Implementación sobre FPGA de un algoritmo de compresión de imágenes hiperespectrales bajo el estándar CCSDS 123.0-B-1

Daniel Báscones García

Doble grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

Trabajo de Fin de Grado

17 de junio de 2016

Directores:

Carlos González Calvo
Daniel Mozos Muñoz

Agradecimientos

En este punto es donde habitualmente se coloca una lista de todos los que durante el largo recorrido que acaba aquí te han acompañado. Nunca me ha gustado la frase de *siempre se ha hecho así y funciona*, así que me arriesgaré a tomar un camino diferente. Además, de esta manera evito la vergüenza de olvidar a alguien.

Quien requiere mención ya la tiene en portada. Son las personas más importantes en este proyecto y sin las cuales no habría nacido. Muchas son las ocasiones en las que no sabes qué hacer, y encuentras a alguien que te saca de la confusión. Ese era mi caso hasta que me topé con mi introducción al mundo de la investigación. Sin aquella providencial charla aún estaría buscando lazarillos.

Toda investigación necesita además una buena financiación para salir adelante, y el sobremesa con el monitor panorámico no ha salido de mi bolsillo. Mi vetusto portátil de hace 8 años no habría sido capaz de compilar siquiera este documento. Ha hecho que no me pase incontables tardes en la facultad buscando un laboratorio libre.

Están los que en cualquier momento te mandan una chorrada por *whatsapp*, o aún mejor, *telegram*, que aunque tiene que ver con tu proyecto menos que una jarra verde con asas, te saca esas sonrisas que rompen la monotonía de trabajar en lo mismo durante meses.

También son motor quienes por orgullo propio quieren competir y enseñarte que lo suyo es mejor. Te animas a superarlos y en el proceso te superas también a ti mismo. Tanto en la oficina como en el terreno de juego la competición (siempre que sea sana) es buena.

¿Y qué es la vida sin cabreos? Todos a los que he conseguido enfadar o han conseguido enfadarme, os saludo. Sin un poco de sal se perdería el sabor de la vida.

Y por supuesto, los que te hacen entender que el dicho "los amigos se cuentan con los dedos de una mano" no es una farsa. Sabéis quienes sois, una herramienta que lo mismo vale para un roto que para un descosido. Cuando queráis nos tomamos cerveza, queso y tarta. Queda pendiente. Como dice la canción[1]:

I could be the rain in your desert sky

I could be the fire in your darkest night

I could be your sun when it's cold outside

I could be your rock when there's nowhere to hide

I could be your curse or your angel

It's all in how you love me.

[1] *How you love me*, 3LAU

Resumen

A lo largo de la historia, nuestro planeta ha atravesado numerosas y diferentes etapas. Sin embargo, desde finales del cretático no se vivía un cambio tan rápido como el actual. Y a la cabeza del cambio, nosotros, el ser humano.

De igual manera que somos la causa, debemos ser también la solución, y el análisis a gran escala de la tierra está siendo un punto de interés para la comunidad científica en los últimos años. Prueba de ello es que, cada vez con más frecuencia, se lanzan gran cantidad de satélites cuya finalidad es el análisis, mediante fotografías, de la superficie terrestre.

Una de las técnicas más versátiles para este análisis es la toma de imágenes hiperespectrales, donde no solo se captura el espectro visible, sino numerosas longitudes de onda. Suponen, eso sí, un reto tecnológico, pues los sensores consumen más energía y las imágenes más memoria, ambos recursos escasos en el espacio.

Dado que el análisis se hace en tierra firme, es importante una transmisión de datos eficaz y rápida. Por ello creemos que la compresión en tiempo real mediante FPGAs es la solución idónea, combinando un bajo consumo con una alta tasa de compresión, posibilitando el análisis ininterrumpido del astro en el que vivimos.

En este trabajo de fin de grado se ha realizado una implementación sobre FPGA, utilizando VHDL, del estándar CCSDS 123. Éste está diseñado para la compresión sin pérdida de imágenes hiperespectrales, y permite una amplia gama de configuraciones para adaptarse de manera óptima a cualquier tipo de imagen.

Se ha comprobado exitosamente la validez de la implementación comparando los resultados obtenidos con otras implementaciones (software) existentes. Las principales ventajas que presentamos aquí es que se posibilita la compresión en tiempo real, obteniendo además un rendimiento energético muy prometedor.

Estos resultados mejoran notablemente los de una implementación software del algoritmo, y permitirán la compresión de las imágenes a bordo de los satélites que las toman.

Palabras clave

Estándar CCSDS 123, FPGA, Imagen hiperespectral, Algoritmos de compresión, Satélite, VHDL, AVIRIS.

Abstract

Throughout history, our planet has endured many different climates. However, a change as fast as the current one had not been seen since the end of the cretacic era. We, as humans, are leading this change.

Being the cause, it is our duty to also be the remedy. The analysis of the earth as a whole has been a hot topic for the scientific community in the past few years. Proof of which is that, with increasing frequency, satellites are being launched whose objective is to analyze, using photography, our planet's surface.

One of the most versatile techniques for this purpose is hyperspectral imaging, where not only the visible spectrum is captured, but also lots of different wavelenghts. This presents, however, a technical challenge, given the increased power consumption of the sensors, and the increased storage needs for the captured data. Both are scarse resources up in space.

Since data processing is done back on Earth, it is of our interest a quick and effective link with the satellites. This is why we believe that real time compression using FPGAs is the ideal solution, combining both low power chips and a high compression rate, opening the possibility of an uninterrupted analysis of the planet we live in.

In this work, an implementation of the CCSDS 123 standard has been made using VHDL, aiming for its use on FPGAs. The standard's goal is to losslessly compress hyperspectral images, allowing optimal compression for any kind of image with its many parameters.

Validity of the algorithm has been tested against previous implementations, with our hardware one giving the same results as its software counterparts. The main advantage we get is the ability to compress in real time, paired with a very promising power efficiency.

These results notably improve those of the software implementation, and will allow on-board compression of images in the same satellites they are taken from.

Keywords

CCSDS 123 Standard, FPGA, Hyperspectral image, Compression algorithms, Satellite, VHDL, AVIRIS.

Índice General

Índice de figuras

Índice de tablas

Capítulo 1

Motivaciones y objetivos

La creciente necesidad de recopilar datos de todo tipo es evidente. En concreto, respecto a lo que aquí nos atañe, datos en forma de imágenes hiperespectrales que nos permitan un estudio exhaustivo de la superficie terrestre.

Para poder tomar dichas imágenes, necesitamos un satélite en órbita, y por tanto, también los medios necesarios para descargarlas a tierra firme. A fin de realizar una etapa previa de compresión para optimizar el ancho de banda se ha diseñado el estándar CCSDS 123[9].

A lo largo de este trabajo iremos viendo cómo y por qué la utilización de FPGAs para implementar este algoritmo resulta una solución idónea y eficaz. Se verá primero una introducción teórica a las entrañas del algoritmo. Pasaremos después a ver los distintos detalles de la implementación y se realizará un estudio de cómo influyen los divesos parámetros del algoritmo en la eficiencia de compresión y la ocupación y consumo de la FPGA.

El ser humano es curioso por naturaleza. Cada vez que mueve la vista hacia un lugar, se pregunta qué hay ahí, y por qué está ahí. Desde los animales más veloces, hasta los paisajes más inhóspitos, no hay nada estático. Si algo se quiere investigar en profundidad, se requiere de una instantánea que recopile la mayor información posible, y fije nuestro objetivo en la dimensión temporal.

Desde hace más de mil años [1] numerosos científicos han ido estudiando las particularidades que presentan diversas sustancias, como las sales de plata, ante la luz. Observaron que una exposición de esta sustancia a la luz producía un oscurecimiento proporcional a la cantidad y tiempo que se expusiera.

Aun así, no fue hasta 1839, cuando se diera a conocer el primer procedimiento fotográfico ideado por Louis Daguerre, el daguerrotipo [2]. Esta primitiva forma de fotografía era muy lenta, por lo que solo se empleaba para paisajes, y además producía nocivos vapores de mercurio (si bien en la época no se conocían tanto como ahora sus peligros). Pero era un comienzo que haría que se impulsara con mucha fuerza un nuevo método de hacer ciencia y arte.

Dos décadas después, en 1861, James Clerk Maxwell consigue tomar la primera fotografía en color [3]. Para ello toma tres fotografías sucesivas en blanco y negro, con la particularidad de que coloca filtros rojo, verde y azul, uno para cada una. Así, al unir posteriormente los resultados de las tres, consigue que la mezcla de los tres colores primarios de la luz de una muestra muy fiable de la realidad que capturaba.

Este método dejó insatisfechos a muchos, por su tediosa y precisa preparación. Motivados

por una manera más sencilla de tomar fotografías, serían los hermanos Lumière quienes en 1907 [4] consiguieran tomar una imagen a todo color mediante una única placa, sin necesidad de utilizar filtros, ni tener que recombinar varios colores posteriormente. Sucesivas mejoras sobre este método serían las que dictaran toda la fotografía del siglo XX, hasta que a finales del mismo, en la década de los 70, se hicieron los primeros avances hacia la fotografía digital por parte de Kodak y Steven Sasson [5]. A principios del siglo XXI esta nueva tecnología digital acabaría definitivamente con la analógica.

Pero no solo nos hemos interesado por fotografías en el espectro visible. A finales del siglo XIX, William Crookes observó que ciertos gases, al aplicarles descargas eléctricas, producían imágenes borrosas al estar cerca de placas fotográficas. Sería Wilhelm Conrad Röntgen, físico alemán, quien investigara este fenómeno más a fondo, descubriendo que los gases emitían una energía misteriosa, que llamó "Rayos X". Descubrió que eran capaces de atravesar materiales bastante gruesos, que sin embargo paraban en seco la luz visible. Observó que una de las cosas que atravesaban eran los humanos, y sería él quien tomara la primera radiografía, técnica que ha sido pilar de la medicina contemporánea. Posteriormente, Marie Curie ganaría el premio Nobel, junto a su marido y Henri Becquerel, por la investigación realizada sobre este fenómeno, cuya causa (y efectos) consiguieron explicar.

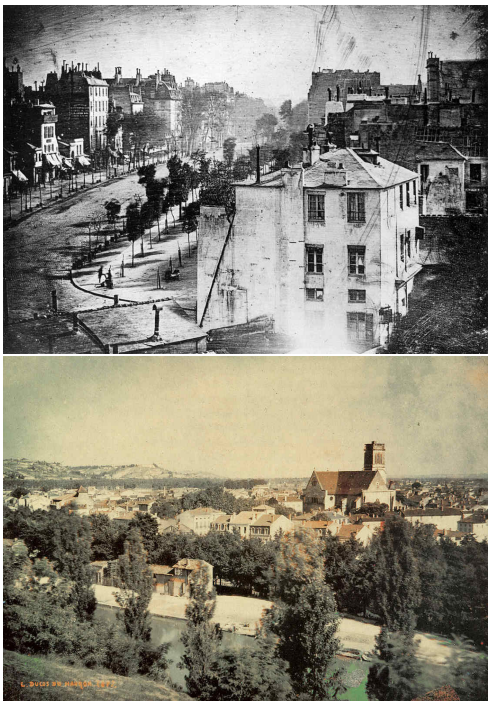


Figura 1.1: Arriba: París, *Boulevard du Temple*, 1838, Daguerre. Abajo: Una de las primeras fotografías a color. 1877, Louis Ducos du Hauron

Al otro lado del espectro, y un siglo antes, Willian Herschel descubría la radiación infrarroja, una fuente de calor más allá de la región rojiza del espectro visible, que midió con un simple termómetro al hacer pasar la luz del sol por un prisma [6]. Habría que esperar hasta la segunda guerra mundial para que la radiación infrarroja fuera de interés, cuando se empezaron a colocar sensores en los misiles para que se dirigieran automáticamente a los puntos calientes del enemigo. La primera cámara que capturase infrarrojos se haría esperar hasta 1960, cuando la empresa sueca AGA puso a la venta el primer modelo comercial [7].

No ha sido hasta las últimas dos décadas cuando se han podido unir todas estas tecnologías en lo que conocemos como imágenes hiperespectrales. De manera similar a lo que ocurriera con Maxwell, hasta ahora había que tomar las imágenes de diferentes longitudes de onda por separado. Los avances de los últimos años han permitido que un único sensor digital aglomere todos los tipos de longitudes de onda y permita, en un tiempo razonable, tomar imágenes en una nueva dimensión, añadiendo a cada píxel información sobre centenares de longitudes de onda, en lugar de restringirnos a una en concreto.

Un área de mucho interés para las imágenes hiperespectrales es la captura desde el espacio. Así permiten analizar la composición de terrenos, estudiar fugas de productos químicos, el estado de los bosques o, lo que habitualmente motiva la creación de nuevas tecnologías, descubrir bases militares camufladas.

Los algoritmos utilizados para el procesamiento de estos datos tienen una complejidad computacional muy alta: Cualquier generalización de algoritmos para imágenes bidimensionales, pasa a tener una sobrecarga del orden del número de bandas como mínimo. Por ejemplo en una

simple convolución pasaríamos de un número cuadrático de operaciones a un número cúbico, siendo cada operación también de coste cúbico respecto al tamaño de la convolución. Y para que funcionemos medianamente rápido, los datos tienen que ser accesibles con rapidez, por lo que en muchas ocasiones necesitaremos cargar la imagen entera en memoria antes de operar con ella.

Así, tanto la información a procesar, como los algoritmos que la procesan, han crecido varios órdenes de magnitud en espacio y complejidad respectivamente. Por tanto es crucial optimizar al máximo qué y dónde se procesa.

El hecho de que sea un satélite el que tome las imágenes limita bastante el almacenamiento y procesado. No es solo que no podamos mandar un supercomputador al espacio (todavía). Si pudiéramos, no tendríamos energía suficiente para alimentarlo. Los errores por rayos cósmicos serían mucho más probables, sobre todo si almacenáramos la información durante mucho tiempo. Por tanto, todo el procesamiento de las imágenes debe realizarse en tierra firme, y deben enviarse lo más rápido posible. No solo por evitar errores, sino porque el porcentaje de tiempo que un satélite apunta a antenas receptoras es limitado, y debe aprovecharlo para enviar todos los datos que toma en momentos sin cobertura.

Aunque se ha ido abaratando en los últimos años, gracias a innovaciones como los cohetes reutilizables [8], el lanzamiento de un satélite al espacio no es algo que pueda permitirse cualquiera. Años y años de planificación para asegurarse que ningún componente falle. Miles de pruebas para tener bajo control cualquier desviación del plan original. Y si algo va mal, basura espacial irreparable que orbitará la tierra hasta que acabe desintegrándose en la atmósfera.

Los avances en FPGAs han abierto una nueva posibilidad para la computación espacial. Ya no es necesario diseñar los procesadores en tierra, ahora se puede enviar silicio reconfigurable al espacio y adaptarlo a las necesidades de cada momento. Rediseñar los circuitos y actualizarlos en caso de que no funcionen adecuadamente. Y no solo de cara a solucionar problemas. Numerosos algoritmos pueden beneficiarse de determinados recursos no disponibles en un procesador genérico. Quizá solo queramos hacer un tipo de operación, pero el procesador debe llevar unidades funcionales suficientes para satisfacer cualquier necesidad, lo cual habitualmente resulta en grandes cantidades del procesador desaprovechadas. Al llevar una FPGA, podemos en cada momento adaptar no sólo el software sino el hardware disponible, a fin de optimizar al máximo la eficiencia de los algoritmos.

- ¿Sólo operas con números de 8 bits? ¿Para qué utilizar un sumador de 32, cuando uno más corto va a ser mucho más rápido?
- ¿El algoritmo no utiliza operaciones en punto flotante? ¿Para qué tener esas unidades funcionales, cuando podemos aprovechar el silicio para duplicar las de enteros?
- ¿Operaciones con vectores de longitud variable? En cada momento podemos paralelizar las mismas con el circuito óptimo.
- ¿Necesidad de gran cantidad de memoria con muy baja latencia? Podemos implementarla en el propio chip.

Pero sin duda la gran ventaja que ofrecen este tipo de sistemas es la posibilidad de cambiar los objetivos planeados inicialmente. Si en algún momento se nos ocurre una mejora o un cambio en el sistema, podemos hacerlo. No hace falta lanzar otro satélite. No hace falta idear trucos y trampas que nos permitan hacer a medias cosas que no se podían hacer en el diseño original. Podemos tener un nuevo satélite en órbita, cuando queramos, sin movernos de casa.

Vamos a ver cómo aprovechar las FPGAs bajo el estándar CCSDS 123[9] como compresoras de imágenes. El algoritmo tiene numerosas variables que ayudan a mejorar la compresión según

el tipo de sensor o imagen con la que estemos tratando. El utilizar una configuración u otra hará que necesitemos almacenar números más o menos grandes, realizar más o menos etapas, o tratar con bucles con número variable de iteraciones. Por esto y por lo mencionado anteriormente, las FPGAs son una plataforma ideal para su desarrollo ya que nos permiten una compresión a medida y siempre en tiempo real, al adaptar el hardware a las necesidades del algoritmo.

A lo largo de las siguientes páginas, iremos viendo en profundidad cómo adaptaremos nuestro algoritmo ante los diferentes escenarios, cambiando la manera de predecir, codificar, almacenar o enviar las imágenes. Pero en primer lugar, vamos a dar un repaso a las bases teóricas sobre las que han sido diseñadas estas ideas.

Capítulo 2

Análisis hiperespectral

El análisis hiperespectral se encarga de recoger y procesar información a lo largo del espectro electromagnético. De manera similar a como una fotografía recoge información de los colores rojo, verde y azul, una imagen hiperespectral (Figura 2.1) recoge el espectro de cada pixel de la imagen, tomando datos de no solo tres sino potencialmente cientos de longitudes de onda, que se suelen extender fuera del espectro visible.

Las aplicaciones van desde astronomía y agricultura (para lo que se necesitan sensores a bordo de aeronaves o satélites) hasta el campo médico o la vigilancia.

Cada material tiene una firma espectral única, y es la combinación de estas la que forma un pixel de una imagen hiperespectral. El posterior análisis de los datos puede servir, entre otros, para identificar los diferentes materiales presentes en una imagen [11].

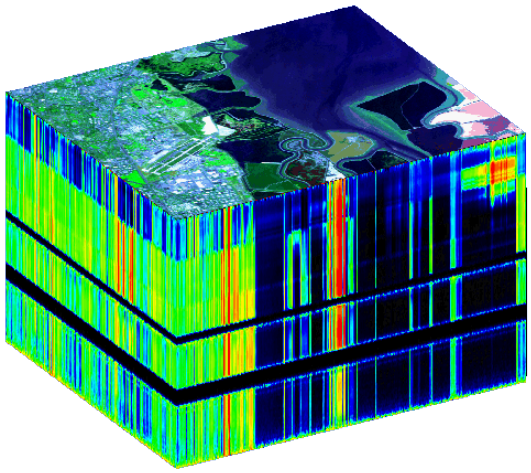


Figura 2.1: Ejemplo de imagen hiperespectral. *"image cube" JPL's AVIRIS. 20-08-1992*

La principal ventaja de las imágenes hiperespectrales es que, dado que se toma un espectro muy completo en cada punto, no hace falta tener conocimiento previo de qué se fotografía para saber sobre qué trata la imagen. Un procesamiento posterior permite extraer todos los datos necesarios, sin necesidad de guiar el proceso introduciendo datos no capturados por el sensor. Además las imágenes hiperespectrales son muy útiles de cara a la clasificación de los objetos que representan, ya que contienen más información que una imagen tradicional.

El coste y la complejidad del tratamiento de las imágenes son sus principales desventajas. Se necesitan ordenadores con mucho almacenamiento y muy rápidos para conseguir unos tiempos de proceso razonables, dado que los conjuntos de datos pueden exceder los cientos de megabytes.

Una de las mayores barreras a superar es cómo enviar la información hiperespectral tomada por satélites, bien siendo ellos mismos quienes preprocesen los datos y envíen solo lo más importante, o bien reduciendo las cantidades de información transmitida mediante el uso de algoritmos de compresión.

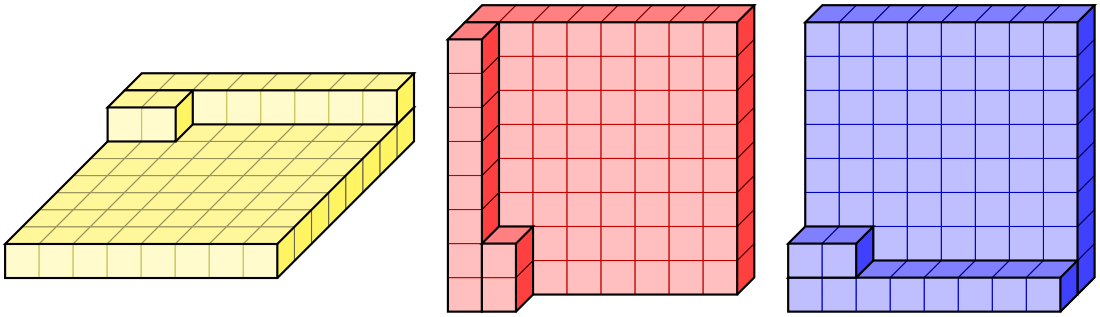


Figura 2.2: Las tres formas principales de recorrer una imagen: BSQ, BIP y BIL respectivamente.

2.1. Organización de la información

Una imagen hiperespectral añade una tercera dimensión a las imágenes tradicionales permitiendo moverse por las cantidades de luz que reflejó la escena para cada longitud de onda estudiada. Al tomar una imagen tradicional, los datos del sensor vienen habitualmente en un flujo lineal como imagen ráster [12]. Sin embargo en el caso de las imágenes hiperespectrales, los sensores tienen diversas maneras de proporcionar los datos[1]. Principalmente son tres: por bandas (BSQ), por píxeles (BIP) y por líneas (BIL). Podemos verlas en la Figura 2.2.

BSQ *Band Sequential*: La imagen se toma por bandas espectrales. Para cada longitud de onda presente en la imagen final, se toma su valor en todos los pixeles antes de pasar a la siguiente.

BIP *Band Interleaved by Pixel*: La imagen es tomada pixel por pixel, adquiriendo los datos de todas las longitudes de onda de un pixel a la vez.

BIL *Band Interleaved by Line*: La imagen se divide en líneas. Para cada línea, se toman sucesivamente las muestras de cada longitud de onda. Esto se repite para cada línea hasta tomar la imagen completa.

2.2. Funcionamiento del sensor AVIRIS

El sensor AVIRIS (*Airborne Visible / Infrared Imaging Spectrometer*) contiene 224 detectores diferentes, detectando longitudes de onda entre los $380nm$ y $2500nm$. Así pues cada muestra de un píxel está separada de las adyacentes aproximadamente unos $10nm$, lo cual le confiere una resolución espectral muy fina [13].

Es utilizado principalmente para tomar información de la superficie terrestre a bordo de aviones. La altura y velocidad del avión son factores importantes a la hora de tomar las imágenes, ya que determinan en gran medida la resolución de la imagen, referida en este caso a la cantidad de terreno que representa cada pixel.

Los 224 sensores capturan a la vez la información de un pixel concreto de la imagen. Para producir la imagen completa, se van realizando barridos sucesivos donde en cada uno se consigue una sección de la imagen de 614 pixeles por 224 bandas. El número de estas secciones que se toma suma un total de 512, formando una "escena" (Figura 2.3). Teniendo en cuenta que los sensores toman muestras en forma de enteros de 16 bits [14], la imagen completa llega a ocupar unos $614 \times 512 \times 224 \cdot 16/8 \approx 140MB$.

[1] En este trabajo, llamaremos x e y a las dimensiones de la imagen como tal, y z a la dimensión en el espectro, es decir, al número de bandas espectrales que tiene una imagen.

Figura 2.3: Imagen que obtiene el sensor AVIRIS en un barrido, resultado de unir varias *escenas*. 2016 AVIRIS Flights.

Además, el sensor no suele tomar imágenes sueltas, sino que se concatenan para formar barridos del suelo que abarcan cientos de kilómetros (teniendo en cuenta que una escena varía entre los 2 y 11km de ancho). El total al final del día puede llegar a sumar hasta 76GB de datos.

2.3. Flujo de trabajo

El algoritmo desarrollado comprimirá las imágenes hiperespectrales a fin de reducir carga en la transmisión y almacenamiento de los datos.

Para la compresión los datos vendrán secuencialmente según la especificación del sensor (secciones 2.1 y 2.2). En primer lugar un preprocesador realizará análisis de frecuencias para reducir la entropía de las muestras de entrada. Posteriormente, pasará los datos al codificador, que mediante codificación adaptativa reducirá el tamaño de los mismos sin incurrir en pérdidas de información. Este flujo se observa en la Figura 2.4

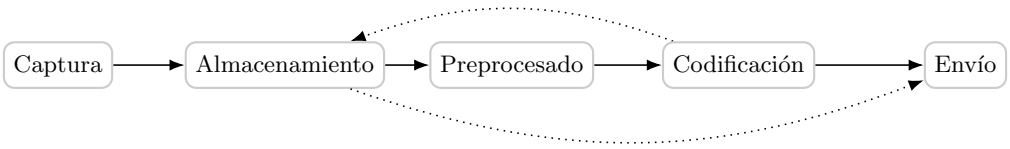


Figura 2.4: Flujo de información para la compresión de imágenes hiperespectrales

La transmisión de estos datos comprimidos será mucho más simple en cuanto a cantidad de información a tratar, pero debemos tener en cuenta que para la descompresión de una muestra, es vital tener todas las anteriores, dado que las técnicas de compresión aprovechan similitudes para reducir la información. Por tanto, pese a reducir los datos a transmitir, tenemos que tener el cuidado de introducir suficientes métodos de redundancia o comprobaciones para asegurar que no perdemos información por errores.

Capítulo 3

Hardware reconfigurable

Desde la creación de los primeros ordenadores, el diseño de circuitos ha ido mejorando progresivamente, incluyendo más transistores en menor área de silicio gracias a mejoras en técnicas fotolitográficas. Si bien la tecnología ha mejorado exponencialmente el rendimiento, la técnica ha sido siempre la misma. Basarse en puertas lógicas extremadamente simples como bloques de construcción cuya combinación hace microchips complejos con grandes prestaciones.

Pero un microchip es un circuito estático. Tras el diseño del mismo y su impresión en silicio, permanece inalterable. Es posible, obviamente, diseñarlo de tal manera que un programa ejecutándose en él realice diferentes tareas, pero si presenta alguna limitación física para ciertos objetivos, siempre va a estar presente.

En 1980 [24], se cocía una nueva arquitectura cuya importancia cada vez está más a la par con los circuitos tradicionales. Xilinx presenta la FPGA en 1985. Basándose en las ideas utilizadas por la PROM[1] y PLD[2], Xilinx diseña un dispositivo que no solo tiene puertas programables (PLD) sino que además tiene interconexiones programables entre las puertas.

Desde entonces, la cantidad de elementos programables ha ido creciendo hasta las decenas de millones en dispostivos modernos, donde se funde esta tecnología con elementos estáticos de alto rendimiento, para poder asignar circuitería muy potente a elementos críticos del sistema [25].

3.1. Estructura de una FPGA moderna

Una FPGA consiste en una matriz de bloques lógicos reprogramables, y de interconexiones (también reprogramables) entre los mismos. Tras diseñar un circuito mediante lenguajes de descripción hardware como VHDL, se puede implementar directamente en la placa, con lo que se obtiene mucho mayor rendimiento que con una simple simulación del comportamiento mediante un programa en lenguajes tradicionales como C. La ventaja de las FPGAs es que son *reprogramables*, y el circuito que implementan puede cambiarse a gusto del usuario cuándo y cómo quiera.

Por muy específico que sea un circuito, hay ciertos elementos presentes en casi todos ellos: memoria, multiplicadores, multiplexores... Si bien en un primer momento podía parecer idóneo el hacer una FPGA totalmente programable, donde cada uno de esos componentes fuera sintetizado directamente en la placa, esto limita fuertemente las capacidades del dispositivo.

Cada bloque lógico necesita una serie de componentes mínimos para que la interconexión de varios pueda generar cualquier función que nos propongamos. Muy frecuentemente, un circuito

[1] *Programmable Read Only Memory*
[2] *Programmable Logic Devices*

no necesitará todos los recursos de los bloques que utilice, pero se extenderá a lo largo de todos ellos, haciendo que su rendimiento no sea óptimo. Esto es una desventaja que tenemos que aceptar cuando estemos diseñando lógica muy específica.

Actualmente, una FPGA contiene no sólo la parte reprogramable, sino numerosos circuitos ya implementados de la manera óptima. Esto permite que las partes genéricas de un diseño sean implementadas sin el coste adicional que supondría utilizar la parte reprogramable. Por ejemplo, podemos tener cientos de MB de memoria RAM en módulos externos, mientras que en las placas más punteras sólo se podrían alcanzar cifras de decenas en caso de implementarse en la zona reprogramable [26].

3.2. Ventajas para el procesamiento a bordo

Es este maridaje de flexibilidad y rendimiento el que ha hecho de las FPGAs un recurso cada vez más apetitoso para codificación en tiempo real, computación de alto rendimiento, comunicaciones, visión artificial, encriptación y, por supuesto, aplicaciones aeroespaciales [27][28][29].

En este trabajo se presenta una aplicación dirigida a la compresión de imágenes hiperespectrales en el espacio. Una FPGA es idónea para esta aplicación porque:

- Permite la aplicación en tiempo real del algoritmo en cuestión, CCSDS 123.
- La modularidad del diseño permite detectar fallos e implementar soluciones al vuelo, cosa que con un circuito prefijado sería mucho más complicado.
- Cualquier avance que surja en el proceso puede ser incorporado o probado en una situación real, con la seguridad de que ante cualquier problema se puede revertir a estados anteriores ya probados.
- Si en algún momento cambian los objetivos de la misión, la FPGA puede adaptarse a nuevas necesidades.
- Consigue reducir costes gracias a que la reprogramación de un mismo chip puede hacerlo servir para múltiples propósitos que de otra manera tendrían que ser cubiertos por varios chips.
- Proporciona paralelismo masivo de datos y procesos gracias a la gran cantidad de I/O y de elementos reconfigurables.

Es decir, que cumpliendo con las necesidades exigidas, las FPGAs brindan además una miríada de posibilidades idóneas para el ámbito científico.

Capítulo 4

Estándar CCSDS 123

El comité asesor para sistemas de datos espaciales (*Consultative Committee for Space Data Systems* (CCSDS)) fue fundado en 1982 por las principales agencias aeroespaciales del mundo, para servir de foro de discusión de problemas comunes en el desarrollo y operación de estos sistemas. Hay más de un centenar de asociados, y entre las principales agencias colaboradoras se encuentran la NASA, la ESA, la JAXA, la RSFA[1] y muchas otras.

Desde su fundación, se encuentra desarrollando activamente recomendaciones para estándares sobre sistemas de datos e información espaciales. Con ello se pretende promover la interoperabilidad y soporte entre las agencias espaciales involucradas (tanto planeado como de contingencia), y crear nuevas oportunidades para misiones futuras [15].

Entre los estándares diseñados se encuentra el estándar CCSDS 123, que intenta paliar el problema de la transmisión de grandes cantidades de datos sobre imágenes hiperespectrales, dando pautas para la implementación de un algoritmo de compresión a utilizar antes de la transmisión de los datos.

Los potenciales beneficios que se pretenden conseguir son, entre otros:

- Reducir el ancho de banda de los canales de comunicación.
- Reducir los requerimientos sobre almacenamiento y buffering.
- Reducir el tiempo de transmisión fijado un ancho de banda.

4.1. Visión general

El estándar formaliza una versión del algoritmo FL (*Fast Lossless*) [16] de compresión sin pérdida, utilizando únicamente aritmética de enteros a fin de reducir su carga computacional simplificando las operaciones.

4.1.1. Notación

Tanto para describir el estándar como para describir los circuitos, hay un gran número de variables y términos que pueden resultar confusos. Por ello se introduce un glosario (Apéndice B) donde poder acudir ante cualquier duda. Contiene la nomenclatura y definición de todos los términos utilizados en el documento. Los términos de la base teórica, al aparecer únicamente en este capítulo, no se encuentran en el glosario.

[1] NASA: *National Aeronautics and Space Administration*, ESA:*European Space Agency*, JAXA: *Japan Aerospace Exploration Agency*, RSFA: *Russian Federal Space Agency*

4.1.2. Datos de entrada

Los datos de entrada consisten en una imagen hiperespectral. Ésta es un array tridimensional de números enteros $s_{z,y,x}$, donde x, y indican las coordenadas espaciales que podríamos encontrar en una imagen normal, y z es el índice en las bandas espectrales. Cada muestra tendrá una profundidad de entre 2 y 16 bits, y los tamaños máximos de la imagen de hasta 2^{16} se denotarán por N_X, N_Y, y N_Z. Como es habitual, los índices de la imagen en cualquier dimensión serán $(0...N-1)$. La forma de obtener estas muestras (sección 2.1) da como resultado tres ordenaciones diferentes de los datos (BSQ, BIL, BIP), todas ellas tenidas en cuenta en el estándar.

4.1.3. Datos de salida

La salida del algoritmo es un flujo de bits desde el cual se puede recuperar la imagen original sin pérdida. Debido a la naturaleza del algoritmo, el tamaño de los datos de la salida será variable aun siendo las imágenes de entrada de dimensiones idénticas. Posteriores etapas en el tratamiento de la imagen, como el envío, deberán tener en cuenta este hecho para una correcta interpretación y tratamiento de los datos de salida.

4.2. Funcionamiento del algoritmo

Los datos de entrada irán llegando al compresor de manera secuencial, según lo indicado en la sección 2.1. El compresor estima los valores de las muestras venideras utilizando modelos de predicción lineal. Las diferencias (δ) entre las predicciones y los valores reales son codificadas sin pérdida en los datos comprimidos de salida. Esta compresión predictiva es una forma de DCPM (*differential code pulse modulation*) descrita en [17]. Podemos observar el flujo en la Figura 4.1

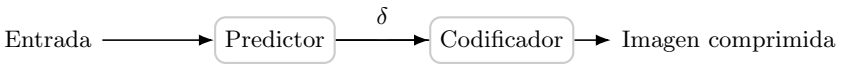


Figura 4.1: Esquema del compresor

Las predicciones se basan en un modelo adaptativo lineal que predice el valor de la siguiente muestra basado en valores de muestras cercanas en un pequeño vecindario tridimensional. El residuo de la predicción (diferencia entre la muestra predicha y verdadera) se mapea a un entero de igual número de bits que las muestras de entrada, que es la salida del predictor.

La compresión se realiza procesando sin pérdida de información los residuos δ calculados por el predictor. Parámetros internos de la compresión son ajustados según se produce para conseguir mejores resultados según las características de la imagen concreta.

4.2.1. Predictor

Describimos primero la predicción asumiendo valores de tipo real para las muestras, para luego pasar a su adaptación a valores enteros. Esta segunda versión producirá resultados lo suficientemente cercanos que ampliamente justifican la utilización de este método de menor coste computacional.

La predicción se hará en una única pasada por la imagen en cualquiera de las ordenaciones[2] posibles. Al adaptarse el predictor a nivel de banda de la imagen, las predicciones serán las mismas para cualquier ordenación, si bien, obviamente, no serán calculadas en el mismo orden.

[2]BSQ, BIL, BIP

La predicción se basa en valores vecinos, indicados en la Figura 4.2. A través de estos valores $s_{z,y,x}$ se obtendrán la muestra predicha $\hat{s}_{z,y,x}$ y el residuo de predicción $\delta_{z,y,x}$.

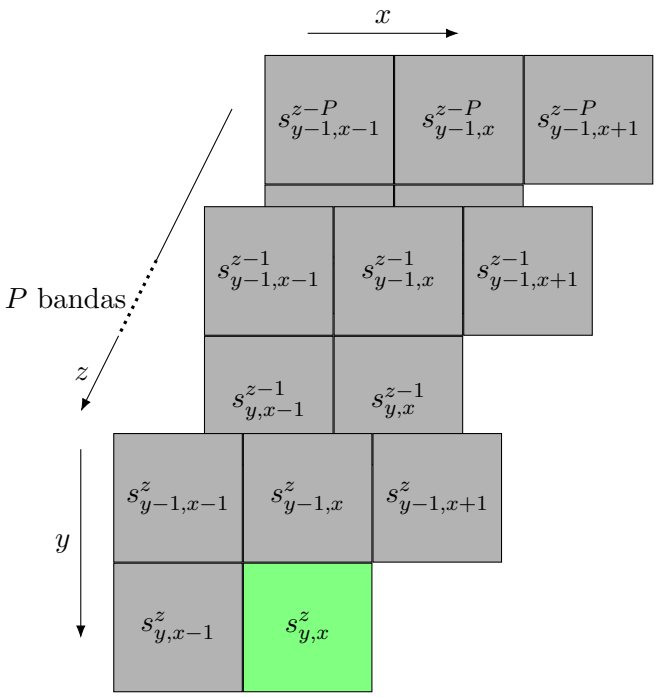


Figura 4.2: Vecindario de píxeles usado en las predicciones. $s^z_{y,x} = s_{z,y,x}$.

En primer lugar se calcula una media local $\mu_{z,y,x}$ de los valores vecinos en la misma banda. Hay dos formas de calcular esta media, utilizando la predicción *orientada a vecinos* (Ecuación (4.1)) u *orientada a columnas* (Ecuación (4.2)). Lo observamos en la Figura 4.3

$$\mu_{z,y,x} = \frac{1}{4}\left(s_{z,y,x-1} + s_{z,y-1,x-1} + s_{z,y-1,x} + s_{z,y-1,x+1}\right) \tag{4.1}$$

$$\mu_{z,y,x} = s_{z,y-1,x} \tag{4.2}$$

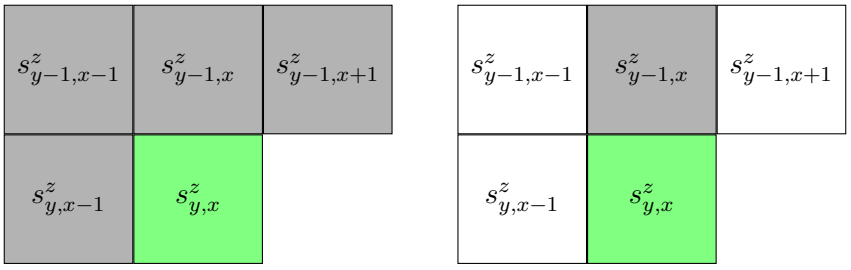


Figura 4.3: Vecindario de píxeles usado para las medias.

Este valor $\mu_{z,y,x}$ se puede interpretar como una estimación preliminar de $s_{z,y,x}$, que posteriormente será mejorada utilizando pesos ajustados adaptativamente para predecir, precisamente, la desviación de esta estimación. Utilizando estas medias y muestras, se genera el vector de diferencias dado en la Ecuación (4.3)

$$\Psi_{z,y,x} = \begin{pmatrix} s_{z,y-1,x} - \mu_{z,y,x} \\ s_{z,y,x-1} - \mu_{z,y,x} \\ s_{z,y-1,x-1} - \mu_{z,y,x} \\ s_{z-1,y,x} - \mu_{z,y,x} \\ s_{z-2,y,x} - \mu_{z,y,x} \\ \vdots \\ s_{z-P_z^*,y,x} - \mu_{P_z^*,y,x} \end{pmatrix} \tag{4.3}$$

Donde $P_z^* = min\{P, z\}$. Nótese que la Ecuación (4.3) da el vector para cuando se utiliza el *modo completo* de predicción. En el *modo reducido* las tres primeras entradas (*diferencias direccionales*) no estarían presentes. Estas diferencias direccionales son, por orden de aparición, la *norte, oeste* y *noroeste* (*N, W, NW*). Como con todos los cálculos hasta ahora, se ajustan debidamente en caso de estar en los bordes de la imagen según [9, 4].

La razón de tener dos métodos de predicción (completo y reducido) y dos formas de calcular las medias (vecinos y columna) es debida a que diferentes tipos de imágenes tendrán mejores tasas de compresión utilizando unos u otros.

El valor predicho $\hat{s}_z^*(t)$ es igual a la media local mas una suma ponderada de las diferencias vecinas:

$$\hat{s}_z^*(t) = \mu_z(t) + \boldsymbol{V}_z^T(t)\boldsymbol{\Psi}_z(t) \tag{4.4}$$

Donde $\boldsymbol{V}_z^T(t)$ es un vector de pesos de dimensión idéntica a $\boldsymbol{\Psi}_z(t)$. Existe uno por cada banda. Dichos vectores se actualizan tras cada cálculo de $\hat{s}_z^*(t)$ utilizando el algoritmo del signo, variante del algoritmo LMS *Least mean square*. Para ello, en primer lugar, se computa el error:

$$\epsilon_z(t) = s_z(t) - \hat{s}_z^*(t) \tag{4.5}$$

Que se utiliza para actualizar los pesos de la siguiente manera:

$$\boldsymbol{V}_z^T(t+1) = \boldsymbol{V}_z^T(t) + sgn(\epsilon_z(t)) \cdot 2^{-\alpha(t)} \cdot \boldsymbol{\Psi}_z(t) \tag{4.6}$$

Donde $sgn(a)$ vale 1 cuando $a \geq 0$ y -1 en caso contrario.

Así, si el valor predicho era mayor que el verdadero, el signo será negativo y los pesos decrecen. En caso contrario aumentan, en ambos casos en la magnitud de $2^{-\alpha(t)} \cdot \boldsymbol{\Psi}_z(t)$. El valor de $\alpha(t)$ controla que haya una convergencia más rápida, o que se toleren mejor los valores atípicos. Comienza en un valor inicial dado por el usuario que aumentará hasta llegar a un límite. Cuanto menor es, más rápido convergen las estimaciones, pues la magnitud en la que se cambian los pesos aumenta.

4.2.1.1. Fundamento matemático

En el algoritmo LMS [18] se utiliza el método del mayor descenso para encontrar los pesos que minimizan una función coste objetivo [19] [20]. En el caso de LMS la función coste es el cuadrado de la señal del error (de donde toma el nombre):

$$C_z(t) = E\left\{|\epsilon_z(t)|^2\right\} \tag{4.7}$$

Donde $E\{\cdot\}$ indica el valor esperado. Sin embargo, optimizar la Ecuación (4.7) nos lleva a terminar utilizando en la actualización de pesos la Ecuación (4.8):

$$\boldsymbol{V}_z^T(t+1) = \boldsymbol{V}_z^T(t) + \epsilon_z(t) \cdot 2^{-\alpha(t)} \cdot \boldsymbol{\Psi}_z(t) \tag{4.8}$$

Que, teniendo mejor convergencia que la Ecuación (4.6), tiene dos multiplicaciones, lo cual eleva el coste computacional bastante cuando hablamos de aplicaciones en tiempo real. Por tanto, para llegar a la Ecuación (4.6), partimos de la siguiente función de error:

$$C_z(t) = E\left\{|\epsilon_z(t)|\right\} \tag{4.9}$$

Para aplicar el método del mayor descenso tomamos derivadas parciales respecto a las entradas del vector de pesos:

$$\begin{aligned}
\nabla_{\boldsymbol{V}_z^T} C_z(t) &= \nabla_{\boldsymbol{V}_z^T} E\left\{|\epsilon_z(t)|\right\} = E\left\{\nabla_{\boldsymbol{V}_z^T}(\epsilon_z(t)) \cdot sgn(\epsilon_z(t))\right\} && (4.10)\\
\nabla_{\boldsymbol{V}_z^T}(\epsilon_z(t)) &= \nabla_{\boldsymbol{V}_z^T}(s_z(t) - \hat{s}_z^*(t)) && (4.11)\\
&= \nabla_{\boldsymbol{V}_z^T}(s_z(t) - \mu_z(t) - \boldsymbol{V}_z^T(t)\boldsymbol{\Psi}_z(t)) = -\boldsymbol{\Psi}_z(t) && (4.12)
\end{aligned}$$

Así, obtenemos el resultado:

$$\nabla C_z(t) = -E\left\{\boldsymbol{\Psi}_z(t) \cdot sgn(\epsilon_z(t))\right\} \tag{4.13}$$

Donde $\nabla C_z(t)$ es un vector que apunta hacia la mayor pendiente de la función coste en el punto (z, y, x). Para encontrar el mínimo del peso, debemos pues restarlo:

$$\boldsymbol{V}_z^T(t+1) = \boldsymbol{V}_z^T(t) - \alpha(t) \cdot \nabla C_z(t) = \boldsymbol{V}_z^T(t) + \alpha(t) \cdot E\left\{\boldsymbol{\Psi}_z(t) \cdot sgn(\epsilon_z(t))\right\} \tag{4.14}$$

Donde $\alpha(t)$ es el paso, que será elegido por el usuario. Ahora bien, para este cálculo necesitaríamos conocer el valor de $E\left\{\boldsymbol{\Psi}_z(t) \cdot sgn(\epsilon_z(t))\right\}$. En su lugar, podemos utilizar el estimador no sesgado:

$$\hat{E}\left\{\boldsymbol{\Psi}_z(t) \cdot sgn(\epsilon_z(t))\right\} = \frac{1}{N}\sum_{i=0}^{N-1} \boldsymbol{\Psi}_z(t-i) \cdot sgn(\epsilon_z(t-i)) \tag{4.15}$$

Tomando $N = 1$ en la Ecuación (4.15) y uniéndolo con la Ecuación (4.14), obtenemos la Ecuación (4.6) que buscábamos.

4.2.1.2. Adaptación a números enteros

El algoritmo descrito está preparado para funcionar sobre números reales. La aritmética de números reales es mucho más compleja y costosa en tiempo de cara a implementarse en hardware. Además, las muestras recibidas desde los sensores son números enteros que habría que transformar a reales, aumentando aún más el coste. Son estas dos de las razones por las que se decide adaptar el algoritmo a números enteros [10]:

- La media $\mu_{z,y,x}$ desaparece, y en su lugar se emplea la *suma local* $\sigma_{z,y,x} = 4\mu_{z,y,x}$, que siempre será valor entero.
- Las diferencias también son escaladas en un factor de 4 para producir valores enteros $\boldsymbol{U}_{z,y,x} = 4\boldsymbol{\Psi}_{z,y,x}$.

- El vector de pesos se escala en 2^{Ω} y se redondea y ajusta a $\Omega + 3$ bits, lo cual hace que $\boldsymbol{W}_z(t) \approx 2^{\Omega}\boldsymbol{V}_z(t)$. El ajuste es equivalente en términos reales a restringir los pesos al intervalo $[-4, 4]$

- El valor $\alpha(t)$ es sustitudo por $\rho(t)$ [9, p. 26] que no es igual pero sirve el mismo propósito.

- Se computa un valor de muestra predicho escalado aproximado al doble del valor predicho [10, p. 25].

- Se calcula el error *escalado*, aproximadamente el doble del error de predicción [10, p. 25].

4.2.1.3. Mapeo final

Tras hacer todos los cálculos, el residuo de predicción $\Delta_z(t) = s_z(t) + \hat{s}_z(t)$ se tiene que pasar al codificador. Pero éste sólo funciona con entradas de números enteros sin signo, por tanto se realiza un mapeo invertible $\delta : \left[-2^{D-1}, 2^{D-1} - 1\right] \rightarrow \left[0, 2^D - 1\right]$. Cualquiera puede valer, pero lo ideal sería que se consiguieran valores de salida lo más pequeños posibles. En [10, p. 27] se propone y explica uno de estos mapeos.

4.2.2. Codificador

Tras la etapa de predicción, los residuos son enviados al codificador. Aquí se utiliza una variante del LOCO-I [22] para la compresión, siendo los resultados codificados con el método de Golomb de potencias de dos (GPO2)[21].

Cada residuo es codificado utilizando códigos binarios de longitud variable. Estos códigos son seleccionados al vuelo basándose en estadísticas sobre los residuos procesados anteriormente. Las estadísticas se mantienen por separado para cada banda espectral, haciendo que, sin importar el orden de entrada (ver sección 2.1), el tamaño de compresión resultante sea el mismo.

La longitud de cada dato codificado está acotada superiormente por un parámetro dado por el usuario, U_{max}, que permite una implementación más sencilla y reduce el coste de codificar valores atípicos[3]. Se hace pues un mapeo de los residuos sin signo a códigos binarios prefijos[4].

Para codificar un entero sin signo δ mediante el GPO2, se formula de la siguiente forma:

$$\delta = u * 2^k + r \tag{4.16}$$

Donde u y r son cociente y resto, respectivamente al dividir entre 2^k. El código resultante consistirá en la representación unaria de u, seguida de un cero (para cumplir la propiedad del prefijo), seguida de r en binario. En caso de que u sea mayor o igual al umbral U_{max}, se codifican U_{max} ceros seguidos de δ en binario. Este límite se impone por comodidad algorítmica.

En el codificador se cuenta con un contador $\Gamma(t)$ y un acumulador $\Sigma_z(t)$ dependiente de la banda. El cociente $\Sigma_z(t)/\Gamma(t)$ estima $\delta_z(t)$, y se utiliza para elegir el parámetro k que da el cociente, siguiendo la Ecuación (4.17):

$$2^K \leq \frac{\Sigma_z(t)}{\Gamma(t)} + \frac{49}{128} \tag{4.17}$$

[3]Un valor es considerado atípico cuando el cociente [21] resultante en el código de Golomb supera el umbral U_{max}

[4]Un código tiene la propiedad del prefijo cuando ninguna palabra del código es prefijo de otra

4.2.2.1. Fundamento matemático

La justificación de elegir estos parámetros se encuentra en [23]. El contador $\Gamma(t)$ llevará en cada momento la cuenta de cuántas muestras van acumuladas en $\Sigma_z(t)$, y $\Sigma_z(t)$ una aproximación de la suma de todas las muestras anteriores, dando un peso exponencialmente mayor a los valores más cercanos. Hay que notar que ambos valores serán inicializados según parámetros del usuario para comenzar la codificación sin variaciones abruptas en las salidas iniciales.

Podemos hacer unos cálculos previos que nos ayudarán tanto a simplificar como a entender el propósito de estas variables. Damos las definiciones de $\Sigma_z(t)$ y $\Gamma(t)$ en las Ecuaciones (4.18) y (4.19).

$$\Sigma_z(t) = \begin{cases} \Sigma_z(t-1) + \delta_z(t-1) & \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \left\lceil \dfrac{\Sigma_z(t-1) + \delta_z(t-1)}{2} \right\rceil & \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \tag{4.18}$$

$$\Gamma(t) = \begin{cases} \Gamma(t-1) + 1 & \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \left\lceil \dfrac{\Gamma(t-1)}{2} \right\rceil & \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \tag{4.19}$$

En primer lugar, observamos que los valores que toma el contador $\Gamma(t)$ son, desde $t = 1$:

$$2^{\gamma_0}, \ldots, 2^{\gamma^*} - 1, 2^{\gamma^*-1}, \ldots, 2^{\gamma^*} - 1, 2^{\gamma^*-1}, \ldots, 2^{\gamma^*} - 1, \ldots \tag{4.20}$$

Podemos deducir una fórmula directa para el contador, uniendo las Ecuaciones (4.19) y (4.20), que vemos en la la Ecuación (4.21).

$$\begin{array}{rcl} \Gamma_0 & = & 2^{\gamma^*} - 2^{\gamma_0} \\ \Gamma_a(t) & = & t - (2^{\gamma^*} - 2^{\gamma_0} + 1) \bmod 2^{\gamma^*-1} \\ \Gamma(t) & = & \begin{cases} 2^{\gamma_0} - 1 + t & t \leq \Gamma_0 \\ 2^{\gamma^*-1} + \Gamma_a(t) & t > \Gamma_0 \end{cases} \end{array} \tag{4.21}$$

De manera análoga deducimos una fórmula para el número $\eta(t)$ de veces que hemos dividido entre dos:

$$\eta(t) = \begin{cases} 0 & t < 2^{\gamma^*} - 2^{\gamma_0} + 1 \\ \left\lfloor \dfrac{t - \left(2^{\gamma^*} - 2^{\gamma_0} + 1\right)}{2^{\gamma^*-1}} \right\rfloor & c.c. \end{cases} \tag{4.22}$$

Y obtenemos que, denotando $2^{\gamma^*} - 2^{\gamma_0} = \ell$ el acumulador sigue la fórmula:

$$\begin{array}{rcl} \Sigma_z^Z(t) & = & \displaystyle\sum_{i=1}^{\min(t,\ell)} \delta_z(i) \\ \Sigma_z^i(t) & = & \displaystyle\sum_{j=1}^{2^{\gamma^*-1}} \delta_z(i \cdot 2^{\gamma^*-1} + j + \ell) \\ \Sigma_z^N(t) & = & \displaystyle\sum_{i=0}^{(t-\ell-1) \bmod 2^{\gamma^*-1}} \delta_z(i + 1 + \ell + (\eta(t) - 1) \cdot 2^{\gamma^*-1}) \\ \Sigma_z(t) & = & \dfrac{\Sigma_z^Z(t)}{2^{\eta(t)}} + \displaystyle\sum_{i=0}^{\eta(t)-2} \frac{\Sigma_z^i(t)}{2^{\eta(t)-i-1}} + \Sigma_z^N(t) \end{array} \tag{4.23}$$

Lo cual evidencia que los valores más recientes tienen un peso *exponencialmente* mayor que los antiguos. Las razones por las que se utiliza este método de reajuste son discutidas en [22, 3.3.2, 3.4].

Capítulo 5

Implementación FPGA

En este capítulo, pasamos a dar una explicación detallada de la implementación del estándar en FPGA. Posibles cambios o mejoras serán discutidos en las secciones 6.2.8 y 6.3.

El diseño aquí explicado contiene una alta modularidad para permitir la operación del estándar en cualquiera de los modos sugeridos, y así poder realizar comparativas de espacio ocupado y tiempos de reloj para las diferentes configuraciones. En los módulos dependientes de la configuración, se especificará cuándo y por qué son o no sintetizados, y las ventajas que presenta el módulo concreto para esa configuración del algoritmo.

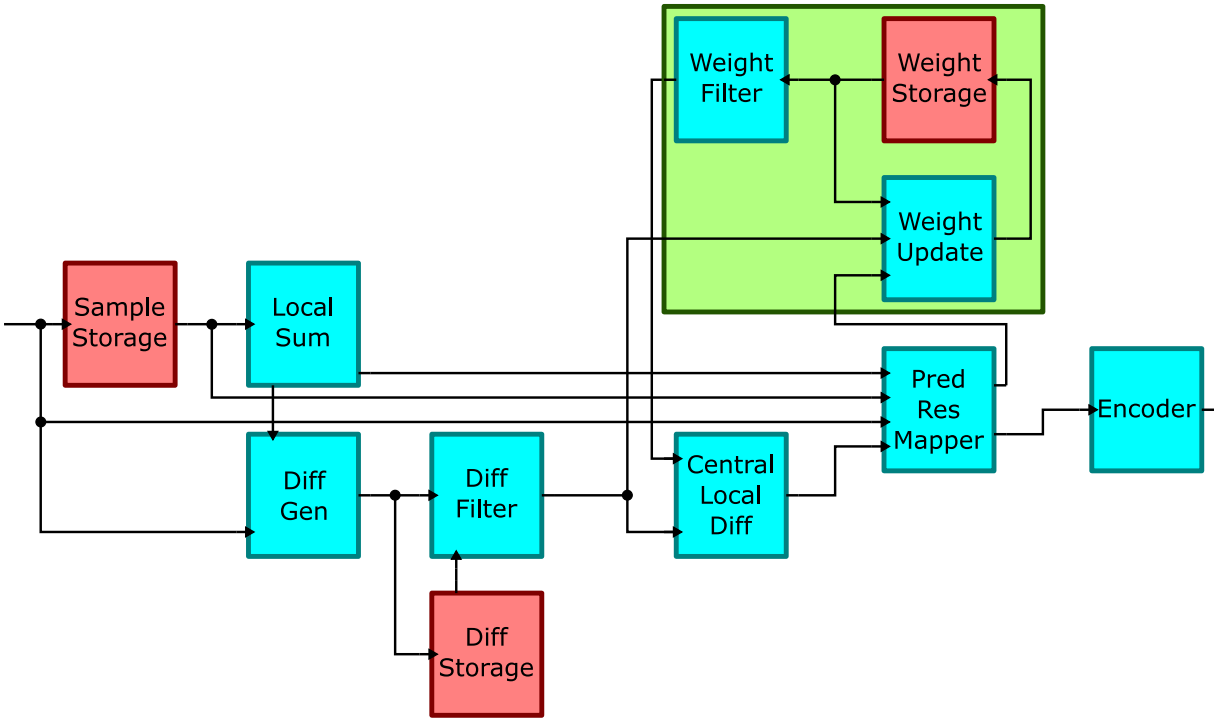


Figura 5.1: Disposición general del circuito. En azul los módulos combinacionales, en rojo los secuenciales.

El algoritmo se implementa en dos grandes bloques, el predictor (sección 4.2.1) y el codificador (sección 4.2.2). Además, se utilizan módulos auxiliares que implementan funciones compartidas entre ambos, como un contador, un generador de *flags*, y elementos de entrada/salida. El esquema general del algoritmo puede verse en la Figura 5.1.

5.1. Módulos

5.1.1. Predictor

El predictor es el encargado de transformar los valores de entrada $s_{z,y,x}$, en valores $\delta_{z,y,x}$ que le serán enviados al codificador. Se implementa a continuación el algoritmo descrito en [9].

5.1.1.1. Suma local

En primer lugar se calcula una suma local $\sigma_{z,y,x}$ a partir de los valores de entrada cercanos, dependiendo de si se utilizan sumas orientadas a vecinos o a columna. En el primer caso:

$$\sigma_{z,y,x} = \begin{cases} s_{z,y,x-1} + s_{z,y-1,x-1} + s_{z,y-1,x+1} + s_{z,y-1,x+1} & y > 0, 0 < x < N_X - 1 \\ 4s_{z,y,x-1} & y = 0, x > 0 \\ 2(s_{z,y-1,x} + s_{z,y-1,x+1}) & y > 0, x = 0 \\ s_{z,y,x-1} + s_{z,y-1,x-1} + 2s_{z,y-1,x} & y > 0, x = N_X - 1 \end{cases} \tag{5.1}$$

Dado que las muestras $s_{x,y,z}$ son enteros sin signo, las multiplicaciones por potencias de dos se realizan concatenando ceros a la derecha de los valores. Para evitar el envío de las coordenadas enteras, las comprobaciones de límite llegan ya hechas al módulo, de manera que no implementamos ningún comparador localmente. La operación más costosa (caso $y > 0, 0 < x < N_X - 1$) se realiza mediante una reducción paralelizando sumas, a fin de tener sólo dos líneas de sumadores en lugar de tres. La implementación hardware toma la forma:

```
σ_{z,y,x} <=
    s_{z,y,x-1} & "00" when y_0 else
    (s_{z,y-1,x} + s_{z,y-1,x+1}) & "0" when x_0 else
    s_{z,y,x-1} + s_{z,y-1,x-1} + (s_{z,y-1,x} & "0") & "0" when x^h else
    (s_{z,y,x-1} + s_{z,y-1,x-1}) + (s_{z,y-1,x} + s_{z,y-1,x+1})
```

En el caso de sumas orientadas a columna, tenemos:

$$\sigma_{z,y,x} = \begin{cases} 4s_{z,y-1,x} & y > 0 \\ 4s_{z,y,x-1} & y = 0, x > 0 \end{cases} \tag{5.2}$$

Que, aplicando las mismas ideas, transformamos a:

```
σ_{z,y,x} <=
    s_{z,y,x-1} & "00" when y_0 else
    s_{z,y-1,x} & "00"
```

Observamos que en ambos casos $\sigma_{0,0,0}$ no está definido en la ecuación matemática, pero sí toma valor en la implementación hardware, que en ocasiones podría ser no válido. Si $x = y = z = 0$, se toma el valor `s_{z,y,x-1} & "00"` en ambos casos. Esto no es un problema ya que, aunque el valor de entrada no esté definido, posteriormente no se verá involucrado en más cálculos. Así podemos simplificar los circuitos en lugar de darle valores por defecto en casos extremos.

5.1.1.2. Almacenamiento de muestras

En nuestro algoritmo, se asume que las muestras, que llegan en un orden determinado, lo hacen sólo una vez. Además hemos visto en la sección 5.1.1.1 la necesidad de utilizar muestras

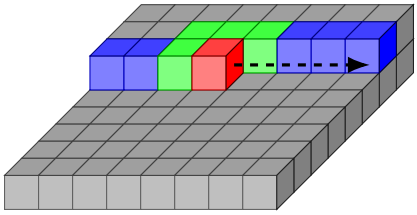


Figura 5.2: Recorrido BSQ y las muestras necesarias (verde) y almacenadas (azul) para la actual (rojo). En gris las que se han procesado y ya no son necesarias.

anteriores (siempre de la misma banda). Estos dos hechos nos fuerzan a implementar algún tipo de almacenamiento donde guardar las muestras que serán necesarias en un futuro.

Las muestras vienen de tres formas diferentes, (ver sección 2.1). Esto, unido a que tenemos dos maneras de hacer la suma, nos lleva a pensar en un principio en la necesidad de implementar seis módulos diferentes, cada uno optimizado para guardar el mínimo número posible de muestras de acuerdo a su configuración específica.

Sin embargo, esto no será necesario. Vamos a definir primero el almacenamiento para la suma orientada a vecinos, para después comprobar que podemos reaprovechar esos módulos y así ahorrar implementaciones. Las muestras necesarias serán pues $s_{z,y,x-1}, s_{z,y-1,x-1}, s_{z,y-1,x}, s_{z,y-1,x+1}$.

- En el modo BSQ, el recorrido se realiza utilizando un orden raster para cada banda. Se sigue el bucle:

  ```
  for z in 0 to N_Z - 1 loop
      for x in 0 to N_X - 1 loop
          for y in 0 to N_Y - 1 loop
  ```

  Por tanto las muestras $s_{z,y,x-1}$, $s_{z,y-1,x-1}$, $s_{z,y-1,x}$, $s_{z,y-1,x+1}$ han sido recibidas, respectivamente, $1, N_X + 1, N_X, N_X - 1$ ciclos atrás, como muestra la Figura 5.2. Bastará pues con tener espacio suficiente para $N_X + 1$ muestras anteriores. Dado que no es una cantidad muy grande, no merece la pena llevarla a un bloque de RAM, lo que aparentemente nos deja con la opción de implementarla como memoria RAM distribuida sobre la propia FPGA. Pero podemos mejorar aún más los tiempos si lo implementamos como una cola FIFO en un registro de desplazamiento, donde el último valor ($s_{z,y,x-1}$), y los tres primeros ($s_{z,y-1,x-1}, s_{z,y-1,x}, s_{z,y-1,x+1}$) serán los únicos que necesitemos acceder. Esto tiene la ventaja de tener una huella mucho menor sobre la FPGA, y de reducir considerablemente el ciclo de reloj (sección 6.2.6).

- En el modo BIP, se recorren las muestras pixel a pixel, como dicta el bucle:

  ```
  for x in 0 to N_X - 1 loop
      for y in 0 to N_Y - 1 loop
          for z in 0 to N_Z - 1 loop
  ```

  Ahora son necesarias muchas más muestras para almacenar, ya que $s_{z,y,x-1}$, $s_{z,y-1,x-1}$, $s_{z,y-1,x}$, $s_{z,y-1,x+1}$ han sido recibidas, respectivamente, $N_Z, N_Z * (N_X + 1), N_Z * N_X, N_Z * (N_X - 1)$ ciclos atrás, como muestra la Figura 5.3.

  Es precisamente fijándonos en esa Figura 5.3 donde vemos que este caso no es tan trivial como el anterior. Si guardáramos todas las muestras en una memoria, necesitaríamos leer de cuatro en cuatro. La única manera de implementarlo con los recursos de la placa sería cuadruplicando la información en cuatro lugares diferentes, lo cual es un desperdicio de espacio considerable. Para evitar esto, se pueden implementar tres memorias de manera

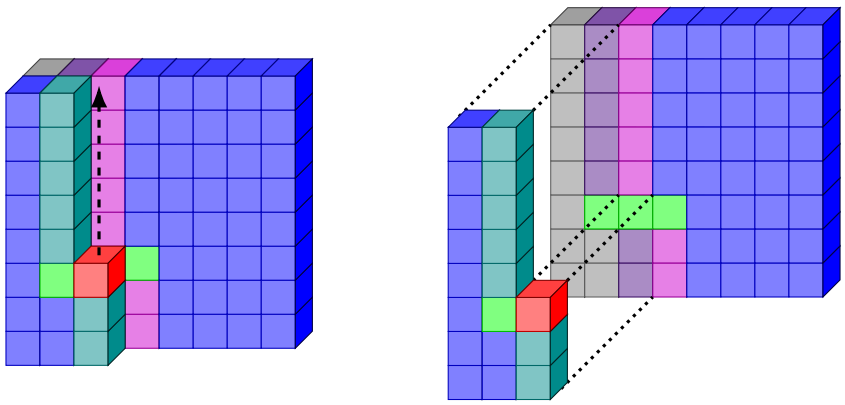


Figura 5.3: Recorrido BIP. En rojo la muestra actual. Verde las necesarias. Azul oscuro el almacenamiento principal. El resto de colores indican secciones más pequeñas de memoria.

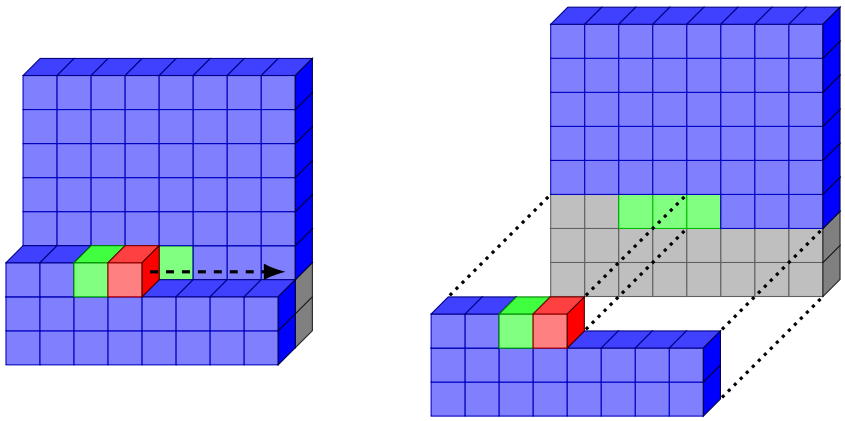


Figura 5.4: Método de almacenamiento de muestras para recorrido BIL. Mismo esquema de colores que en la Figura 5.2

análoga al BSQ, con una capacidad de N_Z muestras cada una. Esto hace que ya no necesitemos cuatro memorias, sino una sola, de capacidad $N_Z * (N_X - 2)$. Esta memoria ya tiene un tamaño considerable, y como veremos, al no haber excesivas restricciones en el tiempo de reloj, se puede implementar en bloques de RAM. No obstante, ante un tiempo de reloj más estricto podría forzarse su síntesis como memoria distribuida o como registro de desplazamiento.

- Por último, en el modo BIL, el recorrido se realiza siguiendo el bucle:

```
for x in 0 to N_X - 1 loop
    for z in 0 to N_Z - 1 loop
        for y in 0 to N_Y - 1 loop
```

Vuelven a ser necesarias, como en BIP, del orden de exactamente $N_Z * N_X + 1$ muestras previas. En este caso, al contrario que ocurría en BIL, no necesitamos una memoria tan compleja. Bastará almacenar temporalmente la muestra inmediatamente anterior ($s_{z,y,x-1}$), para luego meterla a una memoria que, $N_Z * N_X - 2$ ciclos después la extraiga a $s_{z,y-1,x+1}$. Otros dos registros temporales concatenados a la salida serán los encargados de almacenar $s_{z,y-1,x}$ y $s_{z,y-1,x-1}$. Todo queda más claro en la Figura 5.4

Ahora ya queda más claro por qué podemos reaprovechar estos almacenamientos para el modo de suma orientada a columnas:

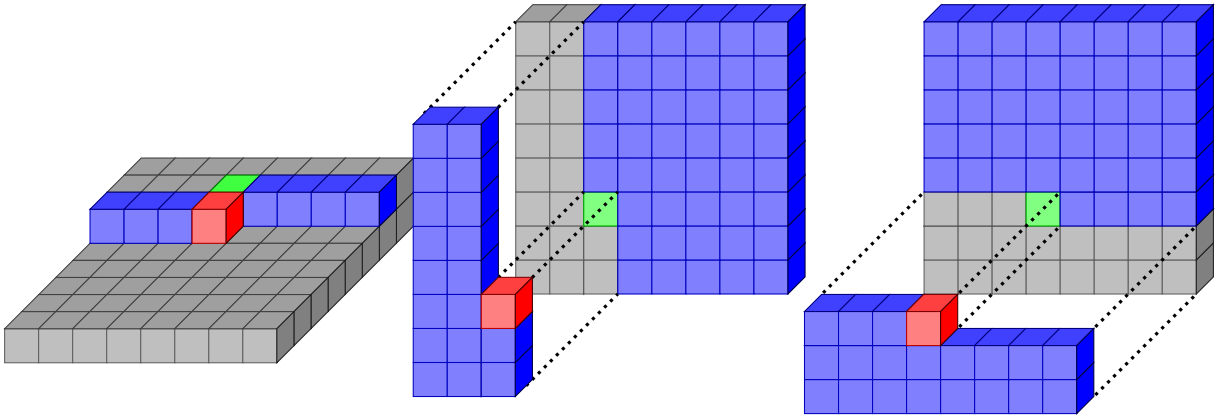


Figura 5.5: Método de almacenamiento de muestras para suma orientada a columnas. Respectivamente BSQ, BIP, BIL.

- En BSQ, el sistema es el mismo que en la Figura 5.2, ignorando la última muestra guardada (que correspondería a $s_{z,y-1,x-1}$). Al hacer la síntesis del módulo, como el puerto quedará desconectado, el propio sintetizador se encargará de eliminar los registros que guardaban esa muestra, con lo que habremos utilizado el mínimo espacio posible.
- En BIP, se eliminaría toda la columna de memoria correspondiente a la muestra $s_{z,y-1,x-1}$ (color morado en Figura 5.3). De nuevo, el sintetizador se encargará de hacerlo por nosotros.
- En BIL nos sobra, al igual que en BSQ, un único registro. De nuevo no es ningún problema dejarle el trabajo al sintetizador.

Podemos ver cómo quedaría la memoria en los tres casos en la Figura 5.5.

5.1.1.3. Diferencias

Tras hacer la suma local, esta se emplea para calcular las diferencias locales. Existe una diferencia central siempre presente, y tres diferencias direccionales que aparecen cuando el algoritmo se configura para utilizar el modo de predicción completa. Se definen como:

$$d_{z,y,x} = 4s_{z,y,x} - \sigma_{z,y,x} \tag{5.3}$$

$$\begin{aligned}
d^N_{z,y,x} &= \begin{cases} 4s_{z,y-1,x} - \sigma_{z,y,x} & y > 0 \\ 0 & y = 0 \end{cases} \\
d^W_{z,y,x} &= \begin{cases} 4s_{z,y,x-1} - \sigma_{z,y,x} & x > 0, y > 0 \\ 4s_{z,y-1,x} - \sigma_{z,y,x} & x = 0, y > 0 \\ 0 & y = 0 \end{cases} \\
d^{NW}_{z,y,x} &= \begin{cases} 4s_{z,y-1,x-1} - \sigma_{z,y,x} & x > 0, y > 0 \\ 4s_{z,y-1,x} - \sigma_{z,y,x} & x = 0, y > 0 \\ 0 & y = 0 \end{cases}
\end{aligned} \tag{5.4}$$

Al implementarlo, nos queda algo de la forma:

```
d_{z,y,x} <= (s_{z,y,x} & "00") - σ_{z,y,x}
d^N_{z,y,x} <=
    0 when y_0 else
    (s_{z,y-1,x} & "00") - σ_{z,y,x}
```

```
d^W_{z,y,x} <=
    0 when y_0 else
    (s_{z,y-1,x} & "00") - σ_{z,y,x} when x_0 else
    (s_{z,y,x-1} & "00") - σ_{z,y,x}
d^NW_{z,y,x} <=
    0 when y_0 else
    (s_{z,y-1,x} & "00") - σ_{z,y,x} when x_0 else
    (s_{z,y-1,x-1} & "00") - σ_{z,y,x}
```

A partir de estas diferencias se generará el vector de diferencias [9, 4.5.3], que incluirá, en caso de predicción completa, las diferencias direccionales $d^N_{z,y,x}, d^W_{z,y,x}, d^{NW}_{z,y,x}$. Dado que es el único lugar donde se utilizan, si tratamos con predicción reducida la síntesis de los circuitos que generan estas diferencias direccionales será omitida.

5.1.1.4. Diferencia local central predicha

Utilizando las diferencias $d_{z,y,x}$ y unos pesos $\omega_z^{(i)}(t)$, explicados más adelante, tenemos que hacer un producto escalar de vectores para hallar la diferencia local central predicha $\hat{d}_z(t)$, utilizando la fórmula:

$$\hat{d}_z(t) = \boldsymbol{W}_z^T(t)\boldsymbol{U}_z(t) \tag{5.5}$$

Donde $\boldsymbol{W}_z(t)$ y $\boldsymbol{U}_z(t)$ son vectores formados por los pesos y las diferencias, de longitud dependiente del modo de predicción del algoritmo (reducida o completa). Para ver qué contienen los vectores puede verse [9, 4.5.3, 4.6.2]. En resumen, el vector de diferencias $\boldsymbol{U}_z(t)$ contiene las diferencias $d_{z,y,x}$ centrales de bandas previas, más las diferencias direccionales de la muestra actual, y el vector de pesos $\boldsymbol{W}_z(t)$ contiene una serie de valores que ayudan en la predicción.

Como producto vectorial, es importante darse cuenta de que vamos a hacer la suma de (potencialmente) decenas de productos. Es la parte más costosa del algoritmo, y que como luego se ha comprobado en el Capítulo 6 es atravesada por el camino crítico. Así pues debemos ejecutarla de la manera más eficiente posible. Para ello hay que asegurarse de que la reducción de las sumas se realice en un número logarítmico de etapas, pues una concatenación de sumadores haría que los tiempos de cálculo se disparasen. Afortunadamente, el sintetizador ya lo hace automáticamente al recibir un código de la forma:

```
d̂_z(t) := 0
for i in 0 to P* - 1 loop
    d̂_z(t) := d̂_z(t) + u_z^(i)(t) * ω_z^(i)(t);
end loop;
```

El tiempo así conseguido $t_{\hat{d}} = t_* + \log_2(P^*) \cdot t_+$, donde t_* es el tiempo de una multiplicación, y t_+ de una suma, es mucho mejor que el de la aproximación ingenua, que costaría $t_* + P^* \cdot t_+$.

5.1.1.5. Filtros de peso y diferencias

Debido a la manera de manejar la memoria, y de resetear los valores del algoritmo, sería posible que tras procesar una imagen, se "colaran" valores de la anterior compresión en la nueva. Esto es debido a que al iniciar la compresión, no tenemos bandas previas, y el producto escalar de la sección 5.1.1.4 siempre trabaja con el tamaño máximo de vector, con lo que datos de bandas inexistentes podrían afectar al resultado.

Para evitar esto, filtramos los vectores de diferencias y pesos para quedarnos exclusivamente con los datos que nos interesan, colocando ceros en las posiciones que no existen para la banda actual. De esta manera se posibilita el procesamiento consecutivo de imágenes.

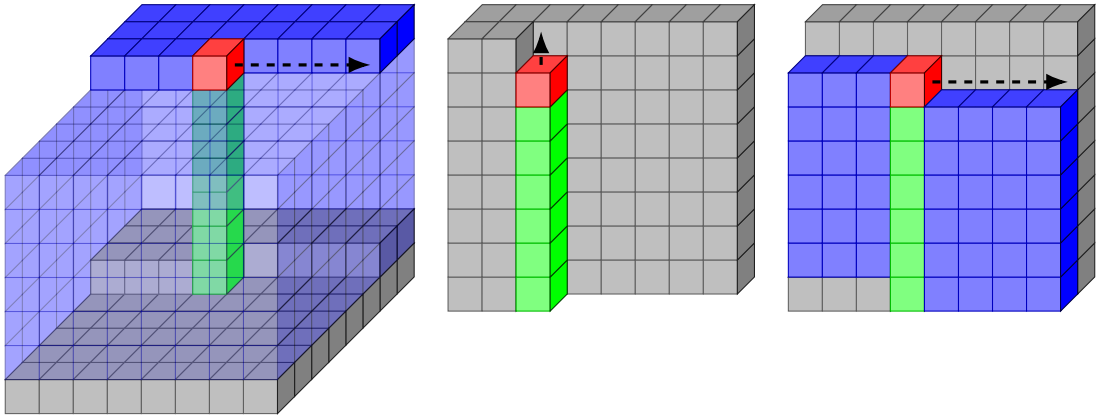


Figura 5.6: Respectivamente, recorridos BSQ, BIP, BIL para las diferencias almacenadas (azul), necesarias (verde) y actual (rojo). En este caso, $P = 6$

5.1.1.6. Almacenamiento de diferencias

Es necesario almacenar las diferencias centrales $d_{z-i,y,x} \forall i \in [1, P]$ ya que se utilizan en el cálculo de $\hat{d}_z(t)$. El número de diferencias a almacenar viene dado por P. Al igual que ocurría con las muestras, el almacén de diferencias varía en tamaño según cómo se recorra la imagen. Tenemos de nuevo tres alternativas, esta vez con diferencias mucho mayores de espacio necesario:

- En modo BSQ se necesita una gran cantidad de memoria. Las diferencias $d_{z-i,y,x}$ son calculadas $i \cdot N_X \cdot N_Y$ ciclos atrás, lo cual implica que la memoria alcanza un tamaño de $N_Y \cdot N_X \cdot P$. Como necesitamos muestras separadas en intervalos de tamaño $N_X \cdot N_Y$, la solución consistirá en llevar una memoria de ese tamaño para cada banda anterior.

  Como ya ocurriera antes, las memorias se pueden implementar como registros de desplazamiento de tamaño elevado (que dan mejor rendimiento (sección 6.2.6), como memoria distribuida, o como memoria de bloque. La elegida será la memoria de bloque por su amplia disponibilidad y adecuado tiempo de respuesta. No habrá una única memoria (eso forzaría P puertos de lectura, lo cual replicaría recursos) sino una por banda.

  Observamos el esquema de la memoria en la Figura 5.6

- En modo BIP, las diferencias han sido calculadas con las muestras inmediamente anteriores, con lo que basta almacenar las P anteriores en un registro de desplazamiento, desde el que acceder a todas a la vez.

- Por último, para BIL, como sólo necesitaremos guardar $N_X \cdot P$ muestras, utilizaremos P registros de desplazamiento, uno para cada línea escaneada. Sería igualmente válido utilizar P memorias (en bloque o distribuidas, son de tamaño no muy grande) pero en ningún caso una sola, pues se replicaría P veces para permitir las P lecturas simultáneas.

5.1.1.7. Almacenamiento e inicialización de pesos

Cada banda, comprimida individualmente, utiliza un vector de pesos para dar la mejor predicción posible. El vector $\boldsymbol{W}_z(t)$ consta de P^* componentes (enteros con signo de tamaño $\Omega+3$), que se utilizan para el producto escalar con $\boldsymbol{U}_z(t)$ (sección 5.1.1.4). Veamos cómo conseguir el mejor aprovechamiento de la memoria para almacenar estos vectores:

- En modo BSQ, dado que las bandas se recorren enteras antes de pasar a la siguiente, nos basta almacenar un único vector de pesos en cada momento, que es el correspondiente a la

banda actual. Tras cada banda, el vector almacenado tomará los valores de inicialización elegidos por el usuario.

```
if t_0 then
    W <= W_z^init
else
    W <= W(t+1)
end if;
```

- En los modos BIL y BIP, todos los vectores están activos durante la compresión completa, por lo que debemos guardarlos todos. La diferencia es que en BIP necesitamos uno diferente cada ciclo, y en BIL uno diferente cada N_X ciclos. En ambos casos implementamos una pila de tamaño N_Z en forma registro de desplazamiento[1]. En el modo BIP, en cada ciclo se mete el vector actual a la pila, y se saca el siguiente, así cada ciclo disponemos del vector de la banda actual. En modo BIL, se modifica la cima de la pila durante el recorrido de la línea, y ante un cambio de banda se hace la operación de meter el vector de la banda vieja y sacar el de la nueva. Con cada marco de la imagen, al volver a la primera banda, volveremos a disponer del primer vector.

  La inicialización de vectores, en ambos casos, se hace de golpe para todos en cuanto llega la señal de reset al módulo.

```
W <= W_0

--BIP
if rst = '1' then
    W_i <= W_i^init   ∀i ∈ [0, N_Z − 1]
elsif not t_0 then
    W_i <= W_{i+1}   ∀i ∈ [0, N_Z − 2]
    W_{N_Z−1} <= W_0(t+1)
end if;

--BIL
if rst = '1' then
    W_i <= W_i^init   ∀i ∈ [0, N_Z − 1]
elsif not t_0 then
    if not x^h then
        W_0 <= W_0(t+1)
    else
        W_i <= W_{i+1}   ∀i ∈ [0, N_Z − 2]
        W_{N_Z−1} <= W_0(t+1)
    end if;
end if;
```

5.1.1.8. Valor predicho de muestra y error

En la Ecuación (5.6) viene dada la fórmula que nos da el *valor predicho de muestra escalado* $\tilde{s}_z(t)$.

[1]La cantidad de FPGA necesaria es proporcional linealmente al número de bandas, luego no es muy costosa su implementación

$$
\begin{aligned}
\tilde{s}'_z(t) &= \left\lfloor \frac{\text{mód}^*_R \left[\hat{d}_z(t) + 2^\Omega \left(\sigma_z(t) - 4s_{\text{mid}}\right)\right]}{2^{\Omega+1}} \right\rfloor + 2s_{\text{mid}} + 1 \\
\tilde{s}_z(t) &= \begin{cases} \text{clip}\left(\tilde{s}'_z(t), \{2s_{\text{mín}}, 2s_{\text{máx}} + 1\}\right) & t > 0 \\ 2s_{z-1}(t) & t = 0, P > 0, z > 0 \\ 2s_{mid} & t = 0 \wedge (P = 0 \vee z = 0) \end{cases}
\end{aligned}
\tag{5.6}
$$

Vemos que existe una operación mód^*_R que podría potencialmente implicar pérdida de información si R es muy pequeño. En [10, 4.2.5] dan la fórmula para calcular un R suficiente R^* que no provoque redondeos nunca, dada en la Ecuación (5.7). Como la implementación es lo suficientemente rápida (Capítulo 6), podemos permitirnos el usar R^* como tamaño de registro en las operaciones intermedias de $\tilde{s}'_z(t)$, consiguiendo así la mejor predicción posible. Si fuera necesario, no supondría problema alguno usar valores $R < R^*$ para ganar velocidad, si bien la compresión podría ser peor.

$$
\begin{aligned}
R^* &= \Omega + 2 + \left\lceil \log_2 \left((2^D - 1)(8P + \kappa) + 1\right)\right\rceil \\
\kappa &= \begin{cases} 1 & \text{modo reducido} \\ 19 & \text{modo completo} \end{cases}
\end{aligned}
\tag{5.7}
$$

El valor predicho de muestra y el error de predicción escalado vienen dado por la sencilla Ecuación (5.8)

$$
\begin{aligned}
\hat{s}_z(t) &= \left\lfloor \frac{\tilde{s}_z(t)}{2} \right\rfloor \\
e_z(t) &= 2s_z(t) - \tilde{s}_z(t)
\end{aligned}
\tag{5.8}
$$

El código por tanto no será complejo, y como siempre ganamos velocidad utilizando desplazamientos y concatenaciones para dividir y multiplicar:

```
s'z(t) <= shift_right(mód*R [d̂z(t) + 2^Ω (σz(t) − 4s_mid)], Ω + 1) + (s_mid & "0") + 1

s̃z(t) <=
    s_mid & "0" when t0 ∧ (P0 ∨ z0) else
    s_{z−1}(t) & "0" when t0 else
    0 when s'z(t) < 0 else
    2 ∗ s_máx + 1 when s'z(t) > 2 ∗ s_máx + 1 else
    s'z(t)

ŝz(t) <= s̃z(t)(D downto 1)

ez(t) <= (sz(t) & "0") − ("0" & s̃z(t));
```

5.1.1.9. Actualización de pesos

Tras el procesado de cada muestra, los pesos se actualizan a fin de predecir mejor la siguiente. Para ello se emplea la Ecuación (5.9), que se aplica a cada componente individual del vector.

$$
\boldsymbol{W}_z(t+1) = \text{clip}\left(\boldsymbol{W}_z(t) + \left\lfloor \frac{1}{2}\,\text{sgn}^+\left[e_z(t)\right] \cdot 2^{-\rho(t)} \cdot \boldsymbol{U}_z(t) + 1 \right\rfloor, \{\omega_{\text{mín}}, \omega_{\text{máx}}\}\right) \tag{5.9}
$$

Para su cálculo, necesitamos conocer previamente el valor de $\rho(t)$:

$$\rho(t) = \text{clip}\left(v_{\text{mín}} + \left\lfloor \frac{t - N_X}{t_{\text{inc}}} \right\rfloor, \{v_{\text{mín}}, v_{\text{máx}}\}\right) + D - \Omega \tag{5.10}$$

Aquí realizamos una observación. El valor de ρ está acotado por:

$$-24 \leq v_{\text{mín}} + D_{\text{mín}} - \Omega_{\text{máx}} \leq \rho(t) \leq v_{\text{máx}} + D_{\text{máx}} - \Omega_{\text{mín}} \leq 21 \tag{5.11}$$

Donde $D_{\text{mín}}, D_{\text{máx}}, \Omega_{\text{mín}}, \Omega_{\text{máx}}$ indican los valores mínimos y máximos que toman D y Ω en las diferentes configuraciones del algoritmo. Podemos utilizar esto para deducir que en la Ecuación (5.9), el valor de registro necesario para no provocar desbordamiento en las operaciones es de $\text{máx}\,(\Omega + 3, D + 3 + \text{mín}\,(1, v_{min} + D - \Omega)) + 1$.

$D + 3$ es el tamaño de los elementos de $\boldsymbol{U}_z(t)$. Por otra parte, sólo nos hemos fijado en el límite inferior de ρ, $v_{\text{mín}} + D - \Omega$ pues es el que provoca un desplazamiento a izquierdas, el límite superior desplazaría a derechas dividiendo los valores, luego no puede conllevar desbordamiento. La otra opción es que el tamaño sea máximo por parte del vector de pesos $(\Omega + 3)$. Por último sumamos 1 para evitar desbordamiento. Esta aproximación, si bien utiliza comparaciones y operaciones de bastantes bits, no es excesivamente costosa en tiempo respecto a otras soluciones como un cálculo del número de cifras del resultado para ajustarlo a límites [2].

La Ecuación (5.11) nos proporciona el tamaño de registro necesario para calcular ρ sin desbordamiento. Al no ser muy grande, nos basta con un entero con signo de 6 bits, cuyo tamaño podemos fijar para cualquier configuración para estar seguros de que no hay problema.

En cuanto a $\boldsymbol{W}$, lo único que nos queda ver es cómo calcular sgn^+. Esto es muy sencillo ya que en la representación de complemento a dos, el bit más significativo nos da el signo del número, luego bastará consultarlo para ver si tenemos que cambiar de signo el producto al que acompaña.

Con todo esto visto, los códigos quedan como siguen:

```
ρ*(t) <= v_mín + shift_right(t − N_X, t_inc)

ρ(t) <=
    v_mín + D − Ω when ρ*(t) < v_mín else
    v_máx + D − Ω when ρ*(t) > v_máx else
    ρ*(t) + D − Ω

for i in 0 to P* − 1 generate
    U_z^{*(i)}(t) <=
        U_z^{(i)}(t) when e_z(t) >= 0 else
        −U_z^{(i)}(t)

    U_z^{**(i)}(t) <=
        shift_right(U_z^{*(i)}(t), ρ(t)) when ρ(t) >= 0 else
        shift_left(U_z^{*(i)}(t), −ρ(t))

    W_z^{*(i)}(t) <=
        W_z^{(i)}(t) + shift_right(1 + U_z^{**(i)}(t), 1)
```

[2] Probando la alternativa (calcular el logaritmo) los resultados de síntesis daban peor tiempo que esta aproximación.

```
    W_z^(i)(t+1) <=
        ω_mín when W_z^*(i)(t) < ω_mín else
        ω_máx when W_z^*(i)(t) > ω_máx else
        W_z^*(i)(t)
end generate;
```

5.1.1.10. Residuo de predicción

Lo último que hace el predictor es generar el *residuo de predicción mapeado* $\delta_z(t)$. Para ello se definen las siguientes cantidades auxiliares:

$$\begin{aligned} \Delta_z(t) &= s_z(t) - \hat{s}_z(t) \\ \theta_z(t) &= \text{mín}\left(\hat{s}_z(t) - s_{\text{mín}}, s_{\text{máx}} - \hat{s}_z(t)\right) \end{aligned} \tag{5.12}$$

Y el cálculo de $\delta_z(t)$ queda como sigue:

$$\delta_z(t) = \begin{cases} |\Delta_z(t)| + \theta_z(t) & |\Delta_z(t)| > \theta_z(t) \\ 2\,|\Delta_z(t)| & 0 \leq (-1)^{\tilde{s}_z(t)}\Delta_z(t) \leq \theta_z(t) \\ 2\,|\Delta_z(t)| - 1 & c.c \end{cases} \tag{5.13}$$

Con esto conseguimos que todos los valores de $\delta_z(t)$ sean enteros positivos de a lo sumo D bits, que pasamos al codificador. El código de esta parte es:

```
Δ_z(t) <= s_z(t) - s̃_z(t)
θ_z(t) <=
    s̃_z(t) when s̃_z(t) < s_máx - s̃_z(t) else
    s_máx - s̃_z(t)

δ_z(t) <=
    |Δ_z(t)| + θ_z(t) when |Δ_z(t)| > θ_z(t) else
    shift_left(|Δ_z(t)|, 1) when
        s̃_z(t)(0) = '0' and Δ_z(t)(D) = '0' and θ_z(t) ≥ Δ_z(t) or
        s̃_z(t)(0) = '1' and Δ_z(t)(D) = '1' and θ_z(t) ≥ -Δ_z(t) or
        Δ_z(t) = 0 else
    shift_left(|Δ_z(t)|, 1) + 1
```

Los cálculos de $\Delta_z(t)$ y $\theta_z(t)$ no presentan complicación, pero para el caso de $|\Delta_z(t)|$ tenemos que tener cuidado con el exponente del -1. El primer caso lo solucionamos con una comparación. Para el segundo, observamos que si $\Delta_z(t) = 0$ la desigualdad se cumple siempre. En caso contrario, la primera desigualdad se cumplirá solo si los signos de $(-1)^{\tilde{s}_z(t)}$ y $\Delta_z(t)$. El primer signo lo tenemos viendo la paridad de $\tilde{s}_z(t)$, y el segundo viendo el bit más significativo. La segunda desigualdad la comprobamos de la manera habitual, teniendo cuidado de cambiar a $\Delta_z(t)$ de signo si era negativo.

5.1.2. Codificador

Para la codificación partimos de los valores $\delta_z(t)$ obtenidos del predictor. Gracias a un contador y un acumulador, las entradas se transformarán en un flujo de bits que, mediante código prefijo, contendrá la información comprimida de todos los $\delta_z(t)$, suficiente para realizar la operación inversa y descomprimir la imagen.

Vimos ya las fórmulas de contador y acumulador en las Ecuaciones (4.18) y (4.19), y la forma de calcular combinacionalmente el contador en la Ecuación (4.21). Pasamos en primer lugar a ver el código del contador:

```
Γ(t) <=
    2^γ0 − 1 + t when t ≤ 2^γ* − 2^γ0 else
    2^(γ*−1) + ((t − 2^γ* + 2^γ0 − 1) mód 2^(γ*−1))
```

Gracias al cual nos evitamos posibles problemas que pueda haber si lo almacenáramos (entre otros, en el modo BIL habría que llevar N_Z contadores diferentes).

El acumulador ya no puede ser calculado combinacionalmente pues depende de todos los $\delta_z(t)$ procesados con anterioridad. Así pues, tenemos el siguiente código:

```
Σz(t) <=
    Σz^init when t_0 else
    shift_right(Σz(t − 1) + δz(t − 1) + 1, 1) when Γ(t − 1) = 2^γ* − 1 else
    Σz(t − 1) + δz(t − 1)
```

Para hacer la codificación, calculamos los valores $k_z(t)$ y $u_z(t)$ correspondientes a u y k de la Ecuación (4.16) con una ligera variante, siguiendo las ecuaciones [9, 44,45]:

```
k'z(t) <= Σz(t − 1) + shift_right(49 · Γ(t − 1), 7)

process
    kz(t) <= D − 2
    if Γ(t) > k'z(t) then
        kz(t) <= 0
    else
        for i in 1 to D − 2 loop
            if shift_left(Γ(t), i) > k'z(t) then
                kz(t) <= i - 1
                exit;
            end if;
        end loop;
    end if;
end process;

uz(t) <= shift_right(δz(t), kz(t))
```

Vemos que para el cálculo de $k_z(t)$, en lugar de hacer una costosa divisón despejando en la Ecuación (4.18), hacemos $D-2$ desplazamientos de los cuales nos quedamos con el primero que cumpla la desigualdad buscada.

Finalmente se preparan los datos de salida. En nuestro caso, siguiendo la sección 4.2.2, vamos a sacar tres valores:

- El número de ceros que preceden al código $Z_z(t)$.
- El código en sí $C_z(t)$.
- El número de bits del código utilizados $B_z(t)$.

La forma de calcularlos es como sigue:

```
Zz(t) <=
    0 when t0 else
```

```
        u_z(t) when u_z(t) < u_máx else
        u_máx

C_z(t) <=
        δ_z(t) when t_0 else
        shift_right("1" & shift_left(δ_z(t), D − k_z(t)), D − k_z(t))
            when u_z(t) < u_máx else
        δ_z(t)

B_z(t) <=
        D when t_0 else
        k_z(t) + 1 when u_z(t) < u_máx else
        D
```

La parte interesante está en la realización de dos desplazamientos consecutivos, a derechas y a izquierda, para quedarse con los $k_z(t)$ bits menos significativos y añadir un uno al principio, que formará el código prefijo junto a los ceros indicados por $Z_z(t)$.

5.1.3. Otros

5.1.3.1. Contador

Es necesario conocer las coordenadas actuales para una muestra concreta, pues muchas de las ecuaciones calculadas dependen de valores de x, y, z o t. Como vimos en la sección 2.1, la forma de recorrer la imagen varía, así que necesitamos contar de manera diferente dependiendo de si estamos en modo BSQ, BIL o BIP.

Así pues, existe un contador base que recibe tres límites N_i, N_j, N_k y que devuelve tres valores $i \in [0, N_i - 1]$, $j \in [0, N_j - 1]$, y $k \in [0, N_k - 1]$. El contador incrementa primero la i, luego la j, y por último la k, reseteando cada una al llegar a su límite, e incrementando la siguiente en ese momento. Los tres modos se consiguen haciendo diferentes mapeos de x, y, z y N_X, N_Y, N_Z:

- Para recorrer en modo BSQ:

$$x \mapsto i, y \mapsto j, z \mapsto k, N_X \mapsto N_i, N_Y \mapsto N_j, N_Z \mapsto N_k$$

- En modo BIL:

$$x \mapsto i, z \mapsto j, y \mapsto k, N_X \mapsto N_i, N_Z \mapsto N_j, N_Y \mapsto N_k$$

- Y en modo BIP:

$$z \mapsto i, x \mapsto j, y \mapsto k, N_Z \mapsto N_i, N_X \mapsto N_j, N_Y \mapsto N_k$$

5.1.3.2. Generador de límites

Salvo la coordenada z, del resto sólo necesitamos saber si son cero o sin son máximas. Por tanto es mucho más práctico hacer comparaciones previas, y llevar por la FPGA el dato ya comparado (que sólo es un bit), que hacer las comparaciones en cada módulo que las necesite (más bits que llevar, y duplicación de recursos).

Necesitamos, en las múltiples configuraciones del algoritmo, los siguientes valoes:

$$t_0, y_0, x_0, z_0, x^h$$

Donde el subíncide cero indica que valen cero, y el superíndice h que tienen su valor máximo.

Con esta combinación contador/generador de límites, dejamos al programa de síntesis el trabajo de decidir si es necesario o conveniente el duplicar contadores, enviar las señales enteras en lugar de comparadas, etc. Así síntesis para diferentes dispositivos podrán tener diferentes optimizaciones de área o tiempo.

5.2. Pruebas virtuales

La primera aproximación para ver si de verdad funciona una implementación para FPGA, es comprobar con simulaciones que las cosas van yendo bien. Es prácticamente imposible que funcione en placa no habiendo funcionado en simulación. Así que, como la lógica de primer orden dicta, necesitamos primero que funcione en simulación para luego pasar a placa.

Lo ideal es probar con los casos extremos que se nos ocurran, e introducir algún ejemplo real. Con imágenes hiperespectrales no es tan fácil ya que la disponibilidad es limitada. Una muy buena forma de superar estas limitaciones es utilizando conjuntos de datos aleatorios. Ya sean completamente aleatorios o con alguna generación dirigida (Apéndice A) que permita una aproximación más veraz a la realidad.

Para las primeras pruebas se han decidido utilizar datos completamente aleatorios, con la función `uniform` proporcionada por la librería `ieee.math_real`. Esta función, dados dos valores enteros positivos (semillas), genera un tercer número real entre 0 y 1. La gran ventaja es que dadas las mismas semillas iniciales, la función generará la misma secuencia de números independientemente del ordenador donde se ejecute, por lo tanto los tests son reproducibles con sólo saber las semillas, evitando transferencias de cientos de megabytes que suponen las imágenes hiperespectrales.

El esquema general de proceso para realizar este tipo de tests es el siguiente:

```
testing: process
    variable test_input: integer;
    --seed values for random generator
    variable seed1, seed2: positive;
begin
    --test random inputs
    loop
        --generate random values
        next_integer(seed1, seed2, <inteval>, test_input);
        --assign values
        input <= to_unsigned(test_input, input'length);
        --wait for signals to propagate
        wait for clk_period;
        --compute expected output
        test_output := f(test_input);
        --check output
        assert to_integer(output) = test_output
            report "output not equal" & LF
            severity failure;
    end loop;
end process;
```

La parte importante del esquema se encuentra en la llamada a `uniform` [30], que encapsula `next_integer`. Las semillas, a las que el usuario dará los valores que quiera en el intervalo

[1, 2147483562] son modificadas por la función, para en cada llamada generar valores diferentes. (Nótese que una llamada con las mismas semillas produce el mismo resultado).

Tras generar el valor, se pasa al circuito y se le da tiempo para que se estabilice y se generen los resultados que debe dar. En este punto se calcula el resultado teórico, que ha sido programado en "software" en lugar de "hardware". La comprobación con **assert** hace que ante un fallo, la ejecución se pare y se devuelvan los datos deseados sobre el estado del módulo VHDL, para así poder depurar con mayor comodidad los errores.

Otro método empleado para los tests es el de utilizar archivos de datos reales en lugar de números aleatorios. La ventaja que presenta respecto al anterior es que el comportamiento del algoritmo se ajustará más a la realidad, si bien los casos extremos serán mucho más extraños y por tanto la robustez será menos probada.

El esquema general para estos tests es igual que el anterior, pero en lugar de leer valores aleatorios se leen de fichero. Los ficheros con los que tratamos contienen datos binarios. Para tratarlos, abrimos el archivo como de caracteres y leemos de uno en uno, transformando a bytes numéricos. La apertura de fichero se realiza con:

```
type t_char_file is file of character;
file infile: t_char_file open read_mode is "Path_to_file";
variable char_buffer: character;
```

En el caso de las imágenes hiperespectrales, tenemos ficheros provenientes del sensor AVIRIS, cuyas muestras son de 16 bits cada una. Para leerlas, leemos dos bytes consecutivos y los pegamos con una suma y multiplicación, como sigue:

```
read(infile, char_buffer);
read_value := character'POS(char_buffer);
read(infile, char_buffer);
read_value := read_value*256 + character'POS(char_buffer);
```

Para lo que hemos necesitado la biblioteca **std.textio**. No es muy cómodo tratar con ficheros en VHDL. De hecho una opción muy común es el utilizar programas auxiliares que transformen ficheros a constantes que se carguen en bibliotecas mediante **use**. Sin embargo en este caso no era viable ya que con las imágenes hiperespectrales estamos hablando de ficheros que llegan a los cientos de megabytes, y transformar esos datos binarios a constantes en VHDL produciría archivos de gigabytes de tamaño.

También es importante guardar los resultados de las pruebas simuladas para utilizarlos posteriormente en la verificación sobre placa. Ya hayamos creado los tests con números aleatorios o con archivos, será necesario guardar para cada ciclo de reloj todos los valores de los puertos tanto de entrada como de salida, con la esperanza de que introduciendo los mismos datos que se utilizaban en las simulaciones, se obtengan los mismos resultados.

Para ello utilizamos el siguiente esquema:

```
testing: process
    file outfile: text is out  "out_test.bin";
    variable, output_data: line;
begin
    write(output_data, "constant res: res_t := (""", right, 35);
    --reset
    for i in 0 to SAMPLES - 1 loop
        --read input value and test results
        --write outputs to file for later use
        write(output_data, test_output, right, <bits_of_output>);
        if i /= SAMPLES_TO_TEST - 1 then
```

```
            write(output_data, """,""", right, 3);
        end if;
    end loop;
    write(output_data, """);", right, 3);
    writeline(outfile, output_data);
end process;
```

Y así conseguimos un archivo con las variables de prueba ya listas para copiar, pegar, y realizar comprobaciones en placa, donde estas variables se guardarán ya en la memoria interna de la misma.

5.3. Pruebas físicas

Tras las pruebas simuladas, pasamos a realizar pruebas directamente sobre la placa. Para ello, realizamos previamente una simulación y vemos el resultado que debería obtenerse.

Con estos datos preparamos un test que contiene todos los datos de entrada utilizados y los esperados de salida. Al cargarlo en la placa, se cargarán las entradas y una a una se compararán con las salidas. En caso de dar un resultado correcto, se encenderán unos leds de la placa que contiene la FPGA, y si falla el test, se encenderán otros diferentes.

El esquema general es como sigue:

```
when IDLE =>
    if (enable = '1') then
        estado_new<=GENERAR;
    end if;

when GENERAR =>
    enable_circuit <= '1';
    data_in <= input_data(to_integer(unsigned(cont_i)));

    if (cont_i=max_i-1) then
        cont_i_new <= (0 => '1', others => '0');
        estado_new <= CORRECTO;
    else
        cont_i_new<=cont_i+1;
    end if;

    if (data_out /= output_data(to_integer(unsigned(cont_i))) then
        estado_new<=ERROR;
    end if;

when CORRECTO =>
    leds <= "01010101";

when ERROR =>
    leds <= "00110011";
```

Una vez activamos las pruebas, se comienzan a introducir secuencialmente todas las entradas de prueba, y comprobando todas las salidas. Si en algún momento se detecta un fallo, se cambia al estado de error y se indica mediante un cambio en los leds. En caso de terminar sin problemas, se pasa al estado correcto y se muestra un patrón diferente en los leds. Este método tiene la

Item	Descripción	Rango	Implementado
1	Muestras con signo	N/A	✓
2	Muestras sin signo	N/A	✓
3	N_X	$1 \dots 2^{16}$	✓$1 \dots 2^{16}$
4	N_Y	$1 \dots 2^{16}$	✓$1 \dots 2^{16}$
5	N_Z	$1 \dots 2^{16}$	✓$1 \dots 2^{16}$
6	D	$2 \dots 16$	✓$2 \dots 16$

Tabla 5.1: Valores permitidos en la implementación respecto a la imagen

limitación de que no se pueden probar arrays muy grandes de datos pues tienen que caber en la memoria de la FPGA. Aun así, es suficiente para realizar las pruebas necesarias.

5.4. Conformación con el estándar

El estándar se cumple a rajatabla para las opciones implementadas, habiendo comparado resultados con la implementación en `java` del grupo GICI [31] y con la implementación en `C` de Luca Fossati [32]. Tanto las dos implementaciones software como la implementación hardware han producido los mismos resultados ante los mismos datos de entrada y configuraciones.

En las tablas 5.1 a 5.4 observamos los parámetros de la implementación que se deben detallar [9, A] para mostrar la conformación con el estándar.

Algunas opciones no se implementan por no ser práctica su inclusión en la FPGA. Si se quisieran, un coprocesador que las añadiera sería la solución ideal. El motivo es que son elementos como la cabecera de la imagen que, dada una configuración concreta del algoritmo, siempre será igual. Las razones de no implementar algunas opciones son las siguientes:

- La cabecera se ha considerado prescindible pues una vez hecha la configuración del algoritmo en placa, va a ser siempre igual, por tanto se puede añadir posteriormente sin complicar la lógica del circuito para añadirla al vuelo. El ajuste a tamaño de palabra tampoco se ha considerado necesario ya que es algo que se puede relegar al canal de comunicación, que tendrá que utilizar los métodos adecuados para transmitir en múltiplos del número de bytes que utilice.
- La profundidad de entrelazado no es algo que se utilice en la práctica para la **toma** de imágenes. Éstas son tomadas siguiendo una de las tres ordenaciones BSQ, BIL y BIP. Sí se emplea para guardar las imágenes, pero a la hora de tratar en tiempo real con su captura, no es necesario implementarlo.
- Por último, la codificación por bloques se ha obviado pues, pese a su mayor complejidad, no introduce mejoras significativas respecto a la compresión por muestras, e incluso puede dar menor rendimiento [10, 3.3.1].

El resto de opciones de implementación han sido programadas exitosamente, admitiéndose la totalidad de los rangos sugeridos. Esto permitirá la adaptación a cualquier sensor hiperespectral.

Item	Descripción	Rango	Implementado
1	P, bandas para predicción	$0 \dots 15$	✓$0 \dots 15$
2	Modo de predicción completa	N/A	✓
3	Modo de predicción reducida	N/A	✓
4	Suma orientada a vecinos	N/A	✓
5	Suma orientada a columna	N/A	✓
6	Ω, resolución de pesos	$4 \dots 19$	✓$4 \dots 19$
7	Pesos por defecto	N/A	✓
8	Pesos a medida	N/A	✓
9	Q, resolución de pesos	$3 \dots \Omega + 3$	✓$3 \dots \Omega + 3$
10	R, tamaño de registro	$D + \Omega + 3 \dots 64$	✓$D + \Omega + 3 \dots 64$
11	$v_{\text{mín}}$, valor inicial de escalado de pesos	$-6 \dots v_{\text{máx}}$	✓$-6 \dots v_{\text{máx}}$
12	$v_{\text{máx}}$, valor final de escalado de pesos	$v_{\text{mín}} \dots 9$	✓$v_{\text{mín}} \dots 9$
13	t_{inc}, Intervalo de cambio del escalado	$2^4, 2^5 \dots 2^{11}$	✓$2^4, 2^5 \dots 2^{11}$

Tabla 5.2: Valores permitidos en la implementación respecto al predictor

Item	Descripción	Implementado
1	Compresión de la imagen	✓
2	Ajuste a tamaño de palabra	✗
3	Cabecera de imagen	✗
4	Codificación de la tabla de pesos	N/A
5	Compresión del cuerpo de la imagen	✓
6	Orden de compresión BI	✓BIL, BIP
7	Orden de compresión BSQ	✓
8	Profundidad de entrelazado, M	✗
9	Codificador adaptativo por muestras	✓
10	Codificador adaptativo por bloques	✗

Tabla 5.3: Valores permitidos en la implementación respecto al codificador

Item	Descripción	Rango	Implementado
1	Inicialización de acumulador	N/A	✓
2	Exponente inicial del contador γ_0	$1 \dots 8$	✓$1 \dots 8$
3	Constante de inicialización K	$0 \dots D - 2$	✓$0 \dots D - 2$
4	Tamaño de reescalado del contador γ^*	máx $\{4, \gamma_0 + 1\} \dots 9$	✓máx $\{4, \gamma_0 + 1\} \dots 9$
5	Límite de longitud unaria U_{max}	$8 \dots 32$	✓$8 \dots 32$

Tabla 5.4: Valores permitidos en la implementación respecto al codificador adaptativo por muestras

Capítulo 6

Resultados experimentales

6.1. Punto de partida

Para esta sección vamos a mostrar los resultados obtenidos con diferentes configuraciones del algoritmo, variando recorridos de la imagen, métodos de compresión y métodos de predicción, entre otros parámetros.

Es importante comentar que no sería posible utilizar un procesador de propósito general para ejecutar este algoritmo. Con un Intel i7 de última generación se consiguen unos resultados de aproximadamente $15 - 20MB$ comprimidos por segundo, lejos de los $\approx 27MB/s$ que recopila el sensor. Por tanto algún tipo de aceleración es necesaria, ya fuera por ejemplo mediante operaciones vectoriales, GPUs, o mediante hardware específico como es el caso en las FPGAs.

Las resultados mostrados corresponden a compilaciones sobre la placa de Xilinx *Virtex 7, XC7VX690T-FFG1761*. Si bien los datos porcentuales dependen de la placa, la cantidad absoluta de recursos utilizados no debería variar demasiado entre FPGAs con elementos programables de similares características. Posteriormente se mostrarán resultados en la placa (también de Xilinx) *Space-Grade Virtex-4QV, XQR4VSX55-10CF1140*, más pequeña pero cuyo uso es apto en misiones espaciales. Vemos a continuación los datos técnicos [33]:

Modelo	Slices	Celdas lógicas	Max. D. RAM	Max. B. RAM	I/O	DSP
XQR4VSX55	24576	55296	$384kb$	$5760kb$	640	n/a
XC7VX690T	108300	693120	$10888kb$	$52920kb$	1000	3600

Aquí, la D y B de la RAM se refieren respectivamente a memoria distribuida (implementada sobre los slices) y en bloques específicos de memoria. Vemos que el modelo apto para espacio tiene bastantes limitaciones respecto al otro, pero no será problema como veremos posteriormente.

En principio todos los resultados serán asumiendo imágenes del tamaño proporcionado por el sensor AVIRIS, cuyas dimensiones son de $512 \times 614 \times 224$ con una profundidad de 16 bits cada muestra. Los parámetros del algoritmo que se utilizarán, a menos que se diga lo contrario, son:

$$\begin{array}{ccc} N_X = 512 & N_Y = 614 & N_Z = 224 \\ D = 16 & \Omega = 19 & P = 3 \\ v_{min} = -1 & v_{max} = 3 & t_{inc} = 6 \\ \gamma_0 = 1 & \gamma^* = 6 & u_{max} = 16 \\ & k_z = 5 & \end{array}$$

La inicialización de los vectores de peso es la dada por defecto [9, 4.6.3.2], y para los acumuladores se utiliza k_z como constante global. Se utiliza el modo de suma **orientada a vecinos**, la **predicción completa**, y el orden de imagen **BSQ**.

El elegir estos parámetros base es debido a que son los utilizados en [10, C-1] para probar diferentes tasas de compresión. Nótese que en dicha referencia, el valor de R es ajustado a 64. Aquí, por eficiencia, tomamos el mínimo valor que no provoque desbordamiento, como regía la Ecuación (5.7).

Asumimos además, para demostrar la validez del circuito en tiempo real, que la entrada de datos proviene de un sensor AVIRIS. Este sensor proporciona una línea de 512 píxeles (cada uno de 224 bandas) en 8,3ms. Esto implica que recibimos 13817831 muestras por segundo, o lo que es lo mismo, debemos ser capaces de procesar una muestra cada 72,37ns.

A lo largo de estos tests se utilizarán como referencia cinco imágenes para ver los ratios de compresión. Dos de ellas han sido generadas (Apéndice A) proceduralmente a fin de demostrar la viabilidad de realizar pruebas con datos sintéticos.

Para su generación se ha utilizado un pluguin en `python` para el programa McEdit [39], que aprovecha el potente generador de terreno del juego Minecraft[1]. Cruzando los datos de este generador con los de liberías de firmas espectrales [34][35] conseguimos imágenes sintéticas muy similares a lo que se podría obtener de un terreno real.

Existe además la posibilidad de introducir artefactos como interferencia de ruido para ver cúal sería el comportamiento del algoritmo ante situaciones no idóneas.

Así, se han elegido las siguientes cinco imágenes para las pruebas, con dos siendo sintéticas:

JR *Jasper Ridge*, imagen de la reserva biológica del mismo nombre, cerca de San Francisco.

WTC *World Trade Center*, imagen tomada tras los atentados del 11-S en las torres gemelas.

CUP *Cuprite*, en Nevada. Esta imagen es de dimesiones $N_X = N_Y = 350, N_Z = 188$.

G1 Imagen generada sintéticamente con ruido (Apéndice A).

G2 Segunda imagen generada sintéticamente, esta vez sin ruido.

6.2. Análisis de los resultados

En esta sección tratamos los resultados obtenidos experimentalmente (Apéndice C). Algunas particularidades observadas son debidas a la FPGA donde se han realizado las pruebas. No obstante, el comportamiento general del algoritmo será, como veremos, adecuado para cualquier FPGA.

6.2.1. Según bandas utilizadas para la predicción

Nos fijamos en la Figura 6.1. Observamos que la ocupación, en cuanto a lógica, aumenta de manera lineal conforme aumentamos P. Esto es de esperar ya que con cada incremento de P, estamos añadiendo un elemento a los vectores $\boldsymbol{\omega}$ y $\boldsymbol{U}$, el cual tiene que ser multiplicado y posteriormente sumado. El resto del circuito se mantiene igual. El número de bloques lógicos ocupados oscila entre los 2000 y 4000, suficientemente bajo como para caber en cualquier FPGA.

En cuanto a memoria la cosa cambia. Para los recorridos BIL y BIP se mantiene constante rondando los 70 módulos necesarios, pero para el modo BSQ pasa de necesitar un único módulo con $P = 0$ a casi 1200 con $P = 6$. Esto podría ser problemático de no ser porque para valores de $P > 3$ la compresión (Figura 6.3a) no solo no mejora sino que empeora en algunos casos. Cualquier FPGA moderna tendrá memoria suficiente para $P < 2$, pero para vlaores de $P \geq 2$,

[1] `https://minecraft.net/es/`

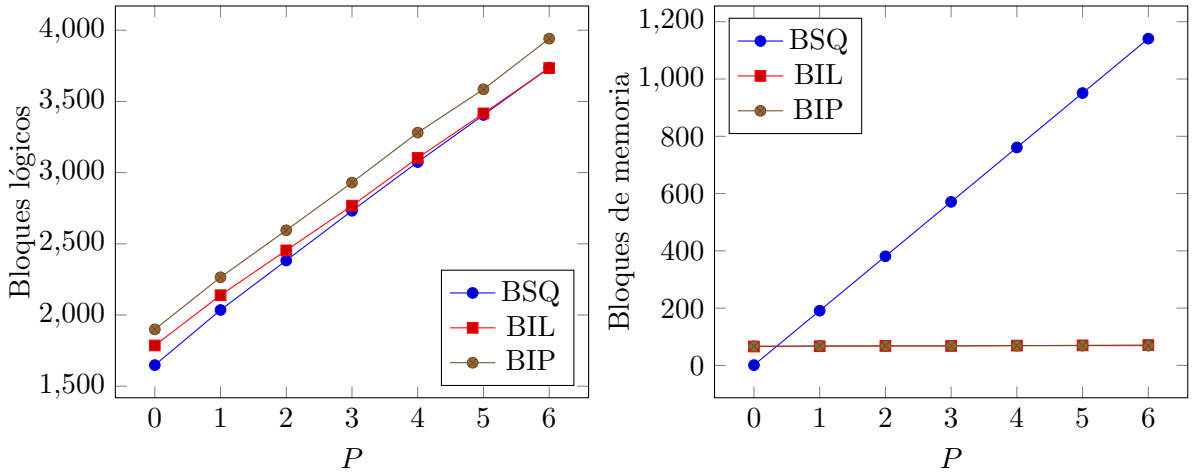


Figura 6.1: Evolución del consumo de recursos de la FPGA según P

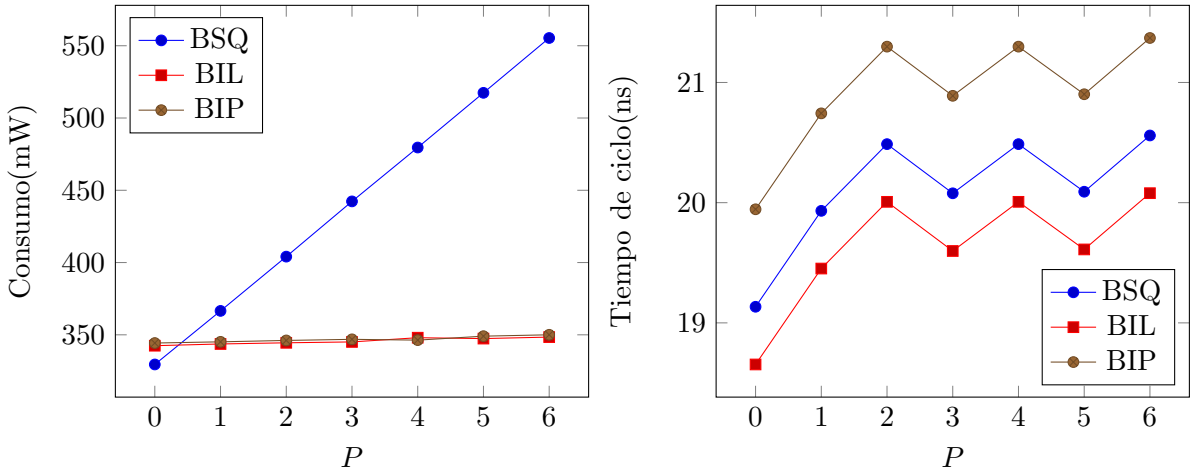
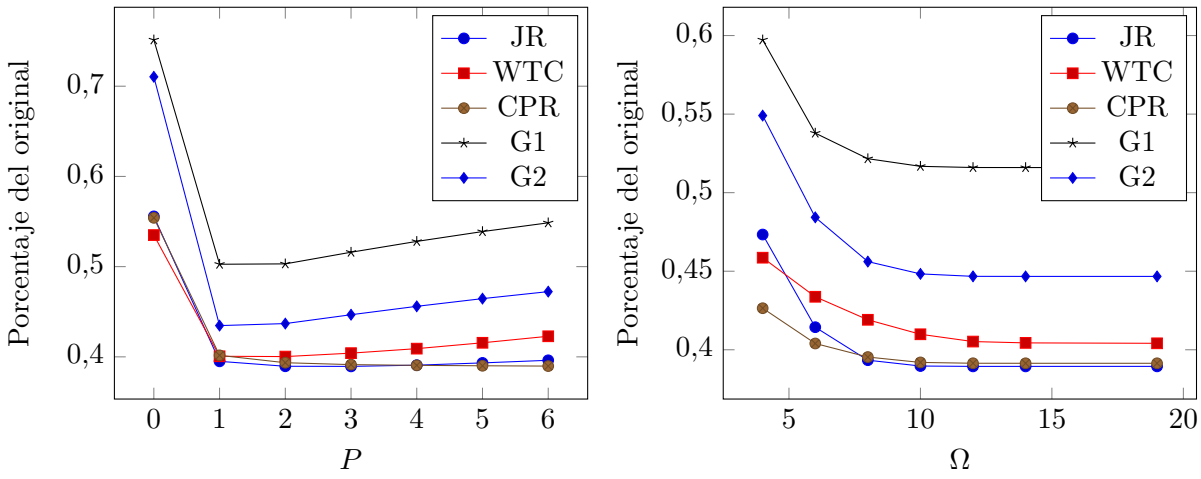


Figura 6.2: Evolución de ciclo mínimo y consumo según P



(a) Compresión según P

(b) Compresión según Ω

Figura 6.3: Tasa de compresión según diferentes valores de parámetros.

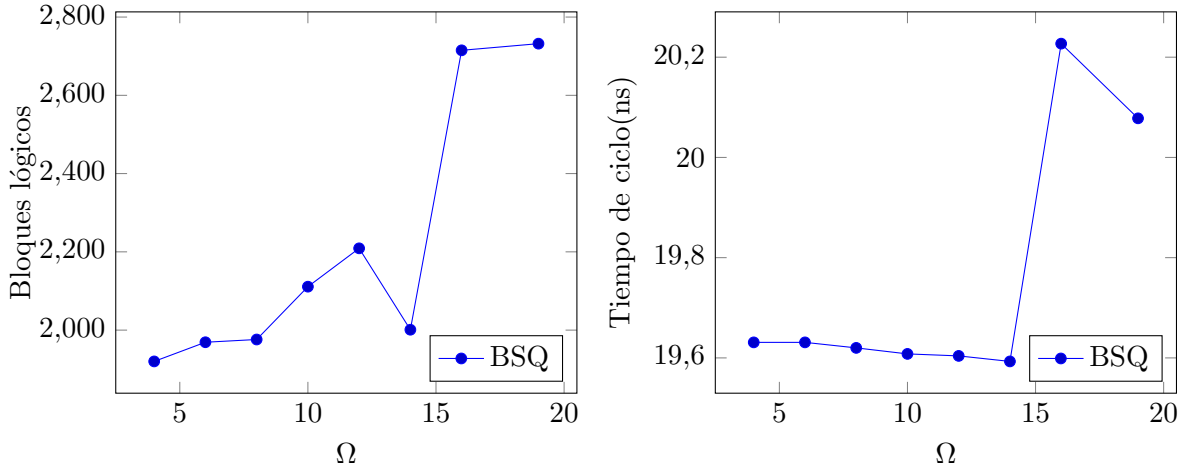


Figura 6.4: Evolución de los bloques lógicos utilizados y el tiempo de ciclo según Ω

el modo BSQ no puede implementarse debido a los requerimientos tan grandes de memoria. Nótese, eso sí, que para imágenes más pequeñas esto no sería un problema, pues la cantidad de memoria necesaria en BSQ es proporcional a $N_X \times N_Y$.

Tenemos también metadatos del circuito en la Figura 6.2, que nos indica algo crucial para las aplicaciones espaciales: el consumo. Mientras que con BIL y BIP somos capaces de afinar la predicción sin aumentar notablemente el consumo, para hacerlo en el modo BSQ el consumo se dispara, debido a que tenemos que mantener muchos más datos "vivos" en memoria. En cuanto al tiempo de ciclo, oscila sobre los $20ns$ y, teniendo en cuenta que necesitamos que sea menor que $72ns$, tenemos margen suficiente.

6.2.2. Según Ω

Variando el parámetro Ω, observamos en la Figura 6.3b una clara mejora de los ratios de compresión según crece, bajando desde compresiones de entre $45 - 60\,\%$ hasta el $40 - 50\,\%$ alcanzado a partir de $\Omega > 10$.

Por otro lado, en la Figura 6.4, vemos que la ocupación en placa y el tiempo de ciclo se mantienen más o menos estables hasta llegar a $\Omega = 16$, donde pegan un gran salto. Esto es debido a la placa concreta donde se han realizado los tests, ya que pasa a implementar parte de la lógica de los multiplicadores en placa en lugar de utilizar los DSP, con la consiguiente pérdida de rendimiento. En cualquier caso, no es un problema grave ya que tras el salto, los valores vuelven a estabilizarse, y además, dado que la compresión ya no mejora, se puede ajustar Ω a un valor previo al salto si se desea reducir el tamaño del circuito.

La cantidad de memoria necesaria y el consumo no se muestran pues están estabilizados en todas las configuraciones en los 571 módulos y $440mW(\pm 0{,}5\,\%)$ respectivamente.

6.2.3. Ocupación de placa según recorrido y predicción

Podemos observar en la Figura 6.5 el impacto que tiene el orden de muestreo de la imagen y los distintos modos de predicción sobre el circuito.

En primer lugar, se observa el gran impacto que tiene sobre el circuito el utilizar los modos más simples de predicción, reduciendo hasta un $39,32\,\%$ los recursos necesarios en cuanto a lógica se refiere. Por parte de la memoria, no se incluye gráfica pues se mantiene constante en 571 bloques para BSQ y 68 para BIL y BIP.

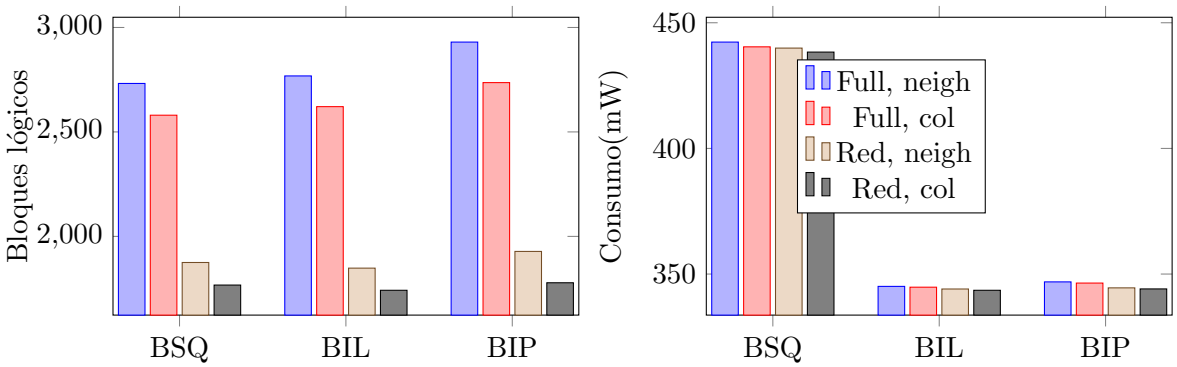


Figura 6.5: Bloques lógicos y memoria utilizada según recorrido de imagen y modos. Modos de predicción completo (full) o reducido (red), modo de suma local orientada a vecinos (neigh) o a columna (col).

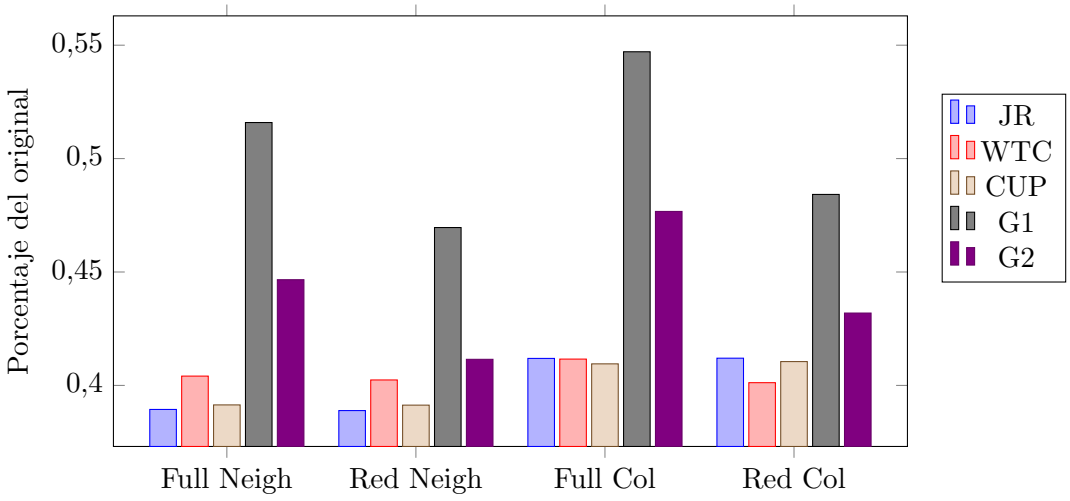


Figura 6.6: Compresión de las distintas imágenes según modos. Modos de predicción completo (full) o reducido (red), modo de suma local orientada a vecinos (neigh) o a columna (col).

El consumo, de nuevo, es bastante alto en BSQ debido al alto número de bloques de memoria necesarios para guardar las diferencias (recordemos que trabajamos con $P = 3$). Como no afecta a la compresión, lo más sensato será, siempre y cuando el sensor lo permita, realizar recorrido BIL o BIP.

En cuanto a la selección de un método u otro (Figura 6.6), se observa que las imágenes reales como mejor se comportan es en modo de predicción completo y con suma orientada a vecinos. Visto que no supone un sobrecoste muy grande (sí en porcentaje, pero no en valor absoluto), lo ideal será implementar estos modos de predicción.

6.2.4. Mejora del ciclo con segmentación

Una técnica muy simple pero que puede ayudar a mejorar mucho el tiempo de ciclo es la segmentación. Observando la disposición del circuito en la Figura 5.1, se evidencia la potencialidad de introducir dos registros intermedios. Uno que nos separe del compresor, y otro situado justo después del filtro de diferencias. Podemos ver cómo queda el circuito tras la segmentación en la Figura 6.7

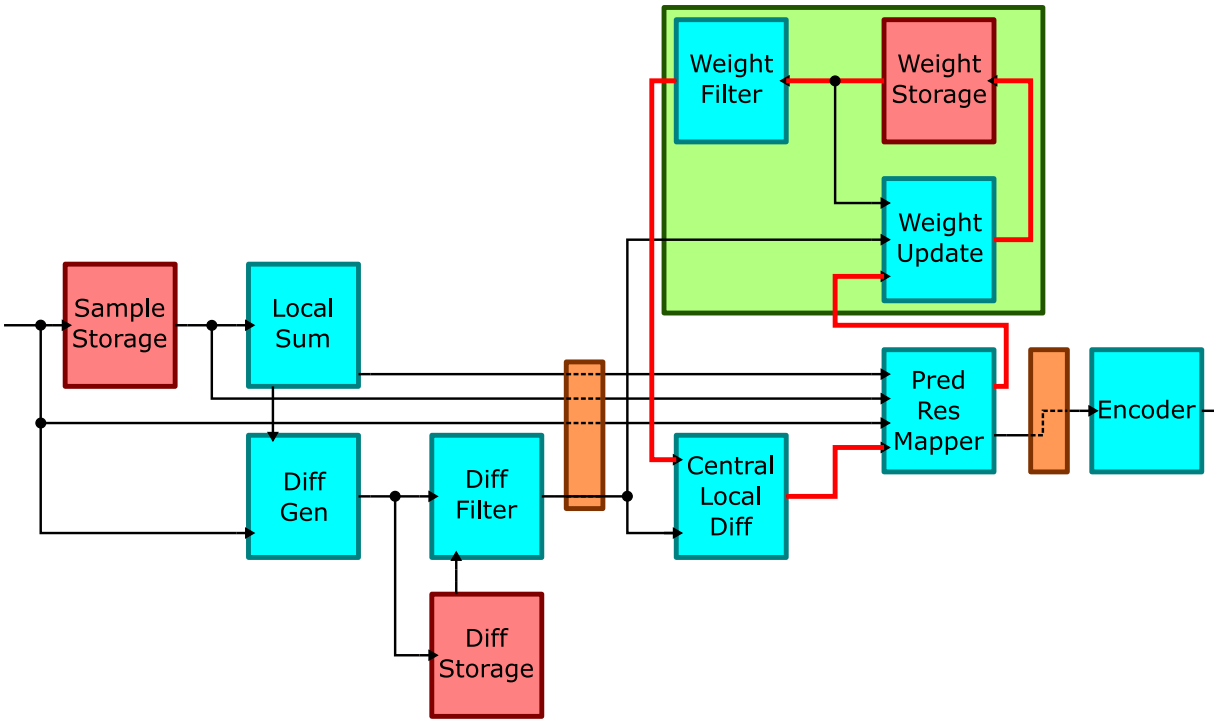


Figura 6.7: Disposición general del circuito segmentado. En azul los módulos combinacionales, en rojo los secuenciales. En naranja los registros para la segmentación, y en rojo el camino crítico.

El camino crítico, que nos limita la frecuencia de operación del circuito, no admite más segmentación debido a la necesidad de utilizar valores de una iteración previa en la siguiente, como veíamos en la sección 5.1.1.9. Por tanto no es posible mejorar el ciclo de reloj añadiendo más segmentación: estamos obligados a mejorar el camino crítico si queremos mejorar el circuito.

A la vista de la Figura 6.8, es claro que la segmentación mejora considerablemente el tiempo de ciclo, reduciéndolo hasta un 26,13 % del original. Además, es claro que segmentar el predictor sale "gratis" en cuanto a recursos, por tanto sería un punto de mejora clave al no introducir costes extra, si bien hay que tener en cuenta que los datos de salida se obtendrían con uno o dos ciclos de retraso.

No se introducen otros datos en esta sección pues el consumo se mantiene estable en torno a

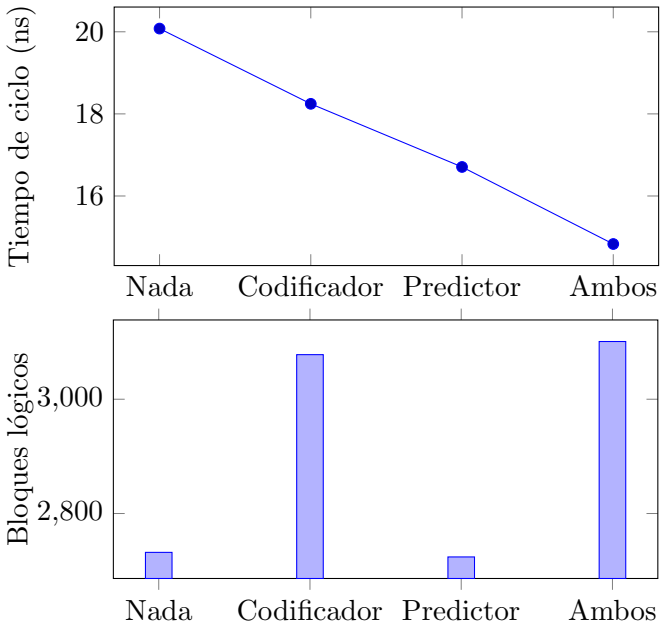


Figura 6.8: Tiempo de ciclo y bloques lógicos utilizados al introducir segmentación en el circuito.

los $441mW$, y los recursos de memoria utilizados son independientes de la segmentación.

6.2.5. Tamaño de la imagen y memoria

A la vista de la Figura 6.9, es claro el efecto que tienen las dimensiones de la imagen en cuanto a recursos de memoria necesarios.

En el modo BSQ, la cantidad utilizada depende directamente de N_X y N_Y, y en modo BIL y BIP depende de N_X y N_Z. El hecho de que la pendiente de BSQ sea mayor es porque depende además de P. Es decir, en caso de que $P \leq 1$, el consumo de memoria será similar al del los modos BIL y BIP, pero en el momento en que queramos afinar la compresión aumentando P, será mucho más rentable hacer el recorrido de otra manera diferente.

También observamos que el aumentar determinadas dimensiones puede no tener repercusión sobre los recursos utilizados en nuestro algoritmo. Por ejemplo, en los modos BIL y BIP, podemos aumentar indefinidamente la dimensión N_Y, y en el modo BSQ la dimensión N_Z, sin que tenga efecto alguno en el rendimiento, no siendo así para la coordenada N_X, que siempre fuerza un incremento de memoria.

Los bloques lógicos utilizados son, para cualquier dimensión, los mostrados en la Figura 6.1 para $P = 3$. Dado que sólo cambia la cantidad de memoria, pero los bloques lógicos se mantienen igual, los tiempos de ciclo oscilan todos en torno a los $20{,}1ns, 19{,}8ns, 21ns$ para BSQ, BIL y BIP respectivamente, al igual que muestra la Figura 6.2 para $P = 3$. El consumo, proporcional a la cantidad de memoria, será mayor en la configuración BSQ, de manera similar a como mostraba la mencionada Figura 6.2.

6.2.6. Según tipo de memoria

En las FPGAs, podemos decidir si las memorias se implementan en los bloques designados para ello o distribuidas entre los bloques lógicos de propósito general. Esta segunda aproximación es más costosa en cuanto a espacio pero proporciona reducciones de tiempo de ciclo. Introducimos además una tercera, gracias a la naturaleza de la memoria de este estándar, que

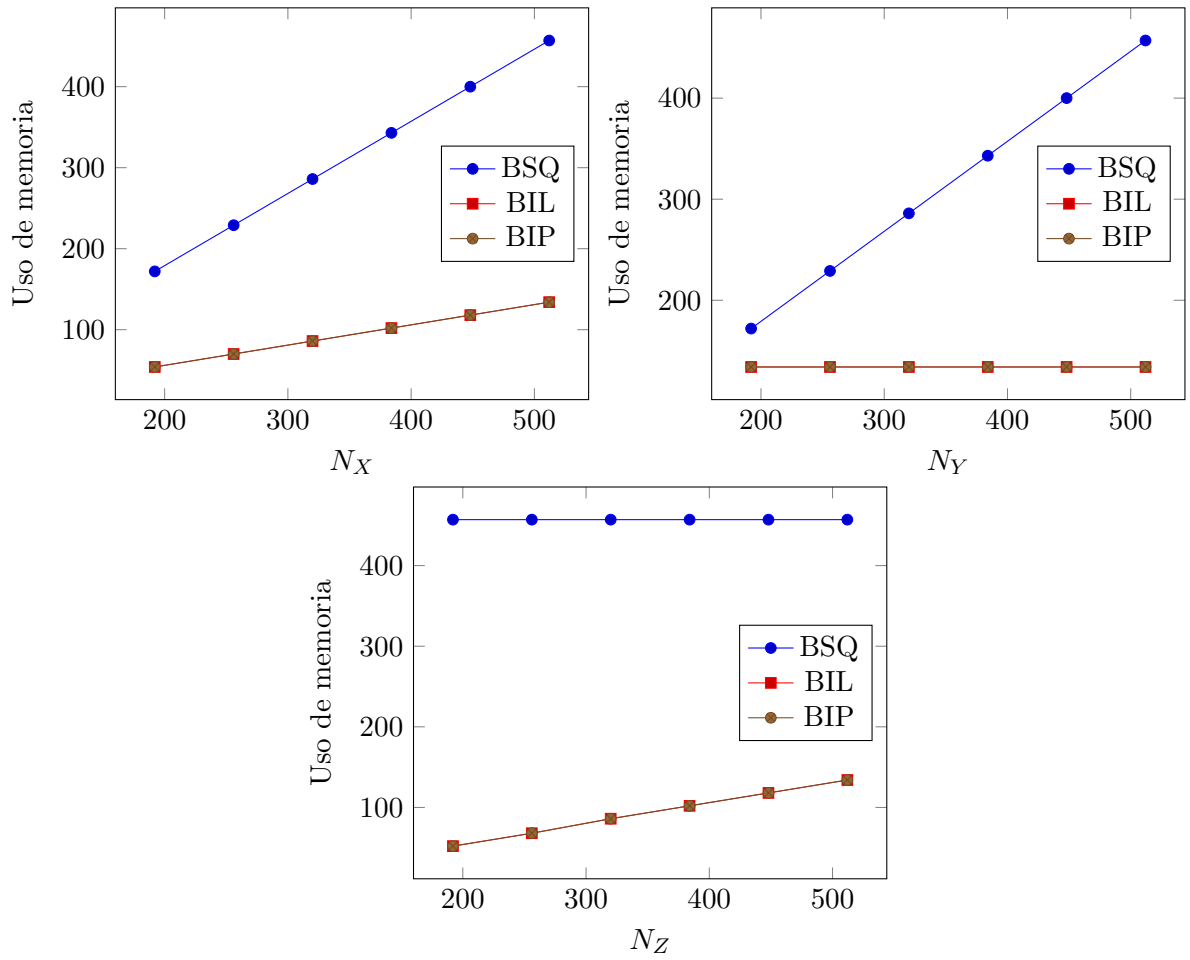


Figura 6.9: Uso de memoria según dimensiones de imagen a comprimir. Tamaño base es $512 \times 512 \times 512$, varía únicamente la dimensión indicada en el eje de abscisas. $P = 3$

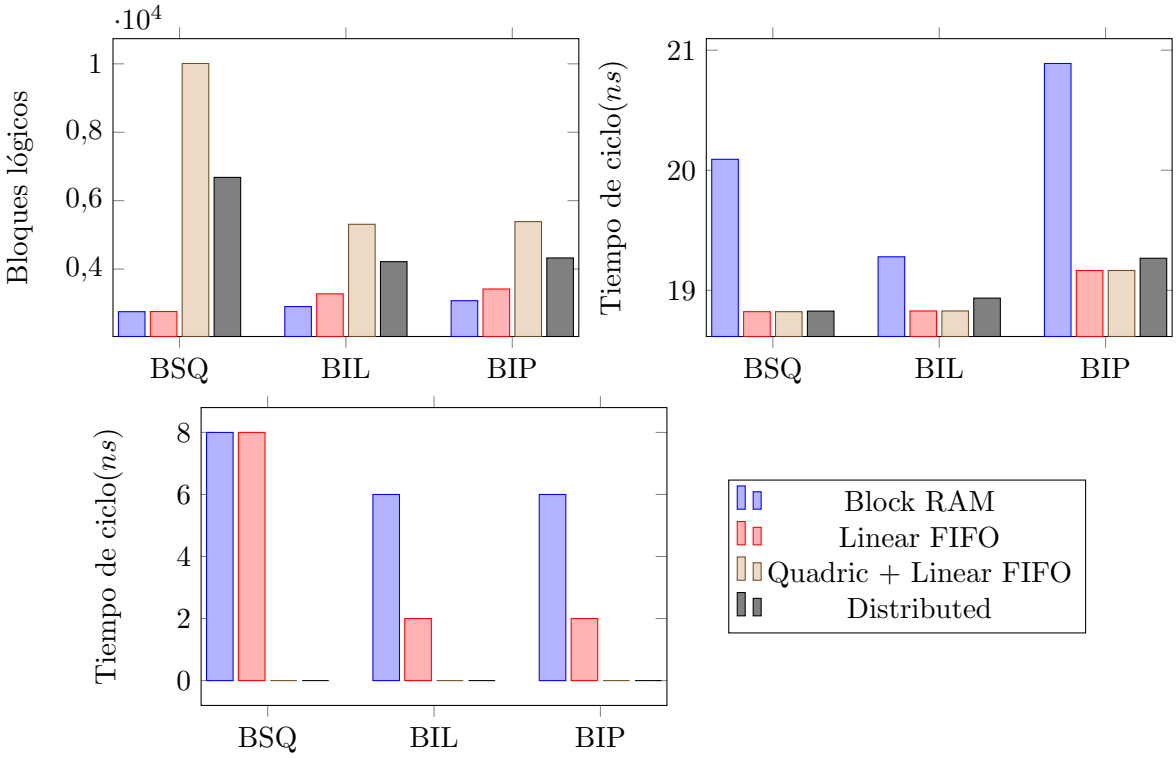


Figura 6.10: Tiempo de ciclo, bloques lógicos y bloques de memoria utilizados según distribución de la memoria. Dimensión de imagen $64 \times 64 \times 64$.

es su implementación en forma de cola FIFO con registros de desplazamiento.

En la versión de cola FIFO, podremos hacer que las memorias pequeñas (de tamaño del lado de la imagen), las memorias grandes (de tamaño del lado al cuadrado) o ambas se implementen como FIFO (ver secciones 5.1.1.2, 5.1.1.6 y 5.1.1.7). Podemos ver la comparativa de todas las aproximaciones en la Figura 6.10.

Los resultados revelan que pasar las memorias pequeñas de tamaño lineal a colas FIFO es la solución ideal. No incrementan excesivamente el uso de placa y consiguen una notable mejora del ciclo de reloj respecto a las memorias en bloque. Además reducen ligeramente la carga en memoria.

Pasar a colas también las memorias de tamaño cuadrático supone un gran sobrecoste para los recursos en placa, al igual que forzar implementación de memoria distribuida. Los tiempos de ciclo sí mejoran respecto a la implementación de memoria en bloque en similar medida a la solución anterior. Por tanto el único factor que podría ser decisivo es la cantidad de bloques de memoria disponibles, ya que estas aproximaciones no necesitan ninguno.

6.2.7. FPGA adaptada al espacio

Se observa en la Figura 6.11 los resultados obtenidos en una placa que podría resistir las condiciones en el espacio, donde un satélite tomaría las imágenes hiperespectrales.

El ciclo ha aumentado respecto a la *Virtex-7*, pero aun así, con un máximo de $42ns$, es suficiente para la compresión en tiempo real. La cantidad de lógica utilizada es muy similar, con el máximo de 2700 bloques muy por debajo del límite de 24576 de la placa. El consumo aumenta hasta los $800mW$ de media.

Sin embargo surge un problema. Las necesidades de memoria para el modo BSQ son excesivas para la capacidad de esta placa, a menos que se utilizase $P = 0$. Por tanto, siempre que sea

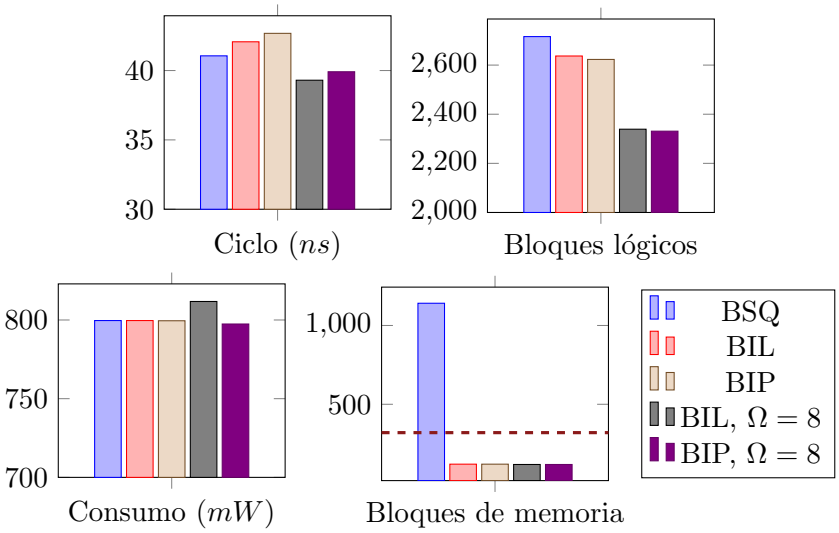


Figura 6.11: Ciclo, bloques lógicos y memoria necesaria para diferentes configuraciones en una placa preparada para el espacio. (*Virtex-4QV*)

posible, debería utilizarse uno de los modos BIL o BIP. En cuanto a variar otros parámetros, observamos que cambiar Ω no afecta al consumo u ocupación de la placa en exceso, y manteniendo un valor alto se consigue una mejor compresión (Figura 6.3b).

6.2.8. Con paralelización

El algoritmo CCSDS 123 comprime las bandas individualmente, dependiendo de muestras previas de la misma banda, y de datos obtenidos en las anteriores P bandas para el mismo pixel. Esto implica que existe la posibilidad de paralelizar el algoritmo y trabajar con varias bandas *a la vez*. De hecho, si la FPGA es suficientemente grande, podremos trabajar con **todas** las bandas a la vez.

Esto es posible sólo en el recordido BIP, que al darnos el pixel entero de golpe permitirá este cálculo simultáneo de los datos comprimidos para todas las bandas. También se podría con recorrido BIL introduciendo un buffer de reordenamiento a la entrada que nos diera las entradas como si fueran BIP. Esto supondría un coste en espacio del orden de $N_X \cdot N_Z$. Sin embargo, un recorrido en BSQ requeriría espacio del orden de $N_Z \cdot N_X \cdot N_Y$, para lo cual es más rentable guardar en memoria y releer en otro orden con posterioridad.

Simplemente replicando N_Z veces los diferentes módulos de cálculo, e interconectándolos según sea necesario, tendremos paralelizado el algoritmo. Además se debería conseguir (teóricamente) un tiempo de ciclo en FPGA similar al del algoritmo no paralelizado. Esto es debido a que los cálculos en una banda solo dependen de P bandas anteriores, por lo que las dependencias de datos sólo existen de manera local y no global entre bandas. Un esquema del entrelazado de módulos se observa en la Figura 6.12.

Los datos que compartirán entre las instancias del algoritmo se limitan a las diferencias $d_z(t)$, que ya no hará falta almacenar pues se utilizan todas en el mismo ciclo en que se generan. La cantidad de almacenamiento necesario tampoco cambia, pues al hacer este recorrido BIP paralelo únicamente necesitamos guardar las N_X muestras previas en cada banda, para un total de $N_Z \cdot N_X$.

Si bien el algoritmo original ya es capaz de comprimir en tiempo real las imágenes tomadas por el sensor AVIRIS, nunca está de más explorar hasta dónde se podría aumentar la velocidad del algoritmo. La tecnología no para de mejorarse, y es obvio que en un futuro se tomarán

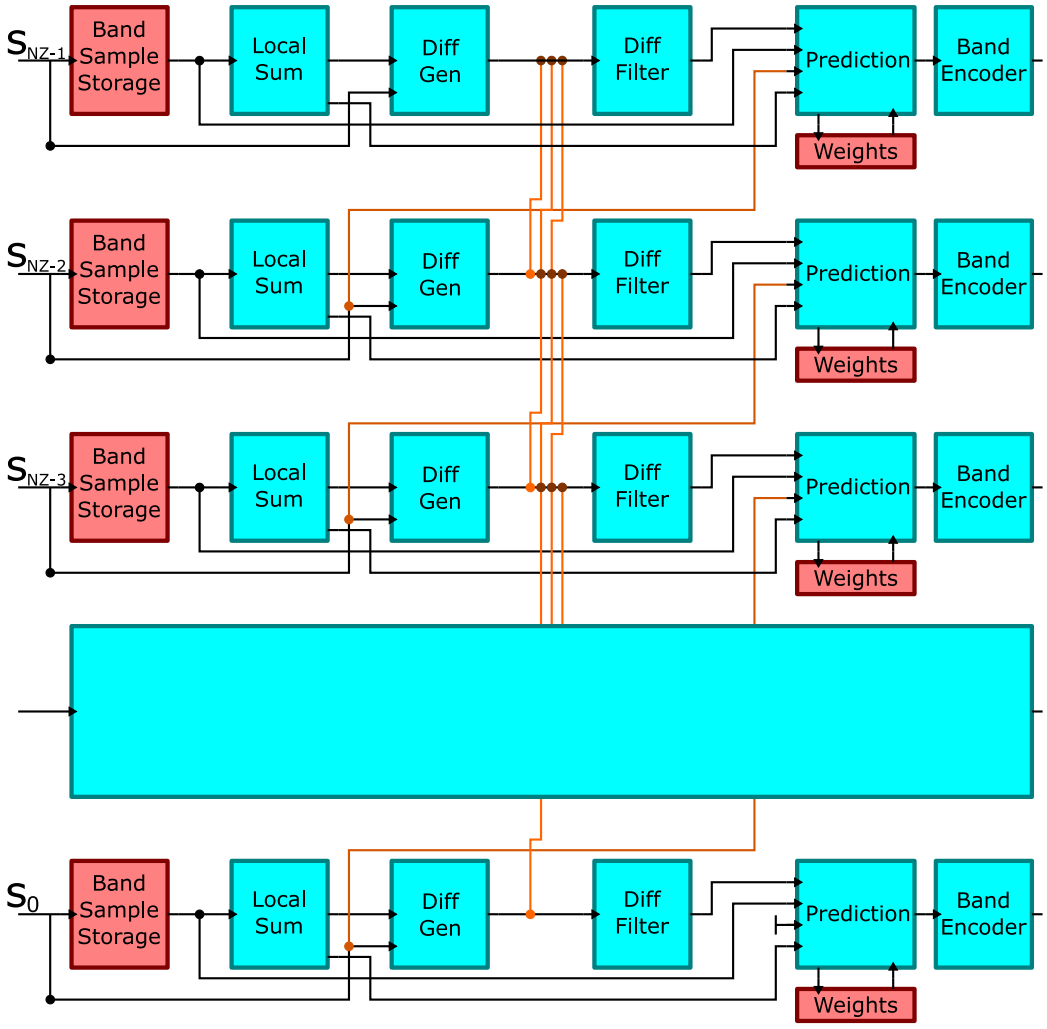


Figura 6.12: Esquema de interconexión para la paralelización del algoritmo. En naranja se destacan las conexiones extra. El gran bloque intermedio representa el circuito para las bandas intermedias, copia de los demás.

las imágenes más rápido y serán más grandes, o incluso se pase a tomar vídeo en lugar de instantáneas.

Cabría esperar que esta paralelización acelerase en un factor proporcional a N_Z la velocidad del algoritmo. Veamos los resultados:

Algoritmo	E/S	Tiempo de ciclo	Velocidad	Speedup
Secuencial (BIP)	$2B/ciclo$	20,889ns	95,744MB/s	–
Paralelo (BIP)	$446B/ciclo$	21,220ns	21,017GB/s	219,5×
	Celdas lógicas	RAM	I/O	DSPs
Secuencial (BIP)	2930	68	47 (paralelo)	6
Paralelo (BIP)	411137	112	678 (serie)	1344

Efectivamente, se consigue una mejora muy notable en la velocidad. Estamos comprimiendo del orden de 20 **GigaBytes** de datos por segundo. Teniendo en cuenta que la tasa de datos del sensor AVIRIS es de $27,64MB/s$, podemos afirmar que aún queda un largo recorrido para saturar de datos este algoritmo paralelo.

Quizá lo más notable de esta paralelización es que conseguimos un speedup que alcanza el 98 % del máximo teórico. Hay que notar, eso sí, que sería necesario un preprocesamiento antes de entrar los datos a la FPGA, pues ésta sólo tiene 850 pines de entrada, y si no serializamos las muestras, estaríamos hablando de un total de $(16+5+5+17)\cdot 224 = 9632$ pines necesarios.

6.3. Posibles cambios al algoritmo base

La parte más costosa del algoritmo es el cálculo que veíamos en las Ecuaciones (5.5) y (5.6). Modificando ligeramente los resultados (lo que implicará una menor compresión), podemos hacer este cálculo más rápido. En primer lugar observamos que:

$$\begin{aligned}\left\lfloor \frac{\text{mód}_R^*\left[\hat{d}_z(t) + 2^\Omega\left(\sigma_z(t) - 4s_{\text{mid}}\right)\right]}{2^{\Omega+1}} \right\rfloor &\approx \left\lfloor \text{mód}_{R-\Omega-1}^* \left[\frac{\hat{d}_z(t) + 2^\Omega\left(\sigma_z(t) - 4s_{\text{mid}}\right)}{2^{\Omega+1}}\right]\right\rfloor \\ &\approx \left\lfloor \text{mód}_{R-\Omega-1}^* \left[\frac{\hat{d}_z(t)}{2^{\Omega+1}} + \frac{\left(\sigma_z(t) - 4s_{\text{mid}}\right)}{2}\right]\right\rfloor\end{aligned} \tag{6.1}$$

Además dado que:

$$\hat{d}_z(t) = \boldsymbol{W}_z^T(t)\boldsymbol{U}_z(t) = \sum_{i=0}^{P^*-1} \omega_z^{(i)}(t)\cdot u_z^{(i)}(t) \tag{6.2}$$

Podemos deducir que la Ecuación (6.1) es similar a:

$$\begin{aligned}&\left\lfloor \text{mód}_{R-\Omega-1}^* \left[\frac{\sum_{i=0}^{P^*-1}\omega_z^{(i)}(t) * u_z^{(i)}(t)}{2^{\Omega+1}} + \frac{\left(\sigma_z(t) - 4s_{\text{mid}}\right)}{2}\right]\right\rfloor \\ \approx &\left\lfloor \text{mód}_{R-\Omega-1}^* \left[\sum_{i=0}^{P^*-1}\left(\frac{\omega_z^{(i)}(t)}{2^{\lfloor \Omega + 1/2\rfloor}}\cdot\frac{u_z^{(i)}(t)}{2^{\lfloor \Omega/2\rfloor+1}}\right) + \frac{\left(\sigma_z(t) - 4s_{\text{mid}}\right)}{2}\right]\right\rfloor\end{aligned} \tag{6.3}$$

Donde ganamos dos cosas:

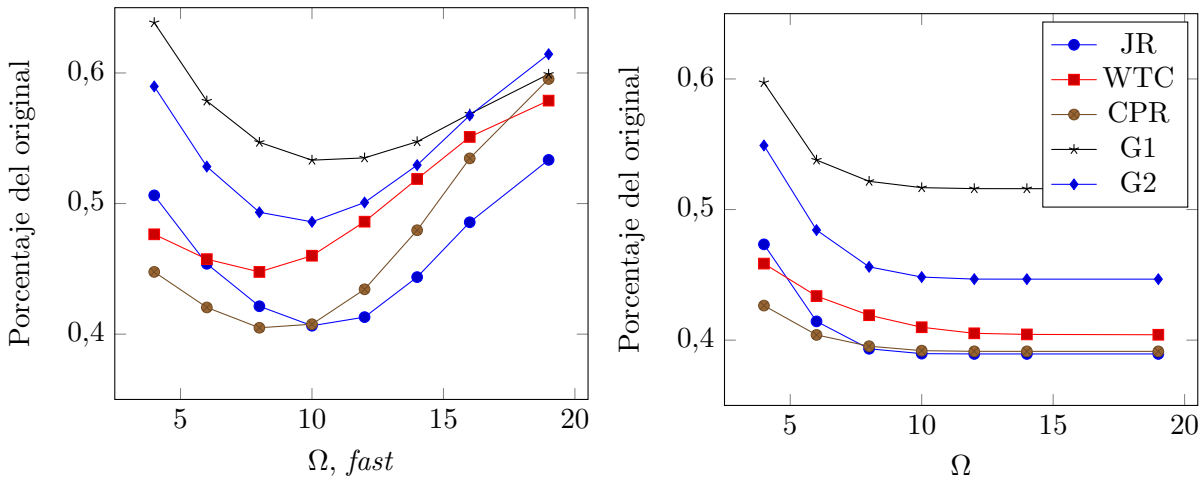


Figura 6.13: Evolución de compresión según Ω

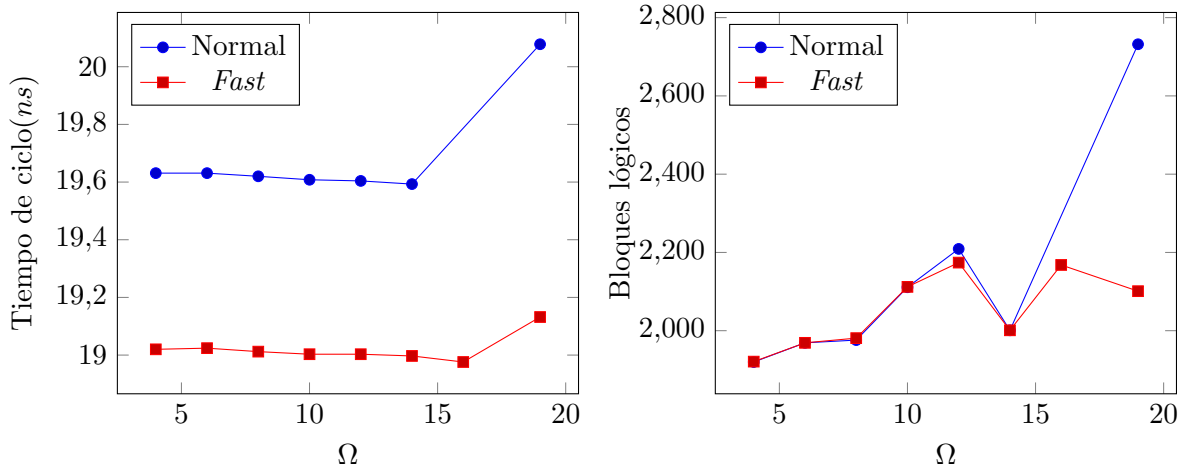


Figura 6.14: Recursos utilizados con el modo *fast* Ω

- Reducimos el tamaño de las multiplicaciones en $\Omega + 1$ bits.
- Reducimos el tamaño de los registros en $\Omega + 1$ bits.

Esto hace que el cálculo sea más rápido, si bien el resultado ya no es conforme con el estándar y habría que descomprimir teniendo en cuenta estas fórmulas. Si llamamos a esta configuración *fast*, podemos ver los cambios introducidos en las Figuras 6.13 a 6.15. Deducimos de las mismas que, si bien el ciclo es más rápido, la compresión empeora en un porcentaje mayor, por lo tanto sólo deberíamos introducir este cambio si el tiempo de ciclo resultase muy justo para nuestras necesidades.

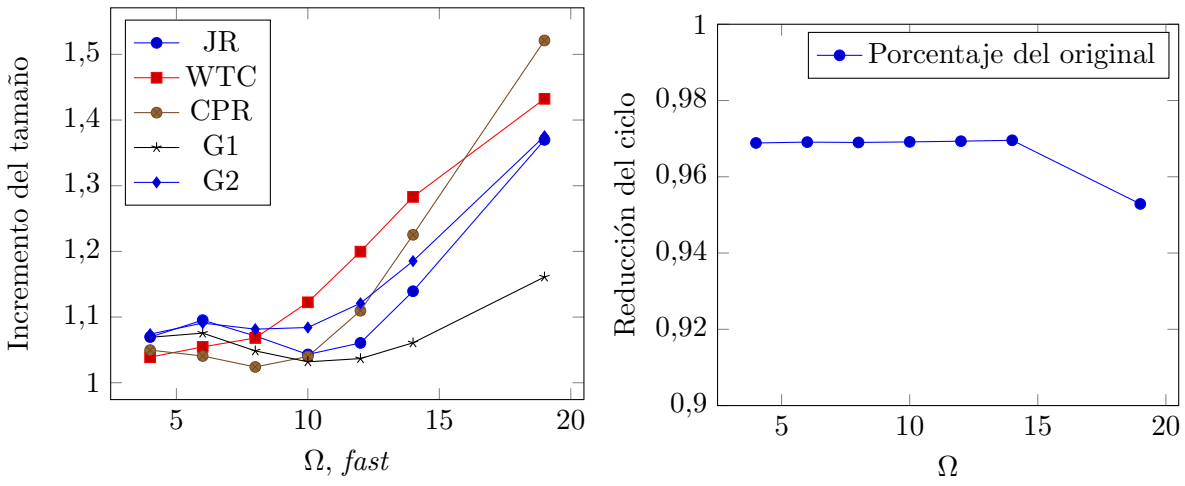


Figura 6.15: Cambios al utilizar el modo *fast*

Capítulo 7

Conclusiones

Con el desarrollo de las tecnologías en los últimos años, se ha hecho factible (y muchas veces necesario) el análisis y estudio de cantidades masivas de datos.

Las imágenes que tomamos para el análisis de la superficie terrestre han aumentando en tamaño y complejidad, pasando del blanco y negro a los cientos de bandas de una imagen hiperespectral.

Para transmitir tales cantidades de datos de manera eficiente es necesario realizar algún tipo de compresión. En general, cuando hacemos una foto, basta mantener la suficiente información como para que el ojo humano no se de cuenta de la reducción de datos, y podemos asumir pérdidas al comprimir. En el ámbito científico es importante mantener al completo la información que proporciona el sensor, para posibilitar un análisis exhaustivo de los datos.

Con ese fin surge el estándar CCSDS 123, diseñado para comprimir sin pérdida imágenes hiperespectrales. En este trabajo se ha realizado una implementación en VHDL, con el objetivo de ser utilizada en hardware reconfigurable tipo FPGA.

Se ha estudiado cómo los diferentes parámetros del algoritmo afectan a la ocupación en placa, al consumo energético, y a la compresión, a fin de encontrar la configuración idónea (secciones 6.2.1 a 6.2.3, 6.2.5 y 6.2.6). También se han propuesto posibles cambios o mejoras para aumentar la velocidad de compresión de cara a un futuro con sensores más rápidos (secciones 6.2.4, 6.2.8 y 6.3). Además se ha analizado en detalle la implementación en FPGA apta para el espacio, mostrando que tiene espacio suficiente para realizar la compresión (sección 6.2.7).

Los resultados son muy positivos. La velocidad de compresión es mucho mayor que la de toma de imágenes, por lo tanto existe margen para la mejora de sensores antes de tener que replantear el algoritmo. El consumo de energía es bastante inferior al vatio, precio muy barato a pagar cuando hablamos de una reducción en datos que ronda el $55 - 60\,\%$. La huella sobre la FPGA es mínima, con lo cual se pueden aprovechar los recursos sobrantes para otros análisis y experimentos.

Además, a fin de facilitar la realización de pruebas se ha programado un completo generador de imágenes sintéticas (Apéndice A) con el que se pueden crear imágenes bajo demanda simulando multitud de terrenos o escenas habituales en la superficie terrestre.

Es clara, pues, la factibilidad y ventajas de implementar el estándar CCSDS 123 en FPGA. Podemos comprimir con el máximo detalle y a tiempo real las imágenes más complejas, y aún sobran recursos sobre la placa para otros menesteres.

Los tiempos de ciclo obtenidos seguirán posibilitando la compresión para futuros sensores más rápidos y precisos que los actuales en varios órdenes de magnitud, todo ello manteniendo el bajo consumo necesario en un satélite.

Se enumeran a continuación los resultados sobre detalles más concretos:

- Los parámetros elegidos como óptimos para la compresión en [10], son también los más adecuados para la implementación sobre placa.
- En cuanto al consumo de recursos lógicos, es tan bajo que cualquier configuración del algoritmo cabrá en cualquier placa sin problema, por tanto debemos elegir la que proporcione mejor compresión.
- Sin embargo, la memoria es el factor limitante en el modo BSQ cuando incrementamos el valor de P. Si el sensor tomara la imagen en este modo, deberíamos sacrificar porcentaje de compresión reduciendo el valor de P hasta que la memoria necesaria quepa en placa.
- Entre los modos BIL y BIP la utilización de memoria es muy bajo. Ambos utilizan cantidad muy similar de recursos, y obtienen rendimientos similares de consumo y velocidad. Sensores que capturen en estos modos pueden aumentar sus resoluciones ya que el algoritmo tiene muy bajo coste para ellos.
- Es posible implementar el circuito en FPGAs aptas para uso espacial. Con un coste energético bastante bajo, conseguimos reducir la cantidad de datos transmitida a menos de la mitad. Esto en consecuencia reduce el coste energético de la transmisión.
- El circuito es suficientemente rápido para cubrir las necesidades de los sensores existentes hoy en día. No obstante, es posible realizar mejoras para adaptarse a un muy posible aumento de tamaño del sensor, velocidad de toma de datos, etc.
- Existe la posibilidad de implementar no sólo uno sino varios compresores sobre una misma placa. Esto permitiría utilizar mecanismos de redundancia para mitigar posibles errores en un entorno espacial, o incluso realizar la compresión simultánea de varias imágenes diferentes a la vez.

Apéndice A

Generación de imágenes

Uno de los mayores problemas a los que se enfreta quien trabaja con imágenes hiperespectrales es la escasez de estas. Aún es una tecnología bastante nueva, y la mayor parte de las imágenes tomadas no están disponibles al público general. Pero, como es obvio, para probar un algoritmo que trabaje sobre un conjunto de datos, necesitamos una muestra grande para comprobar que funciona bien.

La forma de generar datos para probar algoritmos suele ser aleatoria. Para algoritmos generales, donde la entrada es más o menos aleatoria, o que se comportan igual frente a entradas aleatorias y reales, basta generar un conjunto de datos aleatorios del tamaño y tipo necesario mediante funciones básicas de librería.

En el caso de las imágenes hiperespectrales, los datos no son totalmente aleatorios. Más bien todo lo contrario. Bandas de longitudes de onda similares son muy parecidas entre sí, y píxeles vecinos en la misma banda tampoco tienen variaciones abruptas. De hecho, es esta similitud la que aprovecha el algoritmo CCSDS 123 para realizar una compresión eficiente.

A.1. Primera aproximación

Así pues, la generación de datos de prueba no se puede realizar de cualquier manera, y tiene que respetar las características que encontraríamos en una imagen habitual, es decir:

- Es sacada desde un medio aéreo (satélite, avión,...), como vista cenital.
- Implicado por el primer punto, la temática principal es la naturaleza y entornos urbanos.
- No tiene excesivas variaciones de elementos a lo largo de la imagen, es bastante uniforme.

Pensando pues en una situación real, la imagen es, en cada pixel, un conjunto de las reflectancias del terreno que representa en las diferentes bandas adquiridas. Obviamente un pixel puede incluir más de un material, con lo que el resultado sería una media ponderada de las reflectancias de los diferentes materiales.

Existen bibliotecas [34] [35] que contienen la *firma espectral*[1] de diversos materiales encontrados en la naturaleza. Aprovechando estos recursos, una primera aproximación a la generación podría ser utilizando ruido perlin [36] para cada material, indicando procentajes sobre la totalidad del terreno. Estos mapas de ruido se combinarían uno a uno con la firma espectral de cada material para finalmente dar un resultado que cumpliría las características descritas.

[1] La firma espectral de un material viene dada por la interacción de un material frente a radiación electromagnética para diferentes longitudes de onda. Representa cuánta cantidad de radiación refleja el material.

El problema de esta aproximación es que, si bien es rápida y eficaz, no es excesivamente potente, pues todas las imágenes serían relativamente similares y suaves. Una solución (efectiva pero laboriosa) sería utilizar sistemas más avanzados de generación de ruido, que no fueran independientes para cada material sino que se interrelacionaran para crear escenas más realistas.

A.2. Segunda aproximación

En los últimos años, la popularidad de videojuegos donde los mapas son generados proceduralmente ha crecido exponencialmente [37]. El hecho de que cada vez que se juega sea una experiencia única aumenta el valor de juego, evitando la necesidad de comprar uno nuevo al terminarlo. Además, se suele permitir que incluso personas terceras puedan hacer el diseño de nuevos mapas e incluso los vendan.

Este nuevo paradigma de los videojuegos ha hecho que aparezcan numerosas herramientas para creación, edición y generación de mapas. Como siempre, el mundo que vemos es la inspiración para crear nuevo contenido, por tanto esta generación suele producir resultados muy similares a los que podemos ver en la naturaleza. Los algoritmos empleados están basados en generadores de ruido como los mencionados anteriormente. Pero la principal ventaja frente a los mencionados en la Apéndice A.1 es que ya están hechos.

Por tanto podemos aprovechar un generador de terreno de videojuegos que nos diseñará todos los mapas aleatorios que queramos, y además al ser procedural [38], podremos reproducir cuantas veces queramos el mismo resultado con solo guardar la semilla de generación.

No obstante, el terreno generado tiene un formato y características adaptadas al juego para el que se genera. Así pues necesitamos traducir e interpretar estos datos, además de transformarlos, obviamente, a una imagen hiperespectral.

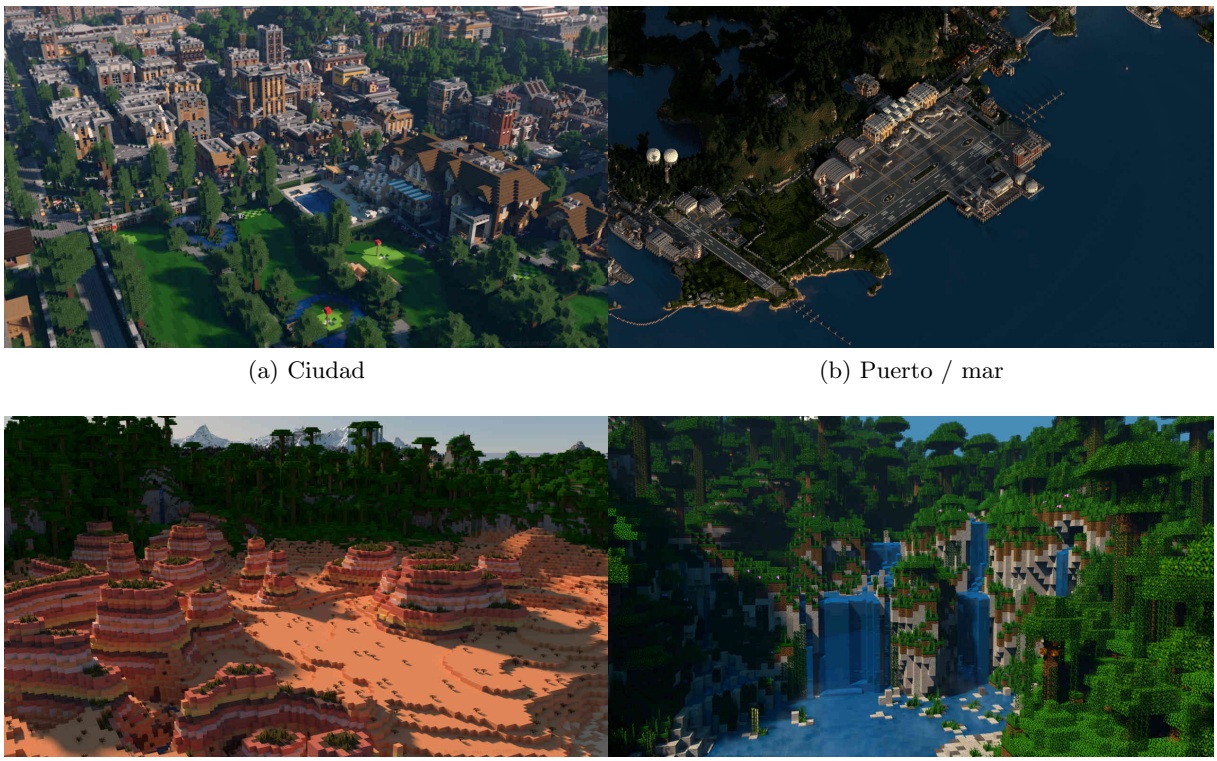
Para ello pensamos en cómo se toma (en el mundo real) una imagen de este tipo. Ya sea con avión o satélite, estamos hablando de imágenes tomadas cenitalmente de un terreno. En cada pixel, nos llega la luz reflejada de la sección de terreno que representa. Esta luz será una mezcla de la que reflejan los materiales individuales del píxel, de manera ponderada con la cantidad (en superficie) de cada uno.

En nuestra generación, por tanto, tendremos que hacer una partición de la vista cenital del mapa en secciones cuadradas del mismo tamaño. Para cada sección, miramos la cantidad de cada material y teniendo cada material mapeado a su firma espectral, hacemos una combinación de todos los materiales de la sección para generar el pixel hiperespectral correspondiente.

Con estas ideas en mente, lo único que nos falta por decidir es el videojuego cuyo generador utilizaremos. Hay numerosos juegos con estas características. Pero el que más destaca entre todos es Minecraft. ¿Por qué?:

- Generador de terreno altamente personalizable y variado.
- Gran cantidad de herramientas de terceros para generación y modificación de mapas libres.
- Formato de mapa extremadamente sencillo de utilizar. Los mapas se guardan como matrices tridimensionales donde cada elemento de la matriz indica el material en esa posición del mapa.
- Soporte de terceros para modificaciones directas del código del juego.

Podemos ver unos ejemplos de qué tipos de terrenos podríamos utilizar para generar las imágenes en la Figura A.1



(a) Ciudad (b) Puerto / mar

(c) Rocas (d) Jungla

Figura A.1: Ejemplos de terrenos desde donde obtener imágenes

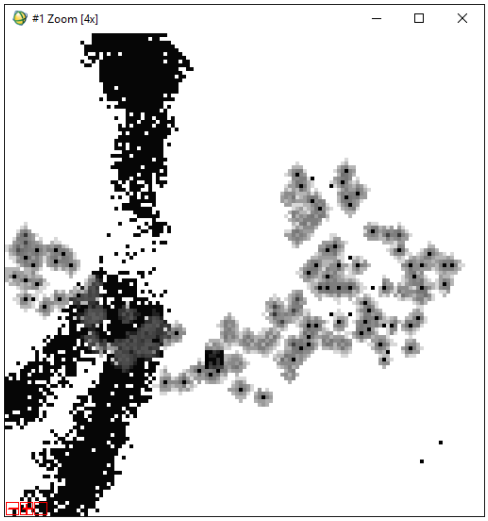
Es decir, no sólo es el juego con mejor generador de terreno, sino que además es el más sencillo de utilizar para nuestros fines. El proceso será el siguiente:

- Crear una biblioteca que asocie elementos del juego con sus firmas espectrales.
- Tomar un mapa ya generado, generar uno propio y/o modificarlo.
- Elegir una sección del mapa de la cual queremos extraer la imagen hiperespectral.
- Recorrer la sección elegida e ir cruzando los bloques individuales con la biblioteca generada anteriormente.
- Para cada pixel de salida, sumar ponderadamente los espectros de todos los materiales que contiene.
- Guardar los datos generados en un archivo, y los metadatos en otro para permitir su lectura desde visores de imágenes hiperespectrales como ENVI [41].

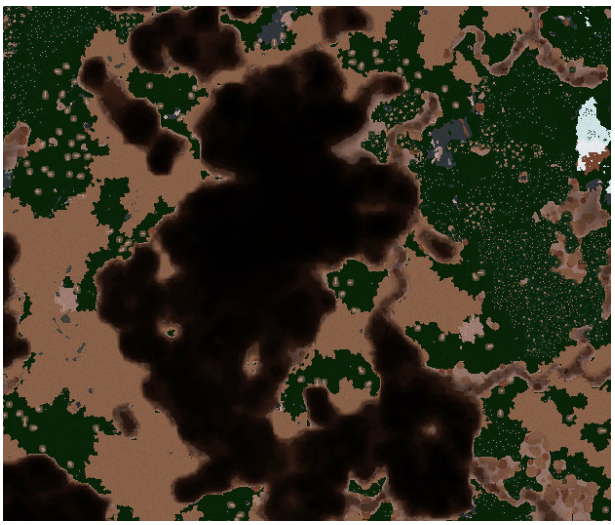
Una herramienta que nos facilitará enormemente el trabajo es el editor de mapas MCEdit [39]. Permite cargar pluguins en python y proporciona una interfaz para seleccionar el trozo de mapa que queramos y leer los datos de los bloques. Esto hace que el procesado de los ficheros de Minecraft se haga de manera totalmente transparente y no haya que preocuparse del parseo.

Otros programas como worldpainter [40] permiten una edición de mapas mucho más cómoda y a mayor escala que la incluida en el juego. Los podremos utilizar en caso de querer modificar el terreno a nuestro gusto para generar una imagen concreta sobre la que probar la compresión.

Con todo esto en mente, se ha diseñado el plugin de MCEdit *hyperspectral image generator*. Éste contiene una base de datos con las firmas espectrales de los materiales de los bloques de minecraft más comunes además de su transparencia (para por ejemplo si hay agua sobre tierra,



(a) Primera prueba con éxito. Imagen de 128x128x20. 30s para renderizar



(b) Simulación de imagen de AVIRIS. 512x614x224, 300s para renderizar

Figura A.2: Resultados de generaciones

tener en cuenta que se reflejarían ambas cosas), y es muy sencillo cambiarlas y/o añadir otras nuevas. Esto se hará traduciendo firmas espectrales de [34] a una base de datos en python admisible por el plugin.

Seleccionando la porción de terreno de la que queremos la imagen hiperespectral, el algoritmo generará el resultado como si una cámara hubiera tomado la foto de manera cenital. Como podemos elegir la firma de cada material, podremos simular escenas de otros astros asignando, por ejemplo, firmas de materiales lunares a los elementos del terreno.

Podemos configurar parámetros como el número de bandas a generar, la longitud de onda de las mismas, si queremos que se realice interpolación a posteriori para generar ruido (hay que tener en cuenta que los píxeles son de un único material, a menos que haya otro con transparencia encima), etc.

La salida consiste en dos archivos, una cabecera de imagen con los metadatos (ancho, alto, número de bandas...) y otro con los datos de reflectancia de cada uno de los píxeles de la imagen, ordenados en BSQ. Un par de ejemplos de generación pueden verse en la Figura A.2.

Apéndice B

Glosario de símbolos

Nota: La notación involucra para muchos símbolos el indicar a qué coordenada concreta se refieren. Por ejemplo $s_{z,y,x}$ se refiere a la muestra de la imagen en las coordenadas (z, y, x). En ocasiones será más cómodo utilizar la notación $s_z(t)$, donde $t = y * N_X + x$, o la notación $s^z_{y,x}$, utilizada por comodidad tipográfica.

Las tablas B.1 a B.3 indican el significado de los diferentes símbolos utilizados. Están extraídas de [9, D].

Símbolo	Significado	Referencia
BSQ	Recorrido secuencial por bandas	sección 2.1
BIL	Recorrido entrelazado por líneas	sección 2.1
BIP	Recorrido entrelazado por píxel	sección 2.1
x, y, z	Índices de coordenada de imagen	[9, 3.2.1]
t	Índice alternativo de coordenadas	[9, 3.4]
$s_{x,y,x}$	Muestra de imagen	[9, 3.2.1]
N_X, N_Y, N_Z	Dimensiones de la imagen	[9, 3.2.2]
D	Profundidad en bits de las muestras	[9, 3.3.1]
$s_{min}, s_{max}, s_{mid}$	Valores extremos y central de muestra	[9, 3.3.2]

Tabla B.1: Notaciones sobre la imagen

Símbolo	Significado	Referencia
P	Bandas utilizadas para la predicción	[9, 4.2]
P^*_z	Bandas para predicción desde la banda z	[9, 4.2]
C_z	Número de diferencias locales para predecir en la banda z	[9, 4.3.2]
$\sigma_{z,y,x}$	Suma local	[9, 4.4]
$d_{z,y,x}$	Diferencia local central	[9, 4.5.1]
$d^O_{z,y,x}$	Diferencias direccionales. $O = \{N, NW, W\}$	[9, 4.5.2]
$\boldsymbol{U}_z(t)$	Vector de diferencias	[9, 4.5.3]
Ω	Resolución de pesos	[9, 4.6.1]
$\omega_{min}, \omega_{max}$	Valores mínimos y máximo de pesos	[9, 4.6.1.3]
$\omega^O_z(t)$	Valores de pesos. $O = \{N, NW, W, (i)\}$	[9, 4.6.2]
Λ_z	Vector de inicialización de pesos	[9, 4.6.3.3]
Q	Resolución inicial de pesos	[9, 4.6.3.3]
$\tilde{s}_z(t)$	Valor de muestra predicho escalado	[9, 4.7.1]
$\hat{d}_z(t)$	Diferencia local central predicha	[9, 4.7.1]
R	Tamaño en bits del registro utilizado en predicción	[9, 4.7.1]
$\hat{s}_z(t)$	Valor predicho de muestra	[9, 4.7.2]
$e_z(t)$	Error de predicción escalado	[9, 4.8.1]

$\rho(t)$	Factor de escalado de pesos	[9, 4.8.2]
v_{min}, v_{max}	Factores de escalado de peso inicial y final	[9, 4.8.2]
t_{inc}	Intervalo de cambio del factor de escalado	[9, 4.8.2]
$\delta_z(t)$	Residuo de predicción mapeado	[9, 4.9]
$\Delta_z(t)$	Residuo de predicción	[9, 4.9]
$\theta_z(t)$	Menor distancia de $\hat{s}_z(t)$ a los extremos s_{min}, s_{max}	[9, 4.9]

Tabla B.2: Notaciones sobre el predictor

Símbolo	Significado	Referencia
$\Sigma_z(t)$	Acumulador	[9, 5.4.3.2.2]
$\Gamma(t)$	Contador	[9, 5.4.3.2.2]
γ_0	Exponente de contador inicial	[9, 5.4.3.2.2.2]
k'_z	Parámetros de inicialización del contador	[9, 5.4.3.2.2.3]
K	Constante de inicialización del contador	[9, 5.4.3.2.2.3]
γ^*	Factor de reescalado del contador	[9, 5.4.3.2.2.4]
U_{max}	Límite de longitud unaria	[9, 5.4.3.2.3.1]
$k_z(t)$	Parámetro de codificación de longitud variable	[9, 5.4.3.2.3.3]
$u_z(t)$	Tamaño del código unario	[9, 5.4.3.2.3.3]

Tabla B.3: Notaciones sobre el codificador

Apéndice C

Datos en crudo

Utilizamos diferentes configuraciones para las pruebas. Para la nomenclatura se han utilizado códigos de 2-3 caracteres separados por barras bajas, que indican los cambios realizados respecto a la configuración base definida en la sección 6.1. Sus significados son los siguientes:

Configuración	Detalles
`def`	Configuración por defecto (sección 6.1)
`bil`	Modo de imagen entrelazada por línea
`bip`	Modo de imagen entrelazada por píxel
`col`	Suma orientada a columnas
`red`	Predicción en modo reducido
`d8`	$D = 8$
`o4`	$\Omega = 4$. Otros valores de Ω son indicados de manera similar, con un número diferente acompañando a la "o"
`p0`	$P = 0$. Otros valores de P se indican análogamente, como en el caso anterior.
`par`	Modo paralelo para el recorrido BIP (sección 6.2.8)
`pip_enc`	Segmentación en el codificador
`pip_pre`	Segmentación en el predictor
`pip_enc_pre`	Segmentación en predictor y codificador
`064`	Tamaño de imagen de $64 \times 64 \times 6$
`128`	Tamaño de imagen de $128 \times 128 \times 128$
`512x512x512`	Tamaño de imagen de $512 \times 512 \times 512$, otros tamaños indicados de forma análoga, variando una de las dimensiones. Dimensiones dadas como $N_X \times N_Y \times N_Z$
`lin`	Uso de colas FIFO para las memorias de tamaño lineal respecto al lado de la imagen.
`qua`	Análogo al anterior pero para memorias de tamaño cuadrático
`dis`	Uso de RAM distribuida en lugar de en bloque
`fst`	Modo rápido de cálculo de la predicción
`spq`	En placa apta para el espacio

Tabla C.1: Nomenclatura para resultados

Para cada configuración, indicamos en la siguiente tabla los resultados obtenidos:

Configuración	Period	Logic	BRAM	DSP	Total	Dynamic	Static Power
Valor máximo	72ns	433200	1470	3600		(mW)	
`def`	20.078	2732	571	6	442.31	104.89	337.43
`bil`	19.598	2768	68	6	345.09	19.43	325.65

bip	20.889	2930	68	6	346.88	21.21	325.67
col	17.524	2580	571	6	440.42	103.01	337.41
col_bil	17.980	2621	68	6	344.77	19.12	325.65
col_bip	18.628	2736	68	6	346.41	20.74	325.67
red	14.528	1875	571	3	439.92	102.51	337.40
red_bil	15.589	1848	67	3	344.05	18.42	325.62
red_bip	15.930	1928	67	3	344.50	18.87	325.63
col_red	14.528	1767	571	3	438.33	100.95	337.38
col_red_bil	15.584	1742	67	3	343.54	17.92	325.62
col_red_bip	15.921	1778	67	3	344.08	18.46	325.62
d8	18.346	1481	331	6	392.56	60.80	331.76
o4	19.631	1920	571	6	439.99	102.59	337.40
o6	19.631	1969	571	6	440.31	102.90	337.41
o8	19.620	1976	571	6	440.64	103.23	337.41
o10	19.608	2111	571	6	439.98	102.58	337.40
o12	19.604	2209	571	6	440.86	103.45	337.41
o14	19.593	2001	571	6	440.46	103.06	337.41
d8__o8	18.348	1344	331	6	392.40	60.64	331.76
p0	19.134	1648	1	3	329.49	5.43	324.07
p1	19.932	2035	191	4	366.55	38.05	328.50
p2	20.487	2383	381	5	404.09	71.13	332.95
p4	20.487	3074	761	7	479.58	137.67	341.90
p5	20.091	3405	951	8	517.42	171.03	346.39
p6	20.559	3737	1141	9	555.36	204.46	350.90
p0__bil	18.654	1786	66	3	342.47	16.88	325.59
p1__bil	19.452	2139	67	4	343.67	18.05	325.62
p2__bil	20.007	2454	68	5	344.48	18.83	325.65
p4__bil	20.007	3103	69	7	348.00	22.29	325.71
p5__bil	19.611	3416	70	8	347.40	21.68	325.72
p6__bil	20.079	3734	71	9	348.43	22.68	325.75
p0__bip	19.945	1899	67	3	344.32	18.70	325.63
p1__bip	20.743	2265	68	4	345.15	19.50	325.66
p2__bip	21.298	2595	68	5	346.11	20.45	325.66
p4__bip	21.298	3281	69	7	346.33	20.64	325.69
p5__bip	20.902	3585	69	8	349.07	23.35	325.72
p6__bip	21.370	3941	69	9	350.02	24.29	325.73
par	21.220	411137	112	1344			
Segmentaciones del circuito							
pip_enc	18.245	3078	571	6	441.96	104.54	337.42
pip_pre	16.707	2724	571	6	442.07	104.65	337.42
pip_enc_pre	14.830	3101	571	6	442.33	104.91	337.43
Tamaño de imagen reducido ($128 \times 128 \times 128$)							
128	20.082	2779	29	6	343.40	18.59	324.80
128_bil	19.450	2931	12	6	337.12	12.74	324.38
128_bip	20.885	3117	12	6	337.75	13.36	324.38
128_col	17.394	2550	29	6	342.46	17.67	324.8
128_col_bil	17.914	2725	12	6	336.24	11.87	324.37
128_col_bip	18.632	2845	12	6	337.25	12.87	324.38
128_red	14.466	1761	29	3	341.49	16.71	324.79
128_red_bil	15.514	1867	11	3	334.70	10.37	324.33
128_red_bip	15.855	1951	11	3	334.69	10.36	324.33
128_col_red	14.466	1653	29	3	340.82	16.04	324.78

128_col_red_bil	15.532	1761	11	3	334.32	9.99	324.33
128_col_red_bip	15.868	1788	11	3	334.17	9.84	324.33
Diferentes memorias para tamaño ($64 \times 64 \times 64$)							
064	20.091	2755	8	6			
064_lin	18.822	2759	8	6			
064_qua	18.822	10010	0	6			
064_dis	18.827	6681	0	6			
064_bil	19.279	2904	6	6			
064_bil_lin	18.828	3277	2	6			
064_bil_qua	18.828	5310	0	6			
064_bil_dis	18.935	4217	0	6			
064_bip	20.889	3076	6	6			
064_bip_lin	19.165	3418	2	6			
064_bip_qua	19.165	5387	0	6			
064_bip_dis	19.267	4326	0	6			
Diferentes tamaños de imagen							
512x512x512_bsq	20.078	2667	457	6			
448x512x512_bsq	20.232	2673	400	6			
384x512x512_bsq	20.232	2553	343	6			
320x512x512_bsq	20.232	2668	286	6			
256x512x512_bsq	20.232	2626	229	6			
192x512x512_bsq	20.232	2540	172	6			
512x448x512_bsq	20.232	2642	400	6			
512x384x512_bsq	20.232	2639	343	6			
512x320x512_bsq	20.232	2639	286	6			
512x256x512_bsq	20.232	2625	229	6			
512x192x512_bsq	20.232	2625	172	6			
512x512x448_bsq	20.232	2642	457	6			
512x512x384_bsq	20.232	2642	457	6			
512x512x320_bsq	20.232	2642	457	6			
512x512x256_bsq	20.232	2641	457	6			
512x512x192_bsq	20.232	2641	457	6			
512x512x512_bil	19.806	2738	134	6			
448x512x512_bil	19.806	2769	118	6			
384x512x512_bil	19.806	2644	102	6			
320x512x512_bil	19.806	2766	86	6			
256x512x512_bil	19.752	2728	70	6			
192x512x512_bil	19.752	2634	54	6			
512x448x512_bil	19.806	2738	134	6			
512x384x512_bil	19.806	2738	134	6			
512x320x512_bil	19.806	2738	134	6			
512x256x512_bil	19.806	2733	134	6			
512x192x512_bil	19.806	2733	134	6			
512x512x448_bil	19.806	2738	118	6			
512x512x384_bil	19.806	2737	102	6			
512x512x320_bil	19.806	2737	86	6			
512x512x256_bil	19.752	2731	68	6			
512x512x192_bil	19.752	2731	52	6			
512x512x512_bip	21.065	2868	134	6			
448x512x512_bip	21.062	2899	118	6			
384x512x512_bip	21.052	2780	102	6			
320x512x512_bip	21.052	2896	86	6			

256x512x512_bip	21.047	2861	70	6			
192x512x512_bip	21.039	2774	54	6			
512x448x512_bip	21.065	2868	134	6			
512x384x512_bip	21.065	2868	134	6			
512x320x512_bip	21.065	2868	134	6			
512x256x512_bip	21.065	2863	134	6			
512x192x512_bip	21.065	2863	134	6			
512x512x448_bip	21.062	2868	118	6			
512x512x384_bip	21.056	2867	102	6			
512x512x320_bip	21.052	2867	86	6			
512x512x256_bip	21.047	2858	68	6			
512x512x192_bip	21.043	2858	52	6			
Versión rápida (pero menos precisa) de cálculo							
fst_o4	19.020	1921	571	6			
fst_o6	19.024	1969	571	6			
fst_o8	19.012	1981	571	6			
fst_o10	19.003	2112	571	6			
fst_o12	19.003	2174	571	6			
fst_o14	18.997	2001	571	6			
fst_o16	18.976	2168	571	6			
fst_o19	19.132	2101	571	6			
Sobre placa apta para el espacio							
Valor máximo	72ns	24576	320	512	-	-	-
spq_bsq	41.060	2716	1141	12	799.64	54.56	745.07
spq_bil	42.073	2637	120	12	799.64	54.56	745.07
spq_bip	42.682	2623	120	12	799.51	54.44	745.07
spq_bil_o8	39.303	2339	118	6	811.76	66.10	745.66
spq_bip_o8	39.913	2331	118	6	797.45	52.49	744.97

Tabla C.2: Datos experimentales para diferentes configuraciones

Y por último, los resultados de compresión de las diferentes imágenes utilizadas, definidas en la sección 6.1.

Imagen	JR	WTC	CUP	G1	G2
Tamaño original	140836864	140836864	46060000	140836864	140836864
def	54846456	56914536	18028136	72667392	62911596
red	54773468	56677392	18026048	66140884	57962328
col	58012136	57980188	18865396	77059656	67149256
red_col	58032212	56514120	18911592	68201200	60831092
Variando la resolución de pesos Ω					
o4	66664780	64589072	19642492	84111528	77329460
o6	58358488	61086372	18608556	75756412	68206768
o8	55400700	59023244	18210644	73463800	64239032
o10	54877468	57724300	18053264	72779204	63136960
o12	54844912	57071076	18028796	72673716	62917140
o14	54844708	56952036	18028220	72669816	62911676
o19	54846456	56914536	18028136	72667392	62911596
Variando las bandas para predicción P					
p0	78263560	75345112	25521112	105809424	100053068
p1	55635532	56426536	18501340	70791820	61213884
p2	54865728	56378568	18124324	70849560	61529952
p4	55038800	57617992	17989864	74356124	64222632

p5	55391960	58531848	17968948	75892120	65424872
p6	55798504	59547932	17953272	77257032	66517964
Variando Ω para el modo de cálculo rápido					
fst_o4	71302224	67102532	20619060	89930320	83052852
fst_o6	63922544	64422820	19366132	81480036	74406188
fst_o8	59345064	63045844	18648796	77026816	69480524
fst_o10	57243528	64793652	18774880	75097684	68445784
fst_o12	58170544	68464240	20005528	75347156	70532012
fst_o14	62488140	73069984	22089660	77089412	74565336
fst_o16	68402184	77603980	24624024	80085652	79927460
fst_o19	75128044	81521356	27423708	84370412	86530760

Tabla C.3: Tamaños comprimidos de las diferentes imágenes

Bibliografía

[1] *Cómo se concibió la fotografía: breve acercamiento a su génesis*, Retina Magazine, 24 de abril de 2009, http://www.retinamagazine.com/print.php?idnota=45

[2] Hannavy, John, ed. *Encyclopedia of Nineteenth-Century Photography* p. 365, Routledge, 2013, ISBN 1-135-87326-7.

[3] James Clerk Maxwell, *The theory of the primary colours*, British Journal of Photography, 9 de agosto de 1861.

[4] *La révélation des couleurs*, http://www.autochromes.culture.fr/

[5] Gareth A. Lloyd, Steven J. Sasson, *Electronic still camera*, Patente de U.S nº 4131919, https://www.google.com/patents/US4131919

[6] Michael Rowan-Robinson, *Night Vision: Exploring the Infrared Universe* p. 23, Cambridge University Press, 2013.

[7] *Historia de la cámara termográfica*, Academia Testo, 2010, http://www.academiatesto.com.ar/cms/historia-de-la-camara-termografica

[8] *Reusability: The key to making human life Multy-Planetary*, SpaceX news, 10 de junio de 2015, http://www.spacex.com/news/2013/03/31/reusability-key-making-human-life-multi-planetary

[9] *Lossless Multispectral & Hyperspectral Image Compression*, CCSDS 123.0-B-1, mayo de 2015.

[10] *Lossless Multispectral & Hyperspectral Image Compression*, CCSDS 120.2-G-1, diciembre de 2015.

[11] Carlos González Calvo, *Procesamiento a bordo de imágenes hiperespectrales de la superficie terrestre mediante hardware reconfigurable*, Tesis doctoral, UCM, 2011.

[12] *Raster Graphics*, Wikipedia, https://en.wikipedia.org/wiki/Raster_graphics

[13] *AVIRIS concept*, Jet Propulsion Laboratory, California Institute of Technology, http://aviris.jpl.nasa.gov/aviris/concept.html

[14] *AVIRIS instrument*, Jet Propulsion Laboratory, California Institute of Technology, http://aviris.jpl.nasa.gov/aviris/instrument.html

[15] *About CCSDS*, http://public.ccsds.org/about/default.aspx

[16] M. Klimesh, *Low-Complexity Lossless Compression of Hyperspectral Imagery via Adaptive Filtering*, IPN Progress Report 42-163, 15 de noviembre de 2005.

[17] Cutler, C.C., *Differential quantization of communication signals*, Patente de U.S. nº 2,605,361, 29 de julio de 1952, https://www.google.com/patents/US2605361

[18] B. Widrow and M. E. Hoff, Jr, *Adaptive Switching Circuits*, IRE WESCON, agosto de 1960, P. 96-104.

[19] Ioan Tabus, *Variants of the LMS algorithm*, 2015-2016, https://www.cs.tut.fi/~tabus/course/ASP/SGN2206LectureNew5.pdf

[20] Varios autores, *Least mean squares filter*, Wikipedia, https://en.wikipedia.org/wiki/Least_mean_squares_filter

[21] Golomb, S.W, *Run-length encodings*. IEEE Transactions on Information Theory, IT–12(3):399–401, 1966, http://urchin.earth.li/~twic/Golombs_Original_Paper/

[22] Marcelo J. W, Gadiel Seroussi, Guillermo Sapiro, *The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS*, HP labs, http://www.labs.hp.com/research/info_theory/loco/HPL-98-193R1.pdf

[23] A. Kiely, *Selecting the Golomb Parameter in Rice Coding*, IPN Progress Report 42-159, 15 de noviembre de 2004, http://ipnpr.jpl.nasa.gov/progress_report/42-159/159E.pdf

[24] *History of FPGAs*, c. 2004, https://web.archive.org/web/20040822113650/http://filebox.vt.edu/users/tmagin/history.htm

[25] Varios autores, *Field-programmable gate array, Modern developments*, Wikipedia, https://en.wikipedia.org/wiki/Field-programmable_gate_array#Modern_developments

[26] *UltraScale+ FPGAs, Product Tables and selection guide*, Xilinx, 2015-2016, http://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf#VUSP

[27] *Accelerating High-Performance Computing With FPGAs*, Altera, octubre 2007, https://www.altera.com/en_US/pdfs/literature/wp/wp-01029.pdf

[28] *H.264 Encoder - Micro Footprint* A2e Technologies, http://www.a2etechnologies.com/products.html, http://www.xilinx.com/products/intellectual-property/1-1lk9j1.html

[29] *Aplicaciones Generales de una FPGA*, Genera Technologies, 2008, http://www.generatecnologias.es/aplicaciones_fpga.html

[30] Jose A. Torres, *Package Math Real*, 1993, https://www.csee.umbc.edu/portal/help/VHDL/math_real.vhdl

[31] GICI group, *Empordá*, Octubre 2011, http://www.gici.uab.es

[32] Luca Fossati, *Compressor for the CCSDS 123 Multispectral & Hyperspectral Image compression upcoming standard* 12 de Abril de 2012.

[33] Xilinx, *Virtex 7 series, Virtex 4QV family overview*, 2014-2015, http://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf, http://www.xilinx.com/support/documentation/data_sheets/ds653.pdf.

[34] *ASTER spectral library*, California institute of Technology, http://speclib.jpl.nasa.gov/search-1

[35] R. N. Clark, G. A. Swayze, R. Wise, K. E. Livo, T. M. Hoefen, R. F. Kokaly, and S. J. Sutley, *USGS Digital Spectral Library splib06a* U.S. Geological Survey, Data Series 231. 2007, http://speclab.cr.usgs.gov/spectral.lib06/

[36] Perlin, Ken *An Image Synthesizer*, SIGGRAPH Comput. Graph. 19 (0097-8930): 287—296

[37] Tracy McVeigh, *Minecraft: how a game with no rules changed the rules of the game for ever* The Guardian, 16 de noviembre de 2013, https://www.theguardian.com/technology/2013/nov/16/minecraft-game-no-rules-changed-gaming

[38] Varios Autores, *Procedural generation* Wikipedia, 2016, https://en.wikipedia.org/wiki/Procedural_generation

[39] Codewarrior, *MCEdit: World Editor for Minecraft*, 2016, http://www.mcedit.net/

[40] *World Painter*, pepsoft, 2016, http://www.worldpainter.net/

[41] *ENVI*, Harris Geospatial Solutions, 2016, http://www.harrisgeospatial.com/ProductsandSolutions/GeospatialProducts/ENVI.aspx

[42] Graham Edgecombe, *Minecraft ID List* 2010-2016, http://minecraft-ids.grahamedgecombe.com/