eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN

SISTEMAS ELECTRÓNICOS AVANZADOS

TRABAJO FIN DE MÁSTER

ESTUDIO DE IMPLEMENTACIÓN DE SOCS SOBRE FPGAS DE BAJO COSTE PARA APLICACIONES DE IA (INTELIGENCIA ARTIFICIAL)

Estudiante *Prieto, López, Pablo*
Director/Directora *Basterretxea, Oyarzabal, Koldobika*
Departamento *Grupo de Diseño en Electrónica Digital*
Curso académico *2020/2021*

Bilbao, 09, Septiembre, 2021

Resumen

El interés actual por trasladar el grueso de la computación de multitud de aplicaciones al borde (*Edge Computing*), en particular en sistemas basados en Inteligencia Artificial y Machine Learning *IA/ML*, responde a la necesidad de garantizar la respuesta en tiempo real y la seguridad de los sistemas en caso de fuga de información, requisitos que la computación en la nube (*Cloud Computing*) no puede satisfacer. Los sistemas *ADAS* (*Advanced Driving Assistance Systems*) son un campo de aplicación creciente que responde a esta demanda, con requisitos muy exigentes en términos de coste, consumo, velocidad y seguridad, que un sistema en el borde puede satisfacer.

Este trabajo forma parte de un proyecto que investiga la aplicabilidad de las imágenes hiper-espectrales para mejorar las prestaciones y la robustez de los *ADAS*. En concreto, se estudian diferentes opciones de *IPs* de microprocesadores de licencia abierta para el diseño de *SoCs* (*System-On-Chip*) sobre *FPGAs* de gama baja/media con aplicación al procesamiento de imágenes hiper-espectrales. Como alternativa a los microprocesadores *"Softcore"*, también se ha estudiado un *SoC* basado en un microprocesador *"Hardcore"*, en concreto un PSoC de Xilinx de la Familia Zynq-7000.

En primer lugar, el trabajo caracteriza el rendimiento de los microprocesadores analizados en una tarea de preprocesamiento de imágenes requerida en cámaras hiper-espectrales de 25 bandas de tipo snapshot con filtro en mosaico. En segundo lugar, se utiliza una arquitectura de *SoC* genérica con *DMAs* para analizar la forma óptima de transferir información entre los diferentes *IP* Cores que conforman el diseño y la memoria externa. Durante el desarrollo del trabajo también se han explorado varias herramientas de diseño de código abierto utilizadas en la síntesis/simulación/verificación de diseños *HDL* como alternativa a herramientas comerciales de mayor implantación.

Palabras Clave : System-On-Chip, FPGA, Microprocesadores, DMA, Imágenes Hiper-espectrales, ADAS.

Abstract

The current interest in moving the bulk of the computing of a multitude of applications to the edge (Edge Computing), particularly in systems based on Artificial Intelligence and Machine Learning (AI/ML), responds to the need to ensure real-time response and security of systems in case of information leakage, requirements that Cloud Computing cannot satisfy. ADAS systems are a growing field of application that responds to this demand, with very demanding requirements in terms of cost, consumption, speed and security, which a system at the edge can satisfy.

This work is part of a research proyect about the applicability of hyperspectral imaging to improve the performance and robustness of ADAS. Specifically, different options of open license microprocessor IPs are studied for the design of SoCs (System-On-Chip) on low/mid-range FPGAs with application to hyperspectral image processing.As an alternative to "Softcore"microprocessors, a SoC based on a "Hardcore"microprocessor, in particular a Xilinx PSoC of the Zynq-7000 Family, has also been studied.

First, the work characterizes the performance of the analyzed microprocessors on an image preprocessing task required on 25-band snapshot-type hyperspectral cameras with mosaic filtering. Secondly, a generic SoC architecture with DMAs is used to analyze the optimal way to transfer information between the different IP Cores that make up the design and the external memory. During the development of the work, several open source design tools used in the synthesis/simulation/verification of HDL designs have also been explored as an alternative to more widely deployed commercial tools.

Keywords : System-On-Chip, FPGA, Microprocessors, DMA, Hyper-Spectral Images, ADAS.

Laburpena

Egun, aplikazio ugarien konputazio-zamak ertzera eramateko (Edge Computing) interes handia dago, bereziki Adimen Artifizialean eta Machine Learning-ean (AA/ML) oinarritutako sistemetan. Joera horren arrazoi nagusia informazio-ihesak ekiditea, segurtasuna bermatzea eta sistemen denbora errealeko erantzuna bermatzea dira, hodeian konputatzeak (Cloud Computing) ezin bai ditu baldintza horiek bete. Gidatzen laguntzeko sistema aurreratuak edo ADAS (Advanced Driving Assistance Systems) gero eta aplikazioeremu garrantzitsuagoa da, eta eskari horiei erantzun behar dioten sistemak garatzea eskatzen du, oso eskakizun zorrotzak bai dituzte kostuari, kontsumoari, abiadurari eta segurtasunari dagokienez.

Lan hau ADASen prestazioak eta sendotasuna hobetzeko irudi hiper-espektralen aplikagarritasuna ikertzen duen proiektu baten parte da. Zehazki, gama baxu/ertaineko FPGAtan irudi hiper-espektralak prozesatzeko SoCak (System–On-Chip) gauzatzeko lizentzia irekiko mikroprozesadoreen IPen hainbat aukera aztertzen dira. Nukleo "bigun" edo softcore mikroprozesadoreen alternatiba gisa, nukleo zurrun edo hardcore mikroprozesadorean oinarritutako SoC bat ere aztertu da, zehazki Zynq-7000 familiako Xilinxen PSoC bat.

Lehenik eta behin, lana honek analizatutako mikroprozesadoreen errendimendua ezaugarritzen du mosaiko iragazkia duten snapshot motako 25 bandako kamera hiper-espektraletan beharrezkoa den irudien aurre-prozesamenduko ataza batean. Bigarrenik, DMAak dituen SoC generikoko arkitektura bat erabili egin da kanpoko memoria eta sistema osatzen duten IP nukleo desberdinen artean informazioa transferitzeko modurik egokiena aztertzeko. Lana garatu bitartean, HDL diseinuen sintesian/simulazioan/egiaztapenean erabiltzen diren kode irekiko diseinu-tresnak ere aztertu dira, ezarpen handieneko tresna komertzialen alternatiba gisa.

Keywords : System-On-Chip, FPGA, Mikroprozesadoreen, DMA, Irudi Hiper-espektralen, ADAS.

Índice general

Índice de figuras

Índice de tablas

Acrónimos

ADAS Advanced Driver Assistance Systems.

ASIC Application-Specific Integrated Circuit.

BIP Band Interleved by Pixel.

BSP Board Support Package.

CNN Convolutional Neural Network.

DMA Direct Memory Access.

DSP Digital Signal Processor.

FPGA Field Programable Gate Array.

FPS Frames Per Second.

FPU Floating Point Unit.

GPU Graphics Processor Unit.

HDL Hardware Description Language.

HLS High Level Synthesis.

IoT Internet-Of-Things.

IP Intellectual Property.

MCU Microcontroller Unit.

MPU Microprocessor Unit.

NN Neural Network.

PL Programable Logic.

PS Processing System.

RISC Reduced Instruction Set Computer.

RTL Register Transfer Level.

SG Scatter/Gather.

SIMD Single Instrucction - Multiple Data.

SoC System-On-Chip.

1. CAPÍTULO

Introducción

Durante todo el año 2019, se produjeron un total de 104.800 accidentes en carretera en España, por lo que resulta un número muy alto de accidentes que se debe reducir[4]. Nuevas tecnologías como la conducción autónoma o semi-autónoma supondrán una significante reducción en el número de accidentes. Estos sistemas de conducción, también llamados *Advanced Driver Assistance Systems (ADAS)*, se pueden clasificar en cinco niveles, que van desde sistemas de frenado de emergencia hasta una conducción automática.

En la actualidad, la gran mayoría de diseños de *ADAS* existentes están situados en el nivel 2, esto significa que el conductor debe estar totalmente centrado en la conducción, pero el vehículo incorpora funciones de emergencia automatizadas además de una monitorización del estilo de conducción.

Se espera que estos sistemas de ayuda a la conducción puedan llegar a reducir hasta un 40 % el número de colisiones y llegar a reducir hasta un 29 % el número de accidentes con implicación de muertes[5]. Por eso, es necesario realizar un estudio de las plataformas embebidas disponibles capaces de dar un alto rendimiento y eficiencia.

Sistemas como el frenado de emergencia ante la detección de peatones o ciclista u otros automóviles, avisador del ángulo muerto, sensores de aparcamiento... Ya se encuentran implementados y funcionando en el mundo real gracias a sensores de ultrasonidos, radar, lidar y más concretamente cámaras.

Gracias a las normativas que se están desarrollando, se facilitará la tarea de diseño y de implementación de estos sistemas. Pero, uno de los problemas que complicaran la integración de estos sistemas en el mundo real, es el estado de las carreteras. Habrá que

adaptar, mejorar y señalizar correctamente estas infraestructuras para que la integración de los ADAS suponga un gran beneficio.

En particular, la gran mayoría funciones que incorporan los *ADAS* se deben a sistemas de detección de objetos por cámara (límites de la carretera, coches, personas, señales. . .) y se han desarrollado hasta el punto de obtener sistemas con una gran eficacia bajo condiciones controladas. Sin embargo, la robustez de estos sistemas en condiciones de conducción reales (iluminación escasa, condiciones meteorológicas adversas...) es precaria, por lo que sigue siendo un tema comprometido que se debe investigar de manera profunda.

Una de las soluciones propuestas para mejorar el rendimiento de estos sistemas de visión inteligente es el uso de imágenes hiper-espectrales. Este tipo de imágenes con diferentes bandas espectrales que van desde el espectro visible hasta el infrarrojo, han permitido que aumentar la robustez de los sistemas y su eficiencia

Aunque las imágenes hiper-espectrales hayan significado una gran cambio en la tendencia en el diseño de sistemas para *ADAS*, aún queda mucho camino por recorrer. La alta exigencia y competitividad de la industria automovilística obligan a diseñar un sistema que se tiene que amoldar perfectamente a sus requisitos como es una alta eficiencia, alta seguridad y un precio competitivo. Debido a estos requerimientos la tarea de diseño y realización de pruebas es tediosa y complicada, pero los beneficios que aporta a la población en general son múltiples y diversos.

Pero, para que los sistemas *ADAS* sean capaces de satisfacer todos los requerimientos de la industria, es necesario introducir un sistema embebido en el "borde".

Aunque, en la actualidad la industria IoT sigue empleando la computación en la "nube" y el *Big Data*, se opta la computación en el "borde", o, *Edge Computing*. Este tipo de diseño es capaz de satisfacer ampliamente los requerimientos de los sistemas *ADAS*, en especial los requisitos de seguridad y respuesta en tiempo real, que son críticos en sistemas de visión inteligente.

Otra de las razones por la que se opta por la computación en el borde es su menor coste, ya que no necesita de servidores y *Clusters* en la nube. Al tratarse de una industria altamente competitiva en lo que a precio se refiere, toda mejora que suponga reducir costes será un gran avance. Desarrollar un dispositivo fácil de implementar que sea capaz de dar una respuesta en tiempo real simplificará la tarea de lanzar al mercado con mayor rapidez los *ADAS*.

Por ello, realizar un estudio sobre las plataformas hardware disponibles, y más concreta-

mente sobre las *Field Programable Gate Array (FPGA)*, es esencial en la tarea de desarrollar un sistema basado en computación en el "borde"funcional. La propuesta de dispositivo hardware se basa en un *System-On-Chip (SoC)* sobre una *FPGA*, ya que se trata de una plataforma adecuada para la ejecución de algoritmos de inteligencia artificial, necesarios para analizar imágenes hiper-espectrales. Además las FPGAs son plataformas adecuadas para realizar cálculos complejos, gracias a aceleradores hardware, o a la posibilidad de paralelizar procesos. Por lo que son una opción excelente en la computación en sistemas ubicados en el borde.

Así, en este trabajo se plantea la exploración de distintas alternativas para el desarrollo de arquitecturas *SoC* basadas en *IPs* de microprocesadores con licencia de uso libre con aplicación al procesamiento de imágenes hiper-espectrales en el ámbito del ADAS.

1.1. Contexto

Actualmente existe un gran interés en la ejecución de algoritmos *IA/ML* directamente sobre procesadores embebidos en el "borde"de los sistemas (*Edge Computing*), y aliviar así los diversos problemas derivados de un esquema de computación en la nube (*Cloud Computing*): Altas latencias, saturación de las infraestructuras de procesamiento, necesidad de gran ancho de banda, fiabilidad en las comunicaciones, problemas de seguridad...

Sin embargo, la ejecución de estos algoritmos complejos en plataformas embebidas con recursos computacionales limitados supone un gran reto de diseño que implica el desarrollo de arquitecturas heterogéneas de computación en las que se combina la ejecución por software sobre microprocesadores embebidos de las tareas de gestión de *I/Os* y preprocesamiento de datos con la aceleración por hardware de los procesos más pesados.

Más aun, en aquellas aplicaciones que requieren de gran trasiego de datos en *streaming*, la gestión de almacenamiento temporal y de las transferencias entre los distintos módulos de un *SoC*, así como entre este y las memorias externas exige un diseño cuidadoso de los procesos de comunicación y transferencia para no malograr el ancho de banda potencialmente alcanzable en el propio procesador.

En este trabajo se va a estudiar la viabilidad de implementar sistemas de procesamiento de *IA* sobre dispositivos *FPGA* de tamaño y consumo medio y bajo coste para aplicaciones que requieran de un rendimiento relativamente elevado en la fase de inferencia como son los algoritmos empleados en *ADAS*. En particular se va a estudiar la aplicabilidad y viabilidad de distintos *Intellectual Property (IP) Cores* de microprocesadores en la generación

de imágenes hiper-espectrales y la transferencia de datos a través del microprocesador, memoria externa e *IP Cores* propios.

Adicionalmente, se va a explorar el uso de herramientas de código abierto para el diseño, síntesis y verificación de *SoCs* realizados en lenguaje *VHDL*. Se trata de verificar si estas herramientas permiten soslayar las severas limitaciones intrínsecas al uso de las herramientas comerciales habituales proporcionadas por los fabricantes de *FPGAs* para los estándares más recientes de este lenguaje como puede ser VHDL-08.

1.2. Objetivos

Este Trabajo Fin de Máster es un proyecto de investigación centrado en el análisis de las diferentes alternativas existentes para la implementación en *FPGA* de bajo/medio coste, un *SoC* en los que se puedan integrar sistemas de *IA* basados en algoritmos de *ML*.

Como objetivo principal se va a realizar el diseño de un *SoC* sobre una *FPGA* de bajo coste para la implementación de sistemas de *IA* con aplicación al procesamiento de imagen hiper-espectral con aplicación al *ADAS* evaluando distintas opciones de implementación.

Para llegar al cumplir con el objetivo principal es necesario establecer unos objetivos intermedios

- Evaluar el rendimiento de las distintas alternativas de *Soft IP Cores* de licencia libre para la implementación de *SoCs* de alto rendimiento sobre FPGAs de rango bajo y medio.
- Caracterizar una arquitectura genérica de *SoC* con *DMAs* para la optmización de la transferencia de datos entre el procesador, la memoria externa y eventuales coprocesadores harrdware integrados.
- Evaluar el uso de herramientas de código abierto en el diseño, síntesis y verificación de *SoCs* sobre FPGAs de Xilinx.

1.3. Beneficios del Proyecto

Este Trabajo Fin de Máster aportara una serie de beneficios que facilitaran realizar el diseño definitivo de un *SoC* que sea funcional y cumpla los requisitos que la industria automovilística propone.

El mayor beneficio que se obtendrá con este estudio, es dar una visión sobre el estado del arte de los microprocesadores capaces de ejecutar algoritmos de *IA/ML*. Se pretende dar una visión amplia sobre los *IP Cores* y herramientas disponibles para facilitar el trabajo de selección de componentes.

Se espera obtener buenos resultados durante el proceso de caracterización de los microprocesadores al ejecutar algoritmos de *IA/ML*. De estos estudios se espera aportar una propuesta de microprocesador que facilite la tarea de diseño de un *System-On-Chip* para poder emplear en etapas futuras del proyecto.

La caracterización de las distintas opciones de dispositivos hardware para la realizar las transferencias de datos dentro del *SoC* facilitara la posterior implementación de la red neuronal de clasificación para el sistema de detección de objetos.

Otro beneficio sera la proposición de como deben estar estructurados los datos en memoria y como acceder a ellos de manera rápida y sencilla. Con esto se pretende minimizar el número de accesos a memoria, aumentando el rendimiento.

2. CAPÍTULO

Estado del Arte

2.1. Visión Inteligente

La Visión Inteligente son el conjunto de técnicas y tecnologías capaces de realizar el análisis y procesado de imágenes de manera automática. El principal objetivo de la visión inteligente es analizar objetos a través de imágenes sin necesidad de interacción humana. Esta tecnología es aplicada a día de hoy en muchos campos diferentes: Industria alimentaria, conducción asistida, industrial textil, inspección de componentes electrónicos...

Para poder establecer un sistema de visión inteligente es necesario disponer de una infraestructura. Esta se debe componer al menos de una cámara, un procesador y una fuente de luz. Para poder obtener un buen rendimiento, la adquisición de la imagen debe realizarse cuidadosamente, es decir, obtener una imagen con poco ruido y un buen contraste, para liberar carga computacional del procesador. Normalmente, para obtener un buen rendimiento y una respuesta rápida, también se suele añadir un paso de preprocesado de la imagen para eliminar la mayor cantidad de píxeles que no aporten información relevante[6].

En los *ADAS*, la visión inteligente se implementa a través de los sistemas de ayuda a la conducción, concretamente en los sistemas de detección de objetos. En este tipo de automóviles se pueden encontrar diferentes tecnologías que se emplean en los sistemas de detección de objetos para asistir a la conducción: *RADAR*, *LIDAR* o Visión por computador.

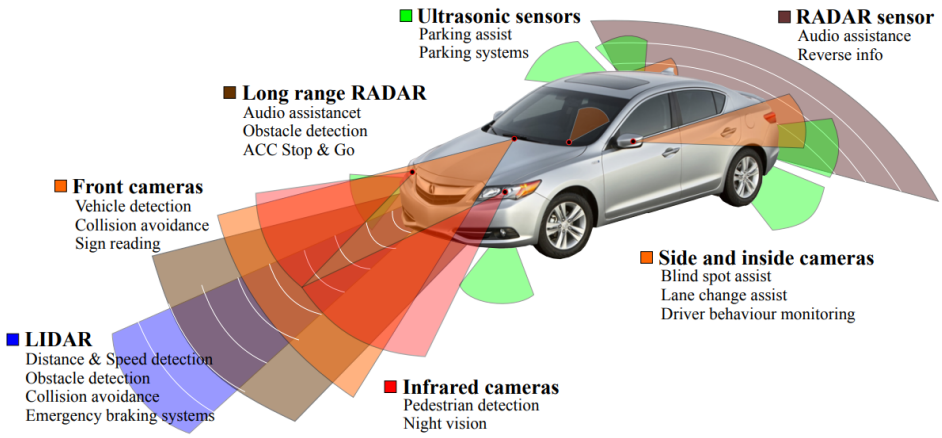

Figura 2.1: Sensores y Posición en ADAS[1]

A diferencia de sensores como son el *RADAR* y el *LIDAR*, los sistemas basados en visión inteligente están muy condicionados por las condiciones meteorológicas. Como ventaja, la visión inteligente, tiene un campo de visión más amplio, además, es capaz de analizar y categorizar elementos.

En la industria de la conducción asistida, la visión artificial ha supuesto un gran cambio gracias a toda la información que puede procesar: señales de tráfico, semáforos, posición de objetos, límites de la carretera...

Sistemas complejos en los que se empleaban tecnologías como *RADAR* o *LIDAR* se han visto superados en rendimiento por sistemas de análisis de imágenes por visión artificial en los que se emplea una sola cámara. Actualmente, estos sistemas de análisis de imágenes incluyen elementos de *IA* y *ML*, ya que se basan en redes neuronales, *NN*, y procesos de aprendizaje e inferencia del sistema.[7].

2.1.1. Inteligencia Artificial y Visión Inteligente

En la actualidad, los sistemas de visión computerizada emplean técnicas y herramientas de *IA/ML* como son máquinas vectoriales, redes neuronales convolucionales, análisis de componentes[8].

Más concretamente, en la computación inteligente, que es el conjunto de metodologías empleado para manejar problemas difíciles de resolver, como es la visión inteligente, es necesario emplear algoritmos de *IA* y *ML*. Las técnicas más representativas son los sistemas borrosos o *Fuzzy*, redes neuronales o algoritmos evolutivos. Además, estas técnicas

se pueden llegar a combinar como pueden ser sistemas neuro-borrosos. El empleo de estos tipos de algoritmos se debe a las no linealidades y elementos complejos que se pueden presentar en los sistemas de visión inteligente.

El uso de *DPMs* (*Deformable Part Models*) ha sido empleado en la visión inteligente hasta la llegada de las *Convolutional Neural Network (CNN)*, que a día de hoy, es la técnica más avanzada de detección de objetos basados en imágenes[9]. Estas técnicas basadas en redes neuronales son capaces de aprender a partir de un conjunto de muestras, para posteriormente aplicar el conocimiento adquirido y dar una respuesta, aprendizaje e inferencia.

Las redes neuronales, y más recientemente, el *Machine Learnning* Extremo, *ELM* (*Extreme Machine Learning*), se emplean en los *ADAS* para llevar a cabo diversos procesos inteligentes como son el procesado de imágenes, el estudio de la conducción o reconocer distracciones por parte del conductor. El excelente rendimiento de todos estos sistemas confirma que la Inteligencia Artificial y el *Machine Learning* son muy importantes en el desarrollo de *ADAS*[10].

Las redes neuronales se basan en una capa de entrada y una de salida, pero, por medio se pueden encontrar múltiples capas que son las encargadas de implementar el algoritmo de aprendizaje profundo, o *Deep Learning*. Gracias a las múltiples capas del algoritmo, se pueden obtener datos clave de cada imagen y realizar un análisis preciso[11].

Las redes neuronales convolucionales se basan en una arquitectura secuencial, Figura 2.2, donde se reciben los datos a la entrada y la información se procesa de manera secuencial en cada una de las capas. Este tipo de redes neuronales se pueden dividir en en tres grandes capas: capa convolucional, capa de agrupación y capa no lineal. La capa convolucional se encarga de computar sobre las características de la entrada los kernels convolucionales. La capa de agrupación se encarga de asegurar una robustez de las características de la imagen aplicando diferentes filtros. Esto sirve para evitar la distorsión de los datos, además, en algún caso, los filtros también se pueden encargar de reducir el tamaño de las características para así reducir la carga computacional.

Para conformar una red neuronal profunda es necesario apilar múltiples capas convolucionales y de agrupación. La arquitectura deber ser jerárquica, las capas bajas de la red neuronal son las encargadas de capturar las características de bajo nivel como son "borde"s o texturas. Las capas altas son las encargadas de capturar estructuras más abstractas o información semántica, la cual es útil en el proceso de clasificación.

El inconveniente más importante que tiene la Inteligencia Artificial, sobre todos las meto-

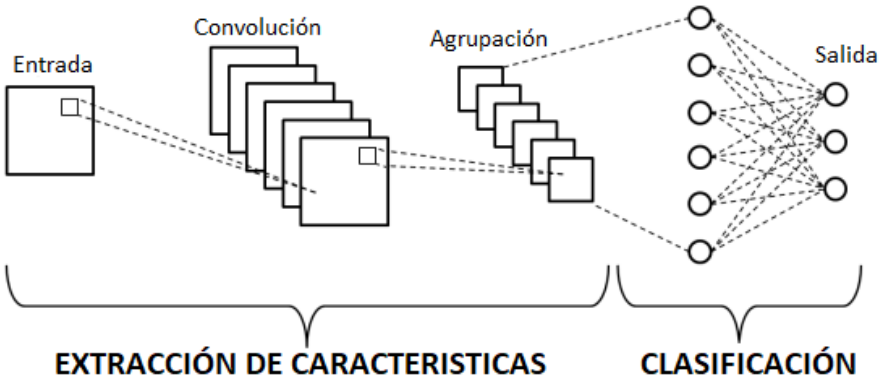

Figura 2.2: Ejemplo de Arquitectura de CNN

dologías que se han comentado anteriormente, es que requieren de una gran cantidad de datos y recursos computacionales. Por esta razón, llevar a cabo sistemas con un alto rendimiento es una tarea complicada, y más aún, si se quiere mantener el consumo de potencia bajo. Por estas razones se plantean los aceleradores hardware, con el fin de mantener los requerimientos de tamaño-tiempo por debajo de un límite.

Inteligencia Artificial en FPGAs

Los dispositivos más adecuados en la actualidad para correr estos algoritmos complejos como son las *CNNs* son *Graphics Processor Unit (GPU)*, pero, las *FPGAs* están emergiendo como una fuerte alternativa. Aunque el ancho de banda que ofrecen los algoritmos de IA en *GPUs* es alto, pero, gracias a proyectos como *Catapult*[12] o *Brainwave* de Microsoft[13], el interés en emplear *FPGAs* es creciente. Ventajas como el alto rendimiento con bajo consumo, paralelismo y la lógica reconfigurable son factores de peso en el cambio de tendencia hacia las *FPGAs*.

En el caso de implementaciones de algoritmos complejos como son las *CNNs* las *FPGAs* son alternativas más flexibles que las *GPUs* al poder soportar diversas modificaciones sobre los algoritmos. Uno de los mayores puntos débiles en la implementación de algoritmos *CNN* en *FPGAs* es la falta de una herramienta de alto nivel que sea capaz de implementar redes *CNN* en *FPGAs* de una manera sencilla. Por eso, se requiere de un gran conocimiento de programación a nivel bajo con la desventaja de ser un proceso tedioso y lento[14].

Aunque las *FPGAs* se caracterizan por implementar diseños hardware complejos es recomendable realizar un co-diseño hardware/software para poder alcanzar una alta eficiencia

en la implementación de redes neuronales.

Esto se debe a que el rango de optimización en hardware es limitado. Existen diversos métodos de optimización, desde los más básicos como es una aproximación al techo del modelo, roofline model[15] o mas complejas como es el pipelining o las estructuras con arrays sistólicos[16]. Estos métodos de optimización junto a la cuantización de los datos, optimización del flujo de datos o la reutilización de datos, pueden ayudar a incrementar el rendimiento de algoritmos *IA* en *FPGAs*.

2.1.2. Visión Inteligente Embebida

Es importante entender y analizar los aspectos de la conducción a la hora de implementar los sistemas *ADAS*. Sin embargo, la implementación de estos sistemas en un entorno real no es una tarea sencilla. La gran mayoría de los algoritmos se desarrollan sobre un ordenador, que cuando van a ser implementados sobre dispositivos embebidos su rendimiento se ve degradado de manera significativa.

Es por ello, que es necesario cumplir una serie de requisitos para poder ser implementados en un sistema embebido: confianza, rendimiento en tiempo real, bajo coste, tamaño pequeño y consumo bajo[17].

El sistema debe ser confiable, es decir, que sea justificable la implementación del servicio. Debe ser tanto seguro para el usuario como para el entorno que le rodea. También debe ser integro, que no se produzcan ninguna fuga de información o alteración de los datos.

El sistema debe alarmar de los fallos al usuario, pero debe ser robusto para poder distinguir entre un error real o un falso positivo. Los falsos positivos pueden crear situaciones peligrosas que pueden crear una sensación de falsa seguridad en el usuario.

Asegurar que el algoritmo de detección es robusto es esencial, pero también que el sistema sea capaz de generar una respuesta a tiempo ante cualquier perturbación también es clave. Dependiendo de los requerimientos del algoritmo, habrá que adecuar la potencia de computación del sistema. En el caso de realizar un filtrado el nivel de potencia de computación es mucho menor que en un proceso de segmentación y clasificación de objetos.

Así que, dependiendo de en que tipo de algoritmo se base el sistema de visión inteligente, habrá que adaptar el sistema embebido con el fin de asegurar en cada caso una respuesta rápida.

Otro de los requerimientos es el bajo coste del dispositivo. Debido a la alta competitividad del mercado, es necesario desarrollar sistemas que tengan un bajo coste.

Por último, asegurar un bajo consumo es importante en todos los sistemas embebidos, pero es especialmente relevante en aplicaciones del sector automovilístico. Se debe a que en este sector, la eficiencia es una de las características más valoradas en un automóvil.

2.1.3. Imágenes Hiper-espectrales en Visión Inteligente

Los sistemas de detección de objetos es una de las técnicas más importantes en la visión inteligente. Estos sistemas se basan en la extracción de características como son la intensidad, color, textura, orientación... Sin embargo, estos modelos pueden fallar cuando todos los objetos tienen un color similar al fondo que les rodea.

Este fenómeno de no ser capaz de distinguir el objeto del fondo se llama metamerismo, que está causado por la limitación de la visión humana, que solo es capaz de percibir los colores primarios generados. Este mismo problema se puede aplicar a las cámaras *RGB*, que solo son capaces de producir tres canales de colores. Una de las soluciones para aliviar el metamerismo es proveer una descripción detalla de la respuesta espectral percibida del objeto. Las imágenes hiper-espectrales son conjuntos de espectros de diferente longitud de onda desde el espectro visible hasta el infrarrojo, Figura 2.3.

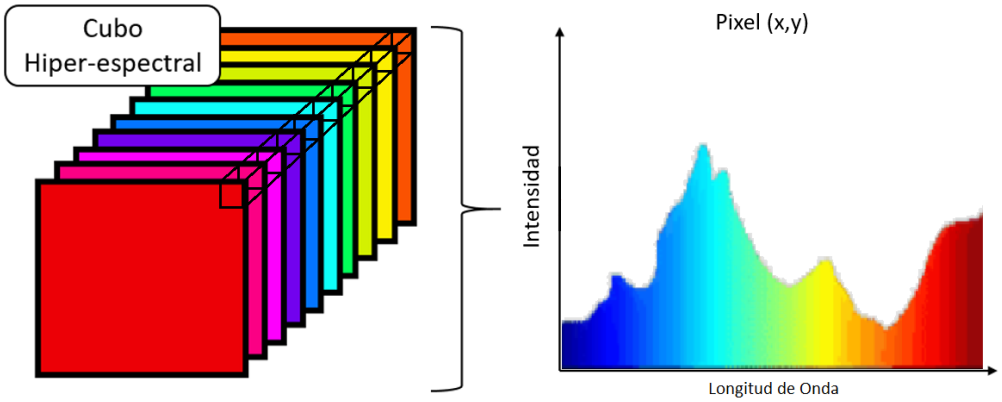

Figura 2.3: Cubo Hiper-espectral

Para poder extraer información del cubo hiper-espectral es necesario realizar una serie de tareas, pre-procesado de la imagen, reconocimiento de patrones, segmentación, separación espectral, regresión y clasificación y por último procesado de la imagen[18].

El pre-procesado de la imagen es un paso que no se le da una alta relevancia, pero gracias a este paso se pueden obtener resultados óptimos. La minimización de los píxeles muertos o que no contengan información reduce la carga computacional del algoritmo.

El reconocimiento de patrones se encarga de identificar la relación entre los píxeles con el fin de identificar cuales son los objetos principales en la imagen. El método más empleado es el análisis de componentes principales (*PCA*). Este algoritmo no necesita de un proceso de entrenamiento previo para poder identificar los patrones.

En el proceso de segmentación, se dividen los píxeles en diferentes grupos considerandos sus propiedades espectrales. Aunque estos métodos agrupan los píxeles según sus características espectral, no se pueden considerar métodos de clasificación ya que no requieren de un proceso de entrenamiento para la implementación del algoritmo.

La separación espectral se encarga de identificar las texturas de cada uno de los píxeles, con el fin de darlos el número correcto de componentes. El resultado final es un conjunto de imágenes para cada componente espectral. La diferencia con los procesos de reconocimiento de patrones es que estos algoritmos no se centran en estudiar las diferencias entre los píxeles, si no que, estudian las características espectrales de cada píxel.

La regresión y la clasificación son métodos supervisados, ya que es necesario un entrenamiento previo del algoritmo para asegurar una implementación robusta. Al tratarse de un método supervisado, se requiere de un proceso de validación para asegurarse de la capacidad del modelo para predecir en su fase de inferencia. Una vez que se ha completado el entrenamiento, el algoritmo sera capaz de realizar tareas de predicción e identificación píxel a píxel en cada imagen. Existen muchos algoritmos para entrenar estos algoritmos: mínimos cuadrados, regresión lineal, maquinas vectoriales o redes neuronales.

Por último, el procesado de la imagen se encarga de analizar la distribución, forma, cantidad, de los objetos identificados.

Aunque la utilización de imágenes hiper-espectrales en detección de objetos y patrones está bastante extendida en las herramientas de visión artificial, todavía existen algunos desafíos que superar en su implementación.

El primer problema que se puede encontrar es la limitada base de datos libres de derechos, debido al relativo alto coste de los dispositivos capaces de adquirir imágenes hiper-espectrales. El segundo, es que los sistemas tradicionalmente empleados como son los basados en escala de grises, no pueden ser empleados directamente sobre la imagen hiper-espectral, por lo que es necesario desarrollar métodos capaces de explorar las propiedades

espectrales y espaciales de la imagen. Otro problema es que la calidad de la imagen se ve deteriorada debido al ruido y la pobre resolución espacial.

Imágenes Hiper-espectrales en FPGAs

Las *FPGAs* son dispositivos ideales para realizar la clasificación y generación de imágenes hiper-espectrales, porque al ser dispositivos reconfigurables y de bajo consumo son ideales para dispositivos integrados en el ecosistema *Internet-Of-Things (IoT)*. Aunque, son dispositivos muy favorables para implementar algoritmos de *IA/ML*, no se pueden manejar grandes cantidades de datos por lo que el número de bandas debe ser limitado. Este proceso debe realizarse antes de la implementación tras realizar un estudio previo, ya que si no se limita, todas las bandas pueden llegar a emplearse en el algoritmo, por lo que el rendimiento y consumo de la *FPGA* se ve afectado negativamente[19].

Las *FPGAs* ofrecen un alto rendimiento porque se pueden llegar a implementar un camino óptimo para los datos de cada tarea computacional. Estos caminos pueden aprovechar el paralelismo y la inclusión de una arquitectura de memoria personalizada, además de poder llegar a ejecutar diferentes tareas en paralelo para poder conseguir un mayor ancho de banda. Emplear estos caminos de datos optimizados, no solo aumentaran el ancho de banda, si no que también, se reducirá el consumo de potencia, aumentando la eficiencia del dispositivo[20].

2.2. Edge Computing

A día de hoy, gracias al crecimiento del *IoT*, los horizontes de la computación en el "borde" (*Edge Computing*) han crecido ampliamente, se estima que para 2025 se lleguen a superar los 70 mil millones de dispositivos[21]. Los dispositivos *IoT* tienen diversas y numerosas aplicaciones en un amplio número de sectores, como el cuidado de la salud, el industrial, el alimenticio, el aeroespacial, transporte... Cada dispositivo *IoT* se encarga de recoger los datos de manera continua, lo que requiere de un análisis rápido de los mismos para poder tomar decisiones en tiempo real, especialmente en aplicaciones donde una respuesta rápida es crucial: vehículos autónomos, redes eléctricas, mantenimiento predictivo, procesos automatizados...

Por esta razón, la computación en el "borde" toma una mayor relevancia, ya que surge la necesidad de procesar los datos obtenidos lo más cercano posible al sensor que ha recibido

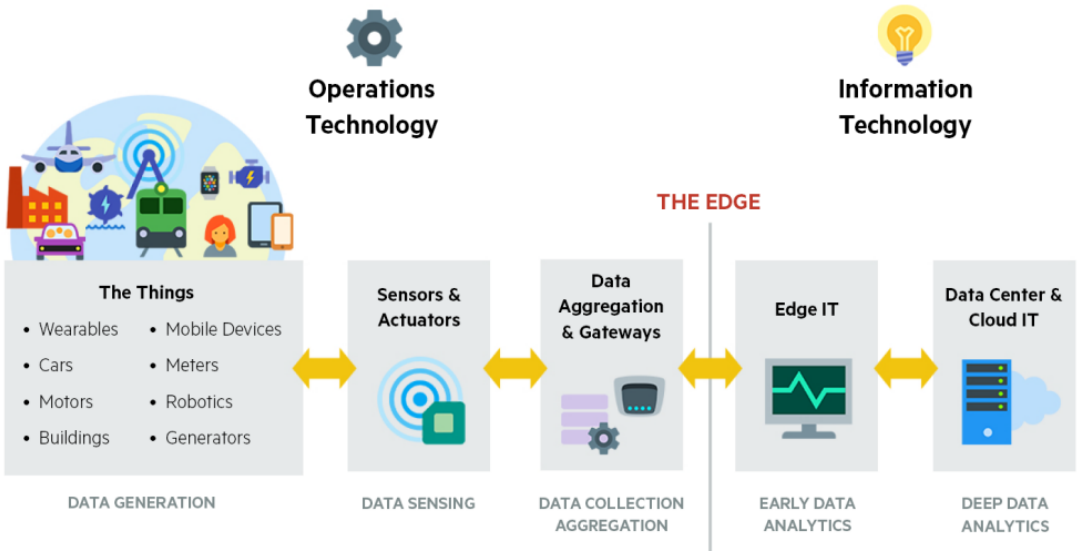

Figura 2.4: IoT (Internet Of Things) y su Flujo de Datos[2]

esos datos. Con esto se evitan las trasferencias de datos sin procesar que son las causantes de las grandes latencias en los sistemas a la hora de generar una respuesta.

Debido a este gran crecimiento de la computación en el "borde", la computación en la nube, muy importante años atrás, se está quedando relegada a un segundo plano en la industria *IoT*. Pero para comprender aún más él porque de este cambio hacia la computación en el "borde", lo mejor es comparar ambas opciones.

2.2.1. Computación en la Nube

La computación en la "nube"se basa en la ejecución de algoritmos sobre datos en que están disponibles en la "nube". Los datos generados por el dispositivo son transferidos a un *Data Center* a través de la red y estos no son procesados hasta que la transmisión ha finalizado. Esta tecnología ha seguido desarrollándose en estos últimos años, se ha conseguido distribuir la carga computacional, paralelizar los procesos, almacenar grandes cantidades de datos. Pero, a pesar de su desarrollo, la computación en la "nube"sigue teniendo puntos débiles que afectan a su rendimiento para dispositivos *IoT*, estos se comentaran los siguientes párrafos[22].

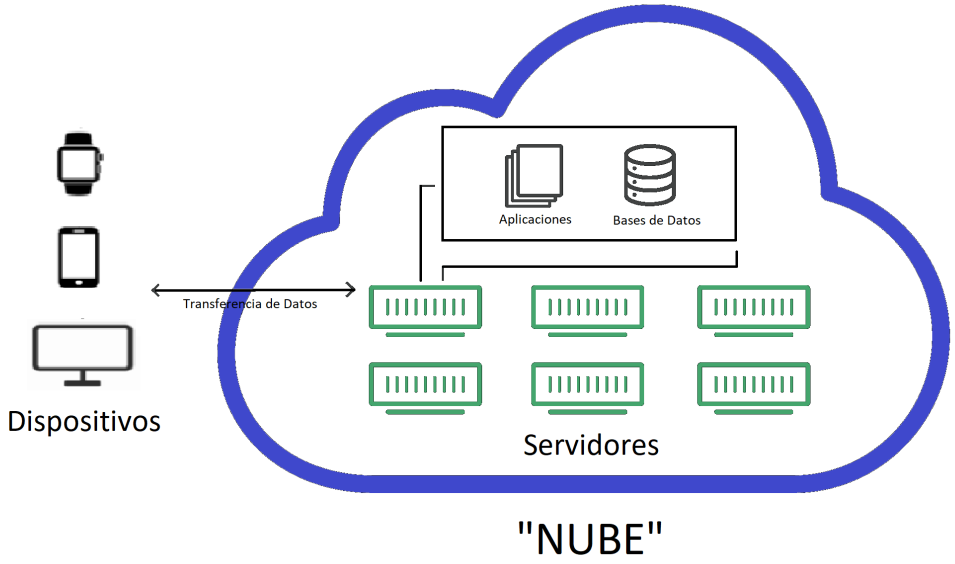

Figura 2.5: Computación en la Nube

2.2.2. Computación en el Borde

La computación en el "borde" se basa en la ejecución de algoritmos sobre los datos que se producen de los dispositivos y sensores en el "borde" de los sistemas. Es decir, el procesado de los datos se realizará antes de subir o después de descargar los datos de la "nube", un paso intermedio entre el dispositivo generador de datos y el almacenamiento en la "nube"[23].

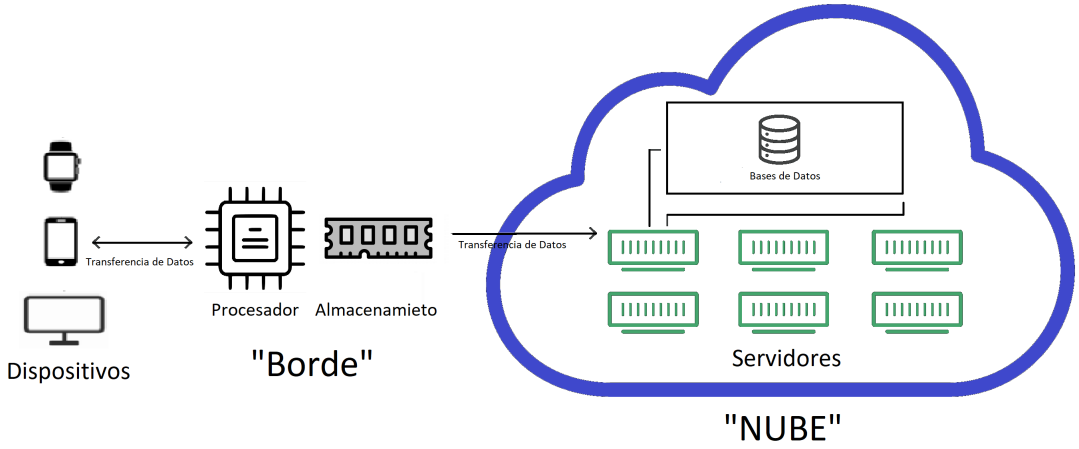

Figura 2.6: Computación en el Borde

En otras palabras, la computación en el "borde" provee servicios y realiza cálculos en el "borde" entre la red y la obtención de los datos. Se basa en migrar la capacidad de la

computación, almacenamiento y recursos de la "nube" al dispositivo, para proveer servicios y cumplir con los requisitos críticos de la industria *IoT*: Ejecución en tiempo real, optimización de los datos, seguridad, privacidad...

2.2.3. Aspectos Críticos en la Industria IoT

Tradicionalmente, la industria *IoT*, empleaba la computación en la "nube"para el análisis de los datos. Si bien, los centros de datos disponen de la capacidad para procesar y almacenar una gran cantidad de datos, su mantenimiento es costoso y requiere de una gran cantidad de energía. Además, no solo es costosa, sino que la transferencia de datos a la "nuberequiere de mucho tiempo y provoca un retraso en la respuesta. Tampoco es de utilidad almacenar todos los datos producidos por los dispositivos *IoT*, ya que solo una pequeña parte de estos datos puede resultar útil.

Debido a estos factores limitantes en la computación en la "nube", en la actualidad, las empresas están migrando hacia la tecnología de la computación en el "borde". Esta tecnología es capaz de satisfacer los aspectos críticos de la industria *IoT*: Respuesta rápida, tiempo real, seguridad, bajo consumo de energía y bajo coste. Al no ser necesario transferir los datos para su procesado en la "nube", el dispositivo actúa en tiempo real. La velocidad de procesado se ve incrementada, por lo tanto, el dispositivo puede ofrecer una respuesta rápida. Estos factores son muy importantes en sectores donde el tiempo es crítico como pueden ser la conducción inteligente o la monitorización de vídeo.

La seguridad es otro de los requisitos esenciales en la industria *IoT*, por eso, la computación en el "borde" aporta ventajas frente a la computación en la "nube". Esto se debe a que, en la computación en la "nube", es necesario enviar todos los datos antes de procesarlos, por lo que la seguridad se ve comprometida a pérdidas de datos y filtrados de información. Sin embargo, en la computación en el "borde", no es necesario subir los datos antes de procesarlos, por lo que se reduce el riesgo en la transmisión a la red. También, en la computación en el "borde", cuando los datos se ven comprometidos por un mal funcionamiento del dispositivo, solo se ven afectados los datos que están guardados en el almacenamiento local.

Otro de los aspectos en los que favorece la computación en el "borde" a la industria *IoT* es en el almacenamiento de los datos. Al contrario que en la computación en la "nube", la cantidad de datos almacenados en la computación en el "borde" es mucho menor, por lo que se emplean un menor número de recursos y energía. Además, la computación en

el "borde", tiene la ventaja que también puede enviar datos a la "nube" en el caso de ser útiles posteriormente.

Como último aspecto crítico a destacar, decir que la computación en el "borde" no depende de servidores y/o conexión de red para su funcionamiento, lo que hace que el dispositivo pueda funcionar siempre.

Resumiendo, se puede decir que la computación en el "borde" es más eficiente, consume menos recursos al no necesitar un gran ancho de banda, y por lo tanto, la capacidad de computación se ve mejorada[24].

Pero estas ventajas de la computación en el "borde" no quieren decir que se deje atrás a la computación en la "nube", ambas deben coexistir para seguir desarrollando productos y servicios *IoT*. Características de la computación en el "borde" como el *Real-Time* son necesarias para actuar frente a un problema en un instante de tiempo, pero también el análisis de los datos almacenados en la "nube" es esencial para conocer el funcionamiento de un sistema. También para interrelacionar datos de diferentes dispositivos es esencial disponer de la computación en la "nube", ya que el sistema puede componerse de más de un dispositivo.

En definitiva, se puede decir que la computación en el "borde" es una ventaja para la computación en la "nube". De esta manera, la computación en la "nube" se puede dedicar al almacenamiento y procesamiento de grandes cantidades datos. Por lo tanto, tendrá efecto en decisiones o respuestas a largo plazo, mientras que la computación en el "borde.ªctuará en instantes cortos de tiempo cumpliendo con los requisitos del *Real-Time*.

2.2.4. Factores de Riesgo de dispositivos IA/ML en el borde

Como se ha visto anteriormente, incorporar la tecnología de *Edge Computing* junto a algoritmos de *IA/ML* es beneficioso, pero también supone un desafío a la hora de realizar el diseño del dispositivo. El primer desafío que se puede encontrar es el procesamiento y consumo de energía. La *IA* se basa en un software de entrenamiento e inferencia. El entrenamiento enseña a un modelo a identificar los parámetros más relevantes para poder interpretar los datos eficientemente y se trata de una tarea con un uso intensivo de energía y recursos. La inferencia son las predicciones del modelo que se basan en el aprendizaje.

En la computación en la "nube" esta tarea de entrenamiento se realizaría en la "nube" para posteriormente implementar el modelo obtenido, software, en el dispositivo *IoT* para poder realizar inferencia. Sin embargo, en la computación en el "borde", las dos tareas de

la *IA* se implementan en el dispositivo *IoT*, por lo que la capacidad de procesamiento del hardware se ve más demandada. Este mayor consumo de energía plantea un problema, por lo que se requiere de un equilibrio de la capacidad de procesamiento frente a la energía al diseñar el dispositivo *IoT*.

El almacenamiento de datos y la seguridad presentan el segundo desafío. La gran mayoría de los datos generados por el dispositivo *IoT* se almacenaran en el propio dispositivo, transfiriendo una pequeña parte a la "nube". En el diseño habrá que tener en cuenta la capacidad de memoria y la distribución de esta para almacenar tanto datos como los propios parámetros del algoritmo *IA* correctamente.

En definitiva, para realizar la computación en el "borde" de una manera eficiente, es necesario desarrollar hardware con un alto poder de procesamiento, un bajo consumo de energía, y un software eficiente para realizar el aprendizaje y la inferencia de una manera eficiente. El último problema es la personalización. Debido a que los dispositivos *IoT* abarcan un espectro de sectores/escenarios muy diferentes, no existen estándares y normas específicas que seguir.

2.2.5. Hardware para Edge Computing

Como se ha visto en los apartados anteriores, el *Edge Computing* y la Inteligencia Artificial están íntimamente relacionados, pero para tratarse de un sistema en el "borde"no es necesario que se implementen algoritmos de *IA* o *ML*, ya sea de inferencia o de aprendizaje. Sin embargo, el hardware diseñado específicamente para la implementación de algoritmos de *IA/ML*, generalmente, es válido para el *Edge Computing*[25].

A continuación se explicara más a fondo los cuatro grandes grupos en los que se puede diseñar un sistema de *Edge Computing*: Procesadores de propósito general (*MCU* o *MPU*), hardware basado en *GPU*, *ASIC* y *FPGA*.

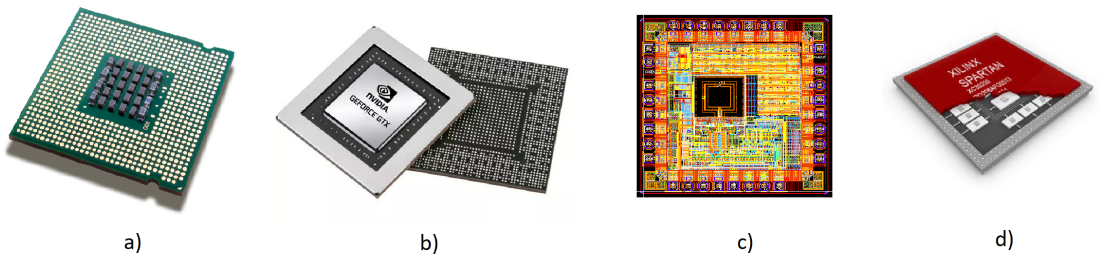

Figura 2.7: a) Microprocesador, b) GPU, c) ASIC y d) FPGA.

Procesadores de Propósito General

Los procesadores de propósito general se tratan de *Microcontroller Unit (MCU)* o *Microprocessor Unit (MPU)* que no han sido diseñados para una aplicación concreta. Estas dos opciones son excelentes en aplicaciones *IoT* y son capaces de ejecutar en software redes neuronales bastante simples. Implementar algoritmos de *IA* complejos en esta serie de dispositivos supone un gran desafío a la hora de optimizar el código para obtener un alto rendimiento[26].

Debido a esta limitación en la capacidad de procesado, estos dispositivos se emplean más como *End Points* que como sistemas de *Edge Computing*. Algunos diseñadores y fabricantes de microprocesadores han desarrollado a lo largo de los años paquetes y librerías de software específicas para ejecutar algoritmos de *ML* de forma eficiente en estas plataformas. Por ejemplo en el caso de los microcontroladores de STMicroelectronics existe X-CUBE-AI, una expansión que amplía el potencial de su entorno de desarrollo STM-CUBE, permitiendo una conversión automática de *ANNs* previamente entrenadas a hardware de bajos recursos[27].

Por otro lado, ARM ha desarrollado sus propios procesadores para sistemas *IoT*. Específicamente el Cortex-M55 y la microNPU Ethos para la aceleración de la inferencia de algoritmos sobre dispositivos sencillos[28]. Además, si se tiene en cuenta que la gran mayoría de dispositivos móviles e *IoT* incluyen *SoCs* de ARM, las posibilidades de implementación de algoritmos *IA/ML* son amplias y variadas.

Pero no solo los fabricantes o diseñadores de microprocesadores proporcionan librerías o código optimizado para *IA/ML*. Por ejemplo, empresas punteras en la computación en la "nubeçomo Google o Facebook, han adaptado sus plataformas como Tensor Flow[29] o Pytorch[30] para la implementación de modelos *ML* en dispositivos *IoT* basados en microprocesadores.

Hardware Basado en GPU

Las *Graphics Processor Unit (GPU)*, son aquellas plataformas capaces de ejecutar programas paralelos a más altas velocidades que un procesador. Concretamente en las rutinas que requieran el uso de cálculos tipo *MACC* (*Multiply Accumulate*), lo que la hace adecuada para computar algoritmos de *IA/ML*, y más específicamente en la tarea de aprendizaje profundo. Al igual que en los microprocesadores, la implementación de los algoritmos se realiza en software. Pero al contrario del código software secuencial empleado en los

procesadores de propósito general, en las *GPUs*, el código software (CUDA) está pensado para su ejecución en paralelo[31][32]. Fabricantes de *GPUs* como NVIDIA como desarrolladores de terceros han lanzado al mercado una gama amplia de *APIs* y librerías específicas para *IA/ML*.

Como consecuencias de ello, el uso de GPUs como dispositivos de computación en el "borde"para algoritmos de *IA/ML* es el más extendido a día de hoy. Sin embargo, características como el *throughput* (ancho de banda), el área o el consumo del dispositivo son peores que los que se pueden obtener con procesadores *FPGA* o *ASIC*.

Hardware Basado en ASIC

Los *Application-Specific Integrated Circuit (ASIC)*, son circuitos integrados que están diseñados y fabricados específicamente para una aplicación concreta. Los *ASIC* pueden llegar a ser muy adecuados para sistemas en los que se requiera *Edge Computing*, debido a su pequeño tamaño, bajo consumo de energía, mayor rendimiento y seguridad. En su contra esta su poca flexibilidad, por lo que no son procesadores adecuados para implementar diferentes algoritmos de Inteligencia Artificial[33][34].

Tampoco se caracterizan por una facilidad a la hora de diseñar el circuito digital que conforma el algoritmo, ya que para realizar un diseño simple, requiere de un conocimiento previo bastante alto. Por último, el coste de fabricación del circuito supone un contratiempo, debido a su alto precio, porqué, para que sea rentable, habría que fabricar un número muy alto de dispositivos.

Hardware Basado en FPGA

Una *FPGA* es un circuito integrado configurable, es decir, el diseñador es el encargado en diseñar la estructura hardware del circuito integrado. Los aceleradores hardware basados en *FPGA* pueden llegar a conseguir un alto rendimiento en la capacidad de computación con baja energía, alto paralelismo, alta flexibilidad y alta seguridad[35].

El uso de *FPGAs* o de *SoCs* con *FPGA* (*PSoC*) ofrece la posibilidad de adaptar el diseño del procesador para optimizar el rendimiento o el ancho de banda, llegando a alcanzar el punto óptimo del sistema llamado *Roofline Model*. Existen diversos ejemplos de implementaciones de procesadores específicos implementados sobre *FPGA/PSoC* para la aceleración hardware de algoritmos de *IA* en el "borde". Por ejemplo, en este paper, se

implementa un acelerador hardware de redes *CNN* en una *FPGA* de Xilinx de la serie Virtex-7 para poder ejecutar algoritmos de *IA/ML*[15]. El acelerador hardware se centra en la resolución del problema del rendimiento de cálculo y el ancho de banda de la memoria (*Roofline Model*). Al analizar cuantitativamente los dos factores anteriormente mencionados empleando diferentes técnicas de optimización, se propone una solución con un mejor rendimiento y un consumo menor de recursos en la *FPGA*. Siguiendo los mismos conceptos del artículo nombrado anteriormente, en este artículo se propone un acelerador de redes *CNN* diseñado sobre un *PSoC* Zynq-7000, para la clasificación de imágenes[36].

Estos dos ejemplos ofrecen una idea sobre los niveles de eficiencia que pueden llegar a obtenerse con un análisis cuidadoso de los recursos empleados en la aceleración hardware del algoritmo. A parte de los aceleradores hardware publicados por investigadores, uno de los mayores fabricantes de *FPGA*s ha incluido en sus entornos de diseño librerías software y de *IP Cores* específicos para el diseño de procesadores *IA* en sus arquitecturas (*Vitis AI* de Xilinx)[37].

2.2.6. Conclusiones sobre los Dispositivos Hardware Disponibles

Tras haber valorado las diferentes opciones disponibles en el mercado, se ha optado por la *FPGA* como plataforma para la implementación de algoritmos *IA/ML* para este Trabajo Fin de Máster. Esto se debe principalmente a la posibilidad de aceleración hardware de los algoritmos, ofreciendo una mayor velocidad de computación. Además cumple otro de los objetivos que es la flexibilidad y la capacidad de implementación de diferentes algoritmos sin necesidad de realizar un re-diseño total del *SoC*.

Debido a las ventajas que ofrecen las *FPGAs* en el diseño de *SoCs* e *IP Cores* de algoritmos, es la mejor opción para implementar los algoritmos *IA* propios que son necesarios para realizar un diseño de *SoC* capaz de procesar imágenes hiper-espectrales.

En la Tabla 2.1, se comparan punto a punto las diferentes opciones para valorar el porqué de la elección de la *FPGA* como la mejor manera para la ejecución de algoritmos de *IA/ML*.

Tabla 2.1: Compartiva Hardware

	Velocidad	**Consumo**	**Implementación**	**Flexibilidad**	**Coste**
MCU	Baja	Bajo	Media	Media	Bajo
GPU	Alta	Alto	Media	Media	Medio
ASIC	Alta	Bajo	Baja	Baja	Alto
FPGA	Media	Bajo	Media	Alta	Medio

2.3. Diseño de SoCs para IA en FPGAs y PSoCs

Gracias a la lógica reconfigurable que ofrecen las *FPGA*s y los *PSoCs* (*Programable System-On-Chip*), el diseño de *SoCs* capaces de incorporar aceleradores hardware de *IA* se ha visto favorecido.

Principalmente, el *SoC* se compone de una *CPU*, una memoria (*RAM/ROM*) y una serie de periféricos para poder realizar las comunicaciones del exterior con la *CPU*. Pero en algunos casos, los *SoCs* pueden implementar elementos hardware para realizar procesado de datos capaces de liberar de carga computacional a la *CPU*. Estos elementos hardware son claves en la implementación de algoritmos de *IA* en *SoCs*, ya que, mejoran en rendimiento a algoritmos implementado por software al ofrecer paralelismo, mayor velocidad de procesado, menor carga computacional...

Pero para poder obtener el máximo rendimiento en un *SoC* en el que se vayan a implementar algoritmos de *IA* la aceleración hardware es esencial. Un acelerador hardware es un dispositivo digital hardware, que se encarga de realizar los algoritmos de procesado y computación, para librar al procesador de carga computacional, y, de esta manera, obtener un mayor rendimiento del algoritmo.

Para poder realizar el diseño hardware del acelerador de algoritmos *IA* es necesario emplear diferentes metodologías para obtener un bloque hardware funcional y con un buen rendimiento. Las dos metodologías más extendidas para el diseño de bloques hardware para *FPGAs* son el *High Level Synthesis (HLS)* y *Register Transfer Level (RTL)*, que se van a comparar a continuación.

2.3.1. High Level Synthesis vs Register Transfer Level

En estos últimos años, los diseños de circuitos electrónicos digitales han aumentado de complejidad y tamaño, por lo que se ha tenido que automatizar procesos, debido a la necesidad de acelerar el proceso de diseño.

A más complejo sea el diseño del circuito electrónico, más necesario es aumentar el nivel de abstracción del hardware con el fin de acelerar el proceso. Debido a esta necesidad, nacen las herramientas *HLS* que facilitan el proceso del diseño de circuitos electrónicos digitales.

Las herramientas *HLS* se encargan de traducir y compilar código de alto nivel, como es C, C++ o MATLAB, para obtener una representación de código *RTL*. Los diseños se pueden optimizar automáticamente o se pueden emplear algoritmos específicos para locatar los diferentes módulos, señales de reloj y nets del diseño.

Esta metodología permite un incremento en la productividad, debido a que es necesario emplear un menor tiempo para obtener resultados. Esto se debe a que la cantidad de código que debe escribir el programador es mucho menor, además, se reducen los posibles errores[38].

Por eso, la metodología *HLS*, es muy relevante para el diseño de sistemas embebidos e una *FPGA*. Porque, al aumentar las capas de abstracción hace que sea posible manejar diseños complejos sin mucho esfuerzo. Por eso es una buena opción para el prototipado rápido y una rápida salida al mercado.

En el paper UG998 de Xilinx, se confirma que el tiempo empleado en el diseño de una aplicación empleando herramientas *HLS* es mucho menor que empleando el método tradicional *RTL*, Figura 2.8. En su contra juega, que el rendimiento obtenido de los módulos *HLS* es peor que el que se puede llegar a obtener con un diseño *RTL*.

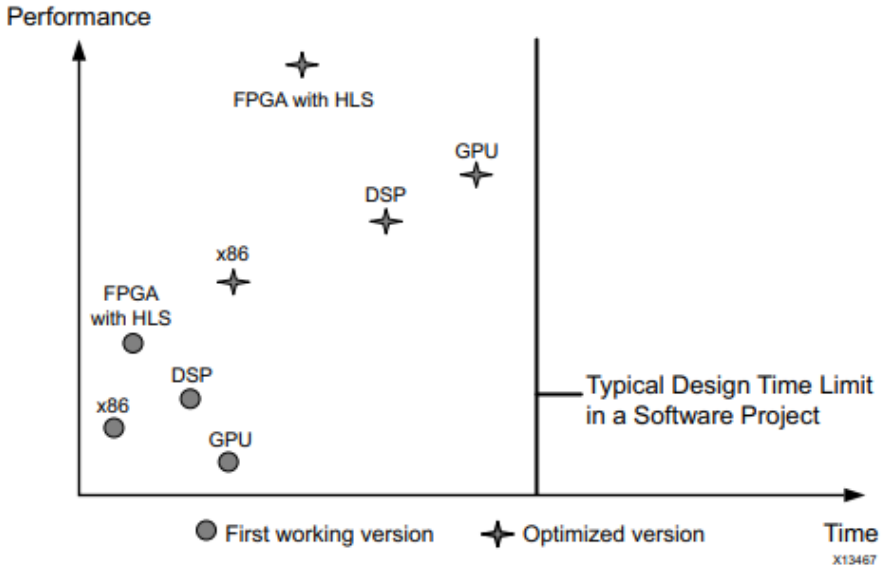

Figura 2.8: Tiempo de Diseño vs Rendimiento con Vivado HLS.

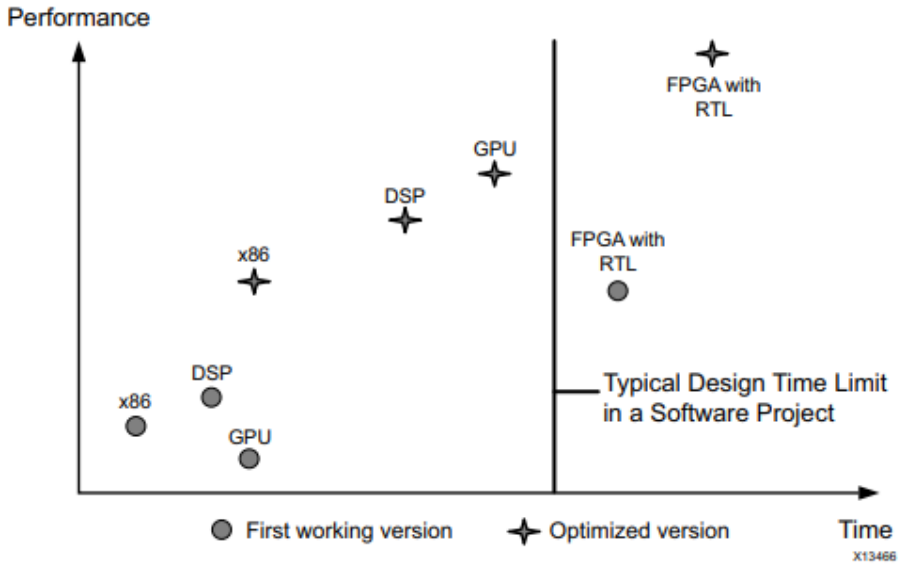

Figura 2.9: Tiempo de Diseño vs Rendimiento con un Diseño RTL

El código *RTL* también es una capa de abstracción sobre el diseño hardware, pero de mucho menos nivel que la *HLS*. En este tipo de metodología se definen las conexiones de las puertas lógicas que conforman el circuito electrónico digital. En este caso el lenguaje de diseño es del tipo *Hardware Description Language (HDL)*, con Verilog o VHDL como los más importantes.

El hardware definido con código *RTL* se puede agrupar en lógica secuencial o lógica combinacional. La lógica secuencial se emplea cuando se quiere tener un mayor control del flujo de datos del diseño, además, las diferentes salidas del diseño hardware no solo dependerán de la entrada, también dependerán de las entradas/salidas previas. La lógica combinacional sirve para diseñar componentes hardware cuya salida solo dependa de las entradas en el mismo instante de tiempo, además el diseño carece de elementos de memoria, ya que esto lo haría ser del tipo lógica secuencial.

Para obtener un diseño *RTL* funcional es necesario que el código Verilog o VHDL contenga una arquitectura de circuito sintetizable, esto vendrá definido por la herramienta, no por los estándares del lenguaje VHDL o Verilog. Si el componente es sintetizable o no se comentara más profundamente en apartados siguientes.

Una desventaja que tiene este tipo de metodología frente a la *HLS* es el contenido del código. Mientras que en un diseño *HLS* el programador se centra en el funcionamiento del sistema completo, en un diseño *RTL*, el programador tiene que centrarse en cómo estructurar el hardware. Es decir, declarar las señales de control, los relojes del sistema,

los registros, memorias...

Por eso es más fácil cometer errores durante la programación en un diseño *RTL* que uno *HLS*. Además de requerir un conocimiento amplio en hardware, ya que un diseño sencillo en *HLS* puede llegar a complicarse a nivel *RTL*.

Sin embargo, la metodología de diseño *RTL*, permite tener el control sobre los elementos hardware y su funcionamiento. Esto permite optimizar su funcionamiento para obtener un mayor rendimiento.

Más concretamente en el apartado de creación de hardware para algoritmos de *ML* los elementos como la memoria, la arquitectura de procesado, o la capacidad de procesado en paralelo son elementos hardware de alto consumo de energía, por lo que su implementación en dispositivos *IoT* es inviable. Pero, en la actualidad está surgiendo la tendencia del diseño de este hardware *ML* en código *RTL*, esto permite reducir el consumo de energía[39].

Estrategias de optimización como la arquitectura *Fully Pipelined* minimizan los accesos a los datos almacenados en la memoria *RAM* dinámica. También la activación de los relojes o de componentes hardware solo cuando son necesarios minimizan el consumo de energía del diseño.

Otra de las estrategias es el ajuste de la precisión de los buses de datos. Es fácil ajustar el tamaño de los buses de datos en código *RTL* gracias al formato coma fija que existe en los estándares del *IEEE* para el código *VHDL*. Además, está comprobado que los algoritmos de *ML* son bastante resistentes a la cuantización de bits, hasta se pueden dar casos extremos en los que una señal interna de un componente se puede ver reducida a una cuantización en binario, o "1.º "0".

2.3.2. Herramientas para la Verificación de IP Cores

Existen diversas aplicaciones y software para la verificación de diseños *HDL* e *IP Cores*, pero siempre se recurre a las mismas herramientas debido a una serie de limitaciones.Las limitaciones puestas por los diseñadores de las *FPGAs* como son Xilinx o Intel, hacen que su software sea necesario para realizar tareas como el *bitstream* del diseño para poder probarlo en hardware. Pero, en la actualidad, las herramientas de síntesis y simulación de diseños *HDL* están creciendo en popularidad. Esto se debe a la incorporación de los estándares de código HDL más recientes[40].

Estos estándares, aunque están aceptados por el Instituto de Ingenieros Eléctricos y Electrónico, *IEEE*, no están implementados en las herramientas de los vendedores de *FPGAs*. Esto hace que sea necesario recurrir a las herramientas de software libre para poder sintetizar y simular los diseños. Destacar que aunque sean sintetizables los diseños, esto no significa que sean implementables, para ello, habrá que traducir el código *HDL* en una netlist con las herramientas de software libre con el fin de que herramientas como Vivado, propiedad de Xilinx, sean capaces de generar un *bitstream*.

A continuación, se explicara de que son capaces las dos herramientas que más se han empleado en este Trabajo Fin de Máster, con el fin de explicar la ventajas que aportan cada una de ella y el porqué es necesario emplearlas.

Vivado

La herramienta principal por excelencia en el diseño de *SoCs* para *FPGAs* de Xilinx ha sido Vivado. Se trata de una herramienta diseñada para la síntesis, implementación, simulación y programación de diseños *HDL* en *FPGAs* de Xilinx. Es una herramienta esencial a la hora de realizar diseños ya que, a parte de disponer de una biblioteca de *IP Cores* bastante amplia, se puede importar *IP Cores* propios que se pueden sintetizar y simular dentro de la propia herramienta.

Aunque Vivado es una herramienta básica a la hora de realizar diseños *HDL* si dispones de una *FPGA* de Xilinx, hay diversos problemas que pueden limitar su uso. El mayor problema que se puede encontrar es que Vivado no es la herramienta más adecuada para los estándares actuales de *VHDL*. Esto se debe a que el soporte ofrecido por Vivado es completo para *VHDL-93* y es capaz de interpretar algunas estructuras de *VHDL-08*, pero para estándares superiores a *VHDL-08* no existe compatibilidad alguna.

El problema de la compatibilidad de Vivado con los nuevos estándares, dificulta el proceso de adaptación del algoritmo de *IA/ML* de software a hardware, ya que muchas de las nuevas sentencias o construcciones del lenguaje *VHDL* facilitan estas tareas.

Estructuras como el tratamiento de datos de coma fija o coma flotante, la declaración de *inputs/outputs* como generic, condicionales con flanco de subida del reloj... La falta de estas estructuras, disponibles en *VHDL-08*, dificultan el diseño de *IP Cores* propios más complejos.

Es por eso que otras herramientas de software libre se están alzando como alternativas

en el diseño de *SoCs*, ya que son capaces de interpretar los nuevos estándares. Estas herramientas además, permiten la verificación del diseño mediante simulaciones.

Herramientas de Software Libre

Herramientas como GHDL, de la que se hablara más adelante, o FuseSoC permiten la creación de *IP Cores* fácilmente incorporando los nuevos estándares, además de permitir la simulación y compilación de los mismos. Pero uno de los problemas que surge es la creación del *bitstream*.

Debido a que solo las herramientas de Xilinx son capaces de generar el *bitstream* para sus *FPGAs*, es necesario emplear tanto Vivado como herramientas de software libre para realizar un pre-procesado de los *IP Cores* antes de implementarlos.

GHDL Es una herramienta de software libre que analiza y simula diseños VHDL, además en las últimas versiones, es capaz de sintetizar diseños VHDL[41]. Principalmente se diferencia de las demás herramientas de análisis y simulación, porque primero realiza una compilación del código VHDL a lenguaje máquina, evitando el uso de un lenguaje intermedio como es C o C++. Esto permite que el código sea más rápido, por lo que el tiempo de análisis es menor.

Pero, la parte más interesante en este proyecto de la herramienta, es la nueva opción de síntesis del código VHDL. Esta opción permite sintetizar y traducir un diseño VHDL, es decir, la herramienta es capaz de generar una *netlist* de VHDL de un diseño con características hasta VHDL-2008.

Una netlist es la descripción del conexionado de un circuito electrónico. Estas se componen de nodos, instancias de componentes o atributos de los mismos. Gracias a esta nueva característica, los diseños de aceleradores hardware realizados en VHDL-2008 se pueden "traducir" a VHDL-93 para su posterior implementación en una *FPGA* o *PSoC*.

La utilización del estándar VHDL-2008 en la creación de *IP Cores* de *IA*, es debido a que las características del lenguaje VHDL-93 dificultan la tarea de diseño en código *RTL* del algoritmo. Además, en VHDL-2008, se puede llegar a generar un *IP Core* totalmente parametrizable, por lo que se puede adaptar a múltiples arquitecturas, ya que se pueden modificar los parámetros de la red neuronal, como son las capas, nodos y parámetros fácilmente.

FUSESOC Otra herramienta muy empleada para el desarrollo de SoCs es FuseSoC[42]. Se trata, al igual que GHDL, de una herramienta de software libre disponible solo para sistemas UNIX. FuseSoC tiene el propósito de facilitar el proceso de desarrollo e integración de diseños hardware.

Se basa en aumentar la reutilización de *IP Cores* en la creación y simulación de *SoCs*. No se encarga de realizar la síntesis o la implementación, si no que su propósito es realizar un fichero formato CAPI2. En este fichero se hace referencia a todos los ficheros que componen el proyecto y gracias a diferentes variables se le puede indicar que sintetizador debe usar, o la FPGA para la que generar el bitstream.

Actualmente, tiene soporte para lanzar diferentes simuladores, sintetizadores, como son Icarus Verilog, GHDL o Vivado.

Esta herramienta facilita el proceso diseño ya que se pueden crear librerías de IP Cores, establecer parámetros para modificar fácilmente el diseño.

2.3.3. Diseño de SoCs en FPGAs

Gracias a la que las *FPGAs* son dispositivos de lógica reconfigurable, esto permite la implementación de *SoCs* con una amplia variedad de procesadores. Esto permite emplear diferentes arquitecturas de procesador como puede ser la arquitectura basada en instrucciones *Reduced Instruction Set Computer (RISC)*, como la empleada en los procesadores de ARM.

Los microprocesadores que se implementaran sobre *FPGAs* son del tipo *Softcore Microprocessor*. Estos procesadores se caracterizan porque se implementan utilizando la lógica programable. La gran mayoría de estos procesadores *Soft* solo implementan un solo núcleo de procesador, por lo que la capacidad de paralelismo y computación se ve limitada frente a sistemas multi-núcleo. Esto se debe al tamaño de la *FPGA*, en *FPGAs* de gama baja, solo se podrá implementar sistemas que utilicen un solo procesador, mientras que en *FPGAs* de gama más alta, cuyo tamaño es mayor, se podrá llegar a implementar sistemas de procesado multi-núcleo.

Concretamente para las *FPGAs* de Xilinx, existen varias alternativas de *CPU Soft Core* en el mercado. La gran mayoría de los procesadores se basan en procesadores basados en conjuntos de instrucciones *RISC*, como pueden ser los procesadores de ARM o Xilinx (Microblaze). En los últimos años, los procesadores RISC-V, también basados en conjuntos de instrucciones *RISC*, se están popularizando como alternativa a los ARM, ya

que ofrecen características similares en hardware que implementan (*Floating Point Unit (FPU)* o multiplicador Hardware) y, además, tienen licencia *Open-Source*. Por lo que no es necesario pagar por su uso en diferentes diseños.

Procesador ARM Cortex-M3

El microprocesador *Cortex-M3*, es un procesador de 32-bit diseñado por ARM basado en una arquitectura ARMV7-M. Se trata de un procesador diseñado para un alto rendimiento, bajo coste y eficiente para plataformas *IoT*. Es procesador no dispone de *FPU*, por lo que tomara más ciclos de reloj realizar cualquier operación con número de tipo float, que será un elemento a tener en cuenta a la hora de evaluar el rendimiento para el código generador de cubo hiper-espectrales.

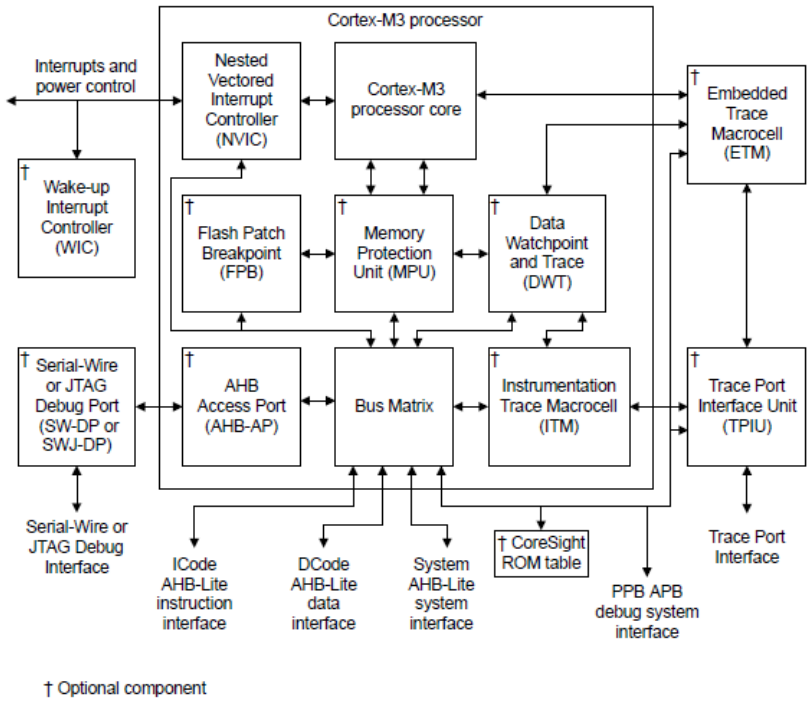

Figura 2.10: Esquema Hardware del Cortex-M3

Lo que sí que incluye este microprocesador, es un multiplicador de 32x32-bit, que, según el *datasheet*, se asegura que las operaciones de multiplicación se harán en un solo ciclo[43].

Procesador Xilinx Microblaze

Al igual que con el microprocesador de ARM, se trata de un *Soft Core* basado en instrucciones *RISC* y arquitectura Hardvard. Una de las ventajas que tiene este microprocesador frente al Cortex-M3 de ARM, es la diversidad de configuraciones disponibles. Este procesador es muy configurable para adaptarlo a las necesidades del sistema[44].

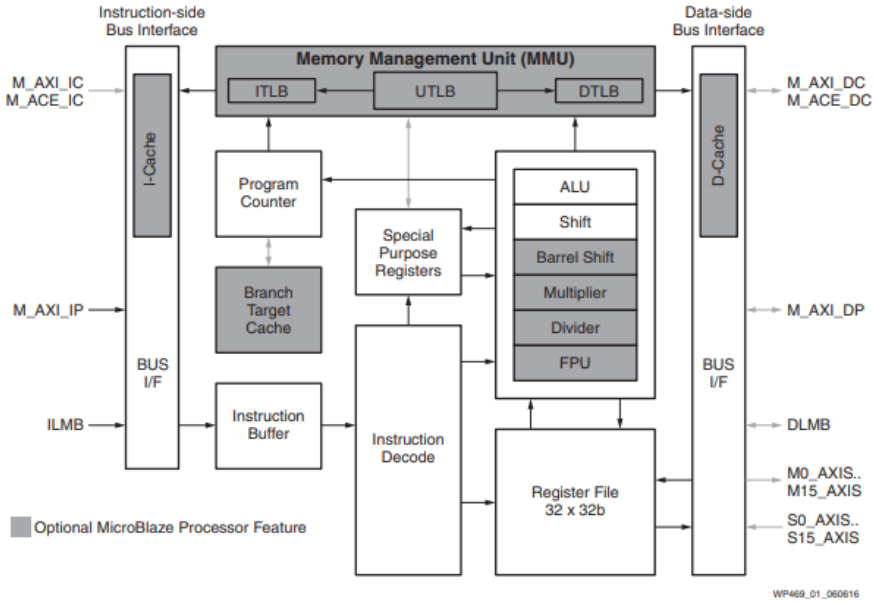

Figura 2.11: Esquema Hardware del Xilinx Microblaze

Este microprocesador dispone de una memoria cache tanto para instrucciones como para datos, *FPU*, multiplicador, divisor... Estos elementos hardware incluidos en el microprocesador son claves para aumentar el rendimiento, ya que es necesario un menor número de instrucciones de reloj.

Xilinx Zynq-7000 APSoC

El Zynq-7000 AP SoC, a diferencia de los microprocesadores anteriormente, se trata de un *SoC* implementado en silicio, es decir, no se trata de un *Soft-Core* implementado en una *FPGA*, si no que se encuentra un *SoC* con el microprocesador implementado en silicio además de una *FPGA*[45].

El *SoC* se compone básicamente de dos partes: *PS* y *PL*. En la parte *Processing System (PS)*, se encuentra el microprocesador ARM, concretamente un *Dual-Core* ARM Cortex-A9 y una serie de periféricos como es la UART o la memoria *DDR*. El procesador viene

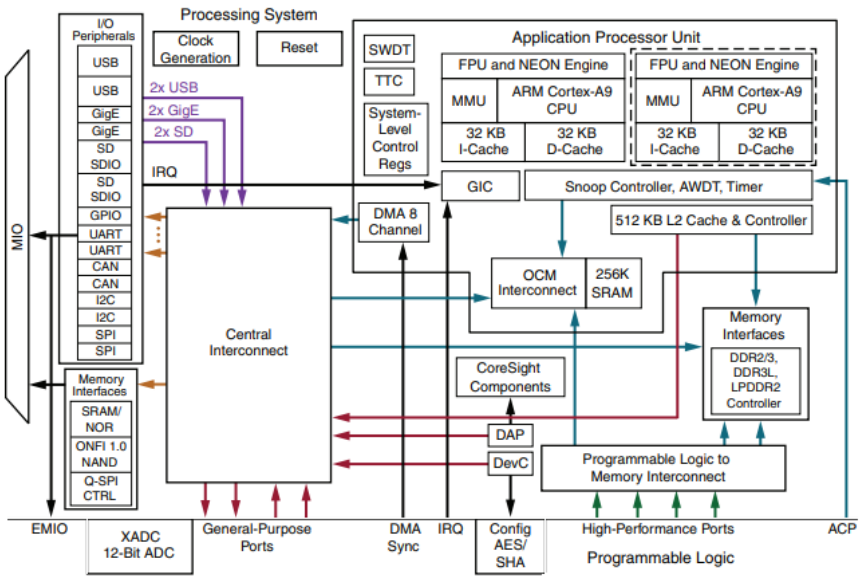

Figura 2.12: Esquema Hardware del Xilinx Zynq-7000 APSoC

implementado con cache de datos e instrucciones, memoria *ROM*, memoria *RAM*, controladores de acceso directo a memoria e interfaces de entrada/salida para periféricos. La parte *Programable Logic (PL)*, se basa en una *FPGA*.

Una de las características a tener en cuenta en este tipo de *SoCs*, es que se centra en una arquitectura sobre la parte *PS*. Siempre el primer elemento en arrancar en el *SoC* es el procesador, y, posteriormente se configura la parte *PL*. La parte *PS* puede trabajar sin necesidad de configurar la parte *PL*, pero no al revés.

Al tratarse de un procesador *Dual-Core*, puede funcionar de diversos modos. Es posible que solo esté funcionando solo uno de los dos núcleos. La otra opción es que ambos núcleos estén funcionando al mismo tiempo, ya sea de manera simétrica o asimétrica. En el caso de funcionamiento simétrico se aprovechan los dos núcleos para trabajar sobre solo un mismo sistema. De manera asimétrica cada uno de los núcleos funciona con un sistema diferente, ya sea *Standalone* o un sistema operativo.

Una de las características más importantes que incluye el Zynq-7000 *APSoC* es el módulo NEON. Se trata de un método de optimización de velocidad de ejecución, además, ayuda a incrementar el rendimiento y puede ser crucial para bajar el consumo de potencia.

NEON se basa en técnicas *Single Instrucction - Multiple Data (SIMD)*, que es capaz de procesar una gran cantidad de datos, almacenados en unos registros específicos, en paralelo empleando tan solo una instrucción. Si el código es paralelizable con opera-

ciones vectoriales repetitivas, la optimización puede aumentar el rendimiento de manera significante. Particularmente, es muy útil en procesado de señales digitales o algoritmos multimedia como son *FFTs*, multiplicación de matrices, procesado de video, imagen...

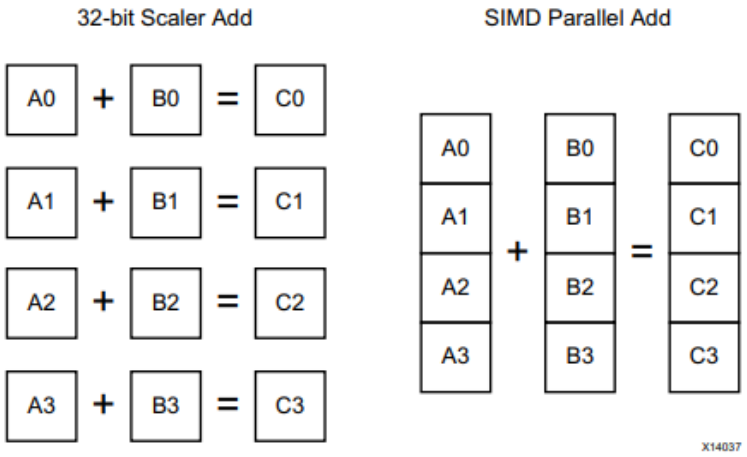

Figura 2.13: Ejemplo de Suma de 32-bit con NEON[3]

En el caso del microprocesador de 32-bit empleado en el Zynq-7000, es ineficiente realizar multiplicaciones de 8-bit o 16-bit, ya que son necesarias más instrucciones para poder manejar el overflow por cómo está diseñada la *ALU*, que es específica para 32-bit.

En la Imagen 2.13 se puede ver un ejemplo de la optimización realizada por el módulo NEON. En este caso la suma de de varios operandos de 32-bit, se puede agrupar como una única suma de vectores, simplificando una serie de operaciones a una única instrucción

En definitiva, con la optimización empleando el módulo NEON, se puede conseguir hasta un incremento de x4 o x8 en algoritmos que empleen el *Digital Signal Processor (DSP)*, un incremento del 60-150 % en el procesado de vídeos, una mayor eficiencia y un menor consumo de potencia[3].

Otra de las ventajas que aporta es la óptima utilización de la memoria cache. Ya que el módulo NEON se encarga de trabajar con grandes cantidades de datos. Minimizar el número de accesos a datos desde la memoria externa es clave para obtener un rendimiento alto. El módulo NEON incluye instrucciones que soportan el entrelazado y des-entrelazado que, dependiendo del caso, pueden suponer un aumento significativo en el rendimiento. Si se usan de manera correctamente, se pueden llegar a guardar/almacenar múltiples registros desde la memoria.

Una desventaja que presenta el módulo NEON es el orden de los factores en la computación. En el caso de números enteros el orden no importa, ya que siempre se produce el

mismo resultado. Pero, en el caso de emplear número de coma flotante, el orden de los factores si que puede afectar debido a la precisión del código. Por ello, hay que prestar atención al resultado para comprobar si es el deseado.

Aunque, el módulo ofrezca un acceso a los datos no alineados en memoria. Si se quiere obtener el máximo rendimiento posible es necesario ser cuidadoso. Los datos en memoria deben estar bien distribuidos y alineados.

Por estas razones, emplear un *SoC* que contenga un módulo NEON en el procesado de imágenes hiper-espectrales es un gran beneficio. Esto se debe a que un gran peso de la computación se lo lleva el filtrado de la imagen, por lo que se podrán paralelizar las operaciones del filtro obteniéndose un mayor rendimiento.

Procesadores RISC-V

Los RISC-V son una serie de procesadores que se basan en una arquitectura ISA, más concretamente en la arquitectura RISC, un conjunto de instrucciones reducidas. Se caracteriza por ser hardware libre y abierto, lo cual facilita su uso al no tener royalties. Esto facilita el diseño, la fabricación y la venta de chips basados en esta arquitectura. El set de instrucciones que se utilizan en los RISC-V se caracteriza por su pequeña longitud, su rapidez y su bajo consumo.

Actualmente, hay disponibles muchas versiones de estos procesadores y se puede obtener la información de la página de RISC-V[46]. En esta página se puede encontrar información sobre los Cores y *SoCs* que se están realizando tanto en grupos de investigación como en empresas.

En este caso, que lo que se quiere realizar es la implementación del *IP Core* sobre un bus AXI, interfaz empleada por Xilinx, se ha realizado una búsqueda de los diferentes *IP Cores* y herramientas disponibles por parte de los fabricantes para poder integrar el *Core* con el bus en la *FPGA*. A continuación se muestra una recopilación de las empresas que más avanzadas están en el tema:

- SiFive[47]: Se basan en diseñar IP Cores personalizables, estos pueden estar agrupados en ‘alto rendimiento’, ‘mayor eficiencia’ o ‘Linux’ [5]. Un problema de estos Cores es que utilizan su propio bus, el TileLink, por lo que hay que añadir un puente de TileLink a AXI Bridge para poder conectar el *Core* a un bus AXI. Concretamente, ya hay interfaces creadas para poder conectar correctamente el Core con el bus.

Este Core en concreto es de 32 bits. Por lo que los microprocesadores de SiFive sí se podrían implementar en una FPGA basada en un SoC de bus AXI.

- VexRiscV[48]: Es un procesador configurable de 32 bits escrito en SpinalHDL [7]. Con esta configuración se puede obtener un procesador de 494 LUTs y 505 FF hasta 2530 LUTs y 2013 FF, este último sería para correr Linux. También están diseñados para correr en FPGAs, concretamente en la serie Artix-7 el Core más simple puede llegar a trabajar a 233MHz. Sí que dispone de comunicación para bus AXI, pero en su contra juega que está escrito en otro lenguaje diferente a los que puede interpretar Vivado.

- Bluespec[49]: Es otro suministrador de Cores de RISC-V, además de herramientas para trabajar con ellos. Actualmente lo que emplean es Open Source y también permiten la modificación de los Cores. El Core Piccolo de Bluespec podría servir para trabajar en un SoC con bus AXI ya que incluye una interfaz AXI4-Lite. Este Core se basa en un set de instrucciones RV32IM.

- Pulp[50]: Al igual que Bluespec y SiFive son diseñadores de Cores. Según la página Web disponen de periféricos para realizar las interconexiones del Core con buses AXI y son compatibles con FPGAs de Xilinx. En la actualidad se dedican más al diseño de SoCs, dejando la parte de Cores a lowRISC.

- LowRISC[51]: Es una empresa que realiza diseños sobre RISC-V, estos colaboran directamente con Pulp. En su página de GitHub disponen de tutóralos para la implementación de Cores y concretamente uno para la Arty A7 con el Core Ibex, que se trata de un Core basado en RISC-V de 32 bits con un pipeline de 2 etapas, RV32IMC.

3. CAPÍTULO

Descripción de la Propuesta

La propuesta de este Trabajo Fin de Máster es evaluar el rendimiento de diferentes *IPs* de microprocesadores para el procesado de imágenes hiper-espectrales, realizando una infraestructura de *SoC* genérica para el ámbito de la aplicación. Se pide evaluar también el acceso y transferencia de datos con *Direct Memory Access (DMA)*.

Se ha comenzado por realizar una revisión de las actuales herramientas EDA para desarrollo de sistemas sobre *FPGA/PSoC*, tanto comerciales como de código abierto, así como, de la disponibilidad de núcleos *IP* exentos de pago por licencia para el procesamiento de datos relacionados con la *IA*.

Se han evaluado el rendimiento de diferentes microprocesadores ejecutando un código de generación de cubos hiper-espectrales para cámaras tipo snapshot. A partir de esto se ha podido construir un análisis comparativo para determinar las mejores opciones a futuro para la aplicación. También, se han estudiado diferentes formas de acceso a memoria, optimización de código y de aceleración por hardware del código.

En la segunda parte se ha analizado la forma óptima de ralizar la transferencia de datos entre memoria externa, microprocesador e *IP Cores* usando *DMAs*. Esto servirá para realizar el posterior análisis del cubo hiper-espectral con coprocesadores hardware integrados en la *FPGA*. Todavía no se dispone de los coprocesadores del sistema de procesamiento final, por lo que se han utilizado *FIFOs* simple en la partición lógica del dispositivo para emular la transferencia de datos de un *IP* genérico para estudiar el rendimiento de las *DMAs*.

Para realizar los diferentes test los diseños de *SoC* genéricos se dispone de la base de datos

desarrollada por el grupo GDED (Imágenes hiper-espectrales tomadas con una cámara snapshot de 25 bandas). Se han realizado diferentes test de rendimiento y consumo para poder valorará la aplicabilidad de los diferentes sistemas desarrollados.

Por último se han podido extraer conclusiones sobre las estudios realizados para en un futuro realizar el diseño de un *SoC* capaz de generar y analizar imágenes hiper-espectrales para una aplicación implementada dentro de un *ADAS*.

3.1. Metodología Experimental

Para realizar los experimentos de una manera correcta y equitativa se ha establecido un orden de pasos a seguir. En el proceso de caracterización del código software de preprocesado de imágenes hiper-espectrales se ha evaluado el tiempo de ejecución de cada una de las tareas que lo componen y, para ello, se han seguido los siguientes pasos.

Lo primero que se ha realizado ha sido la carga en la memoria externa de la imagen en crudo, obtenida de una cámara snapshot, que se va a procesar y de las imágenes de referencia de blanco y negro. Esto se ha realizado en el proceso de depurado, se ha creado un archivo hexadecimal con las imágenes y se ha establecido como fichero de inicialización de la memoria externa.

El segundo paso ha sido ejecutar el código compilado sobre el microprocesador y observar que la ejecución paso a paso era correcta. Por último se ha medido el tiempo de ejecución de cada una de las tareas del código de preprocesado de imágenes hiper-espectrales con el *IP Core* Timer. El tiempo de ejecución se ha medido en número de ciclos de reloj, y Una vez se ha acabado de ejecutar el código se ha procedido a convertir el número de ciclos de reloj a segundos para poder comparar los diferentes microprocesadores.

El proceso de caracterización de las *DMAs* ha sido muy similar al anterior. En este caso solo ha sido necesario cargar la imagen hiper-espectral de 25 bandas en la memoria externa. También se ha optado por cargar la imagen con un fichero hexadecimal que se carga al iniciar la depuración.

El segundo paso ha sido enviar el paquete más grande que se quiere transferir, en este caso se corresponde a el cubo hiper-espectral al completo, y comprobar que la *DMA* transfiere los datos de manera correcta.

Por último, se ha realizado la caracterización realizando transferencias de distintos tamaños de paquetes, comenzando por el paquete más grande, una imagen hiper-espectral

de 25 bandas, hasta el más pequeño posible para cada una de las *DMAs*, 2 píxeles cada paquete. En todas las pruebas realizadas para esta caracterización se ha medido el tiempo necesario para realizar una transferencia de una imagen hiper-espectral de 25 bandas completa, ya sea sólo en un paquete o en múltiples paquetes de datos.

3.2. Arquitecturas de SoCs para la experimentación

Para realizar la evaluación de los diferentes microprocesadores disponibles se ha realizado un diseño de *SoC* genérico. Se ha optado por un diseño sencillo y similar en todos los casos para poder compararlos, en la medida de lo posible, en condiciones equivalentes. El *SoC*, Figura 3.1 consta de un microprocesador, un módulo *IP* para la medida de tiempos de ejecución, la memoria externa para el almacenamiento de las imágenes, y una UART para intercambiar información de la ejecución al exterior.

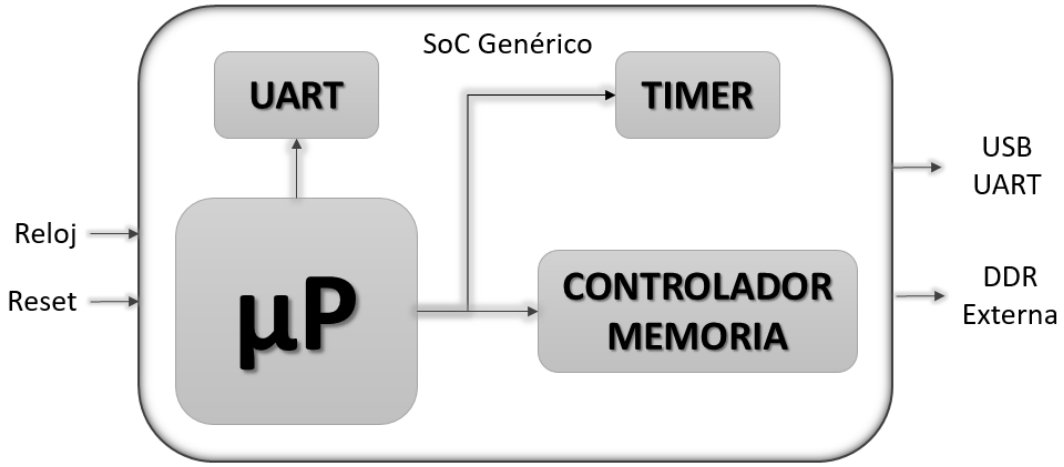

Figura 3.1: Esquema del Diseño del SoC Genérico

En el caso de testeo de la diferentes *DMAs* Hardware se ha añadido un módulo IP de *DMA* y una FIFO al diseño anterior, Figura 3.2. La idea de este *SoC* es testear las diferentes opciones de *DMAs* en igualdad de condiciones.

La FIFO emula los buffer de entrada/salida de un *IP* de coprocesamiento *IA* genérica . Aunque el rendimiento y el ancho de banda no es directamente comparable a lo que se puede llegar a obtener con la FIFO, la estructura de organización de los paquetes que transmite la *DMA* y el cómo se pueden organizar los datos en la memoria para poder

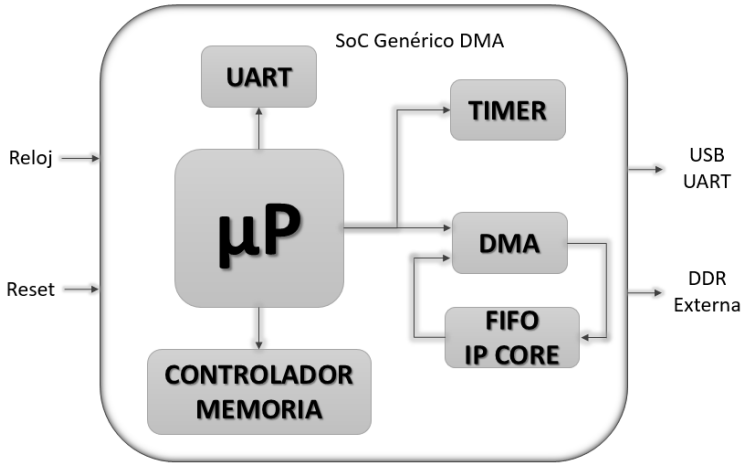

Figura 3.2: Esquema del Diseño del SoC Genérico con DMA

acceder a ellos fácilmente, es equivalente a la que se va a emplear en un futuro para la versión futura del *SoC*.

4. CAPÍTULO

Desarrollo del Proyecto

En este capítulo se explicará detalladamente el orden y tareas que se han seguido para obtener los resultados de este proyecto. Primeramente se planteo unas tareas iniciales sencillas como era realizar una revisión y estudio del arte de las opciones disponibles, tanto *IP Cores* de microprocesadores como de herramientas software con el fin de, al final del proyecto, haber sido capaz de desarrollar un *SoC* genérico para evaluar el rendimiento de generación y transferencia de imágenes hiper-espectrales.

4.1. Descripción de Tareas

La tarea principal de este Trabajo Fin de Máster es evaluar los diferentes *IP Cores* de microprocesadores disponibles para realizar un diseño básico básico de *SoC* capaz de procesar imágenes hiper-espectrales. Para ello se ha realizado diferentes pruebas a los microprocesadores para evaluar en que medida son adecuados para la ejecución de este tipo de tareas.

Una vez se han analizado las características técnicas de los diferentes microprocesadores, se ha procedido ha realizar diferentes ejemplos sencillos de implementación de *SoCs* en una *FPGA* de rango medio. El fin de esta tarea ha sido familiarizarse con el proceso de diseño y las herramientas necesarias para la implementación de *SoCs*.

Se ha puesto especial interés en el desarrollo de un *SoC* basado en una arquitectura RISC-V, ya que esta es una opción de futuro que cobra fuerza rápidamente como alternativa a

otras *IP* licenciadas. Pero, debido a una serie de inconvenientes que se explicaran más adelante no se ha podido obtener un *SoC* funcional. Ya

La primera prueba que se ha realizado ha sido estudiar el rendimiento que ofrece, tiempo y consumo, en la generación un cubo hiper-espectral de 25 bandas a partir de la información proporcionada por una cámara tipo snapshot. Se ha estudiado como realizar el procesamiento software lo más veloz posible haciendo uso de los recursos disponibles en cada uno de los microprocesadores utilizados: *FPU*, multiplicador hardware, *SIMD*...

Destacar que en este punto. ha sido necesario modificar el diseño del *SoC* que incluye el Xilinx Microblaze para obtener el máximo rendimiento posible. Normalmente para microprocesadores *Soft-Core* se emplea el reloj generado por el controlador de memoria como reloj principal para el microprocesador y los periféricos. En este caso, ya que el reloj máximo que puede generar el controlador de memoria es de 83.33MHz, esto se debe a que el componente hardware solo permite 1/4 de la frecuencia máxima. Por esta razón, ha sido necesario añadir otro reloj encargado de controlar el microprocesador y los periféricos excepto la memoria *DDR*.

También como el AXI Timer es de 32-bit, ha sido necesario reducir la frecuencia de conteo hasta los 5MHz, debido a los elevados tiempos de ejecución de código testeado.

Se han analizado qué fases de la generación del cubo hiper-espectral son las que más recursos consumen para poder realizar una comparación entre los microprocesadores con el fin de obtener resultados claros y precisos para futuras aplicaciones.

Se ha estudiado también el tiempo de acceso a memoria *DDR*, movimiento de datos entre el microprocesador, y la optimización del número de accesos, y se han ensayados técnicas de reutilización de datos utilizando memoria local con el fin de optimizar el rendimiento de algunas tareas, en particular, de los filtros espaciales. Para ello, se ha implementado en el mismo *SoC* empleado en tareas anteriores una *DMA*. De esta manera se ha podido comparar las diferentes opciones de *DMA* disponibles para disponer de datos con el fin de tomar una decisión en futuras fases del proyecto sobre que hardware emplear.

Esta tarea será clave en fases futuras del desarrollo del *SoC* final debido a que el rendimiento final del sistema dependerá fuertemente del ancho de banda que se pueda obtener en la transmisión de datos entre el dispositivo y la memoria externa.

Por último se han obtenido conclusiones sobre todos los test realizados para facilitar el proceso de selección de componentes hardware en futuros diseños de la aplicación.

4.2. Descripción de Equipo

El equipo utilizado para este Trabajo Fin de Máster han sido tanto elementos hardware como software. Se han empleado diferentes *FPGAs* de Xilinx y el software necesario para poder implementar, programar y verificar los diferentes *SoCs* diseñados para evaluar los diferentes *IP* de microprocesadores y de *DMAs*.

4.2.1. Hardware Empleado

Las *FPGAs* empleadas son los kit de evaluación de la Arty A7-100T 4.1 y la MicroZed 4.2 . La Arty A7-100T se basa en una *FPGA* de la serie Artix-7. Se trata de una *FPGA* bastante flexible, es decir se puede utilizar en muchos campos, de bajo consumo y bajo coste. Además están diseñadas para aplicaciones en las que sea necesario un ancho de banda alto para los *DSP* [52].

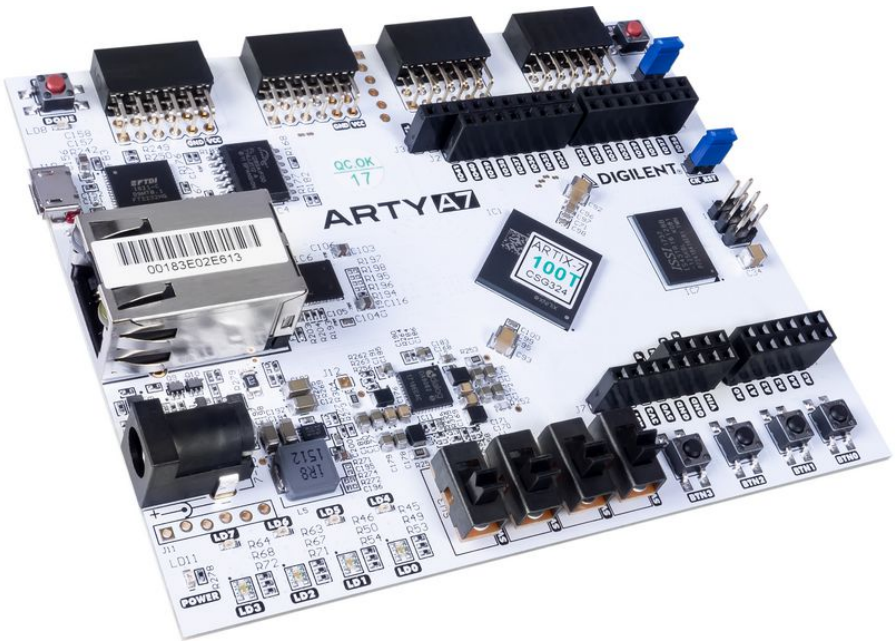

Figura 4.1: Kit de Evaluación de la Arty A7 100T

La MicroZed 7z020 es un *APSoC*, básicamente se puede definir como un *SoC* y una *FPGA* en el mismo circuito integrado. Esta placa desarrollo se basa en un Zynq-7000 y debido a su diseño es capaz de trabajar directamente con software o puede trabajar como un *System-On-Module* (*SOM*). El circuito integrado principal se divide en dos partes: el sistema de procesamiento, *PS*, y la parte de lógica programable, *PL*.

En el parte *PS*, se puede encontrar un *SoC* en silicio con diversos periféricos ya implementados: USB-UART, memoria *DDR*, QSPI, GPIOs... Se basa en un procesador dual de ARM: Cortex-A9 Dual Core. La parte *PL*, es la *FPGA* incluida en el microchip junto al *SoC*, esta parte se puede comunicar con el *SoC*, pero para poder funcionar es necesario primero arrancar la parte *PS* del microchip antes que la *PL*.

Este microprocesador es capaz de llegar a funcionar hasta 667Mhz, además, incluye multiplicador y unidad de coma flotante por hardware, elementos clave en nuestro diseño. También, incluye un módulo NEON para realizar cálculos vectoriales y optimizaciones en el acceso a memoria *DDR*.

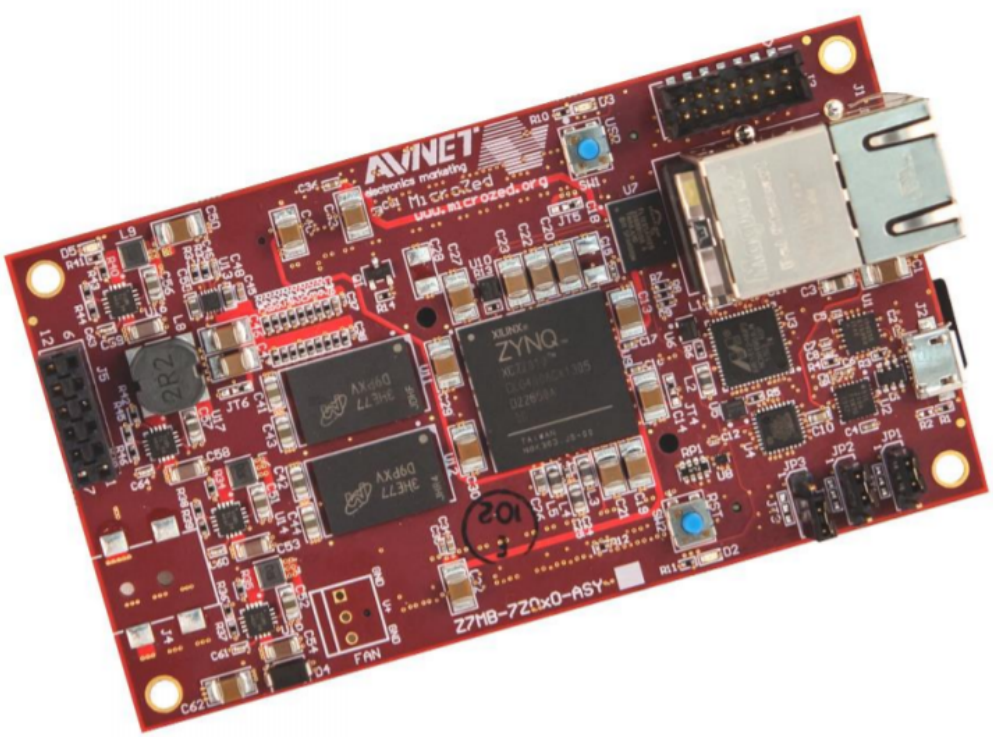

Figura 4.2: Kit de Evaluación de la MicroZed 7z020

4.2.2. Software Empleado

En cuanto al software se han empleado principalmente las herramientas proporcionadas por Xilinx como es Vivado y Xilinx SDK. Vivado es la herramienta por excelencia para el diseño y síntesis de diseño *HDL* para *FPGAs* de Xilinx. Esta herramienta permite realizar el diseño de *SoCs* y de *IP Cores* propios para su implementación en *FPGAs* de Xilinx.

Xilinx SDK, es el programa encargado de compilar el software para el diseño realizado

en Vivado, también se encarga de realizar el *BSP* del microprocesador. El *Board Support Package (BSP)* se puede definir como la capa de software que contiene los drivers específicos del hardware, además de otras rutinas, que permiten correr un sistema operativo o un software sobre el microprocesador.

Otra herramienta empleada es el Keil uVision de ARM, que se ha empleado para realizar el software a ejecutar sobre el Cortex-M3. Se trata de un SDK, bastante diferente a la mayoría de SDKs disponibles, ya que no se basa en Eclipse. Este entorno emplea una cadena de compilación diferente al SDK de Xilinx, ya que se emplea la cadena de compilación ARM GNU, mientras que los SDK basados en Eclipse, la cadena de compilación es GNU GCC.

Dependiendo de la cadena de compilación el lenguaje empleado en el software puede variar, lo que hace que haya que portar el código para poder cambiar de cadena de compilación sin afectar al resultado final.

Como programas de código abierto se ha empleado GHDL. Este programa se ha empleado para generar una netlist de VHDL-93 a partir de IP Cores escritos en VHDL-08. Es necesario emplear este programa si se quiere implementar *IP Cores* escritos en los estándares más recientes en Vivado. También destacar que con este programa se pueden analizar y sintetizar *IP Cores* aunque la sintetización de los *SoCs* se ha realizado empleando Vivado.

4.3. Procedimientos

La primera fase del proyecto se ha basado en realizar un estudio de los microprocesadores disponibles, Microblaze, Cortex-M3, RISC-V y el APSoC Zynq-7000, para poder comprender sus características hardware: disponibilidad de FPU, disponibilidad de multiplicador hardware, módulos SIMD, capacidad de la memoria local. Se han realizado múltiples ejemplos para poder realizar una correcta implementación de los microprocesadores y los diferentes periféricos necesarios para realizar un estudio del tiempo de ejecución del código software.

4.3.1. Desarrollo de SoCs para Experimentación

En el caso del Cortex-M3 este paso ha requerido de un proceso de aprendizaje algo mayor, ya que hay que emplear herramientas externas además de scripts para poder combinar

el bitstream con el código software, generando un bitstream con el software ya escrito en la memoria local del Cortex-M3. Esto se debe a que el *IP Core* está diseñado para ser programado a través del programa propio de ARM, Keil uVision, pero lo que es el diseño del *SoC* y la generación de las funciones correspondientes al *BSP* se realiza con las herramientas habituales de Xilinx Vivado y Xilinx SDK.

En el caso de los microprocesadores RISC-V, el desarrollo de SoCs es una tarea complicada, por eso, se ha dedicado una sección específica para describir el trabajo realizado con los microprocesadores basados en esta arquitectura hardware. En este caso no se ha podido obtener un SoC funcional, por lo que no se ha tenido en cuenta esta opción en las pruebas realizadas en el proyecto.

La segunda fase del proyecto ha sido realizar el diseño del *SoC* básico que permita caracterizar la ejecución del procesamiento de un cubo hiper-espectral, tanto para el Cortex-M3 y el Microblaze. El *SoC* se compone de una UART, de un controlador de memoria *DDR*, y un AXI Timer para calcular el tiempo de ejecución. Para el diseño final se podría eliminar la UART y el AXI Timer, con el fin de simplificar el *SoC*, pero son útiles a la hora de realizar *profilings* de tiempo y recibir *feedback*.

En el caso del ARM Cortex-M3 se ha realizado un diseño de bloques muy simple para poder comunicar con la memoria DDR externa de la FPGA. El diseño completo se puede ver en el Anexo, Figura A.1, pero a continuación se explicaran las particularidades del diseño. Respecto al microcontrolador, Figura 4.3, esta configurado para ejecutar el código desde la memoria interna del microprocesador, ya que, también tiene la opción de ejecutar código desde memoria externa. Además, incluye unos puertos de debug *JTAG* que se han rutado a un PMOD de la Arty A7.

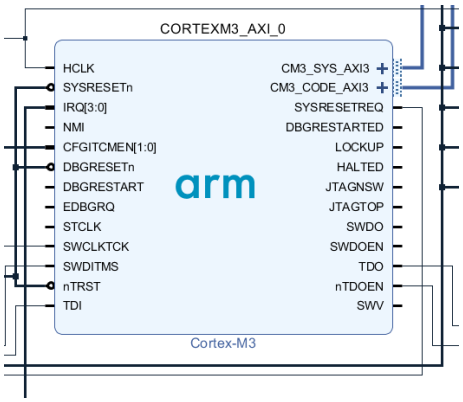

Figura 4.3: Detalle del IP Core Cortex-M3

Respecto a los relojes empleados en el diseño destacar que son los recomendados en las

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)		Drives
		Requested	Actual	Requested	Actual	Requested	Actual	
clk_out1	clk_out1	166.667	166.667	0.000	0.000	50.000	50.0	BUFG
clk_out2	clk_out2	200.000	200.000	0.000	0.000	50.000	50.0	BUFG

Figura 4.4: Detalle de los Relojes del Diseño de Bloques

guías del controlador de memoria MIG-7 y del propio microprocesador, Figura 4.4. La frecuencia máxima que se puede obtener del controlador de memoria es de 83.33MHz, esto se debe al tipo de memoria que viene definido en la Arty-A7, porque el *"Speed Grade"* esta fijado a 1:4 y la frecuencia máxima de la *DDR* externa es 333.33MHz.

Los periféricos empleados son *IP Cores* de Xilinx, Figura 4.5. El AXI Timer esta configurado solo para contar ciclos de reloj y la UART y la memoria DDR están configuradas gracias a Vivado. En algunos kits de desarrollo, Vivado ofrece la opción de poder arrastrar los componentes hardware disponibles desde la pestaña *"Board"*, Figura 4.6. Esto facilita mucho el proceso de implementación, sobretodo a la hora de implementar controladores de memoria, ya que el proceso de implementación de este tipo de componentes es muy complicado.

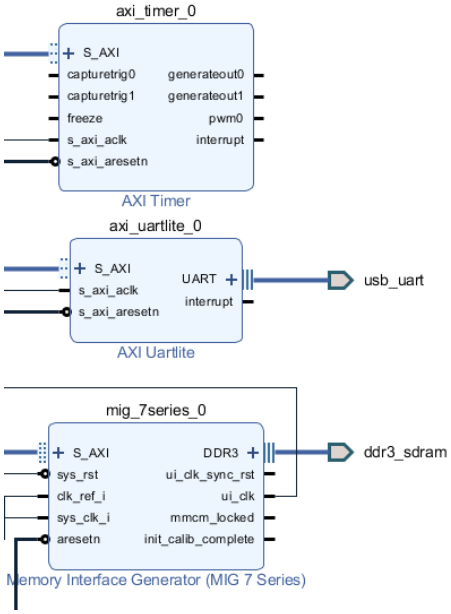

Figura 4.5: Periféricos del Diseño de Bloques

En el caso del Xilinx Microblaze se ha realizado un diseño muy similar al del Cortex-M3. Los periféricos y el controlador de memoria empleado es el mismo, ya que se emplea la misma FPGA, por este motivo, la frecuencia máxima que se puede obtener del controlador es de 83.33MHz.

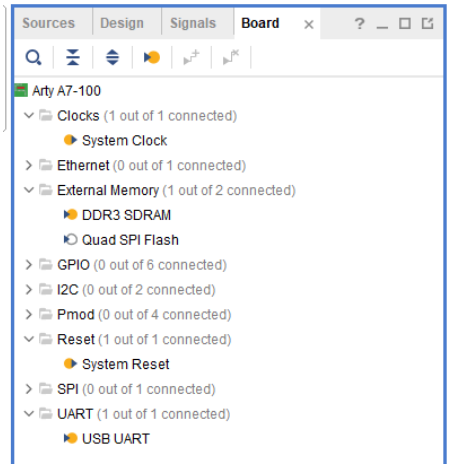

Figura 4.6: Pestaña Board de Vivado

Para solucionar este problema de baja frecuencia de operación se ha optado por emplear más relojes en el diseño, para poder trabajar a 200MHz, Figura 4.7. También, para poder medir de manera correcta el tiempo de ejecución, se ha decidido bajar la frecuencia del AXI Timer a 5MHz aunque se pierda algo de resolución.

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)		Drives
		Requested	Actual	Requested	Actual	Requested	Actual	
clk_out1	clk_out1	166.667	166.667	0.000	0.000	50.000	50.0	BUFG
clk_out2	clk_out2	200.000	201.389	0.000	0.000	50.000	50.0	BUFG
clk_out3	clk_out3	5.000	4.993	0.000	0.000	50.000	50.0	BUFG
clk_out4	clk_out4	220.000	201.389	0.000	0.000	50.000	50.0	BUFG

Figura 4.7: Detalle de los Relojes del Diseño de Bloques para el Xilinx Microblaze

Para el Cortex-A9, al venir ya el *SoC* definido en silicio, no es necesario realizar el diseño, pero sí la configuración de los periféricos, tanto de la memoria *DDR* como de la UART y la frecuencia de reloj del *SoC*, Figura4.8.

En este caso, las frecuencias de reloj sí que son muy superiores a los *SoCs* implementados sobre la Arty-A7. Se ha empleado un reloj a 667MHz para el microprocesador y 533.333MHz para la memoria DDR, ambas son las frecuencias máximas que se pueden emplear.

Para la generación del cubo hiper-espectral se dispone de dos opciones, un código generado por MATLAB Coder a partir de código .m que ya estaba disponible, y otro realizado de manera manual. Destacar que todo el código empleado en este proyecto es código C y, tras compilar a lenguaje ensamblador y crear un fichero hexadecimal, se ejecuta directamente sobre el microprocesador.

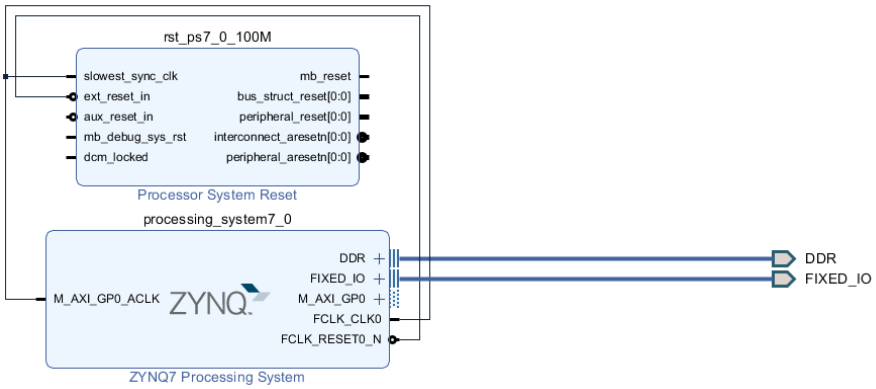

Figura 4.8: Diseño de Bloques para el Zynq-7000

En las pruebas se ha analizado el tiempo de ejecución de cada una de las tareas necesarias para generar cubos hiper-espectral y los recursos empleados en el diseño del SoC. Las pruebas se han realizado empleando todos los elementos hardware como son *FPU* o multiplicador que faciliten y optimicen los cálculos necesarios para cada una de las tareas.

Es necesario indicar en el compilador de software que emplee estos elementos hardware. Para comprobar que sí se hace uso de estos dispositivos hardware, se puede analizar el código ensamblador generado para observar qué instrucción se emplea. En la Figura 4.9, se puede observar que solo se emplean instrucciones de suma a los registros, lo que significa que la multiplicación se hace por hardware.

```
  corrected_rawimage[i] = (double)(int)qY * coef_WhiteREF[i] * 4.0;
5c:   13000000    addk    r24, r0, r0
60:   13200000    addk    r25, r0, r0
```

Figura 4.9: Código Ensamblador para Multiplicador Hardware

Una vez que se ha analizado el rendimiento de los diferentes *SoC* en la generación de imágenes hiper-espectrales, se procede a realizar un *SoC* con una *DMA* para analizar el movimiento de datos entre el microprocesador, la memoria *DDR* y los diferentes *IP Cores* de la *FPGA*.

Para ello, sobre el *SoC* realizado del Cortex-M3 se han realizado diferentes implementaciones de *DMA*, con o sin *Scatter/Gather (SG)* para evaluar cuál de las opciones ofrece mayor rendimiento y adecuación al proyecto. Es decir, cómo deben estar organizados los datos en memoria para obtener el máximo rendimiento de cada una de las implementaciones de *DMA*. Se analizarán también el tamaño de los paquetes y la relación tamaño del paquete a enviar y tiempo necesario para enviar un cubo hiper-espectral completo.

Ya que no se dispone actualmente de los aceleradores hardware para clasificación y procesado de las imágenes, se sustituirán estos bloques por una FIFO propia.

La FIFO empleada para el estudio de las diferentes *DMAs* es una FIFO propia diseñada en VHDL-08 para asemejar el proceso de implementación a como se realizará el procedimiento de transferencia de datos con los coprocesadores hardware. Para ello se ha sintetizado y procesado la FIFO con GHDL para obtener una netlist de VHDL-93 para poder generar el *bitstream* desde Vivado.

El comando empleado para la generación de la netlist de VHDL-93 es `--synth`. A parte de generar una netlist, también se puede emplear este comando para realizar una comprobación rápida para verificar si el componente es sintetizable.

Antes de emplear el comando `--synth` es necesario que todas las ficheros *HDL* que forman el componente hayan sido analizados previamente.

En las pruebas se ha analizado el tiempo empleado para transmitir un cubo hiper-espectral completo desde la memoria externa a la FIFO que emula al coprocesador hardware. En la Figura 4.10se puede ver un esquema de una transferencia de una imagen hiper-espectral completa en 50 paquetes cada uno de 44172 píxeles, es decir, la DMA realiza 50 transferencias de 44172 píxeles a la FIFO para enviar una imagen hiper-espectral completa. Se ha estudiado el número de imágenes por segundo dependiendo del tamaño del paquete de cada una de las transmisiones de la *DMA*. Esto se ha realizado para poder obtener datos sobre cuál es el número óptimo de transferencias a realizar para enviar un cubo hiper-espectral completo que los *IP Cores* de procesado sean capaces de manejar sin perder datos, garantizando un buen funcionamiento.

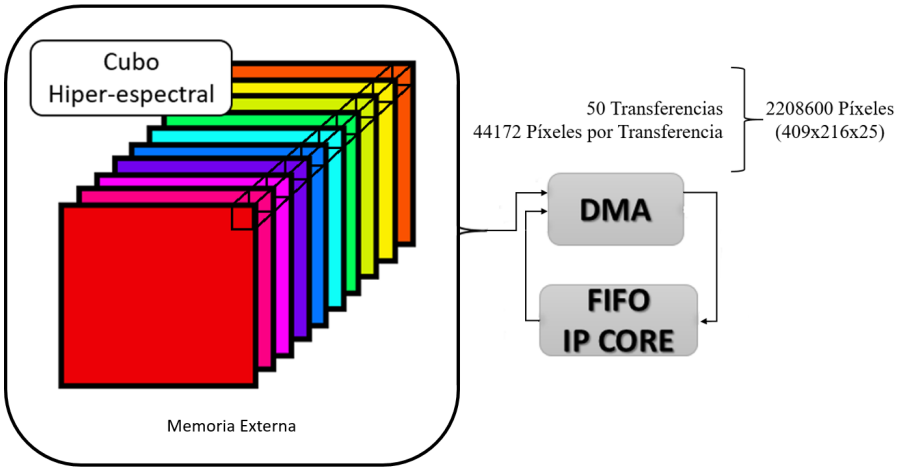

Figura 4.10: Esquema de Transferencia de 50 paquetes desde la memoria externa a la FIFO

Por último, de todos los datos obtenidos de los diferentes estudios, se han tomado decisiones sobe los microprocesadores y las alternativas de *DMAs*. Estos *IP Cores* seleccionados son los que mayor beneficio aportarán a la aplicación de *ADAS*..

4.3.2. Desarrollo de SoCs con RISC-V

En este proyecto se ha intentado desarrollar diferentes *SoCs*, empleando diferentes implementaciones de *IP Cores* de microprocesadores RISC-V, pero no se ha llegado a obtener ningún *SoC* operativo en el que evaluar las tareas de procesamiento bajo estudio. A continuación se explicará los pasos que se han seguido en el desarrollo de los diferentes sistemas y los problemas que se han encontrado para dos microprocesadores de arquitectura RISC-V, concretamente *IPs* de microprocesadores de SiFive y lowRISC.

Implementación Con IP Cores de SiFive

SiFive es la empresa que más está aportando al desarrollo de procesadores RISC-V, tanto en Soft *IP Cores* como en implementaciones en silicio (“Open-Five”) y placas de desarrollo. Disponen de un amplio catálogo de procesadores RISC-V que comparten características con los procesadores de ARM, Figura 4.11

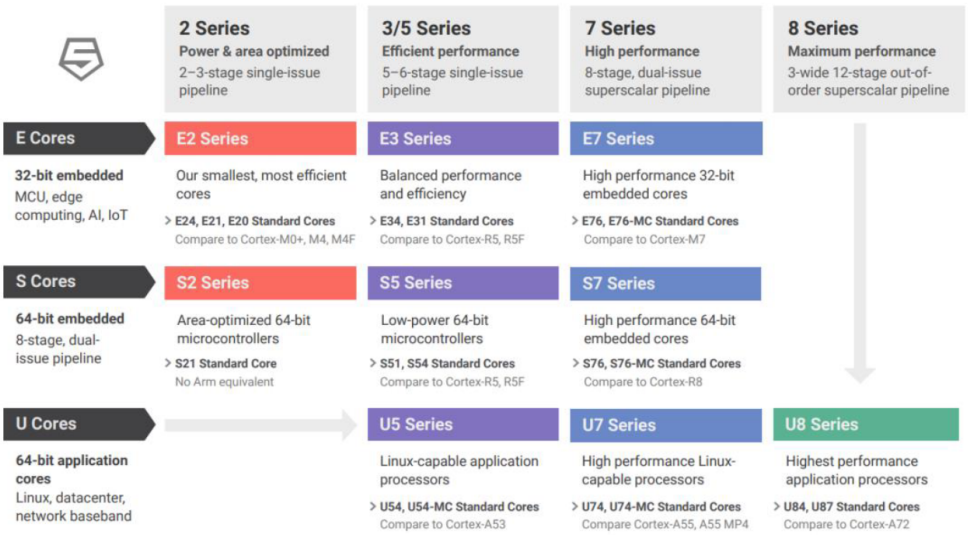

Figura 4.11: Catalogo de IP Cores de SiFive

Una posible ventaja sobre el resto de los diseñadores de *Soft IP Cores* es la posibilidad de crear un procesador personalizado, ya que disponen de un generador de *IP Cores*. Pero,

la realidad, es que, una vez se ha diseñado el procesador, este queda como una petición, y, de manera gratuita, solo se tiene acceso a la versión de prueba de los microprocesadores.

Se ha probado a descargar un Core de SiFive, concretamente el E24 base, para analizar qué ficheros se generan con el diseñador de código[53]. Concretamente, el ejemplo de implementación que ofrece SiFive es compatible con la *FPGA* de rango medio que se ha empleado en el desarrollo de este proyecto .

La implementación se puede realizar tanto con la herramienta que nos proporciona SiFive, FreedomStudio, o con Vivado. FreedomStudio es un entorno SDK, basado en Eclipse, muy similar al uVision de ARM. En él, se puede escribir/editar código, seleccionar el *BSP*, realizar el proceso de depurado y programar la *FPGA*. La primera aproximación que se realizó con el Core de SiFive fue un SoC basado en una UART y una memoria DDR3 para poder leer y escribir datos, Figura 4.12

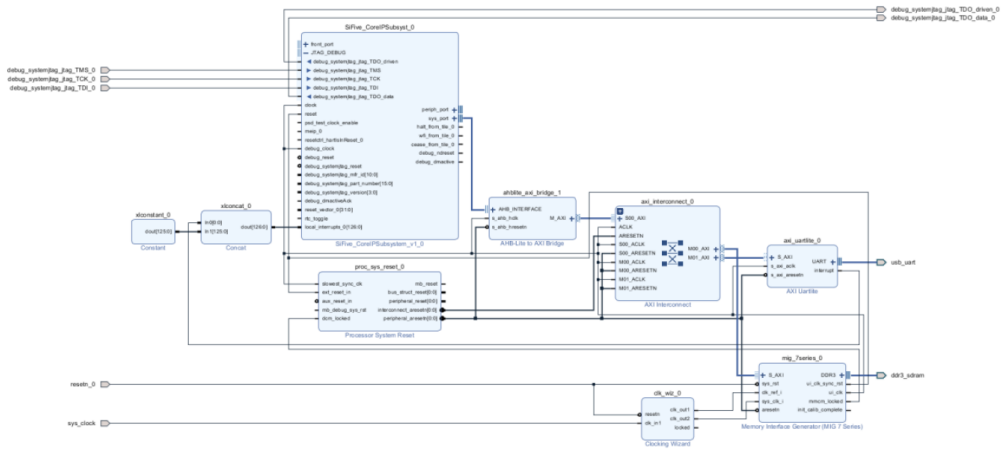

Figura 4.12: Diseño de Bloques con el Core de SiFive

Primero, a partir del código *RTL* proporcionado por SiFive, se realizó un *IP Core*, para poder definir las interfaces del procesador y asignar las direcciones de memoria. Posteriormente, se añade este IP Core al Design Block con la UART y la memoria DDR. Destacar que hubo que añadir un módulo puente de bus AHB a bus AXI para poder realizar la conexión con los periféricos, ya que el protocolo de comunicaciones empleado por Xilinx es siempre AXI o AXI-Lite.

Sí que se llegó a realizar un SoC que se puede sintetizar e implementar, pero el problema surge cuando se exporta el diseño al SDK de Vivado para crear el *BSP*. El primer problema que aparece es que el SDK de Vivado no reconoce el Core de *SiFive* como procesador instanciado en el diseño. Esto se debe, a que Xilinx no reconoce los *IP Cores* de RISC-V

como procesadores, debido a que no están incluidos en su base de librerías. Por lo tanto no se puede obtener el *BSP* con todos los *IP Cores* del diseño. Este proceso ha quedado descartado ya que el tiempo de desarrollo es muy elevado.

SiFive nos proporciona el *BSP* del procesador, pero lo que se necesita, es integrar el *BSP* del procesador y de los *IP Cores* de Xilinx incluidos en el diseño en un mismo archivo. Una solución posible es realizar el archivo con todos los *BSP* a mano, pero esto implica crear un proyecto por cada *IP Core* instanciado en el diseño para obtener el *BSP* de cada uno de los componentes hardware.

Se ha preguntado por el foro de Xilinx si existía alguna manera de realizar el *BSP* para un procesador de RISC-V con sus herramientas, pero se obtuvo una respuesta negativa que deja claro que no van a trabajar para proporcionar soporte. Esto se debe a la relación de Xilinx con ARM y además de sus propios procesadores como el Microblaze, por lo que no ven interés en favorecer el crecimiento de estos *Soft Cores* de RISC-V.

Hay que destacar, que aunque el diseño es implementable, aunque este no cumple las restricciones de tiempo, Figura 4.13. Este problema se podría llegar a solucionar ajustando la configuración de las herramientas de implementación, ya que el reloj empleado en este diseño es de 83.33MHz, al igual que en el *SoC* desarrollado para el Cortex-M3. Además el número de LUTs empleadas en el diseño es bastante alto, llegando a ocupar un 83 %, por lo que si hubiera que ampliar con más periféricos el *SoC*, este sería un factor limitante, Figura 4.14.

Timing	Setup
Worst Negative Slack (WNS):	-16.62 ns
Total Negative Slack (TNS):	-39506.984 ns
Number of Failing Endpoints:	6010
Total Number of Endpoints:	30104
Implemented Timing Report	

Figura 4.13: Error de Timming con el Core de SiFive

Otro problema que se ha encontrado durante el trabajo con los Cores de SiFive es que para descargar el código a la placa es necesario disponer de un debugger concreto: “Olimex ARM-USB-TINY-H”. Se ha realizado una pregunta en el foro de SiFive sobre si existe alguna alternativa al debugger para descargar el código, pero no se ha obtenido respuesta aún. Debido a este problema, no se ha podido comprobar si el bitstream que viene por

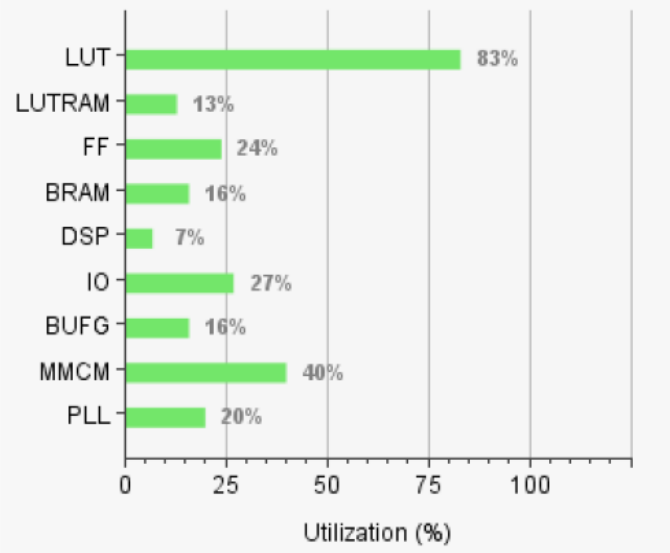

Figura 4.14: Reporte de Utilización con el Core de SiFive

defecto con el Core puede llegar a funcionar en una *FPGA* de Xilinx, ya que no se ha podido descargar y probar el código en placa.

Implementación de IP Cores con lowRISC

Otra de las empresas que realizan *Cores* y *SoCs* basados en RISC-V es lowRISC. Se ha elegido esta otra alternativa debido a que tienen un ejemplo sobre la Arty A7 35T y parece ser que el software se descarga a la placa junto al bitstream, como ocurre con el Cortex M3 de ARM.

Esta empresa utiliza otras herramientas alternativas a Vivado para realizar los *SoCs* y programar la *FPGA*:

- Compilador de RISC-V.
- FuseSoc.
- Software para sintetizar e implementar: Vivado, iVerilog...

Lo primero que es necesario para poder compilar código C y C++, es el RISC-V GNU Toolchain, este compilador se puede descargar directamente del GitHub de RISC-V. Lo que permite este software es compilar el software diseñado para el Core y generar un .elf/.hex, con lo drivers del Core y el software, para poder incluirlo con el bitstream en la programación de la *FPGA*.

El segundo programa es FuseSOC[42], que es un diseñador de *SoCs* empleando *IP Cores*. Es la herramienta básica en la creación de *SoCs* y *Cores* empleada en este apartado. Se basa en un fichero .core en el cual se instancian los ficheros del diseño y se indican también qué ficheros son *RTL*, cuales restricciones y cuales testbench. Otra característica de este programa es que permite sintetizar e implementar el diseño empleando el p que se programa que desee el usuario, por ejemplo iverilog, verilator, incluso soporta Vivado. Gracias a este programa se simplificara la tarea de añadir uno a uno los ficheros que componen el *SoC*, ya que se pueden listar y este programa es el encargado de generar el bitstream empleando Vivado.

El tercer requisito es tener instalado un software para realizar la síntesis y la implementación del *SoC*. En este caso se va a realizar empleando Vivado. Lo primero es realizar el archivo .core para FuseSOC con los ficheros *RTL* que componen el Core Ibex.

La primera toma de contacto con el entorno de lowRISC es con el ejemplo que incluyen. En el ejemplo, se ha programado la *FPGA* Arty A7 35T con un *SoC* basado en el Core Ibex de lowRISC. El ejemplo se basa en el parpadeo de los leds de la placa cada vez que se escribe una palabra por el bus de datos.

El ejemplo ya incluye el fichero .core necesario para trabajar con FuseSOC. El fichero empleado para la realización del *SoC* se puede ver en el Anexo explicado cómo se han de llamar a los ficheros y como se organiza el código.

La programación se ha realizado en C, pero es necesario generar un fichero .vmen para inicializar la memoria con ese programa. Para ello se debe compilar el fichero .c con la toolchain de RISC-V 32 bit. Esta toolchain se puede descargar desde el propio GitHub de RISC-V. Para realizarlo se compila el fichero Makefile incluido, en el que se define cual es el fichero C, cual es la arquitectura del procesador, donde está definido el compilador y los ficheros de salida.

La programación de la *FPGA* se puede realizar con FuseSOC directamente, pero también se puede realizar con Vivado, ya que se dispone del bitstream. En este ejemplo viene como opción programar la *FPGA* utilizando FuseSOC, pero se ha decidido optar por el Hardware Manager de Vivado.

Realmente, lo que hace FuseSOC para programar la *FPGA* es lanzar Vivado en modo batch y programar el target como se haría de una manera manual con el Hardware Manager.

Tras programar la *FPGA*, siguiendo los pasos del tutorial, se ha obtenido el bitstream,

pero al igual que con SiFive, no se ha conseguido un buen resultado al implementar. No se puede decir que funcione correctamente ya que no se puede visualizar los leds parpadeando.

Ya que durante el proceso de síntesis e implementación no se ha obtenido ningún error, se puede llegar a la conclusión que el error se puede deber a una mala asignación del software en la memoria SRAM dedicada para él. También se ha probado a ejecutar el Makefile que crea el proyecto de Vivado con FuseSOC, pero se han obtenido otro error. El error generado es que el número de BUFGCTRL es de 33, cuando el máximo admitido por la *FPGA* Arty A7 es de 32. Este error no se llega a entender muy bien, ya que el ejemplo estaría diseñado para funcionar en la Arty A7, Figura 4.15.

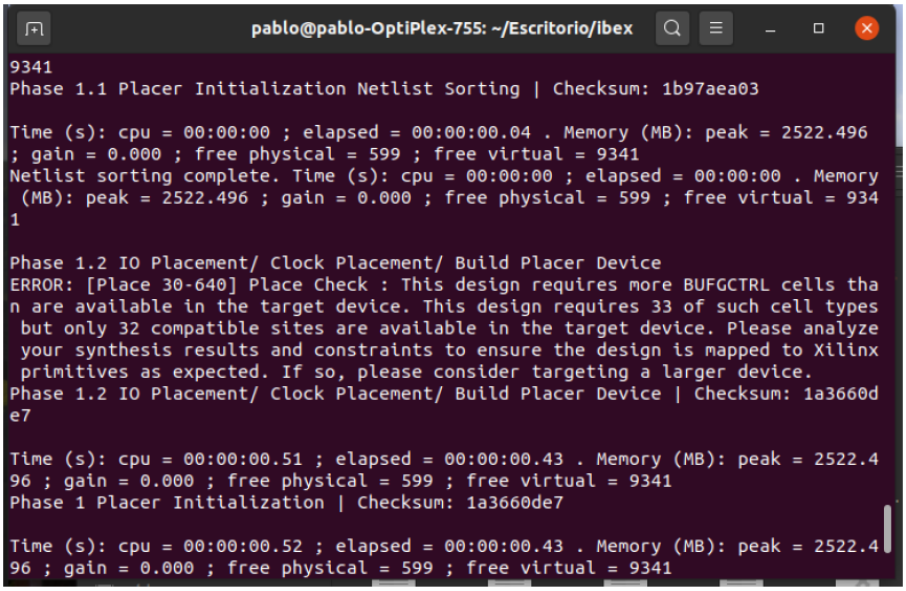

Figura 4.15: Error Obtenido con el Makefile de lowRISC

Como alternativa a FuseSoC se ha intentado desarrollar el *SoC* con Vivado, ya que se dispone del código *RTL* que forma el microprocesador. Al tratarse de un proyecto de Vivado autogenerado, se ha realizado la implementación añadiendo uno a uno los ficheros *RTL* que conforman el *SoC*. Se han introducido al proyecto los mismos ficheros que se mencionan en el fichero para FuseSOC y en el Makefile anteriormente empleados. En este caso no se ha obtenido un error al generar el bitstream, además, el bitstream es funcional. Es decir, los leds parpadean alternativamente, Figura 4.16.

Tras examinar más afondo el código *RTL*, se comprueba que es un *SoC* muy simple, ya que no dispone de periféricos. Solo dispone de una SRAM, el procesador, un *IP Core* de

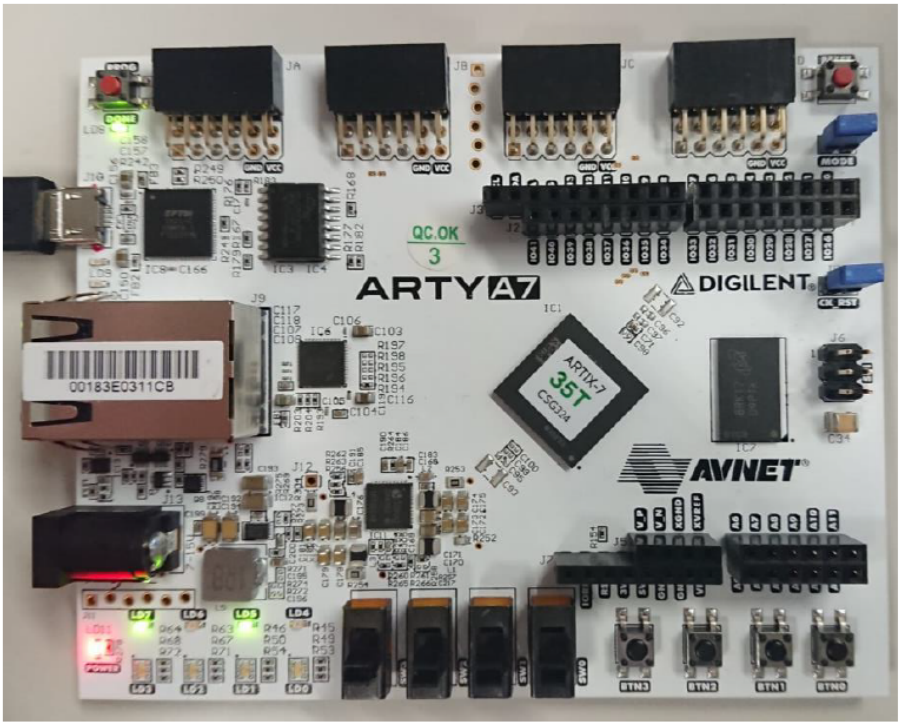

Figura 4.16: FPGA con RISC-V Ibex Funcionando

reloj y reset y la conexión directa de los leds a los buses de comunicación, sin emplear una interfaz GPIO.

Se podría llegar a editar el código *RTL* para añadir los módulos de GPIO y demás interfaces, pero llevaría mucho tiempo, además de posteriores problemas de compatibilidad, ya que emplean un protocolo de comunicación propio que describen en la datasheet del procesador.

Conclusiones sobre los Microprocesadores RISC-V

Tras haber realizado varias implementaciones con diferentes *IP Cores* de diferentes empresas, se ha llegado a la conclusión de que sí existe un mercado amplio de Cores de RISC-V pero que requieren herramientas alternativas a las utilizadas normalmente.

Muchas de estas herramientas, como el compilador de RISC-V, es necesario disponer de un sistema operativo basado en UNIX, ya sea OS X o LINUX. También destacar que Vivado no está disponible para todos los sistemas operativos basados en UNIX, además, la compatibilidad con las versiones más recientes de algunos sistemas operativos es nula, por lo que hay que instalar librerías de forma manual para poder ejecutarlo.

Uno de los principales inconvenientes es que el lenguaje *HDL* empleado por los diseñadores es Verilog o SystemVerilog, muy diferente a VHDL por lo que la adaptación y comprensión del código *RTL* se complica.

Además, para poder añadir el módulo *RTL* al diseño de bloques o realizar un *IP Core* a partir del código *RTL*, es necesario que al menos el fichero superior esté escrito en VHDL o Verilog. Esto implica que, como casi todos los *Cores* están escritos en SystemVerilog, habría que crear o traducir a Verilog/VHDL el fichero superior.

Otro de los inconvenientes, es que todo lo que se ha encontrado es código *RTL*, nunca un *IP Core* para Vivado, como sí sucede con el procesador Cortex-M3 de ARM.

Aunque en este intento de desarrollo de *SoC* no se haya conseguido realizar un un diseño basado en RISC-V funcional para realizar la experimentación objeto de este TFM. Sí que se ha comprobado que funcionan, ya que, varios de los bitstreams que venían por defecto han podido ser ejecutados en la *FPGA*.

Se puede decir que el principal problema viene en la asignación del código en la memoria RAM para que el *Core* acceda a ello. Este punto sería uno de los que habría que trabajar de manera más concienzuda. A su vez, se utilizan diferentes formatos, aunque todos ellos se basan en código en hexadecimal, (vmem, mem, bram_tcl, hex), lo que también complica entender cómo se realiza la asignación del código, ya que no es un patrón que se repita en cada *SoC*.

Por ahora, el camino de los RISC-V en *FPGA* sigue necesitando ser explorado profundamente, por lo que el desarrollo de *SoCs* se complica. Además, para poder ejecutar algoritmos complejos, como son los requeridos por algoritmos de Inteligencia Artificial, es necesario incluir multiplicadores hardware y unidad de coma flotante, de la que muchos *IP Cores* de RISC-V no disponen.

5. CAPÍTULO

Resultados

En este capítulo se procederá a comentar los resultados obtenidos en la caracterización de la ejecución del algoritmo de generación de cubos hiper-espectrales en los diferentes microprocesadores analizados, tanto para el código generado a mano, como para el código C generado por MATLAB Coder desde código .m. Para ello se medirá el tiempo empleado por el microprocesador para realizar cada uno de los procesos necesarios para generar el cubo hiper-espectral (409x216x25 que hace un total de 2208600 píxeles por imagen) a partir de una imagen en crudo.

5.1. Código Realizado a Mano

El código realizado a mano sigue el siguiente esquema que se explicará a continuación:

1. Recortar la Imagen Raw : Recorte y enmarcado de la imagen.
2. Corrección de Reflectancia : Cálculo de la reflectancia corregida (Eliminación del ruido estático) y normalizada.
3. Extracción y alineación de Bandas : Extracción de las 25 bandas que forman el cubo, se pasa de una imagen 2D a 3D.
4. Filtro de Medianas (3x3) : Aplicación de un filtro de medianas de tamaño 3x3.
5. Corrección Espectral : Filtrado del cubo espectral por una matriz de corrección.

6. Normalización de las Bandas : Normalización de la reflectancia por banda.

Hay un paso que es el procesado de las imágenes de referencia del blanco y del negro, para poder realizar la extracción de bandas correctamente. Este paso se asume que se realiza de manera *offline*, por lo que las imágenes procesadas (recortadas y sin ruido estático) ya se encuentran presentes en la memoria *DDR* del *SoC* para proceder a la generación del cubo hiper-espectral, de tamaño 409x216 píxeles cada una de las 25 bandas.

5.1.1. ARM Cortex-M3

Los resultados obtenidos de la generación del cubo hiper-espectral en el Cortex-M3 son los que se pueden ver en la Tabla 5.2. Son unos resultados malos, ya que tarda mas de tres minutos en generar un cubo hiper-espectral a partir de la imagen .^{en} crudo" original. Esto se debe en gran medida a que no dispone de FPU el microprocesador, por lo que el tratamiento de números float llevará mucho mas tiempo que en los otro microprocesadores. Además, tampoco se puede optimizar las multiplicaciones por Hardware, ya que no dispone de módulos de multiplicación vectorial como sí lo hace el Cortex-A9 en el Zynq-7000.

El diseño de los *SoCs* es el mismo en el caso del código generado manualmente y el generado por MATLAB, por lo que los recursos empleados en la *FPGA* es el mismo. En la Tabla 5.1 se pueden ver los recursos que consume el *SoC*.

Tabla 5.1: Recursos Empleados en el SoC ARM Cortex-M3

Recursos	**Utilización**	**Disponible**	**% Utilización**
LUT	20881	63400	32.94
LUTRAM	1180	19000	6.21
FF	13593	126800	10.72
BRAM	40	135	29.63
DSP	3	240	1.25
IO	56	210	26.67
MMCM	2	6	33.33
PLL	1	6	16.67

Además, destacar, que aunque es bastante configurable el *IP Core* del microprocesador, tanto el *Stack* como el *Heap* del microprocesador está bastante limitado, por lo que la utilización de memoria dinámica es muy limitada, lo que hace necesario aumentar el número de accesos a memoria externa, aumentando el tiempo de ejecución. A diferencia

de los otros dos *SoCs* realizados, en este se ha adaptado el código y variables a punteros, por lo que se han tenido que aumentar el número de bucles del código para poder realizar las mismas operaciones que en los otros casos.

El *Stack* del microprocesador es la zona de memoria interna del microprocesador, en el que se almacenan las variables locales de las diferentes funciones del software. El *Heap* es la zona de memoria interna que se encarga de almacenar las variables globales, además de poder alojar memoria dinámica, aunque esto último es recomendable no hacerlo en microprocesadores.

Tabla 5.2: Tiempos de Ejecución ARM Cortex-M3

	Tiempo Empleado (s)
Recortar Imagen	2.6149877
Corrección de Reflectancia	11.767429
Extracción de Bandas	1.773919
Alineación de las Bandas	16.147022
Filtro de Medianas (3x3)	31.2992083
Corrección Espectral	114.6741931
Normalización de las Bandas	11.228335

5.1.2. Xilinx Microblaze

Los resultados obtenidos para el Xilinx Microblaze son mejores que los obtenidos con el Cortex-M3 de ARM. Esto se debe principalmente a que sí dispone de *FPU* el microprocesador, y que la frecuencia de reloj es mayor que la empleada en el Cortex-M3. Aunque son mejores los resultados obtenidos con este microprocesador, los resultados no están cerca de los deseados. El tiempo total de generación del cubo hiper-espectral es cercano al minuto, y se puede ver el desglose de tiempo en la Tabla 5.4

Los recursos empleados por el Xilinx Microblaze se pueden ver en la Tabla 5.3. Son muy similares a los recursos empleados en el caso del Cortex-M3. El número de *DSP* es mayor debido a que se emplea un multiplicador de 64 bits en el caso del Xilinx Microblaze, mientras que en el Cortex-M3 es solo de 32 bits. Además, recordar que este microprocesador sí dispone de *FPU*.

La limitación de la frecuencia de reloj viene dada por el controlador de la memoria externa, que en la Arty A7 100T, el máximo que se puede alcanzar es de 83.3333MHz. En el caso de no necesitar de memoria *DDR* externa, sí que se podría subir la velocidad del microprocesador hasta los 250MHz, pero para esta aplicación es necesario disponer de

Tabla 5.3: Recursos Empleados en el SoC Xilinx Microblaze

Recursos	Utilización	Disponible	% Utilización
LUT	15121	63400	23.85
LUTRAM	1981	19000	10.43
FF	14455	126800	11.40
BRAM	20	135	14.81
DSP	6	240	2.50
IO	52	210	24.76
MMCM	2	6	33.33
PLL	1	6	16.67

una memoria *DDR* para almacenar las imágenes y los datos obtenidos del análisis de las imágenes.

En este caso para poder obtener el máximo rendimiento por parte del *SoC* se ha decidido añadir otro reloj que sea el encargado de controlar el microprocesador y los periféricos excepto la memoria *DDR*. Aunque el diseño es funcional no es recomendable no emplear el reloj del controlador de memoria como reloj general del diseño. Emplear un reloj diferente para cada periférico puede llevar a fallos en el protocolo debido a una mala sincronización de la comunicación entre los distintos *IP Cores* del diseño.

Tabla 5.4: Tiempos de Ejecución Xilinx Microblaze

	Tiempo Empleado
Recortar Imagen	0.4152
Corrección de Reflectancia	4.7071
Extracción de Bandas	1.5689
Alineación de las Bandas	2.3954
Filtro de Medianas (3x3)	11.7403
Corrección Espectral	40.5473
Normalización de las Bandas	2.6732

5.1.3. Xilinx Zynq-7000 APSoC

Los resultados obtenidos con el *APSoC* Zynq-7000 son significativamente mejores a los obtenidos con los otros microprocesadores. Esto se debe a que además de incluir *FPU* y multiplicador hardware, la frecuencia del microprocesador es de 667MHz, es decir 8 veces mayor que en el caso del ARM Cortex-M3 estudiado anteriormente. Destacar que este *APSoC* no se puede comparar directamente con los anteriores, ya este no es un *Soft-Core* como los anteriores.

Otra factor que lo hace diferente de los otros dos microprocesadores es la disponibilidad de un módulo *SIMD*, para multiplicaciones vectoriales, NEON. En la Tabla 5.5 se puede encontrar el desglose de tiempos sin la optimización con el módulo NEON. Estos resultados obtenidos son bastante buenos en comparación con los anteriores, pero aun así para realizar un análisis en tiempo real(que para esta aplicación serían unos 3-4FPS mínimo) de las imágenes no se puede asegurar todavía un buen funcionamiento.

En este caso no se presentan los recursos empleados en el diseño del *SoC*, porque, el *SoC* ya esta implementado en silicio, por lo que la *FPGA* queda libre al completo.

En el caso optimizado con el módulo NEON, los resultados son aún mejores, estableciéndose por debajo de dos segundos la generación del cubo hiper-espectral. Las operaciones del filtro de medianas y la corrección espectral, que eran las tareas que más tiempo le llevan al microprocesador, se ven mejoradas hasta un x3.7 en el caso del filtro y en un x5.98 en el caso de la corrección espectral. Estos resultados si que se acercan a los deseados para poder ofrecer una respuesta en tiempo real. Si se opta por eliminar el tiempo de la corrección espectral (tarea que se puede acelerar por hardware al tratarse de una multiplicación de matrices), se estaría cerca de los requerimientos de latencia del sistema. Otra opción para mejorar la latencia, puede ser eliminar el preprocesado de la imágenes o acelerar por hardware el filtro.

Comparando ambos resultados con el Zynq-7000, se puede observar que hay tareas como es recortar la imagen capturada por la cámara, en las que apenas hay optimización por parte del módulo NEON, pero en las tareas en las que se puede aplicar la optimización con el multiplicador vectorial, como son el filtro de medianas y la corrección espectral, se puede observar una optimización favorable en el tiempo de ejecución del código. El tiempo total se ve reducido de 8.7054s a 1.8023s, es decir, gracias a la optimización el código es 4.83 veces más rápido.

Tabla 5.5: Tiempos de Ejecución Xilinx Zynq-7000 APSoC sin Optimización por NEON

	Tiempo Empleado (s)
Recortar Imagen	0.0021
Corrección de Reflectancia	0.3799
Extracción de Bandas	0.1189
Alineación de las Bandas	0.4179
Filtro de Medianas (3x3)	1.8330
Corrección Espectral	5.7372
Normalización de las Bandas	0.216

Por lo tanto, se puede decir que el módulo NEON es esencial para obtener un buen ren-

Tabla 5.6: Tiempos de Ejecución Xilinx Zynq-7000 APSoC con Optimización por NEON

	Tiempo Empleado (ms)
Recortar Imagen	1.8894
Corrección de Reflectancia	107.9691
Extracción de Bandas	90.9324
Alineación de las Bandas	53.7630
Filtro de Medianas (3x3)	494.1723
Corrección Espectral	959.1678
Normalización de las Bandas	94.6476

dimiento, y que sí se quiere implementar habrá que tener en cuenta un *SoC* que disponga de un módulo de multiplicación vectorial, con el fin de acelerar las tareas que requieran muchos cálculos similares y repetitivos como son el filtro de medianas y la corrección espectral.

5.2. Código Generado Por MATLAB

En este caso el código ha sido generado con MATLAB Coder a partir de un script de MATLAB. Se ha realizado un estudio similar al estudiado anteriormente, se ha evaluado el tiempo de ejecución de cada uno de los procesos en el generador de cubos hiperespectrales. El fin es poder comparar estos resultados con los obtenidos con el código .m compilado con MATLAB Coder para obtener ficheros .C.

5.2.1. ARM Cortex-M3

En el caso de Cortex-M3 no se ha conseguido implementar este código generado por MATLAB. Esto se debe a la gran cantidad de variables locales empleadas y en especial, al tamaño de las variables, que superan el tamaño disponible en el *Stack*. En un principio esta memoria interna sí que se podría aumentar de tamaño, pero, al aumentar el tamaño del *Stack*, la aplicación no se ejecuta, ya que solo tiene soporte el *IP Core* para tamaños de memoria interna de 32kB. Por lo que se hace imposible actualmente implementar el código en este caso, debido a que el código más las variables ocupan más de 32kB.

Esto significa que el modelo de *Soft-Core* diseñado por ARM es un poco limitado, lo que complica la tarea de diseño. Hay que adaptar el código minuciosamente, minimizando el número de variables globales y locales, intentando sustituirlas por punteros.

Este inconveniente de sustituir variables locales por punteros hace que el tiempo de acceso a memoria se vea incrementado, por lo que tampoco es una opción viable, ya que el rendimiento será bajo.

5.2.2. Xilinx Microblaze

El código generado en el caso del Xilinx Microblaze es funcional, pero con un rendimiento muy bajo, los tiempos son mucho mayores respecto a lo obtenido con el código manual. Esto se debe a la gran cantidad de bucles y la gran utilización de la memoria dinámica, lo que hace que los accesos a la memoria *DDR* se multipliquen respecto al caso del código manual.

Los resultados que se pueden ver en la Tabla 5.7 y reflejan que este microprocesador no es una buena opción si se requiere capacidad de cálculo sobre grandes cantidades de datos.

Tabla 5.7: Tiempos de Ejecución Xilinx Microblaze con el código generado por MATLAB

	Tiempo Empleado (s)
Corrección de Reflectancia	32.8893
Extracción de Bandas	1.6936
Filtro de Medianas (3x3)	481.9980
Corrección Espectral	26.9528
Normalización de las Bandas	82.9365

Destacar que la diferencia con el código manual es muy grande. Esto se debe a la poca optimización que hace MATLAB de los bucles al traducir el script a código C. La reutilización de datos en el filtro de medianas por parte del código manual reduce el tiempo de acceso a memoria, ya que, los datos permanecen en la memoria cache del propio microprocesador. Como se puede observar en la Figura 5.1, el código para la primera iteración del filtro de medianas carga los nueve valores correspondientes a la ventana de 3x3, mientras que, las siguientes iteraciones del filtro guardan los 6 datos que se pueden reutilizar y sólo se carga de la memoria externa los tres nuevos datos.

Es por eso, que reutilizar datos es esencial si se quiere mejorar el rendimiento del código y reducir el tiempo de ejecución.

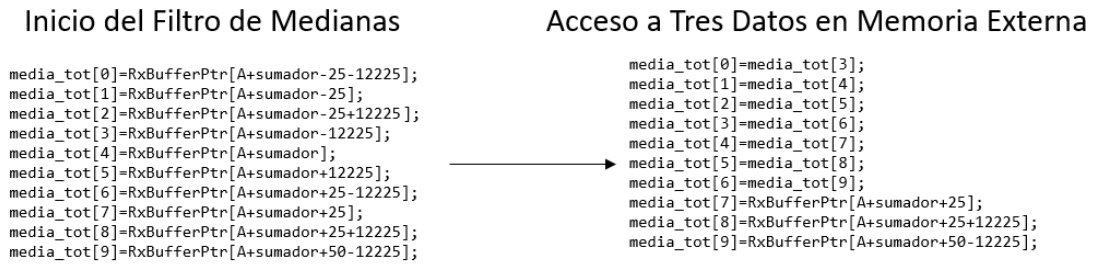

Figura 5.1: Código del Filtro de Medianas Manual para Reutilizar Datos

5.2.3. Xilinx Zynq-7000 APSoC

En este caso, los resultados son mucho más favorables que los obtenidos con el Xilinx Microblaze. En este caso se puede ver que el *SoC* es mucho más óptimo en acciones como el acceso a memoria y la optimización del código para la reutilización de datos almacenados en la memoria dinámica.

También se puede ver en que tareas actúa la optimización con el módulo NEON. En las tareas que requieren de cálculos sobre los datos se puede observar que el tiempo de ejecución se reduce a la mitad.

Aún viendo que estos datos son mejores que los obtenidos que con el Xilinx Microblaze, el tiempo de ejecución con el código manual sigue siendo más bajo. Por lo tanto, aunque requiere más tiempo escribir un código para generar cubos hiper-espectrales, el rendimiento que se puede obtener es mucho mayor.

Tabla 5.8: Tiempos de Ejecución Xilinx Zynq-7000 APSoC sin Optimización por NEON del código generado por MATLAB

	Tiempo Empleado (s)
Corrección de Reflectancia	0.2678
Extracción de Bandas	0.2916
Filtro de Medianas (3x3)	21.6372
Corrección Espectral	0.7018
Normalización de las Bandas	1.8506

5.3. Comparativa de los Resultados para el Cubo Hiper-espectral

De los resultados obtenidos se puede ver que se obtiene un rendimiento mayor con el código manual que con el generado por MATLAB. En las Tablas 5.10 5.11 se pueden compa-

Tabla 5.9: Tiempos de Ejecución Xilinx Zynq-7000 APSoC con Optimización por NEON del código generado por MATLAB

	Tiempo Empleado (s)
Corrección de Reflectancia	0.1525
Extracción de Bandas	0.2915
Filtro de Medianas (3x3)	9.1790
Corrección Espectral	0.5282
Normalización de las Bandas	0.9537

rar el tiempo necesario para cada una de las tareas de generación del cubo hiper-espectral. En tareas con mucha carga computacional como es el filtro de mediana el rendimiento obtenido es significativamente mejor, reduciendo tiempos hasta casi en valores 40 veces menores. Pero, curiosamente, la corrección espectral es peor en el caso de del código manual, esto se debe a la memoria dinámica, que aunque es recomendable no emplearla en microprocesadores, sí que puede mejorar el rendimiento en el caso de la utilización de variables locales en las funciones.

Comentar que para el código manual se puede ver una gran diferencia entre el Cortex-M3 y el Xilinx Microblaze en tiempo de ejecución. Esto se debe principalmente a la frecuencia de reloj y la existencia de *FPU*, ya que en función de recursos empleados son muy similares. La comparativa de recursos empleados en ambos *SoCs* se puede ver en la Figura 5.2. Se puede decir que para obtener un buen rendimiento por parte del multiplicador es necesario alimentar el microprocesador con una frecuencia de reloj alta, superior a 200MHz.

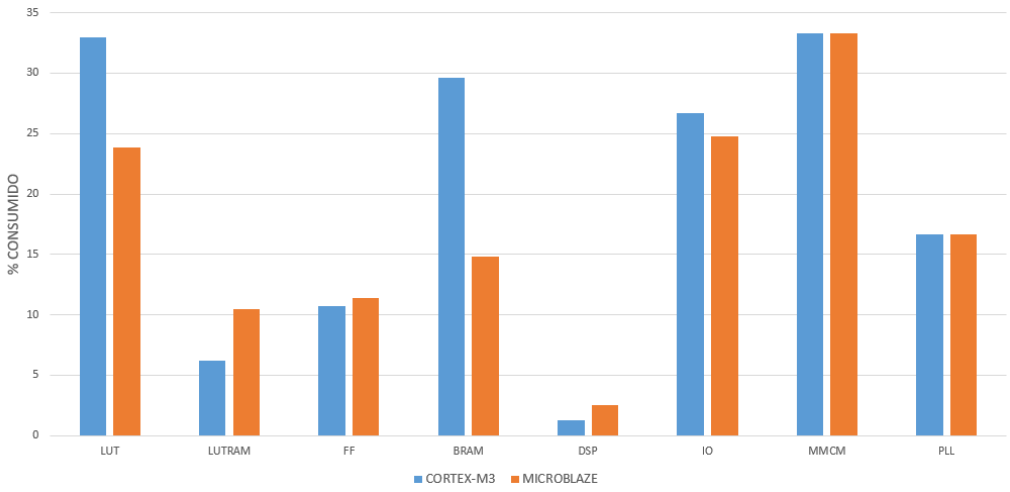

Figura 5.2: Comparativa de consumo entre los sistemas con mayor rendimiento

En la Figura 5.3, se puede ver la comparativa el tiempo empleado en cada una de las

Tabla 5.10: Tiempos de Ejecución del código manual par cada microprocesador

	Cortex-M3	**Microblaze**	**Zynq-7000**	**Zynq-7000 Optimizado**
Recortar Imagen	2,6149s	0,4152s	0,0021s	0,0018s
Corrección de Reflectancia	11,7674s	4,7071s	0,3799s	0,1079s
Extracción de Bandas	1,7739s	1,5689s	0,1189s	0,0909s
Alineación de las Bandas	16,1470s	2,3954s	0,4179s	0,0537s
Filtro de Medianas (3x3)	31,2992s	11,7403s	1,8330s	0,4941s
Corrección Espectral	114,6741s	40,5473s	5,7372s	0,9591s
Normalización de las Bandas	11,2283s	2,6732s	0,2160s	0,0946s
Tiempo Total	189,5050s	64,0477s	8,7052s	1,8025s

Tabla 5.11: Tiempos de Ejecución del código manual par cada microprocesador

	Microblaze	**Zynq-7000**	**Zynq-7000 Optimizado**
Corrección de Reflectancia	32,8893s	0,2678s	0,1525s
Extracción de Bandas	1,6936s	0,2916s	0,2915s
Filtro de Medianas (3x3)	481.9980s	21,6372s	9,1790s
Corrección Espectral	26.9528s	0,7018s	0,5282s
Normalización de las Bandas	82.9365s	1,8506s	0,9537s
Tiempo Total	626.4702s	24,7490s	11,1049s

tareas del microprocesador que mejor rendimiento ofrece para ambos tipos de códigos. El *SoC* que mejor rendimiento en ambos casos es el Zynq-7000 optimizado, esto se debe a su módulo NEON, su alta frecuencia de reloj y su diseño implementado en silicio y no en lógica reconfigurable. La mayor diferencia de tiempos entre ambos sistemas se puede observar en el filtro de medianas, esto se debe a la reutilización de los píxeles empleados en el cálculo anterior, ya que se ha diseñado el código C para ello.

Por ello, se puede decir que es necesario una alta capacidad de computación para poder implementar el algoritmo de generación del cubo hiper-espectral de manera adecuada. Además, se puede observar la utilidad del módulo NEON en la optimización de cálculos que se pueden optimizar de manera vectorial para obtener un mayor ancho de banda.

Se puede decir que en todos los sentidos que es más óptimo emplear el código C escrito a mano, ya que ofrece un mayor rendimiento y un mayor control en que se esta haciendo en cada una de las tareas de manera precisa.

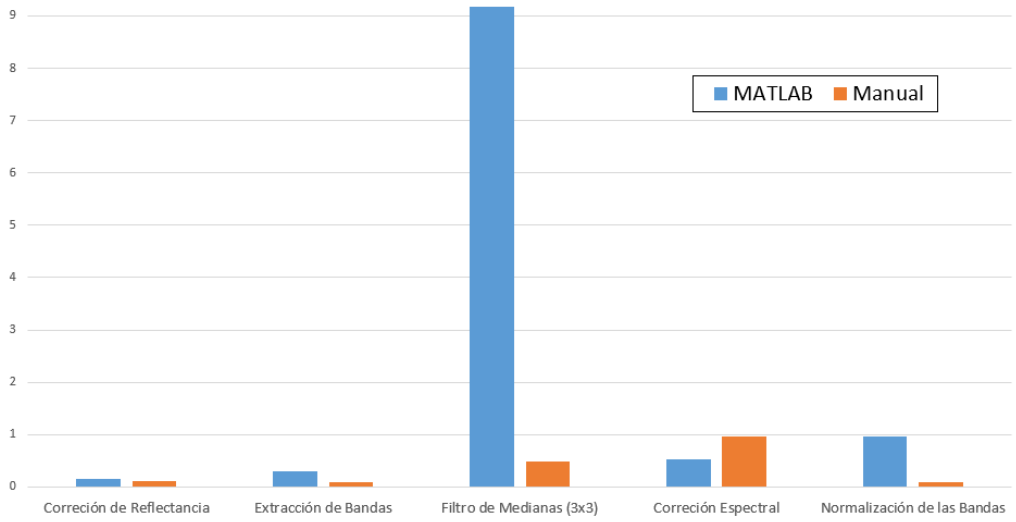

Figura 5.3: Comparativa de tiempos de ejecución entre los sistemas con mayor rendimiento

5.4. Análisis de las Transferencias de Datos con DMAs

En este apartado se comentaran los resultados obtenidos de las dos infraestructuras de *SoCs*. La FIFO empleada para emular la transferencia de datos de entrada/salida del *IP* del coprocesador neuronal, es una FIFO simple de 32-bits. El rendimiento de tiempo no será comprable con el tiempo de procesado que tendrá la red neuronal, pero se pueden extrapolar los resultados.

El *IP Core* empleado en la *DMA* es propiedad de Xilinx, el cual se comunica través del protocolo AXI con el microprocesador, mientras que la transferencia de datos se realiza con el protocolo AXI4-Stream.

Se trata de un elemento hardware que asegura un gran ancho de banda en los accesos directos a memoria, liberando a la *CPU* de realizar los movimientos de datos. El movimiento de datos a través de la *DMA* de alta velocidad ocurre entre la memoria y el destino es a través de los puertos *memory-maped to stream (MM2S)* y *stream to memory-mapped (S2MM)*, ambos con interfaz AXI4-Stream.

Los puertos *MM2S* y *S2MM* pueden operar de manera independiente. También, la *DMA* provee 4kB de protección de límites en las direcciones, ráfagas de mapeo automático, como también tiene la capacidad de poner en cola múltiples peticiones de transferencias empleando todo el ancho de banda disponible por los buses AXI4-Stream.

Además, el *IP Core* proporciona la colocación de datos a nivel de bytes, permitiendo la

lectura y escritura de la memoria comenzando en cualquier dirección.

Se puede seleccionar que la *DMA* incluya señales de control de flujo en los puertos *MM2S* y *S2MM* para poder enviar, recibir, datos de la aplicación del usuaria al *IP* de destino. También se puede seleccionar la inclusión del motor *Scatter/Gather*.

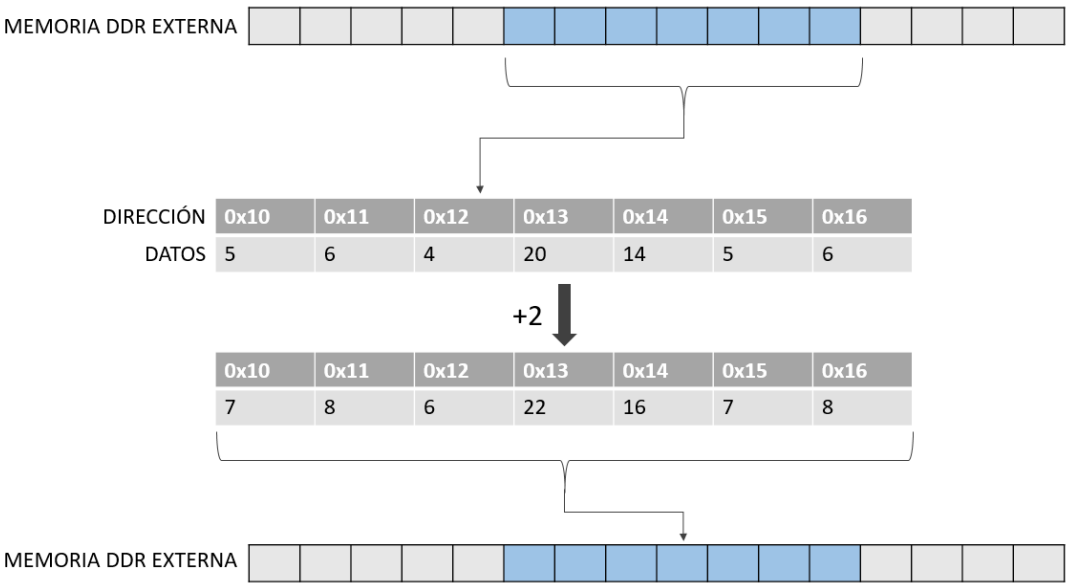

Figura 5.4: Ejemplo de Acceso a Memoria Simple

El motor *Scatter/Gather* permite el acceso, tanto lectura como escritura, a datos de manera no correlativa, Figura 5.5. Esto permite no tener que modificar el formato *Band Interleved by Pixel (BIP)*, que es como se encuentran las imágenes hiper-espectrales almacenadas, para realizar una transferencia de una sola banda espectral. Es raro encontrar que los datos a procesar se encuentren correlativos en memoria, Figura 5.4, por lo que hay que o re-colocar los datos en memoria, o adaptar el *IP Core* a como están colocados los datos en memoria, o emplear un acceso *SG*.

Se han evaluado las dos alternativas, *DMA* Simple accediendo a los datos de manera correlativa, o *DMA* con *SG* accediendo con un offset igual de ancho que el ancho de la imagen hiper-espectral. Se realizaran comparativas de tiempo de transferencia de una imagen completa para diferentes tamaños de transferencias para ambos casos con el fin de obtener conclusiones sobre cuál es la opción más adecuada para esta aplicación.

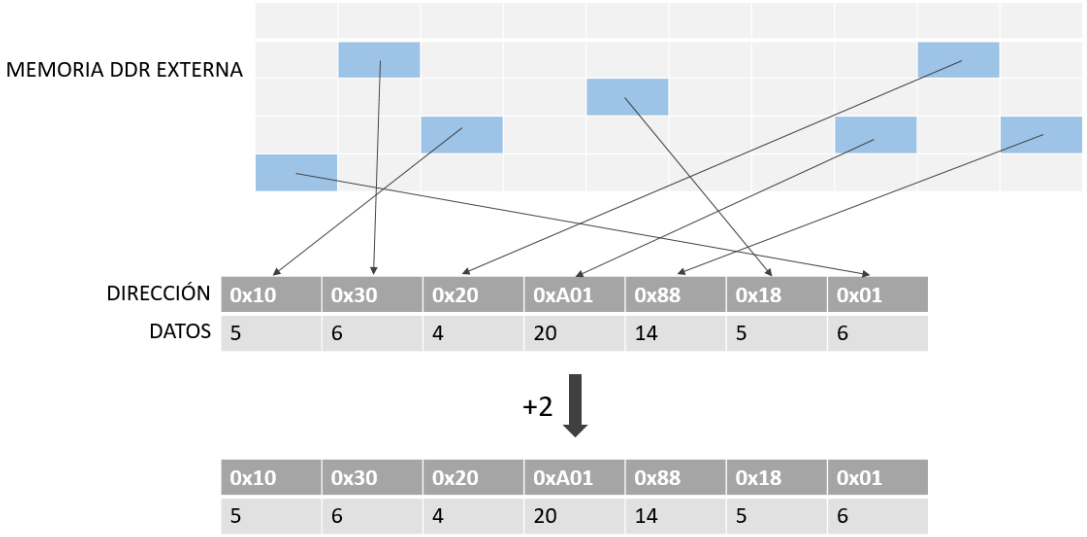

Figura 5.5: Ejemplo de Acceso a Memoria con Scatter/Gather

5.4.1. Transferencia de Datos con DMA Simple

En este caso se realizara una transferencia del primer bit de cada imagen, ya que las imágenes hiper-espectrales están organizadas en formato *BIP*. Este formato es uno de los principales empleados en la codificación de imágenes hiper-espectrales. Se caracteriza por almacenar en memoria el primer píxel de cada banda en orden secuencial, seguido por el segundo píxel de todas las bandas y así, de este modo, hasta llegar al último píxel de la última banda. En este caso el cubo hiper-espectral es de 409x216x25 que hace un total de 2208600 píxeles por imagen.

Debido a que la información de cada píxel es de 12-bit, es necesario enviar dos píxeles por cada dato de 32-bit, así, de esta manera, aprovechar todo el ancho de banda que puede llegar a dar la *DMA*. Por esta razón, el número mínimo de píxeles que se puede enviar son los dos primeros píxeles de cada banda, un total de 50 píxeles.

Para poder interpretar este formato de manera correcta es necesario establecer constante un número de bandas y el ancho y alto de la imagen. En esta prueba se han realizado varios ensayos para poder obtener datos sobre cuantas imágenes por segundo se pueden transmitir. En la Tabla 5.12 se pueden ver alguno de los resultados de las pruebas realizadas, se puede observar que la evolución del número de imágenes por segundo evoluciona de manera logarítmica hasta un límite de 75.38Frames Per Second (FPS).

En la Tabla 5.13 se puede ver el tiempo de transferencia de un cubo completo en una sola transferencia de la *DMA*. Habría que tener en cuenta que estos resultados obtenidos

Tabla 5.12: Datos de diferentes transferencias realizadas con la DMA

Nº Píxeles	**50**	**200**	**1800**	**5400**	**20450**
Tiempo del Envió 1 Imagen	0.4653s	0.1382s	0.02989s	0.01879s	0.01471s
Imágenes Por Segundo	2.1487FPS	7.2331FPS	33.4509FPS	53.2099FPS	67.9553FPS

Tabla 5.13: Envió de todos los píxeles en una sola transferencia

	2208600 Píxeles
Tiempo del Envió 1 Imagen	0.01326s
Imágenes Por Segundo	75.3826FPS

son mejores que los que se pueden llegar a obtener con e la red neuronal de clasificación, debido a que el coprocesador hardware necesita de un mayor tiempo de procesado por lo que el número de FPS será menor. También habría que añadir un buffer anterior al coprocesador neuronal para poder gestionar correctamente los datos y no sobrecargar el *IP Core* en el caso de enviar más datos de los que puede procesar, ya que se pueden originar pérdidas.

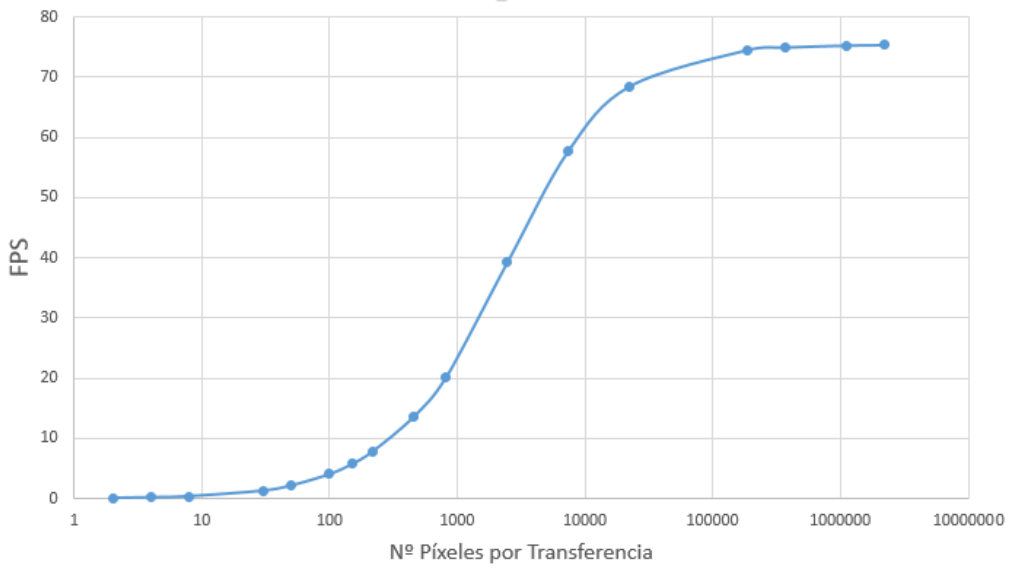

Figura 5.6: Relación FPS con el número de píxeles por transferencia

En la Figura 5.6 se puede observar que por mucho que se aumente el número de píxeles por transferencia el límite esta alrededor de los 75FPS. Este límite de saturación se puede observar para paquete superiores a 100000 píxeles por transferencia. Destacar que para que se observe mejor la evolución de la curva el eje del número de píxeles por transferencia esta en escala logarítmica.

5.4.2. Transferencia de Datos con DMA Scatter/Gather

En el caso de la *DMA* con *SG* se envían los píxeles con un offset mínimo entre cada uno para, al contrario del caso anterior, enviar los píxeles de cada banda y no un píxel de cada una de las bandas. Al tener un mínimo de offset de alineación entre los descriptores, no es posible enviar paquetes de píxeles más pequeños de 50 píxeles por transferencia.

Tabla 5.14: Datos de diferentes transferencias realizadas con la DMA SG

Nº Píxeles	**50**	**216**	**2454**	**22086**	**552150**
Tiempo del Envió 1 Imagen	1.0141s	0.2347s	0.0337s	0.0155s	0.0133s
Imágenes Por Segundo	0.986FPS	4.2597FPS	29.6105FPS	64.3694FPS	74.9377FPS

Tabla 5.15: Envió de todos los píxeles en una sola transferencia con SG

	2208600 Píxeles
Tiempo del Envió 1 Imagen	0.01327s
Imágenes Por Segundo	75.3287FPS

Los resultados para paquetes grandes son muy similares estando cercano a los 75FPS al igual que con la *DMA* sin *SG*. El problema se hace evidente cuando se quiere acceder a paquetes pequeños, ya que se encuentran algunos problemas. El tiempo de acceso a cada descriptor se nota más a cuantos más paquetes se quieran transmitir. Además, existe un offset mínimo entre cada descriptor, por lo que es imposible realizar transferencias por debajo de ese offset mínimo.

En la Figura 5.7, se puede observar que la curva de imágenes por segundo y número de píxeles por transferencia es muy similar, pero en el siguiente apartado se procederá a comparar ambas alternativas de manera más detallada.

5.4.3. Comparativa de los Resultados con las DMA

A continuación se procederá a comparar ambas opciones de *DMA*. La decisión no solo dependerá del número máximo de imágenes por segundo capaz de transmitir, si no, la forma más óptima de transferir los datos de la memoria a la *DMA*.

Si se quiere transmitir en formato *BIP* la imagen la mejor opción es la *DMA* simple, ya que es más sencilla su implementación y es un poco más rápida que la opción con *SG*. Esto se debe que la generación del cubo hipe-espectral almacena el cubo en memoria

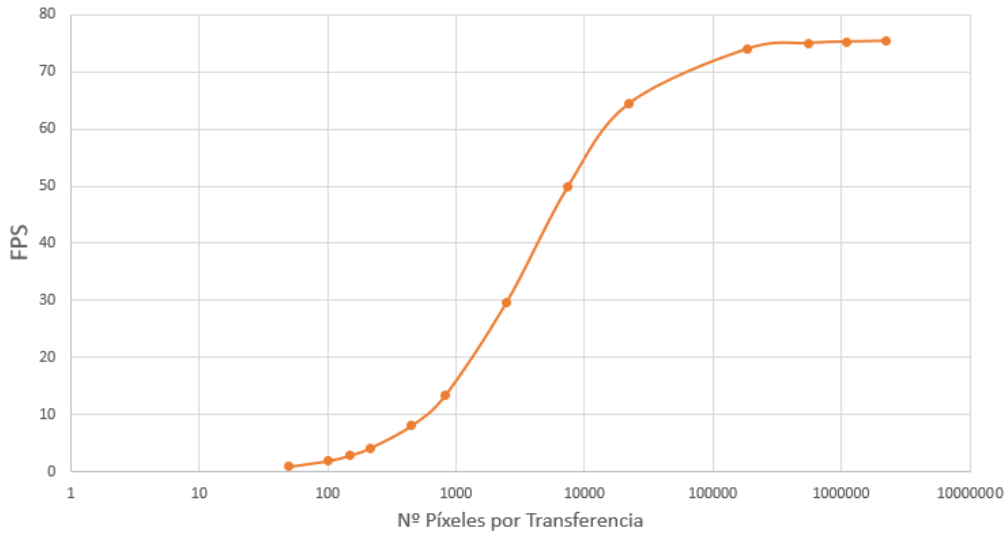

Figura 5.7: Relación FPS con el número de píxeles por transferencia con SG

en este formato. La comparativa entre las velocidades de transmisión de los paquetes se puede ver en la Figura 5.8.

Se puede observar que cuando se transmiten paquetes grandes el número de FPS no se ve afectado significativamente. Sin embargo, para paquetes más pequeños sí que se puede llegar a ver una diferencia de casi 10FPS a favor de la *DMA* simple. Para paquetes más pequeños de 200 píxeles la diferencia de FPS se vuelve a reducir pero en contra de la *DMA* con *SG* juega el tamaño mínimo del paquete.

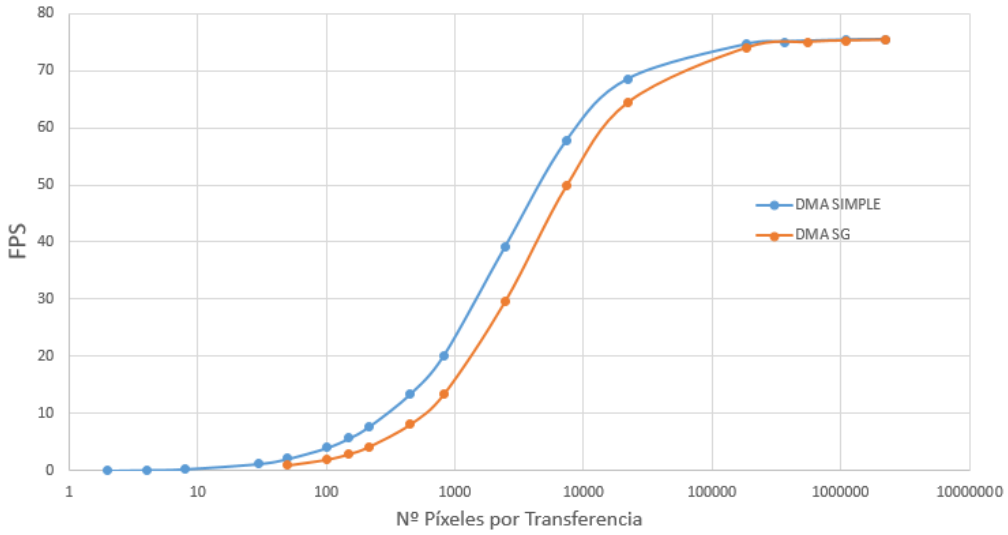

Figura 5.8: Relación FPS con el número de píxeles por transferencia con SG de ambas DMAs

En el caso de querer transmitir las imágenes en formato *BSQ, Band Sequential*, es más

óptima la *DMA* con *SG*. El formato *BSQ* se caracteriza por almacenar de manera secuencial cada una de las bandas, es decir, primero todos los píxeles de la primera banda, y de esta manera, hasta el último píxel de la última banda. Destacar que en este caso el paquete mínimo a enviar son los 50 primeros píxeles y se obtendrían unos FPS más bajos que con la opción de *DMA* simple.

En el caso de querer enviar el cubo hiper-espectral en formato *BSQ* al filtro de medias, habría que comparar si emplear una *DMA* simple y un componente hardware que se encargue de codificar la imagen de formato *BIP* a *BSQ*, ofrece un mayor rendimiento que la *DMA* con *SG*.

En un principio, y tras los analizar los estudios y test realizados, se llega a la conclusión que la *DMA* simple ofrece un mayor rendimiento. Además, el proceso de implementación y de aprendizaje es mucho menor, ya que comprender como funcionan la *DMA* con *SG* y los descriptores es un proceso tedioso.

Además en memoria hay que disponer de un espacio para los descriptores, por lo que es necesario reservar memoria tanto para los descriptores de envió de datos como para los de recepción de los datos. Por lo tanto, en un sistema en el que la memoria va justa para almacenar diversos datos, el uso de una *DMA* con *SG* se complica.

6. CAPÍTULO

Futuras Líneas de Investigación

Tras haber llegado al final de este trabajo de investigación, todavía quedan algunas líneas que investigar y testear para poder llegar a un modelo de sistema embebido funcional para poder implementar en un *ADAS*. Actualmente el desarrollo de procesadores RISC-V, que son procesadores Open-Source, pueden ser otra opción a la hora de seleccionar un *IP Core* de microprocesador para el sistema.

En este trabajo no se ha incluido ningún microprocesador RISC-V debido a que el proceso de diseño basado en una *CPU* RISC-V es complicado. Esto se debe a que Xilinx no ofrece soporte para este tipo de microprocesadores, lo que hace que haya que emplear programas alternativos a los habituales. También, la falta de ejemplos, manuales, y que todavía es un tema que se esta desarrollando en la actualidad, hace que el proceso de aprendizaje para la implementación de diseños sea una ardua tarea.

Cuando el proceso de implementación de microprocesadores RISC-V sea más sencillo y accesible, sería recomendable realizar las mismas pruebas que a los microprocesadores estudiados en este trabajo. Es interesante analizar bien los resultados de los procesadores RISC-V ya que es una opción muy interesante al ser Open-Source.

Un aspecto a tener en cuenta es la organización del cubo hiper-espectral en memoria. En un principio el cubo está almacenado en un formato *BIP, Band Interleved by Pixel*. Otras opciones de almacenamiento son *BSQ, Band Sequential*, o *BIL, Band Interleved by Line*. De entre estas opciones habrá que seleccionar la más adecuada para facilitar el cálculo al filtro de medias para poder facilitar la transmisión de datos y así reducir el número de transferencias *DMA*-Filtro.

El siguiente paso a tomar sería realizar los test de tiempos sobre las *DMAs* cuando se disponga de los *IP Cores* de los aceleradores hardware de clasificación y procesado. Recalcar que el número de imágenes por segundo que se podrá llegar a obtener será menor que el obtenido en los análisis realizados.

Por último, una vez se disponga del sistema completo, habrá que realizar los estudios de consumo de energía con los elementos elegidos de microprocesador y de *DMA*. Habrá que caracterizar el consumo del sistema y realizar las pruebas pertinentes para asegurar su buen funcionamiento.

7. CAPÍTULO

Conclusiones

Tras haber realizado diversos estudios y pruebas, se puede decir que se han llegado a cumplir los objetivos planteados al comienzo del Trabajo Fin de Máster.

Se ha realizado un estudio exhaustivo sobre tres diferentes opciones de microprocesadores disponibles con licencia gratuita para *FPGAs* de Xilinx. Sobre estos microprocesadores se han realizado pruebas para conocer el rendimiento que pueden ofrecer al ejecutar el preprocesamiento de los cubos hiper-espectrales de una cámara snapshot de 25 bandas. Se ha ejecutado la generación del cubo hiper-espectral obteniéndose buenos resultados en algunos de los *SoCs* propuestos.

Aunque no se han obtenido *SoCs* funcionales con los microprocesadores RISC-V, el estudio y la evaluación de las distintas implementaciones ensayadas permiten concluir que es útil para futuros diseños. Sobretodo, el estudio realizado sobre las herramientas de código abierto que son necesarias emplear en el desarrollo de un *SoC* basado en arquitectura RISC-V, permite conocer herramientas alternativas a Vivado que sí incluyen estándares más actuales de lenguajes *HDL*, por lo que se pueden llegar a realizar diseños más complejos y parametrizables.

En el caso de las pruebas realizadas sobre el Cortex-M3 y el Microblaze, el rendimiento que ofrecen esto microprocesadores de pequeño tamaño es inaceptable para la aplicación en cuestión. Los tiempos de ejecución del código de generación de imágenes hiper-espectrales son muy altos, tanto para el código manual como para el generado por MATLAB. Si que se ha obtenido una mejora significativa del código optimizado manualmente con respecto al código generado automáticamente por MATLAB, por lo que

podemos concluir que la reutilización de datos, la optimización del uso de la memoria local y el control detallado sobre las transferencias entre *IP Cores* es esencial si se aspira a ejecutar este tipo de procesamiento sobre dispositivos de rango medio/bajo. Una solución posible para este bajo rendimiento en estos microprocesadores es realizar el preprocesado de la imagen hiper-espectral también con coprocesadores hardware.

Como era de esperar, el máximo rendimiento entre las arquitecturas ensayadas corresponde al caso del Zynq-7000 con microprocesador "*"Hardcore"* . Esto se debe mayoritariamente gracias a diversos módulos que incluye como se ha comentado en el apartado de resultados y la alta frecuencia de reloj.

La gran mejora que se puede ver del tiempo de ejecución entre el código manual y el código generado por MATLAB se debe en gran parte a el control sobre los datos en memoria cache, ya que el tiempo de acceso es mucho menor que a la memoria externa.

Por lo tanto, para poder realizar un *SoC* que de el rendimiento suficiente como para procesar las imágenes en tiempo real es necesario que disponga de multiplicadores hardware y FPU hardware. También, como se ha podido observar en los análisis del Zynq-7000, un módulo de tipo SIMD como es el NEON, facilita la tarea de implementación, ya que al ser capaz de realizar multiplicaciones vectoriales, el número de iteraciones para procesar una imagen hiper-espectral se ve reducido significativamente.

Por otra parte se han evaluado las diferentes opciones para la transferencia de datos. También se han obtenido modelos funcionales y ambos pueden ofrecer un alto rendimiento. Pero, para esta aplicación, la más sencilla *DMA* de programación simple ha resultado ser la más eficiente al ofrece un número más alto de imágenes por segundo. Además, no es necesario reservar espacio en memoria, como sí necesita la opción con *Scatter/Gather* para almacenar la información de los descriptores.

En definitiva, tras observar los resultados obtenidos, las recomendaciones para el diseño de un *SoC* de procesamiento de imágenes hiper-espectrales serían la necesidad de utilizar microprocesadores "Hardcore" de alto rendimiento, la aceleración por hardware de algunas fases nucleares del procesamiento como los filtros espaciales y la corrección espectral y una *DMA* sin *SG*. También es necesario disponer de módulos FPU, multiplicador hardware y SIMD para poder acelerar el código software lo máximo posible.

Uno de los mayores problemas que se ha encontrado es la poca integración entre las diferentes herramientas software. Realizar un diseño para FPGAs de Xilinx es siempre dependiente de Vivado, por lo que debe adaptarse el diseño a los estándares de código HDL que Vivado es capaz de interpretar.

Las herramientas libres como GHDL sí que hacen esta tarea más sencilla, pero para un principiante, el hecho de utilizar Linux o una consola como Mingw puede generar dificultades.

Otro problema es en el caso de los RISC-V ya que no hay un estándar para la programación, además de que es necesario Linux para poder compilar los proyectos. Hay disponible mucho software pero la escasa documentación y la ausencia de soporte supone una limitación seria por el momento.

En lo que respecta al aspecto formativo del trabajo, se han obtenido amplios conocimientos sobre implementación de diversos microprocesadores y *DMAs*, su configuración y la generación de software para estos. Además, se ha trabajado con el módulo NEON, cuyo uso va a ser fundamental en posteriores etapas del sistema embebido para *ADAS*, ya que liberará de carga computacional a la *CPU*.

También se han obtenido conocimientos para realizar la síntesis y simulación de componentes hardware escritos en lenguaje *HDL* con herramientas *Open-Source*. Esto permite utilizar estándares más recientes del lenguaje, por lo que se pueden realizar diseños más complejos. Estas herramientas son muy útiles pero la falta de generación del bitstream, además del aprendizaje previo y la falta de una interfaz gráfica hacen que acceder a ellas sea más complicado.

En mi opinión, se han llegado a satisfacer todos los objetivos planteados con creces. Se han adquirido una serie de conocimientos útiles y que ayudaran en futuras etapas del proyecto. Sobretodo, la exploración de las herramientas alternativas a Vivado para la síntesis, simulación y verificación de *IP Cores*, facilitarán la tarea de implementación de diseños más complejos.

Por último, concluir que aunque se han producido muchos inconvenientes y no ha sido sencillo el desarrollo de este proyecto, los resultados obtenidos han merecido las horas de trabajo dedicadas.

Anexos

A. ANEXO

Diseño de Bloques de los System-On-Chip

A continuación se muestran los diseños de *SoCs* que se han realizado para evaluar y testear los diferentes microprocesadores y DMAs.

- Figura A.1: *SoC* empleado para la evaluación del Cortex-M3.
- Figura A.2: *SoC* empleado para la evaluación del Xilinx Microblaze.
- Figura A.3: *SoC* empleado para la evaluación de las diferentes *DMAs*.

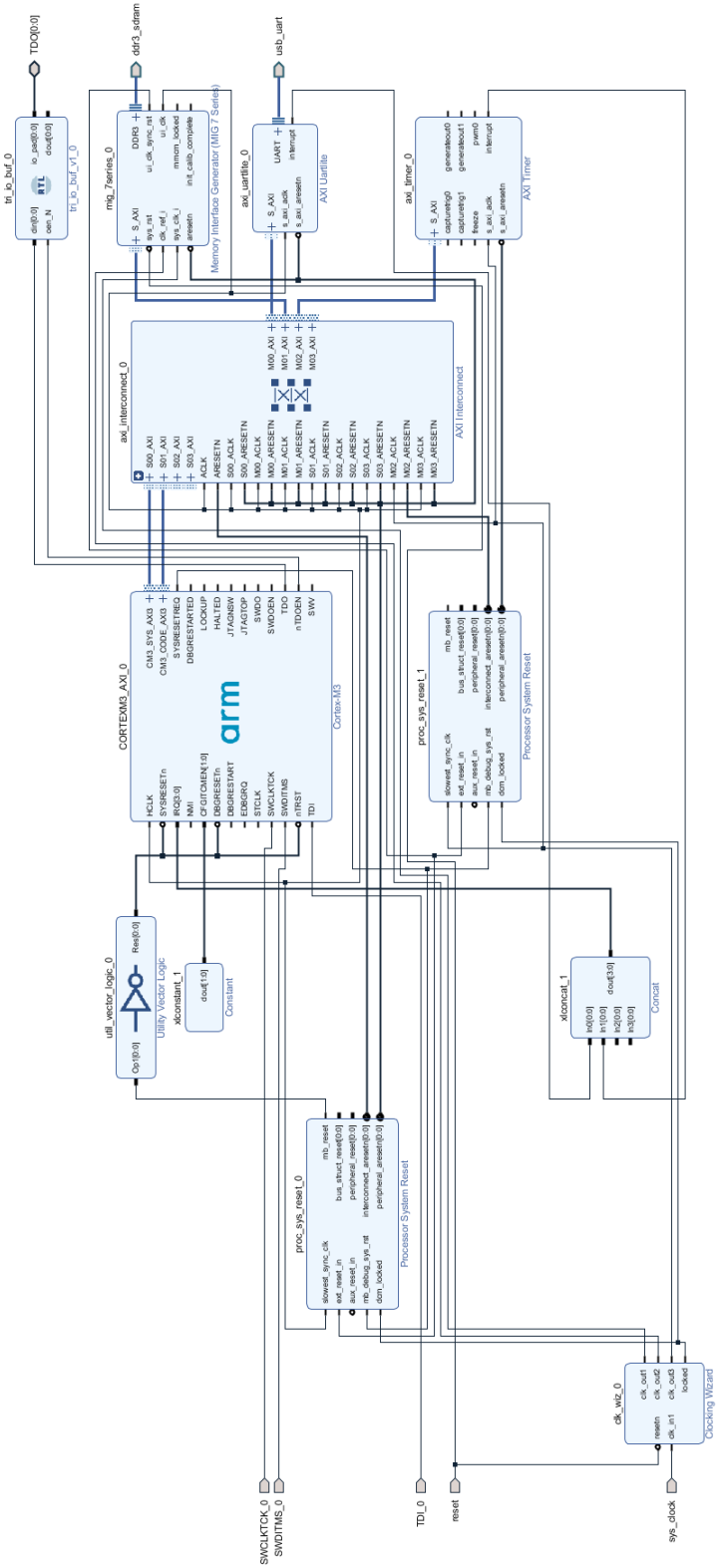

Figura A.1: Esquema del Diseño del SoC con el Cortex-M3

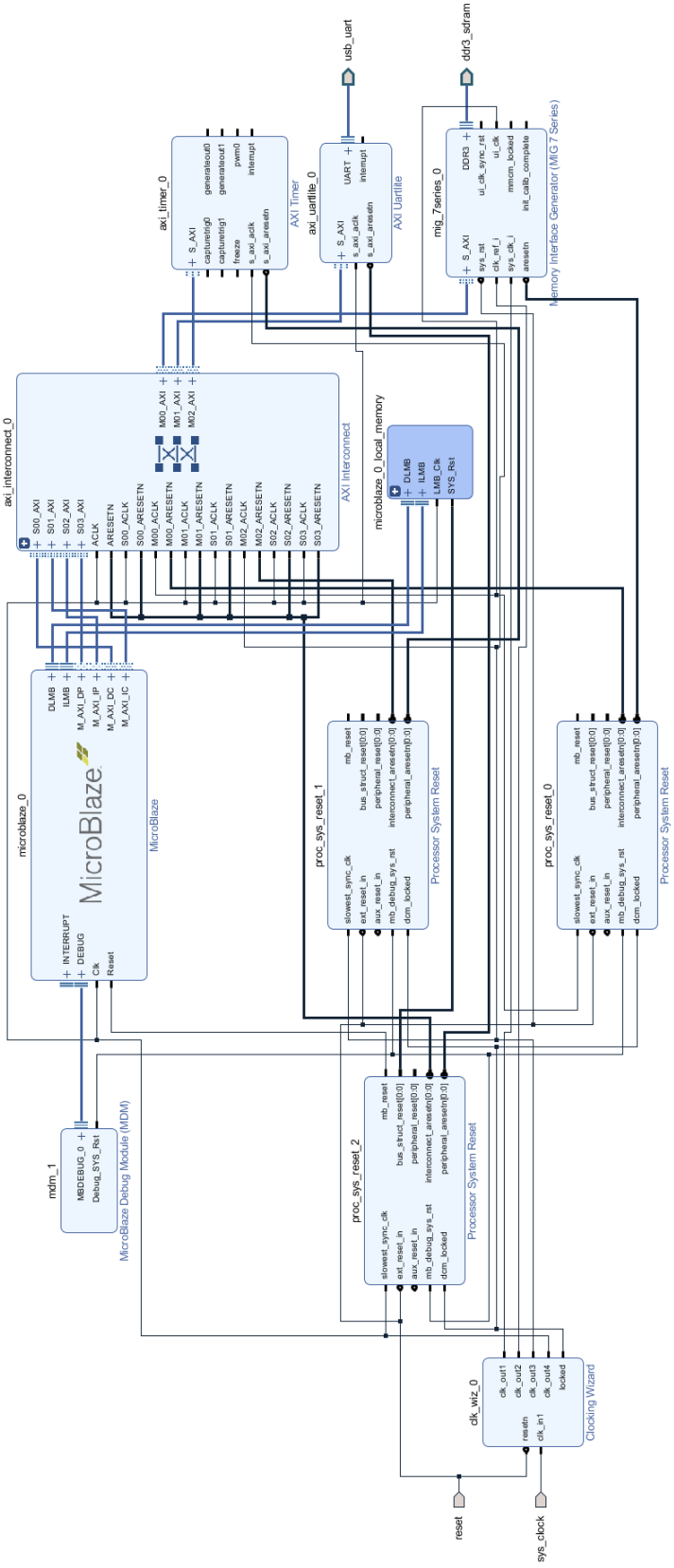

Figura A.2: Esquema del Diseño del SoC con el Xilinx Microblaze

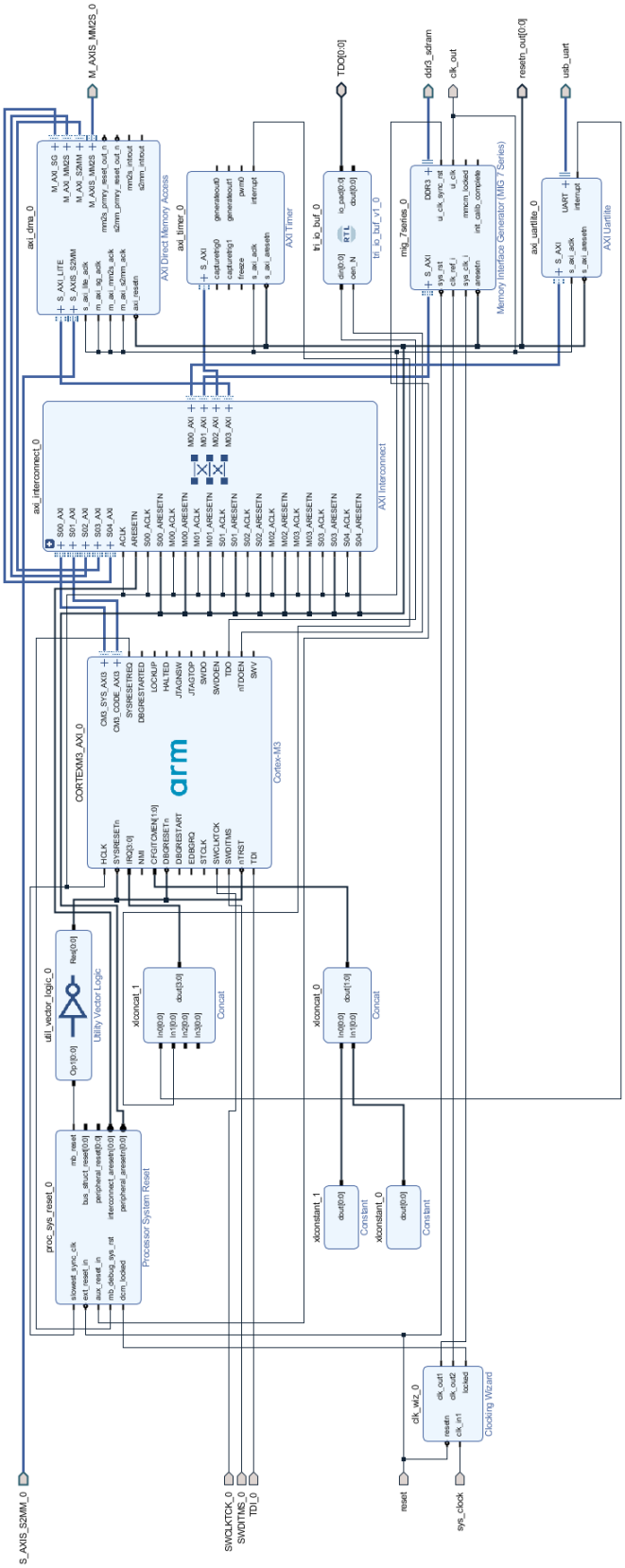

Figura A.3: Esquema del Diseño del SoC con el Cortex-M3 y la DMA

B. ANEXO

Ejemplo Código .core para FuseSoC

A continuación se muestra el fichero top_artyA7.core, que es el tipo de archivo empleado en FuseSoC para realizar el diseño con todos los *IP Cores*.

```
CAPI=2:
# Copyright lowRISC contributors.
# Licensed under the Apache License, Version 2.0, see LICENSE for details.
# SPDX-License-Identifier: Apache-2.0
name: "lowrisc:ibex:top_artya7:0.1"
description: "Ibex example toplevel for Arty A7 boards (both, -35 and -100)"
filesets:
    files_rtl_artya7:
        depend:
            - lowrisc:ibex:ibex_core
            - lowrisc:ibex:fpga_xilinx_shared
        files:
            - rtl/top_artya7.sv
        file_type: systemVerilogSource

    files_constraints:
        files:
            - data/pins_artya7.xdc
        file_type: xdc

parameters:
    # XXX: This parameter needs to be absolute, or relative to the *.runs/synth_1
    # directory. It's best to pass it as absolute path when invoking fusesoc, e.g.
    # --SRAMInitFile=$PWD/sw/led/led.vmem
    # XXX: The VMEM file should be added to the sources of the Vivado project to
    # make the Vivado dependency tracking work. However this requires changes to
    # fusesoc first.
    SRAMInitFile:
```

```
29          datatype: str
30          description: SRAM initialization file in vmem hex format
31          default: "../../../../../examples/sw/led/led.vmem"
32          paramtype: vlogparam
33
34      # For value definition, please see ip/prim/rtl/prim_pkg.sv
35      PRIM_DEFAULT_IMPL:
36      datatype: str
37      paramtype: vlogdefine
38      description: Primitives implementation to use, e.g. "prim_pkg::ImplGeneric".
39
40  targets:
41      synth:
42          default_tool: vivado
43          filesets:
44              - files_rtl_artya7
45              - files_constraints
46          toplevel: top_artya7
47          parameters:
48              - SRAMInitFile
49              - PRIM_DEFAULT_IMPL=prim_pkg::ImplXilinx
50          tools:
51               vivado:
52                  part: "xc7a100tcsg324-1" # Default to Arty A7-100
```

En este fichero se pueden ver los ficheros necesarios para realizar el SoC:

- Ficheros *RTL*:
  - Dependencias: Son los IP Cores de los que depende el SoC. Estos deben estar incluidos en las librerías de FuseSoC para poder trabajar con ellos.
  - Archivos RTL: En este caso solo es uno y define el top del SoC.
- Ficheros de Constrains: Es donde se definen los pines y las conexiones de la placa.

También se indican los parámetros del diseño de *IP Cores*. Esta parte solo será necesario incluirla cuando haya parámetros que necesiten ser definidos o inicializados. En este caso se inicializa la memoria con un archivo .vmen, que es el software del diseño.

La última parte del código se corresponde a los targets para la síntesis e implementación. En este caso se emplea Vivado, y se tiene en cuenta los ficheros que se quiere sintetizar, los parámetros y la FPGA para la cual se va a sintetizar e implementar.

Bibliografía

[1] Mahdi Rezaei. *Computer Vision for Road Safety: A System for Simultaneous Monitoring of Driver Behaviour and Road Hazards*. PhD thesis, University of Auckland, 07 2016.

[2] Marc Tesch. Iot and predictive analytics: Fog and edge computing for industries, January 2018.

[3] Xilinx. Boost software performance on zynq-7000 ap soc with neon application note (xapp1206), 2014.

[4] DGT. Tablas estadísticas. https://www.dgt.es/es/seguridad-vial/estadisticas-e-indicadores/accidentes-30dias/tablas-estadisticas/, 2020.

[5] A Benson, B.C. Tefft, A.M. Svancara, and W.J. Horrey. Potential reduction in crashes, injuries and deaths from large-scale deployment of advanved driver assistance systems. Technical report, AAA Foundation for Traffic Safety, 2018.

[6] V. Nandini, R. Deepak Vishal, C. Arun Prakash, and S. Aishwarya. A review on applications of machine vision systems in industries. *Indian Journal of Science and Technology*, 2016.

[7] Carlos Ramos, Juan Carlos Augusto, and Daniel Shapiro. Ambient intelligencethe next step for artificial intelligence. *IEEE Intelligent Systems*, 2008.

[8] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[9] Pedro F. Felzenszwalb, Ross B. Girshick, David Mcallester, and Deva Ramanan. Object detection with discriminatively trained part based model. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2013.

[10] Amir Tjolleng, Kihyo Jung, Wongi Hong, Wonsup Lee, Baekhee Lee, Heecheon You, Joonwoo Son, and Seikwon Park. Classification of a driver's cognitive workload levels using artificial neural network on ecg signals. *Applied Ergonomics*, 2017.

[11] Abdallah Moujahid, Manolo Dulva Hina, Assia Soukane, Andrea Ortalda, Mounir El Araki Tantaoui, Ahmed El Khadimi, and Amar Ramdane-Cherif. Machine learning techniques in adas: A review. In *Proceedings on 2018 International Conference on Advances in Computing and Communication Engineering, ICACCE 2018*, 2018.

[12] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, 2015.

[13] Microsoft. Microsoft unveils project brainwave for real-time ai. https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave, 2017.

[14] Arish Sateesan, Sharad Sinha, K.G. Smitha, and A.P. Vinod. A survey of algorithmic and hardware optimization techniques for vision convolutional neural networks on fpgas. *Neural Processing Letters*, 53:1–47, 06 2021.

[15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA 2015 - 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.

[16] Kevin Kiningham. Design and analysis of a hardware cnn accelerator. In *University of Stanford*, 2017.

[17] Gorka Velez and Oihana Otaegui. Embedding vision-based advanced driver assistance systems: a survey. *IET Intelligent Transport Systems*, 11, 08 2016.

[18] José Manuel Amigo. *Hyperspectral Imaging*, volume 32. Elsevier, 1 edition, September 2019.

[19] Kento Tajiri and Tsutomu Maruyama. Fpga acceleration of a supervised learning method for hyperspectral image classification. In *Proceedings - 2018 International Conference on Field-Programmable Technology, FPT 2018*, 2018.

[20] Sebastian Lopez, Tanya Vladimirova, Carlos Gonzalez, Javier Resano, Daniel Mozos, and Antonio Plaza. The promise of reconfigurable computing for hyperspectral imaging onboard systems: A review and trends. In *Proceedings of IEEE*, 2013.

[21] Statista. Iot market size worldwide 2017-2025. https://www.statista.com/statistics/976313/global-iot-market-size/.

[22] Michael Armbrust, Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, 04 2010.

[23] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 2016.

[24] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE Access*, 2020.

[25] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. Edge machine learning for ai-enabled iot devices: A review. *Sensors*, 2020.

[26] Congfeng Jiang, Tiantian Fan, Honghao Gao, Weisong Shi, Liangkai Liu, Christophe Cérin, and Jian Wan. Energy aware edge computing: A survey. *Computer Communications*, 2020.

[27] XCUBE. X-cube-ai - ai expansion pack for stm32cubemx - stmicroelectronics. https://www.st.com/en/embedded-software/x-cube-ai.html.

[28] ARM. High-performing ai solutions to transform our digital world – arm. https://www.arm.com/solutions/artificial-intelligence.

[29] TensorFlow. Tensorflow lite para microcontroladores. https://www.tensorflow.org/lite.

[30] PyTorch. Pytorch. https://pytorch.org/.

[31] Jongmin Jo, Sucheol Jeong, and Pilsung Kang. Benchmarking gpu-accelerated edge devices. In *Proceedings - 2020 IEEE International Conference on Big Data and Smart Computing, BigComp 2020*, 2020.

[32] José M. Cecilia, Juan Carlos Cano, Juan Morales-García, Antonio Llanes, and Baldomero Imbernón. Evaluation of clustering algorithms on gpu-based edge computing platforms. *Sensors*, 2020.

[33] Yuan Lei, Peng Luo, Chi Hong Chan, Xiao Huo, Yiu Kei Li, and Mei Kei Ieong. Low power ai asic design for portable edge computing. In *2020 IEEE 15th International Conference on Solid-State and Integrated Circuit Technology, ICSICT 2020 - Proceedings*, 2020.

[34] Maurizio Capra, Riccardo Peloso, Guido Masera, Massimo Ruo Roch, and Maurizio Martina. Edge computing: A survey on the hardware requirements in the internet of things world. *Future Internet*, 2019.

[35] Teng Wang, Chao Wang, Xuehai Zhou, and Huaping Chen. A survey of fpga based deep learning accelerators: Challenges and opportunities. *Proceedings - 21st IEEE International Conference on High Performance Computing and Communications, 17th IEEE International Conference on Smart City and 5th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2019*, 2018.

[36] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going deeper with embedded fpga platform for convolutional neural network. In *FPGA 2016 - Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016.

[37] Xilinx. Vitis ai. https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html, 2020.

[38] Xilinx. Introduction to fpga design with vivado high-level synthesis, 2019.

[39] Mushin Gurel. *A Comparative Study between RTL and HLS for Image Processing Applications with FPGAs*. PhD thesis, University of California, 2016.

[40] Maria Muñoz-Quijada, Luis Sanz, and Hipolito Guzman-Miranda. Sw-vhdl co-verification environment using open source tools. *Electronics*, 2020.

[41] Tristan Gingold. Ghdl: free and open-source analyzer, compiler, simulator and (experimental) synthesizer for vhdl. https://ghdl.github.io/ghdl/about.html.

[42] Olof Kindgren. Fusesoc. https://fusesoc.readthedocs.io/en/latest/index.html.

[43] ARM. Arm cortex-m3 designstart fpga-xilinx edition user guide. https://developer.arm.com/documentation/101483/latest.

[44] Vaibhav Kale. Using the microblaze processor to accelerate cost-sensitive embedded system development. Technical report, Xilinx, 2016.

[45] Xilinx. Zynq-7000 soc data sheet: Overview. Technical report, Xilinx, 2018.

[46] RISC-V. Risc-v. https://riscv.org/exchange/cores-socs/.

[47] SiFive. Sifive. https://scs.sifive.com/core-designer/.

[48] SpinalHDL. Vexrisc-v. https://github.com/SpinalHDL/VexRiscv.

[49] Bluespec. Bluespec. https://bluespec.com/supported-cores/.

[50] Pulp Platform. Pulp platform. https://pulp-platform.org/.

[51] lowRISC. Ibex. https://ibex-core.readthedocs.io/en/latest/01_overview/index.html.

[52] Iryna Svyd, Oleksandr Maltsev, Liliia Saikivska, and Oleg Zubkov. Review of seventh series fpga xilinx. In *25th Conference of Theoretical and Applied Aspects of Device Development on Microcontrollers and FPGAs*, 2019.

[53] SiFive. *SiFive E24 Manual*, 20g1.03.00 edition, June 2020.