ULPGC Universidad de Las Palmas de Gran Canaria | Instituto Universitario de Microelectrónica Aplicada IUMA

Máster Universitario en Electrónica y Telecomunicación Aplicadas (META)

Trabajo Fin de Máster

Implementación MPSoC FPGA de Algoritmos de *Machine Learning* para Aplicaciones Clínicas usando Metodologías de diseño de Alto Nivel

Autor: Mario Daniel Guanche Hernández
Tutor(es): Pedro Pérez Carballo
Sonia Raquel León Martín
Fecha: mayo de 2023

IUMA
t +34 928 451 150
+34 928 451 086
f +34 928 451 083
e: iuma@iuma.ulpgc.es
w: www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Versión: 1 – 20 julio 2020

ULPGC Universidad de Las Palmas de Gran Canaria | Instituto Universitario de Microelectrónica Aplicada IUMA

Máster Universitario en Electrónica y Telecomunicación Aplicadas (META)

Trabajo Fin de Máster

Implementación MPSoC FPGA de Algoritmos de *Machine Learning* para Aplicaciones Clínicas usando Metodologías de diseño de Alto Nivel

HOJA DE FIRMAS

Alumno/a:	**Mario Daniel Guanche Hernández**	Fdo.:
Tutor/a:	**Pedro Pérez Carballo**	Fdo.:
Tutor/a:	**Sonia Raquel León Martín**	Fdo.:

Fecha: **mayo de 2023**

IUMA
t +34 928 451 150
+34 928 451 086
f +34 928 451 083
e: iuma@iuma.ulpgc.es
w: www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Versión: 1 – 20 julio 2020

ULPGC Universidad de Las Palmas de Gran Canaria | Instituto Universitario de Microelectrónica Aplicada IUMA

Máster Universitario en Electrónica y Telecomunicación Aplicadas (META)

Trabajo Fin de Máster

Implementación MPSoC FPGA de Algoritmos de *Machine Learning* para Aplicaciones Clínicas usando Metodologías de diseño de Alto Nivel

HOJA DE EVALUACIÓN

Calificación:

Presidente _____ Fdo.:

Secretario _____ Fdo.:

Vocal _____ Fdo.:

Fecha: mayo 2023

IUMA
t +34 928 451 150
+34 928 451 086
f +34 928 451 083
e: iuma@iuma.ulpgc.es
w: www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Agradecimientos

A mi familia por su apoyo incondicional.

A mis tutores, Pedro Francisco Pérez Carballo y Sonia Raquel León Martín, por el asesoramiento.

Al servicio de soporte técnico del IUMA, por asistirme en la instalación e instrucciones requeridas para el uso de las aplicaciones en este trabajo.

Resumen

En este Trabajo Fin de Máster (TFM) se estudia la utilización de aceleradores *hardware* basados en MPSoC (*Multiprocessor System on Chip*) FPGAs (*Field-programmable Gate Array*) aplicados para preprocesar imágenes hiperespectrales con fines médicos, más concretamente para diagnosticar posibles casos de cáncer de piel. Para que las imágenes hiperespectrales resulten útiles, su contenido debe preprocesarse. Con ello, se minimiza en dicha información la influencia de factores ajenos a la naturaleza de la imagen.

Tras el preprocesado, pueden utilizarse las imágenes hiperespectrales resultantes para extraer conclusiones acertadas, automatizadas y útiles para el caso de uso. Para ello, el uso de algoritmos de *machine learning* resulta de gran idoneidad. En este TFM, se considera el algoritmo "k-means", que se emplea para determinar regiones en la piel captada en la imagen según sea tejido sano o afectado.

El preprocesado de imágenes hiperespectrales y el algoritmo "k-means" requieren una carga computacional elevada en una ejecución secuencial. Por ello, resulta muy conveniente implementar aceleración *hardware* que explote las posibilidades de paralelismo de estos algoritmos, reduciendo con ello su tiempo de ejecución y su consumo energético.

Para lograr dicha aceleración, se utilizan dispositivos MPSoC FPGAs sobre un sistema de prototipado Xilinx Zynq UltraScale+ MPSoC ZCU102. Su elemento central es un MPSoC que contiene, entre otros recursos, un sistema de multiprocesamiento y una FPGA con lógica programable (PL, *Programmable Logic*).

La aplicación *hardware/software* a implementar está formada por una aplicación *software* (app.) ejecutada como *host* y funciones de aceleración *hardware* o *kernels*. El host se ejecuta en la APU (*Application Processor Unit*) basada en un sistema de multiprocesamiento ARM Cortex A53. Los *kernels* se ejecutan en la FPGA. Para comunicar datos y control entre la APU la FPGA se utiliza un sistema de buses en chip basados en interfaces AMBA AXI4 configurables de alto ancho de banda disponibles en el MPSoC.

Desde la perspectiva de la metodología de desarrollo, la programación de la aplicación se realiza en C/C++. Por otra parte, el diseño de los *kernels* se realiza desde una

descripción algorítmica mediante la utilización de técnicas de síntesis de alto nivel (HLS). En esta metodología, el diseño de los *kernels* parte de una descripción funcional, sin información temporal (*untimed*) desarrollada en C/C++ combinada con un conjunto de directivas de síntesis que permiten especificar la arquitectura *hardware* del *kernel*. La interacción entre el *host* y los *kernels* se programa usando OpenCL (*Open Computing Language*). Tanto los *kernels* como el *host* se desarrollan utilizando la aplicación Vitis 2021.2 de Xilinx.

Siguiendo la metodología HLS, la verificación de la aplicación se realiza mediante técnicas de emulación *software* y, posteriormente, mediante emulación *hardware*. La emulación *software* permite la verificación funcional del algoritmo descrito para los *kernels*. Se realiza en C/C++, compilando ya sea sobre un *host* Linux x86 o mediante la utilización de QEMU. La emulación *hardware* posibilita verificar la implementación *hardware* de los *kernels* y obtener información sobre las prestaciones de esta.

Por último, se realiza el diseño final para su prototipado. Tanto el *bitstream* como los componentes *software* (sistema operativo + aplicación) se cargan desde una tarjeta SD (*Secure Digital*) en el MPSoC para su ejecución sobre el hardware real.

Tras desarrollar este trabajo, se obtiene una aplicación capaz de clasificar distintas regiones en imágenes hiperespectrales de pacientes afectados de cáncer de piel. Esta aplicación efectúa el preprocesado de la imagen y el posterior *clustering* de los píxeles, mediante aceleración *hardware* en FPGA. Esta aceleración ha supuesto una mejora significativa de la velocidad de cómputo y del consumo energético de la aplicación.

Por otra parte, a esta aplicación se la ha dotado de parámetros de configuración estáticos, además de cierta versatilidad para reconfigurar los *kernels* FPGA. Esta aplicación puede operar con imágenes hiperespectrales de diferentes dimensiones y formatos. Asimismo, la aplicación posee algunas opciones de configuración que facilitan su adaptabilidad a otras circunstancias de funcionamiento, incluyendo la implementación en otros dispositivos. Adicionalmente, se ha desarrollado un configurador que asiste al usuario en el ajuste de los *kernels* para aprovechar óptimamente la FPGA.

Abstract

This Master's Thesis (TFM) studies the use of hardware accelerators based on MPSoC (Multiprocessor System on Chip) FPGAs (Field-programmable Gate Array) applied for processing hyperspectral images for medical purposes, more specifically for diagnosing possible cases of skin cancer. In order to make hyperspectral images useful, their content must be pre-processed. This minimizes the influence of factors unrelated to the nature of the image on the information. After preprocessing, it is desirable to use the resulting hyperspectral images to extract accurate, automated and useful conclusions for the use case. To do so, utilizing machine learning algorithms is highly suitable. In this work, the "k-means" algorithm is considered, using it to determine regions in the skin captured in the image depending on whether it is healthy or affected tissue.

Pre-processing hyperspectral images and the k-means algorithm involve a high computational load in a sequential execution. For this reason, it is highly desirable to implement hardware acceleration that exploits the parallelization possibilities of these algorithms, thereby reducing their runtime and energy consumption.

To achieve this acceleration, MPSoC FPGAs are used on a Xilinx Zynq UltraScale+ MPSoC ZCU102 prototyping board, which embodies an MPSoC as central element containing, among other resources, a multiprocessing system and an FPGA with programmable logic (PL, Programmable Logic). The hardware/software application to implement consists of a software application (app.) running as host and hardware acceleration functions or kernels running on the FPGA. The host runs on the APU (Application Processor Unit) based on an ARM Cortex A53 multiprocessing system. The kernels run on the FPGA. For communicating data and control between the APU and the FPGA, an on-chip bus system based on configurable high-bandwidth AMBA AXI4 interfaces available on the MPSoC is used

From the point of view of the development methodology, the programming of the application is carried out in C/C++. On the other hand, the design of the hardware kernels is based on an algorithmic description using high-level synthesis techniques (HLS). In this methodology, the design of the kernels starts from a functional description, without

temporal information (untimed) developed in C/C++ combined with a set of synthesis directives that allow specifying the hardware architecture of the kernel. The interaction between host and kernels is programmed using OpenCL (Open Computing Language). Both kernel and host are done using the Xilinx Vitis 2021.2 application.

Following the HLS methodology, the verification of the application is carried out by using software emulation techniques and, subsequently, hardware emulation. Software emulation provides a functional verification of the algorithm described for the kernels is carried out. This is done in C/C++, compiling either on a Linux x86 host or using QEMU. Hardware emulation allows to verify the hardware implementation of the kernels and to obtain information about its performance. Finally, the final design is prototyped. Both the bitstream and the software components (operating system + application) are loaded from an SD (Secure Digital) card into the MPSoC for execution on the real hardware.

After developing this work, an application capable of classifying different regions in hyperspectral images of patients affected by skin cancer is obtained. This application performs image pre-processing and subsequent pixel clustering by using hardware acceleration via FPGA. This acceleration has led to a significant improvement in the computational speed and energy consumption of the application.

On the other hand, this application has been provided with static configuration parameters, as well as a certain versatility for reconfiguring the FPGA kernels. This application can operate with hyperspectral images of different dimensions and formats. It also has some configuration options to facilitate the adaptability of the application to other operating circumstances, including its implementation in other devices. In addition, a configurator has been developed to assist user when adjusting kernels for optimal use of the FPGA.

Tabla de contenidos

Índice de figuras

Índice de Tablas

Índice de Ecuaciones

Índice de códigos

Acrónimos

ACE	AXI Coherency Extension
ACP	Accelerator Coherency Port
AES	Advanced Encryption Standard
ALU	Arithmetic-Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
AMD	Advanced Micro Devices
API	Application Programming Interface
APU	Application Processor Unit
ARM	Advanced RISC Machine
AXI	Advanced eXtensible Interface
BRAM	Block RAM
BIL	Bands Interleaved by Line
BIP	Bands Interleaved by Pixel
BSQ	Band Sequential
CCD	Charge-Coupled Device
CCI	Cache-Coherent Interconnect
CMOS	Complementary Metal Oxide Semiconductor
CLB	Configurable Logic Block
CPU	Central Processing Unit
CRC	Cyclic Redundancy Checking
CSU	Configuration and Security Unit
DDR	Double Data Rate
DMA	Direct Memory Access
DNS	Domain Name System
DRAM	Distributed RAM
DSP	Digital Signal Processing
ECC	Error Correcting Code
EMIO	Extended Multiplexed Input Output
EXT4	Fourth Extended File System
FEC	Forward Error Correction
FF	Flip-Flop
FinFET	Fin-shaped Field Effect Transistor
FIFO	First In – First Out
FPD	Full-Power Domain
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GIC	Generic Interrupt Controller
GNU	GNU’s Not Unix
GPU	Graphics Processing Unit
GTH	Gigabit Transceiver H
HLS	High-Level Synthesis
HPCP	High Performance Coherent Port
II	Initiation Interval
IP	Internet Protocol

IRQ	Interrupt Request
MOESI	Modified Owned Exclusive Shared Invalid
MPSoC	MultiProcessor System on Chip
MMU	Memory Management Unit
LED	Light-Emitting Diode
LPD	Low-Power Domain
LPDDR	Low Power Double Data Rate
LUT	Look-Up Table
NMSC	Non-Melanoma Skin Cancer
OCM	On-Chip Memory
OTG	On The Go
OpenCL	Open Computing Language
PC	Personal Computer
PCIe	Peripheral Component Interconnect express
PL	Programmable Logic
PS	Processing System
PMU	Platform Management Unit
RAM	Random Access Memory
RGB	Red, Green, and Blue
ROM	Read-Only Memory
RPU	Real-Time Processing Unit
RSA	Rivest Shamir and Adleman
RTL	Register-Transfer Level
SAM	Spectral Angle Mapper
SATA	Serial Advanced Technology Attachment
SCU	Snoop Control Unit
SD	Summed Distances
SD	Secure Digital
SEOM	Sociedad Española de Oncología Médica
SFF	Spectral Feature Fitting
SHA	Secure Hash Algorithm
SIMD	Single Instruction Multiple Data
SMMU	System Memory Management Unit
SoC	System on Chip
SRAM	Static RAM
SSH	Secure Shell
SVM	Support Vector Machine
TCF	Target Communications Framework
TCM	Tightly Coupled Memory
TFM	Trabajo Fin Máster
TSMC	Taiwan Semiconductor Manufacturing Company
XOR	eXclusively-OR
XRT	Xilinx Runtime
XSCT	Xilinx Software Command-line Tool
ULPI	UTMI+ Low Pin Interface
USB	Universal Serial Bus

UTMI+	USB 2.0 Transceiver Macrocell Interface
VCU	Video Codec Unit
VFAT	Virtual File Allocation Table
VNIR	Visible and Near Infrared

Capítulo 1. Introducción

1.1. Introducción

Las imágenes hiperespectrales constituyen una herramienta de gran utilidad para su aplicación con fines analíticos. Entre sus ventajas se encuentra que constituye una operación de análisis no invasivo que no requiere la destrucción de las muestras captadas para efectuarse. Además, constituye una técnica no ionizante que no requiere de contacto físico. Estas cualidades ocasionan que las imágenes hiperespectrales se apliquen en múltiples ámbitos y disciplinas, tales como la medicina, el análisis de productos alimentarios, la agricultura o la astronomía [1], [2], [3].

Cada imagen hiperespectral constituye una matriz de datos tridimensional, también denominada hipercubo. Cabe destacar que dos de estas dimensiones se corresponden al largo y ancho espacial de la imagen. Asimismo, la otra dimensión se corresponde al dominio espectral, es decir, a las longitudes de onda electromagnética a considerar en la "visualización" y análisis por parte de la aplicación de la imagen. Esta tridimensionalidad constituye una característica que las imágenes hiperespectrales comparten con las imágenes RGB (*Red, Green and Blue*) y las imágenes multiespectrales. No obstante, las imágenes hiperespectrales se distinguen de las otras dos categorías al considerar una mayor cantidad de longitudes de ondas o bandas espectrales. Dicho aspecto se traduce en un dimensionado espectral de mayor longitud como se representa en la Figura 1. De esta manera, las imágenes hiperespectrales constituyen un recurso que permite aunar información espacial, obtenida en una imagen estándar, con la información espectral,

obtenida mediante espectroscopio, consiguiendo para ambos un nivel de resolución intermedio [1], [4].

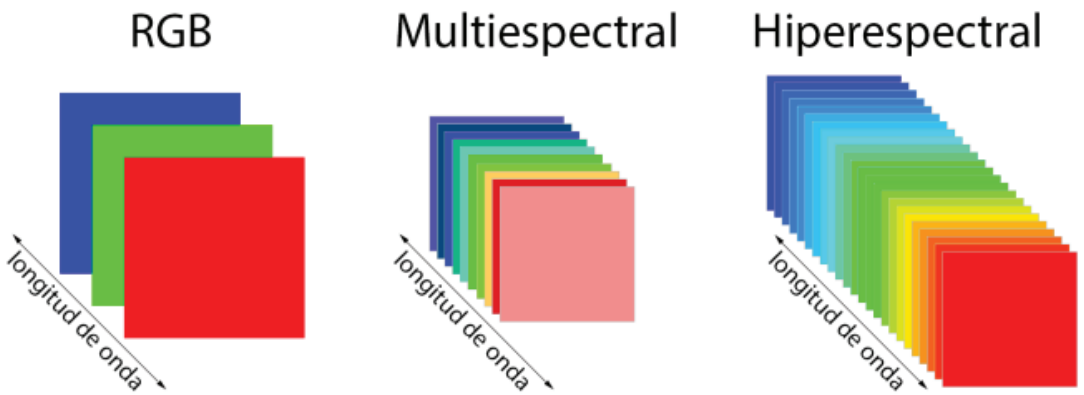


Figura 1: Comparativa imágenes RGB, multiespectrales e hiperespectrales [5]

Mediante la combinación de información espacial y espectral, las imágenes hiperespectrales permiten determinar no solo la naturaleza de los elementos materiales observados, sino que también la distribución de estos. Para que esta información resulte útil, es necesario eliminar la influencia de factores ajenos a la composición de los materiales captados en la imagen hiperespectral. Entre estos factores se encuentra la iluminación desigual a lo largo tanto del rango espacial como espectral de la imagen, la sensibilidad no uniforme de la cámara de captación y el ruido eléctrico introducido por la cámara. Al eliminar estas influencias, se obtiene la firma espectral de cada píxel, la cual se debe al espectro de reflexión y absorción en las distintas bandas espectrales que caracterizan y distinguen entre sí a los elementos materiales observados [2], [3], [6], [7].

El caso de utilización de las imágenes hiperespectrales de este Trabajo Fin de Máster (TFM) se engloba en el ámbito de la medicina, más concretamente, en la detección de cáncer de piel. Esta enfermedad constituye uno de los tipos de cáncer más comunes a nivel mundial, distinguiéndose entre la categoría no melanoma o NMSC (*Non-Melanoma Skin Cancer*) y el melanoma [2]. De acuerdo con los datos de la Sociedad Española de Oncología Médica (SEOM), en el año 2020 se han diagnosticado en España 6.179 casos de melanoma [8]. Sin embargo, la situación de los cánceres de piel no melanoma resulta más grave, puesto que constituyen la tipología de cáncer más frecuente a nivel mundial, superando a la suma de los restos de casos de cáncer existentes. Dentro de los NMSC existen a su vez

diversas tipologías, algunas de las cuales resultan benignas mientras que otras resultan malignas [9].

Para que el tratamiento de estas patologías resulte eficaz, es necesario que la afección se detecte precozmente. Sin embargo, la distinción de cáncer piel, así como de su tipología, mediante inspección visual directa presenta bastante incertidumbre. En consecuencia, es frecuente que el médico, incluso siendo un dermatólogo experimentado, se vea obligado a recurrir a métodos invasivos como la biopsia para aseverar si el tipo de afección cutánea del paciente pertenece a una tipología benigna o maligna [2]. Ante esta situación, la utilización de imágenes hiperespectrales resulta de gran idoneidad, ya que permiten visualizar un espectro electromagnético de la imagen más amplio y preciso que el del ojo humano. De esta manera, se logra que las diferencias entre el tejido cutáneo en función de su afección resulten más evidentes. Dichas diferencias se deben a cómo los cambios bioquímicos y morfológicos que definen a las distintas lesiones se manifiestan ópticamente. Por otra parte, al ser la utilización de imágenes hiperespectrales un método no invasivo y no ionizante, se garantiza de manera absoluta que su aplicación no perjudica al tejido objeto de la evaluación [2].

Aunque la información contenida en la imagen hiperespectral facilita determinar con mayor certeza la naturaleza y la distribución de lo observado, sigue requiriéndose el análisis de los datos de la imagen hiperespectral para determinar conclusiones correctas y útiles para el problema. Dicho proceso de análisis puede automatizarse mediante algoritmos de *machine learning*, minimizándose con ello el tiempo de diagnóstico y la carga de trabajo del usuario [2], [6], [7]. En el caso de la identificación de afecciones cutáneas, la aplicación de *machine learning* resulta especialmente útil, debido a que la evaluación es fácilmente "matematizable", aunque se requiere para ello de una cantidad de datos elevada.

Para la clasificación de los píxeles, en este trabajo se ha optado por emplear un algoritmo de *machine learning* de *clustering* denominado "k-means". Dicho algoritmo subdivide las muestras, correspondientes a los píxeles de la imagen, en un conjunto de agrupaciones o *clústeres*. Esta subdivisión se efectúa a partir de la semejanza relativa presente entre las muestras en función de los atributos que se evalúen. Estos atributos se

corresponden con las bandas espectrales de la imagen hiperespectral utilizadas. El objetivo de esta segmentación consiste en que los píxeles más similares entre sí se agrupen en un mismo *clúster*, maximizando al mismo tiempo las diferencias en la firma espectral de los píxeles ubicados en *clústeres* distintos [2], [10].

Idealmente, los píxeles pertenecientes a un mismo *clúster* identifican un estado de afección de la piel similar a lo largo de la región de la piel en la que se distribuyen. Para que esta situación ocurra, deben escogerse bandas espectrales representativas, es decir, aquellas en las que el coeficiente de absorción del tejido varíe más significativamente entre los diferentes estados de afección. Además, los estados de afección deben poseer una influencia predominante con respecto a los demás factores en la variación de la firma espectral de los píxeles [2], [10]. La aplicación desarrollada permite configurar durante la compilación la cantidad de *clústeres* que el algoritmo "k-means" debe determinar. Sin embargo, los resultados y el estudio mostrados en [2] determinaron que los mejores resultados de *clustering* para este ámbito de uso se obtienen para una segmentación en 3 *clústeres*.

Desde la perspectiva de la implementación, las imágenes hiperespectrales presentan como inconveniente la inmensidad de datos que estas contienen y que deben evaluarse en la aplicación para lograr un nivel de resolución tanto espacial como espectral de la imagen que resulte suficiente. En consecuencia, la ejecución de los algoritmos de utilización y evaluación de imágenes hiperespectrales en un único hilo de ejecución requieren de un consumo de tiempo y energía elevados. Esta situación se empeora al aplicar algoritmos de *machine learning* para la clasificación de los píxeles de la imagen hiperespectral. Esto último se debe a la elevada cantidad de reevaluaciones de las muestras a clasificar que estos algoritmos precisan. Para sobrellevar estas circunstancias, resulta idóneo explorar la explotación de la computación en paralelo mediante aceleradores *hardware* que permitan un elevado grado de paralelismo del procesamiento [11]. Como recursos *hardware* que posibiliten esta función de aceleración se encuentran las GPUs (*Graphics Processing Unit*) y las FPGAs (*Field Programmable Gate Array*), utilizándose esta última tipología para esta aplicación.

La aplicación desarrollada con este trabajo está orientada a ser implementada en una placa de prototipado ZCU102 desarrollada por la empresa Xilinx. El elemento central de esta placa de prototipado consiste en un MPSoC (*MultiProcessor System on Chip*) entre cuyos recursos *hardware* se encuentran una APU (*Application Processor Unit*) de 4 núcleos y una FPGA. Este último recurso está constituido por varios tipos de módulos *hardware* de funcionalidad simple que son replicados intensivamente a lo largo de la FPGA. Se utiliza la APU como *host* de la aplicación con aceleración *hardware* a implementar. Por otra parte, la FPGA es responsable de implementar los aceleradores *hardware* de la aplicación.

Aunque la placa de prototipado de referencia sea la ZCU102, en el desarrollo de este TFM se ha procurado que la aplicación sea lo más fácilmente adaptable posible a otras circunstancias de funcionamiento. En este sentido, se han establecido una serie de parámetros de configuración, así como un procedimiento, que permite ajustar la configuración de los aceleradores *hardware* a cualquier placa de prototipado que disponga de FPGA para la función de aceleración. Estos ajustes requieren modificaciones mínimas del código fuente del usuario, y permiten optimizar el aprovechamiento de los módulos *hardware* que disponga la FPGA para conseguir el máximo rendimiento temporal. Además, se disponen de otros aspectos configurables tales como el formato de los datos de la imagen hiperespectral de entrada, las dimensiones de la imagen hiperespectral y el formato de entrada de la imagen hiperespectral. Estos dos últimos parámetros son dinámicamente ajustables, es decir, pueden cambiarse de una ejecución de la aplicación a la siguiente sin necesidad de recompilar.

1.2. Objetivos

El objetivo de este Trabajo Fin Máster consiste en el análisis y diseño de una aplicación de *machine learning* para el análisis de imágenes hiperespectrales, incluyendo el preprocesamiento de las imágenes y recurriendo a la aceleración *hardware* mediante FPGA para optimizar su rendimiento. En dicho desarrollo se utiliza como referencia una aplicación análoga basada en aceleración *hardware* mediante GPUs. La programación de la aceleración *hardware* en FPGA se efectúa siguiendo la metodología HLS (*High-Level Synthesis*). Este objetivo principal se subdivide en los siguientes objetivos operacionales:

O1. Estudiar la aplicación.

O2. Definir la arquitectura de la aplicación, incluyendo partición *hardware*/*software.*

O3. Modelar en alto nivel del sistema y verificación funcional.

O4. Implementar la arquitectura sobre plataforma MPSoC.

O5. Evaluar los resultados obtenidos

O6. Documentar el trabajo realizado

1.3. Estructura del documento

El contenido de este documento se estructura en diez capítulos, tal como se describe a continuación. En el capítulo 1. se contextualiza e introduce el trabajo abordado, definiendo los objetivos a cumplir. En el capítulo 2 se describen algunos algoritmos de *machine learning,* fundamentalmente algoritmos de *clustering*, puesto que la funcionalidad de *machine learning* implementada está englobada en esta categoría.

En el capítulo 3 se exponen diferentes aspectos acerca de las imágenes hiperespectrales y del tratamiento de la información que incluyen, clave para programar la utilización de estas imágenes. En el capítulo 4 se describe la aplicación basada en aceleración *hardware* mediante GPUs que se toma como referencia para el desarrollo de esta aplicación.

Los capítulos siguientes están orientados a la metodología de diseño e implementación. Así en el capítulo 5. se describe brevemente la placa de prototipado ZCU102 utilizada para ejecutar la aplicación sobre el MPSoC disponible. En el capítulo 6 se presentan los recursos de programación en OpenCL utilizados para la transferencia de datos y sincronización entre el *host* y los *kernels* FPGA. Para completar esta sección, en el Capítulo 7 se explica la metodología de diseño. Esta última explicación incluye la forma de emplear las directivas de compilación requeridas para controlar la síntesis de los distintos *kernels* implementados en la FPGA, así como su impacto en los resultados obtenidos.

Para concluir el trabajo, en el capítulo 8. se describe el sistema en chip desarrollado, así como los procedimientos efectuados para facilitar su depuración, su ajuste y su ejecución en la placa de prototipado. En el capítulo 9 se exponen y discuten los resultados obtenidos, tanto en términos de utilización de recursos *hardware*, como en términos de rendimiento temporal y de los resultados de salida que genera su ejecución. Finalmente, en el capítulo 10 se exponen las conclusiones de este trabajo y se indican líneas futuras propuestas como resultado de la experiencia adquirida durante el desarrollo de este trabajo.

Capítulo 2. *Machine learning*

2.1. Introducción

De acuerdo con Arthur Samuel, un algoritmo de *machine learning* se define como aquel que aprende y mejora su efectividad de manera autónoma evaluando los datos recibidos. Para ello, recurren al análisis estadístico con el fin de establecer predicciones lo más certeras posible. Estos algoritmos pueden subdividirse en tres clases [12]:

a. **Aprendizaje supervisado.** Estos algoritmos precisan, en primer lugar, tomar múltiples muestras de datos que poseen un valor de salida predefinido. Una parte de estas muestras se emplean en el entrenamiento del algoritmo. Dicho proceso consiste en ajustar la funcionalidad y los parámetros de las operaciones internas del algoritmo con el objetivo de maximizar que los datos salientes se asemejen a los valores de salida preestablecidos para cada muestra. Posteriormente, se utilizan el resto de las muestras de salida predefinida para verificar que las operaciones del algoritmo estén bien ajustadas. Estos procesos posibilitan que el algoritmo sea capaz de determinar qué salida le corresponde a cada muestra de entrada posterior sin necesidad de que se le especifique explícitamente cuál es la salida esperable.

   Este tipo de algoritmos se utilizan, por ejemplo, en la detección de *spam* en los mensajes de correo electrónico. El valor saliente puede ser la pertenencia de la muestra a alguna categoría preestablecida o algún valor cuantitativo. En la primera situación, se considera que el algoritmo de *machine learning* es de clasificación. En el otro caso, se considera como un algoritmo de regresión [12], [13], [14].

b. **Aprendizaje sin supervisión.** Estos algoritmos comienzan directamente con la evaluación de muestras no preclasificadas, sin ningún entrenamiento previo. El objetivo de estos algoritmos puede consistir en establecer agrupaciones basándose en las similitudes entre las muestras entrantes. Los algoritmos de estas características se denominan algoritmos de *clustering*.

   Por otra parte, se encuentran los algoritmos cuyo propósito consiste en identificar correlaciones entre los atributos o características de las muestras evaluadas. Dichos algoritmos se categorizan como algoritmos de asociación, [12], [13], [14].

c. **Aprendizaje mediante refuerzo.** El algoritmo, también definido como agente en esta categoría, aprende al interactuar con el entorno considerado. Este aprendizaje puede entenderse desde la dinámica de premio y castigo. Asimismo, cada acción ejecutada por el algoritmo conlleva una valoración. Esta valoración puede resultar negativa (castigo) o positiva (recompensa) en función de las consecuencias que originan y de cómo estas se ajustan a lo deseable. El objetivo del algoritmo consiste en maximizar la recompensa. Por consiguiente, dicho algoritmo reiterará aquellas acciones que otorgaron una mayor valoración positiva. Simultáneamente, el algoritmo evitará las acciones que conllevaron una valoración negativa [12], [13], [14].

2.2. Información de entrenamiento

La fiabilidad y calidad de los datos de entrada constituyen aspectos claves para la veracidad de los resultados de *machine learning*. Esto es relevante en las muestras de entrenamiento supervisadas. No obstante, las muestras pueden contener errores o defectos de información. Se necesita, por tanto, que el sistema presente cierta capacidad tratar estos defectos sin que se modifiquen los resultados obtenidos.

Los defectos más relevantes identificados son los siguientes [10], [14]:

a. **Atributos irrelevantes**. Es posible que algunos de los atributos considerados para las muestras resulten inútiles, puesto que carezcan de influencia y correlación con la categoría a la que cada muestra pertenezca. Este hecho implica una carga

computacional innecesaria, aunque inevitable en aquellos algoritmos en los que el establecimiento de los atributos esté automatizado, sin ser el usuario o programador quien directamente los escoja [10], [14].

b. **Ausencia de atributos representativos**. Contrariamente al caso anterior, otro defecto que puede ocurrir consiste en la desconsideración en el cómputo del algoritmo de atributos representativos, es decir, que poseen gran utilidad para discernir la pertenencia de las muestras a las categorías existentes. Este defecto puede ocasionar incongruencias en el criterio de clasificación determinado, e incluso impedir que las muestras sean clasificadas correctamente por el algoritmo [10], [14].

c. **Atributos redundantes.** Ocurre cuando 2 o más atributos pueden inferirse mutuamente. Para ejemplificar este defecto, considérese un caso donde las muestras de entrenamiento son un conjunto de personas y 2 de los atributos son sus fechas de nacimiento y sus edades. En dicho caso, aunque la información de estos atributos fuera relevante para la clasificación, se puede escoger prescindir de uno cualquiera de ellos. Dicha decisión no afectaría significativamente a los resultados obtenidos de manera relevante y supondría un ahorro de carga computacional. No obstante, la importancia de este defecto en comparación con los comentados en los párrafos anteriores es muy minoritaria [10], [14].

d. **Valor desconocido**. Puede suceder que el valor de algunos atributos se desconozca para algunas muestras [10], [14].

e. **Valores de atributos erróneos**. Los valores de los atributos pueden presentar cierto margen de error. Entre las causas de este defecto podemos citar: recurrir a fuentes de información poco fiables, erratas, la inexactitud de los instrumentos de medida o la equivocación del usuario [10], [14].

f. **Categorías mal definidas.** Esta condición puede causar que las muestras no se adecúen correctamente a las categorías definidas atendiendo a los valores de sus atributos. Consecuentemente, una mínima variación en los valores de los atributos de la muestra puede traducirse en una errónea clasificación de la misma [10], [14].

2.3. Clasificadores lineales y polinómicos

Los clasificadores lineales y polinómicos constituyen algoritmos de *machine learning* supervisados. Para entender estos clasificadores, deben entenderse las muestras de entrenamiento como puntos distribuidos en un espacio vectorial cuyas dimensiones se corresponden a los atributos de las muestras. Estas muestras de entrenamiento se emplean para determinar las regiones que se asocian a cada clase. Asimismo, las muestras posteriores se clasificarán dependiendo las regiones delimitadas [10], [14].

La correcta delimitación de las regiones recae en la correcta definición y determinación de sus fronteras. Dichas fronteras se describen matemáticamente mediante una o varias ecuaciones lineales o polinómicas. Cada una de estas ecuaciones permitirán establecer una separación binaria entre las muestras pertenecientes a una determinada clase y las demás. En el caso de las ecuaciones polinómicas, el grado deberá ser mayor cuanto más complejas resulte la frontera que describe. Con respecto a la ecuación lineal, esta se ajustará a la fórmula expuesta en la ecuación (1). En dicha ecuación, "x_i" identifica a los atributos de las muestras a clasificar. Además, los parámetros "w_i" indican la ponderación otorgada al atributo que multipliquen, pudiendo ser un valor positivo o negativo. Por último, "w_0" representa el valor umbral [10], [14].

$$w_0 + \sum_{i=1}^{n} w_i x_i = 0 \qquad (1)$$

El propósito del entrenamiento de los clasificadores lineales consistirá en establecer el valor de los parámetros de ponderación ("w_i" y "w_0"). Asimismo, la categorización de la muestra a clasificar depende del resultado obtenido al considerar el valor de sus atributos. En este aspecto, la muestra se clasificará como perteneciente a una clase o a otra según esta tome un valor positivo o negativo. Si el resultado es 0, significa que la muestra se localiza exactamente en la frontera entre clases que establece la ecuación. En este caso excepcional, la clasificación de la muestra puede asignarse de manera aleatoria, o bien a la clase mayoritaria entre las muestras de entrenamiento.

Existen varios algoritmos para determinar los coeficientes w consistentes en el ajuste progresivo de los mismos mediante estrategias de ensayo-error. Entre estos

algoritmos se encuentra el de *Perceptron*, que actualiza los coeficientes mediante sumas y restas, y el de *WINNOW*, que recurre a la multiplicación y a la división para dicha actualización. Otro caso de algoritmo de clasificación lineal o polinómica relevante se corresponde al algoritmo SVM (*Support Vector Machine*) [10], [14].

2.3.1. *Support Vector Machine* (SVM)

En clasificadores polinómicos, el *Support Vector Machine* (SVM) constituye un procedimiento que, aparte de determinar una ecuación que separe un conjunto de muestras preclasificadas en función de 2 clases preexistentes, pretende maximizar la distancia de separación entre la representación de esta ecuación en el espacio vectorial y las muestras. Para ello, se establecen como referencia dos muestras, una en cada región delimitada por la ecuación. Dichas muestras deben pertenecer a la clase correspondiente a dicha región y ser las más próximas en sus respectivas clases al hiperplano definido. Las dos muestras que cumplan estos requisitos se denominan vectores de soporte o *support vectors* [10]. Para su determinación, se ajusta la ecuación de clasificación expuesta en la ecuación (2) para los vectores de soporte.

$$\left| w_0 + \sum_{i=1}^{n} w_i x_i \right| = 1 \qquad (2)$$

De esta manera, si el clasificador polinómico es correcto, todas las muestras de entrenamiento deberán originar un resultado igual o superior a la unidad. Los coeficientes "w_i" que ponderan la influencia de un determinado atributo "x_i" se determinan como la diferencia de valor en dicho atributo entre los vectores de soporte. La ecuación (2) presenta 2 vertientes, una correspondiente a las muestras que presentan una clasificación negativa (y=-1) y otra para las que presentan una clasificación positiva (y=1). Asimismo, en el caso de las muestras clasificadas como negativas el resultado del argumento de la función de extracción del valor absoluto debe resultar menor o igual a -1, mientras que las otras muestras deberán dar un resultado igual o superior a 1. Considerando "y_-" la clasificación de las muestras pertenecientes a las clases negativas e "y_+" la de las muestras de la clase positiva, las muestras de entrenamiento deben ajustarse a las inecuaciones expuestas en la ecuación (3) [13].

$$y_+\left(w_0+\sum_{i=1}^{n} w_i x_i\right)\geq 1$$
$$y_-\left(w_0+\sum_{i=1}^{n} w_i x_i\right)\geq 1 \qquad (3)$$

Dependiendo de la exigencia de la aplicación con respecto a las clasificaciones erróneas, el SVM se subdivide en dos modalidades denominadas *hard-margin* y *soft-margin*. La modalidad *hard-margin* constituye la opción que establece que el clasificador SVM desarrollado debe categorizar correctamente cada una de las muestras de entrenamiento. Este hecho implica que, al menos en el espacio vectorial creado por los atributos de las muestras, las clases deben ser linealmente separables. Por otro lado, la *soft-margin* consiente una cierta cantidad de errores como aceptables [15].

Estas versiones básicas del clasificador solo resultan útiles para problemas de clasificación lineales. Sin embargo, es posible extender la funcionalidad a problemas de clasificación no lineales mediante la incorporación de *kernels*. En este contexto, un *kernel* se define como una función matemática cuyas variables independientes se corresponden a atributos de las muestras. Como resultado, se obtienen una serie de nuevos atributos derivados matemáticamente de los atributos originales.

La inclusión de nuevos atributos para la descripción de las muestras introduce nuevas variables para expresar la ecuación lineal del clasificador. De esta manera, se facilita la posibilidad de que las muestras sean linealmente separables, reinterpretando la ecuación lineal resultante como una ecuación no lineal desde la perspectiva de los atributos originales. Para ello, es suficiente con invertir la operación de los *kernels* sobre la determinación de los atributos derivados de la ecuación lineal [15].

Las funciones *kernel* pueden ser de diversa índole. Sin embargo, dependiendo de la distribución de las muestras a lo largo del espacio vectorial, la utilidad de la función escogida para posibilitar una separación lineal entre las clases varía notablemente [15].

Algunos de los *kernels* más comúnmente usados son los siguientes:

***Kernels* radiales**. Estos *kernels* se ajustan a la ecuación expuesta en (4). En dicha ecuación, “x” representa un conjunto de los atributos iniciales de la muestra, mientras “x’”

constituye los valores para cada atributo en los que se maximiza el valor del atributo derivado, es decir, el resultado de la ecuación [15].

$$K(x, x') = e^{-\gamma\|x-x'\|^2} \tag{4}$$

La denominación de este *kernel* como radial se debe a que, considerando "x" y "x'" vectores con una cantidad de dimensiones cualquiera, el valor de salida del *kernel* para cualquier muestra disminuirá cuanto más se aleje del punto de referencia "x'" dentro del espacio vectorial. Asimismo, la dirección en la que se oriente dicho distanciamiento carecerá de efecto sobre el resultado del *kernel*. En la Figura 2 se representa la utilización de los *kernels* para establecer la separación lineal entre clases [15].

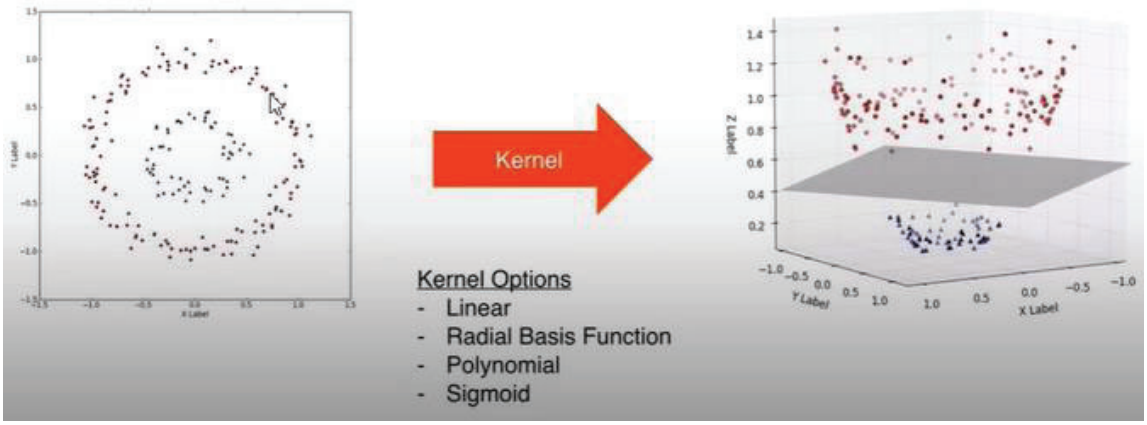


Figura 2: Aplicación de kernel radial para separar clases linealmente [16]

Otro parámetro relevante es "γ", el cual establece cuán significativo es el decaimiento del resultado del *kernel* para un mismo grado de divergencia entre "x" y "x'". Desde el punto de vista gráfico, este hecho implica que cuanto mayor sea el valor de "γ" más estrecha será la concavidad de la representación del *kernel* en función del parámetro "x". Debido a ello, la utilización de *kernels* con mayor "γ" permite dibujar, desde la perspectiva del espacio vectorial formado por los atributos originales, fronteras más complicadas y con mayores grados de curvaturas. Un ejemplo de este caso se presenta en la Figura 3 [15].

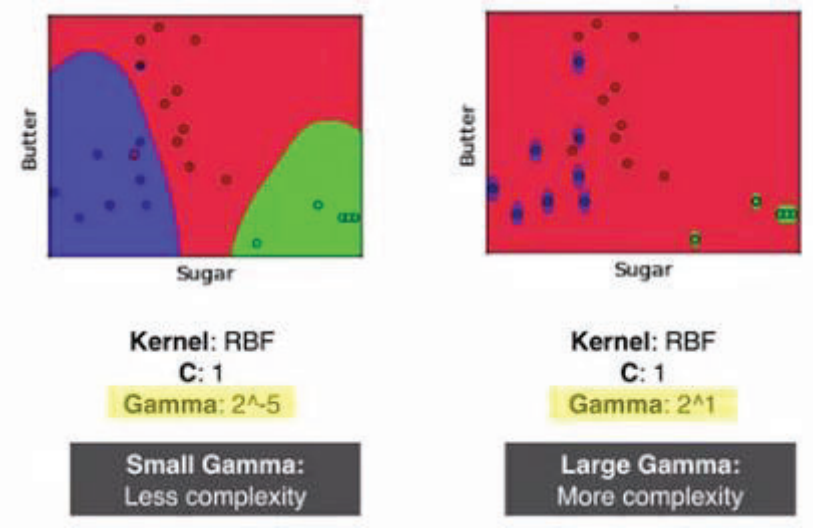


Figura 3: Caso de "γ" reducida (parte izquierda) y elevada (parte derecha) [17]

Cabe destacar que un mayor "γ" reduce la probabilidad de que el clasificador inferido coloque las muestras de entrenamiento preclasificadas en regiones no correspondientes a sus clases. Sin embargo, un valor excesivo de "γ" conduce al sobreajuste u *overfitting*, es decir, que el clasificador desarrollado esté demasiado particularizado para el conjunto de muestras de entrenamiento utilizado, siendo incapaz de categorizar correctamente otros conjuntos de muestras con un grado de acierto razonable [15].

2.3.2. Clasificadores binarios múltiples

En los algoritmos de clasificación lineal y polinómica, cada ecuación constituye un clasificador binario, puesto que solo distingue en dos categorías según el resultado que devuelva la muestra. Por ello, si se desea clasificar las muestras en más de dos clases, la manera más habitual de proceder consiste en aplicar múltiples ecuaciones. Cada una de las ecuaciones se asigna unívocamente a una de las clases definidas para la clasificación. De esta manera, el cometido de cada ecuación consiste en separar las muestras pertenecientes a la clase asignada de las demás. Cada una de estas ecuaciones se extrae utilizando un algoritmo de clasificación binario, como los algoritmos de *Perceptron*, *WINNOW* y SVM indicados anteriormente [10], [14].

Considerando que cada muestra debe pertenecer solamente a una clase, debe suceder, desde la perspectiva aritmética, que cada muestra retorne un resultado positivo para una de las ecuaciones y un resultado negativo para las ecuaciones restantes, considerando el resultado positivo como la asignación a la clase asociada a la ecuación.

Desde la perspectiva gráfica, esto implicaría que las regiones delimitadas por las ecuaciones no se intercepten. No obstante, esta situación puede no suceder debido a que las muestras de entrenamiento resulten demasiado escasas o poco adecuadas para ajustar la ecuación con la precisión adecuada a la región del espacio vectorial en la que se encuentran las muestras pertenecientes a la categoría que distinga.

Por otra parte, puede ocurrir que las regiones sean demasiado complejas para el grado de ecuación considerado, en el caso de los algoritmos de *Perceptron* y *WINNOW*, o para la capacidad de complejidad de las ecuaciones que posibilitan los *kernels* en el caso del SVM. En estos casos, las regiones de las clases no podrían acotarse con suficiente precisión, siendo posible que las aproximaciones a estas regiones que efectúen las ecuaciones se entrecorten. Para solventar esta circunstancia, suele considerarse que la muestra pertenece a la clase cuya ecuación devuelve un resultado más elevado [10], [14].

Este procedimiento para la clasificación de las muestras en más de dos clases solo es fiable si la cantidad de clases definidas es relativamente pequeña, en torno a 3 y 5. El motivo de ello se encuentra en que considerar más clases ocasionaría una relación demasiado desproporcionada entre las muestras de entrenamiento preclasificadas como positivas y las de valor negativo en cada ecuación, siendo los casos de clasificación negativa mayoritarios. En consecuencia, sería difícil la correcta clasificación de las muestras en aquellos casos en los que los valores de los atributos posean cierto grado de error en las muestras [10], [14].

2.4. *Clustering*

Un algoritmo de *clustering* se define como aquel que divide un conjunto de muestras no preclasificadas en varias agrupaciones o *clústeres*. Asimismo, los valores que los atributos de una muestra deben poseer para que esta se asigne a un determinado *clúster* no se encuentran predefinidos. En su lugar, cada muestra se asignará al *clúster* que contenga las muestras con el mayor grado de semejanza promedio. En consecuencia, cada *clúster* agrupará las muestras que presenten una mayor similitud entre ellas y, simultáneamente, una mayor diferenciación en promedio a las muestras asignadas a otros *clústeres.* En la Figura 4 se presenta un ejemplo con un conjunto de muestras distribuidas,

en función del valor de sus atributos "*height*" y "*weight*", en un espacio vectorial bidimensional. En dicho ejemplo, los *clústeres* a establecer resultan evidentes, pudiendo seguir las subdivisiones redondeadas [10], [14].

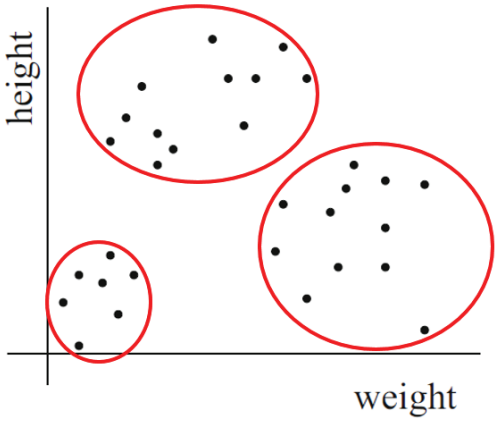


Figura 4: Ejemplo de detección de clústeres [10]

Durante el trascurso del algoritmo, los *clústeres* deben designarse según su ubicación en el espacio vectorial, su tamaño o su distribución. Para esta identificación, puede recurrirse a los centroides. El centroide de un *clúster* indica la posición obtenida al promediar los valores para cada coordenada o atributo de las muestras pertenecientes al *clúster*. De esta manera, el criterio de asignación de *clúster* más habitual se basa en calcular, para cada agrupación existente, la distancia vectorial que separa la muestra a clasificar del centroide. Consecuentemente, la muestra se asocia al *clúster* cuyo centroide se halle más próximo y, por tanto, similar [10], [14].

Una misma muestra no debe pertenecer a varios *clústeres*. No obstante, las muestras pueden reclasificarse durante la ejecución del algoritmo. La cantidad de *clústeres* a considerar es variable. Las restricciones en esta cantidad se corresponden, en el límite inferior, a un solo *clúster* que agruparía la totalidad de las muestras evaluadas. El límite superior se corresponde a tantos *clústeres* como muestras. En este último caso, cada *clúster* abarcaría una sola muestra. La cantidad de *clústeres* puede ser preestablecida por el usuario, o ser estimada por el algoritmo. Los algoritmos de *clustering* resultan de gran utilidad, y en muchos casos, estos algoritmos se utilizan para complementar aplicaciones de *machine learning* de aprendizaje supervisado [10], [14].

"k-means" está considerado como un algoritmo de *clustering*. Dicho algoritmo comienza estableciendo una cantidad fija de "k" agrupaciones, las cuales contienen inicialmente una misma cantidad de muestras. La asignación inicial de muestras a cada *clúster* puede ser aleatoria, introduciéndose una muestra diferente en cada *clúster*. No obstante, si se conocen criterios básicos sobre el resultado esperable, las asignaciones iniciales pueden efectuarse en función de dichos criterios. Esta última alternativa reduciría el tiempo de ejecución que el algoritmo requerirá. Tras establecer los *clústeres* iniciales, se calculan sus respectivos centroides [10], [14].

Posteriormente, el algoritmo evalúa las distancias dentro del espacio vectorial entre una de las muestras y los *clústeres*, representados estos últimos mediante sus centroides. Si el *clúster* calculado como más cercano difiere del asignado a la muestra, esta se reasigna al mencionado primer *clúster*. Dicho cambio modifica el contenido de los *clústeres* implicados, teniendo que recalcularse los centroides de estos. Este proceso se reitera para cada muestra varias veces hasta lograr que estas se encuentren en el *clúster* más próximo [10], [14].

Uno de los principales inconvenientes del algoritmo "k-means", consiste en la influencia que posee la inicialización de los centroides con respecto a las agrupaciones finales resultantes. Este hecho puede exponerse tomando como referencia un conjunto de muestras que siguen la distribución bidimensional indicada en la Figura 5 [10], [14].

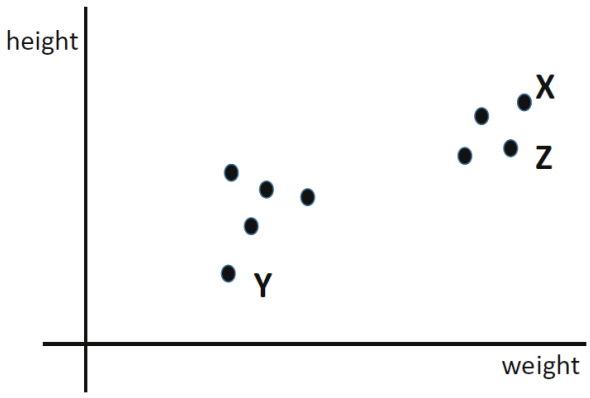


Figura 5: Distribución de muestras en un espacio bidimensional [10]

A partir de la distribución presentada en la Figura 5, supóngase que el usuario ordena la subdivisión de las muestras en 2 *clústeres*. Si el algoritmo elije las muestras "X" e "Y" para inicializar estos *clústeres*, la ejecución converge correctamente hacia las

agrupaciones evidentes. Sin embargo, si se escogiesen las muestras "X" y "Z" para dicha inicialización, puede suceder que las agrupaciones resultantes no fueran las deseables, incluso aunque se cumpliesen los criterios de terminación del algoritmo. Esto ocurre ya que las agrupaciones deseables están muy dispersas entre sí. Además, las agrupaciones deseables contendrán varias muestras que realmente serán más próximas al centroide del *clúster* opuesto [10], [14].

Para solventar esta circunstancia debe ponderarse, en primer lugar, la adecuación del agrupamiento obtenido. Para este propósito, se crea la variable Distancias Sumadas o SD. El valor de esta variable constituye el sumatorio de las distancias entre cada muestra y el centroide del *clúster* asignado a ella. Un menor valor de SD representa una mayor idoneidad de las agrupaciones obtenidas. En consecuencia, el algoritmo probará distintas alternativas de inicialización de los *clústeres*, eligiéndose las agrupaciones resultantes que ofrezcan un menor SD. Esta opción solo es útil si en las alternativas probadas se mantiene constante la cantidad de *clústeres* a obtener [10], [14].

Al ser la cantidad de *clústeres* un parámetro preestablecido por el programador resulta muy probable que el valor elegido no sea el más conveniente teniendo en cuenta las semejanzas relativas entre las muestras. Para determinar la opción más idónea, lo más sencillo consiste en probar el algoritmo para distintas cantidades de *clústeres*, escogiendo aquella que otorgue mejores resultados. La idoneidad de cada opción puede cuantificarse calculando el SD de cada opción y dividiéndolo para la cantidad *clústeres* establecida. La cantidad de *clústeres* óptima será aquella que resulta en un cociente de menor valor. La división resulta necesaria para contrarrestar el efecto de una mayor cantidad de *clústeres* en la reducción del valor de SD obtenido [10], [14].

Otra manera de determinar la cantidad de *clústeres* adecuada consiste en partir de un valor prefijado y aumentar o disminuir dicho valor por medio de subdividir o unificar respectivamente los *clústeres* obtenidos. Para valorar si una agregación de grupos aproxima el algoritmo a una mejor clasificación y, por lo tanto, debe efectuarse, ha de medirse la distancia entre los dos *clústeres* implicados. Asimismo, si esta distancia resulta pequeña en comparación con la distancia media entre los *clústeres*, la agregación debe realizarse. Con respecto a la separación, esta debe hacerse en aquellos *clústeres* que

presenten una distancia promedia entre sus muestras relativamente grande. En caso de efectuarse, se desasignarán todas las muestras pertenecientes al *clúster* afectado para su posterior subdivisión en dos nuevos *clústeres*. Para que esta reorganización sea efectiva, suele ser suficiente con inicializar los nuevos *clústeres* con las muestras más distantes entre sí, reubicando las demás muestras afectadas mediante el algoritmo de "k-means" estándar [10], [14].

Otra variante interesante del algoritmo "k-means" consiste en la subdivisión jerárquica. En esta versión, el algoritmo "k-means" clasifica las muestras inicialmente en dos *clústeres*. Posteriormente, se repite el algoritmo para cada *clúster* por separado, obteniéndose cuatro *clústeres* en total. Estas particiones en dos de los *clústeres* se reiteran hasta alcanzar la cantidad de *clústeres* máxima que indique el programador, o hasta alcanzar la distancia mínima de separación entre *clústeres* cercanos [10], [14].

El algoritmo "k-means" es aplicable con éxito a la inmensa mayoría de necesidades de *clustering*. Excepcionalmente, la capacidad de distinguir agrupaciones por parte de este algoritmo pierde efectividad en aquellos casos minoritarios donde existen agrupaciones que siguen una distribución cuya morfología presenta concavidades. Una distribución de estas características se muestra en la Figura 6 [10], [14].

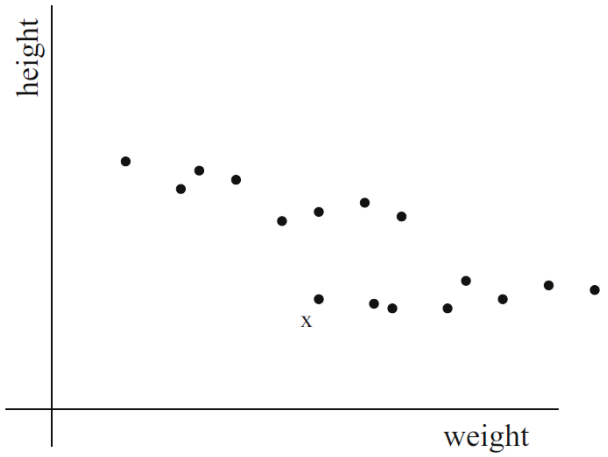


Figura 6: Ejemplo de distribución de muestras donde "k-means" no sería eficaz [10]

En la Figura 6, la subdivisión de las muestras en *clústeres* más evidente consistiría en dos *clústeres* que presentan una morfología alargada con respecto al eje horizontal. No obstante, estas agrupaciones no resultan tan evidentes para el algoritmo. Consecuentemente, es muy probable que la muestra identificada como "x" se asigne

erróneamente al *clúster* incorrecto. Dicha situación se debe a que, si se considera como posición del *clúster* la de su centroide, el *clúster* más próximo a la muestra según los cálculos no resultaría ser el visualmente evidente. Por ello, una distribución de las muestras como la de la Figura 6 requeriría redefinir la distancia entre dos *clústeres* como la mínima existente entre dos muestras asignadas a *clústeres* opuestos. Sin embargo, esta modificación aumentaría la carga computacional que precisa el algoritmo. Esto sucede ya que se tendrían que calcular "N_A x N_B" distancias entre muestras para determinar la separación entre dos *clústeres*, siendo "N_i" la cantidad de muestras agrupadas en cada *clúster* [10], [14].

Otra alternativa para un caso de *clustering* como el requerido para el conjunto de muestras de la Figura 6 se basaría en la agregación jerárquica. La ejecución de esta variante se inicia considerando la misma cantidad de *clústeres* que muestras, siendo la asignación unívoca. Durante el trascurso del algoritmo, se localiza el par de agrupaciones que se ubican más próximos entre sí para su unificación. Esta última operación se repite hasta alcanzar la condición de terminación. Dicha condición se cumple cuando la cantidad de *clústeres* se reduce por debajo de un umbral mínimo específico, o cuando la distancia mínima encontrada entre *clústeres* supere un determinado valor [10], [14].

2.5. Conclusiones

Con este capítulo, se ha descrito las principales características de los algoritmos de *machine learning*, además de la subdivisión de las categorías de los algoritmos, ya sean de aprendizaje supervisado, aprendizaje no supervisado o aprendizaje mediante refuerzo. Posteriormente, se describen varios aspectos de las muestras de entrada que deben considerarse para el correcto y eficaz funcionamiento de estos algoritmos. Por último, se han descrito brevemente algunos algoritmos de *machine learning* existentes, exponiendo sus funcionalidades, así como sus restricciones acerca de su utilidad. Dentro de estos algoritmos se encuentran los algoritmos de *clustering*. Esta última categoría se corresponde a la funcionalidad de *machine learning* implementada en la aplicación objeto de este trabajo. Las características más concretas acerca de esta implementación se detallan en próximos capítulos.

Capítulo 3. Imágenes hiperespectrales

3.1. Introducción

Las imágenes hiperespectrales aúnan la información espacial, propia de una imagen convencional, y la información espectral obtenida, por ejemplo, mediante espectroscopio. De esta manera, se obtiene una imagen en la que cada píxel se representa a partir de los niveles de intensidad de decenas de bandas espectrales, lo que conforma la firma espectral del píxel. Esta información permite caracterizar los coeficientes de absorción o reflexión en cada banda espectral evaluada, los cuales divergen en función del material fotografiado en ese píxel. Debido a ello, las imágenes hiperespectrales suponen un recurso útil para el análisis no invasivo, ya que permiten detectar la presencia y distribución de diversos materiales y compuestos en la imagen captada [1].

El término "hiperespectral" es ampliamente utilizado para referirse a imágenes con múltiples bandas espectrales más allá de las componentes RGB de una imagen a color. Sin embargo, en la literatura también se encuentran otros términos denominados "multiespectral" y "ultraespectral". La diferencia entre estos conceptos radica en la cantidad de bandas espectrales que están implicadas y el grado de selectividad que, consecuentemente, deben presentar estas bandas para resultar de utilidad. Asimismo, el término "multiespectral" designa habitualmente en la literatura a aplicaciones que involucran entre 4 y 9 bandas espectrales. Por otra parte, la utilización del término "hiperespectral" suele centrarse en aplicaciones que gestionan entre 10 y 100 bandas espectrales. Por último, el concepto "ultraespectral" suele reservarse para aplicaciones que requieren más de 100 bandas espectrales [4], [18].

Independientemente de la terminología, la cantidad de bandas espectrales a utilizar en una aplicación debe escogerse considerando las necesidades del sistema que dicha decisión compromete. Asimismo, una mayor cantidad de bandas espectrales confiere una mayor capacidad para discernir los elementos materiales presentes en la imagen, pudiendo ser necesaria si dichos materiales presentan firmas espectrales similares. Sin embargo, una mayor cantidad de estas bandas suele conllevar mayor complejidad y carga computacional. Además, también implica recurrir a cámaras más sofisticadas y por ello costosas, que logran una mejor resolución espectral de la imagen en detrimento de su resolución espacial o de la rapidez de su captura [4], [18].

En una primera aproximación, puede considerarse que las aplicaciones de imágenes hiperespectrales se adecúan a cualquier rango del espectro electromagnético que resulte de interés. No obstante, la mayor parte de estas aplicaciones se restringe al espectro visible e infrarrojo cercano o VNIR (*Visible and Near Infrared*). En menor medida, este rango espectral se amplía en el espectro infrarrojo, abarcando el infrarrojo de onda media y, en algunas situaciones más específicas, el infrarrojo de onda larga. En otras ocasiones excepcionales, también se amplía el rango espectral considerado con la región del espectro ultravioleta próximo al visible [4], [18], [19].

Entre los principales motivos de esta convergencia respecto a los rangos espectrales utilizados se encuentra la disponibilidad de los sensores, los cuales presentan un mejor grado de desarrollo y asequibilidad en estas regiones espectrales y muy especialmente en el VNIR. Además, como se observa en la Figura 7, los rangos espectrales predilectos para imágenes hiperespectrales coinciden con las componentes espectrales más abundantes que presenta la luz solar que alcanza la superficie terrestre. Debido a ello, la limitación a estos rangos espectrales resulta prácticamente obligatoria en aplicaciones a campo abierto [4], [18].

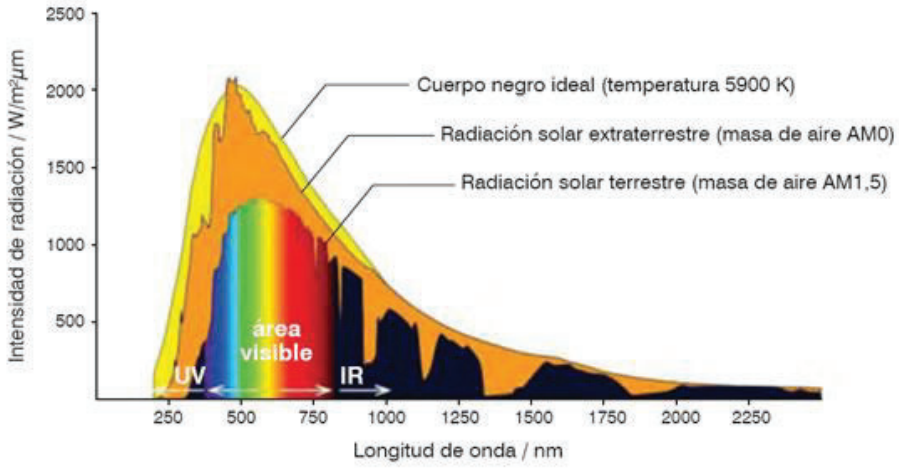


Figura 7. Composición espectral de la radiación solar en la tierra [7], [20]

3.2. Factores a considerar para el preprocesamiento

Para utilizar la adquisición de imágenes hiperespectrales como herramienta analítica, es necesario considerar una serie de aspectos específicos de este campo de uso que afectan en los valores medidos por las cámaras hiperespectrales pero que no son relativos a la composición del material fotografiado. Estos aspectos son los siguientes [7], [19].

a. **Sensibilidad no uniforme del instrumento de captación.** La captura de las imágenes se debe fundamentalmente a la sensibilidad del material de captación al efecto fotoeléctrico. Sin embargo, es frecuente que la intensidad de la respuesta eléctrica del material varíe en función de la frecuencia de la señal electromagnética irradiada sobre él, aún sin cambiar la intensidad de esta última. En la Figura 8 se muestra la sensibilidad típica de un sensor CCD (*Charge-Coupled Device*), el tipo de sensor más ampliamente utilizado en cámaras hiperespectrales [7], [19].

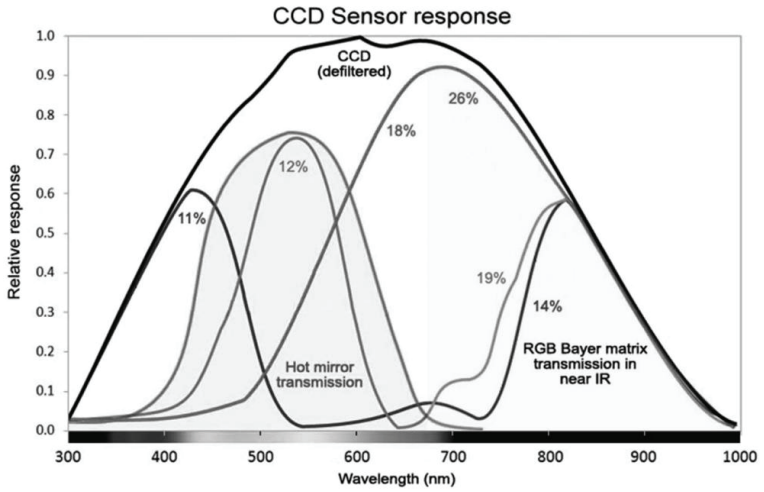


Figura 8. Sensibilidad del sensor CCD estándar a la luz según la longitud de onda [21]

Como puede observarse, la sensibilidad del sensor carece de uniformidad, y se encuentra acotada para cada longitud de onda. Para eliminar la influencia de esta desigual respuesta sobre la imagen captada se recurre a la calibración en una etapa del preprocesado de la imagen. También cabe destacar que el sensor CCD ofrece una sensibilidad deficiente en las frecuencias extremas del rango espectral que es capaz de evaluar. Este hecho suele ocurrir con el resto de los sensores utilizados en aplicaciones de imágenes hiperespectrales. En consecuencia, la imagen hiperespectral extraída presenta los mayores errores de cuantificación en las frecuencias extremas incluso después del preprocesado. Por ello, se descarta una cierta cantidad de frecuencias extremas de la imagen, evitando así comprometer la fiabilidad promedia de la información utilizada de la imagen hiperespectral y, por ende, de las conclusiones analíticas obtenidas [7], [19].

b. ***Offset.*** El elemento de captación puede devolver, debido a su polarización y ruido eléctrico, un valor distinto de cero para la señal eléctrica en ausencia absoluta de iluminación. Para evitar que estas interferencias falseen los resultados medidos de intensidad en las frecuencias espectrales evaluadas, se captura una imagen con el sensor completamente a oscuras. A dicha captura se la denomina referencia de negro, y es restada a las imágenes hiperespectrales captadas posteriormente para anular la influencia de la polarización en los valores de señal eléctrica obtenidos. Por otra parte, para minimizar la influencia del ruido eléctrico debe recurrirse a

operaciones de filtrado de los datos de la imagen, debido principalmente a la naturaleza aleatoria de este [7], [19].

c. **Iluminación no uniforme.** Para que la imagen captada por la cámara resulte de utilidad analítica, es necesario conocer la intensidad de la luz que incide sobre la región fotografiada en cada una de las bandas espectrales evaluadas. Con dicha información puede determinarse los coeficientes de absorción y reflexión de cada píxel y en cada banda espectral de la imagen y, por consiguiente, distinguir los materiales observados en la imagen siempre que se hayan escogido bandas espectrales representativas para su diferenciación. Sin embargo, la fuente de luz rara vez irradia con una intensidad uniforme para las bandas espectrales consideradas. Esta circunstancia sucede tanto si se utiliza una fuente de luz natural, típicamente la luz solar, o una artificial. A modo de ejemplo, en la Figura 9.(a) y la Figura 9.(b) se exponen la composición espectral de 2 lámparas desarrolladas por la empresa Effilux expresamente para la aplicación de imágenes hiperespectrales [7], [19].

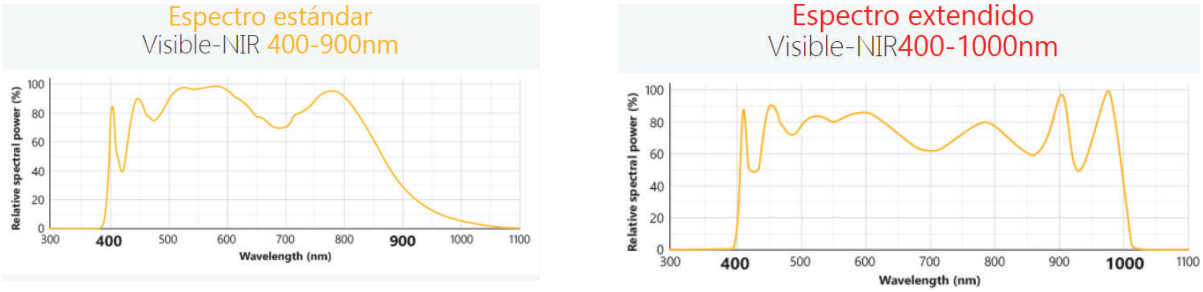


Figura 9. Espectro de la lámpara Effilux: (a) espectro estándar, (b) espectro extendido [22]

d. **Absorción del medio de propagación.** Debe tenerse en cuenta que el medio de propagación de la luz en la gran mayoría de las aplicaciones es aéreo. Dicho medio introduce una atenuación no uniforme de las componentes espectrales de la luz, la cual resulta significativa cuando la distancia de separación entre la cámara y la región captada son considerables. Dicha situación ocurre con la captura de imágenes hiperespectrales al aire libre utilizando el sol como fuente lumínica. Además, la falta de uniformidad en la respuesta del medio de transmisión, en este caso la atmósfera, a las componentes espectrales de la luz se agrava como resultado

de la enorme variabilidad temporal de su efecto debida a la hora del día y a la variabilidad meteorológica [7], [19].

e. **Sombreado y *scattering*.** Relacionado con los aspectos anteriormente evaluados en los apartados c y d, si la región fotografiada no es plana puede producirse cierto sombreado en algunas partes de la imagen. Por otra parte, la luz incidente puede proceder no solo de la fuente de luz de referencia, sino también de otros elementos que reflejan parte de esa luz sobre la región captada. Dicho fenómeno se conoce como *scattering*. Estos aspectos ocasionan que la irradiación electromagnética presente variabilidad tanto en el dominio espectral como a lo largo de la extensión espacial de la imagen [7], [19].

Para poder eliminar esta variabilidad de los resultados obtenidos de la imagen se recurre a la referencia de blanco. Dicho recurso consiste en la captura de una imagen de un material que presente un coeficiente de reflexión idealmente absoluto en todas las frecuencias espectrales estudiadas. La captura de la referencia de blanco debe realizarse en condiciones lumínicas que se asemejen lo mejor posible a las de las imágenes objeto del estudio [7], [19].

f. **Saturación del sensor**. Los sensores de las cámaras hiperespectrales constituyen componentes electrónicos no lineales. Debido a ello, es necesario regular la intensidad lumínica reflejada hacia el sensor para que no se sature. Además, el sensor debe poseer un margen dinámico lo suficientemente amplio para que se identifique la variabilidad lumínica de la imagen captada, manteniendo una respuesta lineal ante dicha luminosidad [23], [24]. Por otra parte, es necesario limitar el tiempo de exposición del sensor a la captura de la imagen ya que si no se produce una acumulación excesiva de cargas eléctricas que conduce igualmente a la saturación [24].

3.3. Organización de los datos

La información contenida en una imagen hiperespectral presenta una distribución tridimensional, conformando un hipercubo. Asimismo, dos de sus dimensiones constituyen el ancho y el alto, es decir, la extensión espacial de la imagen. Por otra parte, la tercera dimensión hace referencia al espectro de la imagen, que equivale al rango de frecuencias

electromagnéticas captadas en cada píxel de la imagen. Atendiendo a estas tres dimensiones, los datos de la imagen hiperespectral pueden serializarse y almacenarse en tres formatos básicos distintos según el orden en el que se recorran sus dimensiones. En muchas ocasiones, estos formatos no presentan el mismo grado de idoneidad en función del tipo de cámara utilizada. Esto se debe a que la mayoría de las cámaras hiperespectrales no efectúan una captación simultánea en las tres dimensiones de la información de la imagen. Por este motivo, algunos formatos de almacenamiento y serializado se asemejan mejor al orden en que la información de la imagen es captada [19].

a. **Formato BIP (*Bands Interleaved by Pixel*).** Los datos de la imagen hiperespectral se ordenan recorriendo primero la dimensión espectral, y luego las dimensiones espaciales. En consecuencia, se indica la firma espectral del píxel al completo antes de proseguir con el siguiente píxel. Este tipo de formato se corresponde con el método de captación de las cámaras de escaneo punto a punto o *whiskbroom*. En dichas cámaras, el dispositivo solo es capaz de capturar la firma espectral de un solo pixel, por lo que la imagen hiperespectral se conforma con varias capturas sucesivas de la cámara mientras esta efectúa el barrido del área de la imagen a captar [19].

b. **Formato BIL (*Bands Interleaved by Line*).** La imagen hiperespectral se serializa empezando con una de las dimensiones espaciales, continuándose con la dimensión espectral y, por último, con la otra dimensión espacial. Este formato se adecúa al orden de captación de la información de la imagen hiperespectral en las cámaras de escaneo línea a línea o *pushbroom*. Dichas cámaras capturan de manera simultánea la información espectral de una línea de la imagen. Asimismo, la imagen al completo se construye barriendo una de las dimensiones espaciales de la imagen mediante capturas sucesivas [19].

c. **Formato BSQ (*Band Sequential*).** Los datos de la imagen hiperespectral se organizan recorriendo primero las dimensiones espaciales de la imagen y, luego, la dimensión espectral. Este formato está orientado a cámaras que en una sola captura pueden abarcar la totalidad del área de la imagen, pero siendo incapaces de discernir en el momento de la captura varias componentes espectrales. En este tipo de cámaras,

cada componente espectral suele separarse de las demás por medio de un filtro antepuesto a la superficie de captación de la cámara, el cual es traspasado por solo una de las componentes espectrales que se desea captar. Estos filtros pueden estar diseñados para una única componente espectral o ser ajustables a cualquiera de las componentes espectrales objeto del análisis. En la primera opción, la imagen se conforma alternando los filtros en las sucesivas capturas, mientras que en la otra se utiliza un mismo filtro regulado electrónicamente [19].

Para mostrar gráficamente cómo se organizan los datos de la imagen hiperespectral según el formato, en la Figura 10 se representa la utilización de estos formatos en el caso de 2 imágenes tricromáticas, siendo una de 3x3 píxeles y la otra de un dimensionado genérico. Asimismo, el criterio de ordenación de los datos en cada formato para estos ejemplos es extrapolable a cualquier dimensionamiento espacial y espectral de la imagen.

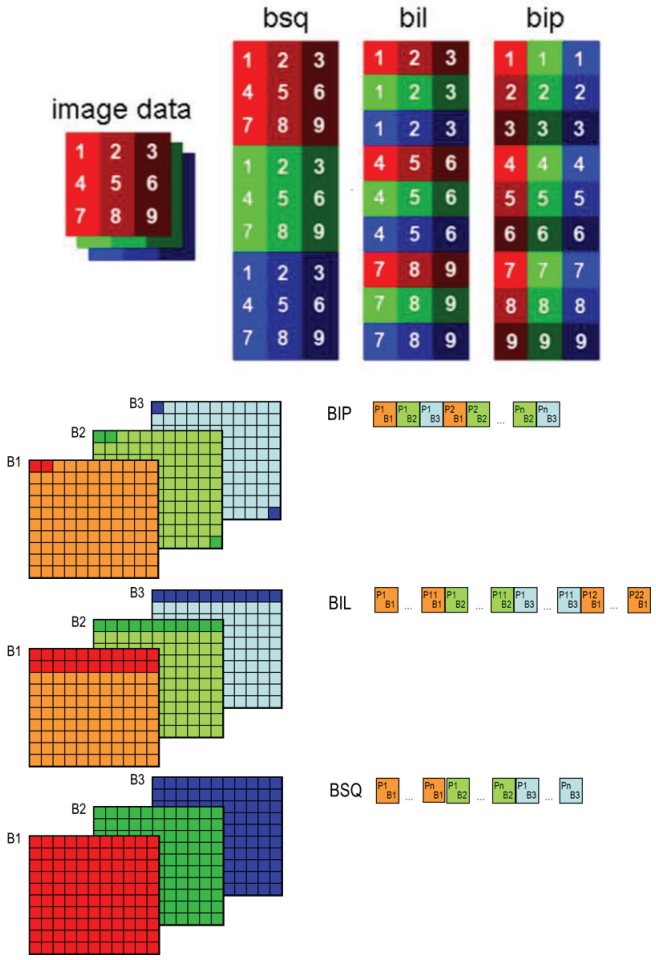


Figura 10. Formatos básicos de ordenación de los datos de una imagen hiperespectral [25]

3.4. Identificación de muestras

Como se mencionó con anterioridad, el propósito de las imágenes hiperespectrales consiste en posibilitar la identificación de las muestras fotografiadas, así como su distribución. Dicho proceso requiere conocer las firmas espectrales de referencia de las posibles muestras a identificar. Dicha información suele extraerse de bases de datos oficiales, donde se detallan las firmas espectrales genéricas de la materia individuales, recurriendo a la espectroscopia o análisis espectral intensivo para ello. De esta manera, la identificación de los elementos presentes en la imagen objetivo se logra mediante una evaluación comparativa. Para ello, puede recurrirse a los siguientes procedimientos matemáticos [7], [26].

a. ***Spectral Angle Mapper* (SAM).** Consiste en determinar el grado de semejanza entre dos firmas espectrales mediante una medida angular. Para su cálculo, se considera que las firmas espectrales son vectores ubicados en un espacio vectorial multidimensional donde cada dimensión se corresponde a una componente espectral cuantificada en ambas firmas espectrales. Para su cálculo se aplica la ecuación (5), donde $\vec{u}$ y $\vec{v}$ representan las firmas espectrales a comparar expresadas como vectores [7], [26].

$$SAM = acos\left(\frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}\right) \tag{5}$$

Desde el punto de vista gráfico, el valor del SAM devuelto por la ecuación (5) depende únicamente del ángulo formado entre los vectores donde $\vec{u}$ y $\vec{v}$, considerando como referencia el punto de origen del espacio vectorial. Es especialmente reseñable que el módulo de los vectores no influye en el valor de SAM obtenido. Esta cualidad resulta provechosa en la comparación de firmas espectrales, ya que desde el punto de vista físico el módulo solo cuantifica el nivel de brillo del píxel cuya firma espectral se evalúa, lo cual no resulta útil para identificar el material fotografiado en el píxel. Sin embargo, dicho material se detecta por medio de las ratios de absorción o reflexión que suceden en cada componente espectral del píxel. Por este motivo, si estas ratios se cuantifican

uniformemente, el material captado en el píxel se infiere por medio de las razones de proporción mantenidas entre ellas. Dichas proporcionalidades afectan a la dirección de los vectores que representan las firmas espectrales, pero no a su módulo [7], [26].

Cabe destacar que, dado que las componentes espectrales solo pueden adquirir valores iguales o superiores a 0, el resultado de la ecuación (5) se encuentra acotado entre 0° y 90°. En el caso de 90°, este valor indica que los vectores son perpendiculares entre sí, siendo absoluta la divergencia entre las firmas espectrales que representan. Por otra parte, el valor extremo inferior 0° especifica que los vectores forman un ángulo nulo, convergiendo en dirección y sentido dentro del espacio vectorial. Este último caso implicaría que las firmas espectrales comparadas equivalen a una misma composición material [7], [26].

b. ***Spectral Feature Fitting*** **(SFF).** Consiste en comparar la firma espectral del píxel con las firmas espectrales de los materiales de referencia. Para ello, se utiliza la porción de la firma espectral de referencia que resulte más distintiva, procurando anular completamente el resto del espectro del píxel y su influencia en el proceso computacional. Para calcular el SFF, se adapta la firma espectral del píxel y la referencia a una misma escala. Posteriormente, el grado de semejanza se determina aplicando la suma de mínimos cuadrados [7], [26].

Dadas las características de funcionamiento de estos procedimientos, el SAM y el SFF tienen distintos campos de aplicación. Por un lado, el SAM resulta más propicio para la identificación en la imagen de materiales pertenecientes a categorías genéricas donde, incluso la firma espectral de referencia del material, no se considera exenta de cierta variabilidad. Como ejemplo de una categoría genérica se encuentra la vegetación, pues varios píxeles pueden captar únicamente esta categoría y divergir en su firma espectral en función del espécimen y la región del mismo que captaron [7], [26].

Por otra parte, el SFF se adecúa mejor a aquellas categorías donde puede concretarse una firma espectral unívoca. Un ejemplo de este tipo de categorías serían los minerales. Este último hecho se debe a que, si se analizan varias muestras de un mismo

tipo de mineral puro, se observa que la firma espectral no varía al cambiar de muestra [7], [26].

3.5. Conclusiones

En este capítulo se han explicado los aspectos más relevantes que deben considerarse para implementar el análisis con imágenes hiperespectrales de manera efectiva en una aplicación. Entre dichos aspectos se incluyen aquellos que afectan a la captura de la imagen, pero no son relativos a la respuesta espectral de los materiales. También se detallaron los formatos de organización básicos de los datos de una imagen hiperespectral, así como la conveniencia de cada uno. Finalmente, se detallan dos procedimientos para identificar las firmas espectrales obtenidas por cada píxel con el material al que representa. Cabe destacar que gran parte de estos conceptos serán utilizados en el diseño de la aplicación objetivo de este proyecto.

Capítulo 4. Sistema de referencia

4.1. Introducción

En este capítulo, se explica la aplicación para el análisis de imágenes hiperespectrales desarrollada en [2], y que ha sido tomado como referencia para el desarrollo de esta aplicación. Dicha aplicación está implementada utilizando aceleración *hardware* basada en GPUs y se subdivide en una serie de etapas. Las etapas de interés a considerar son la cadena de preprocesado y la segmentación automática, las cuales se detallan a continuación.

4.2. Cadena de preprocesado

El objetivo de esta etapa consiste en utilizar los datos de la imagen hiperespectral captados por una cámara y efectuar sobre ellos una serie de operaciones matemáticas que los transformen. Asimismo, el objetivo de estas transformaciones consiste en preparar los datos para que estos sean útiles en etapas posteriores que, a partir de dichos datos, detecten y analicen aspectos presentes en la región captada. Esta etapa de preprocesado se subdivide en las siguientes subetapas [2].

Calibrado. Consiste en eliminar la influencia del *offset*, de la iluminación no uniforme y de la sensibilidad de la cámara no uniforme (ver 3.2.) en la captación y medida de la firma espectral de los elementos fotografiados en la imagen. Para ello, se utiliza una referencia de blanco, consistente en una pieza altamente reflectante que refleja el 99% de la luz incidente, y una referencia de negro obtenida tras realizar una captura con la cámara hiperespectral a utilizar a oscuras y con la lente tapada [2].

Eliminación de bandas espectrales extremas. Esta eliminación se debe a que las cámaras hiperespectrales presentan una mala sensibilidad en los límites de su espectro de captación. En consecuencia, la información de estas bandas espectrales es poco fiable para su utilización con fines analíticos. Por este motivo, es conveniente no tener en consideración un cierto margen de bandas extremas que, para esta aplicación, se corresponderán a las 4 primeras y 5 últimas bandas del espectro captado en la imagen [2].

Filtrado. Se efectúa una operación de filtrado por píxel con el propósito de minimizar la influencia del ruido espectral en los datos de la imagen hiperespectral a utilizar. Este filtrado consiste en que el dato a filtrar se promedia utilizando una cantidad simétrica tanto de datos anteriores y posteriores dentro del espectro del píxel al que pertenece el dato a filtrar. Esta cantidad permanecerá fija durante todo el recorrido por los datos de cada píxel, siendo las únicas excepciones a esta afirmación los datos que se ubican próximos al límite superior y/o inferior del espectro del píxel. En estos últimos, dado que el dato no cuenta con suficientes datos anteriores y/o posteriores para el promediado habitual, se reducirá la cantidad de datos a promediar en el filtrado a la máxima cantidad posible que garantice una utilización simétrica de datos anteriores y posteriores al dato a filtrar. Para esta aplicación, la cantidad de datos implicados en el promediado de manera regular serán 5 [2].

Normalizado. El objetivo de esta etapa consiste en adaptar los datos de la imagen hiperespectral a una escala entre 0 y 1. Con ello se pretende homogenizar la amplitud de las firmas espectrales de los píxeles, mejorando con ello la utilidad analítica de los datos [2].

4.3. Segmentación automática (o *clustering*)

En esta siguiente etapa se subdividen los píxeles preprocesados en la etapa anterior en píxeles que representen tejido cutáneo sano y píxeles que representen tejido afectado de categoría PSL (*Pigmented Skin Lession*). Para ello se aplica el algoritmo “k-means”. Como se ha explicado en el apartado 2.4. , el propósito de este algoritmo consiste en subdividir un conjunto de muestras en función de la semejanza relativa de sus atributos en una serie de agrupaciones o *clústeres*. En esta aplicación, las muestras se corresponderán a los

píxeles, los atributos a las bandas espectrales que componen la firma espectral de cada píxel. Asimismo, la subdivisión de las muestras se establece en 3 *clústeres.* Estos *clústeres* se inicializan con píxeles escogidos aleatoriamente [2].

Una vez terminado el *clustering*, se procede a determinar los centroides en función de si identifican piel sana o piel afectada. Para ello, se aplicará el algoritmo SAM para medir la semejanza relativa entre estos centroides y una serie de firmas espectrales de referencia. Estas últimas firmas espectrales se encuentran preestablecidas y asociadas a las categorías objeto de la clasificación de los píxeles. De esta manera, la categorización de cada centroide se corresponderá a la de la firma espectral de referencia más similar a este, lo cual analizado desde la perspectiva aritmética se corresponderá a aquella firma espectral para la que el resultado del SAM se minimice [2].

4.4. Conclusiones

En este capítulo se ha estudiado el algoritmo seguido por las etapas de la aplicación de referencia. Las etapas consideradas para el desarrollo de este trabajo son las etapas de preprocesamiento y la de segmentación o *clustering* automático.

Asimismo, la etapa de preprocesamiento se subdivide en varias subetapas. La primera de ellas se corresponde al calibrado, para eliminar la influencia del *offset*, la iluminación no uniforme y la sensibilidad no uniforme de la cámara. La siguiente a la eliminación de bandas espectrales extremas, debido a la poca fiabilidad de la cámara en dichas regiones del espectro. Posteriormente, se encuentra el filtrado para la eliminación de ruido y el normalizado para homogenizar la amplitud de las firmas espectrales de los píxeles.

En la aplicación de referencia se utilizan aceleradores basados en GPU para dar soporte a la ejecución. En este caso, por el contrario, se utiliza un dispositivo MPSoC FPGA para acelerar el algoritmo “k-means” usando la placa de prototipado ZCU102 [2].

Capítulo 5. Zynq Ultrascale+ MPSoC

5.1. Introducción

Para la implementación del sistema se ha seleccionado el dispositivo MPSoC Zynq UltraScale+. Se trata de un MPSoC heterogéneo constituido por diferentes módulos diferenciados y optimizados para determinadas funciones donde es posible combinar la funcionalidad *hardware* con la funcionalidad *software*. En lo referente a la funcionalidad *software* se emplean distintas unidades de procesamiento englobadas dentro del Sistema de Procesamiento (PS, *Processing System*) del MPSoC. Además, se encuentran los recursos de lógica programable (PL, *Programmable Logic*) [14], [27], encargados de la funcionalidad *hardware*.

Zynq UltraScale+ constituye un conjunto de dispositivos MPSoC FPGA similares diseñados por Xilinx. La familia de dispositivos Zynq UltraScale+ permite dotar de multiprocesamiento a las funcionalidades *software*. La fabricación del dispositivo se ajusta a un proceso con transistores CMOS FinFET de 16 nm de TSMC (*Taiwan Semiconductor Manufacturing Company*), logrando la capacidad de integración en un *chip* necesaria para un MPSoC. Además, esta tecnología permite una mayor velocidad, además de un menor consumo de potencia en comparación con la fabricación usada anteriormente (FinFET de 20 nm): aumento del 50 % en la velocidad de operación del dispositivo y reducción del 60 % del consumo de potencia [14], [28], [29], [30].

Los módulos *hardware* que agrupan los dispositivos Zynq UltraScale+ presentan distintas variaciones según la subfamilia a la que estos dispositivos pertenezcan. Dichas subfamilias se distinguen entre sí como EG, CG y EV. Los cambios más significativos entre

las subfamilias respecto a los recursos *hardware* que incorporan se muestran en la Tabla 1 [14], [28].

Tabla 1: Componentes hardware en función de la subfamilia (adaptada de [28])

Funcionalidad	Descripción	Sub-Familia		
		CG	EG	EV
Application Processor Unit (APU)	Basado en la arquitectura ARM Cortex-A53, con soporte de SO y ejecución de aplicaciones (2 a 4 *cores*)	2 *cores*	4 *cores*	4 *cores*
Real-Time Processing Unit (RPU)	ARM Cortex-R5 *cores* para tareas en tiempo real	2 *cores*	2 *cores*	2 *cores*
Graphics Processing Unit (GPU)	Soporte gráfico dedicado	-	✓	✓
Video Codec Unit (VCU)	Implementado como un *hard IP* en el PL, proporciona soporte para los formatos de compresión de vídeo estandarizados H.265 y H.264	-	-	✓
Configuration and Security Unit (CSU)	Funcionalidad de arranque seguro, Soporte para ARM TrustZone®, y monitorización de tensiones y temperatura	✓	✓	✓
Platform Management Unit (PMU)	Para la gestión de potencia, seguridad e integridad funcional del sistema	✓	✓	✓
Memoria	256kB OCM, interfaces para memorias externas de varios tipos	✓	✓	✓
Controladores para *Direct Memory Access* (DMA)	Dos controladores DMA de 8 canales, uno en cada dominio de potencia	✓	✓	✓
High performance interfaces (PS)	PCIe Gen2, USB3.0, SATA 3.1, DisplayPort, Gigabit Ethernet	✓	✓	✓
Bloques IPs integrados en el PL	PCI Express	✓a	✓a	✓b
	150G	-	✓a	-
	100G Ethernet MAC	-	✓	-

C: *Dual APU*, *Dual RPU*. E: *Quad APU*, *Dual RPU*, *Single GPU*. G: *General Purpose*. V: Vídeo.
a. Incluidos en un subconjunto de dispositivos. b. Incluidos en todos los dispositivos

En el caso de la placa de prototipado utilizada, ZCU102 [28], el dispositivo integrado es el Zynq UltraScale+ XCZU9EG-2FFVB1156 MPSoC. Dicho dispositivo se engloba dentro de la subfamilia EG. Entre los componentes incorporados en este MPSoC se encuentran una unidad de procesamiento o APU constituida por cuatro núcleos ARM Cortex-A43, una unidad de tiempo real o RPU (*Real-Time Processing Unit*) de dos núcleos basado en ARM Cortex-R5 y una GPU ARM Mali-400. En la Figura 11 se exponen los componentes que conforman el MPSoC y los submódulos que los integran [14], [28].

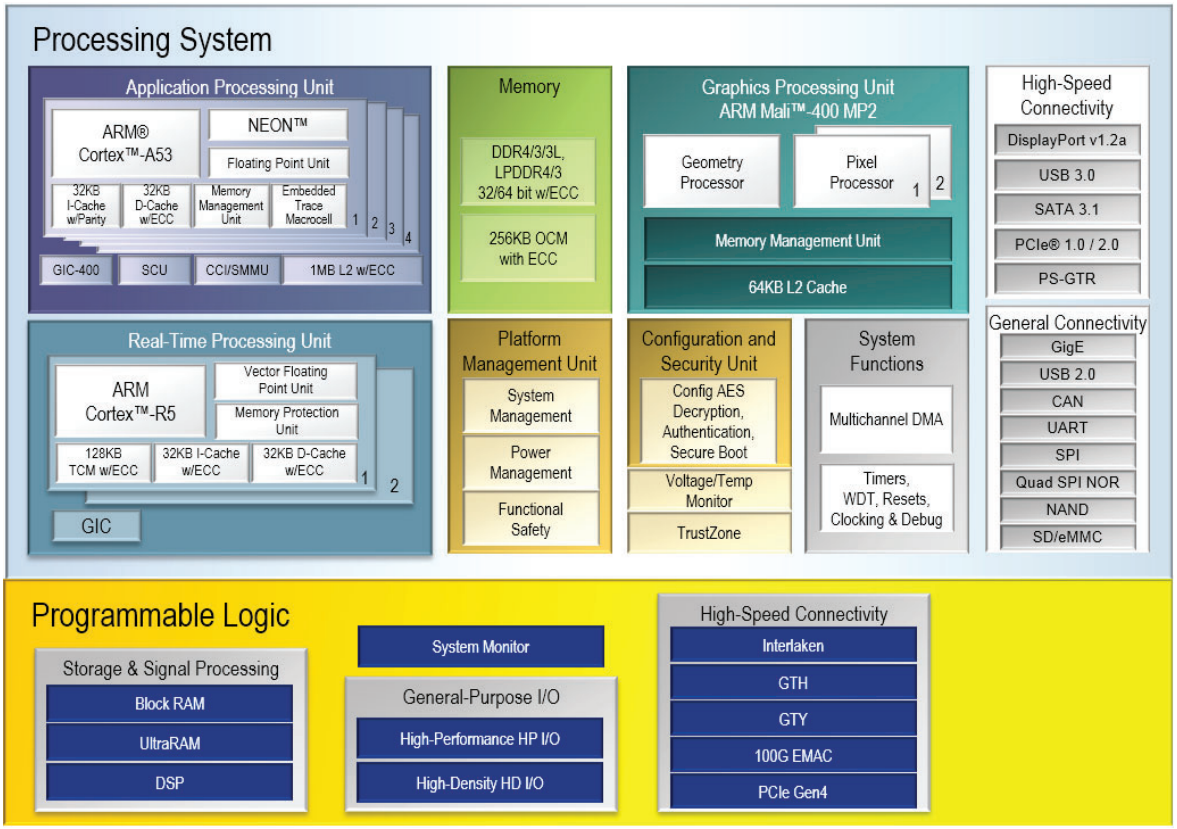


Figura 11: Módulos hardware del MPSoC a utilizar [31]

Al igual que los demás MPSoC de la familia Zynq UltraScale+, este MPSoC dispone de memoria *On-Chip* y múltiples interfaces de entrada y salida, entre otros recursos pertenecientes al PS. Respecto a la PL, esta se ajusta a la arquitectura Xilinx UltraScale [14], [32]. Los recursos *hardware* se interconectan siguiendo el diagrama de bloques expuesto en la Figura 12.

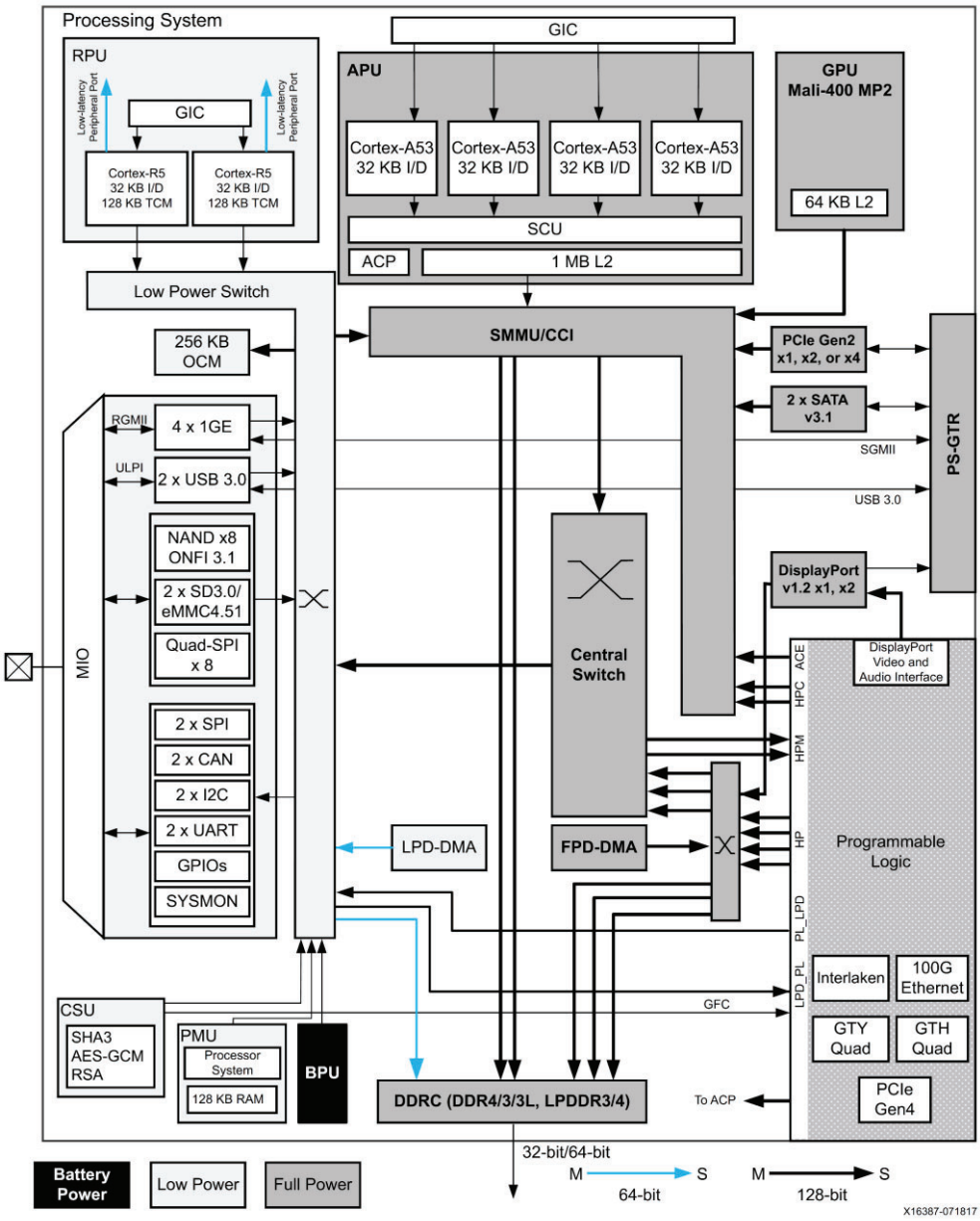


Figura 12: Diagrama de bloques de la arquitectura Zynq UltraScale+ MPSoC [32]

5.2. Sistema de procesamiento

Como se indicó con anterioridad, los elementos centrales del sistema de procesamiento o PS consisten en un conjunto de microprocesadores. Entre estos microprocesadores se encuentran los de propósito general (APU basado en ARM Cortex A53), los de tiempo real (RPU basado en ARM Cortex R5) y GPU (usando ARM Mali-400). Para este diseño se utiliza únicamente la APU. También se incluyen en el PS recursos de

memoria y distintas interfaces [14], [33]. A continuación, se describen los recursos del PS de relevancia para este trabajo.

5.2.1. APU basada en ARM Cortex-A53

Como puede verse en la Figura 13, la APU está definida por cuatro núcleos ARM Cortex A53 los cuales siguen una arquitectura ARMv8-A. Dicha arquitectura se caracteriza por poder operar en 64 y 32 *bits*, pudiendo ejecutar los conjuntos de instrucciones A64, A32 y T32 [14], [33].

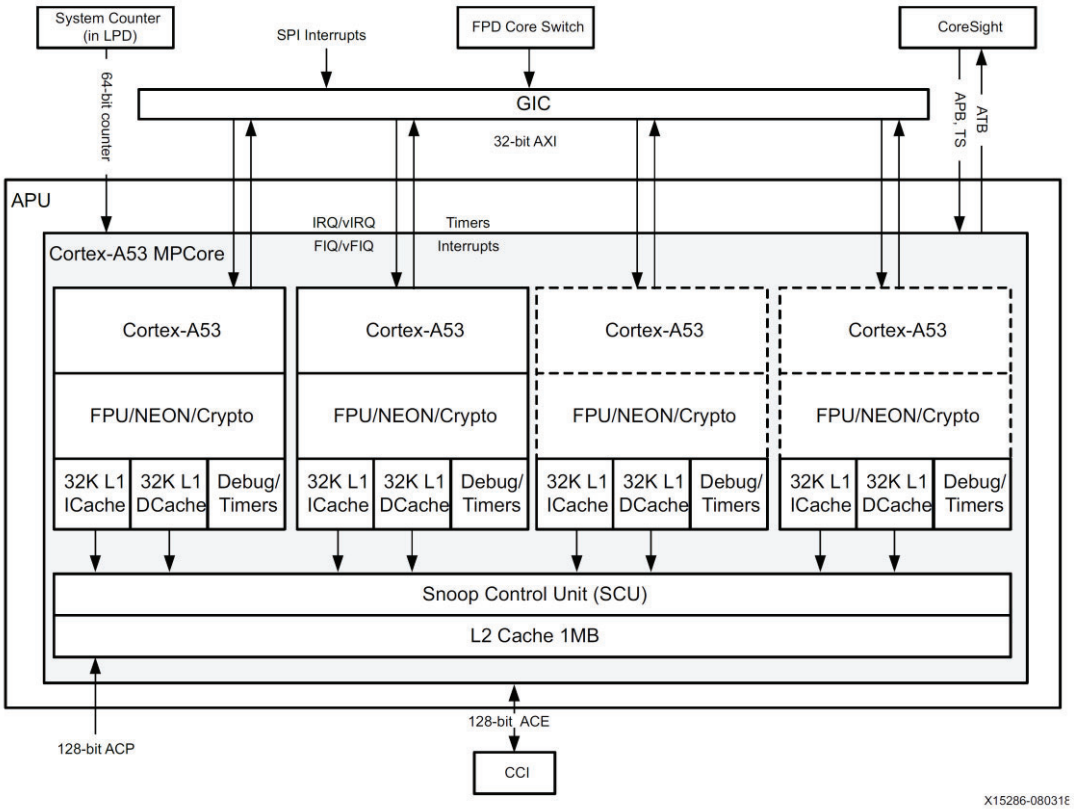


Figura 13: Diagrama de bloques de la APU [34]

Para mejorar el nivel de seguridad, la APU incorpora la funcionalidad ARM TrustZone [14], [28], [33]. Otro aspecto destacable es que los núcleos pueden funcionar con o sin sistema operativo. En este sentido, pueden establecerse diferentes configuraciones según se coordine el funcionamiento de los núcleos de la APU [14], [33]. Las posibles configuraciones serían:

a. **Configuración simétrica.** En este caso, los núcleos ejecutan un sistema operativo común, típicamente Linux. Con esta configuración, los diferentes procesos de la pueden ser ejecutados en cualquiera de los núcleos de la APU [14], [33].

b. **Configuración asimétrica supervisada.** En este caso, el funcionamiento de los núcleos es totalmente independiente, por lo que no hay interrelación entre ellos. Cada núcleo puede utilizar un sistema operativo diferente, por lo que será necesario disponer de un sistema hipervisor que pueda arbitrar la ejecución paralela de las diferentes aplicaciones en cada procesador [14], [33].

Entre sus especificaciones técnicas de funcionamiento se destacan la alimentación independiente de cada uno de los núcleos. Esto posibilita activar cada núcleo de forma individualizada, sin depender del estado de activación del resto de la APU. Por otra parte, esta APU puede operar con valores representados en punto flotante de precisión simple y doble gracias a su FPU (*Floating Point Unit*). Otro aspecto destacable es que la APU incorpora una extensión *hardware* denominada ARM NEON *Advanced*, lo que permite el modo de ejecución SIMD (*Single Instruction Multiple Data*). Por último, esta APU puede operar con una frecuencia de reloj máxima de 1,5 GHz [14], [34].

Un aspecto clave de la implementación de la aplicación de este trabajo es la interacción entre la APU y la lógica programable (PL). Para ello, se dispone de las interfaces ACP (*Accelerator Coherency Port*) y ACE (AXI *Coherency Extension*). Aparte de la transmisión y recepción de datos, estas interfaces poseen funcionalidad de encriptación. Dicha funcionalidad implementa los estándares AES (*Advanced Encryption Standard*), SHA1/256 y RSA (*Rivest Shamir and Adleman*) [14], [34].

En lo referente a la temporización, se dispone en la APU de varios recursos. Estos medios incluyen el soporte de ARM para *timers* genéricos [14], [27], dos bloques IP (*Intelectual Property*) consistentes cada uno en tres *timers* contadores [14], [35], [36], un *timer watchdog* y un *timer* global del sistema. Para la implementación de interrupciones, se incorpora el módulo GIC 400, el cual posee funcionalidad suficiente para establecer priorización entre ellas [14], [27].

Con respecto a los recursos de memoria interna de la APU, cada núcleo de esta incorpora su propia memoria caché (L1) de 64 Kbyte. Dichas memorias se asignan a partes iguales para almacenar tanto datos como instrucciones.

Por otra parte, también se dispone de una memoria caché L2 de 1 MB que es compartida por todas las CPUs de la APU y que es utilizada como recurso compartido con el resto de componentes de la plataforma para permitir la comunicación de estos con la APU [14], [28], [37]. Para la gestión de esta memoria es de especial relevancia el bloque SCU (*Snoop Control Unit*), que se encarga de actualizar el contenido de los registros de las caché L1 que representan variables compartidas entre los procesadores de la APU, mediante transacciones realizadas siguiendo el protocolo MOESI (*Modified Owned Exclusive Shared Invalid*) [14], [34]. Asimismo, cada núcleo de la APU dispone de su propia MMU (*Memory Management Unit*) que se encarga del mapeo de direcciones virtuales a direcciones físicas [14], [34].

5.2.2. Memoria PS

Aparte de los recursos de almacenamientos propios de cada módulo de procesamiento, el PS incluye las memorias OCM (*On-Chip Memory*) y el controlador DDR (*Double Data Rate*) [14], [28], [34].

La OCM constituye una memoria RAM (*Random Access Memory*) de 256 kB. Esta memoria puede utilizarse a una frecuencia de reloj máxima de 600 MHz. La interacción de esta memoria con el resto de los módulos se efectúa mediante interfaz AXI esclava de 128 *bits*. Adicionalmente, esta memoria incorpora corrección de errores ECC [14], [28], [34].

Con respecto al controlador DDR, este constituye un subsistema de control de acceso a memoria. Este recurso puede operar siguiendo los estándares DDR3, DDR3L, LPDDR3 y DDR4, con un ancho del bus de 64 o 32 *bits*, y también el estándar LPDDR4 de 32 *bits*. Además, incluye ECC en cualquiera de los estándares anteriores. Su mayor velocidad de transferencia de datos es de 2400 Mb/s, la cual se logra bajo el estándar DDR4. La mayor capacidad de almacenamiento SRAM (*Static RAM*) que el DDR puede gestionar es de 32 GB [14], [27]. Para intercomunicarse con el resto de módulos, este recurso puede recurrir a 6 interfaces AXI [14], [28], [34].

5.3. Lógica programable (PL)

La lógica programable del MPSoC engloba los recursos de la FPGA. Estos recursos consisten en un conjunto de módulos *hardware* que se caracterizan por estar intensamente replicados dentro del PL. Además, estos módulos poseen una funcionalidad simple, pero con amplias posibilidades de configuración. Con ello, la utilidad de estos recursos será implementar los aceleradores *hardware* a utilizar. Dicha implementación requiere la capacidad física de interacción entre los recursos de la FPGA. La configuración de los módulos de la FPGA es especificada mediante un archivo binario (*bitstream*). Los recursos del PL se clasifican en los siguientes [14], [27].

- *Configurable Logic Block* o CLB, organizados en *Slices*.
- Bloques DSP.
- Almacenamiento: Ultra RAM, Block RAM, y RAM Distribuida.
- Bloques especiales: PCIe, Controladores de alta velocidad.

En la Tabla 2 se indica la cantidad de los recursos disponibles en el dominio PL del MPSoC XCZU9EG-2FFVB1156E [14], [27].

Tabla 2: Cantidad de recursos de la parte PL [32], [38]

Recurso	Disponibilidad
CLB	274 K
DRAM (mediante LUTs)	8.8 Mb
BRAM	32.1 Mb
DSP	2520
Transceptores GTH 16.3Gb/s	24

5.3.1. CLBs

Cada CLB en la arquitectura UltraScale contiene un *slice*. Cada *slice* está formado por 8 LUTs (*Look Up Tables*), 16 *flip/flops* y multiplexores. Además, este módulo posee lógica de acarreo *look-ahead* de 8 *bits* para las operaciones aritméticas que se pueden conectar en cascada a otros CLBs. Estas conexiones posibilitan la implementación mediante CLBs de operaciones aritméticas con datos cuya longitud en *bits* sea superior a 8. Respecto a las LUTs, cada una de ellas puede configurarse para operar como una LUT de 6 entradas o como dos LUTs de 5 entradas que generan salidas independientes, pero comparten las

mismas entradas. Las salidas de las LUTs pueden ser multiplexadas según se muestra en el esquemático de la Figura 14 (F7, F8 y F9 *muxes*) de tal forma que se pueden implementar funciones de hasta 9 *bits* de entrada [14], [39].

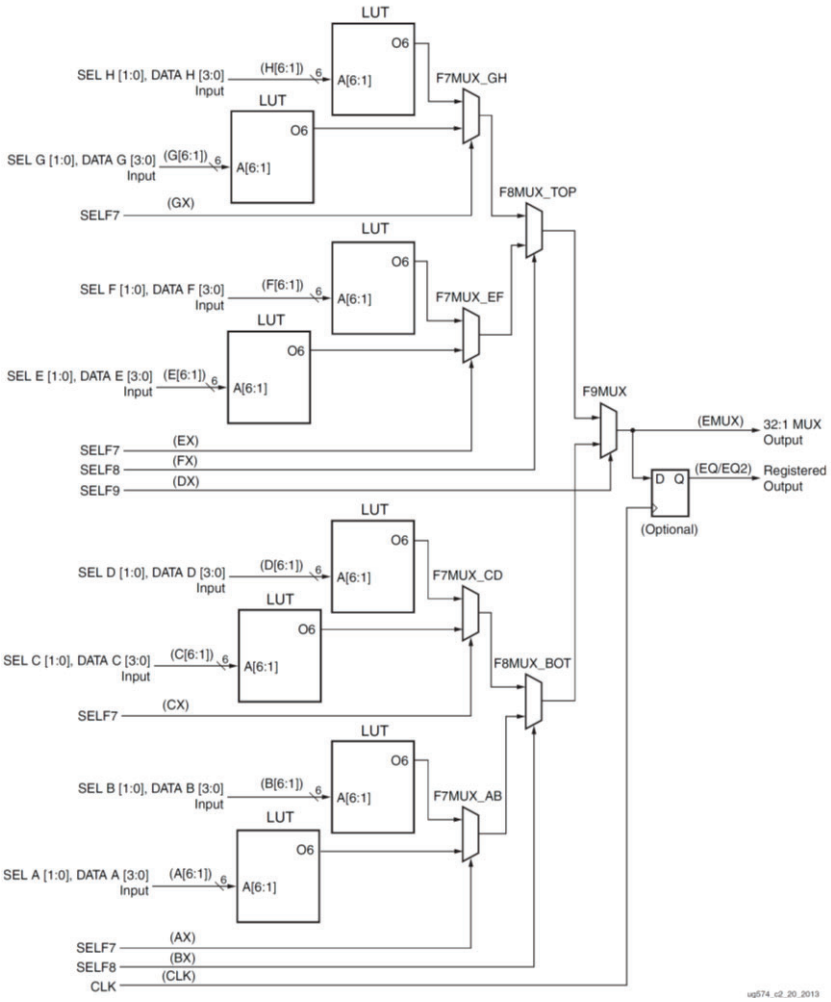


Figura 14: Conexionado entre las LUTs y los multiplexores dentro de un CLB [39]

Hay dos tipos de *slices* en la arquitectura del PL. Por una parte, en el SLICEL las LUTs se utilizan para implementar funciones lógicas. En el SLICEM, además las LUTs se pueden utilizar para implementar memoria RAM distribuida y de registro de desplazamiento que requiera la arquitectura del acelerador *hardware* a implementar. En el caso de operar como memoria RAM, esta tendrá un dimensionado máximo de 512 *bits*, mientras que el dimensionado del registro de desplazamiento será de 256 *bits* [14], [39].

Otra característica configurable del CLB es su salida. En este sentido, cada CLB posee cuatro salidas por cada LUT, dos salidas combinacionales y dos salidas registradas. Las salidas combinacionales se corresponden a la salida de las LUTs, o de alguno de los

multiplexores. Por otra parte, las salidas registradas se corresponden a las salidas de dos *flip/flops*. La entrada de estos *flip/flops* puede asociarse a las dos salidas de cada LUT, el valor de acarreo asociado a cada LUT, la suma saliente de cada LUT con el correspondiente *bit* de acarreo, la salida de los multiplexores conectados directamente a las LUTs y las entradas de la CLB no asignadas a ninguna LUT (Figura 15) [14], [39].

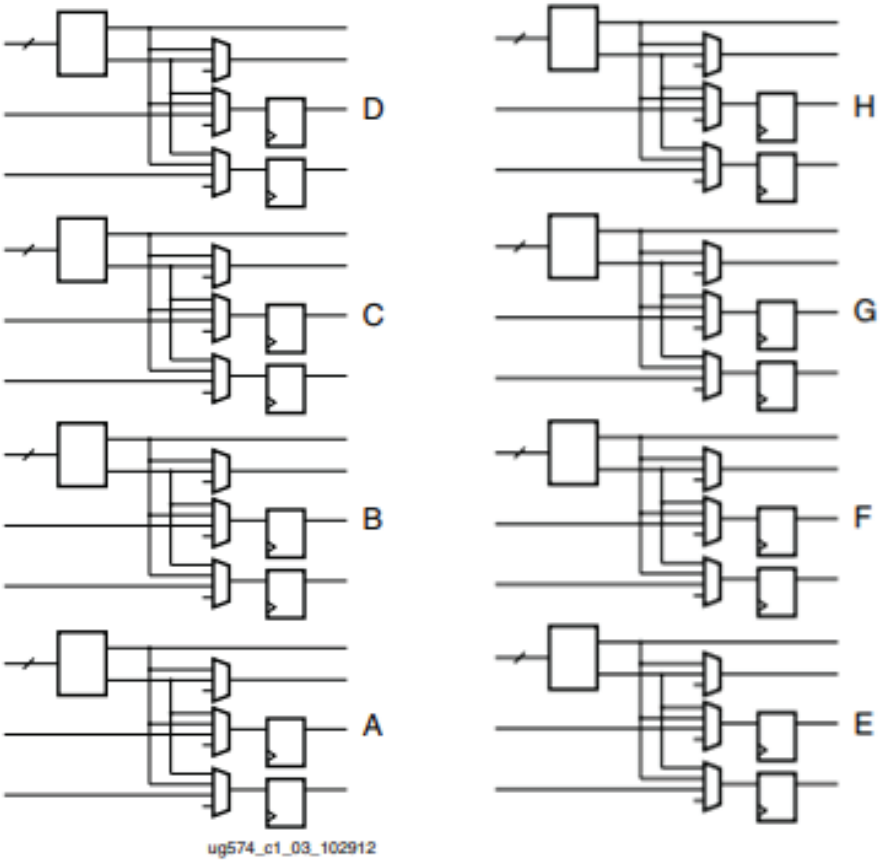


Figura 15. LUTs y elementos de almacenamiento de un Slice [39]

5.3.2. Bloques DSP48E2

Su uso más habitual consiste en implementar operaciones aritméticas en la arquitectura *hardware* que, por su complejidad, su implementación mediante CLBs conllevaría una utilización excesiva de recursos PL [14], [39]. En la Figura 16 se presenta la arquitectura de un módulo DSP. Dicho módulo está formado por un sumador de 48 *bits*, un sumador previo de 27 *bits*, un multiplicador de 27x18 *bits*, una ALU y un detector de patrones [14], [27], [40].

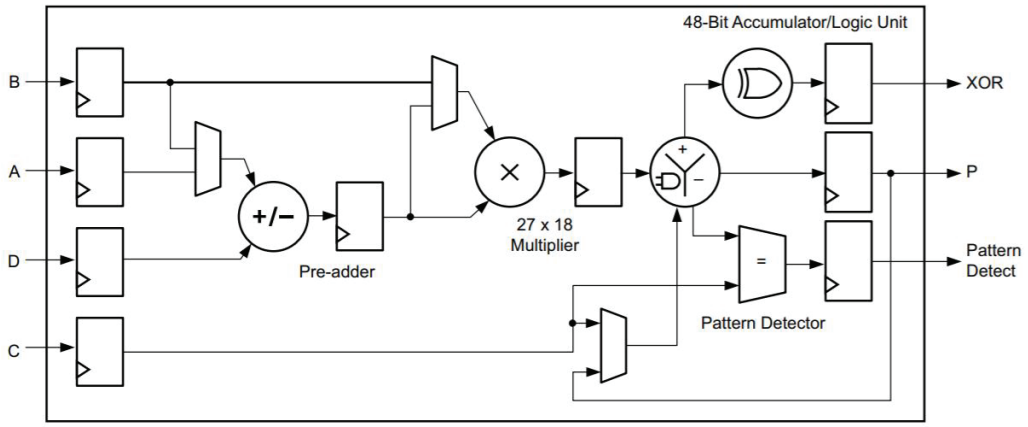


Figura 16: Esquemático del DSP48E2 [27], [40]

En la Figura 16, el pre-*adder* opera con operandos de 27 *bits*. Los datos de entrada al multiplicador son de 27 x 18 *bits*, acompañado de un registro acumulador. Estos datos pueden poseer signo siguiendo la representación en complemento a 2. El siguiente bloque es una ALU con funcionalidad SIMD, adquiriendo los datos de 2 buses de entrada de 48 *bits* y generando 2 salidas de 24 *bits* o 4 salidas de 12 *bits*. Esta ALU es configurable para efectuar operaciones aritméticas de suma, resta y suma acumulativa. En lo que respecta a operaciones lógicas, la ALU es capaz de realizar operaciones lógicas con datos de 48 *bits* [14], [40].

El resultado de la ALU puede asignarse a la salida del DSP. Opcionalmente, este resultado puede dirigirse a un módulo detector de patrones o en una XOR. Entre las aplicaciones del detector de patrones, se encuentra el redondeo convergente o simétrico. Además, este detector de patrones puede emplearse en la implementación de funciones lógicas de 96 *bits* cuando opera de manera conjunta con la funcionalidad lógica de la ALU. En lo que respecta al módulo XOR, este es útil para implementar estrategias de FEC (*Foward Error Correction*) y CRC (*Cyclic Redundancy Checking*) para la detección y corrección de errores en los datos [14], [40].

Con respecto al rendimiento temporal, los bloques DSP48E2 pueden funcionar en modo segmentado para facilitar la conexión en cascada de varios bloques [14], [27], [40].

5.3.3. Memorias RAM

Como recursos de memoria, la lógica programable incluye módulos de memoria específicos en todos los MPSoC de la familia UltraScale, facilitando el almacenamiento

temporal de los datos. Se incluyen tanto las memorias BRAM como los bloques UltraRAM de alta capacidad [14], [27].

Una BRAM consiste en un bloque de memoria síncrono de 36 Kbit que presentan dos puertos de reloj. Dicho bloque es configurable para operar como un solo módulo de memoria, o como 2 bloques de memoria de acceso independiente de 18 Kbit. Con respecto al tamaño de las palabras almacenadas por estos bloques de memoria, esta es ajustable siguiendo las configuraciones 32K x 1, 16K x 2, 8K x 4, 4K x 9, 2K x 18, 1K x 36 o 512 x 72. Estas configuraciones pueden establecerse en cada uno de los puertos de la BRAM de manera que difieran entre sí. Las memorias pueden funcionar como *Single Port* (un puerto de escritura y uno de lectura), *Single-Dual Port* (un puerto de escritura y dos de lectura) o *True Dual Port* (dos puertos independientes de escritura y lectura). Presentan tres modos de lectura/escritura: *read-first*, *write-first* y *don't-change.* Las memorias se pueden modificar *byte* a *byte*. De igual forma, es posible crear FIFOs en base a las memorias, incluyendo la señalización correspondiente (*empty, full, almost-empty, almost-full, error*). Respecto a la gestión de errores, las BRAMs presentan funcionalidad de corrección de errores de 1 *bit* y detección de aquellos de 2 *bits*. Para ello, se recurre a la introducción en el código de 8 *bits* adicionales [14], [27].

En lo que respecta a los bloques de memoria UltraRAM, estos consisten en memorias síncronas de un único reloj de alta densidad de 288 Kbit con acceso mediante puerto dual síncrono. Este tipo de módulos se encuentra solamente en los MPSoCs de las familias UltraScale*+*. Cada uno de estos puertos opera con un ancho de registros de 9 *bytes*, y pueden utilizarse indistintamente para operaciones de lectura y escritura. Para la gestión de errores, las memorias UltraRAM incorporan 8 *bits* de redundancia mediante codificación *Hamming*. Con ello, se implementa corrección de errores de 1 *bit* y detección de errores de 2 *bits* [14], [27]. A diferencia de los BRAMs estos bloques de memoria no pueden configurarse como memorias ROM [14].

5.4. Comunicación PS – PL

Considerando que el acelerador *hardware* se implementa en el bloque PL y el resto de la aplicación se ejecuta en el bloque PS del MPSoC, es necesario disponer de interfaces

que interconecten estos bloques y ofrezcan prestaciones aceptables para las transferencias de datos. Además, estas interfaces deben posibilitar la sincronización entre la ejecución *software* y la aceleración *hardware*. Para esta intercomunicación, el MPSoC incluye interfaces AXI4, organizadas en distintos dominios de potencia [33]. En este trabajo se utilizan interfaces AXI4 en el dominio *Full-Power* (FPD). En la Figura 17 se exponen las interfaces de intercomunicación PS-PL existentes en el MPSoC considerado [14].

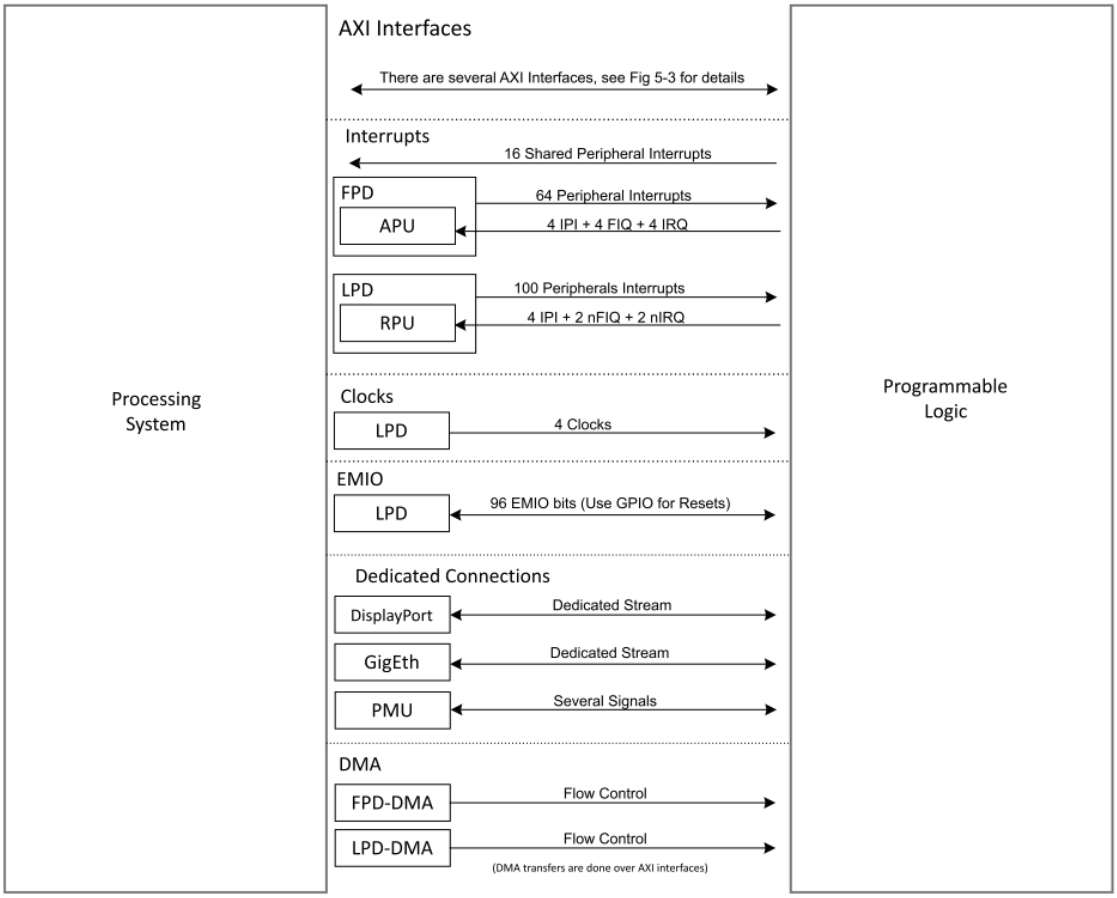


Figura 17: Conexiones entre PS y PL [33]

5.4.1. Interfaz AXI

La interfaz AXI4 es un protocolo de bus en chip capaz de operar en modo maestro-esclavo con soporte para transmisiones a ráfagas [14], [33]. Cada interfaz AXI está constituida por 5 canales cuyos cometidos son:

- **Canal de dirección de lectura**. Este canal permite que módulo maestro comunique la dirección donde se van a leer los datos en el módulo esclavo. También, se utiliza para especificar los *bits* de control de dicha operación de lectura [14], [33].

- **Canal de dirección de escritura**. Este canal es utilizado por el módulo maestro para comunicar la dirección escritura en la memoria del módulo esclavo. Además, permite indicar los *bits* de control de esta operación de escritura [14], [33].
- **Canal de lectura de datos**. Es el canal utilizado para que el módulo esclavo de la comunicación AXI pueda comunicar el dato de lectura solicitado por el módulo maestro [14], [33].
- **Canal de escritura de datos**. Este canal permite que el módulo maestro pueda comunicar el dato a escribir en el módulo esclavo [14], [33].
- **Canal de respuesta de escritura.** Se utiliza para que el módulo maestro sea informado de la terminación de la transacción de escritura. Dicha información es especificada por el módulo esclavo [14], [33].

En la Figura 18, se exponen gráficamente los canales existentes en una interfaz AXI.

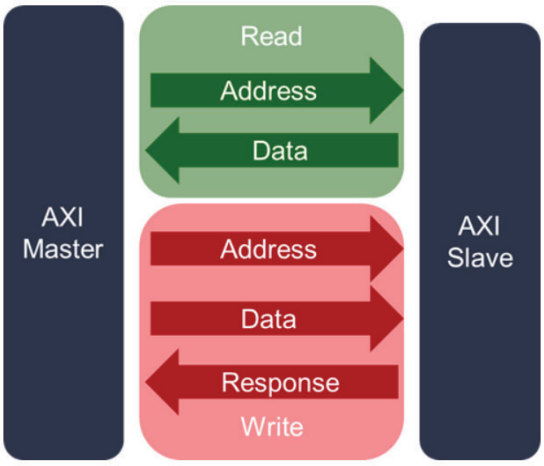


Figura 18: Canales de una interfaz AXI estándar [41]

Se establecen grados de prioridad para regular el acceso de múltiples módulos a una misma memoria. Esta prioridad suele ser mayor para las transferencias que precisan baja latencia como puede ser el caso de la comunicación de datos entre la APU y la RPU. Por otra parte, se encuentran las transferencias entre la GPU y la lógica programable (PL). Este último tipo de transferencias suele caracterizarse por ser poco exigente acerca de la latencia. No obstante, suelen requerir un *throughput* elevado [14], [33].

Un caso particular de tráfico de datos es el denominado tráfico isócrono, definido como aquel que precisa que los datos se transmitan manteniendo una misma velocidad de

transferencia. En consecuencia, constituye un tráfico con escasos requisitos de latencia, salvo cuando la transferencia se encuentra próxima al *timeout*. Para esta última situación, el tráfico isócrono se convierte en el más prioritario. Las transferencia de imagen y vídeo suelen ser un claro ejemplo de este tipo de tráfico [14], [33].

5.4.2. Transferencias PS-PL

Aparte de intercomunicar módulos *hardware* agrupados ya sea en el propio PS o el PL del MPSoC, se incluyen interfaces AXI que interconectan PS y PL. Con ello, se posibilita la transferencia y sincronización entre los módulos del PS y los módulos del PL. Estas conexiones se resumen en la Figura 19 [14].

El módulo CCI (*Cache-Coherent Interconnect*) constituye un módulo de comunicación con el controlador de caché que mantiene la coherencia en la información compartida por varios módulos intercomunicados [14], [33]. Esta coherencia implica que los datos compartidos entre diferentes módulos se encuentren actualizados dentro de sus respectivas memorias caché. Además, se dispone del módulo *System Memory Management Unit* (SMMU) para la conversión de direcciones de memoria entre el PS y el PL. De esta manera, el PL puede utilizar direcciones virtuales [14], [28].

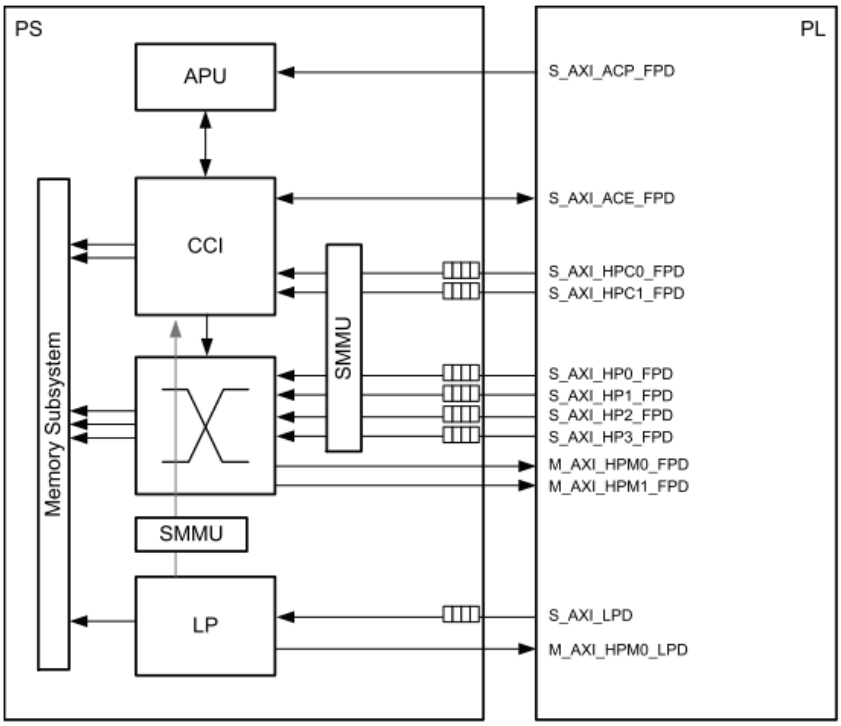


Figura 19: Conexiones AXI PS – PL [33]

En la Figura 19 se indican los nombres de las interfaces AXI, utilizando "S" para interfaces esclavas y "M" para interfaces maestras. Además, la terminación de estos nombres consiste en 3 letras finales para especificar el dominio de potencia en el que se encuentra disponible la interfaz. Asimismo, si la terminación es LPD (*low-power domain*) se trata del dominio de bajo consumo de potencia. En caso contrario, el nombre termina como FPD (*full-power domain*) lo que implica se habilita cuando el sistema funciona a máximo rendimiento.

Las interfaces que interconectan la PS y la PL pueden clasificarse en las siguientes categorías [14]:

- **ACE (*AXI Coherency Extension*).** Su principal variación en comparación con el interfaz AXI estándar consiste incorporar 5 canales adicionales. De estos canales, 3 se emplean para garantizar la coherencia de los datos sin necesidad de codificación *software*, mientras que los 2 restantes se aplican para procesos de verificación [14], [28]. Con ello, también se establece el orden en las transferencias [34]. Esta interfaz permite un ancho de 128 *bits* [14], [28].

- **HPCP (*High Perfomance Coherent Port*).** La principal característica de esta interfaz consiste en permitir una rápida transferencia de los datos. Para ello, se incluyen colas FIFO en cada puerto, lo que facilita la transferencia de datos a ráfagas [14], [28]. El ancho de la interfaz HPCP también puede configurarse a 128, 64 o 32 *bits* [14], [34].

- **ACP (*Accelerator Coherency Port*).** Esta interfaz permite un acceso asíncrono y coherente a memoria caché efectuado desde la PL hacia la APU. Esto simplifica la programación *software*, puesto que la PL puede incorporar múltiples módulos maestros al igual que los núcleos de la APU. Para poder garantizar la coherencia, se recurre al módulo SCU (*Snoop Control Unit*). Esta interfaz posee un ancho de 128 *bits* [14], [34].

- **Interfaces AXI para LPD *(Low Power Domain*).** Esta interfaz permite al PL acceder a las memorias OCM y TCM, las cuales tienen una latencia menor. Con respecto al ancho de trasferencia, este puede ajustarse a 128, 64 y 32 *bits* [14], [34].

5.5. Conclusiones

En este capítulo, se han estudiado los componentes que constituyen el MPSoC de la placa de prototipado ZCU102. Entre estos recursos se encuentran los pertenecientes a la parte de lógica programable, entre los cuales se encuentran CLBs, DSPs, FFs y RAMs. Estos módulos se utilizan para implementar los aceleradores *hardware* que requiere la aplicación.

Por otra parte, el MPSoC contiene recursos agrupados en el PS o sistema de procesamiento. La mayoría de estos recursos consisten en microprocesadores, entre los cuales se halla una APU de 4 núcleos ARM Cortex-A53. Este módulo será el responsable de ejecutar el *host* de la aplicación.

Para interconectar la PS con la parte de lógica programable de la aplicación se dispone de varias interfaces. Entre estas interfaces se encuentran las interfaces AXI4, que se utilizarán en esta aplicación para comunicar la APU con la PL y, tanto posibilitar la interacción y coordinación entre el *host* y los aceleradores *hardware*.

Capítulo 6. OpenCL para C++

6.1. Introducción

Una parte fundamental de la aplicación consiste en la preparación y sincronización de las transferencias entre el *host* y los aceleradores *hardware*. Para ello se hace uso del estándar OpenCL (*Open Computing Language*), que constituye una extensión de C y C++ para la programación de sistemas heterogéneos. Con ese fin, esté estándar define un conjunto adicional de datos, estructuras y funciones.

Entre las ventajas que presenta OpenCL pueden citarse su portabilidad fruto de su amplia aceptación y compatibilidad con diferentes fabricantes de *hardware* entre los que se incluye NVIDIA, AMD, Intel, etc. Además, OpenCL puede emplearse para la programación y sincronización con el *host* de aceleradores *hardware* implementados en distintos tipos de arquitecturas, incluyendo CPUs, GPUs o FPGAs [42], [43], [44]. Es importante tener en cuenta que OpenCL no es portable en términos de prestaciones.

Con OpenCL es posible modelar sistemas de computación paralela, ya sean homogéneos o heterogéneos, como es el caso de los MPSoC. En cualquier caso, el modelo define un único *host* y uno o más dispositivos de cómputo o aceleradores. Por tanto, podemos decir que una aplicación OpenCL se divide en dos partes (Figura 20): un programa anfitrión que se ejecuta en el *host*, que se encarga de gestionar la aplicación mediante APIs OpenCL y un conjunto de *kernels* que se ejecutan en los dispositivos de aceleración *hardware* y generalmente están definidos en el propio OpenCL (es posible realizar su diseño por otros medios).

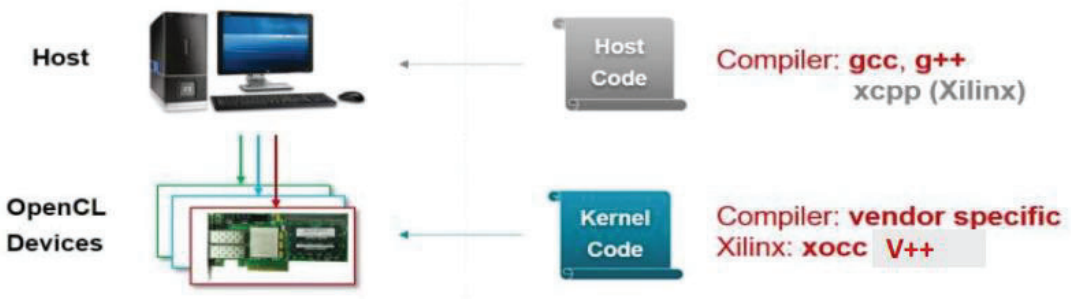


Figura 20. Modelo de desarrollo con OpenCL [45]

La interacción entre el *host* y el dispositivo OpenCL se realiza mediante colas de comandos. Estos comandos pueden ser de tres tipos: comandos de ejecución en los *kernels,* comandos para el acceso a memoria, y comandos de sincronización (Figura 21) [45].

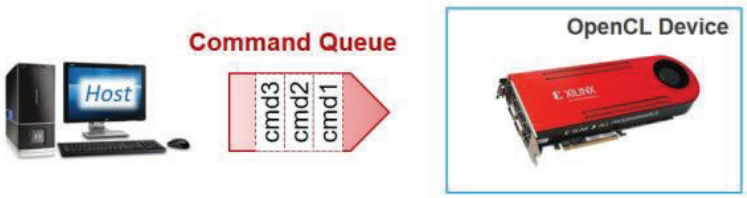


Figura 21. Ejemplo de colas de comandos [45]

Entrando más en detalles, la utilización de OpenCL se realiza mediante APIs definidas en el lenguaje. Para la invocación de estas APIs se utilizan los *wrappers* C++ especificados en el código fuente en C/C++ del *host*. Cada uno de estos *wrappers* constituye una función en C++ que encapsula las funciones equivalentes en C que permiten invocar la misma operación con un nivel de abstracción inferior al del *wrapper*.

Por tanto, siguiendo el modelo de programación orientado a objetos de C++, la invocación de los recursos de OpenCL se realiza mediante la creación de objetos de las clases correspondientes y la posterior invocación de funciones miembro de estos objetos [42], [44]. La inclusión de dichas clases en la programación del *host* para su utilización se especifica al incluir el archivo de cabecera “cl2.hpp”. Las instancias de estas clases deben poseer las relaciones de dependencia, agregación y herencia de clases expuestas en la Figura 22 para constituir y programar el sistema heterogéneo. En dicha figura, las relaciones de dependencia se definen como “navegabilidad”. Las relaciones de agregación consisten en que el concepto representado por la clase a la que se dirige la agregación

agrupa al representado por la otra. Por otra parte, la cantidad de instancias de una determinada clase que la otra clase puede abarcar se detalla en la Figura 22 mediante la leyenda “Cantidad”.

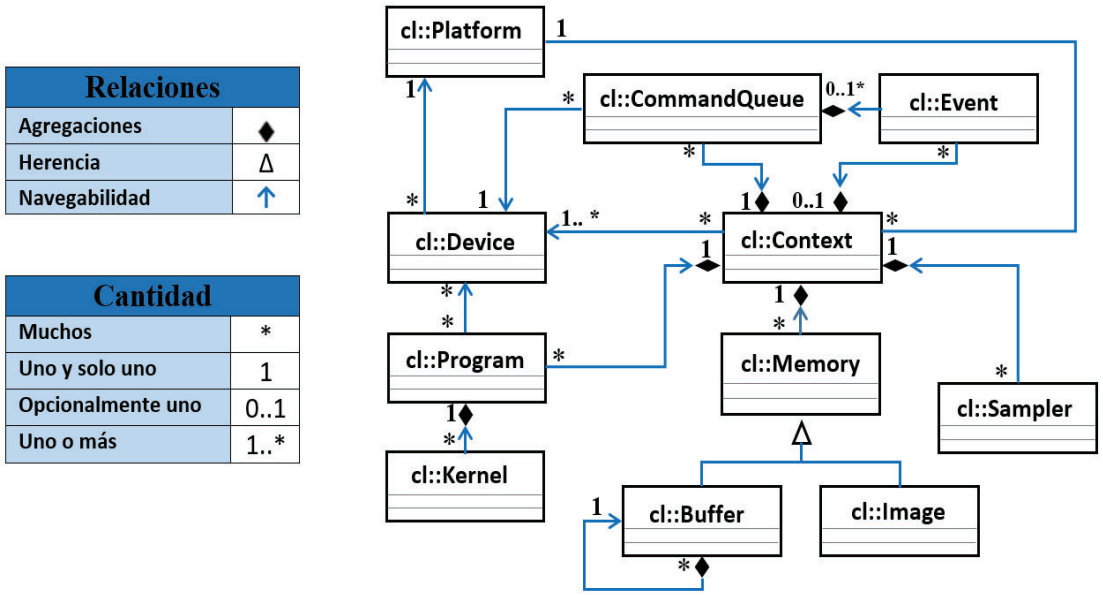


Figura 22: Relaciones entre clases de OpenCL para C++ (adaptado de [44]*)*

Los recursos de programación de OpenCL permiten la adaptabilidad de sistemas heterogéneos cuyos recursos *hardware* implicados y su utilización son muy divergentes entre sí.

6.1.1. Modelo de ejecución

La ejecución de un programa OpenCL se produce en dos partes: *kernels* que se ejecutan en uno o más dispositivos OpenCL (OpenCL *device*) y un programa *host* que se ejecuta en el procesador central. El programa *host* define el contexto de los núcleos y gestiona su ejecución.

El programa *host* define el *kernel* y luego emite una serie de comandos para ejecutar ese *kernel* en un dispositivo OpenCL. Hay muchos detalles que van de la mano con la comprensión de cómo se realiza la ejecución de un *kernel* en el dispositivo. Algunos de los términos que querrá familiarizarse incluyen elementos de trabajo (*work-items*), grupos de trabajo (*work-groups*) y *NDRange*.

El sistema heterogéneo por configurar debe poseer un único *host*, el formado por un microprocesador que ejecuta una aplicación *software*. Dicho *host* gestiona los demás

dispositivos de cómputo, constituyendo estos últimos los aceleradores *hardware*. Esta gestión implica la sincronización entre las diversas operaciones gestionadas e incluso entre dichas operaciones y la ejecución del *host* [44]. Entre las operaciones a gestionar por el *host* en OpenCL se incluye programar y configurar los aceleradores *hardware*, así como ordenar la ejecución de las funciones de aceleración que dichos aceleradores implementan.

Por otra parte, se encuentran las operaciones de transferencia de datos entre los elementos de cómputo del sistema. Dichas transferencias incluyen aquellas entre el *host* y los aceleradores *hardware,* entre los aceleradores *hardware* o incluso entre funciones de aceleración *hardware* [44].

En la Figura 23 se muestra la arquitectura de un sistema heterogéneo programado con OpenCL. En ella, *OpenCL device* identifica a los aceleradores *hardware* y *Compute Unit* a la implementación de una función de aceleración en dichos aceleradores. Además, cada una de las operaciones que conforman una función de aceleración se denomina un *processing element*.

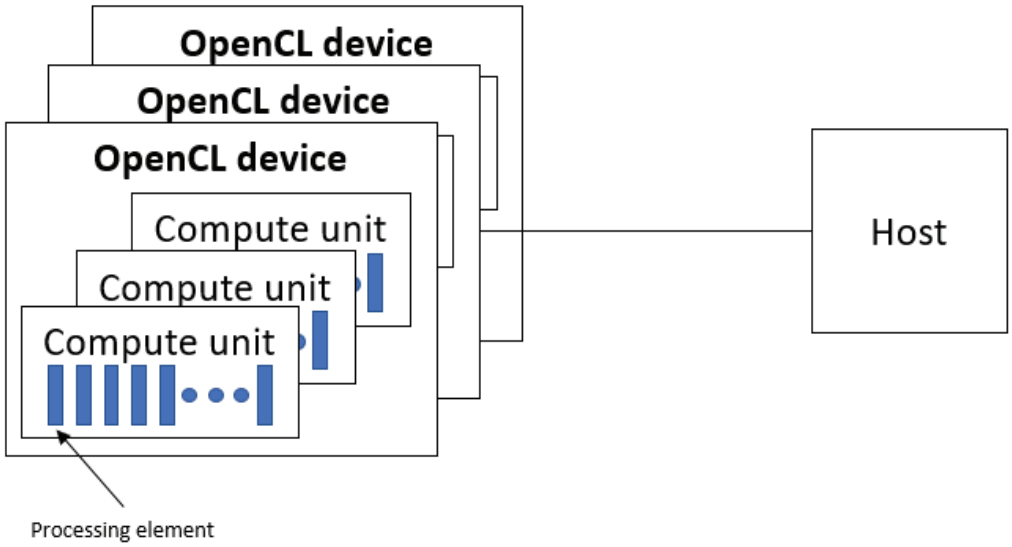


Figura 23: Arquitectura de un sistema heterogéneo adaptado al estándar OpenCL [44]

La especificación OpenCL define las colas de contexto y comandos. El *host* define un contexto para la ejecución de los *kernels*. El contexto incluye los siguientes recursos:

1. *Devices*. La colección aceleradores *hardware* que va a utilizar el *host*.
2. *Kernels*. Las funciones de aceleración *hardware* que se ejecutan en los aceleradores *hardware*.

3. *Program Objects*. El código fuente del programa y los ejecutables que configuran los aceleradores *hardware*.
4. *Memory Objects*. Un conjunto de objetos de memoria visibles para el *host* y los aceleradores *hardware*. Los objetos de memoria contienen valores que pueden ser operados por instancias de un *kernel.*

El *host* crea y manipula el contexto utilizando funciones de la API de OpenCL. Además, el *host* crea una estructura de datos llamada *Command-Queue* para coordinar la ejecución de los *kernels* en los dispositivos. Adicionalmente, el *host* coloca el comando en la cola de comandos que luego se programa en los dispositivos dentro del contexto. Estos incluyen:

- Comandos de ejecución del *kernel*: Ejecuta un *kernel* en los elementos de procesamiento de un dispositivo.
- Comandos de memoria: Transfiere datos a, desde o entre objetos de memoria, o asigne y desasigne objetos de memoria desde el espacio de direcciones de *host*.
- Comandos de sincronización: Restringe el orden de ejecución de los comandos.
- Modelo de memoria

Existen tres tipos de objetos de memoria: *Buffers*, *images* y *pipes*. Los objetos tipo *buffer* representan zonas de memoria continuas que están disponibles para el de acceso de lectura/escritura mediante punteros por los *kernels.* Los objetos tipo memoria almacenan imágenes de forma optimizada, aunque no están soportadas en el perfil *embedded,* usado es este trabajo. Por último, en los objetos tipos *pipes*, los datos se almacenan en FIFOs, aunque solamente son accesibles entre dispositivos OpenCL, siguiendo un modelo de cómputo *DataFlow*.

Los elementos de trabajo o *work-items* que ejecutan un *kernel* tienen capacidad para acceder a cuatro regiones de memoria distintas (Figura 24):

- Memoria global. Esta región de memoria permite el acceso de lectura/escritura a todos los elementos de trabajo de todos los grupos de trabajo. Los elementos de trabajo pueden leer o escribir en cualquier elemento de un objeto de memoria. Las lecturas y escrituras en la memoria

global se pueden almacenar en caché en función de las capacidades del dispositivo.

- Memoria de constantes. Es una región de la memoria global que permanece constante durante la ejecución de un núcleo. El *host* asigna e inicializa objetos de memoria colocados en esta zona de memoria.
- Memoria local. Se trata de una región de memoria local para un grupo de trabajo. Esta región de memoria se puede utilizar para asignar variables compartidas por todos los elementos de trabajo de ese grupo de trabajo. Se puede implementar como regiones dedicadas de memoria en el dispositivo OpenCL. Como alternativa, la región de memoria local se puede asignar a secciones de la memoria global.
- Memoria privada. Esta región de memoria es privada para un elemento de trabajo. Las variables definidas en la memoria privada de un elemento de trabajo no son visibles para otro elemento de trabajo [45].

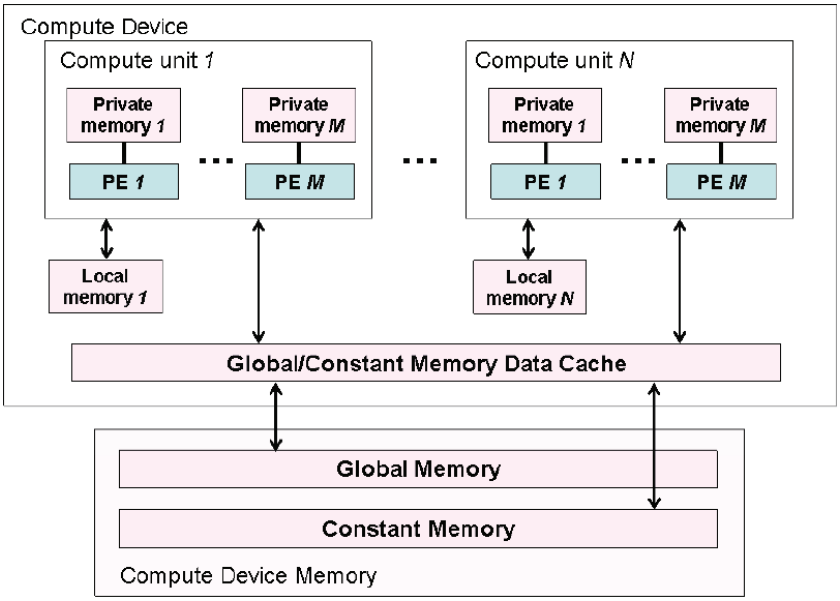


Figura 24. Arquitectura del modelo de memoria (adaptado de [45]*)*

6.1.2. Xilinx Runtime (XRT)

La librería Xilinx Runtime (XRT) permite la comunicación entre el *host* y el acelerador en función del tipo de sistema que se desarrolla. Este recurso se implementa como una combinación de espacio de usuario y componentes de controlador de *kernel*. XRT admite

tarjetas aceleradoras basadas en PCIe (basado en *host* x86 para sistemas basados en la nube) y arquitectura integrada basada en MPSoC (ARM Cortex-A9 o A53 para sistemas empotrados), proporcionando una interfaz de *software* estandarizada para la lógica programable de Xilinx.

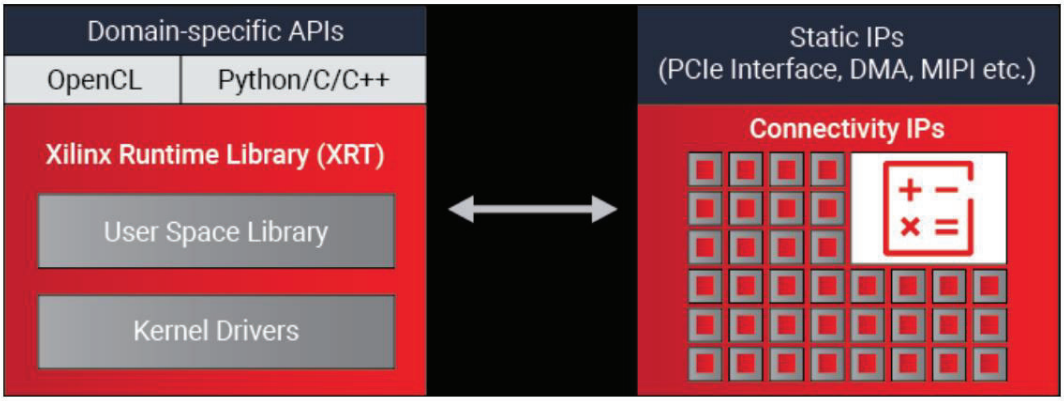


Figura 25. Comunicación utilizando XRT [46]

En la Figura 26 se muestran detalles de la organización por capas de la librería. Como se puede apreciar, proporciona soporte para las API de OpenCL que utilizan Xilinx Runtime (XRT) basado en Linux para programar los *kernels* y controlar el movimiento de datos [47].

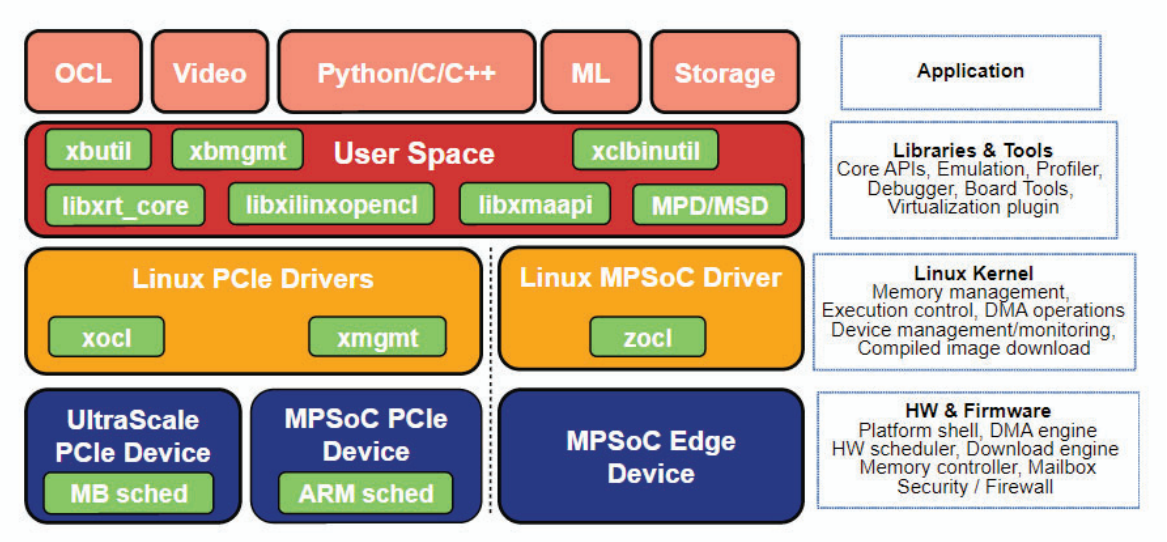


Figura 26. Xilinx Runtime (XRT) Stack [47]

En este capítulo, se describen las clases, así como sus funciones, que se utilizan en la programación de la sincronización e intercomunicación entre el *host* y los *kernels* FPGA. Estas clases se especifican siguiendo el orden en el cual deben inicializarse dentro de la programación del *host*.

6.2. cl::Platform

El objetivo de esta clase consiste en identificar las plataformas sobre las que se ejecuta la aplicación. Una plataforma constituye una forma de implementar la funcionalidad mediante las APIs de OpenCL. Es dependiente del fabricante del *hardware* a utilizar [42], [48].

La plataforma OpenCL a utilizar será la denominada "Xilinx", puesto que es la que está disponible para la placa de prototipado Zynq UltraScale+ MPSoC ZCU102 al que se destina el diseño y ejecución de la aplicación. En esta aplicación, se emplean las funciones **get**, **getInfo** y **getDevices** pertenecientes a esta clase.

6.2.1. Función get

Esta función permite obtener todas las plataformas que se encuentran accesibles para el *host*. Este conjunto de plataformas se introduce en un vector de objetos **cl::Platform**, correspondiéndose cada objeto a una plataforma diferente. El vector donde se almacena la información de las plataformas encontradas se declara de manera externa a la función. Para la escritura de este vector, deberá especificarse a la función un puntero que indique la dirección de memoria donde se ubique. Para ello, deberá utilizarse el parámetro "platforms" de esta función [42], [44]. En el Código 1 se presenta un ejemplo de la utilización de esta función dentro de la aplicación desarrollada. En dicho código, debe destacarse que "err" es una variable que almacena un valor que identifica si la función de OpenCL invocada se ha ejecutado con éxito o, en caso contrario, identifica el error producido. Esta variable está presente en todas las invocaciones de funciones de OpenCL.

```
cl_int err;
std::vector<cl::Platform> platforms;
err = cl::Platform::get(&platforms);
```

Código 1: Utilización de la función cl::Platform::get

6.2.2. Función getInfo

Esta función permite obtener información acerca de la plataforma a la que hace referencia el objeto **cl::Platform** que invoca esta función. La información concreta que se desea obtener es especificada a este como un parámetro "cl_int" del "template" de la

invocación de esta función. Cada tipo de información se corresponde unívocamente a un valor numérico identificativo. No obstante, para facilitar que el código resulte más inteligible, el programador puede especificar una macro que equivale a dicho valor numérico. El nombre de cada macro y la información a la que identifica se expone en la Tabla 3. Asimismo, el tipo de variable de la información devuelta se corresponde a un *array* de "char" en C y a una *string* en C++ [42], [44].

Tabla 3: Macros que identifican diferentes datos de las plataformas [42]

Nombre de la macro	Significado
CL_PLATFORM_NAME	Nombre que identifica a la plataforma
CL_PLATFORM_VENDOR	Empresa vendedora de la plataforma
CL_PLATFORM_VERSION	Versión más avanzada de OpenCL que soporta la plataforma
CL_PLATFORM_PROFILE	Indica si la plataforma soporta la totalidad del estándar OpenCL (opción FULL_PROFILE) o solo la versión empotrada de dicho estándar (EMBEDDED_PROFILE).
CL_PLATFORM_EXTENSIONS	Conjunto de extensiones que soporta la plataforma.

En la aplicación desarrollada, la utilización de la función **cl::Platform::getInfo** realiza la consulta del nombre identificativo de la plataforma evaluada, es decir, de la información identificada con la macro "CL_PLATFORM_NAME". Dichas consultas se realizan en la ejecución de la aplicación para las plataformas contenidas en el vector "platforms" y obtenidas tras invocar la función **cl::Platform::get** como se mostró en el Código 1. Con ello, se busca aquella plataforma cuyo nombre sea "Xilinx", correspondiéndose a la plataforma utilizada en esta aplicación. En Código 2 se muestra el extracto del código que implementa dicha funcionalidad.

```
cl::Platform platform;
const std::string vendor_name = "Xilinx";

for (i = 0; i < platforms.size(); i++){
    platform = platforms[i];
    std::string platformName=platform.getInfo<CL_PLATFORM_NAME>(&err);
    if (!(platformName.compare(vendor_name))){
        std::cout << "Plataforma "<< platformName.c_str() <<
        " encontrada" << std::endl;
        break;
    }
}
```

Código 2: Utilización de la función cl::Platform::getInfo

6.2.3. Función getDevices

Esta función permite obtener el conjunto de dispositivos disponibles que pueden operar con la plataforma indicada. Utiliza dos parámetros, especificados en la cabecera de la función. El primero se denomina "type", y es una variable del tipo "cl_device_type" que permite concretar qué tipos de dispositivos se desean obtener. El siguiente parámetro se denomina "devices" y constituye un puntero al vector de objetos **cl::Device** que almacena los dispositivos encontrados con las características especificadas por "type". El parámetro "type" puede adquirir un conjunto de valores numéricos reducidos, correspondiéndose cada uno de estos valores a una macro diferente. Asimismo, la relación entre cada macro y la implicación del valor de "type" al que hace referencia es la expuesta en la Tabla 4 [42], [44].

Tabla 4: Macros que identifican especificaciones acerca de los dispositivos a buscar [42]

Nombre de la macro	Significado
CL_DEVICE_TYPE_ALL	Cualquier dispositivo que pueda asociarse con la plataforma
CL_DEVICE_TYPE_DEFAULT	Dispositivos del tipo por defecto establecido por la plataforma
CL_DEVICE_TYPE_CPU	Procesador del *host*
CL_DEVICE_TYPE_GPU	Dispositivos que incorporan GPUs
CL_DEVICE_TYPE_ACCELERATOR	Dispositivos externos utilizados para aceleración computacional.

En el código del *host*, el tipo de dispositivo a buscar se corresponde con el indicado mediante el valor de la macro "CL_DEVICE_TYPE_ACCELERATOR", debiendo ser un dispositivo válido para la aceleración *hardware* compatible con la plataforma "Xilinx". En el Código 3 se muestra su utilización.

```
std::vector<cl::Device> devices;
err = platform.getDevices(CL_DEVICE_TYPE_ACCELERATOR, &devices);
return devices[0];
```

Código 3: Utilización de la función cl::Platform::getDevices

6.3. cl::Device

Esta clase se emplea para hacer referencia a un dispositivo *hardware* capaz de operar como acelerador *hardware* de la aplicación [42], [43], [44], [48]. En esta aplicación,

se utiliza un único dispositivo correspondiente a la FPGA e identificado en la codificación del *host* con el nombre "device".

Análogamente al caso de **cl::Platform**, la clase **cl::Device** dispone de una función miembro denominada **getInfo** que permite extraer diferentes datos del dispositivo al que hace referencia el objeto que la invoca. El tipo de información a extraer por esta función puede concretarse especificando el valor de un parámetro de *template* utilizado por la misma. La diversidad de datos del dispositivo que pueden consultarse es mucho más amplia que la encontrada en el caso de **cl::Platform** habiendo un total de 50 [42], [44].

6.4. cl::Context

Esta clase permite establecer un contexto. En OpenCL, el contexto es un recurso que permite agrupar a un conjunto de dispositivos que operan conjuntamente en la ejecución de un proceso común. Dichos dispositivos deben disponer además de una plataforma común. Los dispositivos englobados en contextos diferentes no podrán efectuar transferencias entre ellos de manera directa. Sin embargo, sí podrán efectuar transferencias entre ellos y el *host* [42], [44].

En este caso se utiliza un único contexto, puesto que se emplea un único dispositivo para aceleración *hardware*. La creación del objeto **cl::Context** para esta aplicación se muestra en el Código 4, siendo "context" el nombre con el que se referencia dicho objeto.

```
context=cl::Context(device, nullptr, nullptr, nullptr, &err);
```

Código 4: Creación del objeto cl::Context

6.5. cl::CommandQueue

Esta clase referencia una cola de comandos de OpenCL. Según la metodología de OpenCL, una cola de comandos representa un recurso *software* mediante el cual el *host* ordena a un determinado dispositivo que efectúa operaciones de lectura, escritura y de ejecución de una función. Igualmente, en la nomenclatura de OpenCL, las funciones ejecutadas por un acelerador *hardware* se denominan *kernels* [42], [44], [49].

Para la utilización en el *host* de una cola de comandos, esta cola debe inicializarse asignándosele un dispositivo y un contexto. Para ello, se invoca el constructor de un objeto **cl::CommandQueue**, especificándose el contexto y el dispositivo asociados a la cola de comandos a crear. Este último aspecto se logra por medio de los parámetros "context" y "device" respectivamente. Además, el constructor a utilizar dispone de un tercer parámetro denominado "properties". Este último parámetro permite habilitar diferentes características de funcionamiento relativas a la cola de comandos. Las opciones se identifican con un valor numérico por medio de la macro correspondiente. Asimismo, las macros asociadas a esos valores y las implicaciones que establecen son las siguientes [42], [44], [49]:

- CL_QUEUE_PROFILING_ENABLE: posibilita la evaluación mediante objetos **cl::Event** de aspectos temporales de la ejecución de cada comando encolado en la cola de comandos [42], [44], [49].
- CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE: si no se indica esta opción, los comandos se ejecutarán en el mismo orden en el que fueron introducidos por parte del *host* en la cola de comandos. Sin embargo, al habilitarse esta opción se permite que los comandos encolados se ejecuten en un orden distinto cuando se considere conveniente de cara al rendimiento de la aplicación [42], [44], [49].

Los objetos **cl::CommandQueue** se configuran en el código del *host* únicamente con la opción "CL_QUEUE_PROFILING_ENABLE". En el Código 5 se evidencia la creación de los objetos **cl::CommandQueue** utilizados en el *host*.

```
q_in = cl::CommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE,
&err);
q_exe = cl::CommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE,
&err));
q_out = cl::CommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE,
&err));
```

Código 5: Creación de los objetos cl::CommandQueue

La clase **cl::CommandQueue** posee un conjunto de funciones miembro que pueden utilizar cada uno de los objetos instanciados de esta clase. Las funciones a utilizar son las siguientes [42], [49].

6.5.1. Función enqueueWriteBuffer

Con esta función, se invoca un comando de escritura de datos desde el *host* hacia el acelerador *hardware*. Como primer parámetro se especifica una referencia a un objeto **cl::Buffer** que identifica la región de memoria del acelerador *hardware* en la que se solicita escribir. El segundo parámetro es una variable del tipo "cl_bool" denominada "blocking". Dicho parámetro puede poseer el valor 0, identificado con las macros "CL_FALSE" y "CL_NON_BLOCKING", o el valor 1, identificado con las macros "CL_TRUE" y "CL_BLOCKING". En el caso del valor 0, la ejecución del *host* podrá proseguir una vez que el comando de escritura haya sido encolado, sin necesidad de esperar a que se efectúe y finalice la ejecución de dicho comando. En el caso del valor 1, la ejecución del *host* se detendrá hasta que se culmine la ejecución del comando de escritura [42], [49].

Los dos siguientes parámetros son del tipo "size_type" y se denominan "offset" y "size". Como sus nombres indican, el parámetro "offset" identifica la posición del *byte* dentro de la región de memoria identificada con un objeto **cl::Buffer** a partir de la cual se encuentran los *bytes* afectados por esta operación de escritura. Por otra parte, "size" especifica la cantidad de *bytes* a escribir. El siguiente parámetro es un puntero "void*" denominado "ptr" que permite especificar la dirección de memoria dentro del *host* a partir de la cual se encuentran los datos a escribir [42], [49].

Los dos últimos parámetros de la cabecera se utilizan para la sincronización de este comando con el *host* y con otros comandos encolados tanto en esta como en otra cola de comandos. Dichos parámetros consisten en un puntero a un vector de objetos **cl::Event** y un puntero a un objeto **cl::Event**, los cuales se denominan "wait_list" y "event" respectivamente. El parámetro "wait_list" permite indicar las dependencias para la ejecución del comando actual. Dichos comandos deben finalizar antes de empezar la ejecución el comando actual. Para ello, cada uno de los comandos a esperar debe tener asignado un **cl::Event**. Por otro lado, el parámetro "event" permite asignar un objeto

cl::Event al comando, el cual notificará en todo momento el estado de ejecución del comando. La especificación de estos últimos parámetros es de carácter opcional [42], [49].

En el Código 6 se presenta un ejemplo de utilización de esta función. Como la ejecución del comando encolado por dicha función no se sincroniza con ningún comando de OpenCL anterior mediante eventos, el parámetro "wait_list" se especifica como "nullptr". Sin embargo, este comando tiene asignado el primer evento del vector de eventos "filterInEvs" para, entre otras funcionalidades, sincronizarlo con comandos de OpenCL posteriores. Asimismo, el comando invocado es no bloqueante y constituye una transferencia de "BYTES_PER_TRANSFER" *bytes* del contenido almacenado en el *host* en la dirección de memoria "hst_filter_data". Dicho contenido se transfiere a la región de memoria de la FPGA asociada al objeto **cl::Buffer** "filter_data" con un *offset* dentro de esta región de 0 *bytes*.

```
err=q_in.enqueueWriteBuffer(filter_data, CL_FALSE, 0,
BYTES_PER_TRANSFER, hst_filter_data, nullptr, &(filterInEvs[0]));
```

Código 6: Utilización de la función cl::CommandQueue::enqueueWriteBuffer

6.5.2. Función enqueueTask

Esta función invoca la ejecución por parte del acelerador *hardware* de una determinada función asociada en el *host* a un objeto **cl::Kernel**. Este objeto **cl::Kernel** se introduce como referencia con el primer parámetro de la invocación de esta función. Opcionalmente, el usuario puede especificar dos parámetros adicionales denominados "events" y "event". Con respecto a la función **enqueueWriteBuffer**, el propósito del parámetro "events" en la función **enqueueTask** es análogo al del parámetro "wait_list". Por otra parte, la utilidad del parámetro "event" es idéntica a la del parámetro homónimo de la función **enqueueWriteBuffer** [42], [44], [49].

En el Código 7 se muestra un caso de utilización de la función **enqueueTask** dentro de la codificación del *host*. En dicho ejemplo, el objeto **cl::Kernel** a utilizar se corresponde a un *kernel* ejecutado en la FPGA para operaciones de filtrado de firmas espectrales y que se ubica dentro del *array* "krnls" que guarda todos los objetos **cl::Kernel** utilizados en la aplicación. Además, su ejecución se inicia tras terminar todos los comandos de OpenCL asociados a alguno de los eventos guardados dentro del vector "filterInEvs". Entre estos

últimos comandos se encuentra la invocación de **enqueueWriteBuffer** del Código 6. Para sincronizaciones posteriores, se le asigna a la invocación del Código 7 el primer evento del vector "filterExeEvs".

```
err = q_exe.enqueueTask(krnls[_FILTER_KRNL],
(cl::vector<cl::Event>*)&filterInEvs, filterExeEvs.data());
```

Código 7: Utilización de la función cl::CommandQueue::enqueueTask

6.5.3. Función enqueueReadBuffer

Esta función invoca un comando de lectura por parte del *host* de datos generados por la ejecución del acelerador *hardware*. La parametrización de esta función es idéntica a la de la función **enqueueWriteBuffer** anteriormente descrita. La única diferencia, en términos funcionales, entre las funciones **enqueueReadBuffer** y **enqueueWriteBuffer** se encuentra en qué región de memoria se escribe y lee como resultado de la consecución del comando invocado. En el caso de **enqueueReadBuffer**, la región de memoria a escribir es la del *host*, que es direccionada con el puntero "ptr". Por consiguiente, la región de memoria a leer corresponde a la del acelerador *hardware*, que es direccionada mediante el objeto **cl::Buffer** y el valor de "offset" correspondiente. Como puede observarse, esta situación es la inversa de la ocurrida con la función **enqueueWriteBuffer** [42], [49].

En el Código 8 se presenta el uso de esta función dentro de la aplicación del *host*. En dicha invocación se encola un comando no bloqueante para la transferencia hacia el *host* de "BYTES_PER_TRANSFER" *bytes* de datos almacenados en la región de memoria de la FPGA asociada al objeto **cl::Buffer** "filter_output" considerando un *offset* de 0 *bytes*. Dichos datos se guardan en el *host* considerando como punto de inicio en la región de memoria la dirección del dato del índice "frame_preproc_idx" dentro del *array* "filtered_image". En este *array* se guarda la totalidad de la imagen hiperespectral filtrada. Como establece el penúltimo parámetro de la invocación del Código 8, el comando de OpenCL asociado a esta invocación debe esperar a la ejecución del comando encolado tras la invocación del Código 7. A la invocación del Código 8 se le asigna el primer evento del vector "filterOutEvs".

```
err = q_out.enqueueReadBuffer(filter_output, CL_FALSE, 0,
BYTES_PER_TRANSFER, &(filtered_image[frame_preproc_idx]),
(cl::vector<cl::Event>*)&filterExeEvs, &(filterOutEvs[0]));
```

Código 8: Utilización de la función cl::CommandQueue::enqueueReadBuffer

6.5.4. Función finish.

Tras su invocación, esta función detiene la ejecución del *host* hasta que todos los comandos de la cola de comandos se finalicen [44], [49]. En el Código 9 se muestra el uso de esta función. En este caso, el propósito de la invocación consiste en detener la ejecución del *host* hasta que finalice la ejecución en la FPGA del *kernel* encolado mediante el objeto **cl::CommandQueue** "q_exe" tras la invocación del Código 7.

```
err=q_exe.finish();
```

Código 9: Utilización de la función cl::CommandQueue::finish

Otro aspecto característico con respecto a los comandos de OpenCL es que cada uno de ellos posee un valor numérico que identifica su tipo. En el caso del comando de lectura de *buffer*, introducidos por la función **enqueueReadBuffer**, el valor identificativo se asocia a la macro "CL_COMMAND_READ_BUFFER". En cambio, si el comando es de escritura del *buffer*, es decir, el especificado por la función **enqueueWriteBuffer**, el valor identificativo se encuentra asignado a la macro "CL_COMMAND_WRITE_BUFFER". Por último, los comandos de ejecución de funciones o *kernels* en un acelerador *hardware* introducidos por la función **enqueueTask** se identifican con el valor de la macro "CL_COMMAND_TASK" [42].

Como se indicó con anterioridad, las funciones **enqueueWriteBuffer** y **enqueueReadBuffer** encolan transferencias entre el *host* y el acelerador *hardware*, diferenciándose básicamente en la dirección de dicha transferencia. En la Figura 27 se expone gráficamente la relación de estas funciones con las transferencias que encola.

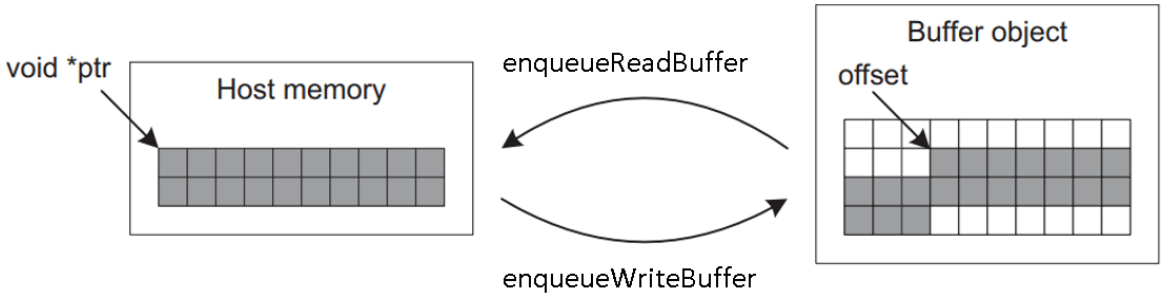


Figura 27: Transferencias mediante enqueueReadBuffer y enqueueWriteBuffer [42]

6.6. cl::Program

Esta clase hace referencia a la programación de los aceleradores *hardware*. En OpenCL, un programa constituye una agrupación de *kernels* que se encuentran en un mismo archivo binario de un mismo dispositivo. Este archivo binario se genera tras la compilación y se utiliza para programar el acelerador *hardware* [42], [44], [50]. Un objeto **cl::Program** puede contener múltiples binarios de múltiples aceleradores *hardware* diferentes [42]. En el caso de que el acelerador *hardware* sea una FPGA de Xilinx, el archivo binario de la FPGA será un archivo *bitstream* de extensión ".xclbin".

Con respecto a las compilaciones que generan los binarios para la programación de los aceleradores *hardware*, el estándar OpenCL ofrece dos alternativas diferentes. La primera opción se corresponde a efectuar la compilación durante la ejecución del *host*. Por otra parte, la segunda opción consiste en introducir los archivos binarios de los aceleradores *hardware* ya compilados con anterioridad [44],[50]. Esta segunda alternativa es la utilizada en este trabajo.

Para ambas opciones con respecto a la compilación, los *bytes* del contenido de cada archivo fuente y de cada binario deben transformarse en un *array* de "char". En cada caso, cada *array* debe agruparse con una variable que almacena la longitud en *bytes* que dicho *array* posee. De esta manera, se crea un objeto del tipo **std::pair<const char*, ::size_t>** para el caso de un archivo fuente, y del tipo **std::pair<const void*, ::size_t>** para el caso de un archivo binario.

Estos objetos se añaden como elementos de un vector que almacena el conjunto de archivos fuente o de archivos binarios a utilizar. En el caso de los binarios, el vector se define con el tipo de variable **Binaries** declarado dentro de **cl::Program**. Si se tratan de los

archivos fuentes, entonces se utiliza el tipo de variable **Sources**, cuya declaración también se ubica dentro de la clase **cl::Program** [42], [51].

Todas estas operaciones se programan en la codificación del *host* en la función "setBinary" para el caso de los archivos binarios. La codificación de esta función se muestra en el Código 10. Siguiendo su parametrización, esta función introduce el contenido del archivo binario de nombre "xclbinFileName" en uno de los **cl::Program::Binaries** contenidos por el *array* "bins". El objeto **cl::Program::Binaries** se indica mediante el índice "bin_idx".

```
inline void setBinary(const char* xclbinFilename, unsigned char
bin_idx){
    std::cout << "Leyendo " << xclbinFilename << std::endl;
    //-----Comprobacion xclbin existe
    FILE* fp = fopen(xclbinFilename, "r");
    if(fp == nullptr){
        printf("ERROR: Archivo %s no encontrado\n", xclbinFilename);
        exit(EXIT_FAILURE);
    }
    fclose(fp);
    //------------------------------
    std::cout << "Cargando '" << xclbinFilename << "'\n";
    std::ifstream bin_file(xclbinFilename, std::ifstream::binary);
    bin_file.seekg (0, bin_file.end);
    unsigned nb = bin_file.tellg();
    bin_file.seekg (0, bin_file.beg);
    buf[bin_idx] = new char[nb];
    bin_file.read(buf[bin_idx], nb);
    // Creating Program from Binary File
    bins[bin_idx].push_back({buf[bin_idx], nb});
}
```

Código 10: Función setBinary

Dentro de la cabecera del constructor del objeto **cl::Program**, el primer parámetro se corresponde al contexto **cl::Context**. Dicho contexto debe de ser común para todos los aceleradores *hardware* a programar con la instancia construida. En el caso de la programación con binarios precompilados, el siguiente parámetro constituye una referencia al vector de objetos **cl::Device** que guarda la colección de aceleradores *hardware* a programar. Posteriormente, el tercer parámetro representa una referencia al vector **cl::Program::Binaries** que almacena los binarios, existiendo una correspondencia unívoca entre los aceleradores *hardware* y los binarios considerados. Opcionalmente, puede especificarse el parámetro "binaryStatus", el cual constituye un puntero a un vector de variables "cl_int". Cada elemento de este vector notifica la carga con éxito de un

determinado binario en el acelerador *hardware* asignado [42]. En el Código 11 se presenta un ejemplo de la utilización del constructor de **cl::Program** dentro de la aplicación desarrollada. Al no utilizarse "BinaryStatus", dicho parámetro se especifica como puntero nulo.

```
program[_FILTER_BIN] = cl::Program(context, {device},
bins[_FILTER_BIN], nullptr, &err);
```

Código 11: Creación del objeto cl::Program

6.7. cl::Kernel

Esta clase permite identificar en la programación del *host* a los *kernels* que son ejecutados en los distintos aceleradores *hardware*. Un *kernel* representa una función que ejecuta el acelerador *hardware*. Para la construcción de esta clase, el usuario debe especificar dos parámetros. El primer parámetro se corresponde al programa **cl::Program** en el que se encuentra el *kernel* a invocar. Por otra parte, el segundo parámetro permite indicar el nombre de la función *kernel* a la que el objeto hace referencia [42], [44]. En el Código 12 se muestra un ejemplo de invocación del constructor de esta clase dentro de la codificación del *host* desarrollada. En dicha invocación, el objeto **cl::Program** considerado es el mismo que se especificó para el Código 11.

```
krnls[_FILTER_KRNL] = cl::Kernel(program[_FILTER_BIN], "multFilter",
&err);
```

Código 12: Creación de objeto cl::Kernel

Para este trabajo se utiliza solamente una función miembro de la clase **cl::Kernel** denominada "setArg". Esta función permite establecer el valor de los argumentos de entrada del *kernel* de manera previa a la invocación de su ejecución. Alternativamente, esta función permite asociar al argumento un objeto OpenCL utilizado como intermediario en la transferencia de ese argumento entre el *host* y el *kernel*. Este objeto puede ser del tipo **cl::Image** o **cl::Buffer** [42], [44], utilizándose este último.

Con respecto a la parametrización de la función "setArg", el primer parámetro se denomina "index". El cometido de este parámetro es especificar el parámetro de la función

kernel al que esta invocación de la función "setArg" afectará. Para ello, cada parámetro se identifica con un índice entre 0 y la cantidad de parámetros de la función *kernel* menos uno, incluyéndose ambos valores. El orden en la asignación de índice identificativo es de izquierda a derecha. En consecuencia, el índice 0 se corresponde al primer parámetro de la cabecera de la función *kernel*.

El segundo parámetro de "setArg" se utiliza para especificar desde el *host* el valor que adquiere dicho parámetro o el objeto de OpenCL destinado a gestionar las transferencias de ese parámetro entre el *host* y el *kernel* [42], [44]. Para este caso, se utiliza únicamente la segunda opción, como se muestra en el Código 13 para la totalidad de los parámetros del *kernel* "multFilter", cuyo objeto **cl::Kernel** fue creado en el Código 12.

```
err=krnls[_FILTER_KRNL].setArg(0, filter_data);
err=krnls[_FILTER_KRNL].setArg(1, filter_nBands);
err=krnls[_FILTER_KRNL].setArg(2, filter_lastFrame);
err=krnls[_FILTER_KRNL].setArg(3, filter_output);
err=krnls[_FILTER_KRNL].setArg(4, filter_min);
err=krnls[ FILTER KRNL].setArg(5, filter scale);
```

Código 13: Utilización de la función cl::Kernel::setArg

Los segundos parámetros en las invocaciones del Código 13 son objetos **cl::Buffer**. Las asociaciones de estos objetos con los parámetros del *kernel* "multFilter" se contemplan al comparar el Código 13 con el encabezado de la función principal de este *kernel*. Dicho encabezado se expone en el Código 14.

```
void multFilter(trans_unsized_d data[_TRANSFER_N_DATA],
                trans_band_id nBands[1],
                trans_n_filters_frame_id nFrames[1],
                trans_unsized_d output[_TRANSFER_N_DATA],
                trans_unsized_d min[_TRANSFER_N_DATA],
                trans_unsized_d scale[_TRANSFER_N_DATA])
```

Código 14: Cabecera de la función principal del kernel "multFilter"

6.8. cl::Buffer

Constituye una clase derivada de la clase **cl::Memory** de OpenCL. La clase **cl::Buffer** instancia objetos cuyo cometido es asociar una región de memoria del *host* con un parámetro del *kernel.* Este parámetro del *kernel* puede ser tanto uno que el *kernel* lee y cuyo contenido procede del *host* como uno que el *kernel* escribe para que sea recibido por

el *host* [42], [44]. Desde la perspectiva del *kernel*, cada parámetro se guarda en una región de memoria global dentro de la FPGA, utilizando bloques BRAM como recursos de memoria.

En la invocación del constructor de un objeto de la clase **cl::Buffer**, el primer parámetro es el contexto **cl::Context** a asociar al objeto. Este contexto debe coincidir con el asignado al programa y al acelerador *hardware* en el que se ubica el *kernel* para el cual se emplea este *buffer*. Posteriormente, se establece un parámetro del tipo "cl_men_flags" denominado "flags", que permite configurar algunas características relativas al *buffer* y a su utilización.

Las opciones de configuración vienen identificadas con un valor numérico asociado a una macro. Asimismo, la relación entre la macro asociada a ese valor y su implicación en la configuración de **cl::Buffer** es la expuesta en la Tabla 5. Pueden especificarse cualquiera de las tres primeras opciones combinadas con cualquiera de las tres últimas. Para ello, el valor de "flags" debe ser la suma aritmética o lógica de los valores de cada configuración deseada [42].

Tabla 5: Macros que identifican diferentes opciones de configuración de cl::Buffer [42]

Nombre de la macro	Significado
CL_MEM_READ_WRITE	El *buffer* se utiliza tanto para la lectura como para la escritura por parte del *host* de datos transferidos entre el *host* y el *kernel*.
CL_MEM_WRITE_ONLY	El *buffer* se utiliza solo para que el *kernel* pueda enviar datos generados al *host*.
CL_MEM_READ_ONLY	El *buffer* se utiliza únicamente para que el *kernel* lea datos enviados por el *host*.
CL_MEM_USE_HOST_PTR	El objeto de memoria accederá a la región de memoria especificada por el parámetro "host_ptr".
CL_MEM_COPY_HOST_PTR	El objeto de memoria establecerá la región de memoria especificada por el por el parámetro "host_ptr".
CL_MEM_ALLOC_HOST_PTR	Se asigna para la transferencia una región de memoria del *host*.

El tercer parámetro de **cl::Buffer** permite indicar la longitud en *bytes* del *buffer* utilizado en la transferencia entre el *host* y el *kernel*. El siguiente parámetro se corresponde a un puntero a "void" denominado "host_ptr", el cual indica la región de memoria a utilizar como *buffer* [42], [44].

La creación de objetos **cl::Buffer** se muestra en el Código 15 para el caso de los argumentos del *kernel* "multFilter". Al no utilizarse "host_ptr", este parámetro se especifica como puntero nulo en todas las invocaciones del constructor. Entre los argumentos del *kernel* se encuentran tanto datos de entrada al *kernel* como datos generados por el mismo y a transferir hacia el *host*. Por ello, los objetos **cl::Buffer** asignados a datos de entrada al *kernel* se configuran con el *flag* "CL_MEM_READ_ONLY", mientras que los correspondientes a datos generados por el *kernel* se establecen con el *flag* "CL_MEM_WRITE_ONLY".

```
const size_t DATA_ARRAY_BYTES = _N_FILTERS*sizeof(trans_unsized_d);

filter_data = cl::Buffer(context, CL_MEM_READ_ONLY, DATA_ARRAY_BYTES,
nullptr, &err);

filter_nBands = cl::Buffer(context, CL_MEM_READ_ONLY,
sizeof(trans_band_id), nullptr, &err);

filter_lastFrame = cl::Buffer(context, CL_MEM_READ_ONLY,
sizeof(trans_n_filters_frame_id), nullptr, &err);

//Creacion del buffer de datos de la imagen de salida
filter_output = cl::Buffer(context, CL_MEM_WRITE_ONLY,
DATA_ARRAY_BYTES, nullptr, &err);

//Creacion del buffer de minimos de los pixeles de salida
filter_min = cl::Buffer(context, CL_MEM_WRITE_ONLY, DATA_ARRAY_BYTES,
nullptr, &err);

filter_scale = cl::Buffer(context, CL_MEM_WRITE_ONLY,
DATA_ARRAY_BYTES, nullptr, &err);
```

Código 15: Creación de objetos cl::Buffer

6.9. cl::Event

Las instancias de esta clase se utilizan como elementos para gestionar aspectos temporales relativos a los comandos de OpenCL ordenados por el *host*. Para su utilización, cada instancia debe asignarse a una invocación de un comando de OpenCL diferente durante la ejecución del *host* [42].

En este caso, los objetos **cl::Event** se utilizan contenidos en vectores **cl::vector**. La especificación e introducción de un objeto **cl::Event** dentro de un vector **cl::vector** se presenta en el extracto del Código 16, donde "filterInEvs" constituye el vector que

almacena los eventos asociados a comandos de OpenCL de transferencia de datos desde el *host* hacia el *kernel* "multFilter". Con esta configuración, cada objeto **cl::Event** puede utilizarse para estos propósitos [42].

```
filterInEvs.push_back(cl::Event());
```

Código 16: Creación e introducción en vector de objeto cl::Event

6.9.1. Establecimiento de una función *callback*

Se declara una función en el *host* cuya ejecución comienza al finalizarse el comando de OpenCL al que se asocia el objeto **cl::Event**. Esta función recibe la designación de función *callback* y es asignada al objeto **cl::Event** que corresponda por medio de la invocación por parte de este último de la función **setCallback** declarada como función miembro dentro de **cl::Event**. En la parametrización de la función **setCallback**, el primer parámetro se denomina "type" y especifica el estado de ejecución del comando OpenCL asociado al objeto **cl::Event** que activará la ejecución en el *host* de la función *callback* asignada.

El siguiente parámetro constituye un puntero a la función *callback* establecida, mientras que el último se corresponde a un puntero a la región de memoria donde se especifican los parámetros de entrada que la función *callback* utilizará. Cabe destacar que todas las variables globales podrán utilizarse en la ejecución de los *callbacks*, aunque no sean pasadas a estos como parámetros.

En el Código 17 se muestra un caso de utilización presente en la codificación del *host* de la función **setCallBack**. Mediante dicho código, se establece que, cuando concluya la ejecución de la transferencia encolada por la invocación de **enqueueWriteBuffer**, se ejecute en el *host* la función "filterDataSent". Como no desea pasarse ningún parámetro a esta función, el tercer parámetro de **setCallBack** se especifica como puntero nulo. La asignación de *callback* a un evento solo puede efectuarse cuando dicho evento ha sido asignado previamente a algún comando de OpenCL.

```
err=q_in.enqueueWriteBuffer(filter_data, CL_FALSE, 0,
BYTES_PER_TRANSFER, hst_filter_data, nullptr, &(filterInEvs[0]));

err=filterInEvs[0].setCallback(CL_COMPLETE, &filterDataSent, nullptr);
```

Código 17: Utilización de la función cl::Event::setCallback

Si se compara el Código 17 con el Código 6, se observa que el objeto **cl::Event** considerado es el mismo. Asimismo, la ejecución del Código 17 dentro de la codificación del *host* establece que, tras la finalización del comando de OpenCL encolado al ejecutar el Código 6, se ejecute en el *host* la función "filterDataSent". Como la función "filterDataSent" no requiere parámetros de entrada para su ejecución, aunque sí para su especificación, el tercer parámetro en el Código 17 se especifica como puntero nulo. Todas las funciones *callback* asignadas a objetos OpenCL deben cumplir con el formato de cabecera expuesto en el Código 18, donde <función> debe reemplazarse por el nombre con el que se identificará en el código fuente al *callback* [42].

```
void CL_CALLBACK <función>(cl_event event, cl_int status, void*
data)
```

Código 18: Formato de cabecera de las funciones callback de OpenCL [42]

Como se observa en el Código 18, el primer parámetro se corresponde a un "cl_event" denominando "event". Este parámetro representa el atributo básico del objeto **cl::Event** para el cual se invoca el *callback*. Dicho atributo posee una utilidad idéntica a la del objeto **cl::Event** que lo contiene, aunque su utilización se restringe a funciones en C de OpenCL.

Debido al tipo del primer parámetro, dentro de las funciones *callback* se aplican exclusivamente funciones en C de OpenCL para obtener información del evento que invoca el *callback*. El siguiente parámetro es del tipo "cl_int" denominado "status", el cual notifica el estado de ejecución del comando OpenCL asociado al objeto **cl::Event** para el cual se establece esta función *callback*. Como último parámetro se encuentra un puntero "void*" denominado "data". Este parámetro puede aplicarse para el traspaso a las funciones *callback* de parámetros adicionales [42].

6.9.2. Sincronización

La utilización de objetos **cl::Event** permite establecer el orden en el que se ejecutan los comandos de OpenCL ordenados por el *host* incluso cuando este orden no coincida con el orden en que los comandos son encolados. Para configurar este orden se disponen de varias alternativas. Se utilizan los parámetros "wait_list" y "event" presentes en las funciones que encolan estos comandos. Cabe recordar que "wait_list" establece los

comandos OpenCL que el comando OpenCL objeto de la invocación debe esperar a que terminen de ejecutarse. Esta "wait_list" consiste en un puntero al vector donde se almacenan los objetos **cl::Event** asignados a los comandos OpenCL a esperar. Por otra parte, el parámetro "event" constituye un puntero a un objeto **cl::Event** asociado al comando OpenCL invocado. Este último evento puede encontrarse en la "wait_list" de otro comando OpenCL, ocasionando con ello que este tenga que esperar a la terminación del comando OpenCL actual [42].

Un ejemplo de esta sincronización se muestra en el Código 19. En dicho código se establece, mediante el objeto **cl::Event** contenido por el vector "filterExeEvs", que una transferencia de datos desde la FPGA hacia el *host* que encola la función **enqueueReadBuffer** solo puede comenzar cuando la ejecución de *kernel* que genera dichos datos termine su ejecución. A su vez, esta ejecución del *kernel*, encolada por la función **enqueueTask**, solo puede iniciarse cuando todos los comandos de OpenCL asociados a eventos del vector "filterInEvs" hayan concluido.

```
err = q_exe.enqueueTask(krnls[_FILTER_KRNL],
(cl::vector<cl::Event>*)&filterInEvs, filterExeEvs.data());

err = q_out.enqueueReadBuffer(filter_output, CL_FALSE, 0,
BYTES_PER_TRANSFER, &(preproc_image[frame_preproc_idx]),
(cl::vector<cl::Event>*)&filterExeEvs, &(filterOutEvs[0]));
```

Código 19: Sincronización mediante eventos de comandos de OpenCL

Por otra parte, los objetos **cl::Event** pueden utilizarse para sincronizar la ejecución de los comandos de OpenCL con la ejecución del *host*. Para esto último, puede utilizarse la función "waitForEvents" declarada como función estática dentro de la clase **cl::Event**. Al invocarse, esta función detiene la ejecución del *host* hasta que se terminen de ejecutar todos los comandos OpenCL que tengan asignados objetos **cl::Event** pertenecientes a la "wait_list" apuntada como parámetro por la invocación de "waitForEvents" [42].

6.9.3. Medición de tiempo

La asignación de un objeto **cl::Event** a un comando OpenCL permite determinar aspectos temporales de la ejecución de dicho comando. Para ello, el objeto **cl::Event** del comando OpenCL a evaluar debe invocar la función **getProfilingInfo**. Asimismo, esta función posee un parámetro de *template* que permite especificar qué información con

respecto a la temporización del comando OpenCL se desea obtener. Este parámetro dispone de cuatro opciones válidas, correspondiéndose cada una de ellas a un valor numérico identificativo. Cada uno de estos valores se asocia con una macro.

En la Tabla 6 se expone la correlación existente entre la macro especificada y la implicación que dicha macro posee en la información extraída [42].

Tabla 6: Macros que identifican instantes de tiempo de los comandos OpenCL [42]

Nombre de la macro	Significado
CL_PROFILING_COMMAND_QUEUED	Devuelve el instante de tiempo en el que se encoló el comando.
CL_PROFILING_COMMAND_SUBMIT	Devuelve el instante de tiempo en el que el comando es admitido por el acelerador *hardware* responsable de su ejecución.
CL_PROFILING_COMMAND_START	Devuelve el instante de tiempo en el que comienza la ejecución del comando.
CL_PROFILING_COMMAND_END	Devuelve el instante de tiempo en el que se finaliza la ejecución del comando.

Todos los datos obtenidos mediante la función **getProfilingInfo** se expresan en nanosegundos y poseen una extensión de 64 *bits*. Sin embargo, el instante de tiempo medido y el devuelto con cada una de estas opciones puede no coincidir exactamente con el real. Esta posibilidad ocurre debido a la inexactitud en la medición temporal que el acelerador *hardware* responsable de ejecutar el comando ocasiona por falta de resolución. Desde la perspectiva de la programación interna de la función **getProfilingInfo**, esta recurrirá a la invocación de la función **clGetEventProfilingInfo**. Esta última función de OpenCL está desarrollada para C y cumple con el mismo propósito que la función **getProfilingInfo**, aunque su parametrización se especifica de forma diferente [42].

En la codificación del *host* se usa la función **clGetEventProfilingInfo** dentro de las funciones *callback* para medir el tiempo trascurrido en la ejecución de la aceleración *hardware*. Para el caso del *callback* asignado en el Código 17, la utilización de la función **clGetEventProfilingInfo** se muestra en el Código 20. Con dicha invocación, se extrae el instante de inicio del comando de OpenCL al que se le asignó el evento "event". Esta información consiste en un dato numérico de 8 *bytes*, resultado de "sizeof(cl_ulong)", que se almacena en la variable "filterInit". En "sizeInitGot" se devuelve la cantidad de *bytes* que requiere el almacenamiento del dato temporal consultado, con el objetivo de evaluar posteriormente que coincide con "sizeof(cl_ulong)".

```
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
                        sizeof(cl_ulong), &filterInit,
                        &sizeInitGot);
```

Código 20: Utilización de la función clGetEventProfilingInfo

Al igual que otras clases de C++ para la utilización de OpenCL, la clase **cl::Event** dispone de una función miembro denominada **getInfo**. Dicha función posee un parámetro de *template* que permite especificar qué información relativa a la instancia de **cl::Event** se desea obtener por medio de un valor numérico identificativo. Cada uno de estos valores numéricos se encuentran asociados a una macro. Asimismo, la correlación entre la macro y la información a la que hace referencia es la expuesta en la Tabla 7 [42].

Tabla 7: Macros que identifican la información a extraer relativa al evento cl::Event [42]

Nombre de la macro	Significado
CL_EVENT_CONTEXT	Devuelve el contexto asociado al evento
CL_EVENT_COMMAND_QUEUE	Devuelve la cola de comandos asociada al evento
CL_EVENT_COMMAND_EXECUTION_STATUS	Devuelve el estado de ejecución en el que se encuentra el comando de OpenCL asociado al evento
CL_EVENT_COMMAND_TYPE	Informa sobre el tipo de comando de OpenCL asociado al evento
CL_EVENT_REFERENCE_COUNT	Indica la cantidad de veces que se ha asociado un comando OpenCL a este evento

6.10. Conclusiones

En este capítulo se han descrito los recursos de OpenCL a aplicar para gestionar desde la programación *software* del *host* la intercomunicación entre este y el acelerador *hardware*. En esta aplicación, el acelerador *hardware* se corresponde a una FPGA. La amplia mayoría de estos recursos de OpenCL se usan por medio de APIs de C++. Dichas APIs permiten un nivel de abstracción mayor en la invocación de operaciones mediante el estándar OpenCL que las funciones de C análogas. Sin embargo, la ejecución de las APIs de C++ se apoya en las funciones de OpenCL originales y desarrolladas para C. Por este motivo, estas funciones de C++ reciben la designación de *wrappers*.

Debido al modelo de programación orientado a objetos de C++, la utilización de los *wrappers* de C++ consiste en instanciar objetos de diferentes clases mediante los que se invocan funciones pertenecientes a dichas instancias. Cada clase constituye e identifica, en

el código fuente, un concepto funcional y abstracto del estándar OpenCL (**cl::Device**, **cl::CommandQueue**, **cl::Buffer**). Asimismo, cada función miembro invocada representa una operación en la que el elemento principal involucrado se corresponde al abstraído por el objeto OpenCL que lo invoca.

Una parte importante de la gestión de la interacción entre el *host* y el acelerador *hardware* consiste en la sincronización entre las diferentes órdenes o comandos que el *host* ordena al acelerador *hardware*. Para ello, se configuran los objetos **cl::CommandQueue** utilizados para introducir estos comandos en el acelerador *hardware* de manera que los comandos se ejecuten en el mismo orden en que estos son introducidos. Para la sincronización de comandos de OpenCL introducidos mediante diferentes objetos **cl::CommandQueue** se recurre a eventos de OpenCL instanciados como objetos **cl::Event**.

Por otra parte, debe sincronizarse la ejecución del *host* con la de las diversas funciones o *kernels* que ejecuta el acelerador *hardware*, así como con las transferencias entre *host* y el acelerador *hardware*. Para ello se dispone, entre otras opciones, de la función bloqueante **finish** de la clase **cl::CommandQueue**. Esta función detiene la ejecución del *host* hasta que todos los comandos introducidos al objeto **cl::CommandQueue** que la invoca se finalicen. Otro aspecto de la aplicación que se cubre mediante recursos de OpenCL es la medición de los tiempos de transferencia y ejecución. Para esto último, pueden utilizarse objetos **cl::Event**, ya que estos almacenan el tiempo de inicio y fin de los comandos de OpenCL a los que son asignados.

Capítulo 7. Metodología de diseño en Vitis

7.1. Introducción

En este capítulo, se describe la metodología seguida para el desarrollo de la aplicación de este trabajo. Dicha metodología está basada en el flujo de desarrollo de aceleradores para aplicaciones de Xilinx Vitis. El flujo de desarrollo diferencia el tipo de implementación (sistema empotrado o centro de datos), el tipo de lenguaje utilizado ya sea OpenCL, C/C++ o VHDL/Verilog, el nivel de descripción del IP (usando un lenguaje de alto nivel o a partir de un diseño RTL). En cada uno de estos escenarios, lo pasos a seguir estarán adaptados a las necesidades de diseño.

En este trabajo se explota la metodología de desarrollo de *kernels* de aceleración en alto nivel, usando Vitis HLS a partir de un modelo *untimed*, con el desarrollo del *host* en la CPU basado en OpenCL, tal como se explicó en el capítulo anterior. Para explicar este procedimiento, se describe el flujo de diseño general para el desarrollo de la aplicación. Dicho flujo incluye distintos tipos de emulación para la verificación progresiva del diseño de la aplicación de manera previa a su implementación en la FPGA.

La metodología explicada incluye la conversión de la aplicación que se ejecuta en el *host* o anfitrión en un archivo ejecutable para su ejecución en el microprocesador al que se orienta su desarrollo. Igualmente se presenta el flujo de diseño seguido por el compilador para establecer la configuración de la FPGA donde es necesario realizar la síntesis de alto nivel, controlada mediante directivas (pragmas en este caso) y restricciones de diseño: parte de los archivos fuente que describen los *kernel*s de aceleración *hardware* que la FPGA

debe implementar, se especifican las directivas de optimización mediante pragmas (**#pragma HLS**) utilizadas para dirigir la compilación de los *kernels* FPGA.

Para finalizar este capítulo dedicado a los aspectos metodológicos se explica el proceso de preparación de la plataforma de cara al prototipado final. Esto incluye el procedimiento y archivos para el arranque, depuración y validación de la plataforma final sobre la placa de prototipado.

7.2. Flujo de diseño

En el Capítulo 6 se han introducidos los conceptos básicos sobre OpenCL, lenguaje de referencia para el desarrollo de la aplicación del *host*. Igualmente se han presentado los mecanismos de comunicación basados en XRT, en este caso para aplicaciones empotradas.

En este apartado se presenta el flujo general seguido para desarrollar la aplicación, tanto a nivel de *host* como a nivel de acelerador o dispositivo OpenCL.

Durante el desarrollo de la aplicación, ha sido necesario recurrir a procedimientos de verificación con el objetivo de evaluar y depurar su correcto funcionamiento, además de sus prestaciones. Estos procedimientos se han efectuado, en primer lugar, para secciones consecutivas de la aplicación y, en última instancia, para la aplicación en su conjunto.

La verificación de la aplicación incluye dos tipos de emulaciones soportados en la metodología de Vitis. Es necesario completar los distintos tipos de emulación para poder ejecutar la aplicación en la placa de prototipado. De esta manera, la metodología de diseño de la aplicación se muestra en la Figura 28 y se detalla en la Figura 29.

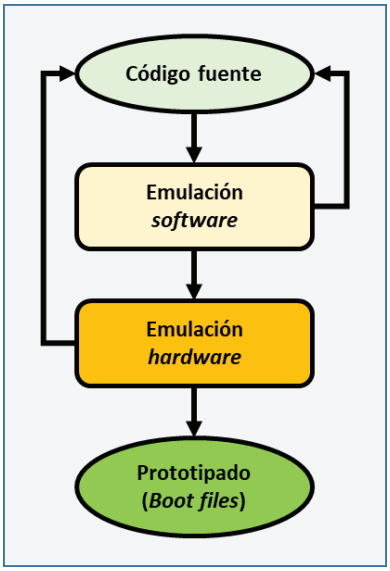


Figura 28. Flujo principal de diseño (adaptado de [45]*)*

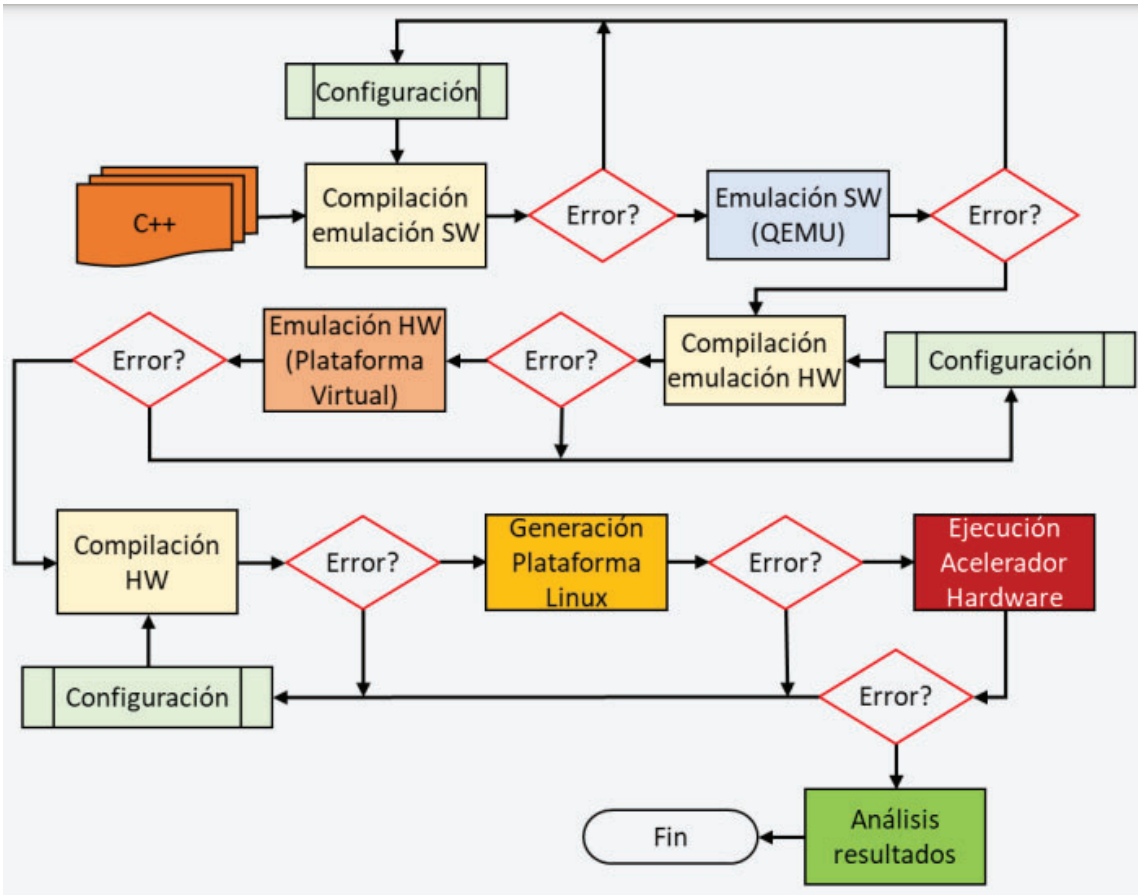


Figura 29: Metodología de diseño de la aplicación [14]

Como se muestra en la Figura 29, las emulaciones aplicadas se denominan emulación *software* y emulación *hardware*.

Ambos procesos de emulación se subdividen en dos etapas. En la primera etapa, se compila la aplicación para el modo de emulación considerado. En la etapa siguiente, se emplea el emulador QEMU para ejecutar la aplicación según ese mismo modo de emulación. Cada opción de emulación posee las siguientes características [14], [52].

Emulación *software*. Esta opción de emulación permite depurar el código del *host* y de los *kernels*, verificando que está libre de errores y que el algoritmo es funcionalmente correcto. Consiste en interpretar que tanto el código del *host* como la descripción HLS de los *kernels* FPGA van a ejecutarse como código *software* en su implementación. Este hecho posibilita la compilación rápida del código fuente de la aplicación. Sin embargo, esta emulación solo permite verificar el algoritmo utilizado en los *kernels* FPGA. Debido a ello, esta opción no permite evaluar la viabilidad de implementación *hardware* de estos *kernels*, así como la frecuencia y utilización de recursos [14], [52].

Desde la perspectiva *hardware*, esta emulación se ejecuta realmente en un microprocesador x86 si esta se implementa en plataformas “Data Center”. No obstante, para el caso de un procesador empotrado será un ARM Cortex A9 o A53. Este último caso será el considerado en este TFM. Por ello, la ejecución de la emulación *software* deberá emular el microprocesador ARM a utilizar en la implementación final de la aplicación [14], [52].

Con respecto al paralelismo, cabe destacar que cada réplica de los *kernels* FPGA que se utilice en la aplicación se ejecutará en la emulación *software* como un hilo de ejecución independiente. Este aspecto permitirá imitar la utilización de múltiples *kernels* FPGA en paralelo que puede darse en la implementación final de la aplicación. Sin embargo, la ejecución dentro de cada uno de estos hilos se efectúa secuencialmente. Debido a ello, la emulación *software* no puede imitar el paralelismo interno de los *kernels* FPGA que ocurre en su implementación *hardware* [14], [52].

Emulación *hardware*. En esta opción, la compilación del código del *host* es análoga a la observada en la emulación *software*. Sin embargo, la descripción de los *kernels* FPGA se transforma, tras la compilación, en un modelo de comportamiento RTL ejecutado sobre un simulador *hardware*, siendo la herramienta Vivado la opción por defecto. Este cambio implica un aumento significativo del tiempo de compilación de la aplicación. No obstante,

este tipo de emulación ofrece la ventaja de poder evaluar aspectos relativos a la implementación *hardware* en FPGA de los *kernels* de la aplicación, ya que posibilita una visión *cycle-accurate* de la ejecución. Además, este modo de emulación permite obtener estimaciones tanto de aspectos temporales como de utilización de recursos de la lógica programable (PL) por parte de los *kernels* FPGA [14], [52].

Para ejecutar las emulaciones de la aplicación debe arrancarse, tras completar las correspondientes compilaciones, el emulador QEMU. Desde la interfaz gráfica de Vitis, dicho emulador puede invocarse mediante la opción "Configure Emulator". Al seleccionarse esta opción, se abre la ventana expuesta en la Figura 30, a partir de la cual puede ordenarse el arranque de QEMU al presionarse el botón "Start". Este emulador debe desactivarse tras terminar la emulación presionando el botón "Stop" de la misma ventana [53].

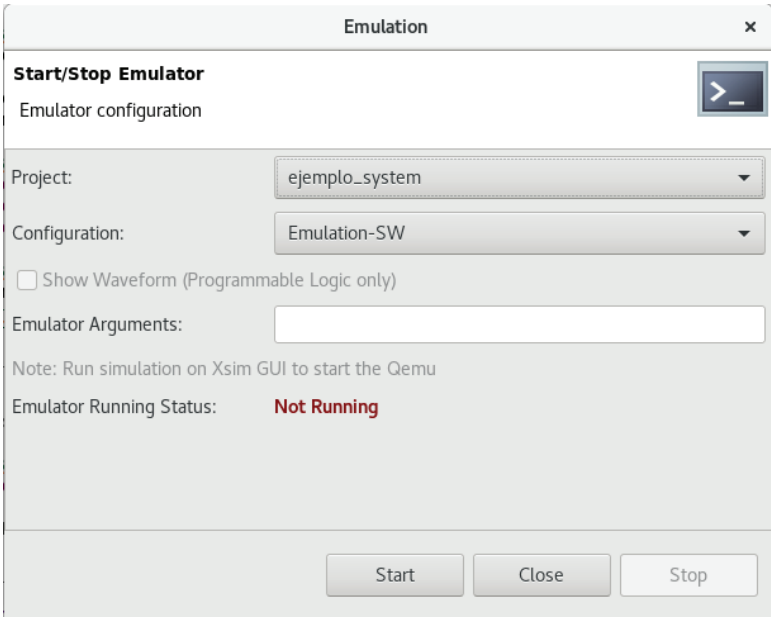


Figura 30: Arranque del emulador QEMU

Como parte del entorno de emulación, el emulador QEMU utiliza su propio sistema de archivos, análogo al que se emplea la placa de prototipado. Este este sistema de archivos se define en un archivo conocido como "Root FS" y de tipo "ext4", el cual debe introducirse en el flujo de diseño de la aplicación. Dicho archivo puede indicarse desde la interfaz gráfica de Vitis. Para ello, desde el proyecto de Vitis que contiene el código fuente de la aplicación,

al archivo “<nombre del proyecto>_system.sprj” y especificarse el archivo “Root FS” dentro de la opción homónima. Esta opción se señala en la Figura 31.

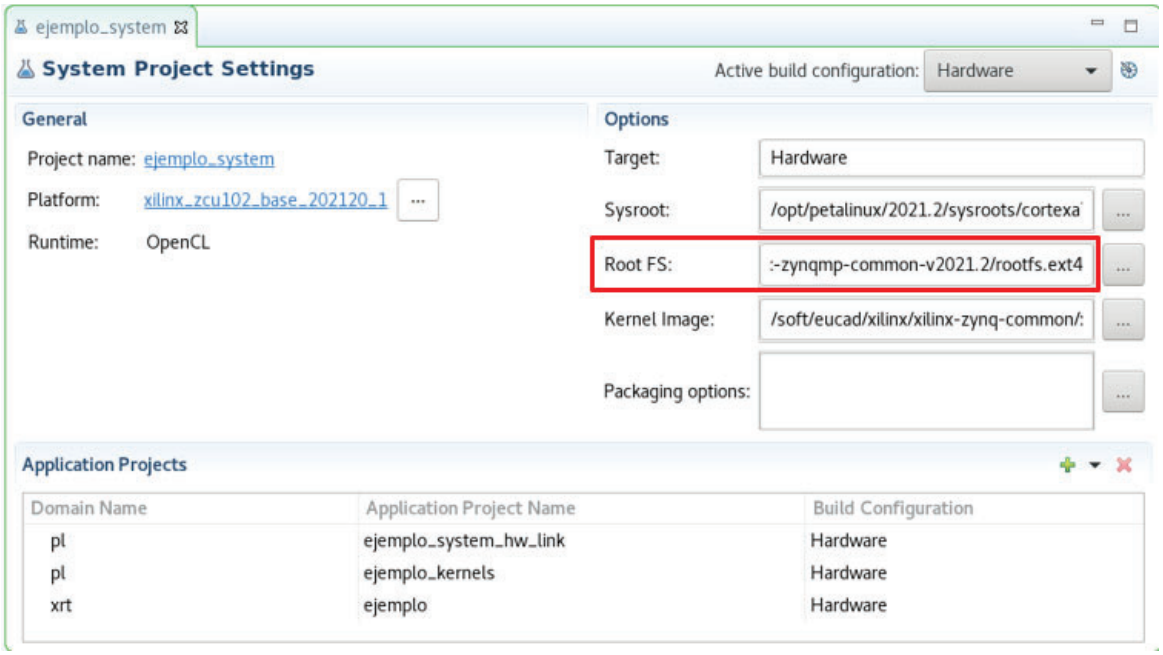


Figura 31: Especificación del Root FS

Durante el proceso de emulación puede ser necesario transferir ficheros desde el sistema de archivos del PC hacia el del QEMU y viceversa [53]. Ambos tipos de transferencias pueden efectuarse por medio de un agente TCF (*Target Communications Framework*) [53], [54]. Para ello, debe invocarse la conexión de un agente TCF desde la ventana de comandos XSCT (*Xilinx Software Command-line Tool*) provista por Vitis IDE [53], [55]. El comando utilizado en esta invocación se muestra en el Código 21.

```
connect –host 127.0.0.1 –port 1440
```

Código 21: Invocación de conexión del agente TCF [53]

Una vez se establece la conexión con el agente TCF, pueden invocarse desde la ventana de comandos XSCT las transferencias de archivos entre el PC y el QEMU. Para ello, debe utilizarse el comando expuesto en el Código 22 si el fichero ha de transferirse desde el sistema de archivos del PC hacia el QEMU. Este es el caso, por ejemplo, de los ficheros “.dat” que almacenan una imagen hiperespectral que la aplicación debe evaluar. Por otra parte, si el fichero debe transferirse desde el sistema de archivos de QEMU hacia afuera, debe invocarse el comando expuesto en el Código 23. Tanto en el Código 22 como en el

Código 23, "<host_path>" hace referencia al directorio dentro del sistema de archivos del PC, mientras que "<target_path>" se refiere al directorio dentro del sistema de archivos que establece QEMU [53].

```
tfile copy –from-host <host_path> <target_path>
```

Código 22: Transferencia de archivos desde el PC hacia el QEMU [53]

```
tfile copy –to-host <target_path> <host_path>
```

Código 23: Transferencia de archivos desde QEMU hacia el PC [53]

Tras verificar con éxito el funcionamiento de la aplicación mediante las emulaciones correspondientes, se procede a su implementación en la placa de prototipado. Al igual que las emulaciones, este proceso requiere de una compilación previa a la ejecución. De acuerdo con la nomenclatura de Xilinx, esta compilación se denomina "compilación *hardware*" y, coherentemente, la consecuente ejecución en placa se identifica como "ejecución *hardware*" [14], [52].

Desde el punto de vista de la optimización, es posible realizar tanto la optimización de la aplicación en el *host* como en los *kernels.* Las distintas optimizaciones realizadas en el *host* se centran en la planificación de la ejecución de los *kernels* (Figura 32) y en la optimización del movimiento de datos (Figura 33).

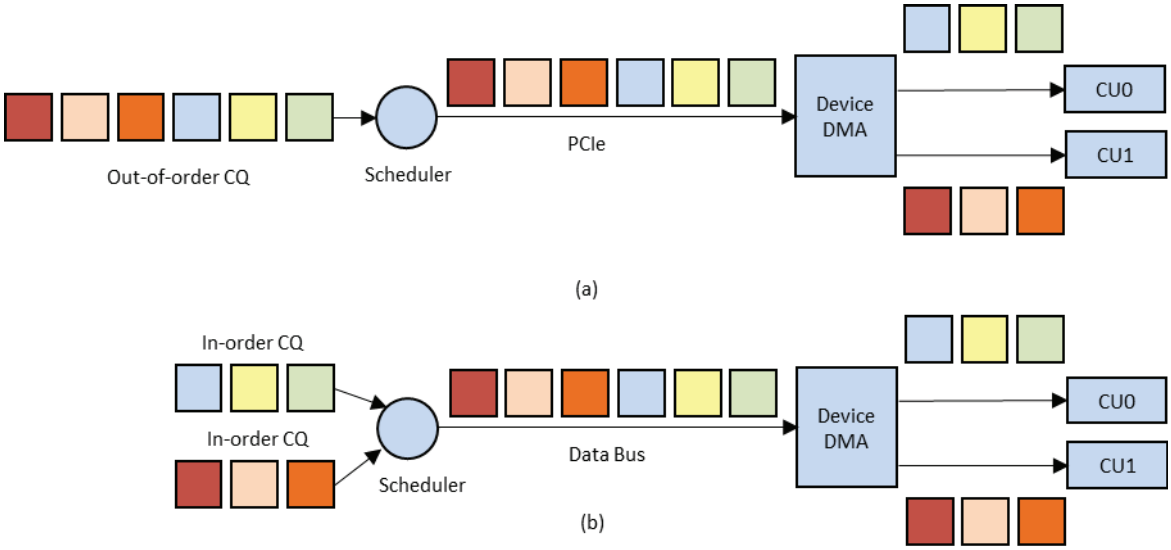


Figura 32. Planificación de las colas de procesamiento. (a) Planificación fuera de orden de una cola de comandos vs. (b) planificación en orden en varias colas de comandos [45]

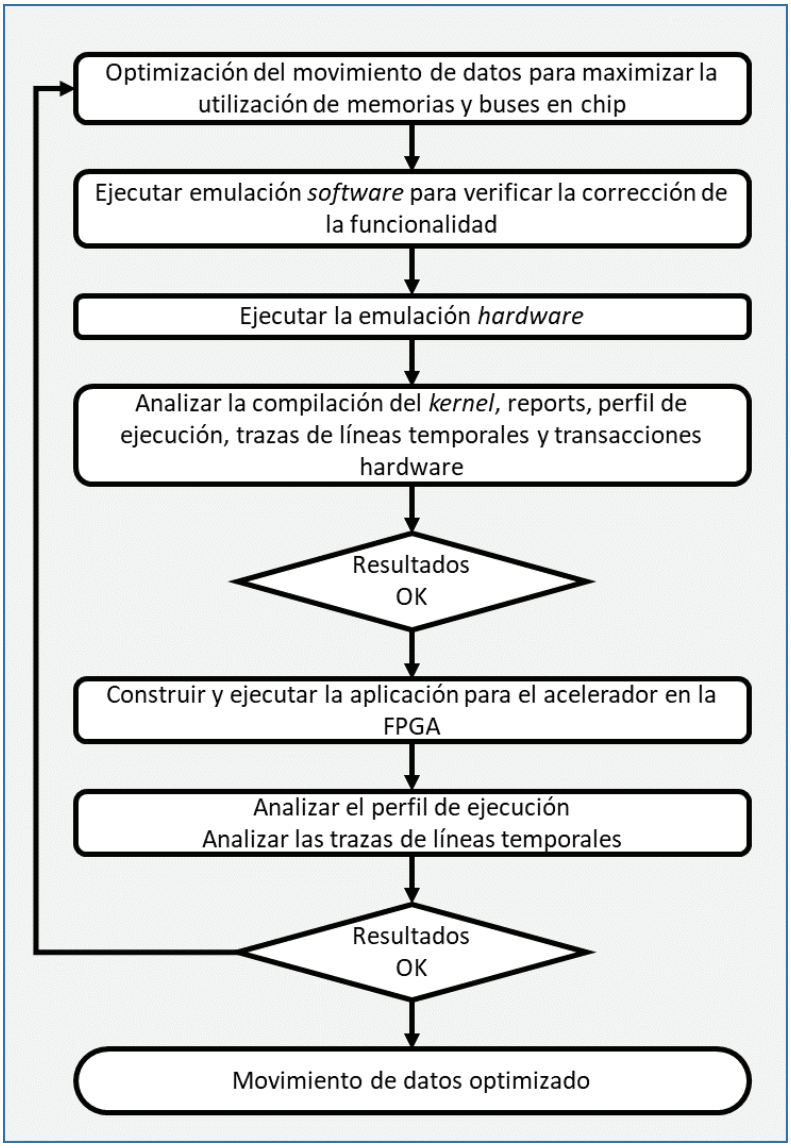


Figura 33. Proceso de optimización en el host [45]

7.3. Diseño del *host*

La codificación del *host* en esta aplicación se desarrolla en C/C++, utiliza las librerías de OpenCL para controlar las interacciones del *host* con los *kernels* FPGA. Para el caso de las aplicaciones "Data Center", la codificación del *host* se compila mediante GNU (*GNU's Not Unix*) para C++ (g++). Sin embargo, para las aplicaciones empotradas, se utiliza un compilador cruzado de GNU orientado a microprocesadores ARM para arquitecturas de 64 *bits* (aarch64-linux-gnu-g++) [52].

En la primera etapa de la compilación del *host*, los archivos de C/C++ que constituyen el código fuente se compilan convirtiéndose cada uno en un archivo objeto de

extensión ".o" (opción -c del compilador). Posteriormente, los archivos ".o" se enlazan constituyendo el ejecutable del *host*, ejecutable tanto para QEMU como para el *host*. Esta segunda etapa se efectúa al especificar la opción -l [52].

Las interacciones del *host* con los aceleradores *hardware* pueden programarse recurriendo a la API de XRT (*Xilinx Runtime*) o a la API de OpenCL. En ambos casos, debe especificarse la librería correspondiente durante la etapa de enlazado de la codificación del *host*. Estas librerías se denominan xrt_coreutil, para el caso de XRT, y OpenCL, para el caso de la utilización de APIs OpenCL. La aplicación de este trabajo utiliza la API de OpenCL como capa de aplicación, aunque finalmente utiliza XRT tal como se explicó anteriormente, por lo que únicamente será necesario incluir las librerías OpenCL [44], [52].

La depuración de la codificación del *host* se efectúa utilizando GDB. Con este depurador, puede ajustarse la codificación del *host* recurriéndose a un rediseño y depuración reiterativos, siguiendo con ello el diagrama expuesto en la Figura 34 [56].

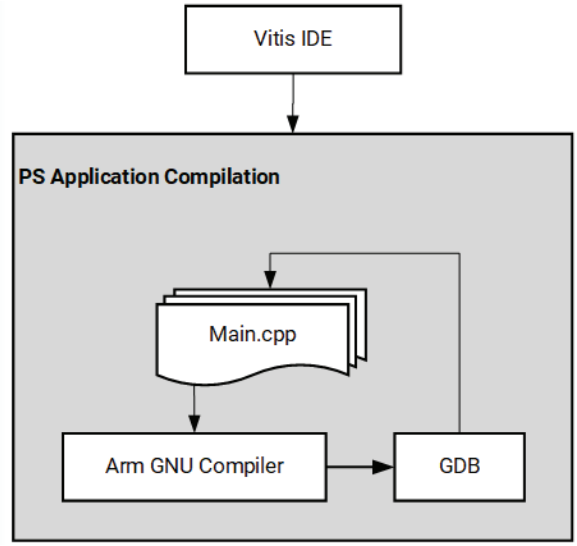


Figura 34: Diseño y depuración de la codificación del host [56]

7.4. Diseño del kernel

Durante la compilación de la aplicación, la codificación de los *kernels* FPGA se transforma en un archivo binario de extensión ".xclbin", recurriendo para ello al compilador "v++" que a su vez utiliza Vivado. Este archivo puede cargarse directamente en

la FPGA, configurando con ello a dicho dispositivo para la ejecución de los *kernels* FPGA agrupados dentro de este archivo.

7.4.1. Flujo de diseño

En la Figura 35 se expone gráficamente el proceso de compilación y agrupación de los *kernels* FPGA dentro de un archivo binario [14], [52].

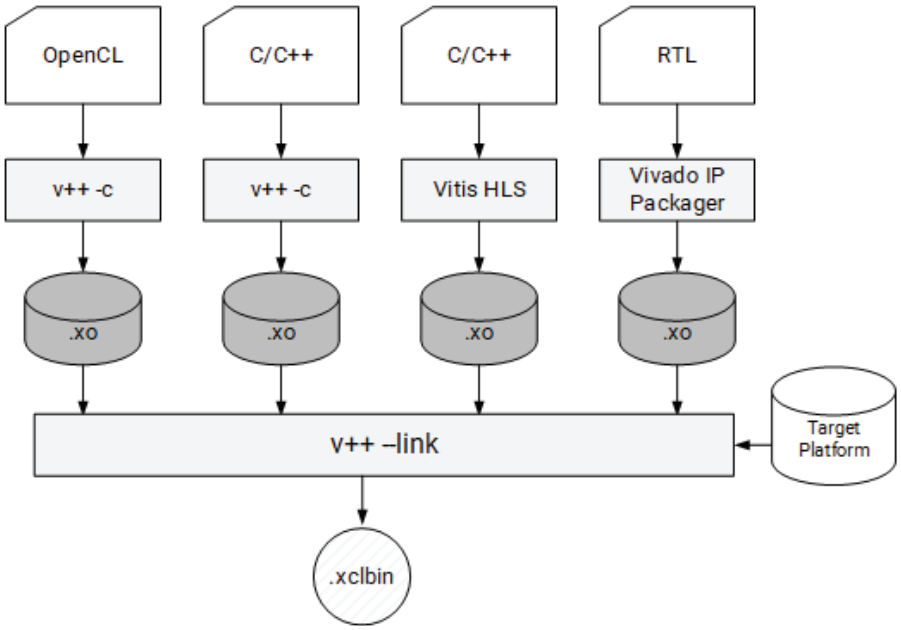


Figura 35: Generación de archivos binarios de la FPGA [52]

Como se observa en la Figura 35, la codificación de cada *kernel* FPGA puede describirse en C/C++, OpenCL o como un diseño RTL, utilizándose para el último caso un lenguaje de descripción *hardware* (VHDL o Verilog). En este caso, se parte de una descripción algorítmica sin información temporal (*untimed*) en C/C++ adaptada para la realización de la síntesis de alto nivel en Vitis HLS. Utilizando el compilador correspondiente, la codificación de cada *kernel* FPGA se convierte en un archivo Xilinx Object (".xo"). Estos archivos ".xo" son enlazados por la instrucción "v++ --link" que, recurriendo a la información acerca de la plataforma donde se implementará la aplicación "Target Platform", genera el binario de extensión ".xclbin" [14] [52].

Para obtener la arquitectura *hardware* de los *kernels* FPGA se ha optado por la metodología de síntesis de alto nivel (HLS). Dicha metodología consiste fundamentalmente en especificar el diseño de los *kernels* FPGA desarrollando una descripción funcional de los mismos mediante el lenguaje de programación C/C++. Sin embargo, en ausencia de

recursos de especificación adicionales, se permite al compilador Vitis HLS [57] tomar decisiones sobre la implementación *hardware* del *kernel* FPGA para cumplir con los requisitos funcionales y de prestaciones.

Hay dos aspectos que configuran la solución final. En primer lugar, la utilización de tipos de datos optimizados para precisión arbitraria. Además de los tipos nativos de C/C++, se aportan un conjunto de tipos de datos de precisión arbitraria (ap_*) tanto para enteros como para punto fijo con objeto de optimizar la implementación del diseño en cuanto a frecuencia ya a la utilización de recursos.

El mecanismo de control de distintos aspectos arquitecturales y de requisitos está basado en la utilización de directivas (pragmas) y de restricciones de diseño (frecuencia de reloj, dispositivo, …). Estas directivas y restricciones serán interpretadas por el compilador Vitis HLS.

7.4.2. Directivas de optimización

A continuación, se explican las principales directivas de optimización utilizadas.

7.4.2.1 HLS interface

Para el desarrollo de los *kernels*, debido al modelo de memoria utilizada es necesario soportar dos tipos de transferencia:

- Envío de datos mediante punteros en memoria global, lo que implica un bus AXI4_MM.
- Datos escalares que se escriben o leen en los registros del *kernel* (AXI_LITE)

Vitis HLS soporta transferencias de hasta 512 *bits* de ancho de palabra con objeto de aumentar las prestaciones finales.

Mediante la directiva **#pragma HLS interface** insertada en el código se permite especificar y configurar las interfaces, puerto físico y protocolo que se asigna a cada parámetro de entrada/salida del *kernel* FPGA para su implementación *hardware*. La declaración de esta directiva permite la especificación de múltiples parámetros [57], [58], [59], [60]:

- Parámetro “mode”. Indica el protocolo de intercomunicación entre el *host* y el *kernel* FPGA que se establecerá para un determinado parámetro del *kernel*. Además, permite declarar los mecanismos de control que se utilizarán en la transferencia de los datos correspondientes al parámetro del *kernel*. En esta aplicación se utilizará la opción “m_axi” [57], [58], [60].
- Parámetro “port”. Indica el objeto del *kernel* FPGA al que se asocia esta directiva, especificando para ello el nombre que el parámetro posee en el código fuente [57], [58], [60].
- Parámetro “bundle”. Esta opción permite regular que varios objetos del *kernel* a los que se les asigna un mismo protocolo para su transmisión y/o recepción, compartan o no un mismo puerto de interfaz AXI. Su aplicación consiste en especificar como “bundle” un nombre determinado para la interfaz, que identifique a un puerto físico. De esta manera, todos los parámetros del *kernel* que coincidan en el nombre especificado como “bundle” utilizarán el mismo puerto físico para su transmisión/recepción en la implementación *hardware* del *kernel* FPGA. En el Código 24, se presentan varios casos de utilización de “bundle” y en la Figura 36 cómo esto afecta a la identificación y asignación de puertos tal y como se observa en la herramienta de análisis [57], [58], [60].

```
#pragma HLS INTERFACE m_axi port=data bundle=gmem
#pragma HLS INTERFACE m_axi port=output bundle=gmem

#pragma HLS INTERFACE m_axi port=min bundle=gmem_min
#pragma HLS INTERFACE m_axi port=scale bundle=gmem_scale

#pragma HLS INTERFACE m_axi port=nBands bundle=gmem_info
#pragma HLS INTERFACE m_axi port=nFrames bundle=gmem_info
```

Código 24: Utilización del parámetro “bundle” de la directiva “#pragma HLS interface”

M_AXI

Interface	Data Width (SW->HW)	Address Width	Latency	Offset
m_axi_gmem	16 -> 256	64	64	slave
m_axi_gmem_info	8 -> 16	64	64	slave
m_axi_gmem_min	16 -> 256	64	64	slave
m_axi_gmem_scale	16 -> 256	64	64	slave

Figura 36: Efecto del parámetro “bundle” en los puertos de entrada salida del kernel

- Parámetro “max_read_outstanding”. Este parámetro se utiliza para la configuración del modo *burst* en las interfaces AXI4 identificadas como “m_axi” en la directiva. Su propósito consiste en configurar la cantidad de transferencias de datos de la interfaz física que deben leerse seguidamente en el *kernel* FPGA en modo *burst* [57], [58].
- Parámetro “max_write_outstanding”. Su utilidad es similar a la del parámetro “max_read_outstanding”, utilizándose para configurar la cantidad de transferencias de datos de la interfaz física que deben escribirse seguidamente desde el *kernel* FPGA en modo *burst* [57], [58], [60].

7.4.2.2 HLS unroll

El desenrollado de bucles es una técnica de optimización del *kernel* que permite aumentar su paralelismo. Los bucles pueden desenrollarse total o parcialmente. La dependencia de datos en los bucles impactará en las prestaciones finales. En la Figura 37 se ejemplifica el efecto de esta directiva.

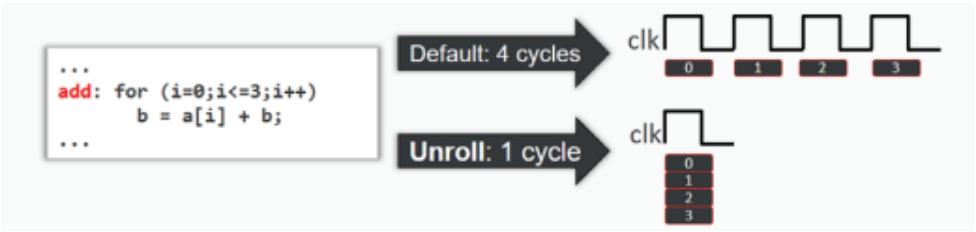


Figura 37. Ejemplo de utilización de HLS unroll [45]

Esta directiva se utiliza dentro de la descripción funcional de los bucles. La directiva **#pragma HLS unroll** permite controlar el paralelismo de la implementación del bucle, estableciendo que la misma pueda componerse de múltiples instancias del cuerpo del bucle. Además, estos diseños serán capaces de operar de manera plenamente simultánea entre sí [57], [58], [59], [60].

Con ello, se consigue paralelizar la ejecución de las iteraciones de los bucles, minimizando el retardo temporal a costa de aumentar la utilización de recursos de lógica programable. El impacto sobre la utilización de recursos puede ser significativo en función del número de operaciones del bucle y del número de iteraciones a desenrollar. Para el uso

adecuado de esta directiva debe tenerse en cuenta las siguientes categorías de bucles que pueden existir en la descripción HLS [57], [58], [59], [60]:

- Bucles con cantidad de iteraciones fijas. Son aquellos bucles para los cuales el compilador HLS puede determinar, durante la compilación, cuántas iteraciones del bucle se ejecutarán cuando se invoque la ejecución. Este hecho implica que dicha cantidad de iteraciones se mantiene con independencia de la ejecución del *kernel* FPGA y de los parámetros de entrada de dicha ejecución [57], [58], [60].
- Bucles con cantidad de iteraciones variables. Son aquellos en los que la cantidad de iteraciones del bucle no pueden ser determinados por el compilador sin garantías de que dicha cantidad no varíe para ninguna de las ejecuciones del bucle tras la compilación. Este hecho suele deberse a la dependencia que la cantidad de iteraciones del bucle puede poseer con respecto a los parámetros de entrada de la ejecución del *kernel* [57], [58], [60].

En ausencia de parámetros adicionales, el compilador interpreta que debe efectuar un "desenrollado completo" (o *full unrolling*) del bucle objeto de esta directiva. Este desenrollado implica que se implementarán tantas instancias RTL capaces de operar de acuerdo con el código fuente interno del bucle como cantidad de iteraciones requiera dicho bucle. En consecuencia, cada una de estas instancias ejecutará una única iteración de bucle, y todas las iteraciones se ejecutarán concurrentemente. Este desenrollado completo requiere que el bucle posea una cantidad de iteraciones fijas, y que así sea identificado por el compilador HLS [57], [58], [59], [60].

La sintaxis de la directiva **#pragma HLS unroll** permite especificar, opcionalmente, una serie de parámetros de acuerdo con la sintaxis expuesta en el Código 25 [57], [58], [60].

```
#pragma HLS unroll factor=<N> skip_exit_check
```

Código 25: Sintaxis de la directiva "#pragma HLS unroll" [57], [60]

Los parámetros presentes en esta directiva son los siguientes:

- El parámetro "factor=<N>" permite especificar cuántas instancias del bucle se implementarán en paralelo y serán capaces de ejecutar el código fuente interno

del bucle, siendo "N" dicha cantidad. Este parámetro facilita el establecimiento de un "desenrollado parcial" (o *partial unrolling*) del bucle. Este desenrollado parcial implica que la cantidad de instancias RTL no necesariamente coincide con la cantidad de iteraciones del bucle. En consecuencia, se paralelizará la ejecución de las iteraciones del bucle, pero seguirá existiendo dicha secuencialidad en la ejecución de estas iteraciones. El desenrollado parcial es factible tanto si la cantidad de iteraciones del bucle es fija como variable [57], [58], [59], [60].

- El parámetro "skip_exit_check" se utiliza para eliminar definir la evaluación de la salida d ejecución de las instancias RTL del bucle. Su uso es útil únicamente en el caso de desenrollado parcial. La cantidad de instancias especificadas en un desenrollado parcial no necesariamente debe ser divisor de la cantidad de iteraciones del bucle. Sin embargo, si se incumple dicho aspecto, ello implica que cada instancia debe implementar la comprobación de la condición de salida de manera individualizada. Esto se debe a que, para las últimas iteraciones del bucle, se requiere activar únicamente la ejecución de algunas de las instancias implementadas, pero no la totalidad. Por defecto, esta funcionalidad de comprobación se implementa de manera particularizada para cada instancia. No obstante, la especificación de "skip_exit_check" anula su implementación, lo cual puede ser conveniente para minimizar la utilización de recursos de lógica programable siempre que dicha funcionalidad adicional no se requiera. Asimismo, al aplicar esta opción será responsabilidad de programador asegurar que, para cualquier ejecución del bucle, la cantidad de iteraciones sea múltiplo de la cantidad de instancias RTL del cuerpo del bucle. Esta opción puede aplicarse tanto en bucles con cantidad de iteraciones fijas como variables [57], [58], [60].

7.4.2.3 HLS array_partition

Normalmente, los *arrays* especificados dentro de la descripción HLS de los *kernels* FPGA se implementan en la lógica programable en un solo bloque de memoria. Esta situación puede limitar los accesos simultáneos de lectura y escritura de los distintos datos

del *array* a la cantidad de puertos de lectura y escritura que este bloque de memoria disponga físicamente. Este hecho restringe el procesamiento en paralelo de los datos de un mismo *array* [57], [58], [59].

Para resolver esta situación, se dispone de la directiva **#pragma HLS array_partition**, que permite configurar el "particionado" del *array*. Dicho particionado consiste en subdividir el *array* en varios *arrays* de menor tamaño o, en el caso de la partición completa, en sus elementos individuales.

Desde la perspectiva *hardware*, cada una de estas subdivisiones o particiones se implementan en un bloque de memoria diferente, con puertos de acceso independiente. En consecuencia, se incrementa la utilización de bloques de memoria de la lógica programable. Sin embargo, dicho incremento posibilita el acceso simultáneo a las distintas particiones, facilitando el paralelismo en el acceso a los datos del *array*. Esta directiva puede asignarse a *arrays* unidimensionales y multidimensionales. Su sintaxis se presenta en el Código 26 [57], [58], [59], [60].

```
#pragma HLS array_partition variable=<name> \
type=<type> factor=<int> dim=<int>
```

Código 26: Sintaxis de la directiva "#pragma HLS array_partition" [57]

Esta directiva soporta los siguientes parámetros:

- Parámetro "variable". Identifica al *array* al que se le asigna, siendo "name" el nombre de dicho *array* en la descripción HLS.
- El parámetro "type" especifica el tipo de particionado efectuado, pudiendo ser exclusivamente alguna de estas tres opciones [57], [58], [60] (Figura 38).
    - "cyclic". En este tipo de particionado, la asignación de bloque de memoria para los elementos del *array* se efectúa de forma entrelazada. Asimismo, si se efectúa un particionado del *array* en 3 bloques de memoria, el primer elemento del *array* que se les asigne almacenar a cada bloque serán los elementos de índice 0, 1 y 2 respectivamente y si existen. Posteriormente, se asignaría como segundo elemento de estos bloques los datos del índice 3, 4 y 5 del *array* si los hubiere. Este tipo de

particionado es el idóneo cuando las exigencias de paralelismo del *kernel* FPGA requieren únicamente el acceso simultáneo de una cantidad definida de datos del *array* ubicados en índices contiguos al menor coste posible en utilización de recursos de memoria de la FPGA.

- "block". Este particionado subdivide el *array* en agrupaciones de elementos de índices contiguos, asignando a cada agrupación un bloque de memoria distinto dentro de la implementación del *kernel* FPGA.
- "complete". Constituye el particionado completo. Este particionado implica que a cada elemento del *array* se le asigna un bloque de memoria diferente en la implementación *hardware*. Con ello, se posibilita el acceso completamente simultáneo a la totalidad de los datos del *array*. Este particionado es el asignado por defecto cuando no se especifica el parámetro "type" en la directiva [57], [58], [60].

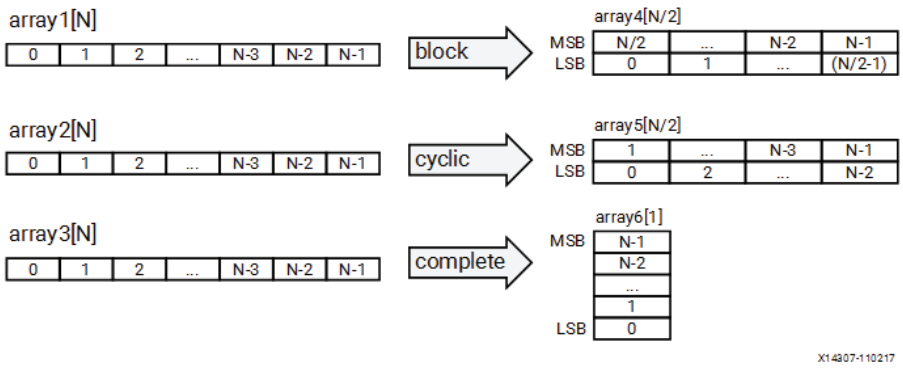


Figura 38. Tipos de partición de los bloques de memoria [57]

- Parámetro "factor". La introducción de este parámetro solo es posible, además de obligatoria, en el caso de que se haya establecido un particionado tipo "cyclic" o "block". En ambos casos, este parámetro indica la cantidad de *arrays* y, por consiguiente, de bloques de memoria entre los que se repartirán los datos del *array* a particionar, siendo "int" dicha cantidad.
- Parámetro "dim". Se utiliza en *arrays* multidimensionales para indicar en qué dimensiones debe efectuarse el particionado solicitado por la directiva. Los valores que puede tomar este parámetro se encuentran entre 0 y la cantidad de dimensiones que posee el *array*, ambos inclusive. Si se indica 0, el particionado

especificado se aplica a todas las dimensiones del *array*. Si se especifica otro valor válido, el particionado se aplica a la dimensión del *array* que dicho valor identifique. La asignación del valor identificativo se efectúa ordenadamente, siguiendo el orden de izquierda a derecha en el que se especifican las dimensiones del *array*.

7.4.2.4 HLS pipeline

Esta directiva se aplica a bucles y a funciones con el propósito de regular el paralelismo escalar o *pipelining* de los mismos. El establecimiento de *pipelining* permite aumentar significativamente las posibilidades de incrementar el *throughput* de la aplicación. Esto se debe a que posibilita la subdivisión de la ejecución del *kernel* FPGA en múltiples etapas que se pueden ejecutar concurrentemente para invocaciones de ejecución contiguas de los bucles o funciones. En la Figura 39 se expone gráficamente el efecto del *pipelining* [57], [58], [59], [60].

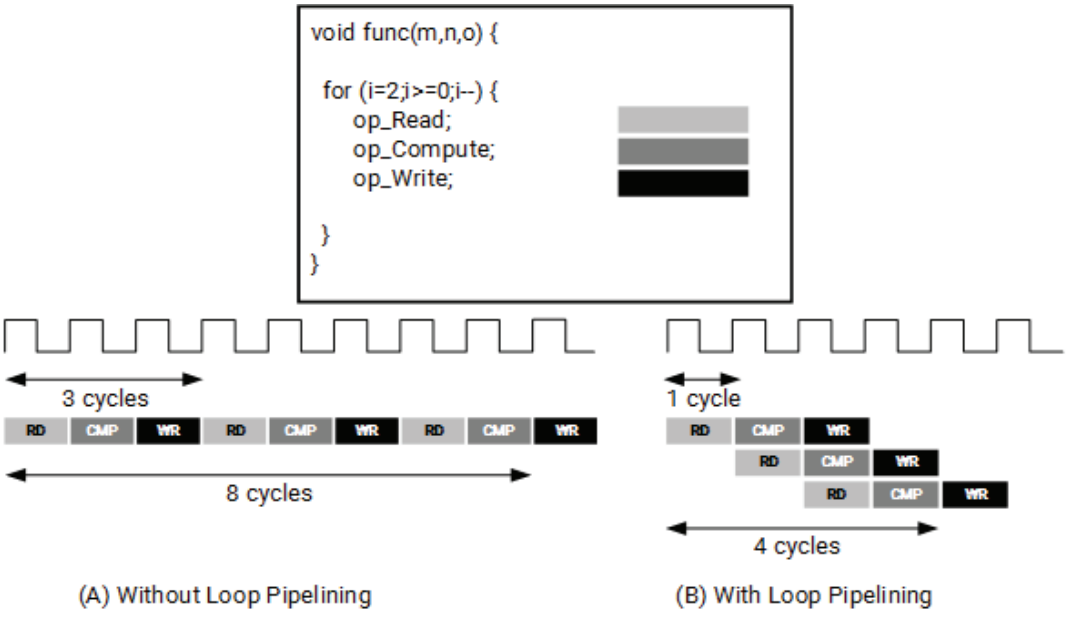


Figura 39: Ejecución sin pipelining (opción A) y con pipelining (opción B) [57], [60]

Esta directiva igualmente presenta algunos parámetros de configuración.

- Intervalo de Iniciación ("II" - *Initiation Interval*). Permite establecer la separación temporal, en ciclos de reloj del *kernel* FPGA, existente entre dos invocaciones de ejecución contiguas de la implementación del bucle o función al que se aplica la directiva. La especificación de este parámetro es

de carácter opcional. En su ausencia, el compilador HLS utiliza el mínimo intervalo de iniciación. Si no es factible cumplir con el intervalo de iniciación exigido, el compilador HLS notificará de ello al usuario por medio de un *warning* y, en su lugar, establecerá el intervalo de iniciación mínimo posible.

- Parámetro "off". Esta opción deshabilita el *pipelining* de la función o bucle a la que la directiva se asigne. Esta posibilidad de configuración resulta útil cuando, bien por otra directiva o bien por la configuración por defecto del compilador HLS, el compilador establece el *pipelining* de la función o bucle en ausencia de una directiva que lo regule explícitamente.

7.4.2.5 HLS inline

Normalmente el compilador opta por implementar cada función como un submódulo dentro del *kernel* para cumplir con la funcionalidad especificada. Este comportamiento por defecto se modifica en el caso de funciones pequeñas. Para estas funciones, el compilador opta por implementar la funcionalidad que dichas funciones requieren no como un submódulo dentro del *kernel* FPGA, sino como una funcionalidad incluida en la función principal, pudiendo además crear varias réplicas particularizadas para cada invocación dentro de la descripción HLS. Esta última alternativa se denomina *inlining*. La directiva **#pragma HLS inline** posee únicamente estas tres variantes [57], [58], [60].

- Opción **#pragma HLS inline**. Esta opción establece que se debe efectuar el *inlining* de la función a la que se le asigna esta directiva [57], [58], [60].
- Opción **#pragma HLS inline off**. Establece que la funcionalidad descrita en la función a la que se le asigna esta directiva se implemente como submódulo dentro del *kernel* FPGA, no efectuándose el *inlining* [57], [58], [60].
- Opción **#pragma HLS inline recursive**. Esta directiva establece que se realice el *inlining* de todas las funciones que aparezcan como funciones invocadas directa o indirectamente por la función a la que se le asigne esta directiva. Desde la perspectiva de la implementación *hardware*, esto implicaría que no existan submódulos dentro del módulo que implementa la totalidad de la funcionalidad de la función a la que se le establezca esta directiva. En cada una de las funciones

invocadas, las implicaciones de esta directiva pueden anularse especificando la directiva **#pragma HLS inline off** [57], [58], [60].

Desde la perspectiva de la utilización de recursos de PL, efectuar *inlining* de las funciones puede tener implicaciones tanto positivas como negativas. Es posible que aumente significativamente la utilización de recursos PL al efectuar el *inlining* de la función. El motivo de ello se encuentra en que se efectuarían múltiples implementaciones particularizadas dentro del *kernel* FPGA de la funcionalidad de esta función, en vez de efectuar una sola implementación que sirva para todas las invocaciones de la función. Sin embargo, aporta facilidades para compartir recursos PL lo que se traduce en una disminución en la utilización de recursos. Sin embargo, en ausencia de *inlining* la funcionalidad de la función de la descripción HLS se suele implementar en un único submódulo *hardware*.

Desde el punto de vista temporal, la utilización del *inlining* es clave para determinar las posibilidades de *pipelining* de un *kernel* FPGA. El motivo de ello se encuentra en que, si no se efectúa el *inlining* de una determinada función, entonces se considera que la ejecución de esta debe constituir una etapa dentro de la ejecución del *kernel* indivisible desde la perspectiva de instauración del *pipelining*. Debido a esto último, el intervalo de *pipelining* mínimo que se puede establecer para este *kernel* quedaría restringido por la latencia de la función sobre la que no se efectúa *inlining*. Cabe destacar que el establecimiento explícito de *inlining* de una función no puede convivir con la asignación de una directiva de *pipelining* a dicha función. En este sentido, la función a la cual se le realiza *inlining* hereda las exigencias de *pipelining* de la función invocadora [58].

En la perspectiva analítica, la aplicación del *inlining* resulta inconveniente. Esto se debe a que la herramienta utilizada para evaluar las posibilidades temporales y de utilización de recursos de los *kernels* FPGA indican esta información no solo para los propios *kernels*, sino que también para los submódulos y bucles que los componen. La gran mayoría de estos submódulos se corresponden a funciones en la descripción HLS de los *kernels*. Esto permite estimar la utilización de recursos PL que las implementaciones de las invocaciones de estas funciones requieren. Esta información será clave a la hora de estimar el nivel de paralelismo en la ejecución del *kernel* FPGA que, desde la perspectiva de la utilización de

recursos PL, puede establecerse. Esto se debe a que un mayor grado de paralelismo requerirá, en la gran mayoría de los casos, introducir una mayor cantidad de réplicas de los módulos *hardware* que implementan cada función en el *kernel* FPGA. La aplicación de directivas *inlining* rompen la jerarquía del diseño y por tanto reducen la información proporcionada con relación a costes de recursos y temporales de las funciones de nivel inferior [57], [60].

Debido a las ventajas que implica el *inlining* en la implementación final de los *kernel*s, se ha optado por aplicar *inlining* a todas las funciones que, bien de manera directa o indirecta, son invocadas por la descripción HLS de los *kernels* FPGA. Sin embargo, para el desarrollo, optimización y análisis de la implementación de estas funciones, tanto desde la perspectiva temporal como de utilización de recursos PL, se ha recurrido a anular temporalmente el *inlining* de estas funciones, introduciendo además las exigencias de *pipelining* que se pretende que la implementación *hardware* de la función sea capaz de cumplir dentro del *kernel* FPGA. En el Código 27 y el Código 28 se muestran un ejemplo de los cambios efectuados [57].

```
void uptRawCtrCoord(px_sum_d* ptr_rawOldCtrDat,
px_sum_d* ptr_rawNewCtrDat, sized_d px_coord){
#pragma HLS INLINE //OFF
//#pragma HLS PIPELINE //II=C_KMEANS_II

      (*ptr_rawOldCtrDat)-=px_coord;
      (*ptr_rawNewCtrDat)+=px_coord;
}
```

Código 27: Función de la descripción HLS con inlining

```
void uptRawCtrCoord(px_sum_d* ptr_rawOldCtrDat, px_sum_d*
ptr_rawNewCtrDat, sized_d px_coord){
#pragma HLS INLINE OFF
#pragma HLS PIPELINE II=C_KMEANS_II

      (*ptr_rawOldCtrDat)-=px_coord;
      (*ptr_rawNewCtrDat)+=px_coord;
}
```

Código 28: Función de la descripción HLS con inlining anulado

7.4.2.6 HLS function_instantiate

Esta directiva viene acompañada del parámetro “variable”, siguiendo la sintaxis mostrada en el Código 29.

```
#pragma HLS function_instantiate variable=<variable>
```

Código 29: Sintaxis de la directiva "#pragma HLS function_instantiate" [57], [60]

Esta directiva es aplicable a las funciones del diseño, siendo "variable" el nombre del parámetro de llamada de la función a la que se asigna esta directiva. Por defecto, las funciones se implementan como un único módulo RTL. De esta manera, cada invocación de la función descrita en el código fuente corresponderá, en la implementación *hardware*, a la activación y ejecución del bloque del *kernel* FPGA responsable de ejecutar cada invocación de la función que implementa. Sin embargo, esta directiva establece que, para cada valor de llamada actual presente en las invocaciones de la función, se instancie un bloque específico. Con ello, se logra que múltiples invocaciones de la función se ejecuten concurrentemente en su implementación *hardware*. En cada uno de los diseños RTL, el parámetro "variable" debe poseer un valor constante establecido en tiempo de compilación, en torno al cual se personalizará el diseño. Este hecho permitirá simplificar la lógica de control de cada instancia RTL de la función [57], [60]. En el Código 30 se muestra la utilización de la directiva, donde se obtendrá como resultado tres bloques diferentes optimizados para cada valor de "incr".

```
char func_sub(char inval, char incr) {
#pragma HLS INLINE OFF
#pragma HLS FUNCTION_INSTANTIATE variable=incr
 return inval + incr;
}
void func(char inval1, char inval2, char inval3,
 char *outval1, char *outval2, char * outval3)
{
 *outval1 = func_sub(inval1, 1);
 *outval2 = func_sub(inval2, 2);
 *outval3 = func_sub(inval3, 3);
}
```

Código 30: Ejemplo de la directiva "#pragma HLS function_instantiate" [57], [60]

7.4.2.7 HLS dependence

Esta directiva permite informar al compilador acerca de las dependencias de datos internas del bucle, es decir, cuando dos o varias operaciones requieren acceder a un mismo dato. En este sentido, existen dos tipos de dependencias [57], [58], [60]:

- Dependencias en iteraciones independientes. La dependencia de los datos se encuentra entre operaciones de una misma iteración del bucle.
- Dependencias entre iteraciones. La dependencia de los datos se encuentra entre operaciones entre distintas iteraciones del bucle.

Cabe destacar que el compilador Vitis HLS posee cierta capacidad para detectar estas dependencias en los bucles. Sin embargo, en ciertas circunstancias puede ocurrir que no haya dependencia de datos y el compilador no pueda percatarse de ello. En consecuencia, ocurre una falsa dependencia que obliga, sin necesidad funcional o algorítmica, a limitar el rendimiento en aspectos temporales de la aplicación, especialmente en lo referente al *pipelining*. Una situación en la que puede ocurrir la falsa dependencia es, por ejemplo, cuando se requiere el acceso a *arrays* mediante indexación dependiente de una variable. Otra situación en la que puede suceder resulta cuando, para que no exista dependencia de datos, es necesario considerar alguna condición que se cumplirá con los datos entrantes al bucle [57], [58], [60].

Debido a estas limitaciones, se provee la directiva **#pragma HLS dependence**, la cual permite notificar al compilador dependencias de datos detectadas por el programador. La sintaxis de esta directiva es la mostrada en el Código 31 [57], [58], [60].

```
#pragma HLS dependence variable=<variable> <class> \
<type> <direction> distance=<int> <dependent>
```

Código 31: Sintaxis de la directiva "#pragma HLS dependence" [57], [60]

Los parámetros que admite esta directiva son los siguientes:

- Parámetros "variable" y "class". Son excluyentes, debiendo el usuario de optar por alguno de ellos de manera obligatoria y exclusiva en cada especificación de esta directiva. Si elige "variable" se deberá especificar el nombre de la variable en torno a la cual se especifican aspectos relativos a las dependencias de datos existentes dentro del bucle. Por otra parte, si se escoge "class" el usuario podrá especificar "array" o "pointer", lo cual originará que las indicaciones introducidas por la directiva se apliquen a todas las variables accedidas dentro del bucle que coincidan con la tipología especificada.

- Parámetro “type”. Puede adquirir solamente los valores “intra” e “inter”. Si se indica “intra”, ello implica que la directiva notifica acerca de dependencias de datos que ocurren entre operaciones de una misma iteración. Contrariamente, si se indica “inter”, la directiva especificará acerca de dependencias de datos entre operaciones que se ejecutan en iteraciones del bucle distintas. La introducción de este parámetro es opcional y, en el caso de no especificarse, el compilador Vitis HLS establece la opción “inter”.
- Parámetro “direction”. Su especificación es opcional y solo se aplica en el caso de que la directiva indique dependencia entre iteraciones del bucle. Su objetivo consiste en detallar el orden entre lecturas y escrituras sobre el dato que origina la dependencia de datos. Las opciones que pueden especificarse son RAW (*Read After Write*), WAR (*Write After Read*) y WAW (*Write After Write*).
- Parámetro “distance”. Parámetro opcional que se aplica en el caso de que se especifique la existencia de dependencia entre iteraciones del bucle. Este parámetro detalla cuántas iteraciones más adelante, como mínimo y con respecto a cualquier otra iteración de referencia se encuentra otra iteración con la cual exista dependencia. Esta distancia se especifica con el valor de “int”.
- Parámetro “dependent”. Señaliza la existencia de dependencia con las características detalladas por los parámetros anteriores de la directiva. En caso de que exista, deberá especificarse el valor “true”, mientras que “false” se especificará en el caso contrario. Es muy recomendable especificar este parámetro. El compilador Vitis HLS asumirá el valor “false” en ausencia de indicación [57], [58], [60].

7.5. Generación de ficheros de la plataforma del sistema

Tras crearse el ejecutable del *host* y el archivo “.xclbin” para programar la FPGA, se procede al prototipado. Este último procedimiento se ordena al solicitar que el compilador “v++” ejecute la opción “-p” o “--package”. Este empaquetamiento consiste en la agrupación, en un único archivo, de varios archivos que deben utilizarse en la placa de prototipado (Figura 40). El archivo resultante posee la extensión “.img” [52].

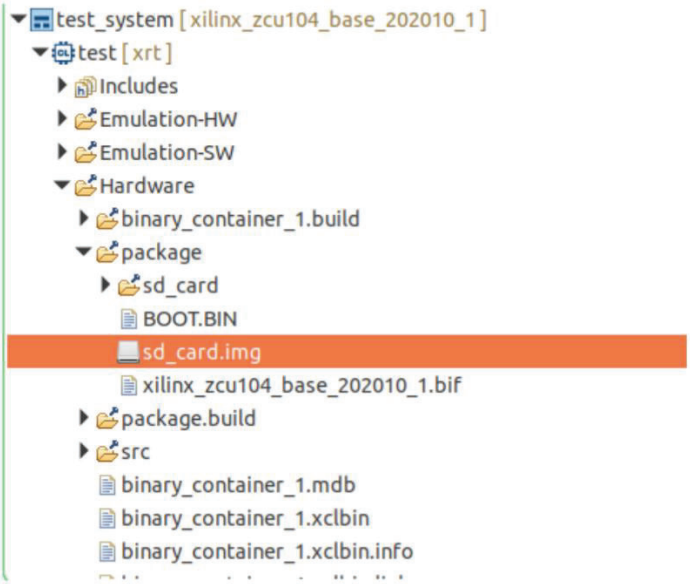


Figura 40. Ejemplo de ficheros producidos por Vitis

Para la compilación de aplicaciones destinadas a sistemas empotrados, entre los archivos a empaquetar se incluyen el ejecutable del *host* y el binario ".xclbin" para la configuración de la FPGA.

Por otra parte, deben incluirse en el empaquetamiento archivos de soporte adicionales. Entre estos últimos archivos se encuentra la configuración del arranque de la placa; el "Root FS", que establece el sistema de archivos raíz dentro de la placa de prototipado y un *kernel* de Linux, que instala una versión del sistema operativo dentro de la placa. Con respecto al *kernel* de Linux, este archivo puede indicarse desde la interfaz gráfica de Vitis. Para ello, debe utilizarse la opción "Kernel Image" presente dentro del proyecto Vitis de la aplicación en el archivo "<nombre del proyecto>_system.sprj". Esta opción se señala en la Figura 41 [52], [61].

El empaquetamiento realizado varía en función del modo de arranque de la placa de prototipado. Dicho modo de arranque puede indicarse al compilador "v++" durante el empaquetamiento mediante la opción "--package.boot_mode", siendo la opción por defecto el arranque mediante tarjeta SD (*Secure Digital*) [52], [62].

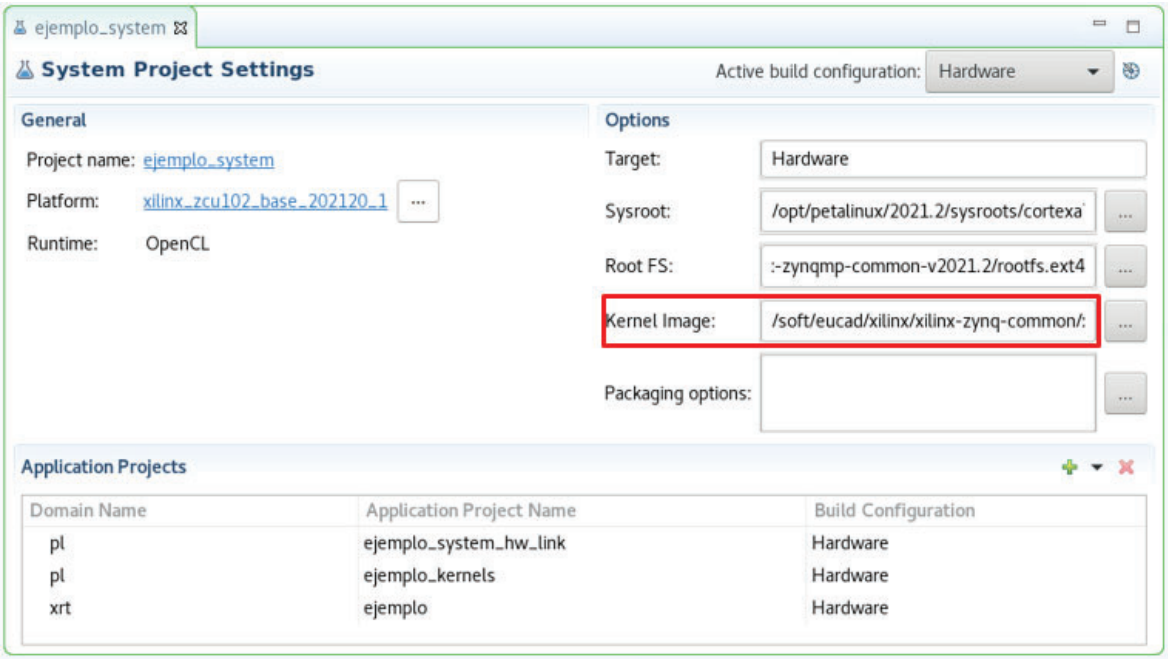


Figura 41: Especificación del kernel de Linux

7.6. Conclusiones

En este capítulo se ha detallado el flujo de diseño utilizado para el desarrollo de la aplicación. Dicho flujo se ajusta al considerado en la metodología Vitis HLS de Xilinx y facilita el desarrollo y depurado progresivo de aplicaciones que recurren a la aceleración *hardware* mediante FPGA para su ejecución. Este flujo de diseño presenta variaciones en función del tipo y de la ubicación, con respecto a la FPGA, del microprocesador que ejecuta la aplicación anfitriona. Además, el flujo de diseño difiere en función de la API utilizada para realizar las interacciones entre el *host* y los *kernels* FPGA y del modo de arranque de la placa de prototipado donde se ejecuta, al menos, la aceleración *hardware*.

Para el caso de este trabajo, el flujo de diseño se corresponde al de una aplicación donde el *host* se ejecuta en un microprocesador de 64 *bits* empotrado en la placa de prototipado. Otras características relevantes para el flujo de diseño son que el *host* utiliza la API de OpenCL para la interacción con los *kernels* FPGA, y que la aplicación resultante se introduce en la placa de prototipado mediante tarjeta SD.

Un aspecto característico del flujo de diseño aplicado son las distintas directivas **#pragma HLS** que se utilizan en la descripción HLS de los *kernels* FPGA a implementar en la aplicación. Estas directivas son importantes para optimizar la implementación *hardware* de

la descripción funcional de los *kernels*. Por otra parte, también se han indicado las distintas opciones de configuración que presentan las distintas directivas. Entre las directivas presentes, las más destacables son las directivas **#pragma HLS array_partition** y **#pragma HLS unroll**. Con ellas, se establece en la implementación *hardware* el grado de paralelismo en el procesamiento exigido por la configuración estática de los *kernels* FPGA.

Otra directiva intensivamente utilizada es la denominada **#pragma HLS inline**. Esta directiva será especialmente útil para mejorar las posibilidades de *pipelining* mínimo de la ejecución de los aceleradores *hardware*. Además, la posibilidad de regular el *inlining* de las funciones de la descripción HLS permite garantizar el grado de replicación de la implementación *hardware* de las funcionalidades descritas. Con ello, puede distribuirse la descripción de estas funcionalidades en distintas funciones, facilitando con ello la legibilidad y remodelación del código. Por último, es destacable la directiva **#pragma HLS pipeline**, gracias a la cual se ordena el *pipelining* a exigir al *kernel* FPGA que corresponda.

Capítulo 8. Aplicación desarrollada

8.1. Introducción

En este capítulo se explican aspectos de la programación del código presentes en todas o varias etapas de esta aplicación. En primer lugar, esta explicación comienza exponiendo las opciones de configuración que se han facilitado para que el usuario pueda reprogramar fácilmente la aplicación. A continuación, se prosigue con la explicación de algunos aspectos generales de la programación en C del *host* y de la descripción HLS de los *kernels* FPGA.

Posteriormente, se describen cada una de las etapas que conforman la aplicación, relativas tanto a la descripción HLS de los *kernels* FPGA como a la programación del *host*. A continuación, se explica un procedimiento para la configuración estática de los *kernels* FPGA basado en una hoja de cálculo. Finalmente se detalla el procedimiento aplicado para realizar el prototipado del MPSoC, aportando un conjunto de conclusiones obtenidas del desarrollo del sistema.

8.2. Opciones de configuración

Se ha dotado al código de cierta versatilidad que permita que la aplicación se ajuste fácilmente a distintas condiciones de funcionamiento. Algunos de estos ajustes son de carácter dinámico, es decir, se efectúan cambiando los datos de entrada al invocar la ejecución de la aplicación sin necesidad de volver a compilar. Sin embargo, otros ajustes son de carácter estático, lo cual implica que, para que se apliquen, es necesario recompilar la aplicación.

Esta aplicación está planificada para que su invocación se efectúe mediante líneas de comandos. Para invocar la aplicación es necesario indicar el fichero en formato ".dat" que contiene la imagen hiperespectral de entrada. Opcionalmente, el usuario podrá indicar en línea de comandos las dimensiones y el criterio de ordenación que siguen los datos de la imagen hiperespectral.

El proceso para extraer los parámetros de la invocación en línea de comandos e introducirlos en el funcionamiento interno de la aplicación se codifica en la función "getSpecifications" del código "main.cpp" (Código 32). En ausencia de estas especificaciones, la aplicación adquirirá una configuración por defecto.

```
inline void getSpecifications(int argc, char** argv, hst_dim_d* dim,
char* img_format, char** filename){
	if(argc < 2){
		printf("ERROR: Es necesario especificar el fichero que "
"contiene la imagen hiperespectral\n");
		exit(EXIT_FAILURE);
	}
	*filename = argv[1];

	//Caso especificacion en ventana de comandos
	if(argc >= 5 ){
		//Caso sobran especificaciones de la ventana terminal
		if(argc > 6){
			printf("ERROR: Demasiados parametros\n");
			exit(EXIT_FAILURE);
		}

		for(int i = 0; i < 3; i++){
			dim[i] = _STR_TO_NUM(argv[i+2]);
		}

		//Caso se incluye especificacion del interleave
		if(argc == 6){
			getSpecifiedFormat(argv[5], img_format);
		}
		checkExtractedDims(dim);
	//Caso especificacion con fichero de cabecera
	}else if(argc == 3){
		char* parameters[] = {"lines", "samples", "bands",
		"interleave"};
		getHDRSpecifications(argv[2], parameters, dim, img_format);
		checkExtractedDims(dim);

	/*Caso especificacion con fichero de cabecera
	 * incluyendo aparte el criterio de ordenacion.*/
	}else if(argc == 4){
		char* parameters[] = {"lines", "samples", "bands"};
		getHDRSpecifications(argv[2], parameters, dim);
		getSpecifiedFormat(argv[3], img_format);
		checkExtractedDims(dim);
```

```
        }
}
```

Código 32: Función "getSpecifications"

Como se observa en la función "getSpecifications", la aplicación puede extraer los parámetros de la invocación si se especifican siguiendo alguno de estos formatos:

- Invocación <nombre ejecutable> <imagen hiperespectral>. Especifica únicamente la imagen hiperespectral a procesar y evaluar, dejando los demás parámetros con la configuración por defecto.
- Invocación <nombre ejecutable> <imagen hiperespectral> <archivo de cabecera>. Especifica la imagen hiperespectral de entrada y un archivo de texto en el que se especifican los demás parámetros. En este archivo, las dimensiones se identifican como "samples" para el ancho, "lines" para el largo y "bands" para la dimensión espectral. El criterio de ordenación de los datos de entrada se identifica como "interleave". En el Código 33 se ejemplifica el contenido de este archivo. Para extraer los parámetros relevantes, la función "getSpecifications" invoca la función "getHDRSpecifications" (Código 34) del archivo "main.cpp".

```
ENVI
bands = 125
data type = 12
interleave = bsq
header offset = 0
wavelength units = Nanometers
byte order = 0
wavelength = {450, 454, 458, 462, 466, 470, 474, 478, 482, 486,
490, 494, 498, 502, 506, 510, 514, 518, 522, 526, 530, 534, 538,
542, 546, 550, 554, 558, 562, 566, 570, 574, 578, 582, 586, 590,
594, 598, 602, 606, 610, 614, 618, 622, 626, 630, 634, 638, 642,
646, 650, 654, 658, 662, 666, 670, 674, 678, 682, 686, 690, 694,
698, 702, 706, 710, 714, 718, 722, 726, 730, 734, 738, 742, 746,
750, 754, 758, 762, 766, 770, 774, 778, 782, 786, 790, 794, 798,
802, 806, 810, 814, 818, 822, 826, 830, 834, 838, 842, 846, 850,
854, 858, 862, 866, 870, 874, 878, 882, 886, 890, 894, 898, 902,
906, 910, 914, 918, 922, 926, 930, 934, 938, 942, 946}
lines = 50
samples = 50
```

Código 33: Contenido de ejemplo de un archivo de cabecera

```
inline void getHDRSpecifications(char* filename, char** datanames,
hst_dim_d* dim, char* img_format=NULL){
  FILE *formatFile = fopen(filename, "r");
  if(formatFile == NULL){
    printf("ERROR: Archivo %s no encontrado\n", filename);
    exit(EXIT_FAILURE);
  }
  std::vector<unsigned char> buscar = {0, 1, 2};
  if(img_format != NULL){
    buscar.push_back(3);
  }
  //Recorrido letra a letra por el archivo
  for(int letra = fgetc(formatFile); (buscar.size() != 0) &&
      (letra != EOF); letra = fgetc(formatFile)){
    //Recorrido nombre a nombre a buscar
    for(unsigned char k=0; k<buscar.size(); k++){
      auto idx = buscar[k];
      char* target_ptr = datanames[idx];
      //Caso letra coincide con letra inicial del identificador
      if(letra == *target_ptr){
        fpos_t tmp_pos;
        fgetpos(formatFile, (fpos_t*)&tmp_pos);
        //Se evalua hasta donde continua la semejanza
        target_ptr++;
        do{
          letra = fgetc(formatFile);
        }while((letra == *target_ptr) && (*(++target_ptr) != '\0'));
        //Caso la cadena identificadora encontrada
        if(*target_ptr == '\0'){
          /*Se obvian espacios en blanco entre el nombre y el =*/
          do{
            letra = fgetc(formatFile);
          }while(letra == ' ');
          /*Se cumple si el nombre en el HDR es exactamente igual al
           * del parametro*/
          if(letra == '='){
            if(idx < 3){
              dim[idx] = getFromCharToInt<hst_dim_d>(formatFile);
              eraseElement(buscar, idx);
            }else if(idx == 3){
             //Quita espacios sobrantes
             do{
               letra = fgetc(formatFile);
             }while((letra == ' ')&&(letra != EOF));
             if(letra != EOF){
               std::vector<char> formatName;
               while((letra != ' ')&&(letra != '\n')&&(letra != EOF)){
                 formatName.push_back((char)letra);
                 letra = fgetc(formatFile);
               }
               formatName.push_back('\0');
               getSpecifiedFormat(formatName.data(), img_format);
               eraseElement(buscar, idx);
             }
           }else{
             printf("WARNING: Mas identificadores que parametros a "
             "extraer\n");
           }
          }
```

```
        //Caso la cadena identificadora incompleta
        }else{
          fsetpos(formatFile, (fpos_t*)&tmp_pos);
        }
      }
    }
  }
  fclose(formatFile);
  //Error no todos los elementos solicitados fueron encontrados
  if(buscar.size() != 0){
    for(unsigned char k : buscar){
      printf("ERROR: El parametro \"%s\" no se encuentra en el archivo
```

Código 34: Función "getHDRSpecifications"

- Invocación <nombre ejecutable> <imagen hiperespectral> <archivo de cabecera> <*interleaving*>. La especificación de los datos en este formato es similar a la del anterior. Sin embargo, la especificación del criterio de ordenación de los datos de la imagen se establecerá con el último parámetro de la invocación de manera directa.

- Invocación <nombre ejecutable> <imagen hiperespectral> <ancho> <largo> <longitud espectral>: Aparte de la imagen hiperespectral de entrada, esta invocación especifica las dimensiones de la imagen de manera directa. El criterio de ordenación de los datos de la imagen hiperespectral de entrada se mantiene con la configuración por defecto.

- Invocación <nombre ejecutable> <imagen hiperespectral> <ancho> <largo> <longitud espectral> <*interleaving*>. Se especifica la imagen hiperespectral de entrada a la aplicación, además de todos los parámetros especificables por línea de comandos, de manera directa.

Con respecto a la imagen hiperespectral de entrada, la aplicación es capaz de reajustarse dinámicamente a cualquiera de los criterios de ordenación de los datos de dicha imagen (BIP, BIL y BSQ). En ausencia de especificación, la aplicación considerará el formato BSQ como el formato de ordenación de los datos de entrada, tal y como se especifica en la inicialización de la variable "HS_IMAGE_FORMAT", definida en el archivo "main.hpp" y responsable de contener en la aplicación esta información.

Cabe destacar que, si el criterio de ordenación de los datos se especifica en la invocación de la aplicación, su introducción en la aplicación se describe en la función "getSpecifiedFormat" (Código 35) declarada en el archivo "main.cpp".

```
inline void getSpecifiedFormat(std::vector<char> param,
                               char* img_format){
  const char* nombresBSQ[] = _NOMBRES_BSQ;
  const char* nombresBIL[] = _NOMBRES_BIL;
  const char* nombresBIP[] = _NOMBRES_BIP;
  const char** casos[] = {nombresBSQ, nombresBIL, nombresBIP};
  //Recorrido por casos de formatos
  for(const char** caso : casos){
    //Recorrido nombres identificativos posibles del formato
    for(const char** nombre_ptr=caso; nombre_ptr[0][0] != '\0';
nombre_ptr++){
      const char* nombre= *nombre_ptr;
      int k;
      //Recorrido por caracteres del nombre identificativo
      for(k=0; (nombre[k] != '\0') && (nombre[k]==param[k]); k++);
      //Caso coincidencia de nombre
      if(nombre[k]==param[k]){
        if(caso==nombresBSQ){
          printf("Establecida serializacion de la imagen BSQ\n");
          *img_format=2;
          return;
        }else if(caso==nombresBIL){
          printf("Establecida serializacion de la imagen BIL\n");
          *img_format=1;
          return;
        }else if(caso==nombresBIP){
          printf("Establecida serializacion de la imagen BIP\n");
          *img_format=0;
          return;
        }
      }
    }
  }
  //Solo se llega aqui si el nombre de formato especificado no esta
  //registrado
  printf("ERROR: Criterio de serializacion de la imagen "
         "desconocido\n");
  exit(EXIT_FAILURE);
}
```

Código 35: Función "getspecifiedFormat"

El establecimiento dinámico del criterio de ordenación de los datos de la imagen hiperespectral se efectúa por medio de la especificación de un nombre o cadena de caracteres que identifique dicho criterio.

Cada criterio de ordenación puede poseer una cantidad múltiple y variable de nombres identificativos, los cuales se especifican en el archivo "main.hpp" con las macros

"_NOMBRES_BSQ", "_NOMBRES_BIL" y "_NOMBRES_BIP" para los criterios de ordenación BSQ, BIL y BIP respectivamente. La especificación de estas macros se expone en el Código 36. Por motivos funcionales, la cadena de caracteres "\0" debe indicarse al final de cada lista de nombres identificativos de un mismo criterio de ordenación.

```
#define   _NOMBRES_BSQ      {"BSQ", "bsq", "2", "\0"}
#define   _NOMBRES_BIL      {"BIL", "bil", "1", "\0"}
#define   _NOMBRES_BIP      {"BIP", "bip", "0", "\0"}
```

Código 36: Declaración de nombres identificativos de los criterios de ordenación

Respecto a las dimensiones de la imagen hiperespectral, el ancho y largo espacial, así como la cantidad de bandas espectrales de las imágenes a utilizar son dinámicamente configurables. Sin embargo, para el funcionamiento de la aplicación es necesario que estas dimensiones se mantengan dentro de unos márgenes que se establecen estáticamente.

Las dimensiones de la imagen se especifican durante la invocación de la aplicación, bien mediante su indicación directa o mediante la especificación de un archivo ".hdr" accedido por la aplicación y que contiene dicha información. En ausencia de especificación, se optará por considerar el dimensionamiento por defecto de la imagen. Las dimensiones por defecto se establecen en tiempo de compilación mediante las macros "_DEFAULT_X" para el ancho de la imagen, "_DEFAULT_Y" para el largo de la imagen y "_DEFAULT_BANDS" para la cantidad de bandas espectrales. Estas macros se indican en el archivo "datatype.h", en la sección del código expuesta en el Código 37.

```
//----------------------CONF-------------------------
#ifndef _DEFAULT_X
/**@brief Especifica la cantidad de
 * columnas de pixels que por
 * defecto tiene la imagen. */
#define _DEFAULT_X                45
#endif

#ifndef _DEFAULT_Y
/**@brief Especifica la cantidad de
 * filas de pixels que por defecto
 * tiene la imagen. */
#define _DEFAULT_Y                45
#endif

#ifndef _DEFAULT_BANDS
/**@brief Especifica la cantidad de
 * bandas espectrales que por
 * defecto tiene la imagen. */
```

```
#define _DEFAULT_BANDS           100
#endif
//--------------------CONF END----------------------
```

Código 37: Especificación de las dimensiones por defecto de las imágenes hiperespectrales

La cantidad de píxeles de la imagen también es dinámicamente configurable siempre y que dicho valor, aparte de ser positivo y no nulo, sea inferior o igual a la cantidad máxima de píxeles por imagen que la aplicación es capaz de aceptar. Dicha cantidad máxima es ajustable durante la compilación de la aplicación por medio de la macro "_MAX_PX".

El valor de esta macro no debe ser modificado por el usuario de manera directa. En su lugar, el código fuente está organizado para que el usuario especifique el valor que desee de máxima cantidad de píxeles por imagen por medio de la macro "_USER_MAX_PX". Si este valor es igual o superior a la cantidad de pixeles de la imagen indicada por defecto, entonces dicho valor se establece como el valor de la macro "_MAX_PX". En caso contrario, se ignora el valor de la macro "_USER_MAX_PX" y, en su lugar, se ajusta "_MAX_PX" al valor de la cantidad de píxeles por defecto en la imagen, notificando mediante el aviso conveniente dicho cambio.

Cabe destacar que la cantidad de píxeles por defecto de la imagen hiperespectral se especifica en la macro "_DEFAULT_PX" como producto entre "_DEFAULT_X" y "_DEFAULT_Y". "_DEFAULT_PX" no debe modificarse directamente.

Con respecto al valor de la macro "_MAX_PX", esta no se encontrará sujeta a restricciones significativas aparte de poseer un valor positivo no nulo. En este sentido, las restricciones en el ajuste se limitan a no requerir una cantidad de *bits* para la representación en binario natural sin signo del valor de "_MAX_PX" superior a 64. Además, deberá tenerse en cuenta que la especificación de una mayor cantidad máxima de píxeles supondrá, no solo una mayor ocupación de memoria en la parte PS de la plataforma, sino que también un aumento en la utilización de recursos de lógica programable debido a un mayor aumento del tamaño de diferentes registros de conteo relativos a la identificación de los píxeles en los *kernels* FPGA.

El ajuste dinámico de la cantidad de bandas espectrales de la imagen se encuentra restringido a establecer un valor igual o inferior en cantidad de bandas útiles al valor de la macro "_MAX_BANDS". Esta macro se detalla en el archivo "datatype.h", y su valor puede ser ajustado por el usuario directamente en el momento de la compilación de la aplicación.

Por otra parte, la cantidad de bandas útiles deberá ser superior o igual a la cantidad de datos por promediado que regularmente se aplica en los módulos de filtrado del *kernel* de filtrado. Esta cantidad de datos por promediado se especifica mediante la macro "_AVERAGING" especificada en el archivo "amounts.h". En lo que respecta a las restricciones en el valor de "_MAX_BANDS", estas son semejantes a las que posee la macro "_MAX_PX", además de deber no ser inferior al valor de "_AVERAGING".

Cabe destacar que, con las posibilidades de configuración de esta aplicación, las bandas útiles no se corresponden necesariamente a la totalidad de las bandas espectrales de la imagen hiperespectral de entrada. Asimismo, estas bandas útiles se obtienen al excluir del conjunto de bandas espectrales totales de la imagen una cierta cantidad de bandas espectrales extremas configurada en tiempo de compilación. La cantidad de bandas espectrales a excluir se definen en "_QUIT_LOWER_SPECTRA" para las bandas del extremo inferior, es decir, de menor longitud de onda, y "_QUIT_UPPER_SPECTRA" para las bandas del extremo superior (Código 38). Estas macros se encuentran declaradas en el archivo "datatype.h", pudiendo el usuario configurarlas a cualquier valor que desee entre el valor nulo (0) y unos valores lo suficientemente pequeños para que en la ejecución de la aplicación queden al menos tantas bandas útiles como el valor de "_AVERAGING". Además, "_QUIT_LOWER_SPECTRA" y "_QUIT_UPPER_SPECTRA" pueden configurarse para valores distintos, pudiendo con ello implementarse en la aplicación una eliminación de bandas espectrales asimétrica.

```
#define   _QUIT_LOWER_SPECTRA                    4
#define   _QUIT_UPPER_SPECTRA                    5
```

Código 38: Bandas espectrales extremas residuales de las imágenes hiperespectrales

Con respecto a los *kernels* FPGA, un aspecto común que los caracteriza consiste en la posibilidad de ajustar en tiempo de compilación el nivel de procesamiento completamente en paralelo y simultáneo de los datos que cada *kernel* es capaz de efectuar.

Este grado de paralelismo no incluye la opción de *pipelining*, la cual se establecerá aparte para mejorar aún más las posibilidades de *throughput* de los *kernels* FPGA. Este paralelismo del procesamiento será directamente ajustable en el código fuente desde la perspectiva del usuario y en el momento de la compilación. Para ello, el usuario deberá reescribir el valor correspondiente a la macro que describe dicho aspecto del funcionamiento del *kernel* FPGA.

Para el ajuste de este procesamiento en paralelo de los *kernels*, el usuario deberá considerar los siguientes aspectos:

1. Uso de recursos de lógica programable de la FPGA. Un mayor grado de paralelismo en el procesamiento de los datos de cada *kernel* FPGA conlleva que una mayor cantidad de módulos que efectúen el procesamiento de estos datos se implementen en la FPGA, lo que implica un aumento en la utilización de recursos de la FPGA requeridos por el *kernel*. En consecuencia, la cantidad de recursos de lógica programable que posee la FPGA a utilizar como soporte físico puede resultar insuficiente para el *kernel*. Además, cabe destacar que, si se desea que el *kernel* FPGA a ajustar conviva en una misma configuración de la FPGA con otros *kernels*, debe tenerse en cuenta el grado de ocupación de recursos de la FPGA que precisarán los demás *kernels*. Esto se debe a que, aunque la FPGA posea recursos de lógica programable suficientes para implementar el *kernel* a ajustar con el nivel de paralelización exigido, dicha implementación puede ser a costa de restringir significativamente la disponibilidad de recursos de lógica programable para los demás *kernels* FPGA. En consecuencia, estos *kernels* no podrían coexistir en la misma configuración de la FPGA que el *kernel* ajustado; o sí podrían, pero con un grado de paralelización en su procesamiento deficiente.

2. *Throughput*. Al aumentar la cantidad de datos que el *kernel* es capaz de procesar en paralelo, se aumenta la velocidad con la que dicho *kernel* puede tomar los datos de entrada y de generar los correspondientes datos de salida. Por lo tanto, se aumenta el *throughput*.

3. Ancho de las interfaces físicas PS-PL (transferencia *host-kernel* FPGA). Cabe destacar que un importante cuello de botella debido al soporte físico de la aplicación se corresponde a la velocidad de transferencia posible entre el *host* y el *kernel* FPGA. Dicha velocidad se encuentra limitada por las capacidades físicas de velocidad, así como de transferencia en paralelo, que poseen las interfaces físicas que interconectarán, en la aplicación, al microprocesador que ejecuta el código *software* del *host* y a la FPGA que ejecuta la aceleración *hardware* que ofrecen los *kernels* FPGA. En consecuencia, las interfaces físicas suponen una limitación en el *throughput* de la aceleración *hardware*. Por este motivo, si se alcanza dicho límite, no tendrá sentido seguir aumentando el grado de procesamiento en paralelo del *kernel*. El motivo de ello se debe a que supondrá un aumento de la utilización de recursos de lógica programable que, desde la perspectiva temporal, estarán gran parte del tiempo inactivas. Asimismo, dicho "sobredimensionamiento" en el grado de paralelización del *kernel* solo tendría sentido para minimizar el consumo energético de la ejecución de estos *kernel*. Este último aspecto ocurriría debido a que un grado de paralelización sobredimensionado posibilitaría mantener el *throughput* con un ciclo de reloj del *kernel* FPGA más lento.

Para configurar el grado de paralelización del procesamiento en los *kernels* FPGA, el usuario deberá recurrir a modificar el valor de la macro "_N_FILTERS" para el caso de los *kernels* de la etapa de preprocesado.

Por otra parte, deberá modificar el valor de las macros "_KMEANS_PARALLEL" y "_SAM_PARALLEL" para ajustar, respectivamente, el grado de procesamiento en paralelo de los *kernels* de las etapas "k-means" y *SAM*. Todas estas macros se encuentran presentes en el archivo "amounts.h" del código fuente de la aplicación.

Como se mencionó con anterioridad, el usuario podrá configurar en el *kernel* de filtrado la cantidad de datos de promediado que se utilizarán en el filtrado de los datos de los píxeles, para lo cual se recurre a la macro "_AVERAGING". Dicha macro debe ajustarse a un valor positivo, no nulo e impar, además de cumplir con las restricciones con respecto a la cantidad de bandas espectrales útiles que se mencionó con anterioridad.

Otro aspecto configurable del código relativo a la etapa de preprocesamiento corresponde al formato de interpretación de los datos de entrada de la imagen hiperespectral en los *kernel*s FPGA para su procesamiento. Este formato se detalla con la definición del tipo de variable "unsized_d". Asimismo, el usuario podrá configurar la longitud en *bits* que tendrá cada dato de este tipo de variable a cualquier valor entre 1 y 64 *bits*. Para ello, se debe asignar la longitud en *bits* que desea a la macro "_INT_PART" que se describe en el archivo "datatype.h". En el Código 39 se muestra la especificación de "unsized_d". Téngase en cuenta que "_AP_UFIXED" representa una función macro que crea un tipo de variables sin signo con decimales en aritmética de punto fijo de, en este caso, "_INT_PART" *bits* de parte entera y "_DEC_PART" *bits* de parte decimal.

```
//----------------------CONF-----------------------
#define _INT_PART            16
//--------------------CONF END---------------------
#define _DEC_PART            0

/**Tipo de dato antes de normalizar*/
typedef _AP_UFIXED(_INT_PART, _DEC_PART) unsized_d;
```

Código 39: Declaración del tipo de variable "unsized_d"

8.3. Funciones macro para la creación de variables de precisión arbitraria

Para mejorar las prestaciones de la aplicación, un aspecto clave es lograr que la implementación de los *kernels* FPGA maximicen su rendimiento. Esto implica conseguir que dichos *kernels* posean las prestaciones funcionales exigidas con la menor utilización de recursos de lógica programable y de tiempo de ejecución posible. Para ello, uno de los aspectos claves desde la perspectiva del código fuente consiste en minimizar el tamaño en *bits* que corresponderá al registro de cada variable del código fuente de los *kernels* FPGA para su implementación RTL.

En el caso de las variables enteras positivas, este tamaño debe corresponderse al mínimo necesario para representar el máximo valor que dicha variable puede obtener en las múltiples y posibles ejecuciones que puedan ocurrir tras su compilación. De esta manera, se garantiza el funcionamiento de la aplicación para cualquier combinación de datos de entrada introducidos dinámicamente.

Al mismo tiempo, se minimiza la utilización de recursos de lógica programable en la implementación de los registros tanto como resulte posible. Para la declaración de estas variables dentro del *kernel* FPGA se utilizarán las clases de C++ “ap_int<W>”, para las variables con signo, y “ap_uint<W>” para las variables sin signo. Ambas variables fueron desarrolladas por Xilinx, siendo especificadas en el archivo “ap_int.h” en las que “W” representa la longitud en *bits* requerida para dicha variable en su implementación RTL.

Para lograr que el ajuste de dicha longitud se efectúe de forma óptima y automática, es necesario disponer de una API que determine, en el momento de la compilación, la cantidad de *bits* mínimos requeridos en una representación sin signo de un valor positivo conocido en el momento de la compilación. Dicho valor resultante puede obtenerse especificando “UnsignedBitWidth<N>::Value”. En esta última declaración, “N” representa el valor positivo del que se desea determinar la cantidad de *bits* que se requiere para su representación sin signo. Asimismo, “Value” se define como una constante estática que contendrá el valor buscado. Dicha constante está declarada dentro de la clase de C++ “UnsignedBitWidth”, como se muestra en el Código 40 y cuya declaración se encuentra en el fichero “x_hls_utils.h” provisto por Xilinx.

```
template  < unsigned int _Num, unsigned int _I=_Num/2>
class UnsignedBitWidth
{
public:
    static const unsigned int Value = 1 +
UnsignedBitWidth<_Num,_I/2>::Value;
};

template <unsigned int _Num>
class UnsignedBitWidth<_Num, 0>
{
public:
    static const unsigned int Value = 1;
};
```

Código 40: Declaración del atributo “Value” de la clase “UnsignedBitWidth”

Esta API se utiliza, combinada con la declaración “ap_uint<W>”, para construir una función macro denominada “_AP_UINT_FOR_MAX_VAL”. Dicha función establecerá, en múltiples secciones del código y de manera automática, el tipo de variable sin signo requerido con el mínimo tamaño en *bits* que la implementación *hardware* de cada variable

requiere. La declaración de esta macro se detalla en el Código 41, donde "_val" es el parámetro que representa el mayor valor que se podrá requerir que la variable contenga.

```
#define _AP_UINT(_len) ap_uint<_len>

#define _U_BITS_IN_VAL(_val) UnsignedBitWidth<_val>::Value

#define  AP UINT FOR MAX VAL( val)  AP UINT( U BITS IN VAL( val))
```

Código 41: Función macro "_AP_UINT_FOR_MAX_VAL"

Por otra parte, se utilizarán las clases "ap_ufixed" y "ap_fixed" para especificar las variables con y sin signo respectivamente, con decimales y de precisión arbitraria que se requieran. Dichas clases son especificadas por Xilinx en C++, siendo descritas en el archivo "ap_fixed.h". Asimismo, este tipo de clase se han utilizado, junto con los demás recursos de programación provistos por Xilinx, para crear múltiples funciones macros especificadas dentro del archivo de la aplicación "datatype.h". Estas funciones cumplen el objetivo de ajustar el tamaño, tanto de la parte entera como decimal, de múltiples variables declaradas dentro de la descripción HLS de los *kernels* FPGA.

Estos ajustes se efectúan de manera que garanticen que las variables puedan cumplir con los requisitos de almacenamiento de valores y resultados que los *kernels* requieran y, al mismo tiempo, procurarán que estos tamaños sean los mínimos indispensables.

Entre estas funciones se encuentran las siguientes:

- Función _SUM_AP_UFIXED(_in_int_size, _in_dec_size, _iters). Esta función se utiliza para establecer el tipo de variable de precisión arbitraria a utilizar en aquellas variables encargadas de almacenar el sumatorio de "_iters" variables de precisión arbitraria que poseen cada una como máximo "_in_int_size" *bits* de parte entera e "_in_dec_size" *bits* de parte decimal. En el Código 41 se ejemplifica la utilización de esta función para el tipo de variable "filter_sum_unsized_d". Este tipo de variable se asigna a los registros que deben almacenar la sumatoria de "_AVERAGING" datos del tipo "unsized_d".

```
typedef _SUM_AP_UFIXED(_INT_PART,_DEC_PART,_AVERAGING)
filter_sum_unsized_d;
```

Código 42: Utilización de función macro "_SUM_AP_UFIXED"

- Función _MULT_NORM_AP_UFIXED(_in_dec_size, _iters). Esta función se aplica para configurar el tipo de variable de precisión arbitraria necesario para aquellas variables que almacenen el producto de "_iters" variables de precisión arbitraria de valor normalizado (valor entre 0 y 1) y que poseen cada una como máximo "_in_dec_size" *bits* decimales. La declaración de esta función macro se muestra en el Código 43.

```
#define _MULT_RES_SIZE(_in_size, _iters) (_in_size*_iters)

#define _MULT_NORM_AP_UFIXED(_in_dec_size, _iters) _AP_UFIXED(1, \
_MULT_RES_SIZE(_in_dec_size, _iters))
```

Código 43: Declaración de la función macro "_MULT_NORM_AP_UFIXED"

8.4. Conversión de datos entre el *host* y el *kernel*

Debido al soporte *hardware,* el tamaño posible de las variables en el *host* se encuentra limitado en comparación con las posibilidades de los *kernels* FPGA. Además, el código del *kernel* utiliza variables con aritmética de punto fijo, las cuales no existen como variables primitivas en la programación C/C++ del *host*.

En este sentido, por cada tipo de variable utilizada para definir parámetros de entrada o salida de los *kernels* se crearán tres tipos de variables distintos. Desde la perspectiva de la nomenclatura, estos tipos variables se distinguen entre ellos añadiendo un "prefijo" a sus nombres:

- Tipo de variable con el prefijo "hst_": Identifica el tipo de variable con el que se encuentran almacenadas en el *host* los datos que son transmitidos y/o recibidos por los *kernels* FPGA.
- Tipo de variable con el prefijo "trans_": Identifica a los tipos de variables que se emplean como intermediarias entre el *host* y el *kernel* FPGA.

- Tipo de variable sin prefijo: Identifica a los tipos de variables con el que se utilizan y procesan internamente los datos en los *kernels* FPGA.

En el Código 44 se ejemplifica la creación de estos "subtipos" de variables para el caso del tipo de variable "unsized_d". En dicho ejemplo, "_DAT_SIZE" representa la longitud en *bits* de "unsized_d".

```
typedef _AP_UFIXED(_INT_PART, _DEC_PART) unsized_d;
#if _DAT_SIZE <=8
      typedef unsigned char hst_unsized_d;
      typedef unsigned char trans_unsized_d;
#elif _DAT_SIZE <=16
      typedef unsigned short hst_unsized_d;
      typedef unsigned short trans_unsized_d;
#elif _DAT_SIZE <=32
      typedef unsigned int hst_unsized_d;
      typedef unsigned int trans_unsized_d;
#elif _DAT_SIZE <= 64
      typedef unsigned long long hst_unsized_d;
      typedef unsigned long long trans_unsized_d;
#else
      #error Tamaño en bits de cada dato de la imagen muy grande
#endif
```

Código 44: Subtipos de variables para la conversión de datos "unsized_d"

8.5. *Kernels* FPGA

A continuación, se describen distintos aspectos relacionados con el diseño de los *kernels* de aceleración implementados en la FPGA.

8.5.1. Inicialización de los *kernels* desde el *host* y configuración de las comunicaciones *host-kernel*

La programación de la FPGA con el *bitstream* que corresponda en función del *kernel* a utilizar se invoca desde el *host* por medio de la API de OpenCL. Esta operación constituye parte del proceso por el que se inicializa la utilización de cada *kernel* FPGA. Otros procesos de la inicialización consisten en establecer mecanismos de sincronización y comunicación entre el *host* y los *kernels* FPGA. Desde la perspectiva de la programación del *host*, se recurre a la creación y utilización de objetos **cl::Event** para la sincronización de las operaciones entre el *host* y los *kernels*.

Por otra parte, la comunicación entre el *host* y los *kernels* se gestionan mediante objetos **cl::CommandQueue** y **cl::Buffer**. En el archivo "recOpenCL.cpp" se describen las

funciones a utilizar en la inicialización de estos aspectos desde el *host*. Entre estas funciones se encuentra una denominada "init", cuya programación se expone en el Código 45.

```
void init(){
	cl_int err;
	device = get_xilinx_device();
	OCL_CHECK(err, context=cl::Context(device, nullptr, nullptr,
nullptr, &err));
	OCL_CHECK(err, q_in  = cl::CommandQueue(context, device,
CL_QUEUE_PROFILING_ENABLE, &err));
	OCL_CHECK(err, q_exe = cl::CommandQueue(context, device,
CL_QUEUE_PROFILING_ENABLE, &err));
	OCL_CHECK(err, q_out = cl::CommandQueue(context, device,
CL_QUEUE_PROFILING_ENABLE, &err));

	//Crea un binario por cada configuracion (.xclbin) de la FPGA
	for(unsigned char k=0; k < _N_BINARIOS; k++){
		setBinary(xclbinNames[k].c_str(), k);
	}
}
```

Código 45: Función "init"

El propósito de esta función consiste en inicializar aspectos comunes a todos los *kernels* FPGA a utilizar e invocar desde el *host*. Esto incluye la inicialización de los objetos de OpenCL de las clases **cl::Device**, **cl::Context**, **cl::CommandQueue**. También se incluye en dicha función el establecimiento del acceso por parte del *host* a los archivos binarios con los que se debe programar la FPGA en función del *kernel* a utilizar. Cada uno de estos binarios se identifica como un objeto **cl::Program::Binaries** y su compilación se efectúa juntamente con la aplicación.

Por otra parte, se dispone para cada *kernel* FPGA de una función encargada de inicializar recursos y ejecutar operaciones de programación específicos para cada *kernel*. Estos aspectos incluyen, en primer lugar, la inicialización de los recursos de OpenCL responsables de la programación de la FPGA con el binario que corresponda mediante el objeto **cl::Program**; y de establecer la identificación del *kernel* en el código del *host*, mediante el objeto **cl::Kernel**. Posteriormente, se especifica en estas funciones la inicialización de los objetos **cl::Buffer** y la asociación de estos al parámetro del *kernel* que corresponda mediante la función **setArg** del objeto **cl::Kernel**.

A continuación, se crean los objetos **cl::Event** necesarios para identificar los comandos de OpenCL que el *host* envía durante la utilización del *kernel* FPGA. Estos objetos **cl::Event** se agrupan en 2 o 3 vectores:

- El primer vector se corresponde a los eventos **cl::Event** asociados a comandos de introducción de datos de entrada en los *kernels* FPGA. Cabe recordar que, en esta aplicación, estos comandos son encolados por la función **enqueueWriteBuffer**. Este tipo de vector se identifica con la terminación "InEvs".

- El segundo vector de objetos **cl::Event**" se asigna a los eventos destinados a comandos de OpenCL para la invocación de la ejecución de los *kernels* FPGA. Dichos comandos se encolan al invocar la función **enqueueTask**. Los vectores utilizados para este propósito poseen un nombre identificativo que termina en "ExeEvs".

- El último vector se corresponde a eventos asociados a comandos de OpenCL para el envío al *host* de datos calculados por el *kernel* FPGA, es decir, comandos encolados al invocar la función **enqueueReadBuffer**. Los vectores empleados para esta función se identifican con un nombre que termina en "OutEvs".

Si el *kernel* FPGA devuelve un solo parámetro o *array* a su salida, el objeto **cl::Event** que referencia su comando de extracción se especifica en el código fuente como independiente, sin introducirse en ningún vector. En este último caso, el nombre asignado en el código fuente al objeto termina en "OutEv".

Por último, en estas funciones se ordena la transferencia desde el *host* hacia el *kernel* FPGA de aquellos parámetros cuyo valor no se cambiarán durante las sucesivas ejecuciones efectuadas por el *kernel* FPGA con los datos de una misma imagen hiperespectral. En el Código 46 se muestra la codificación de la función de inicialización para la etapa de filtrado.

```
void initFilter(hst_n_filters_frame_id lastFrame, hst_band_id
nBands){
  cl_int err;
  //-----Se configura FPGA para el kernel "preproc"------
  std::cout << "Programando dispositivo " <<
  device.getInfo<CL_DEVICE_NAME>() <<" con "<<
  xclbinNames[_FILTER_BIN].c_str() << std::endl;
  program[_FILTER_BIN] = cl::Program(context, {device},
                         bins[_FILTER_BIN], nullptr, &err);
  std::cout <<"Programacion del dispositivo terminada"<<std::endl;
  //Crear kernels
  krnls[_FILTER_KRNL] = cl::Kernel(program[_FILTER_BIN],
                        "multFilter", &err);
  //Creacion del buffer de datos de la imagen de entrada
  const size_t DATA_ARRAY_BYTES =
_N_FILTERS*sizeof(trans_unsized_d);
  filter_data = cl::Buffer(context, CL_MEM_READ_ONLY,
                DATA_ARRAY_BYTES, nullptr, &err);
  filter_nBands = cl::Buffer(context, CL_MEM_READ_ONLY,
                  sizeof(trans_band_id), nullptr, &err);
  filter_lastFrame = cl::Buffer(context, CL_MEM_READ_ONLY,
           sizeof(trans_n_filters_frame_id), nullptr, &err);
  //Creacion del buffer de datos de la imagen de salida
  filter_output = cl::Buffer(context, CL_MEM_WRITE_ONLY,
  DATA_ARRAY_BYTES, nullptr, &err);
  //Creacion del buffer de minimos delos pixeles de salida
  filter_min = cl::Buffer(context, CL_MEM_WRITE_ONLY,
                       DATA_ARRAY_BYTES, nullptr, &err);
  filter_scale = cl::Buffer(context, CL_MEM_WRITE_ONLY,
                       DATA_ARRAY_BYTES, nullptr, &err);

  //Asociacion variables argumento al multFilter
  err=krnls[_FILTER_KRNL].setArg(0, filter_data);
  err=krnls[_FILTER_KRNL].setArg(1, filter_nBands);
  err=krnls[_FILTER_KRNL].setArg(2, filter_lastFrame);
  err=krnls[_FILTER_KRNL].setArg(3, filter_output);
  err=krnls[_FILTER_KRNL].setArg(4, filter_min);
  err=krnls[_FILTER_KRNL].setArg(5, filter_scale);

  //Creacion de Eventos
  filterInEvs.push_back(cl::Event());
  filterInEvs.push_back(cl::Event());
  filterInEvs.push_back(cl::Event());
  filterExeEvs.push_back(cl::Event());
  filterOutEvs.push_back(cl::Event());
  filterOutEvs.push_back(cl::Event());
  filterOutEvs.push_back(cl::Event());

  err=q_in.enqueueWriteBuffer(filter_nBands, CL_FALSE, 0,
  sizeof(trans_band_id), &nBands, nullptr, &(filterInEvs[1]));
  err=q_in.enqueueWriteBuffer(filter_lastFrame, CL_FALSE, 0,
  sizeof(trans_n_filters_frame_id), &lastFrame, nullptr,
  &(filterInEvs[2]));
  err=q_in.finish();
}
```

Código 46: Función de inicialización de la etapa de filtrado

8.5.2. Interfaces de entrada/salida

Para esta aplicación es importante diferenciar entre dos grados de paralelismo, presentes para cualquiera de los *kernels* FPGA:

1. Paralelismo de las transferencias de datos. Se refiere a cuántos datos pueden transferirse simultáneamente entre el *host* y la FPGA. Esta limitación viene establecida por las interfaces físicas a utilizar en la intercomunicación de estos dos componentes de la aplicación, así como por la configuración que se establezca para dichas interfaces.
2. Paralelismo del procesamiento. Indica cuántos datos es capaz de procesar simultáneamente el *kernel* FPGA, sin incluir el *pipelining*.

Dichos grados de paralelismo no coincidirán necesariamente en los *kernels*. Por ello, la descripción HLS de cada uno de los *kernels* dispondrá de un archivo con la terminación "_interface.cpp" donde, entre otros aspectos, se ajustan y correlacionan el paralelismo de la transferencia, el paralelismo del procesamiento y el *pipelining* de *kernel* FPGA para el que se asigne dicho fichero. Además, el objetivo, desde la perspectiva del código fuente, es que estos aspectos se ajusten con un elevado grado de automatización. Esto quiere decir que el usuario solo tendrá que establecer el grado de paralelismo del procesamiento deseado y el formato de los datos de entrada y salida del *kernel*, ajustándose los demás aspectos de manera óptima sin necesidad de que el usuario realice modificaciones adicionales del código fuente del *kernel*.

Otros aspectos detallados en los archivos terminados en "_interface.cpp" son las operaciones intermedias entre la llegada de los datos a procesar a la entrada del *kernel* y el procesamiento de dichos datos. Análogamente, se detallan las operaciones a efectuar con los datos de salida desde que estos son generados dentro del *kernel*, hasta que estos se establecen en los puertos de salida preparados para su transferencia hacia el *host*. Dichas operaciones suelen consistir en cambios en el formato de los datos y en la introducción o extracción de datos de FIFOs. Para el caso del *kernel* de filtrado, en el Código 47 se muestra el tratamiento de los datos a filtrar previo a su procesamiento. Por otra parte, en el Código 48 se expone la preparación de los datos filtrados para su envío desde el *kernel* hacia el *host*.

```
void filterToStr(trans_unsized_d data[_TRANSFER_N_DATA],
preproc_FIFO data_str[_N_FILTERS]){
#pragma HLS FUNCTION_INSTANTIATE variable=data
#pragma HLS FUNCTION_INSTANTIATE variable=data_str
#pragma HLS INLINE
#pragma HLS array_partition variable=data cyclic
factor=C_PREPROC_DATA_PER_HW_TRANSFER
#pragma HLS array_partition variable=data_str complete

  static unsized_d loc_data[_TRANSFER_N_DATA];
  #pragma HLS array_partition variable=loc_data complete

  //Extraccion de los datos de la trama recibida desde el host
  LOOP_PREPROC_TO_STR_I: for(transfer_id idx=0;
                             idx<TRANSFER_N_DATA; idx++){
  #pragma HLS UNROLL factor=C_PREPROC_DATA_PER_HW_TRANSFER
skip_exit_check
  #pragma HLS DEPENDENCE class=array type=inter dependent=false
  filterConvertInputUnsized(data[idx], &(loc_data[idx]));
  }

  //Introducción de los datos en la FIFO de entrada al kernel
  LOOP_PREPROC_TO_STR_II: for(filter_id k=0; k<N_FILTERS; k++){
  #pragma HLS UNROLL
    filterWriteInStr(data_str[k], loc_data[k]);
  }
}
```

Código 47: Función HLS para introducir los datos a filtrar en el kernel de filtrado

```
void filterFromStr(preproc_FIFO output_str[_N_FILTERS],
trans_unsized_d output[_TRANSFER_N_DATA]){
#pragma HLS array_partition variable=output_str complete
#pragma HLS array_partition variable=output cyclic \
factor=C PREPROC DATA_PER_HW_TRANSFER
#pragma HLS INLINE
  static unsized_d loc_output[_TRANSFER_N_DATA];
  #pragma HLS array_partition variable=loc_output complete

  //Lectura no bloqueante de datos de la FIFO de salida
  LOOP_PREPROC_FROM_STR_I: for(filter_id idx=0;
                               idx<N_FILTERS; idx++){
  #pragma HLS UNROLL
    filterNonBlockingReadFromStr(output_str[idx], loc_output[idx]);
  }

  //Conversion "sized_d" -> "trans_sized_d"
  LOOP_PREPROC_FROM_STR_II: for(transfer_id idx=0;
                                idx<TRANSFER_N_DATA; idx++){
  #pragma HLS UNROLL factor=C_PREPROC_DATA_PER_HW_TRANSFER \
  skip_exit_check
  #pragma HLS DEPENDENCE class=array type=inter dependent=false
    filterConvertOutputUnsized(loc_output[idx], &(output[idx]));
  }
}
```

Código 48: Funcion HLS para preparar los datos filtrados para su envío al host

En esta aplicación, se utilizan interfaces “m_axi” para la transferencia entre el *host* y la FPGA. Para su configuración, cabe recordar que cada interfaz “m_axi” tiene limitado a 512 *bits* el máximo grado de paralelización de las transferencias entre el *host* y el *kernel*. Debido a ello, si se requiere transferir un *array* de datos de más de 512 *bits* a procesar y generar para la salida de manera completamente paralela dentro del *kernel*, será necesario dividir la transferencia de los arrays en varios ciclos de reloj, ajustando el intervalo de *pipelining* del *kernel* FPGA a como mínimo dicha cantidad.

Para minimizar la latencia de la transferencia de los *arrays*, debe tenerse en cuenta el criterio por defecto que sigue el compilador HLS para establecer el ancho de las interfaces “m_axi”. Asimismo, se ha observado que, si el tamaño en *bits* del *array* es inferior a 512 *bits*, el ancho del puerto “m_axi” asignado al *array* se establece al tamaño mínimo para asegurar la transferencia en 1 solo ciclo de reloj del *array*. Cabe destacar que el ancho del canal de transmisión de datos de estas interfaces solo es ajustable exclusivamente a las configuraciones de 32, 64, 128, 256 o 512 *bits* [57] según la especificación AXI4.

Por otra parte, si el tamaño del *array* es superior a 512 *bits*, el puerto “m_axi” se configurará con el mayor ancho de transferencia que sea divisor de la longitud del *array* a transmitir. En consecuencia, el intervalo de *pipelining* del *kernel* se limita en su valor mínimo al cociente entre el ancho en *bits* del *array* y el ancho del canal para la transferencia de los datos del puerto físico. Cabe destacar que, desde la perspectiva de maximizar el *throughput*, esta configuración no sería la deseable. Esto se debe a que si en su lugar se establece el ancho del canal para la transferencia de datos a su máximo valor se puede reducir la cantidad de transferencias entre *host* y el *kernel* necesarias para transmitir la totalidad de un *array*. En consecuencia, se reduce la cantidad de ciclos de reloj que requiere la transferencia del *array*. Con ello, se posibilita un *pipelining* en la ejecución del *kernel* menor, siempre y cuando no existan otras cuestiones internas de la implementación del *kernel* FPGA que restrinjan las posibilidades de *pipelining*.

Para lidiar con esta situación, se recurre a sobredimensionar el *array* al valor múltiplo del ancho máximo del puerto “m_axi” inmediatamente superior al tamaño útil y requerido del *array*. Dicho sobredimensionamiento no ocurre cuando el *array* posea una longitud inferior al ancho máximo de la transferencia de datos del puerto “m_axi”.

Este proceso se ejecuta de manera automatizada por parte del compilador HLS. Un ejemplo del código en C/C++ especificado para este proceso se detalla en el Código 49. En dicho código, la cantidad de datos numéricos de los *arrays* transferidos entre el *host* y el *kernel* se identifica como la macro "_TRANSFER_N_DATA".

Cabe destacar que "_ROUND_UP" constituye una función macro declarada para esta aplicación en el archivo "datatype.h" del código fuente. Dicha función establece que el valor del primer parámetro se redondee al múltiplo de valor superior más cercano del segundo parámetro. Asimismo, la macro "_PREPROC_PARALLEL_DATA" equivale a la macro "_N_FILTERS", y "_HST_DAT_SIZE" al tamaño en *bits* de los datos transferidos. El Código 49 se desarrollará análogamente para cada *kernel*.

```
#define _PREPROC_BITS_PER_TRANSFERS (_HST_DAT_SIZE* \
_PREPROC_PARALLEL_DATA)

#if _PREPROC_BITS_PER_TRANSFERS <= 512
      #if  PREPROC BITS PER TRANSFERS <= 32
            #define _PREPROC_AXI_WORD  32
      #elif _PREPROC_BITS_PER_TRANSFERS <= 64
            #define _PREPROC_AXI_WORD  64
      #elif _PREPROC_BITS_PER_TRANSFERS <= 128
            #define _PREPROC_AXI_WORD  128
      #elif _PREPROC_BITS_PER_TRANSFERS <= 256
            #define _PREPROC_AXI_WORD  256
      #else
            #define _PREPROC_AXI_WORD  512
      #endif
      #define _PREPROC_DATA_PER_HW_TRANSFER _N_FILTERS
      #define _TRANSFER_N_DATA _N_FILTERS
#else
      #define _PREPROC_AXI_WORD  512
      #define _PREPROC_DATA_PER_HW_TRANSFER (_PREPROC_AXI_WORD /\
      _HST_DAT_SIZE)
      #define _TRANSFER_N_DATA _ROUND_UP(_N_FILTERS, \
      _PREPROC_DATA_PER_HW_TRANSFER )
      #warning Datos 'data' y 'output' del kernel 'preproc' no \
      transferibles en un ciclo de reloj
#endif
```

Código 49: Configuración del tamaño de los arrays de transferencia entre el host y el kernel

Para minimizar el intervalo temporal entre transferencias de fragmentos sucesivos de un mismo *array* a un ciclo de reloj, se recurre a la configuración del modo *burst* de los puertos "m_axi" a utilizar. En este sentido, cabe recordar que la interfaz "m_axi" presenta dos modos *burst* de configuración independiente. Dichos modos se corresponden uno a la lectura y el otro a la escritura de datos desde la perspectiva del *kernel*. Para su

configuración, se recurre a las opciones "max_read_burst_length" y "max_write_burst_length" de la directiva **#pragma HLS INTERFACE** provista por Xilinx. El uso de esta directiva se muestra en el Código 50 [63].

```
#pragma HLS INTERFACE m_axi port=data bundle=gmem \
max_read_burst_length=C_PREPROC_BURST \
max_write_burst_length=C_PREPROC_BURST
```

Código 50: Configuración del burst para una interfaz "m_axi"

Independientemente del ancho del canal de datos de la interfaz "m_axi", la cantidad de datos transferible en modo *burst* se encuentra limitada a 4 KB [57].

Otro aspecto destacable con respecto a la transmisión/recepción de datos recurrente entre el *host* y los *kernels* consiste en que los *kernels* se han diseñado para operar siguiendo un modo de funcionamiento de *streaming*. Esto implica que cada *kernel* leerá, separado en fragmentos, los datos de la imagen hiperespectral, generando la salida correspondiente a cada fragmento de datos. La cantidad de datos de estos fragmentos coincidirá, en cada *kernel*, al grado de paralelización del procesamiento que se haya establecido.

El funcionamiento en *streaming* de los *kernels* se ha establecido para no precisar que la totalidad de los datos de la imagen hiperespectral, así como los datos de salida que con ellos se generan, sean almacenados en la FPGA. Con ello, se logra minimizar la utilización de recursos de memoria de la FPGA, especialmente en imágenes de grandes dimensiones. La decisión de usar el funcionamiento en modo *streaming* evita que el aumento de la ocupación de recursos de PL, en función de las dimensiones máximas de la imagen, limiten el grado de ejecución en paralelo implementable en los *kernels* FPGA, así como las dimensiones de dicha imagen hiperespectral a procesar.

Por otra parte, el funcionamiento en *streaming* permite lograr un mayor grado de concurrencia entre las ejecuciones de estos *kernels* y del *host*.

8.5.3. Implementación de la configuración estática

La configuración estática de los *kernels* FPGA se establece mediante macros de C. Estas macros se utilizan de manera directa para la configuración de macros secundarias, el dimensionamiento de tipos de variables, y el ajuste de la compilación y ejecución del *host*.

Desde la perspectiva de la nomenclatura, todas estas macros tienen la característica común de que sus nombres vienen precedidos por guion bajo "_". En múltiples ocasiones, estas macros se transforman en el código fuente en una constante con tipo de variable. Estas versiones transformadas se identifican y correlacionan con la macro en función de sus nombres de la siguiente manera:

- Constante con el nombre de la macro sin "_". El valor de esta constante es equivalente al de la macro de C. No obstante, el tipo de esta constante es el empleado para variables enteras sin signo en la descripción HLS de los *kernels*, dimensionada a la cantidad mínima de *bits* necesaria para representar el valor de esta macro. Se utilizan en la descripción en C de los *kernels* FPGA. En el Código 51 se expone la creación de este tipo de constante para la macro "_TRANSFER_N_DATA".

```
typedef _AP_UINT_FOR_MAX_VAL(_TRANSFER_N_DATA) transfer_id;
const transfer_id TRANSFER_N_DATA = (transfer_id)_TRANSFER_N_DATA;
```

Código 51: Constante HLS de precisión arbitraria para la macro "_TRANFER_N_DATA"

- Constante con el nombre de la macro prefijado con "C". Esta constante coincide en valor con la macro de C. Sin embargo, el tipo de esta constante se corresponde al tipo nativo de C sin signo que posea el dimensionamiento mínimo y necesario para almacenar el valor de la macro. Estas constantes se utilizan en el *host* y en la configuración de las directivas **#pragma HLS**. La creación de esta constante se ejemplifica en el Código 52 para el caso de la macro "_FILTER_II" correspondiente al intervalo de iniciación en el *pipelining* del *kernel* de filtrado.

```
#define _FILTER_II  ((_PREPROC_BURST > \
_MIN_FILTER_II)?_PREPROC_BURST:_MIN_FILTER_II)
#if _FILTER_II < 1<<8
      typedef unsigned char c_filterII_id;
#elif _FILTER_II < 1<<16
      typedef unsigned short c_filterII_id;
#elif _FILTER_II < 1<<32
      typedef unsigned int c_filterII_id;
#elif _FILTER_II <= ~(0L)
      typedef unsigned long long c_filterII_id;
#else
      #error La cantidad de ciclos de reloj que separan ejecuciones \
            del kernel "multFilter" sucesivas (_FILTER_II) es excesiva
#endif

const c_filterII_id C_FILTER_II = (c_filterII_id)_FILTER_II;
```

Código 52: Generación de la constante de C para la macro "_FILTER_II"

8.5.4. Particionado de los *arrays*

En la mayor parte de las ocasiones, los *arrays* constituyen registros en los que la totalidad de los datos deben ser escritos y/o leídos de manera simultánea y en una misma ejecución del *kernel* FPGA. Por este motivo se asigna, a la mayoría de los *arrays* de las descripciones HLS, un particionado completo.

Por el contrario, algunos de estos *arrays* son escritos y/o leídos por segmentos de datos contiguos. Para ello, se utiliza el particionado cíclico, siendo el factor de particionado el grado acceso paralelo requerido por la ejecución. En algunas ocasiones puntuales, existen múltiples secciones de la descripción HLS que precisan acceso simultáneo a un mismo *array*, siendo los grados de paralelización que necesitan diferentes. En este último caso, se establece un factor de particionado igual al mayor grado de acceso en paralelo de datos contiguos del *array* que precise el *kernel* FPGA.

8.5.5. Medición de los tiempos de ejecución y transferencia

Un aspecto clave para evaluar el rendimiento de un acelerador *hardware* consiste en determinar el tiempo que transcurre tanto en la ejecución de los *kernels* que se especifiquen como en las transferencias entre el *host* y dicho acelerador *hardware*. Para ello, pueden utilizarse objetos **cl::Event** y funciones *callback* como se indica en el apartado 6.9.1. Dentro de estos *callbacks*, puede recurrirse a la función **clGetEventProfilingInfo**, como se expone en el apartado 6.9.3.

Experimentalmente, se ha observado que la asignación reiterativa de *callbacks* y la escritura reiterativa de los resultados temporales obtenidos ralentizan la ejecución de la aplicación. Debido a ello, en la aplicación final solo se asignan los *callbacks* necesarios para medir el tiempo trascurrido entre 2 iteraciones consecutivas de un mismo *kernel* FPGA. Estos *callbacks* son los asociados a las transferencias hacia los *kernels* FPGA de fragmentos del hipercubo. Todos los *callbacks* desarrollados se describen en el archivo "recOpenCL.cpp". Las funciones *callback* utilizadas se asemejan a la función "filterDataSent" expuesta en el Código 53.

```
static cl_ulong filterInit=0;
void CL_CALLBACK filterDataSent(cl_event event, cl_int status,
                                void* data){
    static unsigned long long iteration = 0;
    static size_t sizeInitGot = 0;
    cl_ulong prevInit = filterInit;
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
                            sizeof(cl_ulong), &filterInit,
                            &sizeInitGot);

    if(sizeInitGot != sizeof(cl_ulong)){
        printf("Tamano del dato de instante de inicio no es "
               "sizeof(cl_ulong) = %lu\n", sizeof(cl_ulong));
        fprintf(logFile, "Tamano del dato de instante de inicio no "
                "es sizeof(cl_ulong) = %lu\n", sizeof(cl_ulong));
        exit(EXIT_FAILURE);
    }
    cl_ulong iterTime = filterInit - prevInit;
    fprintf(logFile, "Iteracion %llu de 'multFilter' tarda %lu ns\n",
            iteration, iterTime);
    iteration++;
}
```

Código 53: Ejemplo de función callback asignada a evento de OpenCL cl::Event

Como se observa en el Código 53, el tiempo de iteración de cada *kernel* FPGA se calcula como una diferencia entre instantes de inicio de transferencias hacia el *host* de fragmentos del hipercubo. Los instantes concretos comparados son el de la iteración actual del *kernel* FPGA, guardado en "filterInit", y el de la iteración anterior, guardado en "prevInit". En consecuencia, el tiempo calculado incluye tanto la ejecución del *kernel* FPGA como las transferencias de datos requeridas en la iteración evaluada entre el *host* y dicho *kernel*.

8.6. Etapa de preprocesamiento

Esta etapa se corresponde al calibrado, eliminación de bandas espectrales extremas residuales, filtrado y normalizado de la imagen hiperespectral. Para este proyecto, se considera que dicha imagen se encuentra calibrada con anterioridad a la ejecución de la aplicación. Para la implementación y ejecución de esta etapa se recurre a dos *kernels* FPGA, siendo uno de ellos responsable del filtrado de los datos de la imagen y el otro de su normalizado. Ambos *kernels* deben operar con un mismo grado de paralelización en su procesamiento. Para cambiar esta paralelización, el usuario debe modificar, únicamente, la macro _N_FILTERS presente en el archivo "amounts.h" y recompilar la aplicación.

Debido al funcionamiento en *streaming* y píxel a píxel del *kernel* de filtrado, los datos de la imagen hiperespectral deberán enviarse siguiendo una ordenación específica. Este orden consiste en que los datos se envíen en fragmentos o tramas de _N_FILTERS datos, donde cada una de estas tramas contendrá únicamente datos pertenecientes a una misma banda espectral de píxeles contiguos. El envío de estas tramas se sucederá recorriendo primero el espectro y luego las dimensiones espaciales de la imagen. En la Figura 42 se muestra el criterio de ordenación a seguir tomando como referencia el formato BSQ. En este ejemplo, se considera que la paralelización del procesamiento en el *kernel* de filtrado es de 6 datos. No obstante, el criterio de ordenación expuesto es análogo para cualquier grado de paralelización exigido.

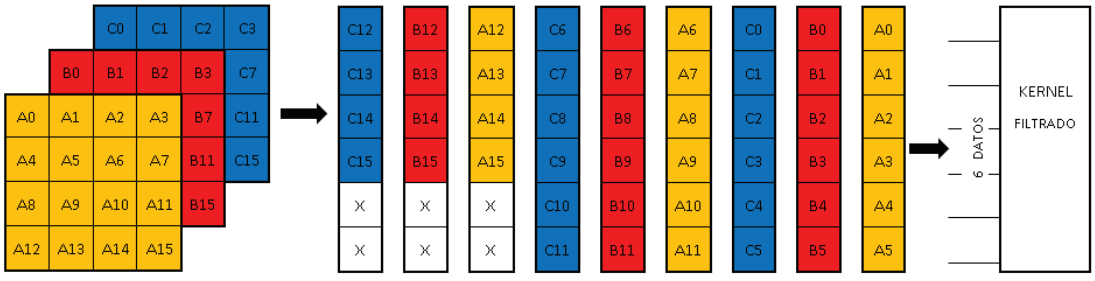


Figura 42: Ordenamiento de datos de la imagen hiperespectral para transferirlos al kernel de filtrado

El código fuente responsable de esta ordenación en el envío de los datos de la imagen hiperespectral se encuentra en el fichero "FromFileToFPGA", siendo la función principal para ello la función "transfer" (Código 54). Para el caso de las imágenes en formato BSQ, dicha función escoge la opción de la subfunción "transfer_BSQ_Preproc" (Código 55).

```
void transfer(FILE* src, hst_unsized_d* HSCube, hst_dim_d dims[3]){
  lastBand = (hst_band_id)(dims[2] - _QUIT_TOTAL_SPECTRA - 1);
  const hst_px_id N_PX = dims[0]*dims[1];
  //Si N_PX <= _N_FILTERS, la primera y última tanda de pixeles
  //evaluados coincide
  lastPxFrame = (N_PX > _N_FILTERS)?(N_PX - _N_FILTERS):0;
  //caso BSQ
  if(HS_IMAGE_FORMAT == 2){
    //Captura dato de relleno de datos inutiles si necesario
    #if _CLEAN_USELESS_DATA == 1
      const long long INC_BANDS = N_PX*sizeof(hst_unsized_d);
      const long long REJECTED_BLOCK=_QUIT_LOWER_SPECTRA*INC_BANDS;
      getFirstDat(src, REJECTED_BLOCK);
    #endif

    transfer_BSQ_Preproc(src, HSCube, dims);

  //caso BIL
  }else if(HS_IMAGE_FORMAT == 1){

    //Captura dato de relleno de datos inutiles si necesario
    #if _CLEAN_USELESS_DATA == 1
      const long long INC_BANDS = dims[0]*sizeof(hst_unsized_d);
      const long long REJECTED_BLOCK=_QUIT_LOWER_SPECTRA*INC_BANDS;
      getFirstDat(src, REJECTED_BLOCK);
    #endif

    if(dims[0] >= _N_FILTERS){
      if((dims[0] % _N_FILTERS) != 0){
        transfer_smallBIL_Preproc(src, HSCube, dims);
      }else{
        transfer_smallperRowBIL_Preproc(src, HSCube, dims);
      }
    }else{
      if((_N_FILTERS % dims[0]) != 0){
        transfer_bigBIL_Preproc(src, HSCube, dims);
      }else{
        transfer_bigperRowBIL_Preproc(src, HSCube, dims);
      }
    }

  //caso BIP
  }else if(HS_IMAGE_FORMAT == 0){
    //Captura dato de relleno de datos inutiles si necesario
    #if _CLEAN_USELESS_DATA == 1
      const long long INC_BANDS = sizeof(hst_unsized_d);
      const long long REJECTED_BLOCK=_QUIT_LOWER_SPECTRA*INC_BANDS;
      getFirstDat(src, REJECTED_BLOCK);
    #endif

    transfer_BIP_Preproc(src, HSCube, dims);

  //caso formato desconocido
  }else{
    printf("ERROR: Formato de imagen hiperespectral desconocido\n");
    exit(EXIT_FAILURE);
  }
```

```
}
```

Código 54: Función "transfer"

```
inline void transfer_BSQ_Preproc(FILE* src, hst_unsized_d* HSCube,
hst_dim_d dims[3]){
  /* Saltos unitarios (variables "inc_"), limites (variables "bound_"
   * ) y recorridos (variables "along_") en bytes de la dimension
   * espectral ("bands") o del conjunto de pixeles ("px").*/
  const unsigned int inc_px=sizeof(hst_unsized_d);
  const unsigned int inc_bands=dims[0]*dims[1]*inc_px;
  const unsigned int bound_px=inc_bands;
  const unsigned int along_bands=inc_bands*(dims[2]
                                  _QUIT_TOTAL_SPECTRA);
  //Tamano en bytes de las bandas espectrales de inicio rechazadas.
  const unsigned rejected_block = _QUIT_LOWER_SPECTRA*inc_bands;

  //Posicion en bytes de comienzo de lectura
  long long pos;
  fgetpos(src, (fpos_t*)&pos);
  pos+=rejected_block;//+offset inicio

  //Salto en bytes a la siguiente tanda de pixeles paralelos a
  //transmitir
  const unsigned int nxt_ppx = _N_FILTERS*inc_px;

  //Constantes para regular transiciones en bytes de avances entre
  //coordenadas.
  const int back_bands_nxt_ppx = nxt_ppx-along_bands;
  //Limite en bytes excedido cuando falte una última tanda
  const int last_ppx = bound_px - nxt_ppx + rejected_block;
  //Una iteracion por tanda de pixeles procesados en paralelo completa
  while(pos < last_ppx){
    const unsigned int bound_pos_atband = pos + along_bands;
    //Recorrido banda a banda
    do{
      fsetpos(src, (fpos_t*)&pos);
      /*Lectura del fichero de los pixeles de la tanda
       * para una sola banda.*/
      fread(_PTR_FILE_SAVE_READ, sizeof(hst_unsized_d), _N_FILTERS,
            src);
      //Intercambio y sincronizacion con la FPGA
      filterExchange(HSCube);
      //Continuar a la siguiente banda
      pos+=inc_bands;
    }while(pos<bound_pos_atband);
    //Continuar a la siguiente tanda
    pos+=back_bands_nxt_ppx;
  }
  //Caso especial última tanda
  //Tamano última tanda
  const unsigned int bound_k = (bound_px-pos+rejected_block)/inc_px;
  const unsigned int bound_pos_atband = pos + along_bands;
  //Recorrido banda a banda
  do{
    fsetpos(src, (fpos_t*)&pos);
    /*Lectura del fichero de los pixeles de la tanda
     * utiles para una sola banda.*/
```

```
    fread(_PTR_FILE_SAVE_READ, sizeof(hst_unsized_d), bound_k, src);
    /*Relleno datos inutiles del frame de N_FILTERS datos
    si necesario*/
    #if _CLEAN_USELESS_DATA == 1
      fillUselessDataFromNowNFILTERSFrame(bound_k);
    #endif

    //Intercambio y sincronizacion con la FPGA
    filterExchange(HSCube);

    //Continuar a la siguiente banda
    pos+=inc_bands;
  }while(pos < bound_pos_atband);
/* Envio tandas de _N_FILTERS datos que no completaron
* una tanda de _PREPROC_PARALLEL_DATA datos
*/
#if _PREPROC_PARALLEL_DATA > _N_FILTERS
  if(in_idx != 0){
    /* Relleno datos inutiles de la tanda de
     * _PREPROC_PARALLEL_DATA datos si necesario*/
    #if _CLEAN_USELESS_DATA == 1
      /*Caso cantidad de bandas utiles mayor o igual que la cantidad
de tandas
       * de _N_FILTERS datos contenidas en la trama de envio --> Se ha
       * invocado "fillUselessDataFromNowNFILTERSFrame" para todas las
tandas
       * de _N_FILTERS datos*/
      if((dims[2] - _QUIT_TOTAL_SPECTRA) >=
(_PREPROC_PARALLEL_DATA/_N_FILTERS)){
        fillUselessNFILTERSFrames(bound_k);
      /*Caso no se ha invocado "fillUselessDataFromNowNFILTERSFrame"
       *para todas las tandas de _N_FILTERS datos*/
      }else{
        fillUselessNFILTERSFrames(_N_FILTERS);
      }
    #endif //_CLEAN_USELESS_DATA == 1
    coreFilterExchange(HSCube);
    in_idx=0;
  }
#endif //_PREPROC_PARALLEL_DATA > _N_FILTERS
  //Espera de fin de las operaciones del kernel
  cl_int err;
  err=q_out.finish();
}
```

Código 55: Función "transfer_BSQ_Preproc"

Con la codificación especificada en este archivo, la aplicación está preparada para ajustarse dinámicamente a cualquiera de los formatos de imágenes hiperespectrales vistos en el apartado 3.3. (BIP, BIL y BSQ), consiguiendo en cualquiera de los casos que estos datos lleguen al *kernel* de filtrado en el orden adecuado. Además, el código de este fichero será el responsable de eliminar las bandas extremas no deseadas, para lo cual se limitará a no extraer los datos correspondientes del archivo ".dat" que contiene los datos de la imagen

y, por consiguiente, no enviándolos al *kernel*. Cabe recordar que la cantidad de bandas espectrales a eliminar de los extremos superior e inferior del rango espectral de la imagen se establecen en tiempo de compilación, siendo especificadas mediante constantes macro "_QUIT_UPPER_SPECTRA" y "_QUIT_LOWER_SPECTRA" respectivamente.

La extracción de los datos ordenadamente en esta etapa constituye una de las operaciones más complejas realizadas por el *host*. Asimismo, dada las limitaciones de paralelización que suele presentar el código ejecutado en el *host*, esta sección de la aplicación constituye el principal cuello de botella en el *throughput* de la aplicación. Por este motivo, se prestó especial atención en el código contenido en este fichero con el objetivo de minimizar el consumo de ciclos de reloj que su funcionamiento requiere. Para la extracción y posterior introducción en archivos de los datos utilizados y generados por esta aplicación se recurre a las funciones especificadas por la librería de C "stdio.h".

El filtrado de los datos de la imagen hiperespectral corresponde al *kernel* denominado "multFilter". Además, este *kernel* deberá determinar el valor filtrado mínimo y máximo de cada píxel, enviando tanto el valor mínimo como la diferencia entre el valor máximo y mínimo al *host*. La función HLS que concentra el cómputo de este *kernel* se denomina "filter", la cual se encuentra en el Código 56.

```
void filter(trans_unsized_d data[_TRANSFER_N_DATA], trans_unsized_d
output[_TRANSFER_N_DATA], trans_unsized_d min[_TRANSFER_N_DATA],
trans_unsized_d scale[_TRANSFER_N_DATA]){
#pragma HLS array_partition variable=data cyclic \
factor=C_PREPROC_DATA_PER_HW_TRANSFER
#pragma HLS array_partition variable=output cyclic \
factor = C_PREPROC_DATA_PER_HW_TRANSFER
#pragma HLS array_partition variable=min cyclic \
factor = C_PREPROC_DATA_PER_HW_TRANSFER
#pragma HLS array_partition variable=scale cyclic \
factor = C_PREPROC_DATA_PER_HW_TRANSFER
#pragma HLS INLINE

  static preproc_FIFO data_str[_N_FILTERS];
  #pragma HLS array_partition variable=data_str complete

  static preproc_FIFO output_str[_N_FILTERS];
  #pragma HLS array_partition variable=output_str complete

  static unsized_d loc_scale[_N_FILTERS];
  #pragma HLS array_partition variable=loc_scale complete

  static unsized_d loc_min[_N_FILTERS];
```

```
#pragma HLS array_partition variable=loc_min complete

if(bnd_idx < loc_nBands){
  filterToStr(data, data_str);
}
filterStr(data_str, rst_min, rst_max, output_str, loc_min,
          loc_scale);
```

Código 56: Función "filter"

Asimismo, todos los datos de salida generados por este *kernel* son utilizado en la etapa posterior de normalizado. Este *kernel* está preparado para operar con múltiples imágenes hiperespectrales de manera consecutiva y sin necesidad de recompilación para ello. Dichas imágenes pueden ser de distintas dimensiones siempre que no violen las restricciones estáticamente establecidas, como se indicó en el apartado 8.2. En la Figura 43 se muestra un esquema básico del *kernel* de filtrado.

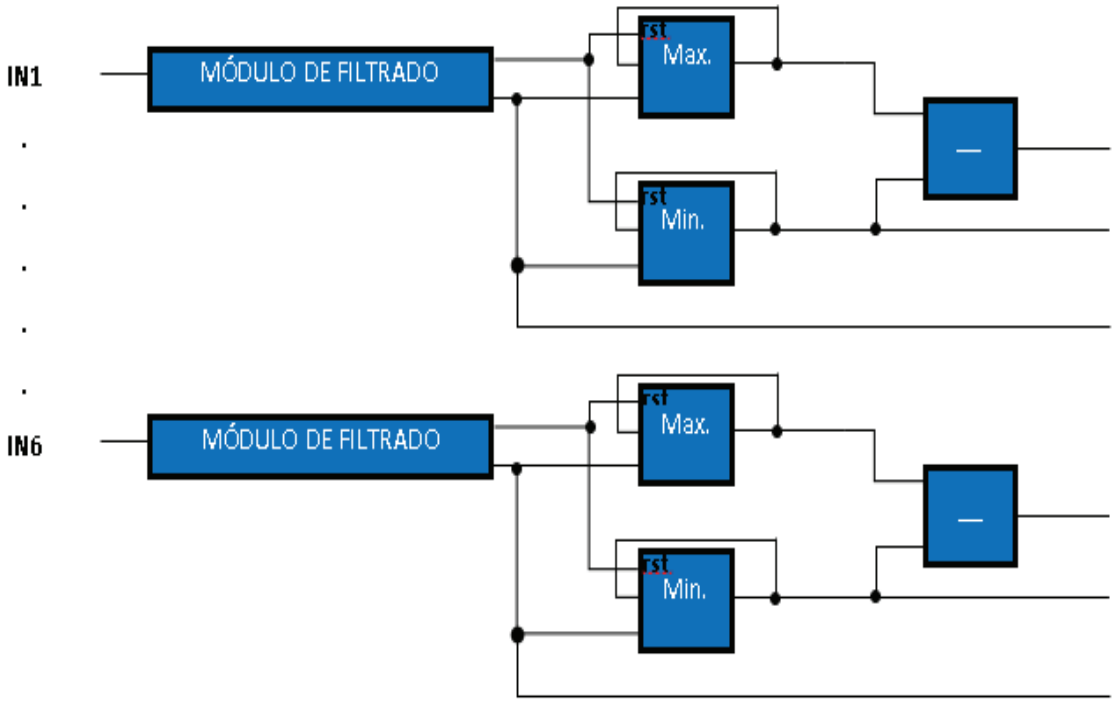


Figura 43: Esquema básico del kernel de filtrado

El módulo de filtrado se declarará en el código fuente como un objeto de la clase de C++ denominada "Filter" y creada para esta aplicación. En el *kernel* de filtrado, se implementan tantos módulos de filtrado como grado de procesamiento en paralelo se especifique para este *kernel*. Dicha clase posee una serie de atributos cuyos propósitos y significados funcionales se detallan en el propio código fuente de la aplicación. La

inicialización de estos atributos vendrá especificada en el constructor de la clase, siendo la detallada en el Código 57.

```
Filter(){
#pragma HLS array_partition variable=inner_reg complete

      filterIter=(band_id)0;
      dataInReg=(inner_reg_idx)0;
      sum_reg=(filter_sum_unsized_d)0;
      filterIterBelowAvg=true;
      newOutput=true;

      for(inner_reg_idx k=0; k<AVERAGING; k++){
      #pragma HLS UNROLL
            inner_reg[k] = (unsized_d)0;
      }
}
```

Código 57: Constructor de la clase "Filter"

Por otra parte, la ejecución del módulo de filtrado se detalla en la función miembro "filterRun" (Código 58). Cabe destacar que la operación de filtrado es análoga, desde el punto de vista funcional, al algoritmo de filtrado expuesto en el apartado 4.2. , devolviendo los mismos resultados que dicho algoritmo para las mismas entradas. Sin embargo, la forma en la que se concretará este algoritmo en su implementación se particulariza para operar sobre FPGA, optimizándose para dicho *hardware*.

```
void filterRun(preproc_FIFO& input, unsized_d* output,
               bool* out_ready, band_id nBands){
#pragma HLS FUNCTION_INSTANTIATE variable=input
#pragma HLS FUNCTION_INSTANTIATE variable=output
#pragma HLS FUNCTION_INSTANTIATE variable=out_ready
#pragma HLS INLINE

  const bool FILTER_ITER_OVER_LIM = filterIter >= nBands;
  const unsized_d NEW_VALUE = (FILTER_ITER_OVER_LIM)?
                                ((unsized_d)0):input.read();
  //Si primeras AVERAGING Bandas, sum_reg-=0
  shift_concurrente(NEW_VALUE);
  //Si filterIter >= nBands sum_reg+=0
  sum_reg+=NEW_VALUE;
  //filterIterBelowAvg= filterIter < AVERAGING
  if(filterIterBelowAvg){
    dataInReg++;
  }else if(FILTER_ITER_OVER_LIM){
    shift_concurrente((unsized_d)0);
    dataInReg-=2;
  }
  //Caso hay dato de salida a transmitir
  *output = sum_reg/dataInReg;
  *out_ready = newOutput;
```

```
  //Actualizacion 'filterIter'
  if(FILTER_ITER_OVER_LIM){
    //Reseteo 'filterIter' y 'dataInReg'
    if(dataInReg <= ((inner_reg_idx)1)){
      shift_concurrente((unsized_d)0);
      dataInReg=0;
      filterIter=0;
      filterIterBelowAvg=true;
      newOutput=true;
    }
  }else{
    if(filterIter < ULTIMO){
      newOutput = !newOutput;
      filterIterBelowAvg=true;
    }else{
      filterIterBelowAvg=false;
    }
    filterIter++;
  }
}
```

Código 58: Función "filterRun"

Para el funcionamiento de este módulo, será necesario que el mismo disponga de un registro que opere como una FIFO que acepte la entrada y salida simultánea de los datos que almacena. La utilización de este registro se especifica en el código fuente dentro de la función "shift_concurrente" englobada dentro de la clase "Filter" y cuyo código se adjunta en el Código 59.

```
void shift_concurrente(unsized_d input){
#pragma HLS INLINE
#pragma HLS array_partition variable=inner_reg complete
      static unsized_d temp_reg[_ULTIMO];
      #pragma HLS array_partition variable=temp_reg complete

      LOOP_SHIFT_CONCURRENTE_I: for(inner_reg_idx k=0; k<ULTIMO;
      k++){
      #pragma HLS UNROLL
            temp_reg[k]=inner_reg[k];
      }
      sum_reg -= inner_reg[ULTIMO];

      inner_reg[0] = input;
      LOOP_SHIFT_CONCURRENTE_II: for(inner_reg_idx k=0; k<ULTIMO;
      k++){
      #pragma HLS UNROLL
            inner_reg[k+1] = temp_reg[k];
      }
}
```

Código 59: Función "shift_concurrente"

Conectado a continuación de cada módulo de filtrado se encuentran dos módulos cuya finalidad consiste en determinar el valor máximo y mínimo del conjunto de datos de salida del módulo de filtrado al que se encuentran asociados. La descripción HLS de estos módulos se detalla en las funciones “minOp” y “maxOp” (archivo “GetMinAndScale.cpp”). El objetivo funcional de estos módulos consistirá en determinar los valores máximo y mínimos presentes en la versión filtrada de cada píxel. Asimismo, dado que los datos filtrados de un mismo píxel se generarán de manera serializada, el funcionamiento de estos módulos consiste en comparar el nuevo valor generado por el módulo de filtrado con el valor mínimo y máximo registrado por el módulo, actualizando los valores registrados cuando el resultado de la comparación así lo requiera.

Además, estos módulos vendrán provistos de un *reset* declarado en la descripción HLS, el cual debe activarse únicamente cuando los módulos comienzan la evaluación de un nuevo píxel. De estos datos se enviarán al *host* únicamente los mínimos calculados, los cuales se emplearán en el normalizado de los datos.

Por otra parte, los valores máximos y mínimos se utilizan en el *kernel* de filtrado para determinar los valores de escala de cada píxel. Estos valores se enviarán al *hos*t para utilizarse en la etapa de normalizado. Asimismo, su cálculo se corresponde, en el *kernel* de filtrado, al cálculo de la diferencia entre el valor máximo y mínimo de cada píxel. La excepción a esta situación ocurre cuando dichos valores máximo y mínimo coinciden para dicho píxel. En este último caso se establecerá a 1 el valor de escala. Esta última operación se describe, en la descripción HLS, en la función “getScale” definida en el archivo “GetMinAndScale.cpp”. Cabe destacar que este *kernel* puede operar con un intervalo de *pipelining* mínimo de 2, requiriendo para ello la utilización de 4 puertos físicos de interconexión entre el *host* y la FPGA.

Con respecto a la interfaz de entrada, la introducción de los datos del hipercubo a filtrar en el procesamiento del *kernel* de filtrado requiere de una serie de procedimientos adicionales una vez que los datos llegan a los puertos de entrada a dicho *kernel*. Dichos procedimientos se describen en la función “filterToStr” (Código 60), correspondiéndose a dos operaciones.

```
void filterToStr(trans_unsized_d data[_TRANSFER_N_DATA],
                 preproc_FIFO data_str[_N_FILTERS]){
#pragma HLS FUNCTION_INSTANTIATE variable=data
#pragma HLS FUNCTION_INSTANTIATE variable=data_str
#pragma HLS INLINE
#pragma HLS array_partition variable=data cyclic \
factor=C_PREPROC_DATA_PER_HW_TRANSFER
#pragma HLS array_partition variable=data_str complete

  static unsized_d loc_data[_TRANSFER_N_DATA];
  #pragma HLS array_partition variable=loc_data complete

  //Conversion "trans_unsized_d" -> "unsized_d"
  LOOP_PREPROC_TO_STR_I: for(transfer_id idx=0; idx<TRANSFER_N_DATA;
  idx++){
  #pragma HLS UNROLL factor=C_PREPROC_DATA_PER_HW_TRANSFER\
  skip_exit_check
  #pragma HLS DEPENDENCE class=array type=inter dependent=false
    filterConvertInputUnsized(data[idx], &(loc_data[idx]));
  }

  //Introduccion de los datos en la FIFO de entrada
  LOOP_PREPROC_TO_STR_II: for(filter_id k=0; k<N_FILTERS; k++){
  #pragma HLS UNROLL
    filterWriteInStr(data_str[k], loc_data[k]);
  }
}
```

Código 60: Función "filterToStr"

La primera operación consiste en cambiar el formato de los datos a filtrar de su formato de transferencia (tipo de variable "trans_unsized_d") al formato con el que se procesan dichos datos dentro del *kernel* (tipo de variable "unsized_d"). Esta operación se especifica en la descripción HLS para el caso de un solo dato de entrada por medio de la función "filterConvertInputUnsized".

La siguiente operación consiste en introducir los datos transformados dentro de la FIFO de entrada al *kernel* de filtrado que le corresponda. Esta operación se especifica con la función "filterWriteInStr".

Para la interfaz de salida del *kernel* de filtrado, se dispone en la descripción HLS de dos funciones diferentes. Una de estas funciones se denomina "filterFromStr" (Código 61), la cual se utiliza para preparar los datos filtrados del hipercubo que el *kernel* genera para su transferencia hacia el *host*.

```
void filterFromStr(preproc_FIFO output_str[_N_FILTERS],
                   trans_unsized_d output[_TRANSFER_N_DATA]){
#pragma HLS array_partition variable=output_str complete
#pragma HLS array_partition variable=output cyclic \
factor=C_PREPROC_DATA_PER_HW_TRANSFER
#pragma HLS INLINE
  static unsized_d loc_output[_TRANSFER_N_DATA];
  #pragma HLS array_partition variable=loc_output complete

  //Lectura no bloqueante de FIFOs de salida
  LOOP_PREPROC_FROM_STR_I: for(filter_id idx=0; idx<N_FILTERS;
                               idx++){
  #pragma HLS UNROLL
    filterNonBlockingReadFromStr(output_str[idx], loc_output[idx]);
  }

  //Conversion "unsized_d" -> "trans_unsized_d"
  LOOP_PREPROC_FROM_STR_II: for(transfer_id idx=0;
                                idx<TRANSFER_N_DATA; idx++){
  #pragma HLS UNROLL factor=C_PREPROC_DATA_PER_HW_TRANSFER \
  skip_exit_check
  #pragma HLS DEPENDENCE class=array type=inter dependent=false
    filterConvertOutputUnsized(loc_output[idx], &(output[idx]));
  }
}
```

Código 61: Función "filterFromStr"

Otra de las funciones se denomina "loadToOutput" (Código 62), la cual se emplea para preparar para la transferencia a los valores mínimo y de escala de los píxeles cada vez que se termina de procesar la totalidad de un conjunto de píxeles procesados en paralelo.

```
void loadToOutput(unsized_d fromKernel[_N_FILTERS], trans_unsized_d
output[_TRANSFER_N_DATA]){
#pragma HLS FUNCTION_INSTANTIATE variable=fromKernel
#pragma HLS FUNCTION_INSTANTIATE variable=output
#pragma HLS array_partition variable=fromKernel complete
#pragma HLS array partition variable=output cyclic \
factor=C_PREPROC_DATA_PER_HW_TRANSFER
#pragma HLS INLINE

  static unsized_d loc_fromKernel[_TRANSFER_N_DATA];
  #pragma HLS array_partition variable=loc_fromKernel complete

  //Transferencias registro -> registro
  LOOP_LOAD_TO_OUTPUT_I: for(filter_id idx=0; idx<N_FILTERS; idx++){
  #pragma HLS UNROLL
    filterRegToRegTransfer(fromKernel[idx], &(loc_fromKernel[idx]));
  }

  //Conversion "unsized_d" -> "trans_unsized_d"
  LOOP_LOAD_TO_OUTPUT_II: for(transfer_id idx=0;
                              idx<TRANSFER_N_DATA; idx++){
  #pragma HLS UNROLL factor=C_PREPROC_DATA_PER_HW_TRANSFER \
  skip_exit_check
  #pragma HLS DEPENDENCE class=array type=inter dependent=false
```

```
    filterConvertOutputUnsized(loc_fromKernel[idx], &(output[idx]));
  }
}
```

Código 62: Función "loadToOutput"

Para el caso de la función "filterFromStr", la primera operación consiste en extraer los datos filtrados a transferir en paralelo de las FIFOs en las que se introdujeron una vez fueron calculados, habiendo una asignación unívoca para cada dato. Esta extracción se detalla en la función "filterNonBlockingReadFromStr", consistiendo en una lectura no bloqueante en la FIFO.

La siguiente operación consiste en transformar los datos leídos de las FIFOs del formato en el que fueron originados (tipo de variable "unsized_d") al formato a utilizar para su transferencia (tipo de variable "trans_unsized_d"). Esta operación se corresponde en la descripción HLS a la función "filterConvertOutputUnsized". El caso de la función "loadToOutput" es semejante al de la función "filterFromStr". Sin embargo, la primera operación en esta función consistirá en transferir los datos a unos registros diferentes con la intención de asegurar que no se sobrescriban. Esta operación se detalla para el caso de un solo dato en la función "filterRegToRegTransfer".

La coordinación entre el *host* y el *kernel* de filtrado se contiene principalmente en la función "coreFilterExchange" (Código 63) descrita en el archivo "FromFileToFPGA.cpp".

```
inline void coreFilterExchange(hst_unsized_d* filtered_image){
  cl_int err;

  //Invocacion de la transferencia de datos desde el kernel
  //'multFilter' hacia el host
  err=q_exe.finish();
  //Espera fin envio de datos host-preproc
  err=q_in.enqueueWriteBuffer(filter_data, CL_FALSE, 0,
     BYTES_PER_TRANSFER, hst_filter_data, nullptr,
     &(filterInEvs[0]));
  err=filterInEvs[0].setCallback(CL_COMPLETE, &filterDataSent,
  nullptr);

  //Invocacion de ejecucion del kernel 'multFilter'
  err=q_out.finish();
  err = q_exe.enqueueTask(krnls[_FILTER_KRNL],
      (cl::vector<cl::Event>*)&filterInEvs, filterExeEvs.data());
  /*Solo se generan datos a la salida del 'multFilter' tras la
  ultima transferencia si se envia una banda posterior a la
  numero _AVERAGING, o por lo menos la última banda espectral
  enviada se encuentra en una posicion impar*/
```

```
  if((currentBand >= _ULTIMO)||((currentBand % 2) == 0)){
    //Invocacion de la transferencia de datos desde el kernel
    //'multFilter' hacia el host
    err = q_out.enqueueReadBuffer(filter_output, CL_FALSE, 0,
          BYTES_PER_TRANSFER, &(filtered_image[frame_preproc_idx]),
(cl::vector<cl::Event>*)&filterExeEvs, &(filterOutEvs[0])))

    frame_preproc_idx+=_N_FILTERS;

    //Caso ya se transfirio la última banda de la tanda pixeles en
    //filtrado
    if(currentBand >= lastBand){
      currentBand=0;
      lastBandsFromPx(filtered_image, minPerPx, scalePerPx);
    //Caso no se transfirio la última banda de la tanda pixeles en
    //filtrado
    }else{
      currentBand++;
    }
  }else{
    currentBand++;
  }

  /*Reapuntar el puntero 'hst_preproc_data' a la direccion
  * de escritura de la proxima tanda a enviar*/
  if(hst_filter_data != filter_inputs){
    hst_filter_data = filter_inputs;
  }else{
    hst_filter_data = &(filter_inputs[_N_FILTERS]);
  }
}
```

Código 63: Función "coreFilterExchange"

En dicha función puede observarse que siempre se efectúa una introducción de datos de la imagen hiperespectral y al menos una ejecución del *kernel* de filtrado. Sin embargo, la extracción de datos filtrados de la imagen hiperespectral a la salida del *kernel* no siempre se realiza para cada invocación de la función "coreFilterExchange". Esta situación se debe al algoritmo de filtrado que sigue el *kernel*, por lo que la generación de un dato filtrado de una determinada posición dentro de la firma espectral del píxel requiere datos del píxel sin filtrar que corresponden a posiciones posteriores dentro de la firma espectral.

Debido al algoritmo de filtrado se ocasiona un desfase, para cada tanda de píxeles procesados en paralelo por el *kernel* de filtrado, entre el índice de los datos no filtrados y extraídos de la imagen hiperespectral, y el índice de los datos filtrados generados por el *kernel*. Por ello, cuando se ha realizado el envío de los últimos datos de cada tanda de

píxeles filtrados en paralelo, se invoca a la función "lastBandsFromPx" (Código 64) desarrollada en el archivo "FromFileToFPGA.cpp".

```
inline void lastBandsFromPx(hst_unsized_d* preproc_image,
hst_unsized_d* mins, hst_unsized_d* scales){
  static cl_int err;
  for(hst_band_ext_id k=0; k<_PREPROC_INS_WITH_NO_OUT; k++){
    err=q_out.finish();
    err = q_exe.enqueueTask(krnls[_FILTER_KRNL],
       (cl::vector<cl::Event>*)&filterInEvs, filterExeEvs.data());
    err = q_out.enqueueReadBuffer(filter_output, CL_FALSE, 0,
      BYTES_PER_TRANSFER, &(preproc_image[frame_preproc_idx]),
      (cl::vector<cl::Event>*)&filterExeEvs, &(filterOutEvs[0]));

    frame_preproc_idx+= _N_FILTERS;
  }

  err = q_out.enqueueReadBuffer(filter_min, CL_FALSE, 0,
        BYTES_PER_TRANSFER, &(mins[currentPx]),
        (cl::vector<cl::Event>*)&filterExeEvs, &(filterOutEvs[1]));
  err = q_out.enqueueReadBuffer(filter_scale, CL_FALSE, 0,
        BYTES_PER_TRANSFER, &(scales[currentPx]),
        (cl::vector<cl::Event>*)&filterExeEvs, &(filterOutEvs[2]));

  //Caso no más pixeles que evaluar de la imagen
  if(currentPx >= lastPxFrame){
    currentPx=0;
  //Caso quedan pixeles a evaluar de la imagen
  }else{
    currentPx+=_N_FILTERS;
  }
}
```

Código 64: Función "lastBandsFromPx"

Esta función se encarga de invocar las ejecuciones del *kernel* de filtrado, así como las posteriores extracciones de los datos filtrados que estas generan, requeridas para obtener los últimos datos filtrados de los píxeles procesados en paralelo y que se quedaron pendientes tras el envío al *kernel* de los últimos datos sin filtrar de dichos píxeles. En esta función, también se especifica la extracción de los valores mínimos y de escala producidos al terminar de generar todos los datos filtrados de la tanda de píxeles filtrados en paralelo.

En lo que respecta al *kernel* de normalizado, este se denominará "normalizers" en la descripción HLS. Sobre el funcionamiento de este *kernel*, cabe destacar que el inicio de las ejecuciones para el normalizado de los datos de una tanda de píxeles procesados en paralelo viene precedido de 2 ejecuciones. Dichas ejecuciones se requieren para introducir el mínimo y la escala a utilizar en el normalizado de los píxeles de la tanda a normalizar.

Este modo de funcionamiento posibilita que los valores mínimos y de escala, así como los datos filtrados de los píxeles puedan enviarse a través de un mismo puerto físico.

El diseño de este *kernel* está constituido, fundamentalmente, por módulos de normalizado que se ajustan a la descripción HLS definida en la función "normalizer" (Código 65) dentro del archivo "normalizers.cpp". Este *kernel* de normalizado solo precisa de 3 puertos físicos para operar con el mínimo intervalo de *pipelining*, el cual será de 1 siempre que el ancho de la transferencia de datos entre el *host* y la FPGA así lo posibiliten.

```
void normalizer(norm_in_FIFO& val_str, unsized_d offset, unsized_d
scale, norm_out_FIFO& out_str){
#pragma HLS FUNCTION_INSTANTIATE variable=val_str
#pragma HLS FUNCTION_INSTANTIATE variable=out_str
#pragma HLS INLINE
  const unsized_d dat = val_str.read();
  const tosized_d num = dat - offset;
  const sized_d res = num/scale;
  out_str.write(res);
}
```

Código 65: Función "normalizer"

La introducción de los datos filtrados a normalizar una vez que estos alcanzan los puertos de entrada del *kernel* de normalizado requiere de 2 operaciones de manera previa a su utilización en la FPGA. Estas operaciones se ordenan en la descripción HLS por medio de la invocación de la función "normToStr" (Código 66) para un mismo conjunto de datos a normalizar en paralelo.

```
void normToStr(trans_unsized_d data[_NORMALIZER_IN_TRANSFER],
norm_in_FIFO data_str[_NORMALIZER_PARALLEL]){
#pragma HLS FUNCTION_INSTANTIATE variable=data
#pragma HLS FUNCTION_INSTANTIATE variable=data_str
#pragma HLS INLINE
#pragma HLS array_partition variable=data cyclic \
factor=C_NORM_DATA_PER_HW_IN_TRANSFER
#pragma HLS array_partition variable=data_str complete

  static unsized_d loc_data[_NORMALIZER_IN_TRANSFER];
  #pragma HLS array_partition variable=loc_data complete

  //Conversion "trans_unsized_d" -> "unsized_d"
  LOOP_NORM_TO_STR_I: for(transfer_in_id idx=0;
  idx<NORMALIZER_IN_TRANSFER; idx++){
  #pragma HLS UNROLL factor=C_NORM_DATA_PER_HW_IN_TRANSFER \
  skip_exit_check
  #pragma HLS DEPENDENCE class=array type=inter dependent=false
    normConvertInputUnsized(data[idx], &(loc_data[idx]));
  }
```

```
  //Escritura no bloqueante de datos de entrada en FIFOs
  LOOP_NORM_TO_STR_II: for(norm_parallel_id k=0;
                           k<NORMALIZER_PARALLEL; k++){
  #pragma HLS UNROLL
    normWriteInStr(data_str[k], loc_data[k]);
  }
}
```

Código 66: Función "normToStr"

La primera operación consiste en transformar los datos filtrados transferidos por el *host* de su formato de transferencia, correspondiente al tipo de variable "trans_unsized_d", al formato con el que se procesarán en el *kernel* de normalizado, indicado este último por el tipo de variable "unsized_d". Esta operación se detalla para el caso de un único dato en la función "normConvertInputUnsized". La siguiente operación consiste en introducir los datos transformados en las FIFOs que correspondan. Esta operación se describe en el caso de un solo dato con la función "normWriteInStr". Para cada conjunto de datos a normalizar en paralelo, la asignación de FIFO a cada dato es unívoca.

Para la introducción de los valores mínimos y de escala de los píxeles a normalizar en paralelo en el momento en el que se requieran, se recurre a la función "loadToReg" (Código 67). En esta función, la operación a ejecutar consiste en transformar los valores transferidos desde el *host* del formato "trans_unsized_d" al formato "unsized_d". Para ello se invoca, al igual que en la función "normToStr", a la función "normConvertInputUnsized".

```
void loadToreg(trans_unsized_d fromHost[_NORMALIZER_IN_TRANSFER],
               unsized_d regs[_NORMALIZER_PARALLEL]){
#pragma HLS FUNCTION_INSTANTIATE variable=fromHost
#pragma HLS FUNCTION_INSTANTIATE variable=regs
#pragma HLS array_partition variable=fromHost cyclic \
factor=C_NORM_DATA_PER_HW_IN_TRANSFER
#pragma HLS array_partition variable=regs complete
#pragma HLS INLINE

  static unsized_d loc_fromHost[_NORMALIZER_IN_TRANSFER];
  #pragma HLS array_partition variable=loc_fromHost cyclic \
  factor=C_NORM_DATA_PER_HW_IN_TRANSFER

  // Conversion "trans_unsized_d" -> "unsized_d"
  LOOP_LOAD_TO_REG_I: for(transfer_in_id idx=0;
                          idx<NORMALIZER_IN_TRANSFER; idx++){
  #pragma HLS UNROLL factor=C_NORM_DATA_PER_HW_IN_TRANSFER \
  skip_exit_check
  #pragma HLS DEPENDENCE class=array type=inter dependent=false
    normConvertInputUnsized(fromHost[idx], &(loc_fromHost[idx]));
```

```
  }
  //Transferencia registro -> registro
  LOOP_LOAD_TO_REG_II: for(norm_parallel_id idx=0;
idx<NORMALIZER_PARALLEL; idx++){
  #pragma HLS UNROLL
    normRegToRegTransfer(loc_fromHost[idx], &(regs[idx]));
  }
}
```

Código 67: Función "loadToReg"

Con respecto a la interfaz de salida, los datos normalizados por el *kernel* requieren de dos operaciones para preparar estos datos para su transferencia hacia el *host*. Este procedimiento se establece en la descripción HLS como la función "normFromStr" (Código 68).

```
void normFromStr(norm_out_FIFO output_str[_NORMALIZER_PARALLEL],
trans_sized_d output[_NORMALIZER_OUT_TRANSFER]){
#pragma HLS array_partition variable=output_str complete
#pragma HLS array_partition variable=output cyclic \
factor=C_NORM_DATA_PER_HW_OUT_TRANSFER
#pragma HLS INLINE
  static sized_d loc_output[_NORMALIZER_OUT_TRANSFER];
  #pragma HLS array_partition variable=loc_output complete

  //Lectura no bloqueante de FIFOs de salida
  LOOP_NORM_FROM_STR_I: for(norm_parallel_id idx=0;
                            idx<NORMALIZER_PARALLEL; idx++){
  #pragma HLS UNROLL
    normNonBlockingReadFromStr(output_str[idx], loc_output[idx]);
  }

  //Conversion "sized_d" -> "trans_sized_d"
  LOOP_NORM_FROM_STR_II: for(transfer_out_id idx=0;
                             idx<NORMALIZER_OUT_TRANSFER; idx++){
  #pragma HLS UNROLL factor=C_NORM_DATA_PER_HW_OUT_TRANSFER \
  skip_exit_check
  #pragma HLS DEPENDENCE class=array type=inter dependent=false
    normConvertOutputSized(loc_output[idx], &(output[idx]));
  }
}
```

Código 68: Función "normFromStr"

La primera operación consiste en la lectura no bloqueante de los datos normalizados en paralelo a transferir de las FIFOs que correspondan, existiendo una asignación de FIFO unívoca para cada dato. Esta operación se especifica en la descripción HLS como la función "normNonBlockingReadFromStr". La siguiente operación consiste en transformar los datos normalizados del formato en el que fueron generados, correspondiente al tipo de variable "sized_d", al formato con el que se enviarán hacia el

host, correspondiente al formato "trans_sized_d". Esta operación se corresponde a la función "normConvertOutputSized".

La función principal donde se desarrolla el código que gestiona las transferencias entre el *host* y el *kernel* de normalizado se denomina "ToNormalizers" (Código 69), el cual se describe en el archivo "ToNormalizers.cpp". Esta función invoca a su vez a la función "sendMinAndScale" (Código 70), antes del envío de cada nueva tanda de píxeles a normalizar en paralelo, para el envío e introducción al *kernel* de normalizado de los valores mínimos y de escala a utilizar en el normalizado de cada píxel de dicha tanda.

```
void ToNormalizers(hst_unsized_d* filtered_image, hst_unsized_d* mins,
     hst_unsized_d* scales, hst_band_id nBands, hst_im_dat_id nImData){
  const hst_im_dat_id PX_PARALL_FRAME_INTERVAL =
                              _NORMALIZER_PARALLEL*nBands;
  hst_im_dat_id data_idx=0;
  //Recorrido por la totalidad de la imagen hiperespectral filtrada
  do{
    sendMinsAndScales(mins, scales);
    const hst_im_dat_id PX_PARALL_FRAME_LIM =
                          data_idx+PX_PARALL_FRAME_INTERVAL;
    do{
      exeNorm(filtered_image, &(data_idx));
    }while(data_idx<PX_PARALL_FRAME_LIM);
  }while(data_idx<nImData);
  //Reset de px_idx para siguiente imagen
  px_idx=0;
  static cl_int err;
  err=q_out.finish();
}
```

Código 69: Función "ToNormalizers"

```
inline void sendMinsAndScales(hst_unsized_d* mins, hst_unsized_d*
scales){
  static cl_int err;
  err=q_exe.finish();

  err=q_in.enqueueWriteBuffer(norm_offset, CL_FALSE, 0,
                   NORM_BYTES_PER_IN_TRANSFER, &(mins[px_idx]),
                   nullptr, &(normInEvs[1]));
  err=q_in.enqueueWriteBuffer(norm_scale, CL_FALSE, 0,
                   NORM_BYTES_PER_IN_TRANSFER, &(scales[px_idx]),
                   nullptr, &(normInEvs[2]));

  err = q_exe.enqueueTask(krnls[_NORM_KRNL], &normInEvs,
        &(normExeEvs[0]));
  err = q_exe.enqueueTask(krnls[_NORM_KRNL], &normInEvs,
        &(normExeEvs[1]));

  px_idx+=_NORMALIZER_PARALLEL;
}
```

```
  hst_im_dat_id data_idx=0;
  //Recorrido por la totalidad de la imagen hiperespectral filtrada
  do{
    sendMinsAndScales(mins, scales);
    const hst_im_dat_id PX_PARALL_FRAME_LIM =
                        data_idx+PX_PARALL_FRAME_INTERVAL;
    do{
      exeNorm(filtered_image, &(data_idx));
    }while(data_idx<PX_PARALL_FRAME_LIM);
  }while(data_idx<nImData);
  //Reset de px_idx para siguiente imagen
  px_idx=0;
  static cl_int err;
  err=q_out.finish();
}
```

Código 70: Función "sendMinAndScales"

Además, la función "ToNormalizers" invoca a la función "exeNorm". Esta última función se emplea para gestionar y sincronizar la introducción de nuevos datos filtrados de la imagen en el *kernel* de normalizado, la ejecución de la normalización de dichos datos, y la posterior extracción de los datos normalizados. Tanto la función "exeNorm" (Código 71) como la función "sendMinAndScale" se encuentran declaradas dentro del archivo "ToNormalizers.cpp."

```
inline void exeNorm(hst_unsized_d* filtered_image, hst_im_dat_id*
ptr_idx){
  static cl_int err;
  const hst_im_dat_id IDX = *ptr_idx;

  err=q_exe.finish();
  err=q_in.enqueueWriteBuffer(norm_input, CL_FALSE, 0,
      NORM_BYTES_PER_IN_TRANSFER, &(filtered_image[IDX]),
      nullptr, &(normInEvs[0]));
  err=normInEvs[0].setCallback(CL_COMPLETE, &normInputSent, nullptr);

  err=q_out.finish();
  err = q_exe.enqueueTask(krnls[_NORM_KRNL], &normInEvs,
        &(normExeEvs[2]));

  err=q_out.enqueueReadBuffer(norm_output, CL_FALSE, 0,
      NORM_BYTES_PER_OUT_TRANSFER, &(normed[IDX]), &normExeEvs,
      &normOutEv);

  *ptr_idx = IDX + _NORMALIZER_PARALLEL;
}
```

Código 71: Función "exeNorm"

8.7. Etapa k-means

En esta etapa, se aplica el algoritmo “k-means” sobre los píxeles de la imagen hiperespectral. El objetivo de este proceso consiste en subdividir los píxeles de la imagen en varias categorías o *clústeres* en función de la similitud relativa existente entre las firmas espectrales de estos píxeles. En la práctica, las similitudes observadas desde la perspectiva computacional se deben a que los píxeles similares representan un tipo de piel y un estado de afección del mismo parecido. Por este motivo, la aplicación de esta etapa permitirá discernir las regiones captadas en la imagen correspondientes a tejido cutáneo sano de las correspondientes a tejido afectado. Sin embargo, tras la ejecución de esta etapa no se conocerá aún a qué estado de la piel se asocian cada una de las regiones determinadas. La ejecución del algoritmo “k-means” será efectuada por el *kernel* FPGA denominado “top_kmeans” (Código 72).

```
extern "C"{
void top_kmeans(trans_cluster_id pxCtr[1],
  trans_sized_d px[_KMEANS_IN_TRANSFER], trans_px_id nPx[1],
  trans_band_id nBands[1],  trans_cluster_id newPxCtr[1],
  bool end[1], trans_sized_d ctrs[_KMEANS_OUT_TRANSFER]){
#pragma HLS INTERFACE m_axi port=px bundle=gmem4\
max_read_burst_length=C_KMEANS_IN_BURST\
max_write_burst_length=C_KMEANS_OUT_BURST
#pragma HLS INTERFACE m_axi port=ctrs bundle=gmem4\
max_read_burst_length=C_KMEANS_IN_BURST\
max_write_burst_length=C_KMEANS_OUT_BURST
#pragma HLS INTERFACE m_axi port=pxCtr bundle=gmem_info
#pragma HLS INTERFACE m_axi port=nPx bundle=gmem_info
#pragma HLS INTERFACE m_axi port=nBands bundle=gmem_info
#pragma HLS INTERFACE m_axi port=newPxCtr bundle=gmem_info
#pragma HLS INTERFACE m_axi port=end bundle=gmem_info
#pragma HLS array_partition variable=px cyclic\
factor=C_KMEANS_DATA_PER_HW_IN_TRANSFER
#pragma HLS array_partition variable=ctrs cyclic\
factor=C_KMEANS_DATA_PER_HW_OUT_TRANSFER
#pragma HLS PIPELINE II=40

  //Centroide del pixel en evaluacion
  static cluster_id loc_pxCtr;

  //Cantidad de pixeles en la imagen
  static px_id loc_nPx = (px_id)_DEFAULT_PX;

  //Notifica la nueva clasificacion del pixel evaluado
  static cluster_id loc_newPxCtr;

  //Carga nuevo centroide del pixel a evaluar
  loc_pxCtr = (cluster_id)(pxCtr[0]);
```

```
    //Ocurre si hay nueva imagen/iteracion de kmeans
    loc_nPx = (px_id)(nPx[0]);
    loc_nBands = (band_id)(nBands[0]);
    changeLim = loc_nPx*_REL_MIN_PX_CHANGES;
    kmeans(loc_pxCtr, px, loc_nPx, loc_nBands, &loc_newPxCtr, ctrs,
    end);
    newPxCtr[0] = (trans_cluster_id)loc_newPxCtr;
}
}
```

Código 72: Kernel "top_kmeans"

Para ejecutar "k-means", debe invocarse la ejecución de este *kernel* reiteradamente. En cada iteración, se transmite al *kernel* o desde el *kernel* una tanda de tamaño fijo de bandas espectrales contiguas. En la descripción HLS, la tanda de bandas espectrales de entrada al *kernel* se identifica como el parámetro "px" de la función "top_kmeans", mientras que la tanda saliente se asocia al parámetro "ctrs". Debido a la condición con la que el *kernel* "top_kmeans" introduce y genera estos datos de bandas espectrales, debe cumplirse que los datos que convivan en una misma tanda de envío o recepción pertenezcan al mismo píxel o centroide. Por ello, si para una determinada iteración del *kernel* no se disponen de suficientes bandas espectrales sin enviar del píxel o del centroide en transferencia para llenar la tanda, los datos sobrantes deben especificarse con valor 0. Además, se deben enviar y recibir, para cada píxel y centroide, secciones consecutivas de su firma espectral entre una ejecución del *kernel* y la siguiente.

La cantidad de datos transferidos en cada tanda de envío/recepción entre el *host* y el *kernel* "top_kmeans" es ajustable mediante la modificación del valor de la macro "_KMEANS_PARALLEL" declarada en el archivo "amounts.h". Esta macro especifica el grado de paralelización del procesamiento a implementar en este *kernel*. Cabe destacar que para que la configuración estática de este *kernel* sea funcional, la modificación del valor asignado a "_KMEANS_PARALLEL" debe acompañarse de la modificación del valor de la macro "_ADDER_REG", la cual se encuentra también declarada en el archivo "amounts.h". La relación que debe existir entre los valores de las macros "_KMEANS_PARALLEL" y "_ADDER_REG" se detalla en el apartado 8.10. .

El *kernel* "top_kmeans" dispone de 4 "modos" de funcionamiento. Cada uno de estos modos implica utilizar las bandas espectrales de entrada o bien producir bandas espectrales de salida mediante un conjunto de operaciones específico. Durante la

ejecución del algoritmo “k-means”, el *kernel* cambiará de un modo de funcionamiento al siguiente entre algunas iteraciones y la iteración siguiente.

Desde la perspectiva de la descripción HLS, el modo de funcionamiento a aplicar en cada iteración viene establecido por un conjunto de variables booleanas estáticas. La determinación de la iteración en la que se cambia de un modo de funcionamiento al siguiente no viene establecida directamente por el *host*, sino que es regulada por una serie de variables estáticas utilizadas a modo de contadores. Sin embargo, el reinicio de estos contadores, y con ello la transición en el modo de funcionamiento, viene establecida por la cantidad de bandas espectrales y de píxeles de la imagen en evaluación. Estas cantidades son especificadas dinámicamente como parámetros de entrada del *kernel* “top_kmeans”, identificándose en la descripción HLS como los parámetros “nPx” para el caso de los píxeles y como “nBands” para el caso de las bandas espectrales.

La influencia de estos parámetros sobre los contadores se debe a que los primeros establecen el límite del recorrido de la gran mayoría de los contadores. Debido a este hecho, la especificación de “nPx” y de “nBands” por parte del *host* permite que el *kernel* ajuste autónomamente el reset de los contadores y las transiciones entre los modos de funcionamiento. En la Figura 44 se muestran las condiciones de transición entre los modos de funcionamiento.

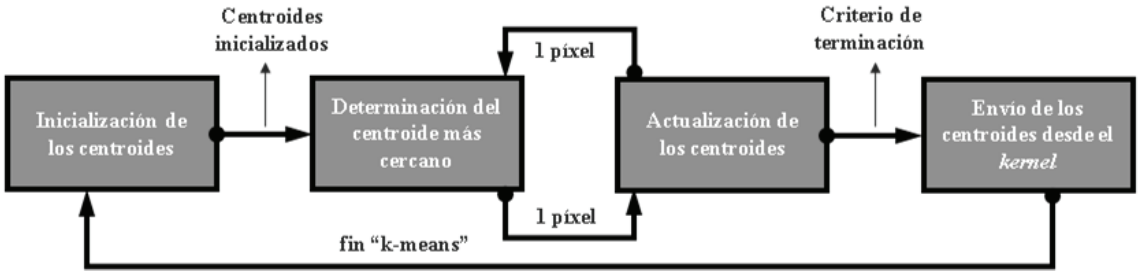


Figura 44: Modos de funcionamiento del kernel “top_kmeans” y transiciones entre ellos

Un aspecto común de los modos de funcionamiento de este *kernel* consiste en que todos ellos leen y/o escriben las firmas espectrales de los centroides calculados en cada momento y guardados internamente en la FPGA. Desde la perspectiva de la descripción HLS, la región de almacenamiento de los datos de estos centroides dentro de la FPGA se asocia al *array* “loc_centroids”. Además, todos los modos de funcionamiento poseen una indexación de acceso a este *array* variable que se modifica de una iteración del modo de

funcionamiento a la siguiente. Esta indexación variable viene establecida por las variables también utilizadas como contadores indicados anteriormente.

Las características específicas de cada modo de funcionamiento se explican a continuación.

8.7.1. Inicialización de los centroides

Este modo de funcionamiento se establece al principio de la ejecución del *kernel* "top_kmeans" con cada nueva imagen hiperespectral. Su procedimiento se especifica en la descripción HLS dentro de la función "initCentroids" (Código 73) del archivo "load.cpp". En cada iteración de este modo de ejecución, el *kernel* utiliza los datos de bandas espectrales recibidos del *host* para inicializar una sección determinada de los registros de los centroides. La ejecución de este modo de funcionamiento en este *kernel* se reitera hasta que se han terminado de inicializar todos los centroides. Cuando esto último suceda, se cambia el modo de ejecución al de determinación del centroide más cercano al píxel en evaluación.

```
void initCentroids(kmeans_FIFO pxStr[_CTR_INIT_PARALLEL],
                   band_id nBands){
#pragma HLS array_partition variable=loc_centroids complete dim=1
#pragma HLS array_partition variable=loc_centroids cyclic \
factor=LOC_CTRS_PARALLEL_ACCESS dim=2
#pragma HLS array_partition variable=raw_loc_centroids complete dim=1
#pragma HLS array_partition variable=raw_loc_centroids cyclic \
factor=RAW_LOC_CTRS_PARALLEL_ACCESS dim=2
#pragma HLS array_partition variable=px_amounts complete //dim=0
#pragma HLS array_partition variable=pxStr complete
#pragma HLS INLINE

  const band_id BAND_IDX_LIM =(nBands>CTR_INIT_PARALLEL)?
            ((band_id)(nBands-CTR_INIT_PARALLEL)):((band_id)0);

  static band_id band_idx = 0;
  static cluster_id ctr = 0;

  static sized_d data[_CTR_INIT_PARALLEL];
  #pragma HLS array_partition variable=data complete

  LOOP_INIT_CENTROIDS_I: for(ctr_init_parallel_id j=0;
                        j<CTR_INIT_PARALLEL; j++){
  #pragma HLS UNROLL
    kmeansFromStrToDat(pxStr[j], &(data[j]));
  }

  //Almacenado de datos del centroide calculados en esta iteracion
  sized_d* const LOC_CTRS_SECT = &(loc_centroids[ctr][band_idx]);
```

```
  px_sum_d* const RAW_LOC_CTRS_SECT =
     &(raw_loc_centroids[ctr][band_idx]);
  LOOP_INIT_CENTROIDS_II: for(ctr_init_parallel_id j=0;
                              j<CTR_INIT_PARALLEL; j++){
  #pragma HLS DEPENDENCE variable=loc_centroids type=inter false
  #pragma HLS UNROLL
    setCtrDat(data[j], &(LOC_CTRS_SECT[j]), &(RAW_LOC_CTRS_SECT[j]));
  }
  px_amounts[ctr] = 0;

  //Actualizacion del contador
  if(band_idx < BAND_IDX_LIM){
    band_idx+=CTR_INIT_PARALLEL;
  }else{
    band_idx=0;
    if(ctr < LAST_CTR){
      ctr++;
    }else{
      ctr=0;
      initCtr=false;
    }
  }
}
```

Código 73: Función "initCentroids"

8.7.2. Determinación del centroide más cercano

Este modo de funcionamiento se activa cada vez que comienza la aplicación del algoritmo "k-means" para un nuevo píxel. La descripción HLS del mismo se detalla en el archivo "assessDistance.cpp", siendo la función homónima la principal en este modo de funcionamiento (Código 74).

```
void assessDistance(sized_d data[_KMEANS_PARALLEL], band_id nBands,
                    cluster_id* new_ctr_idx){
#pragma HLS array_partition variable=loc_centroids complete dim=1
#pragma HLS array_partition variable=loc_centroids cyclic \
factor=LOC_CTRS_PARALLEL_ACCESS dim=2
#pragma HLS FUNCTION_INSTANTIATE variable=data
#pragma HLS FUNCTION_INSTANTIATE variable=new_ctr_idx
#pragma HLS array_partition variable=data complete
#pragma HLS INLINE
  const band_id BAND_IDX_LIM = (nBands>KMEANS_PARALLEL)?
                ((band_id)(nBands - KMEANS_PARALLEL)):((band_id)0);

  static kmeans_adder_out_d dist[_N_CLUSTERS]=
                 {(kmeans_adder_out_d)0};
  #pragma HLS array_partition variable=dist complete

  static mult_sized_d in[_KMEANS_PARALLEL];
  #pragma HLS array_partition variable=in complete

  //Indica que comienza la evaluacion de un nuevo pixel, reseteando
```

```
  //los elementos con memoria
  static bool newPx = true;

  LOOP_ASSESS_DIST_I: for(cluster_id cluster_idx = 0;
                      cluster_idx<N_CLUSTERS; cluster_idx++){
  #pragma HLS PIPELINE II=1
  #pragma HLS UNROLL factor=1

    //Inicializacion datos de entrada
    sized_d* const LOC_CTR = loc_centroids[cluster_idx];
    LOOP_ASSESS_DIST_I_I: for(kmeans_parallel_id idx=0;
                          idx<KMEANS_PARALLEL; idx++){
    #pragma HLS UNROLL
      const bnd_over_parall_id BAND_IDX = bnd_idx + idx;
      const sized_d DAT = LOC_CTR[BAND_IDX];
      in[idx] = coord_diff(data[idx], DAT);
    }
    kmeans_adder.run(in, newPx, cluster_idx, &(dist[cluster_idx]));
  }
  min.minRun(dist, new_ctr_idx);

  //Actualizacion contadores
  if(bnd_idx < BAND_IDX_LIM){
    bnd_idx+=KMEANS_PARALLEL;
    newPx = false;
  }else{
    newPx = true;
    uptCtr = true;
    bnd_idx = 0;
  }
}
```

Código 74: Función "assessDistance"

Para cumplir con su cometido, el procedimiento funcional de este modo de ejecución consiste, en primer lugar, en tomar la firma espectral del píxel y calcular la divergencia en términos absolutos entre esta y la firma espectral de cada centroide en el momento de la ejecución de esta iteración. Esta divergencia se calcula para cada centroide como el sumatorio cuadrático de la diferencia de valor en cada banda espectral entre el píxel y el centroide a comparar. Asimismo, el proceso de cálculo de la diferencia al cuadrado para una banda espectral entre un píxel y un centroide se describe en la función "coord_diff" (Código 75).

```
mult_sized_d coord_diff(sized_d px_coord, sized_d ctr_coord){
#pragma HLS INLINE
      const neg_sized_d DIFF = px_coord - ctr_coord;
      return DIFF*DIFF;
}
```

Código 75: Función "coord._diff"

Entrando más en detalle, los cálculos de estas divergencias se efectúan progresivamente a medida que llegan al *kernel* las diferentes secciones de la firma espectral del píxel a evaluar. Ello implica que, en cada iteración de este modo de funcionamiento, se calculan los sumatorios cuadráticos de las diferencias de valor únicamente para el conjunto de bandas espectrales del píxel que se introduce en el *kernel* en dicha iteración. Estos últimos sumatorios se suman acumulativamente al valor almacenado en un registro diferente para cada centroide.

De esta manera, se logra mediante sumas recurrentes efectuadas en sucesivas iteraciones el cálculo de la divergencia entre el píxel evaluado y cada centroide. Desde la perspectiva de la descripción HLS, los registros utilizados para almacenar los resultados de estas sumas iterativas se especifican como el *array* "dist". El módulo *hardware* responsable de realizar las sumas se encuentra descrito en HLS en la función "run" perteneciente a la clase "Kmeans_Adder" (Código 76) y ubicada en el archivo "kmeans_adder.cpp". Dicho módulo también incluye la funcionalidad de reiniciar los valores de "dist" cuando se comienza el cálculo de las divergencias para un nuevo píxel.

```
void run(in_t in[_KMEANS_PARALLEL], bool rst, cluster_id out_idx,
         out_t* out){
#pragma HLS FUNCTION_INSTANTIATE variable=in
#pragma HLS PIPELINE II=1
#pragma HLS array_partition variable=in        complete
#pragma HLS array_partition variable=inner_reg complete

  const static out_t INIT_VAL_REG = 0;

  static out_t loc_out[_N_CLUSTERS];
  #pragma HLS array_partition variable=loc_out complete

  static out_t loc_out_reg[_N_CLUSTERS] = {0};
  #pragma HLS array_partition variable=loc_out_reg complete

  static out_t feedBack[_N_CLUSTERS];
  #pragma HLS array_partition variable=feedBack complete

  LOOP_KMEANS_ADDER_RUN_I: for(half_parall_in_id inner_idx=0;
                  inner_idx < _KMEANS_PARALLEL/2; inner_idx++){
  #pragma HLS UNROLL
    operation<in_t, in_t, inner_t>(in[2*inner_idx],
              in[2*inner_idx+1], &(inner_reg[inner_idx]));
  }
  #if (_KMEANS_PARALLEL % 2) != 0
    inner_reg[_KMEANS_PARALLEL/2] = (inner_t)in[_KMEANS_PARALLEL - 1];
  #endif
```

```
  half_parall_in_id stage;
  reg_length_id com_offset;
  //Bucle por etapa del arbol de bifurcacion convergente
  LOOP_KMEANS_ADDER_RUN_II: for(com_offset = 0,
       stage=(half_parall_in_id)_ROUND_DIV_UP(_KMEANS_PARALLEL, 2);
       stage > 1; stage=(half_parall_in_id)_ROUND_DIV_UP(stage, 2)){
  #pragma HLS UNROLL
    //Bucle por módulo de la etapa
    LOOP_KMEANS_ADDER_RUN_II_I: for(half_parall_in_id i=0;
    i<(stage/2); i++){
    #pragma HLS UNROLL
      operation<inner_t, inner_t, inner_t>(inner_reg[com_offset+2*i],
      inner_reg[com_offset + 2*i + 1],
      &(inner_reg[com_offset+stage+i]));
    }

    //Caso sobra un dato de esta etapa --> paso directo a la siguiente
    const bool ODD_STAGE = ((stage & 1)!=0);
    if(ODD_STAGE){
      inner_reg[com_offset+stage+stage/2]=
              inner_reg[com_offset+stage-1];
    }
    com_offset += stage;
  }
  //Modulo con memoria
  feedBack[out_idx] = rst?INIT_VAL_REG:loc_out_reg[out_idx];
  operation<inner_t, out_t, out_t>(inner_reg[REG_LENGTH - 1],
                       feedBack[out_idx], &(loc_out[out_idx]));
  loc_out_reg[out_idx] = loc_out[out_idx];
  *out = loc_out[out_idx];
}
```

Código 76: Función "Kmeans_Adder::run"

Tras el cálculo de estas divergencias, debe determinarse el centroide más semejante al píxel evaluado. Para ello, se introducen los datos calculados y almacenados en "dist" en el módulo *hardware* especificado mediante descripción HLS en la función "run" de la clase "Min_ctr" desarrollada en el archivo "min_ctr.cpp". Esta función se muestra en el Código 77. Este módulo obtiene como resultado el centroide más próximo al píxel evaluado, que se envía al *host* cumpliendo así con el propósito de este modo de funcionamiento. En la descripción HLS del *kernel* "top_kmeans", el nuevo centroide obtenido y a enviar desde el *kernel* hacia *host* se identifica en la cabecera de la función homónima como "newPxCtr". Al terminar de evaluar el píxel, se cambia el modo de funcionamiento del *kernel* al de actualización del centroide anteriormente asignado al mismo píxel.

```
void run(in_t in[_KMEANS_PARALLEL], bool rst, cluster_id out_idx,
         out_t* out){
#pragma HLS FUNCTION_INSTANTIATE variable=in
#pragma HLS PIPELINE II=1
#pragma HLS array_partition variable=in        complete
#pragma HLS array_partition variable=inner_reg complete

  const static out_t INIT_VAL_REG = 0;
  static out_t loc_out[_N_CLUSTERS];
  #pragma HLS array_partition variable=loc_out complete
  static out_t loc_out_reg[_N_CLUSTERS] = {0};
  #pragma HLS array_partition variable=loc_out_reg complete
  static out_t feedBack[_N_CLUSTERS];
  #pragma HLS array_partition variable=feedBack complete

  LOOP_KMEANS_ADDER_RUN_I: for(half_parall_in_id inner_idx=0;
    inner_idx < _KMEANS_PARALLEL/2; inner_idx++){
  #pragma HLS UNROLL
    operation<in_t, in_t, inner_t>(in[2*inner_idx], in[2*inner_idx+1],
                            &(inner_reg[inner_idx]));
  }
  //Si N_FILTERS es impar, el último dato de entrada pasa directo a la
  //siguiente etapa comparativa
  #if (_KMEANS_PARALLEL % 2) != 0
    inner_reg[_KMEANS_PARALLEL/2] = (inner_t)in[_KMEANS_PARALLEL - 1];
  #endif
  half_parall_in_id stage;
  reg_length_id com_offset;
  //Bucle por etapa del arbol de bifurcacion convergente
  LOOP_KMEANS_ADDER_RUN_II: for(com_offset=0,
    stage=(half_parall_in_id)_ROUND_DIV_UP(_KMEANS_PARALLEL, 2);
    stage > 1; stage=(half_parall_in_id)_ROUND_DIV_UP(stage, 2)){
  #pragma HLS UNROLL
    //Bucle por módulo de la etapa
    LOOP_KMEANS_ADDER_RUN_II_I: for(half_parall_in_id i = 0;
    i<(stage/2); i++){
    #pragma HLS UNROLL
      operation<inner_t, inner_t, inner_t>(inner_reg[com_offset+2*i],
       inner_reg[com_offset+2*i+1], &(inner_reg[com_offset+stage+i]));
    }

    //Caso sobra un dato de esta etapa --> paso directo a la siguiente
    const bool ODD_STAGE = ((stage & 1)!=0);
    if(ODD_STAGE){
      inner_reg[com_offset+stage+stage/2] =
          inner_reg[com_offset+stage-1];
    }
    com_offset += stage;
  }
  //Modulo con memoria
  feedBack[out_idx] = rst?INIT_VAL_REG:loc_out_reg[out_idx];
  operation<inner_t, out_t, out_t>(inner_reg[REG_LENGTH-1],
                feedBack[out_idx], &(loc_out[out_idx]));
  loc_out_reg[out_idx] = loc_out[out_idx];
  *out = loc_out[out_idx];
}
```

Código 77: Función Min_ctr::run

8.7.3. Actualización de los centroides

En este modo de funcionamiento se actualizan las coordenadas de los centroides afectados por la reclasificación del píxel. Estos centroides se corresponden a aquel en el que se ubicaba anteriormente el píxel y a aquel al que se le ha asignado el píxel. Para esta operación, debe reenviarse al *kernel* desde el *host* las sucesivas secciones de la firma espectral del píxel, y ejecutarse una iteración de este modo de funcionamiento a continuación de cada envío de sección. Además, debe indicarse desde el *host* el centroide anteriormente asociado al píxel en evaluación, el cual se corresponde en la descripción HLS al parámetro "pxCtr". La descripción funcional de este modo de funcionamiento se detalla, fundamentalmente, en la función "uptCtrs" (Código 78) escrita en el archivo "updateCentroids.cpp".

```
void uptCtrs(kmeans_FIFO pxStr[_PARALLEL_UPDATE], band_id nBands,
        px_id nPx, cluster_id new_ctr, cluster_id old_ctr, bool* end){
#pragma HLS array_partition variable=loc_centroids complete dim=1
#pragma HLS array_partition variable=loc_centroids cyclic\
factor=LOC_CTRS_PARALLEL_ACCESS dim=2
#pragma HLS array_partition variable=raw_loc_centroids complete dim=1
#pragma HLS array_partition variable=raw_loc_centroids cyclic\
factor=RAW_LOC_CTRS_PARALLEL_ACCESS dim=2
#pragma HLS array_partition variable=px_amounts complete //dim=0
#pragma HLS FUNCTION_INSTANTIATE variable=pxStr
#pragma HLS FUNCTION_INSTANTIATE variable=end
#pragma HLS array_partition variable=pxStr complete
#pragma HLS INLINE
  const band_id BAND_IDX_LIM = (nBands >
PARALLEL_UPDATE)?((band_id)(nBands - PARALLEL_UPDATE)):((band_id)0);

  static band_id band_idx=0;
  static px_id amountsForOld;
  static px_id amountsForNew;

  static sized_d added[_PARALLEL_UPDATE];
  #pragma HLS array_partition variable=added complete

  LOOP_UPT_CTRS_I: for(parall_update_id k=0; k<PARALLEL_UPDATE; k++){
  #pragma HLS UNROLL
    kmeansFromStrToDat(pxStr[k], &(added[k]));
  }

  const px_id LAST_PX = nPx - 1;

  amountsForOld = px_amounts[old_ctr];
  amountsForNew = px_amounts[new_ctr];

  const bool DIFF_CTRS = old_ctr != new_ctr;
  px_sum_d* const OLD_RAW_CTR_SECT =\
         &(raw_loc_centroids[old_ctr][band_idx]);
  px_sum_d* const NEW_RAW_CTR_SECT =
```

```
        &(raw_loc_centroids[new_ctr][band_idx]);

  LOOP_UPT_CTRS_II: for(parall_update_id k=0; k<PARALLEL_UPDATE; k++){
  #pragma HLS UNROLL
    uptRawCtrCoord(&(OLD_RAW_CTR_SECT[k]), &(NEW_RAW_CTR_SECT[k]),
                   added[k]);
  }

  sized_d* const OLD_CTR_SECT = &(loc_centroids[old_ctr][band_idx]);
  sized_d* const NEW_CTR_SECT = &(loc_centroids[new_ctr][band_idx]);
  LOOP_UPT_CTRS_III: for(parall_update_id k=0; k<PARALLEL_UPDATE;
k++){
  #pragma HLS UNROLL
    uptCtrCoord(&(OLD_CTR_SECT[k]), OLD_RAW_CTR_SECT[k],
            amountsForOld, &(NEW_CTR_SECT[k]), NEW_RAW_CTR_SECT[k],
            amountsForNew);
  }

  //Actualizacion de los contadores
  const bool BAND_BELOW = band_idx < BAND_IDX_LIM;
  const bool PX_OVER_UPT = (px_idx >= LAST_PX);
  //Caso no fin de actualizacion del centroide del pixel
  if(BAND_BELOW){
    band_idx+=PARALLEL_UPDATE;
  //Caso fin de actualizacion del centroide del pixel
  }else{
    band_idx=0;
    if(DIFF_CTRS){
      change++;
      px_amounts[new_ctr] = amountsForNew + 1;
      px_amounts[old_ctr] = amountsForOld - 1;
    }
    uptCtr=false;
    //Caso fin de iteracion de la imagen
    if(PX_OVER_UPT){
      px_idx = 0;
      if((change <= changeLim)||(iter >= MAX_ITER)){
        sendCtr = true;
        iter = 0;
        *end = true;
      //Notificacion de que la iteracion se ha completado con exito
      }else{
        iter++;
      }
      change=0;
    //Caso fin de iteracion de un pixel pero no de toda la imagen
    }else{
      px_idx++;
    }
  }
}
```

Código 78: Función "uptCtrs"

La operación de este *kernel* comienza con la extracción de datos procedentes de FIFOs. Para garantizar el procesamiento en paralelo exigido para este modo de funcionamiento, se implementan tantas FIFOs como grado de paralelización del

procesamiento se establezca. De esta manera, se garantiza el acceso simultáneo a los datos a procesar en paralelo en base a extraer un dato de cada FIFO por cada iteración. El código HLS correspondiente a esta operación se describe en la función "kmeansFromStrToDat" para el caso de una sola FIFO (Código 80).

```
void kmeansFromStrToDat(kmeans_FIFO& pxStr, sized_d* ptr_dat){
#pragma HLS FUNCTION_INSTANTIATE variable=pxStr
#pragma HLS INLINE
  *ptr_dat = pxStr.read();
}
```

Código 79: Función "kmeansFromStrToDat"

Posteriormente, se efectúa la actualización de los registros correspondientes al *array* "raw_loc_centroids". Estos registros se declaran en un *array* bidimensional cuyas dimensiones coinciden con las del *array* "loc_centroids". La función de "raw_loc_centroids" es la de almacenar, para cada banda espectral y para cada centroide, el sumatorio del valor en dicha banda espectral de todos los píxeles asignados al centroide. En el caso del centroide antiguamente asignado al píxel, la actualización consistirá en que, para cada banda espectral enviada de la sección del píxel transferida, se sustraiga el valor del píxel de la región del *array* "raw_loc_centroids" correspondiente a dicho centroide. Por otra parte, en el caso del centroide en el que se reubica el píxel, esta actualización consiste en sumar esos mismos valores del píxel a los datos de "raw_loc_centroids" correspondientes al nuevo centroide. Estas operaciones se describen en la función "uptRawCtrCoord", para el caso de la actualización de ambos centroides en una sola banda espectral. El código de esta función se muestra en el Código 80.

```
void uptRawCtrCoord(px_sum_d* ptr_rawOldCtrDat,
px_sum_d* ptr_rawNewCtrDat, sized_d px_coord){
#pragma HLS INLINE //OFF
//#pragma HLS PIPELINE //II=C_KMEANS_II

      (*ptr_rawOldCtrDat)-=px_coord;
      (*ptr_rawNewCtrDat)+=px_coord;
}
```

Código 80: Función "uptRawCtrCoord"

Tras actualizar los registros del *array* "raw_loc_centroids" correspondientes, se procede a actualizar los registros del *array* "loc_centroids". Asimismo, la lógica requerida para dicha actualización se describe en la función "uptCtrCoord" para el caso de una sola

coordenada para ambos centroides. El código correspondiente a esta función se expone en el Código 81.

```
void uptCtrCoord(sized_d* ptr_oldCtrDat,  px_sum_d rawOldCtrDat,
px_id oldCtrAmounts, sized_d* ptr_newCtrDat,  px_sum_d rawNewCtrDat,
px_id newCtrAmounts){
#pragma HLS FUNCTION_INSTANTIATE variable=ptr_oldCtrDat
#pragma HLS FUNCTION_INSTANTIATE variable=ptr_newCtrDat

#pragma HLS INLINE //OFF
//#pragma HLS PIPELINE //II=C_KMEANS_II

const px_id OLD_CTR_NEW_AMOUNTS = oldCtrAmounts - 1;
const px_id NEW_CTR_NEW_AMOUNTS = newCtrAmounts + 1;

*ptr_oldCtrDat = rawOldCtrDat/OLD_CTR_NEW_AMOUNTS;
*ptr_newCtrDat = rawNewCtrDat/NEW_CTR_NEW_AMOUNTS;
}
```

Código 81: Función "uptCtrCoord"

Con estas últimas operaciones se culmina la actualización de los centroides pertinentes en la sección de la firma espectral asociada a los datos del píxel procesados en dicha iteración. Para esta actualización, deberá determinarse la cantidad de píxeles asociada a cada centroide a actualizar. Dichas cantidades se almacenan en los registros asociados al *array* "px_amounts" de la descripción HLS. Posteriormente, deberá calcularse el nuevo valor a asignar para cada registro del centroide que debe modificarse. Dicho valor se obtiene como el cociente entre el valor del registro de "raw_loc_centroids" asociado y la cantidad de píxeles presentes tras la actualización del centroide.

Para simplificar la lógica de control interna del *kernel* "top_kmeans", el cambio en la clasificación de un píxel siempre conlleva efectuar las operaciones de actualización tanto del centroide en el que se reubica el píxel como del centroide en el que se encontraba el píxel con anterioridad.

Estas actualizaciones ocurrirán incluso en la primera categorización del píxel, cuando este no posee ningún centroide asignado. Por este motivo, todos los *arrays* que guardan datos relativos a los centroides ("raw_loc_centroids", "loc_centroids" y "px_amounts") se han sobredimensionado para almacenar un centroide adicional aparte de los exigidos y útiles para el *clustering*.

Este centroide adicional se actualiza únicamente en operaciones de "supuesta" desasignación del píxel del centroide. De esta manera, se evita tener que regular mediante lógica de control la situación de que el píxel no se encuentre clasificado sin que ello falsee los resultados del proceso de *clustering*.

Cabe destacar que, para esta aplicación, se indica que un píxel no posee ningún centroide asignado cuando el valor identificativo del centroide especificado coincide con la cantidad de centroides a obtener, es decir, con el valor de la macro "_N_CLUSTERS". Dicho valor no coincide con el que identifica a ningún centroide, puesto que todos ellos se identifican con valores entre 0 y la cantidad de centroides a determinar menos 1.

Dentro del código de la función "uptCtrs" se describe al final de este el procedimiento para evaluar si el algoritmo "k-means" implementado con este *kernel* FPGA converge en algún resultado correcto (Código 82). Esta evaluación se efectúa cada vez que el algoritmo "k-means" es aplicado con la totalidad de la imagen hiperespectral, lo cual se notifica por medio de una variable estática utilizada a modo de contador.

```
//Actualizacion de los contadores
const bool BAND_BELOW = band_idx < BAND_IDX_LIM;
const bool PX_OVER_UPT = (px_idx >= LAST_PX);
//Caso no fin de actualizacion del centroide del pixel
if(BAND_BELOW){
  band_idx+=PARALLEL_UPDATE;
//Caso fin de actualizacion del centroide del pixel
}else{
  band_idx=0;
  if(DIFF_CTRS){
    change++;
    px_amounts[new_ctr] = amountsForNew + 1;
    px_amounts[old_ctr] = amountsForOld - 1;
  }
  uptCtr=false;
  //Caso fin de iteracion de la imagen
  if(PX_OVER_UPT){
    px_idx = 0;
    if((change <= changeLim)||(iter >= MAX_ITER)){
      sendCtr = true;
      iter = 0;
      *end = true;
    //Notificacion de que la iteracion se ha completado con exito
    }else{
      iter++;
    }
```

```
      change=0;
    //Caso fin de iteracion de un pixel pero no de toda la imagen
    }else{
      px_idx++;
    }
```

Código 82: Evaluación del criterio de terminación de k-means

Desde el punto de vista aritmético, esta convergencia se observa cuando la cantidad de píxeles cuya clasificación ha cambiado tras la última evaluación de la imagen no supera un determinado umbral. Este umbral se establece estáticamente con el valor de la macro "_REL_MIN_PX_CHANGES", la cual se encuentra declarada en el archivo "amounts.h" (Código 83).

```
#define _REL_MIN_PX_CHANGES  1/10
```

Código 83: Declaración de "_REL_MIN_PX_CHANGES"

Dicho valor se especifica en términos unitarios y relativos, por lo que la cantidad de píxeles cambiados que se establezca como umbral máximo para declarar convergencia depende de la cantidad de píxeles de la imagen evaluada. Para evitar que una posible falta de convergencia con los niveles exigidos por "_REL_MIN_PX_CHANGES" se traduzca en un bucle infinito en las iteraciones del *kernel* "top_kmeans", se establece una condición de salida de dichas iteraciones cuando la cantidad de reevaluaciones de la imagen supera un determinado umbral. Esta cantidad máxima de iteraciones se configura estáticamente mediante la macro "_MAX_ITER" declarada en el archivo "amounts.h" (Código 84).

```
#define _MAX_ITER                                1000
```

Código 84: Declaración de "_MAX_ITER"

En la mayor parte de las ocasiones, cuando se termina con la actualización de los centroides para un píxel, se procede con la determinación del centroide más cercano para el píxel siguiente. La única excepción a este procedimiento ocurre si el píxel evaluado es el último de la imagen y además se ha logrado la convergencia en la clasificación de los píxeles. En este último caso, se establece el modo de funcionamiento para el envío de las firmas espectrales de los centroides obtenidos. Además, se notifica mediante *flag* al *host* que la convergencia en las clasificaciones de los píxeles ha sido lograda. Desde la

perspectiva de la descripción HLS, este *flag* se identifica con el parámetro "end" de la función "top" de este *kernel* (función "top_kmeans").

8.7.4. Envío de los centroides desde el *kernel*

Este modo de ejecución se establece cuando el algoritmo de "k-means" ejecutado por este *kernel* converge. Desde la perspectiva de la descripción HLS, este modo de funcionamiento se corresponde a la función "sendCentroids" escrita en el archivo "load.cpp" (Código 85).

```
void sendCentroids(kmeans_FIFO ctrsStr[_CTR_OUT_PARALLEL], band_id
nBands){
#pragma HLS array_partition variable=loc_centroids complete dim=1
#pragma HLS array_partition variable=loc_centroids cyclic\
factor=LOC_CTRS_PARALLEL_ACCESS dim=2
#pragma HLS array_partition variable=ctrsStr complete
#pragma HLS INLINE
  const band_id BAND_IDX_LIM = (nBands>CTR_OUT_PARALLEL)?
            ((band_id)(nBands - CTR_OUT_PARALLEL)):((band_id)0);

  static band_id band_idx = 0;
  static cluster_id ctr = 0;
  sized_d* const LOC_CTRS_SECT = &(loc_centroids[ctr][band_idx]);

  LOOP_SEND_CENTROIDS_II: for(ctr_out_parallel_id k=0;
                              k<CTR_OUT_PARALLEL; k++){
  #pragma HLS UNROLL
    kmeansFromDatToStr(LOC_CTRS_SECT[k], ctrsStr[k]);
  }

  //Actualizacion de los contadores
  if(band_idx < BAND_IDX_LIM){
    band_idx += CTR_OUT_PARALLEL;
  }else{
    band_idx = 0;
    if(ctr < LAST_CTR){
      ctr++;
    }else{
      ctr=0;
      sendCtr = false;
      initCtr = true;
    }
  }
}
```

Código 85: Función "sendCentroids"

El propósito de esta opción consiste en enviar las firmas espectrales de los centroides desde el *kernel* hacia el *host*. Este funcionamiento se ejecuta tantas veces como sea necesario para enviar las firmas espectrales de los centroides. Para ello, debe tenerse en cuenta la cantidad de bandas espectrales por iteración que se envían, la cual se fija

durante la compilación de la aplicación. Además, ninguna de las tandas de datos enviadas desde el *kernel* en este modo de funcionamiento debe contener bandas espectrales de centroides diferentes. De esta manera, si no hay datos suficientes de un determinado centroide para llenar la trama de datos a enviar, se rellenarán los datos sobrantes con valores a despreciar en el *host*. Cuando se termine de ejecutar las iteraciones correspondientes a este modo de funcionamiento, se reestablece el modo de inicialización de los centroides, preparando así al *kernel* "top_kmeans" para una posible nueva imagen hiperespectral.

8.7.5. Interfaces de entrada/salida

En lo que respecta a la interfaz de entrada, la introducción de las bandas espectrales a procesar en el *kernel* "top_kmeans" será gestionada, en primer lugar, por un componente *hardware* del *kernel* descrito mediante la función "kmeansToStr" (Código 86).

```
void kmeansToStr(trans_sized_d data[_KMEANS_IN_TRANSFER], kmeans_FIFO
data_str[_KMEANS_PARALLEL]){
#pragma HLS FUNCTION_INSTANTIATE variable=data
#pragma HLS FUNCTION_INSTANTIATE variable=data_str
#pragma HLS INLINE
#pragma HLS array_partition variable=data cyclic\
factor=C_KMEANS_DATA_PER_HW_IN_TRANSFER
#pragma HLS array_partition variable=data_str complete

  static sized_d loc_data[_KMEANS_IN_TRANSFER];
  #pragma HLS array_partition variable=loc_data complete

  //Conversion "trans_sized_d" -> "sized_d"
  LOOP_KMEANS_TO_STR_I: for(kmeans_in_transfer_id idx=0;
                            idx<KMEANS_IN_TRANSFER; idx++){
  #pragma HLS UNROLL factor=C_KMEANS_DATA_PER_HW_IN_TRANSFER \
  skip_exit_check
    kmeansConvertInputSized(data[idx], &(loc_data[idx]));
  }

  //Introduccion en FIFOs
  LOOP_KMEANS_TO_STR_II: for(kmeans_parallel_id k=0;
                             k<KMEANS_PARALLEL; k++){
  #pragma HLS UNROLL
    kmeansWriteInStr(data_str[k], loc_data[k]);
  }
}
```

Código 86: Función "kmeansToStr"

En esta función, la primera operación a efectuar, para cada uno de los datos de bandas espectrales a procesar en paralelo, consiste en cambiar el formato de estos.

Asimismo, los datos se convierten del formato en el que son recibidos por el *kernel*, definido como el tipo de variable "trans_sized_d", al formato con el que se utilizan estos datos dentro del *kernel*, referido como el formato "sized_d". Esta operación se detalla en la función "kmeansConvertInputSized". La siguiente operación consiste en introducir, mediante escritura bloqueante, cada uno de estos datos a la FIFO que se le asigne, existiendo una correspondencia unívoca. Dicha operación se describe en la función "kmeansWriteInStr".

Con respecto a la interfaz de salida, la transferencia hacia el *host* de las tramas de datos de los centroides calculados una vez generados requiere de dos operaciones intermedias que se describen en la función "kmeansFromStr" (Código 87).

```
void kmeansFromStr(kmeans_FIFO output_str[_CTR_OUT_PARALLEL],
trans_sized_d output[_KMEANS_OUT_TRANSFER]){
#pragma HLS array_partition variable=output_str complete
#pragma HLS array_partition variable=output cyclic
factor=C KMEANS DATA_PER_HW_OUT_TRANSFER
#pragma HLS INLINE

  static sized_d loc_output[_KMEANS_OUT_TRANSFER];
  #pragma HLS array_partition variable=loc_output complete

  //Lectura no bloqueante de FIFO de salida
  LOOP_KMEANS_FROM_STR_I: for(ctr_out_parallel_id idx=0;
                        idx<CTR_OUT_PARALLEL; idx++){
  #pragma HLS UNROLL
    kmeansNonBlockingReadFromStr(output_str[idx], loc_output[idx]);
  }

  //Conversion "sized_d" -> "trans_sized_d"
  LOOP_KMEANS_FROM_STR_II: for(kmeans_out_transfer_id idx=0;
                             idx<KMEANS_OUT_TRANSFER; idx++){
  #pragma HLS UNROLL factor=C_KMEANS_DATA_PER_HW_OUT_TRANSFER \
  skip_exit_check
  #pragma HLS DEPENDENCE class=array type=inter dependent=false
    kmeansConvertOutputSized(loc_output[idx], &(output[idx]));
  }
}
```

Código 87: Función "kmeansFromStr"

La primera operación consiste en extraer, mediante lectura no bloqueante, los datos del centroide a transferir de las FIFOs que los contienen. Esta operación se especifica para el caso de un solo dato en la función "kmeansNonBlockingReadFromStr". La siguiente operación consiste en transformar los datos extraídos del formato especificado como el

tipo de variable “sized_d”, al formato de transferencia indicado como el tipo de variable “trans_sized_d”. Esta operación se describe en la función “kmeansConvertOutputSized”.

Otras funciones descritas en el archivo “kmeans_interface.cpp” se denominan “kmeansFromStrToDat”, “kmeansFromStrToUselessData” y “kmeansFromDatToStr”. La primera de ellas especifica la lectura bloqueante de las FIFOs que contienen las bandas espectrales de entrada al procesamiento del *kernel* “top_kmeans”.

La siguiente función específica una extracción, mediante lectura bloqueante, de datos de bandas espectrales que carecen de utilidad en la ejecución con la única intención de vaciar las FIFOs. La necesidad de esta última función se debe a que el procedimiento de introducción de bandas espectrales de entrada especificado en la función “kmeansToStr” se efectúa de manera incondicional para permitir que el *kernel* “top_kmeans” pueda operar a frecuencias de reloj más elevadas. Dichos datos no son necesarios únicamente en el modo de funcionamiento de envío de los centroides hacia el *host*, siendo en consecuencia el único modo en el que se emplea la función “kmeansFromStrToUselessData”. La función “kmeansFromDatToStr” se utiliza para indicar la introducción de los datos de los centroides a enviar hacia el *host* en las FIFOs de salida correspondientes.

8.8. Etapa *SAM*

En esta etapa, se determinará la semejanza de cada firma espectral de los centroides determinados en la etapa anterior con unas firmas espectrales de referencia adjuntadas en un archivo y que caracterizan muestras de tejido cutáneo con y sin afección de cáncer de piel. El objetivo de esta etapa consiste en determinar qué referencia espectral se asemeja mejor a cada centroide. Con ello, se identifica el tipo de afección que señaliza el centroide debido a su firma espectral. Según el algoritmo de SAM, la comparación entre un centroide “ctr” y una firma espectral de referencia “ref” se ajusta a la ecuación (6).

$$SAM = arccos\left(\frac{\overrightarrow{ctr} * \overrightarrow{ref}}{|\overrightarrow{ctr}||\overrightarrow{ref}|}\right) \tag{6}$$

Cabe recordar que las coordenadas vectoriales de los centroides y las firmas espectrales de referencia, es decir, las bandas espectrales, poseerán en cualquier caso un

valor positivo. Por consiguiente, el valor de SAM resultante de cada comparación se acotará en el intervalo entre 0° y 90°. Dentro de este rango, se observa que el resultado siempre aumenta de valor cuando decrece el argumento de la función arco coseno. En consecuencia, es posible ordenar de menor a mayor resultado de SAM la totalidad de comparaciones entre los centroides y las firmas espectrales sin calcular el arco coseno. Esto se debe a que dicho orden será siempre el inverso al obtenido si el mismo criterio de ordenación se aplicase a los argumentos de la función arco coseno resultantes de cada combinación. Por este motivo, se excluye tanto del *host* como del *kernel* el cálculo del arco coseno, ya que no es necesario para que esta etapa determine la información buscada, simplificando la implementación. Como resultado, la fórmula modificada del SAM que se aplica en esta aplicación se correspondería a lo indicado en la ecuación (7).

$$\widetilde{SAM} = \frac{\overrightarrow{ctr} * \overrightarrow{ref}}{\left|\overrightarrow{ctr}\right|\left|\overrightarrow{ref}\right|} \tag{7}$$

Sin embargo, desde la perspectiva de cálculo únicamente se busca determinar qué firma espectral de referencia se asemeja mejor a cada centroide. Por ello, puede eliminarse de la ecuación (7) el cálculo del módulo del centroide, ya que dicho valor permanecerá constante entre los resultados a comparar. En definitiva, la fórmula a aplicar será la expuesta en la ecuación (8).

$$\widetilde{SAM} = \frac{\overrightarrow{ctr} * \overrightarrow{ref}}{\left|\overrightarrow{ref}\right|} \tag{8}$$

El producto escalar del numerador se calcula en el *kernel*, mientras que el módulo de cada firma espectral de referencia usada en el denominador se determina en el *host*. Con respecto al *kernel* FPGA, su componente central en la descripción HLS se corresponde a la función “coreSAM” (Código 88) descrita en el archivo “sam_hw_adapt.cpp” y, dentro de la cual, la función principal se denomina “getVectorDotProduct” (Código 89).

```
void coreSAM(sam_input_FIFO refs_str[_SAM_PARALL],
sized_d ctr[_SAM_PARALL], ref_mod_d refMod, ref_signature_id* simRef){
  #pragma HLS FUNCTION_INSTANTIATE variable=simRef
  #pragma HLS array_partition variable=refs_str complete
  #pragma HLS array_partition variable=ctr complete
  #pragma HLS INLINE
  static bool newCtr;

  //Inicializacion al valor maximo (evitara problemas con minCell)
  static sam_adder_out_d toDiv = (~(sam_adder_out_d)0);
  newCtr = ref_iter >= first_ref_iter_in_rst;

  getVectorDotProducts(refs_str, ctr, newCtr, &toDiv);

  if(newCtr){
    maxCell(toDiv, refMod, simRef);
  }

  ref_iter = (ref_iter < last_ref_iter)?
      ((ref_transfers_id)(ref_iter + 1)):((ref_transfers_id)0);
}
```

Código 88: Función coreSAM

```
void getVectorDotProducts(sam_input_FIFO refs_str[_SAM_PARALL],
sized_d ctr[_SAM_PARALL], bool newCtr, sam_adder_out_d* toDiv){
#pragma HLS FUNCTION_INSTANTIATE variable=toDiv
#pragma HLS array_partition variable=refs_str complete
#pragma HLS array_partition variable=ctr complete
#pragma HLS INLINE

  static sam_adder_out_d adderRes[_ADDER_RES];
  static mult_sized_d toAdder[_SAM_PARALL];
  #pragma HLS array_partition variable=toAdder complete

  //Activacion del reset
  if(newCtr){
    *toDiv = adderRes[adderRes_idx];
  }

  //Ejecuta producto escalar
  mult(refs_str, ctr, toAdder);
  SAM_adder.run(toAdder, newCtr, adderRes[adderRes_idx],
&(adderRes[adderRes_idx]));

  //Actualizacion del indice
  adderRes_idx = (adderRes_idx<MAX_ADDER_RES_ID_VAL)?
        (adder_res_id)(adderRes_idx+1):(adder_res_id)0;
}
```

Código 89: Función "getVectorDotProduct"

8.9. Diagrama de bloques de la aplicación

Como se especificó en apartados anteriores, la aplicación se subdivide en un *host*, ejecutado en la parte PS de la placa de prototipado, y tres *kernels* FPGA, ejecutados en la parte PL de dicha placa. El conexionado entre estos dos elementos *hardware* es concretado por la herramienta *software* Vivado, perteneciente al ecosistema de aplicaciones de Xilinx [64].

Partiendo de las directivas especificadas por el usuario y de la configuración por defecto, la arquitectura *hardware* resultante se ajusta al diagrama de bloques expuesto en la Figura 45. Dicho diagrama puede visualizarse al seleccionarse la opción “Open Block Design” desde la interfaz gráfica de Vivado para el proyecto “prj.xpr” asociado a la aplicación.

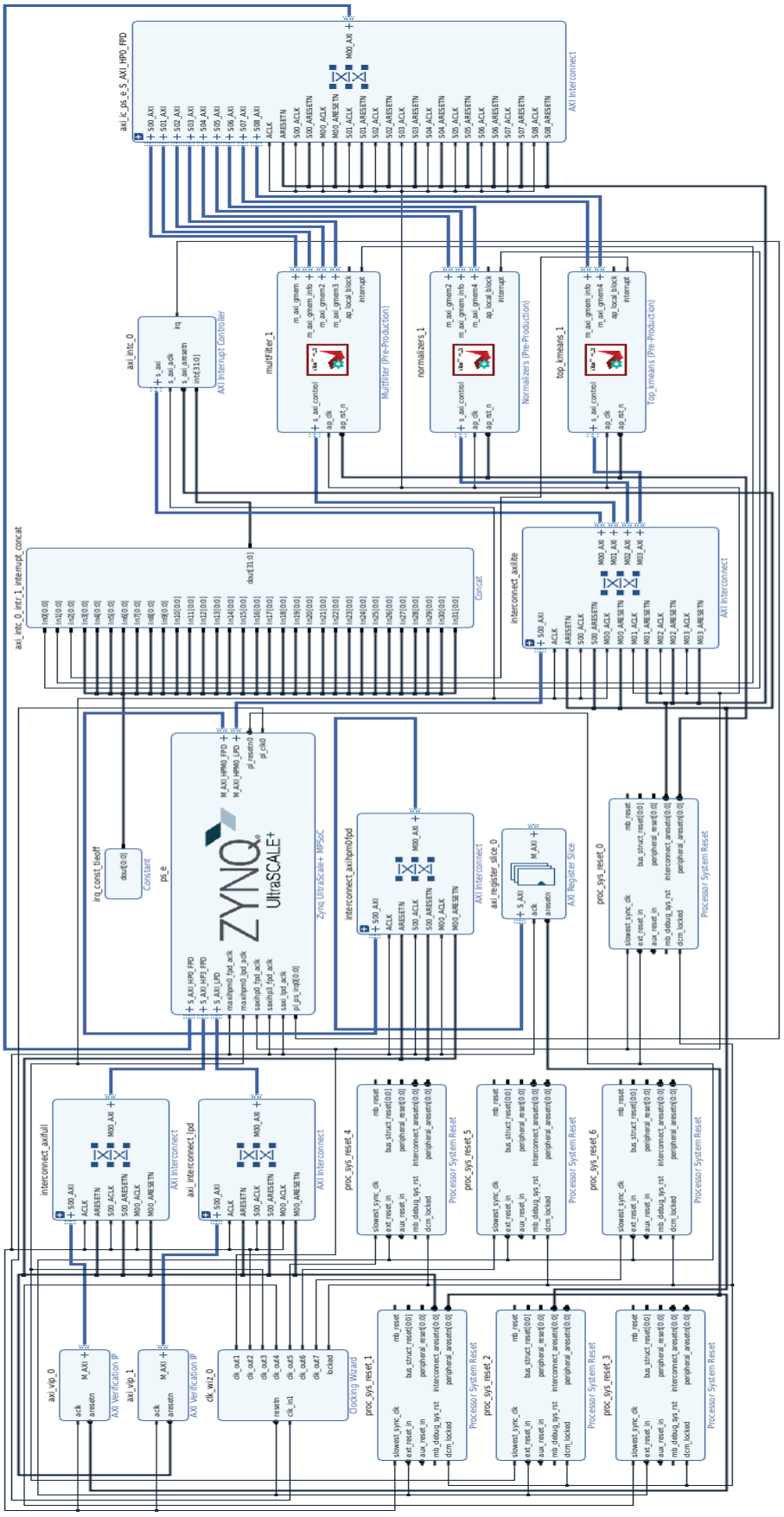


Figura 45: Diagrama de bloques de la aplicación

Dentro de este diagrama, la parte PS de la aplicación se especifica como el módulo "ps_e". Este módulo se expone más detalladamente en la Figura 46. Por otra parte, la sección del diagrama de bloques correspondiente a los *kernels* FPGA se muestra en la Figura 47.

Figura 46: Bloque PS del diagrama de bloques

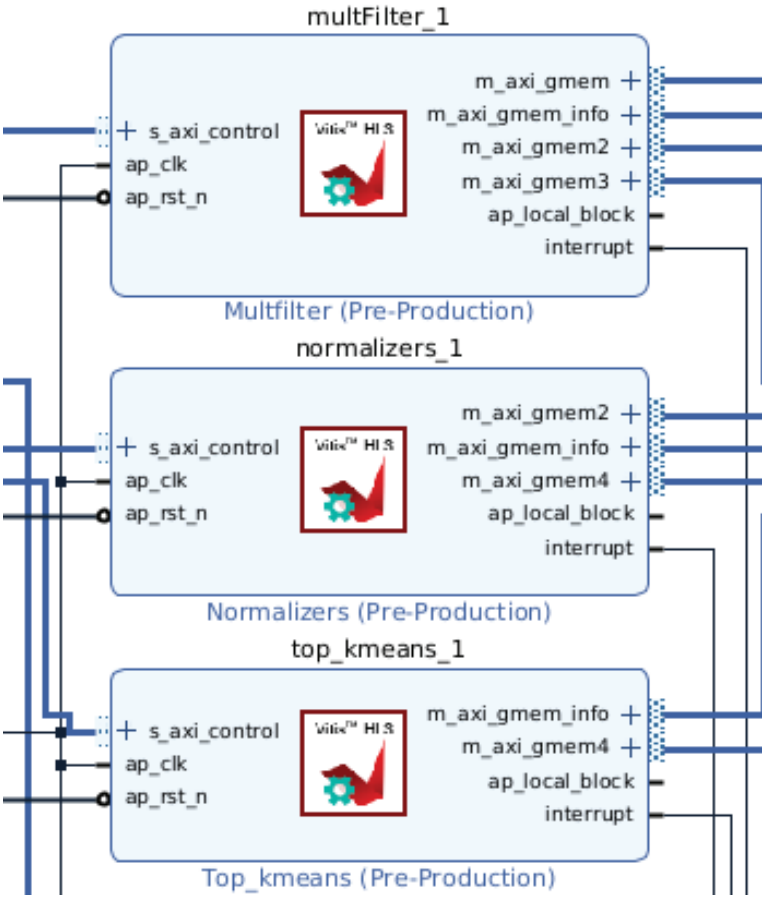


Figura 47: Kernels FPGA en el diagrama de bloques

Como se observa en la Figura 46, el PS utiliza múltiples puertos AXI en la arquitectura de la aplicación. Cada uno de estos puertos comunica en última instancia al PS con otros módulos AXI de la arquitectura o con los *kernels* FPGA de la aplicación utilizando como intermediarios unos *switches* de la tipología "AXI Interconnect" [65].

En la arquitectura de esta aplicación se asigna un "AXI Interconnect" de manera individual a cada puerto AXI procedente del PS, operando dicho puerto como interfaz maestra. Cada uno de estos módulos "AXI Interconnect" conmuta esta interfaz maestra con una o múltiples interfaces AXI esclavas. Estas interfaces esclavas conectan directamente con el módulo final a intercomunicar con el PS. Por otra parte, cada uno de los puertos AXI de cada módulo "AXI Interconnect" posee su propia entrada de reloj y *reset*. Asimismo, la parte PS de la aplicación genera una señal de reloj específica para cada módulo "AXI Interconnect", compartiéndose el mismo *pin* del PS para transmitir dicha señal entre los puertos AXI pertenecientes al mismo bloque "AXI Interconnect". Estos *pines* presentan un nombre similar al del puerto AXI del PS que se conecta al mismo módulo "AXI Interconnect", distinguiéndose fundamentalmente en que el nombre del *pin* incorpora la terminación "_aclk".

Entre los puertos AXI utilizados por el PS se encuentra el "S_AXI_HP0_FPD", el cual conecta al *host* con todas las interfaces "m_axi" especificadas en la descripción HLS de los *kernels* FPGA. Esta conexión se encuentra multiplexada mediante el componente "AXI Interconnect" denominado "axi_ic_ps_e_S_AXI_HP0_FPD" en el diagrama de bloques. Otro puerto AXI relevante es el "M_AXI_HPM0_LPD". Este puerto conecta al PS con las interfaces "s_axi_control" presentes en cada *kernel* FPGA y requeridas para regular la ejecución de estos desde el *host*.

Aparte de las interfaces "s_axi_control", el puerto "M_AXI_HPM0_LPD" conecta con el módulo "AXI Interrupt Controller". Este último módulo se utiliza para configurar y conmutar las interrupciones IRQ (*Interrupt Request*) utilizadas por los *kernels* FPGA para interrumpir la ejecución del *host* [66]. Para solicitar dicha interrupción, cada *kernel* FPGA dispone de un *pin* denominado "interrupt" en el diagrama de bloques. Estos *pins* se concatenan entre sí, como indica el módulo "Concat" del diagrama de bloques [67], de manera previa a ser conectados al módulo "AXI Interrupt Controller". Este último módulo

conecta con el *pin* identificado como "pl_ps_irq0[0:0]" en el módulo del diagrama de bloques correspondiente al PS. Este último *pin* será el utilizado para notificar al *host* las solicitudes de interrupción procedentes de la FPGA.

La introducción de la señal de reloj en los *kernels* FPGA se asocia en el diagrama de bloques al *pin* "ap_clk" presente de manera individual en cada *kernel*. Estos *pines* conectan con la salida "clk_out1" del módulo "Clocking Wizard". El cometido de este último módulo consiste en generar varias señales de reloj de diferentes frecuencias, correspondiéndose la salida "clk_out1" a una frecuencia de 150 MHz [68] que suponen la entrada de los *kernels* implementados en la FPGA. El "Clocking Wizard" precisa de una señal de reloj de entrada, que parte del PS a través del *pin* "pl_clk0" presente en el bloque "ps_e". Además, el "Clocking Wizard" posee un *reset* que es controlado por el PS mediante una conexión directa ("pl_resetn0").

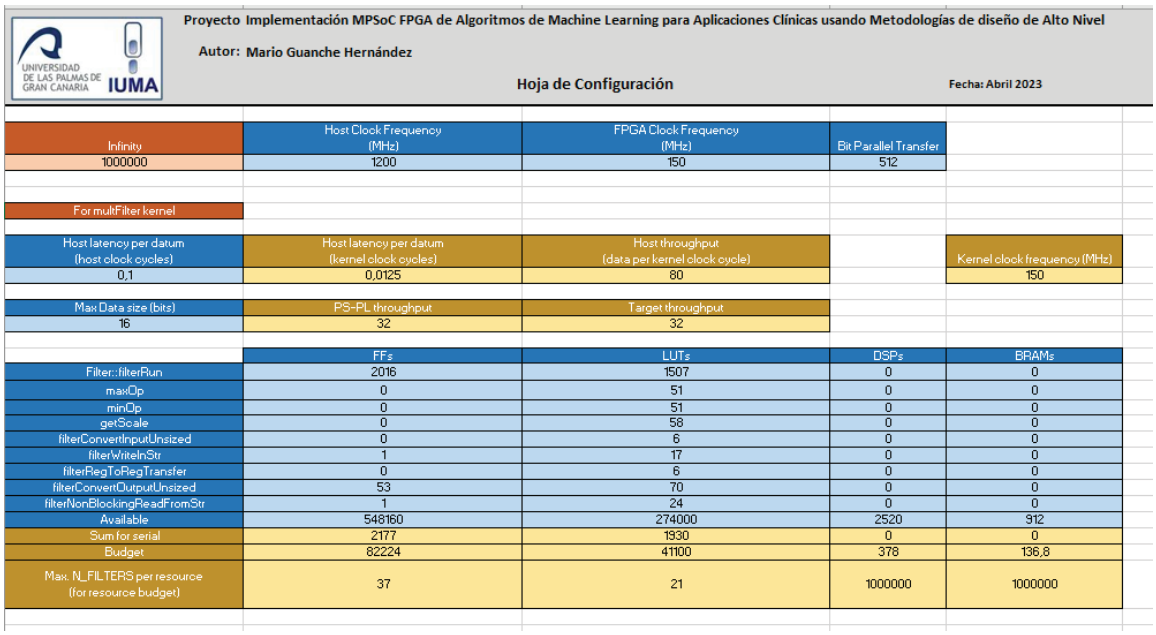
El *reset* de los *kernels* FPGA constituye un *reset* activo a nivel bajo que se identifica en el diagrama de bloques como el *pin* "ap_rst_n" en el módulo de cada *kernel*. La activación del reset de estos *kernels* es controlada por el PS ("pl_resetn0"). Aparte del "Clocking Wizard" y de los *kernels* FPGA, el *pin* "pl_resetn0" permite reiniciar los módulos AXI4. Entre estos módulos AXI se encuentran, por ejemplo, los módulos "AXI Interconnect" mencionados con anterioridad. El reset de los *kernels* FPGA y de los módulos AXI4 se realiza mediante el "Processor System Reset" [69] ("proc_sys_reset_0").

8.10. Configuración de los *kernels* mediante hoja de cálculo

Para determinar la configuración estática idónea con la que establecer los *kernels* FPGA, deben considerarse los aspectos indicados al respecto en el apartado 8.2. Asimismo, para alcanzar el mejor compromiso entre dichos aspectos es necesario conocer múltiples datos numéricos. Entre la información necesaria se encuentran las posibilidades de paralelización de las transferencias PS-PL, la disponibilidad de recursos PL en la FPGA, la proporcionalidad entre el grado de paralelización del procesamiento exigido y la utilización de recursos PL, etc.

A partir de esta información, puede diseñarse una hoja de cálculo que asiste al usuario para determinar la configuración óptima de los *kernels* FPGA teniendo en cuenta

las exigencias algorítmicas de la aplicación y la disponibilidad de recursos de la plataforma MPSoC (Figura 48). Cabe destacar que parte de la información de entrada a esta hoja de cálculo se determina mediante la utilización de herramientas de análisis que requieren modificaciones mínimas y puntuales del código de la aplicación para extraer la información deseada.



Proyecto Implementación MPSoC FPGA de Algoritmos de Machine Learning para Aplicaciones Clínicas usando Metodologías de diseño de Alto Nivel

Autor: Mario Guanche Hernández

Hoja de Configuración

Fecha: Abril 2023

Infinity	Host Clock Frequency (MHz)	FPGA Clock Frequency (MHz)	Bit Parallel Transfer	
1000000	1200	150	512	
For multFilter kernel				
Host latency per datum (host clock cycles)	Host latency per datum (kernel clock cycles)	Host throughput (data per kernel clock cycle)		Kernel clock frequency (MHz)
0,1	0,0125	80		150
Max Data size (bits)	PS-PL throughput	Target throughput		
16	32	32		
	FFs	LUTs	DSPs	BRAMs
Filter::filterRun	2016	1507	0	0
maxOp	0	51	0	0
minOp	0	51	0	0
getScale	0	58	0	0
filterConvertInputUnsized	0	6	0	0
filterWriteInStr	1	17	0	0
filterRegToRegTransfer	0	6	0	0
filterConvertOutputUnsized	53	70	0	0
filterNonBlockingReadFromStr	1	24	0	0
Available	548160	274000	2520	912
Sum for serial	2177	1930	0	0
Budget	82224	41100	378	136,8
Max. N_FILTERS per resource (for resource budget)	37	21	1000000	1000000

Figura 48. Vista parcial de la hoja de cálculo de configuración de los kernels

La hoja de cálculo se divide en varias secciones. La primera sección permite especificar algunos parámetros comunes a la configuración de todos los *kernels* FPGA. Por otra parte, las demás secciones se dedican a cada uno de los *kernels* FPGA de la aplicación. Cada una de ellas se encuentra encabezada por un título que identifica al *kernel* al que se asocia. En la hoja de cálculo se incluyen celdas para especificar datos de entrada, coloreadas en azul, y celdas donde se exponen los valores finales resultantes de los cálculos que debe asignarse, según estos, a determinados parámetros de configuración estática de los *kernels*. Estas últimas celdas se encuentran coloreadas en verde dentro de la hoja de cálculo y se encuentran encabezadas por el nombre de la variable estática del código HLS de la aplicación a la que hacen referencia. También se encuentran otras celdas de color sepia, donde se especifican valores intermedios del proceso de cálculo que permiten depurar los resultados obtenidos. En la Figura 49 se muestra un ejemplo datos relativos al *kernel* “normalizers”.

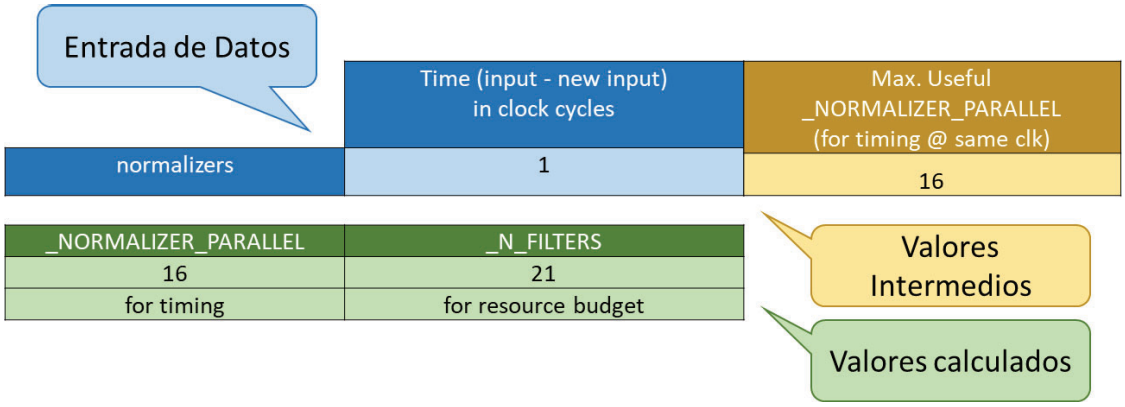


Figura 49: Sistema de colores de la hoja de cálculo

En esta hoja de cálculo se consideran, para cada *kernel* FPGA a configurar, una serie de parámetros cuyas implicaciones a evaluar en los cálculos se corresponden al ámbito del rendimiento temporal. Estos parámetros se utilizarán para determinar el grado de paralelización de cada *kernel* FPGA a implementar para optimizar el *throughput*. Entre los parámetros de entrada podemos citar los siguientes:

- Parámetros comunes:
    - Frecuencia de reloj del *host* (MHz) y de los *kernels* FPGA. Se identifican como “Host Clock Frequency (MHz)”, y “FPGA Clock Frequency (MHz)”.
    - Máximo grado de paralelización posible en *bits* de las transferencias entre el *host* y los *kernels* FPGA considerando la interfaz física de intercomunicación utilizada, identificado como “Bit Parallel Transfer”.
- Parámetros personalizables para cada *kernel*:
    - “Host latency per datum (host clock cycles)”. Este dato representa la cantidad promedio de ciclos de reloj que tarda el *host* en enviar cada nuevo dato al *kernel* FPGA que debe actualizarse recurrentemente en la gran mayoría de las ejecuciones de este último.
    - “Max Data Size (*bits*)”. Este parámetro se corresponde al máximo tamaño en *bits* de cualquier tipo de dato que sea generado o procesado en paralelo por el *kernel* FPGA a configurar. Debido a algunas limitaciones en la configuración de las transferencias entre el *host* y el *kernel*, dicho tamaño máximo debe especificarse redondeado al alza a

un tamaño de dato de valor entero nativo de C, es decir, a 8, 16, 32 o 64 *bits*. Además, el tamaño del tipo de dato no debe ser superior a 64 *bits*.

- "Time (input – new input) in clock cycles". Permite indicar cuál es la cantidad de ciclos de reloj del *kernel* que separan el inicio de ejecuciones sucesivas del mismo debido al funcionamiento interno del *kernel*.

La presencia de estos parámetros temporales en la hoja de cálculo se muestra, ejemplificada para el *kernel* "multFilter", en la Figura 50. No obstante, de entre estos parámetros se exceptúa "Time (input – new input)", el cual se expone en la Figura 51 para el mismo *kernel*. En la Figura 51 también se encuentra un resultado intermedio muy útil denominado "Max. Useful N_FILTERS (for timing @ same clk)". Dicho resultado indica la paralelización del procesamiento del *kernel* "multFilter" máxima que es aprovechable debido a las limitaciones de *throughput* entre las conexiones PS-PL empleadas. Este resultado también puede observarse para los demás *kernels* con una designación similar.

Infinity	Host Clock Frequency (MHz)	FPGA Clock Frequency (MHz)	Bit Parallel Transfer
1000000	1200	150	512
For multFilter kernel			
Host latency per datum (host clock cycles)	Host latency per datum (kernel clock cycles)	Host throughput (data per kernel clock cycle)	
0,1	0,0125	80	
Max Data size (bits)	PS-PL throughput	Target throughput	
16	32	32	

Figura 50: Parámetros temporales comunes y del throughput del kernel "multFilter"

	Time (input - new input) in clock cycles	Max. Useful N_FILTERS (for timing @ same clk)
multFilter	2	64

Figura 51: Parámetro "Time (input – new input)" del kernel "multFilter"

Otra faceta del ajuste de los *kernels* se corresponde a la utilización de recursos de la FPGA. Estos recursos se subdividen en varios tipos, debiéndose especificar la cantidad de cada uno que requiere cada módulo funcional del *kernel* a evaluar. Con ello, puede

determinarse el grado de paralelización del procesamiento del *kernel* que es implementable en la FPGA.

Por lo expuesto en el apartado 7.4.2.5, se precisa efectuar el *inlining* en la implementación final de estos módulos en el *kernel* FPGA. En consecuencia, para determinar la utilización de recursos de cada módulo, debe efectuarse una compilación *hardware* en la que se anule el *inlining* de las funciones en la descripción HLS correspondientes a estos módulos, pero manteniendo las exigencias de *pipelining* que la implementación de dicha funcionalidad debe poseer. Los módulos funcionales se identifican en la hoja de cálculo con el nombre de la función HLS que los describe, siendo los siguientes según el *kernel* FPGA.

- *Kernel* de filtrado "multFilter":
    - "Filter::filterRun". Identifica al módulo de filtrado.
    - "maxOp". Representa el módulo para determinar el valor máximo en la firma espectral filtrada del píxel en evaluación. Cabe recordar que este módulo procesa una sola banda espectral de un solo píxel por ejecución.
    - "minOp". Identifica al módulo que determina el valor mínimo en la firma espectral filtrada del píxel a analizar. A excepción de este aspecto funcional, este módulo es análogo a "maxOp", siendo esperable que coincida en cantidad y tipo de recursos FPGA que utiliza.
    - "getScale". Constituye el módulo que calcula el valor de escala como la diferencia entre el valor máximo y mínimo en la firma espectral filtrada de cada píxel.
    - "filterConvertInputUnsized". Representa el módulo que adapta los datos del hipercubo entrantes al *kernel* "multFilter" del formato para su transmisión "trans_unsized_d" al formato "unsized_d" para su procesamiento dentro del *kernel* FPGA.
    - "filterWriteInStr". Este módulo introduce en una FIFO de entrada al procesamiento del *kernel* "multFilter" el dato transformado por el módulo "filterConvertInputUnsized" asociado.

- "filterRegToRegTransfer". Representa un módulo al que se recurre para el traspaso de algunos datos individuales entre registros dentro del *kernel* "multFilter".
- "filterNonBlockingReadFromStr". Identifica al módulo que efectúa una lectura no bloqueante de una FIFO de salida del procesamiento del *kernel*. De esta lectura, se obtiene un dato filtrado del hipercubo generados por el procesamiento del *kernel*.
- "filterConvertOutputUnsized". Transforma el dato saliente del módulo "filterNonBlockingReadFromStr" asociado del formato "unsized_d" en el que se generó al formato "trans_unsized_d" para su envío.

En la Figura 52 se expone la sección de la hoja de cálculo donde deben indicarse los aspectos de utilización de recursos FPGA de los módulos funcionales del *kernel* "multFilter".

	FFs	LUTs	DSPs	BRAMs
Filter::filterRun	2016	1507	0	0
maxOp	0	51	0	0
minOp	0	51	0	0
getScale	0	58	0	0
filterConvertInputUnsized	0	6	0	0
filterWriteInStr	1	17	0	0
filterRegToRegTransfer	0	6	0	0
filterConvertOutputUnsized	53	70	0	0
filterNonBlockingReadFromStr	1	24	0	0
Available	548160	274000	2520	912
Sum for serial	2177	1930	0	0
Budget	82224	41100	378	136,8
Max. N_FILTERS per resource (for resource budget)	37	21	1000000	1000000

Figura 52: Sección sobre los módulos funcionales del kernel "multFilter"

- *Kernel* de normalizado "normalizers":
  - "normalizer". Este módulo efectúa el normalizado por ejecución de un dato filtrado del hipercubo. Constituye el módulo fundamental del *kernel* "normalizers".

- "normConvertInputUnsized". Representa el módulo que adapta los datos filtrados del hipercubo de entrada al *kernel* "normalizers" del formato para su transmisión "trans_unsized_d" al formato "unsized_d" para su procesamiento dentro del *kernel* FPGA.
- "normWriteInStr". Este módulo introduce, en una FIFO de entrada al procesamiento del *kernel* "normalizers", el dato obtenido del módulo "normConvertInputUnsized" correspondiente.
- "normNonBlockingReadFromStr". Constituye el módulo para la lectura no bloqueante de una FIFO de salida del procesamiento del *kernel* "normalizers". De esta lectura se obtiene un dato normalizado del hipercubo generado tras la ejecución del *kernel* "normalizers".
- "normConvertOutputSized". Transforma el dato saliente del módulo "normNonBlockingReadFromStr" asociado del formato "sized_d" en el que se generó al formato "trans_sized_d" para su envío.
- "normRegToRegTransfer". Representa un módulo al que se recurre para el traspaso de algunos datos individuales entre registros dentro del *kernel* "normalizers".

En la Figura 53 se muestra la parte de la hoja de cálculo donde deben especificarse la utilización de recursos FPGA de los módulos funcionales del *kernel* "normalizers".

	FFs	LUTs	DSPs	BRAMs
multFilter	164299	28380	0	92
normalizer	2319	1789	0	0
normConvertInputUnsized	0	6	0	0
normWriteInStr	1	17	0	0
normNonBlockingReadFromStr	1	24	0	0
normConvertOutputSized	87	43	0	0
normRegToRegTransfer	0	6	0	0
Available	383861	245620	2520	820
Sum for serial	2408	1897	0	0
Budget	76772,2	49124	504	164
Max. NORMALIZER_PARALLEL per resource (for resource budget)	31	25	1000000	1000000

Figura 53: Sección sobre los módulos funcionales del kernel "normalizers"

- *Kernel* "top_kmeans":
    - "coord_diff". Representa el módulo para obtener la diferencia cuadrática entre 2 valores de una misma banda espectral, perteneciendo uno a la firma espectral de un píxel y otro a un centroide.
    - "Min_ctr::minRun". Representa el módulo de obtención del valor mínimo mediante árbol de bifurcación, que se utiliza para determinar el centroide más próximo al último píxel evaluado.
    - "Kmeans_Adder::run". Identifica el módulo sumador mediante árbol de bifurcación realimentado que se emplea para determinar la distancia entre el píxel en evaluación y los centroides. La cantidad de réplicas de este módulo no varía en función del paralelismo del procesamiento del *kernel* "top_kmeans" configurado. Sin embargo, dicho paralelismo sí modifica la cantidad de recursos FPGA que este módulo implica, debido a que modifica la cantidad de módulos de suma "Kmeans_Adder::operation" subyacentes. Estos últimos módulos dentro de "Kmeans_Adder::run" se encuentran en una cantidad equivalente al paralelismo del procesamiento.

      Resultaría más idóneo considerar la utilización de recursos de los módulos "Kmeans_Adder::operation" en lugar de "Kmeans_Adder::run" en la hoja de cálculo. No obstante, esto no es posible debido a que la función "Kmeans_Adder::operation" requiere parametrización de *template*, lo que imposibilita anular el *inlining* de la misma. Por ello, su utilización de recursos de FPGA se estima, en la hoja de cálculo, como el cociente entre la cantidad de recursos *hardware* que conlleva "Kmeans_Adder::run" para un paralelismo determinado y dicho grado de paralelismo. Este último dato se introduce como "Preset _KMEANS_PARALLEL" en la hoja de cálculo.
    - "uptRawCtrCoord". Constituye el módulo para actualizar las sumatorias, de una misma banda espectral, de los valores de los píxeles

pertenecientes a un mismo centroide. En una misma ejecución, este módulo actualiza todos los centroides involucrados.

- "uptCtrCoord". Representa el módulo que recalcula, para una banda espectral, el nuevo valor de los centroides implicados en la reubicación del píxel evaluado.
- "kmeansFromStrToDat". Este módulo se emplea para extraer mediante lectura bloqueante un dato de una FIFO de entrada al procesamiento del *kernel* "top_kmeans". Este dato se corresponde a una banda espectral.
- "kmeansFromStrToUselessDat". Este módulo se emplea para vaciar mediante lectura bloqueante un dato de una FIFO de entrada al procesamiento del *kernel* "top_kmeans". Este dato se corresponde a una banda espectral.
- "setCtrDat". Este módulo se aplica en la inicialización por ejecución de los datos relativos a una banda espectral de un centroide.
- "kmeansFromDatToStr". Este módulo se utiliza para introducir una banda espectral de uno de los centroides generados tras finalizar el *clustering* de la imagen en una FIFO que lo dirige a la salida del *kernel* "top_kmeans".
- "kmeansConvertInputSized". Representa el módulo que adapta los datos normalizados del hipercubo de entrada al *kernel* "top_kmeans" del formato para su transmisión "trans_sized_d" al formato "sized_d" para su procesamiento dentro del *kernel* FPGA.
- "kmeansWriteInStr". Este módulo introduce, en una FIFO de entrada al procesamiento del *kernel* "top_kmeans", el dato obtenido del módulo "kmeansConvertInputSized" correspondiente.
- "kmeansNonBlockingReadFromStr". Constituye el módulo para la lectura no bloqueante de una FIFO de salida del procesamiento del *kernel* "top_kmeans". De esta lectura se obtiene una banda espectral de uno de los centroides obtenidos tras concluir la etapa de *clustering*.

- "kmeansConvertOutputSized". Transforma el dato saliente del módulo "kmeansNonBlockingReadFromStr" asociado del formato "sized_d" en el que se generó al formato "trans_sized_d" para su envío.

En la Figura 54 se muestra la parte de la hoja de cálculo donde deben especificarse la utilización de recursos FPGA de los módulos funcionales del *kernel* "top_kmeans".

	FFs	LUTs	DSPs	BRAMs
normalizers	45698	34766	0	90
prev_kernel_reserved	209997	63146	0	182
kmeansFromStrToDat	1	17	0	0
kmeansFromDatToStr	1	17	0	0
kmeansFromStrToUselessDat	1	17	0	0
Kmeans_Adder::run	121	317	0	0
Kmeans_Adder::operation	21	53	0	0
Min_ctr::run	0	87	0	0
coord_diff	68	87	0	0
setCtrDat	0	6	0	0
uptCtrCoord	3912	2944	0	0
uptRawCtrCoord	0	76	0	0
kmeansConvertInputSized	0	6	0	0
kmeansWriteInStr	1	17	0	0
kmeansNonBlockingReadFromStr	1	24	0	0
kmeansConvertOutputSize	358	1089	0	0

Figura 54: Sección sobre los módulos funcionales del kernel "top_kmeans"

Aparte de conocerse la utilización de recursos FPGA de los módulos funcionales, debe determinarse las cantidades de estos distintos tipos de recursos que pueden utilizarse en la implementación *hardware* del *kernel* a evaluar. Para ello, debe especificarse, en primer lugar, las disponibilidades de recursos FPGA presentes en el MPSoC donde la aplicación es implementada. Dicha información puede consultarse en la documentación técnica de dicho dispositivo y debe indicarse en la fila "Available". Esta fila se ubica en la sección de la hoja de cálculo donde se evalúa la utilización de recursos del primer *kernel* FPGA (Figura 52).

Por otra parte, deben considerarse los recursos utilizados por los *kernels* FPGA cuyas implementaciones *hardware* convivan con el *kernel* a evaluar. Este aspecto se introduce en la hoja de cálculo de 2 maneras diferentes, dependiendo de si el *kernel* FPGA

que comparte los recursos con el *kernel* objeto de la estimación se ejecuta antes o después de este último.

En el caso de los *kernels* FPGA ejecutados con anterioridad, debe obtenerse mediante compilación *hardware* la utilización de recursos de dicho *kernel* FPGA para sustraerla de la disponible para el *kernel* a evaluar. Esta información debe especificarse en las filas cuyos nombres coincidan con los de los *kernel* FPGA de ejecución previa. En el caso, por ejemplo, del *kernel* "top_kmeans" (Figura 54), se encuentra la fila "nomalizers" para indicar la utilización de recursos FPGA que implica el *kernel* homónimo. Sin embargo, no existe ninguna fila denominada "multFilter", correspondiente al primer *kernel* en ejecutarse, dentro de la sección de la hoja de cálculo del *kernel* "top_kmeans". Dicha fila se encuentra únicamente en la sección atribuida al *kernel* FPGA de ejecución inmediatamente posterior al *kernel* "multFilter", siendo este el *kernel* "normalizers". Este suceso se muestra en la Figura 53 anteriormente expuesta.

Si el *kernel* FPGA se ejecuta *a posteriori*, puede reservarse un porcentaje de los recursos FPGA para la implementación *hardware* de dicho *kernel*. Dicha reserva se considera para todos los *kernels* FPGA que, con respecto al *kernel* en evaluación, se ejecutan posteriormente. También se considera para otros aspectos de la implementación *hardware* de los *kernels* FPGA, incluyendo el *kernel* en evaluación, no estimados de forma más precisa en los cálculos. Esta reserva se especifica, para cada *kernel* FPGA, como el porcentaje complementario al indicado en la celda etiquetada como "% usable" dentro de la sección del *kernel* a evaluar. Este último dato indica el porcentaje de recursos asignado a los módulos funcionales del *kernel* en evaluación. Cabe destacar que el "% usable" es con respecto a la disponibilidad de recursos FPGA del MPSoC tras sustraer la utilización de recursos FPGA de los *kernels* de ejecución previa e implementación coexistente con el *kernel* a evaluar.

Tras estas operaciones se obtienen los resultados intermedios de la fila "Budget" para cada *kernel* FPGA, la cual indica los recursos estimados como verdaderamente disponibles para los módulos funcionales del *kernel* en evaluación. A partir de los valores de utilización y disponibilidad de recursos PL, puede determinarse el paralelismo del procesamiento del *kernel* FPGA en evaluación que es implementable. Para el *kernel*

"normalizers", esta paralelización se identifica con el resultado intermedio "Max. _NORMALIZER_PARALLEL (for resource budget)" (Figura 55). No obstante, esta paralelización se indica también para los demás *kernels* con una nomenclatura análoga.

Budget	76772,2	49124	504	164
Max. NORMALIZER_PARALLEL per resource (for resource budget)	31	25	1000000	1000000

Max. _NORMALIZER_PARALLEL (for resource budget)
25

Figura 55: "Budget" y máximo paralelismo implementable del kernel "normalizers"

La utilización de recursos FPGA de los *kernels* calculada es una aproximación. Esto se debe a que obvia las modificaciones que, para cada réplica de los módulos funcionales, ocasiona la aplicación del *inlining*. Tampoco considera la utilización de recursos que implican los elementos de interconexión entre los módulos funcionales y entre el *kernel* y el *host*; y excluye otras funcionalidades del *kernel* FPGA no recogidas en su versión completamente serial por alguna función de la descripción HLS. Por ello, la evaluación de la utilización de recursos FPGA que efectúa esta hoja de cálculo se considera pesimista.

Al comparar los resultados de paralelización del *kernel* FPGA calculados, se obtiene, según la hoja de cálculo, la paralelización del procesamiento que debe tener el *kernel* en evaluación. Esta paralelización es determinada automáticamente en la hoja de cálculo, y puede establecerse debido a limitaciones de la velocidad de la transferencia de datos entre el *host* y el *kernel* FPGA (indicación "for timing") o bien a limitaciones en la disponibilidad de recursos FPGA (indicación "for resource budget"). En la Figura 56 se ejemplifica esta sección de la hoja de cálculo para el *kernel* "multFilter".

	Time (input - new input) in clock cycles	Max. Useful N_FILTERS (for timing @ same clk)
multFilter	2	64

Max. Useful N_FILTERS (for timing)
64

_N_FILTERS	_N_FILTERS_PER_TRANSFER
21	1
for resource budget	

Figura 56: Paralelización resultante del kernel "multFilter"

En el caso del *kernel* "top_kmeans", existe un factor de limitación adicional debido a la máxima cantidad de bandas espectrales útiles que se ha establecido por configuración estática que la aplicación es capaz de manejar. Cabe recordar que esta cantidad se especifica en el código fuente con el valor de la macro "_MAX_BANDS", introduciéndose en la sección de la hoja de cálculo del *kernel* en una celda homónima (Figura 57). Asimismo, ningún grado de paralelización del procesamiento del *kernel* "top_kmeans" superior a "_MAX_BANDS" resultará de utilidad. Este hecho se debe a que, como se indicó en el apartado 8.7. , no pueden convivir datos de píxeles distintos en una misma transferencia desde el *host* hacia el *kernel* "top_kmeans".

_N_CLUSTERS	_N_COMPARATORS (@ _N_CLUSTERS)	_MAX_PX	_MAX_BANDS	Preset _KMEANS_PARALLEL
3	3	"2500 (en principio no se necesita especificar)"	116	6

Figura 57: Parámetros de configuración estática en las estimaciones de "top_kmeans"

Además del grado de paralelización del procesamiento, con esta hoja de cálculo también se determina el valor que el usuario debe especificar a otras macros para que la configuración estática de los *kernels* sea funcional.

En esta aplicación, las macros a configurar de este tipo se denominan "_N_COMPARATORS" y "_ADDER_REG". Ambas macros se encuentran declaradas en el archivo "amounts.h" y su especificación se requiere para la configuración estática del *kernel* "top_kmeans".

El valor de "_N_COMPARATORS" debe ajustarse en función del valor de la macro "_N_CLUSTERS" (Figura 57), mientras que "_ADDER_REG" ha de configurarse en función del valor de "_KMEANS_PARALLEL". Ambas correlaciones se ajustan mediante un mismo algoritmo, "AdderReg", implementado en "Visual Basic", obteniendo el valor que debe especificarse para las macros "_N_COMPARATORS" y "_ADDER_REG". El código fuente de la función "AdderReg" se expone en el Código 90.

```
Function AdderReg(cell As Integer) As Integer
    ' caso especial cell = 0.
    If cell = 0 Then
        AdderReg = 0
    Else
        Dim res As Integer
        res = 0
        Dim n As Integer
        n = cell
        'caso especial cell = 1.
        If n = 1 Then
            AdderReg = 1
        'casos habituales.
        Else
            While (n <> 1)
                Dim n_next As Integer
                n_next = n \ 2
                If ((n - 2 * n_next) <> 0) Then
                    n_next = n_next + 1
                End If
                res = res + n_next
                n = n_next
            Wend
            AdderReg = res
        End If
    End If
End Function
```

Código 90: Función "AdderReg" para la hoja de cálculo

8.11. Prototipado de la aplicación

Tras completar el flujo de diseño descrito en el capítulo 7 para la placa de prototipado Xilinx ZCU102, la aplicación desarrollada se transforma en un conjunto de archivos empaquetados en el archivo de imagen "sd_card.img". Esta imagen contiene los archivos necesarios para arrancar un sistema operativo PetaLinux y cargar un sistema de ficheros que facilite la ejecución de la aplicación [14], [52], [70].

La tarjeta SD requiere dos particiones, creándose ambas mediante la invocación del comando "fdisk" como se muestra en el Código 91. En dicho código, "<dir. tarjeta SD>" constituye el directorio que identifica en el PC al dispositivo de almacenamiento. Este comando debe invocarse desde el modo superusuario. Para implementar la aplicación en la ZCU102, ambas particiones deben ser primarias "p" (Tabla 8) [71], [72] [73].

```
sudo fdisk <dir. tarjeta SD>
```

Código 91: Comando "fdisk" [72]

Tabla 8: Especificación de las particiones (adaptado de [72]*)*

Partición	Partition type	Last sector
1	P	21111220
2	P	

La primera partición debe ajustarse al formato VFAT (*Virtual File Allocation Table*). La segunda partición debe configurarse con el formato EXT4 (*Fourth Extended File System*). Para asignar estos formatos deben invocarse los comandos "mkfs.vfat" y "mkfs.ext4" respectivamente desde el modo superusuario. Estas invocaciones deben ejecutarse siguiendo el formato expuesto en el Código 92, donde "<comando>" se sustituye por el nombre del comando y "<partición>" por el directorio asociado por el PC a la partición que corresponda [72], [73].

```
sudo <comando> <partición>
```

Código 92: Especificación del formato de cada partición [72]

Mediante el comando "dd" se graba la tarjeta SD con la imagen de arranque creada. Este comando debe invocarse en modo superusuario (sudo) (Código 93). Con respecto a los parámetros de dicho comando, "if" especifica el archivo de entrada, en este caso, "sd_card.img" y "of" especifica el directorio de destino que, en esta ocasión, se corresponde a aquel que el sistema operativo de Linux ha montado en la tarjeta SD [14], [74], [71].

```
sudo dd if=<archivo "sd_card.img"> of=<tarjeta SD>
```

Código 93: Introducción de la aplicación en la tarjeta SD

Es preciso configurar la placa para que realice el arranque desde la tarjeta SD. Esta configuración se establece mediante los *switches* del componente SW6 de la placa, que deben estar configurado como aparece en la Figura 58 [14], [75].

Figura 58. Configuración del modo de arranque mediante tarjeta SD

Con esta configuración, se introduce la tarjeta SD y se enciende la placa de prototipado. Tras ello, se produce el arranque de la placa, el cual se señaliza mediante el LED "Init B". Mientras este LED permanece en rojo, implica que el arranque no se ha completado. La finalización del arranque se indica cuando el LED se enciende en verde (Figura 59) [14].

Figura 59: Indicación del fin del arranque de la placa de prototipado

Una vez arrancada la aplicación, se configura una conexión SSH (*Secure Shell*). Para establecer esta conexión, se invoca desde un terminal del PC el comando del Código 94. En dicho comando, "<IP de la placa>" debe reemplazarse por la IP (*Internet Protocol*) que identifique a la placa de prototipado. De la misma forma, se pude asignar un nombre a la placa mediante la configuración en un servidor DHCP [76], [77].

```
ssh root@<IP de la placa>|<DNS de origen>
```

Código 94: Establecimiento de conexión SSH entre el PC y la placa de prototipado

Adicionalmente, se utiliza la conexión SSH para la transferencia de ficheros desde el PC a la placa de prototipado y viceversa mediante el comando "scp". Dicho comando debe seguir el formato expuesto en el Código 95 [78].

```
scp <usuario de origen>@<IP o DNS de origen>:<dir. fichero>  <usuario de destino>@<IP o DNS de destino>:<dir. destino>
```

Código 95: Comando "scp" para la transferencia de archivos

Alternativamente, el usuario puede interactuar con la placa de prototipado directamente, mediante su propia interfaz gráfica basada en X11. La visualización de esta interfaz se efectúa al conectar una pantalla al puerto DisplayPort de la placa. Como periférico de entrada, puede conectarse un teclado y ratón al puerto físico USB 2.0 ULPI (UTMI+ *Low Pin Interface*) PHY de la placa. Para que este teclado sea reconocido por el sistema ejecutado en la placa, es necesario conectar el *jumper* que se indica en la Figura 60. Con ello, se habilita que este puerto físico posea soporte OTG (*On The Go*) [14], [79].

Figura 60: Establecimiento del soporte OTG para la introducción de teclado [14]

Una vez establecido el acceso al sistema ejecutado sobre la placa de prototipado, el usuario debe invocar la aplicación con las opciones correspondientes. Este ejecutable se ubica en la primera partición de la tarjeta SD. Dicha partición se asocia al directorio "/media/sd-mmcblk0p1" dentro del sistema. Las conexiones establecidas con la placa de prototipado se exponen gráficamente en la Figura 61.

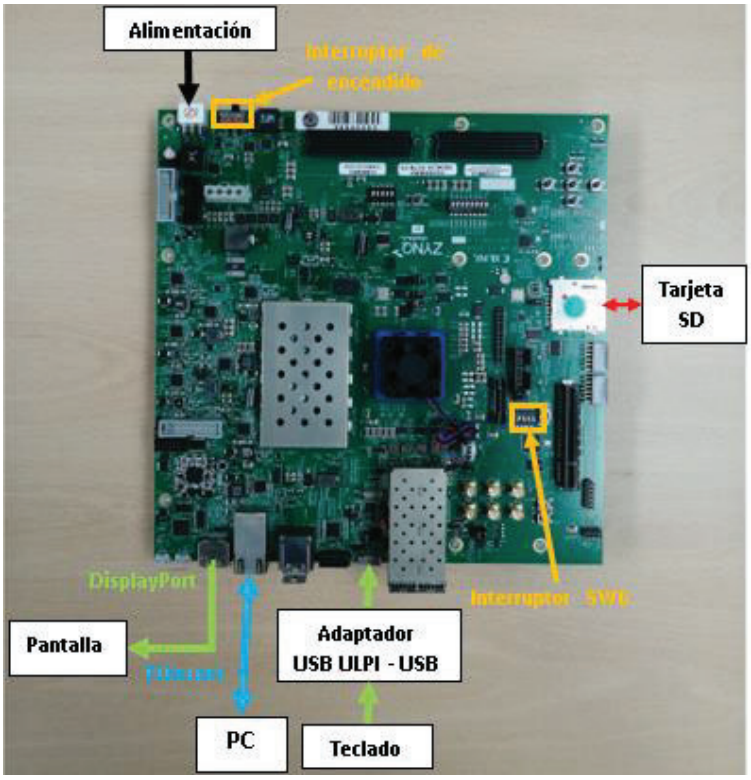


Figura 61: Diagrama de conexionado con la ZCU102

8.12. Conclusiones

En este capítulo se ha descrito la aplicación desarrollada, así como detallado diferentes aspectos claves para su comprensión. Entre las cuestiones tratadas, las más relevantes se corresponden a las opciones de configuración estática y dinámica que se encuentran previstas para el usuario. El objetivo de disponer de estas opciones se corresponde no solo a facilitar el ajuste idóneo de esta aplicación a las circunstancias de funcionamiento requeridas en el caso de uso de este trabajo (dimensiones de la imagen, placa de prototipado, etc.), sino que procurar que la aplicación resulte lo más fácil de adaptar a otras circunstancias de uso.

Con respecto al *host*, se han presentado las distintas funciones que constituyen su código fuente. Asimismo, se han descrito las operaciones efectuadas por el *host* para la captación de los datos de entrada y la sincronización de las operaciones y transferencias entre este y los *kernels* FPGA. Sobre la sincronización, se ha explicado la composición y secuencia con el que deben transferirse los datos desde el *host* destinados a los

aceleradores *hardware*; así como el formato y el orden en el que se obtendrían los datos generados por estos aceleradores.

Con respecto a los *kernels* FPGA, se han detallado los aspectos funcionales y de implementación *hardware* establecidos para los mismos en el código fuente. Asimismo, se han expuesto las opciones de configuración estática que presentan y la forma de regularla, incluyendo para ello una hoja de cálculo que facilita la determinación de la configuración estática óptima para cada *kernel* FPGA. Estos *kernels* se conectan con el *host* mediante un conexionado detallado en este capítulo en un diagrama de bloques del sistema.

Adicionalmente, se han detallado algunos patrones de nomenclatura que se han optado seguir para facilitar el entendimiento del código fuente de la aplicación. Además, se exponen algunos procedimientos y pautas de programación genéricas de las descripciones HLS cuya aplicabilidad es extensible a múltiples aceleradores *hardware* tanto de esta aplicación como de otras. Por último, se ha expuesto el procedimiento para ejecutar la aplicación en la placa de prototipado que utiliza el MPSoC empotrado.

Capítulo 9. Resultados de la implementación

9.1. Introducción

En este capítulo, se describen los resultados de rendimiento obtenidos por la aplicación, centrando la evaluación en los aceleradores *hardware* que incorpora. Este rendimiento depende de la configuración estática de la aplicación, así como del dimensionado, formato y serializado de las imágenes hiperespectrales a clusterizar. En la Tabla 9 se muestra la configuración estática configurada para este trabajo, asignando valores a las macros presentes en el código. Igualmente, en la Tabla 10 se presentan las características de las imágenes hiperespectrales utilizadas en la validación de la aplicación.

Tabla 9: Configuración estática de la aplicación

Parámetro	Valor
_INT_PART	16
_USER_MAX_PX	2500
_MAX_BANDS	116
_QUIT_LOWER_SPECTRA	4
_QUIT_UPPER_SPECTRA	5
_N_FILTERS	16
_AVERAGING	5
_KMEANS_PARALLEL	8
_ADDER_REG	7
_N_CLUSTERS	3
_N_COMPARATORS	3

Tabla 10: Características de las imágenes hiperespectrales para el trabajo

Parámetro	Valor	Parámetro	Valor
Largo	50 px	Ancho	50 px
Tamaño de dato	16 bits		
Bandas espectrales	125	*Interleave*	BSQ

Para la validación de los resultados, se comparan los resultados obtenidos desde un modelo funcional escrito en Matlab con los resultados de *clustering* generados durante la ejecución de la aplicación, ya sea en emulación *software* en el PC como en ejecución *hardware* en la placa de prototipado ZCU102. Se comparan los resultados analiza la adecuación de ambas al caso de uso, además de las semejanzas que presentan entre sí. También se evalúan los tiempos de ejecución medidos respecto a la aceleración *hardware* y a la aplicación en su conjunto.

Igualmente, se evalúa tanto la utilización de recursos PL requeridos para la implementación *hardware* de los módulos funcionales que componen los *kernels* FPGA, de dichos *kernels* y del binario ".xclbin" que los aglutina, como las prestaciones temporales de los *kernels* FPGA. Finalmente se presentan el consumo de potencia obtenido mediante estimación en el entorno Vivado.

9.2. Validación funcional

En consonancia con la información presentada en la referencia [2], las imágenes hiperespectrales de entrada a la aplicación tienen un dimensionado espacial de 50x50 píxeles. Con respecto a la dimensión espectral, la imagen posee 125 bandas espectrales. Las 4 primeras y las 5 últimas bandas espectrales no se consideran en el cómputo de la aplicación. Los datos del hipercubo se encuentran ordenados en la imagen siguiendo el criterio de ordenación BSQ. Dichos datos se representan en formato entero sin signo de 16 *bits*.

El contenido de las imágenes hiperespectrales de partida están disponibles en ficheros de datos de Matlab, con extensión ".mat". La entrada de datos re realiza usando una función en Matlab que lee la información desde los ficheros ".mat" y transforma el contenido de la imagen hiperespectral al formato esperado por la aplicación. Dicha función se denomina "crearImagen". Esta función utiliza las funciones "hypercube" y "enviwrite" para lograr el cambio de formato deseado.

La función "crearImagen" genera los archivos de las imágenes hiperespectrales de entrada y de las imágenes resultantes de las subetapas de preprocesamiento (filtrado y

normalizado). Estos últimos se utilizan para verificar el funcionamiento de las subetapas de preprocesamiento de la aplicación.

Adicionalmente, la función genera, para cada imagen de entrada, el mapa de segmentación o *clustering* resultante de aplicar el algoritmo "k-means" sobre la imagen preprocesada en Matlab. Para ello, la función "crearImagen" utiliza otra función desarrollada para este proyecto y denominada "escribeKmeans". En el Código 96 se expone la función "crearImagen", mientras que la función "escribeKmeans" se muestra en el Código 97.

```
function a=crearImagen(LowX, HighX, LowY, HighY, LowBand, HighBand)
wavelength = {450, 454, 458, 462, 466, 470, 474, 478, 482, 486, 490,
494, 498, 502, 506, 510, 514, 518, 522, 526, 530, 534, 538, 542,
546, 550, 554, 558, 562, 566, 570, 574, 578, 582, 586, 590, 594,
598, 602, 606, 610, 614, 618, 622, 626, 630, 634, 638, 642, 646,
650, 654, 658, 662, 666, 670, 674, 678, 682, 686, 690, 694, 698,
702, 706, 710, 714, 718, 722, 726, 730, 734, 738, 742, 746, 750,
754, 758, 762, 766, 770, 774, 778, 782, 786, 790, 794, 798, 802,
806, 810, 814, 818, 822, 826, 830, 834, 838, 842, 846, 850, 854,
858, 862, 866, 870, 874, 878, 882, 886, 890, 894, 898, 902, 906,
910, 914, 918, 922, 926, 930, 934, 938, 942, 946};

wavelength2 = wavelength(LowBand:HighBand);
wavelength = cell2mat(wavelength);
wavelength2 = cell2mat(wavelength2);

datatype = 'uint16';
normDatatype = 'uint32';

maxScaleVal = 0;
maxNormVal = 0;
switch datatype
    case 'uint8'
        maxScaleVal = 2^8 - 1;
        maxNormVal = 2^8;
    case 'uint16'
        maxScaleVal = 2^16 - 1;
        maxNormVal = 2^16;
    case 'uint32'
        maxScaleVal = 2^32 - 1;
        maxNormVal = 2^32;
end
nameDir = '/home/users/mguanche/ImHiperespectrales/dataCubes/';
nameFile = dir(strcat(nameDir,'*.mat'));

saveFolderName=strcat(nameDir, ['Aquicreados(', int2str(LowX), '_',
int2str(HighX), 'x', int2str(LowY), '_', int2str(HighY), 'x',
int2str(LowBand), '_', int2str(HighBand),')/']);
mkdir(saveFolderName);
```

```
%Iteracion por fichero .mat
for i = 1:size(nameFile, 1)

    %Load HS Cube
    imName = nameFile(i).name;
    nameLoad = strcat(nameDir, imName);
    load(nameLoad, 'calibratedHsCube');

    %Obtencion imagen hiperespectral en version sin recortar
    int_calibratedHsCube = calibratedHsCube;
    int_calibratedHsCube(isnan(int_calibratedHsCube))=0;
    int_calibratedHsCube = rescale(int_calibratedHsCube,0,
    maxScaleVal);
    intToSend_calibratedHsCube = uint16(int_calibratedHsCube);
    hcubeCal = hypercube(intToSend_calibratedHsCube, wavelength);

    %Transformado hipercubo sin recortar en formato .dat y .hdr
    saveFileName = strcat(saveFolderName, erase(imName,".mat"));
    enviwrite(hcubeCal, saveFileName, 'DataType', datatype,
    'ByteOrder', 'ieee-le');

    %Obtencion imagen hiperespectral en la version recortada en
    %largo, ancho y espectro
    int_HsCubeLight = int_calibratedHsCube(LowX:HighX, LowY:HighY,
    LowBand:HighBand);
    hcSize = size(int_HsCubeLight);
    intToSend_HsCubeLight = uint16(int_HsCubeLight);
    hcubeCal = hypercube(intToSend_HsCubeLight, wavelength2);

    %Transformado hipercubo en version recortada en formato .dat y
    %.hdr
    noCornersName = [saveFileName, '(', int2str(LowX), '_',
    int2str(HighX), 'x', int2str(LowY), '_', int2str(HighY), 'x',
    int2str(LowBand), '_', int2str(HighBand),')'];
    enviwrite(hcubeCal, noCornersName, 'DataType', datatype,
    'ByteOrder', 'ieee-le');

    %Filtrado de imagen hiperespectral en version recortada
    int_HsCubeLight = reshape(int_HsCubeLight, hcSize(1)*hcSize(2),
    hcSize(3));
    filteredSmooth = [];
    for  i= 1 : 1 : size(int_HsCubeLight, 1)
        filteredSmooth = [filteredSmooth, smooth(int_HsCubeLight(i,
        :)) ];
    end
    filteredSmooth = filteredSmooth';

    %Creacion hipercubo en version recortada y despues del filtrado
    reshap_filteredSmooth = reshape(filteredSmooth, hcSize(1),
    hcSize(2), hcSize(3));
    intToFile_filteredSmooth = uint16(reshap_filteredSmooth);
    hcubeFiltered=hypercube(intToFile_filteredSmooth, wavelength2);

    %Transformado hipercubo en version recortada y filtrada en
    %formato .dat y .hdr
    filteredName = [saveFileName, '(filtered)(', int2str(LowX), '_',
    int2str(HighX), 'x', int2str(LowY), '_', int2str(HighY), 'x',
    int2str(LowBand), '_', int2str(HighBand),')'];
```

```
    enviwrite(hcubeFiltered, filteredName, 'DataType', datatype,
    'ByteOrder', 'ieee-le');

    %Normalizado de imagen hiperespectral en version recortada
    preProcessedImage = mapminmax(filteredSmooth, 0, maxNormVal);
    toFile_preProcessedImage = reshape(preProcessedImage,
    hcSize(1), hcSize(2), hcSize(3));
    int_toFile_preProcessedImage=uint32(toFile_preProcessedImage);
    hcubePreprocessed = hypercube(int_toFile_preProcessedImage,
    wavelength2);

    %Transformado hipercubo en version normalizada en formato .dat
    % y .hdr
    preprocName = [saveFileName, '(preprocessed)(', int2str(LowX),
    '_', int2str(HighX), 'x', int2str(LowY), '_', int2str(HighY),
    'x', int2str(LowBand), '_', int2str(HighBand),')'];
    enviwrite(hcubePreprocessed, preprocName, 'Datatype',
    normDatatype, 'ByteOrder', 'ieee-le');

    %Creacion mapa de clusterizacion
    [idx,C] = kmeans(mapminmax(filteredSmooth, 0, 1), 3);
    imageCluster = reshape(idx, hcSize(1), hcSize(2), 1);
    kmeansName = [saveFileName, '(clustered)(', int2str(LowX), '_',
    int2str(HighX), 'x', int2str(LowY), '_', int2str(HighY), 'x',
    int2str(LowBand), '_', int2str(HighBand),')'];
    escribeKmeans(kmeansName, imageCluster);
end
end
```

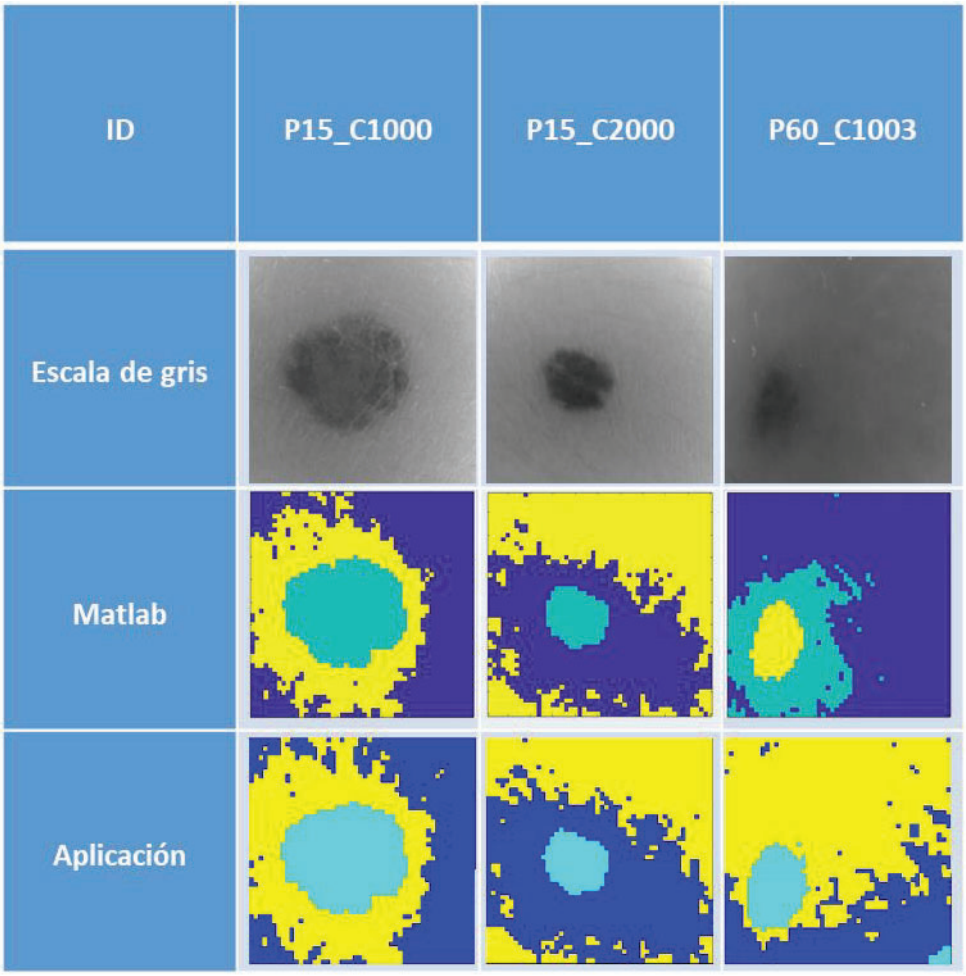
Código 96: Función "crearImagen"

```
function a = escribeKmeans(addr, imData)
    im_size = size(imData);
    wfid = fopen(strcat(addr, '.dat'),'w');
    if(wfid == -1)
        disp([('No se abre ') (addr) (' \n')]);
    else
        totalImage=0;
        for j=1:1:im_size(1)
            for i=1:1:im_size(2)
                success = fwrite(wfid, imData(j, i), 'uint16',
                'ieee-le');
                disp([('Escrito con exito ') int2str(success)
                (' dato')]);
                totalImage = totalImage+success;
            end
        end
        if(totalImage == (im_size(1)*im_size(2)))
            disp('Todos los pixeles se introducen');
        else
            disp([('Se introducen ') int2str(totalImage)
            (' falta ') int2str(im_size(1)*im_size(2)*im_size(3))])
        end
    end
    fclose(wfid);
end
```

Código 97: Función "escribeKmeans"

Tras evaluar el *clustering* efectuado por la aplicación para distintas imágenes de prueba, se observa que este identifica adecuadamente la región afectada por cáncer de piel. Además, existe un elevado grado de similitud entre los resultados de *clustering* obtenidos por la aplicación y los obtenidos mediante Matlab. Esta situación puede validarse visualmente en la Tabla 11. La fila "Matlab" identifica los resultados de *clustering* obtenidos en Matlab, mientras que la fila "Aplicación" identifica los obtenidos por la aplicación desarrollada. La identificación de cada imagen se especifica en la fila "ID" de la Tabla 11. Para facilitar la evaluación, cada imagen se representa en escala de grises en la fila "Escala de gris".

Tabla 11: Clustering en Matlab y de la aplicación



Como se observa en la Tabla 11, tanto la ejecución en Matlab como de la aplicación desarrollada identifican adecuadamente y de manera similar las regiones de la piel en

función de su estado de afección para las imágenes "P15_C1000" y "P15_C2000". Asimismo, las principales divergencias entre los *clusterings* de ambas ejecuciones se encuentran en la frontera entre las regiones ubicadas fuera de la lesión mostrada en la escala de grises. Estas diferencias resultan razonables si se considera que los píxeles en torno a estas fronteras suelen presentar firmas espectrales difíciles de clasificar, ya que poseen un grado de semejanza equiparables a los centroides de ambas regiones. Este suceso se acentúa debido a que estos centroides presentan una similitud entre sí mayor.

Con respecto a la imagen "P60_C1003", se observa una mayor diferencia entre el *clustering* de Matlab y el *clustering* obtenido. Esta divergencia resulta razonable teniendo en cuenta que la lesión en la imagen "P60_C1003" se encuentra muy difuminada. En consecuencia, se dificulta la convergencia en la regionalización de los *clústeres*, especialmente de aquellos distribuidos en regiones de la imagen ajenas a la zona donde se concentra la lesión.

Por otro lado, en el *clustering* realizado por la aplicación desarrollada para la imagen "P60_C1003", se contempla que la región de la esquina inferior derecha se asigna al mismo *clúster* al que se asocia la región donde la lesión cutánea resulta más notoria. No obstante, la diferencia entre ambas regiones que se visualiza en la correspondiente imagen en escala de grises evidencia que esta segmentación resulta incorrecta. Este suceso se justifica por la existencia de píxeles muertos debidos a un defecto de la cámara hiperespectral utilizada en la captura. Este error en la captura origina que la firma espectral de estos píxeles resulte más confusa.

La representación visual del *clustering* obtenido en Matlab se efectúa utilizando la función "imagesc" de Matlab. Esta función se invoca desde la función "kmeansImagen", la cual incluye la ejecución de un algoritmo similar al incluido en la aplicación para obtener el *clustering* correspondiente a la imagen hiperespectral. El código fuente de esta función se presenta en el Código 98.

```
function a = kmeansImagen(filename, LowX, HighX, LowY, HighY,
             LowBand, HighBand)
%HS Cube directory
nameDir = '/home/users/mguanche/ImHiperespectrales/dataCubes/';
compFileName = strcat(nameDir, strcat(filename, '.mat'));

%Labeled Data Directory
nameLabeledData = '/home/users/mguanche/MatlabCode_todo23April/
LabeledData/MeanData.mat';

%Number of cluster to compute k-means
numberOfClusters = 3;

%Color identification (label2rgb)
prismSkin = 22; %Green
prismBenign = 17; %Orange
prismUnknown = 14; %Blue
prismMalign = 19; %Red (color red used to identify lesion)

load(compFileName, 'calibratedHsCube');

%1) Preprocessing Chain
extremeBands = calibratedHsCube(LowX:HighX, LowY:HighY,
LowBand:HighBand);
hcSize = size(extremeBands);
extremeBands = reshape(extremeBands, hcSize(1)*hcSize(2),
hcSize(3));

%Filter smooth to remove noise
filteredSmooth = [];
for j = 1 : 1 : size( extremeBands, 1 )
    filteredSmooth = [filteredSmooth, smooth(extremeBands(j,:))];
end
filteredSmooth = filteredSmooth';

%Normalization
preProcessedImage = mapminmax(filteredSmooth, 0, 1);

clear calibratedHsCube filteredSmooth extremeBands

%2)K-means

%Compute K-Means
[idx,C] = kmeans(preProcessedImage, numberOfClusters);

%Create imageCluster using "k-means" results
imageCluster = reshape(idx, hcSize(1), hcSize(2), 1);
imagesc(imageCluster);

escribeKmeans('/home/users/mguanche/ImHiperespectrales/ImCluster/
P15_C1000(clustered)', imageCluster);
end
```

Código 98: Función "kmeansImagen"

La visualización de los resultados de *clustering* se efectúa al invocar en dicha aplicación la función "viewClustering". Adicionalmente, esta función guarda en un archivo

el *clustering* de la imagen obtenido por la aplicación. Esta función se describe en el archivo "ToKmeans.cpp", siendo su codificación la expuesta en el Código 99.

```
void viewClustering(FILE* resultsFile, hst_cluster_id* ctr_ids,
hst_dim_d y_length, hst_px_id nPx){
	hst_px_id px_idx=0;

	//system("cls");
	//Iteracion por fila
	do{
		//Iteracion por columna
		const hst_px_id Y_LIM = px_idx + y_length;
		do{
			switch(ctr_ids[px_idx]){
			case 0:
				//Azul
				printf("\033[1;34m");
				printf("\033[48;5;12m");
				break;
			case 1:
				//Amarillo
				printf("\033[01;36m");
				printf("\033[48;5;14m");
				break;
			case 2:
				//Cyan
				printf("\033[01;33m");
				printf("\033[48;5;11m");
			}
			printf("##");
			px_idx++;
		} while(px_idx < Y_LIM);

		printf("\033[0m\n");

	}while(px_idx < nPx);
	fwrite(ctr_ids, sizeof(hst_cluster_id), nPx, resultsFile);
}
```

Código 99: Función para la visualización del clustering de la aplicación desarrollada

Con respecto a los aspectos temporales de la ejecución, la aplicación ofrece los resultados expuestos en la Tabla 12. Estos resultados se extrajeron para el modo de ejecución de emulación *software*, en la fila "sw_emu"; y *hardware* o en el MPSoC, en la fila "hw". La penúltima fila, designada como "Mejora (hw/sw_emu)", ofrece el cociente entre el tiempo de ejecución en la emulación *software* y el de ejecución en la placa de prototipado. Esta fila permite determinar el grado de mejora de la velocidad computacional que la aceleración *hardware* implica. Las últimas filas, designadas como "NVIDIA RTX 2080 GPU" y "NVIDIA Tesla K40 GPU", indican los tiempos de ejecución obtenidos para 2 implementaciones de la aplicación desarrollada en [2]. Cabe destacar que la aplicación en

[2] es similar a la desarrollada en este TFM, pero implementada mediante GPUs. Además, dicha aplicación incluye algunas etapas de ejecución adicionales ("SAM" y "SVM") que suponen una carga computacional adicional. Por otra parte, las implementaciones GPUs fueron evaluadas utilizando imágenes hiperespectrales del mismo dimensionado que en este trabajo, pero para casos de cáncer de piel diferentes.

La mayor parte de estos tiempos medidos se corresponden al tiempo promedio que transcurre entre 2 iteraciones consecutivas de un mismo *kernel*. En dichas iteraciones, se tiene en cuenta no solo el tiempo de ejecución de los *kernels*, sino que también el requerido por el *host* en cada iteración del *kernel*, así como el paralelismo y sincronía entre el *host* y los *kernels*. Estos tiempos de iteración se indican en la Tabla 12 en las columnas identificadas como "(1 iter)". Por último, en la columna "Total" se indican los tiempos de ejecución de la aplicación, considerando la imagen hiperespectral "P15_C1000" como referencia y excluyendo la exposición del *clustering* obtenido.

Tabla 12: Tiempos de ejecución del algoritmo

	multFilter (1 iter) (µs)	normalizers (1 iter) (µs)	top_kmeans (1 iter) (µs)	Total (s)
sw_emu	**217190,00**	**220520,00**	**180500,00**	**48885,83**
Hw	**95,56**	**97,32**	**90,43**	**22,77**
Mejora (hw/sw_emu)	**2272,8**	**2265,9**	**1996,0**	**2146,9**
NVIDIA RTX 2080 GPU [2]				**0,8**
NVIDIA Tesla K40 GPU [2]				**0,33**

En la emulación *software*, se observa que los tiempos de ejecución de las iteraciones de los *kernels* "multFilter" y "normalizers" son ligeramente superiores a los del *kernel* "top_kmeans". La causa de ello se encuentra, entre otros aspectos, en que el paralelismo del procesamiento establecido para los *kernels* "multFilter" y "normalizers" es mayor al del *kernel* "top_kmeans". Este paralelismo se efectúa en la ejecución *hardware* de la aplicación. Sin embargo, carecen de efecto en la ejecución para emulación *software*, dado que en dicho modo los *kernels* se ejecutan de forma serial.

Tras evaluar los resultados de ejecución en placa de prototipado, se observa que la utilización de MPSoC con aceleración *hardware* mediante FPGA constituye una mejora muy sustancial del *throughput* en comparación con una implementación enteramente *software*. Esta mejora se encuentra en torno a un factor de 2000 para cada *kernel* y para la ejecución total de la aplicación. Esta mejora del *throughput* supera la obtenida en aplicaciones similares, como las desarrolladas en [14], [80], [81] y [82]. En dichos casos, el *throughput* mejoró en un factor aproximado de 188,77, 2,9, 86 y 28,34. A excepción de [14], las causas de estos sucesos pueden encontrarse en los siguientes aspectos.

- Frecuencia de reloj del *host* más significativamente superior que la de los *kernels* FPGA. En muchos de los casos, la ejecución *software* opera a una frecuencia de reloj superior a la implementada para el sistema de este TFM. Por contraparte, de frecuencia de reloj de la aceleración *hardware* en dichos casos no suele incrementarse en la misma proporción. Debido a ello, la lentitud relativa de los *kernels* con respecto a *host* es más significativa. Esto conlleva que el *host* permanezca ocioso durante mayor tiempo de ejecución. Con ello, la mejora del *throughput* pierde efectividad.
- Menor traspaso de procesamiento a los *kernels.* En algunos casos como en [80], no se implementa la totalidad del algoritmo “k-means” con aceleración *hardware*. Consecuentemente, una porción menos significativa de la aplicación resulta beneficiada de la mejora del *throughput* que dicha aceleración posibilita.

Aunque las características anteriores se encuentran en la mayoría de los casos, ninguna de ellas ocurre en el caso [14], el cual utiliza el mismo MPSoC con la misma frecuencia de reloj tanto para el *host* como para los *kernels*. Esto sucede ya que la aplicación de este TFM difiere del caso [14] en que, como se indicó en el apartado 8.5.2, los *kernels* FPGA siguen un funcionamiento en *streaming*. Esto posibilita que la ejecución de estos *kernels* en cada iteración se paralelice con la preparación por parte del *host* de la próxima trama de datos a procesar en la iteración siguiente. Con ello, se logra una mayor concurrencia *hardware-software* que incrementa la mejora del *throughput* gracias a la aceleración *hardware*.

Se observa que el tiempo de ejecución de las iteraciones del *kernel* "multFilter" es similar al del *kernel* "normalizers". Esto significa que ambos *kernels* FPGA aprovechan con la misma efectividad el grado de paralelismo que poseen desde la perspectiva del rendimiento temporal.

Al aplicarse en la ejecución *hardware* distintas técnicas de paralelismo para los *kernels* FPGA, las diferencias en los tiempos de ejecución de las iteraciones entre estos *kernels* se reducen. Sin embargo, las iteraciones del *kernel* "top_kmeans" siguen siendo un poco más rápidas. El motivo de ello se encuentra en que el *kernel* "top_kmeans" necesita generar datos de salida para su envío al *host* menos frecuentemente que los demás *kernels* FPGA.

9.3. Utilización de recursos

Tras seguir el procedimiento explicado para determinar la utilización de recursos *hardware* de los diferentes módulos funcionales, se han obtenidos los resultados expuestos en la Tabla 13 para el caso del *kernel* "multFilter". Además, en la Figura 62 se incluye una gráfica que permite visualizar el impacto relativo de los módulos funcionales dentro de la utilización de recursos PL de dicho *kernel*. Estos resultados, al igual que los de los módulos funcionales de los demás *kernels* FPGA, fueron visualizados mediante la herramienta Vitis Analyzer.

Tabla 13: Utilización de recursos de los módulos funcionales del kernel "multFilter"

	FFs	LUTs	DSPs	BRAMs
Filter::filterRun	2016	1507	0	0
maxOp	0	51	0	0
minOp	0	51	0	0
getScale	0	58	0	0
filterConvertInputUnsized	0	6	0	0
filterWriteInStr	1	17	0	0
filterRegToRegTransfer	0	6	0	0
filterConvertOutputUnsized	53	70	0	0
filterNonBlockingReadFromStr	1	24	0	0

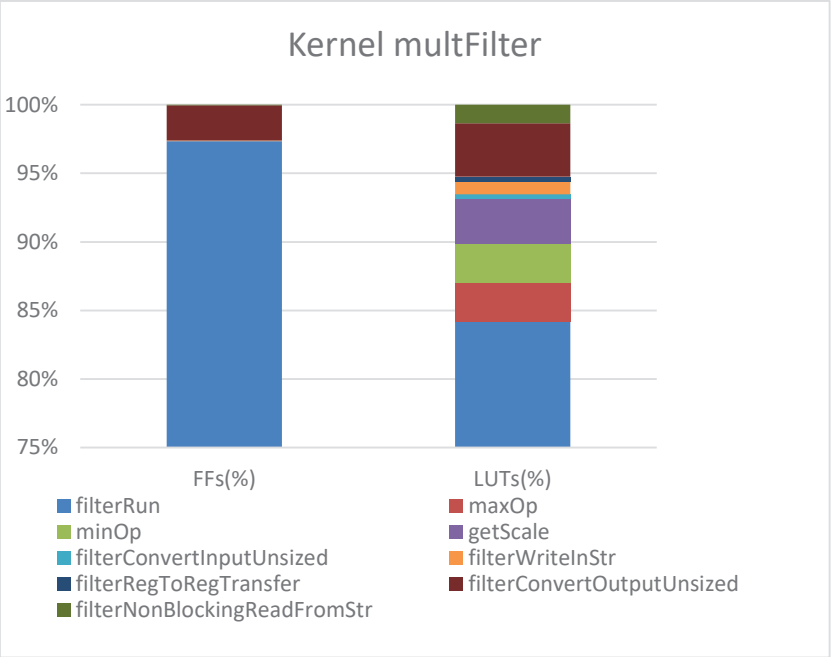


Figura 62: Utilización relativa de LUTs y FFs de los módulos funcionales de "multFilter"

Como se observa en la Tabla 13, la utilización de recursos de lógica programable del módulo funcional "Filter::filterRun" resulta superior al de los demás módulos del *kernel* "multFilter". Además, este módulo implica una ocupación muy mayoritaria de las *flip-flops* utilizadas por el *kernel*, como evidencia la Figura 62. Esto se debe principalmente a que dicho módulo presenta una funcionalidad más compleja que el resto, y engloba la mayor parte de la computación del *kernel*. En consecuencia, el módulo "Filter::filterRun" requiere una cantidad de registros mayor que los demás módulos del *kernel*.

Además, el módulo "Filter::filterRun" es el único módulo del *kernel* "multFilter" que recurre a la división aritmética. Esta operación aritmética resulta más compleja de implementar y, por lo tanto, requiere de una utilización de LUTs y *flip-flops* superior al de las demás operaciones que constituyen el resto del procesamiento que efectúa el *kernel*. Estas otras operaciones se limitan a la suma, resta, transferencia y comparación.

Para el caso del *kernel* "normalizers", la utilización de recursos PL de sus módulos funcionales se muestra en la Tabla 14 de forma absoluta, mientras que en la Figura 63 se comparan en términos relativos.

Tabla 14: Utilización de recursos de los módulos funcionales del kernel "normalizers"

	FFs	LUTs	DSPs	BRAMs
normalizer	2319	1789	0	0
normConvertInputUnsized	0	6	0	0
normWriteInStr	1	17	0	0
normNonBlockingReadFromStr	1	24	0	0
normConvertOutputSized	87	43	0	0
normRegToRegTransfer	0	6	0	0

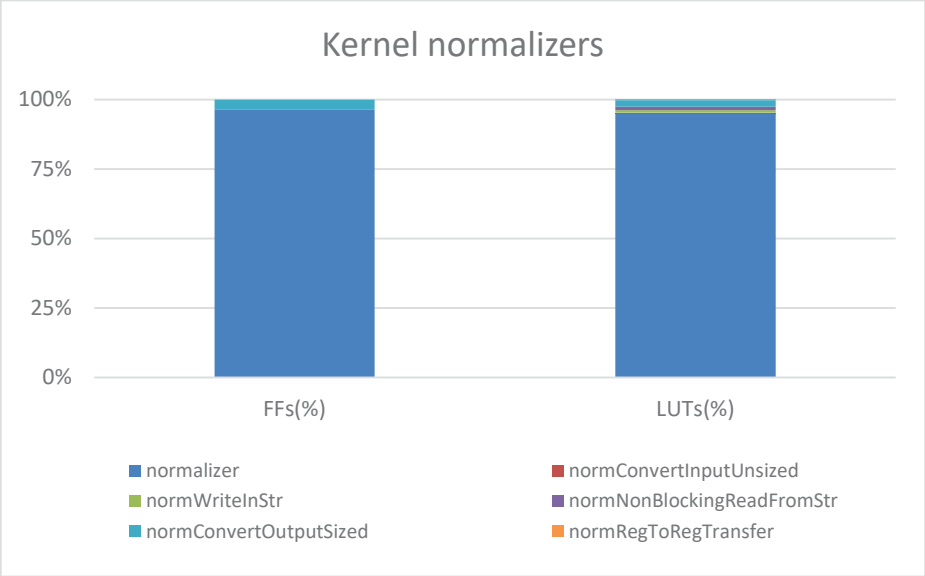


Figura 63: Utilización relativa de LUTs y FFs de los módulos funcionales de "normalizers"

Como se observa en la Figura 63, el *kernel* "normalizers" correspondiente a los módulos "normalizer" representa la parte principal de utilización de recursos del *kernel* ya que abarcan casi la totalidad de la computación del *kernel*. Además, estos módulos son los únicos módulos funcionales del *kernel* "normalizers" que incluye operaciones de división aritmética.

Si se compara con el módulo "Filter::filterRun", la utilización de recursos PL del módulo "normalizer" es un poco superior. Esto parece contradecir que la funcionalidad del módulo "normalizer" es menos compleja y extensa que la del módulo "Filter::filterRun". No obstante, debe tenerse en cuenta que la división en "normalizer" requiere de un divisor cuya longitud en *bits* es muy superior a la del divisor utilizado en el módulo "Filter::filterRun". En este sentido, el divisor en "normalizer" es de 16 *bits* para la configuración estática de la aplicación establecida, mientras que el del módulo

"Filter::filterRun" es de 3 *bits*. En consecuencia, la complejidad de la implementación *hardware* de la división del "normalizer" consume más recursos aun cuando existe una menor variedad de operaciones en este módulo.

En el caso del *kernel* "top_kmeans", la utilización de recursos PL de sus módulos funcionales se indica en la Tabla 15. Adicionalmente, la Figura 64 representa el grado de utilización relativo de estos módulos.

Tabla 15: Utilización de recursos de los módulos funcionales del kernel "top_kmeans"

	FFs	LUTs	DSPs	BRAMs
kmeansFromStrToDat	1	17	0	0
kmeansFromDatToStr	1	17	0	0
kmeansFromStrToUselessDat	1	17	0	0
Kmeans_Adder::operation	21	53	0	0
Min_ctr::run	0	87	0	0
coord_diff	68	87	0	0
setCtrDat	0	6	0	0
uptCtrCoord	3912	2944	0	0
uptRawCtrCoord	0	76	0	0
kmeansConvertInputSized	0	6	0	0
kmeansWriteInStr	1	17	0	0
kmeansNonBlockingReadFromStr	1	24	0	0
kmeansConvertOutputSized	87	43	0	0

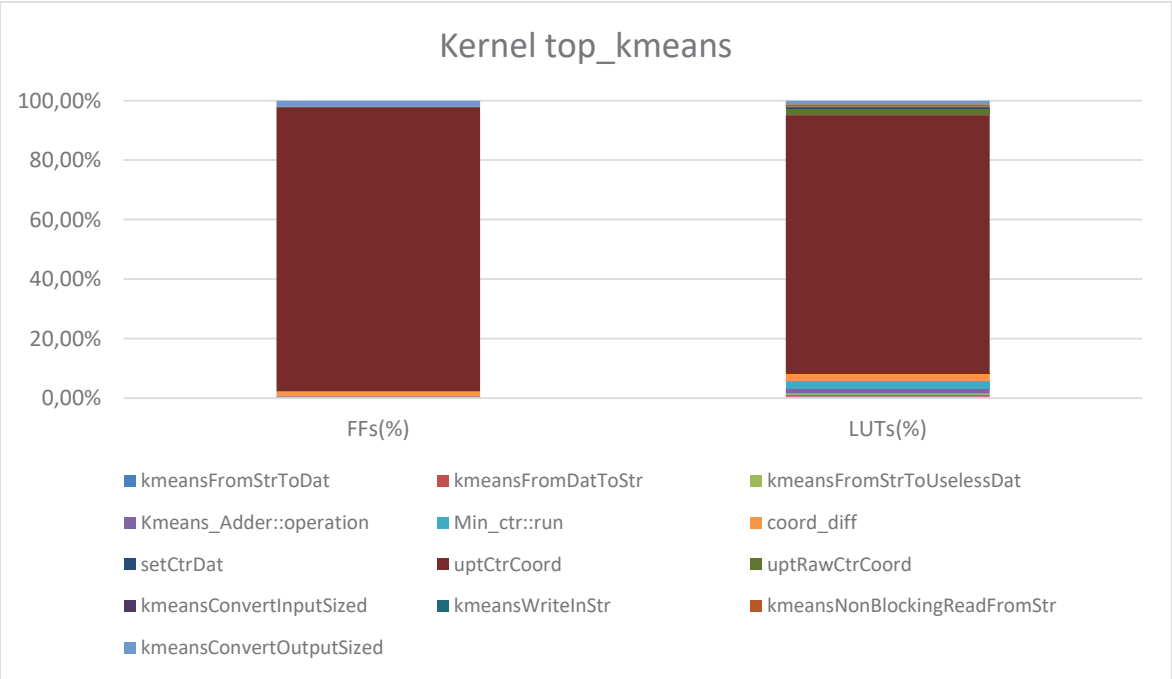


Figura 64: Utilización relativa de LUTs y FFs de los módulos funcionales de "top_kmeans"

Como se muestra en la Figura 64 y en la Tabla 15, la utilización mayoritaria de recursos PL dentro del *kernel* "top_kmeans" se corresponde al módulo "uptCtrCoord". Esto se debe a que dicho módulo es el único de "top_kmeans" que ejecuta división aritmética, mientras que las demás operaciones del *kernel* se limitan a ser suma, resta, multiplicación, transferencia, desplazamiento y comparación. También se contempla que la utilización de recursos PL del módulo "uptCtrCoord" es muy superior al encontrado en los módulos principales de los *kernels* FPGA anteriores. Esto se debe a que la funcionalidad de "uptCtrCoord" implica 2 divisiones por divisores de 12 *bits*, además de un dividendo de longitud mayor al de las divisiones de los demás módulos.

Con el apoyo de la hoja de cálculo descrita anteriormente, se ha optado por establecer un grado de paralelismo del procesamiento en los *kernels* de la etapa de preprocesamiento (*kernels* "multFilter" y "normalizers") de 16 datos (_N_FILTERS=16). Por otra parte, el *kernel* "top_kmeans" se ha configurado para un paralelismo de 8 datos (_KMEANS_PARALLEL=8). Con esta configuración, se obtiene que la utilización de recursos PL de estos *kernels*, así como los porcentajes de uso que implican respecto a la disponibilidad de la FPGA, son los expuestos en la Tabla 16. En esta tabla, la fila "Plataforma" se corresponde a la utilización de recursos FPGA que posee la configuración final de la FPGA, la cual incluye la funcionalidad de todos los *kernels* FPGA a utilizar. Los datos se obtuvieron como resultado de la compilación *hardware* de la aplicación y se visualizaron mediante Vitis Analyzer.

Tabla 16: Utilización de recursos de los kernels FPGA

	FFs	FFs (%)	LUTs	LUTs (%)	DSPs	DSP (%)	BRAMs	BRAMS (%)
multFilter	164298	29	28372	10	0	0	92	5
normalizers	45697	8	34758	12	0	0	90	4
top_kmeans	75114	13	71085	25	0	0	32	1
Plataforma	107970	20	105109	41	25	1	219	24

En los *kernels* de preprocesamiento (*kernels* "multFilter" y "normalizers"), se observa que la utilización de *flip-flops* del *kernel* "multFilter" es superior al del *kernel* "normalizers" para unas mismas exigencias de paralelismo. Esto se debe a que el funcionamiento del *kernel* "normalizers" es fuertemente combinacional.

Un comportamiento más combinacional implica que los datos generados por el *kernel* "normalizers", tras cada iteración, presentan una dependencia menos compleja de cuál sea la iteración y de los datos de entrada introducidos en iteraciones anteriores.

Por otra parte, el valor de los datos de salida del *kernel* "multFilter", e incluso si dichos datos se generan en una determinada ejecución, depende de la iteración concreta en ejecución, y es función de las dimensiones espaciales y espectral de la imagen a preprocesar. Además, el *kernel* "multFilter" precisa de realimentación en los módulos de determinación de los valores máximo y mínimo, así como de la sincronización del *reset* de estos con el fin de la evaluación de cada píxel. Por todo ello, el *kernel* "multFilter" precisa de una mayor cantidad de registros secuenciales en su funcionamiento interno que el *kernel* "normalizers", siendo las memorias *flip-flops* el tipo de recurso PL disponible para ello.

Por otro lado, la utilización de *flip-flops* por parte del *kernel* "top_kmeans" es bastante representativo. Entre los motivos que justifican esta situación se encuentran la necesidad de sincronizar los 4 modos de funcionamiento de este *kernel* FPGA, así como el acceso a la firma espectral de los centroides.

En este último aspecto, cabe recordar que los 4 modos de funcionamiento requieren acceso a los registros que almacenan las firmas espectrales de los centroides, correspondientes al *array* "loc_centroids" en la descripción HLS. Dichos accesos se efectúan además en instantes diferentes dentro de cada iteración, de un modo de ejecución al otro. Además, la transición entre los modos de funcionamiento y el acceso de cada uno de estos al *array* "loc_centroids" requieren de cierta capacidad dinámica. Dicho requisito se debe a la dependencia que presentan, en primer lugar, con respecto a la cantidad de bandas espectrales de la imagen, además de con respecto a la cantidad de píxeles y del contenido de la imagen hiperespectral filtrada.

En lo que respecta a la utilización de LUTs, se observa que es especialmente significativa para el *kernel* "top_kmeans". Entre las causas de ello se encuentra que este *kernel* constituye el único que precisa almacenar tres firmas espectrales en su totalidad. Dichas firmas se corresponden a los tres centroides a determinar por el algoritmo "k-means".

El modo de funcionamiento de actualización de los centroides precisa que, para cada banda espectral y para cada centroide, se incorpore un registro que almacene el sumatorio del valor del conjunto de píxeles asignados al centroide en la banda espectral considerada. Estos registros se corresponden al *array* "raw_loc_centroids" en el código fuente del *kernel*. Estas necesidades de almacenamiento recaen en la utilización de LUTs como memoria RAM distribuida. En consecuencia, se aumenta de forma significativa la utilización de LUTs por parte del *kernel* "top_kmeans" para un mismo grado de procesamiento en paralelo. Este aumento es significativo cuanto mayor sea el dimensionado espectral máximo de la imagen hiperespectral que la aplicación debe poder evaluar sin necesidad de recompilación.

Otro aspecto característico que se observa es la utilización de bloques de memoria BRAM en los *kernels* FPGA, pero no en sus módulos funcionales. Las memorias BRAM se utilizan en las interfaces de entrada y salida del *kernel*. La cantidad de bloques BRAM que consume cada *kernel* depende del tamaño de los datos de entrada y salida de los *kernels* y de la cantidad de puertos físicos que estos poseen. Este último aspecto se regula mediante la utilización del parámetro "bundle" de la directiva **#pragma HLS interface**. El número de puertos se ha minimizado para cada *kernel* sin comprometer las posibilidades de *pipelining* que las arquitecturas *hardware* internas de los *kernels* FPGA posibilitan.

Con respecto a la utilización de recursos FPGA que precisa la implementación de todos los *kernels* FPGA (Tabla 16 fila "Plataforma"), debe considerarse que, para un mismo grado de optimización en la implementación *hardware*, la utilización de recursos FPGA de los *kernels* en su conjunto es algo inferior al del binario que los agrupa. Esto ocurre ya que el binario debe contener, no solo los *kernels* FPGA, sino también la lógica programable implicada en la comunicación de estos *kernels* con el *host*.

Esta situación ocurre en la utilización de los recursos DSPs y BRAMs. Sin embargo, para el caso de las LUTs y *flip-flops* sucede lo contrario. Esto se debe a que las estimaciones de utilización de recursos de los *kernels* FPGA individualmente se efectúan para una optimización de dicha utilización inferior a la aplicada en la generación del binario ".xclbin". Para este último caso, se realizan optimizaciones adicionales consistentes en la utilización

de pocos recursos *hardware* más complejos que reemplacen una implementación funcionalmente equivalente basada en una mayor cantidad de recursos más simples.

Para el cómputo, se dispone de los DSPs como recurso *hardware* complejo. Dicho recurso sustituye la implementación mediante LUTs y *flip-flops* de operaciones complejas, tales como la multiplicación o la división.

Para el almacenamiento de datos internos, los BRAMs constituyen recursos complejos que pueden sustituir la utilización de LUTs como memoria distribuida. Para el caso de *arrays* grandes que no requieran un particionado excesivo, este cambio incrementa la utilización BRAMs, pero reducen en mayor medida la utilización de LUTs, puesto que estos últimos poseen una capacidad de almacenamiento inferior.

En la Figura 65 se visualiza la distribución de la utilización de recursos FPGA de los *kernels* identificada mediante un sistema de colores y obtenida en Vivado. Además, en dicha figura se observa la utilización de recursos de los módulos contemplados en el diagrama de bloques del apartado 8.9. exceptuando el PS y los *kernels* FPGA. Cabe recordar que estos módulos se corresponden fundamentalmente a la interconexión PS-*kernel* y al *reset* de los *kernels*.

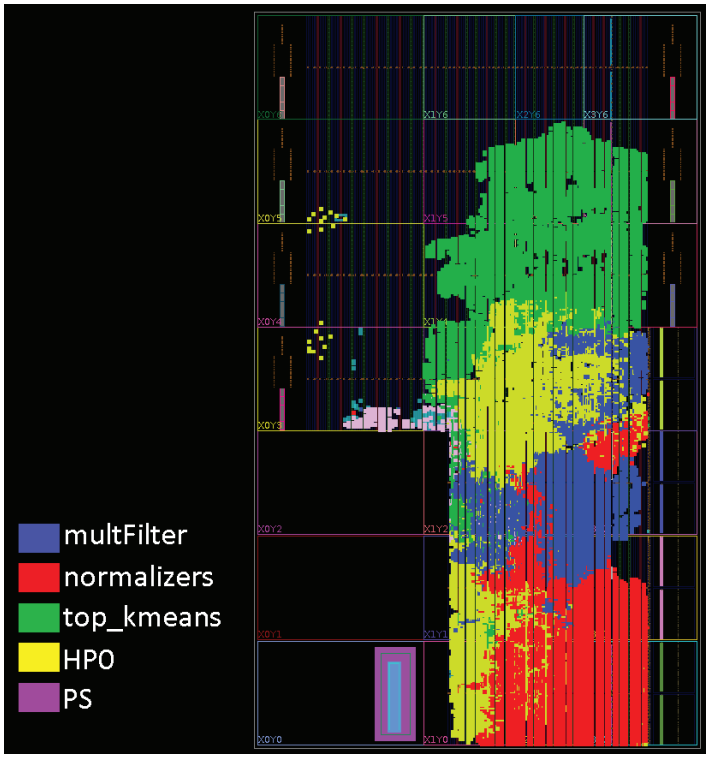


Figura 65: Distribución de la utilización de recursos

En la Figura 65 se observa que los módulos de interconexión entre el PS y los *kernels* FPGA requieren una utilización del PL no desdeñable. Sin embargo, esta utilización se corresponde en su práctica totalidad al módulo "axi_ic_ps_e_S_AXI_HP0_FPD", identificado con coloración amarilla. Como se explicó en el apartado 8.9. , este módulo constituye el *switch* que conmuta el puerto "S_AXI_HP0_FPD" del PS con las interfaces "m_axi" de los *kernels* FPGA.

En la Tabla 17 puede verse cuantitativamente la utilización mostrada en la Figura 65. Estos resultados se obtuvieron mediante Vivado pues, a diferencia de Vitis Analyzer, la primera herramienta permite obtener la utilización de recursos sobre la configuración final de la FPGA especificada mediante el binario ".xclbin". En dicha tabla, la utilización de LUTs se desglosa en las columnas "LUTs as Logic" y "LUTs as Memory". La primera columna se refiere a la utilización de LUTs destinada al cómputo de datos, mientras que la segunda identifica la cantidad de LUTs empleadas como memoria. Además, la fila "Otros" representa la utilización de lógica programable que implican los módulos distintos de los *kernels* FPGA y del módulo "axi_ic_ps_e_S_AXI_HP0_FPD".

Tabla 17: Utilización de recursos dentro del bitstream

	FFs	LUTs	LUTs as Logic	LUTs as Memory	DSPs	BRAMs
multFilter	22342	16887	14540	2347	0	16,5
normalizers	32304	26294	24256	2038	0	12,5
top_kmeans	29391	30271	28689	1582	25	53,0
HP0	21986	22907	21263	1644	0	136,5
Otros	1947	1148	1076	72	0	0,0
Plataforma	107970	97507	89824	7683	25	218,5

9.4. Prestaciones temporales de los *kernels*

Con respecto a la frecuencia de funcionamiento, se ha conseguido que todos los *kernels* FPGA desarrollados en este diseño sean capaces de operar holgadamente con una frecuencia de reloj de 150 MHz, garantizando la fiabilidad de los datos devueltos. Asimismo, las prestaciones temporales logradas para los *kernels* FPGA según la compilación *hardware* son las expuestas en la Tabla 18. Cabe destacar que "Int. Iniciación (ciclos de reloj)" identifica el intervalo de iniciación en el *pipelining* de los *kernels* FPGA. Los valores

de la columna “Latencia (s)” se obtienen considerando una frecuencia de reloj de 150 MHz. La visualización de estos datos se efectuó mediante Vitis Analyzer.

Tabla 18: Prestaciones temporales de los kernels FPGA

	Intervalo de Iniciación (ciclos de reloj)	Latencia (ciclos de reloj)	Latencia (microsegundos)
multFilter	2	239	1,59
normalizers	1	250	1,67
top_kmeans	40	151	1,01

Teniendo en cuenta el intervalo de iniciación, los *kernels* FPGA poseerán una doble capacidad de paralelismo. La primera de dichas capacidades se debe al paralelismo de la arquitectura *hardware*. Este consiste en la capacidad del *kernel* FPGA de procesar simultáneamente varios datos, desde un mismo instante de inicio hasta un mismo instante de fin para cada dato. Esto se logra mediante la replicación dentro del *kernel* de los módulos de procesamiento. La otra capacidad de paralelismo se debe al *pipelining*. Con esto último, varias tramas, formadas cada una por datos procesados simultáneamente, pueden solapar su procesamiento, aunque comenzando y terminando en instantes de tiempo diferentes.

Esta doble capacidad de paralelismo se presenta en la Figura 66, donde “N_DATA” representa la cantidad de datos procesables simultáneamente por parte del *kernel* e “II” el intervalo de iniciación.

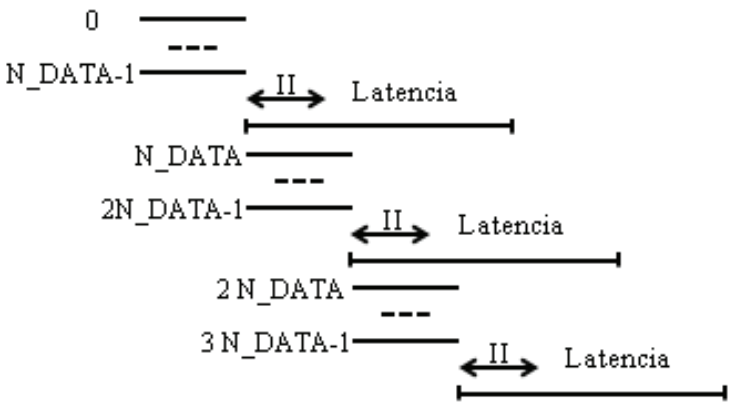


Figura 66: Capacidades de paralelismo de los kernels FPGA

Como se observa en la Tabla 18, se ha conseguido el *pipelining* máximo en el *kernel* “normalizers”, puesto que el intervalo de iniciación es de 1 ciclo de reloj. Esto es posible debido a que los datos siguen una progresión extremadamente lineal en su procesamiento,

sin realimentaciones ni bifurcaciones en la transferencia de los datos hacia las distintas operaciones del *kernel* "normalizers".

Con respecto al *pipelining* del *kernel* "multFilter", se ha logrado un alto grado de eficiencia, al ser el intervalo de iniciación del *kernel* de 2 ciclos de reloj. Este último *pipelining* no pudo optimizarse a un intervalo de iniciación de 1 ciclo de reloj debido a las lecturas múltiples dentro del *kernel* FPGA que precisaban los datos filtrados para una misma iteración del *kernel* de filtrado. En este sentido, cabe recordar que cada dato generado por algún módulo de filtrado del *kernel* "multFilter" debe traspasarse en la misma ejecución hacia la salida y hacia los módulos de determinación de los valores máximo y mínimo que correspondan. Por otra parte, los valores mínimos y máximos de los píxeles evaluados deben transferirse en cada iteración del *kernel* y tras su generación a la realimentación del módulo que los genera y al módulo de cálculo de la escala que se les asigne. Además, en el caso de que en dicha iteración se termine de filtrar algún conjunto de píxeles, los valores mínimos deberán transferirse también hacia la salida en la misma iteración que los genera.

El *kernel* "top_kmeans", no presenta un *pipelining* significativo en comparación a los demás *kernels*. Esto se observa en que el *kernel* "top_kmeans" posee un intervalo de iniciación muy superior al de los otros *kernels*. Dicha situación se debe a que antes de iniciar la evaluación del centroide más cercano a un píxel, es necesario que se termine la actualización de los centroides previa para determinar la firma espectral de los centroides a considerar. Por otra parte, el inicio de la actualización de los centroides requiere que se finalice la etapa previa de cálculo del centroide más cercano para asegurar cuál es el nuevo centroide en el que reubicar el píxel evaluado. Esta interdependencia entre el cálculo del centroide más cercano y la actualización de los centroides imposibilita que pueda establecerse un menor intervalo de iniciación y un mayor grado de *pipelining* en la ejecución del *kernel* "top_kmeans".

Con respecto a la latencia, se ha observado mediante Vitis HLS que, para todos los *kernels*, las operaciones que consumen más ciclos de reloj con, gran margen de diferencia, se corresponden a las operaciones AXI "readreq" (Figura 67) y "writeresp" (Figura 68). En

menor medida, también presentan un impacto significativo las operaciones de división aritmética “udiv” (Figura 69).

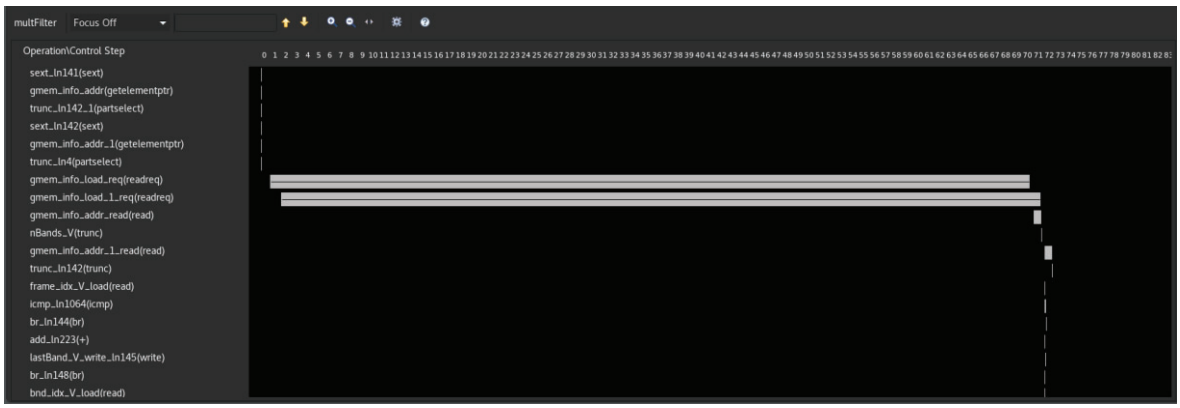


Figura 67: Impacto de la operación “readreq”

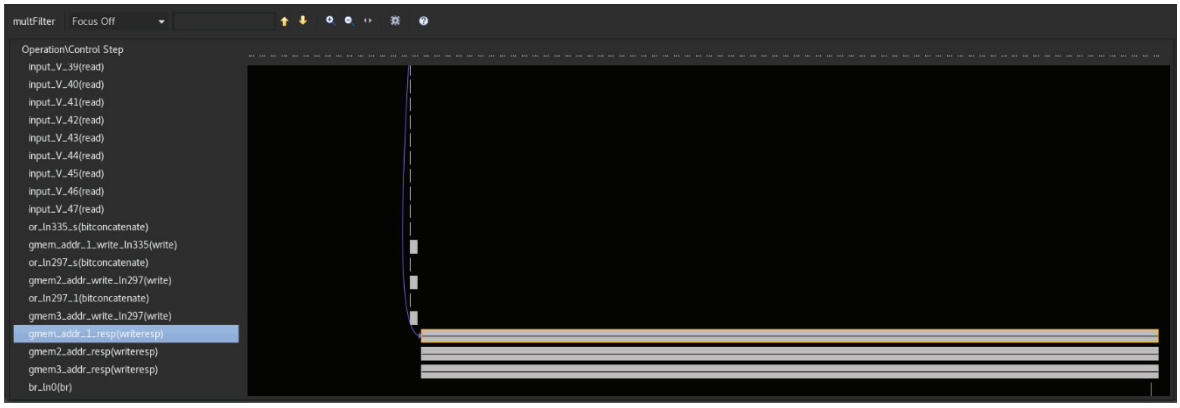


Figura 68: Impacto de la operación “writeresp”

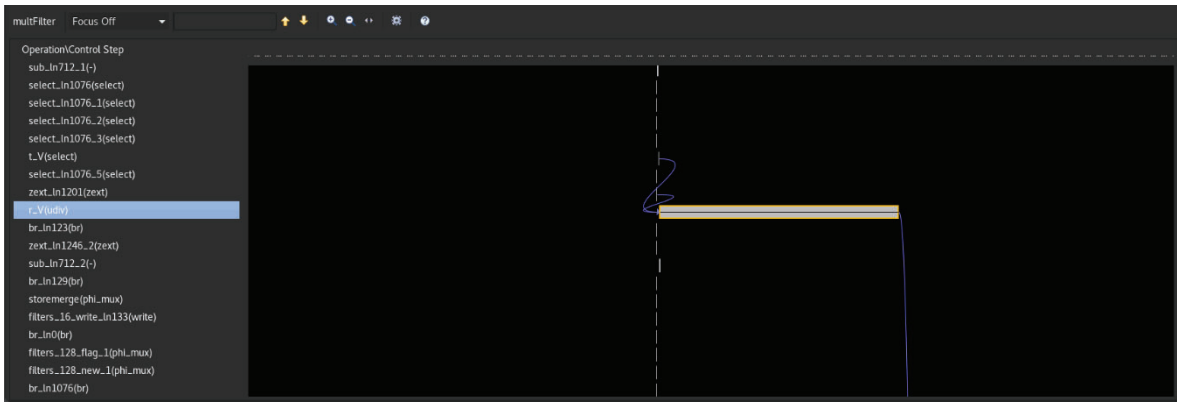


Figura 69: Impacto de la operación “udiv”

Para obtener el diagrama de *timeline* de la aplicación en su ejecución en placa se recurrió a la utilización de funcionalidades XRT ordenadas mediante un archivo “xrt.ini” adjunto al ejecutable del *host*. Para el caso de los *kernels* “multFilter” y “normalizers”, sus diagramas se corresponden a los de la Figura 70 y la Figura 71 respectivamente.

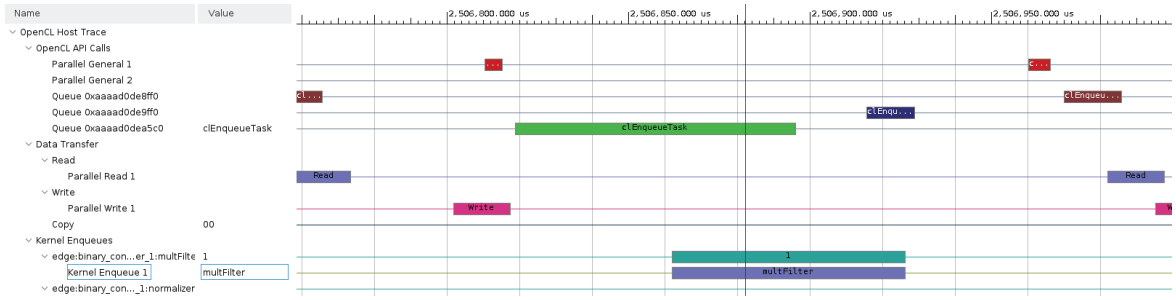


Figura 70: Timeline del kernel "multFilter"

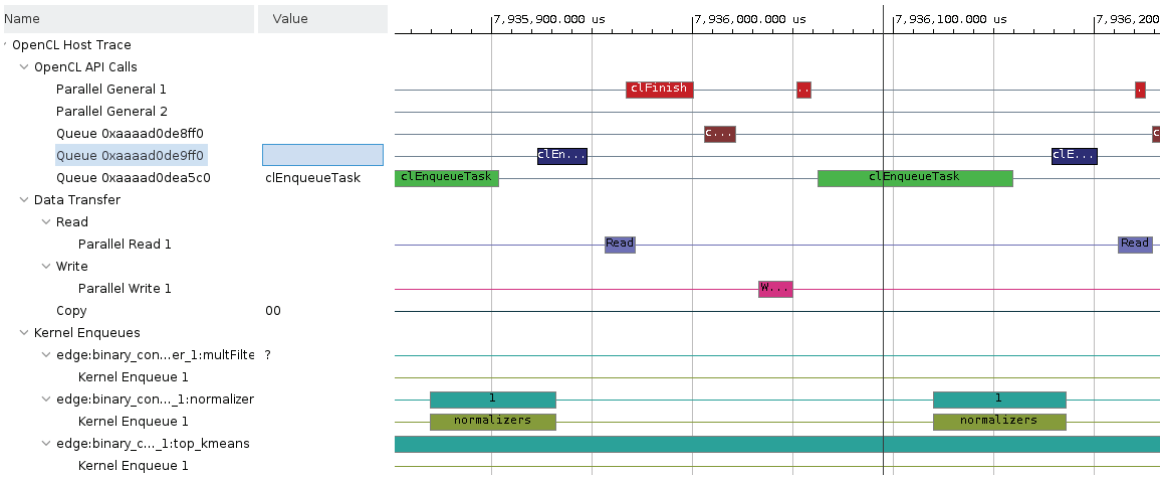


Figura 71: Timeline del kernel "normalizers"

En la Figura 72 se representa el *timeline* del *kernel* "top_kmeans" para aquellas iteraciones que no generan datos de salida. En dicha figura, se muestran 2 iteraciones, donde la primera viene antecedida de 2 transferencias *host-kernel* ("Write") y la siguiente de una sola transferencia. La transferencia común a ambas se corresponde a la trama de bandas espectrales del píxel a evaluar, mientras que la transferencia adicional de la primera iteración es para especificar el centroide previamente asignado al píxel.

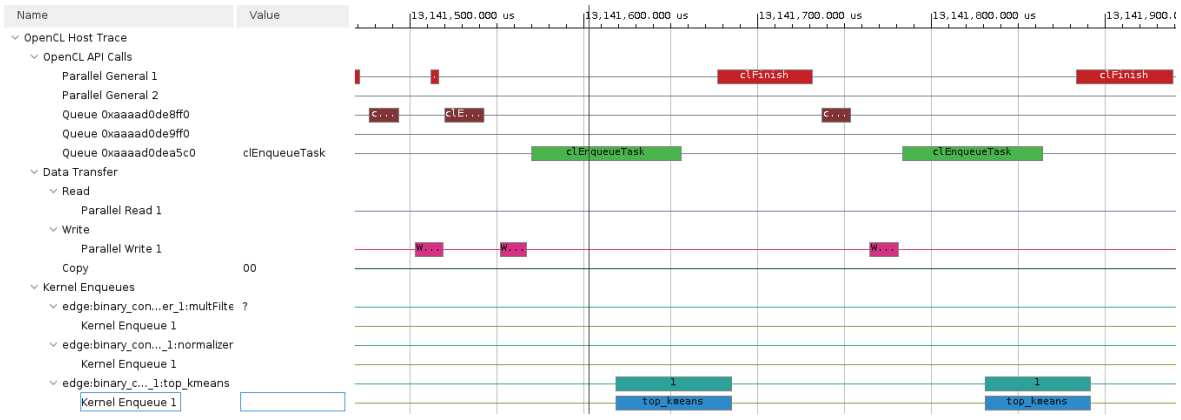


Figura 72: Timeline del kernel "top_kmeans" en etapa sin generación de salida

Para el caso de las iteraciones de “top_kmeans” que generan salida de datos, como el caso de aquellas pertenecientes al modo de funcionamiento de transferencia de los centroides hacia el *host,* el *timeline* se ajusta al mostrado en la Figura 73.

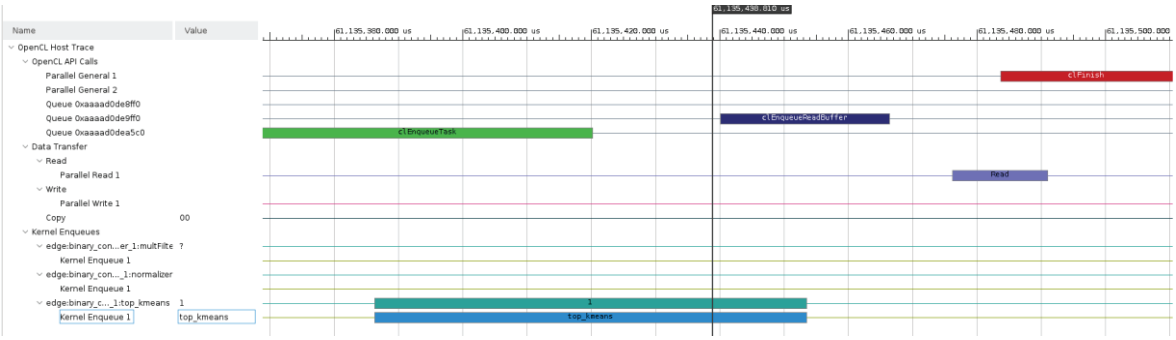


Figura 73: : Timeline del kernel “top_kmeans” en etapa con generación de salida

9.5. Consumo de potencia

Para evaluar la potencia consumida por la ejecución de la aplicación, puede recurrirse a las estimaciones de potencia obtenidas durante la implementación *hardware*. Este proceso, así como las estimaciones, son efectuadas por la herramienta Vivado. Dicha herramienta genera varios informes, y pueden consultarse abriendo desde la interfaz gráfica de Vivado el archivo “prj.xpr” generado tras la compilación *hardware* del proyecto de Vitis de la aplicación. El consumo de potencia es de aproximadamente 6,16 W. Dicho consumo se distribuye de la manera mostrada en la Figura 74.

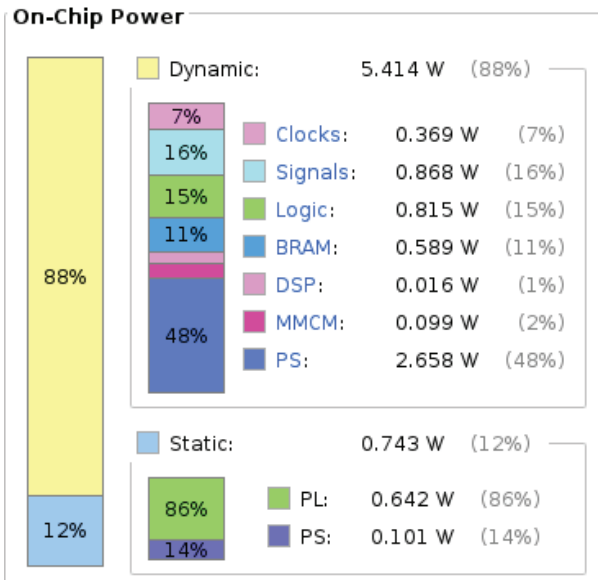


Figura 74: Informe sobre el consumo de potencia de la aplicación

Al analizar el consumo de potencia, hay que tener en cuenta la terminología utilizada por Xilinx tal como se explica en [83]. En este sentido, se considera potencia estática a la potencia suministrada cuando el dispositivo está configurado con su diseño y no se aplica ninguna actividad externamente o se genere internamente. Por el contrario, se conoce como potencia a aquella requerida cuando el dispositivo está ejecutando la aplicación y experimentando actividad de conmutación a medida que los relojes y las rutas de datos alternan entre valores lógicos.

Como es de esperar, se observa que el porcentaje mayoritario del mismo se encuentra en la potencia dinámica, constituyendo el 88% del consumo de potencia total, y se debe al procesamiento y transferencia de los datos. Este hecho resulta razonable teniendo en cuenta la cantidad significativa de instrucciones que requiere la ejecución del sistema. Entre estas instrucciones deben considerarse las implicadas en el procesamiento realizado por los *kernels* FPGA, las operaciones efectuadas por el *host* para la transferencia de datos.

Con respecto al consumo de potencia estática, este presenta una parte minoritaria, siendo el 12 % de la potencia total consumida por la aplicación.

La potencia dinámica que precisa la ejecución del *host* puede identificarse con la denominada como “PS” dentro del informe de potencia que se expuso en la Figura 74. Dicho consumo representa el 48% de la potencia dinámica total que implica la ejecución de la aplicación. Por otra parte, la potencia dinámica que consume la ejecución de los *kernels* en la FPGA puede aproximarse como la suma de las potencias designadas como “Signals”, “Logic”, “BRAM” y “DSP” dentro de ese mismo informe. Dicha suma supone el 43 % de la potencia dinámica total. Es destacable el consumo de potencia de los bloques de memoria “BRAM” representando del orden del 11%. En el caso de los bloques DSP el consumo es más reducido.

Por otra parte, “Logic” se refiere al consumo de potencia debido a la utilización de LUTs de la FPGA, el cual asciende al 15% de la potencia dinámica total de la aplicación. La potencia catalogada como “Signals” indica la porción de la potencia dinámica consumida en la gestión de las transferencias entre el *host* y los *kernels* FPGA [84], la cual engloba el 16 %.

Con estos resultados, se determina que la implementación de la aceleración *hardware* mediante FPGA en esta aplicación supone, no solo una mejora muy significativa del *throughput*, sino que también un ahorro de consumo de potencia. Dicho ahorro se debe, entre otras causas, a que la aceleración *hardware* reduce el tiempo de ejecución de la aplicación. Esto ocurre incluso aunque para implementar dicha aceleración *hardware* se reduzca la frecuencia de reloj a la que opera la aplicación, repercutiendo esta última reducción en un ahorro energético aún mayor. El traspaso de parte de la carga computacional de la aplicación a la FPGA ocasiona que el sistema opere con un consumo de potencia menor.

9.6. Conclusiones

En este capítulo, se ha analizado el rendimiento temporal y la utilización de recursos de lógica programable que implica cada *kernel* FPGA de la aplicación y el binario que los agrupa. En el aspecto temporal, se ha minimizado el intervalo de iniciación entre las ejecuciones de los *kernels* tanto como las dependencias de datos en las descripciones funcionales de estos han posibilitado. La utilización de recursos PL también se ha evaluado para los principales módulos funcionales que componen cada *kernel*, y cuyos grados de replicación en la arquitectura *hardware* dependen del grado de paralelismo del procesamiento establecido.

Por otra parte, se ha valorado el *clustering* efectuado por la aplicación, el cual se ha observado que delimita adecuadamente las regiones de la piel captada en una imagen hiperespectral según el estado de afección de cáncer de piel que presentan. Con respecto al rendimiento temporal de la aplicación, este ha sido evaluado para la totalidad de la aplicación, exceptuando la visualización de los resultados finales, y para una iteración promedio de cada *kernel* FPGA. Asimismo, se han considerado los modos de ejecución de emulación *software*, y ejecución *hardware*. Se ha observado que la aceleración *hardware* mediante FPGA mejora significativamente la velocidad de cómputo de la aplicación. También se ha analizado el consumo de potencia que conlleva la ejecución de la aplicación obteniendo que la aceleración *hardware* implementada reduce dicho consumo.

Capítulo 10. Conclusiones y trabajos futuros

10.1. Conclusiones

A lo largo de este trabajo se ha desarrollado una aplicación que implementa aceleración *hardware* basado en MPSoC y se ha verificado su funcionamiento. Dicha aplicación combina la capacidad informativa de las imágenes hiperespectrales con la capacidad analítica de los algoritmos de *machine learning*, implementándose en este caso el algoritmo "k-means".

El desarrollo de la aplicación ha requerido un conocimiento transversal. Ha sido preciso estudiar conceptos relativos a las imágenes hiperespectrales, *machine learning* y sistemas MPSoC. También ha requerido la utilización de distintos lenguajes de programación. Tanto el *host* como los *kernels* FPGA se programaron en C/C++, si bien estos últimos se programaron siguiendo el enfoque de programación HLS, más próximos a conceptos de diseño *hardware*. Asimismo, las significativas diferencias a nivel de *hardware* que presenta la plataforma, tanto en lo que se refiere al *host* (microprocesador) y a los *kernels* (FPGA) ha ocasionado que el diseño de los *kernels* presente implicaciones y restricciones que divergen significativamente de las encontradas en la codificación del *host*. Además, el diseño HLS de los *kernels* FPGA ha implicado la utilización de directivas **#pragma HLS**, que han permitido dirigir distintos aspectos relacionados con la implementación en FPGA de las descripciones funcionales expresadas en C/C++ sin información estructural ni temporal. Adicionalmente, se recurrió a la programación mediante Matlab para la generación de los ficheros de verificación de la aplicación.

Otro recurso de programación estudiado y aplicado en este trabajo son las APIs de OpenCL. Permiten establecer la sincronización entre las operaciones del *host* y las de los *kernels* FPGA, así como la transferencia de datos entre estos componentes de la aplicación. Además, la utilización de los objetos **cl::Event** y la asignación a estos de funciones *callback* ha permitido establecer las métricas temporales en la ejecución de los *kernels* FPGA.

Con este desarrollo se ha logrado establecer una aplicación que es capaz de efectuar el filtrado, normalizado y *clustering* de imágenes hiperespectrales para la detección de cáncer de piel aprovechando las distintas posibilidades de paralelismo que ofrecen las FPGA para la aceleración *hardware*. Para abordar este diseño, se ha optado por desarrollar y verificar, en primer lugar, la programación del *host* previa a la ejecución del primer *kernel* FPGA. Esta parte del diseño incluye la determinación de los parámetros de la aplicación introducidos mediante línea de comandos, la inicialización de los objetos de OpenCL que se reutilizarán en los diferentes *kernels* FPGA de la aplicación y aquellos que se utilizarán únicamente en el *kernel* de filtrado.

A continuación, se ha desarrollado la arquitectura *hardware* del *kernel* de filtrado. A partir de su diseño, se define el esquema de transferencia de datos de la imagen hiperespectral a dicho *kernel*. A ello le sigue el desarrollo y la verificación de la codificación descrita en el archivo "FromFileToFPGA.cpp". Asimismo, se valida la flexibilidad y versatilidad deseadas para esta aplicación. Entre ellas se pueden citar el tamaño en *bits* de los datos de la imagen hiperespectral, la adaptación dinámica a los formatos de ordenación estándar de los datos del hipercubo, la eliminación asimétrica de bandas espectrales extremas, etc.

Los *kernels* de filtrado, normalizado, *clustering* y SAM de la aplicación mediante igualmente se han desarrollado siguiendo una metodología HLS. Para la verificación de estos *kernels*, la primera etapa consiste en lograr su correcto funcionamiento para emulación *software*. Con este proceso, se asegura la coherencia de la descripción algorítmica. Posteriormente, se compilan estos *kernels* para emulación *hardware*, asegurando con ello su implementabilidad en FPGA y sus prestaciones tanto temporales como de utilización de recursos de la FPGA. Este proceso se efectuó iterativamente, depurando con ello los errores y *warnings* encontrados. Al mismo tiempo, se evalúan

diferentes opciones de codificación, procurando tanto la optimización como la adaptabilidad estática y dinámica de los *kernels*.

La siguiente etapa abordada es la programación del *host,* incluyendo la sincronización y las transferencias de datos entre el *host* y el *kernel* de filtrado. Esta etapa concluye con la verificación de los resultados devueltos por el *kernel* de filtrado.

Una vez que todos los componentes están disponibles se ejecuta la aplicación en modo depuración en emulación *software*. De esta manera, se detectan y corrigen errores de integración tanto del *host* como del *kernel* de filtrado.

Por último, se ejecuta la aplicación en emulación *hardware*, consiguiendo con ello culminar la detección y depuración de errores de funcionamiento tanto de los *kernels,* como de las transferencias de datos. Este proceso se reiteró para cada *kernel*, introduciéndose cada uno de estos de manera progresiva en la aplicación.

Una vez validado el funcionamiento del SoC, se instrumenta el sistema con la funcionalidad necesaria para medir los tiempos de ejecución de los procesos. Para ello, se recurre fundamentalmente a la asignación de *callbacks* y eventos a las operaciones de ejecución de los *kernels* y a las transferencias entre estos *kernels* y el *host*. Estos aspectos se configuran mediante objetos **cl::Event** de la manera definida por los estándares de OpenCL.

Tras esta verificación, se compila la aplicación para su ejecución en la placa de prototipado, obteniendo con ello los resultados y los tiempos de ejecución reales de la aplicación. La introducción y arranque de la aplicación en la placa se efectuó mediante tarjeta SD.

Otro aspecto tratado en esta aplicación ha sido procurar que la codificación, tanto del *host* como del *kernel* resultasen lo más fácilmente adaptables a otras circunstancias de funcionamiento, incluso aunque fueran ajenas a las aplicadas en el caso de uso. En este sentido, la aplicación posee adaptabilidad dinámica a cualquiera de los criterios de ordenación estándar de las imágenes hiperespectrales (BSQ, BIL y BIP). Por otra parte, la aplicación posee adaptabilidad dinámica en las dimensiones espaciales y espectral de la imagen hiperespectral, presentando mínimas restricciones establecidas en tiempo de

compilación. Además, se han implementado, para esta aplicación, otras opciones de configuración estáticas. Entre estas opciones, se incluyen la cantidad de bandas espectrales a desconsiderar en los extremos tanto superior como inferior del rango espectral de la imagen. Otras opciones disponibles son el tamaño en *bits* útiles de los datos de la imagen, el paralelismo del procesamiento de los *kernels* FPGA y la cantidad de datos a promediar en la operación de filtrado.

Para ajustar el paralelismo de los *kernels*, se ha desarrollado una hoja de cálculo en Excel. El propósito de este recurso es asistir al usuario en la configuración de este parámetro. Con ello, se busca facilitar la optimización del rendimiento temporal de la aplicación a la disponibilidad de recursos de lógica programable de la FPGA. Como parte del desarrollo de esta hoja de cálculo, se recurre al lenguaje de programación Visual Basic para definir funciones que automaticen parte de los cálculos.

Como resultado final, se obtiene una aplicación basada en aceleración *hardware* por medio de FPGA. Esta aplicación es capaz de discernir adecuadamente regiones en la piel captada en la imagen hiperespectral en función del estado de afección de cáncer de piel que presenten. Además, la aceleración *hardware* ha implicado una mejora muy significativa de la velocidad de cómputo y el consumo de potencia de la aplicación con respecto a una versión completamente *software*.

10.2. Trabajos futuros

A raíz del desarrollo realizado en este trabajo, se plantean algunas opciones para proseguir y mejorar la aplicación. De esta manera, entre estas opciones se encuentran las siguientes:

1. Implementación de la aplicación en otro entorno de prototipado. Se requiere de un microprocesador para la ejecución del *host*, y lógica programable para la ejecución de los *kernels* FPGA. Además, será conveniente aprovechar la adaptabilidad del código fuente de la aplicación para optimizar el rendimiento de esta en cada uno de los nuevos soportes físicos a probar. Incluido en esta propuesta está la evaluación de la implementación en sistemas de tipo *Data Center*, como por ejemplo en placas Alveo U200.

2. Utilización de multiprocesamiento. Se estudiará la posibilidad de que la funcionalidad del *host* se distribuya en varios microprocesadores, paralelizando con ello su ejecución. En caso de implementarse, se buscará que el código fuente de la aplicación sea lo suficientemente versátil para aprovechar las posibilidades de multiprocesamiento presentes con una intervención por parte del usuario mínima.

3. Programación de la interacción *host-kernel* SAM. Se programa la utilización por parte del *host* del *kernel* SAM tras finalizar la etapa "k-means". Con ello, será posible etiquetar las regiones de *clustering* obtenidas en esta última etapa.

4. Adaptación del algoritmo "k-means" para minimizar su intervalo de *pipelining*. En el algoritmo "k-means" original, la actualización de los centroides se efectúa cada vez que se termina de evaluar la proximidad de un píxel a estos mismos centroides. En consecuencia, existe una interdependencia entre la actualización de los centroides y la determinación del centroide más cercano al píxel que limita significativamente las posibilidades de *pipelining* de la aceleración *hardware* en la etapa de *clustering*.

   Por ello, se plantea estudiar que la aceleración *hardware* se subdivida en dos *kernels* FPGA. Uno de estos *kernels* se ocuparía de evaluar la distancia de los píxeles a los centroides. El otro *kernel* sería el responsable de efectuar la actualización de dichos centroides. Además, se plantea la posibilidad de que la actualización de los centroides se efectuase cuando se han evaluado la proximidad de todos los píxeles de la imagen a los centroides, en lugar de cada vez que un píxel es evaluado. Estos cambios ocasionarían, posiblemente, que se aumentase la cantidad de iteraciones que requiere el algoritmo "k-means" para converger. Sin embargo, al reducirse la interdependencia entre las subetapas de este algoritmo, se puede lograr un *throughput* de la aplicación mayor. Esto se debe a que ambos *kernels* FPGA poseerían un intervalo de *pipelining* muy inferior al del *kernel* implementado actualmente en la etapa de *clustering*.

5. Prueba de la reconfiguración dinámica. Se estudiará la utilización de la reconfiguración dinámica de la FPGA como vía para optimizar el rendimiento de la aplicación en función de la disponibilidad de recursos de lógica programable.

Bibliografía

[1] A. A. Gowen, C. P. O'Donnell, P. J. Cullen, G. Downey, y J. M. Frias, «Hyperspectral imaging – an emerging process analytical tool for food quality and safety control», *Trends in Food Science & Technology*, vol. 18, n.º 12, pp. 590-598, dic. 2007, doi: 10.1016/j.tifs.2007.06.001.

[2] E. Torti *et al.*, «Parallel Classification Pipelines for Skin Cancer Detection Exploiting Hyperspectral Imaging on Hybrid Systems», *Electronics*, vol. 9, n.º 9, 2020, doi: 10.3390/electronics9091503. [Online]. Disponible en: https://doi.org/10.3390/electronics9091503

[3] H. Fabelo *et al.*, «Spatio-spectral classification of hyperspectral images for brain cancer detection during surgical operations», *PLOS ONE*, vol. 13, n.º 3, pp. 1-27, mar. 2018, doi: 10.1371/journal.pone.0193721. [Online]. Disponible en: https://doi.org/10.1371/journal.pone.0193721

[4] J. A. Herrera Ramírez, «Diseño e implementación de un sistema multiespectral en el rango ultravioleta, , visible e infrarrojo : aplicación al estudio y conservación de obras de arte», Universitat Politècnica de Catalunya, 2014 [Online]. Disponible en: https://upcommons.upc.edu/handle/2117/95418

[5] D. Municio Durán, «Técnicas de oversampling aplicadas al análisis de imágenes hiperestrales», Universidad de Extremadura, 2019 [Online]. Disponible en: https://dehesa.unex.es/bitstream/10662/8811/1/TFGUEX_2019_Municio_Duran.pdf

[6] A. U. G. Sankararao, P. Rajalakshmi, S. Kaliamoorthy, y S. Choudhary, «Water Stress Detection in Pearl Millet Canopy with Selected Wavebands using UAV Based Hyperspectral Imaging and Machine Learning», *2022 IEEE Sensors Applications Symposium (SAS)*, pp. 1-6, 2022, doi: 10.1109/SAS54819.2022.9881337. [Online]. Disponible en: https://ieeexplore.ieee.org/document/9881337

[7] M. Borengasser, W. S. Hungate, y R. Watkins, *Hyperspectral remote sensing : principles and applications*, 1.ª ed. CRC Press, 2007 [Online]. Disponible en: http://search.ebscohost.com/login.aspx?direct=true&db=cat07429a&AN=ulpgc.576063&site=eds-live

[8] Sociedad Española de Oncología Médica (SEOM), «Las cifras del cáncer en España», 2020 [Online]. Disponible en: https://seom.org/seomcms/images/stories/recursos/Cifras_del_cancer_2020.pdf

[9] M. C. Cameron *et al.*, «Basal cell carcinoma: Epidemiology; pathophysiology; clinical and histological subtypes; and disease associations», *Journal of the American Academy of Dermatology*, vol. 80, n.º 2, pp. 303-317, 2019, doi: 10.1016/j.jaad.2018.03.060. [Online]. Disponible en: https://www.sciencedirect.com/science/article/pii/S0190962218307758

[10] M. Kubat, *An Introduction to Machine Learning*. Springer International Publishing, 2015 [Online]. Disponible en: https://www.springer.com/gp/book/9783319348865

[11] F. Sierra-Pajuelo, A. Paz-Gallardo, y A. Plaza, «Performance Optimizations for an Automatic Target Generation Process in Hyperspectral Analysis», *Scalable Computing: Practice and Experience*, vol. 17, 2016, doi: 10.12694/scpe.v17i1.1146. [Online]. Disponible en: https://www.researchgate.net/publication/299444370_Performance_Optimizations_for_an_Automatic_Target_Generation_Process_in_Hyperspectral_Analysis

[12] Towards Data Science, «Introduction to Machine Learning for Beginners». [Online]. Disponible en: https://towardsdatascience.com/introduction-to-machine-learning-for-beginners-eed6024fdb08. [Accedido: 24 de abril de 2021]

[13] R. Fernandes de Mello y M. Antonelli Ponti, *Machine Learning: A Practical Approach on the Statistical Learning Theory*. 2018 [Online]. Disponible en: https://doi.org/10.1007/978-3-319-94989-5

[14] M. D. Guanche Hernández, «Diseño de un Acelerador Hardware FPGA para Aplicaciones de Machine Learning usando Plataforma Virtual», Universidad de Las Palmas de Gran Canaria, 2021 [Online]. Disponible en: https://accedacris.ulpgc.es/handle/10553/105183

[15] R. Khanna y M. Awad, *Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers*. Springer Nature, 2015.

[16] J. Ollé, «Qué es una máquina de vectores de soporte (Support Vector Machine) Intuición práctica y referencias», 2021. [Online]. Disponible en: https://www.youtube.com/watch?v=QoRBenaGzzw. [Accedido: 2 de marzo de 2023]

[17] D. Suramanian, «Support Vector Machine (SVM): A Simple Visual Explanation», *Towards AI*, 2019. [Online]. Disponible en: https://pub.towardsai.net/support-vector-machine-svm-a-visual-simple-explanation-part-1-a7efa96444f2. [Accedido: 2 de marzo de 2023]

[18] GIS Geography, «Multispectral vs Hyperspectral Imagery Explained», 2021. [Online]. Disponible en: https://gisgeography.com/multispectral-vs-hyperspectral-imagery-explained/. [Accedido: 26 de septiembre de 2021]

[19] J. Qin, «Hyperspectral imaging instruments», en *Hyperspectral Imaging for Food Quality Analysis and Control*, D. Sun, Ed. Academic Press, 2010, pp. 129-172 [Online]. Disponible en: https://doi.org/10.1016/B978-0-12-374753-2.10005-X

[20] Salvador Escoda S.A, «Energética y geometría solar». [Online]. Disponible en: https://elblogdelinstalador.com/energetica-y-geometria-solar/. [Accedido: 26 de septiembre de 2021]

[21] R. Ocaya, «Versatile CCD based spectrometer with FPGA controller core», *IET Science, Measurement and Technology*, vol. 10, 2016, doi: 10.1049/iet-smt.2016.0063. [Online]. Disponible en: https://www.researchgate.net/figure/Typical-relative-spectral-sensitivity-of-a-tricolour-linear-CCD-Reproduced-from-6_fig1_304246926. [Accedido: 2 de marzo de 2023]

[22] Effilux, «Iluminación LED innovadora para imágenes hiperespectrales y multiespectrales». [Online]. Disponible en: https://www.effilux.com/es/productos/hiperespectrales. [Accedido: 26 de septiembre de 2021]

[23] H. K. Noh, Y. Peng, y R. Lu, «Integration of Hyperspectral Reflectance and Fluorescence Imaging for Assessing Apple Maturity», *American Society of Agricultural and Biological Engineers*, vol. 50, n.º 3, pp. 963-971, 2007, doi: 10.13031/2013.23119. [Online]. Disponible en: https://elibrary.asabe.org/abstract.asp?aid=23119. [Accedido: 24 de septiembre de 2021]

[24] N. E. Ekpenyong, «Calibration and Characterization of Hyperspectral Imaging Systems Used for Natural Scene Imagery», *Optics and Photonics Journal*, vol. 9, n.º 7, pp. 81-98, jul. 2019, doi: 10.4236/OPJ.2019.97009. [Online]. Disponible en: http://www.scirp.org/journal/PaperInformation.aspx?PaperID=93785. [Accedido: 23 de septiembre de 2021]

[25] «Trying to Draw a 24bit image in Java», 2014. [Online]. Disponible en: https://stackoverflow.com/questions/20908987/trying-to-draw-a-24bit-image-in-java. [Accedido: 25 de septiembre de 2021]

[26] «Hyperspectral Analysis: SAM and SFF Tutorial». [Online]. Disponible en: https://www.l3harrisgeospatial.com/docs/whole-pixel_hyperspectral_analysis_tutorial.html. [Accedido: 29 de septiembre de 2021]

[27] Xilinx Inc, «Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)», 2019 [Online]. Disponible en: www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf. [Accedido: 14 de abril de 2021]

[28] L. Crockett, D. Northcote, C. Ramsay, F. Robinson, y B. Stewart, *Exploring Zynq ® MPSoC With PYNQ and Machine Learning Applications*. Glasgow, Scotland, UK: Strathclyde Academic Media, 2019.

[29] Xilinx Inc, «Zynq Migration Guide Zynq-7000 SoC to Zynq UltraScale+ MPSoC Devices» [Online]. Disponible en: www.xilinx.com/support/documentation/user_guides/ug1213-zynq-migration-guide.pdf. [Accedido: 10 de abril de 2021]

[30] Taiwan Semiconductor Manufacturing Company, «16/12nm Technology». [Online]. Disponible en: https://www.tsmc.com/english/dedicatedFoundry/technology/16nm.htm. [Accedido: 10 de abril de 2021]

[31] Xilinx Inc, «Zynq UltraScale+ EG». [Online]. Disponible en: https://www.xilinx.com/content/dam/xilinx/imgs/products/zynq/zynq-eg-block.PNG. [Accedido: 12 de enero de 2023]

[32] Xilinx Inc, «ZCU102 Evaluation Board User Guide», 2019 [Online]. Disponible en: www.xilinx.com/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf. [Accedido: 22 de abril de 2021]

[33] Xilinx Inc, «Zynq UltraScale+ MPSoC Embedded Design Methodology Guide UG1228 (v1.0)», 2017 [Online]. Disponible en: www.xilinx.com/support/documentation/sw_manuals/ug1228-ultrafast-embedded-design-methodology-guide.pdf. [Accedido: 23 de febrero de 2022]

[34] Xilinx Inc, «Zynq UltraScale+ Device Technical Reference Manual», 2019 [Online].

Disponible en: https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf. [Accedido: 2 de marzo de 2021]

[35] A. Taylor, «Introduction to the Zynq Triple Timer Counter Part One: Adam Taylor's MicroZed Chronicles Part 17». [Online]. Disponible en: https://forums.xilinx.com/t5/Xcell-Daily-Blog-Archived/Introduction-to-the-Zynq-Triple-Timer-Counter-Part-One-Adam/ba-p/407537. [Accedido: 22 de febrero de 2022]

[36] Real Digital, «ARM's global timer and ZYNQ's triple timer counter module». [Online]. Disponible en: https://www.realdigital.org/doc/8cfcb505117c049c18d31fa4287285ef. [Accedido: 22 de febrero de 2022]

[37] J. A. Castillo, «Qué es la memoria caché L1, L2 y L3 y cómo funciona», 2019. [Online]. Disponible en: https://www.profesionalreview.com/2019/05/02/memoria-cache-l1-l2-y-l3/. [Accedido: 22 de febrero de 2022]

[38] Xilinx Inc, «Zynq UltraScale+ MPSoC Product Tables and Product Selection Guide», 2016. [Online]. Disponible en: https://docs.xilinx.com/v/u/en-US/zynq-ultrascale-plus-product-selection-guide. [Accedido: 14 de abril de 2022]

[39] Xilinx Inc, «UltraScale Architecture Configurable Logic Block», 2015 [Online]. Disponible en: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf. [Accedido: 25 de febrero de 2021]

[40] Xilinx Inc, «UltraScale Architecture DSP Slice User Guide», 2019 [Online]. Disponible en: https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp. [Accedido: 26 de marzo de 2021]

[41] Xilinx Inc, «AXI Basics 1 - Introduction to AXI», 2021. [Online]. Disponible en: https://support.xilinx.com/s/article/1053914?language=en_US. [Accedido: 2 de abril de 2021]

[42] M. Scarpino, «Foundations of OpenCL programming», en *OpenCL in Action*, New York: Manning, 2012 [Online]. Disponible en: https://www.perlego.com/book/1469171/opencl-in-action-pdf

[43] J. Iparraguirre, «Procesamiento Paralelo», Bahía Blanca, 2016 [Online]. Disponible en: https://www.frbb.utn.edu.ar/hpc/lib/exe/fetch.php?media=2016-10-opencl-intro.pdf

[44] M. Rockel, «Data Acceleration with Vitis: A Traditional FPGA Designers Perspective», 2019. [Online]. Disponible en: https://www.xilinx.com/developer/articles/data-acceleration-with-vitis.html. [Accedido: 15 de marzo de 2023]

[45] ADIUVO, «Building Accelerated Applications with Vitis», 2020 [Online]. Disponible en: https://www.adiuvoengineering.com/vitis-training

[46] Xilinx Inc., «Xilinx Runtime Library (XRT)», 2023. [Online]. Disponible en: https://www.xilinx.com/products/design-tools/vitis/xrt.html. [Accedido: 27 de abril de 2023]

[47] Xilinx Inc., «Xilinx® Runtime (XRT) Architecture», 2022. [Online]. Disponible en: https://xilinx.github.io/XRT/2022.2/html/index.html. [Accedido: 27 de abril de 2023]

[48] The Khronos Group, «OpenCL Programming Model». [Online]. Disponible en: https://github.com/KhronosGroup/OpenCL-

Guide/blob/main/chapters/opencl_programming_model.md. [Accedido: 15 de noviembre de 2022]

[49] M. Guillén Allés, «OpenMP to OpenCL: Aprovechamiento de los recursos heterogéneos del sistema», Universitat Politècnica de Catalunya, 2011 [Online]. Disponible en: https://upcommons.upc.edu/bitstream/handle/2099.1/12473/69587.pdf

[50] Intel FPGA, «Writing OpenCL Programs for Intel FPGAs», 2018. [Online]. Disponible en: https://www.youtube.com/watch?v=jV2NsUUWWws. [Accedido: 17 de noviembre de 2022]

[51] The Khronos Group, «OpenCL C++ Bindings». [Online]. Disponible en: https://github.khronos.org/OpenCL-CLHPP/classcl_1_1_program.html. [Accedido: 17 de noviembre de 2022]

[52] Xilinx Inc, «Building and Running the Application», 2022 [Online]. Disponible en: https://docs.xilinx.com/r/2021.2-English/ug1393-vitis-application-acceleration/Building-and-Running-the-Application. [Accedido: 14 de noviembre de 2022]

[53] Xilinx Inc., «Exchanging Files between Host Machine and Linux Running on QEMU», *Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400)*, 2021. [Online]. Disponible en: https://docs.xilinx.com/r/2021.2-English/ug1400-vitis-embedded/Exchanging-Files-between-Host-Machine-and-Linux-Running-on-QEMU. [Accedido: 27 de febrero de 2023]

[54] eLinux, «TCF», 2013. [Online]. Disponible en: https://elinux.org/TCF. [Accedido: 27 de febrero de 2023]

[55] Xilinx Inc., «Xilinx Software Command-Line Tool», *Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400)*, 2021. [Online]. Disponible en: https://docs.xilinx.com/r/2021.2-English/ug1400-vitis-embedded/Xilinx-Software-Command-Line-Tool?tocId=_QRlJuIrwQXW5DpPkViAkA. [Accedido: 27 de febrero de 2023]

[56] AMD, «PS Software Development», 2022. [Online]. Disponible en: https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/PS-Software-Development. [Accedido: 15 de marzo de 2023]

[57] Xilinx Inc, «HLS Pragma», 2021. [Online]. Disponible en: https://docs.xilinx.com/r/2021.2-English/ug1399-vitis-hls/pragma-HLS-interface. [Accedido: 14 de noviembre de 2022]

[58] Xilinx Inc, «Developing Applications», *Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)*, 2021. [Online]. Disponible en: https://docs.xilinx.com/r/2021.2-English/ug1393-vitis-application-acceleration/Developing-Applications. [Accedido: 21 de noviembre de 2022]

[59] L. Medina Chaveli, «Generación de un Módulo Optimizado de Inferencia en FPGAs con HLS», Universitat Politècnica de València, 2021 [Online]. Disponible en: https://riunet.upv.es/bitstream/handle/10251/178237/Medina - Generacion de un Modulo Optimizado de Inferencia en FPGAs con HLS.pdf?sequence=1

[60] Xilinx Inc., «HLS Pragma», *SDAccel Development Environment Help*, 2018. [Online]. Disponible en: https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/okr1504034364623.html. [Accedido: 14 de noviembre de 2022]

[61] Xilinx Inc., «Using Vitis Embedded Platforms», 2022. [Online]. Disponible en: https://docs.xilinx.com/r/2021.2-English/ug1393-vitis-application-acceleration/Using-Vitis-Embedded-Platforms?tocId=N2b0Yi5vmR~uKOpO~foLPA. [Accedido: 16 de marzo de 2023]

[62] Xilinx Inc., «Vitis Compiler Command», 2022. [Online]. Disponible en: https://docs.xilinx.com/r/2021.2-English/ug1393-vitis-application-acceleration/Vitis-Compiler-Command. [Accedido: 16 de marzo de 2023]

[63] Xilinx Inc, «Controlling AXI4 Burst Behavior», *Vitis High-Level Synthesis User Guide (UG1399)*, 2021. [Online]. Disponible en: https://docs.xilinx.com/r/2021.2-English/ug1399-vitis-hls/Controlling-AXI4-Burst-Behavior. [Accedido: 28 de julio de 2022]

[64] Xilinx Inc., «Software Tools to Develop and Deploy Solutions on all AMD Platforms», 2021. [Online]. Disponible en: https://www.xilinx.com/products/design-tools.html. [Accedido: 1 de marzo de 2023]

[65] Xilinx Inc., «AXI Interconnect». [Online]. Disponible en: https://www.xilinx.com/products/intellectual-property/axi_interconnect.html. [Accedido: 7 de marzo de 2023]

[66] Xilinx Inc., «AXI Interrupt Controller». [Online]. Disponible en: https://www.xilinx.com/products/intellectual-property/axi_intc.html. [Accedido: 7 de marzo de 2023]

[67] Xilinx Inc., «Concat». [Online]. Disponible en: https://www.xilinx.com/products/intellectual-property/xlconcat.html. [Accedido: 7 de marzo de 2023]

[68] Xilinx Inc., «Clocking Wizard». [Online]. Disponible en: https://www.xilinx.com/products/intellectual-property/clocking_wizard.html. [Accedido: 7 de marzo de 2023]

[69] Xilinx Inc., «Processor System Reset Module». [Online]. Disponible en: https://www.xilinx.com/products/intellectual-property/proc_sys_reset.html. [Accedido: 7 de marzo de 2023]

[70] Xilinx Inc., «Vitis Environment Reference Materials», 2021. [Online]. Disponible en: https://docs.xilinx.com/r/2021.2-English/ug1393-vitis-application-acceleration/Vitis-Environment-Reference-Materials. [Accedido: 25 de enero de 2023]

[71] «Linux sudo». [Online]. Disponible en: https://www.javatpoint.com/linux-sudo. [Accedido: 9 de febrero de 2023]

[72] AMD, «Partitioning and Formatting an SD Card», *PetaLinux Tools Documentation: Reference Guide*, 2021. [Online]. Disponible en: https://docs.xilinx.com/r/2021.2-English/ug1144-petalinux-tools-reference-guide/Partitioning-and-Formatting-an-SD-Card. [Accedido: 27 de enero de 2023]

[73] «fdisk command in Linux with examples», *GeeksforGeeks*, 2021. [Online]. Disponible en: https://www.geeksforgeeks.org/fdisk-command-in-linux-with-examples/. [Accedido: 27 de enero de 2023]

[74] K. Verma, «“dd” command in Linux», *GeeksforGeeks*, 2022. [Online]. Disponible en: https://www.geeksforgeeks.org/dd-command-linux/. [Accedido: 25 de enero de 2023]

[75] Xilinx Inc., «Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit - SD Card Boot Mode settings», 2021. [Online]. Disponible en: https://support.xilinx.com/s/article/68682?language=en_US. [Accedido: 25 de enero de 2023]

[76] ssh, «SSH Command - Usage, Options, Configuration». [Online]. Disponible en: https://www.ssh.com/academy/ssh/command#ssh-command-in-linux. [Accedido: 23 de febrero de 2023]

[77] GeeksforGeeks, «ssh command in Linux with Examples», 2019. [Online]. Disponible en: https://www.geeksforgeeks.org/ssh-command-in-linux-with-examples/. [Accedido: 23 de febrero de 2023]

[78] Linuxize, «How to Use SCP Command to Securely Transfer Files», 2020. [Online]. Disponible en: https://linuxize.com/post/how-to-use-scp-command-to-securely-transfer-files/. [Accedido: 26 de enero de 2023]

[79] S. Polsani, «ZCU102 Board Setup», *Xilinx Wiki*, 2019. [Online]. Disponible en: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841937/Zynq+UltraScale+MPSoC+Ubuntu+part+2+-+Building+and+Running+the+Ubuntu+Desktop+From+Sources#ZynqUltraScale＋MPSoCUbuntupart2-BuildingandRunningtheUbuntuDesktopFromSources-ZCU102BoardSetup. [Accedido: 26 de enero de 2023]

[80] T. S. Abdelrahman, «Accelerating K-means clustering on a tightly-coupled processor-FPGA heterogeneous system», *IEEE*, pp. 176-181, 2016, doi: 10.1109/ASAP.2016.7760789. [Online]. Disponible en: https://ieeexplore.ieee.org/document/7760789

[81] R. Raghavan y D. G. Perera, «A fast and scalable FPGA-based parallel processing architecture for K-means clustering for big data analysis», *IEEE*, pp. 1-8, 2017, doi: 10.1109/PACRIM.2017.8121905. [Online]. Disponible en: https://ieeexplore.ieee.org/document/8121905

[82] Y. Wang *et al.*, «TiAcc: Triangle-inequality based Hardware Accelerator for K-means on FPGAs», *IEEE*, pp. 133-142, 2021, doi: 10.1109/CCGrid51090.2021.00023. [Online]. Disponible en: https://ieeexplore.ieee.org/document/9499568

[83] Xilinx Inc., «Power Types», 2022. [Online]. Disponible en: https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology/Power-Types. [Accedido: 2 de marzo de 2023]

[84] Xilinx Inc., «Estimating Power in Xilinx Power Estimator (XPE)», en *Vivado Design Suite User Guide: Power Analysis and Optimization*, 2020.