UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Implementación de redes neuronales convolucionales en Field-Programmable Logic Arrays

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Ana María Ibáñez Álvarez

Tutor: David de Andrés Martínez

Cotutor: Juan Carlos Ruiz García

Curso académico: 2020 / 2021

Resumen

La red neuronal convolucional (CNN), es un algoritmo de aprendizaje automático que ha sido probado como un algoritmo altamente preciso y eficaz, que se ha utilizado en una gran variedad de aplicaciones tales como reconocimiento visual y clasificación de imágenes, entre otros.

Las CNN de última generación tienen un elevado coste computacional, pero su naturaleza modular y su paralelismo hace que sean adecuadas para poder ser aceleradas en plataformas como las matrices de puerta programable de campo (FPGA). Por lo general, las redes neuronales convolucionales, requieren un desarrollo muy largo y complejo para poder ser implementadas o aceleradas usando FPGA, por lo tanto, en este trabajo se va a plantear usa solución para hacer esto de una forma sencilla y sin necesidad de que se tenga un conocimiento previo, ni experiencia en lenguajes de descripción de hardware (HDL). Para conseguirlo, este trabajo propone el uso de una herramienta de generación automática de VHDL; pues este tipo de herramientas, a través de código en C / C++, puede traducir automáticamente cualquier algoritmo descrito con una serie de restricciones a VHDL, inclusive la arquitectura de una red convolucional. Además, también se verán técnicas para mejorar su implementación como optimizadores y cuantificadores. Y para conseguir un mejor desarrollo, se generará un código que sea altamente optimizado, paralelo, reconfigurable, escalable y adaptable a diferentes modelos de una CNN, aunque el trabajo se centrará únicamente en un modelo.

Aparte de la generación automática de código en VHDL, el trabajo también ofrecerá una pequeña introducción a las redes convolucionales, exponiendo cómo se diseñan, cómo están compuestas y cómo funcionan. Además, que mostrarán otras alternativas a las herramientas de generación automática de lenguajes de descripción de hardware, para poder acelerar una red convolucional en un acelerador hardware.

Palabras clave: redes convolucionales, FPGA, paralelismo, capa de la red.

Abstract

The convolutional neural network (CNN) is a machine learning algorithm that has been proven to be a highly accurate and efficient algorithm that has been used in a wide variety of applications like visual recognition or image classification, among others. State-of-the-art CNNs are highly computational, but their modular nature and parallelism make them suitable for acceleration on platforms like field programmable gate arrays (FPGAs). In general, convolutional neural networks require a very long and complex development to be implemented or accelerated using FPGA, therefore, in this project we will propose a solution to do this in a simple way and without the need for prior knowledge or experience in hardware description languages (HDL) in order to accelerate the algorithm in the FPGA. To achieve this, this project proposes the use of an automatic VHDL generation tool. This type of tool, through C / C++ code, can automatically translate any described algorithm into VHDL, including the architecture of a convolutional network. In addition, techniques to improve implementation such as optimisers and quantifiers will also be discussed. To achieve better development we will generate code that is highly optimised, parallel, reconfigurable, scalable and adaptable to different models of a CNN, although the work will focus only on one model.

Apart from automatic code generation in VHDL, the paper will also give a short introduction to convolutional networks, showing how they are designed and implemented. It will also show how they are composed and how they work. In addition, it will show other alternatives to automatic hardware description language generation tools, in order to accelerate a convolutional network in a hardware accelerator.

Tabla de contenidos

Lista de figuras

Lista de tablas

1. Introducción

Las redes neuronales artificiales son uno de los métodos actuales de Aprendizaje Automático o *Machine Learning* [1] que se inspiran en neuronas biológicas reales. Estas neuronas artificiales, al conectarse son capaces de crear las llamadas redes convolucionales (CNN) [2] [3], que son capaces de aprender y realizar una tarea de clasificación o predicción.

Esas redes convolucionales se construyen sobre un conjunto de capas conectadas que, a su vez, están compuestas por unidades que realizan funciones similares a las de las neuronas reales. Por lo tanto, la idea general de su estructura es conseguir que sea semejante a la corteza visual real.

De hecho, las redes convolucionales han ganado gran popularidad no sólo en la clasificación de imágenes y videos, sino también en muchas otras aplicaciones como el reconocimiento de voz, reconocimiento visual de objetos o sectores como la automoción.

La idea de estas redes ha existido desde los años 40 [4] [5], puesto que la unidad de cálculo que intenta modelar el comportamiento de una neurona biológica existe desde entonces. Pero no ha sido hasta las últimas generaciones de las plataformas informáticas de alto rendimiento que se ha podido llevar a cabo esta idea y se ha permitido su evolución.

Desde el principio, las redes neuronales han sabido enfrentarse a algoritmos más tradicionales y superarlos pues consiguen los mismos resultados a la vez que mejoran el tiempo de recepción y la fiabilidad de estos, gracias a su versatilidad y la cantidad de información compleja que pueden aprender y manejar. Sin embargo, la complejidad del aprendizaje automático tiene sus ventajas e inconvenientes. Debido a esto, muchos profesionales aún se muestran escépticos sobre el uso de métodos de aprendizaje profundo, considerándolos como cajas negras debido a su complejidad y la falta de interpretabilidad directa de sus modelos [6].

No obstante, se está haciendo un gran esfuerzo por recuperar la confianza en esos métodos. Por lo que los profesionales de las redes convolucionales han estado tratando de comprender y descubrir los conocimientos del proceso de toma de decisiones que estas aplican, para producir una respuesta o predicción dada una entrada. Así, se ha descubierto, que gracias al rendimiento excepcional de las redes convolucionales y la gran gama de aplicaciones a las que se pueden adaptar, compensa el enorme costo computacional que requieren, puesto que un gran modelo de CNN requiere más de mil millones de operaciones por imagen.

Con la disponibilidad de plataformas potentes como unidades de procesamiento gráfico (GPU) [7] [8], se puede alcanzar este nivel de rendimiento, pero debido al alto consumo de energía en las GPU no es factible integrar estas soluciones en pequeños sistemas portátiles como vehículos inteligentes, drones o dispositivos de IoT [9]. Por lo que, se han considerado diferentes plataformas para la implementación eficiente de las redes convolucionales y las FPGA (Field-programmable gate array) se han presentado como una de las soluciones más prometedoras.

Curiosamente, las FPGA parecen encajar perfectamente con el trabajo, puesto que son reconfigurables, es decir, permiten adaptar su implementación a nuevos algoritmos o modelos y aprovechan el paralelismo inherente en las redes convolucionales y además aportan un ahorro energético.

Por lo que se sabe de las diversas implementaciones que existen de los diferentes tipos de redes, todas coinciden en que se basan en su acceso frecuente a datos y su complejidad de cómputo, por lo que además requiere una implantación eficiente.

1.1 Motivación

Actualmente vivimos en la era de la comunicación y la conectividad, gracias a la cantidad de avances en tecnología que se han realizado en los últimos años. Uno de los avances que más futuro tiene y que nos abre el mundo a un millar de posibilidades es el *Deep-Learning* [10] [11]. Este tipo de tecnología está siendo utilizada en una gran multitud de campos, debido a la gran cantidad de aplicaciones en las que puede ser utilizada como el reconocimiento de imágenes o habla, y la robótica entre otras [12].

Pero esta tecnología conforme se utiliza y se investiga se ha visto que está limitada especialmente por su alto coste computacional y energético en su procesamiento. Y esto supone una gran problemática en muchas de las aplicaciones para las que tenían un prometedor futuro como el sector de la automoción. Por ello, se están estudiando diversas técnicas de cómo mejorar estas limitaciones para así poder sacar todo el rendimiento a esta tecnología.

Después de estudiar cómo se podría mejorar esta tecnología se ha observado que la tendencia mayoritaria es la implantación de la red neuronal convolucional (CNN) en aceleradores hardware, ya que, las CNN se pasan la mayor parte de su tiempo realizando operaciones con matrices. Como los aceleradores teóricamente parecen ideales para realizar esta tarea sería interesante ver si es posible implementar una CNN en un acelerador hardware puesto que además mejoraríamos el coste computacional optimizando la energía.

Por ello este trabajo se ha centrado en realizar un estudio de técnicas que permita mejorar el rendimiento y la eficiencia energética de las redes convolucionales, viendo si esto es posible realizarlo sin sacrificar precisión o incrementar su coste. Además, como esta es la tendencia actual, para realizarlo existe una pequeña gama de entornos y herramientas que pueden ayudar a realizar este estudio.

1.2 Objetivos

El principal objetivo en este trabajo es comprobar si es posible implantar una red convolucional en un acelerador hardware y validar su implementación para comprobar si se obtiene la misma precisión y se mejora el coste energético y de rendimiento. Para ello, definiremos una serie de objetivos específicos:

- Mantener la precisión de clasificación: primero a través de una red entrenada mediante un framework específico se comprobará si es posible adaptar dicha CNN a un acelerador sin perder precisión y fiabilidad.
- Mejora del rendimiento: una vez se tenga una red convolucional creada, se procederá a ejecutarla sobre diferentes hardware para así comprobar las diferencias a la hora de obtener una respuesta.

- Comprobar si los entornos profesionales para implementar redes neuronales en aceleradores permiten que el desarrollador no sea un experto en la materia.

1.3 Estructura

La investigación presentada en este trabajo se centra en la implementación de la aceleración de aprendizaje profundo en FPGA. Las partes restantes del siguiente trabajo de fin de grado están organizadas de la siguiente manera:

- Capítulo 2: Estado del arte.
  Se puede encontrar en este apartado cómo está la situación actual de la tecnología que el estudio pretende analizar, y qué es lo que aporta este trabajo con respecto a otros sobre la misma temática.

- Capítulo 3: Introducción a las matrices de puerta programable en campo.
  Este apartado se centra en exponer que son las FPGA o matrices de puerta programable en campo además de hacer una comparativa con diferentes tipos de hardware para realizar la misma tarea.

- Capítulo 4: Creación de una red neuronal.
  El capítulo se centrará en explicar las redes neuronales y cómo están formadas. Además, se verá la estructura de la red que se va a usar para el estudio.

- Capítulo 5: Implementación de una red convolucional
  En este apartado se verá cómo está construida una red neuronal, explicando a su vez como funciona. Además, se verá cómo ha sido entrenada y cómo se ha implementado en el hardware seleccionado.

- Capítulo 6: Implantación de la red en diferentes equipos.
  En esta sección se analizará los tiempos obtenidos de cada etapa de la red implantado en diferentes hardware, y así comprobar si es rentable implantar toda la red en un acelerador o si por el contario es mejor implantar sólo una sección de la red.

- Capítulo 7: Conclusiones y trabajo futuro.
  Este último apartado se expondrán las conclusiones del trabajo, comentando los objetivos alcanzados y el trabajo futuro.

2. Estado del arte

En los últimos años, se han introducido técnicas que permiten un procesamiento eficiente de las redes convolucionales (CNN) para mejorar su eficiencia energética y rendimiento. Esto se debe a que las redes convolucionales tienen un alto coste computacional y por lo tanto no pueden utilizarse en algunas de las aplicaciones para las que serían aptas. Por ello, se debía mejorar su desarrollo, especialmente si se trata de las *"at the Edge"*, es decir, donde se obtienen los datos. Estos dispositivos son sobre todo sensores, pequeños controladores o microprocesadores, por lo que no disponen de gran capacidad de cómputo y al tratarse muchas veces de dispositivos con batería el consumo se convierte en un factor importante.

A raíz de la problemática, diversos fabricantes y desarrolladores han enfocado soluciones desde diferentes perspectivas. Pero la que parece el mejor enfoque, es la de llevar gran parte de su cálculo a los aceleradores hardware. Esto se debe a que las redes convolucionales pasan la mayor parte de su tiempo multiplicando gran cantidad de matrices, lo que hace que ocupe mucho tiempo de CPU. Sin embargo, el tipo de cálculo es muy sencillo, por lo que parece lógico pensar que, si ese cálculo es llevado a aceleradores, tanto el tiempo, como el rendimiento de la red mejorarán sin contar el consumo energético. Puesto que un acelerador de propósito general está diseñado para soportar este tipo de cálculos.

Por ello, los principales proveedores, han desarrollado entornos de trabajo aptos para trabajar con los principales frameworks para crear y entrenar redes convolucionales como Tensorflow [13], Pytorch [14] o Caffe [15] entre otros.

Se han realizado un gran número de estudios relacionado con estos entornos [16] [17]. Y se puede llegar a la conclusión de que no hay un claro consenso, puesto que cada entorno prima algunas cosas y sacrifica otras. Por ejemplo, Haddoc2 [18] es una herramienta para diseñar automáticamente aceleradores hardware basados en FPGA para redes neuronales convolucionales, pero únicamente es compatible con el framework Caffe, pues es con este framework conforme está diseñado para generar la descripción hardware de la red. Además, esta herramienta se basa en los principios del procesamiento de datos basado en el flujo de los datos e implementa la red convolucional utilizando un enfoque de mapeo directo de hardware lo cual no hacen otras herramientas. Otra herramienta para diseñar redes convolucionales en hardware es Vitis-AI [19] de Xilinx, la cual proporciona un entorno fácil para el desarrollo además de que es compatible con los principales frameworks de entrenamiento, pero también tiene sus limitaciones, puesto que solamente es compatible con algunos de los aceleradores hardware creados por Xilinx y no admite a otros tipos de fabricante. Otras herramientas que proporciona Xilinx es Vitis HLS, herramienta donde un desarrollador escribe un código en C/C++ y el entorno lo traduce a VHDL o Verilog para poder implementar el funcionamiento del algoritmo desarrollado en una FPGA de propósito general. Pero también hay otros fabricantes que están interesados en adaptar las redes convolucionales a su propio hardware como es Intel, que a través de la herramienta Intel HLS [20] permite coger código en C++ y generar código de nivel de transferencia de registro (RTL). Además, esta herramienta acelera el tiempo de verificación sobre RTL elevando el nivel de abstracción para el diseño de hardware FPGA, por lo que es similar a la mencionada anteriormente como Vitis HLS. Incluso este mismo fabricante también ha desarrollado entornos exclusivos para poder implementar las redes neuronales convolucionales como es OpenVINO [21], por lo que se

puede ver como la tendencia es intentar conseguir que las redes neuronales convolucionales sean aceleradas en hardware y es una moda en la que ningún fabricante quiere quedarse atrás.

Todas estas herramientas tienen el mismo objetivo, facilitar la implementación automática de la red neuronal en FPGA, ya que su descripción en un lenguaje de descripción de hardware es muy compleja. De esta forma se pueden evitar errores y facilitar la síntesis, y al implementar en una FPGA se espera de esta forma mejorar el rendimiento y el consumo. Pero tienen problemas, puesto que, en muchas de ellas a la hora de generar el código, la red puede perder precisión y eficiencia. Puesto que si se desea ahorrar energía es muy probable que se pierda eficiencia y si se quiere conservar esta muchas veces el coste se vuelve muy superior. Por lo que se podría deducir que la mejor solución sería diseñar un acelerador conforme a la red que se desee implementar, y su propósito específico. Pero esto aumentaría mucho el coste. Este es el motivo por el que nos centraremos en la mejora que proporcionan las herramientas y en cómo realizando algunos ajustes puede mejorar una red y ser de propósito general.

3. Introducción a las matrices de puerta programable en campo

En este capítulo se va a ofrecer una pequeña introducción sobre las matrices de puerta programable en campo (FPGA, del inglés Field-Programmable Gate Array), destacando sus características, fortalezas y debilidades en comparación con otras plataformas hardware. En concreto, este capítulo se centrará en las unidades de procesamiento de propósito general (CPU), las unidades de procesamiento gráfico (GPU) y los circuitos integrados de aplicación específica (ASIC).

3.1 Posibles plataformas de hardware para el diseño de aceleradores

El diseño de un acelerador hardware es un proceso que está sujeto a los requisitos de la aplicación destino, y la plataforma de implementación final. Normalmente, los sistemas integrados tienen un conjunto de requisitos y están sujetos a restricciones particulares como el tiempo, la potencia, o el tamaño físico. Esas restricciones, requieren que se realicen optimizaciones importantes en los algoritmos antes de aplicarlos al hardware. Para cumplir con los requisitos de diseño bajo las restricciones que tienen, el algoritmo de destino debe investigarse muy bien para identificar la plataforma idónea para el proceso de aceleración.

- **Unidades Centrales de Procesamiento (CPU)**

  Las CPU son los elementos de procesamiento más comunes que se encuentran en los dispositivos electrónicos, como ordenadores personales, teléfonos inteligentes, tabletas, consolas e incluso algunos automóviles. La mayoría de estas CPU son de propósito general, lo que significa que están diseñadas para realizar cualquier tarea, donde se pueden programar de forma sencilla y flexible utilizando software. Además, estas CPU ofrecen un rendimiento aceptable en una amplia gama de cargas de trabajo de computación. Pero dado que estos dispositivos informáticos están basados en secuencias, no son buenos a la hora de ejecutar problemas que dependen de un alto paralelismo, como es la identificación de imágenes, que es lo que realiza este trabajo.

- **Unidades de Procesamiento Gráfico (GPU)**

  Las GPU se dedican a realizar la carga de trabajo relacionadas con el procesamiento gráfico, como su nombre indica. Recientemente, se investigaron las GPU como una plataforma de aceleración para problemas de aprendizaje automático y para otras tareas. Una GPU de gama alta, como NVIDIA TITAN XP, contiene 3840 núcleos de procesamiento de punto flotante que puede funcionar a una velocidad de 1.582 MHz, ofreciendo aproximadamente 547,7 GB/s de ancho de banda de memoria con una velocidad de memoria de 11,4 Gbps, pero todo esto tiene un coste, pues tiene un alto consumo energético, ya que puede alcanzar hasta los 250W. Un procesador con tan alto consumo de energía no es adecuado para dispositivos integrados. Además, con las GPU,

se pueden ejecutar tareas en paralelo independientes; por lo que es posible adaptar algoritmos de aprendizaje profundos, pues el cálculo se realiza en paralelo y se garantiza la independencia de los datos. Pero, aun así, dado el alto consumo energético las GPU, no son plataformas óptimas para llevar a cabo la red convolucional que el trabajo está desarrollando.

- **Circuitos integrados de aplicación específica (ASIC)**

  A la hora de cumplir con los requisitos del sistema, los ASIC son la solución ideal, pues se puede lograr el mayor rendimiento y eficiencia energética. Sin embargo, los ASIC son poco adecuados para realizar los cálculos irregulares y algoritmos dinámicos que evolucionan con el tiempo, ya que no proporcionan ninguna reconfiguración una vez ya se ha fabricado. El algoritmo de CNN es un algoritmo en evolución, y no existe un modelo fijo que se considere modelo representativo. Implementando un completo CNN en un ASIC seguramente seria altamente eficiente (bajo consumo energético y altas prestaciones) pero no sería adaptable pues cualquier cambio en la red haría que fuese necesario volver a diseñar y fabricar un nuevo circuito. Las capas convolucionales, son muy costosas computacionalmente; y acelerar algunos módulos fijos que utilicen tecnología ASIC puede ser eficiente. Pero como este no es un objetivo del trabajo, esta plataforma no es la adecuada.

- **Matrices de puerta programables en campo (FPGA)**

  Si bien es mejor adaptar los algoritmos de naturaleza paralela en GPU, la arquitectura y estructura de las FPGA está diseñada para aplicaciones donde los motores de procesamiento personalizados se pueden construir utilizando bloques lógicos programables para satisfacer las necesidades del algoritmo. En otras palabras, hay menos énfasis sobre la adaptación de algoritmos cuando se trata de desarrollar técnicas de aprendizaje automático para FPGA. Esto permite más libertad para explotar optimizadores a nivel de algoritmo. El rendimiento de la FPGA se puede incrementar con el diseño, y más utilizando formatos de datos de precisión de punto fijo.
  Los optimizadores y técnicas que requieren muchas operaciones complejas se pueden implementar fácilmente en lenguajes de software de alto nivel, pero lo más atractivo de estos optimizadores, como Vitis HLS, es que es posible implementar esas operaciones en aceleradores hardware de forma sencilla.
  Además, la adaptabilidad de la implementación de las FPGA hace posible que una amplia gama de modelos de redes neuronales convolucionales pueda implementarse en el mismo chip. Por lo que, hasta ahora las FPGA es la plataforma más adecuada para el algoritmo que se ha desarrollado en el trabajo, pero la desventaja de la implementación basada en FPGA es que los diseñadores deben usar lenguajes de descripción de hardware para realizar su implementación.

3.2 Matrices de puerta programable en campo

Las FPGA o matrices de puerta programable en campo son dispositivos semiconductores a base de matrices 2D de bloques lógicos configurables o CLB, que se conectan a través de lógica programable [22]. Las interconexiones se asemejan a una red de cables que se disponen horizontal y verticalmente entre bloques lógico, todo esto se puede apreciar en la Figura 3-1 la cual muestra la arquitectura básica de una FPGA y se aprecia como se disponen tanto las interconexiones como los bloques lógicos y de entrada/salida.

Los elementos de una FPGA están controlados por bits de memoria (normalmente SRAM) que permite modificar su funcionamiento e interconexión. En general, estos bits de memoria controlan transistores de paso que permiten conectar las líneas de comunicación, multiplexores para modificar la interconexión de elementos lógicos, y diversos parámetros que permiten modificar las funciones lógicas implementadas, el contenido de las memorias disponibles en el dispositivo, el funcionamiento de sus elementos secuenciales, etc. Todo esto permite programar un flujo de bits de configuración en el dispositivo para implementar cualquier diseño digital. Esta tecnología permite reprogramar múltiples veces la FPGA proporcionándole multitud de funciones desde reproducir el funcionamiento de una puerta lógica o un sistema combinacional hasta complejos sistemas en un chip.

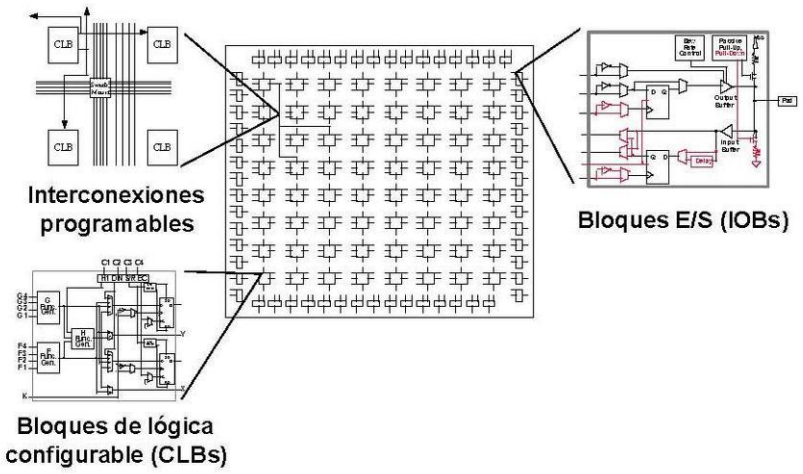


Figura 3-1 Arquitectura interna de una FPGA *[23]*

La primera FPGA comercial de la era moderna fue introducida por Xilinx en 1984 [24], la cual, contenía matrices de bloques lógicos configurables y entradas/salidas. Pero las actuales generaciones de FPGA cuentan con cientos de miles de bloques lógicos configurables e incorporan una gran cantidad de unidades funcionales reforzadas que permiten implementar rápida y eficientemente las funciones más comunes. Es por esto por lo que, dada la gran experiencia de Xilinx en este campo, se ha decidido, a lo largo del trabajo, ceñirnos a este fabricante.

3.2.1 Lógica programable

Los bloques lógicos programables [22] [25] de la FPGA se utilizan para proporcionar el cálculo básico y almacenar los elementos utilizados en sistemas digitales. Un elemento lógico básico contiene:

- LUTs son pequeñas memorias (en principio solo de lectura), que permite implementar funciones lógicas al programarlas con la tabla de verdad deseada. Además, tiene tantos bits de dirección como variables tenga la función lógica y un bit de datos que se corresponde a la salida de la función. Esto permite implementar cualquier tipo de lógica combinacional, fraccionando funciones complejas en diversas LUTs interconectadas entre sí.
- FFs o biestables de tipo D que permiten implementar lógica secuencial almacenando un único bit.
- Multiplexores los cuales permiten modificar la interconexión entre las LUTs y los FFs.
- Circuito de propagación de acarreo o *carry chain* que permite optimizar la propagación de acarreo (+1) para la implementación de contadores.

3.2.2 Bloques funcionales programables especializados

Las arquitecturas de las FPGA se han desarrollado a un ritmo exponencial en los últimos años dando lugar a bloques más especializados, como memoria integrada (RAM de bloque), procesadores de señales digitales (DSP48) que ofrecen la operación de multiplicar y acumular, y núcleos de algún microprocesador (normalmente ARM). Además de incluir gestores de reloj digital, lo cual facilita la distribución de la señal de reloj en frecuencia y fase eliminado el *skew* o desviación del reloj [26], y transceptores de alta velocidad, dotando de las entradas/salidas de alta velocidad. Todo esto hizo que las FPGA se convirtieran en plataformas heterogéneas.

3.2.3 Bloques de entrada y salida programables

Los bloques lógicos y la arquitectura de enrutamiento se interconectan con los componentes o equipos externos de una FPGA a través de los pads de entrada y salida reprogramables. Gracias a estos pads y a un entorno de apoyo permiten formar circuitos lógicos formados por componentes. Además, debido a la variación en el voltaje de suministro y con los estándares de referencia es posible programar los distintos voltajes que van a entrar por cada celda de entrada y salida, y poder evitar errores como inyectar más voltaje del que el circuito necesita o por el contario que no funcione correctamente el diseño por falta de este. Por ello las FPGA son compatibles con diversos estándares, y esto se ha de mirar a la hora de hacer el diseño lógico para poder interpretar correctamente los resultados.

3.2.4 Interconexión programable

Las conexiones entre bloques lógicos y de entrada/salida se proporcionan a través de una red programable. Esta interconexión consta de:

- Bloques de conexión, los cuales permiten la conexión de las entradas y salidas de los CLBs o bloques lógicos con las distintas líneas de encaminamiento (los cables) verticales/horizontales.
- Matrices de interconexión, las cuales permiten la conexión entre las líneas de encaminamiento verticales y horizontales ofreciendo de esta forma un cambio de dirección.

- Líneas de encaminamiento segmentado, las cuales son las líneas verticales y horizontales que recorren el dispositivo y ofrece la conexión entre CLBs y pads de entrada y salida además las hay de diversa longitud, conectando bloques adyacentes cada 2 o cada 4 o cada X dependiendo de la arquitectura (esto se puede consultar en el dataSheet general como ejemplo en la FPGA xcku11p-flga2577-1e [27]).
- La línea de reloj especial, ya que es una línea dedicada distribuida en forma de H para minimizar el *skew* [26].

3.2.5 Mejoras en el rendimiento en una FPGA

En el estudio realizado por Martin Herbort [28] se diseñaron 12 métodos para mejorar las implementaciones de las aplicaciones de computación de alto rendimiento (HPC). En esta sección se han resumido esos métodos para así poder aplicarlos al diseño del trabajo.

- **Método 1: Use un algoritmo óptimo para FPGA**

  Antes de acelerar un algoritmo usando una FPGA, es necesario verificar que este vale la pena acelerarlo, al igual que verificar si es de naturaleza reconfigurable y paralela para así aprovechar la naturaleza del acelerador. Normalmente, un algoritmo óptimo para su aceleración en una FPGA es aquel que requiere un alto rendimiento computacional.

- **Método 2: Use un modo de computación apropiado para FPGA**

  Cuando se habla de modo de computación, se hace referencia a las diferencias entre computación software y hardware. Las configuraciones en la FPGA pueden parecerse a un programa escrito en un lenguaje de alto nivel, donde en lugar de software se especifica hardware. Lo que significa, que los buenos modos de computación en software no son necesariamente buenos modos de computación para el hardware, mientras que la configuración de la aplicación sí que puede mejorar sustancialmente su rendimiento.

- **Método 3: Use estructuras FPGA apropiadas**

  Las FPGA admiten varias estructuras de datos como pilas, árboles o colas de prioridad que son omniscientes en la mayoría de los programas, al igual que muchas de las operaciones básicas como búsqueda o reducción. Las estructuras y operaciones análogas suelen diferir de lo que se obtiene traduciendo directamente las estructuras de software en hardware.

- **Método 4: Vivir con la ley de Amdahl**

  La ley de Amdahl establece que, la aceleración significativa de la aplicación a través de una mejora requiere que se mejore la mayor parte de la aplicación. Pero esto es difícil de alcanzar, especialmente en cuanto a las aplicaciones de alto rendimiento existentes.

- **Método 5: Ocultar la latencia de funciones independientes**

  La ocultación de la latencia puede contribuir a lograr un alto rendimiento en aplicaciones paralelas, especialmente en la latencia introducida por la superposición entre computación y comunicación. En implementaciones de FPGA, en lugar de asignar tareas

a procesadores que deben comunicarse entre sí, la ocultación de latencia establece funciones en el mismo chip para operar en paralelo.

- **Método 6: Use la concordancia de tasas para eliminar los cuellos de botella**

  Las implementaciones de multiprocesador ofrecen cierta flexibilidad en la partición por función o datos; pero en las FPGA las funciones se establecen en el chip, lo que significa que el paralelismo a nivel de función ya está integrado. Además, la concordancia de tasas también se puede encontrar en la potencia computacional ofrecidas en un diseño y las E/S vinculadas a las FPGA de destino, por lo que es mejor hacer coincidir las E/S con el paralelismo deseado.

- **Método 7: Aproveche el hardware específico de la FPGA**

  Las FPGA a menudo se ven como sustratos homogéneos que se pueden configurar en una lógica arbitraria. Hoy en día, las FPGA incluyen módulos DSP [29], memorias en chip y otros elementos de procesamiento. Esos elementos se pueden utilizar para realizar tareas específicas que dan como resultado una mejor implementación y utilización de los recursos de la FPGA. Por ejemplo, el Xilinx VP100 tiene 400 BRAM direccionables de forma independiente, de 32 bits y de cuatro puertos; alcanza un ancho de banda sostenido de 20 TB/s. El uso de este ancho de banda facilita enormemente el alto rendimiento, y es un archivo excepcional de las FPGA de la generación actual.

- **Método 8: Use la precisión aritmética adecuada**

  Esto se habló en la sección 5.2.1, pues se vio que una tecnología de optimización mejora la implementación en un acelerador hardware; pues los microprocesadores de gama alta tienen rutas de datos de 64 bits, que en muchas aplicaciones se pasan por alto ya que sólo son requeridos unos pocos bits, en contraste con los microprocesadores donde las rutas de datos son fijas. Las FPGA permiten la configuración de rutas de datos en tamaños arbitrarios, lo que permite una compensación entre precisión y paralelismo. Este beneficio adicional de minimizar la precisión proviene de una propagación más corta a través de unidades más estrechas.

- **Método 9: Use el modo aritmético apropiado**

  Los microprocesadores brindan soporte para enteros y punto flotante, dependiendo de múltiples características multimedia. Sin embargo, en los sistemas DSP, los problemas de coste a menudo requieren que sólo tengan números enteros. Aunque el software puede emular el punto flotante cuando sea necesario, no es favorable utilizar la representación de los datos en punto flotante en la FPGA pues es muy costosa.

- **Método 10: Minimizar el uso de operaciones aritméticas de alto coste**

  Los costes relativos de las funciones aritméticas en las FPGA son diferentes a los de los microprocesadores. Por ejemplo, la multiplicación de enteros en FPGA es diferente en comparación con la suma, mientras que la división es de magnitud lenta. Por lo que, se

recomienda reemplazar las costosas operaciones aritméticas por operaciones simples. Incluso si una operación costosa, como la división, está totalmente encauzada a ocultar su latencia, el coste sigue siendo alto en el área del chip, especialmente si la lógica debe ser replicada. En una FPGA, no es necesario implementar funciones no utilizadas, lo que produce, que el área recuperada se pueda utilizar para aumentar el paralelismo. Por lo tanto, reestructurar la aritmética con respecto a una función de coste FPGA puede aumentar sustancialmente el rendimiento.

- **Método 11: Cree familias de aplicaciones, no soluciones puntuales**

  Las aplicaciones aritméticas de alto rendimiento suelen ser altamente paralelizables y complejas, dando como resultado variaciones en los algoritmos aplicados, así como en el formato de los datos. Si bien es fácil realizar estas variaciones con la tecnología orientada a objetos, esto es mucho más difícil de implementar en lenguajes de descripción de hardware. Pero si esas variaciones se implementan en HDL, se reduce el coste de desarrollo en un mayor número de usos, con lo que permite una mayor reutilización del diseño y se depende menos de desarrolladores de hardware capacitados para poder realizar las variaciones.

- **Método 12: aplicación de escala para un uso máximo del hardware FPGA**

  El paralelismo es el componente que más contribuye a aumentar el rendimiento, pues forma parte del diseño del acelerador; ya que consiste en instanciar tantos elementos de procesamiento como se pueda. Por ejemplo, el dimensionado automatizado de matrices complejas es cada vez más importante para poder portar aplicaciones a las plataformas FPGA.

La adopción de los métodos antes mencionados en la implementación de aplicaciones para los aceleradores hardware es necesaria para poder lograr un alto rendimiento y evitar la infrautilización de las FPGA. Y dado que la red neuronal convolucional se puede paralelizar, se puede representar en diferentes precisiones, es dinámica, escalable, tiene un alto coste computacional, depende de la memoria y es flexible, es un algoritmo muy recomendable para considerar aplicar los métodos anteriormente mencionados.

En la implementación que se va a realizar en el trabajo, se aprovecharán al máximo el paralelismo disponible de la red neuronal adecuando los recursos hardware disponibles en la FPGA. Además, se utilizará estructuras adecuadas para la implementación, y se aprovecharán los recursos heterogéneos disponibles en la FPGA. Para considerar el modo aritmético apropiado y precisión, se usará una representación en punto fijo para los parámetros en lugar de utilizar el punto flotante.

3.3 FPGA vs procesadores de propósito general

Las ventajas de los sistemas basados en FPGA sobre los sistemas basados en unidades de procesamiento como las CPU o las GPU dan la posibilidad de programar bloques lógicos de propósito general para que hagan funciones específicas en cada momento, y a su vez en otro momento puedan realizar otras funciones. Además, las FPGA se pueden organizar en sistemas especializados de alto rendimiento, pues son aceleradores hardware preparados para realizar tareas muy específicas, lo que da como resultado una velocidad de procesamiento mejorada y un mayor rendimiento. En comparación con las GPU, las FPGA se consideran dispositivos de gran

eficiencia energética en los que se adapta mejor a las aplicaciones basadas en dispositivos dependientes de batería. Todas estas ventajas tienen un coste, por el aumento de la complejidad y la reducción de agilidad durante el tiempo de desarrollo; además de que los desarrolladores deben tener en cuenta los recursos hardware disponibles y el mapeo eficiente del algoritmo destino en la arquitectura de la FPGA. Además, las FPGA pueden superar la capacidad computacional de los procesadores de señales (DSP) [29], pero sobre todo mejoran el consumo energético. Además, al controlar las entradas y salidas a nivel de hardware, son capaces de proporcionar un tiempo de respuesta más rápido. También las FPGA generalmente no usan sistemas operativos, por lo que realmente minimizan los problemas de compatibilidad y aportan una verdadera ejecución paralela y determinista para cada tarea que en ellas se ejecute.

3.4 FPGA vs ASIC

Los circuitos integrados de aplicación específica (ASIC) son dispositivos personalizados creados a medida de un particular, en lugar de ser concebidos para un propósito general como las FPGA. Los sistemas ASIC tienen la ventaja de no tener ningún área o sobrecarga de tiempo debido a la lógica de configuración y las interconexiones genéricas, lo que da como resultado un sistema eficiente y más pequeño. Sin embargo, los sofisticados procesos de fabricación ASIC dan como resultado una duración de desarrollo muy larga y complicada, además de unos altos costes iniciales de ingeniería que exigen una metodología muy concreta desde el primer momento, y una metodología de verificación del diseño muy extensa. Por lo tanto, los sistemas ASIC son adecuados principalmente para aplicaciones donde el precio de fabricación se puede compartir entre una gran cantidad de dispositivos. Por contra, las FPGA con su programabilidad son más adecuadas para prototipos donde los ciclos de desarrollo son cortos, y donde los conceptos se pueden probar y verificar en hardware ahorrando de esta forma todo el largo proceso de diseño y fabricación del ASIC. Además, las FPGA son actualizables, pues, por ejemplo, muchos protocolos de comunicación pueden cambiar con el tiempo y se pueden adaptar, sin embargo, las interfaces basadas en ASIC pueden causar problemas de mantenimiento y compatibilidad con versiones posteriores.

4. Creación de una red neuronal

Este capítulo arroja luz sobre qué son las redes neuronales convolucionales y su estructura, dando una pequeña introducción a las diversas capas que la componen. Además, de mostrar cómo es la red que se ha decido usar para el estudio.

4.1 Redes neuronales convolucionales

Las redes neuronales se caracterizan por estar formadas por un conjunto de unidades llamadas neuronas, las cuales intentan simular el comportamiento de las neuronas biológicas a través de la función que se muestra en la Figura 4-1. Cada neurona está conectada con otras a través de los enlaces y en estos el valor de salida de la neurona anterior es multiplicado por el valor del peso, de forma que los pesos pueden incrementar o disminuir el valor de las neuronas adyacentes y además se van actualizando gracias al algoritmo de propagación hacia atrás el cual busca reducir el valor de la función de error de la red por medio de la actualización de los pesos.

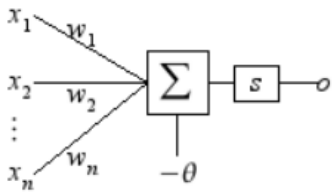


Figura 4-1 Expresa como es el cálculo de una neurona en una red neuronal. Donde x es una entrada, w es el peso de la conexión entre la entrada y la neurona, s es una función de activación, o es la salida de la neurona y Θ es el valor del umbral. La suma ponderada *[30]*.

También cabe resaltar que en las redes neuronales cuando el entrenamiento es supervisado, se suele utilizar un conjunto de datos donde las imágenes ya están etiquetadas y clasificada, pues de esta forma se requiere menos cálculos a la hora de actualizar los pesos. Con este tipo de entrenamiento el conjunto de entrenamiento se pasa varias veces por la red para que aprenda, pues cada imagen al ser diferente en las primeras pasadas del entrenamiento la red falla bastante. Y después de haber entrenado la red varias veces con el conjunto de entrenamiento se pasa un conjunto de prueba donde hay imágenes que nunca han sido pasadas por la red y así se comprueba la precisión que tiene.

Las redes convolucionales son una clase de redes neuronales de retroalimentación, por lo que son adecuadas para operaciones con datos bidimensionales como imágenes. Generalmente comienzan con una capa convolucional, donde se toma la imagen de entrada y se descompone al colocar diferentes filtros de características como bordes, líneas, curvas, etc. Al aplicar estos mapas de características, sacamos esas características que nos interesan de la imagen de entrada y en la última capa de la red se clasifica en clases de salida, utilizando un clasificador.

Una red convolucional típica consta de un número de capas convolucionales, y cuando ya se han obtenido todas las características se aplica una capa de completamente conectada o *fully connected*, donde la mayoría de las operaciones que han sido realizadas se agrupan evitando así un sobreajuste, y más tarde se añade la capa de clasificación, produciendo la salida de red. Una capa en una red convolucional consta de un volumen en 3D de neuronas, como se muestra en la Figura 4-2, es decir, para una red convolucional una imagen tiene alto, ancho y profundidad; al

hablar de profundidad, se refiere a lo que se llama mapas de características, pero no hace referencia al número de capas de la red convolucional.

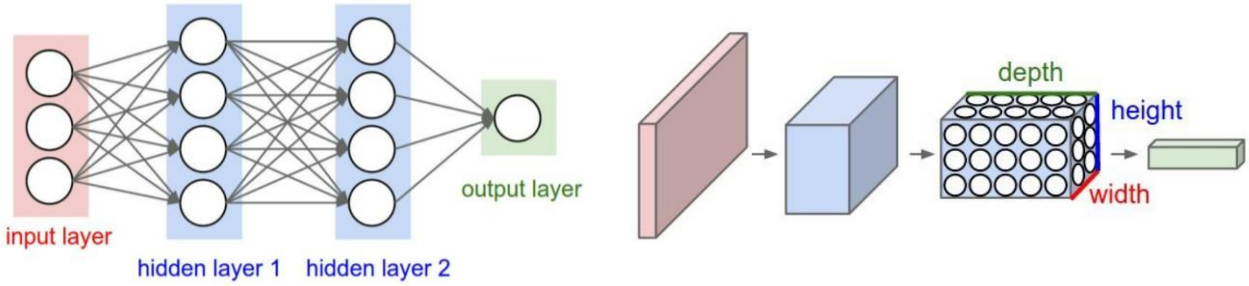


Figura 4-2 Izquierda: se muestra una red neuronal de 3 capas. Derecha: se muestra como la imagen de la izquierda organiza sus neuronas en tres dimensiones (ancho, alto y profundidad), por lo que el volumen de la entrada 3D se transforma en un volumen de salida 3D de neuronas en cada capa *[31]*.

4.2 Capas que puede contener una red neuronal

En esta sección del estudio se va a exponer cómo funcionan y para qué sirven las principales capas de la red, para así poderlas implementar en un futuro y comprobar que la red de estudio del trabajo realiza correctamente cada una de sus capas.

4.2.1 Capa convolucional

La capa convolucional se considera el bloque principal de una red convolucional, y comprende la mayoría de las operaciones en un modelo de CNN. Esta capa, concretamente, realiza una operación matemática llamada convolución, la cual consiste en multiplicar y sumar las diferentes operaciones. La fórmula matemática que define esta capa se muestra en la Figura 4-3 [32]. Además, una expresión más gráfica de su funcionamiento y aplicación se observa en la Figura *4-4*, donde se aprecia como un *kernel* o filtro cuya selección de sus valores depende de la característica que se desee obtener de la imagen de entrada, es multiplicado por un conjunto de entradas, y esas entradas ponderadas se suman. Asimismo, el sesgo, cuyo valor generalmente se agrega a las entradas ponderadas, también se suma para garantizar que la neurona obtenida es correcta.

$$\mathrm{out}(N_i, C_{\mathrm{out}_j}) = \mathrm{bias}(C_{\mathrm{out}_j}) + \sum_{k=0}^{C_{\mathrm{in}}-1} \mathrm{weight}(C_{\mathrm{out}_j}, k) \star \mathrm{input}(N_i, k)$$

Figura 4-3 Expresión matemática de la convolución *[32]*.

Como se puede observar en la Figura 4-4, el número total de parámetros obtenidos es menor a la entrada de la capa; esto se debe a que se aplican receptores de campo local (conectividad local), por lo que, sólo se conectan las neuronas de alrededor, puesto que no hay necesidad de conectarse entre todas, ya que, entre esquinas de una imagen, por ejemplo, no suele haber mucha relación. Una vez hecho esto, la salida de esta capa se convierte en la entrada de la siguiente.

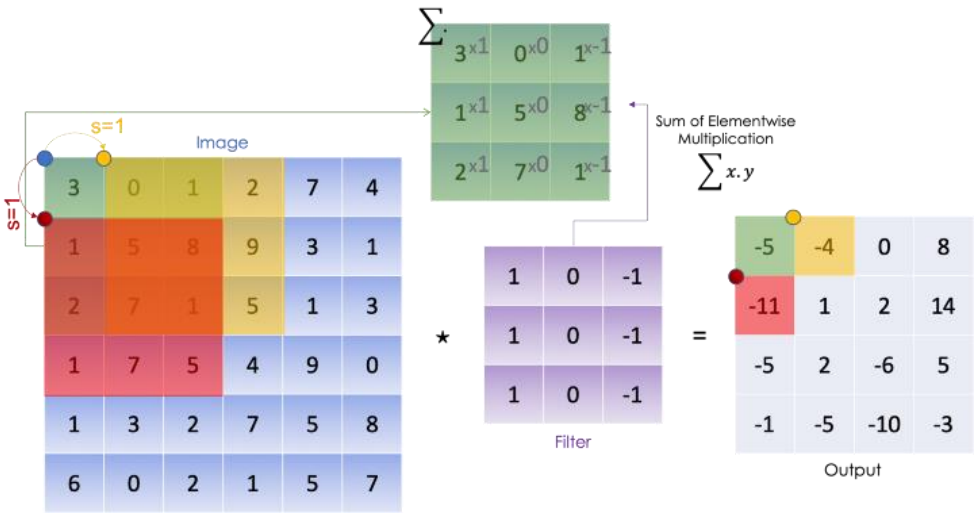


Figura 4-4 Representación de cómo funciona la etapa de convolución; donde con un valor de entrada de tamaño 6x6x1 y un paso o zancada de 1, da como matriz resultante una de tamaño 4x4x1 mostrando el resultado de cada neurona conseguida mediante un color diferente *[33]*.

También hay que resaltar que el tamaño del filtro tiene una importancia significativa, especialmente cuando se operan en las orillas de la matriz o imagen original, pues dependiendo del número de sus filas y columnas, hará que la matriz resultante de esta etapa tenga menos filas y columnas que la original, como se ha visto en el párrafo anterior. Para evitar este problema, se puede utilizar un rellenado de ceros o *Padding*, como se puede ver en la Figura 4-5, el cual consiste en añadir un contorno a la imagen original para así obtener como matriz resultante una del mismo tamaño que la primera sin el rellenado, si es que se desea conservar el tamaño original.

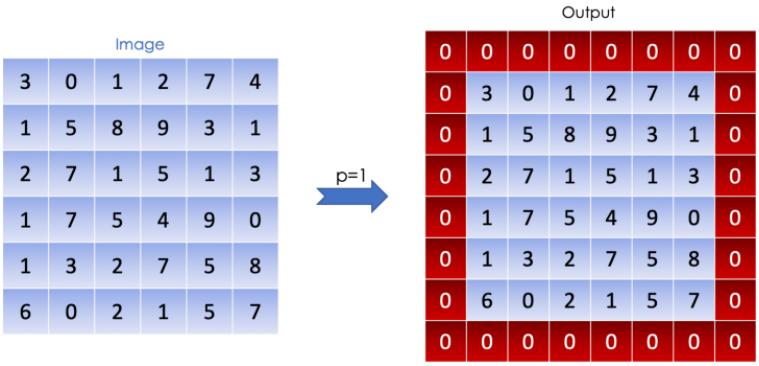


Figura 4-5 Imagen de entrada a la capa convolucional una vez se ha colocado un marco de ceros, en este caso se ha especificado que el marco sea de 1, es decir, que sólo haya una columna y fila a cada lado de la imagen de entrada *[33]*.

4.2.2 Función de activación

La capa de activación se aplica a capa píxel o neurona pues dependiendo del tipo de función que se aplica se definirá el tipo de predicción que puede hacer el modelo. Existen múltiples funciones de activación para una red convolucional, pero en este trabajo se usará una de las más frecuentes, que es la función Relu que se podría definir como $f(x) = MAX\ (0, neurona)$. Esta función, se caracteriza porque elimina las neuronas con valor negativo y les da valor 0, es decir, es una función de umbral cero, de esta forma la red se deshace de información innecesaria y como resultado de esto, la función Relu es la que más fácilmente converge durante el entrenamiento.

4.2.3 Capa de agrupación

Básicamente, la agrupación es una forma de submuestreo que se utiliza para reducir las dimensiones de la función a medida que se va profundizando en la red. Hay varios métodos para realizar las agrupaciones, las más comunes son la agrupación media y máxima. La agrupación basada en la máxima consiste, en que un conjunto de neuronas se submuestra en función de un filtro de agrupación máximo, es decir, se elige el valor máximo del submuestreo y ese es el que pasa a guardarse en el filtro de agrupación, esto se podría definir como $f(x) = MAX(x0, x1, \dots, xn)$. La agrupación por promedio es muy similar, pero en lugar de elegir el máximo del submuestreo se hace una media de este, y es ese valor el que se guarda en el filtro de agrupación, esto se podría expresar como $f(x) = \frac{\sum x0,x1,\dots,xn}{n}$. Un ejemplo de esto se puede ver en la Figura *4-6*.

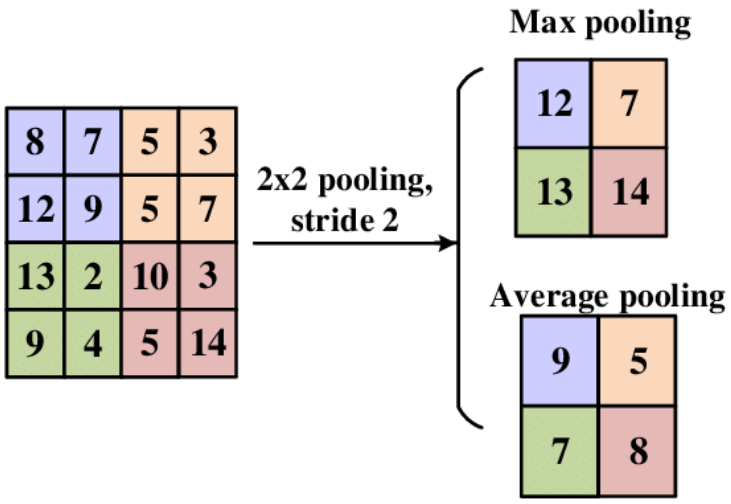


Figura 4-6 Salida de la capa de agrupación, tanto si se hace por promedio (average) como por máximo, para un filtro de 2x2x1 con un paso de dos *[34]*.

Además, hay que tener en cuenta que esta capa tiene la funcionalidad de evitar que la red sea demasiado extensa, por ello el filtro de agrupación ha de ser menor que la entrada, y en función del tamaño de filtro se decidirá el paso o *step*, es decir, en número de filas y columnas en las que se moverá el filtro para elegir otro submuestreo de la entrada, esto también se puede ver en la Figura 4-6 pues cada submuestreo esta seleccionado en un color diferente.

4.2.4 Capa completamente conectada

La capa completamente conectada o *Fully connected* (FC) viene antes que la etapa de clasificación; generalmente coge todas las neuronas que se han obtenido a través de la red y las conecta con todas las neuronas de su etapa anterior, y los parámetros se traducen en las conexiones entre ellas. Su funcionamiento es parecido a la etapa convolucional, pues coge la entrada y la multiplica con los pesos correspondientes, y añade los sesgos respectivamente; esta operación matemática se puede apreciar en la parte izquierda de la función SofMax de la Figura 4-7. Al final de esta etapa se obtiene un vector con el mismo tamaño que las clases a las que puede pertenecer la entrada original.

4.2.5 Capa de clasificación

Esta es la última etapa de una red convolucional, y su principal funcionalidad es clasificar la salida final de la capa anterior (la completamente conectada) en clases específicas, es decir, dará como salida de esta capa un vector con las probabilidades de que la imagen original pertenezca a una clase u otra. En esta capa, lo más frecuente es utilizar la función SoftMax. Básicamente, el clasificador SoftMax convierte las puntuaciones brutas de la clase en una probabilidad de un rango de 0 a 1, donde la más cercana a 1 es la que tiene mayor probabilidad de que pertenezca a esa clase. En la parte derecha de la Figura 4-7 se puede apreciar la operación matemática que realiza esta capa. Además, como las probabilidades de clasificación SoftMax se comparan con las etiquetas reales de las clases disponibles, proporciona una forma de evaluar con precisión el modelo.

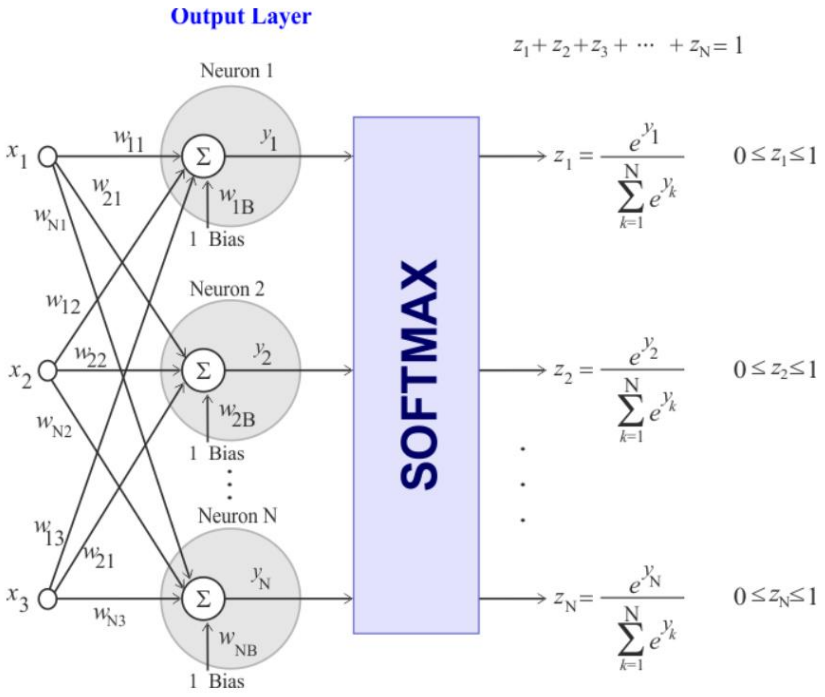
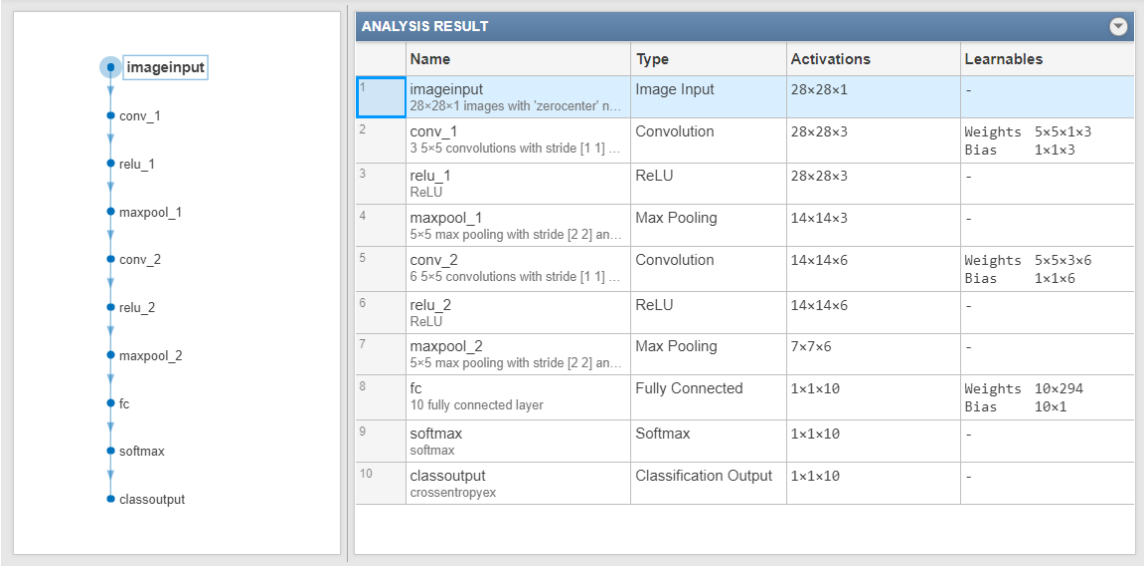


Figura 4-7 La operación matemática que describe el funcionamiento de la capa completamente conectada se encuentra en la parte izquierda de la función SofMax, y en la derecha de la función SoftMax se aprecia la operación matemática que describe la funcionalidad de esta capa *[35]*.

4.3 Diseño de la red convolucional del estudio

Para poder realizar el estudio y probar como funcionaria una red convolucional en una FPGA, primero se ha de decidir qué red se va a implementar, cuáles son las capas que la van a formar, y cuál será su estructura. Para garantizar su funcionamiento correcto y poder identificar y resolver los errores que puedan surgir de forma eficiente se ha decido no usar ninguna de las redes existentes, y crear una red sencilla y así comprobar a qué limitaciones estaría sujeta una red básica. Además, para aumentar la simplicidad del diseño, se ha decidido que las imágenes de entrada de la red serán en blanco y negro, de esta forma la imagen tendrá un único canal y será más sencillo centrarse en su funcionamiento. Por ello se ha decidido usar las imágenes de entrenamiento y de prueba del conjunto de datos MNIST [36].

El conjunto de datos MNIST está formado por un grupo de imágenes de dígitos que van del 0 al 9, por lo que hay 10 clases en los que una imagen se puede clasificar. Estas imágenes fueron tomadas de una variedad de documentos escaneados, normalizados en tamaño y centrado. Por lo que se convierte en un excelente conjunto para evaluar modelos, permitiendo al desarrollador centrarse en el aprendizaje. Cada imagen tiene 28x28 píxeles. Además, este conjunto de datos está dividido en un conjunto preparado para evaluar y comparar modelos, en los que da a disposición del programador 60.000 imágenes para entrenar el modelo, y un conjunto separado de 10.000 imágenes para probarlo.



ANALYSIS RESULT

	Name	Type	Activations	Learnables
1	imageinput 28×28×1 images with 'zerocenter' n…	Image Input	28×28×1	-
2	conv_1 3 5×5 convolutions with stride [1 1] …	Convolution	28×28×3	Weights 5×5×1×3 Bias 1×1×3
3	relu_1 ReLU	ReLU	28×28×3	-
4	maxpool_1 5×5 max pooling with stride [2 2] an…	Max Pooling	14×14×3	-
5	conv_2 6 5×5 convolutions with stride [1 1] …	Convolution	14×14×6	Weights 5×5×3×6 Bias 1×1×6
6	relu_2 ReLU	ReLU	14×14×6	-
7	maxpool_2 5×5 max pooling with stride [2 2] an…	Max Pooling	7×7×6	-
8	fc 10 fully connected layer	Fully Connected	1×1×10	Weights 10×294 Bias 10×1
9	softmax softmax	Softmax	1×1×10	-
10	classoutput crossentropyex	Classification Output	1×1×10	-

Figura 4-8 Estructura de la red convolucional del estudio. Además, se puede apreciar en la columna de activación la salida que se obtendrá en cada capa y los pesos y sesgos de cada una de ellas.

La red convolucional que se va a usar en el trabajo tendrá la estructura que se muestra en la Figura 4-8, donde se puede apreciar que tendrán una primera capa convolucional donde se aplicarán 3 filtros de tamaño 5x5, y dado que por el momento se desea conservar el tamaño de la imágenes a la imagen de entrada se le aplicará un marco o *padding* de 2, esta capa dará como resultado una salida con una imagen de 28x28x3 la cual será la entrada de la siguiente capa, que será una función de activación Relu e irá seguida de una capa de agrupación máxima, donde se usará un filtro de 2x2 y la salida resultante será de 14x14x3. Después, irá otra capa de convolución, donde también se le aplicará un marco o *padding* de 2 como la primera, y se usará 6 filtros de tamaño 5x5, dando lugar a una salida de tamaño 14x14x6, e irá seguida de una función de activación Relu y otra de agrupación máxima, donde el filtro de esta será de 2x2 y la salida resultante de 7x7x6. Por último, se le aplicará la capa de completamente conectada la cual tendrá 7x7x6 = 294 entradas y la de clasificación SoftMax, obteniendo así un vector de probabilidad de clases que irá desde la posición 0 referenciando a la clase del número cero hasta la posición 9 del vector, cuyo contenido tendrá la probabilidad de que la imagen de entrada sea el número 9.

Por lo tanto, a partir de la imagen y estos datos, se puede determinar el número total de pesos necesarios de la red pues son un total de 3184 en nuestro caso y de operaciones de multiplicación y sumas necesarias. Lo cual da una idea de la magnitud del problema, incluso para una red convolucional tan simple como esta y se aprecia la necesidad de acelerar al máximo posible las operaciones manteniendo la precisión.

5. Implementación de una red convolucional

En este capítulo se va a proceder a crear la red convolucional del trabajo descrita en el capítulo anterior apartado 4.3. Para ello, primero se elegirá un entorno para su entrenamiento. A continuación, se realizará la implementación de cada una de las capas y de la red en lenguaje C. Además, se deberán obtener los pesos del framework de entrenamiento para que la red creada tenga los parámetros de entrenamiento, es decir, se implementará la red convolucional en lenguaje C ya entrenada.

5.1 Entrenamiento de una red convolucional

Para que una red convolucional realice la clasificación de imágenes de un conjunto de datos, la red debe estar entrenada. En las redes convolucionales normalmente se usan algoritmos de retropropagación o propagación hacia atrás de errores, es decir, se aplica este método el cual emplea un ciclo de propagación donde primero la red da una salida y esta se compara con la salida deseada y se calcula el error una vez hecho esto y partiendo de la última capa, la salida que ha dado error se propaga hacia todas las neuronas de la capa penúltima a la red, que es la inmediatamente anterior a la salida. Sin embargo, las neuronas de la capa de completamente conectada solo reciben una fracción de la señal total de error, basándose aproximadamente en la contribución relativa que ha aportado cada neurona de esa capa a la salida. Este proceso se repite, capa por capa, hasta que todas las neuronas de la red hayan recibido la señal de error que describa su contribución relativa al error total. Gracias a este proceso se consigue entrenar la red, pues conforme se avanza en el entrenamiento cada vez el error es menor hasta que al final conseguimos una red entrenada. Estas automodificaciones que va haciendo la red al entrenarse se guardan en las matrices de pesos y sesgos, pues al principio del entrenamiento estas matrices están compuestas por valores aleatorios, pero al final de esta ya son valores apropiados para utilizar la red.

Por lo general, los modelos de redes convolucionales se desarrollan utilizando bibliotecas proporcionadas por los principales frameworks de entrenamiento como Tensorflow, Keras o Caffe. Además, el entrenamiento se realiza principalmente mediante una GPU ya que es relativamente fácil de implementar, aunque también se podría hacer en una CPU sin problema. Se suele usar este tipo de hardware para los entrenamientos puesto que está al alcance de cualquier usuario y no requiere de ningún coste adicional.

Para un comienzo rápido y fácil con el entrenamiento de una red convolucional, se recomienda Tensorflow, pues este firmware proporciona una guía completa sobre cómo desarrollar y entrenar una red convolucional, pues todo la información y tutoriales que proporciona están en su web y es de acceso gratuito. Pero, aunque este firmware es mejor a la hora de entender los conceptos y empezar a familiarizarse con las redes convolucionales, en este trabajo no se usará pues los parámetros de pesos y sesgos no los proporciona de forma sencilla al desarrollador y si no estás muy familiarizado con el entorno es difícil acceder a ellos.

En el caso de este proyecto se usará el firmware de entrenamiento Pytorch, pues sí que proporciona muy fácilmente toda la información que compone la red, pues aparte de otorgar los

pesos y sesgos de la red también es sencillo obtener las matrices intermedias o salidas de cada capa de red por separado, lo cual es perfecto para este estudio pues nos da la posibilidad de poder probar cada capa por separado y ver que tiene el mismo funcionamiento que en el framework de entrenamiento.

5.2 Optimización de una red convolucional

Las redes convolucionales son algoritmos de naturaleza paralela y para aprovechar dicha esencia al máximo es mejor explotar el paralelismo disponible dentro de cada capa de forma individual. Además, gracias a que el framework de entrenamiento disponible proporciona módulos para mejorar el funcionamiento de la red y disminuir su tamaño sin perder eficiencia también se aplicará a la red del estudio. De esta forma al explotar el paralelismo se podrá obtener un mejor uso en la FPGA destino explotando al máximo sus cualidades. Estas mejoras se expondrán a continuación.

5.2.1 Módulo de cuantización

Una metodología de optimización que en este tipo de trabajos se debe tener en cuenta es la limitación de datos de precisión. Pues una precisión más baja puede ahorrar una gran cantidad de recursos hardware. Por lo tanto, una reducción eficiente en la precisión del modelo de la red convolucional, teniendo en cuenta el cumplimiento de los requisitos de aplicación, se podría lograr fácilmente, dado que la mayoría de los firmwares de entrenamiento actuales contienen módulos y bibliotecas aptas para poder llevarlo a cabo. En concreto el entorno de entrenamiento de Pytorch permite efectuar esta optimización a la que se llama cuantización.

La cuantización es la técnica para realizar cálculos y almacenar la red en tamaños de bits más bajos que la precisión del punto flotante. Es decir, un modelo de red convolucional cuantificado ejecutará las operaciones de la red con números enteros en lugar de valores en coma flotante; permitiendo así, una representación del modelo más compacto y facilitando de esta forma el uso de operaciones vectorizadas de alto rendimiento en diversas plataformas hardware.

En el modelo que se ha creado, se ha decidido hacer una cuantificación a INT8 y de esta forma conseguir una reducción de 4 veces el tamaño del modelo, y una reducción de 4 veces en los requisitos de ancho de banda de memoria, puesto que el soporte de hardware para cálculos en INT8 suele ser de 2 a 4 veces más rápido en comparación con el cálculo en punto flotante de 32 bits. De esta forma, se evitarán potenciales problemas a la hora de implantar el modelo dentro de un acelerador hardware como una FPGA, pues estos dispositivos no se caracterizan precisamente por su gran capacidad a la hora de conservar datos y dado que el framework seleccionado da la opción de usar dicho módulo se ha decidido intentar explotar todas las mejoras que sean posibles en la red creada para así tener una red convolucional, aunque sencilla, pero con una implantación más profesional.

5.2.2 Matrices intermedias

Todas las capas que forman la red convolucional dan como resultado una salida que, a excepción de la última capa, son la entrada de la capa siguiente. Pues para ahorrar memoria y dado que la red que se ha diseñado es sencilla se ha visto que en las primeras capas da como resultado unas series de matrices donde el resultado puede ser almacenado en un valor diferente al punto flotante, por lo que se ha decidido analizar esto y ajustar lo máximo posible el tipo de dato a usar sin que afecte a la precisión de la red y que de esta forma la red del estudio use únicamente la memoria que necesita.

Para empezar la imagen de entrada en la red será de enteros de 8 bits y al pasar dicha imagen por la red convolucional utilizada se obtiene primero las salidas de la primera convolución. Estas matrices salidas se ha observado que pueden ser enteros de 16 bits ahorrando así la mitad que si fueran en punto flotante. Además, las salidas de la primera activación y de la primera agrupación, también pueden ser enteros de 16 bits, por lo que por el momento se observa que si la primera etapa de la red convolucional del trabajo, todas sus matrices de salida son enteros de 16 bits se proporciona una reducción de ancho de banda de la memoria a la mitad del total de la red, sin afectar a la eficiencia. Pero, sin embargo, esto no se le puede aplicar a la segunda capa de convolución, pues necesita más bits para realizar sus cálculos, por lo que esta segunda parte de la red generara una salida en punto flotante. Y aunque se podría redondear para que de esta forma las salidas de la segunda convolución fueran también enteros se ha decidido dejar esta parte así, porque en caso contrario, la eficacia de la red baja de forma muy considerable, por lo que no es conveniente por un mínimo ahorro de memoria sacrificar el correcto funcionamiento de la red diseñada.

5.2.3 Paralelismo y optimización dentro de los mapas de características de entrada

A la hora de introducir una imagen de entrada en la red convolucional para clasificarla se puede observar que es posible realizar varias cosas. En primer lugar, se ha observado que es posible que las características de entrada, tanto de la red como los mapas de características de entrada a cada capa se pueden procesar en paralelo ya que se pueden combinar para producir una única salida.

Por otro lado, la entrada de una imagen en la red se puede interpretar de muchas formas, pues cada píxel o característica de la imagen puede ser de un tipo de dato, según como el desarrollador dese interpretarlo. En el caso de la red del estudio, se ha decidido que para poder llevar a cabo lo que se ha visto en el apartado anterior, el 5.2.2, se ha decidido que las entradas de la red sean mapas de características de enteros de 8 bits, para así poder ajustar la red y obtener los mismos resultados a la hora de realizar la clasificación y cumplir con los requisitos de aplicación, pero reduciendo la complejidad de los cálculos y la memoria necesaria para llevar a cabo la clasificación.

5.2.4 Paralelismo dentro de los mapas de características de salida

Se ha observado que los mapas de características extraídos de las capas son totalmente independientes entre sí, por lo tanto, todos ellos se pueden calcular en paralelo. En otras palabras, si en un momento dado se está mirando X características de una imagen, entonces es posible ejecutar X procesos paralelos para extraer esas características.

5.2.5 Paralelismo dentro de la convolución

Al realizar la capa de convolución se ha decidido realizarla con la siguiente cabecera, pues de esta forma al hacer genérica la función convolución, se convierte en una capa muy versátil y fácilmente adaptable a cualquier red, pues únicamente habría que redefinir las macros con las características que tendría la red a implementar.

void convolution_2d(int8_t kernel[FIRST_CONV_KERNELS][FIRST_CONV_FEATURES][FIRST_CONV_KERNEL_WIDTH][FIRST_CONV_KERNEL_HEIGHT], **int16_t** imagen[FIRST_CONV_FEATURES][FIRST_CONV_IMAGE_WIDTH][FIRST_CONV_IMAGE_HEIGHT], **int16_t** resultado[FIRST_CONV_KERNELS][FIRST_CONV_FEATURE_WIDTH][FIRST_CONV_FEATURE_HEIGHT]);

En concreto en la capa de convolución creada se ve como se explicó en el apartado 4.1, que la red convolucional tiene un volumen de 3D, pues los filtros de entrenamiento de la cabecera se definen como: número de filtros a ejecutar x tamaño del filtro, y este último se ve claramente como consta de tres dimensiones. Además, este volumen de 3D de la red no solo se aprecian los filtros sino también en la matriz de entrada y en la de salida.

Por otro lado, se ha apreciado que esta capa de convolución es ya de por sí una capa que favorece considerablemente la paralelización pues como se explicó en el apartado 4.2.1 la matriz resultado tiene un tamaño menor a la de entrada lo que obliga a que sean matrices diferentes la de entrada y salida. Aunque en el caso de la red del trabajo ya se explicó que la salida de esta capa tenía el mismo tamaño que la entrada pues al aplicarle un *padding* o marco de 2 esto sería posible, por lo que se podría pensar que con el fin de ahorrar memoria se podría almacenar la matriz resultado en la misma matriz de entrada, pero esto se ha descartado. Debido a que el algoritmo descrito, aunque este en lenguaje C y esto no suponga un problema por el momento, como se tiene la intención de implementarla dentro de un acelerador hardware se ha decidido desde el primer momento aplicar una serie de métodos, los cuales se ven en el apartado 3.2.5, de manera que posteriormente sea más fácil realizar una implementación eficiente. Por lo tanto, como resultado de esto se ha decidido mater los mapas de características de entrada y salida separados de forma que se podría realizar de forma paralela tanto la búsqueda de cada característica al aplicar un filtro como calcular de forma simultánea sobre la misma entrada aplicando distintos filtros. Por lo tanto, gracias a esta medida se puede garantizar que se obtendrán más de una neurona por ciclo de reloj.

5.2.6 Paralelismo dentro de la función de activación

Esta capa en un origen se diseñó con la idea de modificar la matriz de entrada, pues como se explicó en el apartado 4.2.2, únicamente se dedica a remplazar los valores negativos por ceros. Pero al estudiar este diseño e intentar que siga “las buenas prácticas” que se ven en el apartado 3.2.5 se ha creado esta capa para intentar que a la hora del paso a la FPGA sea de forma optimizada. Por ello, se decidió separar las matrices de entrada y de salida. Además, en esta capa podemos apreciar también un paralelismo intrínseco pues como solo nos interesa mirar si los valores son positivos o negativos, se puede decir que cada dato es independiente entre sí, por lo que en el mejor de los casos se podría hacer tantas ejecuciones paralelas como datos tiene la matriz entrada.

5.2.7 Paralelismo dentro de la capa de agrupación

Al implementar la capa de agrupación se ha observado que al igual que la capa de convolución su propia funcionalidad ya le otorga paralelismo pues las operaciones que realiza se pueden paralelizar en función del submuestreo. En otras palabras, por cada submuestreo de la matriz de entrada se obtendrá una salida y al ser todas ellas independientes entre sí, es posible que esta capa tenga tanto paralelismo como grupos de submuestreo haya. Por lo que, podría haber tanto hilos paralelos como tamaño tenga la matriz de salida, pues cada submuestreo como se ha dicho obtendrá una neurona de salida independiente.

6. Implementación de la red neuronal en diferentes equipos

Las redes neuronales convolucionales se pueden implantar en diversas plataformas como en unidades de procesamiento de propósito general (CPU), unidades de procesamiento gráfico (GPU) o FPGA. Como se mencionó en el capítulo 3 del trabajo, las CPU son plataformas que no favorecen la ejecución de redes convolucionales, ya que las infrautilizan; pues estas, son de naturaleza paralela, y la mayoría de las aplicaciones para las que se utiliza es para el procesamiento y reconocimiento de imágenes. Por lo que utilizar una CPU para esto no encaja muy bien, pues las CPU son elementos de procesamiento basado en secuencias y no son aptos para el procesamiento de imágenes ni la toma de decisiones. Si bien esta no es la mejor forma para procesar una red convolucional, las GPU sí que lo son, pues son plataformas que favorecen el entrenamiento de una red neuronal convolucional y obviamente se debe a que estas explotan el paralelismo de las redes convolucionales. Pero, sin embargo, aunque sí favorezcan su entrenamiento, no son buenas para su ejecución, pues este tipo de plataforma tiene un alto consumo energético.

Dado que las CPU no aprovechan el paralelismo disponible en una red convolucional y las GPU no son aptas para dispositivos dependientes de la batería, las FPGA consiguen equilibrar ambas plataformas; es decir, pueden simular la potencia de cálculo de la CPU, pero aprovechando el paralelismo de las redes como las GPU y además siendo eficientes energéticamente. Además, como se dijo en el capítulo 3 las FPGA son el tipo de dispositivos reconfigurables que pueden adaptarse a requisitos de diseño particulares.

Como siempre, la vida no es tan fácil, y el caso de las FPGA no es una excepción, pues para acelerar una red convolucional en una FPGA el diseñador tiene que pasar por un proceso de desarrollo muy largo y complicado utilizando un lenguaje de descripción de hardware (HDL) como VHDL o Verilog. El desarrollo con HDL puede dar como resultado la implementación más optimizada de un acelerador; sin embargo, algunos diseñadores prefieren sacrificar algo de rendimiento para simplificar con esto la implementación y reducir el tiempo de desarrollo. Este es el caso de este trabajo, pues mediante una herramienta de síntesis de alto nivel, como Vivado HLS, la cual ofrece una metodología alternativa para implementar algoritmos en aceleradores hardware, donde se reemplaza el lenguaje de descripción de hardware por lenguajes de medio nivel como C o C++ y mediante la síntesis que proporciona el entorno, traduce ese lenguaje a uno de descripción de hardware.

Por lo que, en este capítulo, se podrá ver, cómo es la herramienta que se ha utilizado para implantar la red convolucional dentro de una FPGA, así como la comparación de resultados de implantarla en distintas plataformas hardware como las mencionadas.

6.1 Lenguajes de descripción de hardware y síntesis de alto nivel

El lenguaje de descripción de hardware y la síntesis de alto nivel, son técnicas muy correlacionadas, ya que comparten objetivos similares que se logran en diferentes metodologías, pues ambos se utilizan para escribir el código de los algoritmos, los cuales se desea que se implanten en circuitos lógicos digitales como FPGA o ASIC.

6.1.1 Lenguajes de descripción de hardware

Los lenguajes de descripción de hardware incluyen, entre otros, a VHDL, Verilog o System C. que son lenguajes de programación especializados. Se utilizan para definir la estructura y el diseño de los circuitos electrónicos digitales. Tanto VHDL como Verilog, son lenguajes que han madurado, y se han especificado mediante estándares como IEEE; pues VHDL, especialmente, se utiliza para describir circuitos digitales, y para la automatización de diseño electrónico; y Verilog, es usado para modelar sistemas electrónicos, pues soporta diseño, prueba e implementación en circuitos analógicos, digítales y de señal mixta a diferentes niveles de abstracción. Mientras que SystemC es una biblioteca basada en C++ que se utiliza para modelar sistemas a nivel de comportamiento.

En el diseño de circuitos digitales, el nivel de transferencia de registro RTL (en inglés *Register Transfer Level*) es una abstracción de diseño, que modela un circuito digital síncrono en términos del flujo de señales digitales (datos) entre los registros de hardware y las operaciones lógicas. De forma que el modelo que los usuarios especifican es donde se encuentran los detalles del algoritmo utilizando una serie de procesos paralelos que operan en vectores de señales binarias, y lo cual deriva en el tipo de datos que representan. Esos procesos paralelos son impulsados por los flancos de subida y bajada de una señal de reloj, y describen la lógica combinacional, las operaciones aritméticas básicas y los registros. Las descripciones de RTL están muy cerca de los buses y las puertas lógicas que están disponibles dentro de la tecnología de la FPGA subyacente. Sin embargo, el proceso de transformar un algoritmo dado en procesos, bloques lógicos y máquinas de estados a nivel de transferencia de registro es muy largo, tedioso y propenso a que se cometan errores. Por lo que, los diseñadores, deben considerar y hacer múltiples diseños, antes de intentar escribir cualquier código; pues los cambios a posteriori son difíciles y costosos de realizar. Por lo tanto, todo este proceso exige mucha intuición y una gran experiencia en los diseñadores, con el fin de tener una implementación totalmente optimizada y funcional del algoritmo deseado. Con lo que se obtiene, que un desarrollador especializado en lenguajes de descripción de hardware (HDL) no es fácil de encontrar, pues es complicado que alguien posea todas las habilidades de programación para los HDL adecuadas.

6.1.2 Síntesis de alto nivel

Para superar las barreras introducidas por el desarrollo con lenguajes de descripción de hardware o HDL, se ha realizado una pequeña búsqueda para aumentar el nivel de abstracción, reducir el tiempo de desarrollo y simplificar la implementación. La síntesis de alto nivel o HLS ofrece a los diseñadores un camino alternativo para la implementación de algoritmos. En HLS, una gran cantidad de detalles de implementación se abstraen y se manejan por el compilador de HLS, el cual, convierte el código desarrollado en un lenguaje de nivel alto o medio, en un lenguaje de descripción de hardware, generalmente a nivel de RTL.

Con HLS, los diseñadores pueden implementar sus diseños a través de bucles, llamadas a funciones, matrices y otras operaciones aritméticas relevantes. Tanto los bucles, las matrices, las llamadas a funciones, etc., se convierten en contadores, multiplexores, multiplicadores, memorias, centro de cálculos y protocolos. La compilación se puede guiar utilizando directivas de compilación con scripts o pragmas, que son metainstrucciones interpretadas directamente por el compilador HLS [37] [38]. Vivado Hight-Level Synthesis o Vivado HLS ofrecido por Xilinx es una de las herramientas más usadas para su realización, debido a la gran experiencia que tiene este fabricante en el tema, y por eso será que se utilizará esta herramienta para realizar este estudio.

Aunque HLS puede proporcionar ciclos de desarrollo más rápidos, además de un seguimiento más fácil para la implementación en hardware y una mayor productividad, las herramientas HLS no ofrecen optimización suficiente para muchas de las aplicaciones, pues la optimización en HLS está limitada y definida por las directivas y programas que están incrustados en la herramienta. De hecho, las herramientas HLS han estado en el mercado durante aproximadamente 18 años; sin embargo, los diseñadores todavía usan lenguajes de descripción de hardware para sus diseños en FPGA.

La tarea de convertir descripciones de software secuencial de alto nivel en arquitecturas de hardware es extremadamente compleja. Aunque las empresas han invertido cientos de millones de dólares y años de investigación sobre la realización de la síntesis de alto nivel [39], cuyos resultados obtenidos aún son dependientes, en gran medida, del estilo de codificación y de los intrincados detalles en el diseño, debido a que los defectos y deficiencias en el compilador sólo se descubren durante esté. Dicho esto, la implementación de algoritmos, usando HDL, es tedioso y complicado, y los niveles de optimización no son precisamente muy altos.

Los modelos de redes neuronales convolucionales varían en el tamaño, sin embargo, los modelos pequeños se consideran grandes para ser implementados, usando un lenguaje de descripción de hardware. En realidad, no es práctico implementar grandes y profundos modelos de redes neuronales convolucionales descritas directamente en HDL. Además, la implementación de modelos de redes convolucionales profundos, usando HLS, podría resultar poco adecuado para esos modelos; pues debido a su falta de optimización, no se lograría el mejor rendimiento posible, pero para redes convolucionales pequeñas sí que podría resultar muy ventajoso utilizar estas herramientas de HLS.

Para superar la cuestión del largo tiempo de desarrollo introducido en el apartado anterior, sobre los lenguajes de descripción de hardware, y de utilización causada por la alta abstracción introducida en este apartado, se ha decidido utilizar, ya que la red a implementar en el estudio es pequeña, una herramienta HLS que será Vitis HLS pues contiene una interfaz gráfica de usuario diseñada para generar automáticamente código VHDL o Verilog para los modelos de redes convolucionales que se deseen crear.

6.2 Herramienta de generación de VHDL

VHDL es uno de los lenguajes de descripción de hardware más comunes para describir o modelar circuitos hardware a nivel de transferencia de registro (RTL). En VHDL, los diseñadores suelen especificar los detalles de su algoritmo utilizando una serie de procesos paralelos que describen algunas combinaciones de la lógica operacional y de los registros aritméticos básicos. Estos procesos son impulsados por flancos ascendentes y descendientes de una señal de reloj, y operan en vectores de señales binarias y tipo de datos derivados de ellas.

Como se expuso anteriormente el proceso de transformar un algoritmo en procesos, bloques lógicos y en máquinas de estado, es muy largo y complicado; y los cambios que se realizan a posteriori son difíciles de implementar, y requieren un alto coste, por lo que los diseñadores deben hacer una gran cantidad de diseños antes de escribir el algoritmo final. Además, como se vio, desarrollar usando HDL requiere una gran experiencia para poder reducir los cambios y garantizar los resultados de un diseño satisfactorio. Además, la dificultad del desarrollo en HDL impide la optimización iterativa, y exige mucha intuición para tener una implementación de los algoritmos totalmente optimizada y funcional. Por lo tanto, el desarrollo con HDL no es utilizado por muchos investigadores y menos por aquellos que no están familiarizados con él. Esto, en realidad, hace que las FPGA sean mucho menos atractivas para acelerar cualquier algoritmo y más una red neuronal convolucional, que es un algoritmo mucho más complicado, y donde una red sencilla como la LeNet-5 [40] podría llevar meses para poder realizar su implementación.

Como se ha visto, no es práctico implementar modelos de redes convolucionales, especialmente si son grandes; usando un lenguaje de descripción de hardware desde cero; por lo que como ya se comentó se usará para eso una herramienta de generación automática, y para ello se ha elegido Vitis HLS.

La herramienta Vitis HLS automatiza gran parte de las modificaciones de código necesarias para implementar y optimizar un código en C o C++ en lógica programable; además de lograr una baja latencia y un alto rendimiento, aunque esto dependerá también de lo complicado que sea el algoritmo descrito. Además, también permite la personalización de su código para implementar estándares de interfaz y optimizaciones específicas para lograr los objetivos del diseño.

En la Figura 6-1derecha, se muestra el flujo de diseño que sigue esta herramienta, pues las entradas incluyen las funciones escritas en C y C++, la cual es la entrada principal de Vitis HLS y puede contener una jerarquía de subfunciones. Además de las restricciones de diseño que se especifican con el periodo e incertidumbre de reloj, y con el objetivo que tiene el dispositivo creado, estas directivas son opcionales, y dirigen el proceso de síntesis para implementar un comportamiento y optimización específicos. También como entrada hay un banco de pruebas de C con todos los archivos necesarios para simular la función C antes de la síntesis y para verificar la salida RTL mediante la co-simulación C o RTL. En cuanto a las salidas producidas por la herramienta, se encuentran los archivos compilados (.xo), los cuales permiten crear funciones hardware compiladas para usarlo en el flujo de desarrollo de la aceleración de la herramienta Vitis HLS. Otra salida son los archivos de implementación RTL en formato de lenguaje de descripción de hardware (HDL). Este es el resultado principal de Vitis HLS, pues es ahí donde se describe el flujo que permite usar el código C/ C++ como fuente para el diseño de hardware. El RTL IP producido por Vitis HLS está disponible en los estándares Verilog (IEEE 1364-2001) y VHDL (IEEE 1076-2000) y se puede sintetizar e implementar en dispositivos Xilinx utilizando la herramienta Vivado Design Suite. La última salida que produce esta herramienta Vitis HLS, es la generación de informes como resultado de la simulación, síntesis y co-simulación.

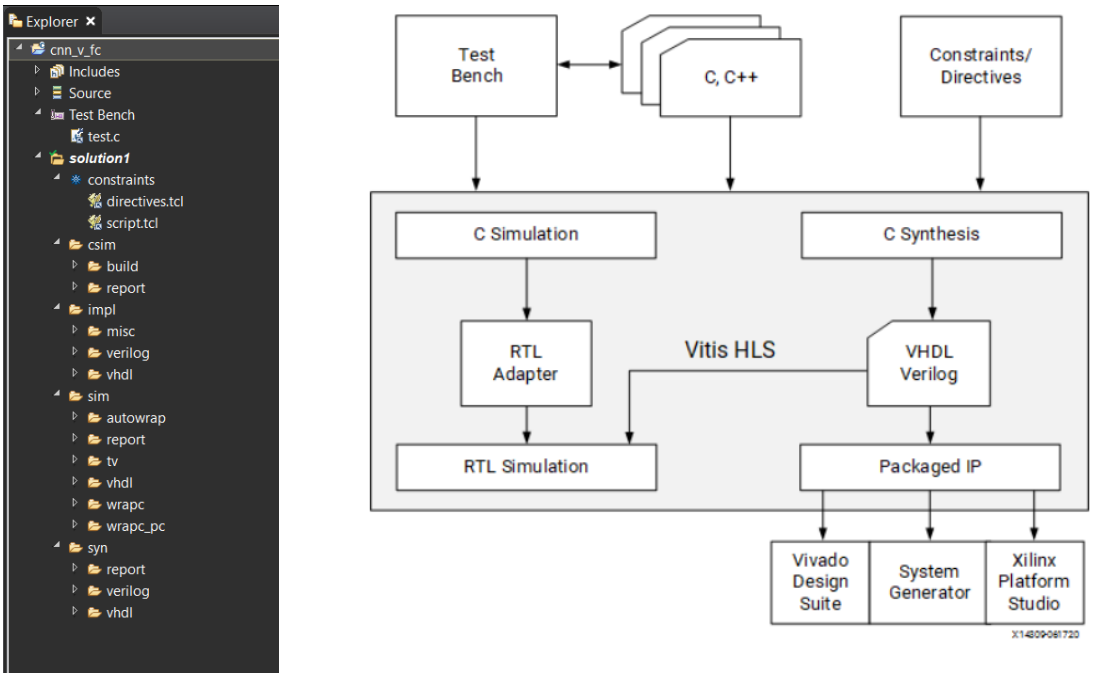


Figura 6-1 Izquierda: estructura de ficheros. Derecha: flujo de diseño de la herramienta Vitis HLS *[41]*

Por lo tanto, Vitis HLS proporciona la estructura de ficheros que se muestra en la Figura 6-1izquierda, donde se muestra como estará estructurado el proyecto y donde se aprecia todos los archivos que genera automáticamente la herramienta como el código en Verilog o VHDL, todos los ficheros necesarios para hacer la implementación o simulación, etc. Además, permite esta herramienta exportar todo su contenido mediante RTL.

Además Vitis HLS permite describir directamente los algoritmos en su entorno usando C/C++ sin necesidad de diseñarlos en otro entorno y luego exportarlos, pues como se muestra en la Figura 6-2, permite compilar el código descrito en C y ofrece un documento con los resultados y errores en el caso de que los hubiera, en concreto en el caso de la Figura 6-2 se han procesado dos imágenes en la red del proyecto y el informe dicta que las ha clasificado correctamente. Por lo tanto, una ventaja que ofrece esta herramienta es que evita si no se desea el tener que realizar el algoritmo en otro entorno y así crearlo directamente que las restricciones de aplicación que sean necesarias.

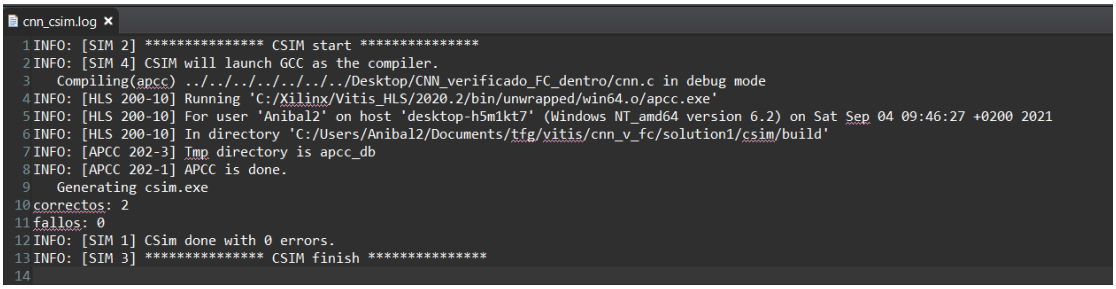
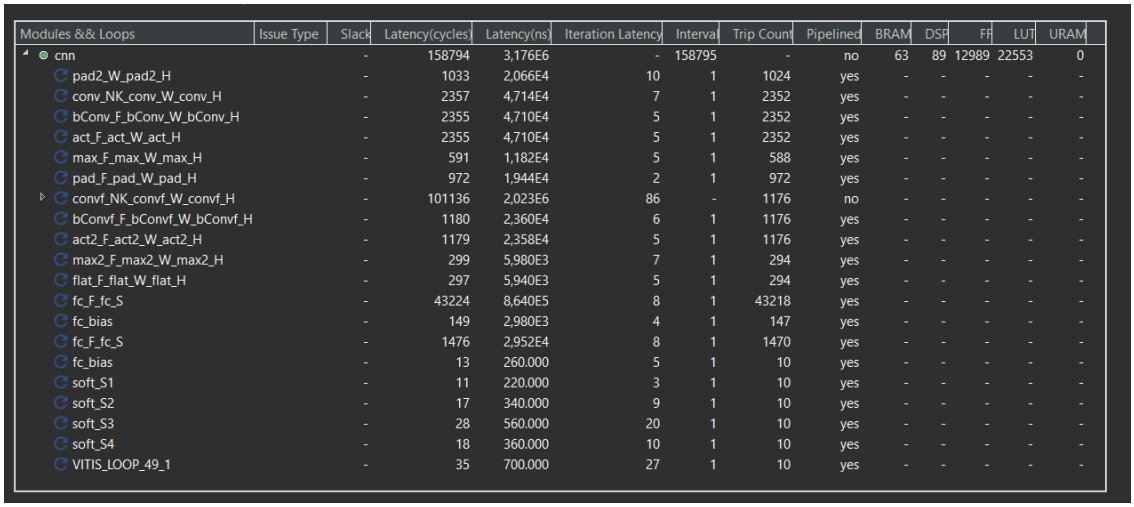


Figura 6-2 Documento generado por la compilación en C del algoritmo

Los proyectos creados mediante Vitis HLS pueden contener múltiples variaciones, llamadas “soluciones” para impulsar la síntesis y la simulación. Cada solución puede apuntar a un flujo de IP de Vivado diferente, y según el flujo de destino, cada solución especificará diferentes restricciones y directivas de optimización; por lo que se puede utilizar dichas directivas para

modificar y controlar la implementación de la lógica interna y de los puertos de entrada y salida, anulando los comportamientos predeterminados de la herramienta si se desea. Por lo tanto, se consigue que en un mismo proyecto se puedan simular diferentes diseños realizando pocos cambios a un nivel que cualquier desarrollador podría realizar, sin mucho esfuerzo, pues el algoritmo que modifica y desarrolla el programador ya estará escrito en código C o C++.



Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
cnn		-	158794	3,176E6	-	158795	-	no	63	89	12989	22553	0
pad2_W_pad2_H		-	1033	2,066E4	10	1	1024	yes	-	-	-	-	-
conv_NK_conv_W_conv_H		-	2357	4,714E4	7	1	2352	yes	-	-	-	-	-
bConv_F_bConv_W_bConv_H		-	2355	4,710E4	5	1	2352	yes	-	-	-	-	-
act_F_act_W_act_H		-	2355	4,710E4	5	1	2352	yes	-	-	-	-	-
max_F_max_W_max_H		-	591	1,182E4	5	1	588	yes	-	-	-	-	-
pad_F_pad_W_pad_H		-	972	1,944E4	2	1	972	yes	-	-	-	-	-
convf_NK_convf_W_convf_H		-	101136	2,023E6	86	-	1176	no	-	-	-	-	-
bConvf_F_bConvf_W_bConvf_H		-	1180	2,360E4	6	1	1176	yes	-	-	-	-	-
act2_F_act2_W_act2_H		-	1179	2,358E4	5	1	1176	yes	-	-	-	-	-
max2_F_max2_W_max2_H		-	299	5,980E3	7	1	294	yes	-	-	-	-	-
flat_F_flat_W_flat_H		-	297	5,940E3	5	1	294	yes	-	-	-	-	-
fc_F_fc_S		-	43224	8,640E5	8	1	43218	yes	-	-	-	-	-
fc_bias		-	149	2,980E3	4	1	147	yes	-	-	-	-	-
fc_F_fc_S		-	1476	2,952E4	8	1	1470	yes	-	-	-	-	-
fc_bias		-	13	260.000	5	1	10	yes	-	-	-	-	-
soft_S1		-	11	220.000	3	1	10	yes	-	-	-	-	-
soft_S2		-	17	340.000	9	1	10	yes	-	-	-	-	-
soft_S3		-	28	560.000	20	1	10	yes	-	-	-	-	-
soft_S4		-	18	360.000	10	1	10	yes	-	-	-	-	-
VITIS_LOOP_49_1		-	35	700.000	27	1	10	yes	-	-	-	-	-

Figura 6-3 Sección de los recursos estimados del informe de la función de síntesis de la herramienta Vitis HLS.

El código creado en la herramienta en C o C++ sería sintetizado de la siguiente manera: Para empezar, los argumentos de las funciones de nivel superior se sintetizarán automáticamente en interfaces de puerto de entrada y salida a nivel de transferencia de registro. Estas interfaces predeterminadas que crea la herramienta dependen del flujo destino, el tipo de datos, la dirección del argumento de la función y el modo de la interfaz predeterminada por el usuario. Las subfunciones de las funciones en C o C++ de nivel superior se sintetizarán en bloques en la jerarquía del diseño RTL, creando un diseño final de RTL que incluya una jerarquía de módulos o entidades que se corresponden con la jerarquía de las funciones C de nivel superior original. Además, Vitis HLS integra automáticamente las subfunciones en las funciones de nivel superior, según sea necesario, para mejorar el rendimiento. También puede deshabilitar la inserción automática especificando el *INLINE* pragma en una subfunción, o usando el *set_directive_inline* y configurándolo a off en su solución. De forma predeterminada, cada llamada de la subfunción C utilizará la misma instancia del módulo RTL. Sin embargo, puede implementar varias instancias del módulo RTL para mejorar el rendimiento. Los bucles en las funciones de C se mantienen enrollados y se canalizan de forma predeterminada para mejorar el rendimiento, pues la herramienta Vitis HLS no los desenrollará a menos que mejore el rendimiento de la solución, como por ejemplo desenrollar bucles anidados para paralelizar el bucle de nivel superior, esto se ve en la Figura 6-3 pues en ese ejemplo de síntesis la herramienta ha considerado necesario desenrollar ciertos bucles como padd2_W y padd2_H. Cuando se enrollan los bucles, la síntesis crea la lógica para la iteración de este, y el diseño RTL ejecutará esta lógica para cada iteración del bucle en secuencia. En cambio, los bucles desenrollados, permiten que algunas o todas las iteraciones se produzcan en paralelo, pero a su vez también consumen más recursos del dispositivo. Las matrices en el código se sintetizan en bloques RAM, LUT RAM o UltraRAM [42] en el diseño final de FPGA, y cuánta cantidad se requiere de cada una para implementar el algoritmo también lo proporciona el informe de síntesis como se muestra en la Figura 6-3. Sin

embargo, si la matriz está en la interfaz de función a nivel superior, la síntesis implementa la matriz como puertos con acceso a una RAM de bloque fuera del diseño. También es importante resaltar que, si se especifica un pragma o directiva en un ámbito particular, ya sea función, bucle o región, entonces el comportamiento predeterminado de la herramienta, como se acaba de describir, será anulado y sustituido por el comportamiento que el pragma especifique. Además, aparte de mostrar los resultados mencionados anteriormente la síntesis también proporciona los tiempos de cada una de las partes que forman el algoritmo (Figura 6-3) al igual que toda la información necesaria de cómo están construidas las interfaces hardware entre otras (Figura 6-4).

Figura 6-4 Sección de las interfaces hardware del informe de síntesis del algoritmo proporcionado por Vitis HLS

Después de la síntesis, se pueden analizar los resultados con los diversos informes producidos por la herramienta y así determinar la calidad de sus resultados. Una vez analizados estos resultados, se pueden crear soluciones para el proyecto, especificando diferentes restricciones y directivas de optimización, y así analizar los nuevos cambios. De esta forma, se pueden comparar los resultados de diferentes soluciones, para ver qué ha funcionado o qué no; y esto se puede repetir hasta que se obtiene el diseño con las características de rendimiento deseadas, pues el uso de múltiples soluciones permite al usuario continuar desarrollando, conservando las soluciones anteriores.

Para que el desarrollador pueda cerciorarse que el algoritmo es correcto Vitis HLS como se ha dicho proporciona una simulación donde se puede ver que los resultados que se generan en cada etapa del algoritmo son los mismo que los producidos al ejecutarlo en lenguaje C. Esto se puede ver en la Figura 6-5, pues se pueden ver cuando se realiza un cambio en una dirección y ver cómo va avanzando el algoritmo. Así en el caso de que no funcione como se desea se puede modificar antes de realizar la implantación en el hardware.

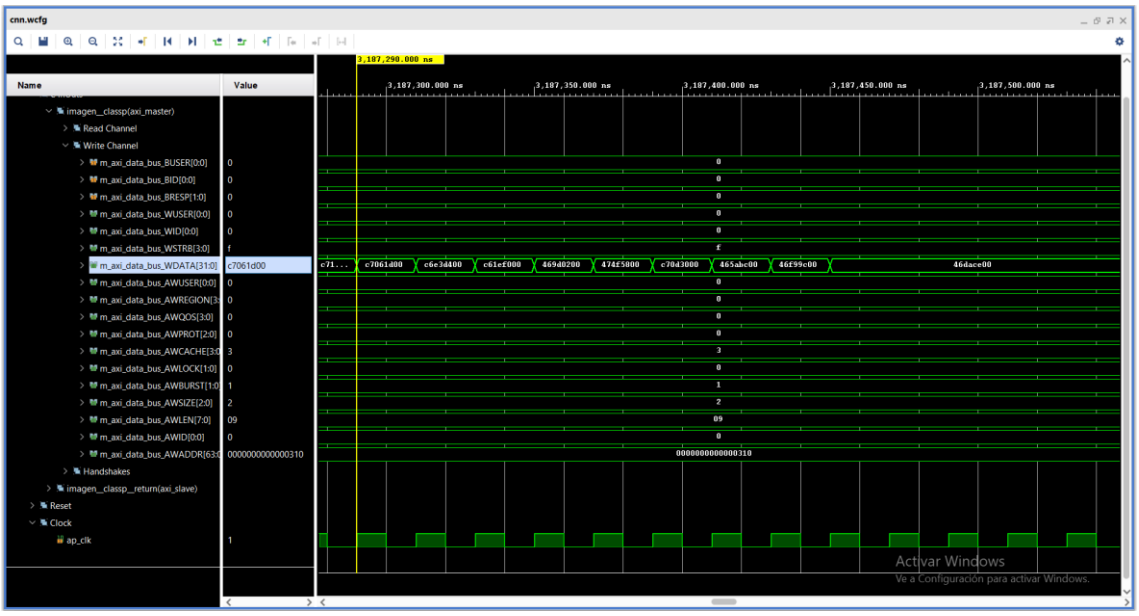


Figura 6-5 Fichero de simulación de la síntesis del algoritmo descrito

6.3 Inserción de la red convolucional en una FPGA

Como se ha visto a lo largo del trabajo, se tiene una red convolucional sencilla descrita en lenguaje C ya entrenada; además de que se ha elegido un entorno apropiado para poder traducir esa red neuronal convolucional a un lenguaje de descripción de hardware como VHDL. Por lo que ya tenemos todos los factores necesarios para poder implementar la red convolucional de trabajo dentro de una FPGA y así poder ver si es posible cumplir el objetivo principal del trabajo.

A la hora de realizar la implementación de la red dentro del hardware, se ha visto que únicamente con seguir unas buenas directivas en el diseño y usar una herramienta adecuada para realizar la traducción del algoritmo, no es suficiente; pues se han tenido que añadir directivas nuevas para poder llevar a cabo tanto la red en si como las diferentes capas que la forman.

6.3.1 Inserción de las capas de la red convolucional

Al implementar cada una las capas de la red del trabajo, se ha visto que es necesario separar las matrices de entrada y salida; pues en caso contrario, la solución se podría guardar en una matriz intermedia, que luego volcaría sus datos en la matriz solución o salida de la capa. Esto funcionaria, pero perjudicaría su implementación en la FPGA, pues al seguir los modelos, apartado 3.2.5, para crear una buena implementación y al investigar cómo funcionan las matrices dentro de una FPGA se ha visto que se deben separar las matrices de entrada y salida.

Las matrices en la FPGA se implementan en bloques de memoria RAM de doble puerto (2 puertos de lectura y 1 de escritura). Pero si se ha de transferir la información de una matriz auxiliar a la salida, entonces se ha de recorrer toda la memoria para transferir la información de un bloque a otro, lo cual, simplemente consume ciclos de reloj sin añadir ningún beneficio, por lo que únicamente se estará aumentando el tiempo a la hora de obtener una salida. Aunque la matriz de datos de entrada sí que se podría reutilizar, pero posteriormente. Esto se podría dar si se ejecutaran varias imágenes sobre la FPGA, haciendo que cuando una imagen salga de una capa y pase a la siguiente, se comience a procesar la nueva, de forma que, aunque no se tuviera aún la salida de clasificación de la imagen de entrada primera, la red comenzara a procesar la siguiente, reutilizando así las entradas.

Aunque la separación de las entradas y salidas de la capa es una decisión que se tomó en el diseño de la capa, y se ha visto en la implementación que efectivamente fue una decisión correcta, al realizar la implementación de la capa se ha visto que no es suficiente. Pues como las FPGA guardan las matrices dentro del mismo bloque de memoria RAM, esto hace que sólo sea posible acceder a dos datos simultáneamente dentro del mismo bloque de memoria, por lo que se ha visto necesario fraccionar las matrices y distribuir su información en diferentes bloques de memoria RAM para así facilitar al máximo el acceso en paralelo. Esto se ha conseguido gracias a la posibilidad que da el entorno elegido de modificar los valores predeterminados; por ello en este caso, se ha decido hacer uso de los pragmas que la herramienta HLS proporciona.

Los pragmas HLS se pueden utilizar para optimizar el diseño, ya sea reduciendo la latencia, mejorando el rendimiento o reduciendo el uso de recursos de área. En el caso concreto de esta etapa, se ha visto la necesidad de usar el *Pragma HLS ARRAY_PARTITION*. Esto permitirá a la imagen de entrada dividirla en matrices más pequeñas, obteniendo como resultado, que la imagen se dividirá en múltiples bloques de memoria más pequeños o en múltiples registros, en lugar de un gran bloque de memoria; un ejemplo de su funcionamiento se ve en la

Figura *6-6*, lo que aumentará efectivamente la cantidad de puertos de lectura y escritura para el almacenamiento de los resultados, y se obtiene una potencial mejora en el rendimiento del diseño. Aunque hay que tener en cuenta que al particionar la matriz en múltiples más pequeñas se requieren más instancias o registros de memoria. Pero mientras el hardware lo permita, esta solución proporciona un extra de paralelismo a la capa.

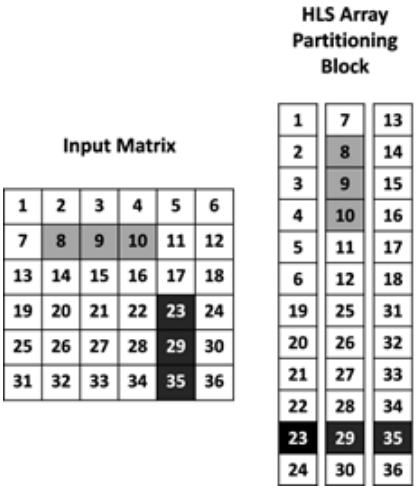


Figura 6-6 Ejemplo de aplicación del *pragma HLS ARRAY_PARTITION [43]*

6.3.2 Inserción de la red del trabajo

Una vez ya se han verificado y acelerado en una FPGA cada capa por separado que formarán la red, se ha procedido a insertar la red convolucional del trabajo en su conjunto. Pero para conseguirlo, se ha visto que la red convolucional a implementar no puede ser genérica, pues al crear los componentes, la FPGA debe saber qué tipo de datos va a utilizar, con cuáles ha de interconectarse y cuál es su tamaño. Por ello, se ha creado una interfaz, donde se definirán los parámetros de la red. De esta forma, a la hora de crear los componentes y todo lo que es necesario para crear la red , se conocerán todas las características necesarias para llevarlo a cabo; y al ser mediante una interfaz, si se desea cambiar de tamaño alguna capa o modificar la red, es más

sencillo; pues aparte de añadir las capas que se quiera, únicamente habrá que modificar esta interfaz y así no hará falta volver a crear el código de cada componente, aunque sí que habría que volver a sintetizar el código para crear la nueva implementación.

También se ha visto la necesidad de definir, mediante un pragma HLS, las salidas y entradas de la red. Para ello, se ha hecho uso del *#pragma HLS INTERFACE*, donde las entradas y salidas del diseño se definen a través de una interfaz de entrada y salida del dispositivo. De esta forma, se han creado 3 interfaces: una de control *(#pragma HLS INTERFACE s_axilite port=return bundle = ctrl_bus*), donde se usa el modo *s_auxilie*, el cual implementa los puertos como una interfaz AXI4-LITE, pues Vivado HLS producirá un conjunto asociado de archivos de controlador C durante el proceso de exportación RTL. Otra, donde se definen las entradas (*#pragma HLS INTERFACE m_axi Depth = 784 port = imagen bundle=src_bus*), donde se expone el tamaño de la entrada al dispositivo; en este caso, la imagen a clasificar tiene un tamaño de 28x28 = 784 píxeles. Y, por último, una interfaz que define la salida de la red *(#pragma HLS INTERFACE m_axi depth=10 port=class bundle=dst_bus*), donde se especifica el tamaño de la salida; en el caso de nuestra red será de 10 pues son las posibilidades que puede ser clasificada la imagen. Estas últimas usarán el modo m_axi, la cual implementará todos los puertos como una interfaz AXI4-Stream.

Una vez se ha realizado la implementación con todas las especificaciones que se ha dicho, se ha obtenido la fFigura 6-7, en la que se puede apreciar donde se encuentra el área de la FPGA que forma la red convolucional en su conjunto, pues esta área esta pintada de azul. Además, también se observa que, al elegir una FPGA lo suficientemente grande, la red se ha podido implementar sin problemas y cómo aumenta su complejidad; pues el área desocupada aún es bastante grande. Además, en la Tabla 1 que se muestra, se puede observar de forma numérica cuantos bloques de RAM, CLB, LUTs, etc, han sido necesarios para implantar la red en la FPGA y puesto que la mayoría de los recursos quedan más del 90 % libres es posible además de implantar la FPGA poder acelerar otro algoritmo en la misma FPGA, teniendo a su vez dos algoritmos diferentes en el mismo hardware. Esto no sería ningún problema pues quedan libres la mayoría de las señales de reloj. Además, en concreto cabe resaltar la frecuencia a la que sea conseguido implantar la red convolucional es de 99,010 MHz.

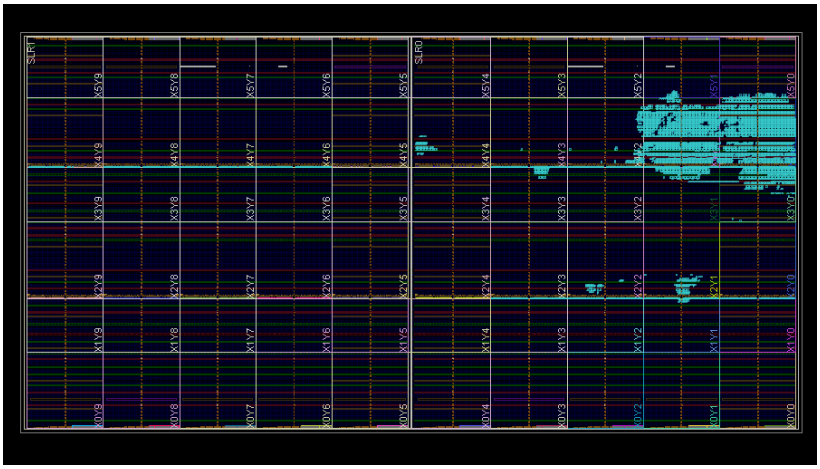


Figura 6-7 Representación del área ocupada por la red convolucional del trabajo dentro de la FPGA.

Hardware utilizado en la FPGA				
	Nº TOTAL	Nº UTILIZADAS	% UTILIZADO	% LIBRE
CLB LUTs	600577	17335	2,89%	97,11%
CLB Registers	1201154	18156	1,51%	98,49%
CARRY8	98520	585	0,59%	99,41%
F7 Muxes	394080	1535	0,39%	99,61%
F8 Muxes	197040	300	0,15%	99,85%
CLB	98520	3869	3,93%	96,07%
LUT as Logic	600577	15863	2,64%	97,36%
LUT as Memory	394560	1472	0,37%	99,63%
Block RAM	1024	30,5	2,98%	97,02%
DSPs	3474	80	2,30%	97,70%
Bonded IOB	832	598	71,88%	28,13%
HPIOBS_M	384	278	72,40%	27,60%
HPIOBS_S	384	277	72,14%	27,86%
HPIOB_SNGL	64	43	67,19%	32,81%
GLOBAL CLOCK BUFFERs	1200	1	0,08%	99,92%

Tabla 1 Representación del hardware utilizado en la FPGA

También, al realizar la implementación mediante la herramienta Vivado [44], se ha podido hacer una estimación del consumo de la red del trabajo al acelerarla en su totalidad dentro de la FPGA. Esta estimación, se puede apreciar en la Figura 6-8, donde se puede apreciar cuanta energía costará cada parte de la FPGA que es necesaria para crear la red (señales, lógica, BRAM, etc.), al igual que se ve la energía total que se necesitará para alimentar la FPGA para que pueda funcionar la red del trabajo. Como se aprecia, la energía que necesita no es mucha, lo que respalda el hecho de que una de las tendencias actuales sea el intentar acelerar las redes convolucionales en estos dispositivos, pues se ha visto que sin perder precisión en la red se puede realizar su trabajo con un coste muy inferior a cualquier GPU.

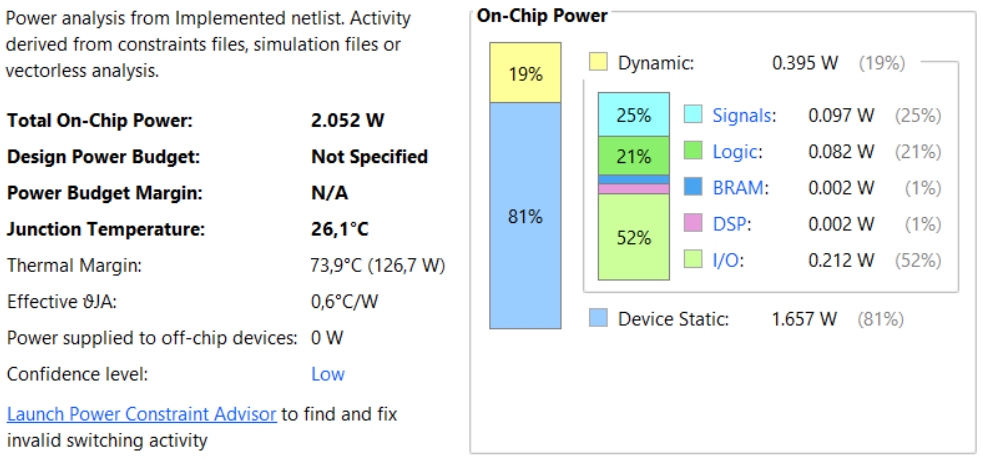


Figura 6-8 Estimación energética de la red convolucional del trabajo dentro de la FPGA

6.4 Comparación de las ejecuciones de una red convolucional sobre diferentes equipos

En esta sección, se va a realizar la comparación, tanto de las capas por separado de la red, como de ella en diferentes equipos. Se ha decidido analizar las capas por separado, debido a que hay que comprobar que en el caso de que la red no cupiera en una FPGA, sería interesante ver qué capas si sería más recomendable realizarlas dentro de un acelerador. En el caso de nuestra red, se ha buscado una FPGA lo suficiente grande para que esto no suceda y sí se pueda implementar toda la red dentro de ella. Pero se ha decidido conocer también esta otra posibilidad, pues es posible que, al implementar redes más grandes, estas sean difíciles de implementar en un FPGA convencional. Primero se verán cada capa por separado y se analizarán los resultados, y por último se analizará la aceleración de la red convolucional del trabajo.

Para llevar a cabo las ejecuciones sobre los diferentes equipos, se han seleccionado los siguientes 3 dispositivos: una ejecución se realizará sobre un I7 de 11th generación a 3000 MHz, otra se realizará sobre una FPGA en concreto (para asegurar que la red quepa sin problema se utilizará la FPGA xcvu5p-flva2104-1-e [27] con el diseño funcionando a 99,010 MHz) y, por último, también se implementará la red en un ARM Cortex-A9 MPCore a 666MHz.

6.4.1 Comparación de la capa convolucional

La primera capa que se ha implementado sobre diferente hardware es la capa de convolución, ya que es la primera que debe ejecutar la red del trabajo. Los resultados obtenidos en sus diferentes implementaciones sobre los diferentes quipos se muestran en la Figura 6-9, pues se puede apreciar de forma rápida y concisa cómo la ejecución sobre el ARM es aproximadamente 438 veces más lenta que si se implementara la capa sobre el acelerador de hardware. Aunque hay que resaltar que, si se compara la ejecución del ARM con la del I7, esta resulta ser 10 veces mejor.

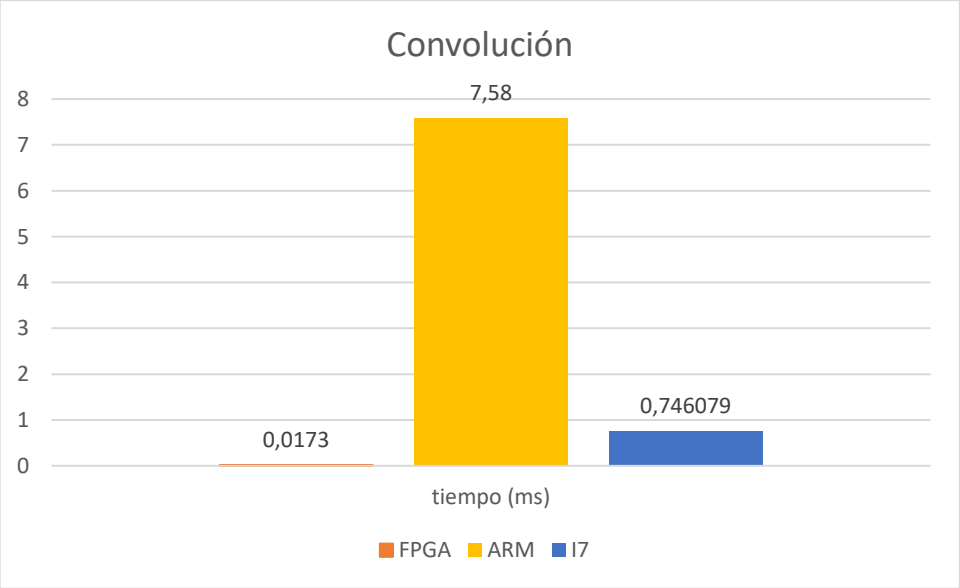


Figura 6-9 Gráfica resultado de las diferentes ejecuciones de la capa de convolución sobre diferente hardware

Por lo que se puede sacar como conclusión de esta capa que, sin lugar a dudas, una ejecución sobre un sistema como el ARM es una mala decisión, pues a la vista queda que hay dos soluciones claramente mejores. Pero de estas dos soluciones, hay que resaltar también que de manera

bastante significativa una resalta sobre la otra; pues como se demuestra en la Figura 6-9 la FPGA en comparación con la ejecución sobre el I7 es 43 veces más rápida. Por lo tanto, se cumple que, como cabria de esperar, vale la pena acelerar esta capa, pues obteniendo siempre la misma salida, el tiempo se reduce de forma considerable.

6.4.2 Comparación de la capa de activación

La capa de activación también se ha querido probar a implementarla por separado, pues, aunque el trabajo que realiza no es muy complicado, sí se ha querido comprobar si las directivas que se han puesto para aumentar su paralelismo (apartado 6.3.1) han mejorado su implementación. El resultado de esto se puede ver en la Figura 6-10; en la cual, se ve como su implementación en la FPGA mejora considerablemente el tiempo con respecto al resto de equipos donde se ha ejecutado esta misma capa. Obteniendo como resultado en la FPGA hasta 30 veces mejor tiempo que si se implementara sobre el ARM, y siendo a su vez 5 veces más rápido que si se implementara sobre el I7.

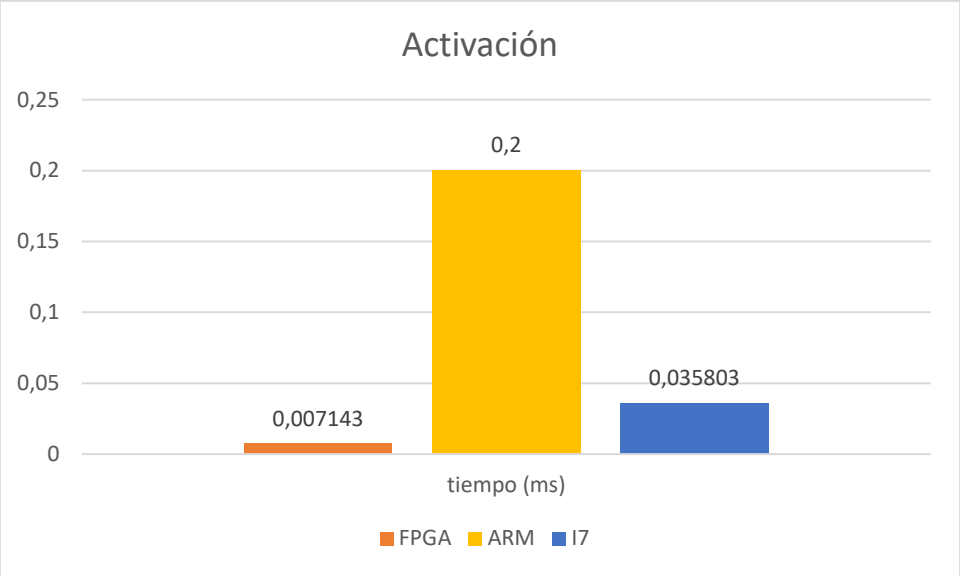


Figura 6-10 Gráfica comparativa de hardware de la capa de activación

Por lo tanto, se puede obtener como conclusión, viendo la Figura 6-10, que efectivamente merece la pena acelerar esta capa dentro de una FPGA; puesto que el tiempo de respuesta es mucho mejor que en comparación con el resto del hardware utilizado. Además, que efectivamente se demuestra que las directivas implementadas también han facilitado que se obtenga este resultado.

6.4.3 comparación de la capa de agrupación

Al implementar la capa de agrupación sobre diferentes dispositivos se ha visto que, efectivamente, al igual que ha pasado con las dos capas anteriores expuestas en las secciones 6.4.1 y 6.4.2, el resultado obtenido es el esperado. Esto se puede ver en la Figura 6-11, pues claramente se puede observar cómo al implementar esta capa en el acelerador hardware, es considerablemente más rentable que en cualquiera de los otros dos equipos. Esto se debe a que, como se puede deducir, a través de la Figura 6-11,la ejecución mediante la FPGA es aproximadamente 120 veces más rápida que mediante la ejecución con el ARM. Esto también sucede si comparamos la FPGA con

la ejecución sobre el I7, pues se ve que esta es aproximadamente 18 veces peor que si se ejecuta la capa sobre la FPGA.

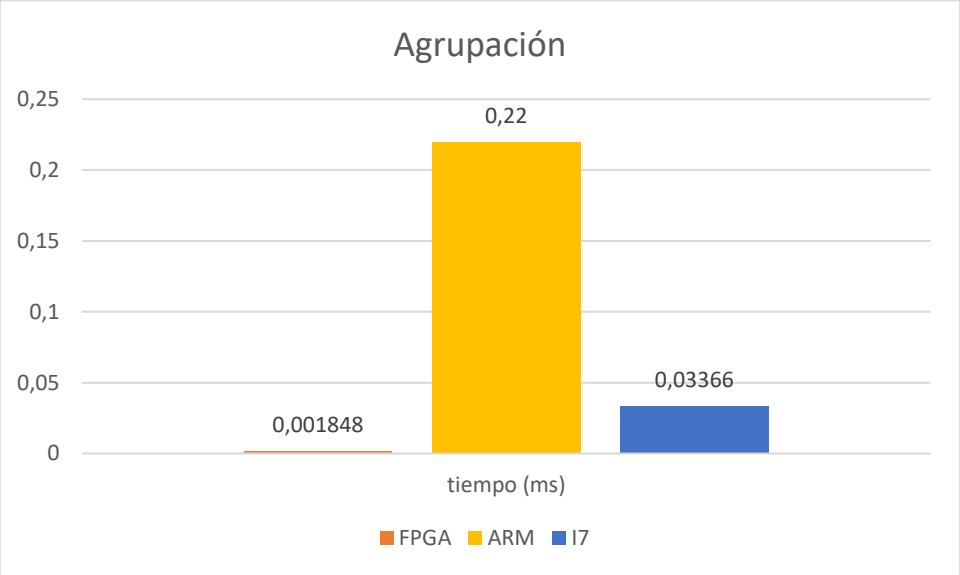


Figura 6-11 Implementación de la capa de agrupación sobre diferentes dispositivos.

6.4.4 Comparación de la capa completamente conectada

Al igual que las capas anteriores, se ha realizado la implementación individual de la capa completamente conectada en los tres equipos hardware seleccionados. Con los resultados obtenidos de las 3 ejecuciones, se ha realizado una gráfica para que sea más sencillo visualizar los resultados, esta se puede ver en la Figura 6-12. Analizando detenidamente la Figura 6-12, se pueden obtener ciertas conclusiones muy interesantes, pues esta capa, al igual que las capas anteriores, al ejecutarla en un sistema empotrado como un ARM, se ve que obtiene peores resultados que si se ejecuta en cualquiera de los otros hardware elegidos para el estudio. Obteniendo que en comparación con una FPGA la ejecución es el doble de lenta, y si se compara con una ejecución sobre un I7, se obtiene que es 5 veces peor. Por lo que se puede decir que no es conveniente usar un sistema empotrado como los actuales.

Pero a diferencia del resto de las capas de la red neuronal convolucional, esta capa es la única que obtiene mejores resultados implementándola por software que mediante un acelerador hardware, pues de esta forma es aproximadamente 2,5 veces más rápido. Por lo que, a la vista de los resultados, se planteará, si al implementar la red al completo, es rentable implementar en un acelerador hardware esta capa; pues a la vista está que, aunque sean unas directivas para aumentar en la medida de lo posible el paralelismo y conseguir una correcta implementación en el hardware, en esta capa, al ser de naturaleza muy secuencial, no se han obtenido los resultados esperados, dando como resultado, el que se plantee implementar esta capa por software.

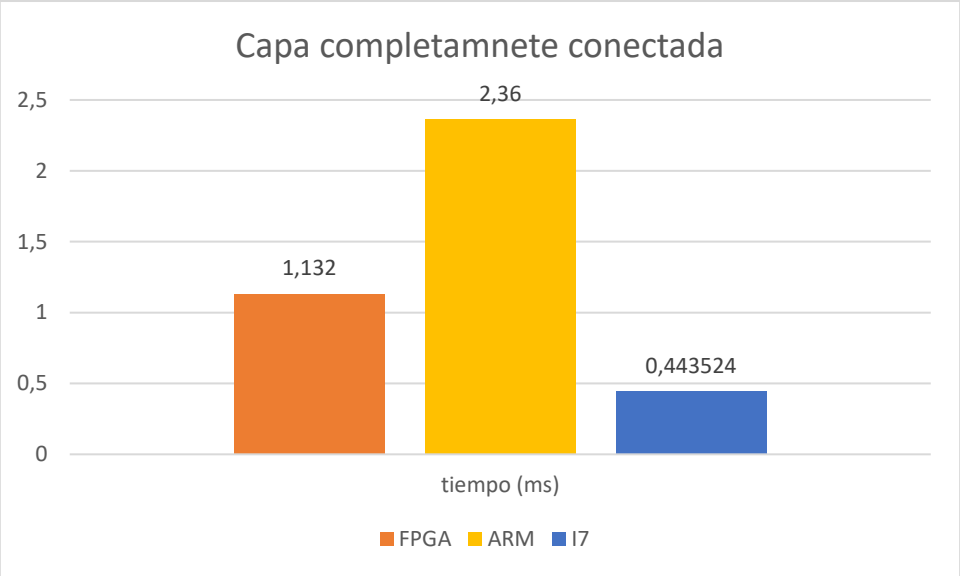


Figura 6-12 Gráfica comparativa de las diferentes ejecuciones en distinto hardware de la capa completamente conectada.

6.4.5 Comparación de la red neuronal convolucional del trabajo

A la hora de implementar la red del trabajo, se ha decido comparar su tiempo de obtención de los resultados de 4 maneras diferentes, para así ver cuál de todas las implementaciones de la red obtiene mejores resultados. Para esto, se ha creado un gráfico, que se muestra en la Figura 6-13, con los datos obtenidos, y así poder analizar de la mejor manera posible los resultados.

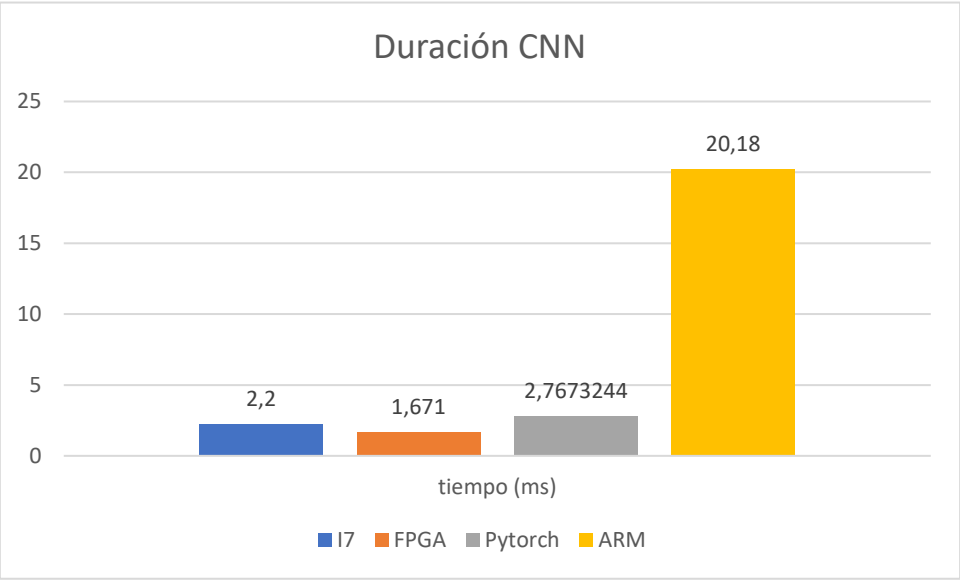


Figura 6-13 Ejecución de la red neuronal convolucional del trabajo sobre diferentes plataformas.

Una vez implementada la misma red convolucional que se ha descrito sobre diferentes plataformas, se ven unos resultados muy interesantes y otros muy esperados. Para empezar, se puede observar cómo cambia mediante la ejecución de la red sobre el ARM, pues en comparación a la ejecución mediante el firmware de entrenamiento seleccionado, es 7 veces peor; y al igual que las capas por separado, se puede ver cómo esta es la peor solución.

Sin embargo, si comparamos las ejecuciones entre el firmware de entrenamiento, la implementación en sobre el I7 y la implementación en la FPGA, se ve claramente cómo los resultados difieren muy poco. Esto es interesante puesto que al analizar las capas por separado se veía como todas las capas, salvo la última (la capa de completamente conectada), se obtenían mucho mejores resultados al acelerarlas; pero al implementar toda la red, incluida la última capa dentro del acelerador, se ha obtenido resultados similares a que si se realizara toda la implementación sobre el I7.

Todo esto parece indicar que las primeras etapas de la red neuronal al ser altamente paralelizables, deberían implantarse en una FPGA, mientras que la etapa de *fully connected* tiene un comportamiento más secuencial, por lo que sería más apta para su implementación realizarla en un procesador de propósito general. Aunque, al insertar la red en su conjunto, se ve cómo lo que se gana de tiempo en el resto de las capas, compensa el poder implementar esta última dentro del acelerador hardware, permitiendo exportar la red al completo a una FPGA; obteniendo unas mejoras, tanto en el tiempo de respuesta, sin sacrificar precisión en la red, como en el componente energético, puesto que los aceleradores están preparados para poder realizar esto.

Aunque sería interesante plantearse a raíz de los resultados como trabajo futuro, la posibilidad de utilizar dispositivos de la familia ZYNQ de Xilinx, las cuales integran un núcleo de procesador ARM junto a la lógica programable. Con ello, podría implementarse las primeras etapas en la FPGA y la *fully connected* en el ARM que se encuentra dentro del mismo chip, reduciendo la latencia de comunicación entre ambos componentes y maximizando los beneficios de ambos componentes.

6.5 Otros entornos de desarrollo

Para esta sección del trabajo se ha decidido probar otras formas para poder implementar redes convolucionales dentro de aceleradores hardware. Como el fabricante con el que se decidió trabajar (Xilinx) pose un entorno propio para el desarrollo de redes neuronales convolucionales y su implementación en hardware, se ha decidido probar su funcionamiento y su sencillez, tanto a la hora de programar las redes convolucionales como a la hora de familiarizarse con el entorno. De esta forma, se podría comprobar cómo simplifican el trabajo con respecto a la sección anterior, además de ver qué nivel de experiencia se ha de tener con el tema en cuestión, cómo de intuitivo es, y qué ventajas ofrece.

6.5.1 Vitis AI

Al analizar el entorno de desarrollo Vitis AI, que Xilinx ha desarrollado especialmente para implementar redes neuronales convolucionales, se ha visto que, esta plataforma está formada por herramientas, bibliotecas, modelos y diseños de ejemplo, principalmente. Además, está diseñado con una alta eficiencia y usabilidad, de forma que pueda liberar todo el potencial de los aceleradores hardware de inteligencia artificial IA de Xilinx en FPGA y ACAP.

La estructura básica de este entorno se muestra en la Figura 6-14, donde se aprecia como contiene compatibilidad con frameworks para la creación y entrenamiento de redes convolucionales, pues admite a los principales, como Caffe, Pytorch o Tensorflow. Permitiendo que sin ser un experto en cómo funciona y cómo se crean las redes neuronales internamente le ofrece al desarrollador la forma de crear y entrenar la red dentro del entorno de desarrollo específico como los mencionados, y a través de unos sencillos pasos. Además, traduce automáticamente esa red entrenada y creada en un framework específico, en un archivo que una FPGA pueda entender e

implementar. Por lo que primero se importa un modelo de red desde cualquier framework de los mencionados, para luego optimizarla eliminando las conexiones redundantes (esto solo será posible realizarlo si se tiene la versión de pago), y también reducirá las operaciones a las necesarias; además de evaluar la efectividad del modelo importado. Una vez el modelo ya está compilado, el entorno pasa a identificar los componentes necesarios para poder implementarlo, obtiene su rendimiento, los puertos de entrada y salida necesarios y los requisitos de transferencia de datos.

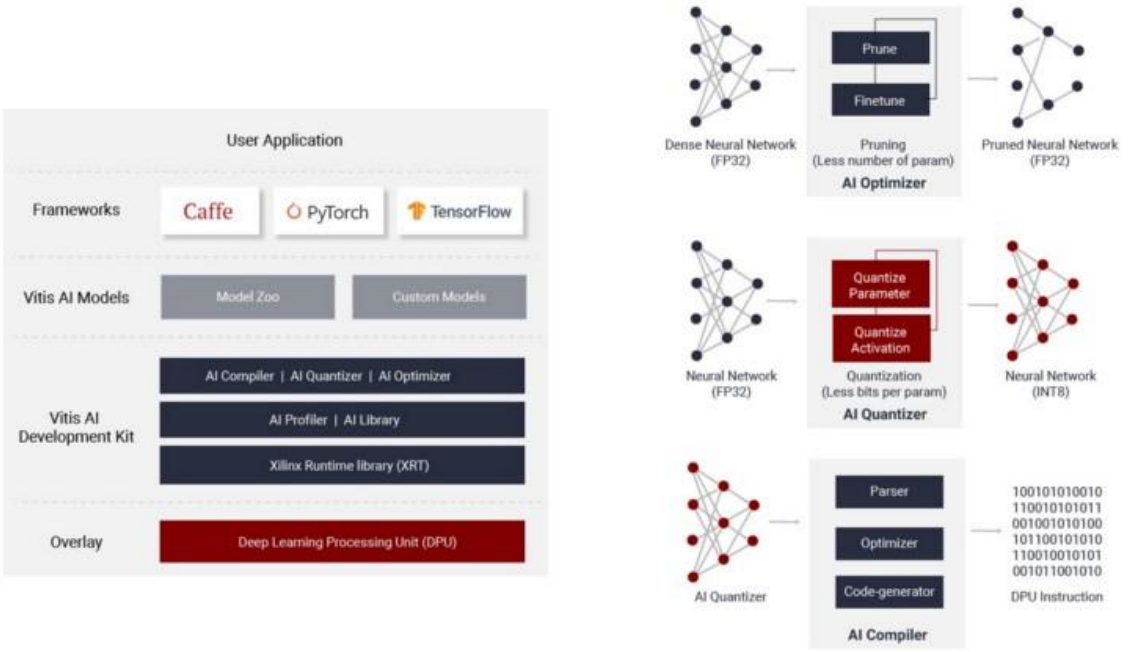


Figura 6-14 Estructura básica de cómo funciona Vitis AI *[45]*

Este entorno permite realizar un particionado, es decir, un proceso que permite dividir la ejecución de un modelo de red neuronal entre la FPGA y el host. Esto es necesario a veces, pues algunos modelos contienen capas que no son admitidas por una FPGA y esa sección de la red debe ser realizada por software. Por otro lado, el particionado también puede ser de utilidad para depurar y explorar diferentes ejecuciones y poder cumplir con un objetivo.

Para probar este entorno, se ha desarrollado una red de tres etapas convolucionales y se ha decidido utilizar Tensorflow para entrenarla, puesto que este framework es uno de los más sencillos de utilizar a la hora de empezar con las redes neuronales. Eso se debe a que Tensorflow ofrece varios niveles de abstracción, por lo que se acopla tanto a las necesidades y a los conocimientos de cualquier desarrollador.

6.5.2 Tensorflow

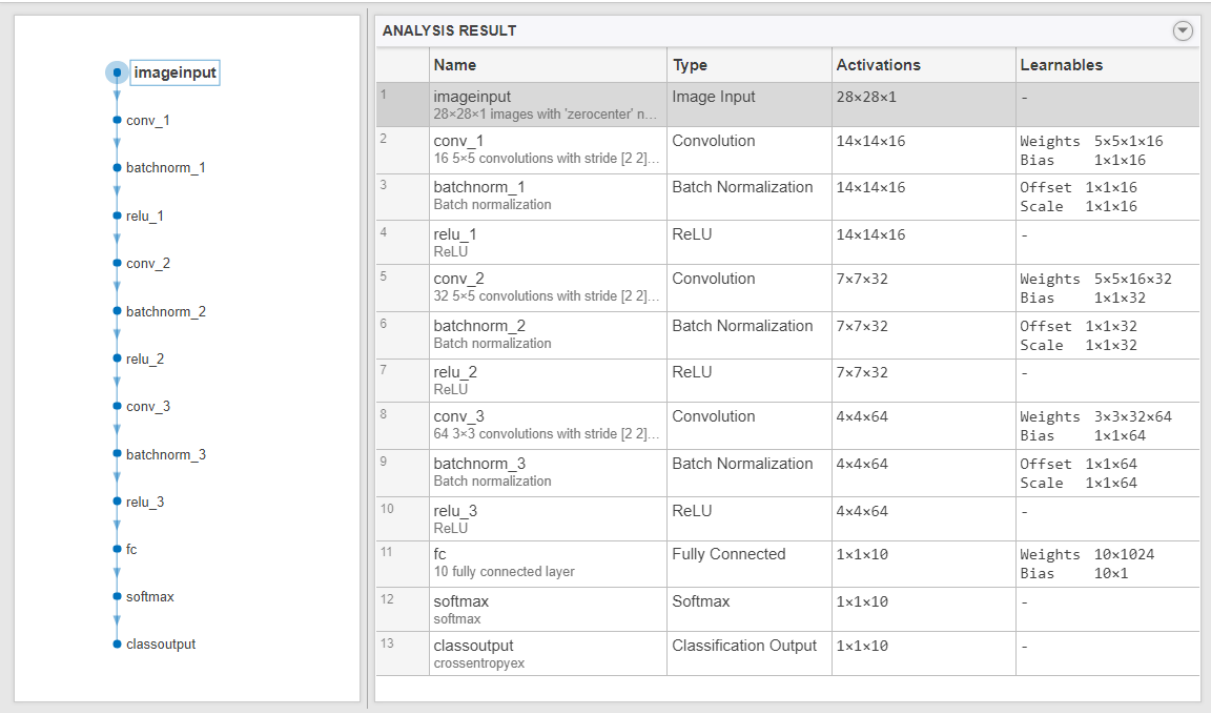
Este framework permite compilar y entrenar modelos mediante la API de alto nivel Keras [46], la cual ayuda a que los primeros pasos con TensorFlow y el aprendizaje automático sean más sencillos. Además, si se necesita una mayor flexibilidad, la ejecución inmediata permite realizar iteración inmediata y depuración intuitiva; y para grandes redes, permite diferentes estrategias, es decir, permite probar diferentes configuraciones de hardware sin cambiar la definición del modelo.

Este entorno ofrece una amplia documentación con ejemplos, en función del nivel del desarrollador, y permite la creación de gráficas e imágenes del entrenamiento y del modelo de la red convolucional para que sea más sencillo tanto interpretar los datos como entender el modelo y su funcionamiento. Además, ofrece bibliotecas que dan acceso a bancos de datos para poder realizar el entrenamiento de forma sencilla, pues se ha decido que como hasta ahora se han utilizado el banco de entrenamiento de NMIST, se continuará usándolo por las razones que se explicaron en su momento.

6.5.3 Implementación de la red convolucional para el entorno de Vitis AI

Una vez seleccionado el framework de entrenamiento, el entorno de desarrollo y el modelo de la red convolucional, como se ha dicho antes, se va a proceder a ver cómo Vitis AI interconecta todos los elementos de forma sencilla.

Para esta sección en concreto, se hará uso de una red algo más grande que la implementada mediante Vitis HLS, pero, aun así, será sencilla para de esta forma poder centrarse más en el entorno que en la red misma. Esta red convolucional, contará de una primera etapa de convolución, donde se le aplicarán 16 filtros diferentes de tamaño 5x5 a una imagen de entrada de tamaño 28x28x1, seguido por una capa de activación Relu. Después, a los resultados obtenidos se les aplicará otra capa de convolución de 32 filtros de tamaño 5x5 y al igual que la anterior, irá seguida de una capa de activación Relu. La siguiente volverá a ser otra capa de convolución de 64 filtros, pero esta vez el tamaño de los filtros será de 3x3, y a sus resultados se le aplicará otra capa de activación Relu. Y, por último, se le aplicará la etapa de completamente conectada o *fully connected* y la capa de clasificación. Esta estructura de la red se puede observar en la siguiente Figura 6-15.



ANALYSIS RESULT

	Name	Type	Activations	Learnables
1	imageinput 28×28×1 images with 'zerocenter' n...	Image Input	28×28×1	-
2	conv_1 16 5×5 convolutions with stride [2 2]...	Convolution	14×14×16	Weights 5×5×1×16 Bias 1×1×16
3	batchnorm_1 Batch normalization	Batch Normalization	14×14×16	Offset 1×1×16 Scale 1×1×16
4	relu_1 ReLU	ReLU	14×14×16	-
5	conv_2 32 5×5 convolutions with stride [2 2]...	Convolution	7×7×32	Weights 5×5×16×32 Bias 1×1×32
6	batchnorm_2 Batch normalization	Batch Normalization	7×7×32	Offset 1×1×32 Scale 1×1×32
7	relu_2 ReLU	ReLU	7×7×32	-
8	conv_3 64 3×3 convolutions with stride [2 2]...	Convolution	4×4×64	Weights 3×3×32×64 Bias 1×1×64
9	batchnorm_3 Batch normalization	Batch Normalization	4×4×64	Offset 1×1×64 Scale 1×1×64
10	relu_3 ReLU	ReLU	4×4×64	-
11	fc 10 fully connected layer	Fully Connected	1×1×10	Weights 10×1024 Bias 10×1
12	softmax softmax	Softmax	1×1×10	-
13	classoutput crossentropyex	Classification Output	1×1×10	-

Figura 6-15 Estructura de la red que se ha creado para esta sección del trabajo, además se puede ver el tamaño de cada una de ellas, así como el tamaño y número de características que se aplican.

Con esta sencilla red convolucional, se obtiene un 98,14 % de precisión. Por lo que podemos deducir que es una buena red para intentar implementarla, dentro de un hardware como un FPGA.

Una vez ya se tiene la red entrenada y evaluada, se guardará el modelo entrenado como un grafo de incidencia y los pesos generados se guardan en otro fichero. Este proceso de inferencia requiere una gran cantidad de cálculos, y un gran ancho de banda de memoria, para así poder satisfacer los requisitos de baja latencia y alto rendimiento que requieren este tipo de aplicaciones. Para ellos se emplean técnicas de cuantificación y poda de canales.

A continuación, es necesario convertir en gráfico de incidencia y los pesos generados en un único archivo, este es conocido como un "gráfico congelado", puesto que todas las variables se convierten en constantes y los nodos del grafo asociados con el entrenamiento, el optimizador y las funciones de pérdida serán eliminadas. Una vez en este punto, se pueden comparar la efectividad del nuevo grafo generado, es decir, el "gráfico congelado" y los resultados obtenidos después del entrenamiento, y así comprobar que efectivamente los resultados obtenidos, aplicando el nuevo gráfico, son muy similares a los obtenidos con el entrenamiento.

En este punto, se continuará convirtiendo el "grafo congelado" de punto flotante a punto fijo. Esto se realizará gracias a la cuantificación, ya que permite utilizar unidades de cálculo de números enteros y representar pesos con menor número de bits. En este caso en concreto, los pesos se encuentran una vez salidos del entrenamiento, en punto flotante de 32 bits y una vez aplicada la cuantificación se convertirán a un formato de entero de 8 bits. Este cuantificador que, nos proporciona Vitis AI, permite reducir la complejidad informática sin perder la precisión de la predicción, ya que el modelo en punto fijo requiere mucho menos ancho de banda de memoria, y como consecuencia se obtendrá una mayor velocidad y eficiencia energética que el modelo en punto flotante.

Al igual que antes, ahora se puede comprobar como el "grafo congelado cuantificado" continúa teniendo una precisión muy similar a la que se obtuvo en el entrenamiento, siendo que ahora los datos ya no están en punto flotante, sino en entero de 8 bits.

Por último, sólo queda compilar el modelo cuantificado en un archivo .xmodel para la placa de evaluación deseada. Vitis AI proporciona una serie de bibliotecas con las diversas configuraciones para realizar este paso con total abstracción. Pero en esta biblioteca, sólo se encuentran las placas más adecuadas para realizar trabajos relacionados con las redes convolucionales. El proveedor Xilinx únicamente oferta la posibilidad de realizar la implementación en esas placas porque ellas contienen un procesador sistólico y lo que hace Vitis AI es obtener el código máquina del modelo creado para poder ejecutar la red. Si se dispone de una FPGA de la serie Versal, entonces el procesador sistólico esta implementado en silicio junto con la lógica programable, con lo que puede acelerarse aún más la ejecución de estas redes. Pero para este trabajo dado, no tenemos ninguna de esas FPGAs a nuestra disposición, por lo que, no se ha podido implementar y obtener los resultados para analizarlos. Pero sí que se ha podido comprobar cómo funciona en entorno.

Por lo que se puede concluir, que el entorno Vitis AI ofrece una alta abstracción para que el desarrollador pueda realizar este tipo de trabajos ,únicamente necesitando saber utilizar los principales framework de entrenamiento, y comprobando que se tiene un hardware compatible con las configuraciones que Vitis AI ofrece en su biblioteca para la implementación de la red; pues si el desarrollador ha de hacer el archivo de implementación, no sólo ha de tener un alto nivel en la programación a bajo nivel, sino además saber cómo está configurada si placa para poder activar todas las vías y señales necesarias.

7. Conclusiones y trabajo futuro

En este trabajo, se ha hecho uso de la herramienta de generación de VHDL,Vitis HLS, para poder realizar la implementación de la red neuronal diseñada y entrenada a lo largo del proyecto. Esta herramienta, desarrollada por Xilinx, ha facilitado el proceso de aceleración de la red neuronal convolucional en hardware, puesto que la programación de la red se ha podido realizar en un lenguaje de nivel medio, permitiendo al desarrollador, centrarse en la funcionalidad de las capas; y no en cómo sería la codificación en un lenguaje de descripción de hardware. Además de ofrecer diferentes tipos de directivas para facilitar y modificar la configuración de la implementación del algoritmo en una FPGA. La herramienta, ofrece una interfaz gráfica de usuario, a través la cual permite al desarrollador describir el código que crea el algoritmo, al igual que describir parámetros para restringir su forma de implementación creando a su vez restricciones y requisitos para llevarla a cabo. Vitis HLS, reduce el tiempo de desarrollo necesario para lograr acelerar una red neuronal o únicamente algunas de sus capas, por lo que supera las barreras introducidas por la complejidad en el desarrollo usando lenguajes de descripción de hardware. Por lo que se puede decir, que se ha cumplido el objetivo de encontrar entornos aptos para poder desarrollar redes neuronales en aceleradores hardware sin ser expertos en su codificación.

En lo referente a la implementación de las capas de la red y de la red del trabajo en su conjunto, se ha podido apreciar a lo largo del capítulo 6, como sí se ha podido acelerar cada una de las capas por separado, obteniendo resultados bastante satisfactorios. Además, también se ha visto que, en lo referente a la capa de completamente conectada, esta es la única que no se ha podido acelerar debido a su naturaleza secuencial; pero sí que se ha apreciado, en concreto, en la red del trabajo, que la aceleración del resto de capas compensa la pérdida de esta; pudiendo así implementar toda la red dentro de la FPGA. Pero cabe resaltar, que la red expuesta en el proyecto es sencilla y pequeña, por lo que muy probablemente si se realizara una red considerablemente más grande y compleja, seguramente la capa de completamente conectada no valdría la pena implementarla dentro de la FPGA.

Dado que la tendencia de futuro es cada vez acelerar redes más complejas, se están desarrollando entornos específicos para esto como Vitis AI. Pues estos entornos, permiten interactuar con los frameworks de entrenamiento, y crear ahí los modelos de red y exportarlos al entorno. De forma, mediante unos sencillos pasos, completamente transparente para el desarrollador, se puede traducir el modelo a una codificación que entienda la FPGA. Además, al ser entornos específicamente diseñados para esto, implementarán correctamente cada una de las capas de la red sin necesidad de que el desarrollador interfiera, e incluso maximizarán en la medida de lo posible las capas que son difíciles de acelerar, como la completamente conectada.

Dado la cantidad de aplicaciones en las que se puede usar las redes convolucionales y la cantidad de entornos y herramientas que los principales fabricantes están desarrollando para poder acelerarlas, se puede ver claramente, como la tendencia de futuro es, claramente implementar las redes cada vez más complicadas en estos tipos de aceleradores hardware para poder crear sistemas empotrados cada vez más robustos y complejos, mejorando su respuesta y siendo energéticamente eficientes.

A partir del desarrollo realizado en el proyecto y de los conocimientos adquiridos a lo largo de este, se ha visto que sería interesante estudiar otras capas se podrían incluir a la red como la de normalización, u otras funciones de la capa de activación y ver funcionarían, que aportarían a la red, de cómo se implementarían, si valdría la pena su aceleración, etc. También sería interesante como se dijo en el apartado 6.4.5, estudiar la implementación de la etapa de completamente conectada en dispositivos de la familia ZYNQ de Xilinx y así ver como funcionaria y como resultaría la implementación de la red convolucional, donde el ARM se encargaría de esta capa y el resto de la red se encargaría de acelerarla la FPGA. Otro estudio interesante que se podría realizar a partir del proyecto realizado es intentar ejecutar de forma paralela diferentes imágenes de forma simultánea y ver cuantas ejecuciones paralelas podría realizar la red y ver si para esto sería necesario replicar la red dentro de la misma FPGA.

7.1 Asignaturas del grado que han ayudado a desarrollar el proyecto

A lo largo del grado se han visto muchas asignaturas en las que se ha enseñado a programar o implementar redes entre otras cosas. Pero una signatura que, sí que ha contribuido de forma activa para poder realizar este proyecto ha sido, Diseño de sistemas digitales o DSD. En esta asignatura se estudió como está el mundo de los aceleradores actualmente, y se enseñó a describir hardware. Pero principalmente y para lo que más ha contribuido su formación al proyecto es que enseña qué es una FPGA y cómo trabajar con ella. Esto ha sido clave a la hora de elegir el hardware para acelerar la red convolucional del trabajo, pues a parte de las ventajas que se han ido exponiendo a lo largo del trabajo, también ha contribuido el que se tenga una experiencia con ellas, por lo que desde un primer momento se ha podido enfocar de una forma óptima la implementación del algoritmo en ellas.

Referencias

[1] Wikipedia, «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico. [Último acceso: 4 9 2021].

[2] healt big data, «healt big data,» [En línea]. Available: https://www.juanbarrios.com/redes-neurales-convolucionales/. [Último acceso: 4 9 2021].

[3] J. I. Bagnato, «Aprende Machine Learning,» 29 11 2018. [En línea]. Available: https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/. [Último acceso: 4 9 2021].

[4] wikipedia, «wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Red_neuronal_convolucional. [Último acceso: 4 9 2021].

[5] soldai, «soldai,» 22 6 2020. [En línea]. Available: https://soldai.com/redes-neuronales-convolucionales/. [Último acceso: 4 9 2021].

[6] wikipedia, «wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Inteligencia_artificial_explicable. [Último acceso: 4 9 2021].

[7] Y. FERNÁNDEZ, «Xataka,» 17 2 2021. [En línea]. Available: https://www.xataka.com/basics/tarjeta-grafica-que-que-hay-dentro-como-funciona. [Último acceso: 4 9 2021].

[8] wikipedia, «wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Unidad_de_procesamiento_gr%C3%A1fico#:~:text=Una%20unidad%20de%20procesamiento%20gr%C3%A1fico,videojuegos%20o%20aplicaciones%203D%20interactivas. [Último acceso: 4 9 2021].

[9] M. Gracia, «Deloitte,» [En línea]. Available: https://www2.deloitte.com/es/es/pages/technology/articles/IoT-internet-of-things.html. [Último acceso: 4 9 2021].

[10] sas, «sas,» [En línea]. Available: https://www.sas.com/es_es/insights/analytics/deep-learning.html. [Último acceso: 4 9 2021].

[11] R. ARRABALES, «Xataka,» 28 9 2016. [En línea]. Available: https://www.xataka.com/robotica-e-ia/deep-learning-que-es-y-por-que-va-a-ser-una-tecnologia-clave-en-el-futuro-de-la-inteligencia-artificial. [Último acceso: 4 9 2021].

[12] Y. LeCun, «atcold,» [En línea]. Available: https://atcold.github.io/pytorch-Deep-Learning/es/week06/06-1/. [Último acceso: 4 9 2021].

[13] tensorflow, «tensorflow,» [En línea]. Available: https://www.tensorflow.org/. [Último acceso: 4 9 2021].

[14] pytorch, «pytorch,» [En línea]. Available: https://pytorch.org/. [Último acceso: 4 9 2021].

[15] Y. Jia, «Caffe,» [En línea]. Available: https://caffe.berkeleyvision.org/. [Último acceso: 4 9 2021].

[16] J. D. A. Ilber Adonayt Ruge Ruge, «scielo,» 22 11 2012. [En línea]. Available: http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S0123-921X2013000200008. [Último acceso: 4 9 2021].

[17] C. T. H. Juan Carlos Moctezuma Eugenio, «ResearchGate,» [En línea]. Available: https://www.researchgate.net/publication/255638595_ESTUDIO_SOBRE_LA_IMPLEMENTACION_DE_REDES_NEURONALES_ARTIFICIALES_USANDO_XILINX_SYSTEM_GENERATOR/link/53da08ca0cf2e38c63364f30/download. [Último acceso: 4 9 2021].

[18] DREAM-A team of ISPR, «DREAM-A team of ISPR,» [En línea]. Available: https://dream.ispr-ip.fr/members/kamel-abdelouahab/haddoc2/#page-content. [Último acceso: 4 9 2021].

[19] Xilinx, «Github,» [En línea]. Available: https://github.com/Xilinx/Vitis-AI. [Último acceso: 4 9 2021].

[20] Intel, «Intel,» [En línea]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html. [Último acceso: 4 9 2021].

[21] Intel, «Intel,» [En línea]. Available: https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html.

[22] Genera Techologias, «Genera Techologias,» [En línea]. Available: https://www.generatecnologias.es/arquitectura_fpga.html. [Último acceso: 4 9 2021].

[23] V. M. Hernando, «ResearchGate,» [En línea]. Available: https://www.researchgate.net/figure/Figura-410-Esquema-basico-de-la-arquitectura-de-una-FPGA-Xilinx-XC4000-Xilinx-web_fig18_39562537. [Último acceso: 4 9 2021].

[24] hacedores, «hacedores,» 15 6 2019. [En línea]. Available: https://hacedores.com/breve-historia-fpga/. [Último acceso: 4 9 2021].

[25] Xilinx, User guide: 7 Series FPGAs Configurable Logic Block, 2016.

[26] wikipedia, «wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/Clock_skew. [Último acceso: 4 9 2021].

[27] Xilinx, Product Specification: UltraScale and UltraScale+ FPGAs Packaging and Pinouts, 2021.

[28] T. V. Y. G. B. S. A. C. J. M. a. D. D. Martin C. Herbordt, «US National Library of Medicine National Institutes of Health,» 1 3 2007. [En línea]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3098506/. [Último acceso: 4 9 2021].

[29] wikipedia, «wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Procesador_de_se%C3%B1ales_digitales. [Último acceso: 4 9 2021].

[30] wikipedia, «wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Neurona_de_McCulloch-Pitts. [Último acceso: 4 9 2021].

[31] Programador clic, «Programador clic,» [En línea]. Available: https://programmerclick.com/article/46491213150/. [Último acceso: 4 9 2021].

[32] Pytorch, «Pytorch,» [En línea]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html. [Último acceso: 4 9 2021].

[33] ICHI.PRO, «ICHI.PRO,» [En línea]. Available: https://ichi.pro/es/matematicas-de-las-redes-neuronales-convolucionales-306971780238 72. [Último acceso: 4 9 2021].

[34] ResearchGate, «ResearchGate,» [En línea]. Available: https://www.researchgate.net/figure/Pooling-layer-operation-oproaches-1-Pooling-layers-For-the-function-of-decreasing-the_fig4_340812216. [Último acceso: 4 9 2021].

[35] WintemplaWeb, «WintemplaWeb,» 22 7 2021. [En línea]. Available: http://sintesis.ugto.mx/WintemplaWeb/01Neural%20Lab/03Classification/04SoftMax/index.htm. [Último acceso: 4 9 2021].

[36] C. C. C. J. B. Yann LeCun, «THE MNIST DATABASE,» [En línea]. Available: http://yann.lecun.com/exdb/mnist/. [Último acceso: 4 9 2021].

[37] Xilinx, Introduction to FPGA Design with Vivado High-Level Synthesis, 2019.

[38] D. McGrath, «Xilinx buys high-level synthesis EDA vendor,» *Times,* 31 1 2011.

[39] Xilinx, Vivado Design Suite User Guide : High-Level Synthesis, 2021.

[40] LeNet-5, convolutional neural networks, «LeNet-5, convolutional neural networks,» [En línea]. Available: http://yann.lecun.com/exdb/lenet/. [Último acceso: 4 9 2021].

[41] Xilinx, «Xilinx,» 5 8 2021. [En línea]. Available: https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/creatingnewvitishlsproject.html. [Último acceso: 4 9 2021].

[42] Xilinx, UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices, 2016.

[43] L. stornaiuolo, «slideshare,» 26 5 2018. [En línea]. Available: https://www.slideshare.net/necstlab/on-how-to-efficiently-implement-deep-learning-algorithms-on-pynq-platform. [Último acceso: 4 9 2021].

[44] Xilinx, Vivado Design Suite User Guide: Getting Started, 2018.

[45] Xilinx, «Xilinx,» 23 3 2020. [En línea]. Available: https://www.xilinx.com/html_docs/vitis_ai/1_1/nvc1570695728850.html. [Último acceso: 4 9 2021].

[46] Keras, «keras,» [En línea]. Available: https://keras.io/. [Último acceso: 4 9 2021].