ULPGC
Universidad de
Las Palmas de
Gran Canaria

Instituto Universitario de
Microelectrónica Aplicada

IUMA

Máster Universitario en
Electrónica y Telecomunicación Aplicadas (META)

Trabajo de Fin de Máster

Análisis de las prestaciones de una capa convolutiva de una red neuronal sintetizada en una FPGA mediante herramientas de alto nivel

Autor: Adrián Rodríguez Molina
Tutor(es):
Sebastián López Suárez (Tutor)
Romén Neris Tomé (Cotutor)
Yubal Barrios Alfaro (Cotutor)

Fecha: septiembre 2021

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

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

Máster Universitario en Electrónica y Telecomunicación Aplicadas (META)

Trabajo de Fin de Máster

Análisis de las prestaciones de una capa convolutiva de una red neuronal sintetizada en una FPGA mediante herramientas de alto nivel

HOJA DE FIRMAS

Alumno/a: Adrián Rodríguez Molina Fdo.:

Tutor/a: Sebastián López Suarez Fdo.:

Cotutor/a: Romén Neris Tomé Fdo.:

Cotutor/a: Yubal Barrios Alfaro Fdo.:

Fecha: septiembre 2021

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

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

Máster Universitario en Electrónica y Telecomunicación Aplicadas (META)

Trabajo de Fin de Máster

Análisis de las prestaciones de una capa convolutiva de una red neuronal sintetizada en una FPGA mediante herramientas de alto nivel

HOJA DE EVALUACIÓN

Calificación: ....

Presidente: Fdo.:

Secretario: Fdo.:

Vocal: Fdo.:

Fecha: septiembre 2021

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

Índice general

Índice de figuras v

Índice de cuadros vii

Bloques de Código ix

1 Introducción 1

1.1 Antecedentes 1

1.1.1 FPGA 2

1.1.2 Remote Sensing 4

1.1.3 Sensores de vídeo en satélites 5

1.1.4 Redes Neuronales Convolucionales 7

1.1.4.1 Mobilenet 9

1.1.5 Proyecto VIDEO 10

1.2 Objetivos 11

1.3 Peticionario 12

1.4 Estructura del Documento 12

2 Estado del Arte 15

2.1 Remote Sensing 15

2.2 Redes Neuronales Convolucionales en FPGAs 18

2.2.1 Motivo de uso de FPGAs 18

2.2.2 Implementaciones de una CNN en FPGAs 19

2.2.3 Arquitecturas de la capa convolucional 20

3 Solución Propuesta 23
3.1 Capa Convolucional . . . 24
3.1.1 Parámetros de la capa convolucional . . . 25
3.1.2 Cálculo de la convolución . . . 26
3.1.3 Capa Convolucional con Imágenes . . . 28
3.2 Hardware al que se orienta la solución . . . 30
3.2.1 FPGA XCKU040-2FFVA1156E . . . 30
3.2.2 Kintex UltraScale FPGA KCU105 Evaluation Kit . . . 31
3.3 Lenguajes . . . 33
3.3.1 Python . . . 34
3.3.2 Lenguaje C++ . . . 35
3.3.2.1 Adaptación a Vitis HLS . . . 35
3.4 Xilinx Vitis HLS . . . 36
3.4.1 Uso de la herramienta . . . 36
3.4.2 Librerías HLS . . . 39
3.4.2.1 *Arbitrary Precision (AP) Data Types* . . . 39
3.4.2.2 Directivas HLS . . . 40

4 Diseño del Bloque IP 45
4.1 Transcripción de Python a C++ . . . 46
4.1.1 Estructura del Código en Python . . . 46
4.1.2 Modificaciones realizadas . . . 47
4.1.3 Resultados de la capa en C++ . . . 49
4.2 Adaptación del código a Vitis HLS . . . 50
4.2.1 Cambios estructurales de la capa . . . 50
4.2.1.1 Gestión de la memoria . . . 50
4.2.1.2 Orden de Iteración . . . 52
4.2.1.3 *Padding* . . . 53
4.2.2 Mejoras de las prestaciones de la capa . . . 54
4.3 Soluciones Generadas . . . 54
4.3.1 Capa *Line* . . . 55
4.3.2 Capa *Block* . . . 56
4.4 Banco de pruebas . . . 57
4.4.1 Generación de la capa . . . 57

4.4.2 Comparación de los resultados de la capa 59
4.4.3 Proceso de generación de las soluciones 60

5 Resultados Obtenidos **61**
5.1 Parámetros de Entrada 61
5.2 Comparación de Resultados 63
5.2.1 Variación del tamaño de las imágenes 63
5.2.1.1 Latencia del Proceso 64
5.2.1.2 Consumo de recursos de las soluciones 66
5.2.2 Variación de las dimensiones de los pesos 68
5.2.2.1 Latencia del Proceso 68
5.2.2.2 Consumo de recursos de las soluciones 70
5.2.3 Variación del *stride* de la capa 71
5.2.3.1 Latencia del Proceso 72
5.2.3.2 Consumo de recursos de las soluciones 74
5.3 Tipo de datos 76
5.3.1 Resultados de latencia obtenidos 77
5.3.2 Consumo de recursos de las soluciones 78

6 Conclusiones y Líneas Futuras **83**
6.1 Conclusiones 83
6.2 Líneas Futuras 85

Bibliografía **87**

Índice de figuras

1.1 Diagrama de la estructura de una FPGA [1] 3
1.2 Ejemplo de efecto *Parallax* [10] 6
1.3 Ilustraciones del Satélite WorldView-2 [12] 7
1.4 Estructura de la red Mobilenet 10

2.1 a) es la imagen de la cámara multiespectral, b) la de la cámara pancromática y c), d) y e) son los resultados después de aplicar distintos métodos de *Pan Sharpening* [25]. 17

3.1 Esquema de una capa convolucional 24
3.2 Esquema de pesos y filtros de la capa convolucional 25
3.3 Proceso de Convolución: primer paso 27
3.4 Proceso de Convolución: segundo paso 28
3.5 Proceso de Convolución para tres dimensiones [40] 29
3.6 Entrada y Salida de la capa convolucional [41] 29
3.7 Detección de las líneas de la carretera [42] 30
3.8 KCU105 Evaluation Kit [43] 32
3.9 Resultados de la síntesis en Vitis HLS 38
3.10 Herramienta de análisis de Vitis HLS 39

4.1 Estructura del código de Python. 46
4.2 Gráfico con la estructura de la capa en C++. 49
4.3 Almacenamiento fila a fila de la imagen. 51
4.4 Desplazamiento de la fila a reutilizar hacia la primera posición de memoria. . 52

5.1 Comparación entre las latencias totales de las dos capas 66

5.2 Comparación entre las latencias totales de las dos capas 70
5.3 Comparación del consumo de recursos de ambas capas al variar las dimensiones de los pesos. 72
5.4 Comparación entre las latencias totales de las dos capas 74
5.5 Comparación del consumo de recursos de ambas capas al variar los *strides*. . 76
5.6 Comparación entre la latencia total de las capas con punto flotante y con punto fijo 78
5.7 Variación de los recursos consumidos por la implementación de las soluciones 5 y 6 al pasar de punto flotante a punto fijo. 80

Índice de cuadros

3.1 Recursos de la FPGA XCKU040-2FFVA1156E [44] 31

5.1 Parámetros de entrada de las capas 62
5.2 Parámetros del primer grupo de resultados 64
5.3 Latencias obtenidas de las capas *Line* (azul) y *Block* (verde) al variar el tamaño de la imagen 65
5.4 Recursos utilizados por las capas *Line* (azul) y *Block* (verde) al variar los tamaños de la imagen (con o sin *padding*) 67
5.5 Diferencias entre los recursos utilizados de ambas capas al variar el tamaño de la imagen (con y sin *padding*) 67
5.6 Parámetros del segundo grupo de resultados 68
5.7 Latencias obtenidas de las capas *Line* (azul) y *Block* (verde) al variar los pesos 69
5.8 Recursos utilizados por las capas *Line* (azul) y *Block* (verde) al variar los pesos 71
5.9 Parámetros del tercer grupo de resultados 73
5.10 Latencias obtenidas de las capas *Line* (azul) y *Block* (verde) al variar los *strides* 73
5.11 Recursos utilizados por las capas *Line* (azul) y *Block* (verde) al variar el desplazamiento de los pesos 75
5.12 Parámetros de entrada de las capas modificadas 77
5.13 Latencias obtenidas de las capas *Line* (azul) y *Block* (verde) al variar el tamaño de la imagen 77
5.14 Recursos utilizados por las capas *Line* (azul) y *Block* (verde) al variar el tipo de dato 79

Bloques de Código

4.1. Convolución en Python 47
4.2. Iteración de la convolución en Python 52
4.3. Estructura de la capa Line 55
4.4. Estructura de la capa Block 56

Resumen

El área del *Remote Sensing* se basa en usar dispositivos equipados con sensores que se desplazan a cierta altura de la superficie terrestre para tomar información sobre el planeta. Dentro de este campo, uno de las plataformas más usadas son los satélites orbitales que, junto con dispositivos de procesamiento a bordo, permiten aplicar algoritmos sobre los datos para potenciales aplicaciones en tierra.

Este Trabajo de Fin de Máster se ha realizado en el contexto del proyecto VIDEO (*Video Imaging Demonstrator for Earth Observation*), un proyecto del programa Horizon 2020 que tiene como objetivo desarrollar un sensor de vídeo que irá a bordo de un satélite y una cadena de procesamiento que se servirá de los datos obtenidos por ese sensor.

Se parte de la capa convolucional de una red neuronal que irá implementada en una FPGA a bordo del satélite. El objetivo de este trabajo es analizar como afectan la variación de los parámetros de entrada de la capa a la latencia y el consumo de recursos de la FPGA.

Para conseguir dicho objetivo, se ha transcrito el código de la capa convolucional de Python a C++ y se han generado dos capas convolucionales con una estructura similar. Una vez hecho esto, se han generado varias configuraciones variando los parámetros de entrada de las capas. Con la ayuda de la herramienta de Síntesis de Alto Nivel de Xilinx, Vitis HLS, se han obtenido los resultados de sintetizar las distintas configuraciones de las capas.

A la vista de los resultados de variar el tamaño de la imagen, aplicándole *padding* o no, las dimensiones de los pesos y el desplazamiento de los pesos sobre la capa, se concluye que todos los parámetros afectan en mayor o menor medida a la implementación de la capa en el dispositivo *hardware*. Los mejores resultados de latencia se consiguen cuando el tamaño de la imagen es el más pequeño de los analizados, mientras que los peores se dan cuando la relación entre el tamaño de los pesos y el desplazamiento de los mismos aumenta.

En términos de recursos lógicos, modificar el tipo de dato de punto flotante a punto fijo ha demostrado ser el cambio que más reduce el consumo y con el que se ocupa menos lógica en el dispositivo.

Abstract

The field of Remote Sensing is based on the use of devices equipped with sensors that move at a certain altitude above the Earth's surface to collect information about the planet. Within this field, one of the most widely used platforms are orbital satellites, which together with on-board processing devices, allow algorithms to be applied to the data acquired to be used on potential applications on ground.

This Master's Thesis is done in the context of the VIDEO project (Video Imaging Demonstrator for Earth Observation), a project of the Horizon 2020 program that aims to develop a video sensor that will be on-board a satellite and a processing chain that will use the data obtained by that sensor.

It is based on the convolutional layer of a neural network that will be implemented on an FPGA on-board the satellite. The goal of this work is to analyze how the variation of the input parameters of the layer affects the latency and resources consumption of the FPGA.

To reach this goal, the convolutional layer source code has been transcribed from Python to C++ and two convolutional layers with a similar structure have been generated. Once this was done, several configurations were generated by varying the input parameters of the layers. With the help of the Xilinx High-Level Synthesis tool, Vitis HLS, the results of synthesizing the different layer configurations have been obtained.

In view of the results of varying the image size, applying padding or not, the dimensions of the weights and the strides of the layer, it is concluded that all parameters affect to a greater or lesser extent to the implementation of the layer on the hardware device. Best latency results are achieved when the image size is the smallest of the ones under analysis, while the worst ones appear when the ratio between the size of the weights and the strides is increased.

In terms of logic resources, changing the data type from floating-point to fixed-point has proven to be the change that reduces consumption the most and takes up the least amount of logic on the device.

Agradecimientos

Me gustaría agradecer a mis tutores el trabajo que han hecho al guiarme durante el desarrollo de este trabajo, a los compañeros del laboratorio de DSI del IUMA por resolver mis dudas y a mi pareja, amigos y familia por apoyarme y animarme durante el tiempo que he estado trabajando en este proyecto.

CAPÍTULO 1: Introducción

1.1 Antecedentes

A la hora de estudiar un objeto, la herramienta que aporta información más fácilmente interpretable para el ser humano son las imágenes, ya que son analizables visualmente. Dando un paso más allá, si se quiere obtener una mayor cantidad de datos se usan vídeos, que ofrecen además información temporal del objeto que se quiere investigar, todo ello sumado a la información espacial proporcionada por una imagen. Este mismo razonamiento se aplica cuando se quieren estudiar la Tierra u objetos que se encuentren sobre la superficie de esta.

Para obtener una cantidad de información relevante de la superficie terrestre, las imágenes se deben tomar desde una cierta distancia con el objetivo de que los sensores que

se usan puedan cubrir un área significativa, sin descuidar la resolución por píxel. La solución a esto es equipar los sensores en dispositivos que se puedan elevar de la superficie una distancia considerable, ya sea dentro de las capas más bajas de la atmósfera como son los aviones o llegando a estar en órbita alrededor del planeta, como los satélites artificiales.

Si bien en el primero de los casos se puede recoger la información una vez el vehículo aterrice o incluso en tiempo real, en función de la arquitectura de procesamiento embarcada, los satélites están diseñados para orbitar durante grandes periodos de tiempo, por lo que es necesario transmitir la información a Tierra para poder procesarla. Los canales que se usan para realizar esta transmisión cuentan con muy poco ancho de banda, por ello requieren de una compresión de los datos a bordo del satélite para ser transmitidos de forma eficiente, compresión que al introducir pérdidas puede inutilizar los datos para algunas de las aplicaciones de interés en tierra.

Para solucionar el problema anteriormente descrito, con el avance de la tecnología se ha optado por introducir dispositivos de cómputo directamente a bordo del satélite, dispositivos que deben cumplir con una serie de requisitos como son la resistencia a la radiación, consumo de potencia limitada y sobre todo que sean dispositivos lo suficientemente estudiados y probados para asegurarse de que no fallen ni necesiten reparaciones, ya que esto es inviable en un satélite.

Este Trabajo de Fin de Máster se centra en desarrollar parte de una red neuronal que se ejecutará a bordo de un satélite y que se implementará en una **FPGA**, que es el dispositivo que se ha elegido en el proyecto en el que se engloba este trabajo para realizar las tareas de cómputo. Esta red neuronal se propone para ser embarcada en un satélite con el objetivo de hacer un seguimiento de objetivos en tiempo real, lo cual sería inviable si la información adquirida por el sensor se realizara en las estaciones ubicadas en tierra.

1.1.1 FPGA

Una FPGA (de las siglas en inglés *Field Programmable Gate Array*) es un dispositivo lógico programable; es decir, un tipo de circuito integrado que puede ser usado para implementar cualquier tipo de circuito integrado modificando las conexiones entre las puertas

lógicas disponibles.

La estructura de estos dispositivos está normalmente organizada de forma que recuerda a islas (Figura 1.1), en la que cada "isla" es un elemento diferente con una función específica. Dichos elementos se detallan a continuación:

- **Elementos lógicos:** partes programables del dispositivo que pueden componer cualquier circuito lógico. Se distinguen *look-up tables* (LUT), multiplexores (MUX), y elementos para generar puertas lógicas genéricas (GAT). Además, se incluyen componentes capaces de registrar las señales lógicas, siendo un ejemplo de esto los *Flip-Flops*.

- **Elementos de entrada y salida:** son bloques que conectan los pines de entrada y salida con las interfaces del dispositivo. También puede incluir sistemas de conexiones estándar como es el PCI.

- **Interconexiones:** incluye los canales de conexión entre los distintos elementos del sistema, los bloques de conexión, que sirven de interfaz entre las interfaces y los elementos lógicos y de entrada y salida; y los *switch blocks*, que se usan para modificar el camino que siguen las señales. Todos estos elementos se usan para comunicar los distintos componentes del dispositivo.

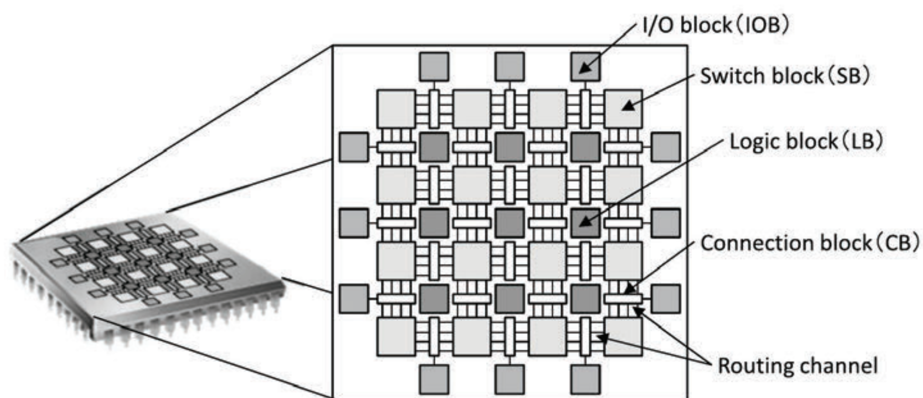


Figura 1.1: Diagrama de la estructura de una FPGA [1]

Aparte de estos componentes, existen otros elementos necesarios para hacer funcionar correctamente el conjunto del dispositivo, y en FPGAs comerciales también es común encontrar bloques con un funcionamiento específico como pueden ser bloques de memoria,

multiplicadores o procesadores integrados en el mismo chip [1].

Debido a las prestaciones que ofrecen, las FPGAs suelen compararse con los dispositivos ASICs, siglas en inglés de *Application Specific Integrated Circuits*, que son circuitos integrados diseñados y fabricados para una aplicación específica. A la hora de realizar la tarea para la que están diseñados, estos ofrecen unas prestaciones en términos de latencia, superficie ocupada y de consumo de potencia superiores a los de una FPGA. Sin embargo, una vez fabricado, un ASIC no puede modificarse. Esto hace que los costes de diseño y el *Time to Market* de una FPGA, unidos a una reducción del gap computacional en los últimos años respecto a los ASICs, la coloquen como una opción más viable para ciertas aplicaciones, así como para la investigación [2].

Es esta posibilidad de reconfigurar el sistema lo que hace que cada vez más se estén usando las FPGAs como dispositivo computacional a bordo de los satélites ya que, si fuera necesario algún cambio del sistema una vez que el dispositivo estuviera en órbita, este se podría realizar *on-the-fly*. Sin embargo, muchos de estos dispositivos no están preparados para los niveles de radiación presentes en el espacio [3]. Siendo conscientes de que las FPGAs son cada vez más usadas en misiones espaciales [4] [5], han desarrollado variantes de estos dispositivos que son tolerantes a la radiación espacial. Un ejemplo de esto es la FPGA de Xilinx **RT Kintex UltraScale**, que cuenta con certificación que indica que es capaz de resistir las condiciones en las que se encontraría si estuviera a bordo de un dispositivo en el espacio [6].

1.1.2 Remote Sensing

Al proceso de estudiar la Tierra obteniendo información de esta a través de distintos tipos de sensores, instalados en satélites o vehículos aéreos, se le conoce como ***Remote Sensing***, aunque este no es el único significado de este término. Dichos sensores pueden ser ópticos, acústicos o basados en microondas y al ir instalados en elementos que se encuentran a gran distancia de la superficie, no se busca tanto el obtener información sobre un punto concreto sino estudiar un área determinada, analizando el contenido de esta a través de diferentes algoritmos o aplicaciones, como detección de objetivos, clasificación o *unmixing*, en el caso de sensores multi- o hiperespectrales.

De los tipos de sensores que se pueden usar en *Remote Sensing*, en este trabajo se tratarán únicamente los sensores ópticos, que se suelen usar para aplicaciones tales como el análisis de los cultivos en agricultura, análisis topográfico, estudio de la meteorología, análisis del clima y estudio de los cambios de este, detección y seguimiento de fenómenos naturales, etc. [7]. Dentro de los sensores ópticos, el trabajo se enfocará únicamente en los sensores de vídeo.

1.1.3 Sensores de vídeo en satélites

Uno de los usos principales que se les ha dado a los satélites es el de obtener información sobre la Tierra a través de diferentes sensores, como son los radiómetros o sensores basados en microondas. Estos sensores se han usado desde los años 60 para obtener datos sobre el comportamiento del clima o para analizar la orografía del terreno [8]. Sin embargo, hubo que esperar hasta 2013 para que ***SkyImaging*** (USA) lanzara el **SKYSAT-1**, el primer satélite artificial capaz de grabar vídeos en alta resolución [9].

Los sensores de vídeo para satélites comparten las mismas características que los sensores de imágenes. De estas características, destacan la alta **resolución espacial** de los mismos, que debido a la distancia entre la superficie que se está captando y el sensor se suele designar no solo como el número de píxeles, sino la cantidad de metros de la superficie a la que equivale cada píxel; la **resolución espectral**, es decir, el número de bandas del espectro electromagnético que es capaz de detectar el satélite; y el **campo de visión**, o **FOV** por sus siglas en inglés, que se define como el ángulo de captura del sensor del satélite, ángulo que afecta directamente al área en kilómetros que cubre cada imagen del satélite.

Además de las características de un sensor de imagen convencional, los sensores de vídeo cuentan con una mayor resolución temporal. A la hora de embarcarlo en un satélite, esta mayor resolución temporal no puede limitarse únicamente a tomar un mayor número de imágenes por segundo, es decir, a aumentar su *frame rate*, sino que además se debe tener en cuenta el movimiento del satélite con respecto a la Tierra. Por esto, los satélites equipados con sensores de vídeo deben contar con dispositivos de seguimiento y estabilización [9], para conseguir mantener el foco sobre un área concreta el tiempo suficiente como para poder grabar un vídeo a un *frame rate* constante.

Otro de los problemas específicos con los que se encuentran los satélites al usar sensores de imagen o vídeo es el efecto *Parallax*. Este efecto ocurre cuando se toman imágenes de una superficie usando un ángulo muy grande de incidencia, lo que provoca que se interprete que algunos píxeles están desplazados en relación a la posición real de la información que se está capturando. Un ejemplo de esto es el representado en la Figura 1.2, en la que se da un ejemplo de una foto tomada desde un satélite (malla morada) que muestra cómo en la imagen se obtiene que existe una nube en un píxel **T'**, de lo que se deduce que la nube se encuentra en la posición **I** de la Tierra. Sin embargo, en la imagen se ha producido efecto *Parallax*, ya que la nube debería encontrarse en el píxel **B'**, estando ésta en el lugar B de la Tierra [10].

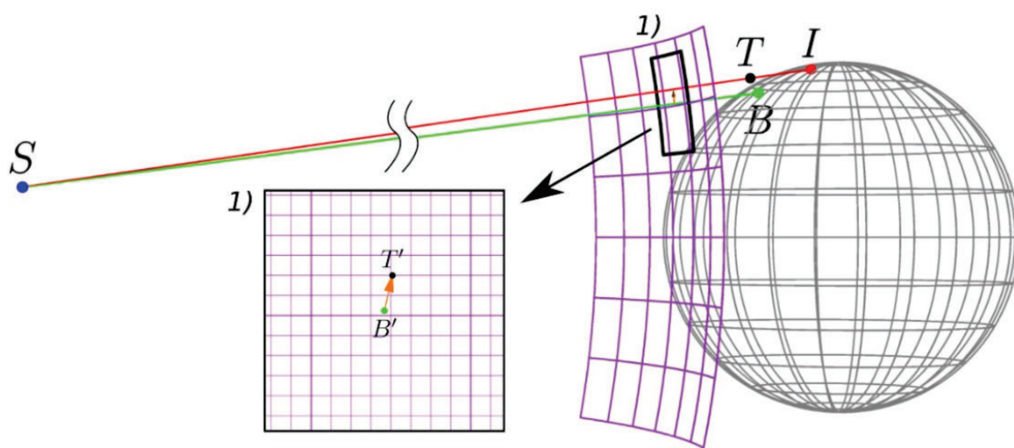


Figura 1.2: Ejemplo de efecto *Parallax* [10]

En cuanto a los tipos de sensores, la principal diferencia se encuentra en la resolución espectral de los mismos, yendo desde sensores pancromáticos, que solo detectan la cantidad de luz reflejada, a sensores multiespectrales que son capaces de detectar alrededor de una decena de bandas y en los que se puede incluir las convencionales cámaras RGB. Estos sensores traen consigo otra diferencia, ya que aquellos con un menor número de bandas son capaces de captar una mayor información espacial y suelen tener mayor resolución. Con el avance de la tecnología en los satélites, se ha logrado solucionar este problema usando fusión de imágenes, que permite aumentar la resolución de los sensores multiespectrales combinando las imágenes tomadas con las obtenidas de un sensor pancromático, como es el caso del satélite **WorldView-2 (WV2)** [11] (Figuras 1.3).

El instrumento que se muestra en la Figura 1.3a cuenta con un sensor pancromático con

(a) Sensor del World View 2: WV110

(b) Satélite World View 2

Figura 1.3: Ilustraciones del Satélite WorldView-2 [12]

una resolución espacial de 0,46 m por píxel y un sensor multiespectral, con una resolución espacial de 1,8 m por píxel. El sensor multiespectral cuenta con ocho bandas: 400-450 nm, 450-510 nm, 510-580 nm, 585-625 nm , 630-690 nm, 705-745 nm, 770-895 nm y 860-1040 nm. Ambos sensores son capaces de almacenar 11 bits de información por píxel. En cuanto a la óptica se refiere, cuenta con una apertura de 1,1 m y una distancia focal de 13,3 m.

1.1.4 Redes Neuronales Convolucionales

Una red neuronal artificial es una cadena de procesamiento formada por distintos nodos que intentan simular las neuronas humanas y que realizan una actividad determinada en el proceso de cómputo. Estos nodos están interconectados entre sí y se transfieren información entre ellos mediante enlaces en la red. La diferencia entre distintas redes neuronales reside en las características de estos nodos, a los que también se les conoce como capas.

Se diferencian principalmente dos tipos de redes neuronales: las de una *single-layer* y las de *multi-layer*. Las redes *single-layer* o de una sola capa se caracterizan porque las entradas de la red están directamente unidas con la salida usando una variación generalizada de una función lineal. A este tipo de redes se les conoce también como *Perceptron* y va-

rían entre ellas dependiendo de las características de la función lineal que se le aplica a la entrada.

Las redes *multi-layer* se caracterizan por contener más de una capa de cómputo. Las capas intermedias entre la entrada y la salida de este tipo de redes se conoce como "capas ocultas", debido a que, a diferencia de las de una sola capa, los cómputos que se hacen en ellas no son fácilmente distinguibles por el usuario de la red. Unos ejemplos de este tipo de redes son las *Restricted Boltzmann machines (RBMs)*, las *Recurrent Neural Networks (RNNs)* o las *Convolutional Neural Networks (CNNs)* [13].

Las redes neuronales convolucionales son redes cuya arquitectura esta inspirada en el comportamiento del cortex visual de un gato. Este tipo de redes se usan normalmente para la detección de objetos a partir de imágenes o para la clasificación de las mismas [13].

Una red neuronal convolucional está formada generalmente por las siguientes cinco capas: una capa de entrada, en la que se realiza un preprocesado de la entrada; una capa convolucional, de la que recibe su nombre y en la que se aplican los pesos de la capa a través de una convolución a los datos de entrada; una capa con una función de activación, que realiza un mapeado no lineal de la salida de la capa anterior; una capa de *pooling* que reduce el *overfitting* del proceso; y un último grupo de capas, conocidas como capas *dense* a las que se conectan todas las neuronas de las capas anteriores y que se usan para calcular la probabilidad de que los datos analizados pertenezcan a una categoría u otra [14].

Este tipo de redes se aplican en un gran número de campos tales como detección de sentimiento del diálogo (*speech-emotion recognition*) [15], detección de dispositivos no autorizados en una red [14] o incluso la clasificación de datos encriptados obtenidos del flujo de información de una red [16].

Sin embargo, debido a las características del proceso de convolución y a los buenos resultados que ofrecen, el principal uso que se le da a estas redes es en el tratamiento de imágenes. Dentro de este campo, se pueden encontrar trabajos sobre el reconocimiento facial o incluso el reconocimiento de la expresión facial [17]; o la clasificación de imágenes relacionadas con lesiones en la piel con el objetivo de detectar aquellas que estén relacionadas con el cáncer [18], por dar dos ejemplos de los muchos que hay.

1.1.4.1 Mobilenet

Después de implementar y analizar varias arquitecturas de redes neuronales convolucionales, en el proyecto dentro del que se desarrolla este trabajo se eligió usar la red **Mobilenet** [19] para realizar el proceso de detección. Esta red fue diseñada con el objetivo de reducir el coste computacional y el tamaño de las redes neuronales convolucionales, para que estas pudieran ser implementadas en dispositivos móviles o en sistemas empotrados con unas capacidades de computación limitadas, como son los que se encuentran en los coches eléctricos; sin reducir demasiado en la precisión de la red.

Para conseguir esto, se han realizado dos modificaciones en una red neuronal convolucional convencional. La primera de las modificaciones consiste en dividir el proceso de convolución en dos, de los que se obtendrá un resultado equivalente al ejecutarlos en serie. Estos procesos se conocen como convolución *depthwise*, en la que se aplica una convolución con varios filtros de tres dimensiones que afecta a todas las bandas de la imagen, obteniendo tantas salidas de dos dimensiones como filtros se hayan usado; y convolución *pointwise*, en el que se usan filtros de $1 \times 1 \times nf$, donde nf es el número de filtros que se usó en la convolución *depthwise*. Una vez realizado este último proceso se obtiene una salida bidimensional.

La segunda de las modificaciones tiene que ver con dos hiperparámetros que se le añaden a la red: α y ρ. Si bien la arquitectura base de la **Mobilenet** ya es pequeña y cuenta con poca latencia, añadiendo estos hiperparámetros se consigue adaptar la red a aplicaciones con unas restricciones mayores. A α se le denomina también como *Width Multipier*, debe tener un valor comprendido entre cero y uno, siendo siempre mayor estricto que 0, y permite reducir de manera uniforme el tamaño de todas las capas de la red. De esta manera, transforma el número de canales de entrada de M a αM y los de salida de N a αN, consiguiendo una reducción del coste computacional y del número de parámetros de aproximadamente α^2.

El segundo de los hiperparámetros, ρ se le denomina también *Resolution Multiplier* e influye en la resolución de la imagen de entrada, reduciendo esta imagen en el factor especificado. Cuenta con el mismo rango de valores que α y consigue una reducción del coste

computacional en un factor de ρ^2.

Se considera arquitectura base de la red cuando ambos hiperparámetros tienen valor uno y hay que tener en cuenta que si ya la red base introduce pérdidas en la precisión, variar los valores de estos parámetros aumentará también dichas pérdidas. La estructura de la red cuenta con 28 capas, si se considera la convolución *depthwise* y *pointwise* como capas independientes, y presenta la distribución que se muestra en la Figura 1.4.

Figura 1.4: Estructura de la red Mobilenet

1.1.5 Proyecto VIDEO

El proyecto **VIDEO**, siglas de *Video Imaging Demonstrator for Earth Observation* [20], es un proyecto europeo financiado mediante el programa **HORIZON 2020**, que busca ofrecer nuevos desarrollos tecnológicos dentro del campo de la observación terrestre mediante satélites, obteniendo vídeos en alta resolución y con un gran campo de visión.

El objetivo de este proyecto es desarrollar un dispositivo de observación usando las últimas tecnologías desarrolladas en cada campo. Estas tecnologías se dividirán en la parte de adquisición, para la que se usarán espejos *"freeform"* (con superficies ópticas no simétricas) para las lentes y materiales con una baja capacidad de deformación y creados usando *Aditive Manufacturing* para la estructura; y una parte de detección, en el que se usarán la nueva generación de cadenas de procesamiento y algoritmos de detección basados en Deep Learning.

En cuanto a los *partners* que componen el proyecto, el coordinador es la empresa **Thales**

Alenia Space France SAS, usuaria final del desarrollo realizado en el proyecto; Poly-Shape será el colaborador encargado de la parte de *Aditive Manufacturing*; **Advanced Mechanical and Optical Sistem SA** se encarga de pulir los espejos; **Pyxalis** es el encargado de fabricar los sensores de vídeo; **Thales Alenia Space España SA** es el encargado de montar el dispositivo completo y la **Universidad de las Palmas de Gran Canaria** se encarga del procesamiento de vídeo, tanto de la detección de objetivos como de la compresión de los datos obtenidos antes de ser mandados a tierra.

Es dentro de esta institución, con un equipo de investigación perteneciente a la división DSI del IUMA, en la parte del proyecto en la que se desarrolla este trabajo. Concretamente, forma parte del doctorado de uno de los integrantes del equipo, Romén Neris Tomé, que se encuentra desarrollando una red neuronal convolucional encargada de la detección de barcos. Esta red irá implementada en una FPGA a bordo del satélite y usará como datos de entrada las imágenes del sensor que se está desarrollando por otro *partner* del consorcio.

1.2 Objetivos

El objetivo de este Trabajo de Final de Máster es adaptar un código escrito en **Python** proporcionado por el equipo de investigación en el que se incluye este proyecto, que describe una de las capas convolucionales de una red neuronal, a un lenguaje de alto nivel como C/C++. Esto permitirá realizar una implementación *hardware* de esta capa siguiendo una metodología de diseño basada en High-Level Synthesis (HLS). Debido a que la capa convolucional tiene varios parámetros de entrada que afectan a su funcionamiento y a los resultados que se obtiene de la red, se realizará un estudio de cómo afecta la variación de estos parámetros de entrada a la implementación de la capa y a los recursos que esta consume. A partir de esta propuesta, se establecen unos objetivos para el trabajo, que quedan estructurados de la siguiente forma:

- Realizar un estudio del código en Python de la capa y transformarlo a un lenguaje sintetizable en alto nivel, siendo C++ el que se usará en este trabajo.

- Adaptar el código en C++ de manera que sea compatible con las herramientas de síntesis de alto nivel, en concreto con **Vivado HLS**, que es la que se usará para realizar el

desarrollo de este trabajo.

- Partiendo de unos parámetros de entrada fijos, realizar el estudio y la aplicación de directivas al código *hardware-friendly* en C++. Con esto se pretende optimizar en la medida de lo posible la estructura que va a ser implementada, en términos de latencia y utilización de recursos lógicos, sin que ello suponga una pérdida de precisión de los resultados de la capa.

- Variar los parámetros de entrada de la red convolucional, estudiando qué cambios son necesarios para realizar dicha variación.

- Observar mediante las herramientas de síntesis y cosimulación los resultados obtenidos para cada configuración de entrada. Se analizarán dichos resultados con el fin de sacar conclusiones sobre el impacto de los parámetros en las prestaciones y recursos usados, y a partir de estos deducir la configuración óptima.

1.3 Peticionario

Actúa como peticionario de este Trabajo de Fin de Máster la División de Sistemas Integrados (DSI) del Instituto Universitario de Microelectrónica Aplicada de la Universidad de Las Palmas de Gran Canaria, *partner* del proyecto VIDEO en el que se encuentra trabajando dicho departamento.

Además de esto, el Trabajo Fin de Máster es requisito indispensable para la obtención del título de Máster en Electrónica y Telecomunicaciones Aplicadas (META) que imparte el IUMA, como institución adscrita a la Universidad de Las Palmas de Gran Canaria.

1.4 Estructura del Documento

Este documento se divide en los siguientes capítulos:

1. **Introducción:** en esta capítulo se detallan los antecedentes del proyecto, los objetivos

del mismo y la estructura que tendrá el documento.

2. **Estado del arte:** capítulo en el que se analiza el estado actual de las tecnologías relacionadas con el trabajo y se analizan publicaciones similares al mismo.

3. **Solución Propuesta:** en esta capítulo se detallan todos los elementos a los que se va a hacer referencia durante el trabajo.

4. **Diseño del Bloque IP:** capítulo en el que se describe el proceso que se ha seguido para desarrollar el trabajo y obtener los resultados del mismo.

5. **Resultados Obtenidos:** en este capítulo se exponen los resultados que se han obtenido a partir del trabajo realizado y se analizan dichos resultados.

6. **Conclusiones y Líneas Futuras:** en este capítulo se expresan las conclusiones obtenidas del trabajo en general y de los resultados del mismo y además se comentan posibles trabajos futuros a raíz de estas conclusiones.

CAPÍTULO 2: Estado del Arte

En este capítulo se hará una revisión del estado del arte, proporcionando algunos ejemplos de investigaciones que se han llevado a cabo en los campos relacionados con este Trabajo de Fin de Máster en los últimos años. Además de comentar dichas investigaciones, se estudiarán implementaciones similares a la que se va a describir en este trabajo para comparar posteriormente los resultados de ambos desarrollos.

2.1 Remote Sensing

En este apartado se estudiarán algunas aplicaciones que se le están dando actualmente a las imágenes tomadas mediante *Remote Sensing* y se comentará cuál es la fuente de dichas imágenes y las características de estas.

Comenzando por el medio de obtener la información, muchas misiones hoy en día hacen públicas las imágenes recogidas por los sensores de los satélites. Esto ofrece a los investigadores una base de datos relevante para sus desarrollos, empleando escenas reales que permiten validar el diseño con datos similares a los que se enfrentarán cuando estén desplegados en una aplicación real. La familia de satélites **Landsat** ofrece una amplia base de datos de imágenes multiespectrales de la superficie terrestre, que se complementa con una descripción de su órbita, lo que permite a los investigadores conocer si estos satélites cuentan con información sobre su área de estudio [21] [22] [23].

Aún con los avances en la tecnología actual la resolución de los sensores de los satélites, tanto desde el punto de vista espacial como el temporal, sigue sin ser suficiente para determinadas aplicaciones. Antes de mejorar la tecnología, desde hace unos años se ha optado por combinar los datos de varios sensores para conseguir mejores resultados en las imágenes, ya que sensores con mayor resolución implican un mayor consumo de potencia y mayor capacidad de procesamiento para gestionar las escenas adquiridas.

Uno de los métodos más comunes es el de instalar dos tipos de sensores en los satélites: un sensor pancromático con una mayor resolución espacial pero sin ninguna información espectral; y una cámara multiespectral, que cuenta con una alta resolución espectral pero que por ello suelen tener menor resolución espacial. Al método usado para combinar los datos de los sensores se conoce como *Pan Sharpening* (figura 2.1), y si bien ya está implementado en muchos dispositivos, se siguen investigando técnicas para mejorar el rendimiento y los resultados de este proceso [24]. Otro método similar a este es el de conseguir imágenes con una alta resolución espacial y temporal combinando imágenes tomadas de sensores que cuentan con una baja resolución espacial pero una alta resolución temporal, y otro con una alta resolución espacial pero una baja resolución temporal [21].

El principal uso que se les da actualmente a las imágenes obtenidas mediante *Remote Sensing* es el de detección e identificación de objetivos en la imagen. Para esto, se usan diferentes algoritmos capaces no solo de separar los elementos que componen la imagen, sino que permiten etiquetar dichos elementos, diferenciándolos del resto. En la literatura especializada de los últimos años se destacan los siguientes: el primero se trata de un trabajo en el que se presenta un algoritmo basado en una red neuronal convolucional, entrenada con un grupo de imágenes de zonas urbanas y no urbanas, extraídas de distintas secciones de

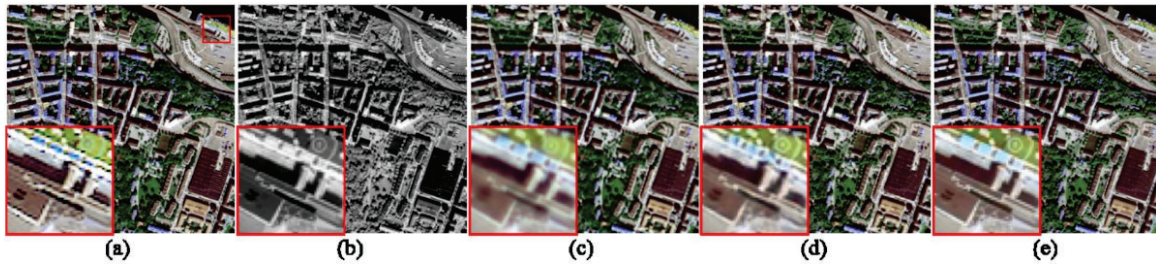


Figura 2.1: a) es la imagen de la cámara multiespectral, b) la de la cámara pancromática y c), d) y e) son los resultados después de aplicar distintos métodos de *Pan Sharpening* [25].

una imagen del satélite Landsat-8 en el que se muestra al completo el área de Beijing y sus alrededores. El objetivo de este algoritmo es detectar y clasificar las secciones del núcleo urbano de Beijing a partir de imágenes extraídas del mismo satélite [26].

La segunda publicación destacada es un análisis de los resultados de diferentes algoritmos, entre los que se incluye una red neuronal convolucional, para detectar el uso que se le da a un área concreta (comercial, viviendas, etc.). A estos algoritmos se les introduce como datos de entrada imágenes de alta resolución obtenidas del satélite **IKONOS** y se comparan los resultados que ofrecen cada uno de ellos [27].

El último artículo presenta un algoritmo específico que usa imágenes extraídas de satélites para detectar los corredores ecológicos cercanos a una zona urbana. A través de los resultados de este, se pretende desarrollar planes de expansión urbanística que respeten el ecosistema de la zona [28].

A través de estos algoritmos de clasificación es posible realizar el estudio de la evolución de zonas a lo largo de un periodo de tiempo, como se demuestra en las investigaciones realizadas en los dos artículos siguientes: el primero es un estudio sobre la evolución de la recuperación de la vegetación en los alrededores de una mina de cobre en China en el que, a través de imágenes obtenidas de los satélites **Landsat** y de algoritmos que usan la información del rango infrarrojo, aportada por las cámaras multiespectrales de estos satélites, se obtuvo un mapa del desarrollo de la vegetación entre los años 2002 y 2019 [22]. El segundo de los estudios hace uso de imágenes de esa misma familia de satélites, además de un algoritmo específico para la detección de agua. Ambos se usaron para estudiar la evolución de una reserva de agua entre tres zonas urbanas de una región de China entre los años 1984

y 2019 [23].

2.2 Redes Neuronales Convolucionales en FPGAs

En este capítulo se describe el estado de las investigaciones actuales sobre la implementación de las redes neuronales convolucionales en FPGAs. Asimismo, se expone en qué se centran esos estudios presentando artículos en los que se implementan capas convolucionales en este tipo de dispositivos y también qué estrategias se están siguiendo para realizar este proceso.

2.2.1 Motivo de uso de FPGAs

Como ya se mencionó en la Sección 1.1.4, las redes neuronales convolucionales han demostrado ofrecer unos muy buenos resultados a la hora de procesar imágenes. A la hora de implementar este tipo de procesamiento a bordo de un satélite, se han estado buscando plataformas que realicen el cálculo de forma eficiente, tanto computacionalmente como en términos de energía consumida, y que además resistan las condiciones ambientales que hay en el espacio. Debido a la gran cantidad de operaciones realizadas por este tipo de redes, una CPU no suele contar con capacidad suficiente para llevar a cabo los cálculos con una latencia aceptable por lo que normalmente, a la hora de ejecutar el algoritmo en Tierra, se usan servidores equipados con GPUs, que si bien tienen un gran rendimiento al realizar un gran número de operaciones en paralelo, cuentan con unos consumos de potencia bastante elevados [29] [30].

Teniendo esto en cuenta, sumado a que aún no se han desarrollado GPUs que demuestren ser tolerantes a la radiación espacial [31], las soluciones por las que se han optado son usar sistemas **ASIC** o **FPGA**. Al comparar estos sistemas, una de las ventajas más significativas con las que cuentan las FPGAs es un tiempo de desarrollo menor que el de los ASICs, además de que el coste del mismo también se ve reducido. Esta característica, teniendo en cuenta que la estructura de las redes convolucionales cambia muy a menudo, es lo que está provocando que los estudios en la actualidad se decanten por emplear FPGAs como dispositivo final [32].

Si bien este tiempo de desarrollo es más reducido que el de los ASICs, el proceso de implementación de las redes, normalmente desarrolladas usando librerías de **Python**, a una FPGA resulta costoso y requiere de unos conocimientos de *hardware* más amplios que, por ejemplo, el proceso para desarrollar una red en una **GPU**. Por esto, y teniendo en cuenta que estas redes están cobrando cada vez más importancia, están apareciendo estudios que proponen métodos de generación automática de bloques IP a partir de esquemas proporcionados por el usuario en lenguajes o *frameworks* de muy alto nivel. Otro de los motivos por los que las FPGAs son usadas en misiones espaciales es la posibilidad de reconfigurarlas. A la hora de tener un dispositivo a bordo de un satélite, la posibilidad de cambiar su funcionamiento durante la misión permite no solo más flexibilidad, sino que puede solucionar errores que se hayan producido en el desarrollo de la misma debidos, por ejemplo, a la radiación. Sobre este tema, se presentan dos estudios.

En el primero de ellos, se presenta una implementación de un modulador de señales en una **Xilinx Ultrascale**, concretamente la denominada como **XCKU060**. Este modulador irá instalado en el satélite y a través de la reconfigurabilidad del dispositivo permitirá cambiar el tipo de modulación de la señal a recibir/emitir [33].

En el segundo de los artículos se aprovecha la reconfigurabilidad que ofrece la FPGA de Xilinx **XCKU5P** para que, en el caso de que una de las partes de la implementación falle debido a las condiciones extremas que se dan en el espacio, poder volver a configurar esa parte del dispositivo de manera que se elimine la posible corrupción que puede haber sufrido el diseño [34].

2.2.2 Implementaciones de una CNN en FPGAs

Entre los estudios existentes, se pone como ejemplo dos artículos que presentan algoritmos que generan automáticamente implementaciones en FPGAs de redes neuronales. En el primero de los artículos, se propone un algoritmo que parte de un modelo de ***TensorFlow***, una librería ampliamente usada para desarrollar redes en **Python**, generando un gráfico en el que se estructuran los elementos. A partir de ese modelo, el algoritmo mapea los componentes de la red con elementos lógicos prediseñados para ser implementados en una FPGA y, a través de las herramientas de síntesis adecuadas, transforma la estructura

que ha generado en una implementación de la red [35].

En el segundo de los artículos, se presenta un estudio de las posibles optimizaciones que se pueden realizar sobre los componentes de una red neuronal en una FPGA. El algoritmo que se propone toma la estructura de la red de un grafo de ***TensorFlow***, analiza las optimizaciones que se pueden aplicar a los componentes de dicha red y, una vez determinadas, compila el diseño para implementarlo en la plataforma [36].

En cuanto a términos de computación, se demuestra que aquellas redes que se implementan en FPGAs cuentan con mejoras significativas en cuanto a la latencia del proceso. De uno de los estudios [37] se extrae que una red implementada en una FPGA (**PYNQ-Z1 Zynq 7z020**) consigue hasta 12,8 veces más rendimiento que una GPU (**ARM Mali-T860MP4**) y 42,38 más rendimiento que una CPU (**ARM Cortex-A72+A53**), aunque no se mencionan las posibles pérdidas de precisión en la predicción que se produjeron en las optimizaciones realizadas, sobre todo si estas influyen en el tipo de precisión de los datos empleada. En otra publicación [36], se compara el *throughput* que ofrecen la FPGA utilizada (Intel Stratix 10) con el que ofrece una GPU (NVIDIA Volta) sobre dos redes diferentes, la ResNet-50 y la MobileNet. Los resultados que se ofrecen en la FPGA son entre 1,5 y 11 veces mejores que los que ofrece la GPU, aunque cuentan con una ligera pérdida de precisión en lo que a las predicciones se refiere.

Sobre el consumo de potencia de estos dispositivos, se destaca un artículo en el que se realiza un estudio sobre cómo se puede reducir el consumo de potencia en las implementaciones que se han realizado. En dicho artículo, estudian la disipación de potencia de los distintos componentes de una FPGA y, aplicando las técnicas que han extraído de dicho estudio, consiguen reducir la potencia frente a distintas implementaciones actuales de la misma red y, de manera más significativa, a un desarrollo de dicha red en una GPU [30].

2.2.3 Arquitecturas de la capa convolucional

En cuanto a las optimizaciones que se realizan a la hora de implementar una red neuronal convolucional en una FPGA, las publicaciones que se han estudiado se centran normalmente en la transferencia de los datos, el tipo de datos que se usa, la gestión de los pesos de

la red y la paralelización del cómputo. A la hora de transmitir los datos en las arquitecturas publicadas se toma normalmente la estrategia de repartir el flujo de datos en varios canales diferentes, entendiendo como flujo de datos tanto el acceso a los pesos como el acceso a los datos de entrada. Si bien este punto es común, las publicaciones difieren en los canales utilizados. Mientras que hay algunas que proponen utilizar los buses AXI de los que disponen comúnmente ciertas FPGAs para transmitir los datos entre capas [37], otros optan por usar los buses AXI para acceder a los datos en memoria y, una vez que esos datos estén en el acelerador *hardware*, transmitirlos entre capas usando **FIFOs**, sistema que además demuestra tener un impacto positivo en el consumo de potencia de la red [30].

Una de las ventajas que supone el usar varios canales de entrada es que se pueden explotar los recursos de la FPGA para realizar el procesamiento de esos datos en paralelo. Entre las publicaciones que describen sus implementaciones, es común que se realice la multiplicación de la matriz de pesos totalmente en paralelo. Un ejemplo de esto se presenta en un artículo que propone una implementación en la que se cuenta con k canales de entrada tanto para la imagen como para los pesos, siendo k el tamaño de la matriz de pesos que se esté usando, consiguiendo así realizar todas las multiplicaciones necesarias a la misma vez para luego sumar los resultados [29]. Otra estrategia es la que se propone en [38], que consiste en contar, para una matriz de pesos fija de 3×3, con tres canales de entrada que almacenen cada uno los tres valores de la entrada que se necesitan multiplicar.

En cuanto al tipo de dato utilizado, existen diferentes propuestas en las publicaciones que se han estudiado. En su gran mayoría, debido a que se produce un impacto significativo en el consumo de recursos y en la velocidad en la que se procesan los datos, la mayoría de las publicaciones suele optar por usar datos en formato de punto fijo, variando el ancho de banda de estos entre 8 bits [37], 16 bits [39] o 32 bits. Otras propuestas optan directamente por usar punto flotante de 32 bits pero, como se ve reflejado en sus resultados, el consumo de recursos aumenta [29].

Para terminar, existen publicaciones que estudian cómo reducir el número de multiplicaciones o el consumo de memoria aprovechándose de las redundancias que aparecen en los pesos de las redes y en aquellos pesos que tienen valor 0. Se destaca una publicación en la que se consigue reducir significativamente el número de multiplicaciones realizadas en la red Alexnet en base a asignar un código a cada peso, de manera que permita evitar aquellas

multiplicaciones por pesos con valor 0 o almacenar pesos redundantes [32].

CAPÍTULO 3: Solución Propuesta

En este capítulo se describen los diferentes elementos y herramientas que se han usado para desarrollar la solución propuesta en este Trabajo de Fin de Máster. En el primer apartado, se describirá con más profundidad una capa convolucional, así como el cálculo que realiza y las características de esta capa. Además, se definirá la plataforma *hardware* hacia la que se orientan las implementaciones que se han desarrollado. En el segundo apartado, se mencionarán las herramientas que se han usado durante el desarrollo del trabajo, entendiendo como herramientas tanto los lenguajes de programación usados como el entorno de desarrollo Xilinx Vitis HLS.

3.1 Capa Convolucional

El proceso que se realiza en una capa convolucional requiere de dos entradas y proporciona una salida única. La información que entra a la capa se divide en dos: los datos de entrada que van a ser procesados y los pesos de la capa, que se aplicarán a los datos de entrada para generar la salida. Además, dependiendo de si la red en cuestión lo requiere o no, puede existir un tercer parámetro de *bias* que se aplicará al proceso de convolución.

La estructura esquemática de la capa se muestra en la figura 3.1. Como se puede observar, tanto los pesos como los datos de entrada tienen el mismo número de bandas (representado en la imagen como *nb*). Después de realizar el cálculo, la salida de la capa tendrá unas dimensiones que dependen de las dimensiones de los pesos y de los datos de entrada, así como del desplazamiento que se realice del *kernel*. Teniendo dos entradas bidimensionales, independientemente del número de bandas de éstas, se generarán tantas salidas bidimensionales como filtros tengan los pesos, lo que en la imagen se representa como *nf*.

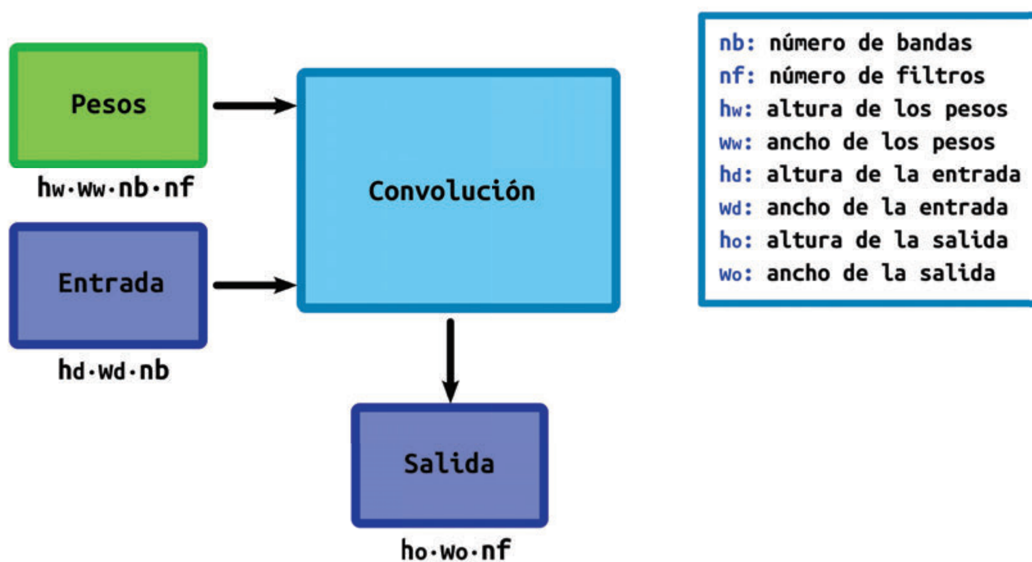


Figura 3.1: Esquema de una capa convolucional

3.1.1 Parámetros de la capa convolucional

En esta sección se describirán los diferentes parámetros de entrada de la capa y se nombrará la terminología que se usa para referirse a ellos.

- **Pesos:** a los que se les denomina también *kernel* (figura 3.2), se de un grupo de datos de alto y ancho variable que cuentan con las mismas dimensiones que la entrada. los pesos se obtienen al entrenar la red neuronal y pueden usarse en la misma red de la que se obtuvieron o almacenarlos y usarlos en otra red igual a la que se usó para obtenerlos pero implementada en un dispositivo diferente.

- **Filtros:** dependiendo de la red, pueden existir varios conjuntos de pesos que aplicar a una misma entrada. Partiendo de una imagen bidimensional con un número nb de bandas, un conjunto de pesos es una matriz de una alto y un ancho determinados y una profundidad de nb bandas. Para una misma zona de la entrada, si se van a aplicar más de uno de estos conjuntos se dice que se cuenta con varios filtros, que generarán tantas salidas para esa zona en concreto como filtros haya.

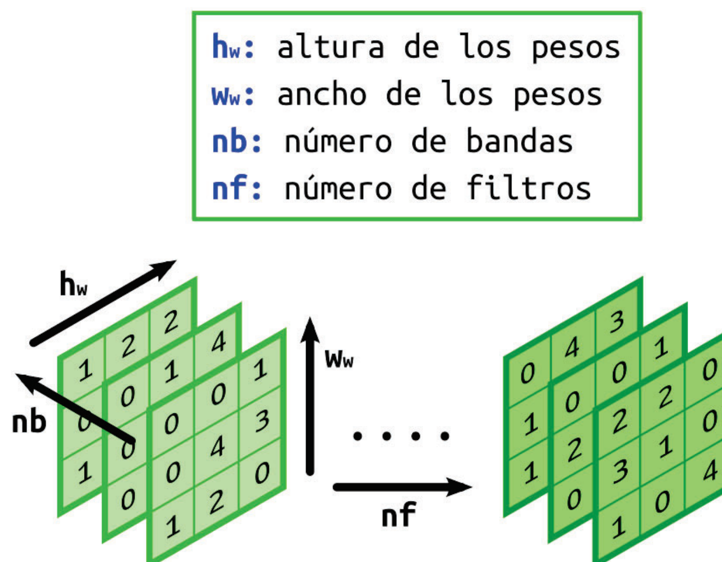


Figura 3.2: Esquema de pesos y filtros de la capa convolucional

- **Desplazamiento de los pesos:** también conocidos como *stride*, se trata de la distancia, tanto horizontal como vertical, que se desplaza los pesos sobre la matriz de datos de entrada para obtener la siguiente salida.

- **Bias:** si la red lo requiere, este será un parámetro que se le sumará a todas las salidas calculadas mediante el proceso de convolución.

- ***Zero Padding:*** se trata de un preprocesado que se le aplica a los datos de entrada antes de realizar la convolución. Este consiste en añadir tantas filas y columnas de ceros alrededor imagen de entrada como sean necesarios. Este proceso se realiza sobre todas las bandas y se computa siguiendo las ecuaciones que se muestran en 3.6. En estas ecuaciones h_w representa el alto de los pesos, s_h el desplazamiento vertical que se realiza, s_w representa el ancho de los pesos y s_w el desplazamiento horizontal. A la hora de realizar las divisiones, el resultado será el cociente entero de las mismas.

$$Pad\ a\ lo\ Alto = max(h_w - s_h, 0) \tag{3.1}$$

$$Pad\ a\ lo\ Largo = max(w_w - s_w, 0) \tag{3.2}$$

$$Pad\ Superior = Pad\ a\ lo\ Alto/2 \tag{3.3}$$

$$Pad\ Inferior = Pad\ a\ lo\ Alto - Pad\ Superior \tag{3.4}$$

$$Pad\ Izquierda = Pad\ a\ lo\ Largo/2 \tag{3.5}$$

$$Pad\ Derecha = Pad\ a\ lo\ Largo - Pad\ Izquierda \tag{3.6}$$

3.1.2 Cálculo de la convolución

Partiendo de una entrada y de unos pesos de dos dimensiones, el proceso de convolución consiste en multiplicar uno a uno todos los pesos por su equivalente en los datos de entrada. Una vez se han obtenido todos los resultados de las multiplicaciones, se suman y se obtiene la salida (figura 3.3).

Después de obtener el resultado de la convolución, se suma el *bias* si lo hubiera y se

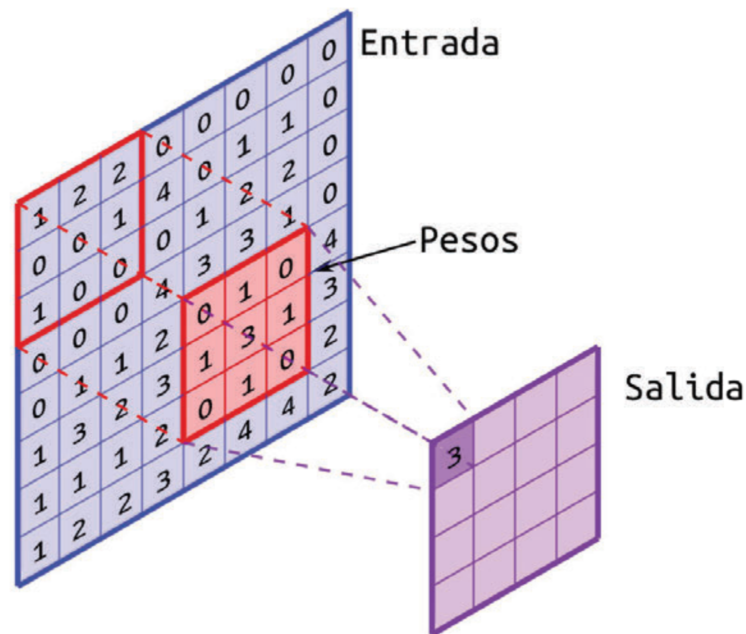


Figura 3.3: Proceso de Convolución: primer paso

pasa al siguiente paso. Para esto, se desplaza el área de aplicación del *kernel* un número determinado de columnas de los datos de entrada, siendo este número un parámetro de la capa conocido como *stride*, y se vuelve a repetir el proceso de cálculo. Una vez finalizado el desplazamiento de las columnas, se desplazará el área un número determinado de filas, repitiéndose el proceso hasta generar la salida, también bidimensional, de la capa (figura 3.4).

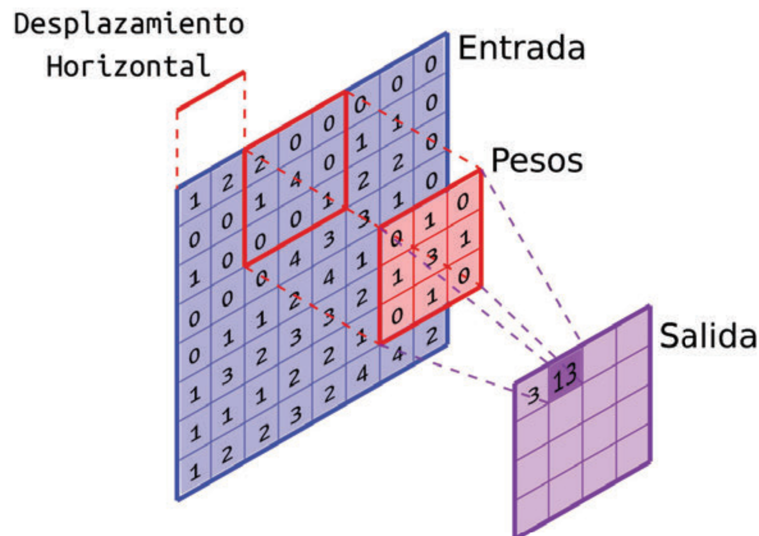


Figura 3.4: Proceso de Convolución: segundo paso

En el caso de que los datos de entrada no sean bidimensionales, como es una imagen RGB, en la que las bandas de los colores formarían una tercera dimensión, el proceso es similar. En este caso, los pesos cuentan con una dimensión extra, de manera que cada banda de la imagen se multiplicaría por los pesos que le correspondiese, pudiendo estos ser los mismos para todas las bandas o diferentes entre sí, y luego se sumarían todos los resultados, proceso que se muestra en la figura 3.5.

3.1.3 Capa Convolucional con Imágenes

Como ya se mencionó en la sección 1.1.4, las redes neuronales convolucionales son ampliamente usadas en el campo de la clasificación de imágenes. El motivo principal es que las características de la capa convolucional permiten que esta extraiga y separe la distinta información presente en las imágenes. Esta información depende directamente de los pesos que se usen para procesar los datos de entrada, por lo que para extraer elementos distintos deberá entrenarse la capa con imágenes en las que se encuentren dichos elementos.

Un ejemplo de esto lo encontramos en el artículo realizado por A. Kalinovsky y V. Liauchuk [41], en el que utilizan este tipo de redes para detectar lesiones relacionadas con la

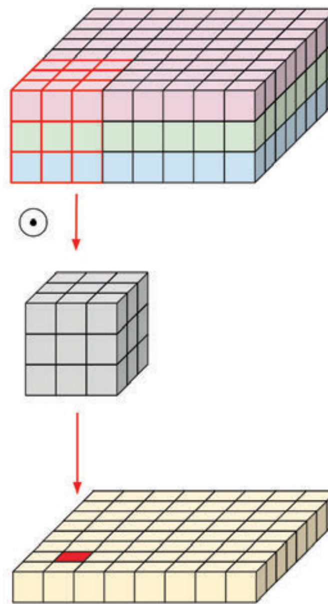


Figura 3.5: Proceso de Convolución para tres dimensiones [40]

tuberculosis en los pulmones. En la figura 3.6 se muestra cómo la capa ha conseguido extraer de la imagen de entrada la parte correspondiente a los pulmones. Una vez extraída la zona de estudio, deberán usarse otras capas de la red para clasificar la imagen, ya que la capa convolucional solo extrae la información, no la etiqueta.

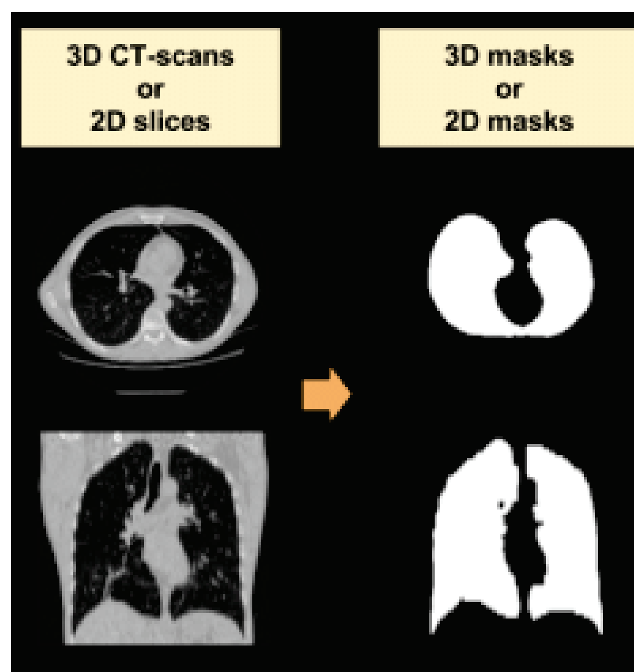


Figura 3.6: Entrada y Salida de la capa convolucional [41]

En la figura 3.7, extraída de un artículo en el que se usa una red neuronal convolucional

para guiar a través de las imágenes a un coche autónomo [42], se puede observar la salida de la capa cuando se entrena para detectar las líneas de la carretera.

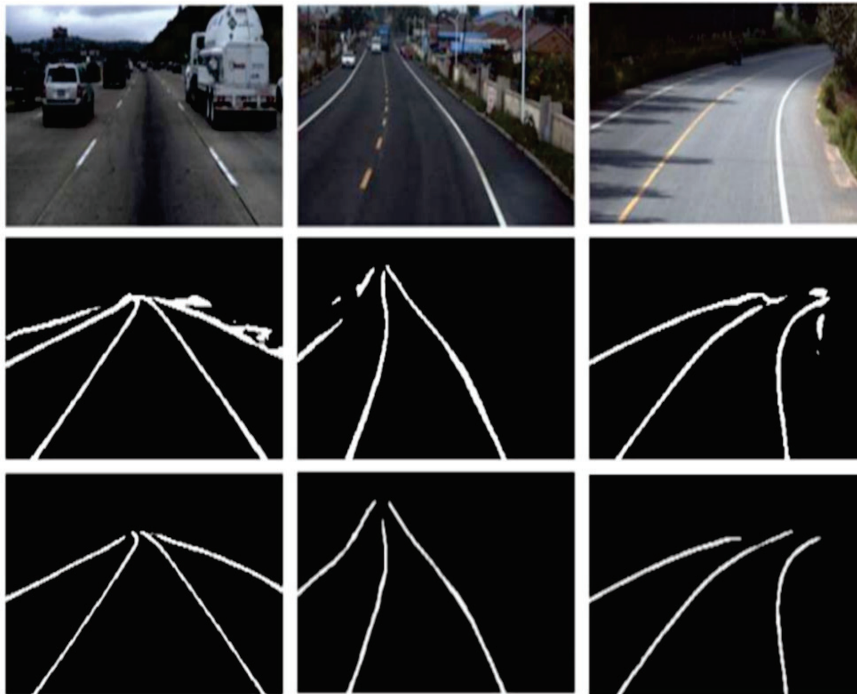


Figura 3.7: Detección de las líneas de la carretera [42]

3.2 Hardware al que se orienta la solución

Todo el diseño que se ha realizado en este Trabajo de Fin de Máster tiene como objetivo implementarse en una FPGA de **Xilinx**, concretamente en una **Xilinx Kintex UltraScale**. De todas las **Kintex Ultrascale** que ofrece el fabricante **Xilinx**, la única que está certificada para resistir la radiación espacial es la **XQRKU060** [6]. Sin embargo, el equipo junto con el que se realiza este Trabajo de Fin de Máster no dispone de este dispositivo, sino que están usando su equivalente comercial a nivel de recursos lógicos disponibles, la FPGA de Xilinx **XCKU040-2FFVA1156E**, instalada en el kit de desarrollo **Xilinx Kintex UltraScale FPGA KCU105 Evaluation Kit** [43].

3.2.1 FPGA XCKU040-2FFVA1156E

Este dispositivo pertenece a la familia **Kintex Ultrascale** de Xilinx, unas FPGAs fabricadas en una arquitectura de 20 nm. El fabricante recomienda esta familia de dispositivos para usos tales como el procesamiento de paquetes en redes de alta velocidad o centros de

datos, procesamiento de imágenes médicas, vídeos de resolución 4k o superior, así como el procesamiento de datos en una red inalámbrica heterogénea. Las características que presenta el dispositivo a nivel de recursos lógicos e interfaces de entrada/salida se muestran en el cuadro 3.1.

Recursos Lógicos		
Células lógicas del sistema	CLB Flip-Flops	CLB LUTs
530 250	484 800	242 400

Recursos de Memoria			
RAM Distribuida	Block RAM/FIFO/ECC (36 kb)	Block RAM/FIFO (18 kb)	Total Block RAM (Mb)
7 050	600	1 200	21,1

Recursos de Reloj	
CMT (1 MMCM, 2 PLLs)	I/O DLL
10	40

Recursos de Entrada/Salida (Máximo número de recursos)			
I/Os de alto rendimiento (único)	I/Os de alto rendimiento (diferencial)	I/Os de alto rango (único)	I/Os de alto rango (diferencial)
416	192	104	48

Dispositivos IP integrados			
DSP Slices	Monitor de Sistema	PCIe®Gen2/2/3	16.3Gb/s Transceivers (GTH/GTY)
1 920	1	3	20

Cuadro 3.1: Recursos de la FPGA XCKU040-2FFVA1156E [44]

3.2.2 Kintex UltraScale FPGA KCU105 Evaluation Kit

A la hora de realizar un diseño sobre la FPGA, los fabricantes suelen proporcionar kits que permiten la programación y prueba del funcionamiento de las implementaciones realizadas, facilitando así el desarrollo en sus productos. En este Trabajo de Fin de Máster se ha usado como kit de desarrollo la **Kintex UltraScale FPGA KCU105 Evaluation Kit** (figura

3.8), que cuenta con las características que se enumeran a continuación:

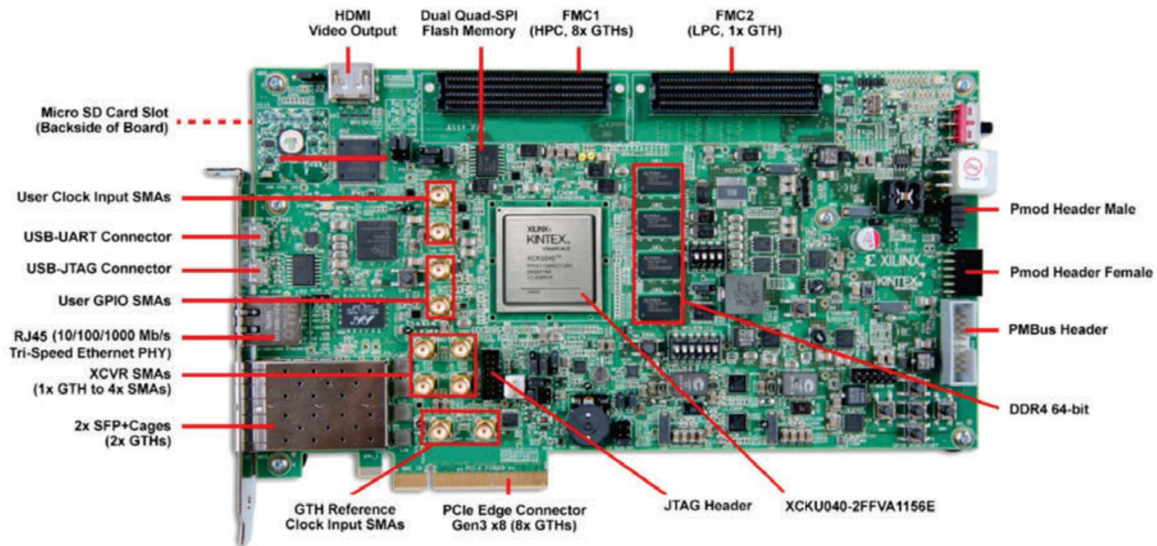


Figura 3.8: KCU105 Evaluation Kit [43]

- **Configuración**
  - Circuito integrado JTAG que permite la configuración a través de USB.
  - Cabezal para conectar un cable JTAG.
  - 2×256 Mb de memoria FLASH Quad SPI.

- **Memoria**
  - 2GB DDR4 de memoria (cuatro dispositivos [256 Mb x 16]) a 1200MHz / 2400Mbps.
  - 64MB (512Mb) de memopria Flash Quad SPI.
  - 8Kb IIC EEPROMI.
  - Un puerto para conectar una tarjeta Micro SD.

- **Redes y Comunicación**
  - Gigabit Ethernet GMII, RGMII and SGMII.
  - 2x SFP / SFP+ cage.
  - Puerto GTX (TX, RX) con cuatro conectores SMA.
  - Conector USART a USB.
  - 8 conectores PCI Express.

- **Conectores de Expansión**

  - Conector FMC-HPC (Partial Population) (Cuenta con hasta 8 GTX Transceiver y señales configurables por el usuario entre 114 únicas o 57 diferenciales (34 LA y 24 HA)).
  - Conector FMC-LPC (Cuenta con hasta 1 GTX Transceiver y señales configurables por el usuario entre 68 únicas o 34 diferenciales (34 LA y 24 HA)).
  - 2 cabezales PMOD.
  - IIC.

- **Elementos de representación visual**
  - Salida de vídeo HDMI.
  - Dispositivo PHY/codec externo que maneja un conector HDMI.
  - 8 LEDs conectados a los GPIOs.

- **Relojes del dispositivo**
  - 8 relojes programables.
  - Relojes de sistema, relojes EMC, relojes de usuario, relojes con un *jitter* atenuado.
  - 2 entradas de reloj SMA.

- **Elementos de control y de entrada/salida**
  - 5 botones direccionales.
  - 4 interruptores DIP.
  - 1 interruptor de rotación.
  - 1 par de entradas y salidas diferencial (SMA).

- **Alimentación**
  - Adaptador de corriente de pared ATX de 12V.

3.3 Lenguajes

En este Trabajo de Fin de Máster se han usado diferentes lenguajes de programación para cumplir los objetivos del mismo. En esta sección se describirán estos lenguajes, explicando cuál fue el cometido de cada uno y qué características presentan.

3.3.1 Python

Python es un lenguaje de programación interpretado de alto nivel, de código libre y de propósito general. Una de las principales fortalezas de este lenguaje de programación es la gran cantidad de librerías disponibles, lo que permite realizar programas para multitud de aplicaciones. Una de estas aplicaciones es el desarrollo de algoritmos de *Machine Learning* y la inteligencia artificial, siendo este el uso que se le dio en este trabajo. De todos las herramientas de *Machine Learning*, este trabajo esta relacionado con dos concretamente: **TensorFlow** y **Keras** [45].

TensorFlow es una plataforma de código libre que ofrece un ecosistema de herramientas, librerías y otro tipo de recursos orientados a realizar modelos de *Machine Learning*. Las características principales de esta plataforma son:

- Permite elaborar y entrenar los modelos de *Machine Learning* de una manera sencilla.
- Es compatible con un gran número de lenguajes y plataformas, lo que permite entrenar y ejecutar los algoritmos en casi cualquier *hardware* con los recursos computacionales suficientes.
- La flexibilidad que ofrece permite crear topologías complejas orientadas a la investigación.

Por otro lado, **Keras** es una librería de alto nivel orientada al desarrollo de redes neuronales. Esta librería, escrita en Python, trabaja en base a plataformas como **TensorFlow** y permite realizar redes neuronales capaces de ser ejecutadas tanto en CPUs como en GPUs, lo que aumenta el rendimiento de estas. Las principales ventajas que ofrece esta librería son:

- Tiene una interfaz simple que permite detectar fácilmente los errores que se cometen al trabajar con la librería.
- Los elementos de Keras están constituidos en forma de bloques, lo que permite realizar modelos en base a unir estos bloques, cumpliendo una serie de requisitos.

- Permite construir nuevos bloques para desarrollar nuevos modelos o investigar sobre el funcionamiento de estos.
- La API de Keras está pensada para reducir el número de acciones que debe realizar el usuario cuando se trabaja en modelos con una estructura común.

El equipo de investigación del IUMA desarrolló y entrenó la red neuronal **Mobilenet** usando estas librerías, y luego a partir del modelo generado desarrollaron un código en Python de las distintas capas de la red. El código de la capa convolucional de esta red descrita en Python fue el punto de partida de este Trabajo de Fin de Máster, con el fin de transcribirlo y realizar la simulación en *hardware* de la capa.

3.3.2 Lenguaje C++

Este lenguaje se usó en este trabajo para desarrollar la capa convolucional de manera que la herramienta de síntesis de alto nivel pudiera interpretar el código y realizar la síntesis y simulación del *hardware*. C++ es un lenguaje de programación orientado a objetos de propósito general. Es una extensión del lenguaje C que, igual que este, necesita ser compilado. Ambos lenguajes son soportados por la herramienta de Vitis HLS.

3.3.2.1 Adaptación a Vitis HLS

Si bien tanto C como C++ están soportados en la herramienta, existen una serie de modificaciones necesarias para que, a la hora de realizar la síntesis, Vitis HLS pueda generar bloques compatibles para la plataforma objetivo que se defina. La documentación de la herramienta [46] enumera las siguientes condiciones para que el código sea sintetizable:

- **Llamadas al sistema**. La función a sintetizar debe contener todos los elementos necesarios para que el diseño funcione. Esto se debe a que aquellas funciones que realicen llamadas al sistema o que dependan del sistema operativo no son sintetizables. Un ejemplo de estas funciones no soportadas son “printf()” o “time()”.
- **Memoria dinámica**. Debido a que aquellas funciones que permiten alojar memoria de

manera dinámica se realizan a partir de llamadas del sistema, no están soportadas durante la síntesis. Sin embargo, a la hora de realizar simulaciones del comportamiento del código a través de la herramienta, se pueden usar siempre que sea mediante una macro que las ignore durante el proceso de síntesis.

- **Funciones Recursivas**. Las funciones recursivas no están soportadas y no son sintetizables.

- **Bucles**. Para poder realizar la síntesis correctamente y obtener los resultados de esta, así como permitir pasos posteriores como co-simulación, todos los bucles deben contar con límites fijos. En el caso de que esto no sea posible, la herramienta ofrece un sistema que permite delimitar el rango de un bucle variable, pero si no se establece este rango correctamente la síntesis tampoco será posible.

- ***Arrays***. A la hora de usar este tipo de estructuras, hay que tener en cuenta que el tamaño de estas puede ser demasiado grande para los recursos disponibles tanto en el sistema en el que se vaya a realizar la simulación, como en la plataforma hacia la que está orientada la síntesis que se va a realizar.

3.4 Xilinx Vitis HLS

Debido a los largos tiempos de desarrollo que puede suponer el realizar una implementación en una FPGA usando directamente un lenguaje de abstracción *hardware* como VHDL o Verilog, se han desarrollado herramientas de alto nivel que permiten, a partir de lenguajes como C/C++, desarrollar implementaciones eficientes con un tiempo de desarrollo significativamente menor. Concretamente, en este Trabajo de Fin de Máster se ha usado el programa proporcionado por Xilinx, **Vitis HLS**, como herramienta de síntesis de alto nivel para realizar la simulación de las distintas soluciones que se pretendían estudiar.

3.4.1 Uso de la herramienta

En esta sección se expondrán los distintos pasos, partiendo de la documentación oficial de la herramienta [46], que se deben seguir para hacer uso de esta.

1. **Creación de un Proyecto**

   Esta herramienta trabaja en base a proyectos en los que se generarán las distintas soluciones para la implementación que se quiera realizar.

   A la hora de crear el proyecto se deberá establecer: el nombre y ruta en el equipo de dicho proyecto; el archivo en el que se encuentre la función principal que se quiera sintetizar, así como el nombre de esta función; los archivos adicionales que se usarán a la hora de realizar la descripción y la simulación del proyecto; la plataforma, ya sea a través del nombre del componente de Xilinx o a través de la plataforma de evaluación en la que esté instalado; y las restricciones de tiempo que se quieran para el sistema.

2. **Simulación en C/C++**

   Para verificar que los resultados obtenidos al ejecutar el código son los deseados, Vitis HLS ofrece herramientas de simulación con las que, a partir de un banco de pruebas o *testbench*, se puede generar una salida de la función principal y comprobar los resultados de ésta.

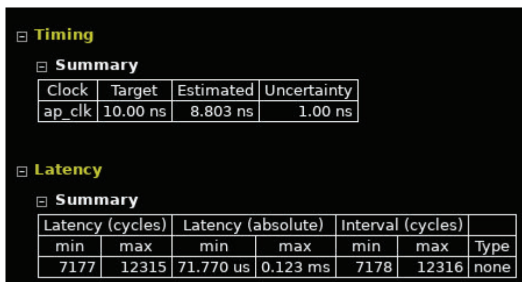
   Para acelerar el proceso de verificación es necesario generar un buen banco de pruebas, que no es más que un código que ejecute la función principal de la solución y compruebe que los resultados sean los correctos, normalmente comparándolos con una *golden reference*. En el caso de que los resultados no sean los deseados, la herramienta de Xilinx ofrece un entorno de depuración que permite monitorizar el programa durante su ejecución.

3. **Síntesis**

   Esta parte de la herramienta transforma la descripción en alto nivel en su equivalente RTL y permite estimar el rendimiento de las soluciones del proyecto, si estas se implementaran en la plataforma que se ha especificado durante la creación del mismo. La información que ofrece es la siguiente, tal como se puede ver en la figura 3.9:

- **Latency:** número de ciclos necesarios para que la función genere todos los valores de salida.
- **Initiation Interval:** número de ciclos que deben pasar antes de que la función pueda volver a procesar un dato de entrada.
- **Loop iteration Latency:** número de ciclos que tarda en completarse una iteración de un bucle.
- **Loop iteration interval:** número de ciclos que tarda en iniciarse la siguiente iteración del bucle.
- **Loop Latency:** número de ciclos que tarda en completarse todas las iteraciones del bucle.

Aparte de esto, la herramienta permite comparar los resultados de sintetizar las soluciones que se encuentren en el proyecto, pudiendo incluso excluir algunas de estas soluciones.



Timing

Summary

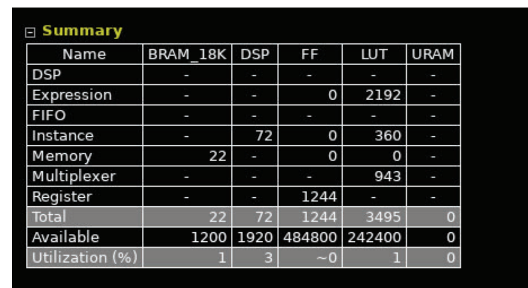
Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.803 ns	1.00 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
7177	12315	71.770 us	0.123 ms	7178	12316	none

(a) Resultados de latencia de la síntesis



Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2192	-
FIFO	-	-	-	-	-
Instance	-	72	0	360	-
Memory	22	-	0	0	-
Multiplexer	-	-	-	943	-
Register	-	-	1244	-	-
Total	22	72	1244	3495	0
Available	1200	1920	484800	242400	0
Utilization (%)	1	3	~0	1	0

(b) Resultados de recursos de la síntesis

Figura 3.9: Resultados de la síntesis en Vitis HLS

4. **Herramienta de Análisis**

Esta herramienta permite observar en un esquema ordenado el número de ciclos que tarda en generarse una salida, así como el número de ciclos que tarda cada una de las operaciones que se realizan durante la ejecución de la función principal, el flujo de datos entre estas operaciones y los posibles problemas que puedan encontrarse en la solución. Toda esta información se refleja de forma visual, tal como se muestra en la figura 3.10. Estas características hacen que la herramienta sea muy útil a la hora de tomar decisiones sobre cuál es la parte concreta del código que se debe modificar u optimizar para mejorar el resultado de la solución.

Además de esto, la herramienta permite generar un esquema en el que se detalla el flujo de datos durante la ejecución, lo que permite discernir más fácilmente las dependencias de datos indeseadas o posibles alternativas al flujo de datos que puedan acelerar la ejecución del código.

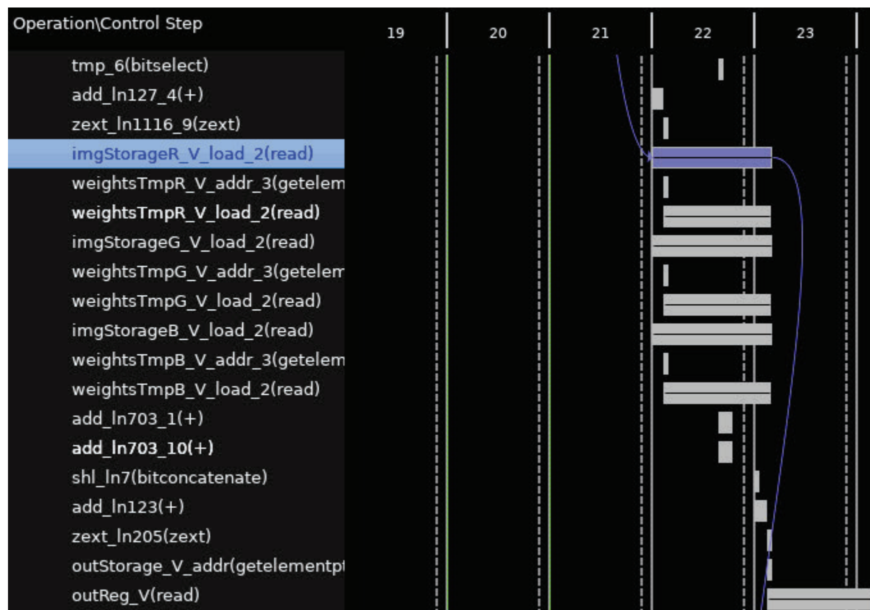


Figura 3.10: Herramienta de análisis de Vitis HLS

3.4.2 Librerías HLS

Para complementar la programación en C/C++, Vitis HLS cuenta con librerías y directivas que ofrecen al usuario un control mayor sobre la implementación *hardware* que se va a hacer del código. En esta sección se comentarán tanto los tipos de datos propios de HLS como los *pragmas* que dan control sobre el comportamiento de la herramienta de síntesis y permiten, entre otras cosas, gestionar los recursos y las interfaces de la implementación que se va a realizar.

3.4.2.1 *Arbitrary Precision (AP) Data Types*

Debido a que las interfaces de comunicaciones de las FPGAs permiten un ancho de datos arbitrario, realizar implementaciones con los tipos de datos estándar de C/C++ (8, 16, 32 o 64 bits) puede ofrecer resultados ineficientes. Para resolver este problema, Vitis HLS

ofrece librerías que permiten designar un ancho de bits específico que permite el control a nivel de bit disponible en los lenguajes de abstracción *hardware* pero trabajando a alto nivel. Estas librerías se dividen en la librería de enteros de **precisión arbitraria "ap_int.h"** y la librería de **punto fijo "ap_fixed.h"**. La librería **"ap_int.h"** permite definir enteros con un tamaño cualquiera con la sentencia `ap_int <N> var`, en la que *N* es el número de bits que se desea y *var* es el nombre de la variable. Usando este tipo de datos no solo se consigue optimizar el ancho de las interfaces, sino que tiene un impacto positivo en la ocupación de memoria y permite usar menos recursos lógicos del dispositivo.

Por otro lado, la librería **"ap_fixed.h"** permite, en un número decimal, designar cuántos bits se dedican a representar la parte entera y cuántos a la parte decimal. Para definir este tipo de datos, hay que usar la sentencia `ap_fixed <W,I,Q,O> var`, en la que *W* es la longitud en bits de la palabra, *I* el número de bits para representar la parte entera, *Q* es el modo de cuantificación, y *O* es el comportamiento cuando se produce *overflow* o *underflow*. Esto no solo ofrece mayor control sobre el diseño, sino que permite observar el efecto de posibles desbordamientos al simular el código en C/C++. Además, este tipo de datos permite sustituir al punto flotante, lo que reduce la latencia del sistema, ya que se necesitan más ciclos para realizar las operaciones con este tipo de datos que con enteros.

3.4.2.2 Directivas HLS

Las directivas son instrucciones que se pueden añadir al código y que permiten modificar el comportamiento del sintetizador. A partir de estas directivas, predefinidas por el entorno de Vitis HLS, se puede optimizar el diseño, reducir la latencia, mejorar el *throughput* o reducir el consumo del recursos del código RTL resultante de la síntesis de alto nivel. A continuación, se mencionarán algunas de las directivas disponibles en el entorno de Vitis HLS:

- **Optimización de área**
  - ***pragma HLS aggregate***: permite agrupar todos los campos de datos de un "struct" en una misma variable formada por un mayor número de bits.

- ***pragma HLS bind_op***: permite vincular una determinada operación matemática a un recurso, de manera que a la hora de generar el modelo RTL el sintetizador asignará dicho recurso a la variable y cálculo designados.

- ***pragma HLS bind_storage***: permite vincular un *array* a un determinado tipo de memoria.

- ***pragma HLS latency***: permite especificar la latencia máxima y mínima de una función, bucle o región.

- ***pragma HLS top***: asigna un nombre a la función para que este pueda ser usado con el comando "set_top".

- **Alineación de Funciones**

  - ***pragma HLS inline***: permite eliminar la jerarquía de funciones en el RTL, implementándolas no como bloques separados, sino como parte de la función principal.

- **Síntesis de Interfaces**

  - ***pragma HLS interface***: permite asignar una de las interfaces disponibles para sintetizar en el entorno de Vitis HLS a los argumentos de la función principal.

- ***Pipeline* a nivel de tareas**

  - ***pragma HLS dataflow***: permite realizar un *pipeline* de las tareas realizadas con los bloques RTL generados a partir de las funciones.

  - ***pragma HLS shared***: permite designar una variable local como una sola memoria compartida entre todas las funciones o puertos que acceden a ella.

  - ***pragma HLS stream***: esta directiva permite registrar los *arrays* a los que se accede de manera secuencial en **FIFOs**, lo que permite realizar una comunicación más eficiente entre funciones del diseño.

- ***Pipeline a nivel de instrucciones***
    - ***pragma HLS pipeline***: permite que las operaciones de una función o un bucle se ejecuten concurrentemente, mejorando así la latencia del proceso.
- **Desenrollado de bucles**
    - ***pragma HLS unroll***: permite desenrollar los bucles, permitiendo que todas las iteraciones del mismo en las que no existen dependencia de datos se ejecuten de forma paralela, mejorando la latencia del diseño y, consecuentemente, incrementando linealmente la utilización de recursos.
    - ***pragma HLS dependence***: permite designar las dependencias (o no dependencias) de datos entre las variables dentro de un bucle para que el sintetizador no detecte dependencias que no existen realmente.
- **Optimización de bucles**
    - ***pragma HLS loop_flatten***: permite unir varios bucles anidados en un solo bucle, lo que permite mejorar la latencia del proceso.
    - ***pragma HLS loop_merge***: une varios bucles consecutivos en un solo bucle, permitiendo así compartir recursos y optimizar la lógica del diseño.
    - ***pragma HLS loop_tripcount***: permite especificar manualmente el número de iteraciones totales que va a realizar un bucle.
- **Optimización de *Arrays***
    - ***pragma HLS array_partition***: permite dividir la memoria en porciones más pequeñas que se implementarán en recursos independientes, aumentando el consumo de estos pero permitiendo un número mayor de accesos concurrentes a la memoria, lo que se traduce en un mejor *throughput*.

- ***pragma HLS array_reshape***: permite modificar las dimensiones de un *array*, pudiendo formar matrices a partir de cadenas o modificando las dimensiones de una memoria n dimensional, siempre y cuando contenga el mismo número de elementos.

CAPÍTULO 4: Diseño del Bloque IP

En este capítulo se describirá el proceso de desarrollo del bloque IP que se usó para generar los distintos resultados que se comparan en este Trabajo de Fin de Máster, proceso que se dividirá en dos partes. En la primera parte, se tomó el código en Python de la capa convolucional y se desarrolló un código equivalente en C++ de una capa que tuviera la misma estructura pero que estuviera escrita en un lenguaje compatible con Vitis HLS. En la segunda parte, se tomó esta capa desarrollada en C++ y se modificó con el fin de que se pudiera generar un bloque IP a partir de ella que además presentara un buen rendimiento desde el punto de vista de la latencia y del consumo de recursos lógicos.

En ambas partes se comprobaron los valores de salida de la capa desarrollada con los de la capa en Python que proporcionó el equipo del IUMA, que se usó como *golden reference*, comprobando que ambos fueran iguales. De esta manera, en el caso de que se incluya la capa desarrollada en la red neuronal, esta no añadiría errores al resultado final, al estar

completamente verificada de forma independiente.

4.1 Transcripción de Python a C++

En esta sección se describirá el procedimiento que se siguió para generar una capa convolucional en C++ a partir del código de una capa escrito en Python. Para esto, primero se estudió tanto la estructura del código en Python como los cálculos que se realizaban internamente en la capa. Luego, se analizó cuales eran los elementos incompatibles entre los dos lenguajes y cómo se podrían sustituir estos elementos. Por último, una vez transcrita la capa, se analizaron las características de la solución que se había generado y se comprobó que los resultados de ambas capas coincidían.

4.1.1 Estructura del Código en Python

Para que los resultados de ambas capas fueran comparables, el proceso para generarlos debe de ser el mismo. Por esto, se estudió la estructura del código en Python para que a la hora de desarrollar el código en C++ ambos procedimientos fueran iguales desde un punto vista computacional. La estructura del código se muestra en la figura 4.1.

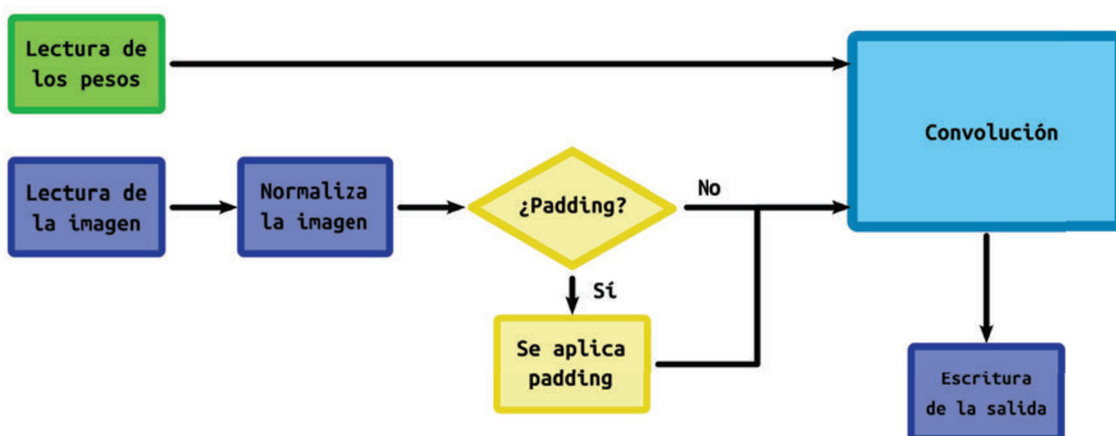


Figura 4.1: Estructura del código de Python.

Como se puede observar, el primer paso es leer tanto la imagen como los pesos, que se leerán desde archivos *.png* y *.txt*, respectivamente. Una vez que se tienen ambos elementos en memoria, se normaliza la imagen y se aplica *padding* en esta, si se ha seleccionado esta opción antes de ejecutar la capa. Usando las dimensiones de la imagen (teniendo en cuenta

el aumento de ésta si se aplicó *padding*), las dimensiones de los pesos y el desplazamiento del kernel que se introduce como parámetro de entrada, el modelo calcula las dimensiones que tiene que tener la salida. Una vez se cuenta con los datos de entrada y las dimensiones de la salida, se realiza la convolución.

Durante este proceso, se iteran sobre los datos de entrada siguiendo la estructura que muestra el pseudocódigo representado en el bloque 4.1. En el código se muestra como, por cada filtro, se itera horizontal y verticalmente, almacenando cada vez en una variable la parte de la imagen sobre la que se va a realizar el cálculo. En los índices de la imagen, **sh** representa el desplazamiento vertical del kernel, **sw** representa el desplazamiento horizontal, y **kh** y **kw** son la dimensión vertical y horizontal del kernel, respectivamente.

```
1 for f in num_filters:
2     for h in out_rows:
3         for w in out_columns:
4             conv_block = image[h*sh - h*sh+kh, w*sw - w*sw+kw, :]
5             out[h, w, f] = sum(conv_block * weights[:, :, :, f])
```

Bloque de Código 4.1: Convolución en Python

Una vez que se ha generado la salida, esta se escribe en un fichero para poder comparar los resultados con las demás capas que se han generado.

4.1.2 Modificaciones realizadas

Durante la transcripción del código, surgieron algunas incompatibilidades que impedían trasladar totalmente la funcionalidad de uno de los lenguajes al otro. Por esto, hicieron falta algunas modificaciones para poder adaptar ambos códigos, de manera que el procedimiento que siguieran ambos para generar una salida fuera lo más similar posible. Estas modificaciones se listan a continuación:

- **Tipo de ficheros:** para acceder a las imágenes, la capa escrita en Python las leía directamente desde el formato ".png". Debido a que C++ no es capaz de leer ficheros

desde este formato, se opto por pasar tanto la imagen, los pesos y la salida a un tipo de fichero binario o ".bin", que es accesible desde ambos lenguajes. Junto con esto, se tuvo que modificar algunas de las funciones para que pudieran acceder a archivos binarios. Concretamente, se modificaron aquellas funciones que se usaban para leer y escribir ficheros, y comprobar errores entre los resultados de varias capas.

- **Orden de los datos:** a la hora de extraer la información de los ficheros, se tuvieron que hacer modificaciones en la forma en la que se interpretaban los datos, debido a que la librería **numpy** de Python cambiaba las dimensiones de los datos de entrada. A la vez que se hacía esto, se adaptó el código para que se pudiera leer y escribir tanto en formato **BIL** (*Band Interleaved by Line*) como en formato **BSQ** (*Band Sequential*).

- **Librerías:** en Python existen librerías que no cuentan con un equivalente en C++. Debido a que en la capa de partida se usaban algunas de estas librerías, se tuvieron que realizar algunas funciones del código usando la estructura base de C. Para el cálculo de la convolución, la capa de partida usaba matrices y funciones propias de la librería **numpy** de Python. Debido a que no existe ningún equivalente en C++ soportado por Vitis HLS, se optó por desarrollar el cálculo de la capa usando las operaciones básicas de C y bucles que recorrieran los datos de entrada.

- **Acceso a la memoria:** los accesos a memoria en la capa de partida se hacen a través de *arrays* de **numpy**, un objeto de la librería que permite hacer cálculos sobre todos los elementos a la vez. Debido a que no es posible usar este tipo de objetos en C++, se optó por almacenar los datos en *arrays* de elementos, siendo tanto las dimensiones como el número de elementos de estas cadenas conocidos de antemano. El acceso a los valores de los datos de entrada se hizo mediante punteros y los cálculos iterando con bucles sobre los elementos de la memoria.

- ***Padding:*** debido a que el *padding* de la imagen de entrada se hacía a través de una librería, se tuvo que generar un proceso para realizarlo en la capa escrita en C++. Este proceso se implementó directamente durante la lectura de la imagen, reservando más cantidad de memoria y escribiendo ceros en aquellos valores que correspondieran a la zona ampliada.

4.1.3 Resultados de la capa en C++

Una vez terminada la transcripción del código, se comprobó que funcionaba correctamente siguiendo el proceso que se muestra en la figura 4.2. En la imagen está representado cómo ambos códigos leen tanto la imagen como los pesos con sus respectivas funciones. Una vez ejecutado todo el proceso de convolución, se escriben ambas salidas en un fichero binario y un tercer código, escrito en Python, se encarga de calcular el error entre ambos ficheros.

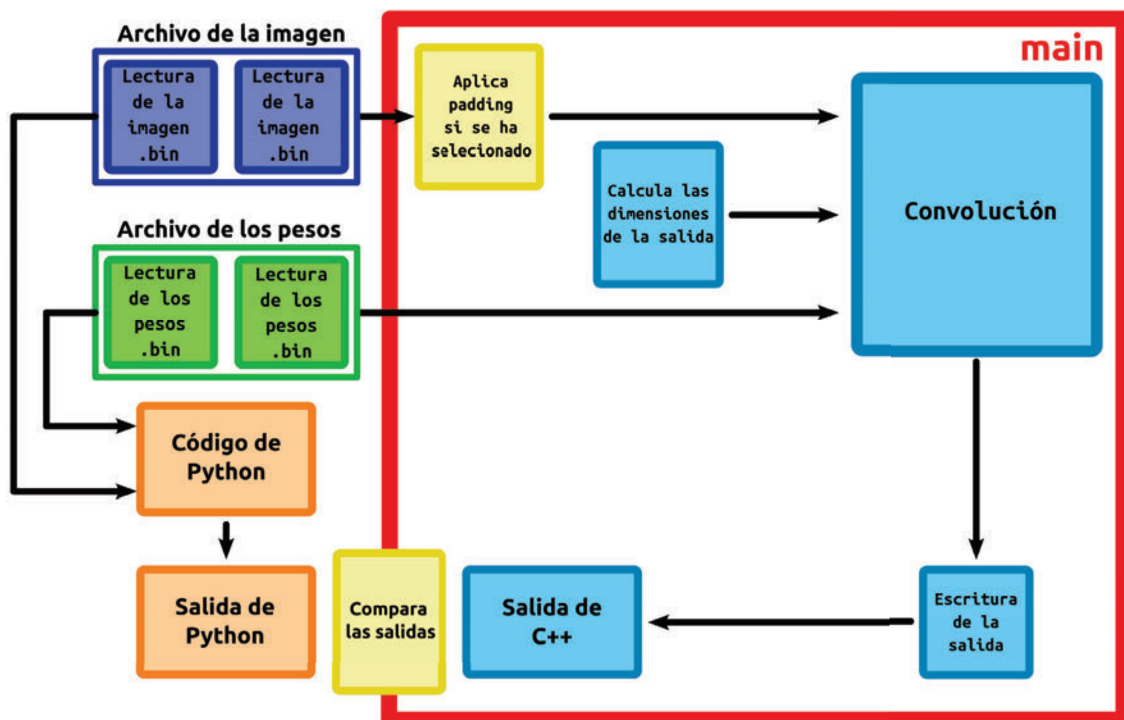


Figura 4.2: Gráfico con la estructura de la capa en C++.

El proceso se realizó tanto para una imagen con *padding* como sin él, obteniendo en ambos casos que el error entre la salida de ambas capas era 0. Una vez comprobado que se había transcrito la capa de un lenguaje a otro de forma correcta, se procedió a adaptar la capa para que pudiera ser sintetizada y que su ejecución tuviera buenas prestaciones, tanto en términos de latencia como de consumo de recursos lógicos.

4.2 Adaptación del código a Vitis HLS

En la sección 3.3.2.1 se enumeran todos los pasos que generalmente hay que seguir para adaptar un código en C++ general a uno que sea sintetizable por la herramienta de Vitis HLS. Aparte de estas modificaciones, para adaptar el código de la capa que se describe en el apartado anterior se tuvieron que hacer cambios adicionales más relacionados con la estructura del propio código. Por tanto, en esta sección se enumerarán los cambios que fueron necesarios para conseguir que el código fuera sintetizable y se mencionarán qué mejoras se realizaron para mejorar el rendimiento del código.

4.2.1 Cambios estructurales de la capa

El código de la capa de la que se partió cargaba la imagen, los pesos y la salida en memoria antes de realizar el cálculo. Esto, si bien presenta una mejora en el rendimiento del cálculo, ya que no se tienen que estar haciendo accesos a memoria constantemente, tiene unos requerimientos de recursos de almacenamiento muy altos. A la hora de implementar la solución *hardware*, incluso al realizar la simulación en según qué equipos, se pueden necesitar más recursos de los que hay disponibles.

Además de esto, la cámara que proporciona las imágenes a la capa envía sus datos en formato **BIL**, es decir, enviando las tres bandas RGB de cada fila de píxeles de la imagen cada vez. Teniendo esto en cuenta, se ideó una serie de cambios en la estructura de la capa que reducen el consumo de recursos, haciendo así posible la implementación de la capa. En los siguientes apartados se enumerarán los cambios realizados.

4.2.1.1 Gestión de la memoria

A la hora de realizar la convolución, si el desplazamiento que se realiza del kernel es menor que el tamaño del propio kernel, será necesario reutilizar parte de la imagen de entrada, lo que implica que se deberán almacenar estos datos de alguna manera. La opción por la que se optó en esta versión, sustituyendo a lo que se hacía en la capa en Python y C++

originalmente, en la que se almacenaba por completo la imagen, consiste en almacenar el mínimo número de información necesaria e ir desplazando la memoria que se va a reutilizar, sobrescribiendo aquellos datos que ya no sean necesarios para el cálculo.

El número mínimo necesario de datos es al menos tantas líneas de la imagen como filas tenga el kernel. De esta manera, suponiendo que el kernel cuente con tres filas, cada vez que lleguen datos de la imagen, que consistirán en las tres bandas de una de las filas de la imagen RGB, se almacenarán en una de las posiciones de la memoria hasta completarla. Este proceso se muestra en la figura 4.3.

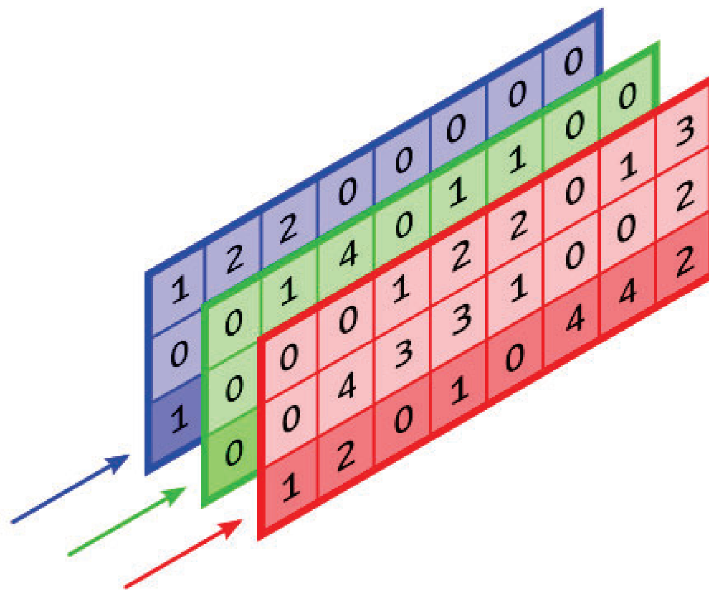


Figura 4.3: Almacenamiento fila a fila de la imagen.

El número de filas que se deberán reutilizar será igual al número total de filas del kernel menos el desplazamiento vertical de este. Esto es, si se supone el mismo kernel de tres filas y un desplazamiento de dos filas, una vez se termine de procesar la última fila, esta se desplazará a la primera fila. Una vez que lleguen nuevos datos, estos se almacenarán en la fila siguiente a la reasignada, como se ilustra en la figura 4.4.

Como los pesos de la capa son constantes durante todo el proceso, estos irán almacenados en memoria y se accederá a ellos cuando sea necesario. Por último, para almacenar la salida de la capa se crea una memoria que contendrá una fila y tantas columnas como se haya calculado para la salida, y esto se repetirá tantas veces como filtros tengan los pesos.

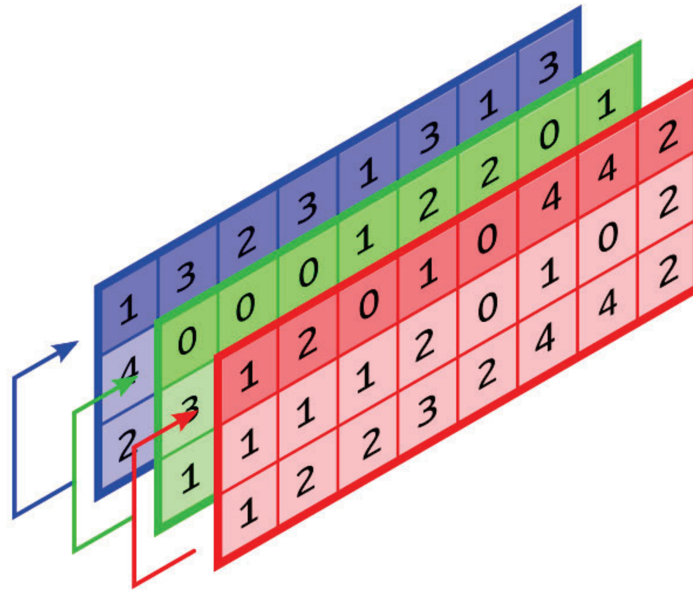


Figura 4.4: Desplazamiento de la fila a reutilizar hacia la primera posición de memoria.

Una vez que se termine de calcular la fila de la salida, ésta se escribirá y se resetearán los valores en memoria a 0.

4.2.1.2 Orden de Iteración

El orden en el que se itera en el código de partida se muestra en el bloque de código 4.1. El bloque IP que se ha diseñado se activará cada vez que reciba datos de la cámara, ya sea para almacenar estos datos en memoria o para procesarlos. Debido a esto, la imagen se iterará horizontalmente dependiendo del número de veces que llegue información al bloque. La iteración de la imagen en el código desarrollado en Vitis HLS queda representada en el pseudocódigo del bloque 4.2.

```
for h in num_ejec:
    for w in out_columns:
```

```
        for f in num_filters:
            convolution_process()
            write_output(w, f)

        img_reasignation(w)
```

Bloque de Código 4.2: Iteración de la convolución en Python

El número de ejecuciones, representado con la variable `num_ejec`, es igual al número de filas de la imagen. La salida, como se muestra en el pseudocódigo, se recorre primero horizontalmente y luego se itera sobre los diferentes filtros. De esta manera, como se representa con la función `img_reasignation(w)`, cuando se termine de procesar las primeras columnas de una fila que deba ser reasignada, éstas se desplazan en la memoria sin necesidad de volver a iterar. Por otro lado, la función `write_output(w, f)` representa que, cada vez que finalice el cálculo de uno de los filtros que componen los resultados de la capa, éste se escribirá en la salida.

4.2.1.3 *Padding*

El proceso de *padding* entre la capa en C++ y la capa *hardware-friendly* que se desarrolló para generar el bloque IP también cuentan con una ligera diferencia. En el primer caso, la imagen se leía al completo y se le iba realizando el *padding* durante el proceso de lectura desde el fichero. Para la capa generada en Vitis HLS se optó por inicializar una memoria con tantos elementos de más como fueran necesarios para realizar el *padding*. Cuando se inicia esta memoria, se le asigna un 0 a todas las posiciones. A la hora de almacenar la imagen de entrada en la memoria, se hace colocando los valores y dejando tantas filas y columnas como sean necesarias, de manera que si, por ejemplo, el *padding* superior es de dos filas y el lateral izquierdo de dos columnas, el primer valor de la imagen se almacenará en la tercera columna de la tercera fila de la memoria.

4.2.2 Mejoras de las prestaciones de la capa

En este apartado se mencionarán aquellos cambios realizados en el código de la capa que afectaron positivamente a las prestaciones de ésta, tanto en términos de latencia como de uso de recursos.

Debido a que en este Trabajo de Fin de Máster se pretenden generar distintas soluciones que cambian la forma en la que se ejecuta la capa, a la hora de realizar una mejora de la latencia y del consumo de recursos en el cálculo se optó por encontrar una solución común a todas las soluciones que se iban a generar.

Entre todas las soluciones, el punto común es el que la imagen de entrada contará en todos los casos con tres bandas. Independientemente del tamaño de la imagen, del kernel o del desplazamiento que se realice de éste, el cálculo siempre contará con tres bandas, de forma equivalente a si la entrada fuera una imagen RGB. Por esto, a la hora de desarrollar la capa se ha optado por dividir las memorias que almacenan tanto la imagen como los pesos en tres, una por cada banda. A la hora de realizar el proceso de convolución, éste se realiza para las tres bandas en paralelo, sumando los resultados obtenidos en tantos registros de salida como columnas tengan los pesos y luego sumando estos registros en la salida.

Junto con esto, el uso de los *pragmas* de Vitis HLS para realizar un *pipeline* de los bucles y la modificación del orden sobre el que se iteraba la imagen también demostraron tener un efecto significativo en el rendimiento del cálculo.

4.3 Soluciones Generadas

Con fines comparativos, en este trabajo se han generado dos capas distintas. La primera de ellas, procesa las líneas de datos que recibe de la cámara una vez que han sido recibidas, a la que se denomina capa “line”; la segunda, almacena en memoria todas las líneas de datos y las procesa una vez que tiene tantas líneas de píxeles como filas tenga el kernel. A esta capa se la designa como “block”.

Ambas capas realizan el mismo procedimiento a la hora de realizar la convolución, pero

cuentan con ligeros cambios estructurales que ofrecen resultados distintos en las simulaciones que se han realizado. En esta sección se describirá la estructura de ambas capas y se explicará su funcionamiento.

4.3.1 Capa *Line*

La estructura de esta capa queda definida en el pseudocódigo que se muestra en el bloque de código 4.3. La función `load_weights()` hace referencia al proceso que almacena los pesos en memoria, almacenando cada banda en una memoria diferente. La función `load_image()` hace referencia al proceso en el que se almacena la línea de la imagen recibida de la cámara, almacenando de nuevo cada banda en una memoria diferente. Por otro lado, la función `process_reused_rows()` representa el proceso en el que se realiza la convolución sobre aquellas líneas de la imagen que son entradas comunes entre una fila de la salida y la siguiente.

```
if first:
    load_weights()

load_image()

for w in out_columns:
    for f in num_filters:
        convolution_process()
        if last_row:
            write_output(w, f)

    if reuse_row:
        img_reasignation(w)

if last_row:
    process_reused_rows()
```

Bloque de Código 4.3: Estructura de la capa Line

El funcionamiento del bloque IP, por tanto, será el siguiente: si es la primera vez que se

ejecuta, cargará los pesos en memoria. Una vez que reciba una línea de la imagen, realizará la convolución y guardará los resultados en la memoria de salida. Cuando reciba una línea que sea entrada común entre dos líneas de la salida, desplazará esta línea de la memoria a la posición que le corresponda. Una vez que se hayan recibido tantas líneas de la imagen como filas tenga el *kernel*, se escribirá la salida y se repetirá el proceso de convolución sobre aquellas filas que deban ser reutilizadas.

4.3.2 Capa *Block*

La estructura de esta capa, representada en el pseudocódigo del bloque 4.4, es muy similar a la anterior. Comparten las funciones de `load_weights()` y `load_image()`, pero varían en el proceso de cálculo de la convolución.

```
if first:
    load_weights()

load_image()

if mem_complete:
    for row in kernel_rows:
        for w in out_columns:
            for f in num_filters:
                convolution_process()
                if last_row:
                    write_output(w, f)

            if reuse_row:
                img_reasignation(w)
```

Bloque de Código 4.4: Estructura de la capa Block

Este bloque IP comienza igual que el anterior, cargando los pesos si es la primera iteración y almacenando los datos recibidos de la cámara en memoria. Sin embargo, la imagen no se empieza a procesar hasta que la memoria esté completa. Una vez que esto ocurre, se recorren las filas de la imagen y se genera la salida a partir de estas, reasignando las filas

que vayan a ser reutilizadas a la parte de la memoria que les corresponda y generando la salida.

4.4 Banco de pruebas

Para poder generar el código de las distintas soluciones y comprobar que el resultado de estas es el correcto, se ha desarrollado un código en Python al que se ha denominado **out_manager.py**. En esta sección se explicarán las partes que componen el código y el proceso que se sigue para generar y comprobar las soluciones que se han comparado.

El código está formado por dos partes. La primera parte del código se encarga de generar y almacenar en un directorio todos los archivos necesarios para simular, sintetizar y comprobar los resultados de la capa. La segunda parte, lee la salida generada por la capa en Python y la compara con todas las soluciones que se hayan generado a partir de la capa de Vitis HLS, dando el grado de error entre ambos modelos.

4.4.1 Generación de la capa

Para generar una solución en concreto se necesitan de antemano algunos archivos que se han de generar manualmente. Primero, se necesitan las imágenes que se vayan a procesar en formato binario, así como el archivo de los pesos que se vayan a aplicar a dicha imagen. Además de esto, es necesario contar con el código de la capa adaptado al tamaño de los pesos que se vayan a usar.

Una vez que se cuente con estos archivos, se podrá ejecutar el código para generar la solución, al que se le pueden asignar los siguientes parámetros de entrada:

- **-s / - -stride_size**, con el que permite seleccionar el desplazamiento de los pesos.
- **-is / - -image_size**, con el que permite seleccionar el tamaño de la imagen que se quiera usar, siempre y cuando se cuente con una archivo binario que contenga una imagen de ese tamaño.
- **-wf / - -weights_file**, que permite seleccionar el fichero del que se leerán los pesos.

- **-pd / - -padding**, a través del que se puede especificar, usando las opciones “same” y “valid” si se usa o no *padding*, respectivamente.

Una vez que se han especificado estos parámetros de entrada, y dependiendo de éstos, el código realiza el proceso que se detalla a continuación, pudiendo realizarse varias veces si se han especificado varios tamaños de imagen de entrada:

1. **Cálculo de la salida de la capa de Python:** el primer paso que realiza el código es ejecutar el programa que calcula los resultados de la capa desarrollada en Python, guardando la salida en un archivo binario y generando un archivo “.log” con los distintos parámetros de la capa.

2. **Obtención de los parámetros:** en el archivo “.log” se escriben todos los parámetros relacionados con la capa, tales como el *padding* realizado o el tamaño de la salida. El siguiente paso que realiza el programa es almacenar en un diccionario de Python todos estos parámetros para generar la solución a partir de ellos.

3. **Creación del fichero de la solución:** una vez que se tienen los parámetros de la solución, se crea un código único a partir de estos y se genera un directorio llamado como ese código y en el que se almacenarán todos los archivos necesarios.

4. **Creación del archivo de cabecera:** lo siguiente que realiza el código es crear un fichero de cabecera para la capa de Vitis HLS en el que se almacenarán todos los parámetros necesarios como *define* de C. Este fichero llevará el nombre de **“conv_layer_header.h”**.

5. **Selección de las capas:** si bien se parte de una estructura base para la capa, cuando se cambian parámetros como el tamaño de los pesos la capa sufre pequeñas modificaciones en el código. Por esto, se cuenta con un programa que elije la capa que le corresponde a cada solución, dependiendo de los parámetros de esta, y los almacena en el fichero que se ha creado para la solución. Este programa está preparado para que si se tienen varios tipos de capa, las desplace todas para poder hacer pruebas con ellas.

6. **Copia de los ficheros restantes:** el último paso consiste en copiar la salida de la capa de Python, los pesos y la imagen que se ha usado al fichero de la solución, de manera que este contenga todos los archivos necesarios para generar un bloque IP y comprobar los resultados.

4.4.2 Comparación de los resultados de la capa

Una vez que se haya realizado la simulación de la capa de Vitis HLS, el programa cuenta con una opción para comparar estos resultados con los que se generaron a partir de Python, pudiendo comprobar así la validez de la solución generada. Esta opción del programa admite dos parámetros de entrada:

- **-fc / - -file_code**, con el que permite seleccionar el fichero en el que se desea realizar la comprobación.
- **-fp / - -fixed_point**. Si este parámetro se encuentra a 0, se interpreta que las soluciones a comparar están en punto flotante. En caso contrario, permite especificar qué configuración de punto fijo se uso para esta solución.
- **-t / - -tolerance**, que permite seleccionar el máximo error absoluto que se permite al comparar ambos resultados de la capa.

Al ejecutar este código, éste busca en la carpeta cuyo código se le haya especificado todos aquellos ficheros que contengan la palabra “vivado” en su nombre, siendo esta la manera que se ha elegido para diferenciar las salidas de Python de las que se generan en Vitis HLS. Una vez que se hayan seleccionado las soluciones, se leen los ficheros que las contienen y se comparan con la solución de Python, obteniendo parámetros de los ficheros que se listan a continuación:

- **Maximum Reference Values**, que es el valor máximo del archivo con el que se comprueban las salidas.
- **Average Reference Value**, que se trata del valor medio del archivo con el que se comprueban las salidas.

- **Average Absolute Error**, que representa el valor medio del error absoluto.
- **Maximum Absolute Error**, que representa el máximo valor de los errores absolutos.
- **Averager Relative Error**, que indica en tanto por ciento la media del error obtenido con respecto al valor de la salida.

A partir de estos valores y de la tolerancia que se le especifique ,se obtendrá si la capa propuesta es viable o si presenta algún error en ella. Todos los resultados de la comparación de las soluciones se almacenan en un archivo denominado “check.log” que se almacena en la carpeta de las soluciones.

4.4.3 Proceso de generación de las soluciones

Teniendo en cuenta las funcionalidades descritas en las secciones anteriores, el proceso para generar cada solución será por tanto el que se enumera a continuación:

1. **Generación de la solución:** usando el programa que se ha desarrollado para ello, se genera la carpeta que contiene todos los archivos necesarios para ejecutar la solución.
2. **Simulación de la solución:** una vez generados, se transportan todos los archivos a un proyecto en Vitis HLS y se realiza la simulación en C de los archivos para comprobar que no existe ningún error en ellos y se guarda la salida generada en un fichero.
3. **Síntesis de la capa:** se realiza la síntesis de la capa con la configuración que se haya seleccionado y se guardan los resultados de esta síntesis.
4. **Comprobación de los resultados:** se comprueba con el código en Python si existe un error entre la salida simulada y la salida de referencia, y si el error es nulo o admisible se admite la solución.

A través de este proceso, se ha agilizado la generación de las soluciones y se ha comprobado que cada una de ellas sería viable a la hora de implementarla como una capa dentro de la red.

CAPÍTULO 5: Resultados Obtenidos

En este capítulo se describirán todas las soluciones generadas en este Trabajo de Final de Máster y se compararán todos los resultados obtenidos de la síntesis de cada una de ellas, comprobando el impacto de las modificaciones en el rendimiento de la capa.

5.1 Parámetros de Entrada

Todas las variaciones que se han realizado en los parámetros de entrada de la capa se han realizado para las dos soluciones descritas en la sección 4.3, comparando no solo cómo afectaban las variaciones en una misma capa, sino si afectaba de manera distinta en una capa o en otra.

Los parámetros de entrada que se han usado para generar las soluciones quedan lista-

dos y etiquetados en el cuadro 5.1:

Número Identificador	Dimensiones de la Imagen	*Strides*	Dimensiones de los pesos	*Padding*
1	128·128·3	2·2	3·3·3·8	No
2	128·128·3	2·2	3·3·3·8	Sí
3	256·256·3	2·2	3·3·3·8	No
4	256·256·3	2·2	3·3·3·8	Sí
5	512·512·3	2·2	3·3·3·8	No
6	512·512·3	2·2	3·3·3·8	Sí
7	512·512·3	2·2	4·4·3·8	No
8	512·512·3	1·1	7·7·3·8	No
9	512·512·3	2·2	7·7·3·8	No
10	512·512·3	3·3	7·7·3·8	No
11	512·512·3	4·4	7·7·3·8	No

Cuadro 5.1: Parámetros de entrada de las capas

La solución que aparece destacada en el cuadro 5.1, es decir, la etiquetada con el número 6, es la configuración que tendrá la capa dentro de la red neuronal **Mobilenet**. Tomando esa configuración como referencia, las demás soluciones se han generado variando un parámetro cada vez. En las soluciones de la 1 a la 6, se han variado las dimensiones de la imagen de entrada comprobando, además de para el tamaño de imagen base, los resultados para una imagen de $128 \cdot 128$ píxeles y de $256 \cdot 256$, con 3 bandas en ambos casos. Además, se ha comprobado para todos los tamaños de imagen los resultados cuando a éstas se le aplicaba *padding* o no.

Para el segundo grupo de soluciones, conformado por las soluciones 5,7 y 9, se ha mantenido el tamaño de la imagen base sin aplicar *padding*, y se han variado el tamaño de los pesos, manteniendo el número de filtros y el número de bandas constante, y variando las dimensiones del *kernel* entre $3 \cdot 3$, $4 \cdot 4$ y $7 \cdot 7$.

Por último, para el grupo de soluciones 8 a 11, se ha mantenido el tamaño de la imagen de entrada y el *kernel* de $7 \cdot 7$ y se ha variado el desplazamiento que se realiza del *kernel* entre 1, 2, 3 y 4 píxeles, tanto en la dirección horizontal como en la vertical.

5.2 Comparación de Resultados

En este apartado se compararan los resultados obtenidos de la síntesis de las distintas configuraciones de las capas que se han desarrollado en este Trabajo de Fin de Máster. De esta síntesis se han obtenido tanto el consumo de recursos de la FPGA utilizados por cada una de las configuraciones como la latencia del proceso convolucional completo.

Para ambas capas, la latencia para almacenar tanto los pesos como la imagen en memoria es la misma, variando esta en función de las dimensiones de los pesos y de la cantidad de píxeles que tenga cada línea de la imagen.

Los recursos usados por las distintas soluciones en el dispositivo final rondan siempre un porcentaje de utilización total inferior al 3 % en DSPs, 2 % en BRAMs y en LUTs y 1 % en *Flip-Flops*. Esto se debe a que, debido a que la capa es parte de una red neuronal, a la hora de desarrollar las capas se priorizó que ocuparan pocos recursos para que fuera viable implementar la red completa, compuesta por capas interconectadas de forma sucesiva.

Por último, a la hora de representar el consumo de recursos, éstos se expresan como el tanto por ciento que ocupa cada recurso en función al máximo uso de dicho recurso que se hace entre las soluciones. Se ha elegido este sistema para poder representar todos los recursos en una misma gráfica con la misma escala.

5.2.1 Variación del tamaño de las imágenes

El primer grupo de resultados es el representado en el cuadro 5.2, que son aquellas configuraciones en las que se han variado las dimensiones de la imagen de entrada, manteniendo los demás parámetros constantes. Además de esto, se compararon los resultados obtenidos cuando se le aplicaba *padding* a la imagen y cuando no.

Número Identificador	Dimensiones de la Imagen	*Strides*	Dimensiones de los pesos	*Padding*
1	128·128·3	2·2	3·3·3·8	No
2	128·128·3	2·2	3·3·3·8	Sí
3	256·256·3	2·2	3·3·3·8	No
4	256·256·3	2·2	3·3·3·8	Sí
5	512·512·3	2·2	3·3·3·8	No
6	512·512·3	2·2	3·3·3·8	Sí

Cuadro 5.2: Parámetros del primer grupo de resultados

5.2.1.1 Latencia del Proceso

En las tablas representadas en el cuadro 5.3 se pueden observar los resultados de latencia del proceso.

Comenzando por la capa *line*, representada en el cuadro azul, se observa que según va aumentando el tamaño de la imagen, la latencia necesaria para generar una salida aumenta casi linealmente, siendo los resultados normalmente el doble de la latencia de la solución anterior. Esto demuestra que, sabiendo que el tamaño de la imagen se duplica entre soluciones, la capa diseñada guarda una relación directa con el tamaño de la imagen que se esté usando. Por otro lado, en la latencia total, si bien el aumento sigue siendo lineal, los resultados son cuatro veces mayores que el anterior, debido a que también se tiene en cuenta en este cálculo el aumento vertical de la imagen.

En cuanto a la diferencia entre aplicar *padding* y no aplicarlo, existe un aumento en las latencias, pero no es significativo. Sin embargo, es destacable que en algunos casos, al aumentar el tamaño de la imagen, el resultado generado con *padding* tiene una latencia menor al que no cuenta con *padding*. Esto es posible ya que, de antemano, se hace una reserva de memoria para el peor caso, que es en el que se aplica *padding*, y dichas posiciones se inicializan con valor 0, por lo que la aplicación de *padding* no supone una latencia extra durante el cómputo de la convolución.

Respecto a la capa *block*, los resultados son muy similares a los obtenidos con la solu-

Número Identificador	Mínima (ns)	Máxima (ns)	Convolución (ns)	Total (ms)
1	31 020	49 720	32 760	3,877
2	31 450	49 760	32 640	3,922
3	62 380	97 090	66 040	15,614
4	62 820	96 490	65 280	15,624
5	125 000	192 000	133 000	62,784
6	125 530	190 000	131 000	62,487

Número Identificador	Mínima (ns)	Máxima (ns)	Convolución (ns)	Total (ms)
1	3 920	104 930	96 840	6,595
2	3 920	106 490	99 840	6,884
3	7 760	208 620	198 050	27,121
4	7 760	210 180	199 680	27,528
5	15 440	415 980	397 800	109,306
6	15 440	417 530	399 360	110,103

Cuadro 5.3: Latencias obtenidas de las capas *Line* (azul) y *Block* (verde) al variar el tamaño de la imagen

ción anterior. Las relaciones entre las latencias siguen siendo lineales y cuentan con la misma tendencia creciente. A la hora de aplicar el *padding*, el aumento en la latencia es mayor en este caso, aunque sigue sin ser significativo y, a diferencia de los resultados anteriores, no se observan tiempos de ejecución menores en las soluciones con un tamaño de imagen mayor.

En la figura 5.1 se muestra la comparación en el tiempo de ejecución total entre la capa *line* y la capa *block*. En color claro, se muestran aquellas soluciones a las que no se le aplicó *padding*. En color más oscuro, aquellas soluciones a las que sí. En todos los casos, las latencias obtenidas de la capa *line* son inferiores a las de la capa *block*, llegando a haber una diferencia del 43,24 % entre ambas en la solución 6.

La máxima diferencia observada entre una solución a la que no se le aplicó *padding* y una a la que sí es de un 1,15 % en la capa *line* y de un 4,198 % en la capa *block*, dándose

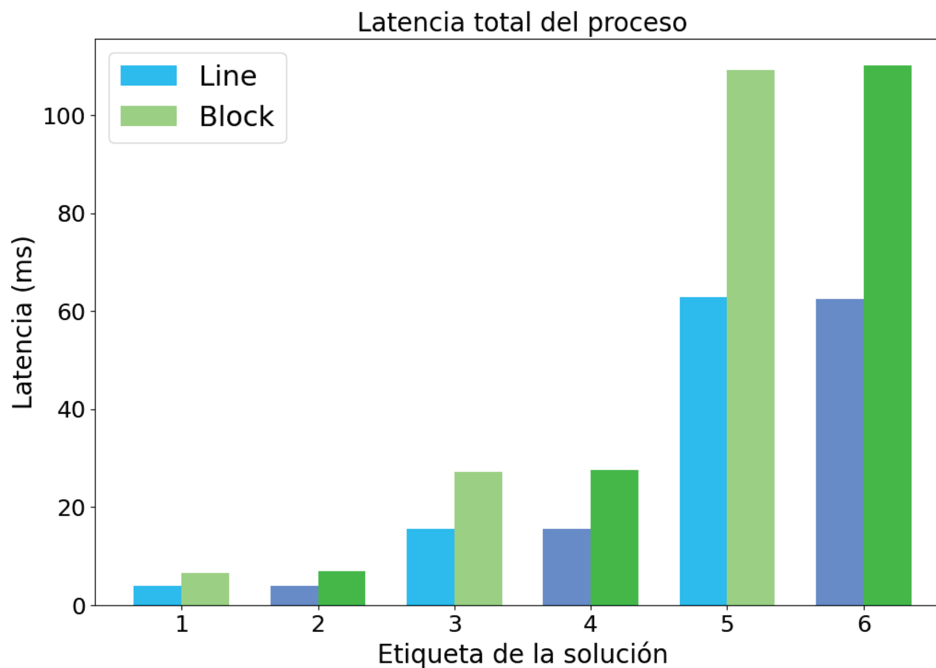


Figura 5.1: Comparación entre las latencias totales de las dos capas

esta diferencia en el primer par de soluciones (solución 1 y 2). Esto no supone un aumento significativo de la latencia, aunque tiene más importancia en la capa *block*. Los demás pares de soluciones (soluciones 3 y 4 y soluciones 5 y 6) presentan una diferencia de un 0,06 % y 0,04 % en la capa *line* respectivamente, y de un 1,48 % y un 0,72 % en la capa *block*. Las diferencias son, por tanto, poco significativas entre una configuración y otra, aunque mayores en la capa *block* que en la capa *line*.

5.2.1.2 Consumo de recursos de las soluciones

Los recursos usados por las soluciones de este grupo se muestran en el cuadro 5.4. En la tabla azul se muestran los resultados obtenidos para los distintos parámetros de entrada en la capa *line*. Se observan en estos resultados que los recursos de la capa permanecen prácticamente constantes entre todas las soluciones que se muestran, variando únicamente las memorias necesarias en aquellas imágenes con unas dimensiones mayores. Entre las soluciones con o sin *padding*, no se observan diferencias significativas.

La capa *block* presenta unos resultados parecidos a la anterior. Los recursos usados en

Número Identificador	BRAM (tot/ %)	DSP (tot/ %)	FF (tot/ %)	LUT (tot/ %)
1	14 / 1	25 / 1	3499 / $\sim$ 0	4041 / 1
2	14 / 1	25 / 1	3547 / $\sim$ 0	4041 / 1
3	14 / 1	25 / 1	3532 / $\sim$ 0	4051 / 1
4	14 / 1	25 / 1	3584 / $\sim$ 0	4066 / 1
5	22 / 1	25 / 1	3565 / $\sim$ 0	4062 / 1
6	22 / 1	25 / 1	3621 / $\sim$ 0	4093 / 1

Número Identificador	BRAM (tot/ %)	DSP (tot/ %)	FF (tot/ %)	LUT (tot/ %)
1	13 / 1	25 / 1	3410 / $\sim$ 0	3612 / 1
2	13 / 1	25 / 1	3439 / $\sim$ 0	3679 / 1
3	14 / 1	25 / 1	3437 / $\sim$ 0	3623 / 1
4	14 / 1	25 / 1	3467 / $\sim$ 0	3689 / 1
5	22 / 1	25 / 1	3467 / $\sim$ 0	3634 / 1
6	22 / 1	25 / 1	3495 / $\sim$ 0	3716 / 1

Cuadro 5.4: Recursos utilizados por las capas *Line* (azul) y *Block* (verde) al variar los tamaños de la imagen (con o sin *padding*)

las distintas soluciones se mantienen constantes, por lo que se puede afirmar que a excepción de la memoria, los recursos necesarios para implementar las capas en la FPGA no se ven afectados por el tamaño de la imagen. tanto la capa *line* como la capa *block* consumen la misma cantidad de recursos de memoria y DSPs.

En el cuadro 5.5 se muestra la diferencia en % de FF y LUTs entre los recursos de la capa *line* y la capa *block*. Como se observa en dicho cuadro, el consumo de *Flip-Flops* y de

Número Identificador:	1	2	3	4	5	6
FF (reducción de *block* respecto a *line*)	2,54 %	3,04 %	2,69 %	3,26 %	2,75 %	3,48 %
LUT (reducción de *block* respecto a *line*)	10,62 %	8,96 %	10,57 %	9,05 %	10,53 %	9,21 %

Cuadro 5.5: Diferencias entre los recursos utilizados de ambas capas al variar el tamaño de la imagen (con y sin *padding*)

lógica de la capa *block* es menor que el de la capa *line* para todas las soluciones que se han comparado en este grupo, variando entre un 10,62 y un 8,96 % de diferencia de LUTs y entre un 2,54 % y un 3,48 % de diferencia entre *Flip-Flops*.

5.2.2 Variación de las dimensiones de los pesos

Las soluciones que se van a estudiar en esta sección son la que se muestran en el cuadro 5.6. Estas soluciones corresponden a aquellas configuraciones en las que se ha mantenido constante tanto el tamaño de la imagen de entrada como el *stride* de la capa, pero se ha variado el tamaño de los pesos.

Número Identificador	Dimensiones de la Imagen	*Strides*	Dimensiones de los pesos	*Padding*
5	512·512·3	2·2	3·3·3·8	No
7	512·512·3	2·2	4·4·3·8	No
9	512·512·3	2·2	7·7·3·8	No

Cuadro 5.6: Parámetros del segundo grupo de resultados

5.2.2.1 Latencia del Proceso

Los resultados de latencia obtenidos al sintetizar el grupo de soluciones de esta sección se muestran en el cuadro 5.7. En la capa *line*, representada en azul, se observa como la variación que se producen en los tiempos de ejecución con pesos de dimensiones $3 \cdot 3$ (solución 5) frente a los tiempos que se observan en los pesos de $4 \cdot 4$ (solución 7) son bastante similares, aumentando debido a que se deben realizar un mayor número de cálculos para generar cada salida.

Por otro lado, la variación entre estas dos soluciones y la solución de kernel con dimensiones $7 \cdot 7$ es mucho más significativa, a pesar de que el tiempo que tarda en realizarse la convolución no llega a ser el doble que el de las soluciones anteriores. Esta diferencia se debe a que entre mayor es la dimensión de los pesos con respecto al desplazamiento que se realiza de estos, mayor es el número de veces que se realiza un cálculo con el mismo pa-

rámetro de entrada y el número de filas de la imagen que son reutilizadas de una salida a otra.

Número Identificador	Mínima (ns)	Máxima (ns)	Convolución (ns)	Total (ms)
5	125 000	192 000	133 000	62,784
7	127 660	252 000	150 000	77,502
9	174 840	1 386 000	220 000	410,704

Número Identificador	Mínima (ns)	Máxima (ns)	Convolución (ns)	Total (ms)
5	15 440	415 980	397 800	109,306
7	15 440	621 650	601 800	161,328
9	15 440	1 568 540	1 540 770	397,691

Cuadro 5.7: Latencias obtenidas de las capas *Line* (azul) y *Block* (verde) al variar los pesos

En cuanto a la capa *block*, representada en verde, se observa que el aumento en los tiempos de ejecución no es tan drástico como en la solución anterior. Esto se debe a que en la estructura de esta capa, la reutilización de las líneas de la imagen es parte del propio proceso de convolución y no se realiza como un proceso separado, como en la solución anterior. Además, se observa que el aumento de latencia entre la solución 7 y la solución 5 es mayor que el que se produce entre la capa 9 y la 7, teniendo en cuenta que en la primera comparación las dimensiones de los pesos aumentan en 1 unidad y en la segunda comparación aumentan tres unidades.

En la figura 5.2 se compara la latencia máxima de las dos capas para cada solución. En la gráfica se puede observar que, para las dos primeras soluciones, la capa *line* cuenta con un tiempo de ejecución menor que la capa *block*. Sin embargo, cuando comienza a aumentar la diferencia entre las dimensiones de los pesos y el *stride*, la capa *line* empieza a obtener peores resultados, llegando incluso a superar el tiempo de ejecución de la capa *block* en la última solución.

Una vez analizados los resultados, se observa que la capa *line* funciona mejor cuando se trabaja con kernel de dimensiones más pequeñas, mientras que la capa *block* mejora su rendimiento cuanto mayor es el tamaño de los pesos.

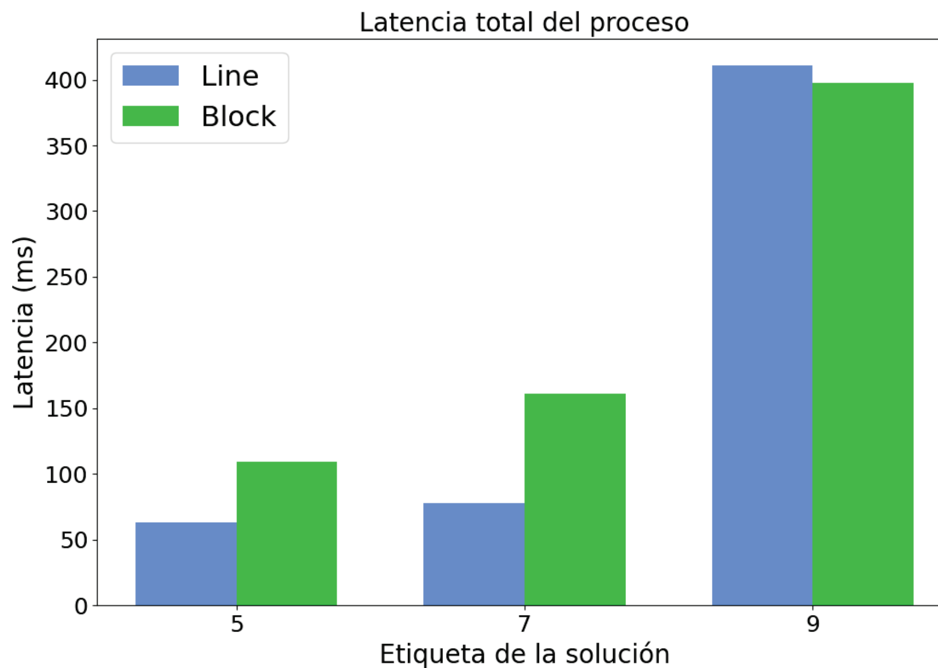


Figura 5.2: Comparación entre las latencias totales de las dos capas

5.2.2.2 Consumo de recursos de las soluciones

En el cuadro 5.8 se muestra el consumo de recursos de las distintas soluciones generadas, comenzando por la capa *line*, representada en azul, y seguida por la capa *block*, representada en verde.

En la primera de las capas, se puede observar que el número de memorias no cambia de la primera a la segunda solución, mientras que de la segunda a la tercera solución sí aumenta el número de memorias usadas. Esto se debe a que entre mayor es la dimensión del kernel, mayor es el número de filas que se deben almacenar y mayor es el consumo de memoria. En cuanto al consumo de DSPs, entre mayor es el número de filas del kernel, mayor cantidad de sumas se tendrán que hacer, por lo que se necesitarán un mayor número de DSPs para hacer las operaciones. Por último, analizando la lógica interna, se observa como aumenta cuanto mayores son las dimensiones de los pesos. En la capa *line* el aumento de los *Flip-Flops* es de un 24,32 % entre las soluciones 5 y 7 y de un 38,17 % entre las soluciones 7 y 9, mientras que las LUTs aumentan un 17,85 % y un 24,67 %, respectivamente.

Número Identificador	BRAM (tot/ %)	DSP (tot/ %)	FF (tot/ %)	LUT (tot/ %)
5	22 / 1	25 / 1	3565 / $\sim$ 0	4062 / 1
7	22 / 1	30 / 1	4432 / 1	4787 / 1
9	34 / 2	30 / 1	6124 / 1	5968 / 2

Número Identificador	BRAM (tot/ %)	DSP (tot/ %)	FF (tot/ %)	LUT (tot/ %)
5	22 / 1	25 / 1	3467 / $\sim$ 0	3634 / 1
7	22 / 1	30 / 1	4304 / $\sim$ 0	4170 / 1
9	34 / 2	30 / 1	5315 / 1	4846 / 1

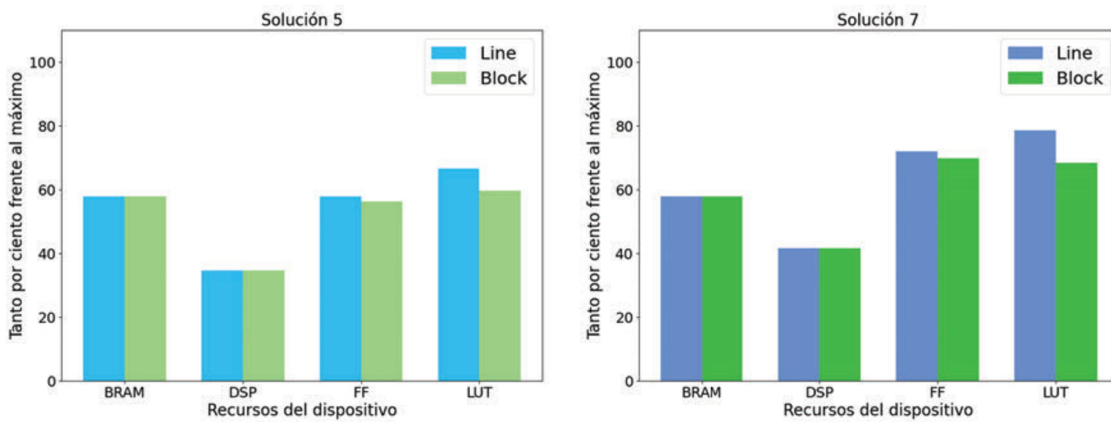
Cuadro 5.8: Recursos utilizados por las capas *Line* (azul) y *Block* (verde) al variar los pesos

En la segunda de las capas, la capa *block*, se observa que el consumo de las memorias y de los DSPs es el mismo que en la capa *line*. Esto es debido a que el cálculo que se hace en ambas capas y la manera de almacenar la información es la misma. En esta capa, los incrementos entre las soluciones 5 y 6 son de un 24,14 % en FFs y de un 14,75 % en LUTs; y entre las soluciones 7 y 9 los incrementos son de un 23,49 % en FFs y de un 16,21 % en LUTs. Si bien se observa un incremento entre las soluciones como en la capa *line*, el aumento que se produce entre ellas en esta capa es menor.

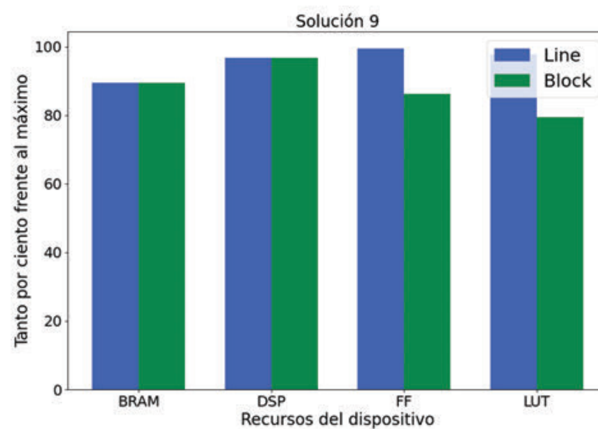
Como se puede observar en la figura 5.3, ambas capas cuentan con el mismo consumo de memorias y de DSPs en todas las soluciones. Sin embargo, la capa *block* tiene un consumo en *Flip-Flops* y LUTs menor que el de la capa *line*, siendo la diferencia en el caso de los *Flip-Flops* mayor cuanto más se aumentan las dimensiones de los pesos.

5.2.3 Variación del *stride* de la capa

En el cuadro 5.9 se muestran las etiquetas y los parámetros de entrada que se analizarán en esta sección. En esta sección, para tener un mayor margen, se han usado como base los pesos con dimensiones $7 \cdot 7$ y se han variado los *strides* de la capa para generar las soluciones. Se han usado como base estos pesos debido a que, si el desplazamiento del *kernel* es mayor que el propio *kernel*, algunas partes de la imagen quedarían sin procesar. De esta manera,



(a) Consumo de recursos al variar los pesos (Soluciones 5 y 7)



(b) Consumo de recursos al variar los pesos (Solución 9)

Figura 5.3: Comparación del consumo de recursos de ambas capas al variar las dimensiones de los pesos.

se ha tenido mayor margen para probar un mayor número de parámetros de entrada.

5.2.3.1 Latencia del Proceso

Los resultados de latencia obtenidos al sintetizar el grupo de soluciones de esta sección se muestran en el cuadro 5.10. La tabla azul representa los resultados de la capa *line* y, como se puede observar, entre mayor es el *stride*, el tiempo de ejecución de la capa disminuye. Esto se debe a que entre mayor es el desplazamiento del *kernel* sobre la imagen, menor es el número de soluciones que se tienen que generar.

De estos resultados, cabe destacar que la solución 8, aquella que cuenta con *strides* $1 \cdot 1$,

Número Identificador	Dimensiones de la Imagen	*Strides*	Dimensiones de los pesos	*Padding*
8	512·512·3	1·1	7·7·3·8	No
9	512·512·3	2·2	7·7·3·8	No
10	512·512·3	3·3	7·7·3·8	No
11	512·512·3	4·4	7·7·3·8	No

Cuadro 5.9: Parámetros del tercer grupo de resultados

es la solución con un mayor tiempo de ejecución de todas las generadas, siendo este 3,54 veces mayor que la solución con unos parámetros de entrada más similares, la solución 9.

Número Identificador	Mínima (ns)	Máxima (ns)	Convolución (ns)	Total (ms)
8	334 230	2 881 620	425 040	1 454,348
9	174 840	1 386 000	220 000	410,704
10	121 930	762 930	147 000	198,388
11	95 460	260 480	110 000	52,997

Número Identificador	Mínima (ns)	Máxima (ns)	Convolución (ns)	Total (ms)
8	15 440	2 578 020	2 550 240	1 298,298
9	15 440	1 568 540	1 540 770	397,691
10	15 440	891 360	863 590	177,140
11	15 440	801 200	773 430	106,102

Cuadro 5.10: Latencias obtenidas de las capas *Line* (azul) y *Block* (verde) al variar los *strides*

La latencia cuando se varía dicho parámetro de entrada en la capa *block* tiene un comportamiento similar a la de la capa anterior, con la diferencia de que el tiempo de ejecución entre las dos últimas soluciones (soluciones 10 y 11) disminuye un 40,10 % en la capa *block*, mientras que en la capa *line* disminuye un 73,28 %.

En la figura 5.4 se muestra cómo varía el tiempo total de ejecución de las distintas soluciones. La imagen muestra como la capa *block* tiene un tiempo de ejecución menor en los tres primeros casos, mientras que en el último caso la capa *line* ofrece unos mejores re-

sultados. La capa *line* presenta el inconveniente de que entre mayor es la diferencia entre los *strides* y el *kernel*, mayor es el número de filas que debe reasignar en memoria y volver a procesar esta capa, lo que la hace menos eficiente cuando esta diferencia se incrementa. Esta diferencia decrece con cada una de las soluciones presentadas, por lo que el rendimiento de la capa *line* aumenta.

5.2.3.2 Consumo de recursos de las soluciones

En el cuadro 5.11 se muestra el consumo de recursos de las distintas soluciones generadas. Comenzando por la tabla azul, en la que se muestran los recursos utilizados por la capa *line*, se puede observar como el número de memorias se va reduciendo a medida que se aumentan los *strides*. Esto se debe a que al aumentar el desplazamiento de la capa, también se reduce el número de salidas generadas y por tanto la memoria necesaria para almacenar estas salidas.

Por otro lado, debido a que los pesos no cambian y, por tanto, no debe hacerse un mayor número de sumas, los DSPs permanecen constantes, salvo por la última solución en la que

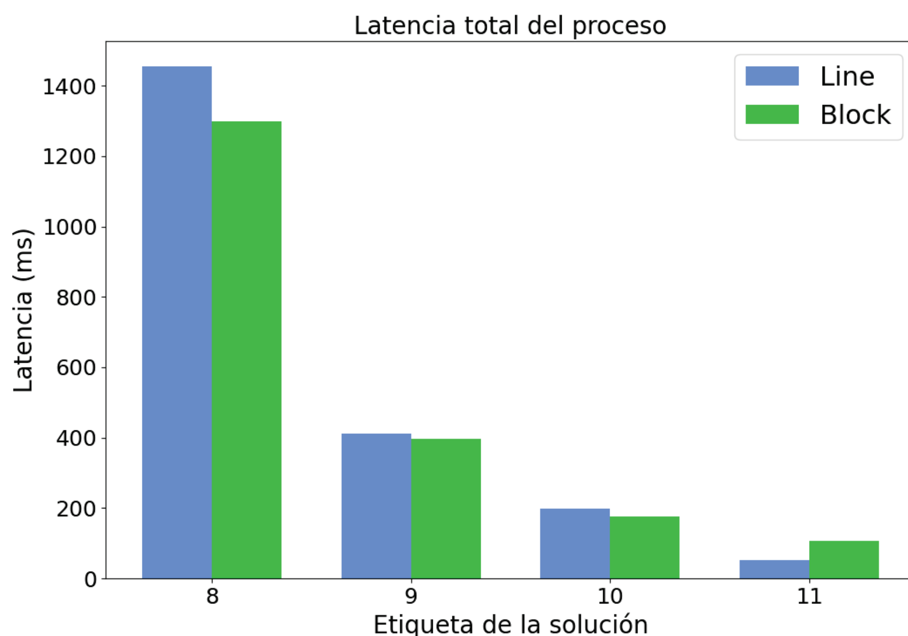


Figura 5.4: Comparación entre las latencias totales de las dos capas

se usa un DSP más. Esto es debido a que un *stride* mayor implica también una mayor complejidad de las operaciones, incluyendo las multiplicaciones, siendo los DSPs los recursos computacionales destinados a realizar estas operaciones de forma eficiente. En cuanto a los *Flip-Flops*, estos permanecen constantes, con pequeñas variaciones al igual que ocurre en el caso de las LUTs.

Número Identificador	BRAM (tot/ %)	DSP (tot/ %)	FF (tot/ %)	LUT (tot/ %)
8	38 / 2	30 / 1	6117 / 1	5939 / 1
9	34 / 2	30 / 1	6124 / 1	5968 / 2
10	34 / 2	30 / 1	6159 / 1	6099 / 2
11	32 / 2	31 / 1	5901 / 1	5573 / 2

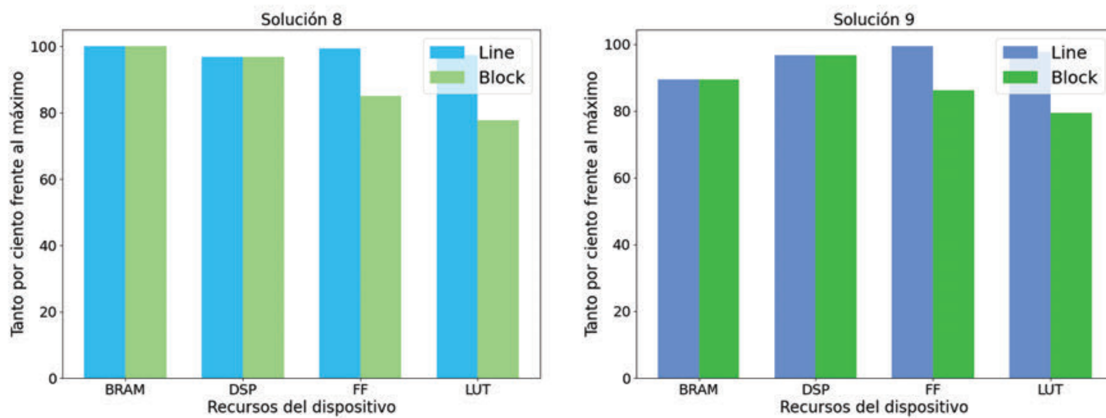
Número Identificador	BRAM (tot/ %)	DSP (tot/ %)	FF (tot/ %)	LUT (tot/ %)
8	38 / 2	30 / 1	5239 / 1	4745 / 1
9	34 / 2	30 / 1	5315 / 1	4846 / 1
10	34 / 2	30 / 1	5278 / 1	4853 / 1
11	32 / 2	30 / 1	5247 / 1	4782 / 1

Cuadro 5.11: Recursos utilizados por las capas *Line* (azul) y *Block* (verde) al variar el desplazamiento de los pesos

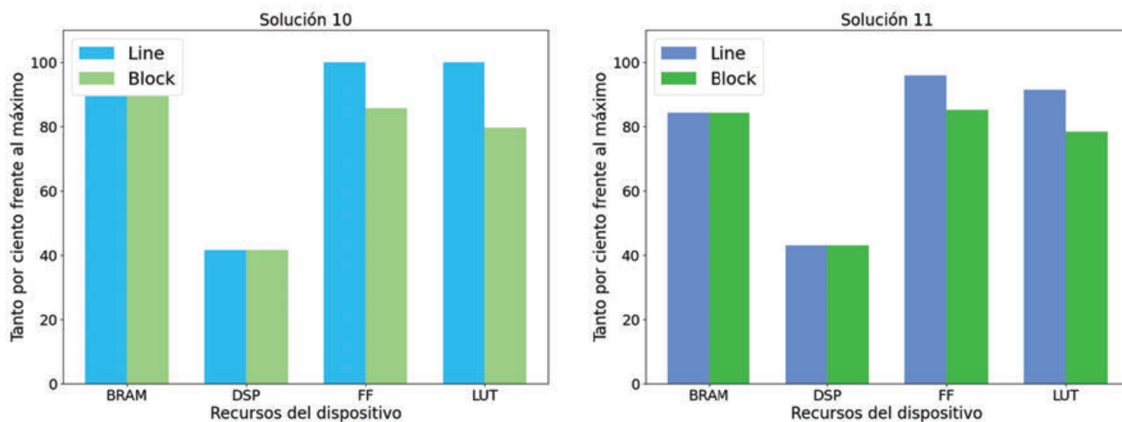
Los resultados de recursos de la capa *block* obtenidos de la síntesis se comportan como los de la capa *line*, manteniéndose constantes excepto por la memoria.

De estos resultados se deduce que el variar los *strides* no afecta significativamente en estas soluciones a los recursos utilizados por el dispositivo, más allá de la memoria necesaria para almacenar la salida de la capa.

En la figura 5.5 se muestra una comparación del consumo de recursos para todas las soluciones. Como se puede observar, al comparar la capa *line* con la capa *block*, los recursos consumidos de memoria y DSPs son los mismos en todos los casos; sin embargo, la capa *block* presenta un consumo menor de *Flip-Flops* y LUTs para todas las soluciones generadas.



(a) Consumo de recursos al variar el desplazamiento de la capa (Soluciones 8 y 9)



(b) Consumo de recursos el desplazamiento de la capa (Soluciones 10 y 11)

Figura 5.5: Comparación del consumo de recursos de ambas capas al variar los *strides*.

5.3 Tipo de datos

La síntesis de todas las soluciones que se analizaron en la sección 5.2 se hizo usando punto flotante como tipo de dato para la imagen, los pesos y la salida. Debido a que las FPGAs suelen ser más eficientes cuando se realizan los cálculos en aritmética entera, se decidió modificar este tipo de dato, sustituyéndolo por valores en punto fijo y así estudiar el efecto que tenía esto en el rendimiento de las capas.

Para realizar el cambio a punto fijo, se usó la librería `ap_fixed.h`, usando como tipo de dato `ap_fixed <32, 5, AP_RND>`. Dentro de la estructura que define el tipo de dato, el primer campo indica que el número será de 32 bits (lo mismo que ocupa un número en punto flotante en la plataforma hacia la que se orienta las soluciones generadas), el segundo valor

indica que se dedicarán 5 bits a la parte entera y el último representa el modo de cuantificación. Las soluciones con las que se realizó la síntesis variando este tipo de dato están representadas en el cuadro 5.12, siendo estas las configuraciones que se usarán en la red final.

Número Identificador	Dimensiones de la Imagen	*Strides*	Dimensiones de los pesos	*Padding*
5	512·512·3	2·2	3·3·3·8	No
6	512·512·3	2·2	3·3·3·8	Sí

Cuadro 5.12: Parámetros de entrada de las capas modificadas

5.3.1 Resultados de latencia obtenidos

La latencia obtenida de las soluciones se muestra en el cuadro 5.13. Como se puede observar, tanto para la capa *line* (azul) como para la capa *block* (verde), se mantiene la misma relación entre la solución con *padding* y la solución a la que no se le aplica, presentando ambas unos resultados en términos de latencia muy similares.

Número Identificador	Mínima (ns)	Máxima (ns)	Convolución (ns)	Total (ms)
5	71 550	122 740	63 750	45,018
6	71 770	123 150	63 750	45,228

Número Identificador	Mínima (ns)	Máxima (ns)	Convolución (ns)	Total (ms)
5	15 440	224 720	206 550	60,537
6	15 440	217 850	199 680	58,985

Cuadro 5.13: Latencias obtenidas de las capas *Line* (azul) y *Block* (verde) al variar el tamaño de la imagen

En la figura 5.6 se observan los tiempos de ejecución totales de las capas generadas con punto fijo frente a las generadas con punto flotante para ambas capas, *line* y *block*. Como se puede observar, el punto fijo mejora en carácter general los resultados de latencia y, después de comprobar los resultados generados por ambas capas con el generado por la capa de Python que se ha empleado como modelo de referencia, el error relativo que existe entre

los resultados es del 0.00001 %, por lo que el cambio de un tipo de dato a otro no introduce un error significativo en la solución.

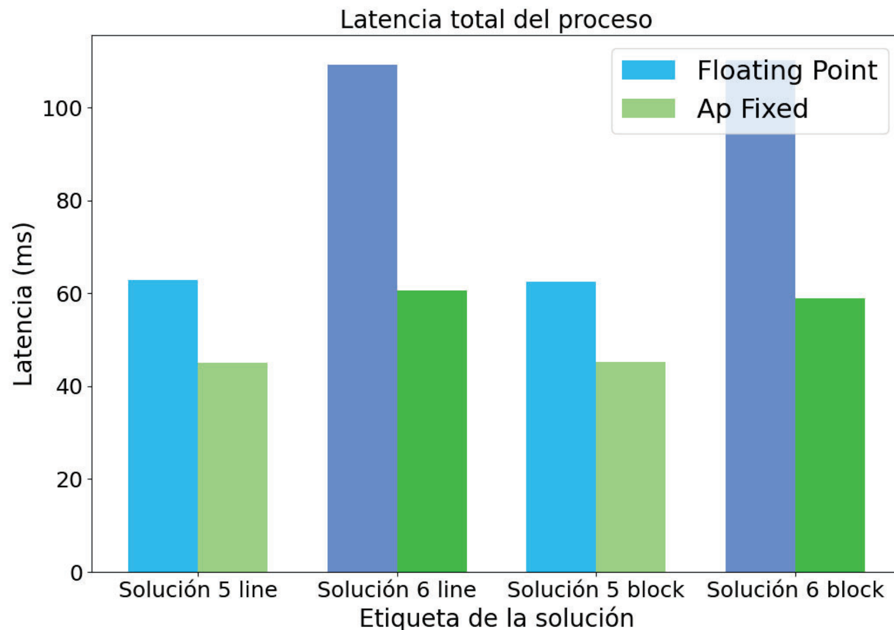


Figura 5.6: Comparación entre la latencia total de las capas con punto flotante y con punto fijo

Comparando ambas capas, la mejora de latencia obtenida en la capa *block* es mucho más significativa que la obtenida en la capa *line*, reduciendo el tiempo de ejecución total en un 44,62 % para la solución 5 y en un 46,43 % para la solución 6, frente a la reducción que se obtuvo en la capa *line* de un 28,30 % y un 27,62 %, respectivamente. Aunque la reducción fuera mayor para la capa *block*, la capa *line* sigue contando con un tiempo de ejecución menor.

5.3.2 Consumo de recursos de las soluciones

Los consumos de recursos obtenidos de la síntesis se muestran en el cuadro 5.14. Comenzando por la cantidad de memoria utilizada, los resultados obtenidos tanto entre las distintas soluciones analizadas como entre capas son los mismos. Esto se debe a que, al contar con los mismos parámetros de entrada, y sabiendo que el *padding* no aumenta significativamente el tamaño de la imagen, la información a almacenar en todos los casos es aproximadamente la misma.

En cuanto a los DSPs, para realizar el cálculo en la capa *line* se consumen el doble que para la capa *block*. Esto es debido a que existen en la capa *line* dos procesos iguales de convolución, el cálculo principal y el cálculo que se realiza para procesar las líneas redundantes.

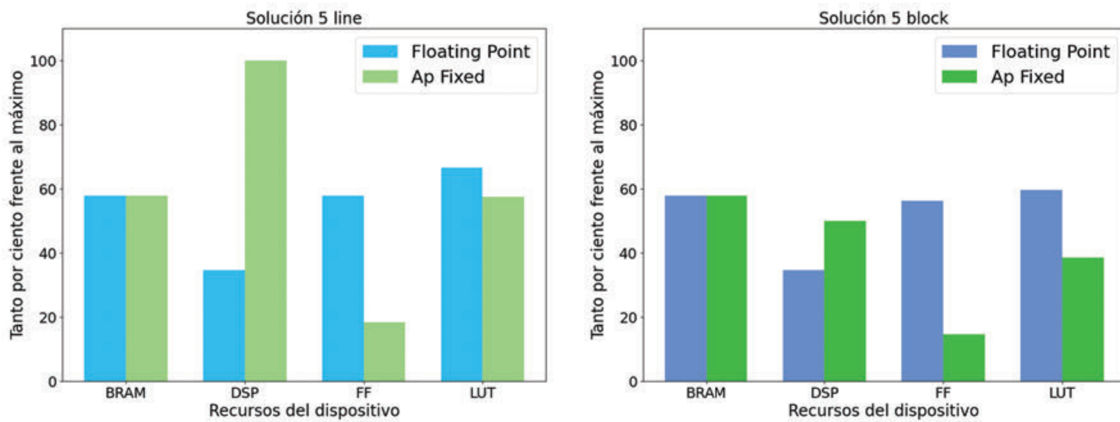
Analizando ahora el consumo de *Flip-Flops* y LUTs, la capa *block* presenta un menor consumo que la capa *line* en ambos recursos. Esto se debe a que la estructura de la capa es más sencilla, lo que hace que la lógica necesaria para implementarla sea menor. Por último, analizando la diferencia entre las soluciones dentro de la misma capa, tanto en la capa *line* como en la capa *block* se observa un aumento del número de *Flip-Flops*, siendo este del 9,31 % en la capa *line* y de 4,22 % en la capa *block*. Al contrario de lo que ocurre con los FFs, el número de LUTs se ve reducido entre soluciones de la misma capa, disminuyendo un 3,69 % en la capa *block* y un 0,29 % en la capa *line*, por lo que la disminución en esta última capa no es significativa.

Número Identificador	BRAM (tot/ %)	DSP (tot/ %)	FF (tot/ %)	LUT (tot/ %)
5	22 / 1	72 / 3	1138 / $\sim$ 0	3505 / 1
6	22 / 1	72 / 3	1244 / $\sim$ 0	3495 / 1

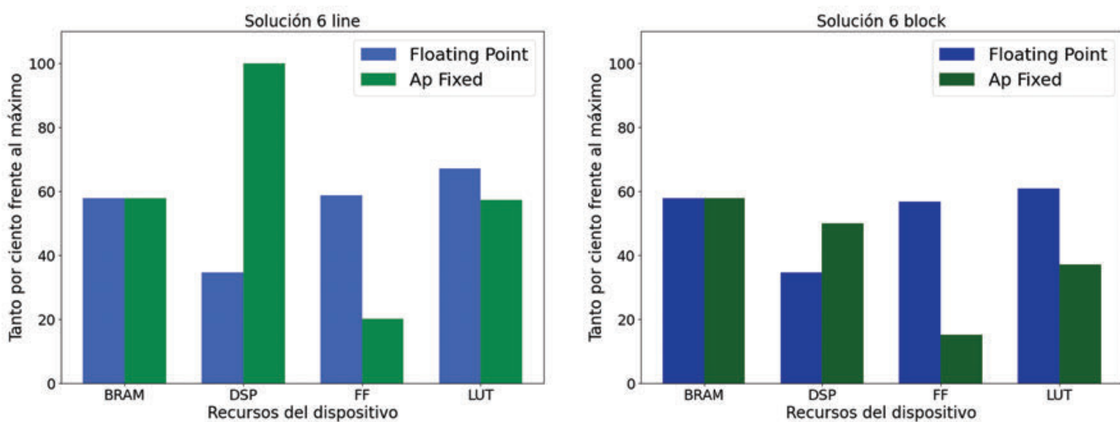
Número Identificador	BRAM (tot/ %)	DSP (tot/ %)	FF (tot/ %)	LUT (tot/ %)
5	22 / 1	36 / 1	899 / $\sim$ 0	2357 / 1
6	22 / 1	36 / 1	937 / $\sim$ 0	2270 / 1

Cuadro 5.14: Recursos utilizados por las capas *Line* (azul) y *Block* (verde) al variar el tipo de dato

En la figura 5.7 se muestra la variación que se ha producido en el consumo de recursos al variar el tipo de dato de punto flotante a punto fijo. Como se observa en las gráficas, el consumo de memorias es constante independientemente de la solución. El número de DSPs, sin embargo, aumenta cuando se usa punto fijo. Esto se debe a que al ancho de banda de los DSPs orientados a aritmética entera con los que cuenta la FPGA que se está usando tienen un ancho de banda de 27 bits [47]. Al usar un número fijo de 32 bits se necesitan el doble de DSPs, debido a que es necesario usar uno extra para manejar los 5 bits restantes.



(a) Comparación de recursos consumidos por la implementación de la solución 5 en ambas capas al variar el tipo de dato



(b) Comparación de recursos consumidos por la implementación de la solución 6 en ambas capas al variar el tipo de dato

Figura 5.7: Variación de los recursos consumidos por la implementación de las soluciones 5 y 6 al pasar de punto flotante a punto fijo.

Además de esto, la comparativa muestra como el consumo de *Flip-Flops* y LUTs es menor en las capas que usan punto fijo. En la solución 5, el número de *Flip-Flops* usados por la capa *line* en punto fijo es un 31,92 % de los que se usan en esa misma capa en punto flotante. Las LUTs usadas por la capa en punto fijo, por otro lado, suponen el 86,29 % de las que se usan en punto flotante.

En cuanto a la capa *block*, el consumo de *Flip-Flops* se reduce a un 25,93 % de la capa en punto flotante, mientras que el consumo de LUTs es un 64,85 % del consumo de la capa antes de cambiar el tipo de dato.

Se observa, por tanto, que el uso del punto fijo supone una reducción de los recursos ló-

gicos necesarios en ambas capas, siendo esta reducción mayor en la capa *block*, que además consume en líneas generales menos recursos que la capa *line*.

CAPÍTULO 6: Conclusiones y Líneas Futuras

En este capítulo se expondrán las conclusiones extraídas de la realización de este Trabajo de Fin de Máster, teniendo en cuenta los resultados obtenidos de la síntesis de las diferentes soluciones. Además de esto, se comentarán las posibles avances que se podrán realizar en un futuro sobre el trabajo realizado.

6.1 Conclusiones

En este trabajo se ha realizado el desarrollo de la estructura del bloque IP de una capa concreta de una red neuronal convolucional que irá implementada en una FPGA. Una vez desarrollada la capa, se han variado los parámetros de entrada de la misma, incluyendo el tamaño de la imagen de entrada, de los pesos y de los *strides*, y se ha observado cómo afecta la variación de estos parámetros a la latencia y consumo de recursos del bloque IP en el

dispositivo final. Además de esto, se ha realizado un estudio de cómo afecta el tipo de dato empleado, tanto a la precisión de los resultados, como a las prestaciones del diseño, en términos de latencia y consumo de recursos.

Para realizar este trabajo, se partió de una capa convolucional desarrollada en Python por uno de los grupos de investigación del IUMA, se estudió el funcionamiento de la capa y se transcribió todo el código de la misma al lenguaje C++, con el objetivo de contar con una capa en un lenguaje compatible con la herramienta de diseño *hardware* en alto nivel Vitis HLS. Una vez que se desarrolló la capa en C++, se comprobó que los resultados obtenidos coincidían con los resultados obtenidos al ejecutar la capa en Python. Una vez que el error entre ambas capas era cero, se comenzó a adaptar la capa para que se pudiera sintetizar en la herramienta de Vitis HLS. En última instancia, se desarrollaron dos capas con una estructura muy similar para comprobar los resultados de la síntesis de ambas. Una vez que la estructura estaba finalizada, con ayuda de un código de Python se generaron los archivos necesarios para generar distintas configuraciones de las capas en las que se variaron las dimensiones de los pesos, el tamaño de la imagen de entrada de la capa, aplicando *padding* o no a las imágenes; y el desplazamiento de los pesos sobre la imagen de entrada.

De los resultados obtenidos, en lo que refiere a latencia destacan: el menor tiempo de ejecución es de 3,877 ms, conseguido por la capa *line* al procesar una imagen de $128 \cdot 128 \cdot 3$ píxeles, con unos *strides* de $2 \cdot 2$, unos pesos de $3 \cdot 3 \cdot 3 \cdot 8$ y sin aplicar *padding* (solución 1); mientras que el mayor tiempo de ejecución es de 1 454,348 ms, obtenido también por la capa *line*, con imagen y pesos con dimensiones de $512 \cdot 512 \cdot 3$ y $7 \cdot 7 \cdot 3 \cdot 8$ respectivamente, sin aplicar *padding* y con un *stride* de $1 \cdot 1$ (solución 8).

En términos de recursos, el menor consumo de memoria es el de 13 BRAMs, resultado que se consiguió al sintetizar la capa *block* con los parámetros de entrada de la solución 1 (cuadro 5.1), mientras que el mayor consumo es de 38 BRAMs y se da en ambas capas cuando la imagen cuenta con $512 \cdot 512$ píxeles, sin *padding*, unos pesos de $7 \cdot 7 \cdot 3 \cdot 8$ y un desplazamiento de capa de $1 \cdot 1$.

Tanto en LUTs como en FFs, el mayor consumo se da en la capa *line*, que ocupa 6.099 y 6.159 de estos recursos, respectivamente, cuando sus parámetros de entrada son una imagen de $512 \cdot 512 \cdot 3$ sin *padding*, unos pesos de $7 \cdot 7 \cdot 3 \cdot 8$ y un *stride* de $4 \cdot 4$. El menor consumo

de *Flip-Flops* se da en la capa *block*, con la que se ocupan 899 cuando se realiza la síntesis con los datos en punto fijo con la imagen de $512 \cdot 512$, sin *padding*, los pesos de $3 \cdot 3$ y un *stride* de $2 \cdot 2$. Por otro lado, el menor consumo de LUTs, 2.270, se consigue con esta misma capa, la misma configuración, pero aplicando *padding* a la imagen de entrada.

A la vista de los resultados obtenidos de la síntesis de las distintas soluciones, se ha observado que todos los parámetros de las capas influyen en el tiempo de ejecución, aumentando la latencia cuando aumenta el tamaño de la imagen y las dimensiones de los pesos, pero disminuyendo esta cuando se aumenta el tamaño de los *strides*. En lo referente al consumo de recursos de la capa, de los resultados obtenidos se llega a la conclusión de que el parámetro que más afecta al consumo de recursos son las dimensiones de los pesos, aunque todos los recursos afectan al consumo de memoria en mayor o menor proporción.

Entre las dos capas, la capa *line* cuenta con un tiempo de ejecución menor en la mayoría de los casos, por lo que es la elección adecuada si se busca una mayor velocidad de respuesta al obtener resultados del bloque IP. Sin embargo, la capa *block* consume en todo momento menos recursos que la capa *line*, por lo que podría resultar más adecuada si esta capa se debe implementar junto con las demás capas de la red en el mismo dispositivo.

Por último, a la vista de los resultados obtenidos, a la hora de implementar la capa en una FPGA es recomendable cambiar el tipo de dato usado a variables en punto fijo, ya que no tiene impacto significativo en los resultados de la capa pero mejora en gran medida la latencia y el consumo de recursos del bloque IP.

6.2 Líneas Futuras

El siguiente paso sobre el trabajo desarrollado sería, usando las herramientas disponibles en Vitis HLS y en los entornos de diseño *hardware* de Vivado, realizar la co-simulación e implementación de las soluciones en la FPGA para comprobar, además de su correcto funcionamiento en el dispositivo final, que los resultados obtenidos de la síntesis se corresponden con la realidad o si estos cambian. Además, una vez implementado en el dispositivo final, sería conveniente comprobar que los resultados obtenidos de la capa son compatibles con la red que está diseñando el equipo del IUMA, o si las restricciones impuestas en

términos de latencia y/o de consumo de recursos cumplen con las especificaciones del proyecto.

A partir de las soluciones obtenidas, se abre también la posibilidad de intentar desarrollar una arquitectura híbrida entre la capa *line* y la capa *block*, para intentar conseguir un punto intermedio entre los buenos resultados de latencia de la primera y el menor consumo de recursos de la segunda.

Además de esto, para comprobar la influencia que tienen estas en el *throughput* del sistema, se podrían probar distintas interfaces de entrada/salida en las capas ya generadas (basadas en *streaming* o en memorias) y comprobar si éstas incrementan las prestaciones del sistema, lo que supondría que en la arquitectura actual las interfaces tipo FIFO implican un cuello de botella.

Por último, aprovechando los códigos generados durante este trabajo, se podría repetir el flujo de diseño con otro tipo de capas de la red neuronal y así generar una configuración que no solo tenga en cuenta los resultados de predicción de la propia red, sino que además tenga en cuenta el impacto de dicha configuración en el dispositivo final en el que se va a implementar.

Bibliografía

[1] H. Amano, *Principles and Structures of FPGAs*. Springer, 2018.

[2] I. Kuon y J. Rose, «Measuring the Gap Between FPGAs and ASICs», *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, n.º 2, págs. 203-215, 2007. doi: 10.1109/TCAD.2006.884574.

[3] Z. Gao, L. Yan, J. Zhu, R. Han, U. Anees y R. Pedro, «Radiation tolerant viterbi decoders for on-board processing (OBP) in satellite communications», *China Communications*, vol. 17, n.º 1, págs. 140-150, 2020. doi: 10.23919/JCC.2020.01.011.

[4] C. C. Wu, L. Qiao y Q. Chen, «Design of a 640-Gbps Two-Stage Switch Fabric for Satellite On-Board Switches», *IEEE Access*, vol. 8, págs. 68 725-68 735, 2020. doi: 10.1109/ACCESS.2020.2986300.

[5] S. Yang, S. Cao, J. Wei, Y. Zhao, H. Han y L. Yan, «Space-Based Computing Platform Based on SoC FPGA», en *2019 IEEE World Congress on Services (SERVICES)*, vol. 2642-939X, 2019, págs. 172-177. doi: 10.1109/SERVICES.2019.00049.

[6] Xilinx, *RT Kintex UltraScale FPGA For Ultra High Throughput and High Bandwidth Applications*, accedido el 04-08-2021, 2021. dirección: https://www.xilinx.com/products/silicon-devices/fpga/rt-kintex-ultrascale.html#documentation.

[7] R. A. Schowengerdt, *Remote sensing: models and methods for image processing*. Elsevier, 2006.

[8] A. Tatem, S. Goetz y S. Hay, «Fifty Years of Earth Observation Satellites», *American Scientist*, vol. 96, págs. 390-398, sep. de 2008. doi: 10.1511/2008.74.390.

[9] A. Xu, J. Wu, G. Zhang, S. Pan, T. Wang, Y. Jang y X. Shen, «Motion Detection in Satellite Video», *Journal of Remote Sensing & GIS*, vol. 06, ene. de 2017. doi: 10.4172/2469-4134.1000194.

[10] Bieliński, «A Parallax Shift Effect Correction Based on Cloud Height for Geostationary Satellites and Radar Observations», *Remote Sensing*, vol. 12, pág. 365, ene. de 2020. doi: 10.3390/rs12030365.

[11] L. Meng y J. P. Kerekes, «Object tracking using high resolution satellite imagery», *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 5, n.º 1, págs. 146-152, 2012.

[12] E. P. Directory, *WorldView-2*, accedido el 13-08-2021, 2021. dirección: https://earth.esa.int/web/eoportal/satellite-missions/v-w-x-y-z/worldview-2.

[13] C. C. Aggarwal y col., «Neural networks and deep learning», *Springer*, vol. 10, págs. 978-3, 2018.

[14] Y. Li, «Research on Application of Convolutional Neural Network in Intrusion Detection», en *2020 7th International Forum on Electrical Engineering and Automation (IFEEA)*, 2020, págs. 720-723. doi: 10.1109/IFEEA51475.2020.00153.

[15] A. B. Abdul Qayyum, A. Arefeen y C. Shahnaz, «Convolutional Neural Network (CNN) Based Speech-Emotion Recognition», en *2019 IEEE International Conference on Signal Processing, Information, Communication Systems (SPICSCON)*, 2019, págs. 122-125. doi: 10.1109/SPICSCON48833.2019.9065172.

[16] Y. Zhou y J. Cui, «Research and Improvement of Encrypted Traffic Classification Based on Convolutional Neural Network», en *2020 IEEE 8th International Conference on Computer Science and Network Technology (ICCSNT)*, 2020, págs. 150-154. doi: 10.1109/ICCSNT50940.2020.9305018.

[17] A. Verma, P. Singh y J. S. Rani Alex, «Modified Convolutional Neural Network Architecture Analysis for Facial Emotion Recognition», en *2019 International Conference on Systems, Signals and Image Processing (IWSSIP)*, 2019, págs. 169-173. doi: 10.1109/IWSSIP.2019.8787215.

[18] K. Pai y A. Giridharan, «Convolutional Neural Networks for classifying skin lesions», en *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON)*, 2019, págs. 1794-1796. doi: 10.1109/TENCON.2019.8929461.

[19] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto y H. Adam, «Mobilenets: Efficient convolutional neural networks for mobile vision applications», *arXiv preprint arXiv:1704.04861*, 2017.

[20] C. EU, *Video Imaging Demonstrator for Earth Observation*, accedido el 03-03-2021, 2019. dirección: https://cordis.europa.eu/project/id/870485.

[21] Y. Song, H. Zhang y L. Zhang, «Remote Sensing Image Spatio-Temporal Fusion via a Generative Adversarial Network Through One Prior Image Pair», en *IGARSS 2020 - 2020 IEEE International Geoscience and Remote Sensing Symposium*, 2020, págs. 7009-7012. doi: 10.1109/IGARSS39084.2020.9324101.

[22] X. Zhang, R. Liu, F. Gan, W. Wang, L. Ding y B. Yan, «Evaluation of Spatial-Temporal Variation of Vegetation Restoration in Dexing Copper Mine Area Using Remote Sensing Data», en *IGARSS 2020 - 2020 IEEE International Geoscience and Remote Sensing Symposium*, 2020, págs. 2013-2016. doi: 10.1109/IGARSS39084.2020.9323698.

[23] X. Wang, W. Jiang, J. Li, J. Wu, Y. Chen, A. Gong, H. Tang y J. Yue, «Using Remote Sensing to Monitor the Water Change of Xiong'an New Area», en *IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium*, 2019, págs. 4415-4418. doi: 10.1109/IGARSS.2019.8897994.

[24] N. R. Govind, C. A. Rishikeshan y H. Ramesh, «Comparison of Different Pan Sharpening Techniques using Landsat 8 Imagery», en *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, 2019, págs. 1-4. doi: 10.1109/I2CT45611.2019.9033659.

[25] M. Kim, H. Lim, S. Yu y J. Paik, «Pan-sharpening of Multispectral Remote Sensing Imagery Using Deep Back-Projection Network», en *2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, 2020, págs. 1-2. doi: 10.1109/ICCE-Asia49877.2020.9277369.

[26] J. Guo, H. Ren, Y. Zheng, J. Nie, S. Chen, Y. Sun y Q. Qin, «Identify Urban Area From Remote Sensing Image Using Deep Learning Method», en *IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium*, 2019, págs. 7407-7410. doi: 10.1109/IGARSS.2019.8898874.

[27] Q. Zhu, X. Sun, Y. Zhong y L. Zhang, «High-Resolution Remote Sensing Image Scene Understanding: A Review», en *IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium*, 2019, págs. 3061-3064. doi: 10.1109/IGARSS.2019.8899293.

[28] C. Yang, H. Li, X. Huang, X. Li, Y. Liu, W. Hong e Y. Zou, «Research on Extraction and Evaluation of Ecological Corridor Based on Remote Sensing and GIS», en *IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium*, 2019, págs. 3464-3467. doi: 10.1109/IGARSS.2019.8899839.

[29] H. V. Phu, T. Minh Tan, P. Van Men, N. Van Hieu y T. Van Cuong, «Design and Implementation of Configurable Convolutional Neural Network on FPGA», en *2019 6th NAFOSTED Conference on Information and Computer Science (NICS)*, 2019, págs. 298-302. doi: 10.1109/NICS48868.2019.9023810.

[30] H. Kim y K. Choi, «Low Power FPGA-SoC Design Techniques for CNN-based Object Detection Accelerator», en *2019 IEEE 10th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, 2019, págs. 1130-1134. doi: 10.1109/UEMCON47517.2019.8992929.

[31] L. Kosmidis, I. Rodriguez, A. Jover-Alvarez, S. Alcaide, J. Lachaize, O. Notebaert, A. Certain y D. Steenari, «GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward», en *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, págs. 1314-1319. doi: 10.23919/DATE51398.2021.9474123.

[32] C. Shu, W. Pang, H. Liu y S. Lu, «High Energy Efficiency FPGA-Based Accelerator for Convolutional Neural Networks Using Weight Combination», en *2019 IEEE 4th International Conference on Signal and Image Processing (ICSIP)*, 2019, págs. 578-582. doi: 10.1109/SIPROCESS.2019.8868502.

[33] A. Kubra, K. R.K., P. Das, Y. Prasad y W. R. Gautam, «Design of A Reconfigurable Digital Modulator For High Bit Rate Data Transmission», en *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2020, págs. 1-6. doi: 10.1109/CONECCT50063.2020.9198620.

[34] C. M. Fuchs, N. M. Murillo, A. Plaat, E. van der Kouwe y P. Wang, «Towards Affordable Fault-Tolerant Nanosatellite Computing with Commodity Hardware», en *2018 IEEE 27th Asian Test Symposium (ATS)*, 2018, págs. 127-132. doi: 10.1109/ATS.2018.00034.

[35] D. T. Kwadjo y C. Bobda, «Late Breaking Results: Automated Hardware Generation of CNN Models on FPGAs», en *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, págs. 1-2. doi: 10.1109/DAC18072.2020.9218692.

[36] M. Hall y V. Betz, «From TensorFlow Graphs to LUTs and Wires: Automated Sparse and Physically Aware CNN Hardware Generation», en *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, págs. 56-65. doi: 10.1109/ICFPT51103.2020.00017.

[37] Y.-C. Ling, H.-H. Chin, H.-I. Wu y R.-S. Tsay, «Designing A Compact Convolutional Neural Network Processor on Embedded FPGAs», en *2020 IEEE Global Conference on Artificial Intelligence and Internet of Things (GCAIoT)*, 2020, págs. 1-7. doi: 10.1109/GCAIoT51063.2020.9345903.

[38] S. Qiao y J. Ma, «FPGA Implementation of Face Recognition System Based on Convolution Neural Network», en *2018 Chinese Automation Congress (CAC)*, 2018, págs. 2430-2434. doi: 10.1109/CAC.2018.8623662.

[39] X. Zhen y B. He, «Research on FPGA High-Performance Implementation Method of CNN», en *2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP)*, 2021, págs. 1177-1181. doi: 10.1109/ICSP51882.2021.9408954.

[40] E. Bendersky, *Depthwise separable convolutions for machine learning*, accedido el 17-08-2021, 2018. dirección: https://eli.thegreenplace.net/2018/depthwise-separable-convolutions-for-machine-learning/.

[41] A. Kalinovsky, V. Liauchuk y A. Tarasau, «Lesion detection in CT images using Deep Learning semantic segmentation tachnique», mayo de 2017.

[42] H. Thadeshwar, V. Shah, M. Jain, R. Chaudhari y V. Badgujar, «Artificial Intelligence based Self-Driving Car», en *2020 4th International Conference on Computer, Communication and Signal Processing (ICCCSP)*, 2020, págs. 1-5. doi: 10.1109/ICCCSP49186.2020.9315223.

[43] XILINX, *Xilinx Kintex UltraScale FPGA KCU105 Evaluation Kit*, accedido el 13-08-2021, 2021. dirección: https://www.xilinx.com/products/boards-and-kits/kcu105.html#overview.

[44] ——, *Kintex UltraScale*, accedido el 13-08-2021, 2021. dirección: https://www.xilinx.com/products/silicon-devices/fpga/kintex-ultrascale.html#documentation.

[45] J. Brownlee, *Deep Learning With Python: Develop Deep Learning Models on Theano and TensorFlow Using Keras*. Machine Learning Mastery, 2016. dirección: https://books.google.es/books?id=K-ipDwAAQBAJ.

[46] XILINX, *Vitis High-Level Synthesis User Guide*, accedido el 04-08-2021, 2020. dirección: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf.

[47] ——, *UltraScale Architecture DSP Slice User Guide*, accedido el 06-09-2021, 2021. dirección: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.