UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Estudio comparativo de implementaciones paralelas para el análisis de imágenes obtenidas de forma remota

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Rubén Macias Igari

DIRIGIDA POR

Carlos González Calvo
Sergio Bernabé García

Estudio comparativo de implementaciones paralelas para el análisis de imágenes obtenidas de forma remota

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

Rubén Macias Igari

Dirigida por los profesores:

Carlos González Calvo
Sergio Bernabé García

Facultad de Informática
Universidad Complutense de Madrid

Madrid, julio 2025

Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas las personas que, de una u otra manera, han estado cerca de mí a lo largo de este camino. A quienes me brindaron su confianza, su tiempo, su apoyo y su cariño, mi gratitud es profunda y permanente. La culminación de esta tesis representa un logro académico, sí, pero también un recorrido vital que ha dejado huella en mi vida personal, por lo que agradecer no es solo un acto formal, sino una necesidad sentida que no tendría fin.

Durante estos años como estudiante de doctorado, he experimentado un crecimiento integral en todos los aspectos de mi vida. Esta etapa ha sido, sin duda, una de las más satisfactorias y enriquecedoras que he vivido. Haber tenido la oportunidad de estudiar en la Universidad Complutense de Madrid, en España, lejos de mis seres queridos y dejando atrás muchas cosas por alcanzar un objetivo de calidad académica, ha sido un desafío con un valor incalculable. La satisfacción que me deja esta experiencia es simplemente indescriptible.

Deseo expresar de manera especial mi más profundo agradecimiento a mis directores de tesis, el Dr. Carlos González Calvo y el Dr. Sergio Bernabé García. Su guía constante, dedicación y profundo conocimiento fueron pilares fundamentales en el desarrollo de este trabajo. Agradezco sinceramente su paciencia, compromiso y las valiosas observaciones que enriquecieron tanto el contenido académico como mi formación como investigador. Más allá de su rol como directores, valoro profundamente el apoyo humano que me brindaron, el cual fue decisivo en los momentos más exigentes del proceso. Su confianza y acompañamiento marcaron una diferencia significativa desde el inicio hasta el final.

Recuerdo con especial afecto la etapa de movilidad académica que realicé entre enero y junio de 2016 en la Facultad de Informática, cuando tuve la oportunidad de conocerlos. Desde aquel momento supe que quería regresar como estudiante de doctorado, y desde entonces siempre conté con su respaldo incondicional. Desde aquellos primeros encuentros, admiré su forma de trabajar y aprender de ustedes ha sido, sin duda, un privilegio. Ha sido un verdadero placer recorrer este camino a su lado.

A lo largo del proceso comprendí que el ritmo del trabajo doctoral exige dedicación, constancia y compromiso con los objetivos trazados. Esta experiencia no solo me formó académicamente, sino que también me transformó personalmente.

Quiero extender también mi agradecimiento al Consejo Nacional de Ciencia y Tecnología (CONACYT) de México por el valioso apoyo brindado a través de la convocatoria “Becas CONACYT para estudios de Doctorado en el Extranjero 2021-1”, de la cual fui beneficiario.

Finalmente, a todas las personas que estuvieron cerca durante esta etapa de mi vida, ofreciendo palabras de aliento, compañía y comprensión:

¡Muchas gracias!

A Emmanuel, Cin, Rubén y Lu.

Resumen

Este trabajo de Tesis Doctoral aborda el desafío del desmezclado espectral en imágenes hiperespectrales mediante una solución optimizada para plataformas FPGA, utilizando el lenguaje de alto nivel DPC++ dentro del entorno Intel oneAPI. La principal dificultad en el análisis de imágenes hiperespectrales radica en la mezcla espectral de materiales dentro de un único píxel, lo que dificulta la identificación de los componentes puros o endmembers. Esta tarea es esencial para diversas aplicaciones de observación remota de la Tierra, tales como la agricultura, la minería y la gestión ambiental.

La investigación presenta una cadena de procesamiento para desmezclar espectralmente los píxeles mixtos de una imagen hiperespectral, que consta de tres etapas principales: 1) estimación del número de endmembers, 2) detección y clasificación automática de los materiales puros, y 3) estimación de la proporción de cada endmember en la mezcla. El proceso ha sido optimizado para su implementación en FPGA, lo que permite una reducción significativa en los tiempos de procesamiento. Uno de los avances clave de esta investigación es la optimización de la implementación en FPGA utilizando DPC++ en lugar de los tradicionales lenguajes de descripción de hardware (HDL, *Hardware Description Language*). Esta elección facilita la descripción de la arquitectura y permite reducir el tiempo de desarrollo. Para cada una de las tres etapas del desmezclado espectral, se aplicaron estrategias de optimización en términos de paralelismo y eficiencia en el uso de los recursos de hardware, lo que resultó en una mejora significativa del rendimiento. Los resultados experimentales mostraron que las implementaciones optimizadas a través de High-Level Synthesis (HLS) mediante el aumento de acumuladores, ejecución de tareas *single-task* en paralelo y reestructuración de código funcional en una única estructura hardware eliminando la necesidad de replicar funcionalidades, entre otras, lograron reducir significativamente los tiempos de procesamiento. Como conclusión, el trabajo demuestra que el uso de técnicas de optimización para plataformas FPGA, combinadas con el entorno DPC++, construye una solución eficaz y flexible para el desmezclado espectral de imágenes hiperespectrales.

Palabras clave: Imagen Hiperespectral, Desmezclado Espectral, Virtual Dimensionality (VD), Automatic Target Detection and Classification Algorithm (ATDCA), Image Space Reconstruction algorithm (ISRA), Data Parallel C++, Hardware Reconfigurable.

Abstract

This doctoral thesis addresses the challenge of spectral unmixing in hyperspectral images through an optimized solution for FPGA platforms, using the high-level programming language DPC++ within the Intel oneAPI environment. The main difficulty in hyperspectral image analysis lies in the spectral mixing of materials within a single pixel, which hinders the identification of pure components or endmembers. This task is essential for various Earth observation applications, such as agriculture, mining, and environmental management.

The research presents a processing chain for spectral unmixing of mixed pixels in a hyperspectral image, which consists of three main stages: 1) estimation of the number of endmembers, 2) automatic detection and classification of pure materials, and 3) estimation of the proportion of each endmember in the mixture. The process has been optimized for FPGA implementation, which allows for a significant reduction in processing times. One of the key advancements in this research is the optimization of the FPGA implementation using DPC++ instead of traditional Hardware Description Languages (HDL). This choice facilitates the description of the architecture and reduces development time. For each of the three stages of spectral unmixing, optimization strategies were applied in terms of parallelism and efficiency in the use of hardware resources, resulting in a significant performance improvement. Experimental results showed that the optimized implementations through High-Level Synthesis (HLS) achieved a significant reduction in processing times, by increasing the number of accumulators, enabling parallel execution of *single-task*, and restructuring functional code into a unified hardware architecture (thus eliminating the need for functional replication), among other enhancements, achieved a significant reduction in processing times. In conclusion, the work demonstrates that the use of optimization techniques for FPGA platforms, combined with the DPC++ environment, provides an effective and flexible solution for spectral unmixing of hyperspectral images.

Keywords: Hyperspectral Image, Spectral Unmixing, Virtual Dimensionality (VD), Automatic Target Detection and Classification Algorithm (ATDCA), Image Space Reconstruction Algorithm (ISRA), Data Parallel C++, Reconfigurable Hardware.

Índice general

Índice de figuras

Índice de tablas

Acrónimos

Acrónimos / Abreviaturas

ALMs Adaptive Logic Modules

ALUTs Adaptive Look-Up Tables

ASIC Application-Specific Integrated Circuit

ATDCA Automatic Target Detection and Classification Algorithm

AVIRIS Airborne Visible Infrared Imagine Spectrometer

DSPs Digital Signal Processors

FPGA Field-Programmable Gate Array

GPU Graphics Processing Unit

HDL Hardware Description Language

HLS High Level Synthesis

HPC High-Performance Computing

ISRA Image Space Reconstruction Algorithm

LABs Logic Array Blocks

LMM Linear Mixing Model

LUTs Look-Up Tables

MLABs Memory LABs

MPSoC Multiprocessor SoC

MPU Multi-Processor Unit

NLMM Nonlinear Mixing Model

NMSE Normalised Mean Square Error

SoC System-on-Chip

VD Virtual Dimensionality

Capítulo 1

Motivaciones y objetivos

El procesamiento de imágenes hiperespectrales obtenidas desde satélites ha sido un área de investigación activa en la comunidad científica durante las últimas décadas [29]. Este proceso se basa en el uso de un espectrómetro de imágenes que captura la luz reflejada (reflectancia) en cientos de longitudes de onda distintas (bandas) para una misma región de la superficie terrestre. El análisis de estas imágenes tiene aplicaciones muy diversas, entre las que destacan la agricultura de precisión [66], la monitorización ambiental [87], la geología [57] y la vigilancia urbana [89], todas ellas con un gran impacto social.

Los avances tecnológicos han permitido que los nuevos sensores capturen más información en menos tiempo, generando un flujo continuo de datos de alta dimensionalidad [92]. Como consecuencia, la capacidad de almacenamiento del satélite se llena rápidamente, y la transmisión de las imágenes a la Tierra mediante enlaces de radio requiere un tiempo considerable. Dada la enorme cantidad de datos recopilados, la compresión y el procesamiento a bordo se han convertido en elementos clave para los satélites equipados con sensores hiperespectrales [71]. Además, el procesamiento de datos hiperespectrales presenta una elevada complejidad computacional [21]. Cualquier adaptación de algoritmos diseñados para imágenes bidimensionales conlleva una sobrecarga proporcional al número de bandas. Por esta razón, el procesamiento de los datos debe realizarse a bordo del satélite, lo que exige tecnologías capaces de satisfacer tanto los requerimientos de rendimiento como las restricciones de carga útil de la misión [64].

En el campo de la computación paralela, encontramos dispositivos con un número reducido de núcleos, alrededor de una docena, como los procesadores actuales conocidos como multinúcleo (*multi-core*); y otros con un gran número de núcleos, conocidos como *many-core*, entre los que destacan las Unidades de Procesamiento Gráfico (GPU, *Graphics*

Processing Units). El alto rendimiento que ofrecen las GPU en tareas de Computación de Alto Rendimiento (HPC, *High-Performance Computing*) conlleva un elevado consumo energético debido a la gran cantidad de unidades de procesamiento que incorporan. El auge de los sistemas en un solo chip (SoC, *System-on-Chip*) ha impulsado el desarrollo de chips que integran múltiples núcleos de procesamiento y una *Field-Programmable Gate Array* (FPGA), combinando la flexibilidad del software con la optimización del hardware. En la búsqueda de mayores niveles de rendimiento y versatilidad, surgen los SoC multiprocesador (MPSoC, *Multiprocessor SoC*), que aumentan la capacidad de cómputo añadiendo más procesadores. Un ejemplo de ello son los dispositivos AMD Xilinx Zynq UltraScale+ (Xilinx fue adquirida por AMD en 2022), que incluyen una Unidad Multiprocesador (MPU, *Multi-Processor Unit*), una GPU (ARM Mali-400, aunque con limitaciones para la computación de propósito general (GPGPU, *General Purpose GPU*) debido a su enfoque en la representación de gráficos y soporte limitado de OpenCL) y una FPGA (solución más eficiente para una aceleración de propósito general), lo que los convierte en una alternativa viable para la computación distribuida.

No obstante, la creciente necesidad de entornos de bajo consumo energético nos obliga a buscar alternativas a estas potentes plataformas, como el uso de aceleradores hardware en FPGA, permitiendo alcanzar valores de eficiencia energética más sostenibles. Se han realizado importantes contribuciones en el desarrollo de implementaciones basadas en FPGAs, una de las tecnologías más eficaces para el análisis y compresión de imágenes hiperespectrales en el espacio en temáticas tan diversas como la estimación del número de endmembers [32, 31], la mezcla espectral [34, 35, 33], la estimación del ruido [63], la detección de objetivos [30], la reducción de la dimensionalidad [28] y diversas técnicas de compresión: sin pérdida [6, 5], con pérdida [7, 8] y casi sin pérdida [9].

A partir de esta experiencia, se ha identificado que la cantidad de algoritmos y variantes desarrolladas para el procesamiento de imágenes hiperespectrales crece a un ritmo mucho más acelerado que su implementación en tecnologías a bordo como las FPGAs. Por ello, es fundamental establecer mecanismos que permitan la generación rápida y eficiente de algoritmos de análisis y compresión para FPGAs, optimizando su rendimiento, consumo energético, utilización de recursos y fiabilidad.

A medida que aumenta la complejidad de las aplicaciones, también crece la dificultad en el diseño de bloques de hardware a bajo nivel. Poseer un conocimiento profundo de los Lenguajes de Descripción de Hardware (HDL, *Hardware Description Language*) para lograr implementaciones optimizadas no es una habilidad ampliamente extendida (en comparación con la programación de software) y requiere una inversión significativa de tiempo en el

diseño. Por este motivo, la tendencia actual es recurrir a otras alternativas de diseño, como la generación de aceleradores a partir de lenguajes de programación de alto nivel más extendidos, como C/C++ u OpenCL. Al elevar el nivel de abstracción, es necesario emplear herramientas de Síntesis de Alto Nivel (HLS, *High Level Synthesis*), lo que permite generalizar el diseño, reducir el tiempo de desarrollo y hacerlo menos dependiente de la plataforma de aplicación, aunque a costa de una pérdida en la calidad de la implementación obtenida.

En los últimos años, esta tendencia también ha llegado al análisis de imágenes hiperespectrales. La literatura reciente recoge estudios en los que, a partir de algoritmos descritos en OpenCL y utilizando herramientas HLS, se han desarrollado implementaciones en FPGA [24, 41, 40]. Aunque los resultados son prometedores, aún están lejos del rendimiento que se lograría con una implementación en VHDL. Por otro lado, el kit de herramientas Intel oneAPI [51] es muy reciente, por lo que apenas existen trabajos relevantes en esta línea, y ninguno aplicado específicamente al análisis hiperespectral. Intel oneAPI es un sistema de programación abierto, gratuito y basado en estándares, que ofrece portabilidad y rendimiento en distintas generaciones de hardware y aceleradores. Este sistema incluye un lenguaje y bibliotecas para la creación de aplicaciones paralelas. En particular, Data Parallel C++ (DPC++) es el lenguaje de programación principal de Intel oneAPI, proporcionando las características necesarias para definir funciones de procesamiento paralelo y ejecutarlas en distintos dispositivos.

El objetivo principal, aprovechando el conocimiento en arquitecturas FPGA y herramientas de síntesis, es desarrollar códigos en DPC++ que se acerquen al rendimiento de una implementación en VHDL. Para ello, realizaremos una comparativa entre la implementación de varios algoritmos representativos del análisis de imágenes hiperespectrales (estimación del número de endmembers, búsqueda de endmembers y reconstrucción de la imagen) utilizando dos enfoques diferentes: implementación directa en VHDL y empleo del lenguaje DPC++ dentro del kit de herramientas Intel oneAPI, teniendo en cuenta la arquitectura FPGA. Para ello, en primer lugar, se realizarán una serie de experimentos para evaluar el rendimiento que ofrecen distintas técnicas de paralelismo en FPGA para el lenguaje DPC++. A partir de estos resultados, podremos desarrollar una guía con recomendaciones para escribir código eficiente en DPC++ para FPGA. Tras este estudio, llevaremos a cabo la implementación en DPC++ para FPGA de un algoritmo de estimación del número de endmebers presentes en la escena (VD, *Virtual Dimensionality*), un algoritmo de extracción de endmembers (ATDCA, *Automatic Target Detection an Classification Algorithm*) y un algoritmo de estimación de abundancias (ISRA, *Image Space Reconstruction Algorithm*). Finalmente, compararemos la implementación directa en VHDL con las implementaciones obtenidas DPC++ para los algoritmos propuestos.

1.1. Aportaciones de esta tesis

La propuesta de esta Tesis Doctoral se centra en el objetivo principal de proponer una nueva cadena de desmezclado espectral paralela para dar solución al problema de los píxeles mezcla en imágenes hiperespectrales empleando herramientas HLS y hardware especializado. Para ello, se pretende que la cadena completa desarrollada sea eficiente en términos computacionales, haciendo uso de un dispositivo FPGA y el conjunto de herramientas que ofrece Intel oneAPI. Esto ha permitido analizar el rendimiento de las técnicas desarrolladas en diferentes contextos a un coste razonable en comparación con otras implementaciones HDL de la literatura.

La consecución del objetivo general anteriormente mencionado se ha llevado a cabo en la presente memoria, abordando una serie de objetivos específicos, los cuales se enumeran a continuación:

- Adquirir conocimientos previos sobre análisis hiperespectral (como imagen hiperespectral, formatos de los datos, representación de los mismos, presentación de resultados, entre otros), esenciales para poder realizar el estudio de los diferentes algoritmos.
- Conocer en detalle el funcionamiento del modelo de programación Intel oneAPI destinado a ser utilizado sobre múltiples arquitecturas hardware a través de un lenguaje de datos paralelos compilados sobre DPC++, basado en los estándares ISO C++ y Khronos Group SYCL.
- Analizar en detalle una serie de algoritmos que puedan integrarse en una cadena de desmezclado espectral, evaluando sus ventajas e inconvenientes desde el punto de vista algorítmico y considerando los parámetros de entrada requeridos. Este análisis incluye la obtención de los conocimientos necesarios sobre todas las técnicas evaluadas, así como sobre otras técnicas de clasificación comúnmente empleadas en el campo del análisis hiperespectral.
- Con respecto al algoritmo VD:
    - Establecer un estudio sobre el funcionamiento del algoritmo VD para encontrar el diseño más óptimo para su implementación.
    - Implementar el algoritmo para su ejecución en harware utilizando una FPGA que optimice el tiempo de ejecución y los recursos utilizados. Comparar la versión optimizada con una implementación en FPGA recientemente desarrollada en VHDL.

  - Ejecutar el algoritmo paralelo propuesto sobre una serie de datos reales provenientes de sensores hiperespectrales y sintéticos que permitan conocer la precisión del número de endmembers extraídos en la ejecución del algoritmo y el rendimiento obtenido.
  - Llevar a cabo un análisis comparativo de los resultados obtenidos.

- Con respecto al algoritmo ATDCA:

  - Establecer un estudio sobre el funcionamiento del algoritmo ATDCA para establecer las distintas estrategias de optimización y utilizar la más adecuada para su implementación eficiente.
  - Desarrollar una nueva versión paralela para su ejecución en hardware utilizando una FPGA que optimice el tiempo de ejecución y los recursos utilizados. Comparar la versión optimizada con una implementación en FPGA recientemente desarrollada en VHDL.
  - Ejecutar el algoritmo paralelo propuesto sobre una serie de datos reales provenientes de sensores hiperespectrales y sintéticos que permitan conocer la precisión de los endmembers extraídos en la ejecución del algoritmo y el rendimiento obtenido.
  - Evaluar la implementación propuesta a partir de los experimentos realizados.

- Con respecto al algoritmo ISRA:

  - Realizar un análisis sobre el funcionamiento del algoritmo ISRA con el fin de determinar el diseño más adecuado para su implementación.
  - Desarrollar una nueva versión paralela para su ejecución en hardware utilizando una FPGA que optimice el tiempo de ejecución y los recursos utilizados. Comparar la versión optimizada con una implementación en FPGA recientemente desarrollada en VHDL.
  - Ejecutar el algoritmo sobre una serie de datos reales provenientes de sensores hiperespectrales y sintéticos que permitan conocer la precisión que se ha logrado en la ejecución del algoritmo y el rendimiento obtenido.
  - Analizar los resultados obtenidos en los experimentos realizados.

- Seleccionar la mejor configuración de cada uno de los algoritmos implementados para integrarlos en un único flujo de trabajo en el que se asegure un procesamiento eficiente y estructurado dentro de una cadena completa de desmezclado espectral.

- Extraer conclusiones del análisis cuantitativo y comparativo realizado, y proponer posibles líneas de investigación futuras.

1.2. Organización de esta tesis

A continuación se describen brevemente los diferentes capítulos en los cuales se estructura la presente memoria de tesis.

- **Capítulo 1. Motivaciones y objetivos**. En este capítulo se describen las principales motivaciones y objetivos del presente trabajo de Tesis Doctoral. También se describe brevemente la estructura del trabajo y sus principales aportaciones.
- **Capítulo 2. Análisis hiperespectral y el uso de hardware especializado**. A lo largo de este capítulo se describen los conceptos más importantes en relación al análisis de imágenes hiperespectrales, los sensores de adquisición y las principales misiones del pasado, presente y futuro en el campo de la observación remota de la Tierra. Además, se exponen los antecedentes y el estado del arte relativo a las técnicas del desmezclado espectral o unmixing para resolver el problema de los píxeles mezcla en este tipo de imágenes con una alta dimensionalidad espectral. El capítulo concluye con un análisis de las principales características y la arquitectura de los dispositivos FPGA para la computación de altas prestaciones en aplicaciones espaciales, destacando finalmente las ventajas e inconvenientes de su utilización.
- **Capítulo 3. *Virtual Dimensionality* (VD)**. En este capítulo se realiza un estudio detallado de uno de los algoritmos más utilizados en la literatura para la estimación del número de materiales puros o endmembers presentes en una escena hiperespectral, el algoritmo *Virtual Dimensionality* (VD). Tras un minucioso estudio, se presenta una implementación *baseline* de la que se parte para después implementar una serie de optimizaciones que permitan la aceleración del algoritmo sobre una plataforma FPGA haciendo uso de DPC++ e Intel oneAPI. Finalmente, se describen los resultados de precisión junto con los resultados experimentales y una comparación con otra implementación hardware del algoritmo en VHDL.
- **Capítulo 4. *Automatic Target Detection and Classification Algorithm* (ATDCA)**. Este capítulo detalla el estudio realizado al algoritmo *Automatic Target Detection and Classification Algorithm* (ATDCA) para determinar las distintas firmas espectrales puras o endmembers que están presentes en una escena hiperespectral. Para ello, se

tienen en cuenta las diversas métricas de distancias existentes en la literatura para la implementación del operador de proyección necesario para poder obtener el conjunto de endmembers. Una vez obtenida la mejor opción, se presenta una versión *baseline* del algoritmo que permite la implementación de diversas optimizaciones capaces de acelerar el algoritmo utilizando el compilador DPC++ sobre Intel oneAPI. La última parte del capítulo concentra los resultados de precisión obtenidos sobre escenas hiperespectrales junto a los resultados experimentales con una comparación de otra implementación hardware del algoritmo utilizando el lenguaje VHDL.

- **Capítulo 5. *Image Space Reconstruction Algorithm* (ISRA)**. En este capítulo se realiza un estudio del *Image Space Reconstruction Algorithm* (ISRA), que sigue un modelo lineal de mezcla para la estimación de las fracciones de abundancia no negativas en píxeles mezcla, su implementación optimizada sobre un dispositivo FPGA utilizando el compilador DPC++ sobre Intel oneAPI y la validación de los resultados de precisión haciendo uso de la métrica *Normalised Mean Square Error* (NMSE). Finalmente, se ofrecen los resultados experimentales obtenidos y una comparación con otra implementación hardware del algoritmo en VHDL.

- **Capítulo 6. Cadena completa de desmezclado espectral**. Este capítulo presenta la implementación de una cadena completa que soluciona el problema de los píxeles mezcla en imágenes hiperespectrales. Con la cadena se describen un conjunto de tareas necesarias para ejecutarse en un único flujo de trabajo en el que se asegura un procesamiento eficiente y estructurado. Por último, se realiza la evaluación del rendimiento computacional de la cadena completa comparándola con otra cadena completa de desmezclado implementada en hardware utilizando el lenguaje VHDL.

- **Capítulo 7. Conclusiones y trabajo futuro**. En este capítulo se muestran las conclusiones obtenidas y se exponen posibles trabajos futuros o fuentes de investigación abiertas que pueden ser continuadas.

- **Publicaciones generadas**. En este apéndice se exponen las publicaciones a través de las cuales se ha difundido una parte del trabajo de investigación llevado a cabo.

- **Bibliografía**. El trabajo presenta una revisión bibliográfica detallada que permitirá al lector explorar con mayor profundidad los diversos aspectos teóricos y relacionados con la implementación de los algoritmos descritos en esta memoria.

Capítulo 2

Análisis hiperespectral y el uso de hardware especializado

En este capítulo se describe el concepto de imagen hiperespectral, detallando las particularidades y características propias de este tipo de imágenes de alta dimensionalidad, así como las características del sensor hiperespectral utilizado en este trabajo. A continuación, se muestra la evolución que han seguido las principales misiones desde las últimas décadas del siglo XX, así como las más recientes y las que están planificadas para los próximos años. Seguidamente, se presenta una visión general de las principales aplicaciones actuales que utilizan imágenes hiperespectrales para llevar a cabo análisis eficientes. Se presta especial atención a los algoritmos basados en el modelo lineal de mezcla, eje central de este trabajo, el cual aborda el desafío de la caracterización subpíxel en imágenes hiperespectrales mediante la identificación de los píxeles espectralmente más puros presentes en la escena. El capítulo concluye con las ventajas e inconvenientes del uso de hardware reconfigurable ofrecido por los dispositivos FPGAs para poder solucionar el problema de la mezcla espectral en imágenes hiperespectrales, con vistas a mejorar la explotación de este tipo de técnicas en aplicaciones reales mediante la utilización de un lenguaje de alto nivel.

2.1. Sensores hiperespectrales

Los sensores hiperespectrales capturan imágenes digitales a lo largo de un amplio conjunto de bandas espectrales estrechamente espaciadas, lo que permite obtener, para cada píxel de la escena, una "firma espectral" distintiva que actúa como una especie de "huella dactilar" del material correspondiente [59]. La información recopilada por un sensor hiperespectral

sobre una escena determinada puede representarse mediante un cubo de datos, en el que dos de sus dimensiones corresponden a la localización espacial de los píxeles (habitualmente denominadas líneas y muestras), mientras que la tercera dimensión recoge la información espectral de cada píxel en múltiples longitudes de onda [20].

La Figura 2.1 muestra la estructura de una imagen hiperespectral, en la que el eje X representa las muestras, el eje Y las líneas, y el eje Z el número de banda, es decir, la longitud de onda correspondiente a cada canal. Para ilustrar el proceso de análisis hiperespectral, la Figura 2.2 presenta un diagrama simplificado que toma como ejemplo el sensor *Airborne Visible Infra-Red Imaging Spectrometer* (AVIRIS), desarrollado por el *Jet Propulsion Laboratory* (JPL) de la NASA. Este sensor opera en un rango espectral que abarca desde 0.4 μm hasta 2.5 μm, utilizando 224 bandas con una resolución espectral aproximada de 10 nm [39].

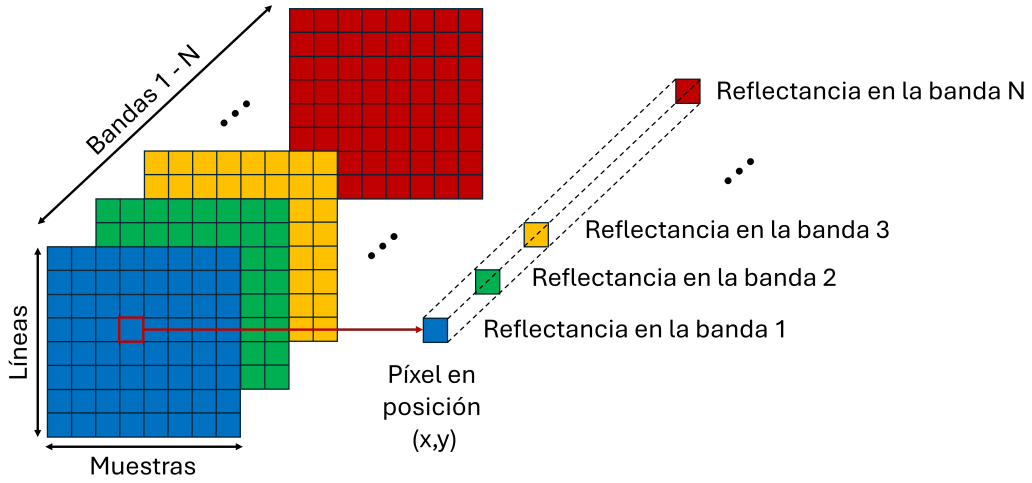


Figura 2.1 Concepto de imagen hiperespectral.

Como se observa en la Figura 2.2, la capacidad de observación del sensor permite obtener una firma espectral detallada para cada píxel de la imagen, definida por los valores de reflectancia registrados a lo largo de distintas longitudes de onda. Esta característica posibilita una caracterización altamente precisa de la superficie terrestre [52]. Cabe destacar que en las imágenes hiperespectrales es habitual la presencia de mezclas a nivel subpíxel. En términos generales, se pueden clasificar los píxeles en dos categorías: píxeles puros y píxeles mixtos [82]. Se considera píxel mixto aquel en el que coexisten múltiples materiales. Este tipo de píxeles constituye la mayoría dentro de una imagen hiperespectral, ya que el fenómeno de mezcla espectral es independiente de la escala y puede producirse incluso a niveles microscópicos [84].

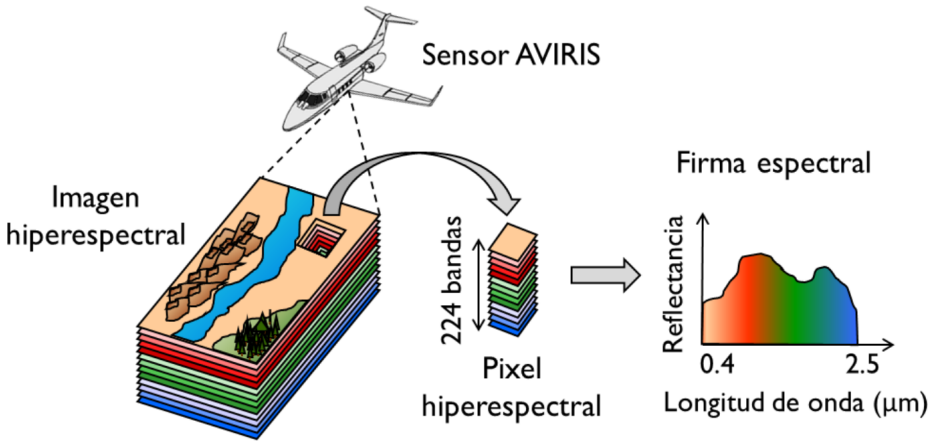


Figura 2.2 Adquisición de una imagen hiperespectral por el sensor AVIRIS.

El avance tecnológico impulsado por la incorporación de sensores hiperespectrales en plataformas de observación remota de última generación ha sido especialmente notable en los últimos años. En este contexto, cabe destacar que dos de las principales plataformas satelitales del pasado, Earth Observing-1 (EO-1) de la NASA y ENVISAT de la Agencia Espacial Europea, estuvieron equipadas con sensores hiperespectrales, lo que permitió la adquisición continua de imágenes de casi toda la superficie terrestre. No obstante, a pesar del significativo progreso en el desarrollo de instrumentos de observación remota, la evolución de las técnicas de análisis de los datos generados por estos sensores no ha seguido el mismo ritmo. En particular, el diseño de métodos avanzados de análisis hiperespectral, capaces de explotar eficientemente la abundante información espacial y espectral contenida en este tipo de imágenes, constituye aún un reto relevante y de gran interés para la comunidad científica [53, 60].

2.1.1. Misiones con sensores hiperespectrales

Las misiones hiperespectrales representan una evolución relativamente reciente en comparación con otras misiones de observación remota, como aquellas basadas en sensores multiespectrales. Si bien la tecnología hiperespectral tiene sus orígenes en investigaciones más antiguas, el desarrollo y despliegue de satélites equipados con sensores hiperespectrales para la observación de la Tierra comenzó a consolidarse a partir de las últimas décadas del siglo XX y los primeros años del siglo XXI.

Si retrocedemos a las décadas de 1970 y 1980, los primeros avances en sensores hiperespectrales surgieron principalmente en laboratorios y a través de proyectos experimentales.

Durante este período, la tecnología hiperespectral tuvo un uso predominante en plataformas terrestres y aéreas. Por ejemplo, hacia finales de la década de 1980 se llevaron a cabo experimentos con sensores hiperespectrales instalados en aeronaves y globos de gran altitud, orientados principalmente a estudios atmosféricos y de cobertura terrestre. Estas iniciativas pioneras fueron fundamentales para sentar las bases de la transición hacia la observación remota de la Tierra mediante satélites. Un ejemplo notable es el sensor Hyperspectral Imager (HIS), probado en plataformas aéreas durante los años ochenta. Aunque no se trataba de un sistema satelital, su desarrollo resultó clave para demostrar la viabilidad de la tecnología hiperespectral en la obtención de observaciones detalladas del entorno natural.

Durante la década de 1990, la tecnología hiperespectral comenzó a integrarse de forma más significativa en misiones espaciales, impulsada por el desarrollo de sensores avanzados capaces de cubrir un amplio rango espectral con alta resolución. Estos avances marcaron el inicio de la era de los satélites hiperespectrales. Un ejemplo destacado es el sensor AVIRIS, desarrollado por la NASA en la década de 1980 y operativo desde 1987. Este instrumento permitió la monitorización detallada de la superficie terrestre gracias a su alta resolución espectral, abarcando un rango de longitudes de onda que va desde los 400 nm hasta los 2500 nm. Los datos proporcionados por AVIRIS resultaron fundamentales para investigaciones relacionadas con la vegetación, la química del suelo, la composición mineralógica y la calidad del agua, siendo utilizado en numerosas campañas científicas. En 1994, la NASA realizó una prueba a bordo del transbordador espacial Endeavour, en el marco del programa Hyperion, con el objetivo de evaluar la viabilidad de emplear sensores hiperespectrales para la observación remota desde el espacio. Este tipo de experimentos sentó las bases para el desarrollo de futuros satélites hiperespectrales.

A partir de los años 2000, se lanzaron varios satélites operacionales equipados con sensores hiperespectrales capaces de proporcionar datos globales sobre la superficie terrestre y la atmósfera. Estos satélites comenzaron a utilizarse en una amplia variedad de aplicaciones, que incluyen la monitorización de la vegetación, la calidad del agua y la contaminación atmosférica. La incorporación de estos sensores supuso una revolución en la observación remota de la Tierra, al permitir la captura de información detallada a lo largo de un amplio rango de longitudes de onda, posibilitando la identificación y análisis precisos de materiales y características tanto en la superficie terrestre como en la atmósfera y los océanos. Un ejemplo destacado es el sensor hiperespectral Hyperion, lanzado a bordo del satélite EO-1 de la NASA en el año 2000. Este satélite fue uno de los primeros en ofrecer observación hiperespectral global desde el espacio. Hyperion cubría un rango espectral de 400 nm a 2500 nm mediante 220 bandas espectrales, lo que permitió realizar estudios detallados sobre la composición de la superficie terrestre, incluyendo vegetación, minerales y aguas superficiales. Aunque

EO-1 fue concebido como una misión experimental, los datos proporcionados por Hyperion fueron fundamentales para mejorar la comprensión de la biodiversidad, la salud de los ecosistemas y el cambio climático. Además, esta misión sirvió como plataforma de prueba para el desarrollo de sensores hiperespectrales futuros. Por otra parte, destaca también el satélite ENVISAT, lanzado por la Agencia Espacial Europea (ESA, *European Space Agency*) en 2002, que incorporaba diversos instrumentos, entre ellos el espectrómetro de imágenes hiperespectrales SPECTRA. Aunque SPECTRA no era un sensor completamente hiperespectral como Hyperion, proporcionó información relevante sobre la composición de la atmósfera y las emisiones de gases de efecto invernadero, contribuyendo a la monitorización ambiental a escala global.

A partir de 2010, la tecnología hiperespectral se integró de manera más profunda en las misiones de observación remota operacionales, lo que permitió que los satélites hiperespectrales alcanzaran mayores niveles de precisión y capacidades mejoradas. Gracias a su capacidad para detectar diferencias espectrales que son invisibles al ojo humano, los sensores hiperespectrales se convirtieron en una herramienta clave en diversas áreas, tales como la monitorización ambiental, la gestión de recursos naturales, la agricultura de precisión y la investigación climática. En la Tabla 2.1, se detallan algunas de las misiones espaciales actuales y futuras que emplean esta tecnología avanzada para mejorar nuestra comprensión del planeta.

	PRISMA	**HISUI**	**EnMap**	**HyspIRI**	**HypXIM**	**CHIME**
País de origen	Italia	Japón	Alemania	EE.UU.	Francia	ESA
Bandas espectrales	239	185	242	214	210	>200
Res. espacial	5-30 m	30 m	30 m	30 m	8 m	30 m
Tiempo de ciclo	3/7 días	3 días	4 días	16 días	5 días	12.5 días
Rango espectral	0.4-2.5 μm	0.4-2.5 μm	0.4-2.5 μm	0.38-2.5 μm	0.4-2.5 μm	0.4-2.5 μm
Res. espectral	10 nm	10-12.5 nm	6.5-10 nm	10 nm	10 nm	$\leq$ 10 nm
Ancho de barrido	30 km	20 km	30 km	120 km	15 km	<20 km
Cobertura terrestre	Completa	Completa	Completa	Completa	Completa	Completa
Lanzamiento	2019	2022	2026	TBD	TBD	TBD
Tiempo de vida	$\approx$ 5 anos	$\approx$ 5 anos	$\approx$ 5 anos	$\approx$ 5 anos	$\approx$ 10 anos	$\approx$ 5 anos

Tabla 2.1 Listado de algunas misiones espaciales de observación remota de la Tierra presentes y futuras, que incluyen sensores hiperespectrales.

Entre todas las misiones mencionadas, destaca la misión *PRecursore IperSpettrale della Missione Applicativa* (PRISMA), que incorporó un satélite de observación hiperespectral lanzado en 2019 con el objetivo de mejorar la comprensión del medio ambiente y de los procesos naturales a gran escala. Entre sus principales objetivos se incluyen la monitorización ambiental, el control de la contaminación y la evaluación de recursos naturales. Otro ejemplo de misiones modernas es la misión *Environmental Mapping and Analysis Program* (EnMap), desarrollada por la Agencia Espacial Alemana (DLR, *Deutsches Zentrum für Luft- und*

Raumfahrt) y lanzada en 2022. El propósito de esta misión es la observación y análisis de la superficie terrestre para aplicaciones ambientales, tales como el estudio de ecosistemas, el cambio climático, la calidad del agua y la agricultura. En definitiva, estas misiones tienen como objetivo alcanzar un conocimiento más profundo de la superficie terrestre, lo que permitirá una mejor gestión ambiental y una mayor sostenibilidad de los recursos naturales.

2.1.2. Aplicaciones con imágenes hiperespectrales

El valor de las imágenes hiperespectrales radica en su capacidad para analizar la composición material de los objetos a distancia, sin necesidad de interacción física. Esto se logra mediante la medición de la cantidad de luz reflejada (reflectancia) en un amplio rango de longitudes de onda contiguas. Esta característica permite una gran variedad de aplicaciones en diferentes campos, como la medicina, donde se emplea, entre otras cosas, para la detección de tumores [3]; la agricultura, para evaluar el estado de los cultivos [42]; y las ciencias geoespaciales, para monitorear las condiciones de la tierra y la atmósfera [17].

El desarrollo de la tecnología hiperespectral en aplicaciones de observación remota de la superficie terrestre ha dado lugar a la creación de instrumentos de medición con una resolución extremadamente alta, tanto en los dominios espacial como espectral, sin descuidar el aspecto temporal. Este notable aumento en la cantidad de información recopilada ha generado nuevos retos en el procesamiento de estos datos. En consecuencia, ha surgido la necesidad de abordar uno de los principales problemas de este tipo de imágenes: la mezcla espectral de los píxeles, con el fin de proporcionar soluciones y respuestas en tiempo real en aplicaciones como la monitorización de incendios forestales, la identificación y seguimiento de personas o vehículos militares, la detección de amenazas biológicas y la vigilancia de derrames de petróleo u otros contaminantes químicos en el mar.

El desmezclado en imágenes hiperespectrales [2, 56] es una técnica empleada para separar los espectros de los distintos materiales o componentes presentes en una escena hiperespectral. Esta técnica resulta fundamental para la interpretación y el análisis de los datos, ya que permite identificar y cuantificar la presencia de diversos elementos en una imagen. Los modelos utilizados para este proceso pueden ser lineales o no lineales (ver Figura 2.3). Los modelos lineales, como el Modelo de Mezcla Lineal (LMM, *Linear Mixing Model*) [15], son algunos de los más comunes en la literatura, ya que asumen que los espectros observados son combinaciones lineales de los espectros puros de los materiales presentes. Por otro lado, los modelos no lineales, como el Modelo de Mezcla No Lineal (NLMM, *Nonlinear Mixing Model*) [46], contemplan interacciones no lineales entre los materiales, lo que los

hace más adecuados en casos de alta reflectancia. En la práctica, el modelo lineal es más flexible, se adapta con mayor facilidad a diversos escenarios de análisis y es el más utilizado por la comunidad debido a su facilidad de implementación y su carácter no supervisado.

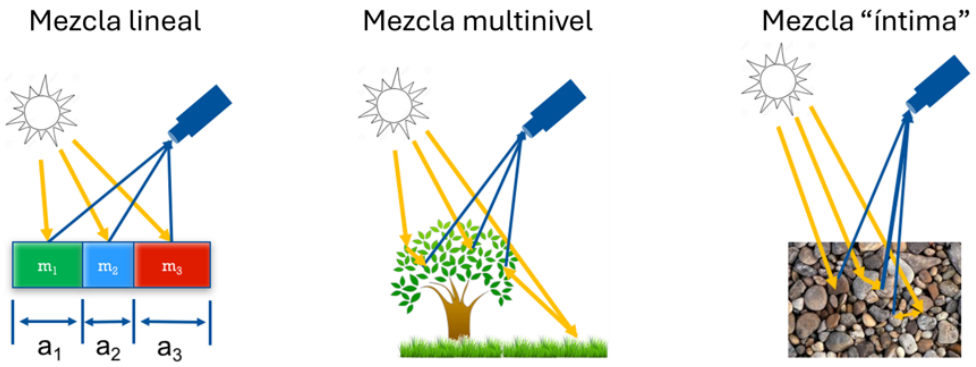


Figura 2.3 Modelo de mezcla lineal frente a modelo de mezcla no lineal.

2.1.3. Algoritmos para el desmezclado espectral

En el contexto de la observación remota de la Tierra, y particularmente en relación con el problema de los píxeles mezcla en las imágenes hiperespectrales, desarrollar una cadena completa de desmezclado representa una tarea compleja y costosa, principalmente debido a la alta dimensionalidad de estas imágenes y a las intensivas operaciones matriciales que se requieren. Generalmente, esta cadena de desmezclado consta de tres etapas principales (ver Figura 2.4): 1) estimación del número de endmembers, 2) identificación de los endmembers y 3) estimación de los mapas de abundancia correspondientes a cada uno de los endmembers presentes en la imagen. Es relevante destacar que las operaciones más costosas en estos algoritmos suelen ser repetitivas, lo que las hace altamente susceptibles a ser implementadas en arquitecturas paralelas, lo que a su vez mejora considerablemente el rendimiento y permite alcanzar tiempos de respuesta en tiempo real o casi real.

1. **Estimación del número de endmembers**. En lo que respecta a la primera etapa, se han desarrollado diversos algoritmos a lo largo de las últimas décadas. Entre los métodos clásicos, se encuentran algoritmos como el *Principal Component Analysis* (PCA) [55] y el *Minimum Noise Fraction* (MNF) [38]. La técnica PCA busca determinar el número de componentes principales necesarios para capturar la mayor variabilidad de los datos. Para ello, proyecta la imagen sobre un sistema de coordenadas que maximiza la varianza de la información de la imagen en relación con el ruido. Por su parte, la

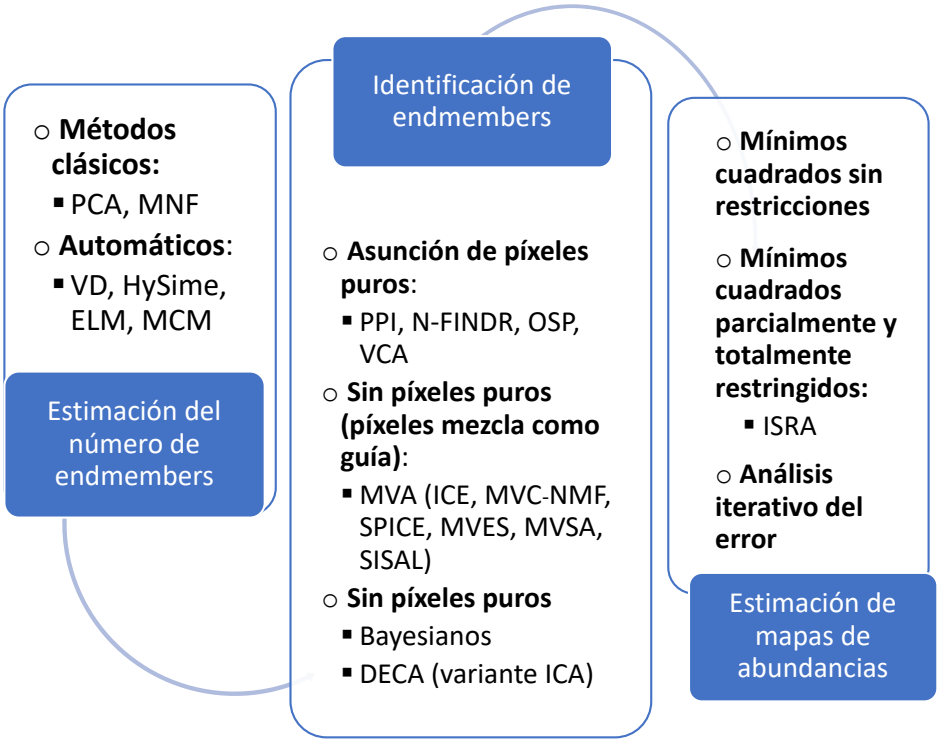


Figura 2.4 Diagrama con las etapas principales del proceso de desmezclado hiperespectral.

técnica MNF es otra transformación comúnmente utilizada, incluso más que la PCA. Esta se basa en ordenar los componentes en función de la relación señal-ruido (SNR). Cuanto mayor sea esta relación, mayor será el número de componentes puros que se pueden identificar en la escena analizada, ya que el resto correspondería al ruido.

Además de los métodos clásicos, existen otros enfoques que buscan determinar de manera automática el número de endmembers o componentes puros. Entre estos se incluyen algoritmos como *Virtual Dimensionality* (VD) [22], *Hyperspectral Subspace identification mínimum error* (HySime) [14], *Eigenvalue likelihood maximization* (ELM) [67] y el *Normal Compositional Model* (MCM) [25]. En el caso del algoritmo VD, se analizan las matrices de correlación y covarianza de los datos hiperespectrales para identificar las regiones que corresponden a señales y las que corresponden a ruido. Así, mediante una prueba estadística, se determina el número de fuentes o señales presentes en la imagen frente a las componentes de ruido. Este método es sencillo de implementar, ya que no toma en cuenta el ruido de la imagen, y actualmente existen diversas implementaciones paralelas en la literatura [11, 32, 54, 91], las cuales podrían ser objeto de comparación con una nueva implementación paralela diseñada en FPGA utilizando un lenguaje de alto nivel.

En cuanto al algoritmo HySime, se puede afirmar que es uno de los métodos más empleados en la literatura para la identificación del número de endmembers, junto con el VD. Esta técnica tiene la particularidad de considerar el ruido presente en una imagen, modelándolo previamente. Para ello, la imagen hiperespectral se proyecta en un subespacio, comparando distintas bandas entre sí para determinar cuáles contienen información útil y cuáles corresponden a ruido. Debido a su complejidad, las implementaciones paralelas disponibles [31, 95] son limitadas, y dado el tiempo de procesamiento que requiere, su incorporación en una cadena de procesamiento completa de desmezclado se ve dificultada.

Además de los algoritmos mencionados previamente, no se debe pasar por alto los métodos ELM y MCM. El primero de ellos se basa en un concepto similar al algoritmo VD, pero sin necesidad de un valor de entrada conocido como umbral de tolerancia, lo que elimina la necesidad de optimizar un valor previo para determinar el número de endmembers. Por su parte, el algoritmo MCM representa otra opción viable, cuya ventaja radica en que asume la posibilidad de que no existan píxeles puros en la imagen, lo cual es especialmente relevante cuando se trabajan con imágenes de baja resolución espacial, como aquellas con una resolución de 20-30 metros, donde es probable que todos los píxeles sean mezclas. No obstante, ambas técnicas podrían ser consideradas como alternativas al algoritmo VD, que es ampliamente reconocido por su sencillez de implementación y su precisión.

2. **Identificación de endmembers**. Esta etapa es considerada la más crítica en el proceso de desmezclado hiperespectral. Debido a su importancia, existen una amplia variedad de algoritmos que abordan tres casos principales: 1) la asunción de la existencia de píxeles puros, 2) la ausencia de píxeles puros, pero la posibilidad de utilizar otros píxeles mezclados como guía para determinarlos de manera virtual en un simplex (representación geométrica en un espacio multidimensional, donde cada dimensión corresponde a una banda espectral y los vértices serían los endmembers o guías), y 3) la falta de píxeles puros en datos altamente mezclados, donde ningún píxel se encuentra cerca de alguna de las caras del simplex, lo que requiere la determinación de un simplex virtual más pequeño que el ideal. Para simplificar el proceso de paralelización y lograr una cadena de procesamiento en tiempo real, nos centraremos en los algoritmos que suponen la presencia de píxeles puros.

   - ***Pixel Purity Index* (PPI)** [16]: Este algoritmo es muy popular y se encuentra disponible en paquetes de software como ENVI. Se basa en la generación de vectores (*skewers*) que particionan una nube de puntos cuando modelamos la

imagen en dos dimensiones, obteniendo los valores extremos en cada proyección mediante un proceso de fuerza bruta (estos valores extremos pueden repetirse en iteraciones posteriores). La principal desventaja de este algoritmo radica en la cantidad de parámetros de entrada que requiere para su ejecución, aunque es altamente paralelizable debido a la ausencia de dependencias de datos.

- **N-FINDR** [94]: Es una alternativa muy popular basada en el concepto de que los endmembers se encuentran en los vértices de un simplex. El proceso iterará buscando el volumen máximo (calculando el determinante de una matriz cuadrada definida por el número de endmembers y restringiendo el número de bandas) del simplex formado por todos los píxeles de la imagen, cuyos vértices serán los endmembers que buscamos. Comparado con el algoritmo PPI, N-FINDR se considera más estable y robusto, ya que solo requiere el número de endmembers que se desean encontrar.
- ***Orthogonal Subspace Projection* (OSP)** [44]: Es un algoritmo automático ampliamente utilizado, cuyo funcionamiento es bastante sencillo y requiere solo un parámetro de entrada. Para identificar el primer endmember, el algoritmo encuentra el píxel más alejado de todos y, a continuación, realiza una proyección ortogonal para obtener el píxel más extremo con respecto a este primer endmember. En las siguientes iteraciones, se buscará una nueva proyección ortogonal en relación con los endmembers obtenidos hasta el momento. Debido a su simplicidad, rapidez y efectividad en implementaciones paralelas disponibles en la literatura [10, 12, 13, 69], destaca la implementación OSP-GS (que realiza el proceso de ortogonalización utilizando el método de Gram-Schmidt) como un candidato para ser comparado con una nueva implementación paralela sobre FPGA.
- ***Vertex Component Analysis* (VCA)** [77]: Es considerado por la comunidad como el algoritmo estándar para la identificación de endmembers. Su funcionamiento es similar al de OSP, pero incorpora un proceso de reducción de ruido previo a la identificación del primer componente o endmember, lo que ayuda a evitar la selección de píxeles anómalos en la imagen. Aunque es un algoritmo más complejo y menos rápido en comparación con los anteriores, su robustez lo ha convertido en una de las opciones más utilizadas.

3. **Estimación de los mapas de abundancias**. Esta es la última etapa del desmezclado espectral en imágenes hiperespectrales. Para llevar a cabo esta tarea, se pueden emplear tres aproximaciones:

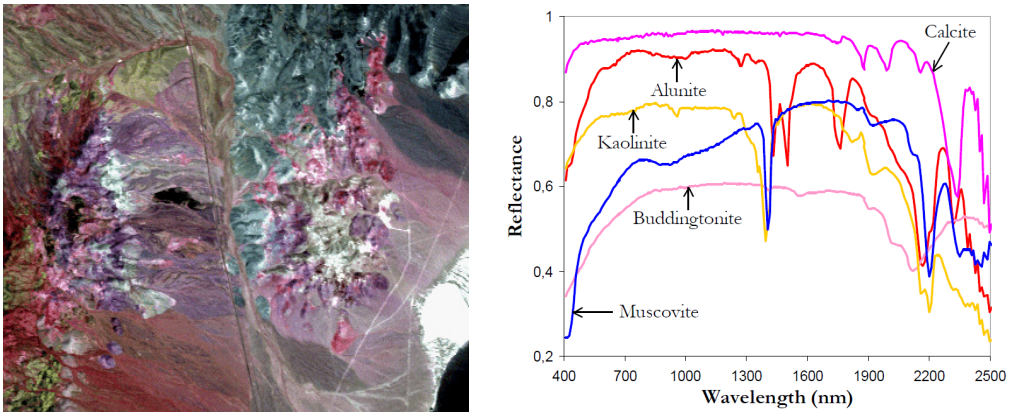
- ***Unconstrained least squares* (LSU)** [83]: Es el método más sencillo y rápido, ya que no impone restricciones de suma unitaria ni de no negatividad en las abundancias. El problema de optimización a resolver es relativamente simple, ya que depende de los endmembers estimados y de cada píxel cuya abundancia se desea estimar. Sin embargo, la solución obtenida podría ser irreal, ya que es posible obtener abundancias negativas.
- ***Partially and fully constrained least squares* (PCLS/FCLS)** [79, 45]: Son métodos que permiten incorporar restricciones de suma unitaria y no negatividad para obtener las abundancias. Para ello, es necesario añadir términos adicionales al problema de optimización a resolver. Entre estas dos restricciones, la de no negatividad se considera la más importante, ya que no tendría sentido físico obtener un valor negativo para la abundancia de un elemento o material puro en la escena analizada. Sin embargo, la restricción de suma unitaria podría ser cuestionable, ya que podríamos estar trabajando con endmembers estimados que no son reales o con píxeles ruidosos. En última instancia, la elección de las restricciones dependerá de si estamos asumiendo píxeles reales o del algoritmo utilizado para identificar los endmembers. Como alternativa, podemos considerar el método *Image Space Reconstruction Algorithm* (ISRA) [23], que es uno de los muchos métodos disponibles para estimar la abundancia de cada endmember en los píxeles de una imagen. Este enfoque busca resolver un problema lineal inverso con restricciones positivas, iterando hasta alcanzar la convergencia, lo que da lugar a mapas de abundancia significativos desde un punto de vista físico, con valores positivos. Una de las principales razones para elegir este método es su naturaleza iterativa, que permite controlar la calidad de los resultados en casos donde se busque una solución funcional y rápida, aunque con la posible concesión de un resultado final menos preciso.
- ***Iterative error analysis* (IEA)** [78]: Este método realiza de manera conjunta la extracción de los endmembers y la estimación de las abundancias. Para ello, partiendo de la imagen original, se calculará el píxel medio de la misma. Luego, se aplicará el método LSU y se obtendrá un primer mapa de abundancia que servirá para calcular una imagen reconstruida y una imagen de error. De esta última, se seleccionará el píxel con el mayor error de reconstrucción, que será considerado el primer endmember. Para extraer los endmembers restantes, se utilizará el conjunto de endmembers obtenidos hasta el momento y se aplicará nuevamente el método LSU para generar una nueva imagen de error.

2.1.4. Imágenes hiperespectrales utilizadas

En este trabajo de investigación se ha utilizado un amplio conjunto de imágenes hiperespectrales para llevar a cabo los resultados experimentales. A continuación, se describen en detalle las tres imágenes seleccionadas en esta memoria, ya que se consideran suficientemente representativas por presentar distintas características en cuanto al número de bandas espectrales y la resolución espacial (número de píxeles). Esta diversidad permite validar la robustez y eficiencia del enfoque propuesto en escenarios con distintas complejidades espectrales y espaciales.

2.1.4.1. AVIRIS Cuprite

La primera de las imágenes que ha sido utilizada para validar los resultados experimentales en el presente trabajo fue adquirida en 1997 por el sensor AVIRIS, operado por el *Jet Propulsion Laboratory* de NASA. En la Figura 2.5(a), se muestra una imagen aérea donde podemos visualizar zonas compuestas principalmente por varios minerales expuestos de interés, incluyendo *alunita*, *buddingtonita*, *calcita*, *caolinita* y *moscovita*, que caracterizan esta región denominada Cuprite, situada en la zona de Nevada, Estados Unidos. Se encuentra disponible online en unidades de reflectancia después de ser corregida atmosféricamente.



(a) (b)

Figura 2.5 (a) Composición en falso color de la escena hiperespectral capturada por el sensor AVIRIS sobre la región minera de Cuprite, en Nevada. (b) Firmas espectrales de los minerales en la librería U.S. Geological Survey utilizadas para propósitos de validación.

En cuanto a las características de esta imagen, está compuesta por un total de 350 líneas y 350 muestras, etiquetada como `f970619t01p02_r02_sc03.a.rfl` en los datos online, con un total de 224 bandas espectrales en el rango de 400 a 2500 nanómetros, una resolución

espacial de 20 metros por píxel y un tamaño total de aproximadamente 50 megabytes. Las bandas 1–3, 105–115 y 150–170 han sido eliminadas antes del análisis debido a la absorción por agua y a la baja relación señal-ruido (SNR, *signal-to-noise ratio*) en estas bandas. Para realizar la validación de la pureza de las firmas espectrales obtenidas por el algoritmo ATDCA-GS, se compararán con las firmas de referencia de los minerales mencionados (ver Figura 2.5(b)), disponibles en la biblioteca *U.S. Geological Survey* (USGS).

2.1.4.2. AVIRIS World Trade Center

La segunda imagen utilizada en este trabajo fue capturada por el sensor AVIRIS el 16 de septiembre de 2001, solo cinco días después de los atentados terroristas que destruyeron las dos torres principales y otros edificios del complejo World Trade Center (WTC) en Nueva York. A diferencia de la primera imagen, esta se encuentra en unidades de radiancia, lo que indica que no ha sido corregida atmosféricamente. Esta condición permite analizar un escenario de procesamiento a bordo, en el cual los datos se examinan de manera inmediata tras ser adquiridos por el sensor. De este modo, la detección de los focos de fuego y la generación de sus mapas de abundancia representan un desafío para el procesado en tiempo real.

En cuanto a sus características, la imagen está compuesta por un total de 614 líneas y 512 muestras, con 224 bandas espectrales en el rango de 400 a 2500 nanómetros, una resolución espacial de 1.7 metros por píxel y un tamaño aproximado de 150 megabytes. A diferencia de la escena anterior, esta imagen cubre un área menos extensa, pero con una resolución mayor en comparación con otras imágenes obtenidas por el mismo sensor, permitiendo así un estudio más detallado.

En la Figura 2.6 se muestra una composición en falso color de la imagen WTC obtenida por el sensor AVIRIS. En esta composición, se emplean las bandas espectrales correspondientes a 1682, 1107 y 655 nanómetros, asignadas a los colores rojo, verde y azul, respectivamente. Dependiendo de la combinación de colores utilizada en la imagen, las áreas con predominancia de vegetación aparecen en tonos verdes, mientras que las zonas afectadas por el fuego se muestran en tonos rojos. El humo proveniente del área del WTC (enmarcado en el rectángulo de color rojo), que se desplaza hacia el sur de la isla de Manhattan, adquiere un tono azul claro en la composición de falso color debido a su alta reflectancia en la longitud de onda de 655 nanómetros.

Figura 2.6 Composición en falso color de la escena hiperespectral capturada por el sensor AVIRIS sobre la zona del WTC en la ciudad de Nueva York, cinco días después del atentado terrorista del 11 de Septiembre de 2001 (imagen izquierda). Localización de los píxeles con alta intensidad infrarroja que indican una fuente de calor (imagen derecha).

2.1.4.3. Sintética

El tercer conjunto de datos utilizado en este trabajo ha sido generado de forma sintética empleando un conjunto de 30 endmembers seleccionados de la librería USGS y siguiendo el procedimiento descrito en [74] para simular patrones espaciales naturales. En cuanto a sus características, la imagen está compuesta por un total de 650 líneas y 750 muestras, con 224 bandas espectrales y un tamaño aproximado de 437 megabytes. La Figura 2.7 muestra una composición en falso color junto con tres ejemplos de mapas de abundancia reales, construidos a partir de firmas espectrales puras o endmembers, para esta imagen simulada.

2.2. Hardware reconfigurable

Tradicionalmente, han existido dos enfoques principales para realizar tareas de computación. El primero consiste en el procesamiento mediante hardware, utilizando circuitos interconectados de forma fija, ya sea a través de la integración en un *Application-Specific Integrated Circuit* (ASIC) o mediante la conexión de componentes discretos en una placa. El

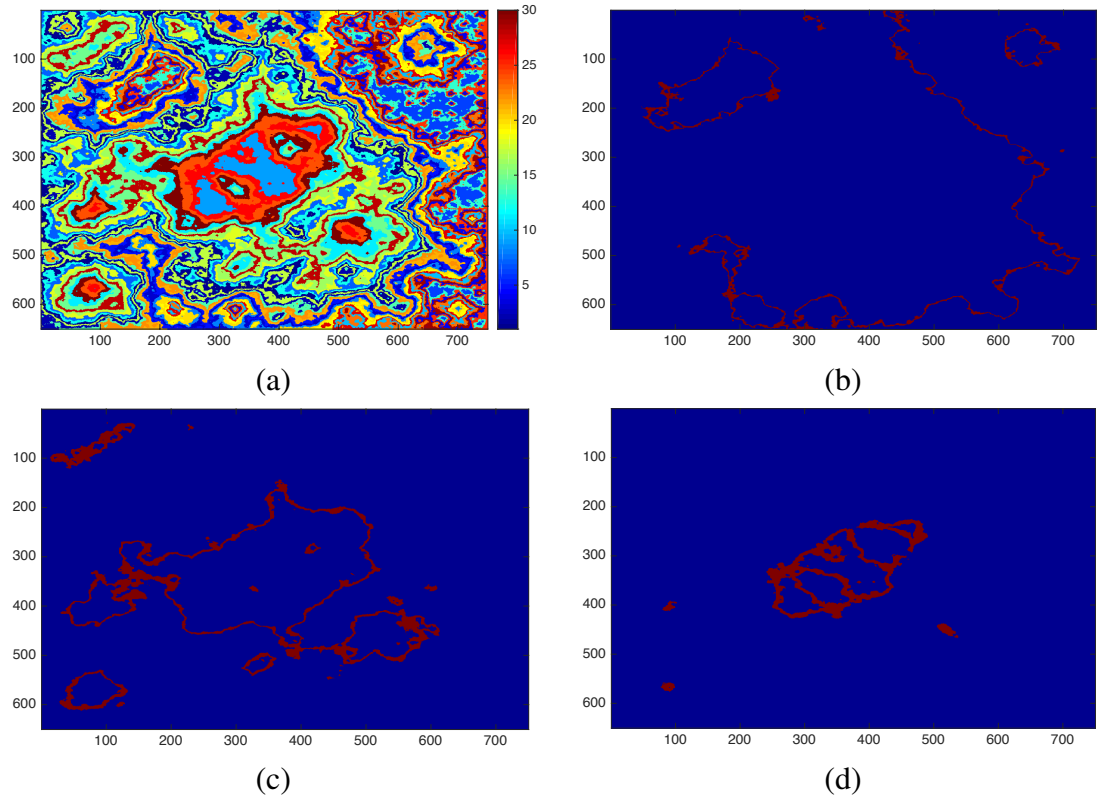


Figura 2.7 (a) Composición en falso color y tres ejemplos de mapas de abundancia reales de endmembers (firmas espectrales puras) en el conjunto de datos hiperespectrales sintéticos. (b) Endmember #3. (c) Endmember #15. (d) Endmember #26.

segundo enfoque es el procesamiento por software, que se apoya en procesadores capaces de ejecutar un conjunto de instrucciones predefinidas.

La primera opción ofrece un alto rendimiento, ya que el hardware se diseña específicamente para una tarea determinada. Esto permite obtener soluciones altamente optimizadas en cuanto a velocidad y consumo energético, empleando únicamente los recursos estrictamente necesarios. No obstante, esta eficiencia viene acompañada de un importante inconveniente: la falta total de flexibilidad. Al no poder modificar el circuito tras su fabricación, cualquier cambio en el diseño obliga a repetir el proceso completo, lo que incrementa de manera considerable el coste económico. Por ello, es imprescindible que el diseño inicial sea meticuloso y definitivo.

En contraste, el procesamiento por software permite definir la lógica de una tarea mediante algoritmos escritos en lenguajes de alto nivel, los cuales son posteriormente traducidos a lenguaje máquina por un compilador. Esta aproximación ofrece una gran flexibilidad, ya que un mismo procesador puede ejecutar múltiples programas o variantes del mismo, facilitando

la introducción de mejoras o modificaciones. Sin embargo, esta ventaja se ve contrarrestada por una menor eficiencia, debida a la secuencia necesaria de lectura, decodificación y ejecución de instrucciones.

En este contexto, las GPUs y el hardware reconfigurable emergen como soluciones intermedias, equilibrando flexibilidad y facilidad de uso con rendimiento y eficiencia energética [43, 90] (ver Figura 2.8). Estos dispositivos permiten modificar la funcionalidad del sistema sin necesidad de un nuevo proceso de fabricación. Aunque no alcanzan la flexibilidad de un procesador de propósito general ni la especialización de un ASIC, combinan beneficios de ambos mundos.

Figura 2.8 Comparación de las diferentes alternativas de diseño.

Una de las opciones más populares dentro del hardware reconfigurable son las FPGAs. A pesar de que su rendimiento puede ser comparable al de las GPUs de la misma generación, las GPUs han ganado mayor popularidad debido a su facilidad de programación. Sin embargo, las FPGAs destacan por su menor tamaño, mayor eficiencia relativa y por no requerir conexión a un *host*, lo que las convierte en una opción especialmente adecuada para aplicaciones en sistemas empotrados.

2.2.1. Arquitectura de una FPGA

La estructura interna de una FPGA puede variar dependiendo del fabricante y de la familia a la que pertenezca, aunque todas comparten una base arquitectónica común que ha sido perfeccionada con el tiempo (ver Figura 2.9). En el caso de Intel Stratix 10, la FPGA está formada por una matriz de Logic Array Blocks (LABs), cada uno de los cuales agrupa

varias Adaptive Logic Modules (ALMs). Cada ALM contiene elementos fundamentales como Look-Up Tables (LUTs), registros, multiplexores y lógica de acarreo, que permiten implementar funciones lógicas complejas y operaciones aritméticas a nivel de hardware.

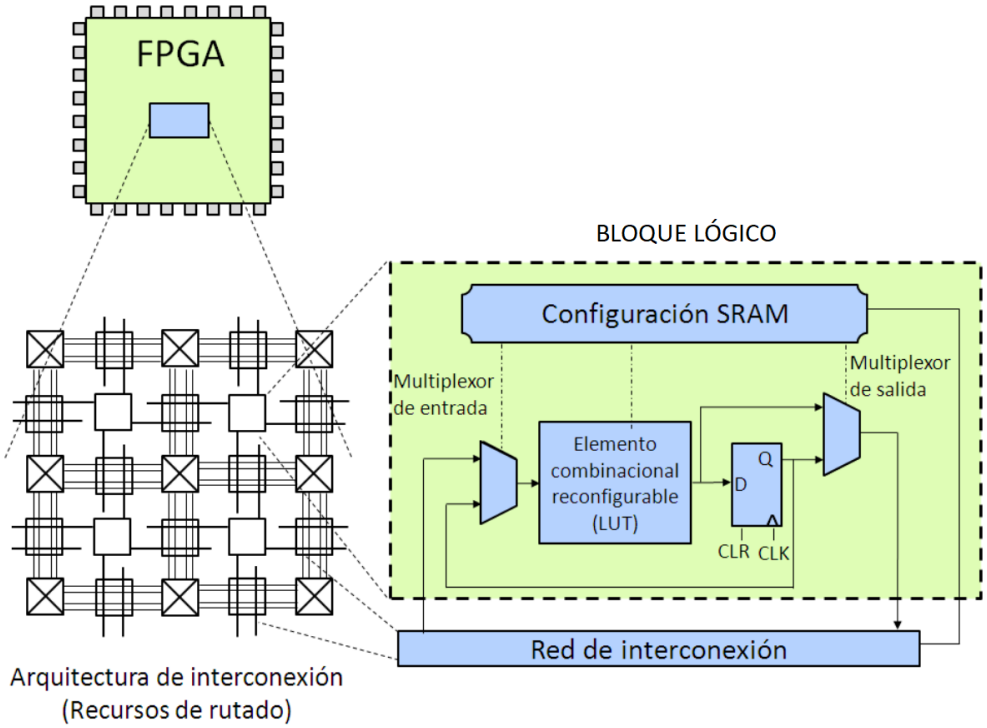


Figura 2.9 Modelo genérico de una FPGA.

La funcionalidad tanto de los bloques lógicos como de la red de interconexión se configura mediante la carga de datos en la memoria de configuración de la FPGA. Esta memoria puede ser de tipo anti-fuse o SRAM, dependiendo del tipo de dispositivo. Las FPGAs basadas en tecnología anti-fuse presentan la ventaja de ser no volátiles, lo cual resulta crítico en sectores como el aeroespacial o el médico. Además, ofrecen una latencia de conexión muy reducida, lo que incrementa su rendimiento. Su principal inconveniente es que, una vez programadas, no pueden reconfigurarse, a diferencia de las FPGAs basadas en SRAM, que permiten múltiples reprogramaciones.

Cada LAB dispone de una red interna de interconexión para sus ALMs y para conectarse al resto de la FPGA. Estas interconexiones se realizan mediante switches programables que definen las rutas eléctricas. Un aspecto destacable de las LUTs, especialmente útil en aplicaciones de alta frecuencia, es que su retardo de propagación permanece constante independientemente de la función implementada.

La interacción de la FPGA con el entorno externo se realiza mediante bloques especiales llamados *Input-Output Blocks* (IOBs). Estos permiten configurar cada pin de entrada/salida para operar como entrada, salida o bidireccional, y son compatibles con múltiples niveles de voltaje, estándares de señal y frecuencias de operación, lo que facilita su integración en distintos sistemas.

Con el objetivo de ampliar las capacidades de procesamiento y almacenamiento, las FPGAs modernas suelen incorporar bloques de memoria RAM dedicada y unidades de procesamiento digital como los *Digital Signal Processors* (DSPs). Esta integración reduce el uso excesivo de LUTs cuando se requiere manejar grandes volúmenes de datos o realizar operaciones matemáticas complejas. Algunas versiones incluso incluyen microprocesadores embebidos, facilitando el desarrollo de sistemas basados en co-diseño hardware/software.

Para el desarrollo de esta tesis se ha empleado una FPGA de la familia Stratix 10, modelo 1SX280HN2F43E2VG, fabricada por Intel. Este dispositivo, construido con tecnología de 14 nm, ofrece una destacada relación entre rendimiento y consumo energético, dispone de un elevado número de bloques DSP y proporciona un ancho de banda de E/S considerable para satisfacer las exigencias de diseños avanzados. La Tabla 2.2 presenta el desglose detallado de los componentes internos de esta FPGA.

Tipo	**Componente**	**Cantidad**
Recursos lógicos	Elementos Lógicos (LEs)	2.753.000
	Módulos Lógicos Adaptables (ALMs)	933.120
	Registros de ALM	3.732.480
	PPLs	24
Recursos heterogéneos	Memoria Integrada Máxima (*Mb*)	244
	Bloques DSP	5.760
	Sistema de Procesador Físico (HPS)	ARM Cortex-A53 de 4 núcleos a 64 bits
Recursos E/S	transceptor PCIe Gen3, 100G Ethernet	1

Tabla 2.2 Tabla de componentes internos de la FPGA Intel Stratix 10 SX 2800.

En el caso específico del modelo 1SX280HN2F43E2VG, se dispone de 2.800.000 elementos lógicos (LEs, *Logic elements*), organizados en bloques de lógica denominados *Logic Array Blocks* (LABs), cada uno compuesto por *Adaptive Logic Modules* (ALMs). Cada ALM es una unidad configurable que permite implementar funciones lógicas, aritméticas y de

registro. Además, una cuarta parte de los LABs puede configurarse como *Memory LABs* (MLABs), dedicados a funciones de memoria.

En cuanto a los recursos de memoria, el dispositivo cuenta con bloques *Embedded SRAM* (eSRAM) de 45 Mb, ideales para aplicaciones que requieren alta velocidad y ancho de banda. También dispone de bloques M20K de 20 kilobits, adecuados para matrices de memoria más grandes con múltiples puertos independientes. Los MLABs, mencionados anteriormente, son bloques de memoria mejorados configurados a partir de los LABs, óptimos para registros de desplazamiento y buffers FIFO anchos y poco profundos.

Además, el modelo 1SX280HN2F43E2VG incorpora un sistema de procesador integrado (HPS, *Hard Processor System*) basado en un procesador ARM Cortex-A53 de 64 bits con cuatro núcleos, funcionando a una frecuencia de hasta 1,5 GHz. Este HPS permite ejecutar sistemas operativos y aplicaciones en el dispositivo, facilitando el desarrollo de sistemas embebidos complejos.

Como se acaba de mencionar, para el desarrollo de esta tesis se ha empleado una FPGA de la familia Intel Stratix 10. Esta elección se justifica por su elevada capacidad lógica, el soporte avanzado para herramientas de desarrollo como Intel oneAPI y DPC++, así como por su rendimiento superior en tareas de cómputo paralelo intensivo. Además, esta arquitectura se encuentra disponible dentro del entorno Intel DevCloud, lo que ha permitido su uso sin necesidad de adquirir hardware adicional y ha facilitado el despliegue de pruebas directamente sobre el dispositivo objetivo. Frente a otras arquitecturas, como AMD Xilinx Versal o Microsemi PolarFire, Stratix 10 ofrece una integración más directa con el flujo HLS utilizado en esta tesis y una mayor madurez en el ecosistema de desarrollo.

2.2.2. Arquitectura interna de Intel Stratix 10

Intel Stratix 10 es una familia de FPGAs de gama alta diseñada para satisfacer las necesidades de procesamiento intensivo en aplicaciones como visión artificial, redes, análisis de datos y computación científica. Esta familia de dispositivos introduce la arquitectura HyperFlex, que representa una mejora significativa con respecto a generaciones anteriores como Stratix V o Arria 10. La arquitectura HyperFlex, combinada con avances en lógica programable, memoria y conectividad de alta velocidad, permite alcanzar frecuencias operativas superiores y una mayor eficiencia energética.

El componente más distintivo de esta arquitectura es precisamente el sistema HyperFlex. A diferencia de arquitecturas tradicionales que utilizan registros solo en puntos específicos de la lógica, HyperFlex introduce una capa adicional de registros distribuidos a lo largo del entramado de interconexiones y bloques lógicos. Estos registros, denominados Hyper-Registers, permiten realizar técnicas avanzadas de retiming automático, así como un pipeline más agresivo en el flujo de datos. Como resultado, se pueden alcanzar frecuencias cercanas a 1 GHz en algunos diseños, manteniendo al mismo tiempo una baja latencia y un consumo dinámico reducido. Esta característica es especialmente relevante en contextos donde se requiere tanto rendimiento como eficiencia energética, como sucede en sistemas embebidos de alta demanda o en aceleración por hardware.

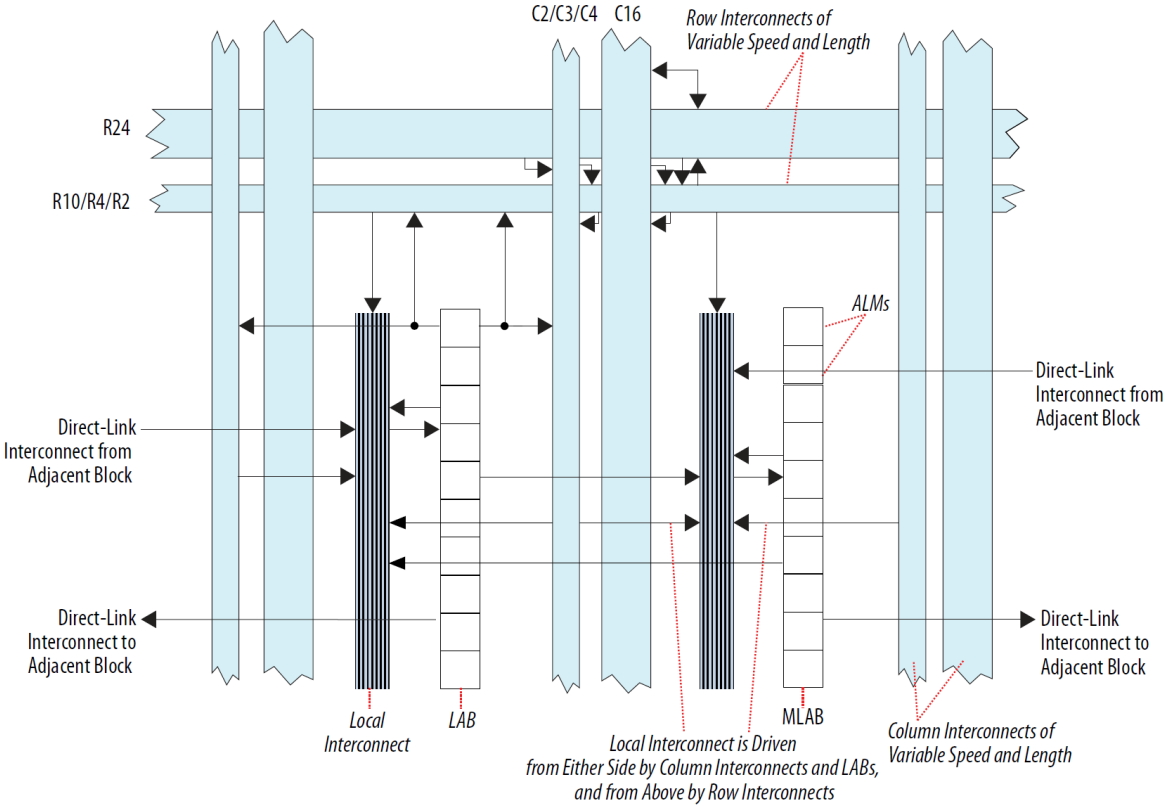


Figura 2.10 Visión general de la estructura del LAB y las interconexiones en Intel Stratix 10 [49].

En el núcleo del dispositivo se encuentra la matriz de lógica programable, construida a partir de bloques conocidos como Logic Array Blocks (LABs) (ver Figura 2.10). Cada LAB agrupa múltiples Adaptive Logic Modules (ALMs) (ver Figura 2.11), que a su vez contienen LUTs (Look-Up Tables), flip-flops y cadenas de acarreo (carry chains). Esta estructura altamente configurable permite implementar desde funciones lógicas simples hasta operadores aritméticos complejos. La organización interna de los ALMs está optimizada

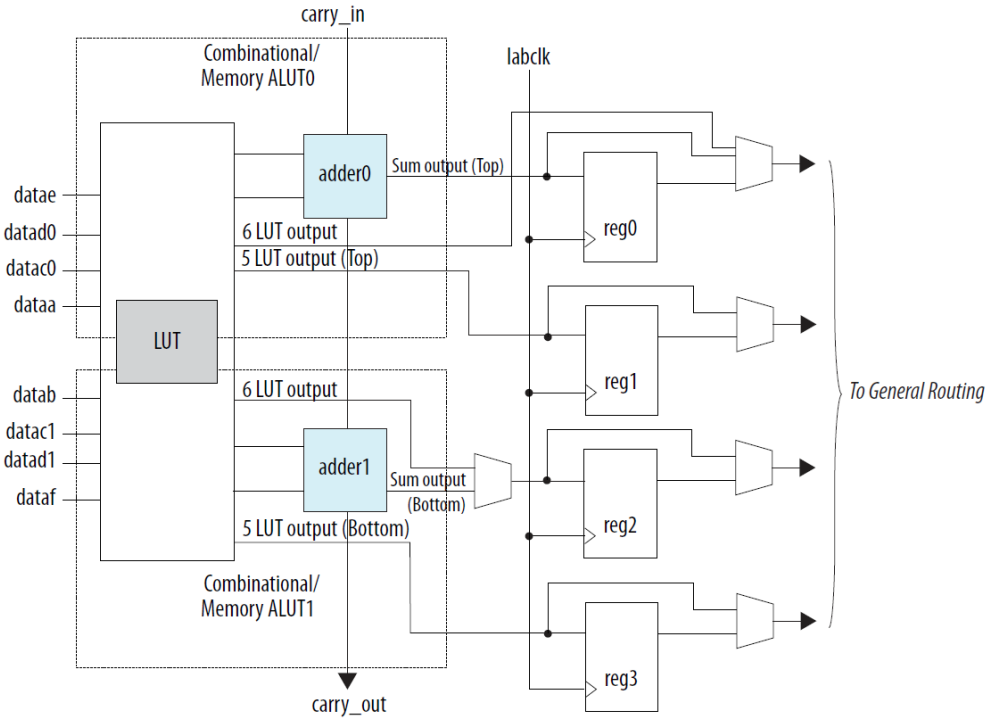


Figura 2.11 Diagrama de bloques a nivel alto del ALM [49].

para ejecutar operaciones combinacionales, registros, multiplexores y funciones aritméticas con alta eficiencia. Además, la arquitectura permite reconfigurar dinámicamente partes de esta lógica durante la ejecución, lo que mejora aún más su flexibilidad.

Stratix 10 también incorpora una jerarquía de memorias embebidas que proporciona almacenamiento local eficiente sin necesidad de acceder constantemente a memorias externas. En primer lugar, se encuentran los MLABs, pequeñas memorias distribuidas en los LABs que son ideales para almacenar datos temporales o constantes. En segundo lugar, los bloques M20K, de 20 kilobits cada uno, permiten implementar RAMs, ROMs y estructuras como colas FIFO, ofreciendo acceso dual y soporte para corrección de errores (ECC). Esta combinación permite al diseñador elegir entre distintas opciones de latencia, ancho de banda y capacidad, según lo requiera la aplicación.

Otro componente clave de la arquitectura Stratix 10 son los bloques DSP (*Digital Signal Processing*), que están diseñados específicamente para acelerar operaciones matemáticas intensivas. Estos bloques incluyen multiplicadores de alta precisión, acumuladores y operadores de suma/resta, compatibles tanto con aritmética de punto fijo como de punto flotante. La capacidad de realizar operaciones en formato IEEE 754, incluyendo precisión simple y media, permite abordar tareas complejas como filtrado digital, transformadas rápidas de Fourier (FFT) o inferencia en redes neuronales. La integración de estos bloques DSP en la

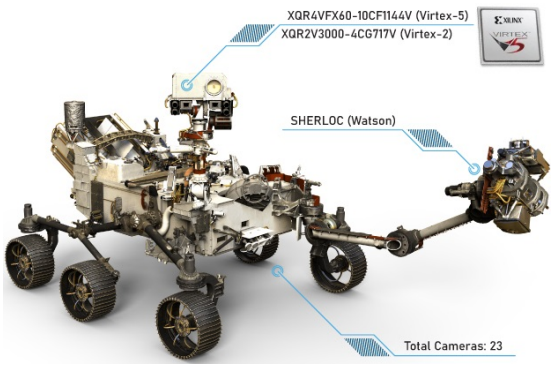
matriz programable reduce la necesidad de recurrir a recursos externos o a lógica general, mejorando la eficiencia del sistema.

Por último, el sistema de interconexión ha sido rediseñado para maximizar el rendimiento general del dispositivo. Stratix 10 utiliza un entramado jerárquico de rutas de señal, con múltiples niveles de conmutadores y segmentos de interconexión de diferentes longitudes, que permiten optimizar el rutado tanto a corta como a larga distancia dentro del chip. Esta red de interconexión se beneficia también de los Hyper-Registers, que permiten dividir rutas largas en segmentos más cortos y equilibrados, reduciendo así la latencia y permitiendo un mayor paralelismo. Esta arquitectura de rutado flexible y de alto rendimiento es fundamental para escalar diseños complejos sin que ello implique penalizaciones en velocidad o consumo.

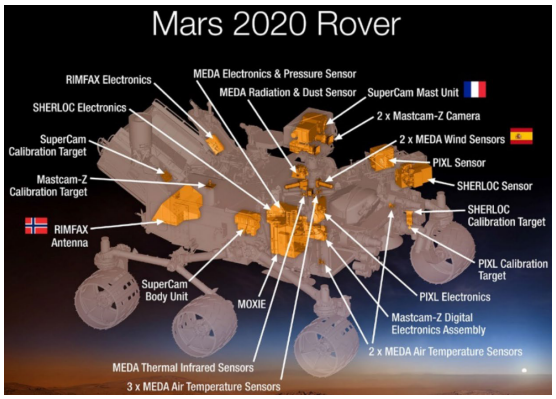
2.2.3. Aplicaciones con FPGA

Las FPGAs han despertado un notable interés en sectores estratégicos como la investigación espacial y los satélites comerciales desde hace décadas. Fabricantes como AMD Xilinx y Microchip Technology Inc han desarrollado dispositivos tolerantes a la radiación que han obtenido la calificación *Qualified Manufacturers List* (QML) de Clase V, cumpliendo con los estándares más rigurosos para aplicaciones espaciales [75, 98]. Estas FPGAs avanzadas permiten construir sistemas de alto rendimiento, robustos y fiables frente a los efectos de la radiación iónica, lo que resulta esencial en misiones expuestas a condiciones extremas [86]. Una de las primeras misiones que demostró la idoneidad de estas tecnologías fue *Solar Anomalous and Magnetospheric Particle Explorer* (SAMPEX), enfocada en el estudio de partículas energéticas y rayos cósmicos [93]. Esta misión utilizó una FPGA desarrollada por Matsushita Electric Corporation como parte de su sistema de procesamiento de datos y control de tolerancia a fallos [68]. Operó exitosamente durante 12 años, consolidando la fiabilidad de estas soluciones.

Desde entonces, muchas misiones han incorporado FPGAs en sus sistemas. En Rosetta, lanzada en 2004, se emplearon más de 400 dispositivos de Actel en etapas de desarrollo y operación. Ejemplos destacados incluyen el instrumento *Rosetta Plasma Consortium* (RPC), que utiliza 23 FPGAs para el control de sus subunidades, y el espectrómetro de masas COSIMA, donde estas gestionan el motor de la unidad manipuladora de muestras [26]. También se integraron en sistemas como GIADA y OSIRIS para controlar mecanismos ópticos y electrónicos [65]. La misión Venus Express de la ESA utilizó hardware reconfigurable basado en una FPGA Virtex de AMD Xilinx, resistente a la radiación, como parte de un SoC en el módulo de captura de imágenes *Venus Monitoring Camera* (VMC) [70]. En el caso del Solar



(a) Perseverance Mars Rover [27].



(b) Sistemas alimentados en el Rover Mars Perseverance por FPGAs AMD Xilinx Virtex [96].

Figura 2.12 FPGAs en misiones Rover.

Orbiter, el instrumento SO/PHI marcó un hito al utilizar reconfiguración dinámica parcial en vuelo, abriendo nuevas posibilidades para misiones futuras [47].

En misiones a Marte, como la del Rover Perseverance, se integraron FPGAs de la familia Virtex 5 de AMD Xilinx para tareas críticas como la entrada, descenso y aterrizaje. La capacidad de reconfiguración permitió reutilizar módulos en distintas etapas de la misión, reduciendo carga útil y costes. Además, se incorporaron algoritmos de aprendizaje autónomo, que mejoraron significativamente el rendimiento respecto a misiones anteriores como la del Curiosity [62]. La Figura 2.12b ilustra los componentes del Rover que utilizan esta tecnología.

En el sector militar, las FPGAs están presentes desde sus inicios, incorporándose en sistemas de navegación, comunicación, guiado de misiles y contramedidas electrónicas. En aviones como el F-35, estos dispositivos son fundamentales para tareas críticas como el control de motores, sistemas de frenos o identificación de objetivos [88, 48]. También se emplean para la seguridad en comunicaciones mediante cifrado y funciones antifalsificación, como ocurre con la gama militar de AMD Xilinx lanzada en 2019.

En el ámbito científico, centros como el CERN utilizan FPGAs en experimentos de física de partículas para el procesamiento en tiempo real, control de sistemas y transmisión de grandes volúmenes de datos [76].

La industria automotriz ha adoptado ampliamente estas tecnologías debido al auge del vehículo eléctrico, la conducción autónoma y la necesidad de computación avanzada. Las FPGAs permiten procesamiento en tiempo real para sistemas de seguridad, asistencia a la

conducción, análisis de entorno y sensores internos de cabina, superando las limitaciones de los microcontroladores tradicionales [72, 97].

Empresas tecnológicas como Microsoft y Amazon también han incorporado FPGAs en sus infraestructuras. Microsoft las utiliza en Azure para acelerar procesos de *Machine Learning* y mejorar su motor de búsqueda Bing [73], mientras que Amazon ofrece instancias de máquinas virtuales con aceleración basada en FPGAs [4].

El sector de las telecomunicaciones constituye la mayor cuota de mercado para las FPGAs, gracias a su uso en redes 3G, 4G y 5G, procesamiento de paquetes, comunicación óptica y gestión del ancho de banda.

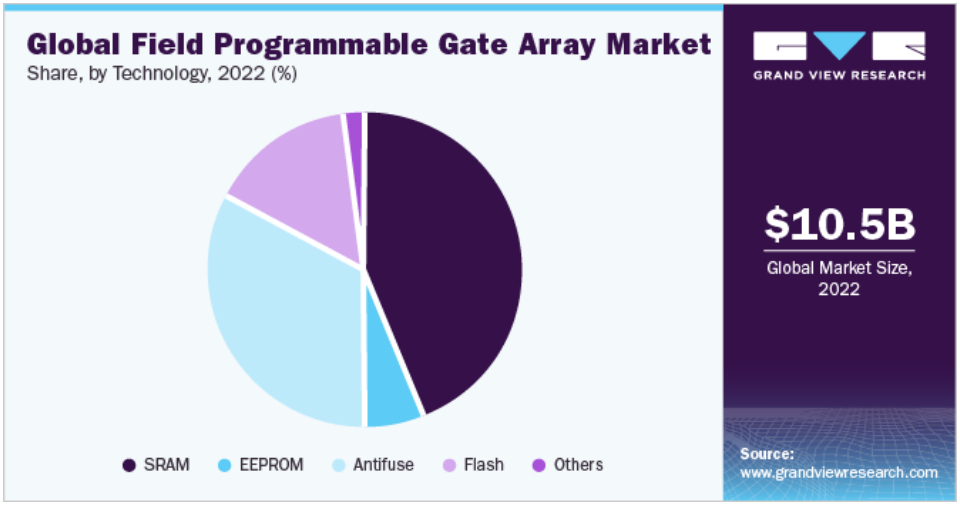


Figura 2.13 Cuota de mercado por tecnología en el año 2022 [37].

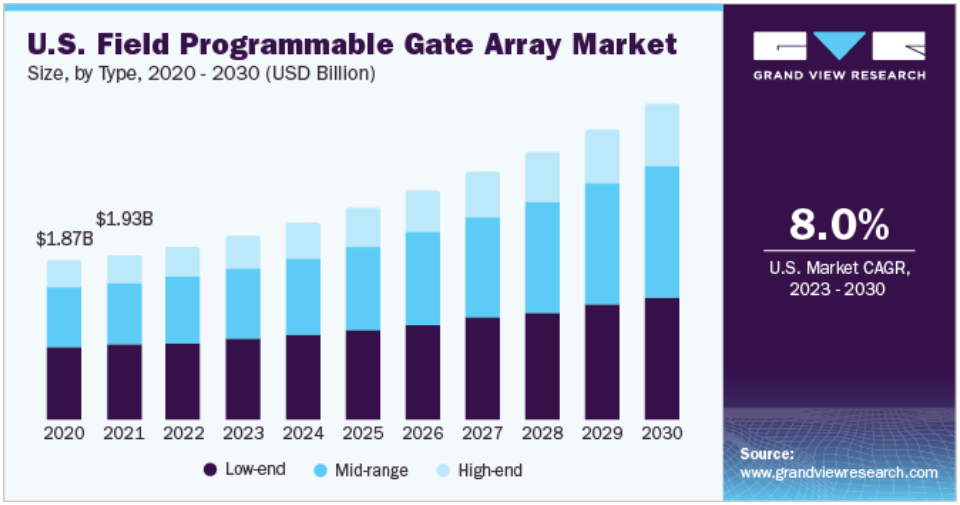


Figura 2.14 Uso y crecimiento de FPGAs por año [37].

Como se observa en la Figura 2.13, el mercado global de FPGAs fue valorado en 10.460 millones de dólares en el año 2022, y se estima que crecerá a una tasa de crecimiento anual del 10.8 % durante el periodo comprendido entre 2023 y 2030 a nivel global y del 8.0 % en el mercado de Estados Unidos (ver Figura 2.14). La región de Asia-Pacífico concentró la mayor cuota de ingresos, con más del 46 % del total en el año 2022, y se espera que mantenga su posición dominante a lo largo del periodo de previsión, impulsada principalmente por su elevada capacidad de fabricación.

2.2.4. Ventajas e inconvenientes de las FPGAs

A continuación, se detallan las principales ventajas e inconvenientes asociados al uso de la tecnología basada en FPGAs.

- Ventajas:

  1. **Rendimiento**. Las FPGAs ofrecen un notable incremento en la velocidad de procesamiento gracias a tres pilares fundamentales: segmentación, paralelismo y ejecución directa en hardware. La segmentación permite dividir una tarea en múltiples etapas, posibilitando la ejecución simultánea de distintas partes del proceso. Además, es factible replicar unidades funcionales para operar en paralelo sobre diferentes conjuntos de datos. Por último, al tratarse de hardware específico, el circuito realiza únicamente las operaciones necesarias, con una ruta de datos optimizada, lo que contrasta con los procesadores convencionales, que deben adaptarse a un conjunto de instrucciones más generalista.
  2. **Consumo**. El consumo energético de estos dispositivos es significativamente inferior al de otras plataformas como procesadores o GPUs, debido tanto a los avances tecnológicos como a la eficiencia de los diseños implementados. Solo las secciones del circuito que intervienen en la ejecución consumen energía, ya que se diseñan a medida del problema. Esta eficiencia energética, unida al alto rendimiento, proporciona una excelente relación entre consumo y capacidad de procesamiento. Las investigaciones actuales continúan enfocadas en seguir reduciendo el consumo.
  3. **Flexibilidad**. Una de las principales fortalezas de las FPGAs es su capacidad de reconfiguración, lo que permite modificar el diseño para corregir errores, adaptarlo a nuevas versiones o incluso redefinir completamente su funcionalidad. Esta característica la convierte en una solución versátil y económica para un entorno tecnológico en constante evolución.

4. **Coste**. La popularización del uso de las FPGAs, junto con los avances en sus procesos de fabricación, ha favorecido una significativa reducción de costes, especialmente en producciones a gran escala. Esto las posiciona como una alternativa cada vez más competitiva frente a soluciones tradicionales.

A estas ventajas se suma la disponibilidad creciente de herramientas y entornos de desarrollo más sofisticados y accesibles, que facilitan el trabajo con estos dispositivos.

- Inconvenientes:

1. **Tiempo de desarrollo**. La implementación de circuitos en FPGAs requiere el uso de un HDL, lo cual demanda conocimientos técnicos especializados, además de procesos de verificación complejos. Los lenguajes más empleados actualmente son VHDL y Verilog. Aunque existen herramientas de síntesis automática desde lenguajes de alto nivel que permiten acortar los tiempos de desarrollo, estas soluciones aún presentan limitaciones y no alcanzan la calidad de diseño obtenida por desarrolladores expertos en HDL.
2. **Tiempo de reconfiguración**. Uno de los retos de esta tecnología es el tiempo necesario para reconfigurar una FPGA, que, aunque se encuentra en el orden de milisegundos, aún representa una desventaja en determinados contextos. Se están desarrollando nuevas tecnologías y arquitecturas, así como técnicas innovadoras de reconfiguración, con el objetivo de reducir este tiempo.
3. **Rutado de señales**. El rutado automático generado por los sintetizadores no siempre es óptimo, por lo que es habitual que un diseñador deba intervenir manualmente para mejorar el trazado de señales y así obtener mejores resultados en términos de rendimiento.

2.2.5. Diseño en FPGA mediante lenguajes de alto nivel

El diseño de sistemas digitales sobre FPGAs ha estado tradicionalmente ligado al uso de lenguajes de descripción hardware (HDL), como VHDL o Verilog. Estos lenguajes, aunque potentes y flexibles, requieren conocimientos especializados y un considerable esfuerzo de desarrollo y verificación. A medida que aumentan la complejidad y el tamaño de los sistemas digitales, esta barrera de entrada se convierte en un obstáculo importante, especialmente para investigadores e ingenieros provenientes del ámbito del software.

Con el objetivo de acortar los ciclos de desarrollo y facilitar la adopción de FPGAs en dominios no especializados en diseño hardware, en la última década ha surgido un fuerte

impulso hacia el uso de lenguajes de alto nivel (HLLs, *High-Level Languages*) para generar aceleradores hardware, apoyado en herramientas de *High-Level Synthesis* (HLS). Estas herramientas permiten describir el comportamiento del sistema en lenguajes como C, C++, OpenCL, o SystemC, y sintetizar directamente circuitos digitales a partir de dicha descripción [18].

Entre las herramientas más representativas se encuentran:

- **AMD Xilinx Vitis HLS**: anteriormente conocida como Vivado HLS, permite generar lógica RTL a partir de código C/C++ orientado a dispositivos de AMD Xilinx. Ha sido ampliamente utilizada en dominios como visión artificial, redes neuronales, y procesamiento de señales.
- **Intel HLS Compiler**: diseñado para su uso con dispositivos Intel (anteriormente Altera), esta herramienta permite la compilación de código C/C++ hacia lógica programable. Además, Intel ha promovido el uso de OpenCL y, más recientemente, de DPC++ como parte del ecosistema oneAPI.
- **OpenCL para FPGAs**: varios fabricantes han adoptado OpenCL como estándar para la programación heterogénea. A través de entornos como Intel FPGA SDK for OpenCL o Vitis Unified Software Platform, es posible escribir *kernels* en OpenCL que luego se implementan en hardware.
- **Data Parallel C++ (DPC++) y oneAPI**: promovido por Intel, DPC++ es una extensión de SYCL (basado en C++) que proporciona una sintaxis familiar a desarrolladores de software y permite desplegar aplicaciones paralelas en múltiples tipos de dispositivos (CPU, GPU, FPGA). Aunque su soporte en FPGA aún se encuentra en expansión, representa una de las aproximaciones más prometedoras en cuanto a portabilidad y productividad [50].

Diversos estudios han comparado el rendimiento y la eficiencia de diseños generados mediante HLS con respecto a implementaciones manuales en VHDL. Si bien las soluciones en HDL siguen ofreciendo mayor control y optimización de recursos, las soluciones en HLS permiten alcanzar resultados competitivos con un menor esfuerzo de desarrollo [58]. Además, las técnicas de optimización en HLS (como *loop unrolling*, *pipelining*, *dataflow*, y *array partitioning*) permiten a los desarrolladores mejorar significativamente el rendimiento sin abandonar el paradigma de alto nivel [99].

Si nos centramos en el uso de DPC++, nos podremos encontrar con una serie de primitivas para poder expresar de manera explícita el paralelismo y optimizar el rendimiento en

arquitecturas heterogéneas, especialmente en dispositivos como FPGAs. Estas primitivas permiten al programador guiar al compilador en la generación de hardware eficiente, adaptado a las características del algoritmo. Entre las principales se encuentran *loop_unroll*, que desenrolla bucles para aumentar el paralelismo a nivel de instrucción; *pipeline*, que organiza las operaciones en etapas superpuestas para maximizar el throughput; y *dataflow*, que permite definir regiones concurrentes del código conectadas mediante canales, promoviendo una arquitectura modular y altamente paralela. Adicionalmente, la primitiva *selector* facilita la elección del dispositivo de ejecución en tiempo de ejecución, lo que mejora la portabilidad del código. El uso adecuado de estas primitivas no solo permite mejorar el rendimiento y la escalabilidad, sino también adaptar el diseño a las restricciones de recursos y latencia propias del hardware de destino.

En el campo del análisis de imágenes, y en particular en el tratamiento de datos hiperespectrales, las herramientas HLS han comenzado a consolidarse como una alternativa viable. Existen trabajos recientes donde algoritmos de reconstrucción espectral o detección de targets han sido implementados mediante HLS con resultados alentadores en términos de aceleración y eficiencia energética [1, 61]. En este contexto, la portabilidad que ofrece oneAPI y el creciente soporte de DPC++ para FPGAs presentan un entorno unificado especialmente atractivo para aplicaciones científicas e industriales, donde la diversidad de arquitecturas es la norma.

En resumen, el uso de lenguajes de alto nivel para el diseño sobre FPGAs está marcando un punto de inflexión en el paradigma de desarrollo hardware. Aunque aún existen retos importantes, como la pérdida de control fino sobre los recursos o las limitaciones de ciertas herramientas en cuanto a expresividad, el avance de los compiladores HLS y la aparición de nuevos estándares como DPC++ están haciendo posible una integración más fluida entre hardware y software, lo que resulta especialmente beneficioso para aplicaciones complejas y de alto rendimiento, como las abordadas en esta tesis.

En el contexto de esta tesis, se ha explorado el uso de DPC++ y herramientas HLS para implementar diversos algoritmos de análisis hiperespectral en plataformas FPGA. El objetivo ha sido evaluar hasta qué punto es posible aproximar el rendimiento de una implementación manual en VHDL, aprovechando al mismo tiempo las ventajas que ofrecen los lenguajes de alto nivel en términos de productividad. A través de una serie de experimentos y comparativas, se han identificado las técnicas de paralelismo más efectivas y se han derivado recomendaciones prácticas para la escritura de código eficiente en DPC++ orientado a FPGAs, contribuyendo así al desarrollo de una metodología de diseño más ágil y escalable.

Capítulo 3

Virtual Dimensionality (VD)

En este capítulo se presenta el ***Virtual Dimensionality*** (VD), un algoritmo diseñado para estimar el número de materiales puros o endmembers presentes en un conjunto de datos multidimensional. VD es ampliamente utilizado en el procesamiento de imágenes hiperespectrales, constituyendo una etapa esencial dentro del proceso de desmezclado espectral. Su importancia radica en que el número de endmembers es un parámetro clave para las etapas posteriores de la cadena, sirviendo como entrada tanto para la identificación de endmembers como para la estimación de abundancias. El algoritmo se basa en el principio de que, si un determinado endmember está presente en la imagen hiperespectral, este debe ser representativo en al menos una banda espectral, donde su detección resulta más sencilla. En la literatura, existen pocas metodologías automáticas para la estimación del número de endmembers, siendo el VD uno de los enfoques más relevantes debido a su flexibilidad. Esto se debe a que incorpora un parámetro adicional de entrada, que permite ajustar la sensibilidad del método, proporcionando un control más preciso sobre la estimación.

A lo largo del capítulo, se examinarán las distintas variantes del algoritmo VD propuestas en la literatura, centrándose en los métodos utilizados para el cálculo de los autovalores de las matrices de correlación y covarianza. Se evaluarán sus ventajas e inconvenientes con el objetivo de seleccionar la alternativa más adecuada para su implementación en hardware, priorizando la eficiencia computacional, evitando operaciones costosas y maximizando el rendimiento en entornos de cómputo acelerado. Para ello, se desarrollará una implementación paralela optimizada en FPGA, utilizando el modelo de programación Intel oneAPI y el lenguaje DPC++. Este enfoque no solo permitirá mejorar el rendimiento del algoritmo con respecto a una versión secuencial base, sino que también reducirá los tiempos de desarrollo al aprovechar un lenguaje de alto nivel, facilitando tanto la programación como la optimización del hardware.

3.1. Fundamentos del VD

Virtual Dimensionality (VD) [20] es un algoritmo ampliamente utilizado en el procesamiento de imágenes hiperespectrales para estimar el número de materiales puros o endmembers en escenas complejas. Su principal objetivo es determinar automáticamente la cantidad de endmembers presentes en una imagen sin requerir información previa. Para lograrlo, VD se basa en técnicas estadísticas y principios de teoría de la probabilidad para analizar la dimensionalidad de la imagen. Este proceso emplea como parámetro de entrada un valor de probabilidad de falsa alarma, P_{fa}, que define un umbral de detección y ajusta la sensibilidad del método.

El funcionamiento de VD se basa en el principio de que, si un endmember específico está presente en una imagen hiperespectral, su firma espectral debe ser representativa en al menos una banda espectral, facilitando así su identificación. Para ello, se calculan los autovalores de las matrices de correlación y covarianza, lo que permite realizar un análisis en cada banda espectral para determinar la presencia o ausencia de un endmember en dicha banda. La estimación de la dimensionalidad se lleva a cabo mediante un detector de Neyman-Pearson, el cual evalúa cuántas veces la prueba estadística no se cumple en cada banda, proporcionando así una estimación del número de endmembers presente en los datos. Formalmente, el algoritmo sigue los siguientes pasos:

1. **Inicialización**: Se parte de un conjunto de datos hiperespectrales $X \in \mathbb{R}^{b \times p}$, donde b representa el número de bandas espectrales y p el número de píxeles.

2. **Cálculo de las matrices de correlación (CM) y covarianza (VM)**: Se calcula CM para determinar el grado de relación lineal entre las bandas espectrales de la imagen de entrada. En el caso de VM, los valores de la diagonal representan la varianza de cada banda, mientras que los valores fuera de la diagonal indican la covarianza entre ellas.

3. **Cálculo de autovalores para las matrices de correlación ($\hat{\lambda}_i$) y covarianza (λ_i)**: Se obtienen los autovalores de ambas matrices, los cuales reflejan la cantidad de varianza explicada por cada autovector, ordenados en orden descendente.

4. **Realización del test estadístico**: Se evalúan dos hipótesis: H_0: $\hat{\lambda}_i - \lambda_i = 0$ (ausencia de endmember en la banda espectral i-ésima) y H_1: $\hat{\lambda}_i - \lambda_i > 0$ (presencia de endmember en la banda espectral i-ésima).

5. **Determinación de la dimensionalidad**: Se establece la dimensionalidad a través del método Harsanyi–Farrand–Chang (HFC) [22], basado en los conceptos previamente

descritos. Este método cuantifica cuántas veces la prueba estadística falla mediante el detector de Neyman-Pearson [80], utilizando como parámetro de entrada la probabilidad de falsa alarma, P_{fa}, la cual determina la sensibilidad del método en la evaluación de las hipótesis anteriores.

La Figura 3.1 ilustra de manera esquemática el flujo del algoritmo VD, mostrando cada una de sus fases principales. En ella se observa cómo, a partir de un conjunto inicial de datos hiperespectrales, el algoritmo aplica de forma consecutiva los siguientes pasos: cálculo de las matrices de correlación y covarianza, cálculo de autovalores para ambas matrices, realización del test estadístico y, finalmente, determinación de la dimensionalidad.

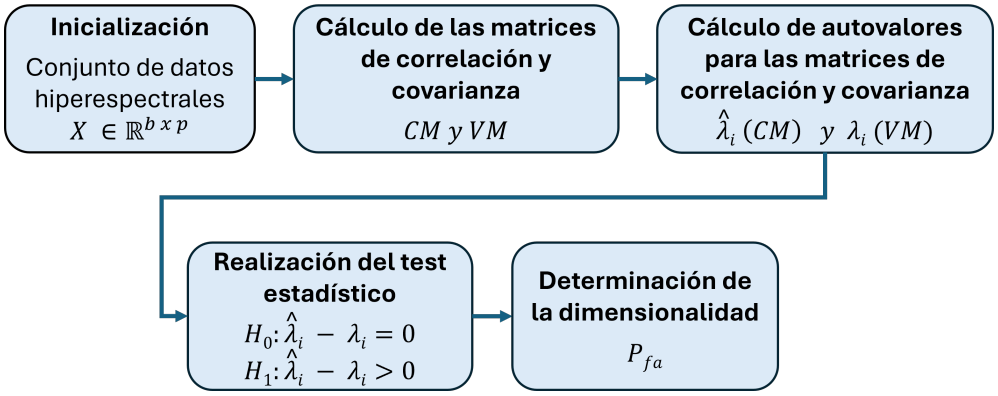


Figura 3.1 Esquema del proceso del VD.

3.2. Variantes algorítmicas del VD

Cuando los datos provienen de un sensor hiperespectral, la matriz de correlación de una secuencia de muestras $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N\}$ se calcula de manera incremental para adaptar la implementación en hardware a la naturaleza en flujo (*streaming*) de estos sensores. Matemáticamente, la matriz de correlación se define como $CM = \frac{1}{N}\sum_{i=1}^{N} \vec{x}_i \vec{x}_i^{\,\mathsf{T}}$ y expandiendo sus coordenadas se desarrolla de la siguiente manera:

$$CM = \frac{1}{N} \begin{pmatrix} | & | & \cdots & | \\ \vec{x}_1 & \vec{x}_2 & \cdots & \vec{x}_N \\ | & | & \cdots & | \end{pmatrix} \begin{pmatrix} — & \vec{x}_1 & — \\ — & \vec{x}_2 & — \\ \vdots & \vdots & \vdots \\ — & \vec{x}_N & — \end{pmatrix}$$

Por lo tanto, cada elemento de la matriz de correlación se calcula como $CM_{i,j} = \frac{1}{N}\sum_{k=1}^{N}(\vec{x}_k)_i(\vec{x}_k)_j$. Asimismo, la fila j-ésima de CM se define como:

$$CM(j,:) = \frac{1}{N}\sum_{k=1}^{N}\vec{x}_k(\vec{x}_k)_j \tag{3.1}$$

El Algoritmo 1 implementa la Ecuación 3.1 para calcular la matriz de correlación de forma incremental.

Algoritmo 1 Cálculo de la matriz de correlación

Entrada: $X = [\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N]$, con $\vec{x}_i \in \mathbb{R}^L$
variable local $\vec{d} \in \mathbb{R}^L$
for $\vec{x}_i \in X$ **do**
  $\vec{d} := \vec{x}_i$
  for $j := 0$ **to** $L-1$ **do**
    if $i = 0$ **then**
      $\vec{t} := \vec{0}$
    else
      $\vec{t} := CM_{j,:}$
    end if
    $CM_{j,:} := \vec{d} \cdot d_j + \vec{t}$
  end for
end for
for $i := 0$ **to** $L-1$ **do**
  for $j := 0$ **to** $L-1$ **do**
    $CM_{i,j} := CM_{i,j}/N$
  end for
end for
Salida: $CM \in \mathbb{R}^{L\times L}$

Si se lograse almacenar la matriz completa de píxeles en la memoria interna, sería posible utilizar algoritmos más eficientes para la multiplicación de matrices, en lugar del método ingenuo de complejidad $O(n^3)$. Un ejemplo de ello es el algoritmo de Coppersmith-Winograd, que alcanza una complejidad de $O(n^{2,3755})$ para la multiplicación de matrices [19], lo cual es asintóticamente más rápido que el enfoque tradicional. Un aspecto crucial de estos algoritmos es que requieren que la matriz completa de píxeles esté almacenada previamente en memoria, lo que impide su ejecución incremental a medida que los píxeles van llegando.

La implementación en hardware propuesta permite calcular la correlación de manera incremental, lo que posibilita ocultar la latencia asociada al cálculo de la correlación bajo la latencia del sensor durante la captura de imágenes, la cual puede ser considerablemente

alta. Así, aunque el cálculo de la matriz de correlación en la implementación en hardware propuesta pueda llevar más tiempo, este proceso queda enmascarado por el tiempo requerido para la captura de imágenes por parte del sensor. En conclusión, cuando los datos provienen de un sensor hiperespectral, la opción más eficiente es calcular la matriz de correlación de forma incremental, tal como se ha explicado.

Por tanto, la mayoría de las variantes algorítmicas del VD se enfocan en la forma de calcular los autovalores de las matrices de correlación y covarianza. Para esta etapa del algoritmo, se pueden emplear distintos métodos para el cálculo de los autovalores en una matriz simétrica, entre los cuales destacan los siguientes:

1. **Método de resolución del polinomio característico**. El proceso consiste en construir el polinomio característico de una matriz y luego resolverlo encontrando sus raíces, que corresponden a sus autovalores. Este procedimiento se basa en la siguiente ecuación:

$$det(A - \lambda I) = 0 \tag{3.2}$$

   Al calcular el determinante de la expresión anterior, se obtiene un polinomio en λ, denominado polinomio característico. En consecuencia, es necesario resolver $p(\lambda) = 0$ para determinar las raíces, que representarán los autovalores de la matriz A. Este método tiene una complejidad de $O(n^3)$ tanto para el cálculo del determinante mediante la factorización LU u otros métodos de eliminación, como para la resolución del polinomio de grado n.

2. **Método Givens**. Este método utiliza una serie de transformaciones ortogonales mediante rotaciones aplicadas a las matrices para eliminar ciertos elementos fuera de la diagonal. Su objetivo principal es hacer ceros los elementos ubicados debajo de la diagonal, con el fin de obtener una matriz triangular superior. Para cada par de elementos a_{ij} (donde $i > j$) de la matriz A, se calcula el ángulo θ necesario para anular el elemento correspondiente. El sistema de ecuaciones utilizado para determinar el ángulo es el siguiente:

$$cos(\theta) = \frac{a_{ii}}{\sqrt{a_{ii}^2 + a_{ij}^2}}, \qquad sin(\theta) = \frac{-a_{ij}}{\sqrt{a_{ii}^2 + a_{ij}^2}} \tag{3.3}$$

   Tras calcular $cos(\theta)$ y $sin(\theta)$, se construye la matriz $G(\theta)$ correspondiente:

$$G(\theta) = \begin{pmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{pmatrix} \tag{3.4}$$

Finalmente, se multiplica la matriz A por la matriz de rotación G sucesivamente para cada par de elementos a_{ij} debajo de la diagonal hasta que todos los elementos debajo de la diagonal sean cero, dando lugar a una matriz triangular superior R. Al final de este proceso, los autovalores de la matriz original se encuentran en la diagonal de la matriz triangular resultante. En cuanto a la complejidad, este método presenta una complejidad de $O(n^2)$ para cubrir todos los elementos fuera de la diagonal, en el peor de los casos. Sin embargo, la complejidad total de aplicar las rotaciones de Givens en el contexto de un algoritmo QR iterativo para calcular los autovalores de una matriz A de tamaño $n \times n$ es de $O(n^3)$.

3. **Método QR**. Este método se basa en la descomposición de una matriz A en el producto de una matriz ortogonal Q (tal que $Q^T Q = I$, siendo I la matriz identidad) y una matriz triangular superior R (es decir, $A = QR$). Una vez obtenida esta primera descomposición, se forma una nueva matriz A' multiplicando R y Q en orden inverso ($A' = RQ$). A continuación, se descompone A' en Q' y R' para obtener una nueva matriz $A'' = R'Q'$ y se repite este proceso iterativamente hasta que las matrices A^k converjan a una forma triangular superior, con los elementos fuera de la diagonal siendo cero. Finalmente, las entradas diagonales de la matriz resultante corresponderán a los autovalores de la matriz original A. Si la matriz A es diagonalizable, estas entradas diagonales serán precisamente los autovalores de A. Los autovectores pueden ser obtenidos a partir de las matrices Q generadas en cada iteración. En cuanto a la complejidad, este método presenta una complejidad de $O(n^3)$, lo que lo convierte en un método eficiente para matrices grandes y densas.

4. **Método de la potencia**. Es un algoritmo iterativo utilizado para encontrar el autovalor dominante, es decir, el autovalor de mayor valor absoluto de una matriz, así como su correspondiente autovector. Se basa en la idea de multiplicar repetidamente la matriz por un vector inicial $x^{(0)}$, que puede ser un vector aleatorio, y observar cómo evoluciona a lo largo de las iteraciones hasta que el vector se alinee con el autovector correspondiente al autovalor dominante de la matriz. En cada iteración, se multiplicará el vector $x^{(k)}$ por la matriz A, lo que producirá un nuevo vector $x^{(k+1)} = Ax^{(k)}$. Después de cada multiplicación, se normaliza el vector $x^{(k+1)}$, dividiéndolo entre su norma o módulo, con el fin de evitar que los valores crezcan demasiado y así mantener la estabilidad numérica. El autovalor dominante se puede aproximar de la siguiente manera:

$$\lambda^{(k)} = \frac{x^{(k+1)T} A x^{(k)}}{x^{(k+1)T} x^{(k)}} \tag{3.5}$$

Se repetirán los pasos anteriores hasta que el vector $x^{(k)}$ y el autovalor $\lambda^{(k)}$ converjan o cambien muy poco entre iteraciones. En definitiva, este método es bastante eficiente para encontrar el autovalor dominante de una matriz, ya que presenta una complejidad de $O(n^2)$, pero no es adecuado si se desea encontrar todos los autovalores de una matriz.

5. **Método Jacobi**. Es un algoritmo iterativo que utiliza un sistema de rotaciones ortogonales para eliminar los elementos fuera de la diagonal de una matriz. El proceso comienza con la matriz $V = I$, que almacenará los autovectores, y selecciona dos índices i y j tales que a_{ij} (un elemento fuera de la diagonal) sea el valor más grande en valor absoluto, el cual será el objetivo para hacer cero. La matriz de rotación P que se aplicará a la matriz A tiene la siguiente forma para sus elementos a_{ii} y a_{jj}:

$$P = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & cos(\theta) & \dots & sin(\theta) & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & -sin(\theta) & \dots & cos(\theta) & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & 1 \end{pmatrix} \tag{3.6}$$

Donde θ es el ángulo de rotación utilizado para hacer cero el elemento a_{ij}. Este ángulo puede calcularse utilizando los elementos a_{ii} y a_{ij} de la siguiente forma:

$$\theta = \frac{1}{2} tan^{-1}\left(\frac{2a_{ij}}{a_{ii} - a_{jj}}\right) \tag{3.7}$$

Una vez calculada la matriz de rotación P, se actualizan las matrices A y V, es decir, $A' = P^T AP$ y $V' = VP$. Este proceso se repetirá hasta que los elementos fuera de la diagonal de A sean lo suficientemente pequeños (por debajo de un umbral predefinido). En este punto, los elementos diagonales de A son los autovalores de la matriz original, y los vectores columna de V corresponden a los autovectores asociados a esos autovalores. Finalmente, la complejidad de este método es de $O(n^3)$.

De todas las alternativas comentadas anteriormente, el método Jacobi puede ser la opción más eficiente al hacer uso de computación de alto rendimiento [32], especialmente si consideramos que el tamaño de la matriz A dependerá del número de bandas espectrales de la imagen hiperespectral (en el caso de matrices pequeñas o medianas). Además, presenta

como ventaja su alta paralelización, ya que cada rotación afecta únicamente a un par de filas y un par de columnas de la matriz, así como su simplicidad.

3.3. Variante seleccionada: VD usando método Jacobi

La variante VD seleccionada para esta implementación ha sido ampliamente utilizada por la comunidad científica en los últimos años, obteniendo excelentes resultados en términos de rendimiento. Por este motivo, este algoritmo es el candidato ideal para compararlo con una versión optimizada en FPGA, la cual ha sido desarrollada utilizando el lenguaje VHDL [32]. Para facilitar su comprensión, a continuación se describen los pasos que se representan en el Algoritmo 2.

Algoritmo 2 Pseudocódigo del algoritmo VD

1: **Entradas**: $\mathbf{X} := [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_p]$, P_{fa}
2: $CR := \frac{X^T * X}{p}$;
3: $CV := \frac{(X-\bar{X})^T * (X-\bar{X})}{p}$;
4: $\lambda^{CR} := eig(CR)$; // se calculan los autovalores de CR
5: $\lambda^{CV} := eig(CV)$; // se calculan los autovalores de CV
6: $t := 0$;
7: **for** $i := 1$ to b **do**
8: $\quad \sigma_i \cong \sqrt{\frac{2}{p}(\lambda_i^{CR})^2 + \frac{2}{p}(\lambda_i^{CV})^2}$;
9: $\quad$ resolver $P_{fa} := \frac{1}{\sigma_i\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{z_i^2}{2\sigma_i^2}} dz_i$ para encontrar y;
10: $\quad$ diff := λ_i^{CR}-λ_i^{CV};
11: $\quad$ **if** diff $> y$ **then**
12: $\quad\quad t := t + 1$;
13: $\quad$ **end if**
14: **end for**
15: **Salida**: t

Las entradas de este algoritmo están compuestas por la imagen hiperespectral, denotada como $\mathbf{X}$, que consta de p píxeles y b bandas espectrales, además de un valor de probabilidad de falsa alarma P_{fa}. Este último se utiliza para establecer la sensibilidad del algoritmo en términos de cuánto error se puede tolerar en la identificación del número total de endmembers presentes en los datos de la imagen. Los autores del método recomiendan utilizar valores de P_{fa} entre 10^{-1} y 10^{-5} [22].

Para la implementación propuesta, se seguirán los pasos descritos en [32], ya que el objetivo es optimizar el método sobre FPGAs, que constituye el hardware de destino de esta Tesis Doctoral. Las principales diferencias con respecto al método HFC-VD tradicional se encuentran en el cálculo del píxel medio junto con la matriz de correlación y la matriz de covarianza a partir de los cálculos anteriores, correspondientes a los pasos 2 y 3 del Algoritmo 2, así como en la forma de calcular los autovalores de ambas matrices, que corresponde a los pasos 4 y 5.

3.4. Implementación HLS del algoritmo VD

Para lograr una implementación HLS eficiente del algoritmo VD sobre FPGAs, es fundamental realizar un perfilado del código que permita identificar las secciones con mayor consumo de tiempo. De este modo, se pueden seleccionar las partes más adecuadas para paralelizar y optimizar su aceleración.

Para un análisis más detallado del perfilado, es importante considerar que el procesamiento del algoritmo VD puede dividirse en tres fases: 1) Cálculo de las matrices de correlación y covarianza; 2) Cálculo de los autovalores para ambas matrices; y 3) Determinación de la dimensionalidad mediante el método HFC, basado en la prueba de dos hipótesis.

Después de ejecutar el perfilado del código VD utilizando tres imágenes de diferentes tamaños, se analizaron los resultados para identificar las secciones con mayor demanda computacional. Al observar los resultados, se encontró que la tendencia del **cálculo de las matrices de correlación y covarianza**, era la parte que más tiempo consumía a medida que el tamaño de la imagen aumentaba, representando entre un **36 % y un 57 %** del tiempo total de procesamiento. Además, cabe destacar que la segunda fase que más tiempo consumía era el **cálculo de los autovalores**, con un consumo de entre un **63 % y un 42 %** respectivamente, para las imágenes AVIRIS Cuprite y AVIRIS WTC. Por lo tanto, los esfuerzos de aceleración se concentrarán en estas dos fases.

Antes de comenzar con las primeras optimizaciones, será necesario disponer de una versión base del código, que presente las mínimas modificaciones posibles, para poder ejecutarlo en una FPGA.

3.4.1. Versión base (*baseline*) del VD

Para llevar a cabo las diversas optimizaciones en la implementación propuesta del algoritmo VD, es fundamental disponer de una versión base que nos permita realizar las comparaciones posteriores. Esta versión debe incluir únicamente los cambios esenciales en relación con una implementación secuencial en C++ y SYCL, la cual será compilada utilizando el compilador DPC++. Para esta versión base, será necesario incorporar los siguientes elementos (indicar que el elemento [0] es optativo pero aconsejable para disponer de un código más legible), tal y como se muestra en el Código 3.1:

[0] Aislar el código en una función donde se realice la llamada al *kernel* que se ejecutará en la placa FPGA, dejando la lectura de la imagen y la creación de los buffers de entrada/salida fuera de esta función.

[1] Definir un selector de dispositivo dirigido al uso de FPGAs, al inicio de la función.

[2] Declarar una *queue* para conectar el código del *host* con el *kernel* que se ejecutará en el dispositivo FPGA.

[3] Crear un dispositivo cola DPC++ utilizando el selector previamente definido.

[4] Utilizar *buffers* para los arrays de entrada y salida de datos. En este caso, se empleará un *buffer* de entrada para la imagen hiperespectral y un *buffer* de salida para almacenar el número de endmembers estimados por el algoritmo.

[5] Emplear *submit* para enviar un conjunto de comandos que se ejecutarán en el dispositivo objetivo.

[6] Utilizar accesores para los *buffers* creados, permitiendo que la FPGA acceda a los datos de manera eficiente.

[7] Usar *single_task* para enviar el kernel a ejecución, asegurando que el contenido de *buffer_VD* se copie a *VD* una vez la función haya terminado.

Código 3.1 Estructura de la versión base para el algoritmo VD.

```
1  void VD(const std::vector<float> &h_image, std::vector<int> &VD, int
       Samples, int Lines, int bands) {
2      #if defined(FPGA_EMULATOR)
3          ext::intel::fpga_emulator_selector device_selector;
4      #else
```

```
5           [1] ext::intel::fpga_selector device_selector;
6       #endif
7
8       try {
9           [2] queue q(device_selector, dpc_common::exception_handler,
10              property::queue::enable_profiling{});
11          [3] buffer buffer_image(h_image);
12          [3] buffer buffer_VD(VD);
13
14          [4] event e = q.submit([&](handler &h) {
15              [5] accessor acc_image(buffer_image,h,read_only);
16              [5] accessor acc_VD(buffer_VD,h,write_only, no_init);
17
18              [6] h.single_task([=]()
19              [[intel::kernel_args_restrict]] {
20                  // Código del kernel VD a ejecutar en FPGA
21              });
22          });
23      }
24      catch (sycl::exception const &e) {
25          // Detección de excepciones en el código del host
26      }
27  }
```

Dado que los accesos a memoria son secuenciales y predecibles, el uso de técnicas como la partición o la distribución en bancos de memoria podría no aportar mejoras significativas e incluso reducir el rendimiento. Al no producirse conflictos de acceso y tratarse de un acceso lineal, el diseño ya se beneficia del comportamiento eficiente de la caché por defecto. El uso de directivas como [intel::partition] e [intel::numbanks(N)], están más orientadas a situaciones donde existen múltiples accesos paralelos o potenciales conflictos que podrían degradar el rendimiento. Si en el futuro se modifica el patrón de acceso o se busca un mayor paralelismo en el *kernel*, sí sería recomendable considerarlas.

Los accesos a memoria se gestionan mediante objetos *buffer* y *accessor*, lo que permite al entorno SYCL construir un grafo de ejecución basado en dependencias de datos. Esto asegura un modelo de sincronismo automático entre *kernels*, donde la ejecución se programa en función de la disponibilidad de los datos requeridos. Posteriormente, se han utilizado colas *in-order* para operaciones secuenciales, y mecanismos explícitos como *event.wait()* para forzar sincronización en situaciones críticas. Este enfoque se alinea con el modelo de ejecución de SYCL, evitando condiciones de carrera sin necesidad de sincronización manual.

Con los cambios previamente mencionados, se dispone de un código base a partir del cual será posible generar un informe para proceder con su optimización. Esta versión no incorpora ninguna técnica de optimización, ya que más adelante se estudiará la viabilidad de implementar posibles mejoras sobre esta versión base. Este informe se obtendrá tras realizar una compilación rápida y proporcionará sugerencias sobre cómo modificar los *kernels* para mejorar el rendimiento. Además, incluirá información esencial para el desarrollo, como vistas de las estructuras (mapeo del código sobre los recursos de la FPGA), diseño de memoria, rendimiento, cuellos de botella (áreas en las que el rendimiento está limitado) y estimaciones de los recursos consumidos por el diseño.

El informe generado proporciona información clave para el análisis del diseño, siendo especialmente útil la vista de planificación, que detalla la planificación estática ciclo a ciclo y la latencia de los grupos de instrucciones, permitiendo examinar el comportamiento del sistema en FPGA y optimizar su rendimiento. En la Figura 3.2, se muestra el diagrama de planificación correspondiente a la versión *baseline*, dividida en cuatro fases: cálculo de las matrices de correlación y covarianza, cálculo de los autovalores de la matriz de correlación, cálculo de los autovalores de la matriz de covarianza y cálculo de las condiciones. Dentro de la fase del cálculo de las matrices de correlación y covarianza, se han identificado las operaciones clave: el cálculo de la matriz de correlación y del píxel medio en paralelo, y el cálculo de la matriz de covarianza. En el siguiente apartado, se analizarán en detalle estos procesos y su impacto en el rendimiento y eficiencia de la implementación.

Antes de abordar las optimizaciones detectadas tras analizar el informe, es fundamental analizar el rendimiento actual para cuantificar con precisión las mejoras introducidas. Para ello, las mediciones del tiempo de ejecución se han realizado mediante funciones de una clase de perfilado de SYCL, tal y como se muestra en el Código 3.2:

Código 3.2 Funciones para realizar el perfilado en el algoritmo VD.

```
1 get_profiling_info<info::event_profiling::command_start>
2 get_profiling_info<info::event_profiling::command_end>
```

En este perfilado, no se incluye ninguna transferencia de datos entre el *host* y el dispositivo *"offload"*, ya que su objetivo principal es medir el tiempo de ejecución de un *kernel*. Es importante señalar que los tiempos así obtenidos se corresponden con ejecuciones reales sobre el dispositivo hardware (en este caso, una FPGA Intel Stratix 10 SX 2800), y se miden una vez completado el flujo completo de síntesis e implementación. Por tanto, no se trata de estimaciones previas ni simulaciones, sino de datos empíricos en entorno real. No obstante, estos tiempos reflejan exclusivamente la ejecución del *kernel* en placa, y no incluyen fases

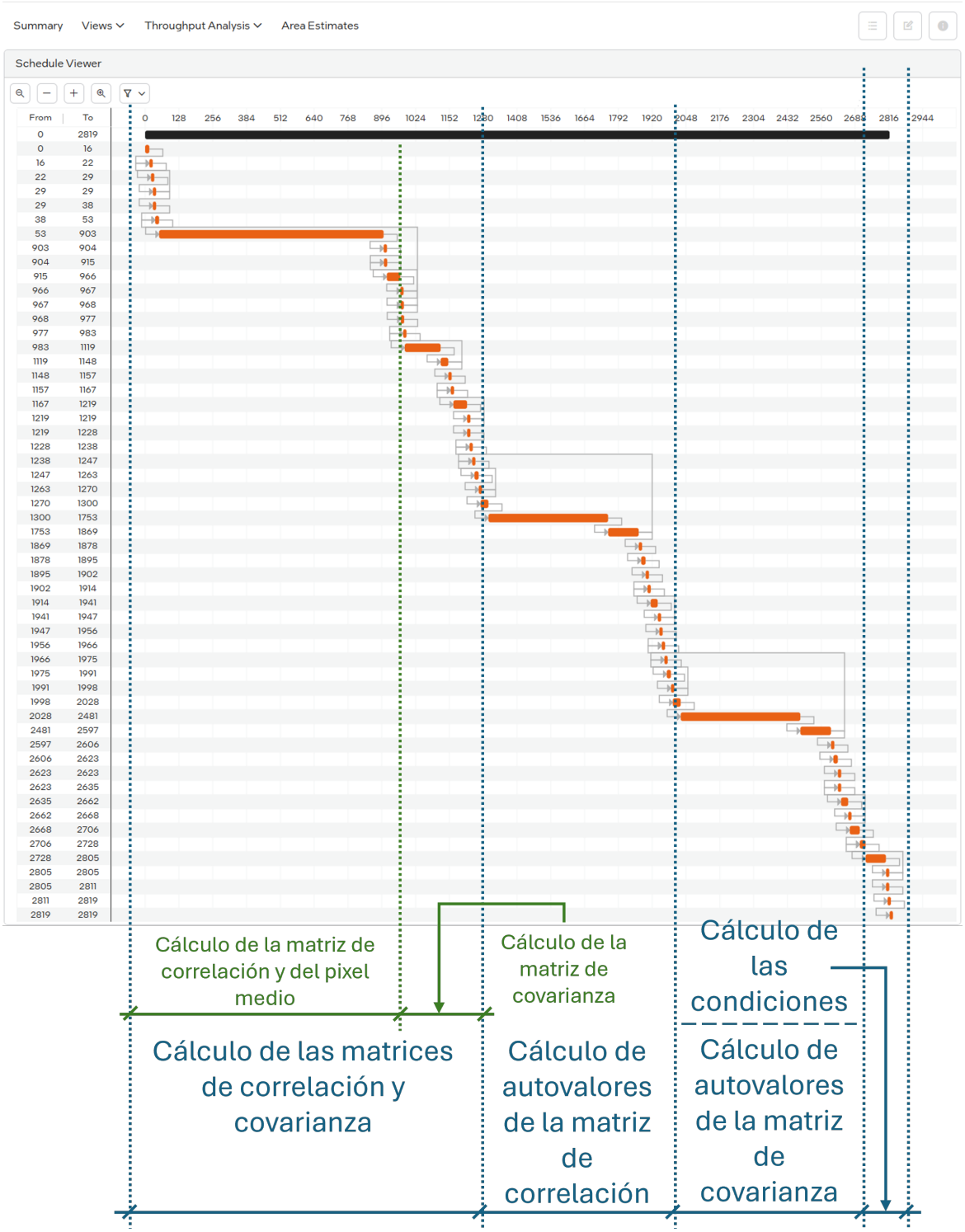


Figura 3.2 Diagrama temporal ciclo a ciclo de la versión VD *baseline*.

previas como la síntesis, *place & route* o generación del bitstream. Para la versión base, se presentan los tiempos obtenidos en hardware, así como los recursos utilizados por esta versión sobre las escenas consideradas, los cuales se detallan en la Tabla 3.1.

Implementaciones	Tiempos (ms)			F_{max}	Recursos				
	AVIRIS Cuprite	AVIRIS WTC	Sintética	(MHz)	ALUTs	FFs	RAMs	MLABs	DSPs
Baseline	60545.3	133326	207583	432	175426 (9%)	220426 (6%)	837 (7%)	1107 (1%)	308 (5%)

Tabla 3.1 Tiempo de procesamiento y recursos utilizados por el algoritmo VD usando la FPGA Intel Stratix 10 SX 2800 para la versión *baseline*.

3.4.2. Versión optimizada del VD

El análisis del diagrama temporal confirma que, también en FPGA, las dos etapas con mayor tiempo de ejecución son el cálculo de las matrices de correlación y covarianza, y el cálculo de los autovalores, mientras que el cálculo de las condiciones presenta un impacto computacional significativamente menor.

Desde el inicio, el algoritmo seleccionado para el cálculo de la matriz de correlación fue diseñado para minimizar el número de accesos a memoria externa, recorriendo la imagen una única vez para realizar el cálculo. Además, el píxel medio se computa de manera simultánea, evitando accesos adicionales a memoria. La matriz de covarianza, por su parte, se deriva directamente de la matriz de correlación y del píxel medio, lo que también elimina la necesidad de acceder a memoria externa. Por lo tanto, esta fase del algoritmo ya se encuentra altamente optimizada, por lo que los esfuerzos de mejora se enfocaron en la segunda fase: el cálculo de los autovalores.

Las optimizaciones implementadas se centraron en la paralelización del cálculo de los autovalores para ambas matrices, dado que estos cálculos son independientes. Además, se trabajó en la optimización del algoritmo de Jacobi (utilizado para el cálculo de autovalores), explorando variantes como la búsqueda exhaustiva, la paralelización de la reducción de elementos independientes fuera de la diagonal y la conversión a enteros. Sin embargo, ninguna de estas técnicas logró una mejora con respecto a la versión original o, en los casos en que sí lo hacía, la pérdida de precisión afectaba negativamente al resultado.

Para permitir la ejecución en paralelo de ambas tareas, se dividió la *single task* original en cuatro *single tasks* con la siguiente estructura:

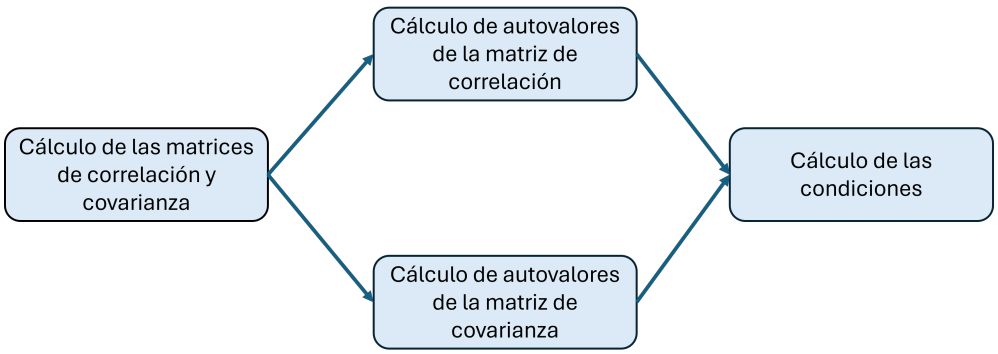


Figura 3.3 Diagrama de flujo de la ejecución del algoritmo VD optimizado.

1. Cálculo de las matrices de correlación y covarianza.
2. Cálculo de los autovalores de la matriz de correlación (en paralelo con la fase 3).
3. Cálculo de los autovalores de la matriz de covarianza (en paralelo con la fase 2).
4. Cálculo de las condiciones (una vez finalizadas las fases 2 y 3).

La Figura 3.3 ilustra esta nueva distribución en *single tasks*. A continuación, el Código 3.3 muestra la implementación de las cuatro *single tasks*, incluyendo las dependencias, sincronizaciones y mediciones de tiempo de cada *single task* y del proceso completo.

Código 3.3 Estructura de la versión parelala usando cuatro single tasks para el algoritmo VD.

```
1  void VD(const std::vector<float> &h_image, std::vector<int> &VD, int
       Samples, int Lines, int bands) {
2      #if defined(FPGA_EMULATOR)
3          ext::intel::fpga_emulator_selector device_selector;
4      #else
5          ext::intel::fpga_selector device_selector;
6      #endif
7
8      try {
9          queue q01(device_selector, dpc_common::exception_handler,
               property::queue::enable_profiling{});
10         queue q02(device_selector, dpc_common::exception_handler,
               property::queue::enable_profiling{});
11         queue q03(device_selector, dpc_common::exception_handler,
               property::queue::enable_profiling{});
12         queue q04(device_selector, dpc_common::exception_handler,
               property::queue::enable_profiling{});
```

```
13
14          buffer buffer_image (h_image);
15          buffer buffer_VD (VD);
16
17          std::vector<double> CM_singleTasks(bands * bands);
18          buffer buffer_CM(CM_singleTasks);
19
20          std::vector<double> VM_singleTasks(bands * bands);
21          buffer buffer_VM(VM_singleTasks);
22
23          std::vector<double> eigenvaluesCM_singleTaks(bands);
24          buffer buffer_eigenvaluesCM(eigenvaluesCM_singleTaks);
25
26          std::vector<double> eigenvaluesVM_singleTaks(bands);
27          buffer buffer_eigenvaluesVM(eigenvaluesVM_singleTaks);
28          auto start_todo = std::chrono::high_resolution_clock::now();
29
30          event e01 = q01.submit ([&] (handler &h) {
31              accessor acc_image (buffer_image, h, read_only);
32
33              accessor acc_CM (buffer_CM, h, write_only, no_init);
34              accessor acc_VM (buffer_VM, h, write_only, no_init);
35
36              h.single_task<class single_task_CM_and_VM> ([=]()[[intel
                    ::kernel_args_restrict]] {
37                  /* Código para el cálculo de las matrices de
                        correlación y covarianza */
38              });
39          });
40
41          q01.wait();
42
43          event e02 = q02.submit ([&] (handler &h) {
44              accessor acc_eigenvaluesCM (buffer_eigenvaluesCM, h,
                    write_only, no_init);
45              accessor acc_CM (buffer_CM, h, read_only);
46
47              h.single_task<class single_task_eigenvaluesCM> ([=]()[[
                    intel::kernel_args_restrict]] {
48                  /* Código para el cálculo de los autovalores de la
                        matriz de correlación */
49              });
50          });
51
```

```
52          event e03 = q03.submit ([&] (handler &h) {
53              accessor acc_eigenvaluesVM (buffer_eigenvaluesVM, h,
                    write_only, no_init);
54              accessor acc_VM (buffer_VM, h, read_only);
55
56              h.single_task<class single_task_eigenvaluesVM> ([=]()[[
                    intel::kernel_args_restrict]] {
57                  /* Código para el cálculo de los autovalores de la
                        matriz de covarianza */
58              });
59
60          });
61
62          q02.wait();
63          q03.wait();
64
65          event e04 = q04.submit ([&] (handler &h) {
66              accessor acc_VD (buffer_VD, h, write_only, no_init);
67              accessor acc_eigenvaluesCM (buffer_eigenvaluesCM, h,
                    read_only);
68              accessor acc_eigenvaluesVM (buffer_eigenvaluesVM, h,
                    read_only);
69
70              h.single_task<class single_task_dimensionality> ([=]()[[
                    intel::kernel_args_restrict]] {
71                  /* Código para el cálculo de las condiciones */
72              });
73          });
74
75          double start = e01.get_profiling_info < info::event_profiling
                ::command_start > ();
76          double end = e01.get_profiling_info < info::event_profiling::
                command_end > ();
77          //convert from nanoseconds to ms
78          double kernel_time = (double)(end - start) * 1e-6;
79          std::cout << "CM␣and␣VM␣matrices␣time␣:␣" << kernel_time << "
                ␣ms␣\n";
80
81          start = e02.get_profiling_info < info::event_profiling::
                command_start > ();
82          end = e02.get_profiling_info < info::event_profiling::
                command_end > ();
83          //convert from nanoseconds to ms
84          kernel_time = (double)(end - start) * 1e-6;
```

```
85          std::cout << "Jacobi␣and␣bublle␣sort␣CM␣time␣:␣" <<
                kernel_time << "␣ms␣\n";
86
87          start = e03.get_profiling_info < info::event_profiling::
                command_start > ();
88          end = e03.get_profiling_info < info::event_profiling::
                command_end > ();
89          //convert from nanoseconds to ms
90          kernel_time = (double)(end - start) * 1e-6;
91          std::cout << "Jacobi␣and␣bublle␣sort␣VM␣time␣:␣" <<
                kernel_time << "␣ms␣\n";
92
93          start = e04.get_profiling_info < info::event_profiling::
                command_start > ();
94          end = e04.get_profiling_info < info::event_profiling::
                command_end > ();
95          //convert from nanoseconds to ms
96          kernel_time = (double)(end - start) * 1e-6;
97          std::cout << "Dimensionality␣calculation␣time␣:␣" <<
                kernel_time << "␣ms␣\n";
98
99          auto end_todo = std::chrono::high_resolution_clock::now();
100         std::chrono::duration<double> elapsed = end_todo - start_todo
                ;
101         std::cout << "Total␣execution␣time:␣" << elapsed.count() << "
                ␣segundos" << std::endl;
102     }
103     catch (sycl::exception const &e) {
104         // Detección de excepciones en el código del host
105     }
106 }
```

Las Figuras 3.4, 3.5 y 3.6 presentan el diagrama temporal ciclo a ciclo de cada una de las fases: cálculo de las matrices de correlación y covarianza, cálculo de los autovalores (idéntico para ambas matrices) y cálculo de las condiciones.

Finalmente, la Tabla 3.2 presenta los tiempos de ejecución y la ocupación de recursos de la versión optimizada, mientras que la Tabla 3.3 detalla los tiempos de procesamiento por fases, evidenciando que el cálculo de autovalores para ambas matrices se realiza en paralelo.

Aunque los resultados puedan no parecer muy optimistas a primera vista, es importante tener en cuenta que el método elegido para el cálculo de la matriz de correlación ha sido diseñado específicamente para procesar las muestras a medida que son capturadas por el

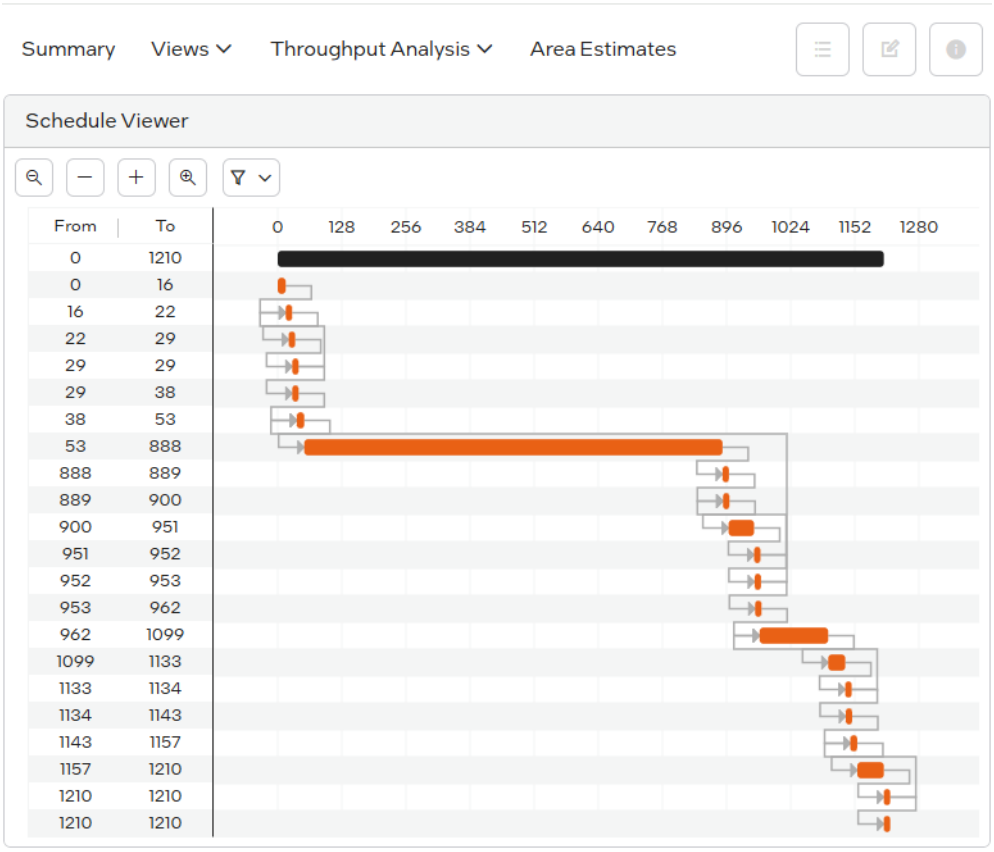


Figura 3.4 Diagrama temporal ciclo a ciclo del cálculo de la matriz de correlación y covarianzas.

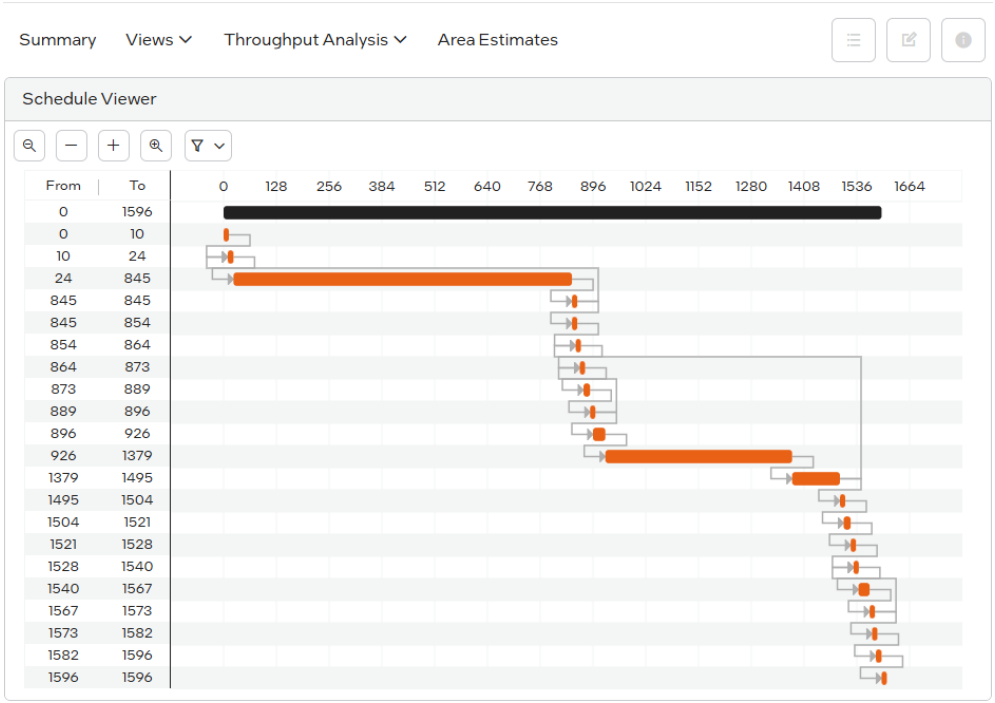


Figura 3.5 Diagrama temporal ciclo a ciclo del cálculo de autovalores y su ordenación.

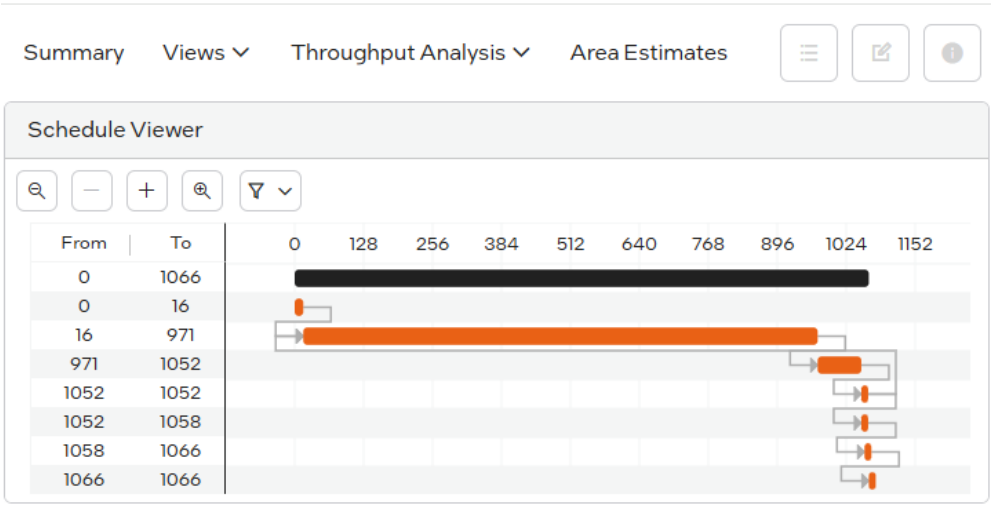


Figura 3.6 Diagrama temporal ciclo a ciclo del cálculo de condiciones para la estimación del número de endmembers presentes en la imagen.

Implementaciones	Tiempos (ms)			F_{max}	Recursos				
	AVIRIS Cuprite	AVIRIS WTC	Sintética	(MHz)	ALUTs	FFs	RAMs	MLABs	DSPs
VD_OPT	42811.3	110127	171157	432	201510 (11 %)	265400 (7 %)	1287 (11 %)	1252 (1 %)	336 (6 %)

Tabla 3.2 Tiempo de procesamiento y recursos utilizados por el algoritmo VD usando la FPGA Intel Stratix 10 SX 2800 para la versión optimizada.

Imagen	Cálculo de CM y VM (ms)	Autovalores CM (ms)	Autovalores VM (ms)	Condiciones (ms)	Total (ms)
AVIRIS Cuprite	24499.5	18285.4	18140.5	0.01536	42811.3
AVIRIS WTC	85941.4	24130.9	24089.4	0.014848	110127
Sintética	133272	37807.4	37805.2	0.01664	171157

Tabla 3.3 Tiempos de procesamiento detallados por fases para el algoritmo VD usando la FPGA Intel Stratix 10 SX 2800 para la versión optimizada.

sensor. De esta manera, la latencia asociada al acceso a memoria en esta fase, que constituye la mayor parte del tiempo en cada iteración, desaparecería. Considerando que el sensor AVIRIS necesita 8.3 ms para capturar 512 muestras con 224 bandas espectrales y que el algoritmo opera a una frecuencia de 432 MHz, se dispone de más de 30 ciclos para procesar cada muestra. Este tiempo es suficiente para realizar los cálculos necesarios para la matriz de correlación y el cálculo del píxel promedio. Por lo tanto, el tiempo dedicado al cálculo de la matriz de correlación se superpondría con el tiempo de captura de la imagen, es decir, prácticamente desaparecería. Es cierto que, después de calcular la matriz de correlación, es necesario calcular la matriz de covarianza, pero este proceso ya no requiere acceso a memoria. En resumen, el tiempo de cálculo de las matrices de correlación y covarianza (CM y VM) se reduciría considerablemente, lo que disminuiría los tiempos totales de procesamiento prácticamente al tiempo de cálculo de los autovalores consiguiendo una reducción del tiempo de procesamiento del 57 % para la escena AVIRIS Cuprite y del 78 % para las otras dos escenas.

3.5. Resultados experimentales

En este apartado se presentan los resultados obtenidos en la evaluación experimental de las diversas optimizaciones aplicadas al algoritmo VD. En primer lugar, se describen las

imágenes hiperespectrales utilizadas en el estudio: la escena Cuprite, que permite la detección de diferentes minerales presentes en la región; la escena World Trade Center, utilizada para identificar píxeles anómalos asociados a los incendios provocados tras los atentados de 2001 en Nueva York; y, por último, una imagen sintética de mayor tamaño que las anteriores, diseñada para evaluar la escalabilidad del enfoque propuesto. A continuación, se presentan las características de la FPGA utilizada en los experimentos, junto con los resultados de precisión, que permiten validar la estimación del número de endmembers obtenidos por la implementación del algoritmo. Posteriormente, se analizan los resultados en términos de rendimiento computacional, evaluando el tiempo de procesamiento, el uso de recursos, la escalabilidad con respecto al tamaño de la imagen y otros factores relevantes. Finalmente, se lleva a cabo una comparación con una implementación alternativa del VD en hardware, desarrollada utilizando el lenguaje VHDL.

3.5.1. Imágenes hiperespectrales utilizadas

Se emplean las imágenes hiperespectrales que fueron descritas en detalle en la Sección 2.1.4. Estas imágenes corresponden a:

- **Escena AVIRIS Cuprite**. Capturada sobre la región minera de Cuprite, Nevada, expresada en unidades de reflectancia.
- **Escena AVIRIS World Trade Center**. Registrada en Nueva York, cinco días después del atentado terrorista, también en unidades de reflectancia, con el objetivo de identificar focos de incendio en la zona afectada.
- **Escena sintética**. Diseñada para evaluar el rendimiento del algoritmo en imágenes de mayor tamaño, permitiendo analizar su escalabilidad y eficiencia computacional.

Estas imágenes proporcionan un conjunto de pruebas representativo para validar el desempeño del algoritmo en distintos escenarios y condiciones de análisis.

3.5.2. FPGA seleccionada

Para evaluar el rendimiento de las diferentes optimizaciones del algoritmo VD, se ha utilizado una de las FPGAs disponibles en el Intel Developer Cloud, concretamente la FPGA Intel Stratix 10 SX 2800, que fue descrita en detalle en la Sección 2.2.2. Esta placa ha sido fabricada con tecnología de 14 nm y cuenta con un total de 933.120 módulos lógicos

adaptables (ALMs), 1.866.240 tablas de búsqueda adaptables (ALUTs) y 3.732.480 flip-flops (FFs). Además, dispone de una serie de recursos heterogéneos, incluyendo 5760 DSPs y 11721 bloques de memoria RAM, así como otros componentes adicionales que no han sido utilizados.

3.5.3. Evaluación de la precisión en VD

En este apartado se evaluarán los resultados de estimación del número de endmembers obtenidos por el algoritmo VD sobre las imágenes reales consideradas en este trabajo. Con este fin, la Tabla 3.4 presenta las distintas estimaciones en función del valor de falsa alarma, P_{fa}, utilizado como parámetro de entrada del algoritmo.

P_{fa}	AVIRIS Cuprite (t)	AVIRIS WTC (t)
10^{-1}	37	61
10^{-2}	28	45
10^{-3}	25	33
10^{-4}	20	31
10^{-5}	19	30

Tabla 3.4 Estimaciones del número de endmembers obtenidos por el algoritmo VD realizando variaciones del valor P_{fa} para cada una de las escenas hiperespectrales consideradas.

A través del valor más bajo de falsa alarma, $P_{fa} = 10^{-5}$, propuesto por los autores del algoritmo original en [22], se obtienen resultados que se aproximan a los obtenidos por otros algoritmos de referencia ampliamente aceptados en la comunidad científica. Un ejemplo de ello es el algoritmo *Hyperspectral Signal Identification with Minimum Error* (HySIME) [14], que reporta valores de $t = 16$ para la escena AVIRIS Cuprite y $t = 23$ para la escena AVIRIS WTC. Este hecho valida el correcto funcionamiento del algoritmo VD. De esta manera, para la segunda etapa del desmezclado espectral, en la que se utiliza el algoritmo ATDCA-GS para la estimación de los endmembers, se emplearán como valores de entrada $t = 19$ para la escena AVIRIS Cuprite y $t = 30$ para la escena AVIRIS WTC.

3.5.4. Evaluación del rendimiento computacional en VD

En este apartado se presenta una evaluación experimental del rendimiento computacional de la implementación del algoritmo VD, considerando las diversas optimizaciones realizadas sobre la FPGA descritas anteriormente. Para la implementación optimizada, se ha calculado

la media de 10 ejecuciones, obteniéndose una desviación estándar despreciable, ya que los resultados de las distintas ejecuciones son prácticamente idénticos.

Con este fin, la Tabla 3.5 muestra los tiempos y recursos utilizados, los cuales han sido obtenidos en la placa para la implementación optimizada *VD_OPT*, partiendo de la versión *baseline* para cada una de las escenas consideradas en este trabajo.

Implementaciones	**Tiempos (ms)**			**F_{max}**	**Recursos**				
	AVIRIS Cuprite	**AVIRIS WTC**	**Sintética**	**(MHz)**	**ALUTs**	**FFs**	**RAMs**	**MLABs**	**DSPs**
Baseline	60545.3	133326	207583	432	175426 (9%)	220426 (6%)	837 (7%)	1107 (1%)	308 (5%)
VD_OPT	42811.3 (1.4×)	110127 (1.2×)	171157 (1.2×)	432	201510 (11%)	265400 (7%)	1287 (11%)	1252 (1%)	336 (6%)

Tabla 3.5 Tiempos de procesamiento y recursos utilizados por el algoritmo VD usando la FPGA Intel Stratix 10 SX 2800 para las versiones *baseline* y optimizada, generalizadas para un tamaño máximo de imagen.

Comparando estos resultados con otra implementación del algoritmo VD en hardware utilizando el lenguaje VHDL [32], los tiempos de ejecución para una FPGA Virtex-7 XC7VX690T fueron de 1.64 s y 4.26 s para las escenas de AVIRIS Cuprite y AVIRIS WTC, respectivamente. Esto muestra que la implementación optimizada realizada es aproximadamente 26× más lenta. Sin embargo, si consideramos que el algoritmo presentado en [32] requiere disponer de la imagen completa para calcular las matrices de correlación y covarianza, mientras que en la versión propuesta este cálculo se realiza de manera simultánea con la captura de datos del sensor, la implementación optimizada resultaría ser 11× veces más lenta para la escena AVIRIS Cuprite y 5.6× veces más lenta para la escena AVIRIS WTC.

No obstante, es importante señalar que la comparación entre ambas versiones debe matizarse debido a las diferencias tecnológicas entre las plataformas empleadas. En particular, la implementación en HLS se ha ejecutado sobre un Intel Stratix 10 SX 2800 a 432 MHz, mientras que la versión en VHDL se implementó sobre una Virtex-7 XC7VX690T a 75.995 MHz. Para reflejar de forma más justa la eficiencia relativa de cada diseño, se ha calculado el número estimado de ciclos de reloj necesarios en cada caso. La versión en HLS requiere aproximadamente 18,504 millones de ciclos para procesar la escena AVIRIS Cuprite y 57,676 millones para ARIVIS WTC, frente a los 124.63 y 323.74 millones de ciclos requeridos por la versión en VHDL, respectivamente. Estas cifras evidencian que, pese a la mayor frecuencia del dispositivo moderno, la versión en VHDL presenta una eficiencia de diseño considerablemente superior, fruto de una optimización a bajo nivel.

A pesar de ello, cabe destacar que el desarrollo de una implementación en HLS requiere significativamente menos tiempo que el necesario para crear una versión optimizada en HDL para FPGAs. Esto convierte al enfoque HLS en una alternativa muy valiosa en entornos donde el tiempo de desarrollo, la portabilidad o la flexibilidad son factores críticos, incluso si se asume una penalización en la eficiencia computacional.

Capítulo 4

Automatic Target Detection and Classification Algorithm (ATDCA)

En este capítulo se presenta el ***Automatic Target Detection and Classification Algorithm*** (ATDCA), un algoritmo diseñado para detectar y clasificar automáticamente objetivos, como materiales puros, en datos multidimensionales. ATDCA es ampliamente utilizado en aplicaciones de procesamiento de imágenes hiperespectrales, donde la identificación precisa de los endmembers (materiales puros) es crucial para tareas como el desmezclado espectral. Este algoritmo se basa en propiedades matemáticas como la proyección ortogonal y métodos iterativos, los cuales permiten extraer las características más relevantes de los datos. Su efectividad radica en su capacidad para operar en entornos donde los píxeles representan combinaciones de múltiples materiales, facilitando así la identificación de las firmas espectrales predominantes en la escena analizada.

A lo largo del capítulo, se analizarán las distintas variantes del ATDCA propuestas en la literatura, evaluando sus ventajas e inconvenientes con el fin de seleccionar la alternativa más adecuada para su implementación en hardware. Se priorizará la eficiencia computacional, evitando operaciones costosas y maximizando el rendimiento en entornos de cómputo acelerado. Para ello, se desarrollarán diversas implementaciones paralelas optimizadas en FPGAs utilizando el modelo de programación Intel oneAPI y el lenguaje de programación DPC++. Este enfoque permitirá no solo mejorar el rendimiento del algoritmo con respecto a una versión secuencial base, sino también reducir los tiempos de desarrollo gracias al uso de un lenguaje de alto nivel, facilitando la programación y optimización del hardware.

4.1. Fundamentos del ATDCA

El ***Automatic Target Detection and Classification Algorithm*** (ATDCA) es un algoritmo ampliamente utilizado en el procesamiento de imágenes hiperespectrales para la detección y clasificación de objetivos en escenas complejas. Su objetivo principal es identificar automáticamente los **endmembers** (materiales puros) presentes en una imagen sin necesidad de información previa. Para ello, ATDCA emplea técnicas basadas en **proyección ortogonal** y análisis estadístico, lo que le permite resaltar firmas espectrales distintivas dentro del conjunto de datos.

El funcionamiento de ATDCA se basa en la premisa de que los materiales puros dentro de una imagen hiperespectral pueden ser detectados mediante la proyección de los datos sobre un subespacio adecuado. El algoritmo recorre la imagen identificando píxeles cuya firma espectral maximiza una medida de separación respecto a los píxeles previamente seleccionados. Esta selección iterativa permite construir un conjunto de endmembers representativos de la escena. Formalmente, el algoritmo sigue los siguientes pasos:

1. **Inicialización**: Se parte de un conjunto de datos hiperespectrales $X \in \mathbb{R}^{b \times p}$, donde b es el número de bandas espectrales y p el número de píxeles.
2. **Proyección ortogonal**: Se calcula la proyección de cada píxel sobre el complemento ortogonal del espacio generado por los endmembers ya seleccionados.
3. **Criterio de selección**: Se elige el píxel que maximiza una métrica de detección, como la energía de la proyección o una medida de distancia.
4. **Actualización**: Se añade el nuevo endmember al conjunto de firmas espectrales detectadas y se repite el proceso hasta alcanzar un criterio de parada, como un número fijo de endmembers o un umbral de variabilidad.

La Figura 4.1 ilustra de manera esquemática el flujo del algoritmo ATDCA, mostrando cada una de sus fases principales. En ella se observa cómo, a partir de un conjunto inicial de datos hiperespectrales, el algoritmo aplica sucesivamente la proyección ortogonal, la selección del píxel y la actualización del conjunto de endmembers, repitiendo este proceso iterativo hasta alcanzar el criterio de parada establecido.

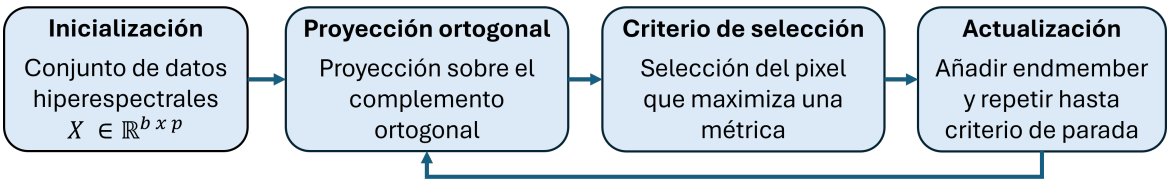


Figura 4.1 Esquema del proceso del ATDCA.

4.2. Descripción del algoritmo y sus variantes

Como se ha mencionado, el algoritmo ATDCA [85] fue diseñado para identificar automáticamente píxeles representativos que permitan construir una matriz de firmas espectrales (endmembers), utilizando un enfoque basado en proyecciones ortogonales (OSP, *Orthogonal Subspace Projection*) [44]. En este caso, el número total de endmembers a extraer depende de la estimación obtenida en la etapa previa de la cadena de desmezclado, a partir de una matriz de entrada $X \in \mathbb{R}^{b\times p}$ que representa la imagen hiperespectral.

Para determinar el primer endmember, denotado como $\mathbf{u}_0$, se calcula el producto escalar de cada píxel consigo mismo y se selecciona aquel que maximiza este valor, es decir, el píxel de mayor longitud en el espacio X. Una vez obtenido el primer endmember, el algoritmo procede a determinar los siguientes aplicando proyecciones ortogonales sobre el subespacio generado por los endmembers previamente seleccionados, de acuerdo con la siguiente expresión:

$$P_{\mathbf{U}}^{\perp} = \mathbf{I} - \mathbf{U}(\mathbf{U}^T\mathbf{U})^{-1}\mathbf{U}^T \tag{4.1}$$

Este proceso se aplicará a todos los píxeles de la imagen hiperespectral, comenzando con $\mathbf{U} = [\mathbf{u}_0]$. A continuación, se determinará el segundo endmember, denotado como $\mathbf{u}_1$, seleccionando el píxel cuya proyección en el complemento ortogonal del subespacio generado por $\mathbf{u}_0$, es decir, $< \mathbf{u}_0 >^{\perp}$, sea máxima. Para encontrar un tercer endmember, $\mathbf{u}_2$, se aplicará un nuevo proyector ortogonal, $P_{\mathbf{U}}^{\perp}$, sobre el subespacio generado por $\mathbf{U} = [\mathbf{u}_0, \mathbf{u}_1]$. La selección se realizará eligiendo el píxel cuya proyección ortogonal en $< \mathbf{u}_0, \mathbf{u}_1 >^{\perp}$ sea la más elevada con respecto a la imagen original. Este procedimiento se repetirá de forma iterativa hasta obtener un conjunto de t endmembers, $\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{t-1}\}$, donde t es un parámetro de entrada del algoritmo, determinado en la etapa previa de la cadena de desmezclado.

En contraste con el enfoque OSP, que emplea proyecciones ortogonales basadas en operaciones matriciales, el método **Gram-Schmidt (GS)** [12] ofrece una alternativa más eficiente para la ortonormalización de los endmembers seleccionados. Este método evita la necesidad

de invertir matrices, lo que reduce significativamente la complejidad computacional y lo hace especialmente adecuado para implementaciones en GPU y FPGA, donde la paralelización es fundamental.

El proceso de Gram-Schmidt comienza con la selección del primer endmember. A partir de este, cada nuevo candidato a endmember se ajusta eliminando la contribución de los endmembers previamente determinados, garantizando que el conjunto resultante sea ortonormal. Esta ortogonalización se logra mediante la siguiente expresión:

$$\mathbf{v}_i = \mathbf{u}_i - \sum_{j=0}^{i-1} \frac{\mathbf{v}_j^T \mathbf{u}_i}{\mathbf{v}_j^T \mathbf{v}_j} \mathbf{v}_j, \tag{4.2}$$

donde:

- $\mathbf{u}_i$ es el candidato a endmember antes de la ortogonalización.
- $\mathbf{v}_j$ representa los endmembers ortonormalizados previamente seleccionados.
- $\mathbf{v}_i$ es el endmember resultante tras la eliminación de la contribución de los endmembers anteriores.

Este procedimiento se repite iterativamente hasta obtener el número total de endmembers requerido. Cada vez que se incorpora un nuevo endmember al conjunto, se ajusta para garantizar que sea ortogonal a los ya seleccionados, construyendo progresivamente una base ortonormal.

En comparación con el enfoque OSP, el método Gram-Schmidt presenta diversas ventajas que lo hacen más eficiente y adecuado para implementaciones en hardware acelerado:

1. **Elimina la inversión de matrices**, reduciendo el coste computacional en comparación con OSP.
2. **Mejora la eficiencia en hardware acelerado**, ya que las operaciones pueden paralelizarse más fácilmente.
3. **Evita acumulación de errores numéricos** en comparación con enfoques puramente matriciales, lo que lo hace más estable en ciertos contextos.
4. **Mayor adaptabilidad a arquitecturas de cómputo paralelo**, como GPU y FPGA, donde la ortonormalización puede distribuirse entre múltiples núcleos de procesamiento.

Gracias a estas ventajas, el método Gram-Schmidt se considera una alternativa optimizada al OSP, especialmente en escenarios donde la eficiencia computacional es crítica.

Además del enfoque tradicional basado en proyección ortogonal, existen diversas alternativas para la implementación del ATDCA. En particular, el operador de proyección $P_{\mathbf{U}}^{\perp}$ puede ser reemplazado por diferentes métricas de distancia que permiten evaluar la similitud espectral entre píxeles de la imagen hiperespectral. Algunas de las principales opciones consideradas en la literatura [81] incluyen:

- **Norma 1 (1-Norm)**. Calcula la distancia entre dos píxeles $\mathbf{x}$ e $\mathbf{y}$ como la suma de las diferencias absolutas de sus componentes espectrales, definida por $\|\mathbf{x}-\mathbf{y}\| = \sum_{i=1}^{b} |x_i - y_i|$

- **Norma Euclidiana (2-Norm)**. Determina la distancia entre dos píxeles $\mathbf{x}$ e $\mathbf{y}$ mediante la raíz cuadrada de la suma de los cuadrados de las diferencias entre sus componentes, definida por $\|\mathbf{x}-\mathbf{y}\|_2 = \sum_{i=1}^{b} |(x_i - y_i)^2|^{\frac{1}{2}}$.

- **Norma infinito (∞-Norm)**. Considera únicamente el valor máximo absoluto de las diferencias entre los componentes espectrales de dos píxeles $\mathbf{x}$ e $\mathbf{y}$, definido por $\|\mathbf{x}-\mathbf{y}\|_\infty = max(|x_i - y_i|, i = 1,2,\ldots,b)$.

- **Distancia del ángulo espectral (SAD, *Spectral Angle Distance*)**. Define la similitud entre dos píxeles $\mathbf{x}$ e $\mathbf{y}$ en función del ángulo formado por sus firmas espectrales en un espacio multidimensional, lo que la hace invariante a cambios de iluminación. Está definida por la siguiente expresión:

$$\text{SAD}\ (\mathbf{x},\mathbf{y}) = \cos^{-1}\left(\frac{\mathbf{x}\cdot\mathbf{y}}{\|\mathbf{x}\|_2 \cdot \|\mathbf{y}\|_2}\right) \tag{4.3}$$

- **Divergencia de información espectral (SID, *Spectral Information Divergence*)**. Mide la diferencia de distribución espectral entre dos píxeles $\mathbf{x}$ e $\mathbf{y}$ utilizando una métrica basada en teoría de la información. Está definida por la siguiente expresión:

$$\text{SID}\ (\mathbf{x},\mathbf{y}) = \text{D}\ (\mathbf{x}\|\mathbf{y}) + \text{D}\ (\mathbf{y}\|\mathbf{x}), \tag{4.4}$$

  donde $\text{D}\ (\mathbf{x}\|\mathbf{y}) = \sum_{i=1}^{b} p_i \cdot \log(p_i/q_i)$. Para ello, definimos $p(x_j) = p_j = x_j/\sum_{i=1}^{b} x_i$ y $q(y_j) = q_j = y_j/\sum_{i=1}^{b} y_i$.

- **La raíz cuadrada del error cuadrático de la media de la suma de las bandas espectrales (BSMSE, *square root of band sum mean squared error*)**. Se realiza la

comparación de dos firmas espectrales minimizando el incremento de la media del error cuadrático, definidos por la siguiente exprexión:

$$\text{BSMSE}\ (\mathbf{x},\mathbf{y}) = \sqrt{\frac{1}{b}\sum_{i=1}^{b}(x_i - y_i)^2} \tag{4.5}$$

- **La raíz cuadrada del error cuadrático de la media de los máximos de las bandas espectrales (BSMME, *square root of band maximum mean squared error*)**. Al igual que la anterior, se minimiza el incremento de la media del error cuadrático tomando el valor máximo de todas las bandas espectrales, definidos por la siguiente expresión:

$$\text{BSMME}\ (\mathbf{x},\mathbf{y}) = |max(x) - max(y)| \tag{4.6}$$

- **La distancia vectorial normalizada (NVD, *normalized vector distance*)**. Se basa en la combinación de una medida de módulo vectorial, por ejemplo la norma 2, con el SAD. Para este criterio, dos firmas espectrales serán iguales si tienen el mismo módulo y un SAD igual a 0. La expresión que lo define es la siguiente:

$$\text{NVD}\ (\mathbf{x},\mathbf{y}) = \frac{\|\mathbf{x}-\mathbf{y}\|_2}{\|\mathbf{x}\|_2 + \|\mathbf{y}\|_2} \tag{4.7}$$

Estas alternativas permiten reducir la carga computacional y optimizar el rendimiento del algoritmo en diferentes arquitecturas paralelas. Sin embargo, los resultados de precisión obtenidos hasta el momento no son del todo concluyentes, ya que el análisis se ha realizado sobre un conjunto limitado de imágenes [81]. Para validar la eficacia de cada enfoque, sería necesario ampliar los experimentos a un mayor número de escenas con características variadas.

4.3. Variante seleccionada: ATDCA-GS

La variante ATDCA-GS ha sido ampliamente utilizada en la comunidad científica durante la última década [13, 69], logrando buenos resultados en términos de rendimiento, consumo energético y eficiencia en el desarrollo del código [10]. Debido a estas ventajas, este algoritmo es el candidato ideal para ser comparado con una versión optimizada en FPGA que ha sido desarrollada haciendo uso del lenguaje VHDL [69].

La versión seleccionada se basa en una implementación mejorada de la presentada en [12], con el objetivo de optimizar su ejecución en FPGAs, que constituyen el hardware objetivo de esta Tesis Doctoral. Para facilitar la comprensión de esta variante, a continuación, se describen los pasos del algoritmo representados en el Algoritmo 3.

Algoritmo 3 Pseudocódigo del algoritmo ATDCA-GS

1: **Entrada:** $\mathbf{X} \in \mathbb{R}^{p \times b}$ y t;
2: $\mathbf{U}$ =[$\mathbf{u}_0 \mid 0 \mid, \cdots, \mid 0$];
3: $\mathbf{B}$ =[$0 \mid 0 \mid, \cdots, \mid 0$];
4: $\mathbf{w} = [\mathbf{1}, \cdots, \mathbf{1}]$;
5: $P^{\perp}_{\mathbf{U}} = [\mathbf{1}, \cdots, \mathbf{1}]$;
6: **for** $i = 1$ to $t-1$ **do**
7: $\quad \mathbf{B}[:,i] = \mathbf{U}[:,i]$;
8: $\quad$ **for** $j = 2$ to i **do**
9: $\quad\quad proj_{\mathbf{B}[:,j-1]}(\mathbf{U}[:,i]) = \frac{\mathbf{U}[:,i]^T \mathbf{B}[:,j-1]}{den[j-1]} \mathbf{B}[:,j-1]$;
10: $\quad\quad \mathbf{B}[:,i] = \mathbf{B}[:,i] - proj_{\mathbf{B}[:,j-1]}(\mathbf{U}[:,i])$;
11: $\quad$ **end for** j
12: $\quad proj_{\mathbf{B}[:,\mathrm{i}]}(\mathbf{w}) = \frac{\mathbf{w}^T \mathbf{B}[:,i]}{\mathbf{B}[:,i]^T \mathbf{B}[:,i]} \mathbf{B}[:,i]$;
13: $\quad den[i] = \mathbf{B}[:,i]^T \mathbf{B}[:,i]$;
14: $\quad P^{\perp}_{\mathbf{U}} = P^{\perp}_{\mathbf{U}} - proj_{\mathbf{B}[:,\mathrm{i}]}(\mathbf{w})$;
15: $\quad \mathbf{v} = P^{\perp}_{\mathbf{U}} \mathbf{X}$;
16: $\quad i = \mathrm{argmax}_{\{1,\cdots,p\}} \mathbf{v}[:,i]$;
17: $\quad \mathbf{u}_i \equiv \mathbf{U}[:,i+1] = \mathbf{X}[:,i]$;
18: **end for**
19: **Salida:** $\mathbf{U} = [\mathbf{u}_0, \mathbf{u}_1, \cdots, \mathbf{u}_{t-1}]$;

Las entradas de este algoritmo están compuestas por la imagen hiperespectral, denotada como $\mathbf{X}$, y el número de endmembers a extraer, denotado como t. La imagen está formada por p píxeles y b bandas espectrales. En esta implementación, la ortogonalidad, representada por $P^{\perp}_{\mathbf{U}}$, se garantiza a través de la base ortogonal $\mathbf{B}$, la cual se actualiza en cada iteración al obtener un nuevo endmember en la matriz $\mathbf{U}$ mediante el método de Gram-Schmidt (GS). Este método evita la necesidad de realizar operaciones inversas para obtener proyecciones ortogonales, a cambio de realizar operaciones más simples y más eficientes para su implementación en hardware. Específicamente, GS selecciona un conjunto finito de vectores linealmente independientes en el espacio $\mathbb{R}^{p \times b}$, en el que se define la imagen hiperespectral, y genera un conjunto de vectores ortogonales a partir de los vectores candidatos seleccionados previamente, almacenándolos en $\mathbf{B}$. Esto se realiza en el paso 10, ajustando los vectores mediante la resta de las proyecciones anteriores.

Tras ajustar las proyecciones, el algoritmo calcula una nueva proyección de $\mathbf{B}$ sobre el vector $\mathbf{w}$. En implementaciones anteriores, se generaba un vector aleatorio $\mathbf{w}$ en cada iteración

para actualizar $P_{\mathbf{U}}^{\perp}$. Sin embargo, en esta implementación se optó por fijarlo en $[\mathbf{1},\ldots,\mathbf{1}]^T$, con el fin de evitar la generación de vectores aleatorios y reducir así el coste computacional del paso 12, logrando además que el algoritmo sea determinista. En el paso 14, se reajustan nuevamente las proyecciones para asegurar que se mantenga la ortogonalidad respecto a los endmembers previamente seleccionados. De esta forma, se obtienen las proyecciones ortogonales de los candidatos a endmembers sobre la imagen hiperespectral, lo que permite identificar la máxima proyección y seleccionar dicho candidato como el nuevo endmember. Finalmente, el algoritmo devuelve todos los endmembers seleccionados en la matriz $\mathbf{U}$.

Es importante señalar que se han implementado varias mejoras algorítmicas en comparación con la versión utilizada anteriormente en [12]. Según el Algoritmo 3, se ha logrado reutilizar $P_{\mathbf{U}}^{\perp}$ a lo largo de las diferentes iteraciones del bucle principal, lo que ha permitido reducir la complejidad en aproximadamente un 50 %, eliminando la necesidad de calcularlo en cada iteración. Además, el cálculo de $\mathbf{B}[:,i]^T\mathbf{B}[:,i]$ también se almacena entre iteraciones, lo que optimiza el tiempo de actualización de $\mathbf{B}$. Por este motivo, el denominador del paso 12 se guarda en el paso 13 y luego se reutiliza en el paso 9 durante las iteraciones siguientes.

4.4. Implementación HLS del algoritmo ATDCA-GS

Para lograr una implementación eficiente del algoritmo ATDCA-GS mediante HLS en FPGAs, es fundamental llevar a cabo un perfilado del código. Este proceso permite analizar en detalle la ejecución del algoritmo, identificando las secciones que presentan mayor consumo computacional y determinando los cuellos de botella que limitan el rendimiento. A partir de este análisis, es posible seleccionar estratégicamente las regiones del código más adecuadas para ser paralelizadas y aceleradas, optimizando así el uso de los recursos del hardware. De esta manera, se maximiza la eficiencia computacional, se reduce el tiempo de ejecución y se mejora el aprovechamiento de la arquitectura FPGA, asegurando un diseño equilibrado entre rendimiento, consumo energético y utilización de recursos.

Para realizar un análisis más preciso del perfilado, es importante considerar que la ejecución del algoritmo ATDCA-GS puede descomponerse en cuatro fases principales:

1. **Cálculo del primer endmember**. Selección del píxel inicial con la mayor intensidad espectral.

2. **Generación del proyector ortogonal** $P_{\mathbf{U}}^{\perp}$. Garantiza la ortogonalidad en la selección de los siguientes endmembers.

3. **Proyección de la imagen** sobre el subespacio ortogonal $P_{\mathbf{U}}^{\perp}\mathbf{X}$. Evalúa la contribución de cada píxel.

4. **Identificación del endmember con la máxima proyección**. Determinación del píxel más representativo para su inclusión en el conjunto final.

Este esquema permite estructurar y optimizar cada fase del algoritmo, facilitando la identificación de posibles mejoras en términos de rendimiento y eficiencia computacional.

Después de realizar el perfilado del código ATDCA-GS utilizando tres imágenes de distintos tamaños, se tomó como referencia la extracción de 19 endmembers [12]. El análisis de los resultados reveló que el **cálculo de las proyecciones** (tercera fase) es la operación con mayor impacto en el tiempo de ejecución, representando entre un **94 % y un 96 %** del tiempo total de procesamiento a medida que aumenta el tamaño de la imagen. La segunda fase más costosa en términos computacionales corresponde a la **extracción del primer endmember** (primera fase), que consume entre un **3.5 % y un 5.5 %** del tiempo total. Debido a estos hallazgos, el enfoque de optimización y aceleración se centrará en estas dos fases, priorizando la reducción de su impacto computacional para mejorar la eficiencia del algoritmo.

Antes de iniciar el proceso de optimización, es fundamental disponer de una versión base del código con mínimas modificaciones, garantizando así su correcta ejecución en una FPGA. Esta versión servirá como punto de referencia para evaluar el impacto de las optimizaciones y asegurar una comparación precisa en términos de rendimiento y eficiencia.

4.4.1. Versión base (*baseline*) del ATDCA-GS

Para llevar a cabo las distintas optimizaciones en la implementación ATDCA-GS, es imprescindible contar con una versión base que sirva como referencia para futuras comparaciones. Esta versión presentará mínimas modificaciones con respecto a una implementación secuencial en C++ y SYCL, la cual será compilada utilizando el compilador DPC++. Para establecer esta versión base, es necesario incorporar los siguientes elementos (indicar que el elemento [0] es optativo pero aconsejable para disponer de un código más legible), tal y como se muestra en el Código 4.1:

[0] Aislar el código en una función donde se realice la llamada al *kernel* que se ejecutará en la placa FPGA, dejando la lectura de la imagen y la creación de los buffers de entrada/salida fuera de esta función.

[1] Definir un selector de dispositivo dirigido al uso de FPGAs.

[2] Crear un DPC++ dispositivo cola (*queue*) usando el selector anterior.

[3] Utilizar buffers para los arrays de entrada y salida de datos de nuestra implementación. En este caso, utilizaremos un buffer de entrada para la imagen hiperespectral y otros dos buffers de salida para almacenar las firmas espectrales y las posiciones de los endmembers.

[4] Uso de *submit* para ejecutar un conjunto de comandos en el dispositivo objetivo.

[5] Utilizar accesores para los diferentes buffers creados, permitiendo que la FPGA acceda a los datos.

[6] Uso de *single_task* para enviar el *kernel* a ejecución. Los contenidos de *buffer_end* y *buffer_P* serán copiados a *h_end* y *P*, respectivamente, cuando la función termine.

Código 4.1 Estructura de la versión base para el algoritmo ATDCA-GS.

```
1  void ATDCA_GS(const std::vector<float> &h_image, std::vector<float> &
       h_end, std::vector<long int> &P, int Samples, int Lines, int bands
       , int num_endmembers) {
2      #if defined(FPGA_EMULATOR)
3          ext::intel::fpga_emulator_selector device_selector;
4      #else
5          [1] ext::intel::fpga_selector device_selector;
6      #endif
7
8      try {
9          [2] queue q(device_selector, dpc_common::exception_handler,
10             property::queue::enable_profiling{});
11         [3] buffer buffer_image(h_image);
12         [3] buffer buffer_end(h_end);
13         [3] buffer buffer_P(P);
14
15         [4] event e = q.submit([&](handler &h) {
16             [5] accessor acc_image(buffer_image,h,read_only);
17             [5] accessor acc_end(buffer_end,h,read_write, no_init);
18             [5] accessor acc_P(buffer_P,h,write_only,no_init);
19
20             [6] h.single_task([=]()
21             [[intel::kernel_args_restrict]] {
22                 // Código del kernel ATDCA-GS a ejecutar en FPGA
23             });
24         });
```

```
25      }
26      catch (sycl::exception const &e) {
27          // Detección de excepciones en el código del host
28      }
29  }
```

Dado que los accesos a memoria son secuenciales y predecibles, el uso de técnicas como la partición o la distribución en bancos de memoria podría no aportar mejoras significativas e incluso reducir el rendimiento. Al no producirse conflictos de acceso y tratarse de un acceso lineal, el diseño ya se beneficia del comportamiento eficiente de la caché por defecto. El uso de directivas como [intel::partition] e [intel::numbanks(N)], están más orientadas a situaciones donde existen múltiples accesos paralelos o potenciales conflictos que podrían degradar el rendimiento. Si en el futuro se modifica el patrón de acceso o se busca un mayor paralelismo en el kernel, sí sería recomendable considerarlas.

Los accesos a memoria se gestionan mediante objetos *buffer* y *accessor*, lo que permite al entorno SYCL construir un grafo de ejecución basado en dependencias de datos. Esto asegura un modelo de sincronismo automático entre kernels, donde la ejecución se programa en función de la disponibilidad de los datos requeridos. Este enfoque se alinea con el modelo de ejecución de SYCL, evitando condiciones de carrera sin necesidad de sincronización manual.

Con los cambios previamente mencionados, se cuenta con un código *baseline* a partir del cual se podrá generar un informe destinado a su optimización. Esta versión no incorpora ninguna técnica de optimización, ya que más adelante se estudiará la viabilidad de implementar posibles mejoras sobre esta versión base. Dicho informe se elabora tras una compilación rápida, lo que permite obtener una serie de recomendaciones sobre cómo modificar los *kernels* para mejorar el rendimiento. Además, proporciona información crucial para el desarrollo, tales como: vistas de las estructuras (mapeo del código sobre los recursos de la FPGA), diseño de la memoria, análisis de rendimiento, identificación de cuellos de botella (áreas del código donde el rendimiento se ve limitado) y estimaciones de los recursos consumidos por el diseño.

El informe generado proporciona información clave para el análisis del diseño, siendo especialmente útil la vista de planificación, que detalla la planificación estática ciclo a ciclo y la latencia de los grupos de instrucciones, permitiendo examinar el comportamiento del sistema en FPGA y optimizar su rendimiento. En la Figura 5.2, se muestra el diagrama de planificación correspondiente a la versión *baseline*, dividida en tres fases: búsqueda del primer endmember, búsqueda del segundo endmember y búsqueda del resto de endmembers.

La distinción entre la búsqueda del segundo y los endmembers restantes responde a que el cálculo del primer proyector es más sencillo, por lo que en la implementación original se trataba de manera diferenciada. Dentro de cada fase, se han identificado las operaciones clave: el cálculo del píxel más brillante, empleado para la selección inicial del endmember; el cálculo del proyector ortogonal, necesario para garantizar la independencia entre endmembers; la proyección de los píxeles de la imagen sobre el proyector, que permite evaluar su relevancia en la selección de endmembers; y la copia del píxel seleccionado desde la imagen al conjunto final de endmembers, asegurando su almacenamiento en la estructura de datos adecuada. En el siguiente apartado, se analizarán en detalle estos procesos y su impacto en el rendimiento y eficiencia de la implementación.

Antes de abordar la primera optimización, es fundamental analizar el rendimiento actual para cuantificar con precisión las mejoras introducidas. Para ello, las mediciones del tiempo de ejecución se han realizado mediante funciones de una clase de perfilado de SYCL, tal y como se muestra en el Código 4.2:

Código 4.2 Funciones para realizar el perfilado en el algoritmo ATDCA-GS.

```
1 get_profiling_info<info::event_profiling::command_start>
2 get_profiling_info<info::event_profiling::command_end>
```

En este perfilado, no se consideran las transferencias de datos entre el *host* y el dispositivo *"offload"*, ya que el enfoque está centrado en medir el tiempo de ejecución de un *kernel*. Es importante señalar que los tiempos así obtenidos se corresponden con ejecuciones reales sobre el dispositivo hardware (en este caso, una FPGA Intel Stratix 10 SX 2800), y se miden una vez completado el flujo completo de síntesis e implementación. Por tanto, no se trata de estimaciones previas ni simulaciones, sino de datos empíricos en entorno real. No obstante, estos tiempos reflejan exclusivamente la ejecución del *kernel* en placa, y no incluyen fases previas como la síntesis, *place & route* o generación del bitstream.

Para la versión *baseline*, se presentan los tiempos obtenidos en hardware, así como los recursos utilizados por esta versión en las diferentes escenas, a través de las Tablas 4.1-4.3. Al ser una primera versión con las mínimas modificaciones requeridas para su ejecución en FPGA, fue necesario realizar tres síntesis distintas, adaptadas a las dimensiones de cada imagen y al número de endmembers a extraer. Se puede observar que la principal diferencia radica en la cantidad de RAM utilizada. Además, al comparar la síntesis de la imagen AVIRIS WTC con la de la imagen sintética, los valores son idénticos, ya que el consumo de memoria depende del número de bandas y de endmembers a extraer, que en ambos casos es el mismo. En las siguientes optimizaciones, se tomó como referencia la síntesis correspondiente a la

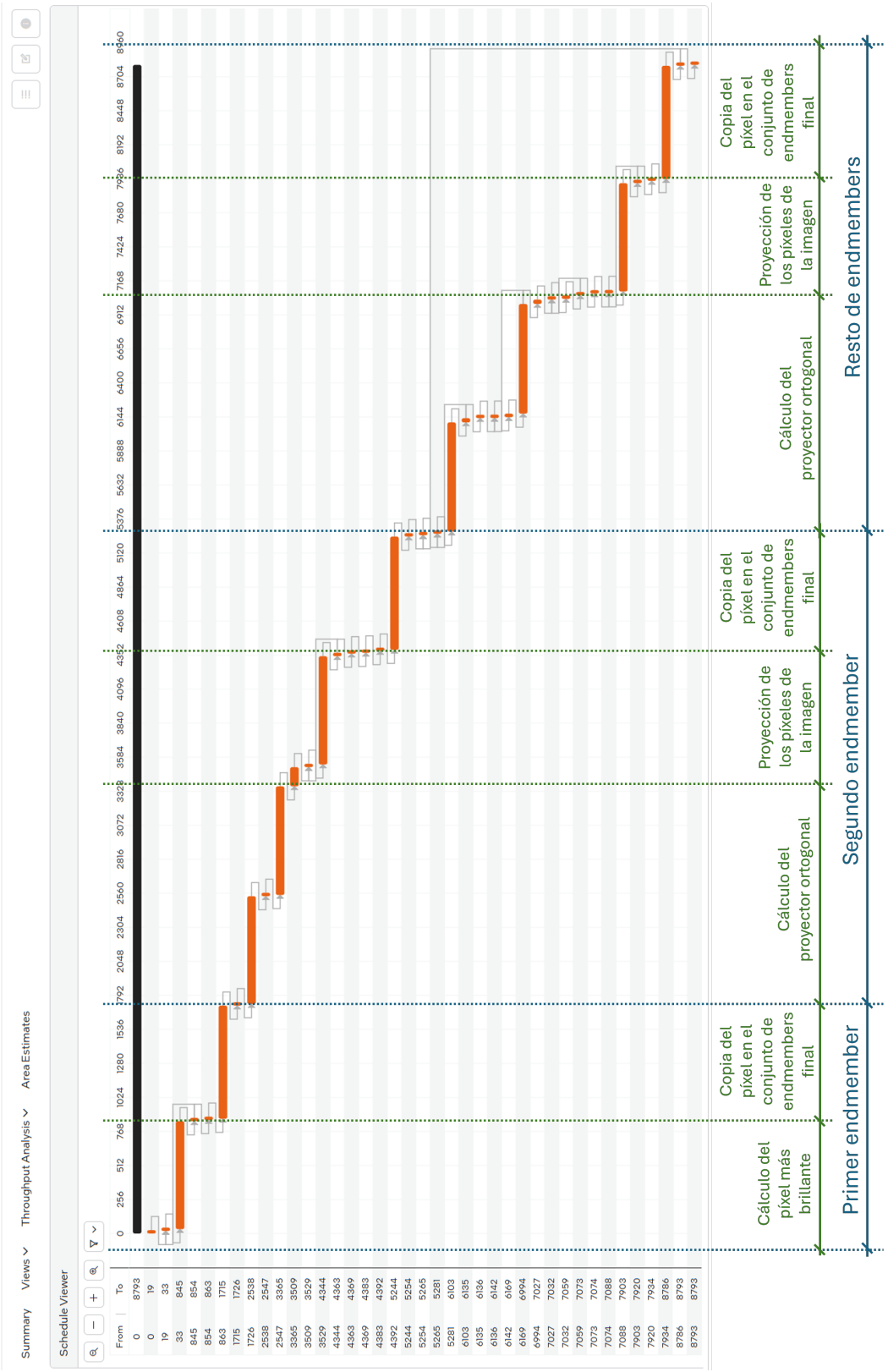


Figura 4.2 Diagrama temporal ciclo a ciclo de la versión ATDCA-GS *baseline*.

imagen AVIRIS Cuprite, y en la versión final, el código se generalizó para admitir cualquier tamaño de imagen y número de endmembers, teniendo una única síntesis para todas.

Implementación	Tiempo (ms)	F_{max} (MHz)	Recursos				
			ALUTs	FFs	RAMs	MLABs	DSPs
Baseline	2345.51	196.61	50482 (3 %)	102527 (3 %)	1267 (11 %)	560 (1 %)	61 (1 %)

Tabla 4.1 Tiempo de procesamiento y recursos utilizados por el algoritmo ATDCA-GS para la imagen AVIRIS Cuprite usando la FPGA Intel Stratix 10 SX 2800 para la versión *baseline*.

Implementación	Tiempo (ms)	F_{max} (MHz)	Recursos				
			ALUTs	FFs	RAMs	MLABs	DSPs
Baseline	10799.9	196.61	50491 (3 %)	102545 (3 %)	2067 (18 %)	560 (1 %)	61 (1 %)

Tabla 4.2 Tiempo de procesamiento y recursos utilizados por el algoritmo ATDCA-GS para la imagen AVIRIS WTC usando la FPGA Intel Stratix 10 SX 2800 para la versión *baseline*.

Implementación	Tiempo (ms)	F_{max} (MHz)	Recursos				
			ALUTs	FFs	RAMs	MLABs	DSPs
Baseline	16743.9	196.61	50491 (3 %)	102545 (3 %)	2067 (18 %)	560 (1 %)	61 (1 %)

Tabla 4.3 Tiempo de procesamiento y recursos utilizados por el algoritmo ATDCA-GS para la imagen sintética usando la FPGA Intel Stratix 10 SX 2800 para la versión *baseline*.

4.4.2. Primera optimización en ATDCA-GS: aumento de acumuladores (ACC)

Al analizar detenidamente el diagrama temporal de la versión *baseline* (ver Figura 5.2), se observó que todas las barras que presentan mayor latencia (o número de ciclos) corresponden, en todos los casos, a accesos a memoria externa. Esto sugiere que se debe priorizar la reducción de este número de ciclos o, alternativamente, intentar realizar varios accesos secuenciales. En la Figura 4.3 se ilustra, como ejemplo de lo mencionado, el diagrama temporal detallado del cálculo del píxel más brillante en la versión baseline. Como puede

apreciarse claramente, la operación que presenta la mayor latencia es *LD*, que se encarga de transferir una muestra del píxel desde la memoria externa hasta la FPGA. Esta latencia es constante, con un valor de 795 ciclos, lo que indica que se debe concentrar en optimizar los accesos a memoria, especialmente en aquellos casos en los que estos se encuentren dentro de los bucles con mayor número de iteraciones.

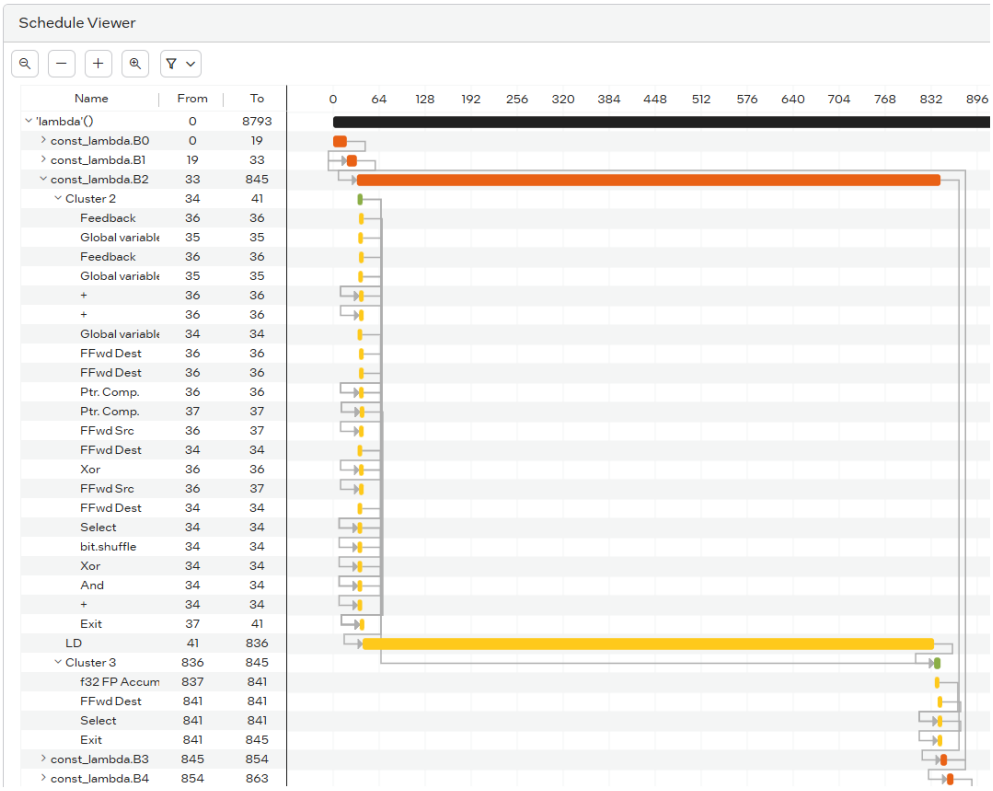


Figura 4.3 Diagrama temporal detallado del cálculo del píxel más brillante en la versión *baseline*.

Al analizar el informe obtenido de la versión baseline, es posible identificar los primeros problemas de rendimiento en la implementación. El primero de estos se presenta durante el cálculo del primer endmember. A continuación, en el Código 4.3 se muestra la porción de código correspondiente a esta parte de la implementación *baseline*:

Código 4.3 Fragmento del kernel para calcular el píxel más brillante en el algoritmo ATDCA-GS.

```
1 //First endmember
2 float bright_actual;
3 for (i = 0; i < SamplesLines; i++) {
4     bright_actual = 0.0;
5     for (k = 0; k < bands; k++) {
6         bright_actual += acc_image[i*bands+k] * acc_image[i*bands+k];
7     }
```

```
8       if (bright_actual > bright) {
9           bright = bright_actual;
10          pos = i;
11      }
12  }}
```

En el código anterior, se puede observar la implementación tradicional del cálculo del primer endmember en una plataforma CPU, en la que se calcula el producto escalar de un vector consigo mismo. Por lo tanto, tendremos que hacer tantos accesos a memoria externa como muestras tiene la imagen (*SampleLines* * *bands*). Este proceso consiste en multiplicar los elementos correspondientes a las mismas posiciones de ambos vectores y luego sumar el resultado en un acumulador. Debido a esto, la siguiente multiplicación y acumulación no puede comenzar hasta que la operación anterior se haya completado, ya que cada iteración del bucle *for* depende del valor de *bright_actual* de la iteración anterior, generando una dependencia en *bright_actual*.

El mismo problema de rendimiento se repite en el cálculo de los demás endmembers, en particular cuando se realiza la proyección de los píxeles de la imagen sobre el vector ortogonal (línea 5 del Código 4.4) y se obtiene el valor máximo de todos ellos. En este caso, surge otra dependencia al intentar obtener el valor de *value*. A continuación, se muestra la porción de código correspondiente a esta parte del proceso:

Código 4.4 Fragmento del kernel para calcular la proyección de los píxeles de la imagen en el algoritmo ATDCA-GS.

```
1   float value;
2   for(iter = 0; iter < SamplesLines; iter += 1) {
3       value = 0.0;
4       for (j = 0; j < bands; j += 1) {
5           value += acc_image[iter * bands + j] * h_f[j];
6       }
7       h_reduction[iter] = value * value;
8   }
9
10  float max_local = 0.0;
11  for(iter = 0; iter < SamplesLines; iter += 1) {
12      if(h_reduction[iter] > max_local) {
13          max_local = h_reduction[iter];
14          pos = iter;
15      }
16  }
```

Se exploraron distintas estrategias para mitigar este cuello de botella. En particular, se evaluó el uso de directivas de síntesis como *loop_unroll* e *ivdep* con el objetivo de forzar el desenrollado del bucle y eliminar las dependencias aparentes. Sin embargo, el sintetizador no permitió el desenrollado automático al detectar una dependencia real en la variable de acumulación. Se probó también el uso conjunto con *ivdep*, pero la operación de suma se completaba antes que el acceso a memoria y la multiplicación, lo que resultaba en acumulaciones incorrectas por falta de sincronización de datos. Aunque la documentación de Intel propone una estructura alternativa específica para este tipo de acumulaciones, dicha solución mostró un consumo reducido de recursos, pero fue menos eficiente que nuestra aproximación. Por ello, se optó finalmente por realizar el desenrollado de forma manual, dividiendo el trabajo entre múltiples acumuladores, lo que resultó en una mejora de rendimiento significativa en nuestro caso concreto.

En hardware, se puede acelerar este proceso aumentando el número de acumuladores, de manera que, en cada ciclo, se inicie una nueva multiplicación y, tras la latencia asociada a esa operación, el resultado se sume al acumulador correspondiente. De esta forma, al finalizar todas las multiplicaciones y operaciones de acumulación, se tendrá el valor del producto escalar distribuido entre todos los acumuladores. Finalmente, se necesitaría sumar todos los valores utilizando un árbol de sumadores. Para facilitar la comprensión de este enfoque, se muestra a continuación el Código 4.5 modificado, simplificado para el uso de cuatro acumuladores en el cálculo del primer endmember:

Código 4.5 Aumento de acumuladores (ACC) en el algoritmo ATDCA-GS.

```
1  float buffer [SAMPLES_LINES];
2  float acc_01, acc_02, acc_03, acc_04;
3
4  //First endmember
5  for (i = 0; i < SamplesLines; i++) {
6      acc_01 = 0.0f;
7      acc_02 = 0.0f;
8      acc_03 = 0.0f;
9      acc_04 = 0.0f;
10     for (k = 0; k < bands; k += 4) {
11         acc_01 += acc_image[i*bands+k]   * acc_image[i*bands+k];
12         acc_02 += acc_image[i*bands+k+1] * acc_image[i*bands+k+1];
13         acc_03 += acc_image[i*bands+k+2] * acc_image[i*bands+k+2];
14         acc_04 += acc_image[i*bands+k+3] * acc_image[i*bands+k+3];
15     }
16     buffer[i] = (acc_01 + acc_02) + (acc_03 + acc_04);
17 }
```

```
18
19 for (i = 0; i < SamplesLines; i++) {
20     if (buffer[i] > bright) {
21         bright = buffer[i];
22         pos = i;
23     }
24 }
```

Las modificaciones anteriores ofrecen varias ventajas, tanto para el cálculo del primer endmember como para la proyección de los píxeles de la imagen sobre el vector ortogonal. Entre estas ventajas, destacan el acceso a memoria secuencial para la lectura de varias muestras de un píxel, la ejecución simultánea de varias operaciones de multiplicación y una reducción considerable del número de iteraciones, ya que las acumulaciones se realizan de forma independiente, lo que, a su vez, conlleva una disminución del tiempo de ejecución. La Figura 4.4 muestra el diagrama temporal detallado del cálculo del píxel más brillante en esta versión con 4 acumuladores. Se observa que la latencia de la operación *LD* ha aumentado hasta 842 ciclos; no obstante, es importante tener en cuenta que, en este caso, se está accediendo a 4 muestras. Finalmente, también se aprecia cómo se llevan a cabo 4 operaciones de acumulación en paralelo.

Para evaluar esta optimización, fue necesario probar diferentes cantidades de acumuladores, desde 4 hasta 64, con el fin de encontrar el mejor equilibrio entre el tiempo de procesamiento y la cantidad de recursos utilizados. Para ello, se realizó un análisis del informe, cuyos resultados se presentan en la Tabla 4.4, utilizando la escena AVIRIS Cuprite.

Implementación	**Tiempo (ms)**	**F_{max} (MHz)**	**Recursos**				
			ALUTs	**FFs**	**RAMs**	**MLABs**	**DSPs**
4_ACC	884.53	196.61	51282 (3%)	104124 (3%)	1231 (11%)	587 (1%)	77 (1%)
8_ACC	675.52	196.61	53137 (3%)	106685 (3%)	1229 (10%)	637 (1%)	101 (2%)
16_ACC	440.66	196.61	56590 (3%)	110374 (3%)	1229 (10%)	701 (1%)	132.5 (2%)
32_ACC	419.48	196.61	65136 (3%)	122077 (3%)	1280.4 (11%)	797 (1%)	260.5 (5%)
64_ACC	437.19	196.61	90663 (5%)	157845 (4%)	1417.3 (12%)	1250 (1%)	487.5 (8%)

Tabla 4.4 Tiempo de procesamiento y recursos utilizados por el algoritmo ATDCA-GS para la imagen AVIRIS Cuprite usando la FPGA Intel Stratix 10 SX 2800 incrementando el número de acumuladores.

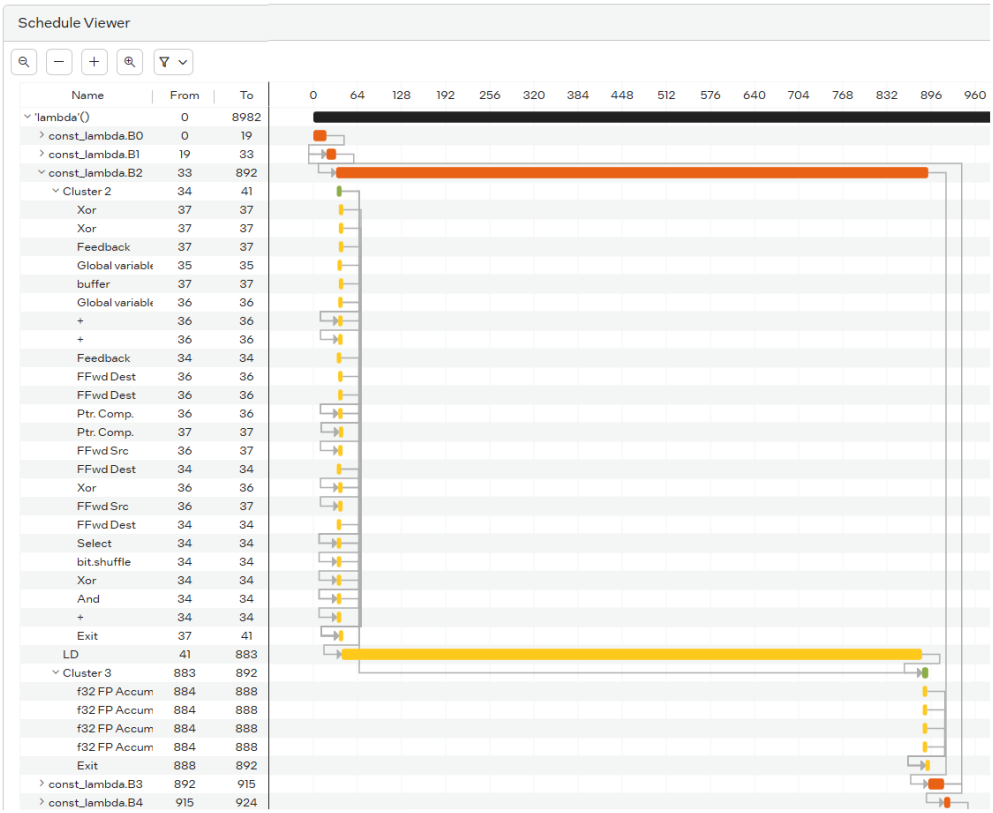


Figura 4.4 Diagrama temporal detallado del cálculo del píxel más brillante en la versión ACC con 4 acumuladores.

En la Figura 4.5 se muestra el diagrama de planificación de la versión con 16 acumuladores, adoptada como solución de compromiso entre rendimiento y utilización de recursos. Aunque el número de ciclos ha aumentado en comparación con la versión *baseline*, es importante destacar que el número de iteraciones se ha reducido considerablemente, lo que se traduce en una mejora del rendimiento global. Esto se refleja claramente en los resultados de tiempo de ejecución presentados en la Tabla 4.4.

4.4.3. Segunda optimización en ATDCA-GS: conversión de punto flotante a entero (INT)

En el uso de FPGAs, es fundamental convertir los datos y operadores en punto flotante a un formato entero siempre que sea posible. Esta conversión ofrece múltiples ventajas, principalmente en términos de rendimiento, eficiencia y facilidad de implementación. Al reducir la complejidad de la lógica y el uso de memoria, se disminuye el número de ciclos de ejecución (ver Figura 4.6), lo que a su vez optimiza el consumo de recursos del sistema.

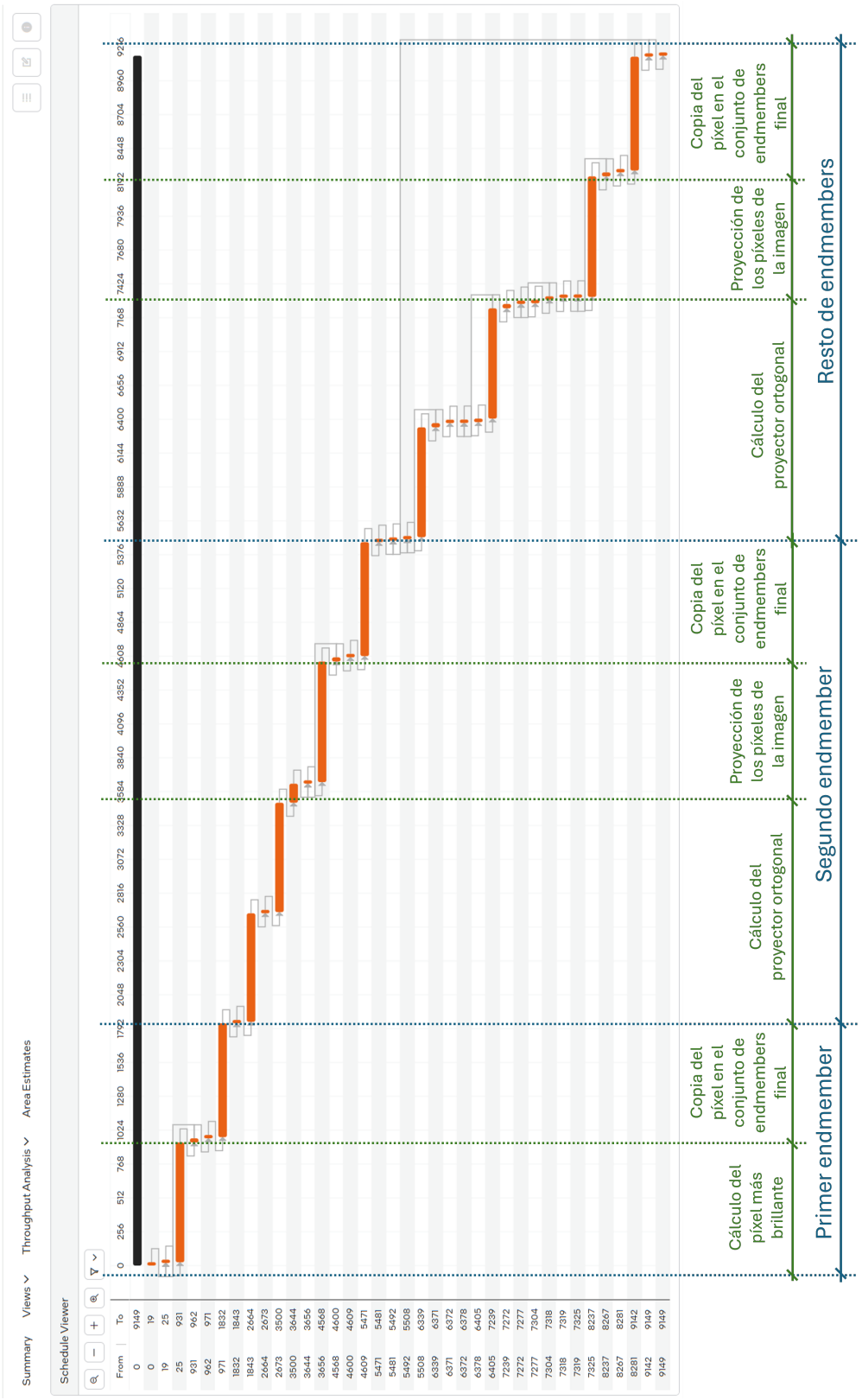


Figura 4.5 Diagrama temporal ciclo a ciclo de la versión con 16 acumuladores.

Aunque la FPGA utilizada cuenta con una cantidad considerable de recursos DSP, es importante destacar que el objetivo final es implementar una cadena completa de desmezclado, compuesta por tres algoritmos distintos. Por esta razón, la optimización del uso de recursos resulta fundamental para garantizar una implementación eficiente. En términos de precisión numérica, el algoritmo ATDCA-GS no requiere una representación de alta precisión, ya que la información manejada por los números en punto flotante puede ser adecuadamente representada dentro de un rango limitado sin afectar significativamente su eficacia.

A continuación, la Tabla 4.5 presenta la mejora obtenida en tiempo de procesamiento y uso de recursos tras aplicar la conversión de datos y operadores de punto flotante a entero en la escena AVIRIS Cuprite.

Implementación	**Tiempo (ms)**	**F_{max} (MHz)**	**Recursos**				
			ALUTs	**FFs**	**RAMs**	**MLABs**	**DSPs**
16_ACC_INT	431.93	196.61	82438 (4%)	136161 (4%)	1420 (12%)	776 (1%)	127.5 (2%)
32_ACC_INT	426.93	196.61	113500 (6%)	170251 (5%)	1458.4 (12%)	1025 (1%)	215.5 (4%)

Tabla 4.5 Tiempo de procesamiento y recursos utilizados por el algoritmo ATDCA-GS para la imagen AVIRIS Cuprite usando la FPGA Intel Stratix 10 SX 2800 para la optimización de convertir los datos y operadores de punto flotante a entero.

Analizando los resultados de las Tablas 4.4 y 4.5 para las configuraciones con 16 y 32 acumuladores, se observa que la conversión de punto flotante a entero mejora ligeramente los tiempos de procesamiento (por ejemplo, de 440.66 ms a 431.93 ms con 16 acumuladores). Sin embargo, esta mejora temporal implica un aumento notable en el uso de recursos lógicos, especialmente ALUTs y FFs, que casi se duplican en algunos casos. En cambio, el consumo de memoria y DSPs se mantiene relativamente estable. Por lo tanto, aunque la optimización mejora el rendimiento, también incrementa significativamente el uso de lógica FPGA, lo que requiere un balance cuidadoso entre coste y beneficio, sobre todo considerando la integración de múltiples algoritmos en una misma cadena de desmezclado.

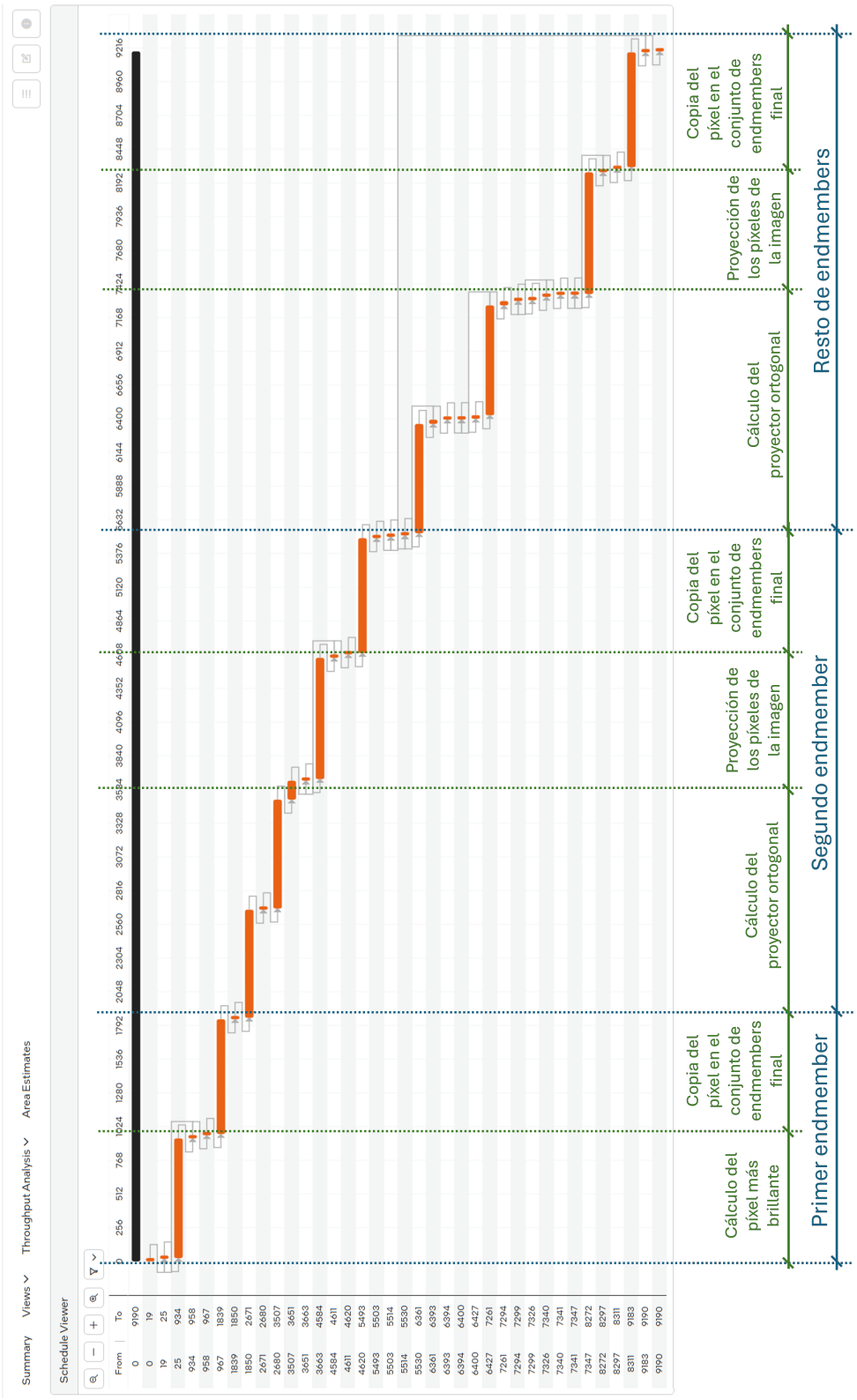


Figura 4.6 Diagrama temporal ciclo a ciclo de la versión con 16 acumuladores y con datos y operadores en formato entero.

4.4.4. Tercera optimización en ATDCA-GS: comparaciones en paralelo (CMP-MAX)

Si se retoma la sección del código que mayor tiempo de ejecución consume, según el perfilado inicial, se puede identificar otro punto crítico: la búsqueda del píxel con la máxima proyección mostrada en el Código 4.6.

Código 4.6 Fragmento de código para la búsqueda del píxel con la máxima proyección en el algoritmo ATDCA-GS.

```
1 float max_local = 0.0;
2 for(iter = 0; iter < SamplesLines; iter += 1) {
3     if(h_reduction[iter] > max_local) {
4         max_local = h_reduction[iter];
5         pos = iter;
6     }
7 }
```

En la versión software original, este proceso se realiza comparando cada valor con el máximo encontrado hasta el momento, actualizándolo si es necesario. Como consecuencia, no se lleva a cabo una nueva comparación hasta que la anterior ha finalizado, lo que introduce una dependencia secuencial que limita el rendimiento. En hardware, el rendimiento del código puede mejorarse ejecutando múltiples comparaciones en paralelo, como se ilustra en el siguiente fragmento de Código 4.7. El número óptimo de comparaciones dependerá de los resultados obtenidos en el informe, siempre que se mantenga un equilibrio entre el tiempo de procesamiento y el uso de recursos de la optimización.

Código 4.7 Fragmento del kernel para realizar múltiples comparaciones en paralelo (CMP-MAX) en el algoritmo ATDCA-GS.

```
1  long int max01 = 0;
2  long int max02 = 0;
3  int pos01;
4  int pos02;
5
6  for(int s = 0; s < SamplesLines; s += 2) {
7      if (h_reduction[s] > max01) {
8          max01 = h_reduction[s];
9          pos01 = s;
10     }
11     if (h_reduction[s + 1] > max02) {
12         max02 = h_reduction[s + 1];
```

```
13            pos02 = s + 1;
14        }
15    }
16
17    long int pos;
18    if (max01 > max02) {
19        pos = pos01;
20    } else {
21        pos = pos02;
22    }
```

La Figura 4.7 y la Figura 4.8 muestran, respectivamente, el diagrama temporal detallado del cálculo de la comparación en modo secuencial y el diagrama temporal detallado del cálculo de dos comparaciones en paralelo. Al contrastarlos, se puede apreciar la mejora obtenida con la ejecución en paralelo.

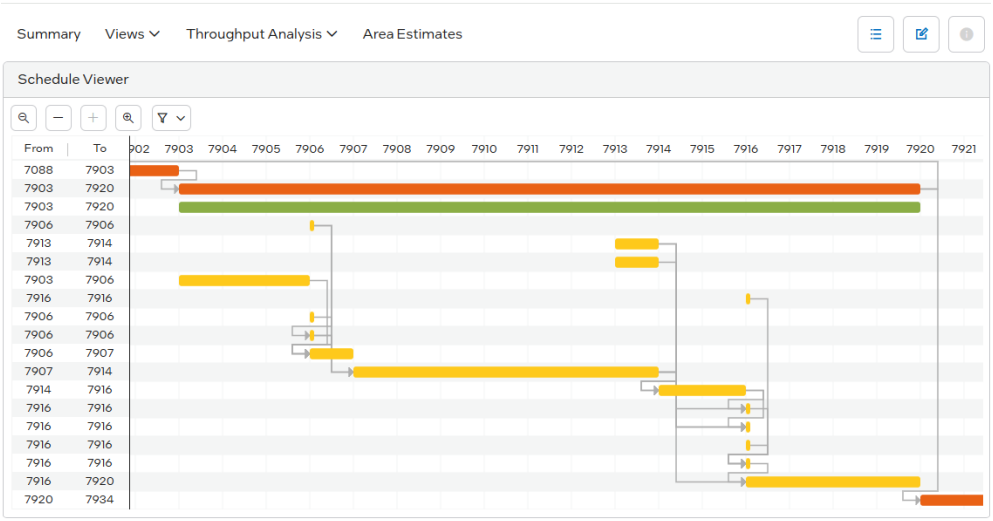


Figura 4.7 Diagrama temporal detallado del cálculo de la comparación en secuencial.

A continuación, la Tabla 4.6 presenta la mejora obtenida en tiempo de procesamiento y uso de recursos tras aplicar la comparación en paralelo de dos y cuatro valores en la escena AVIRIS Cuprite.

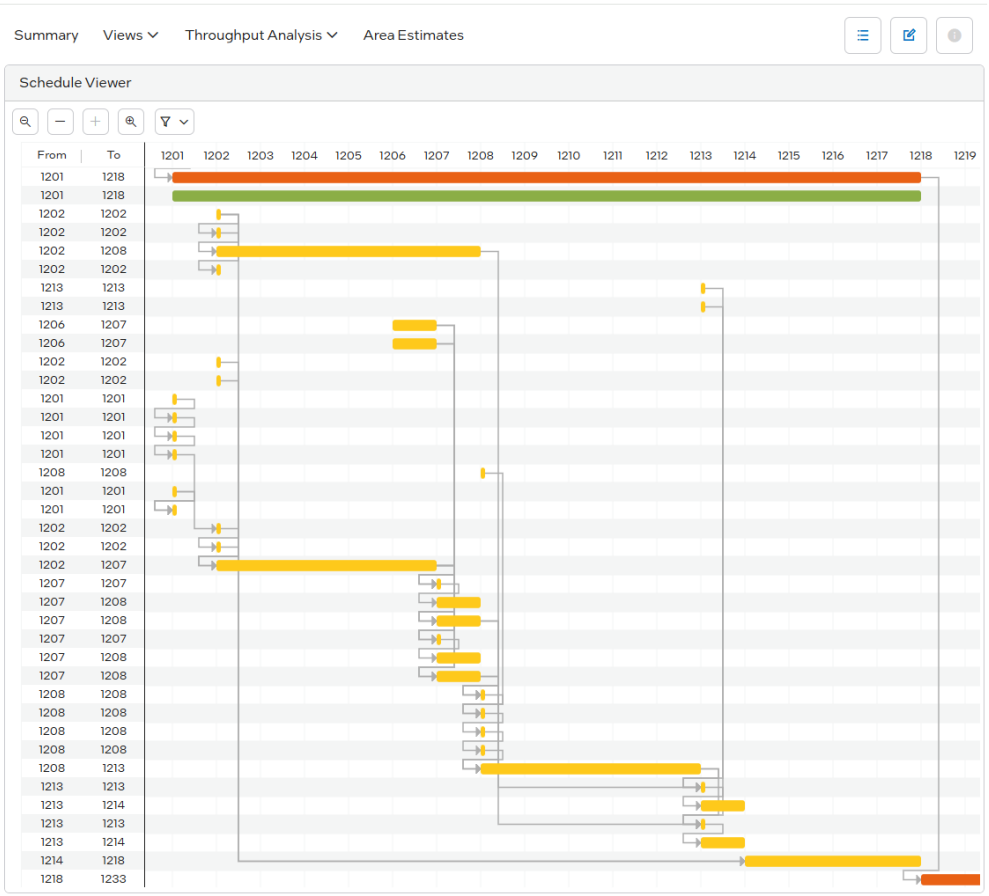


Figura 4.8 Diagrama temporal detallado del cálculo de dos comparaciones en paralelo.

Implementación	Tiempo (ms)	F_{max} (MHz)	Recursos				
			ALUTs	FFs	RAMs	MLABs	DSPs
16_ACC_INT _2CMP-MAX	312.51	196.61	86400 (5 %)	140200 (4 %)	1475 (13 %)	818 (1 %)	117 (2 %)
16_ACC_INT _4CMP-MAX	366.01	196.61	90372 (5 %)	144165 (4 %)	1529.6 (13 %)	861 (1 %)	107 (2 %)

Tabla 4.6 Tiempo de procesamiento y recursos utilizados por el algoritmo ATDCA-GS usando la FPGA Intel Stratix 10 SX 2800 para la optimización de usar varias comparaciones a la hora de obtener el valor máximo en paralelo.

4.4.5. Cuarta optimización en ATDCA-GS: reestructuración del código (UF)

Llegados a este punto donde se han optimizado todas las posibles secciones de código se aplicó una estrategia de mejora global. Una estrategia clave para mejorar la eficiencia de la implementación en FPGA es la reestructuración del código. Cabe destacar que la única parte que variaba entre el cálculo del primer endmember, el segundo y los sucesivos era la obtención del vector ortogonal específico a cada caso. Sin embargo, las operaciones de proyección de los píxeles sobre dicho vector y la búsqueda del valor máximo eran idénticas para todos los endmembers. Por ello, en lugar de replicar tres veces la misma lógica de proyección y comparación, hemos unificado ese bloque común en una única unidad de hardware. En la versión inicial, la existencia de estructuras duplicadas o triplicadas generaba un sobrecoste en el consumo de recursos, aumentando innecesariamente el uso de RAMs, ALUTs y FFs dentro del dispositivo.

Al consolidar estos cálculos en una única unidad de procesamiento, se consigue una reducción significativa en el uso de recursos, lo que no solo libera espacio para futuras optimizaciones o para la integración de los demás algoritmos de la cadena de desmezclado, sino que también mejora el rutado del diseño. Un mejor rutado implica que las señales internas de la FPGA pueden distribuirse de manera más eficiente, reduciendo la congestión en el enrutamiento y minimizando los retardos asociados a la propagación de señales.

Como consecuencia directa, la implementación puede operar a una frecuencia de reloj mayor, ya que la reducción de la complejidad estructural disminuye las restricciones de temporización y permite alcanzar ciclos de reloj más cortos. Esto se traduce en una mejora del rendimiento general del sistema, logrando tiempos de procesamiento más bajos sin comprometer la precisión de los cálculos. En definitiva, esta optimización no solo hace un uso más eficiente del hardware disponible, sino que también incrementa la capacidad de escalabilidad del diseño, permitiendo su adaptación a imágenes de mayor tamaño o configuraciones más exigentes sin un aumento excesivo del consumo de recursos.

En la Figura 4.9, se muestra el diagrama de planificación de la versión final, donde se aprecia claramente la reestructuración del código implementada. Por último, la Tabla 4.7 recoge los resultados obtenidos tras aplicar estas mejoras al código. Es importante destacar que, en esta versión, el código se ha generalizado para admitir cualquier tamaño de imagen (hasta 487,500 píxeles y 224 bandas) y un número máximo de 30 endmembers, lo que permite que todas las imágenes compartan una única síntesis. En caso de requerir un tamaño

mayor o un número superior de endmembers, sería suficiente con modificar la constante correspondiente y volver a ejecutar el proceso de síntesis.

Implementación	Tiempo (ms)	F_{max} (MHz)	Recursos				
			ALUTs	FFs	RAMs	MLABs	DSPs
16_ACC_INT _2CMP-MAX_UF	138.60	480	36897 (2%)	55376 (1%)	402 (3%)	355 (1%)	72 (1%)
16_ACC_INT _4CMP-MAX_UF	149.92	480	90372 (5%)	144165 (4%)	1529.6 (13%)	861 (1%)	107 (2%)

Tabla 4.7 Tiempo de procesamiento y recursos utilizados por el algoritmo ATDCA-GS usando la FPGA Intel Stratix 10 SX 2800 para la optimización de usar una única unidad funcional para realizar la proyección y búsqueda del máximo.

4.5. Resultados experimentales

En este apartado se presentan los resultados experimentales obtenidos tras la aplicación de diversas optimizaciones al algoritmo ATDCA-GS. Primero, se hará un repaso de las imágenes utilizadas y la FPGA seleccionada como plataforma de implementación, además de evaluar la precisión de los endmembers extraídos para validar la efectividad del algoritmo optimizado. A continuación, se analizará el rendimiento computacional de las optimizaciones propuestas, considerando métricas clave como el tiempo de procesamiento, el uso de recursos, la escalabilidad en función del tamaño de la imagen y otros aspectos relevantes. Finalmente, se llevará a cabo una comparación con otra implementación del ATDCA-GS desarrollada en hardware utilizando el lenguaje VHDL, evaluando sus ventajas y diferencias con la versión optimizada.

4.5.1. Imágenes hiperespectrales utilizadas

Nuevamente, se emplean las mismas imágenes hiperespectrales utilizadas en la evaluación del algoritmo VD, las cuales fueron descritas en detalle en la Sección 2.1.4. Estas imágenes corresponden a:

- **Escena AVIRIS Cuprite**. Capturada sobre la región minera de Cuprite, Nevada, expresada en unidades de reflectancia.

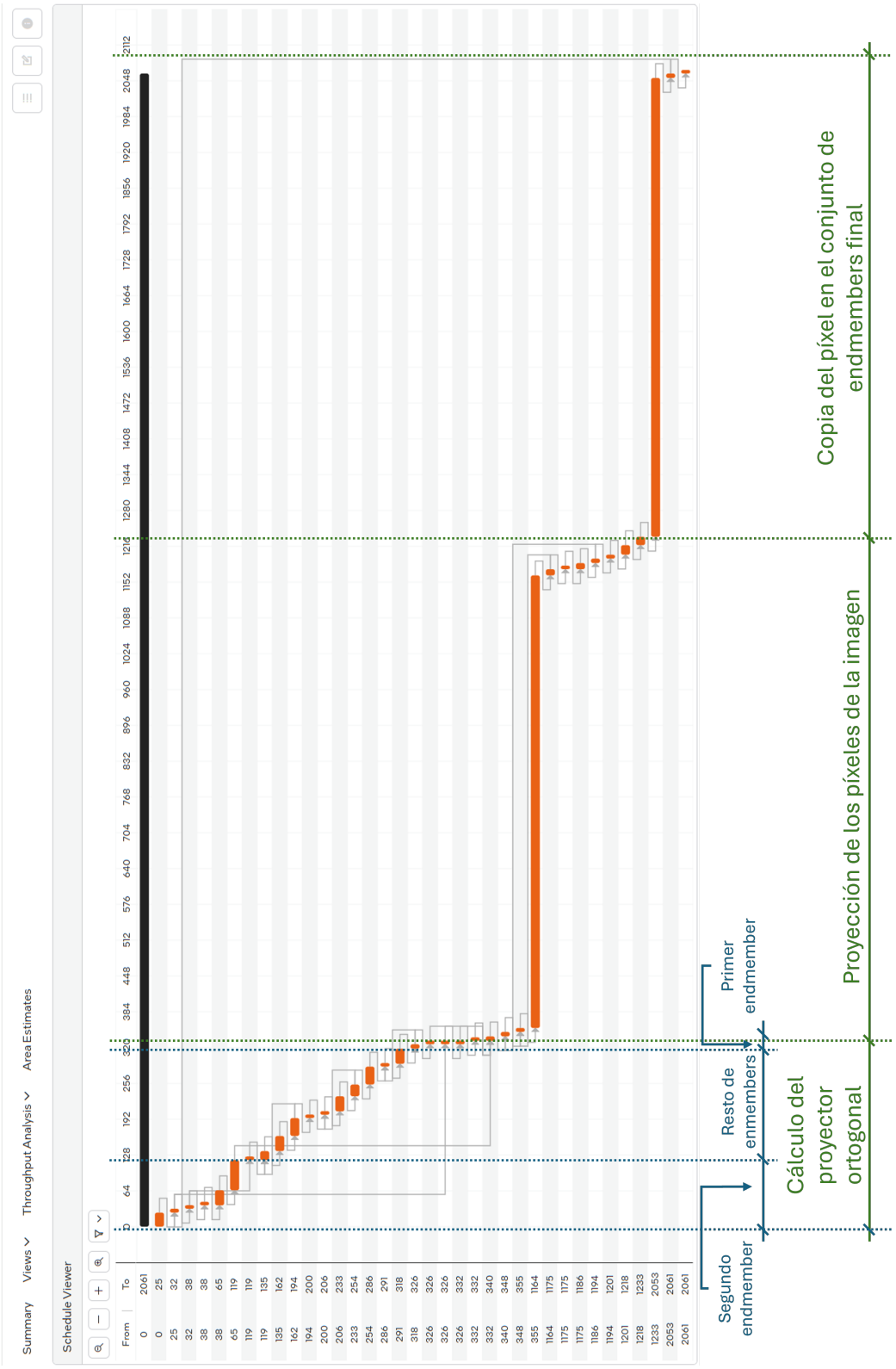


Figura 4.9 Diagrama temporal ciclo a ciclo de la versión con 16 acumuladores, con datos y operadores en formato entero, código reestructurado y 2 comparaciones paralelas.

- **Escena AVIRIS World Trade Center**. Registrada en Nueva York, cinco días después del atentado terrorista, también en unidades de reflectancia, con el objetivo de identificar focos de incendio en la zona afectada.
- **Escena sintética**. Diseñada para evaluar el rendimiento del algoritmo en imágenes de mayor tamaño, permitiendo analizar su escalabilidad y eficiencia computacional.

Estas imágenes proporcionan un conjunto de pruebas representativo para validar el desempeño del algoritmo en distintos escenarios y condiciones de análisis.

4.5.2. FPGA seleccionada

Para evaluar el rendimiento de las diferentes optimizaciones del algoritmo ATDCA-GS, se ha utilizado una de las FPGAs disponibles en el Intel Developer Cloud, concretamente la FPGA Intel Stratix 10 SX 2800, que fue descrita en detalle en la Sección 2.2.2. Esta placa ha sido fabricada con tecnología de 14 nm y cuenta con un total de 933.120 módulos lógicos adaptables (ALMs), 1.866.240 tablas de búsqueda adaptables (ALUTs) y 3.732.480 flip-flops (FFs). Además, dispone de una serie de recursos heterogéneos, incluyendo 5760 DSPs y 11721 bloques de memoria RAM, así como otros componentes adicionales que no han sido utilizados.

4.5.3. Evaluación de la precisión en ATDCA-GS

En este apartado se validará la precisión de los endmembers extraídos por la implementación optimizada del ATDCA-GS utilizando las dos escenas hiperespectrales reales descritas previamente. Para ello, se empleará una de las métricas más utilizadas en la comunidad científica: el ángulo espectral, también conocido como *Spectral Angle Distance* (SAD), cuya fórmula fue presentada en la Ecuación 4.4. Esta ecuación puede expresarse de manera expandida de la siguiente forma:

$$\text{SAD}\,(\mathbf{x}_i - \mathbf{x}_j) = \cos^{-1}\left(\frac{\mathbf{x}_i \cdot \mathbf{x}_j}{\|\mathbf{x}_i\|_2 \cdot \|\mathbf{x}_j\|_2}\right) = \cos^{-1}\left(\frac{\sum_{k=1}^{b} x_i^{(b)} \cdot x_j^{(b)}}{\sqrt{\sum_{k=1}^{b} x_i^{(b)^2}} \cdot \sqrt{\sum_{k=1}^{b} x_j^{(b)^2}}}\right), \quad (4.8)$$

donde $\mathbf{x}_i$ y $\mathbf{x}_j$ representan firmas espectrales asociadas a píxeles de una imagen hiperespectral $\mathbf{X}$ dentro de un espacio b-dimensional. Ambas firmas pueden expresarse de la forma $\mathbf{x}_i =$

$[x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(b)}]$ y $\mathbf{x}_j = [x_j^{(1)}, x_j^{(2)}, \ldots, x_j^{(b)}]$, donde los términos $x_i^{(k)}$ y $x_j^{(k)}$ corresponden al k-ésimo valor espectral de los píxeles $\mathbf{x}_i$ y $\mathbf{x}_j$, respectivamente, con $k \in \{1, 2, \ldots, b\}$.

Los valores de SAD suelen situarse en un rango entre 0 y $\frac{\pi}{2}$, dado que las firmas espectrales de una imagen suelen ser positivas, independientemente de si están expresadas en radiancia (radiación captada por el sensor) o en reflectancia (tras la corrección atmosférica aplicada a los datos).

La Tabla 4.8 muestra los resultados de SAD (en grados) obtenidos para la imagen AVIRIS Cuprite. Para ello, se llevó a cabo un proceso de emparejamiento en el que se identificaron las firmas de referencia del USGS más similares a los endmembers extraídos por la implementación optimizada en hardware. La Tabla 4.8 presenta los valores de SAD correspondientes al mejor emparejamiento para cada mineral. En este estudio, el número de endmembers a extraer se fijó en $t = 19$, valor determinado en la primera etapa del desmezclado espectral mediante el algoritmo VD.

Alunita	**Buddingtonita**	**Calcita**	**Caolinita**	**Moscovita**	**Media**
5.48°	4.08°	5.87°	11.14°	5.68°	6.45°

Tabla 4.8 Similaridad espectral entre los endmembers obtenidos por el algoritmo ATDCA-GS y las firmas espectrales de referencia USGS para los cinco minerales representativos en la imagen AVIRIS Cuprite.

Como se observa en la Tabla 4.8, los valores de SAD obtenidos son relativamente bajos, con un ángulo medio de aproximadamente 6° para los cinco materiales analizados. Recordemos que en esta métrica, un valor de 0° indica una coincidencia perfecta, mientras que 90° representa la máxima discrepancia. Estos resultados sugieren que la implementación optimizada del algoritmo logra identificar endmembers con un alto grado de similitud respecto a las firmas espectrales de referencia del USGS.

Además, se realizaron pruebas con la imagen AVIRIS WTC para detectar las ubicaciones de los focos activos de incendio. En este caso, el número de endmembers a extraer se estableció en $t = 30$, de acuerdo con el valor determinado por el algoritmo VD. En este escenario, el objetivo era evaluar la similitud entre los endmembers detectados y las firmas espectrales de posiciones previamente etiquetadas desde "A" hasta "H" (ver Figura 2.6). La Tabla 4.9 muestra los valores de SAD obtenidos en la comparación, donde se observa que los focos etiquetados como "A" y "C" fueron detectados correctamente, mientras que otros focos de menor tamaño presentaron mayores dificultades en su identificación.

En ambas escenas, los valores de SAD obtenidos han sido satisfactorios, mostrando una alta correlación con los resultados reportados en la literatura para implementaciones

A	B	C	D	E	F	G	H
0.00°	27.16°	0.00°	15.62°	27.81°	3.98°	2.72°	24.26°

Tabla 4.9 Similaridad espectral entre los endmembers obtenidos por el algoritmo ATDCA-GS y las firmas espectrales de las posiciones conocidas correspondientes a los focos con fuegos activos en la imagen AVIRIS WTC.

de software equivalentes [12]. Estos resultados respaldan la validez de la implementación optimizada en hardware y su capacidad para extraer endmembers representativos en diferentes escenarios.

4.5.4. Evaluación del rendimiento computacional en ATDCA-GS

En este apartado, se lleva a cabo una evaluación experimental del rendimiento computacional de la implementación de ATDCA-GS, teniendo en cuenta las optimizaciones aplicadas sobre la FPGA previamente descrita. Para cada una de las versiones optimizadas, se realizaron un total de 10 ejecuciones, cuyos resultados muestran una desviación estándar despreciable, lo que indica que los tiempos de ejecución son prácticamente idénticos entre sí.

La Tabla 4.10 presenta los tiempos de procesamiento y los recursos utilizados, obtenidos en la FPGA, para la mejor versión optimizada del algoritmo. En este caso, se aplicaron las tres optimizaciones mencionadas, partiendo de la versión *baseline* y considerando las diferentes escenas analizadas en este estudio. En cuanto al uso de recursos, se observa que la reestructuración del código contribuyó significativamente a mejorar el impacto de la optimización mediante el uso de 16 acumuladores, resultando en una mayor aceleración. Por otro lado, la conversión de datos y operadores a enteros también tuvo un efecto positivo sobre los resultados, aprovechando las ventajas del cálculo rápido, sin que se incrementara significativamente el uso de recursos en comparación con la versión *baseline*. Finalmente, la reducción en el número de DSPs utilizados en la versión optimizada del algoritmo no se debe únicamente a la paralelización del proceso de comparación para encontrar el valor máximo, sino también a la reestructuración del código mediante la unificación de unidades funcionales. Esta combinación de estrategias permitió disminuir significativamente el uso de recursos sin comprometer el rendimiento, alcanzando una aceleración superior a 18× respecto a la versión *baseline*.

Estos resultados sugieren que la versión HLS optimizada de nuestro algoritmo puede emplearse en entornos de tiempo real, donde se requiere procesar los datos a medida que son obtenidos por el sensor, como en el caso del sensor AVIRIS. Este sensor tiene un tiempo de

Implementaciones	Tiempos (ms)			F_{max}	Recursos				
	AVIRIS Cuprite	AVIRIS WTC	Sintética	(MHz)	ALUTs	FFs	RAMs	MLABs	DSPs
Baseline	2345.51	10799.90	16743.90	196.61	50482 (3%)	102527 (3%)	1267 (11%)	560 (1%)	61 (1%)
16_ACC_INT _2CMP-MAX_UF	138.60 (16.9×)	610.86 (17.7×)	920.20 (18.2×)	480	36897 (2%)	55376 (1%)	402 (3%)	355 (1%)	72 (1%)

Tabla 4.10 Tiempos de procesamiento y recursos utilizados por el algoritmo ATDCA-GS usando la FPGA Intel Stratix 10 SX 2800 para las versiones *baseline* y optimizada, generalizadas para un tamaño máximo de imagen.

barrido rápido, de solo 8.3 ms para capturar 512 muestras con 224 bandas espectrales. Como resultado, es necesario procesar las escenas en menos de 1.986 s (AVIRIS Cuprite), 5.09 s (AVIRIS WTC) y 7.903 s (Sintética), lo que se logra con nuestra implementación, que es capaz de escalar su rendimiento conforme aumenta el tamaño de la imagen de entrada.

Comparando estos resultados con otra implementación del algoritmo ATDCA-GS en hardware utilizando el lenguaje VHDL [69], los tiempos de ejecución para una FPGA Virtex-7 XC7VX690T fueron de 23.3 ms, 94.4 ms y 146.3 ms para las escenas AVIRIS Cuprite, AVIRIS WTC y la imagen sintética, respectivamente. En cambio, la implementación en HLS sobre una Stratix 10 SX 2800 funcionando a 480 MHz presentó tiempos de 38.60 ms, 610.86 ms y 920.20 ms para esas mismas escenas. Esto implica que la versión HLS es aproximadamente 1.66×, 6.47× y 6.29× más lenta, respectivamente.

Sin embargo, una comparación directa de tiempos absolutos puede resultar engañosa al tratarse de dispositivos diferentes. Por ello, se han estimado los ciclos de reloj requeridos por cada implementación. En HLS, la ejecución de las escenas AVIRIS Cuprite, AVIRIS WTC y sintética requiere aproximadamente 18,528 millones, 293,213 millones y 441,696 millones de ciclos, respectivamente. En cambio, la implementación VHDL, ejecutándose a 116 MHz, necesita solo 2,703 millones, 10,950 millones y 16,970 millones de ciclos para cada escena, lo que indica que la versión en VHDL es entre 6.8× y más de 26× más eficiente en términos de uso de ciclos de reloj.

Estos datos reflejan una ventaja clara en eficiencia computacional para la versión HDL, fruto del control exhaustivo sobre la arquitectura hardware. No obstante, es importante subrayar que la solución basada en HLS permite acelerar drásticamente el proceso de desarrollo y facilita la portabilidad entre plataformas. En este contexto, la propuesta optimizada en HLS sigue siendo una alternativa válida y robusta, especialmente en fases tempranas de diseño, validación o en entornos donde los recursos de ingeniería son limitados.

Capítulo 5

Image Space Reconstruction Algorithm (ISRA)

En este capítulo se presenta el ***Image Space Reconstruction Algorithm*** (ISRA), un algoritmo diseñado para estimar la proporción de cada endmember presente en los píxeles de una imagen hiperespectral. ISRA es uno de los métodos iterativos más reconocidos en el ámbito del desmezclado espectral, aunque también se aplica en diversas áreas, como la reconstrucción de imágenes en el análisis de tejidos en medicina o en monitorización ambiental. Estas aplicaciones permiten obtener un análisis más detallado de la composición química y las características de los materiales en la superficie terrestre. El algoritmo se basa en una clasificación supervisada, ya que se parte de un conjunto de endmembers previamente conocidos, los cuales funcionan como una de las entradas del método. Entre las principales ventajas de ISRA, destacan su simplicidad en la implementación y su rápida ejecución.

A lo largo de este capítulo, se examinarán las diversas variantes del ISRA propuestas en la literatura, evaluando sus ventajas e inconvenientes con el objetivo de seleccionar la opción más adecuada para su implementación en hardware. Se priorizará la eficiencia computacional, evitando operaciones costosas y maximizando el rendimiento en entornos de cómputo acelerado. Para ello, se desarrollarán diversas implementaciones paralelas optimizadas en FPGAs utilizando la plataforma Intel oneAPI y el lenguaje de programación DPC++. Este enfoque permitirá no solo mejorar el rendimiento del algoritmo con respecto a una versión secuencial base, sino también reducir los tiempos de desarrollo gracias al uso de un lenguaje de alto nivel, facilitando la programación y optimización del hardware.

5.1. Fundamentos del ISRA

El ***Image Space Reconstruction Algorithm*** (ISRA) [23] es un algoritmo ampliamente utilizado en la implementación de la etapa final del desmezclado espectral, destinado a resolver el problema de la estimación de abundancias. Entre las distintas aproximaciones propuestas en la literatura para abordar este problema, la mayoría sigue un modelo lineal de mezcla para estimar las fracciones de abundancia no negativas en los píxeles mezcla. Para este propósito, cada píxel puede ser modelado mediante la siguiente aproximación:

$$\mathbf{x}_j = \sum_{i=1}^{t} \mathbf{u}_i \cdot \Phi_i + \mathbf{w} = \mathbf{U} \cdot \Phi + \mathbf{w}, \tag{5.1}$$

donde la imagen hiperespectral con b bandas espectrales se denota como $\mathbf{X}$, en la cual cada píxel está representado por un vector $\mathbf{x}_j$=[x_{j1}, x_{j2},...,x_{jn}] $\in \mathbb{R}^b$, donde $\mathbb{R}$ denota el conjunto de números reales. La matriz de endmembers, $\mathbf{U} \in \mathbb{R}^{b \times t}$, contiene las firmas espectrales de los endmembers, siendo $\mathbf{u}_i$ la firma espectral del i-ésimo endmember. El vector de abundancias $\Phi \in \mathbb{R}^t$ contiene las fracciones de abundancia, $\mathbf{w}$ es un vector de ruido, y t representa el número de endmembers. De este modo, se puede resolver el problema de mezcla obteniendo una estimación de abundancias positivamente restringida, es decir, $\Phi_i \geq 0$ para $1 \leq i \leq t$, mediante la siguiente expresión iterativa:

$$\widehat{\Phi}_i^{k+1} = \widehat{\Phi}_i^k \left(\frac{\sum_{l=1}^{n} (u_{il} \cdot x_{jl})}{\sum_{l=1}^{n} (u_{il} \cdot u_{jl}) \cdot \widehat{\Phi}_i^k} \right), \tag{5.2}$$

donde las abundancias de los endmembers para el píxel $\mathbf{x}_j$ se estiman de manera iterativa. En este caso, las abundancias en la iteración $k+1$ para un endmember dado $\mathbf{u}_i$ dependerán de las abundancias estimadas para ese mismo endmember en la iteración k. En base a lo expuesto, la reconstrucción seguirá un proceso iterativo que comprende los siguientes pasos:

1. **Inicialización**: Se comienza la reconstrucción de la imagen mediante una aproximación inicial, que puede ser una estimación simplificada, como asignar una abundancia igual para todos los endmembers, o incluso mediante la proyección directa de los datos.

2. **Corrección espectral**: En cada iteración, el algoritmo ajusta la imagen a reconstruir para que se acerque progresivamente a los datos reales conforme avanza el proceso.

3. **Evaluación del error**: En cada paso de la reconstrucción, se calcula el error utilizando una métrica adecuada, como el error cuadrático medio normalizado (NMSE). Los parámetros del modelo se ajustan para minimizar dicho error en la iteración siguiente.

4. **Convergencia**: El proceso continúa hasta que el error se reduce lo suficiente o se alcanza el número máximo de iteraciones establecido.

La Figura 5.1 ilustra el flujo del algoritmo ISRA, que sigue un proceso iterativo para la reconstrucción de una imagen hiperespectral. El proceso comienza con una estimación inicial de las abundancias, que puede basarse en una asignación uniforme o en una proyección directa de los datos. A continuación, en cada iteración, se aplica una corrección espectral para ajustar progresivamente la imagen reconstruida a los datos reales. Para evaluar la calidad de la reconstrucción, se calcula el error utilizando métricas como el NMSE, permitiendo ajustar los parámetros del modelo en cada paso. Este procedimiento se repite hasta que el error se reduce lo suficiente o se alcanza el número máximo de iteraciones, garantizando así la convergencia del algoritmo.

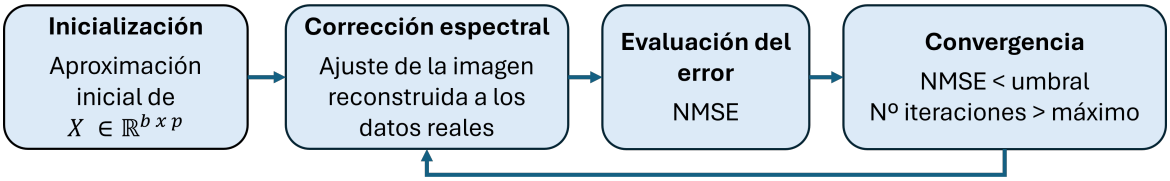


Figura 5.1 Esquema del proceso del ISRA.

5.2. Descripción del algoritmo ISRA

Como se mencionó anteriormente, el algoritmo ISRA es considerado uno de los métodos iterativos más empleados para estimar la proporción de cada endmember presente en los píxeles de una escena hiperespectral. Para realizar esta estimación, es posible abordarlo como un problema de minimización de distancia. Entre las distancias más comunes utilizadas en la literatura, destaca la de mínimos cuadrados:

$$MC(U\Phi, x) = ||U\Phi - x||^2 = \sqrt{\sum_{i=1}^{n} (U_i\Phi_i - x_i)^2} \tag{5.3}$$

A la ecuación anterior se debe imponer la restricción de que las abundancias sean no negativas, ya que, de lo contrario, podrían surgir valores negativos en dichas abundancias,

lo cual carecería de sentido físico. Por lo tanto, este problema de mínimos cuadrados con la restricción impuesta puede resolverse mediante un algoritmo iterativo cuya base es la siguiente:

$$\widehat{\Phi}_i^{k+1} = \widehat{\Phi}_i^k \left(\frac{\sum_{j=1}^{b} (x_j \cdot u_{ji})}{\sum_{j=1}^{b} (u_{ji} \cdot u_j^T) \cdot \widehat{\Phi}} \right), \tag{5.4}$$

donde u_{ji} es un elemento de $\mathbf{U}$ y u_j es el vector de la respuesta espectral de un endmember en todas las bandas j. El término x representa el píxel que se está procesando y $\widehat{\Phi}$ es el vector de abundancias. Para facilitar la comprensión de esta solución iterativa, a continuación, se describen los pasos del algoritmo, los cuales están representados en el Algoritmo 4.

Algoritmo 4 Pseudocódigo del algoritmo ISRA

```
1: Entrada: X ∈ ℝ^{p×b}, U ∈ ℝ^{b×t} y MAX_ITER
2: for i = 1 to p do
3:    for l = 1 to MAX_ITER do
4:       for j = 1 to t do
5:          for k = 1 to b do
6:             num = num + U[k][j]*X[k][i];
7:             for s = 1 to t do
8:                dot = dot + U[k][s]*Φ[s][i];
9:             end for
10:            den = den + dot*U[k][j];
11:            dot=0;
12:         end for
13:         Φ[j][i] = Φ[j][i]*(num/dem);
14:         num=0;
15:         den=0;
16:      end for
17:   end for
18: end for
19: Salida: Φ ∈ ℝ^{p×t};
```

Las entradas de este algoritmo están compuestas por la imagen hiperespectral, denotada como $\mathbf{X} \in \mathbb{R}^{p \times b}$, la matriz de endmembers $\mathbf{U} \in \mathbb{R}^{b \times t}$ y el número máximo de iteraciones por píxel, especificado como MAX_ITER. La ecuación previamente mencionada se descompone en los cálculos correspondientes al numerador y al denominador, para luego dividir y multiplicar por la abundancia anterior, $\Phi \in \mathbb{R}^{p \times t}$.

5.3. Implementación HLS del algoritmo ISRA

Para lograr una implementación eficiente de un algoritmo mediante HLS en FPGAs, es fundamental llevar a cabo un perfilado del código. Este proceso permite analizar en detalle la ejecución del algoritmo, identificando las secciones que presentan mayor consumo computacional y determinando los cuellos de botella que limitan el rendimiento. A partir de este análisis, es posible seleccionar estratégicamente las regiones del código más adecuadas para ser paralelizadas y aceleradas, optimizando así el uso de los recursos del hardware. De esta manera, se maximiza la eficiencia computacional, se reduce el tiempo de ejecución y se mejora el aprovechamiento de la arquitectura FPGA, asegurando un diseño equilibrado entre rendimiento, consumo energético y utilización de recursos.

Salvo por la evaluación de la condición que verifica si se ha alcanzado el número máximo de iteraciones, el código corresponde íntegramente a la fase de corrección espectral. Dado que esta se encuentra dentro de un bucle anidado cuya estructura es sencilla de analizar, no resulta necesario realizar un perfilado detallado. En el caso del algoritmo ISRA, la aceleración se logrará mediante la optimización individual de cada unidad de procesamiento, procurando que cada una ocupe la menor cantidad de hardware posible, ya que la mejora principal se obtendrá a través de la paralelización de dichas unidades.

5.3.1. Versión base (*baseline*) del ISRA

Para llevar a cabo las distintas optimizaciones en la implementación ISRA, es imprescindible contar con una versión base que sirva como referencia para futuras comparaciones. Esta versión presentará mínimas modificaciones con respecto a una implementación secuencial en C++ y SYCL, la cual será compilada utilizando el compilador DPC++. Para establecer esta versión base, es necesario incorporar los siguientes elementos (indicar que el elemento [0] es optativo pero aconsejable para disponer de un código más legible), tal y como se muestra en el Código 5.1:

[0] Aislar el código en una función donde se realice la llamada al *kernel* que se ejecutará en la placa FPGA, dejando la lectura de la imagen y la creación de los buffers de entrada/salida fuera de esta función.

[1] Definir un selector de dispositivo dirigido al uso de FPGAs.

[2] Crear un DPC++ dispositivo cola (*queue*) usando el selector anterior.

[3] Utilizar buffers para los arrays de entrada y salida de datos de nuestra implementación. En este caso, utilizaremos dos buffers de entrada para la imagen hiperespectral y para la matriz de endmembers, así como un buffer de salida para almacenar las fracciones de abundancia.

[4] Uso de *submit* para ejecutar un conjunto de comandos en el dispositivo objetivo.

[5] Utilizar accesores para los diferentes buffers creados, permitiendo que la FPGA acceda a los datos.

[6] Uso de *single_task* para enviar el *kernel* a ejecución. El contenido de *buffer_abundances* será copiado a *h_abundances* cuando la función termine.

Código 5.1 Estructura de la versión base para el algoritmo ISRA.

```
1  void ISRA(const std::vector<float> &h_image, const std::vector<float>
       &h_endmembers, std::vector<float> &h_abundances, int samples, int
       lines, int bands, int num_endmembers, int num_iters)
2      #if defined(FPGA_EMULATOR)
3          ext::intel::fpga_emulator_selector device_selector;
4      #else
5          [1] ext::intel::fpga_selector device_selector;
6      #endif
7
8      try {
9          [2] queue q(device_selector, dpc_common::exception_handler,
10             property::queue::enable_profiling{});
11         [3] buffer buffer_image(h_image);
12         [3] buffer buffer_end(h_end);
13         [3] buffer buffer_abundances(h_abundances);
14
15         [4] event e = q.submit([&](handler &h) {
16             [5] accessor acc_image(buffer_image,h,read_only);
17             [5] accessor acc_endmembers(buffer_end,h,read_only);
18             [5] accessor acc_abundances(buffer_abundances,h,
                   read_write,no_init);
19
20             [6] h.single_task([=]()
21             [[intel::kernel_args_restrict]] {
22                 // Código del kernel ISRA a ejecutar en FPGA
23             });
24         });
25     }
```

```
26      catch (sycl::exception const &e) {
27          // Detección de excepciones en el código del host
28      }
29  }
```

Dentro de la *single task* se encuentra el siguiente Código 5.2 para la implementación del algoritmo ISRA.

Código 5.2 Fragmento de código dentro de la single task para el algoritmo ISRA.

```
1  int num_pixels = samples * lines;
2  int k, i, j, p, s;
3  float num = 0.0;
4  float den = 0.0;
5  float dot = 0.0;
6
7  // Para cad píxel
8  for (p = 0; p < num_pixels; p++) {
9
10      // Inicializar las abundancias al mismo porcentaje
11     for (k = 0; k < num_endmembers; k++)
12         acc_abundances[p * num_endmembers + k] = 1.0 / num_endmembers
               ;
13
14     // Iteraciones
15     for (k = 0; k < num_iters; k++) {
16         // Para cada endmember
17          for (j = 0; j < num_endmembers; j++) {
18             // Para todas las bandas
19             for (i = 0; i < bands; i++) {
20                 num +=  acc_endmembers[j * bands + i] * acc_image[p *
                       bands + i];
21
22                 for (s = 0; s < num_endmembers; s++)
23                     dot += acc_endmembers[s * bands + i] *
                           acc_abundances[p * num_endmembers + s];
24
25                 den += dot * acc_endmembers[j * bands + i];
26                 dot = 0.0;
27             }
28
29             acc_abundances[p * num_endmembers + j] *= (num / den);
30             num = 0.0;
31             den = 0.0;
```

```
32                }
33            }
34        }
```

Dado que los accesos a memoria son secuenciales y predecibles, el uso de técnicas como la partición o la distribución en bancos de memoria podría no aportar mejoras significativas e incluso reducir el rendimiento. Al no producirse conflictos de acceso y tratarse de un acceso lineal, el diseño ya se beneficia del comportamiento eficiente de la caché por defecto. El uso de directivas como [intel::partition] e [intel::numbanks(N)], están más orientadas a situaciones donde existen múltiples accesos paralelos o potenciales conflictos que podrían degradar el rendimiento. Si en el futuro se modifica el patrón de acceso o se busca un mayor paralelismo en el kernel, sí sería recomendable considerarlas.

Los accesos a memoria se gestionan mediante objetos *buffer* y *accessor*, lo que permite al entorno SYCL construir un grafo de ejecución basado en dependencias de datos. Esto asegura un modelo de sincronismo automático entre kernels, donde la ejecución se programa en función de la disponibilidad de los datos requeridos. Posteriormente, se han utilizado colas *in-order* para operaciones secuenciales, y mecanismos explícitos como *event.wait()* para forzar sincronización en situaciones críticas. Este enfoque se alinea con el modelo de ejecución de SYCL, evitando condiciones de carrera sin necesidad de sincronización manual.

Con los cambios previamente mencionados, se cuenta con un código *baseline* a partir del cual podremos generar un informe destinado a su optimización. Esta versión no incorpora ninguna técnica de optimización, ya que más adelante se estudiará la viabilidad de implementar posibles mejoras sobre esta versión base. Dicho informe se elabora tras una compilación rápida, lo que permite obtener una serie de recomendaciones sobre cómo modificar los *kernels* para mejorar el rendimiento. Además, proporciona información crucial para el desarrollo, tales como: vistas de las estructuras (mapeo del código sobre los recursos de la FPGA), diseño de la memoria, análisis de rendimiento, identificación de cuellos de botella (áreas del código donde el rendimiento se ve limitado) y estimaciones de los recursos consumidos por el diseño.

El informe generado proporciona información clave para el análisis del diseño, siendo especialmente útil la vista de planificación, que detalla la planificación estática ciclo a ciclo y la latencia de los grupos de instrucciones, permitiendo examinar el comportamiento del sistema en FPGA y optimizar su rendimiento. En la Figura 5.2, se muestra el diagrama de planificación correspondiente a la versión *baseline*, donde se observa claramente que los mayores tiempos de ejecución se deben a los accesos a memoria externa. En el siguiente

apartado, se analizarán en detalle estos procesos y su impacto en el rendimiento y eficiencia de la implementación.

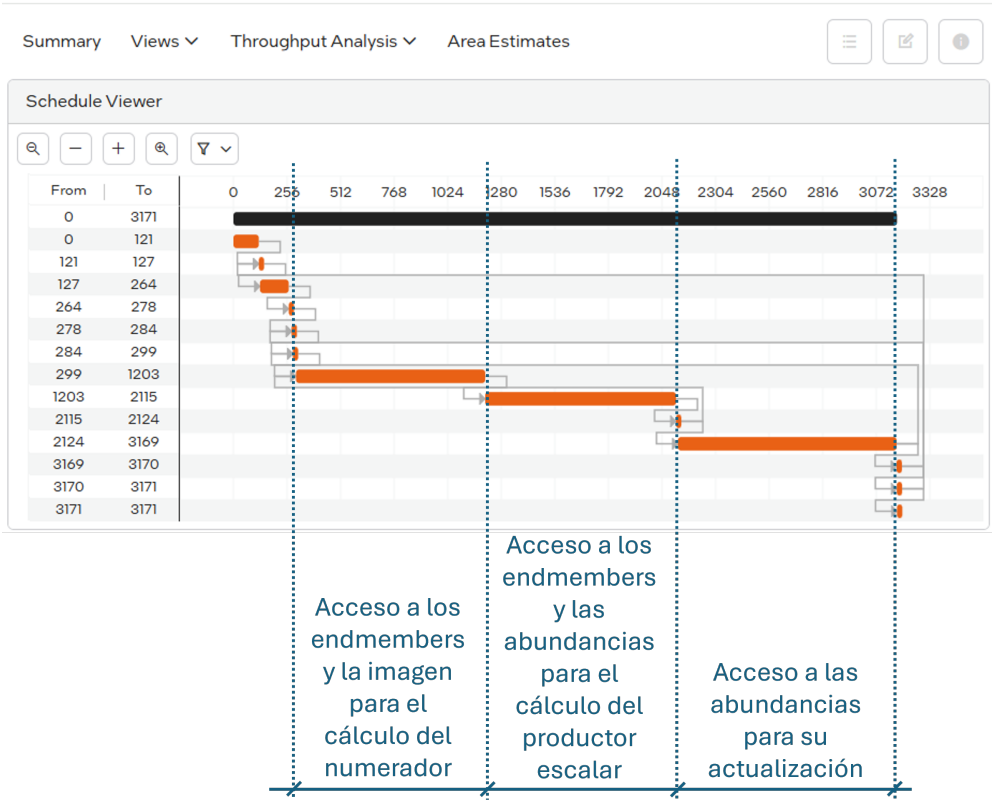


Figura 5.2 Diagrama temporal ciclo a ciclo de la versión ISRA baseline.

Antes de abordar las primeras optimizaciones, es fundamental analizar el rendimiento actual para cuantificar con precisión las mejoras introducidas. Para ello, las mediciones del tiempo de ejecución se han realizado mediante funciones de una clase de perfilado de SYCL, tal y como se muestra en el Código 5.3:

Código 5.3 Funciones para realizar el perfilado en el algoritmo ISRA.

```
1 get_profiling_info<info::event_profiling::command_start>
2 get_profiling_info<info::event_profiling::command_end>
```

En este perfilado, no se consideran las transferencias de datos entre el *host* y el dispositivo *"offload"*, ya que el enfoque está centrado en medir el tiempo de ejecución de un *kernel*. Es importante señalar que los tiempos así obtenidos se corresponden con ejecuciones reales sobre el dispositivo hardware (en este caso, una FPGA Intel Stratix 10 SX 2800), y se miden una vez completado el flujo completo de síntesis e implementación. Por tanto, no se trata de estimaciones previas ni simulaciones, sino de datos empíricos en entorno real. No obstante,

estos tiempos reflejan exclusivamente la ejecución del *kernel* en placa, y no incluyen fases previas como la síntesis, *place & route* o generación del bitstream. Para la versión *baseline*, se presentan los tiempos obtenidos en hardware para un total de 10 iteraciones, así como los recursos utilizados por esta versión en las diferentes escenas, los cuales se detallan en la Tabla 5.1.

Implementaciones	Tiempos (ms)			F_{max}	Recursos				
	AVIRIS Cuprite	AVIRIS WTC	Sintética	(MHz)	ALUTs	FFs	RAMs	MLABs	DSPs
Baseline	1014650	7208060	11177757	480	19537 (1%)	40062 (1%)	391 (3%)	272 (<1%)	28 (<1%)

Tabla 5.1 Tiempo de procesamiento (para 10 iteraciones) y recursos utilizados por el algoritmo ISRA usando la FPGA Intel Stratix 10 SX 2800 para la versión baseline.

5.3.2. Versión optimizada del ISRA

Al analizar detenidamente el diagrama temporal de la versión *baseline* (ver Figura 5.2) se observó claramente que los mayores tiempos de ejecución se deben a los accesos a memoria externa. En la primera fase, se realiza el acceso a los endmembers y a la imagen para el cálculo del numerador (línea 20); en la segunda, se accede a los endmembers y a las abundancias para el cálculo del producto escalar (línea 23); y, finalmente, en la tercera fase, se accede a las abundancias para su actualización (línea 29). Además, es importante señalar que los dos primeros accesos a memoria se realizan dentro de bucles anidados de cuatro y cinco niveles, respectivamente, lo que implica que se ejecutarán un gran número de veces, impactando significativamente en el rendimiento del algoritmo.

Por tanto, la primera modificación consistió en la creación de tres memorias internas en la FPGA: una para almacenar los endmembers, otra para el píxel actual del que se están calculando las abundancias y una tercera para guardar dichas abundancias durante los cálculos. De este modo, se accede a la memoria externa una única vez para leer los endmembers, una vez por píxel para su lectura y una última vez por píxel para escribir el resultado final de las abundancias calculadas. Esta estrategia minimiza los accesos a memoria externa, mejorando significativamente la eficiencia del algoritmo.

La siguiente modificación consistió en optimizar la operación de acumulación de la línea 23 mediante el uso de 16 acumuladores, de manera similar a la estrategia empleada en el algoritmo ATDCA. Esta mejora permitió una mayor paralelización de los cálculos, reduciendo significativamente el tiempo de ejecución.

A continuación, se presenta el Código 5.4 optimizado del *kernel*, el cual integra todas las técnicas mencionadas, incluyendo el uso de memoria interna para minimizar accesos a memoria externa y la implementación de múltiples acumuladores para mejorar la eficiencia computacional.

Código 5.4 Código optimizado del kernel en el algoritmo ISRA.

```
1  float original_abundance = 1.0f / num_endmembers;
2
3  // Leer los endmembers
4  unsigned short endmembers[BANDS * ENDMEMBERS];
5  for (int k = 0; k < num_endmembers; k++)
6      for (int i = 0; i < bands; i++)
7          endmembers[k * bands + i] = acc_endmembers[k * bands + i];
8
9  for (int k = num_endmembers; k < ENDMEMBERS; k++)
10     for (int i = 0; i < bands; i++)
11         endmembers[k * bands + i] = 0;
12
13 // Para cada píxel
14 for (int p = 0; p < pixels_per_task; p++) {
15
16     // Leer el píxel actual
17     unsigned short pixel[BANDS];
18     for (int k = 0; k < bands; k++)
19         pixel[k] = acc_image[p * bands + k];
20
21     // Inicializar las abundancias con el mismo porcentaje
22     float abundances[ENDMEMBERS];
23     for (int k = 0; k < num_endmembers; k++)
24         abundances[k] = original_abundance;
25
26     // Iteraciones
27     for (int k = 0; k < num_iters; k++) {
28
29         // Para cada píxel
30         for (int j = 0; j < num_endmembers; j++) {
31             float num = 0.0f;
32             float den = 0.0f;
33
34             // Para todas las bandas
35             for (int i = 0; i < bands; i++) {
36
37                 float acc_01 = 0.0f;
```

```
38                  float acc_02 = 0.0f;
39                  float acc_03 = 0.0f;
40                  float acc_04 = 0.0f;
41                  float acc_05 = 0.0f;
42                  float acc_06 = 0.0f;
43                  float acc_07 = 0.0f;
44                  float acc_08 = 0.0f;
45                  float acc_09 = 0.0f;
46                  float acc_10 = 0.0f;
47                  float acc_11 = 0.0f;
48                  float acc_12 = 0.0f;
49                  float acc_13 = 0.0f;
50                  float acc_14 = 0.0f;
51                  float acc_15 = 0.0f;
52                  float acc_16 = 0.0f;
53
54                 num +=  endmembers[j * bands + i] * pixel[i];
55
56                 for (int s = 0; s < num_endmembers; s += 16) {
57
58                     acc_01 += endmembers[s * bands + i  ] *
                           abundances[s];
59                     acc_02 += endmembers[(s + 1) * bands + i] *
                           abundances[s + 1];
60                     acc_03 += endmembers[(s + 2) * bands + i] *
                           abundances[s + 2];
61                     acc_04 += endmembers[(s + 3) * bands + i] *
                           abundances[s + 3];
62                     acc_05 += endmembers[(s + 4) * bands + i] *
                           abundances[s + 4];
63                     acc_06 += endmembers[(s + 5) * bands + i] *
                           abundances[s + 5];
64                     acc_07 += endmembers[(s + 6) * bands + i] *
                           abundances[s + 6];
65                     acc_08 += endmembers[(s + 7) * bands + i] *
                           abundances[s + 7];
66                     acc_09 += endmembers[(s + 8) * bands + i] *
                           abundances[s + 8];
67                     acc_10 += endmembers[(s + 9) * bands + i] *
                           abundances[s + 9];
68                     acc_11 += endmembers[(s + 10) * bands + i] *
                           abundances[s + 10];
69                     acc_12 += endmembers[(s + 11) * bands + i] *
                           abundances[s + 11];
```

```
70                      acc_13 += endmembers[(s + 12) * bands + i] *
                            abundances[s + 12];
71                      acc_14 += endmembers[(s + 13) * bands + i] *
                            abundances[s + 13];
72                      acc_15 += endmembers[(s + 14) * bands + i] *
                            abundances[s + 14];
73                      acc_16 += endmembers[(s + 15) * bands + i] *
                            abundances[s + 15];
74
75                  }
76                  float dot = (((acc_01 + acc_02) + (acc_03 + acc_04))
                        + ((acc_05 + acc_06) + (acc_07 + acc_08))) + (((
                        acc_09 + acc_10) + (acc_11 + acc_12)) + ((acc_13 +
                         acc_14) + (acc_15 + acc_16)));
77
78                  den += dot * endmembers[j * bands + i];
79              }
80
81              abundances[j] *= (num / den);
82          }
83      }
84
85      // Escribe las abundancias de ese píxel
86      for (int k = 0; k < num_endmembers; k++)
87          acc_abundances[p * num_endmembers + k] = abundances[k];
88
89  }
```

La Figura 5.3 muestra el diagrama temporal ciclo a ciclo de la versión optimizada del algoritmo ISRA. Se puede observar cómo el acceso a los endmembers se ha desplazado fuera del bucle principal, lo que reduce significativamente los accesos a memoria externa. Además, el número de ciclos por iteración se ha reducido considerablemente, evidenciando la mejora en eficiencia y rendimiento gracias a las optimizaciones implementadas.

Finalmente, una vez optimizada una unidad funcional, se replicó esta misma estructura para ejecutar varias unidades en paralelo, permitiendo el cálculo simultáneo de las abundancias para distintos píxeles. A continuación, se muestra el fragmento de Código 5.5 utilizado para implementar múltiples *single tasks* en paralelo, tomando como ejemplo la ejecución con dos unidades funcionales.

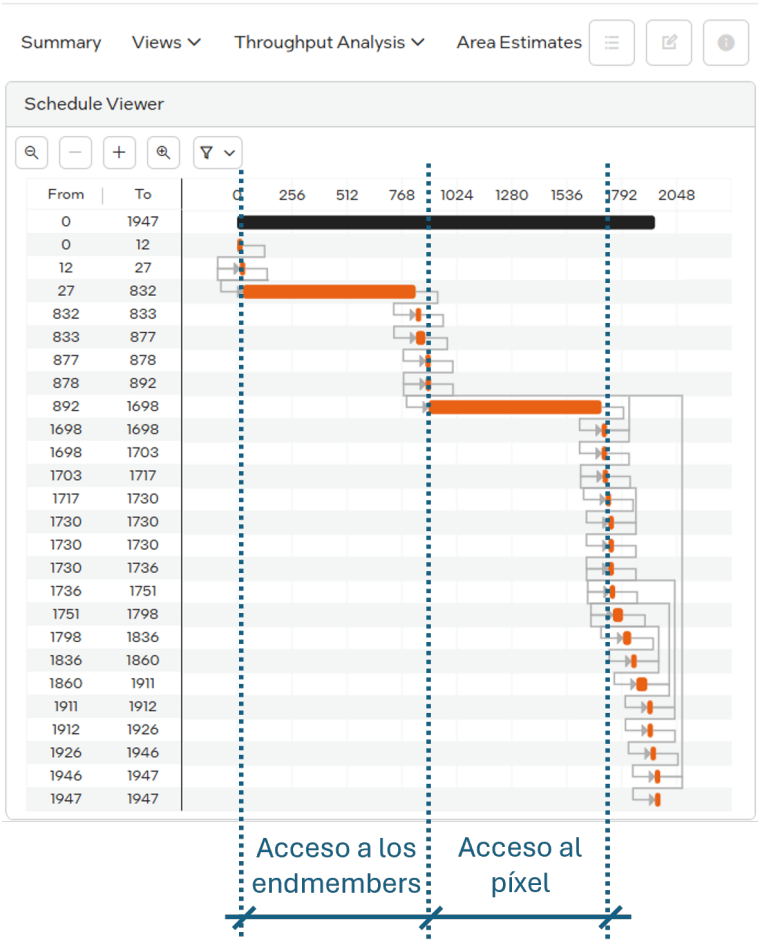


Figura 5.3 Diagrama temporal ciclo a ciclo de la versión ISRA optimizada.

Código 5.5 Código con la implementación de múltiples single tasks en paralelo en el algoritmo ISRA.

```
1  void ISRA(const std::vector<unsigned short> &h_image, const std::
       vector<unsigned short> &h_endmembers, std::vector<float> &
       h_abundances, int samples, int lines, int bands, int
       num_endmembers, int num_iters) {
2      #if defined(FPGA_EMULATOR)
3      ext::intel::fpga_emulator_selector device_selector;
4      #else
5      ext::intel::fpga_selector device_selector;
6      #endif
7
8      try {
9          queue q01(device_selector, fpga_tools::exception_handler,
               property::queue::enable_profiling{});
10         queue q02(device_selector, fpga_tools::exception_handler,
               property::queue::enable_profiling{});
```

```
11
12          // Dividir los píxeles en dos mitades
13          const size_t num_pixels = samples * lines;     // Número de pí
                xeles de la imagen
14          const size_t pixels_per_task = num_pixels / num_tasks; // Nú
                mero de píxeles por task
15          const size_t leftover_pixels = num_pixels - pixels_per_task;
                // Píxeles restantes
16
17          buffer buffer_image01(h_image.data(), range<1>(
                pixels_per_task * bands));
18          buffer buffer_image02(h_image.data() + (pixels_per_task *
                bands), range<1>(leftover_pixels * bands));
19
20          buffer buffer_endmembers(h_endmembers);
21
22          buffer buffer_abundances01(h_abundances.data(), range<1>(
                pixels_per_task * num_endmembers));
23          buffer buffer_abundances02(h_abundances.data() + (
                pixels_per_task * num_endmembers), range<1>(
                leftover_pixels * num_endmembers));
24
25          auto start_todo = std::chrono::high_resolution_clock::now();
26
27          event e01 = q01.submit([&](handler &h) {
28              accessor acc_image(buffer_image01, h, read_only);
29              accessor acc_endmembers(buffer_endmembers, h, read_only);
30              accessor acc_abundances(buffer_abundances01, h,
                    write_only, no_init);
31
32              // Ejecutar el kernel como una única tarea que procesa la
                     primera mitad de los píxeles
33              h.single_task<class single_task_1>([=]() [[intel::
                    kernel_args_restrict, intel::max_concurrency(2)]] {
34                  // Código del kernel ISRA a ejecutar en FPGA
35              });
36          });
37
38
39
40          event e02 = q02.submit([&](handler &h) {
41              accessor acc_image(buffer_image02, h, read_only);
42              accessor acc_endmembers(buffer_endmembers, h, read_only);
```

```
43              accessor acc_abundances(buffer_abundances02, h,
                    write_only, no_init);
44
45              // Ejecutar el kernel como una única tarea que procesa la
                     segunda mitad de los píxeles
46              h.single_task<class single_task_2>([=]() [[intel::
                    kernel_args_restrict, intel::max_concurrency(2)]] {
47                  // Código del kernel ISRA a ejecutar en FPGA
48              });
49          });
50
51      // Esperar que ambas tareas terminen
52      q01.wait();
53      q02.wait();
54
55      auto end_todo = std::chrono::high_resolution_clock::now();
56
57      double start = e01.get_profiling_info < info::event_profiling::
            command_start > ();
58      double end = e01.get_profiling_info < info::event_profiling::
            command_end > ();
59      // Convertir de nanosegundos a milisegundos
60      double kernel_time = (double)(end - start) * 1e-6;
61      std::cout << "First half of pixels time : " << kernel_time << "
            ms \n";
62
63      start = e02.get_profiling_info < info::event_profiling::
            command_start > ();
64      end = e02.get_profiling_info < info::event_profiling::command_end
            > ();
65      // Convertir de nanosegundos a milisegundos
66      kernel_time = (double)(end - start) * 1e-6;
67      std::cout << "Remaining pixels time : " << kernel_time << " ms \n
            ";
68
69      std::chrono::duration<double> elapsed = end_todo - start_todo;
70      std::cout << "Tiempo total de ejecución: " << elapsed.count() <<
            " s" << std::endl;
71
72  }
```

Se realizaron pruebas con diferentes cantidades de unidades en paralelo para analizar la evolución del uso de recursos y el tiempo de ejecución, evaluando el equilibrio entre

eficiencia y consumo de hardware. La Tabla 5.2 muestra el resultado de dicho estudio. A partir de 16 unidades, los resultados no mejoran porque se ha alcanzado el límite de tareas que el *scheduler* es capaz de gestionar simultáneamente. Como consecuencia, en lugar de ejecutarse las 16 tareas en paralelo, el *scheduler* organiza su ejecución en dos grupos de 8 tareas secuenciales. Esta gestión introduce un retardo adicional debido a la planificación y cambio de contexto entre los grupos de tareas, lo que explica que el tiempo de ejecución con 16 tareas sea mayor que con 8.

Implementaciones	**Tiempos (ms)**			**F_{max}**	**Recursos**				
	AVIRIS Cuprite	**AVIRIS WTC**	**Sintética**	**(MHz)**	**ALUTs**	**FFs**	**RAMs**	**MLABs**	**DSPs**
ISRA_OPT_2Tasks	56640.8	332664	515855	480	64751 (3%)	112621 (3%)	643 (5%)	1450 (2%)	108 (2%)
ISRA_OPT_4Tasks	28148.6	172871	268057	480	128075 (7%)	224192 (6%)	1163 (10%)	2900 (3%)	216 (4%)
ISRA_OPT_8Tasks	14720.1	90267.6	139978	480	254835 (14%)	448109 (12%)	2203 (19%)	5800 (6%)	432 (8%)
ISRA_OPT_16Tasks	17201.5	105518	163644	480	508291 (27%)	895599 (24%)	4283 (37%)	11600 (12%)	864 (15%)

Tabla 5.2 Tiempo de procesamiento (para 10 iteraciones) y recursos utilizados por el algoritmo ISRA usando la FPGA Intel Stratix 10 SX 2800 para la versión optimizada.

5.4. Resultados experimentales

En este apartado se presentan los resultados experimentales obtenidos tras la aplicación de diversas optimizaciones al algoritmo ISRA. Primero, se hará un repaso de las imágenes utilizadas y la FPGA seleccionada como plataforma de implementación, además de evaluar la reconstrucción iterativa de cada una de las escenas hiperespectrales para poder validar la efectividad del algoritmo optimizado usando la métrica NMSE. A continuación, se analizará el rendimiento computacional de las optimizaciones propuestas, considerando métricas clave como el tiempo de procesamiento, el uso de recursos, la escalabilidad en función del tamaño de la imagen y otros aspectos relevantes. Finalmente, se llevará a cabo una comparación con otra implementación del ISRA desarrollada en hardware utilizando el lenguaje VHDL, evaluando sus ventajas y diferencias con la versión optimizada.

5.4.1. Imágenes hiperespectrales utilizadas

Nuevamente, se emplean las mismas imágenes hiperespectrales utilizadas en la evaluación de los algoritmos VD y ATDCA, las cuales fueron descritas en detalle en la Sección 2.1.4. Estas imágenes corresponden a:

- **Escena AVIRIS Cuprite**. Capturada sobre la región minera de Cuprite, Nevada, expresada en unidades de reflectancia.
- **Escena AVIRIS World Trade Center**. Registrada en Nueva York, cinco días después del atentado terrorista, también en unidades de reflectancia, con el objetivo de identificar focos de incendio en la zona afectada.
- **Escena sintética**. Diseñada para evaluar el rendimiento del algoritmo en imágenes de mayor tamaño, permitiendo analizar su escalabilidad y eficiencia computacional.

Estas imágenes proporcionan un conjunto de pruebas representativo para validar el desempeño del algoritmo en distintos escenarios y condiciones de análisis.

5.4.2. FPGA seleccionada

Para evaluar el rendimiento de las diferentes optimizaciones del algoritmo ISRA, se ha utilizado una de las FPGAs disponibles en el Intel Developer Cloud, concretamente la FPGA Intel Stratix 10 SX 2800 que fue descrita en detalle en la Sección 2.2.2. Esta placa ha sido fabricada con tecnología de 14 nm y cuenta con un total de 933.120 módulos lógicos adaptables (ALMs), 1.866.240 tablas de búsqueda adaptables (ALUTs) y 3.732.480 flip-flops (FFs). Además, dispone de una serie de recursos heterogéneos, incluyendo 5760 DSPs y 11721 bloques de memoria RAM, así como otros componentes adicionales que no han sido utilizados.

5.4.3. Evaluación de la precisión en ISRA

En este apartado se validará la precisión de las fracciones de abundancia estimadas para reconstruir una imagen hiperespectral a partir de los endmembers ya conocidos, y se estudiará la convergencia del algoritmo. De esta manera, se podrá determinar un número adecuado de iteraciones según la escena analizada. Es importante destacar que la versión hardware propuesta produce resultados idénticos a los de una implementación de software

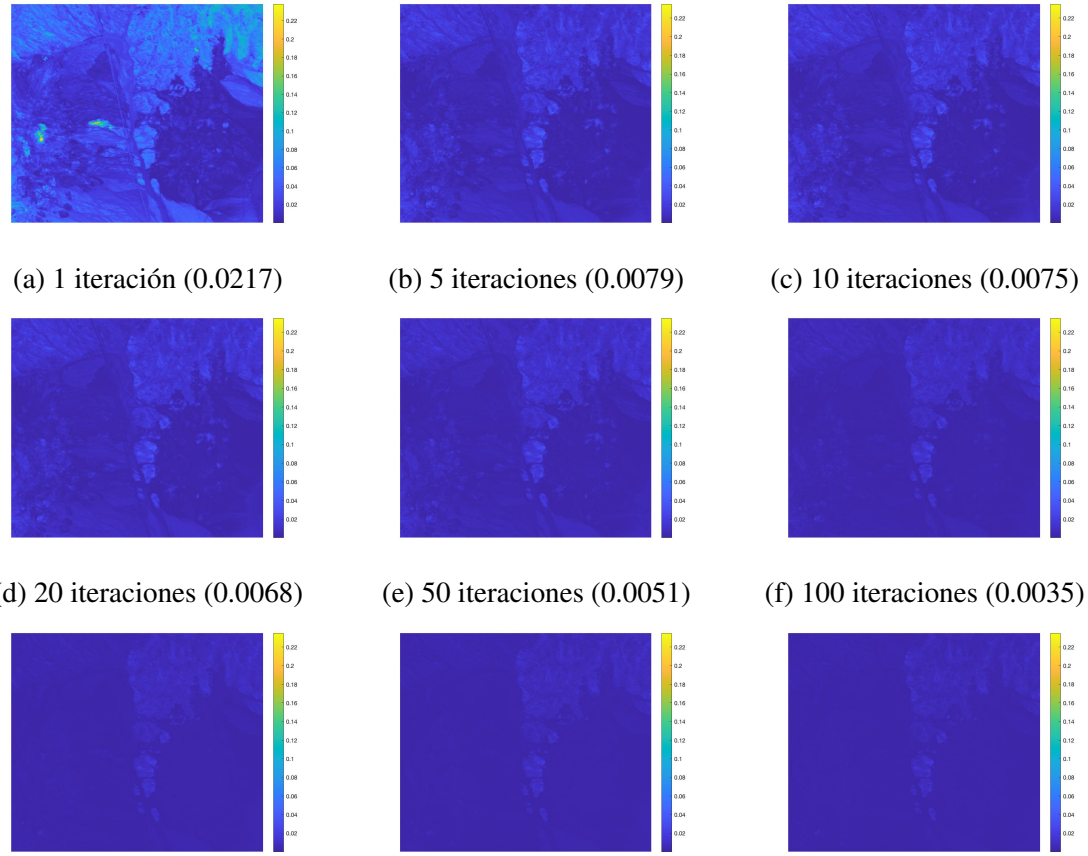
propia. Ambas versiones han sido validadas mediante pruebas realizadas con un extenso conjunto de datos sintéticos con resultados conocidos, con el objetivo de garantizar que ambas proporcionan los mismos resultados en los escenarios evaluados.

Para medir el error generado al reconstruir una imagen a partir de los endmembers ya extraídos y las abundancias estimadas, se puede utilizar el *Normalized Mean Square Error* (NMSE), que es comúnmente empleado en la literatura. Este error se basa en la multiplicación de la matriz de endmembers, **U**, identificada en la etapa anterior, por la matriz de abundancias, Φ, obteniendo así una versión aproximada $\widehat{X}$ de la imagen original. A continuación, se calcula el error de reconstrucción utilizando la siguiente expresión, que normaliza el error cuadrático medio con respecto a la energía de la imagen original:

$$NMSE(X,\widehat{X}) = \frac{\sum_{i=1}^{p}\sum_{k=1}^{b}(x_i^{(k)} - \widehat{x}_i^{(k)})^2}{\sum_{i=1}^{p}\sum_{k=1}^{b}(x_i^{(k)})^2}, \tag{5.5}$$

donde p es el número de píxeles de la imagen hiperespectral, b es el número de bandas, $x_i^{(k)}$ representa la k-ésima banda del píxel x_i de la imagen hiperespectral original, y $\widehat{x}_i^{(k)}$ representa la k-ésima banda del píxel reconstruido $\widehat{x}_i$, siendo $k \in \{1,2,\ldots,b\}$. Esta métrica proporciona una medida relativa del error de reconstrucción, lo que permite comparar el desempeño del modelo independientemente de la escala de la imagen. Un valor de NMSE cercano a cero indica que la imagen reconstruida se ajusta bien a la original, reflejando una representación adecuada mediante la combinación lineal de los endmembers extraídos, **U**, y sus correspondientes abundancias, Φ.

La escena AVIRIS Cuprite se ha utilizado como caso de estudio para evaluar la precisión de las fracciones de abundancia estimadas, con el fin de calcular el valor NMSE entre la escena original y la reconstruida, obtenida a partir de nuestra implementación del algoritmo ISRA propuesto. El único parámetro de entrada para el ISRA (además del conjunto de endmembers y la escena hiperespectral) es el número de iteraciones por píxel durante el análisis. En los experimentos realizados, este número se varió entre 1 y 600 para ilustrar la convergencia del algoritmo, observándose que, a partir de 600 iteraciones, los resultados ya no muestran variaciones significativas. La Figura 5.4 muestra los errores de reconstrucción para cada píxel de la imagen AVIRIS Cuprite considerando desde 1 iteración hasta 600 iteraciones.



(a) 1 iteración (0.0217) (b) 5 iteraciones (0.0079) (c) 10 iteraciones (0.0075)

(d) 20 iteraciones (0.0068) (e) 50 iteraciones (0.0051) (f) 100 iteraciones (0.0035)

(g) 200 iteraciones (0.0024) (h) 300 iteraciones (0.0021) (i) 600 iteraciones (0.0018)

Figura 5.4 Mapas NMSE entre la escena original AVIRIS Cuprite y el conjunto de datos reconstruido considerando desde 1 iteración hasta 600 iteraciones.

5.4.4. Evaluación del rendimiento computacional en ISRA

En este apartado, se lleva a cabo una evaluación experimental del rendimiento computacional de la implementación de ISRA, teniendo en cuenta las optimizaciones aplicadas sobre la FPGA previamente descrita. Para la versión optimizada se realizaron un total de 10 ejecuciones, cuyos resultados muestran una desviación estándar despreciable, lo que indica que los tiempos de ejecución son prácticamente idénticos entre sí.

La Tabla 5.3 presenta los tiempos de procesamiento y los recursos utilizados, obtenidos en la FPGA, para la versión optimizada del algoritmo. En este caso, se partió de la versión *baseline* y se consideraron las diferentes escenas analizadas en este estudio para un total de 10 iteraciones. En cuanto al uso de recursos, se observa que el empleo de 16 acumuladores y la capacidad de ejecutar hasta 8 *single tasks* en paralelo contribuyó significativamente

a mejorar el impacto de la optimización, resultando en una aceleración cercana a 80× en comparación con la versión *baseline*. Además, a modo ilustrativo, la Figura 5.5 presenta los tiempos de procesamiento de la implementación optimizada para las distintas escenas analizadas, en función del número de iteraciones.

Implementaciones	**Tiempos (ms)**			**F_{max}**	**Recursos**				
	AVIRIS Cuprite	**AVIRIS WTC**	**Sintética**	**(MHz)**	**ALUTs**	**FFs**	**RAMs**	**MLABs**	**DSPs**
Baseline	1014650	7208060	11177757	480	19537 (1%)	40062 (1%)	391 (3%)	272 (<1%)	28 (<1%)
ISRA_OPT_8Tasks	14720.1 (68.9×)	90267.6 (79.9×)	139978 (80×)	480	254835 (14%)	448109 (12%)	2203 (19%)	5800 (6%)	432 (8%)

Tabla 5.3 Tiempos de procesamiento (para 10 iteraciones) y recursos utilizados por el algoritmo ISRA usando la FPGA Intel Stratix 10 SX 2800 para las versiones baseline y optimizada.

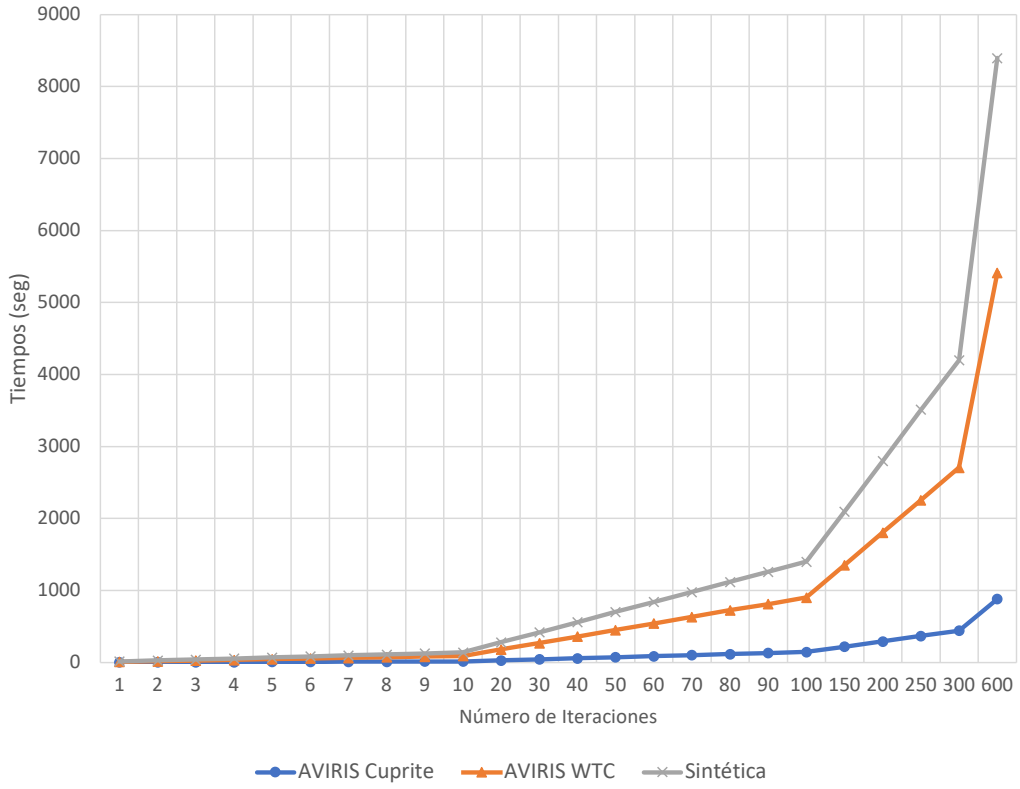


Figura 5.5 Tiempos de procesamiento para la implementación hardware del algoritmo ISRA para las imágenes hiperespectrales consideradas según el número de iteraciones.

Comparando estos resultados con otra implementación del algoritmo ISRA en hardware utilizando el lenguaje VHDL [35], el tiempo de ejecución para una FPGA Virtex-4 XC4VFX60 fue de 0.15 minutos (9,000 ms) para la escena AVIRIS Cuprite. Por su parte, la versión optimizada en HLS, ejecutada sobre una FPGA Intel Stratix 10 SX 2800 a 480 MHz,

presentó un tiempo de 14,720.1 ms. Esto implica que la versión en HLS es aproximadamente 1.64× más lenta.

No obstante, para un análisis más justo, es conveniente considerar la eficiencia en términos de ciclos de reloj. La implementación HLS requiere aproximadamente 7,065 millones de ciclos para ejecutar la escena (14,720.1 ms × 480 MHz), mientras que la implementación en VHDL requiere aproximadamente 394.2 millones de ciclos (9,000 ms × 43.8 MHz). Esto revela que la versión HDL es cerca de 18× más eficiente en uso de ciclos, reflejando una arquitectura altamente optimizada y específica.

A pesar de esta diferencia, cabe destacar que esta es una de las comparativas más equilibradas entre las propuestas HLS y HDL, lo cual demuestra que, utilizando el modelo de programación Intel oneAPI, es posible alcanzar rendimientos competitivos respecto a implementaciones de bajo nivel. Además, la solución basada en HLS ofrece importantes ventajas en cuanto a tiempo de desarrollo, portabilidad y mantenimiento, factores especialmente relevantes en proyectos con restricciones de tiempo o recursos humanos.

Capítulo 6

Cadena completa de desmezclado espectral

En este capítulo se presenta la integración de los algoritmos descritos en los capítulos anteriores para conformar una **cadena completa** que aborde el problema del **desmezclado espectral** en imágenes hiperespectrales. Esta cadena se compone de tres algoritmos implementados en hardware: 1) el algoritmo **VD**, encargado de estimar el número de endmembers; 2) **ATDCA-GS**, responsable de la detección y clasificación automática de materiales puros o endmembers; y 3) **ISRA**, utilizado para calcular la proporción de cada endmember en la mezcla espectral. Para llevar a cabo esta integración, es fundamental considerar diversos aspectos una vez que cada algoritmo ha sido implementado y optimizado de manera individual. Posteriormente, se procederá a su integración en una cadena completa, estableciendo un flujo de trabajo unificado y eficiente.

A lo largo del capítulo se discutirán las consideraciones necesarias para integrar cada algoritmo desarrollado de forma independiente en una única implementación. Esto permitirá obtener una solución optimizada en FPGAs mediante el uso del modelo de programación Intel oneAPI y el lenguaje DPC++. Finalmente, se analizarán los tiempos de procesamiento y el uso de recursos de hardware con el objetivo de evaluar su viabilidad en escenarios de tiempo real.

6.1. Descripción de la cadena completa

El problema de la mezcla es común al obtener imágenes con un sensor hiperespectral de resolución espacial moderada, donde cada píxel representa una combinación de las firmas

espectrales de los elementos puros o endmembers presentes en la escena. Para resolver esta situación, es necesario aplicar una cadena completa de algoritmos que permitan llevar a cabo el desmezclado espectral. En esta Tesis Doctoral, la cadena propuesta consta de tres etapas (ver Figura 6.1):

1. **Estimación del número de firmas espectrales puras (endmembers)**, t, en la escena hiperespectral a través del algoritmo VD. En esta etapa se realiza un proceso de estimación del número de endmembers presentes en la imagen $\mathbf{X}$ con el objetivo de establecer cuántas firmas espectrales puras deberán identificarse en las etapas posteriores.

2. **Identificación y extracción de una colección de endmembers** U a través del algoritmo ATDCA-GS. Esta etapa consiste en la identificación de las firmas espectrales de los endmembers en la escena. Aunque en la literatura existen diversas alternativas, en este trabajo se emplea un algoritmo basado en una aproximación geométrica que asume la presencia de píxeles puros en la imagen.

3. **Estimación de las abundancias** Φ, en la que se estima la proporción de cada endmember para cada uno de los píxeles mediante el algoritmo ISRA. Esta es la fase final del proceso, en la que, después de haber extraído los endmembers, se representa cada píxel de la imagen hiperespectral como una combinación lineal de los píxeles espectralmente puros. En otras palabras, se determina la proporción en la que cada endmember contribuye a la mezcla presente en cada píxel de la imagen.

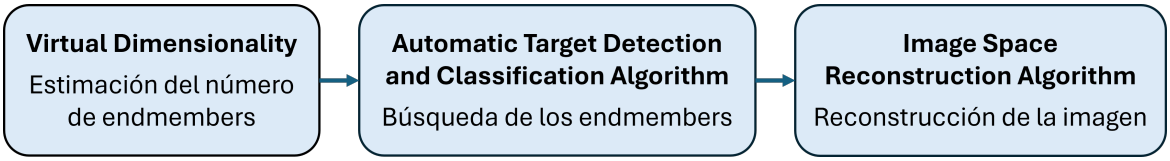


Figura 6.1 Diagrama de bloques que ilustra la cadena de desmezclado completa propuesta aplicada al análisis de imágenes hiperespectrales.

6.2. Implementación HLS de la cadena completa

La Figura 6.2 muestra la versión optimizada utilizada de cada algoritmo y la secuenciación de las tareas de la implementación de la cadena de desmezclado completa propuesta. El flujo de trabajo representado en la imagen se compone de tres etapas principales, cada una con un

conjunto de tareas específicas que se ejecutan de manera secuencial o en paralelo según sus dependencias.

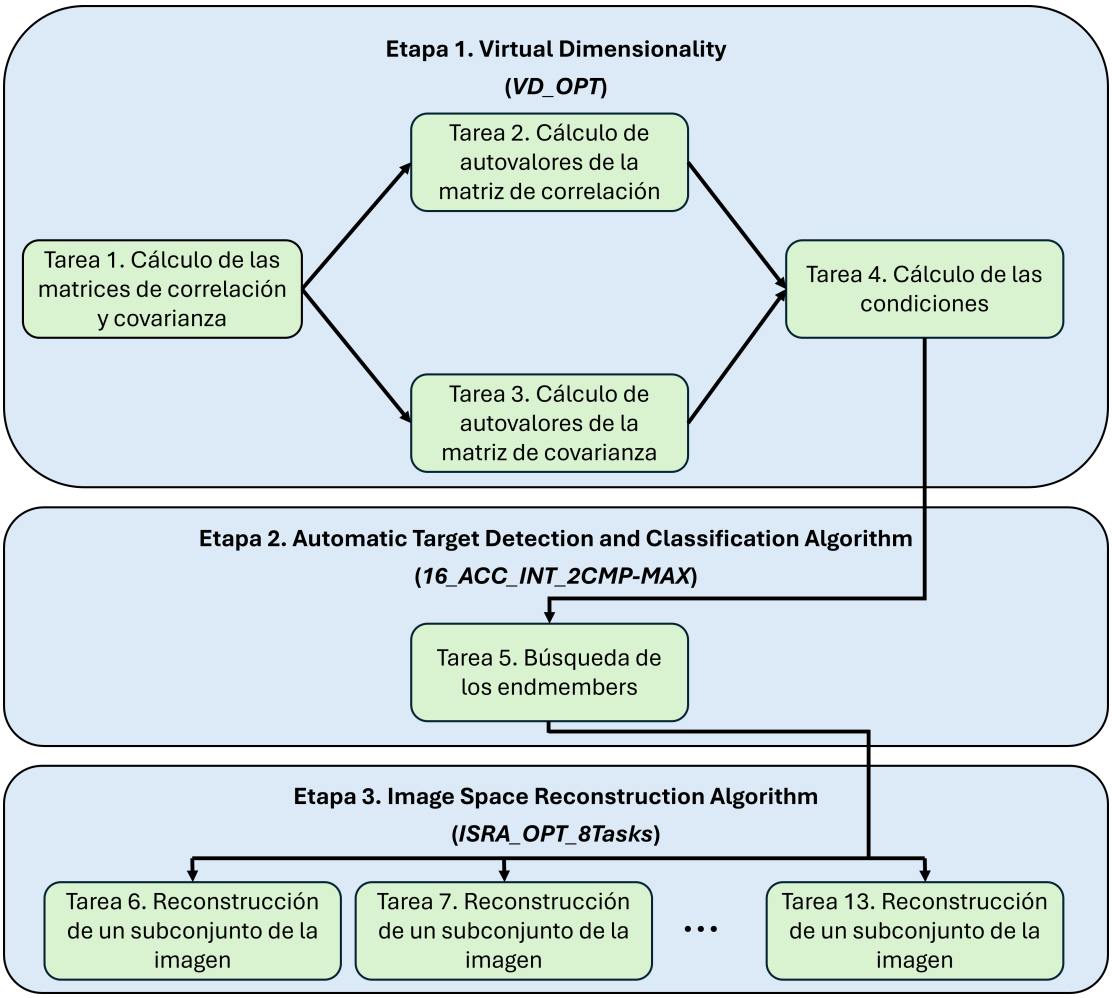


Figura 6.2 Diagrama que ilustra la secuenciación de las tareas de la implementación de la cadena de desmezclado completa propuesta aplicada al análisis de imágenes hiperespectrales.

En la **Etapa 1:** ***Virtual Dimensionality***, se realizan cálculos para la estimación del número de endmembers presentes en la imagen hiperespectral. Primero, en la Tarea 1, se calculan las matrices de correlación y covarianza, información esencial para los siguientes pasos. A partir de estos cálculos, la Tarea 2 obtiene los autovalores de la matriz de correlación, mientras que, en paralelo, la Tarea 3 se encarga del cálculo de los autovalores de la matriz de covarianza. Ambas tareas conducen a la Tarea 4, donde se calculan las condiciones para establecer el número de endmembers y avanzar a la siguiente etapa de la cadena.

En la **Etapa 2:** ***Automatic Target Detection and Classification Algorithm***, se lleva a cabo la detección y clasificación automática de objetivos. En esta fase, la Tarea 5 se enfoca

en la búsqueda de los endmembers, que representan firmas espectrales clave dentro de la imagen analizada. Esta tarea es crucial, ya que sus resultados servirán de base para la etapa final del flujo de trabajo.

Finalmente, en la **Etapa 3:** ***Image Space Reconstruction Algorithm***, se ejecuta la reconstrucción de la imagen en diferentes subconjuntos. Se presentan varias tareas de reconstrucción, que van desde la Tarea 7 a la Tarea 13, encargadas de procesar distintas secciones de la imagen. Aunque la imagen muestra solo algunas de estas tareas, se indica que existen más procesos similares en esta etapa.

En conjunto, este flujo de trabajo permite realizar un análisis detallado de datos mediante la evaluación de la dimensionalidad virtual, la detección de objetivos y la posterior reconstrucción de la imagen, asegurando así un procesamiento eficiente y estructurado. Finalmente, la Tabla 6.1 muestra la utilización de recursos de la cadena completa de desmezclado espectral propuesta.

Implementaciones	$\mathbf{F}_{max}$ **(MHz)**	**Recursos**				
		ALUTs	**FFs**	**RAMs**	**MLABs**	**DSPs**
Cadena completa de desmezclado espectral	432	490461 (26 %)	767287 (21 %)	3646 (31 %)	7407 (8 %)	840 (15 %)

Tabla 6.1 Recursos utilizados por la cadena completa de desmezclado espectral compuesta por los algoritmos VD, ATDCA e ISRA.

6.3. Resultados experimentales

En este apartado se presentan los resultados experimentales obtenidos tras integrar los algoritmos descritos en los capítulos anteriores en un único flujo de trabajo, formando así una cadena completa para el desmezclado espectral. En primer lugar, se realizará una revisión de las imágenes empleadas y la FPGA seleccionada como plataforma para la implementación. A continuación, se analizará el rendimiento computacional de la cadena completa, evaluando el tiempo de procesamiento para cada una de las imágenes hiperespectrales consideradas, utilizando 10 y 100 iteraciones en la etapa de estimación de abundancias. Finalmente, se llevará a cabo una comparación con otra implementación de la cadena completa de desmezclado, desarrollada en hardware mediante el lenguaje VHDL, con el objetivo de evaluar sus ventajas y diferencias respecto a la versión propuesta.

6.3.1. Imágenes hiperespectrales utilizadas

Se utilizan nuevamente las mismas imágenes hiperespectrales que fueron empleadas en la evaluación individual de cada uno de los algoritmos, las cuales fueron descritas en detalle en la Sección 2.1.4. Estas imágenes corresponden a:

- **Escena AVIRIS Cuprite**. Capturada sobre la región minera de Cuprite, Nevada, expresada en unidades de reflectancia.
- **Escena AVIRIS World Trade Center**. Registrada en Nueva York, cinco días después del atentado terrorista, también en unidades de reflectancia, con el objetivo de identificar focos de incendio en la zona afectada.
- **Escena sintética**. Diseñada para evaluar el rendimiento del algoritmo en imágenes de mayor tamaño, permitiendo analizar su escalabilidad y eficiencia computacional.

Estas imágenes ofrecen un conjunto de pruebas representativo, adecuado para validar el desempeño de la cadena completa de desmezclado en diversos escenarios y condiciones de análisis.

6.3.2. FPGA seleccionada

Para evaluar el rendimiento de la cadena completa de desmezclado espectral, se ha utilizado una de las FPGAs disponibles en el Intel Developer Cloud, concretamente la FPGA Intel Stratix 10 SX 2800 que fue descrita en detalle en la Sección 2.2.2. Esta placa ha sido fabricada con tecnología de 14 nm y cuenta con un total de 933.120 módulos lógicos adaptables (ALMs), 1.866.240 tablas de búsqueda adaptables (ALUTs) y 3.732.480 flip-flops (FFs). Además, dispone de una serie de recursos heterogéneos, incluyendo 5760 DSPs y 11721 bloques de memoria RAM, así como otros componentes adicionales que no han sido utilizados.

6.3.3. Evaluación del rendimiento computacional de la cadena completa

En este apartado se realiza una evaluación experimental del rendimiento computacional de la implementación de la cadena completa de desmezclado, considerando las optimizaciones aplicadas a la FPGA previamente descrita. Para la versión optimizada, se llevaron a cabo un

total de 10 ejecuciones, cuyos resultados revelan una desviación estándar insignificante, lo que indica que los tiempos de ejecución son prácticamente idénticos en todas las pruebas.

La Tabla 6.2 muestra los tiempos de procesamiento obtenidos en la FPGA para la versión optimizada de la cadena completa de desmezclado. En este caso, se optó por utilizar las versiones optimizadas de cada uno de los algoritmos, tal como se especificó en los capítulos anteriores. Para este análisis, se consideró variar el número de iteraciones en la etapa ISRA entre 10 iteraciones, que es el mínimo necesario para obtener un error NMSE suficientemente bajo y asegurar una reconstrucción aceptable de la imagen, y 100 iteraciones, con las cuales se lograba un valor NMSE muy bajo, obteniendo así una excelente reconstrucción de la imagen a partir de los endmembers y las estimaciones de abundancias proporcionadas por la cadena completa.

	AVIRIS Cuprite		**AVIRIS WTC**		**Sintética**	
	10 iter.	**100 iter.**	**10 iter.**	**100 iter.**	**10 iter.**	**100 iter.**
Estimación de endmembers (VD)	48.94	48.94	125.87	125.87	195.64	195.64
Extracción de endmembers (ATDCA-GS)	0.17	0.17	0.75	0.75	1.13	1.13
Estimación de abundancias (ISRA)	14.60	145.99	89.68	896.32	139.08	1390.03
Total	63.71	195.10	216.30	1022.94	335.85	1586.80

Tabla 6.2 Tiempos de procesamiento en segundos para la implementación hardware de la cadena completa de desmezclado usando la FPGA Intel Stratix 10 SX 2800 para cada una de las imágenes hiperespectrales consideradas con 10 y 100 iteraciones.

Al comparar estos resultados con los de otra cadena completa de desmezclado implementada en hardware utilizando el lenguaje VHDL [36], el tiempo de ejecución para una FPGA Virtex-4 XC4VFX60 fue de 1.92 minutos (115,200 ms) para la escena AVIRIS Cuprite, utilizando 100 iteraciones en la etapa ISRA. En nuestra propuesta, ejecutada en una FPGA Intel Stratix 10 SX 2800 a 432 MHz, el tiempo de ejecución fue de 195.10 s (195,100 ms). Esto indica que la versión en HLS es aproximadamente 1.69$\times$ más lenta en tiempo absoluto.

Sin embargo, al considerar la eficiencia relativa en función de los ciclos de reloj, se observa una diferencia más significativa. La implementación en HLS requiere aproximadamente 84,973 millones de ciclos (195,100 ms $\times$ 432 MHz), frente a los 5,045 millones de ciclos de la versión en VHDL (115,200 ms $\times$ 43.8 MHz), lo que implica que la versión en HDL es unas 16.8$\times$ más eficiente en este sentido. Esta diferencia se explica, en parte, por el uso de arquitecturas altamente específicas y optimizadas en HDL, frente a una solución más generalista desarrollada en HLS.

Además, es importante destacar que la cadena de desmezclado utilizada en esta tesis difiere de la propuesta en [36], ya que se emplea un algoritmo distinto en la etapa de detección de endmembers. Esta variación metodológica puede afectar tanto el comportamiento compu-

tacional como el flujo general de procesamiento, y debe ser tenida en cuenta al interpretar los resultados.

En consecuencia, se concluye que una implementación HLS de la cadena completa de desmezclado puede ser una opción adecuada en casos donde el tiempo de desarrollo, la flexibilidad o la necesidad de combinar múltiples algoritmos sean factores determinantes, aunque con una penalización en eficiencia frente a soluciones HDL altamente especializadas.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

El análisis eficiente de imágenes hiperespectrales continúa siendo un área de gran actividad dentro de la observación remota de la Tierra, así como en otras aplicaciones de alto impacto para la humanidad, como la agricultura, la minería, la salud y la gestión ambiental. Esta relevancia se ve reflejada en el creciente número de misiones espaciales activas y planificadas, orientadas a obtener información cada vez más detallada del planeta y contribuir a la resolución de los desafíos asociados a los Objetivos de Desarrollo Sostenible (ODS). Para ello, es fundamental poder clasificar los píxeles que componen una escena, los cuales suelen representar mezclas complejas de múltiples materiales. Esta necesidad de descomposición es especialmente relevante en el uso de sensores hiperespectrales, que capturan imágenes con alta resolución tanto espectral como espacial, y puede abordarse mediante técnicas eficientes de desmezclado espectral.

En el presente trabajo se ha desarrollado una nueva cadena de procesamiento para el desmezclado espectral en paralelo, orientada a resolver el problema de los píxeles mezcla en imágenes hiperespectrales de la superficie terrestre, incorporando diversas contribuciones innovadoras en este campo. Para alcanzar este objetivo, se cumplieron los propósitos establecidos al inicio de esta investigación. En primer lugar, se analizó en profundidad uno de los principales desafíos asociados al uso de imágenes hiperespectrales: el problema de la mezcla, cuya comprensión resulta esencial para el cumplimiento de los objetivos de las misiones espaciales, dada la riqueza de información contenida en los datos adquiridos por los sensores. Tradicionalmente, este problema se aborda mediante un modelo de mezcla lineal, el cual requiere una cadena de procesamiento compuesta por la integración de tres

etapas principales: 1) estimación del número de endmembers; 2) detección y clasificación automática de materiales puros (endmembers); y 3) estimación de la proporción de cada endmember en la mezcla espectral.

Una vez identificadas las etapas necesarias para abordar el problema de la mezcla, se realizó un análisis detallado de las principales técnicas correspondientes a cada fase del desmezclado espectral, evaluando sus ventajas e inconvenientes desde una perspectiva algorítmica, con el objetivo de adaptarlas a una implementación hardware sobre una plataforma FPGA. Este tipo de plataforma destaca por ofrecer un rendimiento competitivo frente a otras arquitecturas, especialmente en el procesamiento eficiente de imágenes hiperespectrales, gracias a su bajo consumo energético y a la reducción de costes, lo que la convierte en una de las alternativas más atractivas para aplicaciones de observación remota de la Tierra. No obstante, uno de los principales desafíos sigue siendo el elevado tiempo de desarrollo asociado al uso de lenguajes de descripción hardware (HDL) tradicionales. En este contexto, una solución basada en diseño mediante lenguajes de alto nivel (HLS) representa una contribución innovadora en el uso de dispositivos FPGA. En particular, se seleccionó el lenguaje DPC++, incluido en el conjunto de herramientas Intel oneAPI, el cual permite implementar diversas estrategias de paralelismo sobre FPGA. Esta elección buscó conservar las ventajas inherentes al hardware especializado, al mismo tiempo que se reducen significativamente los tiempos de desarrollo en comparación con lenguajes como Verilog o VHDL.

A partir de la emulación y prueba de varios códigos incluidos en el toolkit de Intel oneAPI, se desarrolló una versión *baseline* para cada uno de los algoritmos seleccionados: 1) *Virtual Dimensionality* (VD), utilizado para la estimación del número de endmembers; 2) *Automatic Target Detection and Classification Algorithm* mediante el proceso de ortogonalización de Gram-Schmidt (ATDCA-GS), para la detección automática de los distintos endmembers; y 3) *Image Space Reconstruction Algorithm* (ISRA), empleado para la estimación de la proporción de cada endmember presente en la mezcla espectral. Esta versión mínima, apta para su ejecución en dispositivos FPGA, permitió evaluar el impacto de cada optimización incorporada y comparar el rendimiento de la implementación desarrollada con respecto a una implementación tradicional basada en HDL. Hasta el momento, las implementaciones existentes de estos algoritmos habían demostrado su eficacia únicamente mediante diseños HDL sobre FPGA, los cuales suelen estar fuertemente ligados al hardware específico de la placa y requieren tiempos de desarrollo considerablemente largos. En este contexto, se elaboró una guía con recomendaciones para la escritura de código HLS eficiente en DPC++, orientada a facilitar el desarrollo sobre este tipo de hardware especializado, manteniendo un equilibrio entre portabilidad, rendimiento y eficiencia en el tiempo de implementación.

Una vez finalizados los códigos base, se aplicó la guía de recomendaciones propuesta al primer algoritmo de la cadena de desmezclado espectral. Antes de iniciar las optimizaciones sobre el algoritmo VD, se seleccionaron estrategias previamente validadas en la literatura por su contribución al rendimiento. Entre ellas se incluyeron: el cálculo incremental de la matriz de correlación a medida que se reciben los datos del sensor, y el uso del método de Jacobi para el cómputo de los autovalores de las matrices de correlación y covarianza. Con este enfoque, se reestructuró la *single task* original dividiéndola en cuatro tareas independientes: 1) cálculo de las matrices de correlación y covarianza; 2) cálculo de los autovalores de la matriz de correlación (en paralelo con la siguiente); 3) cálculo de los autovalores de la matriz de covarianza; y 4) evaluación de las condiciones necesarias una vez completadas las fases 2 y 3. Estas optimizaciones permitieron reducir de forma significativa los tiempos de cálculo de las matrices, disminuyendo el tiempo total de procesamiento hasta aproximarse al tiempo de cálculo de los autovalores. Como resultado, se logró una reducción del tiempo de procesamiento del 57 % para la escena AVIRIS Cuprite y del 78 % para las otras dos escenas evaluadas. Los experimentos mostraron que, si bien nuestra versión optimizada en HLS era inicialmente 26× veces más lenta que una implementación en VHDL, la capacidad de procesar datos sin necesidad de contar con la imagen completa permitió reducir esa diferencia a 11 veces más lenta para la escena AVIRIS Cuprite y 5.6× veces para AVIRIS WTC. Estos resultados reflejan un rendimiento prometedor, especialmente si se considera que el desarrollo en HLS requiere considerablemente menos tiempo que el diseño de una solución equivalente en HDL sobre FPGA.

Para la versión optimizada del algoritmo ATDCA, se seleccionó el método de ortogonalización de Gram-Schmidt (GS), lo que permitió evitar la inversión de matrices y aportó beneficios notables en términos de rendimiento, eficiencia energética y simplicidad en el desarrollo del código. En esta implementación HLS se aplicaron diversas optimizaciones buscando un equilibrio entre rendimiento y uso de recursos: 1) se aumentó el número de acumuladores a 16, lo que facilitó el acceso secuencial a múltiples muestras de un píxel, permitió la ejecución simultánea de operaciones de multiplicación y redujo considerablemente el número de iteraciones; 2) se realizó la conversión de operaciones en punto flotante a enteros, disminuyendo la complejidad lógica y el uso de memoria, lo cual redujo el número de ciclos de ejecución y mejoró el consumo de recursos; 3) se implementaron comparaciones en paralelo durante la búsqueda del píxel con la máxima proyección, permitiendo dos comparaciones simultáneas como compromiso entre tiempo de procesamiento y uso de hardware; y 4) se reestructuró el código para que una única estructura hardware fuera capaz de calcular tanto el píxel más brillante como las proyecciones de todos los píxeles, eliminando así la necesidad de replicar funcionalidades. Estas optimizaciones permitieron operar a una

mayor frecuencia de reloj, hacer un uso más eficiente del hardware disponible y mejorar la escalabilidad del diseño, lo que se tradujo en una única síntesis capaz de admitir cualquier tamaño de imagen (hasta 487.500 píxeles, 224 bandas y 30 endmembers). Los resultados experimentales mostraron que, si bien la implementación en HLS fue aproximadamente $6\times$ veces más lenta que su homóloga en VHDL, se logró procesamiento en tiempo real, con un rendimiento que escalaba favorablemente al aumentar el tamaño de la imagen, lo cual evidencia el potencial del enfoque propuesto para aplicaciones de observación remota a gran escala.

Para el último algoritmo de la cadena de desmezclado espectral, ISRA, el esfuerzo se centró en la fase de corrección espectral, fundamental para la estimación precisa de las abundancias de los endmembers en cada píxel. La aceleración del procesamiento se consiguió mediante dos estrategias complementarias: 1) la optimización individual de cada unidad de procesamiento, con el objetivo de minimizar su consumo de recursos hardware; y 2) la replicación de dichas unidades para habilitar su ejecución en paralelo. Esta combinación permitió maximizar el aprovechamiento del hardware disponible y mejorar de forma significativa el rendimiento global del sistema. Al reducir la complejidad de cada unidad, se facilitó además la escalabilidad del diseño, haciéndolo adecuado para el procesamiento eficiente de imágenes con alta resolución espectral. Los resultados experimentales mostraron que la implementación optimizada en HLS para ISRA era aproximadamente $1.64\times$ veces más lenta que la versión en VHDL. Sin embargo, esta implementación ofreció un rendimiento escalable, lo que permitió procesar imágenes de grandes dimensiones sin comprometer la precisión de los cálculos, logrando una mejora notable en términos de tiempo de desarrollo y facilidad de integración en plataformas FPGA, en comparación con los enfoques tradicionales en HDL.

Finalmente, se presentó la integración de los tres algoritmos para abordar el problema del desmezclado espectral en imágenes hiperespectrales, formando una cadena completa implementada en hardware. De la misma forma que con los algoritmos individuales, esta integración se llevó a cabo mediante una implementación optimizada en plataformas FPGA utilizando el modelo de programación Intel oneAPI y el lenguaje DPC++. El flujo de trabajo propuesto, basado en un flujo de tareas secuenciales y paralelas, permitió una ejecución eficiente de las distintas etapas (y fases) de la cadena. Este enfoque optimiza la utilización de recursos y asegura que cada tarea se ejecute de manera eficaz, respetando las dependencias entre ellas. La implementación en FPGA demostró ser eficaz, logrando un alto nivel de optimización en términos de recursos y rendimiento, con tiempos de procesamiento que fueron evaluados utilizando diferentes configuraciones de iteraciones (10 y 100) en el algoritmo ISRA. Los resultados experimentales mostraron que la implementación de la cadena completa en hardware optimizado es viable para escenarios de desmezclado espectral

en imágenes hiperespectrales, con tiempos de procesamiento que varían dependiendo de la escena y el número de iteraciones. La comparación con una implementación previa en VHDL reveló que, si bien la versión en HLS es ligeramente más lenta (aproximadamente $1.69\times$ veces), ofrece ventajas significativas en términos de flexibilidad, control sobre la secuenciación de las tareas y tiempo de desarrollo. Esto sugiere que la implementación en HLS podría ser adecuada en contextos donde el tiempo de desarrollo y la capacidad de implementar múltiples algoritmos sean factores críticos.

7.2. Trabajo futuro

Los resultados obtenidos en esta investigación abren múltiples líneas de trabajo que pueden explorarse para seguir avanzando en la eficiencia, flexibilidad y aplicabilidad del desmezclado espectral en hardware especializado. Una de las principales áreas de mejora es la optimización del rendimiento del sistema. Aunque el uso de lenguajes de alto nivel (HLS) como DPC++ ha demostrado ser efectivo para reducir significativamente los tiempos de desarrollo frente a enfoques tradicionales en HDL, el rendimiento en tiempo de ejecución aún presenta margen de mejora. En este sentido, futuras investigaciones podrían centrarse en la aplicación de estrategias de paralelismo más avanzadas, como la explotación de estructuras jerárquicas de pipeline, el uso de controladores de flujo dinámicos o la fusión de tareas con balanceo adaptativo, con el fin de mejorar el aprovechamiento de los recursos disponibles en la FPGA y reducir aún más los tiempos de procesamiento.

Otro aspecto de gran interés para el trabajo futuro es la incorporación de modelos de mezcla más complejos. El enfoque actual se basa en el modelo lineal, el cual es ampliamente aceptado y ofrece buenos resultados en muchos escenarios. Sin embargo, en situaciones donde la interacción espectral entre materiales no sigue un comportamiento estrictamente lineal (como en casos de dispersión múltiple, mezclas íntimas o efectos atmosféricos), podría ser beneficioso implementar variantes no lineales del modelo de mezcla. Estas alternativas, aunque más costosas computacionalmente, podrían integrarse en hardware aprovechando los recursos paralelos de la FPGA. En particular, se plantea el estudio de modelos no lineales basados en núcleos (kernel-based), redes neuronales específicas para el desmezclado, o métodos híbridos que combinen aproximaciones físicas y estadísticas.

Asimismo, se considera prometedor explorar la automatización del flujo de desarrollo en HLS. Una posibilidad es desarrollar herramientas que permitan generar automáticamente estructuras paralelizables a partir de una descripción funcional de los algoritmos, facilitando el diseño de arquitecturas adaptadas a diferentes necesidades sin requerir conocimientos

profundos en hardware. Esta automatización permitiría reutilizar componentes optimizados, adaptar la cadena de procesamiento a nuevos algoritmos emergentes, o generar implementaciones específicas en función de los recursos disponibles en cada dispositivo FPGA. En paralelo, también se sugiere profundizar en el análisis del coste-rendimiento de cada etapa de la cadena, con el objetivo de identificar cuellos de botella y mejorar la asignación de tareas en sistemas heterogéneos.

Finalmente, sería interesante evaluar la integración de la cadena desarrollada en entornos de simulación o procesamiento geoespacial más amplios, donde se combinen datos hiperespectrales con información multifuente (como imágenes multiespectrales, datos LiDAR o información geográfica). Esta integración permitiría aprovechar al máximo la riqueza espectral de los datos, habilitando aplicaciones más robustas en campos como el monitoreo ambiental, la agricultura de precisión o la detección temprana de riesgos naturales. La combinación de hardware especializado con técnicas de fusión de datos abre un campo de investigación multidisciplinar que puede ampliar significativamente el impacto de este tipo de soluciones.

Publicaciones generadas

A continuación, se presenta el conjunto de publicaciones mediante las cuales se ha divulgado parte de este trabajo de investigación. Están divididas por publicaciones en revistas con índice de impacto *Journal Citation Reports* (JCR) y artículos en congresos internacionales, ordenadas siguiendo un orden cronológico inverso.

Publicaciones en revistas con índice JCR:

- Rubén Macias, Sergio Bernabé y Carlos González. "High-Level Synthesis Design for an FPGA Implementation of the Image Space Reconstruction Algorithm in Hyperspectral Unmixing". Journal of Supercomputing. En proceso de revisión.
- Rubén Macias, Sergio Bernabé y Carlos González. "FPGA-Based HLS Acceleration with Intel oneAPI for Estimating the Virtual Dimensionality of Hyperspectral Imagery". The International Journal of High Performance Computing Applications. En proceso de revisión.
- Rubén Macias, Sergio Bernabé y Carlos González. "High-Level Synthesis Acceleration for an FPGA Implementation of an Optimized Automatic Target Detection and Classification Algorithm for Hyperspectral Image Analysis with Intel oneAPI". Journal of Computational Science. En proceso de revisión.
- Rubén Macias, Sergio Bernabé, Daniel Báscones y Carlos González, "FPGA Implementation of a Hardware Optimized Automatic Target Detection and Classification Algorithm for Hyperspectral Image Analysis". IEEE Geoscience and Remote Sensing Letters. Volumen 19, páginas 1–5. Julio de 2022. DOI: 10.1109/LGRS.2022.3189109.

Artículos publicados en congresos internacionales:

- Rubén Macias, Sergio Bernabé y Carlos González. “Accelerating the ATDCA Algorithm for Endmember Extraction from Hyperspectral Imagery with Intel oneAPI for FPGAs”. 33rd International Conference on Field-Programmable Logic and Applications (FPL). Páginas 349–350. Gothenburg (Sweden). Septiembre 2023. DOI: 10.1109/FPL60245.2023.00061

Bibliografía

[1] Abdali, E. M., Picone, D., Dalla-Mura, M., and Mancini, S. (2023). Implementation of hyperspectral inversion algorithms on fpga: Hardware comparison using high level synthesis.

[2] Adams, J. B., Smith, M. O., and Johnson, P. E. (1986). Spectral mixture modeling: a new analysis of rock and soil types at the Viking Lander 1 site. *Journal of Geophysical Research*, 91:8098–8112.

[3] Akbari, H., Halig, L., Schuster, D., Osunkoya, A., Master, V., Nieh, P., Chen, G., and Fei, B. (2012). Hyperspectral imaging and quantitative analysis for prostate cancer detection. *Journal of biomedical optics*, 17:076005.

[4] Amazon (2022). Instancias F1 de Amazon EC2. https://aws.amazon.com/es/ec2/instance-types/f1/.

[5] Bascones, D., Gonzalez, C., and Mozos, D. (2017). Parallel Implementation of the CCSDS 1.2.3 Standard for Hyperspectral Lossless Compression. *Remote Sensing*, 9(10).

[6] Bascones, D., Gonzalez, C., and Mozos, D. (2018a). FPGA Implementation of the CCSDS 1.2.3 Standard for Real-Time Hyperspectral Lossless Compression. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 11(4):1158–1165.

[7] Bascones, D., Gonzalez, C., and Mozos, D. (2018b). Hyperspectral Image Compression Using Vector Quantization, PCA and JPEG2000. *Remote Sensing*, 10(6):907.

[8] Bascones, D., Gonzalez, C., and Mozos, D. (2020a). An Extremely Pipelined FPGA Implementation of a Lossy Hyperspectral Image Compression Algorithm. *IEEE Transactions on Geoscience and Remote Sensing*, 58(10):7435–7447.

[9] Bascones, D., Gonzalez, C., and Mozos, D. (2020b). An FPGA Accelerator for Real-Time Lossy Compression of Hyperspectral Images. *Remote Sensing*, 12(16):2563.

[10] Bernabe, S., Garcia, C., Igual, F. D., Botella, G., Prieto-Matias, M., and Plaza, A. (2019). Portability study of an OpenCL algorithm for automatic target detection in hyperspectral images. *IEEE Transactions on Geoscience and Remote Sensing*, 57(11):9499–9511.

[11] Bernabe, S., Jimenez, L. I., Garcia, C., Plaza, J., and Plaza, A. (2018). Multicore real-time implementation of a full hyperspectral unmixing chain. *IEEE Geoscience and Remote Sensing Letters*, 15(5):744–748.

[12] Bernabe, S., Lopez, S., Plaza, A., and Sarmiento, R. (2012). GPU implementation of an automatic target detection and classification algorithm for hyperspectral image analysis. *IEEE Geoscience and Remote Sensing Letters*, 10(2):221–225.

[13] Bernabe, S., Sanchez, S., Plaza, A., Lopez, S., Benediktsson, J. A., and Sarmiento, R. (2013). Hyperspectral unmixing on GPUs and multi-core processors: A comparison. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 6(3):1386–1398.

[14] Bioucas-Dias, J. M. and Nascimento, J. M. P. (2008). Hyperspectral subspace identification. *IEEE Transactions on Geoscience and Remote Sensing*, 46(8):2435–2445.

[15] Bioucas-Dias, J. M., Plaza, A., Dobigeon, N., Parente, M., Du, Q., Gader, P., and Chanussot, J. (2012). Hyperspectral unmixing overview: Geometrical, statistical, and sparse regression-based approaches. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5(2):354–379.

[16] Boardman, J. W., Kruse, F. A., and Green, R. O. (1995). Mapping Target Signatures Via Partial Unmixing of Aviris Data. *Proc. JPL Airborne Earth Sci. Workshop*, pages 23–26.

[17] Camps-Valls, G., Tuia, D., Bruzzone, L., and Benediktsson, J. A. (2014). Advances in hyperspectral image classification: Earth monitoring with statistical learning methods. *IEEE Signal Processing Magazine*, 31(1):45–54.

[18] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., and Czajkowski, T. (2011). Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 33–36, New York, NY, USA. Association for Computing Machinery.

[19] Cenk, M. and Hasan, M. A. (2017). On the arithmetic complexity of strassen-like matrix multiplications. *Journal of Symbolic Computation*, 80:484–501.

[20] Chang, C.-I. (2003). *Hyperspectral imaging: techniques for spectral detection and classification*, volume 1. Springer Science & Business Media.

[21] Chang, C.-I. (2013). *Hyperspectral Data Processing: Algorithm Design and Analysis*. John Wiley & Sons, Hoboken, NJ, USA.

[22] Chang, C.-I. and Du, Q. (2004). Estimation of number of spectrally distinct signal sources in hyperspectral imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 42(3):608–619.

[23] Daube-Witherspoon, M. E. and Muehllehner, G. (1986). An Iterative Image Space Reconstruction Algorthm Suitable for Volume ECT. *IEEE Transactions on Medical Imaging*, 5:61–66.

[24] Domingo, R., Salvador, R., Fabelo, H., Madronal, D., Ortega, S., Lazcano, R., Juarez, E., Callico, G., and Sanz, C. (2017). High-level design using Intel FPGA OpenCL: A hyperspectral imaging spatial-spectral classifier. In *Proceedings of ReCoSoC*.

[25] Eches, O., Dobigeon, N., and Tourneret, J.-Y. (2010). Estimating the number of endmembers in hyperspectral images using the normal compositional model and a hierarchical bayesian algorithm. *IEEE Journal of Selected Topics in Signal Processing*, 4(3):582–591.

[26] EDN (2014). Actel FPGAs Play Critical Role in Uncovering Origins of the Solar System. https://www.edn.com/actel-fpgas-play-critical-role-in-uncovering-origins-of-the-solar-system/.

[27] Fallahlalehzari, F. (2021). How does the Mars Perseverance rover benefit from FPGAs as the main processing units? https://www.aldec.com/en/company/blog/188–how-does-the-mars-perseverance-rover-benefit-from-fpgas-as-the-main-processing-units.

[28] Fenzandez, D., Gonzalez, C., Mozos, D., and Lopez, S. (2019). FPGA implementation of the principal component analysis algorithm for dimensionality reduction of hyperspectral images. *Journal of Real-Time Image Processing*, 16(4):1395–1406.

[29] Goetz, A. F. (2009). Three decades of hyperspectral remote sensing of the earth: A personal view. *Remote Sensing of Environment*, 113:S5–S16. Imaging Spectroscopy Special Issue.

[30] Gonzalez, C., Bernabe, S., Mozos, D., and Plaza, A. (2016). FPGA Implementation of an Algorithm for Automatically Detecting Targets in Remotely Sensed Hyperspectral Images. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 9(9):4334–4343.

[31] Gonzalez, C., Lopez, S., Mozos, D., and Sarmiento, R. (2015). FPGA Implementation of the HySime Algorithm for the Determination of the Number of Endmembers in Hyperspectral Data. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 8:1–14.

[32] Gonzalez, C., Lopez, S., Mozos, D., and Sarmiento, R. (2018). A Novel FPGA-Based Architecture for the Estimation of the Virtual Dimensionality in Remotely Sensed Hyperspectral Images. *J. Real-Time Image Process.*, 15(2):297–308.

[33] Gonzalez, C., Mozos, D., Resano, J., and Plaza, A. (2012a). FPGA Implementation of the N-FINDR Algorithm for Remotely Sensed Hyperspectral Image Analysis. *IEEE Transactions on Geoscience and Remote Sensing*, 50(2):374–388.

[34] Gonzalez, C., Resano, J., Mozos, D., Plaza, A., and Valencia, D. (2010). FPGA Implementation of the Pixel Purity Index Algorithm for Remotely Sensed Hyperspectral Image Analysis. *EURASIP Journal on Advances in Signal Processing*, 2010:969806.

[35] Gonzalez, C., Resano, J., Plaza, A., and Mozos, D. (2012b). FPGA Implementation of Abundance Estimation for Spectral Unmixing of Hyperspectral Data Using the Image Space Reconstruction Algorithm. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5(1):248–261.

[36] González, C. (2011). *Procesamiento a bordo de imágenes hiperespectrales de la superficie terrestre mediante hardware reconfigurable*. Tesis doctoral, Facultad de Informática. Universidad Complutense de Madrid. Disponible en https://docta.ucm.es/bitstreams/fef92bd8-6555-4ff6-bab0-cb333c5a6cb5/download.

[37] GrandviewResearch (2022). Field Programmable Gate Array Market Size, Share and Trends Analysis Report By Technology (SRAM, Antifuse, Flash), By Application (Military and Aerospace, Telecom), By Region, And Segment Forecasts, 2020 - 2027. https://www.grandviewresearch.com/industry-analysis/fpga-market.

[38] Green, A. A., Berman, M., Switzer, P., and Craig, M. D. (1988). A transformation for ordering multispectral data in terms of image quality with implications for noise removal. *IEEE Transactions on Geoscience and Remote Sensing*, 26:65–74.

[39] Green, R. O., Eastwood, M. L., Sarture, C. M., Chrien, T. G., Aronsson, M., Chippendale, B. J., Faust, J. A., Pavri, B. E., Chovit, C. J., Solis, M., et al. (1998). Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS). *Remote Sensing of Environment*, 65(3):227–248.

[40] Guerra, R., Lopez, S., and Sarmiento, R. (2017a). A FPGA implementation for linearly unmixing a hyperspectral image using OpenCL. In *Proceedings of the SPIE*.

[41] Guerra, R., Martel, E., Khan, J., López, S., Athanas, P., and Sarmiento, R. (2017b). On the Evaluation of Different High-Performance Computing Platforms for Hyperspectral Imaging: An OpenCL-Based Approach. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 10(11):4879–4897.

[42] Haboudane, D., Miller, J. R., Pattey, E., Zarco-Tejada, P. J., and Strachan, I. B. (2004). Hyperspectral vegetation indices and novel algorithms for predicting green lai of crop canopies: Modeling and validation in the context of precision agriculture. *Remote Sensing of Environment*, 90(3):337–352.

[43] Hamada, T., Benkrid, K., Nitadori, K., and Taiji, M. (2009). A Comparative Study on ASIC, FPGAs, GPUs and General Purpose Processors in the O(N^2) Gravitational N-body Simulation. In *2009 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 447–452.

[44] Harsanyi, J. and Chang, C.-I. (1994). Hyperspectral image classification and dimensionality reduction: an orthogonal subspace projection approach. *IEEE Transactions on Geoscience and Remote Sensing*, 32(4):779–785.

[45] Heinz, D. and Chein-I-Chang (2001). Fully constrained least squares linear spectral mixture analysis method for material quantification in hyperspectral imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 39(3):529–545.

[46] Heylen, R., Parente, M., and Gader, P. (2014). A review of nonlinear hyperspectral unmixing methods. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 7(6):1844–1868.

[47] Instituto de Astrofisica de Andalucia (2020). El instrumento SO/PHI, a bordo de la mision Solar Orbiter, obtiene el primer mapa magnetico autonomo del Sol. www.iaa.csic.es/noticias/el-instrumento-sophi-bordo-mision-solar-orbiter-obtiene-el-primer-mapa-magnetico-autonomo.

[48] Intel (2022a). FPGA for Military Applications - Intel® FPGA. https://www.intel.es/content/www/es/es/government/products/programmable/applications.html.

[49] Intel (2022b). Intel stratix 10 logic array blocks and adaptive logic modules user guide. https://cdrdv2-public.intel.com/666917/ug-s10-lab-683699-666917.pdf.

[50] Intel (2024). oneapi programming guide. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html.

[51] Intel (2025). Intel oneapi base toolkit. https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit.html.

[52] Jimenez, L. and Landgrebe, D. (1998). Supervised classification in high-dimensional space: geometrical, statistical, and asymptotical properties of multivariate data. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 28(1):39–54.

[53] Jimenez, L., Rivera-Medina, J., Rodriguez-Diaz, E., Arzuaga-Cruz, E., and Ramirez-Velez, M. (2005). Integration of spatial and spectral information by means of unsupervised extraction and classification for homogenous objects applied to multispectral and hyperspectral data. *IEEE Transactions on Geoscience and Remote Sensing*, 43(4):844–851.

[54] Jimenez, L. I., Martin, G., Sanchez, S., Garcia, C., Bernabe, S., Plaza, J., and Plaza, A. (2016). GPU Implementation of Spatial–Spectral Preprocessing for Hyperspectral Unmixing. *IEEE Geoscience and Remote Sensing Letters*, 13(11):1671–1675.

[55] Jolliffe, I. T. (2002). *Principal Component Analysis*. Springer Series in Statistics. Springer-Verlag, New York.

[56] Keshava, N. and Mustard, J. F. (2002). Spectral unmixing. *IEEE Signal Processing Magazine*, 19(1):44–57.

[57] Kurz, T., Buckley, S., and Howell, J. (2013). Close-range hyperspectral imaging for geological field studies: workflow and methods. *Int. J. Remote Sens.*, 34(5):1798–1822.

[58] Lahti, S. and Hämäläinen, T. D. (2025). High-level synthesis for fpgas—a hardware engineer's perspective. *IEEE Access*, 13:28574–28593.

[59] Landgrebe, D. (2002). Hyperspectral image data analysis. *IEEE Signal Processing Magazine*, 19(1):17–28.

[60] Landgrebe, D. A. (2003). *Signal Theory Methods in Multispectral Remote Sensing*. John Wiley & Sons: New York.

[61] Lei, J., Li, Y., Zhao, D., Xie, J., Chang, C.-I., Wu, L., Li, X., Zhang, J., and Li, W. (2018). A deep pipelined implementation of hyperspectral target detection algorithm on fpga using hls. *Remote Sensing*, 10(4).

[62] Lentaris, G., Diamantopoulos, D., Stamoulias, G., Siozios, K., Soudris, D., and Aviles Rodrigalvarez, M. (2012). FPGA-based path-planning of high mobility rover for future planetary missions. In *2012 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012)*, pages 85–88.

[63] Leon, G., Gonzalez, C., Mayo, R., Mozos, D., and Quintana-Orti, E. S. (2019). Noise estimation for hyperspectral subspace identification on fpgas. *The Journal of Supercomputing*, 75(3):1323–1335.

[64] Lopez, S., Vladimirova, T., Gonzalez, C., Resano, J., Mozos, D., and Plaza, A. (2013). The promise of reconfigurable computing for hyperspectral imaging onboard systems: A review and trends. *Proceedings of the IEEE*, 101(3):698–722.

[65] Lopez Jimenez, A. C. (2006). Aplicacion de dispositivos FPGA a la instrumentacion espacial: los instrumentos GIADA y OSIRIS de la mision Rosetta.

[66] Lu, B., Dao, P. D., Liu, J., He, Y., and Shang, J. (2020). Recent Advances of Hyperspectral Imaging Technology and Applications in Agriculture. *Remote Sensing*, 12(16).

[67] Luo, B., Chanussot, J., Doute, S., and Zhang, L. (2013). Empirical automatic estimation of the number of endmembers in hyperspectral images. *IEEE Geoscience and Remote Sensing Letters*, 10(1):24–28.

[68] Mabry, D., Hansel, S., and Blake, J. (1993). The SAMPEX data processing unit. *IEEE Transactions on Geoscience and Remote Sensing*, 31(3):572–574.

[69] Macias, R., Bernabe, S., Bascones, D., and Gonzalez, C. (2022). FPGA implementation of a hardware optimized automatic target detection and classification algorithm for hyperspectral image analysis. *IEEE Geoscience and Remote Sensing Letters*, 19:1–5.

[70] Markiewicz, W., Titov, D., Ignatiev, N., Keller, H., Crisp, D., Limaye, S., Jaumann, R., Moissl, R., Thomas, N., Esposito, L., Watanabe, S., Fiethe, B., Behnke, T., Szemerey, I., Michalik, H., Perplies, H., Wedemeier, M., Sebastian, I., Boogaerts, W., Hviid, S., Dierker, C., Osterloh, B., Böker, W., Koch, M., Michaelis, H., Belyaev, D., Dannenberg, A., Tschimmel, M., Russo, P., Roatsch, T., and Matz, K. (2007). Venus Monitoring Camera for Venus Express. *Planetary and Space Science*, 55(12):1701–1711. The Planet Venus and the Venus Express Mission, Part 2.

[71] Mat Noor, N. and Vladimirova, T. (2013). Investigation into lossless hyperspectral image compression for satellite remote sensing. *Int. J. Remote Sens.*, 34(14):5072–5104.

[72] Maxfield, C. M. (2022). The Selection and Use of FPGAs for Automotive Interfacing, Security, and Compute-Intensive Loads. https://www.digikey.es/en/articles/the-selection-and-use-of-fpgas-for-automotive-interfacing.

[73] McMillan, R. (2016). Microsoft Supercharges Bing Search With Programmable Chips. https://www.wired.com/2014/06/microsoft-fpga/.

[74] Miller, G. S. (1986). The definition and rendering of terrain maps. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 39–48. ACM.

[75] Minh Nguyen for Microsemi Corporation (2018). Microsemi's RTG4™ FPGAs Achieve Qualified Manufacturers List (QML) Class V Qualification. https://www.microsemi.com/blog/2018/08/31/microsemis-rtg4-field-programmable-gate-arrays-fpgas-achieve-qualified.

[76] Musa, L. (2008). FPGAS in high energy physics experiments at CERN. In *2008 International Conference on Field Programmable Logic and Applications*, pages 2–2.

[77] Nascimento, J. M. P. and Bioucas-Dias, J. M. (2005). Vertex Component Analysis: A Fast Algorithm to Unmix Hyperspectral Data. *IEEE Transactions on Geoscience and Remote Sensing*, 43(4):898–910.

[78] Neville, R. A., Staenz, K., Szeredi, T., Lefebvre, J., and Hauff, P. (1999). Automatic endmember extraction from hyperspectral data for mineral exploration. In *Proc. Canadian Symp. Remote Sens.*, pages 21–24.

[79] Nielsen, A. (2001). Spectral Mixture Analysis: Linear and Semi-parametric Full and Iterated Partial Unmixing in Multi- and Hyperspectral Image Data. *Journal of Mathematical Imaging and Vision*, 15:17–37.

[80] Patel, A. and Kosko, B. (2008). Optimal noise benefits in neyman-pearson signal detection. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3889–3892.

[81] Paz, A., Plaza, A., and Plaza, J. (2009). Comparative analysis of different implementations of a parallel algorithm for automatic target detection and classification of hyperspectral images. In Huang, B., Plaza, A. J., and Vitulli, R., editors, *Satellite Data Compression, Communication, and Processing V*, volume 7455, page 74550X. International Society for Optics and Photonics, SPIE.

[82] Plaza, A. and Chang, C.-I. (2006). Impact of initialization on design of endmember extraction algorithms. *IEEE Transactions on Geoscience and Remote Sensing*, 44(11):3397–3407.

[83] Plaza, A., Martin, G., Plaza, J., Zortea, M., and Sanchez, S. (2011). *Recent Developments in Endmember Extraction and Spectral Unmixing*, pages 235–267. Springer Berlin Heidelberg, Berlin, Heidelberg.

[84] Plaza, A., Martinez, P., Perez, R., and Plaza, J. (2002). Spatial/spectral endmember extraction by multidimensional morphological operations. *IEEE Transactions on Geoscience and Remote Sensing*, 40(9):2025–2041.

[85] Ren, H. and Chang, C.-I. (2003). Automatic spectral target recognition in hyperspectral imagery. *IEEE Transactions on Aerospace and Electronic Systems*, 39(4):1232–1249.

[86] Shunlongwei (2021). FPGA robusta a radiacion. https://www.shunlongwei.com/es/st-collaborates-with-xilinx-to-power-radiation-hardened-fpgas/.

[87] Stuart, M. B., McGonigle, A. J. S., and Willmott, J. R. (2019). Hyperspectral imaging in environmental monitoring: A review of recent developments and technological advances in compact field deployable systems. *Sensors*, 19(14).

[88] SudoNull (2019). Hardware components of the onboard MPS of the unified strike fighter F-35. https://sudonull.com/post/27782-Hardware-components-of-the-onboard-MPS-of-the-unified-strike-fighter-F-35.

[89] Taherzadeh, E., Mansor, S. B., and Ashurov, R. (2012). Hyperspectral remote sensing of urban areas: An overview of techniques and applications.

[90] Thomas, D. B., Howes, L., and Luk, W. (2009). A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, page 63–72, New York, NY, USA. Association for Computing Machinery.

[91] Torti, E., Fontanella, A., and Plaza, A. (2018). Parallel real-time virtual dimensionality estimation for hyperspectral images. *J. Real-Time Image Process.*, 14(4):753–761.

[92] Transon, J., d'Andrimont, R., Maugnard, A., and Defourny, P. (2017). Survey of current hyperspectral earth observation applications from space and synergies with sentinel-2. In *2017 9th International Workshop on the Analysis of Multitemporal Remote Sensing Images (MultiTemp)*, pages 1–8.

[93] University of Colorado (2022). Sampex, Solar Anomalous and Magnetospheric Particle Explorer. https://lasp.colorado.edu/home/sampex/.

[94] Winter, M. E. (1999). N-FINDR: an algorithm for fast autonomous spectral end-member determination in hyperspectral data. In Descour, M. R. and Shen, S. S., editors, *Imaging Spectrometry V*, volume 3753, pages 266 – 275. International Society for Optics and Photonics, SPIE.

[95] Wu, X., Huang, B., Wang, L., and Zhang, J. (2016). GPU-Based Parallel Design of the Hyperspectral Signal Subspace Identification by Minimum Error (HySime). *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 9(9):4400–4406.

[96] Xilinx (2021). NASA Rover Exploring Mars Using Adaptive Computing Technology from Xilinx. https://www.xilinx.com/content/dam/xilinx/publications/powered-by-xilinx/xilinx-nasa-mars-case-study.pdf.

[97] Xilinx (2022a). Powering Next-Generation Automotive Systems. https://www.xilinx.com/applications/automotive.html.

[98] Xilinx (2022b). Radiation Tolerant Kintex UltraScale XQRKU060 FPGA Data Sheet. https://docs.xilinx.com/v/u/en-US/ds882-xqr-kintex-ultrascale.

[99] Zhao, J., Feng, L., Sinha, S., Zhang, W., Liang, Y., and He, B. (2020). Performance modeling and directives optimization for high-level synthesis on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(7):1428–1441.