Implementación paralela del Automatic Target Detection and Classification Algorithm haciendo uso de la ortogonalización de Gram Schmidt para el análisis de imágenes hiperespectrales

Andrés Ortiz Loaiza

GRADO EN INGENIERÍA DE COMPUTADORES. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería de Computadores

Madrid, mayo de 2019

Directores:

González Calvo, Carlos
Bernabé García, Sergio

Agradecimientos

Gracias a mis profesores por apoyar mis ideas y hacer de este proyecto algo de lo que he podido disfrutar en todo momento.

Por supuesto gracias a mi madre, la persona que siempre ha estado conmigo en todos los momentos buenos y malos de mi vida.

Índice general

Índice de figuras

Índice de tablas

Resumen

La observación remota de la Tierra ha sido siempre objeto de interés para el ser humano. A lo largo de los años los métodos empleados con ese fin han ido evolucionando hasta que, en la actualidad, el análisis de imágenes hiperespectrales constituye una línea de investigación muy activa, en especial para realizar la monitorización y el seguimiento de incendios o prevenir y hacer un seguimiento de desastres naturales, vertidos químicos u otros tipos de contaminación ambiental.

Debido a la forma en que aparecen los materiales en el entorno natural, es muy habitual que cohabiten materiales diferentes en una misma porción de espacio, por pequeña que sea ésta. Ello hace que la gran mayoría de los píxeles analizados no siempre estén constituidos por la presencia de un único material, sino que estén formados por distintos materiales puros a nivel de subpíxel.

Tradicionalmente, se utilizan para su análisis técnicas de desmezclado espectral que precisan de dos etapas complejas: la primera se basa en la extracción de firmas espectrales puras, también llamadas endmembers, y la segunda consiste en estimar el porcentaje de abundancia de dichos endmembers a nivel de subpíxel. Ambas etapas implican un alto coste computacional y esto supone un problema cuando se desea analizar imágenes hiperespectrales en tiempo real.

Los algoritmos de clasificación y detección de objetivos (targets) tienen unos principios de funcionamiento muy similares a los algoritmos de detección de endmembers y por tanto, habitualmente se utilizan para este fin a pesar de su alto coste computacional. Las FPGAs (Field Programmable Gate Array) ofrecen suficiente rendimiento para realizar este procesamiento además de flexibilidad, pequeño tamaño y bajo consumo. Todo ello sumado a que pueden ser endurecidas para su uso en el espacio hacen de ésta una opción muy viable para el objetivo que se persigue.

En este trabajo de fin de grado se lleva a cabo la implementación en FPGA del algoritmo de detección y clasificación de objetivos conocido como ATDCA-GS (Automatic Target

Detection and Classification Algorithm - Gram Schmidt), que utiliza el concepto de proyección ortogonal de un subespacio, haciendo uso de la ortogonalización de Gram Schmidt para simplificar operaciones complejas.

Para la consecución de dicho TFG, se ha realizado la implementación del algoritmo en el lenguaje de descripción hardware VHDL (Very High Speed Integrated Circuit Hardware Description Language) y además, se ha analizado y optimizado otra implementación previa bajo el paradigma de programación paralela OpenCL. Posteriormente, se han comparado ambas implementaciones optimizadas en términos de precisión y rendimiento sobre plataformas de hardware reconfigurable tipo FPGAs.

Palabras clave

Imágenes hiperespectrales, ATDCA-GS, detección de objetivos, Gram Schmidt, FPGA, VHDL, OpenCL.

Abstract

The remote observation of the Earth has always been an object of interest for the human being. Over the years, the methods used for this purpose have evolved until, at present, the analysis of hyperspectral images constitutes a very active line of research, especially to monitor fires or prevent and monitoring natural disasters, chemical discharges or other types of environmental pollution.

Due to the way in which materials appear in the natural environment, it is very common to cohabit different materials in the same portion of space, however small it may be. This means that the vast majority of pixels analyzed are not always constituted by the presence of a single material, but are formed by different pure materials at the sub-pixel level.

Traditionally, spectral demixing techniques are used for their analysis, which require two complex stages: the first is based on the extraction of pure spectral signatures, also called endmembers, and the second consists of estimating the percentage of abundance of said endmembers at the level of sub-pixel. Both stages involve a high computational cost and this is a problem when you want to analyze hyperspectral images in real time.

The algorithms of classification and detection of targets have some operating principles very similar to the detection algorithms of endmembers and therefore, they are usually used for this purpose despite their high computational cost. FPGAs (Field Programmable Gate Array) offer sufficient performance to make this processing as well as flexibility, small size and low consumption. All this together with the fact that they can be hardened for use in the space make this a very viable option for the objective pursued.

In this End-of-Degree Project, the FPGA implementation of the Automatic Target Detection and Classification Algorithm - Gram Schmidt known as ATDCA-GS, which uses the concept of orthogonal projection of a subspace, is carried out, using the orthogonalization of Gram Schmidt to simplify complex operations.

To achieve this proyect, the algorithm has been implemented in the hardware description language VHDL (Very High Speed Integrated Circuit Hardware Description Language)

and in addition, another previous implementation under the OpenCL parallel programming paradigm was analyzed and optimized. Subsequently, both implementations optimized in terms of accuracy and performance have been compared on reconfigurable hardware platforms such as FPGAs.

Keywords

Hyperspectral images, ATDCA-GS, target detection, Gram Schmidt, FPGA, VHDL, OpenCL

Capítulo 1

Introducción

1.1. Motivación

Una imagen hiperespectral es una imagen de gran resolución espectral que se obtiene a través de sensores capaces de obtener cientos o incluso miles de imágenes sobre el mismo área terrestre pero correspondientes a diferentes canales de longitud de onda. El conjunto de bandas espectrales no está limitado estrictamente al espectro visible sino que también abarca el infrarrojo y el ultravioleta.

En la actualidad, el uso de imágenes hiperespectrales está aumentando considerablemente debido al lanzamiento de nuevos satélites y el interés en la observación remota de la Tierra, que tiene utilidad en ámbitos tan diversos como defensa, agricultura de precisión, geología (detección de yacimientos minerales), valoración de impactos ambientales o incluso visión artificial.

Durante los últimos años se han producido numerosos avances con respecto a la tecnología de los sensores, lo cual ha revolucionado la obtención, la manipulación y el análisis de los datos recopilados. Esta evolución ha conseguido que se pase de tener unas decenas de bandas a tener cientos y la tendencia es que el número siga aumentando. Instituciones como National Aeronautics and Space Administration (NASA) o la European Space Agency (ESA) están continuamente obteniendo una gran cantidad de datos que necesitan ser procesados. Como consecuencia, han surgido nuevos desafíos en el procesamiento de los datos.

Si al aumento de la cantidad de información recopilada le sumamos que muchas aplicaciones actuales y futuras de observación remota requieren capacidades de procesamiento en tiempo real (en el mismo tiempo o menos que lo que tarda el satélite en capturar los datos) o cercano a este tiempo real, se hace imprescindible el uso de arquitecturas paralelas para el procesamiento eficiente [**?**] y rápido de este tipo de imágenes.

El problema principal en el procesamiento de las imágenes hiperespectrales reside en la mezcla espectral, es decir, la existencia de píxeles mezcla en los que cohabitan varios materiales distintos a nivel de subpixel. Estos píxeles conforman la mayor parte de las imágenes hiperespectrales y para su análisis es preciso el uso de algoritmos complejos de alto coste computacional, que hacen que la ejecución de los algoritmos de desmezclado sea lenta y necesite una aceleración o paralelización.

Para abordar este tipo de tareas ha sido ampliamente utilizada la computación paralela a través de procesadores multi-cores, GPUs (Graphics Processing Units) o hardware dedicado como las FPGAs (Field-Programmable Gate Arrays). De todas las alternativas estas últimas presentan una opción eficiente en cuanto a rendimiento ofreciendo tiempos reducidos, además de una menor utilización de recursos siendo de las pocas alternativas que se pueden adaptar en un sensor para poder realizar un procesamiento a bordo en misiones espaciales como Mars Pathfinder o Mars Surveyor [**?**].

Por un lado, VHDL o Verilog son las maneras nativas de programar este tipo de dispositivos, a bajo nivel y más óptimas. Por otro lado, existe una alternativa en OpenCL que permite una programación a alto nivel, más rápida y permitiendo su ejecución en una variedad de arquitecturas pero menos óptima a nivel de recursos hardware que en los dispositivos FPGAs.

1.2. Objetivos

El objetivo general de este trabajo es la implementación paralela sobre FPGA del algoritmo Automatic Target Detection and Classification Algorithm [**? ?**] haciendo uso de la

ortogonalización de Gram Schmidt y de los lenguajes de programación VHDL y OpenCL. Esto permitirá una comparativa muy interesante entre un lenguaje nativo para dicha plataforma (VHDL) y otro paradigma de programación paralela a alto nivel (OpenCL) que podrá ser portado a otras plataformas como procesadores multi-cores, GPUs u otros aceleradores.

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

- Diseño de módulos individuales en VHDL que sirvan para realizar todas las operaciones que se necesitan para la implementación del algoritmo ATDCA-GS.
- Elaboración de una máquina de estados e implementacion del algoritmo usando los modulos individuales.
- Análisis y optimización de una implementación paralela previa en OpenCL del algoritmo.
- Obtención de resultados y realización de comparativas de rendimiento entre ambos lenguajes de programación.

1.3. Organización de esta memoria

Teniendo presentes los anteriores objetivos concretos, se procede a describir la organización del resto de esta memoria, estructurada en una serie de capítulos cuyos contenidos se describen a continuación:

- **Análisis hiperespectral**: se define el concepto de imagen hiperespectral y el modelo lineal de mezcla; se mencionan algunos sensores hiperespectrales (AVIRIS y EO-1 Hyperion) y algunas bibliotecas espectrales (USGS y ASTER); y por último, se presenta la necesidad de paralelización y las plataformas que se pueden utilizar para afrontar el problema de mejora de rendimiento.

- **Tecnología de las FPGAs**: se define de forma breve las tecnologías de las FPGAs.
- **Implementación**: se define el algoritmo ATDCA-GS en serie y se explica la paralelización y optimización que se ha llevado a cabo tanto en VHDL como en OpenCL.
- **Resultados**: se presentan los resultados obtenidos tras la implementación y ejecución del algoritmo en dispositivos FPGAs.
- **Conclusiones y trabajo futuro**: se presentan las principales conclusiones de los aspectos abordados en el trabajo a las que se han llegado y también algunas posibles líneas de trabajo futuro que se pueden desempeñar con relación al presente trabajo.

Capítulo 2

Análisis hiperespectral

2.1. Imágenes hiperespectrales

Una imagen hiperespectral (o imagen de gran resolución espectral) es una imagen en la que cada punto (píxel) viene descrito por un conjunto (vector) de valores espectrales que se corresponden con la reflectancia de dicho píxel en diferentes longitudes de onda a estrechas bandas del espectro [**?**]. Cada uno de esos vectores conforman lo que se denomina "firma espectral" [**? ?**] del píxel en cuestión, y se obtienen gracias a los sensores hiperespectrales en diversos canales espectrales. El número de bandas que se utilizan suele variar desde las decenas hasta los varios centenares, y el conjunto no está limitado estrictamente al espectro visible sino que también abarca el infrarrojo y el ultravioleta.

Se puede ver como un caso particular de las imágenes hiperespectrales a las imágenes RGB [**?**], las más comunmente utilizadas en el día a día. Ellas están compuestas por tres bandas de color y cada una corresponde a una longitud de onda concreta: rojo, verde y azul, respectivamente. Para componer una imagen RGB a partir de una imagen hiperespectral, se pueden seleccionar esas tres longitudes de onda concretas o las más aproximadas que se encuentren.

Para el estudio de imágenes hiperespectrales se hace uso de la propiedad de reflectancia espectral [**?**], que es el porcentaje de energía reflejada sobre la energía incidente como una función de la longitud de onda. La reflectancia es característica de cada material porque

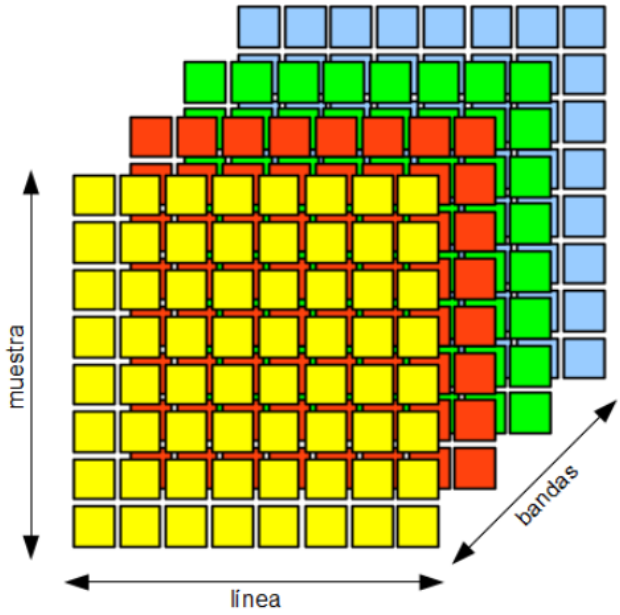


Figura 2.1: *Ejemplo de cubo hiperespectral.*

varía en función de las distintas longitudes de onda para la mayoría de los materiales dado que la energía puede ser absorbida o reflejada en distintos grados. Por ello, el análisis de estas imágenes sirve también para identificar un material a partir de su firma espectral.

Una imagen hiperespectral puede ser representada como un cubo (denominado "hipercubo" o "cubo hiperespectral") en el que los ejes X, denominado "líneas" (lines), e Y, denominado "muestras" (samples), se utilizan para mostrar la ubicación espacial de cada píxel de la imagen. El tercer eje Z, denominado "bandas" (bands), se utiliza para mostrar la reflectancia de cada píxel en cada uno de los canales espectrales que haya obtenido el sensor. En la Figura 2.1 [**?**] se muestra un ejemplo de cubo hiperespectral, en el que se pueden observar las diferentes bandas de longitudes de onda por separado.

Como se puede observar en la Figura 2.2 [**?**], existen dos tipos de píxeles en función de sus firmas espectrales. Si su firma espectral está constituida por un único tipo de material, el píxel se denomina píxel puro o endmember. Si, por el contrario, está constituida por varios materiales distintos a nivel de subpíxel, el píxel se denomina píxel mezcla. Estos últimos serán los que conformen la mayor parte de la imagen hiperespectral, ya que es muy

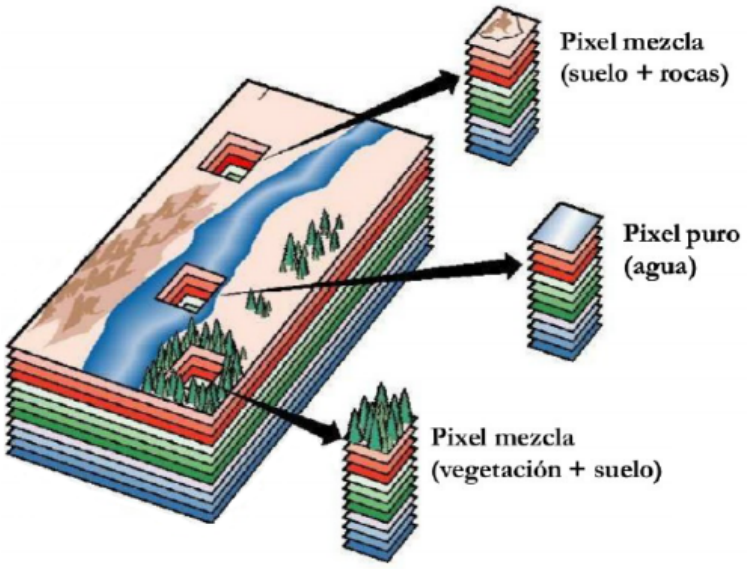


Figura 2.2: *Ejemplo de píxeles puros y píxeles mezcla.*

habitual que cohabiten materiales diferentes en una misma porción de espacio, incluso a nivel microscópico. [**?**].

Otro de los aspectos importantes a destacar de este tipo de imágenes es la resolución espectral [**?**], que es la separación entre longitudes de onda (a menor separación, mayor resolución espectral) y viene determinada por la tecnología del sensor que se emplea. Cuanto mayor sea la resolución espectral, más información se obtendrá y se podrán esperar mejores resultados en el análisis de las imágenes. Sin embargo, es conveniente tener en cuenta que esta ventaja supone también un aumento considerable del coste computacional.

Las imágenes hiperespectrales surgieron a raíz de las imágenes multiespectrales [**?**]. Este tipo de imágenes se diferencia de las anteriores en que la separación entre bandas es muy grande y la imagen dispone de un número muy pequeño de las mismas (de 4 a 20 aproximadamente). A pesar de basarse en los mismos conceptos, la información proporcionada por las imágenes multiespectrales es mucho más limitada y, por tanto, sus aplicaciones y su método de procesamiento son diferentes [**?**].

2.2. Sensores hiperespectrales

El análisis de la Tierra nunca ha dejado de despertar interés o de ser objeto de estudio. Su observación remota se lleva realizando desde hace más de un centenar de años, tradicionalmente empleando cámaras instaladas en globos dirigibles o satélites. No fue hasta el inicio de los años 90 cuando comenzó a surgir la tecnología que serviría para desarrollar los sensores hiperespectrales.

Para obtener las imágenes hiperespectrales, los sensores se basan en la espectroscopia [**?**], que es el estudio de la luz emitida o reflejada por los materiales y su variación de energía con la longitud de onda. Esto se consigue empleando unos instrumentos llamados espectrómetros de imágenes, que son capaces de realizar mediciones espectrales de bandas muy próximas entre sí.

El concepto de imagen hiperespectral tuvo su origen en el Jet Propulsion Laboratory[1], una misión comercial de la NASA encargada del desarrollo de instrumentación para la adquisición de imágenes hiperespectrales, esto es, sensores de gran resolución espacial. Un ejemplo de esta instrumentación es el sensor Airborne Visible Infra-Red Imaging Spectrometer (AVIRIS)[2].

2.2.1. Sensor AVIRIS

AVIRIS es un sensor óptico creado en 1987 capaz de captar imágenes hiperespectrales de 224 canales espectrales (bandas) con longitudes de onda que varían entre 400 y 2500 nanómetros. Tiene la capacidad de obtener información con un ancho entre bandas (resolución espectral nominal) de 10 nanómetros. Está pensado para ser aerotransportado y analizar zonas del espectro visible e infrarrojo [**? ?**].

Este sensor ha realizado tomas de imágenes hiperespectrales en Europa, Estados Unidos, Canadá, Argentina y otras partes de América del Sur mediante el uso de cuatro plataformas

[1] https://www.jpl.nasa.gov
[2] https://aviris.jpl.nasa.gov

de aviones: el jet ER-2, perteneciente al Jet Propulsion Laboratory de la NASA, que vuela a aproximadamente 20 kilómetros sobre el nivel del mar con una velocidad de unos 730 km/h; el turbopropulsor Twin Otter International, desarrollado por la compañía canadiense Havilland Canada, que vuela a 4 kilómetros sobre el nivel del suelo a una velocidad de 130 km/h; el Proteus de Scaled Composites; y el WB-57 de la NASA.

El objetivo principal del proyecto AVIRIS es identificar, medir y monitorizar los componentes de la superficie y la atmósfera terrestre basándose en la absorción molecular y las firmas de dispersión de partículas. Se trata de una investigación centrada principalmente en comprender los procesos relacionados con el medio ambiente y el cambio climático.

2.2.2. Sensor EO-1 Hyperion

Este sensor fue lanzado con éxito en noviembre del año 2000 gracias al programa New Millenium[3] de la NASA. Es capaz de obtener información con una resolución espacial de 30 metros y en 220 bandas espectrales, cubriendo un rango de longitudes de onda de entre 400 y 2500 nanómetros. Cada línea de datos en las imágenes hiperespectrales tomadas está constituida por 256 píxeles.

2.3. Bibliotecas espectrales

Existen bibliotecas espectrales que facilitan el tratamiento y el análisis de imágenes hiperespectrales. Esto se debe a que contienen una elevada cantidad de datos de reflectancia espectral tanto de materiales naturales como desarrollados por el ser humano. A continuación se nombran algunas de ellas.

2.3.1. Biblioteca espectral USGS

Esta biblioteca fue desarrollada por el laboratorio de espectrometría United States Geological Survey (USGS)[4], en Colorado. Contiene información de en torno a 500 reflectancias

[3] http://nmp.nasa.gov
[4] https://www.usgs.gov

espectrales, principalmente de minerales, sobre longitudes de onda que varían entre 0.2 y 3.0 micrómetros.

2.3.2. Biblioteca espectral ASTER

Esta biblioteca, impulsada gracias a la NASA, fue desarrollada por el programa Advanced Spaceborne Thermal Emission and Reflectance Radiometer (ASTER)[5]. Contiene datos referentes a, aproximadamente, 2000 reflectancias espectrales en un rango de longitudes de onda de entre 0.4 y 14 micrómetros. Dicha información fue recogida a partir de materiales tales como minerales, agua, nieve y otros fabricados por el ser humano, y la tarea se llevó a cabo por el Jet Propulsion Laboratory de la NASA, Johns Hopkins University y United States Geological Survey.

2.4. Modelo lineal de mezcla

Para la realización del desmezclado espectral [**?**] en el análisis de imágenes hiperespectrales, existen numerosos algoritmos [**?**] y dos modelos principales. El más utilizado actualmente es el modelo lineal de mezcla, que supone que los espectros recogidos en cada pixel mezcla pueden ser representados mediante una combinación lineal de firmas espectrales puras (endmembers). Esta aproximación asume que los componentes que residen a nivel de sub-pixel aparecen separados en el espacio, lo que hace que los fenómenos de absorción y reflexión de la radiación electromagnética incidente puedan ser caracterizados siguiendo un patrón estrictamente lineal y que los efectos producidos por las dispersiones sean mínimos.

La otra aproximación es el modelo no lineal de mezcla, que tiene en cuenta también aquellos endmembers que se distribuyen al azar a lo largo del campo de visión del sensor, por lo que proporciona una mejor información de la imagen. Sin embargo, para ello requiere información previa acerca de las propiedades físicas de los materiales que conforman la imagen. La Figura 2.3 [**?**] se presenta una comparativa de los dos posibles modelos de

[5] http://asterweb.jpl.nasa.gov

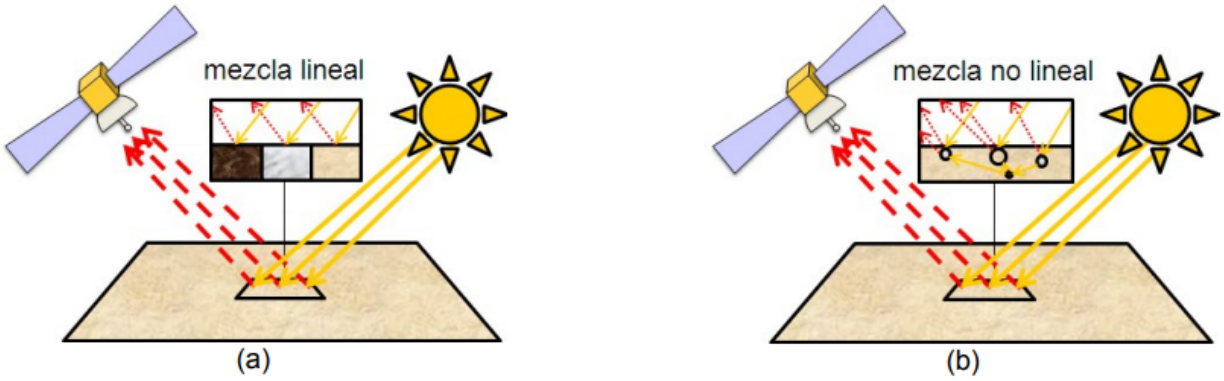


Figura 2.3: *Comparativa entre modelos de desmezclado espectral.*

desmezclado espectral. Como el modelo no lineal no es aplicable a zonas sobre las que no se posee ninguna información, en este trabajo se opta por utilizar el modelo lineal de mezcla.

Como se puede observar en la Figura 2.4 [**?**], el modelo lineal se puede interpretar gráficamente mediante un espacio bidimensional, utilizando un diagrama de dispersión entre dos bandas de la imagen poco relacionadas entre sí. Todos los puntos de la imagen quedan englobados dentro del área formada por los puntos más extremos, es decir, los elementos espectralmente más puros, que serán los mejores candidatos para ser seleccionados como endmembers.

De esta manera se forman sistemas de coordenadas con origen en el centroide de la nube de puntos y los vectores que van de ese punto a los endmembers. Así, cualquier punto de la imagen puede expresarse como combinación lineal de los endmembers entre los que se encuentre. El paso clave a la hora de aplicar el modelo lineal de mezcla consiste, pues, en identificar de forma correcta los elementos extremos de la nube de puntos N-dimensional. Para ello existen numerosos algoritmos de extracción de endmembers, como el Automatic Target Detection and Classification Algorthm empleado en el presente trabajo.

Se ha optado por el algoritmo ATDCA-GS ya que en trabajos previos se ha hecho uso de arquitecturas CPUs o GPUs y el rendimiento obtenido ha sido suficiente para poder realizar un procesamiento en tiempo real, además de que puede mejorar la precisión si se compara con el algoritmo ATDCA-OSP. También se trata de un algoritmo sencillo pero al mismo

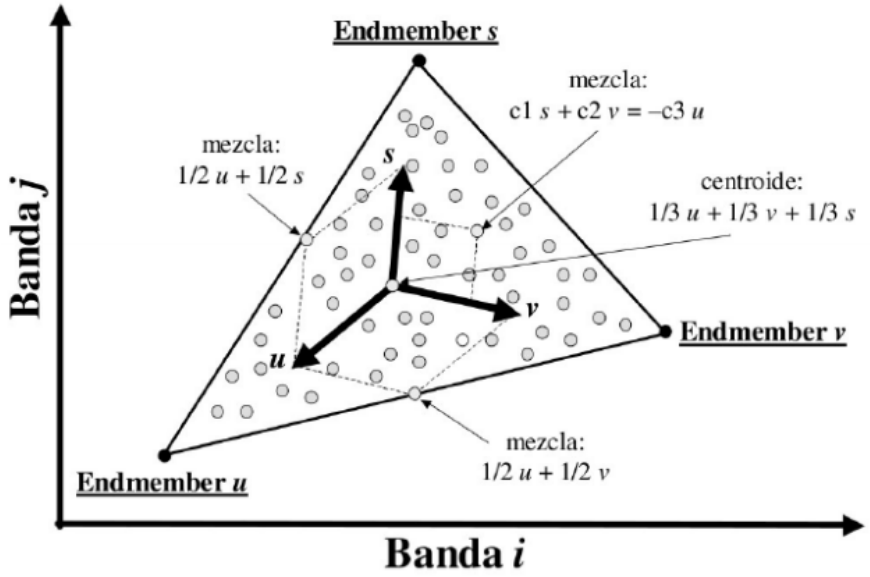


Figura 2.4: *Representación gráfica del modelo lineal de mezcla.*

tiempo bastante robusto, con operaciones no muy costosas en términos computacionales. Por ejemplo, si de nuevo lo comparamos con el algoritmo ATDCA-OSP, la operación de calcular la inversa de una matriz para obtener las proyecciones ortogonales se sustituyen por la optimización del método de Gram Schmidt, de manera que la complejidad y el tiempo de ejecución se reducen considerablemente.

2.5. Necesidad de paralelización

La mayoría de las técnicas de análisis hiperespectral desarrolladas hasta la fecha, incluyendo la utilizada en este trabajo, tienen en cuenta que gran parte de los píxeles obtenidos por el sensor (píxeles mezcla) vienen dados por la contribución de diversos materiales que residen a nivel de sub-píxel. Por ello, son necesarios diseños capaces de modelar este fenómeno de manera adecuada, lo cual supone un coste computacional considerablemente alto.

Además, la elevada cantidad de datos que se obtienen de cada imagen hiperespectral (las bandas puede ser de cientos o incluso miles de longitudes de onda diferentes) no sólo dificulta la manera en que estos deben ser almacenados y procesados en un ordenador determinado,

sino que añade la complicación de que esta información debe ser enviada previamente por el satélite que se ha encargado de la toma de las imágenes.

Existen posibles soluciones al problema de la alta dimensionalidad de los datos [**?**]. Por un lado, se podría realizar una compresión de datos con pérdidas (mayor compresión pero menos información) o sin pérdidas (menor compresión pero manteniendo la información) [**?**]. Por otro lado, se podría recoger de manera selectiva determinadas bandas espectrales en base a los materiales sobre los que se requiera información, por ejemplo, minerales y rocas en zonas mineras; o vegetación y agua en zonas de selva o bosque.

Debido a lo anterior, unido a que muchas aplicaciones de observación remota requieren capacidades de procesamiento en tiempo real, se hace imprescindible el uso de arquitecturas paralelas para el procesamiento eficiente y rápido de este tipo de imágenes. Esta tarea resulta relativamente sencilla puesto que al analizar las imágenes se hace uso de operaciones matriciales cuyo carácter repetitivo las hace altamente susceptibles de ser implementadas en arquitecturas paralelas.

En la actualidad, existen múltiples alternativas hardware para la computación paralela: procesadores multi-cores, GPUs o hardware dedicado como los Circuitos Integrados de Aplicación Específica (ASICs) o las FPGAs. De todas ellas, estas últimas presentan una opción eficiente en cuanto a rendimiento haciendo que los tiempos de respuesta sean reducidos y que el algoritmo se ejecute de manera más eficiente. También utiliza una cantidad menor de recursos, por lo que es una de las pocas alternativas que se pueden adaptar en un sensor para poder realizar un procesamiento a bordo.

Capítulo 3

Tecnología de las FPGAs

Las FPGAs (Field Programmable Gate Arrays) [**?**] son dispositivos hardware programables y reconfigurables de bajo consumo que ofrecen un gran equilibrio entre flexibilidad y eficiencia, motivos por los cuales han sido elegidas para la implementación del algoritmo en este trabajo. Consisten en una matriz de bloques lógicos o LB (Logic Blocks) y una red de interconexión que pueden ser configurados haciendo uso de dispositivos anti–fuse [**?**] o mediante un mapa de bits (bitstream) [**?**] según la funcionalidad que se requiera.

Los LBs interconectados mediante esa red contienen típicamente uno o varios circuitos combinacionales programables Look-Up Table (LUT), uno o varios biestables, lógica adicional y las celdas de memoria SRAM requeridas para la configuración de todos los elementos. Las tareas de entrada/salida se realizan en la periferia del dispositivo mediante LBs o mediante bloques específicos denominados Input–Output–Blocks (IOBs). La figura 3.1 [**?**] muestra la estructura interna simplificada de una FPGA.

Actualmente, se integran también de forma habitual recursos heterogéneos como bloques de memoria RAM dedicada [**?**], multiplicadores, interfaces para periféricos, DSPs (Digital Signal Processor) e incluso microprocesadores. Con la integración de estos elementos arquitectónicos en FPGAs que disponen de una sección de lógica programable por el usuario surge el concepto de Arquitecturas Híbridas [**?**].

El diseño con FPGAs se lleva a cabo mediante un proceso automático conocido como síntesis de alto nivel. Los algoritmos o las tareas deben ser descritos en un lenguaje de alto

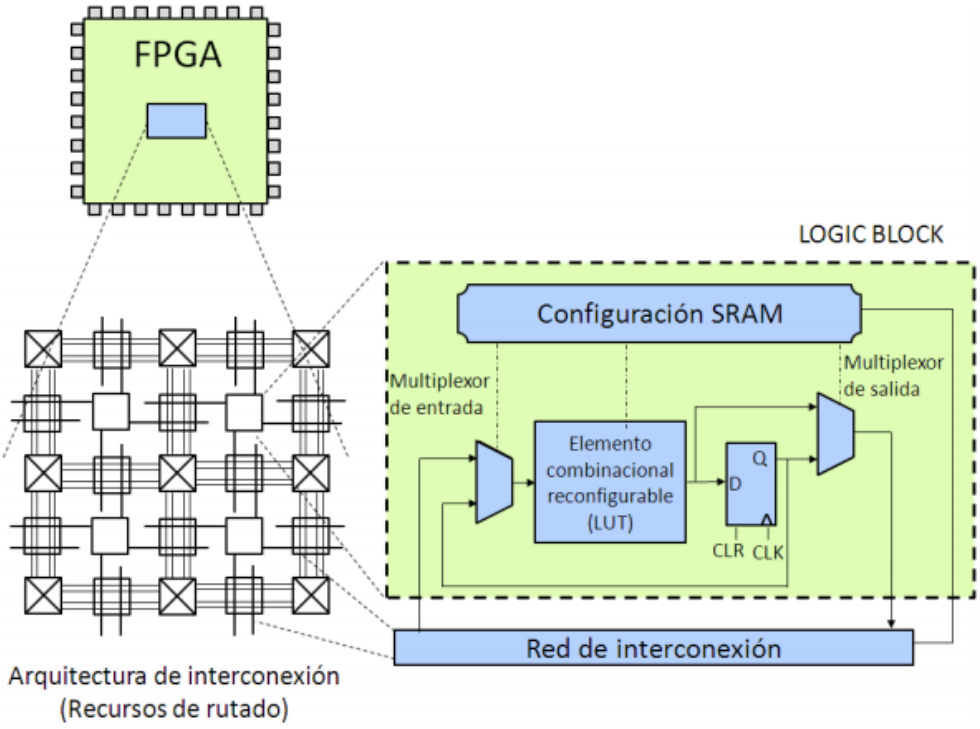


Figura 3.1: *Modelo genérico de una FPGA.*

nivel que esté directamente relacionado con elementos hardware para su fácil traducción a un circuito. En los últimos años se ha utilizado mayormente los lenguajes de descripción hardware (HDL, Hardware Description Language), entre los que destacan Verilog y VHDL.

El primer paso para sintetizar una tarea a hardware es describir el algoritmo en alguno de los HDLs disponibles (por ejemplo, VHDL). El siguiente paso es determinar qué tipo de bloques hardware son necesarios y cómo están conectados entre sí para después, en el tercer paso, asignar bloques básicos configurables concretos de la FPGA (CLBs) y el rutado de señales entre ellos. En el cuarto paso del proceso se obtiene el mapa de bits de configuración necesario para que los LBs utilizados en el diseño del circuito realicen cada uno la funcionalidad necesaria. Por último, es necesario escribir dicho mapa de bits en la memoria de configuración del dispositivo.

Las ventajas que presentan las FPGAs son el aumento de la velocidad de procesamiento [**?**], ya que, además de presentar la posibilidad real de paralelización, una ejecución hardware

es habitualmente más rápida que el código software; la reducción del consumo, ya que se pueden aplicar técnicas para evitar el consumo innecesario; flexibilidad [**?**], ya que existe la posibilidad de reconfigurar los elementos básicos; y el coste reducido debido al aumento de su popularidad y a los avances en la fabricación de su tecnología.

Por otra parte, existen dos aspectos importantes a tener en cuenta [**?**]: uno es el rutado de las señales, ya que el obtenido tras la compilación no suele ser el más eficiente y suele ser necesaria una revisión manual por parte del diseñador. La solución por la que se opta para mejorar este aspecto es el diseño 3D [**?**]. El otro aspecto es el elevado tiempo de configuración debido al gran tamaño que suele tener el mapa de bits necesario. Ya se están dirigiendo numerosos esfuerzos en nuevas arquitecturas o tecnologías que permitan paliar este efecto.

Capítulo 4

Implementación

4.1. Algoritmo ATDCA-GS en serie

Anteriormente se ha presentado el problema que supone la existencia de firmas espectrales mixtas en varios píxeles de las imágenes hiperespectrales. La primera etapa del desmezclado espectral permite obtener los endmembers o firmas espectrales puras presentes, lo cual es necesario en la segunda etapa para el procesamiento posterior de la imagen y para la obtención de la presencia cuantitativa de dichos endmembers en cada pixel [**?**].

Como alternativa a la extracción de endmembers en la primera etapa, existen algoritmos orientados a la detección de targets, como es el caso del algoritmo ATDCA-GS implementado en este trabajo. Los targets son objetivos con características espectrales muy diferentes entre sí. Así, el conjunto de targets de una imagen ofrece una colección de firmas espectrales representativa de los distintos elementos que se pueden encontrar en la imagen.

Se puede realizar una similitud entre targets y endmembers puesto que estos últimos también representan una colección de firmas espectrales muy diferenciadas; sin embargo, es preciso destacar que los endmembers están asociados a firmas espectrales estrictamente puras mientras que los targets son una muestra representativa de los elementos presentes en la imagen.

Generalmente los algoritmos de detección de targets suelen funcionar de manera iterativa, obteniendo cada target en una iteración del algoritmo y, como resultado tras la ejecución,

el conjunto de targets detectados. En algoritmos como el ATDCA también se requiere al comienzo una inicialización del conjunto de targets, siendo el píxel más brillante de la imagen (el de mayor intensidad) el más utilizado como primer target. Pese a no ser la única alternativa viable, se ha demostrado experimentalmente que dicho píxel siempre figura en el conjunto de píxeles de la solución [**?**], por lo que supone una elección correcta.

El algoritmo ATDCA usa el concepto de proyección ortogonal de un subespacio para hallar los targets en la imagen. Después de inicializarlo con el pixel más brillante, el siguiente objetivo será aquel con el valor de proyección más alto y será incluido en el subespacio para repetir el proceso tantas veces como targets se quieran detectar. Para optimizar el rendimiento del algoritmo, se ha empleado la ortogonalización de Gram Schmidt, que permite sustituir operaciones complejas, como invertir una matriz, por operaciones más simples y menos costosas en computación. En el Algoritmo 1 se muestra una implementación de este algoritmo en pseudocódigo.

4.2. Paralelización y optimización del algoritmo ATDCA-GS usando VHDL

Para la implementación en VHDL, se ha construido el módulo descrito gráficamente en la Figura 4.1. A continuación se muestra una secuencia de pasos del funcionamiento del algoritmo en VHDL y en las siguientes subsecciones se mostrarán los sub-componentes que se han empleado y una breve descripción de los mismos.

Usando como referencia el Algoritmo 1, el código VHDL comienza utilizando los módulos Producto Escalar (línea 4) y Píxel Más Distinto (líneas 5 a 8) para calcular el píxel más brillante, esto es, el primer target de la solución.

Acto seguido se procede a calcular el segundo target empleando los módulos Producto Escalar Auxiliar en modo sólo suma y Divisor en la línea 17; y los módulos Producto Escalar Auxiliar en modo sólo suma, Producto Escalar y Divisor en la línea 19. Después de realizar estos cálculos, procede a calcular con ellos el píxel más distinto utilizando los módulos

Algorithm 1 Algoritmo ATDCA-GS

```
# Cálculo del píxel más brillante
max = 0
for i = 0 to r − 1 do
    brillo = H[:,i] * H[:,i]^T
    if brillo > max then
        max = brillo
        pos = i
    end if
end for
U[:,0] = H[:,pos]
P[0] = pos

# Cálculo del resto de targets
for i = 0 to t − 2 do
    UC = U[:,0..i]
    if i == 0 then
        f[:,0] = sum_{k=0}^{nb-1} UC[k, 0] / UC[nb,0]
        u[:,0] = UC[:,0]
        c2[0] = sum_{k=0}^{nb} u[k, 0] / (u[:,0]^T * u[:,0])
    else
        c1 = (u[:,0..(i-1)]^T * UC[:,i]) / (u[:,0..(i-1)] * u[:,0..(i-1)])
        u[:, i] = UC[:,i] - sum_{k=0}^{i-1}(c1 * u[:, k])
        c2 = sum_k(u[:, k]) / (u[:,i]^T * u[:,i])
        f[:,i] = w - sum_{k=0}^{i}(c2 * u[:, k])
    end if

    max = 0
    # Cálculo del píxel más diferente
    for j = 0 to r − 1 do
        result = f[:,i] * H[:,j]
        val = result^T * result
        if val > max then
            max = val
            pos = j
        end if
    end for
    U[:,i+1] = H[:,pos]
    P[i+1] = pos
end for
```

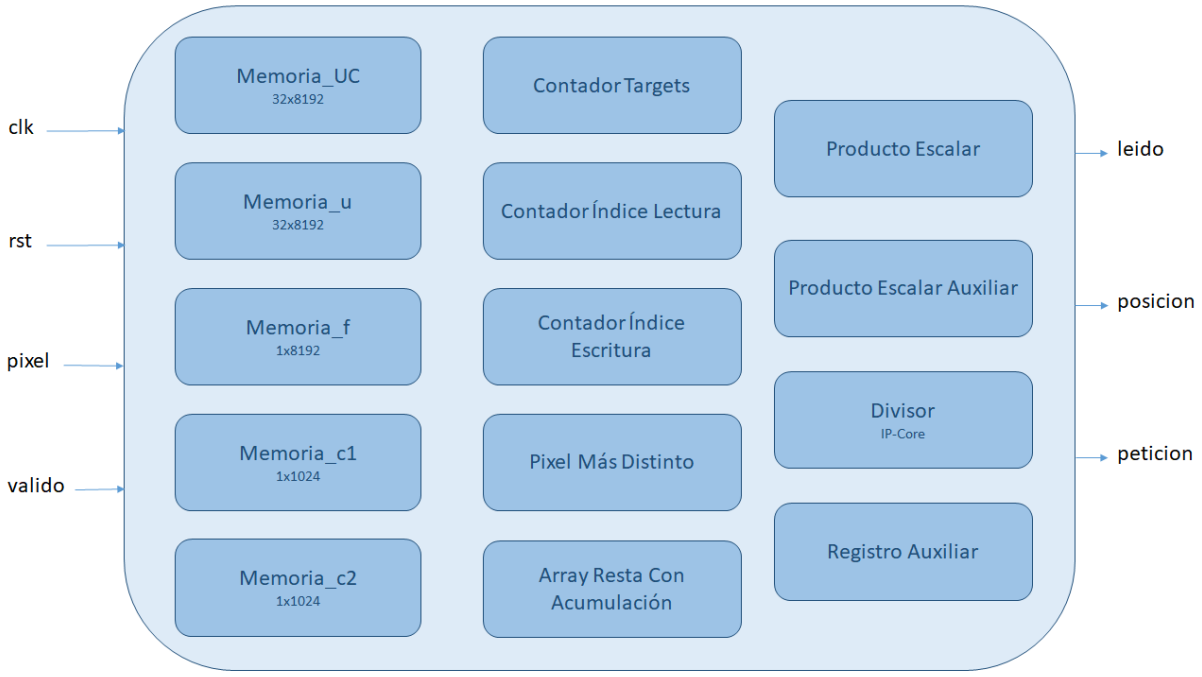


Figura 4.1: *Representación gráfica del módulo ATDCA-GS implementado en VHDL.*

Producto Escalar Auxiliar en modo sólo multiplicación (línea 30) y Píxel Más Distinto (líneas 31 a 35).

Una vez obtenidos los dos primeros targets, el resto se calcularán haciendo uso de los módulos Producto Escalar, Producto Escalar Auxiliar en modo producto escalar y Divisor (línea 21); los módulos Array Resta Con Acumulación y Producto Escalar Auxiliar en modo sólo multiplicación (línea 22); los módulos Producto Escalar Aux en modo sólo suma, Registro Auxiliar, Producto Escalar y Divisor (línea 23); y los módulos Array Resta Con Acumulación y Producto Escalar Auxiliar en modo sólo multiplicación (línea 24). Después de realizar estos cálculos, se procede a calcular con ellos el píxel más distinto de la misma manera que ocurría para calcular el segundo target.

4.2.1. IP-Cores

Un IP-Core es un componente básico ya implementado que se encuentra incorporado en la herramienta IP-Core Generator dentro del software de desarrollo Xilinx ISE. La principal

ventaja que presentan radica en que han sido específicamente diseñados para ser utilizados en dicha herramienta y, por consiguiente, ya han sido completamente optimizados. Se emplean durante el diseño de algoritmos para facilitar enormemente la construcción de operaciones matemáticas complejas y bloques de memoria.

A lo largo de la programación en VHDL del algoritmo ATDCA-GS se han utilizado algunos de los siguientes bloques IP-Core, cuya configuración y descripción se mostrará a continuación.

Operadores de punto flotante

Estos componentes hacen uso de una señal de entrada llamada operation_nd, que indica al módulo que los operadores ya están disponibles y debe comenzar a realizar los cálculos. También hacen uso de una señal de salida llamada rdy (ready), que indica que el componente ha terminado de operar y el resultado está ya disponible.

Estos módulos se han utilizado para trabajar con números en punto flotante de precisión simple (32 bits) y están segmentadas, es decir, admiten la entrada de una secuencia consecutiva de datos. Tras una latencia inicial, se irá produciendo un resultado en cada ciclo y en el mismo orden que llevaba la secuencia de entrada.

- Restador (restadorPF32Bits.xco)
    - IP-core: Floating-point.
    - Functions: Add/Substract (Substract).
    - Floating-point precision of A/B inputs: single (32 bits).
    - Architecture optimizations: Low latency.
    - Family optimizations: No usage (= Logic only).
    - Latency: 0.
    - Cycles per operation: 1.

- Divisor (div_escalar_PF32Bits.xco)
    - IP-core: Floating-point.
    - Functions: Divide.
    - Floating-point precision of A/B inputs: single (32 bits).
    - Architecture optimizations: High speed.
    - Family optimizations: No usage (= Logic only).
    - Latency: Use maximum latency (28).
    - Cycles per operation: 1.
- Comparador (cmp_mayor.xco)
    - IP-core: Floating-point.
    - Functions: Compare (Greater than).
    - Floating-point precision of A/B inputs: single (32 bits).
    - Architecture optimizations: High speed.
    - Family optimizations: No usage (= Logic only).
    - Latency: 0.
    - Cycles per operation: 1.
- Sumador (sumadorPF32Bits.xco)
    - IP-core: Floating-point.
    - Functions: Add/Substract (Add).
    - Floating-point precision of A/B inputs: single (32 bits).
    - Architecture optimizations: High speed.
    - Family optimizations: Full usage (= 2 x DSP48E).

- Latency: 2.
- Cycles per operation: 1.

- Multiplicador (multiplicadorPF32Bits.xco)
  - IP-core: Floating-point.
  - Functions: Multiply.
  - Floating-point precision of A/B inputs: single (32 bits).
  - Architecture optimizations: High speed.
  - Family optimizations: Full usage (= 2 x DSP48E).
  - Latency: 1.
  - Cycles per operation: 1.

Bloques de memoria

Estos componentes son bloques de memoria que permiten seleccionar un tamaño y unas características concretas tales como los tipos de puertos o la prioridad de lectura o escritura. De esta manera se han podido diseñar bloques de memoria específicos según las necesidades del algoritmo. En concreto, se han utilizado 4 bloques de memoria distintos:

- Memoria 2x4096 2 True Dual Port (memoria2x4096_WRITE_FIRST.xco)
  - IP-core: Block Memory Generator.
  - Memory type: True Dual Port RAM.
  - Port A - Read/Write width: 4096.
  - Port A - Read/Write depth: 2.
  - Port A - Operating mode: Write first.
  - Port B - Read/Write width: 4096.

- Port B - Read/Write depth: 2.
- Port B - Operating mode: Write first.

- Memoria 1x1024 (mem1x1024.xco)
  - IP-core: Block Memory Generator.
  - Memory type: Simple Dual Port RAM.
  - Common clock.
  - Port A - Write width: 1024.
  - Port A - Write depth: 2.
  - Port A - Operating mode: Read first.
  - Port B - Read width: 1024.
  - Port B - Read depth: 2.
  - Port B - Operating mode: Read first.
  - Fill Remaining Memory Locations: 0 (Hex).

- Memoria 2x4096 Simple Dual Port (memoria2x4096.xco)
  - IP-core: Block Memory Generator.
  - Memory type: Simple Dual Port RAM.
  - Port A - Write width: 4096.
  - Port A - Write depth: 2.
  - Port A - Operating mode: Write first.
  - Port B - Read width: 4096.
  - Port B - Read depth: 2.
  - Port B - Operating mode: Write first.

- Memoria 32x4096 (memoria.xco)
    - IP-core: Block Memory Generator.
    - Memory type: Simple Dual Port RAM.
    - Port A - Write width: 4096.
    - Port A - Write depth: 32.
    - Port A - Operating mode: Write first.
    - Port B - Read width: 4096.
    - Port B - Read depth: 32.
    - Port B - Operating mode: Write first.
    - Fill Remaining Memory Locations: 0 (Hex).

4.2.2. Módulos VHDL

A continuación se definen los módulos VHDL que se han implementado para este trabajo.

Memoria 32x8192 (mem_32x8192.vhd)

- DESCRIPCIÓN
    - Memoria de doble puerto simple, es decir, permite realizar una escritura (puerto A) y una lectura (puerto B) simultáneamente. Combina dos memorias de 32x4096 para generar una de 32x8192. Tiene una política de Write First, lo que significa que la lectura garantiza que el dato leído esté siempre actualizado. Está estructurada para almacenar datos de 8192 bits, que sirven para representar un máximo de 256 bandas de 32 bits cada una. Esos 32 bits son los necesarios para representar un número en punto flotante de precisión simple. Tiene 32 filas, una por cada target que se vaya a detectar y almacenar durante la ejecución del algoritmo. No se esperan más de 32 como máximo.

- INPUT
    - clka: señal de reloj del puerto A.
    - wea: señal Write Enable que indica si se debe realizar una escritura (1) o no (0).
    - addra: señal de 5 bits que indica la dirección de memoria sobre la que va a trabajar el puerto A.
    - dina: señal de 8192 bits que indica el dato que entra por el puerto A.
    - clkb: señal de reloj del puerto B.
    - addrb: señal de 5 bits que indica la dirección de memoria sobre la que va a trabajar el puerto B.
- OUTPUT
    - doutb: señal de 8192 bits que indica el dato que sale por el puerto B.
- UTILIZADO EN
    - Memoria UC.
    - Memoria u.

Memoria 1x8192 (mem_1x8192.vhd)

- DESCRIPCIÓN
    - Memoria de doble puerto simple, es decir, permite realizar una escritura (puerto A) y una lectura (puerto B) simultáneamente. Combina dos memorias de 2x4096 para generar una de 1x8192. Tiene una política de Write First, lo que significa que la lectura garantiza que el dato leído esté siempre actualizado. Está estructurada para almacenar datos de 8192 bits, que sirven para representar un máximo de 256 bandas de 32 bits cada una. Esos 32 bits son los necesarios para representar un número en punto flotante de precisión simple. Tiene 2 filas pero sólo se utilizará una. Esta fila almacenará la proyección calculada en cada iteración del algoritmo.

- INPUT
    - clka: señal de reloj del puerto A.
    - wea: señal Write Enable que indica si se debe realizar una escritura (1) o no (0).
    - addra: señal de 1 bit que indica la dirección de memoria sobre la que va a trabajar el puerto A.
    - dina: señal de 8192 bits que indica el dato que entra por el puerto A.
    - clkb: señal de reloj del puerto B.
    - addrb: señal de 1 bit que indica la dirección de memoria sobre la que va a trabajar el puerto B.
- OUTPUT
    - doutb: señal de 8192 bits que indica el dato que sale por el puerto B.
- UTILIZADO EN
    - Memoria f.

Memoria c (memoriac.vhd)

- DESCRIPCIÓN
    - Memoria de doble puerto simple, es decir, permite realizar una escritura (puerto A) y una lectura (puerto B) simultáneamente. Tiene una política de Read First, lo que significa que la lectura garantiza que el dato leído no sea el sobreescrito por la escritura. Está estructurada para almacenar datos de 32 bits agrupados dentro de una única fila de 1024 bits. Por tanto, se representan 32 números (número máximo de targets) de 32 bits cada uno. Esos 32 bits son los necesarios para representar un número en punto flotante de precisión simple. Tiene 2 filas pero sólo se utilizará una. Esta fila almacenará el cálculo generado para el target que

corresponda en cada momento en función de la iteración del algoritmo en la que nos encontremos.

- INPUT
  - clk: señal de reloj del puerto A.
  - wea: señal Write Enable que indica si se debe realizar una escritura (1) o no (0).
  - pos: señal de 5 bits que indica qué 32 bits de la única fila de la memoria deben ser modificados en la escritura.
  - dina: señal de 32 bits que indica el dato que entra por el puerto A.
- OUTPUT
  - doutb: señal de 1024 bits que indica el dato que sale por el puerto B.
- UTILIZADO EN
  - Memoria c1.
  - Memoria c2.

Contador (contador_rst.vhd)

- DESCRIPCIÓN
  - Contador módulo n comenzando desde 0. En cada flanco de reloj, si la señal count_up está a 1, se suma 1 al valor actual del contador. Si está a 0, no se produce ningún cambio. Por la señal dout sale el valor por el que vaya la cuenta del contador en cada flanco de reloj. Las señales rst y rst_sync, si están a 1, reinician el contador haciendo que la cuenta comience de nuevo en 0.
- GENÉRICOS
  - n = 32: número de bits del contador.

- INPUT
  - clk: señal de reloj.
  - rst: señal de reset asíncrona.
  - rst_sync: señal de reset síncrona.
  - count_up: señal que indica si se aumenta el valor del contador o se deja tal y como está.
- OUTPUT
  - dout: señal de n-1 bits indicando la cuenta por la que va el contador.
- UTILIZADO EN
  - Contador de Targets.
  - Contador de Índice de Lectura.
  - Contador de Índice de Escritura.
  - Contador en el módulo Pixel Más Distinto.

Producto Escalar (mult_sum_PF32Bits.vhd)

- DESCRIPCIÓN
  - Realiza el producto escalar de las dos entradas A y B, que serán dos vectores de Nx32 bits. Se utilizan N multiplicadores, uno por cada multiplicación A[i]xB[i], y, posteriormente, un árbol de sumadores conectado a los resultados de los multiplicadores, obteniendo al final un numero de 32 bits que indica el producto escalar de los vectores A y B. Cuando el resultado está disponible, se muestra por la salida r en el siguiente flanco de reloj. Se puede observar gráficamente un ejemplo de este módulo para N = 4 en la Figura 4.2.
- GENÉRICOS

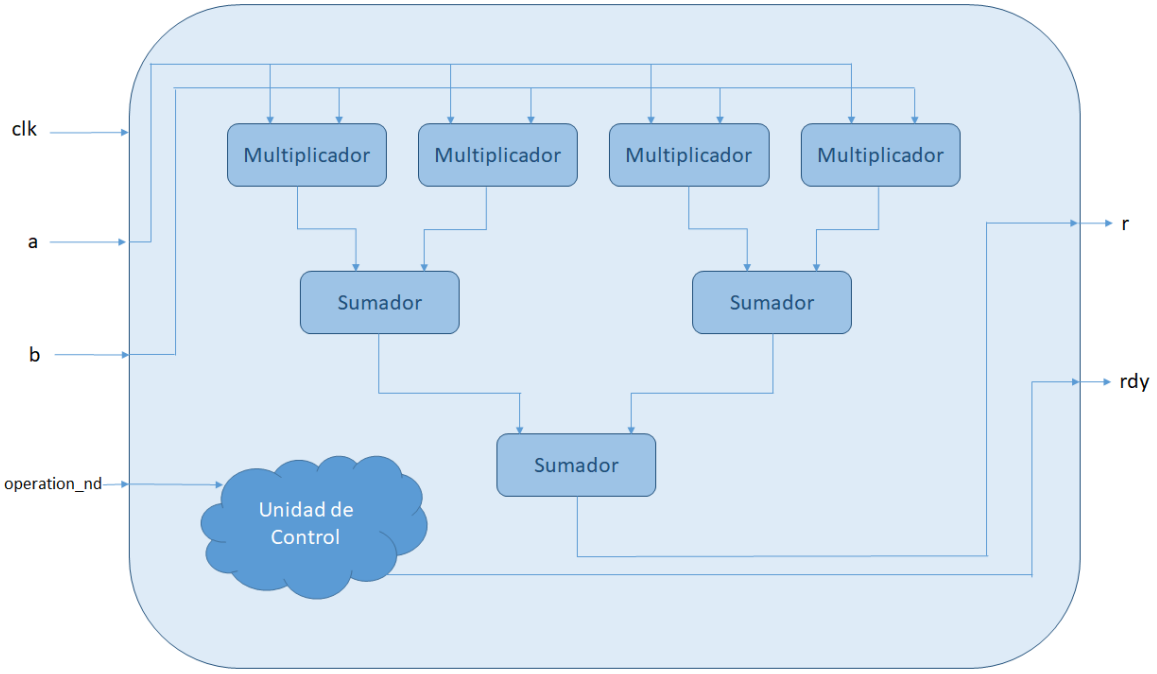


Figura 4.2: *Representación gráfica de un ejemplo del módulo Producto Escalar implementado en VHDL.*

- N = 4: número de elementos que tiene cada vector. Para el algoritmo no se utilizará este valor, sino el número de bandas.
- N_log2 = 2: logaritmo en base 2 de N. Para el algoritmo no se utilizará este valor sino el logaritmo en base 2 del número de bandas.

- INPUT
  - clk: señal de reloj.
  - a: señal de Nx32 bits indicando el vector A.
  - b: señal de Nx32 bits indicando el vector B.
  - operation_nd: señal que indica si los operandos están disponibles.
- OUTPUT
  - r: señal de 32 bits indicando el resultado del producto escalar de los vectores A

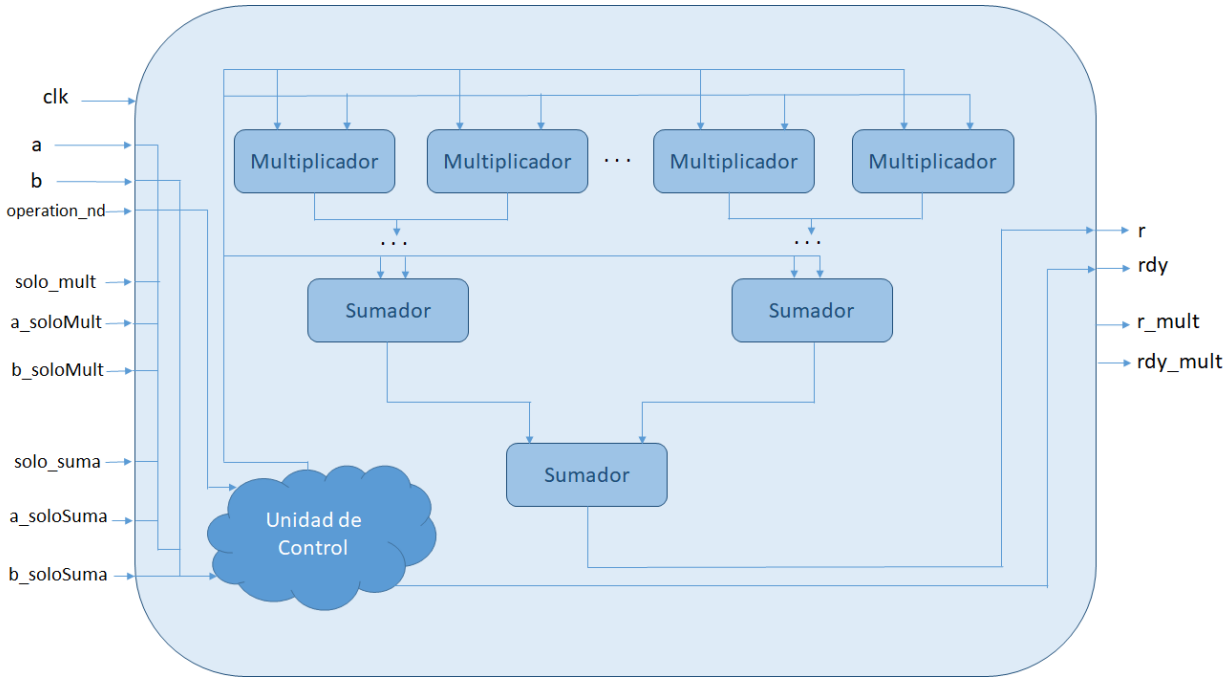


Figura 4.3: *Representación gráfica de un ejemplo del módulo Producto Escalar Auxiliar implementado en VHDL.*

y B.

- rdy: señal que indica que el resultado ya se encuentra disponible.

- UTILIZADO EN

  - ATDCA-GS.

Producto Escalar Auxiliar (mult_sum_aux_PF32Bits.vhd)

- DESCRIPCIÓN

  - En caso de que solo_suma y solo_mult estén a 0, realiza exactamente la misma funcionalidad que el módulo Producto Escalar. En caso de que solo_suma esté a 1, realiza la suma de todos los elementos de los dos vectores de entrada, A y B, empleando para ello un árbol de sumadores. Cuando el resultado está disponible, se muestra por la salida r en el siguiente flanco de reloj y la señal rdy se pone a 1. En caso de que solo_mult esté a 1, realiza la multiplicación elemento a elemento

de los dos vectores de entrada. Cuando se obtiene el resultado, se muestra por la señal r_mult (de Nx32 bits) y la señal rdy_mult se pone a 1. Se puede observar gráficamente un ejemplo de este módulo en la Figura 4.3.

- GENÉRICOS
  - N = 256: número de elementos que tiene cada vector (equivaldrá al número de bandas de la imagen).
  - N_log2 = 8: logaritmo en base 2 de N.
- INPUT
  - clk: señal de reloj.
  - a: señal de Nx32 bits indicando el vector A.
  - b: señal de Nx32 bits indicando el vector B.
  - operation_nd: señal que indica si los operandos están disponibles.
  - solo_mult: indica si se debe realizar la multiplicación elemento a elemento o no.
  - solo_suma: indica si se debe realizar la suma en árbol o no.
  - a_soloSuma: señal de Nx32 bits indicando el vector A para la operación de suma en árbol.
  - b_soloSuma: señal de Nx32 bits indicando el vector B para la operación de suma en árbol.
  - a_soloMult: señal de Nx32 bits indicando el vector A para la operación de multiplicación elemento a elemento.
  - b_soloMult: señal de Nx32 bits indicando el vector B para la operación de multiplicación elemento a elemento.
- OUTPUT

- r: señal de 32 bits indicando el resultado de la operación suma en árbol o la de producto escalar (la que se haya realizado) de los vectores A y B.
- rdy: señal que indica que el resultado de la operación suma en árbol o la de producto escalar (la que se haya realizado) ya se encuentra disponible.
- r: señal de 32 bits indicando el resultado de la operación de multiplicación elemento a elemento de los vectores A y B.
- rdy_mult: señal que indica que el resultado de la operación de multiplicación elemento a elemento ya se encuentra disponible.

- UTILIZADO EN
  - ATDCA-GS.

Registro (registro_rst.vhd)

- DESCRIPCIÓN
  - En cada flanco de reloj, si la señal load está a 1, se introduce el dato de la señal din en el registro. Si está a 0, no se produce esta actualización. Por la señal dout sale lo que haya dentro del registro en cada flanco de reloj con una política de Read First, lo que significa que, si se ha actualizado el contenido del registro (load = 1), será el nuevo dato el que saldrá por dout en ese mismo instante de tiempo. Las señales rst y rst_sync, si están a 1, reinician el registro haciendo que el contenido del mismo sean 0's.
- GENÉRICOS
  - n = 32: número de bits de los datos que se almacenan en el registro.
- INPUT
  - clk: señal de reloj.

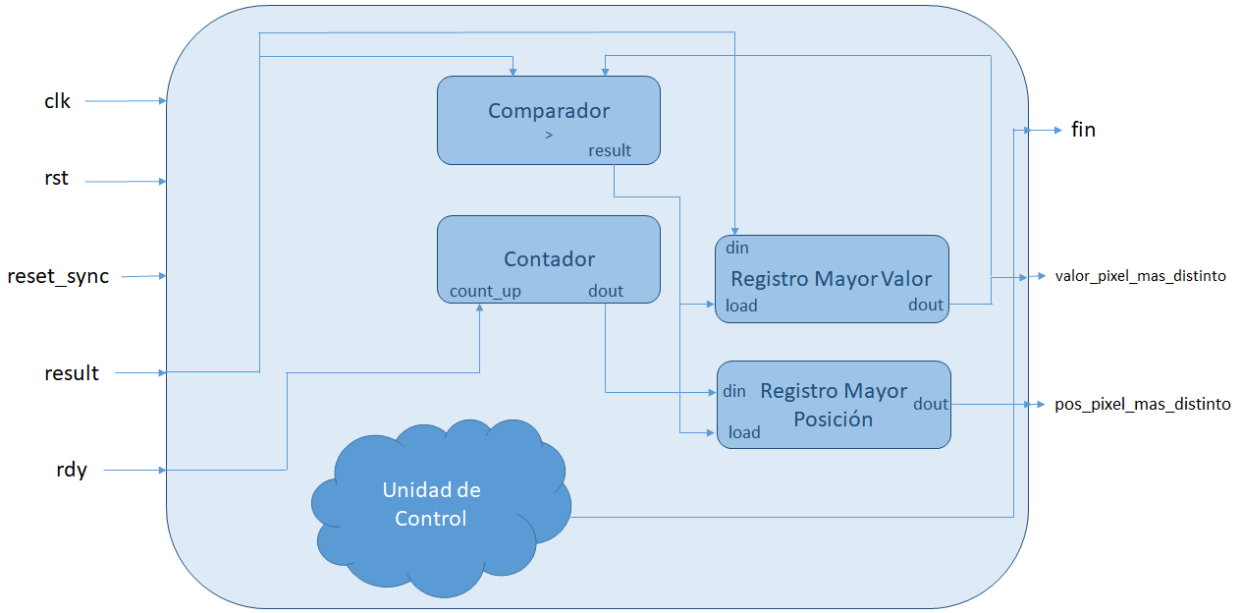


Figura 4.4: *Representación gráfica del módulo Pixel Más Distinto implementado en VHDL.*

- rst: señal de reset asíncrona.
- rst_sync: señal de reset síncrona.
- load: señal que indica si se actualiza el valor del registro o se deja tal y como está.
- din: señal de n bits indicando el dato a introducir en el registro.

▪ OUTPUT

- dout: señal de n bits indicando el dato que se encuentra almacenado en el registro.

▪ UTILIZADO EN

- Registro Mayor Valor en el módulo Pixel Más Distinto.
- Registro Mayor Posición en el módulo Pixel Mas Distinto.
- Registro Auxiliar.

Pixel Más Distinto (pixel_mas_distinto.vhd)

▪ DESCRIPCIÓN

- Recibe un número de píxeles y calcula cuál de ellos es el más brillante (o distinto), es decir, el que tiene mayor valor. Para ello hace uso de dos registros, un contador y un comparador que indica si el operando A es mayor que el operando B. Se puede observar gráficamente la estructura de este módulo en la Figura 4.4.

- GENÉRICOS
  - NUMPIXELES = 2: número de pixeles que se van a analizar. El algoritmo sobreescribirá este valor según el número de pixeles sobre el que se ejecute.
  - NUM_PIXELES_log2 = 1: logaritmo en base dos del número de píxeles. El algoritmo sobreescribirá este valor según el número de pixeles sobre el que se ejecute.

- INPUT
  - clk: señal de reloj.
  - rst: señal de reset asíncrona.
  - reset_sync: señal de reset síncrona.
  - result: señal de 32 bits indicando el siguiente pixel a analizar.
  - rdy: señal que indica que el siguiente pixel a analizar ya se encuentra disponible.

- OUTPUT
  - fin: señal que indica que ya se han analizado todos los píxeles y ya se ha encontrado el pixel más distinto.
  - pos_pixel_mas_distinto: señal de NUM_PIXELES_log2 bits que indica la posición en la memoria del pixel más distinto que se ha encontrado.
  - valor_pixel_mas_distinto: señal de 32 bits que indica el valor del pixel más distinto que se ha encontrado.

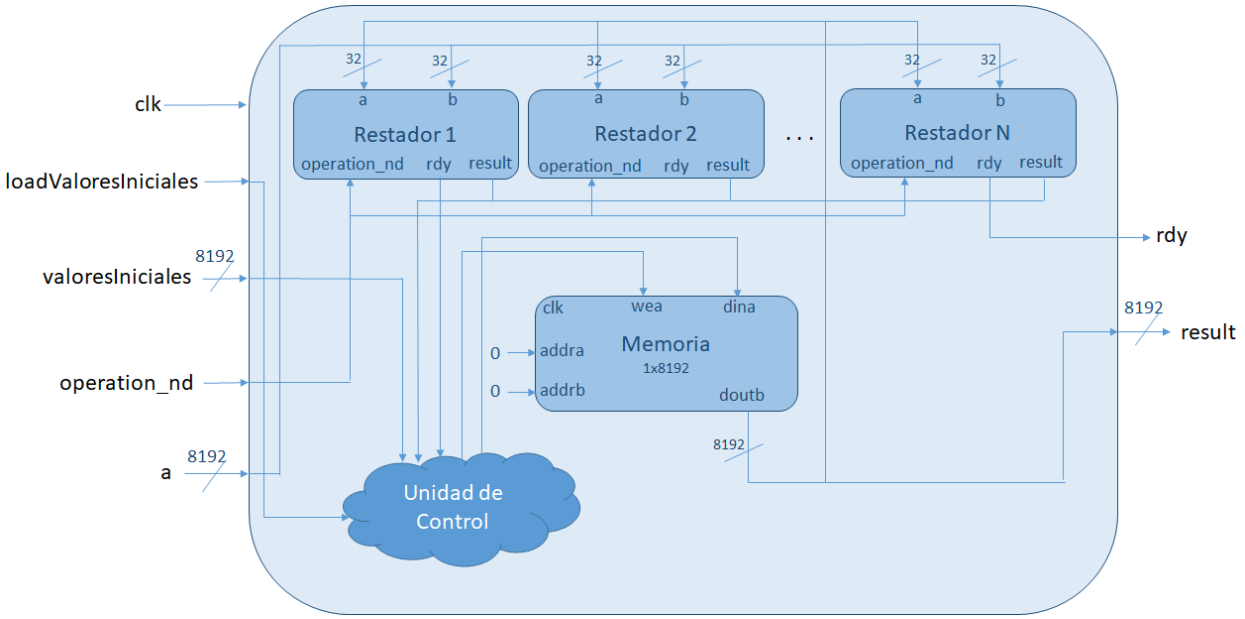


Figura 4.5: *Representación gráfica del módulo Array Resta Con Acumulación implementado en VHDL.*

- UTILIZADO EN
  - ATDCA-GS.

Array Resta con Acumulación (array_restadores_con_acumulacion_PF32Bits.vhd)

- DESCRIPCIÓN
  - Recibe un operando de Nx32 bits y lo resta a un valor que puede ser, o bien un valor inicial cargado previamente a los cálculos, o bien el resultado acumulado de las operaciones anteriores. La resta se realiza como si fuera una operación elemento a elemento de dos vectores de N números en punto flotante, por lo que se utilizan N restadores. Para almacenar el valor acumulado se hace uso de una memoria de 1x8192 bits. Se puede observar gráficamente la estructura de este módulo en la Figura 4.5.
- GENÉRICOS

- N = 256: número de restadores que se van a utilizar (el equivalente al número de bandas de la imagen hiperespectral).

- INPUT
  - clk: señal de reloj.
  - loadValoresInciales: señal que indica si se cargan los valores iniciales o no.
  - valoresInciales: señal de Nx32 bits que representa los valores iniciales que deben cargarse cuando sea necesario.
  - operation_nd: señal que indica si los operandos están disponibles para las restas.
  - a: señal de Nx32 bits que representa los N elementos que se desean restar a los valores acumulados hasta el momento.

- OUTPUT
  - result: señal de Nx32 bits indicando el resultado de las operaciones.
  - rdy: señal que indica que el resultado de las operaciones ya se encuentra disponible.

- UTILIZADO EN
  - ATDCA-GS.

4.3. Paralelización y optimización del algoritmo ATDCA-GS usando OpenCL

Se ha realizado el estudio de una implementación paralela del algoritmo usando el paradigma OpenCL para su posterior optimización[1]. Antes de continuar con el estudio, se comentará una serie de conceptos relacionados con el paradigma de progrmación OpenCL.

[1]https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf

La especificación de OpenCL[2] fue desarrollada para permitir una libertad en términos de implementar aplicaciones que podrían ejecutarse en diferentes arquitecturas. En el paradigma OpenCL, el programa "host", está a cargo de las operaciones de E/S, inicialización de datos y controlando el "device". Además, lanza los códigos "kernel" y la sincronización entre ellos. Entre los principales beneficios se encuentra el amplio rango de dispositivos hardware a ser usados: CPUs, GPUs y FPGAs. Todos ellos podrán ser usados con un esfuerzo moderado en el proceso de codificación.

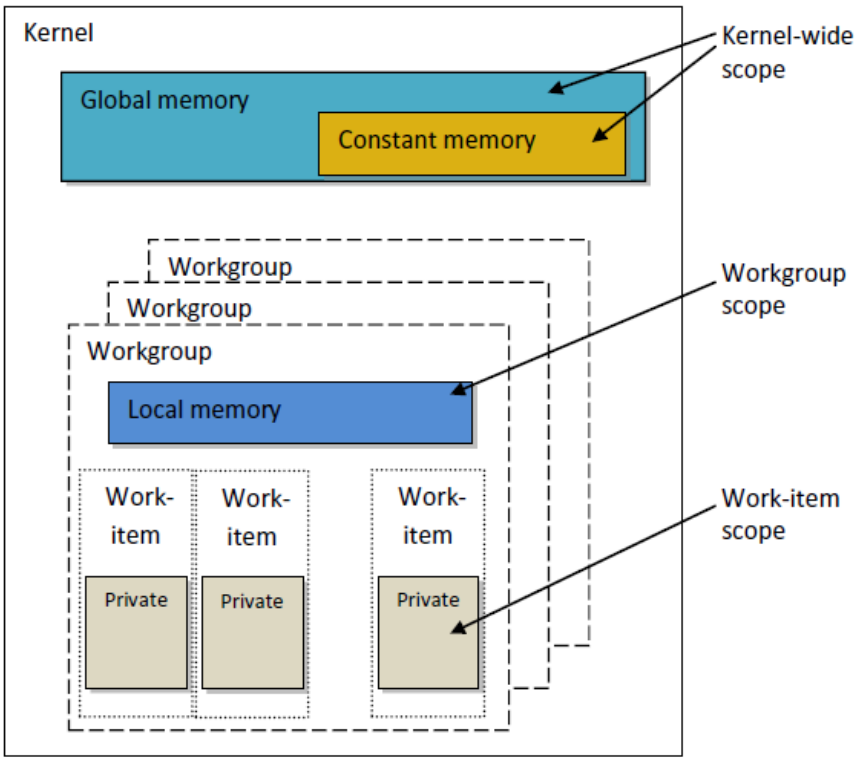


Figura 4.6: *Jerarquía de memoria en OpenCL, y división en work-items.*

Un kernel OpenCL permite expresar paralelismo a través de la ejecución de varios *work-items*. Un grupo de *work-items* forman un textitwork-group que corre sobre una unidad de cómputo individual o "single compute unit". Los *work-items* ejecutan el mismo *kernel* (con un único id), comparte una memoria rápida denominada *local memory* (ver Figura 4.6[3]) y

[2]https://www.khronos.org/opencl
[3]https://www.mql5.com/es/articles/407

puede ser sincronizada con barreras. La dimensión máxima de cada work-group dependerá de la especificación del *device* usado (ver Sección 5).

Una vez que se han conocido los conceptos relacionados con el paradigma de programación paralela OpenCL, pasaremos a analizar una implementación previa en OpenCL del algoritmo ATDCA-GS. En este trabajo previo [**?**] fueron implementados tres kernels: cálculo del píxel más brillante, cálculo de la proyección ortogonal de cada píxel sobre la imagen y obtención de la máxima proyección ortogonal. De cada uno de ellos se mostrará a continuación el pseudocódigo que se ha seguido.

El primer *kernel* sobre el que se ha realizado el estudio, es el correspondiente al cálculo del píxel más brillante. El pseudocódigo estudiado es el descrito en el algoritmo 2. Tras analizarlo y teniendo en cuenta que esta parte consume aproximadamente el **5 %** del tiempo de procesamiento total del algoritmo considerando una imagen hiperespectral real como es Cuprite, se han tenido en cuenta varias consideraciones:

Algorithm 2 Cálculo del píxel más brillante

```
1: global d_image ← Vector inicial X[r], d_bright ← El valor brillo de cada píxel
2: registers bright ← 0, value ← 0
3: id ← get_group_id(0) * get_local_size(0) + get_local_id(0)
   % n_b indica el número de bandas espectrales
4: if id < r then
5:    for k = 0 to n_b do
6:       value ← d_image[id + (k * r)]
7:       bright ← bright + value * value
8:    end for
9:    d_bright[id] ← bright
10: end if
```

1. Empleo de *restrict* en arrays usados en memoria global. De esta manera se evita que los punteros no estén apuntando a ubicaciones en memoria con información. A la hora de definir los parámetros de la función kernel se ha acompañado la variable *restrict* a los arrays declarado en memoria global (*restrict d_ image* y *restrict d_ bright*).

2. Distribución óptima de los datos para favorecer la vectorización. Para ello, la línea 6 del Algoritmo 2 se ha modificado por lo siguiente: d_image[k + (id * n_b)].

3. Desenrollado de bucles (*loop unrolling*). Entre las líneas 4 y 5 del Algoritmo 2 se ha introducido la siguiente directiva: *#pragma unroll num_factor*, permitiendo un factor diferente para el desenrollado del bucle dependiendo del tamaño de la imagen.

Algorithm 3 Reducción para obtener la máxima proyección ortogonal

```
1: global d_bright, d_projection, d_index
2: local s_p[BLK] ← Estructura inicial para almacenar todas las proyecciones
3: local s_i[BLK] ← Estructura inicial para almacenar todos los índices para cada proyección
4: tid ← get_local_id(0)
5: id ← get_group_id(0) * (get_local_size(0) * 2) + get_local_id(0)
6: if (id + get_local_size(0)) ≥ r then
7:    s_p[tid] ← d_bright[id]
8:    s_i[tid] ← id
9: else
10:   if d_bright[id] > d_bright[id + get_local_size(0)] then
11:      s_p[tid] ← d_bright[id]
12:      s_i[tid] ← id
13:   else
14:      s_p[tid] ← d_bright[id + get_local_size(0)]
15:      s_i[tid] ← (id + get_local_size(0))
16:   end if
17: end if
18: barrier(CLK_LOCAL_MEM_FENCE)
    % En esta sincronización, todos los hilos deben esperar la ejecución de todos los hilos en un
    bloque para poder completar la copia en memoria local de s_p y s_i
19: for s=get_local_size(0) / 2  to  s>0; s>>=1 do
20:   if tid < s then
21:      if s_p[tid] ≤ s_p[tid + s] then
22:         s_p[tid] ← s_p[tid + s]
23:         s_i[tid] ← s_i[tid + s]
24:      end if
25:   end if
26:   barrier(CLK_LOCAL_MEM_FENCE)
27: end for
28: d_projection[get_group_id(0)] ← s_p[0]
29: d_index[get_group_id(0)] ← s_i[0]
30: barrier(CLK_LOCAL_MEM_FENCE)
```

El siguiente *kernel* que ha sido analizado es la reducción para obtener la máxima proyección ortogonal. El pseudocódigo estudiado es el descrito en el Algoritmo 3. Tras analizarlo, se ha llegado a la conclusión de que no se podrían realizar grandes optimizaciones en este

kernel ya que el tiempo de ejecución exclusivamente de esta parte está por debajo del **2 %** del tiempo total del algoritmo usando una imagen real como es Cuprite.

Por último, nos encontramos con el kernel que se encarga de computar la proyección ortogonal de cada uno de los píxeles que conforman la imagen. El pseudocódigo estudiado es el descrito en el Algoritmo 4. Esta parte del código constituye aproximadamente un **90 %** en cuanto al tiempo de procesamiento total del algoritmo. La parte más costosa la podemos encontrar entre las líneas 18-21. Al igual que en el primer kernel, se ha optado por las mismas consideraciones: empleo de la variable *restrict*, distribución óptima de los datos para favorecer la vectorización y por supuesto, un desenrollado de este bucle que depende del número de bandas espectrales usado. Para ello, se han realizado las siguientes modificaciones al Algoritmo 4:

Algorithm 4 Cálculo de las proyecciones ortogonales para cada píxel

```
1: global d_image ← Vector inicial X[r]
2: global d_projection ← El valor proyección para cada píxel x_i en X
3: global d_f ← The vector más ortogonal
4: registers sum ← 0, value ← 0
5: local s_df[n_b] ← Estructura inicial d_f con el vector más ortogonal
6: id ← get_global_id(0)
7: if id < r then
8:    if get_local_size(0) < n_b then
9:       for i = get_local_id(0) to n_b do
10:         s_df[i] ← d_f[i]
11:      end for
12:   else
13:      if get_local_id(0) < n_b then
14:         s_df[get_local_id(0)] ← d_f[get_local_id(0)]
15:      end if
16:   end if
17:   barrier(CLK_LOCAL_MEM_FENCE) % Espera hasta que la copia de d_f se haya completado en memoria local
18:   for i = 0 to n_b do
19:      value ← d_image[id + (i * r)]
20:      sum ← sum + value * s_df[i]
21:   end for
22:   d_projection[id] ← sum * sum
23: end if
```

1. Se ha declarado la variable *restrict* a los arrays usados en memoria global (*restrict d_ image*, restrict d_projection y restrict d_f).

2. Se ha empleado otra distribución de los datos para favorecer la vectorización. La línea 19 del Algoritmo 4 ha sido modificada con lo siguiente: d_image[i + (id * n_b)].

3. Uso óptimo del desenrollado de bucles. Entre las líneas 17 y 18 del Algoritmo 4 se ha introducido la siguiente directiva: #*pragma unroll num_factor*, permitiendo un factor diferente para el desenrollado del bucle dependiendo del tamaño de la imagen.

Las optimizaciones aplicadas a los kernels descritos anteriormente han favorecido que el tiempo de procesamiento sea menor aumentando el rendimiento de nuestra implementación. El análisis será cubierto en el capítulo siguiente.

Capítulo 5

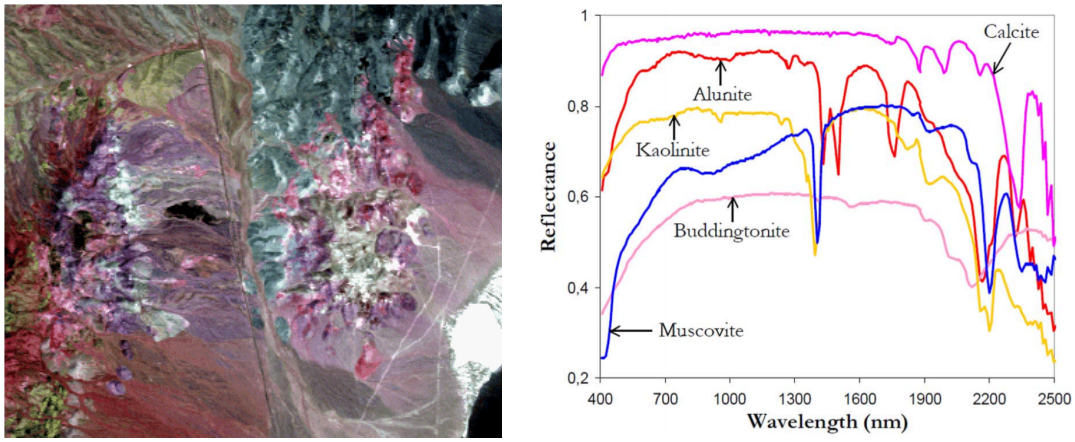
Resultados

5.1. Conjunto de datos hiperespectrales

Para realizar los experimentos, se han utilizado diferentes conjuntos de datos hiperespectrales, tanto reales como sintéticos:

- El primer conjunto de datos corresponde a la conocida escena AVIRIS Cuprite (ver Figura 5.1(a)), recogida en el verano de 1997 y disponible online en unidades de reflectancia después de ser corregida atmosféricamente. La porción utilizada en los experimentos corresponde a un subconjunto de 350 x 350 píxeles del sector, etiquetados como f970619t01p02_r02_sc03.a.rfl en los datos online, que cuenta con 224 bandas espectrales en el rango de 400 a 2500 nanómetros y un tamaño total de alrededor de 50 megabytes. Las bandas 1-3, 105-115 y 150-170 han sido eliminadas antes del análisis debido a la absorción por agua y la baja relación señal-ruido o signal-to-noise ratio (SNR) de estas bandas. La zona es bien conocida mineralógicamente, y tiene varios minerales expuestos de interés, incluyendo alunita, buddingtonita, calcita, caolinita y moscovita. Las firmas de referencia de suelo de los minerales mencionados (ver Figura 5.1(b)), disponibles en la biblioteca U.S. Geological Survey library (USGS), se utilizarán para evaluar la pureza de la firma de los endmembers en este trabajo.

- El segundo conjunto de datos corresponde a una imagen sintética (ver Figura 5.2) que nos ayudará a evaluar la escalabilidad de nuestras implementaciones. La imagen ha



(a) (b)

Figura 5.1: *(a) Composición en falso color de la escena captada por el sensor hiperespectral AVIRIS sobre el distrito minero de Cuprite en Nevada. (b) Firmas espectrales de la biblioteca U.S. Geological Survey de los minerales expuestos de interés.*

sido construida usando un conjunto de 30 firmas espectrales de la librería USGS y el procedimiento descrito en [**?**] para simular patrones naturales espaciales. La imagen sintética resultante está compuesta por un total de 750×650 píxeles y 224 bandas espectrales, dando como resultado un tamaño aproximado de 437 MB. La Figura 5.2 muestra una composición en falso color de la escena simulada y tres ejemplos de mapas de abundancia verdad terreno (construidos a partir de píxeles puros o endmembers).

5.2. Plataformas FPGAs

El algoritmo completo en VHDL se ha implementado en una placa VC709 (ver Figura 5.3), una placa reconfigurable con una sola Virtex-7 XC7VX690T, dos ranuras DDRM DDR3 que admiten hasta 4 GB cada una, un puerto RS232 y algunos componentes adicionales que no se han utilizado en nuestra implementación. La FPGA de Xilinx Virtex-7 XC7VX690T tiene una capacidad total de 693,120 celdas lógicas, 3,600 DSPs y 52,920 Kb de memoria.

Para llevar a cabo la validación del algoritmo en placa, se ha utilizado la plataforma de test que se muestra en la Figura 5.4. Como se puede observar, consta de un microprocesdor tipo MicroBlaze, un controlador de memoria, un puero serie y una unidad reconfigurable,

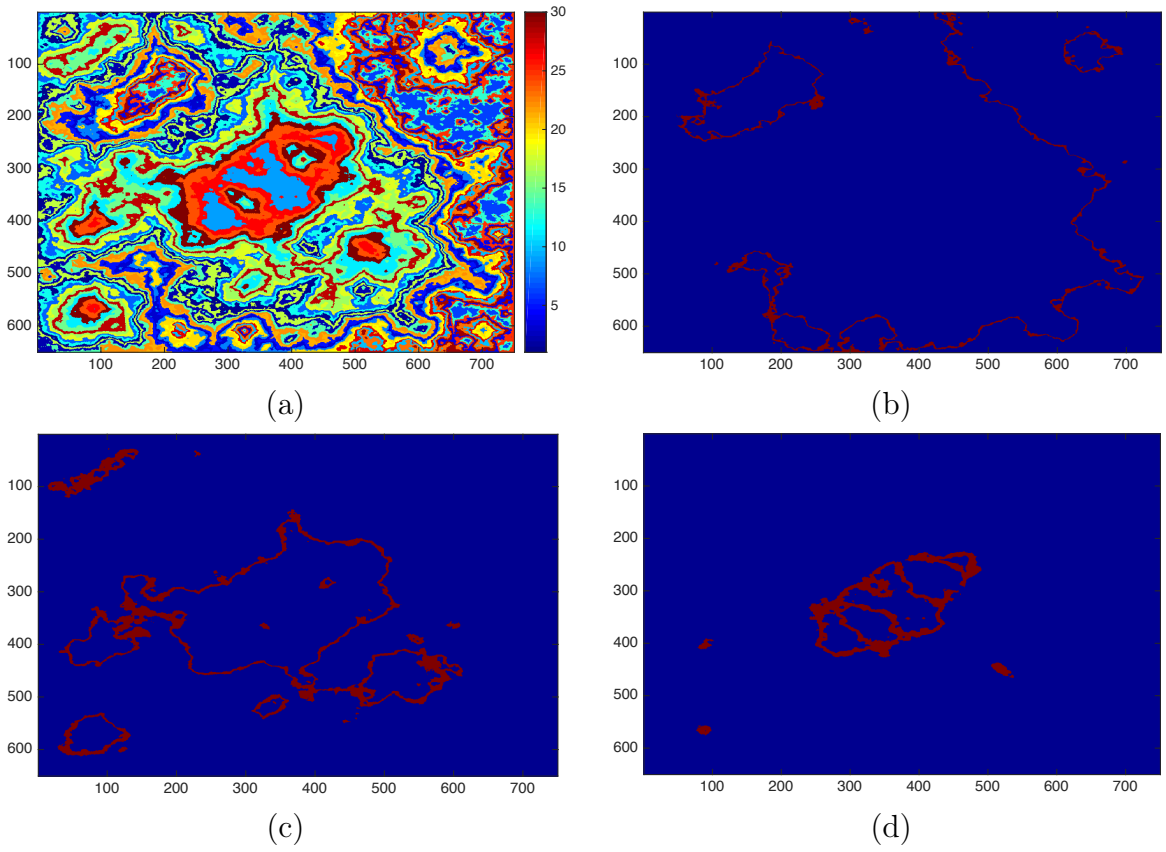


Figura 5.2: *(a) Composición en falso color de la escena sintética. (b) Endmember #3. (c) Endmember #15. (d) Endmember #26.*

todo ello interconectado mediante el bus AXI. Dentro de la unidad reconfigurable, se ubicaría el algoritmo a testear.

Por otro lado, para la implementación en OpenCL se ha hecho uso de la FPGA Intel Arria 10 GX 10AX115S2F45I2SGES (ver Tabla 5.1 para conocer los recursos disponibles y las particulariedades sobre el modelo de memoria en OpenCL). Este dispositivo se encuentra instalado en un servidor con un procesador Intel Xeon E5-1620 v3, con 4 cores físicos cada uno a una frecuencia de reloj de 3.5GHz y 64GB de memoria RAM DDR3. El sistema operativo empleado en este servidor es un CentOS Linux 7 y cuenta con la versión 17.1.0 del Intel FPGA SDK for OpenCL Offline Compiler.

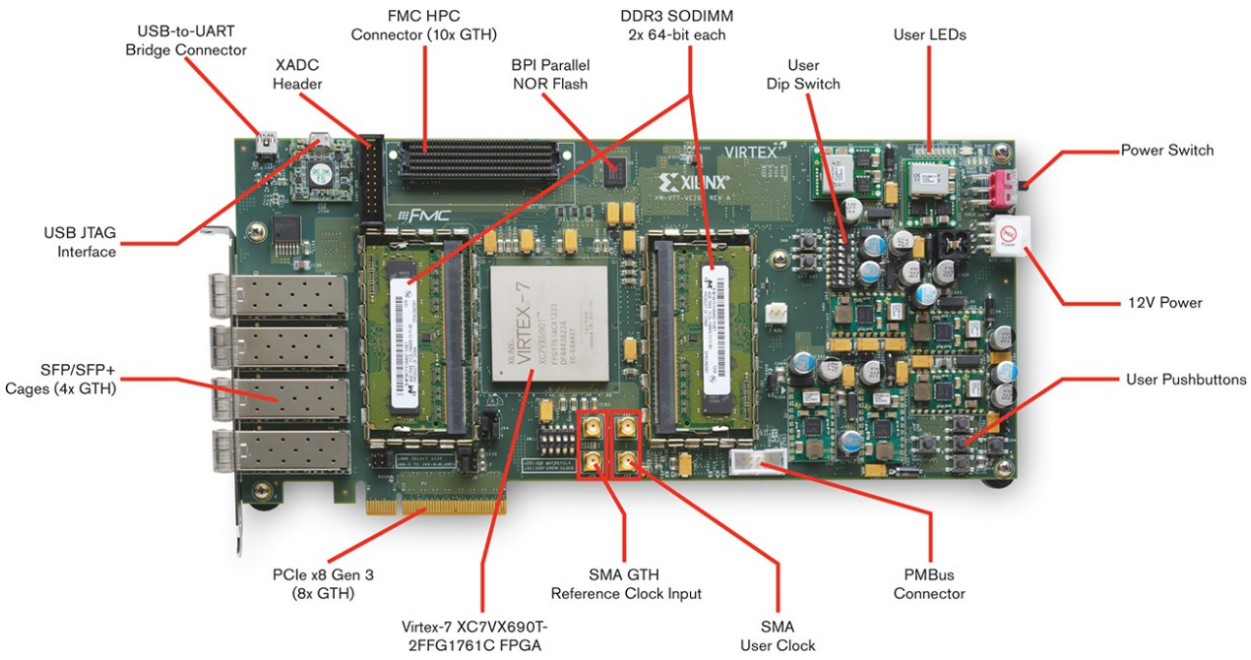


Figura 5.3: *Placa VC709.*

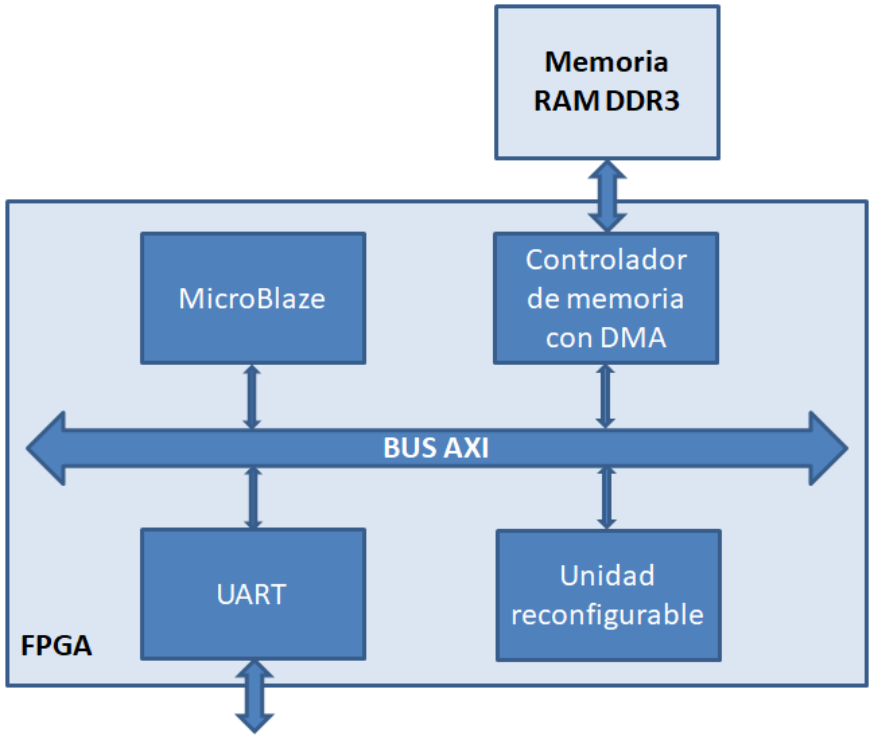


Figura 5.4: *Plataforma de test para la placa VC709.*

	OpenCL	FPGA	Intel Arria 10 GX
Memoria	Global	External	2GB DDR3
	Constant	Cache	32KB DDR3
	Local	Embedded	67Mbits
	Private	Registers	67244Kbits

Tabla 5.1: *Modelo de memoria en OpenCL y recursos disponibles para la FPGA Intel Arria 10 GX.*

5.3. Métricas

5.3.1. Calidad

Antes de describir los resultados obtenidos, se describe primero la métrica utilizada para la comparación cuantitativa en los experimentos realizados. Con el fin de reducir el impacto de las fuentes de interferencia atmosférica en la evaluación realizada, se utiliza el ángulo espectral (AE) entre el endmember más similar detectado por la implementación propuesta y la firma espectral de referencia disponible en cada escena. El AE entre un píxel $X(i,j)$ seleccionado por el algoritmo ATGP-GS y una firma espectral de referencia Si disponible a priori, puede calcularse simplemente como:

$$AE[X(i,j), S_i] = cos^{-1}\frac{X(i,j) \cdot S_i}{|X(i,j)| \cdot |S_i|} \tag{5.1}$$

es decir, el AE mide el ángulo formado por dos vectores n-dimensionales. Como resultado, valores bajos de AE significan una alta similitud espectral entre los vectores comparados. Esta medida de similitud espectral es independiente de la multiplicación de $X(i,j)$ y Si por constantes y, en consecuencia, es independiente ante escalas multiplicativas desconocidas que puedan surgir debido a las diferencias en la iluminación y al ángulo de incidencia. Esto nos puede ayudar a compensar las diferentes condiciones de adquisición para un píxel en la imagen original y para una firma espectral recogida sobre el terreno (como es el caso de las firmas de referencia de la librería USGS utilizadas para la imagen AVIRIS Cuprite).

5.3.2. Rendimiento

Para medir el rendimiento se ha tenido en cuenta la aceleración del algoritmo en un lenguaje respecto al otro (ecuación 5.2) y de esta manera realizar una comparativa entre ambos. Además, se ha identificado cuál es el tiempo límite (máximo) para que el procesamiento del desmezclado espectral de la imagen completa se pueda realizar en tiempo real [**?**], que es uno de los principales objetivos que se persigue en este trabajo.

$$\text{Rendimiento} = \frac{\text{Tiempo OpenCL}}{\text{Tiempo VHDL}} \tag{5.2}$$

Pese a que las pruebas se realizarán sobre imágenes de un tamaño relativamente pequeño, es preciso destacar que si éstas se pueden procesar en tiempo real, las de mayor tamaño también se podrán procesar de esta forma. Esto se debe a que el aumento en el tiempo respecto al tamaño de la imagen es logarítmico, es decir, aumenta poco ante variaciones en el tamaño de las imágenes y/o en el número de endmembers a calcular.

Para medir el rendimiento, primero se analizará el rendimiento en cuanto a recursos empleados en cada implementación con la siguiente metodología: primero, se prestará atención a los recursos empleados (LUTs, LUTRAMs, BRAMs y DSPs) en la FPGA Virtex 7 XC7VX690T con el algorirmo en VHDL para imágenes de 188, 224 y 256 bandas, y se comparará con respecto a los máximos disponibles que ofrece el dispositivo; después, se realizará el mismo análisis pero respecto a la FPGA Intel Arria 10 GX y sus recursos (ALUTs, FFs, RAMs y DSPs), usando primero el algoritmo en OpenCL sin optimizaciones a modo de referencia y después con ellas.

Una vez determinado el rendimiento en cuanto a recursos utilizados, se procederá a evaluar el rendimiento en cuanto a tiempo. En primer lugar, se inferirá una fórmula que calcule el número de ciclos necesarios para analizar cualquier imagen a través del algoritmo en VHDL. Después se analizarán los tiempos de ejecución de las dos implementaciones con la imagen de Cuprite prestando atención para ello a la frecuencia de reloj. A continuación se establecerá una comparativa entre el tiempo teórico máximo alcanzable para un análisis

a bordo y el tiempo real que tarda en ejecutarse el algoritmo en los lenguajes VHDL y OpenCL con la penalización de E/S. Finalmente con esta información se afirmará si se consigue viabilidad para un análisis en tiempo real o no.

5.4. Resultados experimentales

5.4.1. Calidad

Mineral en USGS	Implementación OpenCL en FPGA	Implementación VHDL en FPGA
Alunita	5.48°	5.48°
Buddingtonita	4.08°	4.08°
Calcita	5.87°	5.87°
Kaolinita	11.14°	11.14°
Moscovita	5.68°	5.68°

Tabla 5.2: *Ángulo espectral entre los endmembers extraídos en la escena AVIRIS Cuprite por las diferentes implementaciones de ATGP-GS y las firmas de referencia seleccionadas de la librería USGS.*

En la Tabla 5.2 se muestra la calidad de los resultados experimentales tras la ejecución del algoritmo en OpenCL y en VHDL. Cabe señalar que sólo refleja la menor puntuación de AE de todos los endmembers extraídos con respecto a su firma de referencia en cada caso. Dado que se obtienen los mismos resultados, validamos que las dos implementaciones son correctas.

5.4.2. Rendimiento

En la Tabla 5.3 se pueden observar los resultados de ocupación en la FPGA Virtex-7 XC7VX690T, a partir de los cuales se puede afirmar que los recursos empleados en la implementación del algoritmo desarrollado en VHDL crecen linealmente con el número de bandas. Se observa que el recurso más utilizado son las LUTs y en el caso de un número de bandas elevado (256) apenas se alcanza un 86 % de su utilización.

En el caso de la implementación en OpenCL, los recursos necesarios crecen en la versión

Recurso	LUT	LUTRAM	BRAM	DSP
Disponible	433.200	174.200	1.470	3.600
ATGP-GS (188 bandas)	271.912 (62,8 %)	25.260 (14,5 %)	475 (32,3 %)	1.508 (41,9 %)
ATGP-GS (224 bandas)	311.496 (71,9 %)	29.083 (16,7 %)	555 (37,8 %)	1.792 (49,8 %)
ATGP-GS (256 bandas)	370.264 (85,5 %)	34.397 (19,8 %)	1.013 (68,9 %)	2.118 (58,8 %)

Tabla 5.3: *Resumen de la utilización de recursos en la Virtex–7 XC7VX690T.*

Recurso	ALUT	FF	RAM	DSP
Disponible	707.600	1.415.440	2.531	1.518
ATGP-GS (sin optimizar)	182.679 (25,8 %)	354.680 (25,1 %)	605 (23,9 %)	24 (1,6 %)
ATGP-GS (188 bandas)	210.515 (29,8 %)	425.652 (30,1 %)	941 (37,2 %)	112 (7,4 %)
ATGP-GS (224 bandas)	217.325 (30,7 %)	446.609 (31,6 %)	1.043 (41,2 %)	144 (9,5 %)
ATGP-GS (256 bandas)	226.709 (32,0 %)	463.269 (32,7 %)	1.144 (45,2 %)	160 (10,5 %)

Tabla 5.4: *Resumen de la utilización de recursos en la Intel Arria 10 GX.*

optimizada a medida que se varía el factor de la directiva *#pragma unroll num_factor* para el desenrollado de bucles (ver Tabla 5.4), como consecuencia del tamaño de bandas espectrales. Para las imágenes Cuprite y sintética los factores óptimos han sido 12 y 16, respectivamente. Las demás optimizaciones no varían el porcentaje de recursos de la versión sin optimizar. En este caso se observa que el recurso más utilizado son las RAMs y apenas se alcanza un 46 % de su utilización.

El número de ciclos necesarios para la ejecución del algoritmo implementado en VHDL se puede calcular a partir de la siguiente fórmula:

$$\text{núm.ciclos} = \underbrace{\text{núm.píxeles} + 16}_{\text{cálculo del primer target}} + \underbrace{\overbrace{47}^{\text{proyección}} + \text{núm.píxeles} + 18}_{\text{cálculo del segundo target}} + \underbrace{2 \cdot \text{núm.targets}}_{\text{peticiones y escrituras de targets}} + \underbrace{\overbrace{103(\text{núm.targets} - 2)}^{\text{proyecciones}} + \sum_{i=1}^{\text{núm.targets}-2} (3i) + (\text{núm.píxeles} + 18) \cdot (\text{núm.targets} - 2)}_{\text{cálculo del tercer target en adelante}} \quad (5.3)$$

en la que cabe destacar que el número de ciclos no depende del número de bandas sino únicamente del número de targets a calcular y del número de píxeles que contenga la imagen.

Otro aspecto importante a tener en cuenta es que el ciclo de reloj se mantiene por encima de los 100 MHz y que se mantiene constante a pesar de que aumente el número de bandas espectrales. Teniendo esto en cuenta, para la imagen de Cuprite son necesarios 2.330.135 ciclos, por lo que el algoritmo se ejecutaría en 0,0233 segundos. Para la imagen sintética son necesarios 14.629.747 ciclos, por lo que se ejecutaría en 0,1463 segundos.

Imagen	Implementación OpenCL	Implementación VHDL (plataforma test)	Implementación VHDL (máximo rendimiento)
Cuprite	1,4874 seg	0,3793 seg	0,0233 seg
Sintética	8,2089 seg	1,8191 seg	0,1463 seg

Tabla 5.5: *Tiempo de procesamiento para la implementación del algoritmo ATDCA-GS desarrollado en VHDL y en OpenCL para FPGA.*

Observando la Tabla 5.5 se aprecia cómo los tiempos de ejecución en la plataforma de test para la implementación en VHDL son peores que los téoricos mínimos alcanzables para esta implementación. Esto se debe a que tal y como está desarrollada la plataforma de test, la E/S se convierte en el cuello de botella del sistema. Además, podemos observar cómo los tiempos de ejecución teóricos mínimos se incrementan sin tener en cuenta el número de bandas y sin embargo, para la implementación en la plataforma de test sí depende del número de banbas (se produce una mayor E/S).

Dado que el análisis en tiempo real de las imágenes se consigue cuando el tiempo de procesamiento es menor o igual que 1,986 segundos para la imagen de Cuprite o menor o igual que 7,903 segundos para la imagen sintética, se observa que, para la imagen Cuprite, sí se consigue un análisis eficiente con la implementacion tanto en VHDL como en OpenCL; para la imagen sintética, sí se consigue un análisis eficiente con la implementación en VHDL y, pese a que no ocurra lo mismo con la implementación en OpenCL, sí se alcanza un valor muy cercano al deseado. El tiempo teórico máximo para realizar un análisis a bordo de la imagen real y la sintética se obtiene tomando como referencia el sensor AVIRIS, que tarda 8,3 milisegundos en captar 512 píxeles con 224 bandas espectrales.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

En este Trabajo de Fin de Grado se ha llevado a cabo el diseño y la implementación del algoritmo ATDCA-GS, que utiliza la ortogonalización de Gram Schmidt con el objetivo de optimizar y mejorar el rendimiento de operaciones complejas como es el caso del cálculo de la inversa de una matriz. Se han empleado los lenguajes de programación VHDL y OpenCL y se han evaluado los resultados de su ejecución en FPGA para posteriormente realizar una comparación de rendimiento entre ambas alternativas.

Como parte del diseño, se ha realizado una adaptación del algoritmo adecuándolo al flujo habitual de un diseño específico hardware, minimizando lo máximo posible la cantidad de recursos a emplear y paralelizando las operaciones llevadas a cabo durante la ejecución del algoritmo.

Para realizar una comparativa en cuanto al rendimiento en tiempo de las dos implementaciones, se ha teniendo en cuenta la aceleración de una respecto a la otra haciendo uso de imágenes reales y sintéticas. Además, se ha verificado que, excepto la implementación en OpenCL para imágenes de gran tamaño, el procesamiento en ambas alternativas no supera el tiempo límite (máximo) y por tanto se puede realizar el análisis en tiempo real, cumpliendo uno de los objetivos principales de este trabajo.

Las pruebas de rendimiento en cuanto a recursos empleados en cada implementación

han desvelado que el porcentaje de recursos utilizados aumenta linealmente con el número de bandas. También revela que para un número elevado de las mismas (256) el recurso de mayor uso apenas alcanza un 86 % de utilización en VHDL y un 48 % en OpenCL, por lo que se concluye que el rendimiento es adecuado.

6.2. Trabajo futuro

En primer lugar, sería conveniente mejorar las optimizaciones de la implementación en OpenCL para que permita un análisis en tiempo real al igual que las demás alternativas. Además, dado que la tendencia del tamaño de las imágenes es continuar creciendo cada vez más, la opción de trabajo futuro que parece más evidente es la de conseguir procesar otras imágenes reales de un tamaño aún mayor.

Una posible vía de trabajo futuro para este trabajo sería desarrollar el algoritmo convirtiendo la aritmética de punto flotante a aritmética entera. De esta manera se conseguiría un mejor rendimiento ya que los cálculos serían aún más sencillos y, por lo tanto, el número de recursos necesarios disminuiría a la vez que aumentaría la frencuencia de reloj.

Otro posible camino de continuación podría ser la modificación de la plataforma de test para utilizar el bus PCIe 3x8. De esta manera se reducirían las penalizaciones debidas a la E/S.

Por último, se podría optar por analizar si los kernels de la implementación en OpenCL pueden seguir un modelo de programación paralela a nivel de tarea, de modo que la tarea se refiera a la ejecución de un kernel con un work-group que contenga un work-item y, así, el compilador intente acelerar el único work-item para conseguir un rendimiento mejor.

Apéndice A

Introduction

A.1. Motivation

A hyperspectral image is a high spectral resolution image obtained through sensors capable of obtaining hundreds or even thousands of images on the same terrestrial area but corresponding to different wavelength channels. The set of spectral bands is not strictly limited to the visible spectrum but also covers the infrared and the ultraviolet.

At present, the use of hyperspectral images is increasing considerably due to the launching of new satellites and the interest in remote observation of the Earth, which has utility in areas as diverse as defense, precision agriculture, geology (detection of mineral deposits) , valuation of environmental impacts or even artificial vision.

During the last years there have been many advances with regard to sensor technology, which has revolutionized the collection, handling and analysis of the data collected. This evolution has managed to go from having a few tens of bands to having hundreds and the tendency is for the number to continue increasing. Institutions such as National Aeronautics and Space Administration (NASA) or the European Space Agency (ESA) are continuously obtaining a large amount of data that needs to be processed. As a result, new challenges have arisen in the processing of data.

If we add to the increase in the amount of information collected, many current and future remote observation applications require real-time processing capabilities (in the same time

or less than the satellite takes to capture the data) or close to this real-time, it is essential to use parallel architectures for the efficient [**?**] and fast processing of this type of images.

The main problem in the processing of hyperspectral images lies in the spectral mixture, that is to say, the existence of mixed pixels in which several different materials coexist at the subpixel level. This type of pixels are the most common in hyperspectral images and for their analysis it is necessary to use complex algorithms with a high computational cost, which makes the execution of the demixing algorithms slow and requires acceleration or parallelization.

To address this type of tasks, parallel computing has been widely used through multi-core processors, GPUs (Graphics Processing Units) or dedicated hardware such as FPGAs (Field-Programmable Gate Arrays). Of all the alternatives, the latter present an efficient option in terms of performance, offering reduced times, in addition to a lesser use of resources, being the few alternatives that can be adapted in a sensor to perform on-board processing in space missions such as Mars Pathfinder or Mars Surveyor [**?**].

On the one hand, VHDL or Verilog are the native ways to program this type of devices, at a low level and more optimal. On the other hand, there is an alternative in OpenCL that allows a high level programming, faster and allowing its execution in a variety of architectures but less optimal at the level of hardware resources than in FPGAs devices.

A.2. Objectives

The general objective of this work is the parallel implementation on FPGA of the Automatic Target Detection and Classification Algorithm [**? ?**] making use of the Gram Schmidt Orthogonalization and the programming languages VHDL and OpenCL. This will allow a very interesting comparison between a native language for said platform (VHDL) and another paradigm of parallel programming at a high level (OpenCL) that can be ported to other platforms such as multi-core processors, GPUs or other accelerators.

The achievement of the general objective is carried out in the present memory by ad-

dressing a series of specific objectives, which are listed below:

- Design of individual modules in VHDL that serve to perform all the operations that are needed for the implementation of the ATDCA-GS algorithm.
- Elaboration of a state machine and implementation of the algorithm using the individual modules.
- Analysis and optimization of a previous parallel implementation in OpenCL of the algorithm.
- Obtaining results and performance comparisons between both programming languages.

A.3. Organization of this memory

Bearing in mind the previous specific objectives, we proceed to describe the organization of the rest of this report, structured in a series of chapters whose contents are described below:

- **Hyperspectral analysis**: the hyperspectral image concept and the linear mixing model are defined; some hyperspectral sensors (AVIRIS and EO-1 Hyperion) and some spectral libraries (USGS and ASTER) are mentioned; and finally, the need for parallelization and platforms that can be used to address the problem of performance improvement is presented.
- **FPGAs technologies**: FPGA technologies are defined in a short way.
- **Implementation**: the algorithm ATDCA-GS in series is defined and the parallelization and optimization that has been carried out in both VHDL and OpenCL languages is explained.

- **Results**: the results obtained after the implementation and execution of the algorithm in FPGAs devices are presented.
- **Conclusions and future work**: the main conclusions of the aspects addressed in the work that have been reached and also some possible lines of future work that can be performed in relation to this work are presented.

Apéndice B

Conclusions and future work

B.1. Conclusions

In this end-of-degree project, the design and implementation of the ATDCA-GS algorithm has been carried out, using Gram Schmidt orthogonalization in order to optimizing and improving the performance of complex operations such as the calculation of the inverse of a matrix. The programming languages VHDL and OpenCL have been used and the results of their execution in FPGA have been evaluated to subsequently make a performance comparison between both alternatives.

As part of the design, an adaptation of the algorithm to the usual flow of a specific hardware design has been carried out, minimizing as much as possible the amount of resources to be used and parallelizing the operations carried out during the execution of the algorithm.

To make a comparison in terms of the performance in time of the two implementations, it has been compared the acceleration of one with respect to the other making use of real and synthetic images. In addition, it has been verified that, except for the implementation in OpenCL for large images, the processing in both alternatives does not exceed the time limit (maximum) and therefore the real-time analysis can be performed, fulfilling one of the main objectives of this proyect.

The performance tests in terms of resources used in each implementation have revealed that the percentage of resources used increases linearly with the number of bands. It also

revealed that for a large number of them (256), the resource with the highest use hardly reaches 86 % of use in VHDL and 48 % in OpenCL, so it is concluded that the performance is adequate.

B.2. Lines of future work

In the first place, it would be convenient to improve the implementation optimizations in OpenCL so that it allows a real-time analysis as well as the other alternatives. In addition, since the tendency of the size of the images is to continue growing more and more, the future work option that seems more evident is to be able to process other real images of an even larger size.

A possible future work path for this work would be to develop the algorithm by converting the floating-point arithmetic to whole arithmetic. In this way a better performance would be obtained since the calculations would be even simpler and, therefore, the number of necessary resources would decrease while increasing the clock frequency.

Another possible way of continuation could be the modification of the test platform to use the PCIe 3x8 bus. In this way penalties due to I / O would be reduced.

Finally, another way would be to choose whether the implementation kernels in OpenCL can follow a parallel programming model at task level, so that the task refers to the execution of a kernel with a work-group that contains a work-item and, thus, the compiler tries to accelerate the only work-item to get a better performance.