Arquitectura basada en lógica reconfigurable para compresión de imágenes hiperespectrales

Julián Caba[1], María Díaz[2], Jesús Barba[1], Raúl Guerra[2], José A. de la Torre[1], Fernando Rincón[1], Sebastián López[2] y Juan Carlos López[1]

Resumen— **Uno de los principales problemas a la hora de capturar imágenes hiperespectrales a bordo de un vehículo aéreo no tripulado es el alto coste energético, y por tanto es el factor limitante del tiempo de vuelo. Este artículo presenta una implementación altamente optimizada de un acelerador FPGA del novedoso algoritmo *HyperLCA*. Este algoritmo ha sido adaptado utilizando aritmética de punto fijo con el fin de obtener una solución eficiente para este tipo de tecnología y es capaz de comprimir imágenes en tiempo real. A su vez, se ha analizado cuidadosamente en términos de rendimiento y eficiencia energética, comparando las versiones mononúcleo y multinúcleo de la arquitectura presentada con tres implementaciones del algoritmo basadas en GPU y desplegadas en tres plataformas: Jetson Nano, Jetson TX2 y Jetson Xavier NX. Los resultados demuestran que una solución basada en lógica reconfigurable alcanza niveles de rendimiento similares a los de una GPU de última generación, con una mayor eficiencia en términos de líneas procesados por vatio.**

Palabras clave— **hyperspectral imaging; lossy compression; FPGA; GPU; real-time performance; UAV; parallel computing**

I. Introducción

La tecnología hiperespectral está en pleno auge debido a que este tipo de sensores proporcionan una mayor riqueza de información espectral frente a otras alternativas tradicionales o basadas en tres componentes. Esta característica ha posicionado a las técnicas de análisis hiperespectral como la solución principal para el análisis de áreas terrestres y la identificación y discriminación de materiales superficiales visualmente similares. Como consecuencia, esta tecnología ha adquirido una gran relevancia, siendo ampliamente utilizada para una gran variedad de aplicaciones, como la agricultura de precisión, la monitorización medioambiental, la geología, la vigilancia urbana y la seguridad nacional, entre otras. Sin embargo, el procesamiento de imágenes hiperespectrales va acompañado de la gestión de grandes cantidades de datos, lo que afecta, por un lado, a su rendimiento en tiempo real y, por otro, a los requisitos de los recursos de almacenamiento a bordo. Además, los últimos avances tecnológicos están promoviendo la comercialización de cámaras hiperespectrales con mayores resoluciones espectrales y espaciales. Todo esto hace que el manejo eficiente de los datos, desde el punto de vista del procesamiento, la comunicación y el almacenamiento a bordo, sea aún más desafiante [1], [2].

Tradicionalmente, las imágenes captadas por los vehículos aéreos no tripulados (UAV) o por las plataformas espaciales de observación terrestre no son procesadas a bordo debido a la limitación de energía, que obliga a utilizar dispositivos de baja potencia, y que normalmente no tienen el mismo rendimiento que sus homólogos comerciales [3], [4], [5], [6], [7], [8]. En este sentido, las imágenes se descargan para su posterior procesado *off-line* en sistemas de alto rendimiento basados en Unidades Centrales de Procesamiento (CPU), Unidades de Procesamiento Gráfico (GPU), *Field-Programmable Gate Arrays* (FPGA), o arquitecturas heterogéneas. Lamentablemente, la transmisión de datos desde las mencionadas plataformas de observación introduce importantes retrasos debido a la gran cantidad de datos a transmitir y a la limitación del ancho de banda entre el dispositivo sensor y la plataforma de procesado [9], [6], [10]. Además, las mejoras ofrecidas por los fabricantes de sensorización hiperespectral hace que sea obligatorio alcanzar mayores ratios de compresión y llevar a cabo una actuación de compresión en tiempo real para evitar la acumulación innecesaria de grandes cantidades de datos sin comprimir, a la vez que se facilita la transferencia en los mismos de forma eficiente [11].

En este tipo de escenarios, donde el ancho de banda de la comunicación es determinante para la transmisión de grandes volúmenes de datos, se hace necesario pasar de enfoques de compresión sin pérdidas a técnicas de compresión con pérdidas [12], [13], [14], [15], [16]. Sin embargo, la mayoría de este tipo de compresores son generalizaciones de algoritmos de compresión de imágenes 2D o de vídeo ya existentes [17], que se caracterizan por su elevada carga computacional, sus intensos requisitos de memoria y su naturaleza no escalable. Estas características impiden su uso en aplicaciones con restricciones de potencia y recursos de hardware limitados, como es la compresión a bordo [18], [19].

En este artículo se presenta una arquitectura escalable sobre lógica reconfigurable del algoritmo *HyperLCA*, el cual fue desarrollado como compresor con pérdidas para imágenes hiperespectrales. Dicho algoritmo proporciona un buen rendimiento de compresión con una carga computacional razonable. A continuación se listan las contribuciones de este trabajo.

- Diseño e implementación del algoritmo *HyperLCA* en un dispositivo de lógica reconfigurable

[1]Escuela Superior de Informática, Universidad de Castilla-La Mancha (UCLM), 13071 Ciudad Real, España, e-mail: {julian.caba,jesus.barba,joseantonio.torre, fernando.rincon, juancarlos.lopez}@uclm.es.

[2]Instituto de Microelectrónica Aplicada (IUMA), Universidad de Las Palmas de Gran Canaria (ULPGC), 35001 Las Palmas de Gran Canaria, España, e-mail: {mdmartin,rguerra,seblopez}@iuma.ulpgc.es.

mediante uso de síntesis de alto nivel (High-Level Synthesis, HLS).
- Análisis de la implementación desarrollada de acuerdo a la cantidad de datos que pueden ser procesados en paralelo.
- Comparación de la solución implementada frente a otras soluciones basadas en GPU en términos de líneas procesadas por vatio.

El resto de este trabajo se organiza de la siguiente forma. En las Secciones II y III se exponen las operaciones realizadas en el algoritmo *HyperLCA* y su adaptación a punto fijo, respectivamente. En la Sección IV se detalla la arquitectura desarrollada para este algoritmo sobre lógica reconfigurables. En la Sección V se realiza un análisis de la implementación hardaware y es comparada con otras alternativas basadas en GPU. Por último, en la Sección VI se exponen las principales conclusiones de este trabajo.

II. Algoritmo HyperLCA

El algoritmo *HyperLCA* es un compresor con pérdidas para imágenes hiperespectrales especialmente diseñado para aplicaciones basadas en sensores del tipo *pushbroom/whiskbroom*. Concretamente, esta solución basada en transformada espectral puede comprimir de forma paralela bloques de píxeles de la imagen, descartando la definición de restricciones espaciales entre ellos y por ende, evitando dependencias espaciales. Para ello, hace uso de técnicas de proyección ortogonal, en concreto, del método tradicional de Gram-Schmidt, en aras de obtener el conjunto de píxeles hiperespectrales que mejor representen al resto, $\mathbf{E} = [\mathbf{e}_n, n = 1, ..., BS]$, y su correspondiente vector de proyecciones ortogonales, $\mathbf{V} = [\mathbf{v}_n, n = 1, ..., BS]$.

Una de las características más significativas de este compresor es que permite fijar de antemano una relación de compresión mínima deseada y garantiza que al menos este mínimo se alcanzará para cada bloque. Además, el algoritmo *HyperLCA* preserva los rasgos espectrales más característicos de los píxeles de la imagen que son potencialmente más útiles para posteriores técnicas de análisis hiperespectral, como puede ser la detección de anomalías. A continuación se explican las cuatro etapas de las que se compone este algoritmo.

A. Etapa de inicialización

En primer lugar se debe determinar el número de píxeles, $\mathbf{e}_n$, y vectores de proyección, $\mathbf{v}_n$, que deben extraerse de cada bloque de la imagen ($\mathbf{M}_k$), de acuerdo al ratio de compresión (CR), número de bits por píxel (N_{bits}) y el tamaño del bloque (BS). Este parámetro se obtiene como muestra la Ecuación 1, donde DR es el número de bits por píxel y por banda.

$$p_{\max} \leq \frac{DR \cdot nb \cdot (BS - CR)}{CR \cdot (DR \cdot nb + N_{\text{bits}} \cdot BS)} \qquad (1)$$

Algoritmo 1 Transformación HyperLCA.

Entradas:
$\mathbf{M}_k = [\mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_{BS}]$, p_{max}
Salidas:
$\hat{\boldsymbol{\mu}}$; $\mathbf{E} = [\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_{p_{max}}]$; $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_{p_{max}}]$
Algoritmo:
1: Píxel medio: $\hat{\boldsymbol{\mu}}$;
2: Imagen Centralizada: $\mathbf{C} = \mathbf{M}_k - \hat{\boldsymbol{\mu}} = [\mathbf{c}_1, \mathbf{c}_2, ...\mathbf{c}_{BS}]$;
3: **for** $n = 1$ **to** p_{max} **do**
4: **for** $j = 1$ **to** BS **do**
5: Cálculo del brillo: $\mathbf{b}_j = \mathbf{c}'_j \cdot \mathbf{c}_j$;
6: **end for**
7: Brillo máximo: $j_{max} = \text{argmax}(\mathbf{b}_j)$;
8: Píxel extraidos: $\mathbf{e}_n = \mathbf{r}_{j_{max}}$;
9: $\mathbf{q}_n = \mathbf{c}_{j_{max}}$;
10: $\mathbf{u}_n = \mathbf{q}_n / b_{j_{max}}$;
11: Vector de Proyección: $\mathbf{v}_n = \mathbf{u}'_n \cdot \mathbf{C}$;
12: Sustracción de información: $\mathbf{C} = \mathbf{C} - \mathbf{q}_n \cdot \mathbf{v}_n$;
13: **end for**

B. Etapa de transformación espectral

El compresor *HyperLCA* se basa en una versión modificada del método de ortogonalización Gram-Schmidt para obtener una imagen comprimida y no correlacionada. Esta etapa se encarga de realizar la transformación espectral, que corresponde a la fase del algoritmo con mayor exigencia de cálculo y, por tanto, es la más propensa a ser acelerada.

La imagen hiperespectral será dividida en bloques de tamaño determinado de píxeles hiperespectrales (BS), sobre los cuales se realizará el conjunto de operaciones detallados en el Algoritmo 1. Como se puede observar, la entrada a esta etapa corresponde al bloque de imagen hiperespectral a comprimir, $\mathbf{M}_k$, y el número de píxeles hiperespectrales a extraer, p_{max}. Mientras que las salidas corresponden al píxel medio ($\hat{\boldsymbol{\mu}}$) del bloque entrante, al conjunto de los p_{max} píxeles hiperespectrales más diferentes ($\mathbf{E}$) y sus correspondientes vectores de proyección ($\mathbf{V}$). En primer lugar, se obtiene el píxel medio ($\hat{\boldsymbol{\mu}}$) del bloque hiperespectral de entrada, que es utilizado para centralizar dicho bloque mediante la resta de dicho píxel con cada uno de los píxeles hiperespectrales que contiene el bloque, obteniendo como resultado una versión centralizada del bloque de entrada, $\mathbf{C}$ (línea 2). En segundo lugar, se extraen secuencialmente los p_{max} píxeles más característicos (cuerpo del *for* de la línea 3). En este proceso iterativo, el brillo de cada píxel de la imagen se calcula mediante el producto escalar de cada píxel de la imagen consigo mismo (véase la línea 5). Una vez obtenidos todos los brillos, se selecciona aquel con mayor brillo y se extrae su valor original del bloque de entrada ($\mathbf{M}_k$), almacenando el resultado en $\mathbf{e}_n$ (ver líneas 7 y 8). Después, los vectores ortogonales $\mathbf{q}_n$ y $\mathbf{u}_n$ se definen como se muestran en las líneas 9 y 10, respectivamente. Se emplea $\mathbf{u}_n$ para estimar la proyección de cada píxel de la imagen sobre la dirección del píxel seleccionado, $\mathbf{e}_n$, derivando en el resultado en el vector de proyección, $\mathbf{v}_n$ (línea 11). Finalmente, esta información se resta de $\mathbf{C}$ (línea 12). Estas operaciones se realizan p_{max} veces cambiando el bloque de entrada por el bloque resultante de la línea 12.

Tabla I: Resultados de compresión según el tipo de datos utilizado.

Nbits	BS	CR	SNR				MAD				RMSE			
			I12	I16	I32	F32	I12	I16	I32	F32	I12	I16	I32	F32
12	1024	12	43.01	38.01	43.12	42.75	24.50	39.00	25.00	25.50	3.12	5.55	3.08	3.22
		16	42.27	38.57	42.27	41.97	32.25	41.50	32.25	32.50	3.40	5.21	3.40	3.52
		20	41.31	39.13	41.31	41.06	41.25	46.50	41.25	41.25	3.73	4.88	3.80	3.91
	512	12	42.99	38.95	43.03	42.68	26.25	34.50	25.00	25.00	3.13	4.98	3.12	3.24
		16	42.45	39.36	42.47	42.14	30.75	38.00	30.00	30.50	3.33	4.75	3.33	3.45
		20	41.50	39.92	41.48	41.24	39.50	43.75	39.50	40.00	3.72	4.46	3.72	3.83
	256	12	43.03	40.00	43.02	42.67	25.00	34.50	25.00	25.50	3.12	4.42	3.12	3.25
		16	42.33	40.51	42.32	42.02	34.75	37.25	34.75	35.00	3.38	4.16	3.38	3.50
		20	40.94	40.59	40.59	40.73	54.75	57.75	54.75	54.75	3.96	4.13	3.97	4.06
8	1024	12	41.80	36.91	42.19	41.68	23.25	41.50	22.00	22.25	3.62	6.32	3.47	3.69
		16	41.63	37.42	41.73	41.27	25.50	41.25	25.50	26.50	3.69	5.95	3.65	3.86
		20	41.20	37.89	41.20	40.79	32.25	42.00	32.50	32.25	3.87	5.64	3.87	4.07
	512	12	42.49	37.99	42.70	42.26	23.00	38.00	22.75	22.25	3.33	5.57	3.25	3.42
		16	42.07	38.64	42.09	41.69	27.50	36.00	26.50	26.75	3.49	5.17	3.48	3.65
		20	41.61	39.01	41.60	41.25	31.25	39.50	30.50	31.75	3.68	4.95	3.68	3.84
	256	12	42.79	39.35	42.82	42.39	24.75	35.50	25.00	25.00	3.20	4.76	3.19	3.36
		16	42.15	39.96	42.13	41.76	30.00	37.00	29.75	29.75	3.45	4.43	3.46	3.61
		20	41.80	40.21	41.78	41.44	35.75	37.00	35.75	36.00	3.59	4.31	3.60	3.74

C. Etapa de preprocesado

La salida de la anterior etapa debe ser adaptada antes de su codificación. En un primer paso, se deben escalar los valores de las proyecciones ($\mathbf{V}$), que representan la proyección de cada píxel dentro de $\mathbf{M}_k$ sobre la dirección abarcada por cada vector ortogonal, $\mathbf{u}_n$, en cada iteración. Por lo tanto, los valores de cada elemento de $\mathbf{V}$ están en el rango de $(-1, 1]$. Sin embargo, la siguiente etapa de *codificación de entropía* trabaja exclusivamente con números enteros. En consecuencia, los elementos dentro de $\mathbf{V}$ deben ser escalados como se muestra en la Ecuación 2. Posterioremente se redondean a los valores enteros más cercanos.

$$v_{j_{\text{scaled}}} = (v_j + 1) \cdot (2^{N_{\text{bits}}-1} - 1) \tag{2}$$

La etapa de codificación también extrae las redundancias dentro de los datos en el dominio espectral para asignar la longitud de palabra más corta a los valores más comunes. Como consecuencia, la relación de compresión alcanzada en la etapa de transformación espectral podría incrementarse aún más. Para ello, los vectores de salida de la etapa anterior ($\hat{\boldsymbol{\mu}}$, $\mathbf{E}$ y $\mathbf{V}$ escalado) son procesados individualmente y transformados para que estén compuestos exclusivamente por valores enteros positivos más cercanos a cero, siguiendo las recomendaciones dictadas para la predicción de errores en los *Blue Books del Consultative Committee for Space Data Systems (CCSDS)* [20].

D. Etapa de codificación

Por último, cada vector de salida se codifica de forma individual siguiendo una estrategia de codificación de entropía sin pérdidas basada en el algoritmo Golomb-Rice. Para ello, el parámetro de compresión, M, se estima como el valor medio del vector objetivo. Después, cada uno de sus elementos se divide por M para obtener los resultados de la división, el cociente (q) y el resto (r). Por un lado, el cociente, q, se codifica mediante código unario. Por otro lado, el resto, r, podría codificarse utilizando $b = log_2(M)+1$ bits si M es potencia de 2. Sin embargo, M puede ser en realidad cualquier número entero positivo. Por esta razón, el resto, r se codifica como binario plano utilizando $b-1$ bits para valores de r menores que $2^b - M$, de lo contrario se codifica como $r + 2^b - M$ utilizando b bits.

III. Adaptación del algoritmo *HyperLCA*

La mayor parte del rendimiento de compresión alcanzado por el algoritmo HyperLCA se obtiene en la etapa de transformación espectral, diseñada originalmente con operaciones de punto flotante. Sin embargo, las FPGA son, en general, más eficientes tratando con operaciones de enteros. Además, la ejecución de operaciones de punto flotante en diferentes dispositivos puede producir resultados ligeramente diferentes. En concreto, se ha utilizado el concepto de punto fijo empleando aritmética de enteros y desplazamiento de bits [21]. En este contexto, hemos asumido que el sistema de sensorización hiperespectral codifica las imágenes hiperespectrales con un máximo de 12 bits, que es también el escenario más común [22], [23].

La Tabla I muestra una comparativa de la calidad de los resultados tras el proceso de compresión/descompresión llevado a cabo por el algoritmo HyperLCA en función de distintas configuraciones de sus parámetros de entrada. En este contexto, se ha analizado la información que se pierde tras el proceso de compresión con pérdidas utilizando tres métricas de calidad diferentes: *Relación Señal-Ruido (Signal-to-Noise Ratio o SNR)* representada en la Ecuación 3, *Error Medio Cuadrático (Root Mean Squared Error o RMSE)* calculada de acuerdo a la Ecuación 4 y *Diferencia Absoluta Máxima (Maximum Absolute Difference o MAD)* obtenida mediante la Ecuación 5. El *SNR* y el *RMSE* dan una idea de la información media perdida en el proceso de compresión-descompresión. Los valores más grandes de *SNR* son indicativos de un mejor rendimiento de la compresión. Por el contrario, valores más altos de *RMSE* significan que la compresión con pérdidas ha introducido mayores pérdidas de datos. La métrica *MAD* evalúa la cantidad de información perdida para el peor valor de la imagen reconstruida. Para nuestra aplicación, el rango dinámico es de $2^{12} = 4096$ y, por tanto, el peor valor posible de *MAD* es 4095.

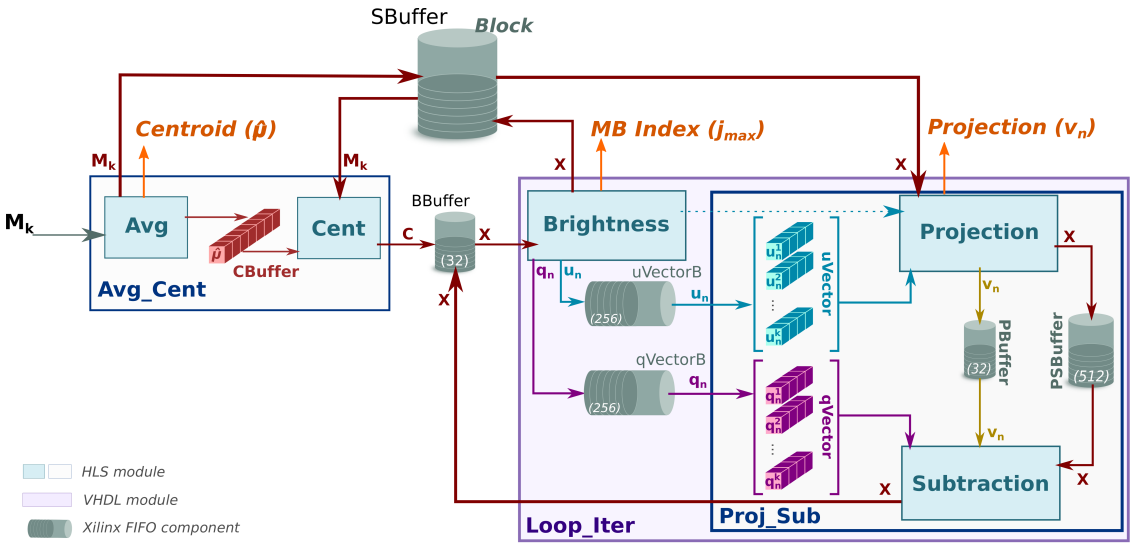

Fig. 1: Diagrama de bloques de la arquitectura implementada para la etapa transformación espectral del algoritmo *HyperLCA*.

En aras de la claridad, las métricas mencionadas se han calculado utilizando la totalidad de las imágenes comprimidas-descomprimidas, es decir, después de que el algoritmo HyperLCA haya terminado de comprimir todos los bloques de la imagen, $\mathbf{M}_K$.

Además, el análisis realizado recoge también una comparativa entre distintas versiones del algoritmo en función del número de bits y la aritmética empleada para representar los datos espectrales recogidos en $\mathbf{M}_K$. La versión *I12* corresponde con la situación anteriormente planteada, empleando lógica entera y 16 bits para guardar cada elemento de $\mathbf{M}_K$ pero asumiendo que el sensor hiperespectral codifica los datos capturados con un máximo de 12 bits. La versión *I16* y *I32* hacen uso también de lógica entera en punto fijo empleando 16 y 32 bits, respectivamente, para representar $\mathbf{M}_K$ y asumiendo que en este caso el sensor puede capturar los datos con un máximo de 16 bits. Finalmente, la versión *F32* es similar a la de *I32* pero empleando aritmética en coma flotante de precisión simple.

Tras el análisis de pérdidas de calidad se puede concluir que la calidad de los resultados de compresión de la versión *I12* es significativamente mejor que la versión *I16*, cuyas diferencias se acentúan para ratios de compresión menores debido a las pérdidas introducidas por la disminución de la precisión de los datos. Sin embargo, el rendimiento de compresión obtenido por ambas versiones no es tan competitivo como el de las soluciones basadas en punto fijo *I32* y en coma flotante *F32*. Se demuestra que este enfoque de *I12* proporciona su mejor rendimiento para ratios de compresión muy altos, *BS* pequeños e imágenes empaquetadas utilizando menos de 12 bits por píxel y por banda. Esto convierte a la versión *I12* en una opción muy interesante para aplicaciones con recursos hardware limitados, y por lo tanto es la versión candidata a implementar sobre lógica reconfigurable.

$$\mathrm{SNR} = 10 \cdot \log_{10}\left(\frac{\sum_{i=1}^{nb}\sum_{j=1}^{np}(I_{i,j})^2}{\sum_{i=1}^{nb}\sum_{j=1}^{np}(I_{i,j} - Ic_{i,j})^2}\right) \quad (3)$$

$$\mathrm{RMSE} = \frac{1}{np \cdot nb} \cdot \sqrt{\sum_{i=1}^{nb}\sum_{j=1}^{np}(I_{i,j} - Ic_{i,j})^2} \quad (4)$$

$$\mathrm{MAD} = \max(I_{i,j} - Ic_{i,j}) \quad (5)$$

IV. Implementación hardware del algoritmo HyperLCA

Las etapas de *transformación espectral* y *codificación* son las candidatas a ser implementadas sobre lógica reconfigurable, donde ambas deben trabajar de forma paralela; mientras que la primera se encarga de los cálculos con mayor requisitos de cómputo, la segunda es alimentada por los resultados de dichas operaciones para abordar su codificación. La etapa de *inicialización* se trata de la parte de configuración, la cual puede ser calculada en tiempo de diseño de acuerdo a los parámetros de entrada, ya que de ello depende diferentes elementos internos del propio acelerador, como son las profundidades de las memorias utilizadas. Por otro lado, la etapa de *preprocesado*, previa a la codificación, se ha integrado en las etapas aceleradas.

A. Transformación espectral

La Figura 1 muestra una visión general de la implementación hardware realizada para la etapa de *transformación espectral*. Los módulos *Avg_Cent*, *Brightness* y *Proj_Sub* han sido modelados e implementados utilizando Vivado HLS, mientras que las memorias y la lógica que integra y orquesta todos los componentes del diseño han sido instanciados e implementados utilizando el lenguaje VHDL. El acelerador tiene una única entrada correspondiente a un bloque hiperespectral ($\mathbf{M}_k$) que será comprimido, mientras que la salida está compuesta por tres elementos (flechas naranjas) cuyo destino es el codificador. Esta salida difiere del algoritmo original con el objetivo de alcanzar un mayor nivel de paralelismo. De este modo, el centroide ($\hat{\boldsymbol{\mu}}$), se obtiene como se indica en el Algoritmo 1, mientras que los p_{max} píxeles hiperespectrales más diferentes ($\mathbf{E}$) no se obtienen directamente. En este sentido, el acelerador proporciona

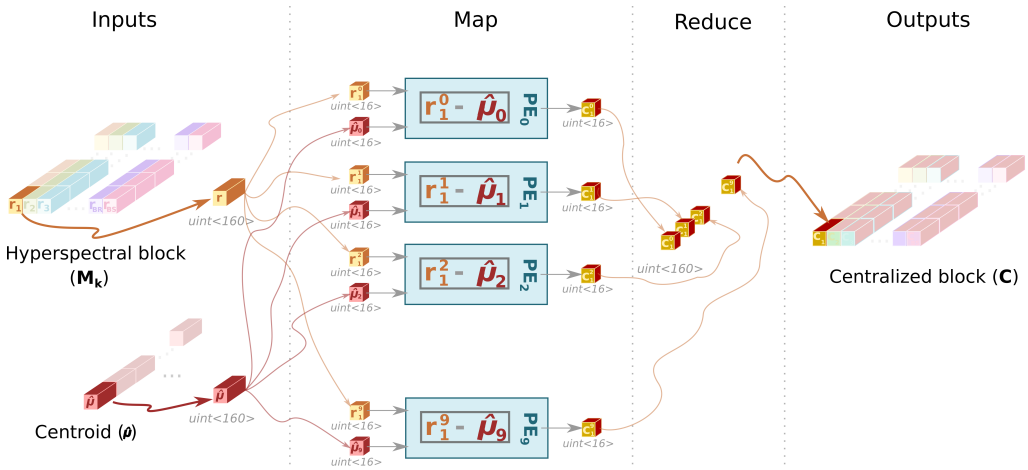

Fig. 2: Estrategia de paralelización de grano fino.

los índices de dichos píxeles (j_{max}) en cada iteración, mientras que el *codificador* es el encargado de obtener cada píxel hiperespectral ($\mathbf{e}_n$) de la memoria externa, en la que se almacena la propia imagen, para construir el vector de píxeles hiperespectrales más diferentes ($\mathbf{E}$). Finalmente, la proyección ($\mathbf{V}$) es proporcionada por el acelerador en partes, es decir, en cada iteración se obtiene una proyección ($\mathbf{v}_n$) que forma parte del vector $\mathbf{V}$.

El módulo *Avg_Cent* implementa las líneas 1 y 2 del Algoritmo 1. En primer lugar, se calcula el centroide o píxel medio ($\hat{\boldsymbol{\mu}}$) del bloque hiperespectral original ($\mathbf{M}_k$) en *Avg* y el resultado se almacena en *CBuffer*, que es compartido con el submódulo *Cent*. Durante esta operación, *Avg* reenvía una copia del centroide al *codificador* y, al mismo tiempo, almacena una copia del bloque de entrada ($\mathbf{M}_k$) en *SBuffer*, es decir, esta memoria interna debe garantizar suficiente espacio para almacenar un bloque hiperespectral completo. La profundidad de *SBuffer* dependerá del tamaño del bloque y de la cantidad de bandas espectrales utilizadas. Seguidamente el bloque original ($\mathbf{M}_k$) es centralizado por el submódulo *Cent*, haciendo uso del píxel medio ($\hat{\boldsymbol{\mu}}$) y del bloque hiperespectral almacenado en la memoria interna *SBuffer*.

Con el objetivo de obtener el mayor rendimiento posible a la solución basada en lógica reconfigurable, se ha implementado una estrategia de paralelización de grano fino aumentando el número de bandas a procesar de forma paralela en cada uno de los submódulos que componen la arquitectura final. La Figura 2 muestra un ejemplo de esta estrategia aplicada al submódulo *Cent*. Las entradas de esta etapa, bloque hiperespectral ($\mathbf{M}_k$) y el píxel medio ($\hat{\boldsymbol{\mu}}$), son leídas en bloques de N-bandas. Para operar de forma paralela es necesario disponer de tantos elementos de procesamiento (PEs) como bandas se deseen procesar. Por lo tanto, el ancho de palabra de cada una de las memorias internas (*CBuffer*, *SBuffer*, *BBuffer*, ...) están ligadas con el número de bandas a procesar en paralelo. En el ejemplo, la entrada es leída de diez en diez bandas, lo que supone un ancho de palabra de 160 bits cuando la representación de cada píxel es de 16 bits. Esta palabra es troceada en tantos trozos como elementos de procesamiento se tengan instanciados, ya que cada uno de estos elementos operarán con una única banda (fase map), mientras la salida se construirá a partir de cada una de la salidas parciales proporcionadas por los elementos de procesamiento (fase reduce).

Los siguientes módulos que entran en juego son los correspondientes al bucle *for* de la línea 3 del Algoritmo 1, los cuales son ejecutados p_{max} veces. El submódulo *Brightness* empieza a operar, en la primera iteración, en el momento que el submódulo *Cent* proporciona datos, mientras que el resto de iteraciones esperará a la salida del submódulo *Subtraction*. Independientemente de la fuente de datos, el submódulo *Brightness* tiene a su entrada una memoria (*BBuffer*) que actúa de buffering. El submódulo *Brightness* ha sido optimizado para lograr que el tiempo que se tarda en obtener el píxel con mayor brillo sea independiente de la ubicación del mismo. En primer lugar, se lee cada píxel hiperespectral del bloque y se calcula su brillo (b_j) como se especifica en la línea 5 del Algoritmo 1. A su vez se almacena dicho píxel en una memoria interna y en *SBuffer*, quedando una copia del bloque hiperespectral con las transformaciones realizadas (línea 5 y asignación en la línea 12 del Algoritmo 1). Una vez calculado el brillo del píxel hiperespectral actual, se actualizará un vector interno que almacena el píxel con mayor brillo. En este caso, las memorias internas de este módulo han sido implementadas siguiendo una estrategia ping-pong buffer, cuya solución se ha desarrollado de forma manual en lugar de automatizar esta optimización a través de el uso de *pragmas*, lo que ha permitido paralelizar el flujo de datos y obtener un mayor rendimiento. Por último, los vectores de proyección ortogonal $\mathbf{q}_n$ y $\mathbf{u}_n$ se obtienen a partir del píxel más brillante (líneas 9 y 10 del Algoritmo 1). Ambos son almacenados en FIFOs independientes cuya profundidad es el doble de la necesaria para evitar el bloqueo de la arquitectura desarrollada. También devuelve el índice del píxel más brillante (j_{max}) para que el *codificador* lea el píxel original de la memoria externa donde se almacena la imagen hiperespectral para construir la salida comprimida.

Aunque el submódulo *Proj_Sub* está representado por dos submódulos separados (*Projection* y *Subtraction*) en la Figura 1, hay que mencionar que ambos realizan sus cálculos en paralelo. El submódulo *Proj_Sub* lee el bloque hiperespectral que fue escrito en *SBuffer* por el submódulo *Brightness* ($\mathbf{X}$) y el vector de proyección ortogonal $\mathbf{u}_n$, para obtener el vector de la imagen proyectada según la línea 11 del Algoritmo 1. Al mismo tiempo, los datos leídos de *SBuffer* se escriben en *PSBuffer*, que puede almacenar hasta dos píxeles hiperespectrales, debido a que el submódulo *Subtraction* comienza justo después de que la primera proyección sobre el primer píxel hiperespectral esté lista, es decir, la ejecución de *Subtraction* tiene un intervalo de inicialización de un píxel respecto del submódulo *Projection*; mientras el píxel i está siendo consumido por *Subtraction*, el píxel $i+1$ está siendo escrito en *PSBuffer*. Durante la proyección del segundo píxel hiperespectral ($i+1$), se puede realizar la sustracción del primero (i) ya que todos los operandos de entrada, incluida la proyección $\mathbf{v}_n$, están disponibles (línea 12 del Algoritmo 1). Cabe destacar que los vectores ortogonales, $\mathbf{q}_n$ y $\mathbf{u}_n$, son copiados y particionados en N arrays, donde N corresponde al número de elementos de procesamiento instanciados en los diferentes submódulos.

La salida del submódulo *Projection* es el vector de la imagen proyectada ($\mathbf{v}_n$), que es escalado y posteriormente enviado al *codificador*. Por otro lado, la salida del submódulo *Subtraction* alimenta de nuevo al bloque *Loop_Iter* en una nueva iteración (ver flecha púrpura, etiquetada como $\mathbf{X}$, en la Figura 1) con los píxeles del bloque transformado en la i^{th} iteración. Esto significa que la etapa *Brightness* puede iniciar la siguiente iteración sin esperar a obtener la imagen completa modificada. Así, el intervalo de inicialización entre iteraciones del bucle se reduce al máximo, ya que *Brightness* comienza cuando los primeros datos sustraídos están listos. En la última iteración, los datos de salida del submódulo *Subtraction* serán descartados, es decir, no se almacenarán en *BBuffer*.

B. Codificador

El *codificador* es el segundo de los aceleradores hardware desarrollados, y es el responsable de realizar en paralelo las tareas del mapeo de errores de predicción CCSDS [20] y de la tarea de codificación de entropía Golomb-Rice [24] a medida que se reciben los diferentes vectores desde el acelerador explicado en la sección anterior. Ambos aceleradores pueden funcionar en paralelo dado que el codificador tarda aproximadamente la mitad del tiempo que la transformación necesita para generar cada vector para el máximo número de PEs, es decir, el máximo rendimiento alcanzado. Por lo tanto, no se produce una situación de contención, reduciendo la presión sobre los FIFOs que conectan ambos bloques y, por consiguiente, requiriendo menos espacio para estos canales de comunicación.

La Figura 3 muestra la estructura interna del *codificador* que ha sido modelado íntegramente con Vivado HLS. Se trata de una arquitectura que comprende tres pasos. Durante el primer paso, el mapeo de predicción y la codificación de entropía de todos los vectores de entrada son realizados por el *coding command generator*. El resultado de este paso es una secuencia de comandos que posteriormente son interpretados por el *bitstream generator*. Este módulo está leyendo continuamente la FIFO *cmd_queue* en busca de un nuevo comando para ser procesado. Un comando contiene la operación así como la palabra a codificar, y el número de bits a generar. Las funciones de codificación simplemente iteran sobre la palabra a codificar y producen una secuencia de bits que corresponde al formato comprimido del bloque hiperespectral. Por último, el tercer paso empaqueta el flujo de bits comprimido en palabras y lo escribe en la memoria. Para esta implementación, el ancho de la palabra de memoria es de 64 bits, el máximo permitido por el puerto de interfaz *AXI Master* para la plataforma FPGA seleccionada. El módulo *Bitstream packet* instancia un pequeño buffer de 64 palabras que se vuelca a la memoria DDR una vez que se ha llenado. De este modo, se optimiza el promedio de ciclos de acceso a la memoria por palabra mediante el uso de ráfagas.

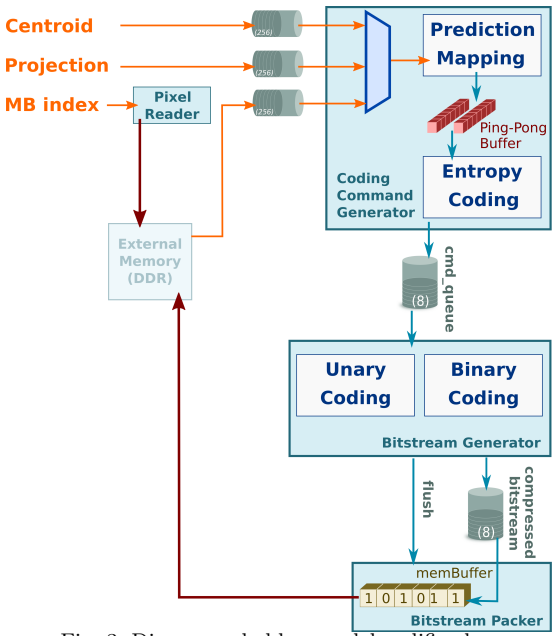

Fig. 3: Diagrama de bloques del codificador.

V. Resultados experimentales

Para evaluar el acelerador de hardware se ha implementado la arquitectura propuesta utilizando las herramientas proporcionadas por Xilinx. El prototipo implementado se ha desplegado sobre una FPGA del mismo fabricante, en concreto una Zynq-7000 con un chip XC7Z020-CLG484. Esta FPGA ha sido seleccionada por su bajo coste y alta flexibilidad, características que la convierten en un dispositivo interesante para ser integrado en plataformas aéreas, como los drones. La cantidad de recursos disponibles en este dispositivo permite instanciar hasta 20 elementos de procesamiento (PEs).

Tabla II: Resultados post-síntesis con diferentes PEs y bloques de 1024 píxeles con 160 bandas espectrales.

Num. PEs	BRAM18K	DSP48E	FlipFlops	LUTs
1	96,0 *(68,57 %)*	9 *(4,09 %)*	7.121 *(6,69 %)*	5.434 *(10,21 %)*
2	94,5 *(67,5 %)*	16 *(7,27 %)*	6.655 *(6,25 %)*	6.257 *(11,76 %)*
4	98,0 *(70 %)*	30 *(13,64 %)*	7.725 *(7,26 %)*	7.416 *(13,94 %)*
8	96,5 *(68,93 %)*	58 *(26,36 %)*	9.910 *(9,31 %)*	9.553 *(17,96 %)*
10	104,5 *(74,64 %)*	72 *(32,73 %)*	11.415 *(10,73 %)*	11.630 *(21,86 %)*
16	95,5 *(68,21 %)*	114 *(51,82 %)*	14.891 *(14 %)*	14.721 *(27,67 %)*
20	95,5 *(68,21 %)*	122 *(64,55 %)*	17.946 *(16,87 %)*	18.851 *(35,43 %)*

El número de PEs a instanciar dependerá directamente del número de bandas espectrales que contenga la imagen, que para el análisis de la arquitectura propuesta se ha establecido en 160 bandas y cuyo tamaño de bloque será de 1024 píxeles. Las imágenes se han adquirido mediante una cámara hiperespectral *especim FX10* de tipo pushbroom montada sobre un dron DJI Matrice 600 [25]. El sensor hiperespectral cubre el rango del espectro entre 400 y 1000 nm utilizando 1024 píxeles espaciales por línea transversal escaneada y 224 bandas espectrales. Los conjuntos de datos se recogieron sobre algunas zonas de viñedos en el centro de la isla de Gran Canaria (Islas Canarias, España), cuyas coordenadas exactas son 27°59′356″N 15°36′25.6″W, a una altura de 45m sobre el suelo, obteniendo una representación aproximada de unos $3cm^2$ por cada píxel. La velocidad del dron fue de 4,5 m/s y la tasa de muestreo fue fijada a 150 LPS (Líneas por Segundo). El modo de captura de imágenes de este sensor provoca que el análisis de los resultados se realice mediante líneas por segundo, en lugar de frames por segundo.

La Tabla II recoge los recursos necesarios para cada una de las versiones del acelerador hardware, dichos datos han sido extraídos de los informes posteriores a la síntesis.

Por otro lado, la Tabla III muestra los resultados post-síntesis del bloque encargado de codificar la salida de la transformada espacial. Los recursos demandados por el codificador no dependen del tamaño del bloque (BS) ni del número de PEs instanciados. Es importante mencionar que la mayoría de los recursos BRAM, FFs y LUTs se asignan a las dos interfaces AXI-Memory que la herramienta HLS genera.

Tabla III: Resultados post-síntesis del codificador para imágenes de 160 bandas espectrales.

BRAM18K	DSP48E	FlipFlops	LUTs
7 *(2.5 %)*	1 *(0.45 %)*	3464 *(3.25 %)*	4106 *(7.71 %)*

La Tabla IV muestra el rendimiento, expresado en líneas por segundo, para una configuración específica del acelerador hardware utilizando una frecuencias de reloj de 150 MHz, donde las dos primeras columnas denotan la configuración del acelerador, en forma de número de bits por píxel (N_{bits}) y ratio de compresión (CR), mientras que el resto corresponden al número de *elementos de procesamiento* instanciados en el propio acelerador. Para las pruebas se ha establecido el tamaño del bloque en 1024 píxeles hiperespectrales, que es el tamaño de la línea o fotograma entregado por el sistema de adquisición.

Tabla IV: LPS alcanzados según las diferentes configuraciones con bloques de 1024 píxeles.

Nbits	CR	Elementos de Procesamiento (PEs)						
		1	2	4	8	10	16	20
12	12	67	140	**275**	**503**	**603**	**857**	**997**
	16	88	**182**	**357**	**636**	**754**	**1.045**	**1.200**
	20	110	**202**	**444**	**772**	**906**	**1.225**	**1.387**
8	12	48	101	**199**	**373**	**451**	**659**	**778**
	16	67	140	**275**	**503**	**603**	**857**	**997**
	20	80	**165**	**325**	**584**	**696**	**974**	**1.123**

Un factor clave es la frecuencia de imagen mínima que debe soportar la aplicación. Lo ideal sería que dicho umbral se correspondiera con la máxima velocidad a la que el sensor hiperespectral empleado puede trabajar, es decir, 330 LPS. Sin embargo, una tasa de entre 150 y 200 LPS son suficientes para obtener imágenes con una calidad aceptable. Por lo tanto, se establece un valor umbral de 200 LPS como rendimiento mínimo para validar la viabilidad del acelerador hardware. En la Tabla IV se han resaltado (celdas en negrita) las configuraciones que serían válidas dado este mínimo. Se puede observar que todas las configuraciones excepto las versiones de 1 y 2 PEs alcanzan el umbral establecido. A su vez, al utilizar valores más bajos para N_{bits} la tasa de compresión se reduce debido al mayor número de vectores $\mathbf{V}$ extraídos en la fase de transformación espectral. Esto significa que hay que realizar más cálculos para comprimir un bloque hiperespectral.

Desde el punto de vista del consumo-rendimiento, se ha comparado la solución presentada en este trabajo con plataformas basadas en GPU, concretamente los resultados se han extraído de la propuesta realizada en el artículo [26], donde el algoritmo *HyperLCA* se ha desplegado sobre las plataformas Jetson Nano y Jetson TX2, y del trabajo [27] donde se incorporó la GPU de última generación Jetson Xavier NX. La Figura 4 muestra la eficiencia energética de cada dispositivo, medida como LPS alcanzados divididos por el consumo energético de cada solución. La imagen muestra cómo varía la eficiencia en relación con el ratio de compresión (CR). Las placas Jetson están diseñadas con un circuito integrado de gestión de energía de alta eficiencia que se encarga de los reguladores de voltaje, y un árbol de energía para optimizar la eficiencia energética. Según [28], [29], [30], los balances de energía típicos de las placas seleccionadas ascienden a 10 W, 15 W y 10 W para los módulos Jetson Nano, Jetson TX2 y Jetson Xavier NX, respectivamente. En el caso de la FPGA XC7Z020-CLG484, el consumo de energía estimado tras la etapa *Place & Route* asciende a 3,74 W a 150

MHz. En base a la tendencia de las gráficas mostradas en la Figura 4, se puede observar que la solución basada en FPGA es mucho más eficiente en términos de consumo de energía que las plataformas basadas en GPU para todas las configuraciones posibles. Por lo tanto, se puede afirmar que la arquitectura basada en lógica reconfigurable para el algoritmo de compresión con pérdidas *HyperLCA* presenta un enfoque más eficiente desde el punto de vista energético.

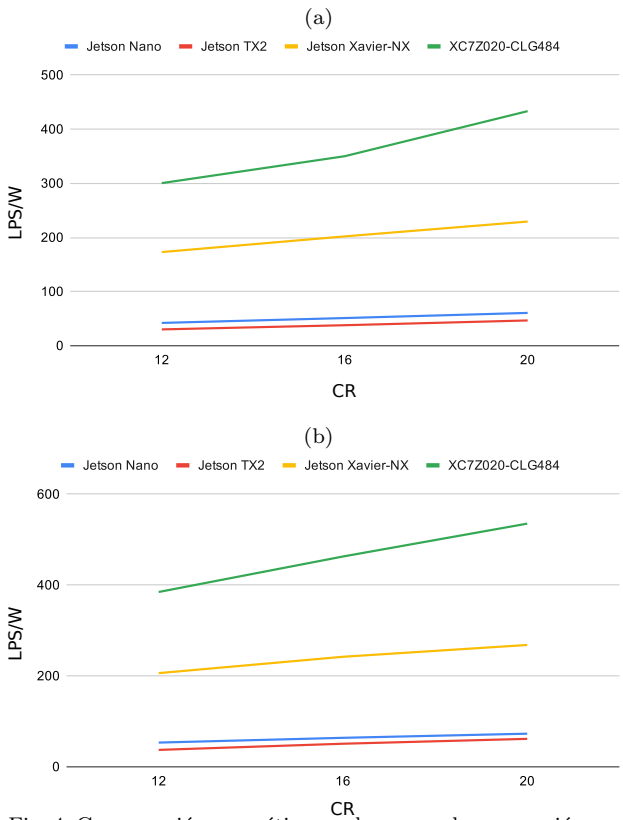

Fig. 4: Comparación energética en el proceso de compresión en términos de LPS/W alcanzados por las soluciones basadas en FPGA y GPU (**a**) LPS/W $N_{bits} = 8$; (**b**) LPS/W $N_{bits} = 12$.

VI. Conclusiones

La modificación del algoritmo HyperLCA para ser ejecutado mediante aritmética de punto fijo supone una mejora sustancial del rendimiento junto con una importante reducción de recursos hardware. En este contexto, se ha implementado la mencionada versión modificada del compresor con pérdidas HyperLCA en un SoC Zynq-7000 con el fin de acelerar su rendimiento y alcanzar una alta tasa de compresión de imágenes hiperespectrales. La solución adoptada combina módulos que utilizan VHDL y modelos HLS sintetizados, dando vida a un flujo de datos eficiente que permite alcanzar una velocidad superior a los 750 LPS en la configuración más exigente, es decir un mayor número de iteraciones (p_{max}).

Además, la comparación en términos de eficiencia energética y rendimiento entre la implementación basada en FPGA desarrollada en este trabajo y un modelo basado en GPU desplegado en tres tarjetas de computación NVIDIA de bajo consumo (Jetson Nano, Jetson TX2 y Jetson Xavier NX) demuestra que la implementación sobre lógica reconfigurable es más eficiente frente a las alternativas sobre GPU.

Por último, aunque el trabajo descrito en este artículo se ha centrado en una aplicación basada en UAV, puede extrapolarse fácilmente a otros trabajos en el ámbito espacial, ya que se ha implementado de forma eficiente todas las etapas de compresión del algoritmo HyperLCA en la parte de la lógica programable (PL) del SoC. Por lo tanto, puede ser fácilmente extrapolado a otras FPGAs certificadas y tolerantes a la radiación para su uso en el espacio.

Agradecimientos

Esta investigación está parcialmente financiada por el Ministerio de Economía y Competitividad (MINECO) del Gobierno de España (proyecto PLATINO, no. TEC2017-86722-C4, subproyectos 1 y 4), la Agencia Canaria de Investigación, Innovación y Sociedad de la Información (ACIISI) de la Consejería de Economía, Industria, Comercio y Conocimiento del Gobierno de Canarias, conjuntamente con el Fondo Social Europeo (FSE) (POC2014-2020, Eje 3 Tema Prioritario 74 (85 %)), la Junta de Comunidades de Castilla-La Mancha (proyecto SymbIoT, no. SBPLY-17-180501-000334), y por el programa Europeo Horizonte 2020 bajo el proyecto SHAPES (GA Nº 857159).

Referencias

[1] Antonio Plaza, Jon Atli Benediktsson, Joseph W Boardman, Jason Brazile, Lorenzo Bruzzone, Gustavo Camps-Valls, Jocelyn Chanussot, Mathieu Fauvel, Paolo Gamba, Anthony Gualtieri, et al., "Recent advances in techniques for hyperspectral image processing," *Remote sensing of environment*, vol. 113, pp. S110–S122, 2009.

[2] Nor Rizuan Mat Noor and Tanya Vladimirova, "Integer klt design space exploration for hyperspectral satellite image compression," in *International Conference on Hybrid Information Technology*. Springer, 2011, pp. 661–668.

[3] Miloš Radosavljević, Branko Brkljač, Predrag Lugonja, Vladimir Crnojević, Željen Trpovski, Zixiang Xiong, and Dejan Vukobratović, "Lossy compression of multispectral satellite images with application to crop thematic mapping: A hevc comparative study," *Remote Sensing*, vol. 12, no. 10, pp. 1590, 2020.

[4] Alberto G Villafranca, Jordi Corbera, Francisco Martín, and Juan Fernando Marchán, "Limitations of hyperspectral earth observation on small satellites," *Journal of Small Satellites*, vol. 1, no. 1, pp. 19–29, 2012.

[5] Rico Valentino, Woo-Sung Jung, and Young-Bae Ko, "A design and simulation of the opportunistic computation offloading with learning-based prediction for unmanned aerial vehicle (UAV) clustering networks," *Sensors*, vol. 18, no. 11, pp. 3751, 2018.

[6] Sebastian Lopez, Tanya Vladimirova, Carlos Gonzalez, Javier Resano, Daniel Mozos, and Antonio Plaza, "The promise of reconfigurable computing for hyperspectral imaging onboard systems: A review and trends," *Proceedings of the IEEE*, vol. 101, no. 3, pp. 698–722, 2013.

[7] Alan D George and Christopher M Wilson, "Onboard processing with hybrid and reconfigurable computing on small satellites," *Proceedings of the IEEE*, vol. 106, no. 3, pp. 458–470, 2018.

[8] S Fu, R Chang, S Couture, M Menarini, MA Escobar, M Kuteifan, M Lubarda, D Gabay, and V Lomakin, "Micromagnetics on high-performance workstation and mobile computational platforms," *Journal of Applied Physics*, vol. 117, no. 17, pp. 17E517, 2015.

[9] José M Bioucas-Dias, Antonio Plaza, Gustavo Camps-Valls, Paul Scheunders, Nasser Nasrabadi, and Jocelyn Chanussot, "Hyperspectral remote sensing data analysis and future challenges," *IEEE Geoscience and remote sensing magazine*, vol. 1, no. 2, pp. 6–36, 2013.

[10] Didier Keymeulen, Nazeeh Aranki, Ben Hopson, Aaron Kiely, Matthew Klimesh, and Khaled Benkrid, "Gpu lossless hyperspectral data compression system for space applications," in *2012 IEEE Aerospace Conference*. IEEE, 2012, pp. 1–9.

[11] Bormin Huang, *Satellite data compression*, Springer Science & Business Media, 2011.

[12] Barbara Penna, Tammam Tillo, Enrico Magli, and Gabriella Olmo, "Transform coding techniques for lossy hyperspectral data compression," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 45, no. 5, pp. 1408–1421, 2007.

[13] Michael W Marcellin and David S Taubman, "JPEG2000: image compression fundamentals, standards, and practice," *Kluwer International Series in Engineering and Computer Science, Secs 642*, 2002.

[14] Lena Chang, Ching-Min Cheng, and Ting-Chung Chen, "An efficient adaptive KLT for multispectral image compression," in *4th IEEE Southwest Symposium on Image Analysis and Interpretation*. IEEE, 2000, pp. 252–255.

[15] Pengwei Hao and Qingyun Shi, "Reversible integer KLT for progressive-to-lossless compression of multiple component images," in *Proceedings 2003 International Conference on Image Processing (Cat. No. 03CH37429)*. IEEE, 2003, vol. 1, pp. I–633.

[16] Andrea Abrardo, Mauro Barni, and Enrico Magli, "Low-complexity predictive lossy compression of hyperspectral and ultraspectral images," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2011, pp. 797–800.

[17] Lucana Santos, Enrico Magli, Raffaele Vitulli, José F López, and Roberto Sarmiento, "Highly-parallel gpu architecture for lossy hyperspectral image compression," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 6, no. 2, pp. 670–681, 2013.

[18] Yubal Barrios, Antonio J Sánchez, Lucana Santos, and Roberto Sarmiento, "Shyloc 2.0: A versatile hardware solution for on-board data and hyperspectral image compression on future space missions," *Ieee Access*, vol. 8, pp. 54269–54287, 2020.

[19] L. Santos, J. F. López, R. Sarmiento, and R. Vitulli, "Fpga implementation of a lossy compression algorithm for hyperspectral images with a high-level synthesis tool," in *2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013)*, 2013, pp. 107–114.

[20] Consultative Committee for Space Data Systems (CCSDS), "*Blue Books: Recommended Standards.*," `{[Online].Available:}https://public.ccsds.org/Publications/BlueBooks.aspx`, (Última consulta: 7 Mayo 2021).

[21] Erick L Oberstar, "Fixed-point representation & fractional math," *Oberstar Consulting*, p. 9, 2007.

[22] Chuanmin Hu, Lian Feng, Zhongping Lee, Curtiss Davis, Antonio Mannino, Charles Mcclain, and Bryan Franz, "Dynamic range and sensitivity requirements of satellite ocean color sensors: Learning from the past," *Applied optics*, vol. 51, pp. 6045–62, 09 2012.

[23] Ashok Kumar, Sanjeev Mehta, Sandip Paul, R. Parmar, and R Samudraiah, "Dynamic range enhancement of remote sensing electro-optical imaging systems," 12 2012.

[24] Paul G Howard and Jeffrey Scott Vitter, "Fast and efficient lossless image compression," in *Data Compression Conference, 1993. DCC'93.* IEEE, 1993, pp. 351–360.

[25] Pablo Horstrand, Raúl Guerra, Aythami Rodríguez, María Díaz, Sebastián López, and José Fco López, "A uav platform based on a hyperspectral sensor for image capturing and on-board processing," *IEEE Access*, vol. 7, pp. 66919–66938, 2019.

[26] María Díaz, Raúl Guerra, Pablo Horstrand, Ernestina Martel, Sebastián López, José F. López, and Sarmiento Roberto, "Real-time hyperspectral image compression onto embedded GPUs," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, pp. 1–18, 2019.

[27] Julián Caba, María Díaz, Jesús Barba, Raúl Guerra, and Jose A. de la Torre and Sebastián López, "Fpga-based on-board hyperspectral imaging compression: Benchmarking performance and energy efficiency against gpu implementations," *Remote Sensing*, vol. 12, no. 22, 2020.

[28] NVIDIA Corporation, "*NVIDIA Jetson Linux Developer Guide 32.4.3 Release. Power Management for Jetson Nano and Jetson TX1 Devices.*," Available Online: `https://docs.nvidia.com/jetson/l4t/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/power_management_nano.html`, (Última consulta: 7 Mayo 2021).

[29] NVIDIA Corporation, "*NVIDIA Jetson Linux Driver Package Software Features Release 32.3. Power Management for Jetson TX2 Series Devices.*," Available Online: `https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/power_management_tx2_32.html`, (Última consulta: 7 Mayo 2021).

[30] NVIDIA Corporation, "*NVIDIA Jetson Linux Developer Guide 32.4.3 Release. Power Management for Jetson Xavier NX and Jetson AGX Xavier Series Devices.*," Available Online: `https://docs.nvidia.com/jetson/l4t/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/power_management_jetson_xavier.html`, (Última consulta: 7 Mayo 2021).