5.22: Different patterns indicating how many bits they shave on each reduced dimension.

5.4.1.8 Optimizing bit-depth

After dimensionality reduction, the same algorithm has been applied so far to every resulting band. This means that every dimension in the embedded subspace obtained after transforming is treated as if it had equal importance.

This doesn't have to be the case. In fact, the first components (specially in the case of PCA) will hold the most information, while the last components will hold the least. This suggests the possibility of varying the amount of bits used for the quantization of each reduced component. For that, multiple *shave patterns* (Figure 5.22) have been tested, which remove a certain amount of bits from each transformed component, with more bits removed as the band index grows.

To see which one is preferable, the same optimization procedure as with the QR and QRC indices has been carried out: test runs have been done on all images, aggregating the results with the best distortion-ratio performance. Results from this are seen in Figure 5.23.

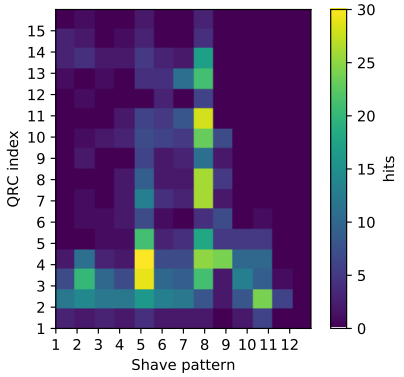

Figure 5.23: Hits indicate a point where that shave pattern optimized distortion-ratio performance for a test image.

Patterns 5 and 8 are the clear winners, not being aggressive on bit shaves, but removing enough to make a significant dent on ratio without affecting image quality. Pattern 5 shaves a

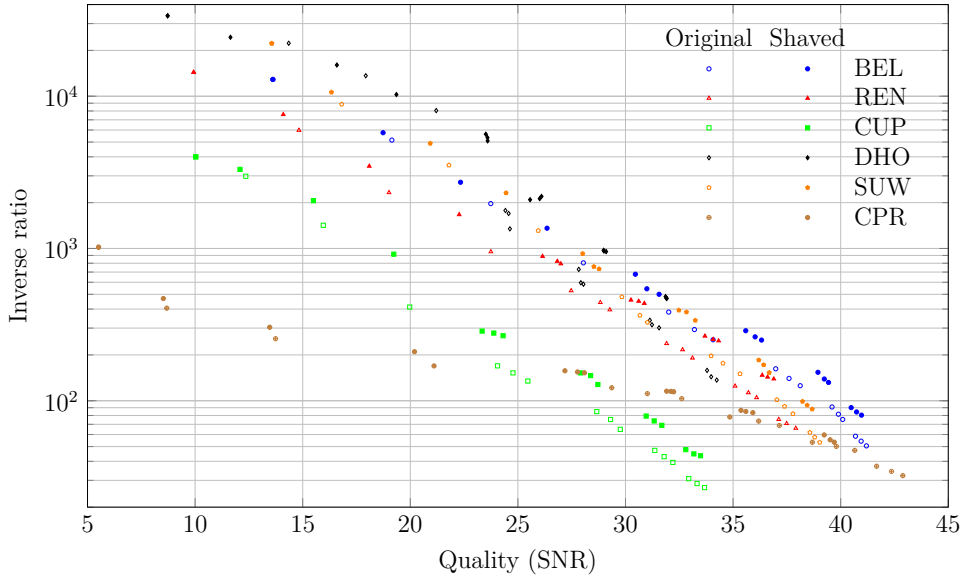

Figure 5.24: Comparison between images compressed with shave pattern 7 and no shave pattern. Filled markers correspond to the shaved version, while hollow ones to the unshaved runs.

bit of the second component, and two bits afterwads. Pattern 8 is the preferred one, halving the information every 2^i component and obtaining the best result more often. With this, the QRC index from Equation (5.2) is used to create the ideal configuration F, given by Equation (5.3):

$$F(j) = \{QRC(j), s_i(j)\} == \left\{ QRC(j), \begin{array}{cc} 0 & i = 0 \\ \lfloor \log_2(j+1) \rfloor & i > 0 \end{array} \right\} \tag{5.3}$$

When applied to the test image set, ratios improved (compared to not using shave patterns) across all image qualities (as seen in Figure 5.24). A clear improvement is seen across the image test set. Shaved (-sh) images present much higher distortion-ratio performances at an average of 1.5 higher compression ratio for the same quality.

5.4.1.9 Visualizing the results

Up to this point, everything has been theoretical, based on the synthetic quality and ratio measures. All parameters have been optimized with distortion-ratio performance in mind, creating the $F(j)$ index to obtain, given a value j, the algorithm configuration to get the best distortion ratio. Low values of j yield great compression, while high values yield high quality. A visualization is presented in Figure 5.25.

Visually, no loss can be perceived until $j = 12$, where color starts fading away. This is an indicator of diminishing spectral resolution, but the spatial resolution can still be resolved quite clearly until $j = 8$. Below that value, blurriness starts growing with the loss of spatial resolution, and spectral resolution is completely lost since the target dimensionality has fallen below the underlying subspace dimension. However, even at the extreme compression of $j = 0$, the image is still recognizable with a size over 20000 smaller than the original.

5.4.1.10 Timing after optimizations

This aggressive theoretical optimization affects performance in different ways. With each additional dimension, wavelet pass, bits used for quantization, clusters... the time it takes to process an image increases. As a lossy algorithm, the target distortion-ratio moves away from perfect reconstruction to good approximation.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
SNR	13.56	16.34	20.98	24.47	28.01	28.55	29.12	32.80	33.09	33.41	36.31	36.55	37.06	38.55	38.77	38.94	39.47	39.66	39.88	40.13
C.R.	22221	10513	4902	2339	934.6	766.6	598.8	341.3	326.4	293.2	167.5	155.7	136.9	94.68	87.60	83.67	62.99	59.97	56.92	45.24
C.T.	7.94	8.14	8.26	8.95	9.45	9.75	10.10	10.35	11.43	11.79	14.88	15.07	16.86	21.40	22.15	22.50	28.08	29.01	33.44	38.74
D.T.	1.84	1.83	2.21	2.52	3.35	5.72	6.00	6.41	6.72	6.99	8.67	9.39	10.00	12.67	13.42	13.70	16.71	17.60	22.50	24.12

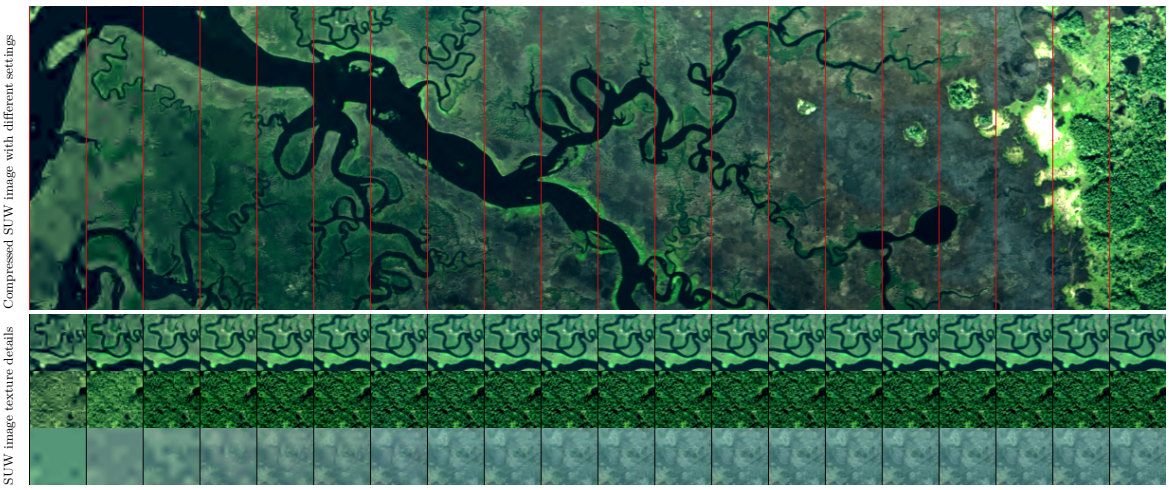

Figure 5.25: Visualization of the SUW image for values of $j = 0 \dots 19$, which control quality and ratio based on the optimizations from the previous sections. C.R is the compression ratio. C.T and D.T are the compression and decompression times respectively.

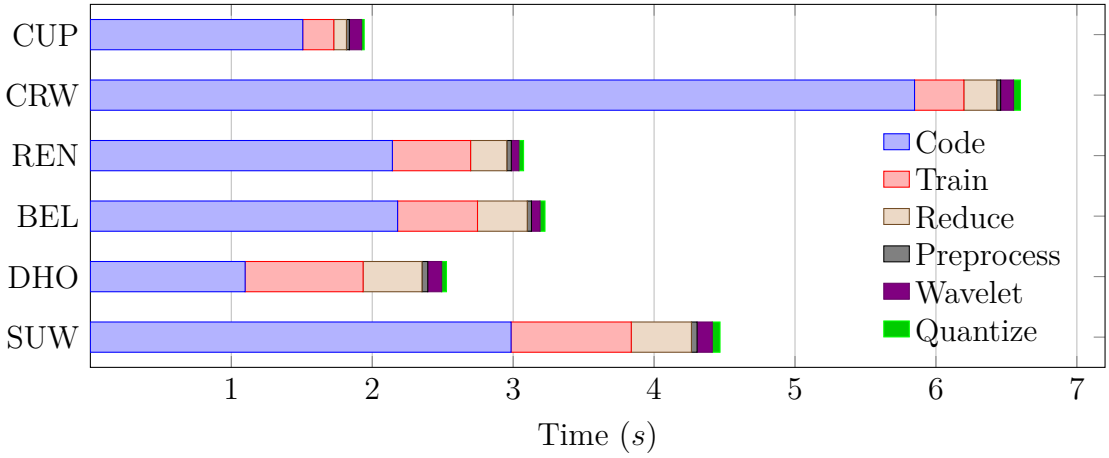

Figure 5.26: Timing results by algorithm steps for JYPEC, using three wavelet passes, 10 bits of depth and a reduction using PCA to four target dimensions training with 1% of samples.

Looking at real-time, these optimizations are too expensive to execute. For example, looking at Figure 5.25, it is seen that the minimum time is almost 8 seconds for the SUW image, which with its $138.24MS$ takes exactly $4.5s$ to traverse with the most advanced sensor AVIRIS-ng. Its old version AVIRIS takes $75.96s$ in obtaining an image of the same size, so even at high quality indices j, real time is met without acceleration for old sensors.

However, to target real-time in the fastest sensors, simplicity is key. Values in Figure 5.26 were obtained with a configuration targeted for fast compression. Quality is lowered by around $1 - 2dB$ on average (compared to the optimized configuration at a similar compression ratio), but is still of quite good quality.

A clear trend is seen across all images. The time it takes to code the image is by far dominating the rest of the algorithm. Even though it is the simplest part, the fact that each bit is processed individually, along with the optimizations made to the training process by using $t = 0.01$, make it the most costly part of the algorithm. It is important to note that this are average values of multiple runs, and the nature of a CPU can introduce slight variations from run to run. Even

	Virtex 7			Virtex 4		
Module	MHz	LUTs	BRAM	MHz	LUTs	BRAM
Tier 1 coder	255	2708	4	99.38	3803	8
BPC-core	248	731	2	127.8	1850	4
BPC-serial	390	142	0	183.4	199	0
MQ	321	1778	0	121.2	1689	1
MQ-I/II	322	1326	0	121.7	989	0
MQ-III	535	47	0	233.8	67	0
MQ-IV	331	231	0	116.4	375	0
FIFOS	927	57	2	293.4	114	2

Table 5.11: Frequency and occupancy for the constituents of the tier 1 coder. Results are for the Virtex-7 and Virtex-4 boards with a depth of 32 set for all FIFO queues.

if some already meet the real-time requirements of AVIRIS-ng, it is important to create a time margin big enough for any unexpected variations to still fit within constraints.

Accelerating the full algorithm in hardware was deemed to be too time consuming: Dimensionality reduction algorithms are complex and rely on floating point arithmetic. Wavelet transforms are easier to do, but require careful management of memory and again, floating point arithmetic. Given that the most costly part is coding, a compromise is made, following Amdahl's law, of only optimizing the coder, which is exactly what is done in Section 4.2.2.

5.4.1.11 Hardware acceleration

The full JPEG2000 tier 1 coder from Section 4.2.2 is implemented in hardware to help accelerate the most costly part of the algorithm. Implementing the full JYPEC algorithm would be too time consuming, and most likely wouldn't even fit on an FPGA. (Just a fast PCA implementation takes up almost the entire Virtex-7 board [67]). Table 5.11 shows the occupancy results for a tier 1 coder module.

Occupancy is very low (for the Virtex-7) at under 3000 LUTs with just 4 BRAM blocks used for FIFOs. These are necessary to hold the 64×64 blocks processed by the algorithm. As for speed, a more careful analysis is needed, since the different submodules of the implementation process different data. The following calculations are done for the Virtex-7:

- The BPC core processes blocks of $64 \times 64 \times 16b = 65536b$ in 44032 cycles (Section 4.2.5), so its expected throughput at the input (when operating at $255MHz$) is $380Mb/s$. This will vary if the image has blocks that are smaller in size or with a height not multiple of four. But in general this is a good approximation.
- The BPC-serial module can output up to $390M$ CxD pairs per second.
- The MQ coder, at its first two stages, can process up to $322M$ CxD pairs per second, and will generate the same amount of updates. The fuser module can keep up with it at $535M$ updates per second, and the same thing goes for the last stage that processes up to $331M$ updates per second.
- FIFOs have no problem working at the required speed.

However, in a practical setting the same clock is used for the full module. The limitation then comes from the speed of $255MHz$ achieved in this case.

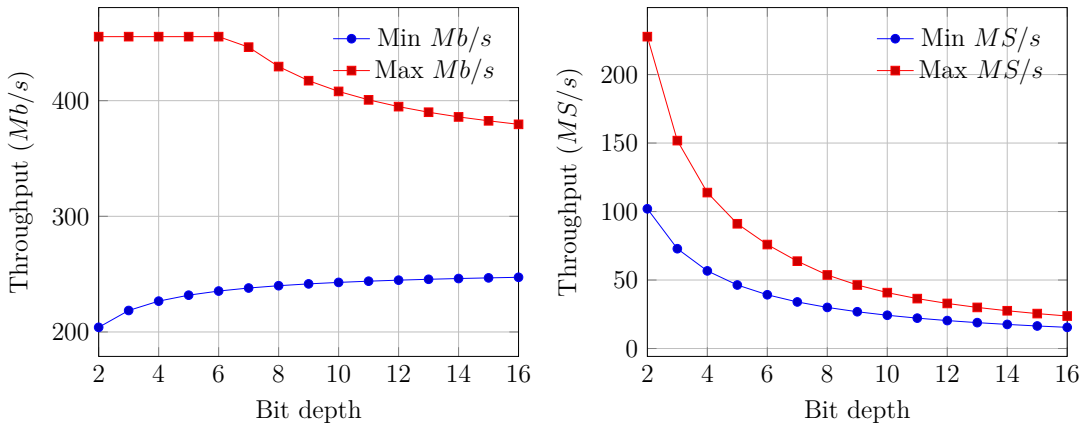

Figure 5.27: Impact of bit-depth on the throughput of the JPEG2000 tier 1 coder. (Virtex-7 FPGA)

The bottleneck is the number of CxD pairs processed by the MQ-coder ($255M$ CxD pairs per second). By studying how many of these are produced by the BPC, the throughput of the full module can be estimated:

The absolute minimum number of CxD pairs that can be generated for a full-sized block of $64 \times 64 \times 16$ happens when every bit is zero. In this case, a total of $15 \cdot 1024 = 15360$ CxD pairs are generated, for a total of 0.234 per input bit. On the other hand, the upper limit is given by a cleanup pass with run-length interruptions at every position, followed by 14 refinement passes for the remaining bit planes. The number of CxD pairs generated in this case is $1024 \cdot 10 + 4096 \cdot 14 = 67584$, or 1.03125 per input bit.

This means that the input rate to generate $255M$ CxD pairs per second ranges from $1.01Gb/s$ to $247.3Mb/s$. The practical range however is limited to the BPC core actual throughput which was $255 * 65536/44032 \approx 380Mb/s$, so the actual range only goes up to that value.

The actual input speed lies in that range, but depends on the redundancy of the data. However, studies have shown the average rate of CxD pairs per bit to be approximately 0.56 [183]. This means that, on average, to sustain $255M$ CxD pairs per second, an input stream of $455Mb/s$ is needed, far exceeding the throughput accepted by the BPC at $380Mb/s$.

Taking this into account, it is expected that the MQ-Coder will be able to keep up with the maximum input throughput at $380Mb/s$, expecting to generate around $380 \cdot 0.56 \approx 213M$ CxD pairs per second, far below the $255M$ CxD pair limit of the full tier 1 coder implementation.

With this in mind, the *expected* speed of the module is, for $16b$ samples, $380/16 = 23.75MS/s$, and the *minimum guaranteed speed* is $247.3/16 = 15.45MS/s$. At first glance, these do not seem sufficient to keep up with the real-time constraint of $30.72MS/s$. However, the tier 1 coder works on the *reduced* dimensionality data, so this will not be a problem in practice.

Repeating these calculations for all bit-depths D, yields the graphs seen in Figure 5.27. It is very interesting to see how the minimum input throughput (corresponding to worst-case scenarios) is lower with smaller bit-depths, due to more contexts being generated per bit in the worst case. The best case, however, reaches the maximum throughput of $455Mb/s$ for low bit-depths due to less passes needed at lower bit-depths (with only 1 for the first bit-plane). For high bit-counts, this value is dominated by the BPC core throughput. For both cases, however, the number of samples processed per second goes down since even when a value increases in Mb/s, it is not compensated by the bit-depth growth.

However, when looking at MS/s, the throughput actually increases, since the speed loss is, proportionally, lower than the bit difference between samples. The same results for the space qualified Virtex 4 FPGA are seen in Figure 5.28.

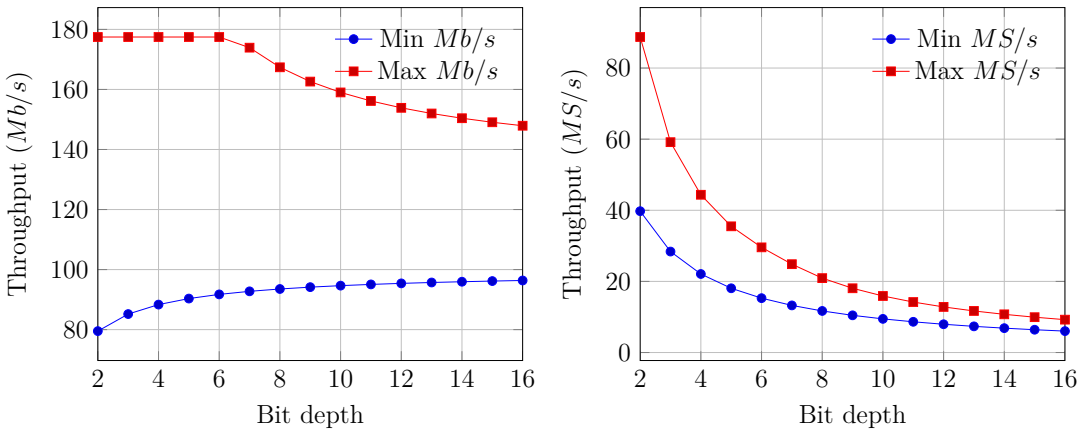

Figure 5.28: Impact of bit-depth on the throughput of the JPEG2000 tier 1 coder. (Virtex-4 FPGA)

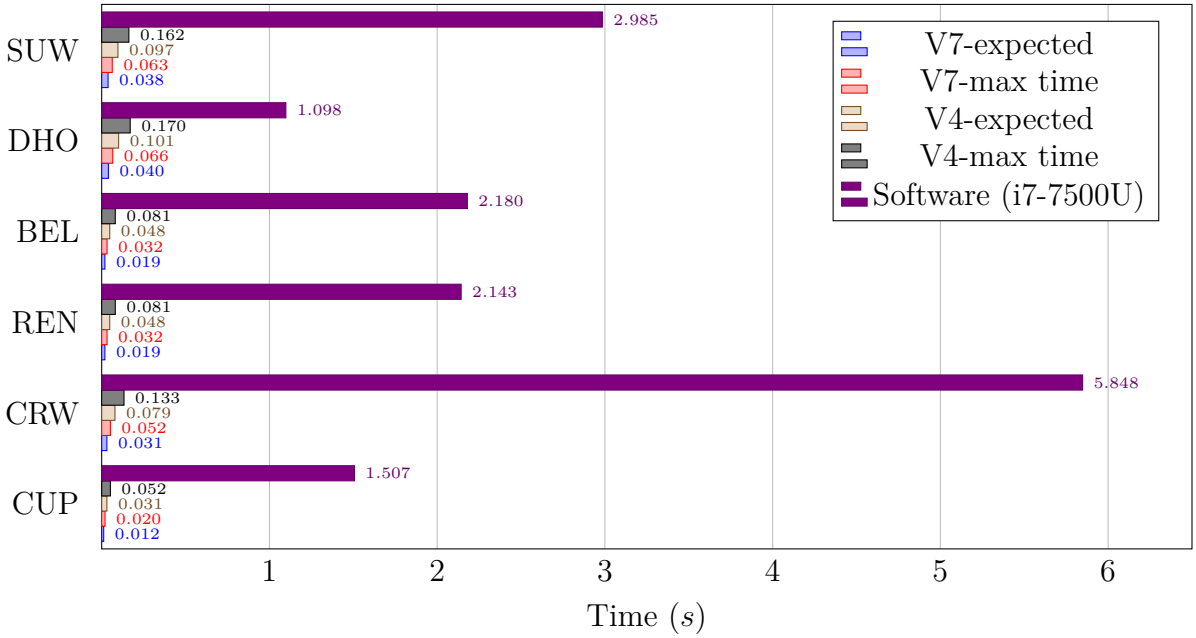

Figure 5.29: Timing results (exclusively from the *coding* part of the algorithm) when compressing images using JYPEC in both software and an FPGA accelerator.

5.4.1.12 Improvement over software

The impact of this FPGA acceleration vs its software counterpart depends of course on the speed of the FPGA module, which in turn depends on the bit-depth of the image after applying the dimensionality reduction algorithm. In this case, the bit-depth selected is 10, so that will be used going forward. For calculations, this means a speed range of $94.6 \rightarrow 159 Mb/s$ (or $9.46 \rightarrow 15.9 MS/s$) on the Virtex-4, and a range of $242.9 \rightarrow 408 Mb/s$ (or $24.3 \rightarrow 40.8 MS/s$) on the Virtex-7. Results will be marked with *minimum* and *expected* to indicate if the value corresponds to the minimum guaranteed throughput or the expected one based on the usual behavior of images.

As an example, if an AVIRIS image of size $614 \times 512 \times 224$ is compressed with target dimension 4 and target bit-depth 10. This means that a total of $10 \times 614 \times 512 \times 4 = 12574720b$ (or $1257472S$) will go through the encoder. At a guaranteed minimum speed of $242.9 Mb/s$, the core, synthesized on a Virtex-7 FPGA, does its part of the compression algorithm in just $51.77 ms$.

Comparing these results over the test set of images against the software implementation, Figure 5.29 is obtained.

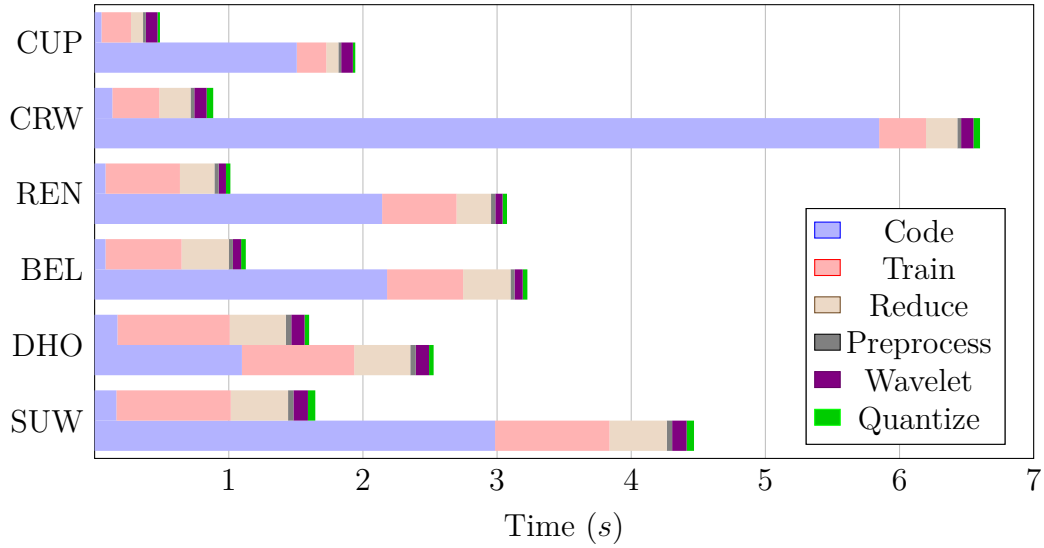

Figure 5.30: The impact of speeding up the coding part of the algorithm in an FPGA, when assuming the *minimum guaranteed* throughput on a Virtex-4.

Speedup	CUP	CRW	REN	BEL	DHO	SUW
Block coding (V4Min)	29.09×	43.99×	26.39×	26.39×	6.440×	18.38×
Block coding (V7Exp)	125.4×	189.7×	113.8×	113.8×	27.77×	79.28×
Whole process (V4Min)	3.98×	7.45×	3.03×	2.86×	1.57×	2.71×
Whole process (V7Exp)	4.33×	8.43×	3.23×	3.02×	1.72×	2.93×

Table 5.12: Speedup values with the different FPGA configurations

.

Hardware acceleration can speed up the tier 1 coder, with averages over 20× in the Virtex-4 minimum speed, and over 100× as the expected speedup on the Virtex-7. Furthermorer, the "minimum speed" is ensured in hardware, meaning that for any image characteristics, it will perform at that speed. In contrast to software, which heavily depends on image characteristics (more redundant images get compressed faster as evidenced by DHO results).

To see the impact of this on the full algorithm, Figure 5.30 shows the different parts of the algorithm, with the time they take to execute, with and without hardware acceleration. The effect is quite noticeable.

As for numbers, the speedup gained is shown in Table 5.12. The coding speedup is much greater than the overall speedup, which is still quite high due to the fact that coding took the most time out of the execution.

5.4.1.13 Achieving real-time performance

Figure 5.31 shows the final results (time wise) of the JYPEC accelerated algorithm.A first version without optimizations was far away from anything close to real-time, due to long training times in the dimensionality reduction algorithm. Sub-sampling, as seen in Section 5.4.1.2 is a great way of drastically reducing this time without affecting the distortion-ratio performance, thanks to the redundancy present in hyperspectral images.

Results were close (and even on some images such as DHO, better) than real time, but still not quite there. A close analysis to the timing diagrams from Section 5.4.1.10 showed that most of the effort was now devoted to the coding step from JPEG2000. Luckily, that could be accelerated in an FPGA due to the nature of the algorithm (simple bit manipulations with great

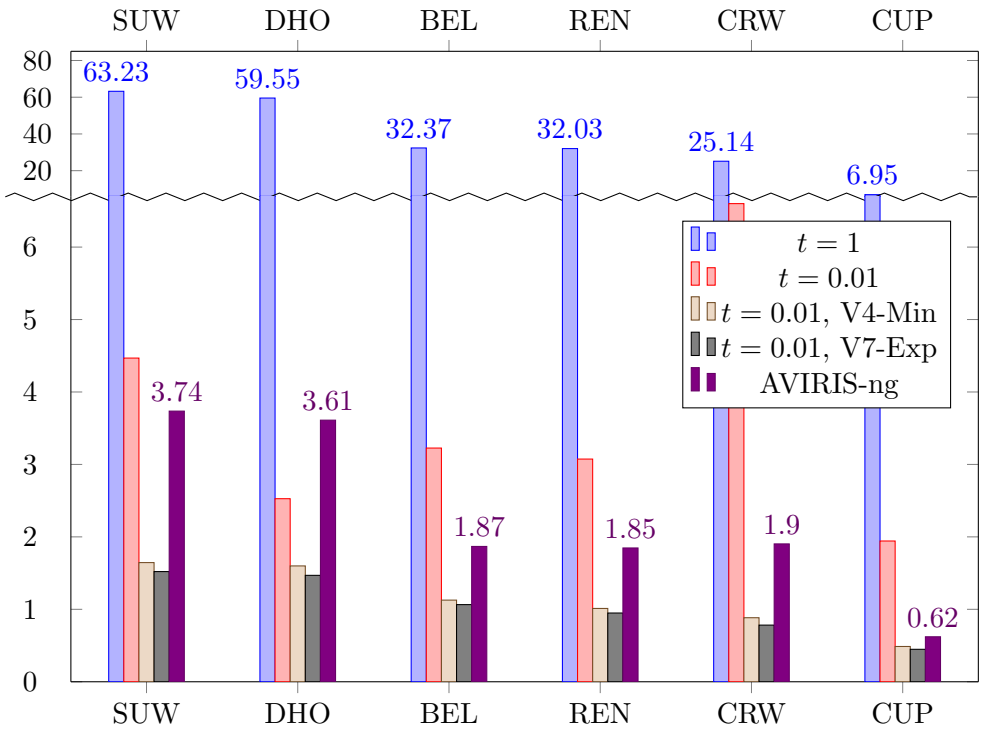

Figure 5.31: Performance of JYPEC through software and hardware optimizations. The effects of training sub-sampling t and JPEG2000 hardware acceleration are shown.

presence of conditional branching, too inefficient in the CPU). The final results show that, even on the slower Virtex-4 board, and assuming a worst case scenario where the input is generating the maximum amount of CxD pairs, the core is still able to beat real time with enough margin.

Care must be taken though with small images such as CUP that have less margin to work with. Others are nearly doubling the speed of real time thanks to hardware-software co-processing, being able to take hits from the unpredictable nature of software execution.

5.5 LCPLC

5.5.1 Software

LCPLC is a simple algorithm whose only meaningful tunable parameters are the slice size, distortion threshold, and quantization step. Quantization and quality do not affect hardware occupancy, with slice size being the only parameter impacting it. Since prediction is done for sub-blocks spanning all bands spectrally with a spatial size equal to the slice size, the predictor is directly affected by this choice. This happens in two ways: first, with bigger slices, less overhead is needed for storing the $\hat{\mu}$ and $\hat{\alpha}$ values, as well as the d flag. However, predictions will be less precise with bigger slices, since they are done for all samples of the slice at once, and big slices mean less adaptability to small local changes.

The algorithm was run with different slice sizes, distortion thresholds and quantization steps to see the distortion-ratio curves. Figure 5.32 shows the results. Aside from the small set of outliers around the $4bpppb$ mark (corresponding to a quantization step of 2^8 and $\gamma = 0$), the rest follow a smooth curve that gains quality quite fast at the start and then gradually slopes up until reaching lossless compression.

Slice sizes of 4×4 and 8×8 are clearly worse than the bigger counterparts, with the best results being obtained at the 32×32 setting, albeit by a narrow margin against the 16×16

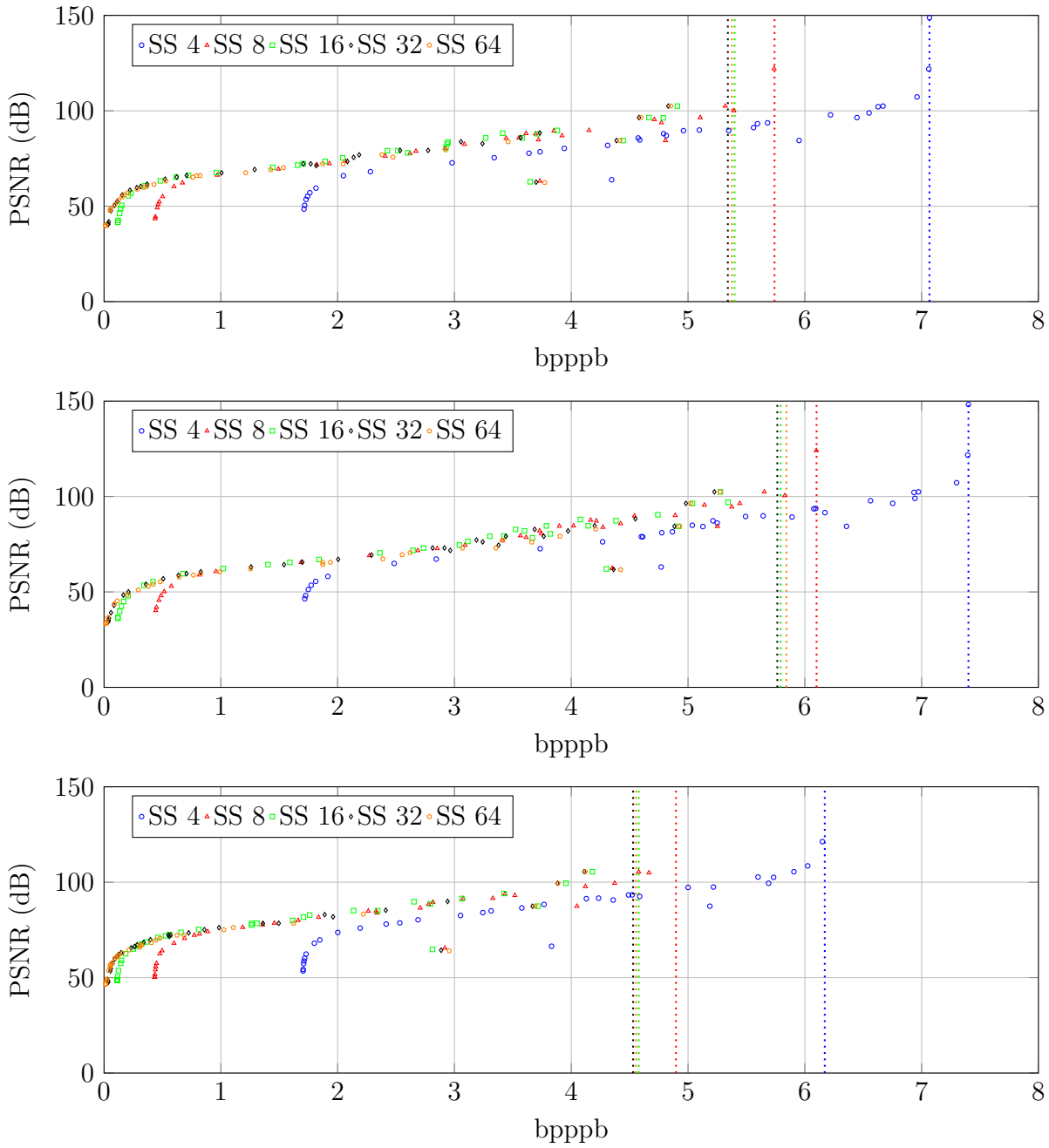

Figure 5.32: The results from compressing respectively the BEL, REN and SUW images on different compression configurations, containing all possible combinations of quantizations steps $0, 2^0, 2^1, 2^2, 2^4, 2^8$ and γ values $0, 0.1, 0.25, 0.5, 1, 3, 5$ for distortion control. Markers show the results for different slice sizes. Vertical lines indicate the *bpppb* at which lossless compression is achieved.

FPGA	Slice	LUT	Regs	DSPs	BRAMs	Occupancy	Power	Frequency
Virtex 7	$\leq 2^4$	5418	5690	5	0.5	1.25%	$0.617W$	$341.65MHz$
	$\leq 2^6$	5448	5706	5	3.5	1.26%	$0.628W$	$341.88MHz$
	$\leq 2^8$	5625	5827	5	4	1.30%	$0.639W$	$341.30MHz$
	$\leq 2^{10}$	5785	5953	5	5.5	1.34%	$0.655W$	$340.83MHz$
	$\leq 2^{12}$	5820	6078	5	17.5	1.34%	$0.709W$	$341.88MHz$
Virtex 5	$\leq 2^4$	6544	5676	5	1	7.99%	$2.650W$	$251.60MHz$
	$\leq 2^6$	6786	5779	5	2	8.28%	$2.668W$	$229.82MHz$
	$\leq 2^8$	6874	5811	5	4	8.39%	$2.710W$	$229.82MHz$
	$\leq 2^{10}$	7077	5942	5	6	8.64%	$2.723W$	$229.82MHz$
	$\leq 2^{12}$	7507	6081	5	18	9.16%	$2.796W$	$242.38MHz$

Table 5.13: Original LCPLC core occupancy results for different maximum slice sizes on the Virtex-7 and Virtex-5 FPGAs. Power is estimated at $300MHz$ and $200MHz$ on the V7 and V5 respectively.

and 64×64 sizes. From a theoretical point of view, smaller sizes are obtained at high gamma values and quantization steps, while higher qualities are achieved at low gamma values and no quantization.

5.5.2 Hardware

From the hardware point of view, only two parameters are important for core occupancy on the FPGA. The maximum slice size that can be processed (this value will be used to size the internal queues) and the maximum image size that can be processed (this value is used to size internal counters for the core that do not have an effect on the processing itself). Only the slice size will be of interest here, since the maximum image size has barely any impact on core occupancy and is set to a default of $4096 \times 4096 \times 4096$, more than enough to meet any hyperspectral image demands. As for other values affecting compression, both the quantization and γ value are configurable in real time, not impacting static hardware resources.

First, a look is taken at the first implementation of the original algorithm from Section 4.3.5, and then it is compared against the modified implementation from Section 4.3.6.

5.5.2.1 Resource use

The original algorithm described in [5] was implemented in both the Virtex-5 and Virtex-7 FPGAs, obtaining the results from Table 5.13.

Core frequency remains mostly constant throughout the experiments, and the differences are due to optimizations within the synthesizer. LUT, register and DSP usage is also fairly constant, with a more noticeable increase in the 32×32 and 64×64 slice size versions. Memory, however, is where the internal queue sizing shows up, by doubling or tripling the amount of blocks in the biggest slice size. However, given the immense amount of blocks available, this has no impact on total resources used, since the LUT usage is higher proportionally. Nonetheless, power is affected, and smaller sizes are prefered for core efficiency.

As for the modified algorithm, the results are shown in Table 5.14. A similar behavior is seen. Frequency is now more stable at different sizes (though this is due to routing and not the actual path being more complex). Memory use is a bit higher at big slice sizes. LUT usage is similar for the modified algorithm due to the increased pipeline being memory heavy and not calculation heavy. Register usage is consistently up to keep up with the increased pipeline. The overall impact in logic is not too significant, except in the odd case of the 32×32 slice size for

FPGA	Slice	LUT	Regs	DSPs	BRAMs	Occupancy	Power	Frequency
Virtex 7	$\leq 2^4$	5544	5950	7	1	1.27%	$0.629W$	$342.58MHz$
	$\leq 2^6$	5598	5965	7	3	1.29%	$0.632W$	$342.81MHz$
	$\leq 2^8$	5880	6062	7	3.5	1.35%	$0.639W$	$342.23MHz$
	$\leq 2^{10}$	6371	6172	7	6	1.47%	$0.672W$	$341.99MHz$
	$\leq 2^{12}$	5963	6245	7	19.5	1.50%	$0.714W$	$321.85MHz$
Virtex 5	$\leq 2^4$	6551	5940	7	1	8.00%	$2.672W$	$258.74MHz$
	$\leq 2^6$	6803	6016	7	2	8.30%	$2.700W$	$252.69MHz$
	$\leq 2^8$	6931	6039	7	5	8.46%	$2.712W$	$258.68MHz$
	$\leq 2^{10}$	7096	6139	7	7	8.66%	$2.732W$	$247.36MHz$
	$\leq 2^{12}$	7432	6253	7	22	9.07%	$2.853W$	$240.66MHz$

Table 5.14: Modified LCPLC core occupancy results for different maximum slice sizes on the Virtex-7 and Virtex-5 FPGAs. Power is estimated at $300MHz$ and $200MHz$ on the V7 and V5 respectively.

	Modified					Original				
Slice size	4^2	8^2	16^2	32^2	64^2	4^2	8^2	16^2	32^2	64^2
Samples	16	64	256	1024	4096	16	64	256	1024	4096
1^{st} slice	202	730	3166	13150	53123	243	918	3624	14387	57562
N^{th} slice	156	301	877	3180	12396	137	332	1103	4173	16448
Intra-slice	59	107	299	1067	4139	84	182	562	2102	8243
Block	21k	38k	109k	389k	1.5M	30k	65k	204k	760k	3M
1^{st} slice *spc*	0.079	0.088	0.081	0.078	0.077	0.066	0.070	0.071	0.071	0.071
N^{th} slice *spc*	0.271	0.598	0.856	0.960	0.990	0.190	0.352	0.456	0.487	0.497
Block *spc*	0.273	0.596	0.845	0.943	0.971	0.192	0.355	0.452	0.485	0.493
MS/s	81.95	179.0	254.4	284.3	292.9	57.59	106.8	135.7	145.6	148.3
MB/s	163.9	358.1	508.8	568.6	585.8	115.2	213.6	271.4	291.1	296.5

Table 5.15: Cycles taken to process the different types of slices. For the first slice, the full time is shown. For the rest, both the time it takes for a slice to be fully processed, as well as the time between subsequent slices enter the pipeline are shown. This last value is more representative of the real time taken to process, since the pipeline hides the full execution time by overlapping slices. $N_z = 360$. *spc* is samples per cycle. $300MHz$ clock and $D = 16$ assumed for throughput calculations.

the Virtex 7 where optimizations place a memory module on LUTs. Two more DSPs are used, which are the two extra DSPs used for the secondary α value calculation, but overall the count is still very low. Power is also affected and is a bit higher to run the memories and registers.

5.5.2.2 Performance of the modified algorithm

The overall resource use is similar in both versions, with more memory used in the modified version given the duplicity of paths. These are used to calculate the two different α options for the next band prediction (either the original or predicted values will be used for it). The modification approximately doubles the speed at which samples can be processed. A more detailed analysis of the performance gain is presented next.

First, Table 5.15 shows the number of cycles taken to process each slice. This is important since the predictor for the first slice is different and slower than the one for the other slices. This induces a penalty for the first slice. Thus, performance will improve with the number of slices.

Results for the first slice are similar, and the difference is due to small optimizations in the number of pipeline stages in the feedback loop, forwarding results to make it faster. The

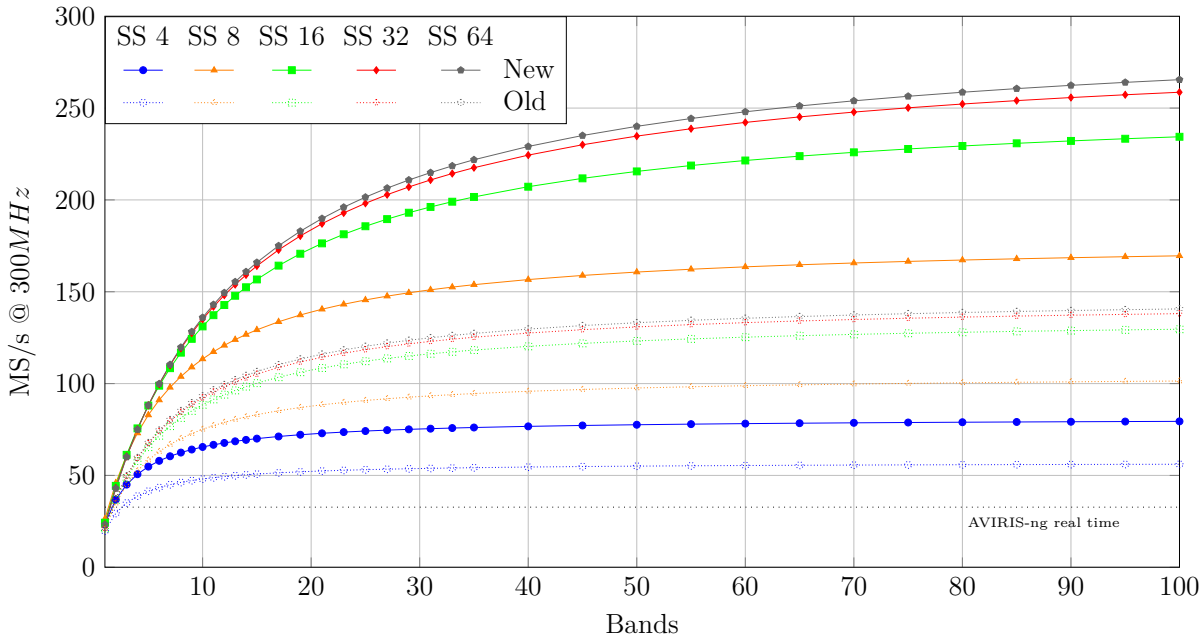

Figure 5.33: Samples per second for LCPLC on the original and modified versions, assuming a $300MHz$ clock. Dotted lines correspond to the original, while solid lines are for the modified.

difference in the rest of slices is more significant, and it is approximately 3/4 lower for the modified version. The difference comes from the fact that the alpha value used for prediction is pre-calculated in the modified version, instead of having to wait for the full d flag to be calculated like in the original version. This reduces the number of "steps" for the full slice from four (average, prediction, error, flag) to three (average, prediction, error).

However, the most interesting improvement comes from the intra-slice results, where the full pipelining achievable in the modified version brings the total cycle count as close as possible to the total number of samples being processed, halving the time of the original version. The penalty for each slice amounts to 43 cycles (time to fill the pipeline between slices), so the relative penalty is much lower with bigger slices and with higher band counts. There is also a second effect at play: since the first slice takes many times the time it takes for the rest, more bands will result in a less noticeable penalty of the first slice (Up to approximately 10× slower on the bigger slice sizes). These effects can be seen in Figure 5.33.

The modified version doubles the throughput of the original in every slice size. The impact of filling the pipeline, and the penalty for the first slice is also noticeable, since low band counts lower efficiency greatly. At band counts above approximately $N_z = 50$, it starts to stabilize reaching values above $250MS/s$ on big slice sizes.

For most of the test images, where $N_Z = 360$, efficiency (*spc* with full block pipelining) reaches 95% on the ideal slice size of 32^2 for the modified version, which is quite close to the ideal one sample per cycle, resulting in a performance of $284.3MS/s$ (vs 145.6 for the unmodified). These facts more than justify the use of the modified version, which only required around 15% more memory resources than the original, producing results with the same distortion-ratio.

For a comparison, the real time performance of an AVIRIS-ng sensor is included in the graph. Every configuration beats it at $N_z > 2$. Even though these speeds might even seem excessive, at high band counts the quite fast throughput can be taken advantage of by lowering clock speed to decrease power consumption.

Chapter 6

Comparison

Throughout the previous chapters, there has been a great level of optimization in all three algorithms:

- From the algorithmic point of view, finding out which small changes could improve performance in any of the different aspects of compressing hyperspectral images. When designing an algorithm, multiple steps are taken along the way that modify the data from its raw starting form to its processed (in this case compressed) form. Different steps can be introduced or modified, and their effects have been studied all throughout.

- From the parameter point of view, once an algorithm is defined, finding out the optimal configurations. It can be made more efficient, needing less operations to complete. Distortion-ratio can be improved by configuration changes alone. Adaptation to user needs can be done to optimize results. The underlying process is the same, but the results are improved.

- Lastly, from the hardware point of view. Multiple options can be chosen: CPUs, GPUs, ASICs and FPGAs are only some of all the available choices. For all of them, small optimizations are possible in the way code is written. Specifically, for FPGAs, taking advantage of different resources such as BRAMs or DPSs can be advantageous, and different hardware tricks can be used to trade off area for performance.

One of the main goals of this thesis is to see how these extensive optimizations work, in three fronts, affects the end result performance when compared to other approaches. In the following sections, all three algorithms will be compared with other implementations, to see which approaches worked best in each case. Lossless, lossy and near-lossless compression will be studied individually. Finally, all of them will be compared as compression algorithms, to determine the trade-offs that make each one ideal for a different scenario.

6.1 CCSDS 123

One of the first interesting comparisons to draw is how efficient different platforms are at executing the algorithms. Since one of the main targets of CCSDS 123.0-B-1 is spaceborne systems, where power is limited, it makes sense to develop an implementation that is not only fast enough, but also consumes as little power as possible. Table 6.1 shows these results for different implementations. An OpenCL implementation was developed to obtain results for the GT 440 and 610 GPUs, as well as the i7-6700 processor.

Platform	Language	Speed	Power	Efficiency
V-5QV FX130T	VHDL	$179.9MS/s$	* $3.04W$	$59.11MS/s/W$
V-5QV FX130T [120]	N/S	$110.0MS/s$	* $3.72W$	$29.54MS/s/W$
V-5QV FX130T [207]	VHDL	$213.0MS/s$	* $4.72W$	$45.13MS/s/W$
V-5QV FX130T [135]	N/S	$55.4MS/s$	* $3.31W$	$16.70MS/s/W$
V-5QV FX130T [180]	VHDL	$11.3MS/s$	$2.35W$	$4.80MS/s/W$
RTAX1000S [180]	VHDL	$4.4MS/s$	$0.17W$	$25.82MS/s/W$
V-4 XC2VFX60 [25]*	VHDL	$116.0MS/s$	* $0.95W$	$122.60MS/s/W$
V-4 LX160 [180]	VHDL	$11.2MS/s$	$1.49W$	$7.51Ms/s/W$
V-5 SX50T [114]	VHDL	$40.0MS/s$	$0.70W$	$57.10MS/s/W$
V-7 XC7VX690T [25]*	VHDL	$219.4MS/s$	$5.30W$	$31.30MS/s/W$
Zynq 7020 [68]	VHDL	$147.0MS/s$	$0.29W$	$489.31MS/s/W$
Zynq 7030 [154]	VHDL	$750.0MS/s$	$0.52W$	$1442.31MS/s/W$
GT 440 [25]*	OpenCL	$62.2MS/s$	$< 65.00W$	$0.96MS/s/W$
GT 610 [25]*	OpenCL	$62.6MS/s$	$< 29.00W$	$2.15MS/s/W$
GTX 560M [91]	CUDA	$297.1MS/s$	$< 75.00W$	$3.96MS/s/W$
2× GTX 560M [91]	CUDA	$329.2MS/s$	$< 150.00W$	$2.19MS/s/W$
i7-6700 [25]*	OpenCL	$35.0MS/s$	$< 65.00W$	$0.54MS/s/W$
i7-2760QM [91]	OpenMP	$118.0MS/s$	$< 45.00W$	$2.62MS/s/W$

Table 6.1: Results for different platforms and their respective performances, sorted by platform. FPGAs, GPUs, and CPUs were targeted. Power values with an asterisk are estimates [233]. A less than symbol indicates maximum TDP (thermal design power) for the platforms, not necessarily the consumed power. Asterisks on the platform indicate own work.

The first thing to note is that, with just a few exceptions, all of the rest reach speeds above the real-time requirement of $30.72MS/s$, whether on FPGA, CPU or GPU. This means that any of those systems are valid candidates. However, it is quite clear that FPGAs and GPUs beat CPUs by quite an ample margin. However, when it comes to efficiency, FPGAs are as much as three to four hundred times more efficient than GPUs or CPUs.

Hardware that adapts to the specific characteristics of the algorithm only uses the required resources and does so in an optimal way, resulting in much more efficient circuits that still meet throughput demands. FPGA versions however take longer to develop, in the case of this work taking around four months including limited testing, versus two weeks for the OpenCL version.

As for the FPGA implementations, different techniques have been used in the literature, having different impact in the total amount of resources required as well as the speed achieved. These values are seen in Table 6.2.

The first implementation [114] is the first adaptation of the Fast Lossless [12] algorithm that ended up being the CCSDS 123.0-B-1 standard. A sample per cycle is processed in this implementation, but at low speeds of $40MHz$. This means that the external RAM running at $100MHz$ can provide the input samples at the required speed. An output packer ensures using LUTs that the outputs are packed in 128-bit words, to output directly to the same RAM. The system is implemented and tested on-board the twin otter aircraft.

A different approach is taken by [180] where, instead of targeting one sample per cycle, the core is simplified to process a sample every few cycles, greatly reducing the required resources, and being able to fit in smaller Space-grade FPGAs. Ordering is changed to BSQ since it provides less dependencies, but more memory is required in the external RAM to store differences from multiple bands. Different image sizes are also supported. Arbitrary size is also explored by [120], realizing that by using BIP mode and enough storage, a full frame can be stored on BRAM to speed up calculations.

Implementation	Regs	LUTs	BRAM	DSP	MS/s	Image size	Order	RAM
Modified FL [114]	1586	12697	8	1	40.0	AVIRIS*	BIP	Yes
HyLoC [180]	1535	2342	0	1	11.3	Any	BSQ	Yes
HP CCSDS [120]	2557	5378	22	6	110.0	Any	BIP	Yes
Parallel CCSDS [25]	141	12783	138	50	179.9	AVIRIS	BIP	No
Efficient CCSDS [68]	2528	3012	84	6	147.0	AVIRIS	BIP	No
Fast 3.3G [207]	9990	9462	83	6	213.0	AVIRIS	BIP	No
SHyLoC [177]	3658	5815	74	8	111.8	AVIRIS**	Any	Opt
Par. Eff. CCSDS [154]	10696	12033	71	30	750.0	AVIRIS	BIP	No
SHyLoC 2.0 [16]	2736	4809	74	10	138.3	AVIRIS	BIP	No

Table 6.2: Different implementations of the CCSDS 123.0-B-1, sorted by chronological order of publication. [25] is own work. *: AVIRIS frame block with 32 frames. **: Different sizes up to AVIRIS. Results correspond to the Virtex-5 FPGA except for [68] (Z7020) and [154] (Z7035).

After that, the concept of parallel execution using BIP mode [25] was introduced. Samples from multiple bands could be compressed at the same time, since dependencies only exist at the difference level, which can be shared. This concept uses more resources than the previously optimized versions (in this case long combinatorial paths result in low register use and high LUT count) but reaches higher performances. The limit is the serial part of the algorithm, where outputs are packed in 32-bit words. The theoretical throughput limit for a Virtex-5 with optimal packing is closer to $250MS/s$.

Another possibility for increasing performance was used in [68], were execution of samples in BIP mode was overlapped in a 15-stage pipeline. For images with over 15 bands on BIP mode, this meant a one sample per cycle performance using a single predictor, and thus resource requirements fell dramatically while retaining parallel performance. The same concept was improved upon in [207], where more resources were used in a deeper variable-depth pipeline, improving again performance.

The concept of arbitrary size compression was again explored in [177] taking advantage of the developed advancements. Arbitrary image sizes can be compressed, with RAM being used only if strictly necessary under some configurations such as BSQ with high P value, this was further improved in [16]. Both algorithms are also capable of using the block encoder from the standard, which offers more options than the simpler sample-oriented encoder.

Finally, the concepts of parallel band execution and pipelined execution were merged in [154] yielding an astounding $750MS/s$ by pipelining packs of 5 bands in a modified pipeline from [68], capable of compressing a single AVIRIS image in $48.82ms$. The power efficiency as well reached if $1442.31MS/s/W$ values that allow it to compress approximately 73735 AVIRIS images using a single Wh of energy.

6.2 JYPEC

As a lossy algorithm, two aspects are compared in the following lines: first, the distortion-ratio performance, and then, the JPEG2000 coder performance compared to other FPGA implementations.

6.2.1 Lossy performance

JYPEC was based on the idea from [57] where PCA and JPEG2000 were combined to get a hyperspectral compression algorithm. It was then demonstrated that this novel algorithm out-

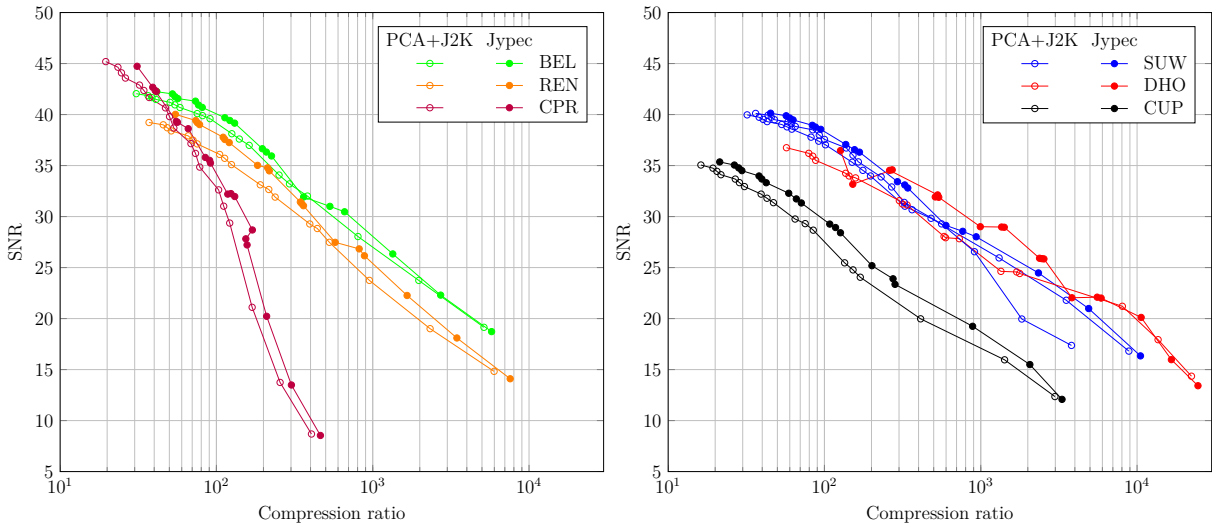

Figure 6.1: Results of optimally applying JYPEC and PCA+JPEG2000 over some test images.

performed 3D wavelet transforms over the hyperspectral data, since higher spectral decorrelation was achieved even though the 3D wavelet transform treats all dimensions as independent.

The improvements from Section 3.3 are mainly the vector quantization step before PCA, and the bit shaving techniques that remove more bits from less important components. When this parameters, along with all the rest that are given optimally by the F value, are fed into both JYPEC and PCA+JPEG2000 over the test set of images, the distortion-ratio curves shown in Figure 6.1 are obtained.

A clear improvement is shown, seeing that both quality and ratio are improving by using the techniques from JYPEC. Even though the images have different characteristics, improvements are seen across the test set. Furthermore, Figure 6.2 shows that this is achieved with faster compression and decompression times, gaining for each configuration both in ratio and quality. Nearly all configurations improve the results, obtaining smaller output sizes and better quality compressed images in lower times.

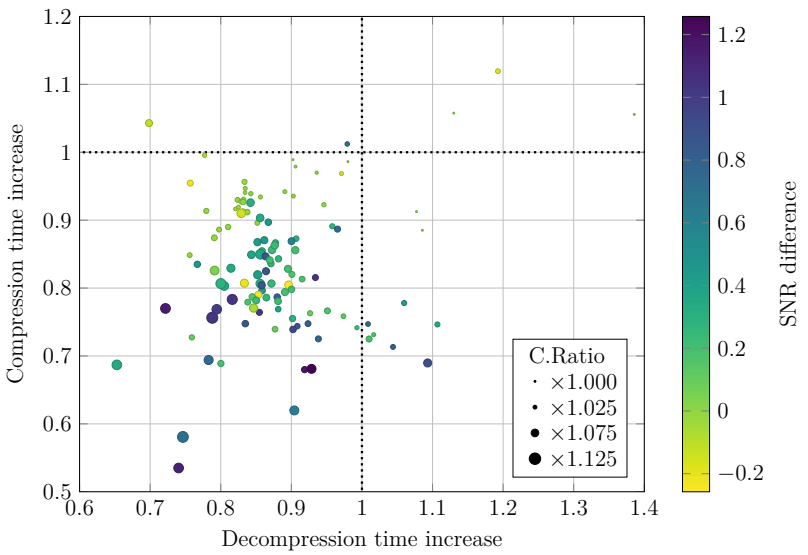

Figure 6.2: Timing and distortion-ratio results at different values of F_j for all test images. Bigger marks indicate a higher compression ratio (always achieved). SNR difference values above 0.0 indicate an increase in SNR.)

6.2.2 Hardware acceleration

The biggest bottleneck in JYPEC was the coding part of JPEG2000. This is not new, and coding has been known to be the bottleneck of JPEG2000 [93, 130, 132, 152, 182].

Coder	Ref	Technology	Frequency	Speed	Slices	BRAM/b
BPC	[86]	APEX20KE FPGA	$51.7MHz$	$73.44Mb/s$	956	n/a**
	[110]	XCV600e-6BG432	$52MHz$	$94.4Mb/s$	n/a	n/a**
	[142]	EP20K600EFC672–3	$100MHz$	$40.5Mb/s$	1850	0**
	This	Virtex-4 FPGA	$127.8MHz$	$190.2Mb/s$	1850	4
	This	Virtex-7 FPGA	$247.8MHz$	$368.8Mb/s$	731	2
MQ	[38]	$0.35\mu m$	$90MHz$	$180MCxD/s$	n/a	n/a
	[129]	$0.35\mu m$	$150MHz$	$300MCxD/s$	n/a	n/a
	[58]	Stratix	$48.8MHz$	$97.7MCxD/s$	1596	8192b
	[58]	$0.18\mu m$	$211.8MHz$	$423.7MCxD/s$	n/a	n/a
	[142]	EP20K600EFC672–3	$26.29Mhz$	$52.58MCxD/s$	1811	n/a
	[182]	Stratix FPGA	$153MHz$	$137.7MCxD/s$	279	1344b
	[165]	$0.18\mu m$	$413MHz$	$413MCxD/s$	n/a	n/a
	[123]	Stratix FPGA	$106.2MHz$	$212.4MCxD/s$	1267	0
	[132]	XC4VLX80 FPGA	$48.3MHz$	$96.6MCxD/s$	6974	1509b
	[132]	$0.18\mu m$	$220MHz$	$440MCxD/s$	n/a	n/a
	[181]	Stratix EP1S10B672C6	$136.9MHz$	$136.9MCxD/s$	695	3301b
	[183]	Stratix FPGA	$146MHz$	$146MCxD/s$	824	428b
	[6]	$0.18\mu m$	$208MHz$	$192.8MCxD/s$	n/a	n/a
	[124]	Stratix II FPGA	$106.2MHz$	$212.4MCxD/s$	1267	1321b
	This	Virtex-4 FPGA	$121.2MHz$	$121.2MCxD/s$	1689	1
	This	Virtex-7 FPGA	$321.5MHz$	$321.5MCxD/s$	1778	0
Tier 1	[130]	$0.35\mu m$	$50MHz$	$36.5Mb/s$	n/a	n/a
	[71]	Virtex II XC2V1000	$50MHz$	$91.18Mb/s$	4420	3120b**
	[174]	Virtex II Pro FG 456	$112MHz$	$181.6Mb/s*$	2504	28
	This	Virtex-4 FPGA	$99.38MHz$	$147.9Mb/s$	3803	8
	This	Virtex-7 FPGA	$255.3MHz$	$380Mb/s$	2708	4

Table 6.3: Comparison of different hardware accelerators for the Tier 1 coder (or parts of it) of JPEG2000. *The value is not specified, but given the architecture, it is expected to have the same relationship with frequency than the one presented here. **Requires external memory for data and/or internal variables.

This has prompted many FPGA and ASIC implementations to arise in order to accelerate the tier 1 coder, or parts of it, in JPEG2000. This also stems from the fact that any image capturing device contains accelerators for JPEG encoding, and a transition to the more complex JPEG2000 coding would mean that accelerators for it would also be required. However, there are few implementations targeting the full tier 1 coder, and usually focus on either the BPC or the MQ-coder only.

Different approaches for accelerating the different modules can be seen in Table 6.3.

A comparison is difficult to draw given the multiple architectures and designs available to choose from. Fortunately, speed is not a problem in the context of hyperspectral image compression if dimensionality has been reduced. The margin in compression speed from Section 5.4.1.11 is enough that even with a slower implementation, it still meets the imposed real-time constraints, since it is so fast already when compared to the software version that time is now dominated by the slow dimensionality reduction part. Hardware-wise, FPGA implementations are very light on resources, being able to instantiate many cores even on small FPGAs, which enables parallel execution on multiple blocks at the same time if necessary. The fastest FPGA design is this work's Virtex-7 synthesis, but this could be due to using newer technology and not necessarily having a better design.

Design	Platform	Frequency	Speed	LUTs	BRAMs	DSPs	Power
[95, Ch.3]	4VLX200	$81MHz$	$70MS/s$	10306	21	9	$1.1W$
[178]	Virtex 5	$80.21MHz$	$30.25MS/s$	7836	4	17	n/a
	4VLX200	$77MHz$	$29.04MS/s$	10015	4	20	n/a
[76]	Virtex 5	$86.96MHz$	$27.9MS/s$	7746	4	25	n/a
	4VLX200	$75.84MHz$	$24.33MS/s$	9283	4	25	n/a
[75]	RTAX2000S	$18.65MHz$	$6.05MS/s$	18101	7	n/a	$0.378W$
[179]	T-C2075	$1150MHz$	$130MS/s$	n/a	n/a	n/a	$< 225W$
This	Virtex 5	$247.36MHz$	$233.25MS/s$	7096	6	7	$2.732W$
	Virtex 7	$341.99MHz$	$322.50MS/s$	6371	7	7	$0.672W$

Table 6.4: Different implementations of LCPLC and their characteristics.

Also, the fact that FPGAs and ASICs can compete with each other, is proof for the viability of using FPGAs directly, instead of spending time and effort in designing a custom ASIC. Specially for such a small core that only needs a very small FPGA. The ASIC version will of course use less power, so in very constrained scenarios it might still make sense. However, software solutions are completely out of the question.

Note that much more efficient BPC exist that deviate from the standard. A pass-parallel architecture, introduced in [66]. Efficiencies of up to $55MPixel/s$ are achieved at $16b$ depths on $0.18\mu m$ CMOS technology [240], later improved to $92.8MPixel/s$ at $10b$ [181] on a Stratix FPGA. Bit-wise, these correspond to $880Mb/s$ and $928Mb/s$ respectively. Even over $1.2Gb/s$ is reported in [129]. They are much faster than standard bit-sequential BPC. This approach was not chosen because timing demands are met nonetheless, and the slight loss in compression ratio lowers the overall algorithm's performance, as well as taking many more resources due to MQ-coder replication to keep up with the BPC output.

6.3 LCPLC

The first LCPLC implementation was done by their authors in [95, Ch.3], targeting a space qualified FPGA, achieving high throughput by decomposing the algorithm into elementary blocks that communicate via ping-pong buffers.

More work was done on simpler HLS implementations [75, 76, 178, 179]. A modular architecture was developed were each main stage of the algorithm had its own module. Multiple RAM blocks were used to interconnect and transmit data between the modules. This meant that data transmission had greater overhead than a direct bus connection, resulting in lower speeds. However, development was much quicker thanks to the ease of use of HLS synthesis.

A different approach was taken in [179] that used a GPU for acceleration, bringing the design to speeds four times higher than the fastest previous FPGA version. The parallel nature of GPUs exploits block predictions processing multiple samples at the same time.

The approach taken in this work, which is that of a very optimized pipeline version of the algorithm, is able to surpass the GPU version previously mentioned, with less resources used than the FPGA work that already existed, thanks to using the more customizable VHDL code instead of HLS synthesis, and AXI-Stream based FIFOs instead of ping-pong buffers. Comparisons can be seen in Table 6.4.

	CCSDS		LCPLC		JPEG			
Image	123.0-B-1	122-0-B	16	32	LS	2000	ESA	LUT
HAW	**2.62**	3.29	3.00	2.92	3.27	3.99	2.94	3.38
MAI	**2.73**	3.36	3.14	3.11	3.36	3.09	3.07	3.51
YEL00	**6.19**	6.7	6.44	6.41	6.95	6.65	6.45	7.15
YEL03	**6.06**	6.54	6.29	6.28	6.83	6.47	6.3	6.93
YEL00	**3.96**	4.63	4.13	4.06	4.73	4.46	4.59	4.85
YEL03	**3.83**	4.5	3.94	**3.83**	4.63	4.31	4.46	4.65
YEL10	3.37	3.94	3.28	**3.12**	4.01	3.68	3.8	3.93
YEL11	3.64	4.31	3.70	**3.60**	4.28	4.1	4.18	4.43
YEL18	**3.91**	4.68	4.04	3.96	4.68	4.51	4.56	4.86
Average	**4.03**	4.66	4.22	4.14	4.75	4.58	4.48	4.85

Table 6.5: Results (*bpppb*) of losslessly compressing multiple images with different algorithms. Data for algorithms other than LCPLC is taken from [33]

6.4 Algorithm comparison

All three algorithms: CCSDS 123.0-B-1, JYPEC, LCPLC and their implementations have their own specific applications and characteristics. CCSDS is lossless, JYPEC is lossy, and LCPLC sits in between with both capabilities being near-lossless. Also, CCSDS and LCPLC are simple enough to be synthesized as stand-alone cores on an FPGA, while JYPEC is just accelerated by an FPGA with a general purpose processor behind.

The common characteristic in all of them is the ability to compress hyperspectral images in real time. With all of them meeting the real-time constraints and the occupancy constraints of the FPGAs, the following lines are devoted to comparing their distortion-ratio performance. They are also measured against other hyperspectral algorithms to see their relative performance.

First, lossless compression is compared in Table 6.5 for multiple lossless algorithms. These include the CCSDS 123.0-B-1 standard and LCPLC algorithm in slice size 16 and 32 that are studied in this work. Also included are the JPEGLS (JPEG lossless) algorithm and lossless version of JPEG2000 (both designed for non-hyperspectral images). As for hyperspectral algorithms, the predecessor to CCSDS 123: CCSDS 122 is included, as well as the LUT compressor [143] and ESA compressor [95, Ch.3] (original LCPLC with no modifications) that target hyperspectral compression.

Even though all algorithms are similar in results, CCSDS 123.0-B-1 generally outperforms the rest, achieving higher compression ratios. LCPLC in its slice size of 32 also achieves some of the best results for some images, with the ESA algorithm very close on some runs. JPEGLS, JPEG2000, CCSDS 122-0-B and LUT all follow way behind, proving that more complex algorithms do not necessarily translate in higher compression ratios. It is surprising however to see how close CCSDS 123.0-B-1 and LCPLC are, considering the much lower complexity of the latter (batch slice predictions instead of per-sample weighted predictions).

As for lossy algorithms, the performance of LCPLC and JYPEC is measured against the newly developed near-lossless CCSDS 123.0-B-2 and PCA+JPEG2000 in Figure 6.3.

It is very evident that JYPEC and PCA+JPEG2000 outperform the others at low *bpppb*. JYPEC is consistently above, thanks mainly to the increase performance due to selective bit-depths applied to the different bands after dimensionality reduction. Both lose performance when approaching the high-quality zone, where the near-lossless algorithms leap forward. Between those, LCPLC is better at low qualities since its batch-prediction characteristic is able to reduce much more the size than the sample-based prediction in CCSDS 123.0-B-2. At high bit-rates,

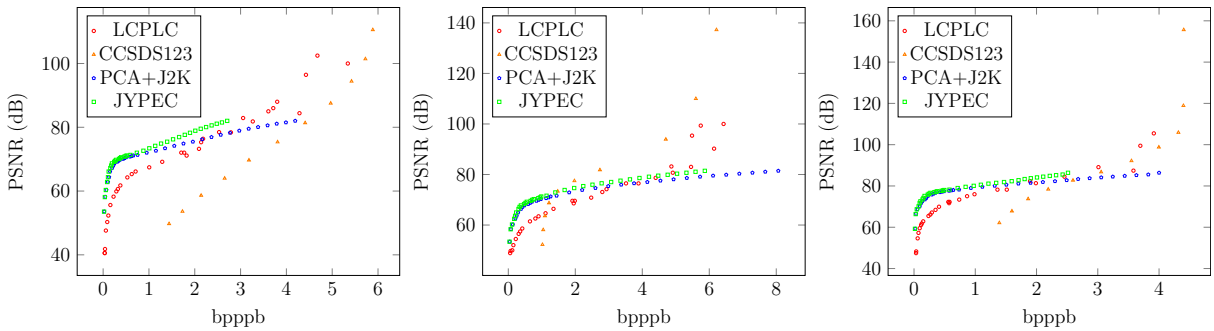

Figure 6.3: Results of applying different algorithms on the BEL, CUP and SUW image respectively.

however, both algorithms are similar, with performance depending on the image characteristics. LCPLC is preferred given its faster performance and low occupancy.

All in all, all algorithms can reach their real-time target, and it is best to use them depending on what the compressed data will be used for. If perfect reconstruction is needed, a lossless algorithm should be used. For high compression of images and storage, a complex algorithm such as JYPEC is ideal. However, its multitude of steps introduce losses that are too high at high bit-rates. For an all-around great algorithm that can perform well in both lossless and lossy situations, a near-lossless algorithm is a safe bet that will in most cases outperform the former two choices.

Chapter 7

Conclusions

7.1 Summary

Hyperspectral images bring endless possibilities in many fields, mainly focusing on different types of remote studies of the Earth. From military purposes such as target detection, to geological studies about mineral abundance, to vegetation analysis or urban surveillance, all of the possible applications have one thing in common: they need massive amounts of data to work with.

The minimum unit of information in the hyperspectral image context can be considered the hyperspectral pixel. By itself, a single pixel spans a wide range of the electromagnetic spectrum, sampling wavelengths at evenly spaced intervals throughout its range. At usually two bytes per sample, a single pixel might end up taking a kilobyte of storage. Millions of these pixels are needed to perform in-depth studies in different fields, and sensors often capture multiple *TeraBytes* of information on their runs in order to build up a strong sample library.

Even though storage has increasingly been made available for cheap, it is not free. Cloud storage services require expensive fees to maintain, and custom solutions targeting hyperspectral imagery are expensive to build. Compression of these images seems like a logical step in the processing flow to reduce these expenses. And doing it in real time is of special interest for being able to set up a capture-compress-store pipeline that avoids the need for either huge buffers or capture stalls.

The possibilities for compressing hyperspectral imagery are endless. Lossy algorithms that preserve the raw information to help with precise scientific studies. Lossless algorithms that target bulk storage of data in a compact way without sacrificing quality. Near-lossless algorithms in which quality or data rates can be controlled obtaining results within set size or reconstruction constraints. All of them with different characteristics but one common objective: reducing data size.

In this thesis, three algorithms (one on each category) have been studied from the algorithmic and implementability point of view. FPGAs have been targeted since some models can be used out of the box in the harsh air or spaceborne conditions that hyperspectral sensors usually work under. Their reconfigurability offers a unique advantage over other systems: performance and low power consumption within a single die.

The three algorithms studied have been: CCSDS 123.0-B-1: an international standard aiming at lossless compression. JYPEC: a custom lossy algorithm based on the most promising published techniques. And LCPLC, a slight modification of a near-lossless algorithm designed for the Exomars mission.

Results show that every option has been able to reach real-time performance with FPGA acceleration. Both CCSDS and LCPLC have been developed as stand-alone cores, while JYPEC

is partially implemented as an accelerator. The first two, thanks to the simple nature of their predictors and encoders, show remarkable performance beating both GPUs and CPUs in speed and power efficiency. The latter is more limited in speed and requires co-processing along a general purpose processor.

7.2 Thesis work

The problem of hyperspectral image compression has been evident since hyperspectral imagery was developed. As such, multiple algorithms were available at the start of this thesis. The first step was thus to study and compare the different options that were available, seeking the best algorithms that might benefit from FPGA execution. Three main types were identified: Lossless, lossy, and near-lossless.

The first type, targeting perfect reconstruction after the compression-decompression process, had the most options available. General data compression is mostly lossless, since most applications compressing text, databases or programs cannot afford to lose a single bit, or the applications depending on them might stop working or completely malfunction. This availability of lossless algorithms translates to hyperspectral compression. An international standard had already been developed to tackle this exact issue. Multiple options had been proposed, and the CCSDS 123.0-B-1 algorithm had been developed as a result. It was simple enough to be implemented on an FPGA, and offered the best compression ratios when compared to other alternatives, being the best candidate for its category.

For the lossy options, multiple options were also available. This stems from the fact that images are one of the most common data types in computing. For their general uses, perfect quality is usually not needed, and losses can be afforded even for scientific studies. Their ideas usually involve some kind of signal processing over the 2D matrix of pixels. These ideas had been adapted to hyperspectral imagery. The best results were achieved by separating the spectral decorrelation from the spatial decorrelation, noticing that they were independent. An algorithm including PCA and JPEG2000 for the spectral and spatial decorrelation respectively was the most promising, and was selected as a starting point for the lossy approach.

Finally, near-lossless compression hadn't been explored as much, with usually lossy or lossless approaches being sufficient for most applications, since they generally outperform near-lossless compression in their respective fields. However, it was still interesting to study an algorithm that, while not being the best at lossy or lossless compression, can seamlessly move between both to adapt to possibly changing constraints. The algorithm designed for the ExoMars mission was selected as a starting point in this category.

All of these options were explored with FPGA implementations in mind. Their specific characteristics were studied aiming for optimizations in the development process, targeting real-time performances.

7.2.1 Timeline

A CCSDS core was first developed in VHDL, adding parameters for all of the configuration options available in the standard. The configuration is static, meaning that the core is synthesized for a specific image size and compression parameters. While this is a problem in ASICs, it can easily be reconfigured on an FPGA, taking advantage of the fully optimized circuit that results from this specificity. A software was also developed to cross-check compression results with both the developed core and existing software implementations. The core was tested, but its complexity was still too high for real-time compression on small space-grade FPGAs. As such, parallelization was then tried as a way of speeding up the module. The type of parallelization

applied restricted some constraints for the algorithm configurations, which luckily were not common characteristics in the available sensors. Real-time was greatly surpassed, achieving results even faster than full GPU implementations, and proving that even old FPGAs were still future proof against sensor improvements.

A more algorithmic approach was first taken for the lossy PCA+JPEG2000 starting point. With the aim of improving the distortion-ratio performance, the literature was extensively researched for different compression techniques and approaches that were tried to see which worked best in the hyperspectral context. These were tried in a custom software, designed to pipeline multiple compression steps to generate a viable compression algorithm. The best results came from Vector Quantization before the PCA step, plus variable bit-rates in the latent component compression. This improved the existing algorithm in both distortion-ratio and compression/decompression speed. A directed search in the parameter space was performed to find out the best possible algorithm configurations to get an optimal distortion-ratio curve across the whole spectrum of qualities. The resulting algorithm was deemed too complex for a full-on hardware implementation, which would've required multiple FPGA cores and memory synchronization between them. Instead, the algorithm was profiled and the JPEG2000 coder found to be the bottleneck. The literature was extensive in this subject, and multiple techniques were tried, arriving at a core that was competitive even with CMOS technology. The combination of CPU+FPGA was able to bring lossy hyperspectral compression to real-time, albeit at the cost of needing a complex system instead of a simple FPGA core, and limiting the algorithm parameters to those that didn't require excessive computing resources. This still limited its applicability in air and spaceborne applications, being more suited for data center processing.

Finally, the near-lossless algorithm LCPLC algorithm from the ExoMars mission was also algorithmically analyzed. With block-oriented predictions instead of sample-oriented predictions, data dependencies were very far apart in the processing flow. A hugely pipelined execution was deemed possible, and extremely optimized simple modules were developed as the pipeline components. Results of this first version already surpassed even the parallelized CCSDS algorithm with a smaller core. Throughout development, it was realized that a small change in the algorithm, with no impact in distortion-ratio, could be introduced halving the pipeline stalls (doubling performance). Even though this increased the resource requirements (specially registers), the modified core still fits in a space-grade FPGA, reaching speeds well above any previous lossy or lossless hyperspectral compression algorithm, being almost ten times faster than the real-time constraint from the AVIRIS-ng sensor.

The algorithms and their implementations were then compared, demonstrating that the specific lossless and lossy algorithms are still the best option in their domains (rather than the near-lossless one). However, the difference is very close, and the adaptable near-lossless algorithm is also far more efficient resource wise and quite faster in compression speed. These facts fairly outweigh the slight loss in compression efficiency in the lossless and extremely lossy extremes.

In any case, the most important factor when determining the algorithm complexity on an FPGA is whether its design is easily portable or adaptable to reconfigurable hardware. Spending time in reviewing and adapting the algorithms for FPGAs is almost as important, if not more, than developing a great implementation. Software tests must be done beforehand to optimize the algorithm and determine its best parameters for the target application. The target platform must be taken into account in design, taking advantage of the fast conditional execution of an FPGA, or the massive parallelism of a GPU. It is also clear that, when that target platform can be an FPGA, the results far outperform the competition.

7.3 Future Work

The results from the near-lossless algorithm indicate that this option is the best for FPGA implementation of hyperspectral compression algorithms. Despite needing more FPGA resources than the other types, even older models are not limited to host the LCPLC implementation. This highly pipelined approach could be used for other compression algorithms, or for other processing algorithms that rely on pipelined steps.

A framework could be built around the individual modules developed, with a specific language indicating how elements connect with and depend on each other. This would semi-automate the generation of these kind of cores. The idea is similar to that of HLS, but in this case falling back to highly hand-optimized modules, which will probably outperform HLS-inferred ones.

This would not be a substitute for HLS, but a niche application for a specific kind of problem, where the benefits of having a custom pipeline are greatly noticeable. It is of special interest for any kind of image or signal processing from a sensor, were usually, in real time, the input data needs to be processed in some way. And specifically, for hyperspectral image processing.

There are many options available, but the flexibility of FPGAs is unpaired. Being cheap and fast when compared to other options, the development of new tools to ease their programmability certainly brings them closer to the average user, and help their integration as accelerators in every platform.

Regarding hyperspectral compression algorithms in general, a unified software framework to implement them, along with a library of standard test images, would help in making comparisons more fair. Results are often given for different images, or as averages of multiple runs for lossless algorithms, making comparisons hard. Lossy algorithms are even harder to compare since quality metrics are sometimes not uniform across papers.

Unifying this into a testing suite where new algorithms could be implemented, would certainly benefit the hyperspectral compression community with a tool able to properly compare performances, since currently comparing different algorithms sometimes involves implementing them again given the unavailability of either the code or the test image suite used.

Not only on the software side, but being also able to test different architectures would highly benefit the conclusions as to what platform is better suited for different scenarios. Testing can be done for multiple platforms once a generic software has been built. GPUs can be targeted for parallelization capabilities. Architectures such as ARM, more power efficient than x86, can also be directly targeted as fully self-contained accelerators. Finally, embedded GPUs and vector processors can be studied under real-time constraints, since they are, as FPGAs, optimized towards power consumption.

The field is still active and thriving, with special focus put into heavy optimizations now after a broad theoretical exploration in the early 2000s. New sensors are being made every year, and are being launched to space even more frequently. With FPGAs growing into every market, their application to hyperspectral imagery will only continue to grow moving forward.

7.4 Publications and contributions

Throughout the development of this thesis, publications have been made across different Journals, as well as some participations in different public forums. Below they are listed in chronological order, along with the importance metrics when appropriate.

- Daniel Báscones, Carlos González, and Daniel Mozos. "FPGA implementation of the CCSDS 123.0-B-1 standard for real-time hyperspectral lossless compression". In: *IEEE*

Journal of Selected Topics in Applied Earth Observations and Remote Sensing 11.4 (2017), pp 1158-1165. Impact factor: 2.777. Q2 in Electrical and Electronic Engineering, Remote Sensing, Image Science and Photographic Technology, Physical and Geography. (Cited 20 times (Google Scholar)).

- Daniel Báscones, Carlos González, and Daniel Mozos. "Parallel implementation of the CCSDS 123.0-B-1 standard for hyperspectral lossless compression". In: *MDPI Remote Sensing* 9.10 (2017), p 973. Impact factor: 3.406. Q1 in Remote Sensing. (Cited 16 times (Google Scholar)).

- Daniel Báscones. "Real-time Hyperspectral image compression using FPGAs". In: *Hipeac ACACES 2018* (2018).

- Daniel Báscones, Carlos González, and Daniel Mozos. "Hyperspectral image compression using vector quantization, PCA and JPEG2000". In: *MDPI Remote Sensing* 10.6 (2018), p 907. Impact factor: 4.118. Q1 in Remote Sensing. (Cited 13 times (Google Scholar)).

- Daniel Báscones, Carlos González, and Daniel Mozos. "An extremely pipelined FPGA implementation of a lossy hyperspectral image compression algorithm". In: *IEEE Transactions on Geoscience and Remote Sensing* (2020). 2018 Impact Factor: 5.63. Q1 in Remote Sensing, Geochemistry and Geophysics, Electrical and Electronic Engineering, Imaging Science and Photographic Technology. (Cited 3 times (Google Scholar)).

- Daniel Báscones, Carlos González, and Daniel Mozos. "An FPGA Accelerator for Real-Time Lossy Compression of Hyperspectral Images". In: *MDPI Remote Sensing* 12.16 (2020), p 2563. Impact factor: 4.118. Q1 in Remote Sensing.

As well as the scientific contributions, the code for all of the projects has been compiled into repositories and made public for anyone to use or expand upon:

- The CCSDS123 software repository [18], with CCSDS 123.0-B-1 and 2 capabilities for both compression and decompression, used to verify the CCSDS 123.0-B-1 FPGA core.

- The Parallel CCSDS123 hardware repository [22], containing all of the modules for parallel and serial CCSDS 123.0-B-1 core generation, with all of the different options and parameters that were presented in this work.

- The JYPEC software repository [19]. It contains the full JYPEC compression suite, as well as packages for hyperspectral image reading and writing, and matrix quality measurements that can be used for other purposes.

- The VYPEC hardware repository [23]. All of the modules used for the implementation of the tier 1 coder of JPEC2000, included in the JYPEC flow, are uploaded in this repository.

- The LCPLC repository [20], containing both the software and hardware implementations of the LCPLC algorithm, using the improved fast version presented in this work.

- The AXI-Modules repository [17], a collection of the basic modules used in LCPLC packaged in a small repository that can be reused for other purposes.

- The MERVI (Matrix ERror VIsualizer) repository [21]. A repository built on top of JYPEC's hyperspectral functionality, to visually see the differences between different hyperspectral images, as well as easily rendering an image using three bands as RGB colors.

Bibliography

[1] Hervé Abdi and Lynne J. Williams. "Principal component analysis". In: *Wiley interdisciplinary reviews: computational statistics* 2.4 (2010), pp. 433–459. arXiv: arXiv:1011.1669v3.

[2] Maleen Abeydeera et al. "4K real-time HEVC decoder on an FPGA". In: *IEEE Transactions on Circuits and Systems for Video Technology* 26.1 (2016), pp. 236–249.

[3] Norman Abramson. "Information theory and coding". In: (1963), p. 201.

[4] Andrea Abrardo, Mauro Barni, and Enrico Magli. "Low-complexity lossy compression of hyperspectral images via informed quantization". In: *Proceedings - International Conference on Image Processing, ICIP* (2010), pp. 505–508.

[5] Andrea Abrardo, Mauro Barni, and Enrico Magli. "Low-complexity predictive lossy compression of hyperspectral and ultraspectral images". In: *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings* (2011), pp. 797–800.

[6] M Ahmadvand and A Ezhdehakosh. "A New Pipelined Architecture for JPEG2000". In: *World Congress on Engineering and Computer Science* II (2012).

[7] Bruno Aiazzi, Stefano Baronti, and Luciano Alparone. "Lossless compression of hyperspectral images using multiband lookup tables". In: *IEEE Signal Processing Letters* 16.6 (2009), pp. 481–484.

[8] Toygar Akgun, Yucel Altunbasak, and Russell M. Mersereau. "Super-resolution reconstruction of hyperspectral images". In: *IEEE Transactions on Image Processing* 14.11 (2005), pp. 1860–1875.

[9] Naveed Akhtar, Faisal Shafait, and Ajmal Mian. "Bayesian sparse representation for hyperspectral image super resolution". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 07-12-June (2015), pp. 3631–3640.

[10] Amazon. *Amazon EC2 F1 Instances*. URL: https://aws.amazon.com/ec2/instance-types/f1/?nc1=h{_}ls (visited on 06/29/2020).

[11] Kishore Andra, T Acharya, and Chaitali Chakrabarti. "Efficient VLSI implementation of bit plane coder of JPEG2000". In: *Applications of Digital Image Processing Xxiv* 4472.December 2001 (2001), pp. 246–257.

[12] Nazeeh Aranki et al. "Fast and adaptive lossless on-board hyperspectral data compression system for space applications". In: *IEEE Aerospace Conference Proceedings*. 2009.

[13] Jean Pierre Ardouin, Josée Lévesque, and Terry A. Rea. "A demonstration of hyperspectral image exploitation for military applications". In: *FUSION 2007 - 2007 10th International Conference on Information Fusion* August 2007 (2007).

[14] ARM. "AMBA 4 AXI4-Stream Protocol Specification v1.0". In: (2010).

[15] *Barn*. Dominio público. URL: commons.wikimedia.org/wiki/File:Barn-yuv.png.

[16] Yubal Barrios et al. "SHyLoC 2.0: A Versatile Hardware Solution for On-Board Data and Hyperspectral Image Compression on Future Space Missions". In: *IEEE Access* 8 (2020), pp. 54269–54287.

[17] Daniel Báscones. *AXI-modules*. 2019. URL: https://github.com/Daniel-BG/axi-modules (visited on 06/01/2020).

[18] Daniel Báscones. *CCSDS 123 Implementation*. 2019. URL: https://github.com/Daniel-BG/CCSDS123 (visited on 04/15/2020).

[19] Daniel Báscones. *Jypec*. 2018. URL: github.com/Daniel-BG/Jypec (visited on 01/17/2018).

[20] Daniel Báscones. *Lcplc*. 2019. URL: https://github.com/Daniel-BG/Lcplc (visited on 09/13/2019).

[21] Daniel Báscones. *Mervi*. 2018. URL: https://github.com/Daniel-BG/Mervi (visited on 06/01/2020).

[22] Daniel Báscones. *Parallel CCSDS*. 2020. URL: https://github.com/Daniel-BG/Parallel{_}CCSDS-123.0-B-1 (visited on 06/01/2020).

[23] Daniel Báscones. *Vypec*. 2018. URL: github.com/Daniel-BG/Vypec (visited on 01/25/2018).

[24] Daniel Báscones, Carlos González, and Daniel Mozos. "Hyperspectral Image Compression Using Vector Quantization, PCA and JPEG2000". In: *Remote Sensing* 10.6 (2018), p. 907. URL: http://www.mdpi.com/2072-4292/10/6/907.

[25] Daniel Báscones, Carlos González, and Daniel Mozos. "Parallel Implementation of the CCSDS 1.2.3 Standard for Hyperspectral Lossless Compression". In: *Remote Sensing* 9.10 (2017), p. 973.

[26] Patrick Billingsley. "On the coding theorem for the noiseless channel". In: *The Annals of Mathematical Statistics* 32.2 (1961), pp. 594–601.

[27] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[28] Asgeir Bjorgan and Lise Lyngsnes Randeberg. "Real-time noise removal for line-scanning hyperspectral devices using a minimum noise fraction-based approach". In: *Sensors* 15.2 (2015), pp. 3362–3378.

[29] J.W. Boardman, F.a. Kruse, and R.O. Green. "Mapping target signatures via partial unmixing of AVIRIS data". In: *Summaries of JPL Airborne Earth Science Workshop* (1995), pp. 3–6. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.26.4610{\&}rep=rep1{\&}type=pdf.

[30] X. Briottet et al. “Military applications of hyperspectral imagery”. In: *Targets and Backgrounds XII: Characterization and Representation* 6239 (2006), 62390B.

[31] M Burrows and D.J Wheeler. *A Block-sorting Lossless Data Compression Algorithm*. Tech. rep. 1994.

[32] CCSDS. “Lossless Multispectral & Hyperspectral Image Compression”. In: May (2012). URL: http://public.ccsds.org/publications/archive/123x0b1ec1.pdf.

[33] CCSDS. “Lossless multispectral and hyperspectral image compression infromational report, 120.2-G-1, Green Book”. In: December (2015).

[34] CCSDS. “Low-complexity lossless and near-lossless multispectral and hyperspectral image compression”. In: February (2019). URL: https://public.ccsds.org/Pubs/123x0b2c1.pdf.

[35] Susanna Celeste. *Daguerreotype process*. CC-BY-SA. URL: https://commons.wikimedia.org/wiki/File:Daguerreotype_process.jpg (visited on 01/17/2020).

[36] Tsung Han Chan et al. “A convex analysis-based minimum-volume enclosing simplex algorithm for hyperspectral unmixing”. In: *IEEE Transactions on Signal Processing* 57.11 (2009), pp. 4418–4432.

[37] Chein I. Chang and Antonio Plaza. “A fast iterative algorithm for implementation of pixel purity index”. In: *IEEE Geoscience and Remote Sensing Letters* 3.1 (2006), pp. 63–67.

[38] Yu Wei Chang, Hung Chi Fang, and Liang Gee Chen. “High Performance Two-Symbol Arithmetic Encoder in JPEG 2000”. In: (2000), pp. 4–7.

[39] Chirot Charitkhuan, Janjai Bhuripanyo, and Rerngwut Choomuang. “FPGA implementation of closed-loop control system for small-sized robocup”. In: *2006 IEEE Conference on Robotics, Automation and Mechatronics* (2006).

[40] Emmanuel Christophe, Corinne Mailhes, and Pierre Duhamel. “Hyperspectral image compression: adapting SPIHT and EZW to anisotropic 3-D wavelet coding”. In: *IEEE Transactions on Image processing* 17.12 (2008), pp. 2334–2346.

[41] Albert Cohen, Ingrid Daubechies, and J-C Feauveau. “Biorthogonal bases of compactly supported wavelets”. In: *Communications on pure and applied mathematics* 45.5 (1992), pp. 485–560.

[42] Marco Conoscenti, Riccardo Coppola, and Enrico Magli. “Constant SNR, Rate Control, and Entropy Coding for Predictive Lossy Hyperspectral Image Compression”. In: *IEEE Transactions on Geoscience and Remote Sensing* 54.12 (2016), pp. 7431–7441.

[43] Thomas Cooley, Gary Seigel, and Ivan Thorsos. “Hyperspectral imaging from space: Warfighter-1”. In: *Space Technology and Applications International Forum* 747.March (1999), pp. 747–752.

[44] Satellite Imaging Corporation. *Aster Mexicali*. URL: https://content.satimagingcorp.com/static/galleryimages/aster-mexi-cali.jpg (visited on 02/24/2020).

[45] Satellite Imaging Corporation. *ASTER Satellite Sensor*. URL: https://www.satimagingcorp.com/satellite-sensors/other-satellite-sensors/aster/ (visited on 02/24/2020).

[46] Satellite Imaging Corporation. *Sentinel-2A (10m) Satellite Sensor*. URL: https://www.satimagingcorp.com/satellite-sensors/other-satellite-sensors/sentinel-2a/ (visited on 05/14/2020).

[47] Satellite Imaging Corporation. *Worldview 3 Satellite Sensor*. URL: https://www.satimagingcorp.com/satellite-sensors/worldview-3/ (visited on 05/14/2020).

[48] Thomas M Cover and Joy A Thomas. “Entropy, relative entropy and mutual information”. In: *Elements of information theory* 2 (1991), pp. 1–55.

[49] Louise H Crockett et al. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[50] T. J. Cudahy et al. “The performance of the satellite-borne hyperion hyperspectral VNIR-SWIR imaging system for mineral mapping at Mount Fitton, South Australia”. In: *International Geoscience and Remote Sensing Symposium (IGARSS)* 1.C (2001), pp. 314–316.

[51] C. C. Cutler. *Differential Quantization of Communication Signals. Patent n. 2605361*. 1952.

[52] Alpha data. *Hyperspectral Imaging and Compression Solutions*. Tech. rep. September. 2018.

[53] Index DataBase. *A database for remote sensing indices*. URL: https://www.indexdatabase.de/db/s.php (visited on 02/24/2020).

[54] *DCT JPEG*. Dominio público. URL: commons.wikimedia.org/wiki/File:Dctjpeg.png.

[55] Diwan P. Ariana and Renfu Lu. “Visible/Near-Infrared Hyperspectral Transmittance Imaging for Detection of Internal Mechanical Injury in Pickling Cucumbers”. In: *ASABE Annual International Meeting*. 2006.

[56] Pier Luigi Dragotti, Giovanni Poggi, and Arturo R.P. Ragozini. “Compression of multispectral images by three-dimensional SPIHT algorithm”. In: *IEEE Transactions on Geoscience and Remote Sensing* 38.1 (2000), pp. 416–428.

[57] Qian Du and James E. Fowler. “Hyperspectral image compression using JPEG2000 and principal component analysis”. In: *IEEE Geoscience and Remote Sensing Letters* 4.2 (2007), pp. 201–205.

[58] Michael Dyer et al. “Concurrency techniques for arithmetic coding in JPEG2000”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 53.6 (2006), pp. 1203–1213.

[59] Wajdi Elhamzi et al. “An efficient low-cost FPGA implementation of a configurable motion estimation for H.264 video coding”. In: *Journal of Real-Time Image Processing* 9.1 (2014), pp. 19–30.

[60] Peter Elias. “Universal Codeword Sets and Representations of the Integers”. In: *IEEE transactions on information theory* 21.2 (1975), pp. 194–203.

[61] Gamal ElMasry et al. “Hyperspectral imaging for nondestructive determination of some quality attributes for strawberry”. In: *Journal of Food Engineering* 81.1 (2007), pp. 98–107.

[62] eoPortal. *Airborne Sensors*. URL: https://directory.eoportal.org/web/eoportal/airborne-sensors (visited on 05/14/2020).

[63] eoPortal. *EnMAP (Environmental Monitoring and Analysis Program)*. URL: https://directory.eoportal.org/web/eoportal/satellite-missions/e/enmap (visited on 05/14/2020).

[64] eoPortal. *JERS-1 (Japan Earth Resources Satellite) Fuyo-1*. URL: https://directory.eoportal.org/web/eoportal/satellite-missions/content/-/article/jers-1 (visited on 05/14/2020).

[65] Suhaib A. Fahmy, Kizheppatt Vipin, and Shanker Shreejith. "Virtualized FPGA accelerators for efficient cloud computing". In: *Proceedings - IEEE 7th International Conference on Cloud Computing Technology and Science, CloudCom 2015* (2016), pp. 430–435.

[66] Hung-chi Fang et al. "High speed memory efficient ebcot architecture for JPEG2000". In: *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS'03.* 2003.

[67] Daniel Fernandez et al. "FPGA implementation of the principal component analysis algorithm for dimensionality reduction of hyperspectral images". In: *Journal of Real-Time Image Processing* 16.5 (2016), pp. 1395–1406.

[68] Johan Fjeldtvedt, Milica Orlandic, and Tor Arne Johansen. "An efficient real-time FPGA implementation of the CCSDS-123 compression standard for hyperspectral images". In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 11.10 (2018), pp. 3841–3852.

[69] J.E. Fowler and J.T. Rucker. "3D wavelet-based compression of hyperspectral imagery". In: *Hyperspectral Data Exploitation: Theory and Applications* pp (2007), pp. 379–407.

[70] Thomas W. Fry and Scott Hauck. "SPIHT Image Compression on FPGAs". In: *Reconfigurable Computing* 15.9 (2008), pp. 565–590.

[71] Manjunath Gangadhar and Dinesh Bhatia. "FPGA based EBCOT architecture for JPEG 2000". In: *Proceedings - 2003 IEEE International Conference on Field-Programmable Technology, FPT 2003* 29 (2003), pp. 228–233.

[72] Bo Cai Gao, Kathleen B. Heidebrecht, and Alexander F.H. Goetz. "Derivation of scaled surface reflectances from AVIRIS data". In: *Remote Sensing of Environment* 44.2-3 (1993), pp. 165–178.

[73] Bo-Cai Gao et al. "Atmospheric correction algorithm for hyperspectral remote sensing of ocean color from space". In: *Applied Optics* 39.6 (2000), p. 887.

[74] Bo Cai Gao et al. "Atmospheric correction algorithms for hyperspectral remote sensing data of land and ocean". In: *Remote Sensing of Environment* 113.SUPPL. 1 (2009), S17–S24. URL: http://dx.doi.org/10.1016/j.rse.2007.12.015.

[75] A. García et al. "FPGA implementation of the hyperspectral Lossy Compression for Exomars (LCE) algorithm". In: *High-Performance Computing in Remote Sensing IV*. Vol. 9247. May. 2014, p. 924705.

[76] Aday García et al. "High level modular implementation of a lossy hyperspectral image compression algorithm on a FPGA". In: *5th Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS)*. 2013, pp. 1–4.

[77] Aman Gayasen et al. "A dual-v dd low power fpga architecture". In: *International Conference on Field Programmable Logic and Applications*. Springer. 2004, pp. 145–157.

[78] GISGeography. *Satellite Pour l'Observation de la Terre (SPOT)*. URL: https://gisgeography.com/spot-satellite-pour-observation-terre/ (visited on 02/24/2020).

[79] Solomon W Golomb. "Run-Length Encodings". In: *IEEE Transactions on Information Theory* 12.3 (1966), pp. 399–401.

[80] Christopher Gordon. "A generalization of the maximum noise fraction transform". In: *IEEE Transactions on geoscience and remote sensing* 38.1 (2000), pp. 608–610.

[81] A. A. Gowen et al. "Hyperspectral imaging - an emerging process analytical tool for food quality and safety control". In: *Trends in Food Science and Technology* 18.12 (2007), pp. 590–598.

[82] Andrew A Green et al. "A transformation for ordering multispectral data in terms of image quality with implications for noise removal". In: *IEEE Transactions on geoscience and remote sensing* 26.1 (1988), pp. 65–74.

[83] GICI Group. *Empordá software*. 2011.

[84] Jie Guo et al. "Efficient VLSI architecture of JPEG2000 encoder". In: *Proceedings of the 2013 6th International Congress on Image and Signal Processing, CISP 2013* 1.May (2013), pp. 192–197.

[85] Kaiyuan Guo et al. "A Survey of FPGA-Based Neural Network Accelerator". In: 9.4 (2017), pp. 1–26. arXiv: 1712.08934. URL: http://arxiv.org/abs/1712.08934.

[86] Amit Kumar Gupta, David S Taubman, and Saeid Nooshabadi. "High speed VLSI architecture for bit plane encoder of JPEG 2000". In: *IEEE Midwest Symposium on Circuits and Systems* (2004), pp. 233–236.

[87] Driss Haboudane et al. "Hyperspectral vegetation indices and novel algorithms for predicting green LAI of crop canopies: Modeling and validation in the context of precision agriculture". In: *Remote sensing of environment* 90.3 (2004), pp. 337–352.

[88] David L. Hall and James Llinas. "An introduction to multi-sensor data fusion". In: *Proceedings of the IEEE* 81.January (1997).

[89] Ahmed Hanafi et al. "FPGA-based secondary onboard computer system for low-earth-orbit nanosatellite". In: *Proceedings - 3rd International Conference on Advanced Technologies for Signal and Image Processing, ATSIP 2017* (2017), pp. 1–6.

[90] Alexander Hofmann et al. "Reconfigurable on-board processing for flexible satellite communication systems using FPGAs". In: *Proceedings of the 2017 Topical Workshop on Internet of Space, TWIOS 2017* (2017), pp. 9–12.

[91] Ben Hopson et al. "Real-time CCSDS lossless adaptive hyperspectral image compression on parallel GPGPU & multicore processor systems". In: *Proceedings of the 2012 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2012* (2012), pp. 107–114.

[92] Harold Hotelling. "Analysis of a complex of statistical variables into principal components." In: *Journal of educational psychology* 24.6 (1933), p. 417.

[93] Yun-Tai Hsiao et al. "High-speed memory-saving architecture for the embedded block coding in JPEG2000". In: *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on.* Vol. 5. IEEE. 2002, pp. V–V.

[94] Su May Hsu and Hsiao Hua Burke. "Multisensor fusion with hyperspectral imaging data: Detection and classification". In: *Handbook of Pattern Recognition and Computer Vision, 3rd Edition* 14.1 (2005), pp. 347–365.

[95] Bormin Huang. *Satellite Data Compression*. Springer Science & Business Media, 2011.

[96] David A Huffman. "A method for the construction of minimum-redundancy codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[97] Aapo Hyvärinen, Juha Karhunen, and Erkki Oja. *Independent component analysis*. Vol. 46. John Wiley & Sons, 2004.

[98] Intel. *Intel Core i7-7500U*. URL: https://ark.intel.com/content/www/es/es/ark/products/95451/intel-core-i7-7500u-processor-4m-cache-up-to-3-50-ghz.html (visited on 04/24/2020).

[99] I ISO. "12232: Photography-Electronic Still Picture Cameras: Determination of ISO Speed". In: *International Organization for Standardization, Geneva, Switzerland* (1997).

[100] ITU. "JPEG2000 Core part 1 -T.800 (08/2002)". In: *Itu* 800 (2002).

[101] Jagokogo. *Parowoz IR*. CC-BY-SA. URL: https://commons.wikimedia.org/wiki/File:ParowozIR.jpg (visited on 01/27/2020).

[102] Jastrow. *Aristotle*. Public Domain. URL: https://commons.wikimedia.org/wiki/File:Aristotle_Altemps_Inv8575.jpg (visited on 01/17/2020).

[103] S D Jayavathi and A Shenbagavalli. "FPGA Implementation of MQ Coder in JPEG 2000 Standard - A Review". In: 28.1 (2016), pp. 76–83.

[104] Worku Jifara et al. "Hyperspectral image compression based on online learning spectral features dictionary". In: *Multimedia Tools and Applications* 76.23 (2017), pp. 25003–25014.

[105] John. *Comparison between JPEG, JPEG2000 and JPEGXR*. GFDL. URL: commons.wikimedia.org/wiki/File:Comparison{_}between{_}JPEG,{_}JPEG{_}2000{_}and{_}JPEG{_}XR.png.

[106] Neelanchal Joshi et al. "Implementation of CCSDS Hyperspectral Image Compression Algorithm on FPGA on board a nanosatellite". In: *8th European Conference for Aeronautics and Aerospace Sciences (EUCASS)*. October. 2019, pp. 0–10.

[107] JPL. *Airborne Visible / Infrared Imaging Spectrometer*. URL: https://aviris.jpl.nasa.gov/aviris/index.html (visited on 02/18/2020).

[108] JPL. *Airborne Visible / Infrared Imaging Spectrometer - Next generation*. URL: aviris-ng.jpl.nasa.gov/ (visited on 06/18/2019).

[109] Jun Qiao et al. "Water Content and Weight Estimation for Potatoes Using Hyperspectral Imaging". In: *ASAE Annual International Meeting*. 2013.

[110] Liu Kai, Wu Chengke, and Li Yunsong. "A high-performance VLSI arquitecture of EBCOT block coding in JPEG2000". In: 23.1 (2006), pp. 1–5.

[111] Nandakishore Kambhatla and Todd K. Leen. "Fast non-linear dimension reduction". In: *Advances in neural information processing systems*. 1994, pp. 152–159.

[112] Phillip W Katz. *String searcher, and compressor using same*. 1991. URL: http://www.google.com/patents?hl=en{\&}lr={\&}vid=USPAT5051745{\&}id=mQkXAAAAEBAJ{\&}oi=fnd{\&}dq=hash+efficient+string+memory.

[113] Didier Keymeulen. *FPGA implementation of lossless and lossy compression of space-based multispectral and hyperspectral imagery*. 2016. URL: http://arc.aiaa.org/doi/pdf/10.2514/6.1984-2018.

[114] Didier Keymeulen et al. "Airborne demonstration of FPGA implementation of Fast Lossless hyperspectral data compression system". In: *Proceedings of the 2014 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2014* (2014), pp. 278–284.

[115] Didier Keymeulen et al. *High Performance Space Data Acquisition and Compression with Embedded System-on-Chip Instrument Avionics for Space-based Next Generation Imaging Spectrometers (NGIS)*. Tech. rep. 2018, pp. 1–20.

[116] a Kiely. "Selecting the Golomb Parameter in Rice Coding". In: *Analysis* 2 (2004), pp. 1–18.

[117] M. S. Kim, Y. R. Chen, and P. M. Mehl. "Hyperspectral reflectance and fluorescence imaging system for food quality and safety". In: *Transactions of the American Society of Agricultural Engineers* 44.3 (2001), pp. 721–729.

[118] Russell A. Kirsch. *Walden Kirsch*. Public Domain. URL: https://commons.wikimedia.org/wiki/File:NBSFirstScanImage.jpg (visited on 01/17/2020).

[119] M Klimesh. "Low-complexity adaptive lossless compression of hyperspectral imagery". In: *Proc. SPIE, vol. 6300, 63000N* (2005), pp. 1–10.

[120] N Kranitis et al. "High Performance CCSDS Image Compression Implementations on Space-Grade SRAM FPGAs". In: March (2016).

[121] Fred A. Kruse, Joseph W. Boardman, and Jonathan F. Huntington. "Comparison of airborne hyperspectral data and EO-1 Hyperion for mineral mapping". In: *IEEE Transactions on Geoscience and Remote Sensing* 41.6 PART I (2003), pp. 1388–1400.

[122] Moez Kthiri et al. "FPGA architecture of the LDPS motion estimation for H.264/AVC Video Coding". In: *Journal of Signal Processing Systems* 68.2 (2012), pp. 273–285.

[123] Nandini Ramesh Kumar, Wei Xiang, and Yafeng Wang. "An FPGA-based fast two-symbol processing architecture for JPEG 2000 arithmetic coding". In: *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings* I (2010), pp. 1282–1285.

[124] Nandini Ramesh Kumar, Wei Xiang, and Yafeng Wang. "Two-symbol FPGA architecture for fast arithmetic encoding in JPEG 2000". In: *Journal of Signal Processing Systems* 69.2 (2012), pp. 213–224.

[125] Tobias H. Kurz et al. "Geological outcrop modelling and interpretation using ground based hyperspectral and laser scanning data fusion". In: *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 37(B8).March 2014 (2008), pp. 1229–1234. URL: http://org.uib.no/cipr/Project/VOG/files/papers/Kurz{_}isprs2008.pdf.

[126] Heesung Kwon and Nasser M. Nasrabadi. "Kernel RX-algorithm: A nonlinear anomaly detector for hyperspectral imagery". In: *IEEE Transactions on Geoscience and Remote Sensing* 43.2 (2005), pp. 388–397.

[127] George A. Lampropoulos et al. "Hyperspectral classification fusion for classifying different military targets". In: *International Geoscience and Remote Sensing Symposium (IGARSS)* 3.1 (2008), pp. 262–265.

[128] Charis Lanaras, Emmanuel Baltsavias, and Konrad Schindler. "Hyperspectral super-resolution by coupled spectral unmixing". In: *Proceedings of the IEEE International Conference on Computer Vision* 2015 Inter (2015), pp. 3586–3594.

[129] Yijun Li and Magdy A Bayoumi. "A three level parallel high speed low power architecture for EBCOT of JPEG 2000". In: *IEEE Transactions on Circuits and Systems for Video Technology* 16.9 (2006), pp. 1153–1163.

[130] Chung-Jr Lian et al. “Analysis and architecture design of block-coding engine for EBCOT in JPEG 2000”. In: *IEEE Transactions on circuits and systems for video technology* 13.3 (2003), pp. 219–230.

[131] Shuang Liang et al. “FP-BNN: Binarized neural network on FPGA”. In: *Neurocomputing* 275 (2018), pp. 1072–1086. URL: https://doi.org/10.1016/j.neucom.2017.09.046.

[132] Kai Liu et al. “A high performance MQ encoder architecture in JPEG2000”. In: *Integration, the VLSI Journal* 43.3 (2010), pp. 305–317. URL: http://dx.doi.org/10.1016/j.vlsi.2010.01.001.

[133] Zni Liu et al. “Automated tongue segmentation in hyperspectral images for medicine”. In: *Applied Optics* 46.34 (2007), pp. 8328–8334.

[134] J. W. Lockwood et al. “Reprogrammable network packet processing on the Field Programmable Port Extender (FPX)”. In: *ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA* (2001), pp. 87–93.

[135] Giorgio Lopez, Ettore Napoli, and Antonio G.M. Strollo. “FPGA implementation of the CCSDS-123.0-B-1 lossless Hyperspectral Image compression algorithm prediction stage”. In: *2015 IEEE 6th Latin American Symposium on Circuits and Systems, LASCAS 2015 - Conference Proceedings* (2015), pp. 0–3.

[136] Guolan Lu and Baowei Fei. “Medical hyperspectral imaging: a review”. In: *Journal of Biomedical Optics* 19.1 (2014), p. 010901.

[137] Jingshan Lu et al. “Monitoring leaf potassium content using hyperspectral vegetation indices in rice leaves”. In: *Precision Agriculture* 0123456789 (2019).

[138] Renfu Lu and Yud-Ren Chen. “Hyperspectral imaging for safety inspection of food and agricultural products”. In: *Pathogen Detection and Remediation for Safe Eating* 3544 (1999), pp. 121–133.

[139] D. Manolakis et al. “Is there a best hyperspectral detection algorithm?” In: *Algorithms and Technologies for Multispectral, Hyperspectral, and Ultraspectral Imagery XV* 7334 (2009), p. 733402.

[140] Dimitris Manolakis, David Marden, and Gary a Shaw. “Hyperspectral Image Processing for Automatic Target Detection Applications”. In: *Lincoln Laboratory Journal* 14.1 (2003), pp. 79–116.

[141] Freek D. van der Meer et al. “Multi- and hyperspectral geologic remote sensing: A review”. In: *International Journal of Applied Earth Observation and Geoinformation* 14.1 (2012), pp. 112–128. URL: http://dx.doi.org/10.1016/j.jag.2011.08.002.

[142] Kuizhi Mei et al. “VLSI design of a high-speed and area-efficient JPEG2000 encoder”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 17.8 (2007), pp. 1065–1078.

[143] Jarno Mielikainen. “Lossless compression of hyperspectral images using lookup tables”. In: *IEEE Signal Processing Letters* 13.3 (2006), pp. 157–160.

[144] Giovanni Motta, Francesco Rizzo, and James A Storer. *Hyperspectral data compression*. Ed. by Giovanni Motta, Francesco Rizzo, and James A. Storer. Springer Science & Business Media, 2006, p. 417.

[145] Richard J. Murphy, Sildomar T. Monteiro, and Sven Schneider. “Evaluating classification techniques for mapping vertical geology using field-based hyperspectral sensors”. In: *IEEE Transactions on Geoscience and Remote Sensing* 50.8 (2012), pp. 3066–3080.

[146] Ahmed N, Natarajan N, and Rao K R. “Discrete Cosine Transform”. In: *IEEE transactions on Computers* January (1974), pp. 90–93.

[147] NASA. *Forest fires in Myanmar*. URL: https://modis.gsfc.nasa.gov/gallery/images/image02232020_250m.jpg (visited on 02/24/2020).

[148] NASA. *Landsat 8*. URL: https://landsat.gsfc.nasa.gov/landsat-data-continuity-mission/ (visited on 05/14/2020).

[149] NASA. *Moderate Resolution Imaging Spectrometer*. URL: https://modis.gsfc.nasa.gov/ (visited on 02/24/2020).

[150] NASA. *The Thematic Mapper*. URL: https://landsat.gsfc.nasa.gov/the-thematic-mapper/ (visited on 02/24/2020).

[151] José M P Nascimento and José M B Dias. “Vertex component analysis: A fast algorithm to unmix hyperspectral data”. In: *IEEE transactions on Geoscience and Remote Sensing* 43.4 (2005), pp. 898–910.

[152] Nopphol Noikaew and Orachat Chitsobhuk. “Dual Symbol Processing for MQ arithmetic coder in JPEG2000”. In: *Image and Signal Processing, 2008. CISP’08. Congress on*. Vol. 1. IEEE. 2008, pp. 521–524.

[153] Amos R. Omondi and Jagath C. Rajapakse. *FPGA Implementations of Neural Networks*. 2006, pp. 1–36.

[154] Milica Orlandić, Johan Fjeldtvedt, and Tor Johansen. “A Parallel FPGA Implementation of the CCSDS-123 Compression Algorithm”. In: *Remote Sensing* 11.6 (2019), p. 673.

[155] Svetlana V. Panasyuk et al. “Medical hyperspectral imaging to facilitate residual tumor identification during surgery”. In: *Cancer Biology and Therapy* 6.3 (2007), pp. 439–446.

[156] Jay Pearlman et al. “The EO-1 Hyperion Imaging Spectrometer”. In: *IEEE Aerospace Conference* (2016), pp. 1–47.

[157] Barbara Penna et al. “Progressive 3-D coding of hyperspectral images based on JPEG 2000”. In: *IEEE Geoscience and Remote Sensing Letters* 3.1 (2006), pp. 125–129.

[158] M. R. Pickering and M. J. Ryan. “Efficient spatial-spectral compression of hyperspectral data”. In: *IEEE Transactions on Geoscience and Remote Sensing* 39.7 (2001), pp. 1536–1539.

[159] Antonio Plaza and Chein-I Chang. “An improved N-FINDR algorithm in implementation”. In: *Algorithms and Technologies for Multispectral, Hyperspectral, and Ultraspectral Imagery XI* 5806 (2005), p. 298.

[160] Robi Polikar. “The Wavelet Tutorial”. In: (), pp. 1–67.

[161] Shen-En Qian. *Optical Satellite Data Compression and Implementation*. 2013. arXiv: arXiv:1011.1669v3.

[162] Majid Rabbani and Rajan Joshi. *An overview of the JPEG2000 still image compression standard*. Vol. 17. 2002, pp. 3–48. URL: papers/science{_}{_}ob{_}overview.pdf.

[163] Majid Rabbani et al. *The JPEG2000 Still-Image Compression Standard*. Tech. rep.

[164] Resmini R.G et al. “Mineral mapping with HYperspectral Digital Imagery Collection Experiment (HYDICE) sensor data at Cuprite, Nevada, U.S.A.” In: *International journal of Remote Sensing* 18.7 (1997), pp. 1553–1570.

[165] Minsoo Rhu and In Cheol Park. "Optimization of arithmetic coding for JPEG2000". In: *IEEE Transactions on Circuits and Systems for Video Technology* 20.3 (2010), pp. 446–451.

[166] DA Roberts, Y Yamaguchi, and RJP Lyon. "Comparison of various techniques for calibration of AIS data". In: *NASA STI/Recon Technical Report N* 87 (1986), pp. 21–30.

[167] Tiphaigne de la Roche. *Giphantie*. 1760.

[168] Alfonso Rodriguez et al. "Scalable hardware-based on-board processing for run-time adaptive lossless hyperspectral compression". In: *IEEE Access* 7 (2019), pp. 10644–10652.

[169] Wim Roelands. "Xilinx - 15 years of innovation". In: *Xcell* 32 (1999). URL: https://www.xilinx.com/publications/archives/xcell/Xcell32.pdf.

[170] Justin T. Rucker, James E. Fowler, and Nicolas H Nicholas H. Nicolas H Younan. "JPEG2000 coding strategies for hyperspectral data". In: *Geoscience and Remote Sensing Symposium, 2005. IGARSS'05. Proceedings. 2005 IEEE International*. Vol. 1. July. IEEE. 2005, 4–pp.

[171] Michael J. Ryan and John F. Arnold. "The lossless compression of aviris images by vector quantization". In: *IEEE Transactions on Geoscience and Remote Sensing* 35.3 (1997), pp. 546–550.

[172] Jason H. Anderson Safeen Huda, Muntasir Mallick. "Clock gating architectures for FPGA power reduction". In: 1 (2009), pp. 112–118.

[173] a. Said and W.a. Pearlman. "A new, fast, and efficient image codec based on set partitioning inhierarchical trees". In: *IEEE Transactions on Circuits and Systems for Video Technology* 6.3 (1996), pp. 1–16.

[174] Taoufik Saidani, Mohamed Atri, and Rached Tourki. "Implementation of JPEG 2000 MQ-coder". In: *International Conference on Design and Technology of Integrated Systems in Nanoscale Era, DTIS'08* (2008), pp. 1–4.

[175] S. Sanjith and R. Ganesan. "A review on hyperspectral image compression". In: *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies, ICCICCT 2014* (2014), pp. 1159–1163.

[176] Temuulen Sankey et al. "UAV lidar and hyperspectral fusion for forest monitoring in the southwestern USA". In: *Remote Sensing of Environment* 195 (2017), pp. 30–43. URL: http://dx.doi.org/10.1016/j.rse.2017.04.007.

[177] Lucana Santos, Ana Gomez, and Roberto Sarmiento. "Implementation of CCSDS Standards for Lossless Multispectral and Hyperspectral Satellite Image Compression". In: *IEEE Transactions on Aerospace and Electronic Systems* 9251.c (2019), pp. 1–1.

[178] Lucana Santos et al. "FPGA implementation of a lossy compression algorithm for hyperspectral images with a high-level synthesis tool". In: *Proceedings of the 2013 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2013*. 2013, pp. 107–114.

[179] Lucana Santos et al. "Highly-parallel gpu architecture for lossy hyperspectral image compression". In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 6.2 (2013), pp. 670–681.

[180] Lucana Santos et al. "Multispectral and Hyperspectral Lossless Compressor for Space Applications (HyLoC): A Low-Complexity FPGA Implementation of the CCSDS 123 Standard". In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 9.2 (2016), pp. 757–770.

[181] Kishor Sarawadekar and Swapna Banerjee. "An Efficient Pass-Parallel Architecture for Embedded Block Coder in JPEG 2000". In: *IEEE Transactions on Circuits and Systems* 21.6 (2011), pp. 825–836.

[182] Kishor Sarawadekar and Swapna Banerjee. "Low-cost, high-performance VLSI design of an MQ coder for JPEG 2000". In: *Signal Processing (ICSP), 2010 IEEE 10th International Conference on*. D. IEEE. 2010, pp. 397–400.

[183] Kishor Sarawadekar and Swapna Banerjee. "VLSI design of memory-efficient, high-speed baseline MQ coder for JPEG 2000". In: *Integration, the VLSI Journal* 45.1 (2012), pp. 1–8. URL: http://dx.doi.org/10.1016/j.vlsi.2011.07.004.

[184] Claude E Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* XXVII.Nº 3 (1948), pp. 379–423.

[185] Jerome M. Shapiro. "Embedded Image Coding Using Zerotrees of Wavelet Coefficients". In: *IEEE Transactions on Signal Processing* 41.12 (1993), pp. 3445–3462.

[186] Hongda Shen and W. David Pan. "Predictive lossless compression of regions of interest in hyperspectral image via Maximum Correntropy Criterion based Least Mean Square learning". In: *Proceedings - International Conference on Image Processing, ICIP* 2016-Augus (2016), pp. 2182–2186.

[187] Tiezhu Shi et al. "Estimation of arsenic in agricultural soils using hyperspectral vegetation indices of rice". In: *Journal of Hazardous Materials* 308 (2016), pp. 243–252. URL: http://dx.doi.org/10.1016/j.jhazmat.2016.01.022.

[188] Jonathon Shlens. "A tutorial on principal component analysis". In: *arXiv preprint arXiv:1404.1100* (2014).

[189] Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi. "The JPEG 2000 Still Image Compression Standard". In: *IEEE Signal Processing Magazine* 18.September (2001), pp. 36–58.

[190] KR Slocum et al. "Trafficability Analysis Engine". In: *CrossTalk, The Journal of Defense Software Engineering* (2003), pp. 28–30.

[191] Marko Slyz and Lei Zhang. "A block-based interband lossless hyperspectral image compressor". In: *Data Compression Conference Proceedings* (2005), pp. 427–436.

[192] Hayden Kwok-Hay So and Robert W . Brodersen. "Improving usability of FPGA-Based reconfigurable computers through operating system support". In: (2006).

[193] Spectir. *Free Data Samples*. URL: www.spectir.com/free-data-samples/ (visited on 01/18/2018).

[194] IEEE Spectrum. *Chip Hall of Fame: Xilinx XC2064 FPGA*. URL: https://spectrum.ieee.org/tech-history/silicon-revolution/chip-hall-of-fame-xilinx-xc2064-fpga (visited on 03/09/2020).

[195] David W.J. Stein et al. "Anomaly detection from hyperspectral imagery". In: *IEEE Signal Processing Magazine* 19.1 (2002), pp. 58–69.

[196] James A. Storer and Thomas G. Szymanski. "Data Compression via Textual Substitution". In: *Journal of the ACM (JACM)* 29.4 (1982), pp. 928–951.

[197] Nasri Sulaiman et al. "Design and Implementation of FPGA-Based Systems - A Review". In: *Australian Journal of Basic and Applied Sciences* 3.October (2009), pp. 3575–3596.

[198] Nasri Sulaiman et al. “FPGA-Based Fuzzy Logic: Design and Applications – a Review”. In: *International Journal of Engineering and Technology* 1.5 (2009), pp. 491–503.

[199] Da-wen Wen Sun. *Hyperspectral imaging for food quality analysis and control*. 2010.

[200] Hiroyuki Tanaka et al. “Implementation of bilateral control system based on acceleration control using FPGA for multi-DOF haptic endoscopic surgery robot”. In: *IEEE Transactions on Industrial Electronics* 56.3 (2009), pp. 618–627.

[201] David Taubman and Michael Marcellin. *JPEG2000 image compression fundamentals, standards and practice*. Vol. 642. Springer Science & Business Media, 2012, p. 773.

[202] Alfredo Thenkabail et al. *Hyperspectral Vegetation Indices*. October. 2011, pp. 309–328.

[203] Prasad S Thenkabail, Ronald B Smith, and Eddy De Pauw. “Hyperspectral Vegetation Indices and Their Relationships with Agricultural Crop Characteristics”. In: *Remote sensing of environment* 71 (1999), pp. 158–182. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.472.6217{\&}rep=rep1{\&}type=pdf.

[204] K. C. Tiwari, M. K. Arora, and D. Singh. “An assessment of independent component analysis for detection of military targets from hyperspectral images”. In: *International Journal of Applied Earth Observation and Geoinformation* 13.5 (2011), pp. 730–740. URL: http://dx.doi.org/10.1016/j.jag.2011.03.007.

[205] Stephen M. Trimberger. “Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology”. In: *Proceedings of the IEEE* 103.3 (2015), pp. 318–331.

[206] R Tsai. “Multiframe image restoration and registration”. In: *Advance Computer Visual and Image Processing* 1 (1984), pp. 317–339.

[207] Antonis Tsigkanos et al. “A 3.3 Gbps CCSDS 123.0-B-1 Multispectral & Hyperspectral Image Compression Hardware Accelerator on a Space-Grade SRAM FPGA”. In: *IEEE Transactions on Emerging Topics in Computing* 6750.c (2018), pp. 1–13.

[208] Tim Tuan et al. “A 90-nm low-power FPGA for battery-powered applications”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pp. 296–300.

[209] Unknown. *Standard Kilogram*. Public Domain. URL: https://commons.wikimedia.org/wiki/File:Standard_kilogram,_2.jpg (visited on 01/17/2020).

[210] Gregory K. Wallace. “The JPEG still picture compression standard”. In: *IEEE Transactions on Consumer Electronics* (1992). arXiv: arXiv:1011.1669v3.

[211] Edward Waltz and James Llinas. *Multi sensor data fusion*. 1990. URL: http://it4sec.org/bg/system/files/02.12{_}Monitor.pdf.

[212] Hongqiang Wang, S. Derin Babacan, and Khalid Sayood. “Lossless hyperspectral-image compression using context-based conditional average”. In: *IEEE Transactions on Geoscience and Remote Sensing* 45.12 (2007), pp. 4187–4193.

[213] Zhou Wang et al. “Image quality assessment: from error visibility to structural similarity”. In: *IEEE transactions on image processing* 13.4 (2004), pp. 600–612.

[214] Marcelo J. Weinberger, Gadiel Seroussi, and Guillermo Sapiro. “The LOCO-I lossless image compression algorithm: Principles and standardization into JPEG-LS”. In: *IEEE Transactions on Image Processing* 9.8 (2000), pp. 1309–1324.

[215] Terry A. Welch. “A Technique for High-Performance Data Compression”. In: *Computer* 17.6 (1984), pp. 8–19.

[216] Michael E. Winter. “N-FINDR: an algorithm for fast autonomous spectral endmember determination in hyperspectral data”. In: *Imaging Spectrometry V*. 1999.

[217] Ian H. Witten et al. “Arithmetic coding for data compression”. In: *Communications of the ACM* 30.6 (1987), pp. 520–540. URL: http://portal.acm.org/citation.cfm?doid=214762.214771.

[218] Chaoyang Wu et al. “Estimating chlorophyll content from hyperspectral vegetation indices: Modeling and validation”. In: *Agricultural and Forest Meteorology* 148.8-9 (2008), pp. 1230–1241.

[219] Jiaji Wu et al. “Lossless compression of hyperspectral imagery via clustered differential pulse code modulation with removal of local spectral outliers”. In: *IEEE Signal Processing Letters* 22.12 (2015), pp. 2194–2198.

[220] Xilinx. *7 Series FPGAs Data Sheet: Overview (DS180)*. 2010. URL: www.xilinx.com/support/documentation/data{_}sheets/ds180{_}7Series{_}Overview.pdfwww.xilinx.com.

[221] Xilinx. *ISE Design Suite 14.7*. URL: www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html.

[222] Xilinx. *Radiation-Hardened, Space-Grade Virtex-5QV Family Data Sheet: Overview*. 2018.

[223] Xilinx. *Space-Grade Virtex-4QV QPro Family Overview*. 2010.

[224] Xilinx. *Virtex-4 FPGA User Guide,* 2008.

[225] Xilinx. *Virtex-5 FPGA User Guide*. 2012. URL: http://www.xilinx.com/support/documentation/user{_}guides/ug190.pdf.

[226] Xilinx. *Vivado Design Suite, 2018.3*. URL: www.xilinx.com/products/design-tools/vivado.html.

[227] Xilinx. *XC2064/XC2018 Logic Cell Array*.

[228] Xilinx. *XC3000 Series Field Programmable Gate Arrays (XC3000A/L, XC3100A/L)*. 1998.

[229] Xilinx. *XC4000E and XC4000X Series Field Programmable Gate Arrays May*. 1999.

[230] Xilinx. *Xilinx and the Birth of the Fabless Semiconductor Industry*. 2013.

[231] Xilinx. *Xilinx develops new class of ASIC*. Tech. rep. 1985. arXiv: arXiv:1011.1669v3.

[232] Xilinx. *Xilinx Virtex-7 FPGA VC709 Connectivity Kit*. URL: https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html#hardware (visited on 03/11/2020).

[233] Xilinx Inc. *Xilinx Power Estimator*. URL: https://www.xilinx.com/products/technology/power/xpe.html.

[234] Lei Xu, Weidong Shi, and Taeweon Suh. “PFC: Privacy preserving FPGA cloud - A case study of MapReduce”. In: *IEEE International Conference on Cloud Computing, CLOUD* June (2014), pp. 280–287.

[235] Hua Yu and Jie Yang. "A direct LDA algorithm for high-dimensional data—with application to face recognition". In: *Pattern recognition* 34.10 (2001), pp. 2067–2070.

[236] Yuan Yuan, Xiangtao Zheng, and Xiaoqiang Lu. "Hyperspectral Image Superresolution by Transfer Learning". In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 10.5 (2017), pp. 1963–1974.

[237] Guido Zavattini et al. "A hyperspectral fluorescence system for 3D in vivo optical imaging". In: *Physics in Medicine and Biology* 51.8 (2006), pp. 2029–2043.

[238] Jing Zhang et al. "Evaluation of JP3D for lossy and lossless compression of hyperspectral imagery". In: *International Geoscience and Remote Sensing Symposium (IGARSS)* 4.June 2017 (2009), pp. 1–5.

[239] Lefei Zhang et al. "Compression of hyperspectral remote sensing images by tensor approach". In: *Neurocomputing* 147.1 (2015), pp. 358–363.

[240] Yi Zhen Zhang et al. "Performance analysis and architecture design for parallel EBCOT encoder of JPEG2000". In: *IEEE Transactions on Circuits and Systems for Video Technology* 17.10 (2007), pp. 1336–1346.

[241] Liu Zhi et al. "Classification of hyperspectral medical tongue images for tongue diagnosis". In: *Computerized Medical Imaging and Graphics* 31.8 (2007), pp. 672–678.

[242] J Ziv and A Lempel. "Compression of lndividual Sequences". In: *IEEE Transactions on Information Theory* 24.5 (1978), pp. 530–536.

[243] Jacob Ziv and Abraham Lempel. "A Universal Algorithm for Sequential Data Compression". In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343.

Glossary

A	Interval length register (JYPEC).
arithmetic coding	Optimal coding technique for a sequence of symbols with known probabilities.
ASIC	Application Specific Integrated Circuit.
BIL	Band interleaved by line. Ordering of hyperspectral data where frames are stored sequentially, and within frames pixels are interleaved by wavelength.
BIP	Band interleaved by pixel. Ordering of hyperspectral data where pixels are stored in sequential order.
bpppb	Bits per pixel per band.
BRAM	Block Random Access Memory in an FPGA.
BSQ	Band sequential. Ordering of hyperspectral data where bands are stored sequentially, in turn stored in raster-scan order.
c	Number of clusters for VQ (JYPEC).
C	Number of parallel cores (CCSDS).
CCSDS	Consultative Committee for Space Data Systems.
CDF	Cohen–Daubechies–Feauveau wavelet transform.
CLB	Configurable Logic Block in an FPGA.
code	Mapping between two sets of symbols.
CPU	Central Processing Unit.
C	Interval base register (JYPEC).
CxD	Context-Data pair.
D	Bith depth of a hyperspectral sample.
DCT	Discrete Cosine Transform.
δ	Prediction residual (CCSDS).
DFT	Discrete Fourier Transform.
DSP	Digital Signal Processing block in an FPGA.
e	Error in prediction (CCSDS).

EBC	Embedded Block Coding.
EBCOT	Embedded Block Coding with Optimal Truncation.
entropy	Amount of information present in the symbols of a data source.
EZW	Embedded Zerotree Wavelet.
FIFO	First In First Out queue.
FL	Fast Lossless algorithm.
FPGA	Field Programmable Gate Array.
frame	The set of pixels that span the width of an image, in turn given by the capture width of the sensor.
golomb code	A code optimal for coding a sequence which symbols follow a geometric distribution.
golomb exp code	A code optimal for coding a sequence which symbols follow an exponential distribution.
GPU	Graphics Processing Unit.
H	Shannon's entropy, the average length limit for transmitting a sequence of symbols of a fixed set.
z	Position of a sample within an image across its height.
HDL	Hardware Description Language.
HH	Subband with highpass horizontal and highpass vertical filters applied.
HL	Subband with highpass horizontal and lowpass vertical filters applied.
hyperspectral image	An image capturing many bands at equally spaced wavelength intervals.
ICA	Independent Component Analysis.
IR	Infrared, referring to a band in the IR wavelength.
ir	Inverse Compression Ratio. Value indicating the fraction of the original size that the compressed data occupies.
JPEG	Joint Photographic Experts Group.
JYPEC	Java hYPerspEctral Compressor.
KLT	Karhunen–Loève Transform.
LCPLC	Low Complexity Predictive Lossless Compression.
LH	Subband with lowpass horizontal and highpass vertical filters applied.
LL	Subband with lowpass horizontal and vertical filters applied.
LMS	Least Mean Square.
lossless	A type of compression where the data recovered after decompression exactly matches the original data before compression.
lossy	A type of compression where the data recovered after decompression can be different than the original, normally an approximation.

LPS	Least Probable State.
LUT	Look Up Table in a CLB.
maxSE	Maximum Square Error.
MS/s	Hyperspectral mega samples per second.
MNF	Minimum Noise Fraction.
monochrome image	An image capturing only one band of a specific wavelength.
MPS	Most Probable State.
MSE	Mean Square Error.
MSR	Mean to Standard deviation Ratio.
μ	Local average of neighbors in CCSDS.
multispectral image	An image capturing a small number of bands of different wavelengths.
near-lossless	A type of compression where either the loss of information or the compression ratio are limited.
NPSNR	Normalized PSNR.
N_X	Number of pixels in each frame of the image, or width.
N_Y	Number of frames of an image, or length.
N_Z	Number of samples on each pixel of the image, or height.
Ω	Weight resolution in CCSDS.
P	Number of bands used for prediction (CCSDS).
panchromatic image	An image capturing all visible light in a single band, averaging the intensity of the range of wavelengths.
PB	Push-broom, type of sensor that has a detector for each sample within a frame.
PCA	Principal Component Analysis.
pixel	The set of sample(s) at the same spatial location within an image.
PowSNR	Power Normalized SNR.
PSNR	Peak SNR.
quantization	Process through which a numeric data sample loses precision.
r	Compression Ratio. Value indicating how many times smaller the compressed data is with respect to the original.
radiance	Absolute amount of light that bounces off the target and is captured by the sensor.
raster order	Traversal of an image by lines, one after another, flattening the spatial matrix of pixels to a single vector.
reflectance	Amount of light that the target reflects relative to how much it is receiving.

RGB	Red, Green and Blue. Usually referring to a three-band image in the red, green, and blue wavelengths.
ρ	Update scaling exponent (CCSDS).
ROI	Region Of Interest.
run length coding	Coding technique that benefits from repetitive sequences of symbols.
$s_{z,y,x}$	A sample of an image at the given coordinates. Also $s_z(t)$.
sample	Concrete value within a hyperspectral image at a fixed spatial and spectral location.
S/s	Hyperspectral samples (S) per second . Used for measuring throughput.
$\hat{s}$	Prediction of a sample s.
σ	Local sum of neighbors in CCSDS.
$s_{\max}$	Minimum value of a sample within its range.
s_{mid}	Central value of a sample within its range.
$s_{\min}$	Minimum value of a sample within its range.
SNR	Signal to Noise Ratio.
spectral signature	A function mapping wavelength to intensity, referring to either pure synthetic spectra or the contents of a hyperspectral pixel.
SPIHT	Set Partitioning In Hierarchical Trees.
SSIM	Structural Similitude Index.
SVD	Singular Value Decomposition.
swath	Width of the image at the ground level, given by the sensor's characteristics.
t	Position of a sample within its band. $t = x + y \cdot N_X$.
T	The result of performing dimensionality reduction on matrix X for JYPEC.
t	Subsampling factor (JYPEC).
$\boldsymbol{U}$	Difference vector (CCSDS).
UV	Ultraviolet, referring to a band in the UV wavelength.
VCA	Vertex Component Analysis.
VQ	Vector Quantization.
VQPCA	Vector Quantization Principal Component Analysis.
$\boldsymbol{W}$	Weight vector (CCSDS).
W	Transformation matrix (JYPEC).
wavelet transform	Transformation applied to a data matrix that separates high and low frequency components of the underlying signal.
WB	Whisk-broom, type of sensor that has detectors for just one pixel, and raster scans the target area to generate an image.

$\bar{W}$	Inverse Transformation matrix (JYPEC).
x	Position of a sample within an image across its width.
X	Matrix of pixels used in JYPEC.
y	Position of a sample within an image across its length.