UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de Soporte de Procesos de Inferencia en FPGAs con High Level Synthesis

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Raül Ferrando Plà

Tutor: José Flich Cardo
Xavier Molero Prieto

Curso 2021-2022

Resumen

En los últimos años la Inteligencia Artificial (AI) se ha convertido en un elemento imprescindible en múltiples ámbitos tecnológicos. Al mismo tiempo que la IA se está desarrollando a nivel de algoritmos, también las arquitecturas de procesamiento se están adaptando para un mejor soporte de la IA. Para un mejor conocimiento de las implicaciones de las arquitecturas con los algoritmos de IA se hace imprescindible el uso de nuevas herramientas que permitan una exploración adecuada mediante el desarrollo de algoritmos óptimos a resolver por la IA.

En este proyecto se desarrollan módulos de cómputo en FPGAs para el correcto soporte de procesos de inferencia de redes neuronales. Se han desarrollado y optimizado diferentes capas de redes neuronales, como capas de pooling y funciones de activación.

Palabras clave: IA, Redes neuronales, FPGA, Xilinx, HLS

Resum

En els últims anys, la Inteligència Artificial (AI) s'ha convertit en un element imprescindible en diversos àmbits tecnològics. Al mismo tiempo que la IA s'està desenvolupant a nivell d'algorismes, també les arquitectures de processament están adaptant-se per a un millor soport de la IA. Per a un millor coneiximent de les implicacions de les arquitectures amb els algorismes de la IA resulta imprescindible el ús de noves ferramentes que permitisquen una exploració adequada mitjançant el desenvolupament d'algorismes òptims a resoldre per la IA.

En aquest projecte es desenvolupen mòduls de cómput en FPGAs per al correcte soport de processos de inferència de reds neuronals. S'han desenvolupat i optimitzat diferents capes de reds neuronals, com per exemple, capes de pooling i funcions d'activació.

Paraules clau: IA, Reds neuronals, FPGA, Xilinx, HLS

Abstract

In recent years, Artificial Intelligence (AI) has become an essential element in multiple technological fields. At the same time AI is developing at the algorithm level, processing architectures are also adapting to better support AI. For a better understanding of the implications of architectures with AI algorithms, it is essential to use new tools which allow adequate exploration through the optimum algorithm development to be resolved by AI.

In this project, computing modules are developed in FPGAs for the correct support of neural network inference processes. Different layers of neural networks have been developed and optimized, such as pooling layers and activation functions.

Key words: AI, Neural Networks, FPGA, Xilinx, HLS

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

1.1 Motivación

En las útimas décadas la tecnología ha experimentado un avance muy rápido, en especial en los últimos años con la aparición de la Inteligencia Artificial (IA), la cuál se aplica cada vez más en la rutina diaria de millones de personas.

Dentro del sector de la IA, destaca el *Machine Learning* el cuál está presente en nuestra vida cotidiana, desde las recomendaciones personalizadas de una tienda online hasta las indicaciones de una aplicación de mapas, pasando por un asistente de voz. Todos estos servicios implementan siempre un sistema de inteligencia artifical que permite ofrecer estos servicios al usuario.

Para obtener un sistema fiable de inteligencia artificial se necesita una gran cantidad de datos y el procesamiento de ellos, para ello se usan las técnicas de *Deep Learning*.

Un ejemplo muy común de aplicación de los procesos de *deep learning* es la segmentación de imágenes (figura 1.1), que permite en el campo médico la detección precoz de determinadas enfermedades.

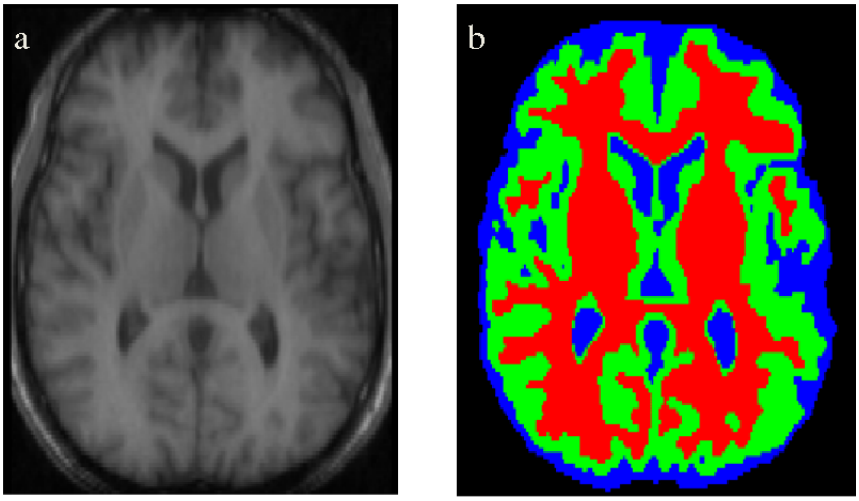

Figura 1.1: Segmentación de imagen de una radiografía de un cerebro humano

En la actualidad los algoritmos de *deep learning* se usan en multitud de aplicaciones. Estos algoritmos tienen dos tipos de procesos claramente diferenciados, la parte de entrenamiento dedicada a la creación del modelo de predicción a partir de datos de entrenamiento, como por ejemplo imágenes, y la de inferencia, referida al uso del algoritmo entrenado por la anterior etapa para así poder realizar una predicción en un entorno en producción.

Los procesos de inferencia requieren de una baja latencia en su ejecución debido a que se suelen usar en sistemas de tiempo real o si se requiere obtener determinada información al instante, es por ello la importancia de aplicar éstos procesos de inferencia en una FPGA, la cuál permite configurar su comportamiento entero pudiendo así conseguir tiempos de respuesta muy bajos.

Una FPGA es una matriz programable de puertas lógicas, compuesta por bloques lógicos configurables mediante interconexiones programables. Las FPGAs pueden ser reprogramadas para una aplicación específica varias veces. En el campo de los dispositivos lógicos también se encuentran los ASIC, que a diferencia de una FPGA no se puede reprogramar ofreciendo por contra una mayor frecuencia de funcionamiento frente a la FPGA.

El uso de las FPGA está orientado cuando se requiere una optimización específica de una determinada aplicación y una futura modificación y mejora de ésta. Las FPGA se usan en el campo de la inferencia de la inteligencia artificial, puesto que ofrecen un menor consumo enérgetico que otras soluciones específicas. El menor consumo enérgetico de este tipo de dispositivos viene dado por el hecho de que se pueden programar partes específicas de la FPGA, dejando inactivas las demás si no se van a usar.

El uso de una FPGA frente a las arquitecturas heterogéneas más comunes, como una GPU (tarjeta gráfica), resulta más práctico cuándo se requiere ejecutar un proceso de inferencia de manera desacoplada, es decir sin la necesidad de requerir un ordenador. Mediante una FPGA adecuada para la aplicación a usar se puede realizar un proceso de inferencia con una latencia baja y de manera eficiente desde el punto de vista energético.

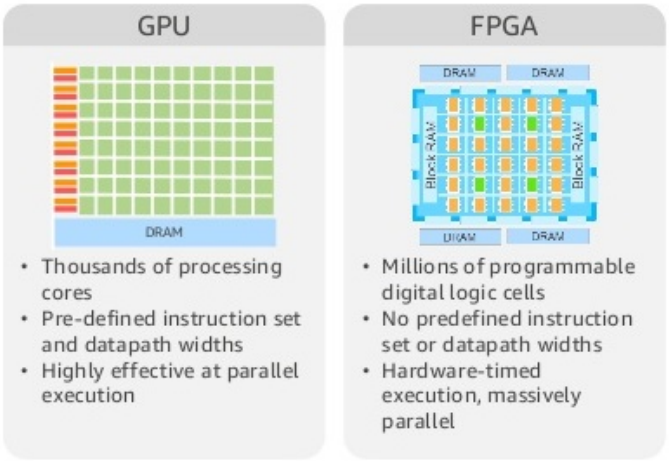

Figura 1.2: Comparativa de una GPU frente a una FPGA

La programación de las FPGA presenta una gran complejidad debido a la multitud de tareas a realizar para lograr una óptima implementación del dispositivo, destacando la técnica conocida como *Floorplanning* [14], que permite elegir la mejor agrupación y conectividad de toda la lógica necesaria para implementar el diseño.

Principalmente en las FPGA se programa su comportamiento mediante lenguajes de descripción de hardware, que permiten especificar el diseño a implementar por la FPGA. Este tipo de aproximación en el desarrollo de la FPGA requiere de una gran inversión de tiempo por parte del diseñador.

Es por ello que para facilitar la programación de estos dispositivos existen herramientas de síntesis de alto nivel (HLS) [7], que permiten, a partir de una primera implementación en un lenguaje de programación de alto nivel (C y C++), ofrecer un diseño final sobre la FPGA directamente. Este tipo de herramientas facilitan la programación de la FPGA, reduciendo costes.

La implementación de las redes neuronales en sistemas reconfigurables (FPGAs), presenta un problema, pues estos algoritmos precisan de un gran número de operaciones en coma flotante, terminando también en tiempos de ejecución elevados. Es por ello que se hace imprescindible una implementación eficiente de estos algoritmos, aplicando para conseguirlo las técnicas de optimización de hardware más comunes y apropiadas.

1.2 Objetivos

El objetivo principal de este trabajo es el de realizar una implementación de las capas más comunes en los procesos de inferencia sobre sistemas reconfigurables, aplicando para ello las optimizaciones oportunas. El fin de este trabajo es el de poder obtener una primera aproximación a la implementación de los procesos de inferencia sobre FPGA.

En cuanto a la implementación del diseño se ha buscado analizar y aplicar directivas de programación de un lenguaje de alto nivel (C y C++) para así poder obtener diseños eficientes sobre FPGA.

Por otra parte, también se va a analizar el impacto de los diferentes formatos de precisión en la implementación de procesos de inferencia sobre FPGAs y cómo afectan al rendimiento del diseño final.

Para ello, por una parte se describirán los algoritmos empleados mediante un lenguaje de alto nivel, para el desarrollo de este trabajo en concreto se ha usado C/C++, para posteriormente adaptar y transformar en una implementación *hardware* usando para ello herramientas de síntesis de alto nivel.

Y finalmente, se obtendrá un análisis de uso de espacio de la FPGA para así poder sacar conclusiones y establecer una posible mejora y optimizaciones oportunas que permitan un incremento en el rendimiento de los procesos de inferencia sobre sistemas reconfigurables.

1.3 Estructura de la memoria

Este trabajo de fin de grado se ha estructurado por capítulos y secciones para ofrecer toda la información de la manera más clara posible y facilitar el entendimiento, quedando de la siguiente manera:

- **Introducción**
  El primer capitulo del trabajo, en el cuál se han expuesto la motivación y los principales objetivos a alcanzar.
- **Redes neuronales**
  En este segundo capítulo se muestra al lector de manera breve cuáles son las distintas implementaciones de algoritmos de redes neuronales, enfatizando en las redes neuronales convolucionales, principal objetivo de desarrollo de este trabajo.
- **FPGAs y High Level Synthesis**
  En este capítulo se describe la arquitectura principal de una FPGA, su funcionamiento al igual que la programación de la misma. Además, también se describe la herramienta de síntesis usada para el desarrollo del diseño final implementado.
- **Implementación con HLS**
  En este otro capítulo se detalla y describe la implementación de un proceso de inferencia de ejemplo diseñado sobre una FPGA.
- **Resultados y analisis de recursos**
  En este capítulo se obtienen los resultados conseguidos con el proceso de inferencia diseñado y muestras de tiempo de ejecuciones.
- **Conclusiones y trabajo futuro**
  Por último, en este capítulo se describen las conclusiones que se han obtenido del trabajo y las posibles mejoras a tratar en un trabajo futuro del mismo.

1.4 Contexto del trabajo

Todo el desarrollo del trabajo se ha integrado dentro del grupo de investigacion de Arquitecturas Paralelas de la UPV (GAP), el cuál centra su línea de investigación en las redes de interconexión para computación paralela, además del estudio de nuevas arquitecturas de computadores. [13]

Este trabajo se ha integrado en un equipo formado por varios miembros, tanto alumnos como profesores, con el objetivo común del desarrollo y estudio de redes neuronales en distintos ámbitos sobre distintos dispositivos hardware objetivo (FPGA, computación paralela y GPUs).

Durante el desarrollo del trabajo se ha trabajado con una retroalimentación constante mediante reuniones periódicas con todo el equipo. La comunicación entre los miembros se ha realizado mediante aplicaciones de comunicación, a su vez que la colaboración de código mediante la herramienta Git.

1.5 Herramientas utilizadas

Durante el desarrollo del trabajo de fin de grado se ha hecho uso de las siguientes herramientas software:

- **GIT**
  Es un sistema de control de versión que permite mantener un registro continuo de los cambios producidos durante el transcurso del desarrollo del proyecto. Además Git también ofrece la coordinación del desarrollo entre varios integrantes. [12]

- **Vitis HLS**
  Proyecto de sintesís de alto nivel encargado de realizar simulaciones y síntesis. Este proyecto aglutina todas las herramientas necesarias para programar una FPGA mediante HLS. [7]

- ***Testbench* en C/C++**
  Programa desarrollado durante todo el trabajo fin de grado implementado en C y C++, para lanzar ejecuciones de distintas simulaciones sobre una FPGA.

- ***OpenCL***
  Es una API de bajo nivel que permite la comunicación entre una máquina host (ordenador) y un dispositivo lógico conectado a la misma, como una GPU o una FPGA. Mediante OpenCL se ejecutan los algoritmos previamente diseñados sobre la FPGA.

- **Microsoft Teams**
  Plataforma de software de comunicación. Se ha usado a lo largo del desarrollo del trabajo para mantener contacto directo mediante reuniones entre los distintos integrantes.

CAPÍTULO 2
Redes neuronales

Antes de ahondar en las redes neuronales, tenemos que entender primero que es el aprendizaje automático. Se puede definir el aprendizaje automático como la implementación de sistemas capaces de realizar una mejora en su comportamiento y resultado, en base a una experencia previa para así aprender a partir de unos datos dados. Los algoritmos de aprendizaje automático permiten obtener unos resultados de manera correcta y gracias al aprendizaje se reduce el número de errores de éstos.

El aprendizaje automático se puede clasificar principalmente en dos grupos:

- **Aprendizaje supervisado** (*Supervised learning*)

  En el aprendizaje supervisado, el sistema realiza una clasificación de los datos, es decir, recibe unos datos de entrada y unas etiquetas asociadas a esos datos, por lo que el algoritmo final debe ser capaz de asociar para un dato de entrada la etiqueta correspondiente. Un ejemplo de aprendizaje supervisado, es el algoritmo perceptrón multicapa.

- **Aprendizaje no supervisado** (*Unsupervised learning*)

  Por otra parte, en el aprendizaje no supervisado, el sistema recibe una gran cantidad de etiquetas y propiedades diferentes asociados a los datos de entrada. En este tipo de aprendizaje, se trata de conseguir que el sistema sea capaz de identificar unos patrones determinados.

2.1 Proceso de entrenamiento

Dentro del aprendizaje supervisado, y tal como se ha mencionado anteriormente, existen dos procesos claramente diferenciados, por un lado los procesos de entrenamiento, que a partir de una gran cantidad de datos se crea un modelo de predicción que será usado posteriormente por un proceso de inferencia que consiga obtener una predicción determinada en un entorno de producción.

Dentro de los procesos de entrenamiento se integran distintas técnicas y elementos que permiten lograr un resultado óptimo del modelo de predicción deseado: [15]

- ***Uso de pesos y gradientes***

En una red neuronal convolucional, existen una serie de funciones lineales representadas en forma de matrices, las cuáles se les aplican una serie de características. Estas características son determinados por los pesos, los cuáles permiten establecer ciertos patrones para el modelo deseado. Por otro lado, está el uso de gradientes, los cuáles permiten conseguir una optimización del modelo final de entrenamiento.

- ***Forward Propagation***

  Ésta parte del proceso de entrenamiento comprende todos los resultados calculados y almacenados de una determinada capa de entrada hacia una capa de salida.

- ***Back Propagation***

  Durante este proceso de entrenamiento se calculan todos los gradientes de los parámetros de una red neuronal. En otras palabras, este tipo de proceso recorre la red neuronal en orden inverso.

2.2 Perceptron multicapa (MLP)

Este tipo de redes son las más comunes, se trata de una red basada en una red neuronal simple, con la diferencia de que el perceptron multicapa puede tener más de una capa oculta dando como resultado una red de mayor complejidad que una red neuronal simple. [4]

El perceptrón multicapa utiliza la técnica de aprendizaje supervisado conocida como *backpropagation* para realizar el entrenamiento de la red. Este tipo de redes tienen la característica de ser totalmente conectadas, es decir cada perceptron de la red está conectado con la siguiente capa. Esta característica ofrece consistencia, pero a la vez demuestra una gran desventaja puesto que el número de parámetros puede crecer considerablemente, pudiendo así aumentar la ineficiencia de la red.

La figura 2.1 muestra un ejemplo de la arquitectura básica de un perceptrón multicapa.

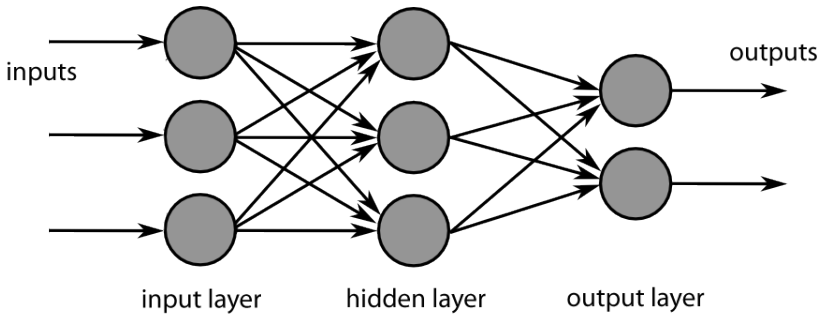

Figura 2.1: Esquema del perceptron multicapa

Cada círculo de la figura representa una neurona de la red. Estas neuronas se organizan en capas, el primer grupo de neuronas son las **entradas** y representan la **capa de entrada** (*input layer*) de la red neuronal. La siguiente agrupación

de neuronas representa la **capa oculta** (*hidden layer*) de la red, la cual realiza los cálculos intermedios de la red. Y finalmente, el último grupo de neuronas son las **salidas**, lo que viene a ser la **capa de salida** (*output layer*) de la red neuronal.

Normalmente, todas las redes neuronales están formadas por neuronas dispuestas y completamente conectadas entre ellas, cómo se observa con las flechas. Las distintas conexiones entre las neuronas tienen asociado un valor llamado **peso**. La principal operación de las capas ocultas de la red, es el de ir multiplicando los valores de una neurona por los pesos de su conexión. Esta operación se puede realizar en todas las capas ocultas de la red, según la configuración.

En resumen, el principal objetivo de una red neuronal es para unos valores determinados de entrada y según las configuraciones de las capas ocultas (operaciones, pesos, etc.), obtener unos valores de salida esperados.

2.3 Redes neuronales recurrentes

Una RNN (*Recurrent Neural Network*) es una clase de red que permite analizar datos estructurados en series temporales. La principal característica de una red neuronal recurrente es la no-linearidad de la red. Es decir, en este tipo de red se incluyen conexiones "hacia atrás", permitiendo así a la red recordar valores previos. [16]

El nombre de redes neuronales recurrentes"se debe a que realizan la misma tarea para cada elemento de la secuencia definida, como por ejemplo para cada elemento de una serie de tiempo, y la salida depende de los resultados anteriores.

En el ejemplo de la figura 2.2, se observa una red neuronal recurrente compuesta por una única neurona, la cuál recibe una entrada (x) produciendo una salida (y), y enviando el valor de esa salida otra vez hacia la misma neurona.

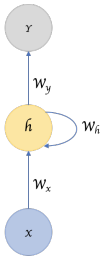

Figura 2.2: Estructura de una red neuronal recurrente de una única neurona

Una red neuronal recurrente es mucho más compleja y completa, está formada por más neuronas, el número de estas viene dado principalmente por la serie de tiempo sobre la que se está tratando. En estas redes más complejas, cada neurona recibe un valor de entrada, como puede ser un valor de peso determinado, y a su vez recibe como entrada el valor de salida de la neurona anterior.

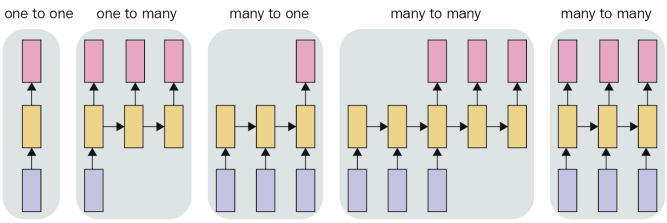

Figura 2.3: Tipos de redes neuronales recurrentes

En la figura 2.3 se muestran los distintos tipos de aplicaciones de una RNN:

- ***One to One***

  La forma más básica de una red neuronal, como se aprecia en la anterior imagen, para una entrada se obtiene una determinada salida. En este caso, la RNN no dispondría del valor intermedio característico de las RNN.

- ***One to Many***

  En esta red neuronal recurrente, para una única entrada se obtienen varias salidas, teniendo en cuenta también el valor intermedio de salida de la neurona anterior.

- ***Many to One***

  Este tipo de RNN se usan cuando se requiere obtener un único valor de salida a partir de múltiples entradas.

- ***Many to Many***

  Finalmente, estas RNN devuelven múltiples valores de salida a partir de varios valores de entrada. El número de datos de entrada puede diferir del de salida.

2.4 Redes neuronales convolucionales

Las CNN (*Convolutional Neural Network*), son un tipo de red neuronal artificial, aplicadas principalmente para operaciones sobre datos de dos o más dimensiones, como por ejemplo imágenes. Estas redes neuronales, se pueden entender como una implementación multicapa del algoritmo perceptrón aplicada a varias dimensiones.

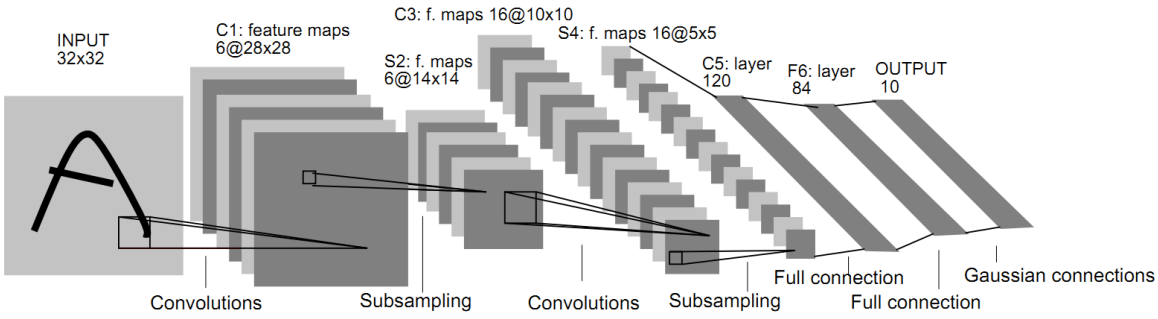

Figura 2.4: Estructura de una red convolucional típica

Las redes convolucionales son una implementación sucesiva de diversas capas dispuestas de manera que los datos de salida de una capa sirven como datos de entrada para la siguienta capa de la red.

Vamos a pasar a resumir y detallar las capas más comunes que forman una red neuronal convolucional:

2.4.1. Capas convolucionales

En esta capa, el dato de entrada convoluciona. En términos matemáticos esto significa que dos funciones se combinan, y se retorna hacia la siguiente capa de la red. La convolución aplicada a una imagen bi-dimensional (expresada en forma de matriz), consiste en que para cada elemento de la matriz, se realiza el cómputo del producto entre el filtro establecido y el elemento de la matriz (píxel de la imagen). En algunas implementaciones de esta capa, también se añade un modificador de pesos o *bias*.

En resumen, cómo se aprecia en la figura 2.5 la salida de la capa convolucional es el resultado de aplicar un filtro, uno distinto por los canales de entrada de la imagen (representado por los colores RGB), y finalmente añadir el valor del *bias*. El tamaño del filtro varía según necesidades, es bastante común aplicar un filtro mediante una matriz bi-dimensional de pequeños tamaños, como por ejemplo un tamaño de filtro de 3x3.

La entrada de la capa convolucional consiste en imágenes de dos dimensiones agrupadas, está agrupacion (Cin) viene indicada por el número de canales de la imagen, por ejemplo una imagen en escala de grises, dispone de un único canal de entrada, mientras que una imagen RGB dispone de tres canales de entrada. El valor de los canales de entrada varía segun las distintas capas convolucionales intermedias. Cada imagen tiene un tamnaño determinado *Hin x Win*, representando la altura y el ancho de la imagen de entrada, respectivamente.

Para realizar la convolución, cada capa convolucional dispone de una agrupación de filtros de tamaño *Kw x Kh* de dimensión *Cin x Cout*, que como se aprecia en la imagen 2.5 se usa para realizar el producto de la región de la imagen seleccionada. Cada capa convolucional produce una agrupación de imagenes de tamaño *Hout x Wout*.

2.4.2. Funciones de activación

Esta capa de las redes convolucionales, aunque se suele integrar a la salida de las capas convolucionales, las mencionaremos como una capa independiente, debido a su gran cantidad de implementaciones distintas.

Las funciones de activación, se acoplan al final de la capa convolucional y se encargan de establecer un umbral determinado e incluso una normalización de los datos, según las necesidades del problema. Existen diversos tipos:

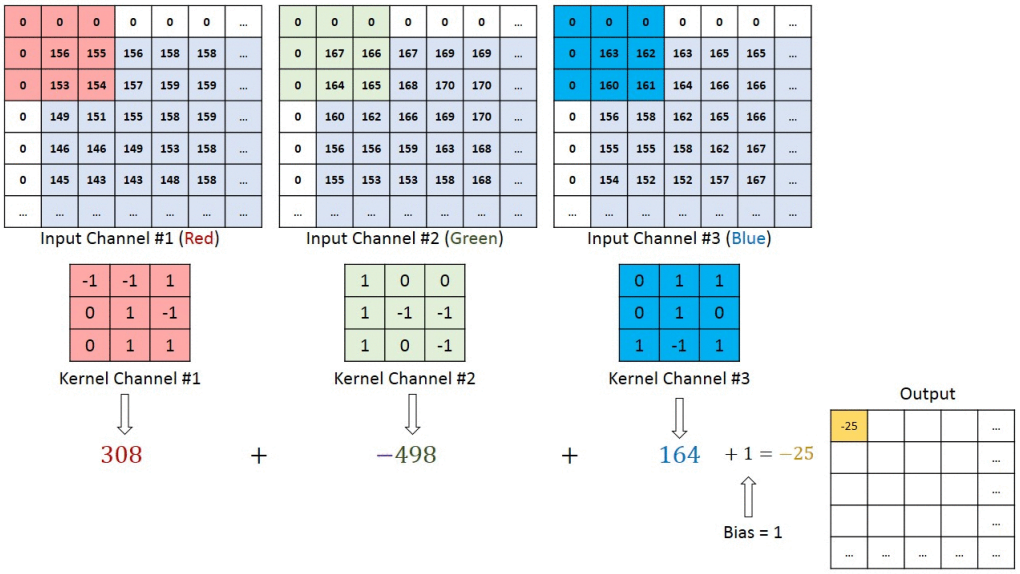

Figura 2.5: Capa convolucional sobre una imagen RGB

ReLU

Esta función de activación, por sus siglas en inglés *Rectified Linear Unit*, es una de las funciones de activación más básicas y usadas, se obtiene como resultado de aplicar la función, cero si el valor de entrada es menor que cero o el mismo valor de entrada en caso contrario, dicho de otra manera se obtiene el máximo entre cero y el valor original de entrada.

Su expresión matemática se denota por:

$$f(x) = max(0, x) \tag{2.1}$$

Sigmoide

Muy parecida a la ReLU en cuánto a funcionalidad, aunque la función sigmoide se usa más para ofrecer optimizaciones de los pesos, al aplicar la segunda derivada en todo el dominio de la función, y estableciendo un umbral positivo entre 0 y 1. La función sigmoide se puede expresar matemáticamente de la siguiente manera:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

En el caso de las redes neuronales convolucionales aplicadas a una imagen de varias dimensiones, estas funciones de activación descritas se aplicarían a cada píxel de la imagen y de su respectiva dimensión. Por ejemplo para una imagen RGB con un tamaño de 256x256, la red realizaría la función de activación sobre 256·256·3 = 196.608 elementos.

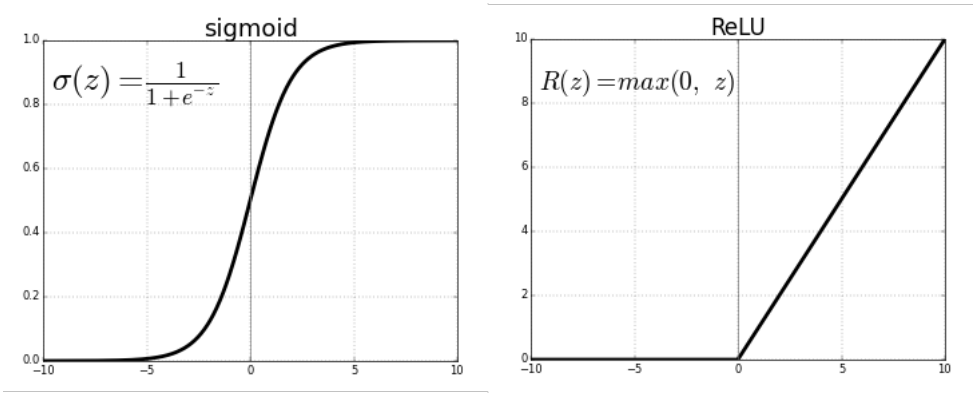

Figura 2.6: Comparación de la Sigmoid y ReLU

2.4.3. Capa de Muestreo (*Pooling layer*)

El muestreo de una red neuronal convolucional permite reducir la dimensionalidad de la red, permitiendo así reducir también la complejidad de cómputo, pero conservando a la vez la información más relevante.

El tipo de muestreo más común es el muestreo por máximos (*maxpooling*), otro también bastante usado es el muestreo de medias (*avgpooling*):

- ***Max pooling.*** Devuelve el máximo valor de la porción de la imagen cubierta por el tamaño de filtro seleccionado. Por ejemplo en la figura 2.7 el tamaño de filtro es de 2x2.
- ***Average pooling.*** La diferencia respecto al anterior muestreo radica en que en este caso se retorna la media de todos los valores de la región cubierta por el tamaño de filtro.

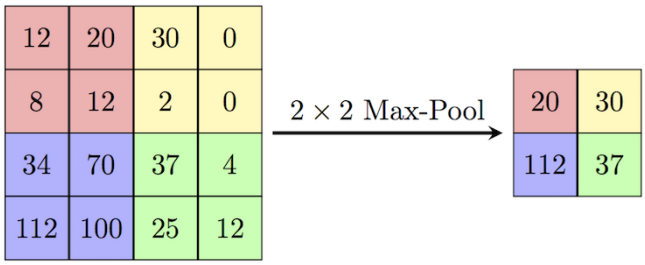

Figura 2.7: Ejemplo de la capa *maxpooling*

CAPÍTULO 3
FPGAs y High Level Synthesis

3.1 FPGA

Una FPGA (*Field Programmable Gate Array*, matriz de puertas programable), es un dispositivo programable formado por una matriz de bloques lógicos reconfigurables capaces de realizar operaciones combinacionales complejas o operaciones lógicas básicas, como por ejemplo circuitos con puertas lógicas. Estos dispositivos permiten la configuración de los bloques lógicos así como su interconexión.

Esta configuración del dispositivo se realiza mediante un lenguaje de descripción de hardware (**VHDL** o **Verilog**, los más comunes), mediante los cuáles permiten describir el comportamiento de los circuitos digitales a partir de un diseño previo, para así llegar a una implementación final del circuito digital sobre una FPGA.

Al contrario de las ASIC (*Application-Specific Integrated Circuit*), las FPGAs son reconfigurables, lo que permite un ahorro de implementación en costes y tiempo. Sin embargo, un inconveniente de las FPGAs respecto a las ASIC es que ofrecen un menor rendimiento.

Las FPGAs nacieron a partir de la combinación de dos tecnologías ya existentes, por una parte de los PLD (*Programmable Logic Devices*) y de los circuitos integrados de aplicación específica (*ASIC*). Estos dispositivos surgieron con la necesidad de tener dispositivos programables que ya ofrecían los PLD, pero con el objetivo de conseguir un rendimiento y un consumo enérgetico bajo cercano a los ASIC.

La primera FPGA que salió al mercado fue creada por Ross Freeman y Bernard Vonderschmitt, fundadores de la acutal *Xilinx Inc.*, empresa líder en la investigación y comercialización de sistemas reconfigurables.

3.1.1. Arquitectura de una FPGA

Cómo hemos mencionado anteriormente, una FPGA está formada principalmente en bloques lógicos, los cuáles tanto su interconexión, su funcionamiento y comportamiento se puede programar. [1]

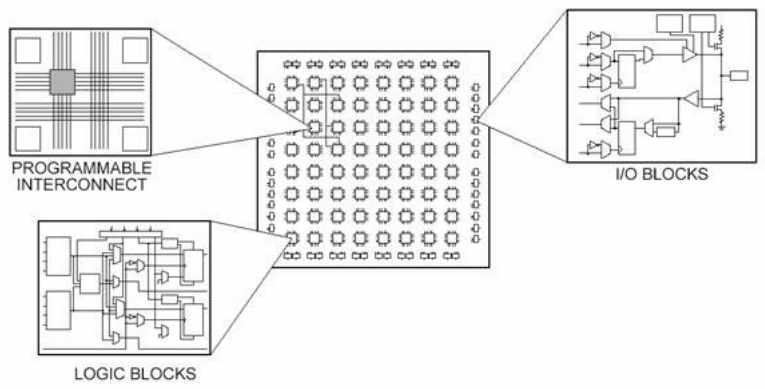

Figura 3.1: Arquitectura de una FPGA

Las FPGAs estan formadas por matrices de bloques lógicos programables los cuáles mediante canales de entrada/salida dispuestos tanto de manera vertical como horizontal, permiten su interconexión, como se muestra en la figura 3.1.

La disposición de los bloques lógicos sobre la FPGA varían en cada fabricante, aunque de manera genérica se establecen en torno a los llamados **CLBs** (*Configurable Logic Blocks*), los cuáles constituyen la parte lógica básica de una FPGA. Estos bloques lógicos, cuándo se establecen conectados, son los encargados de implementar funciones lógicas complejas así como la capacidad de implementar memorias.

Como se aprecia en la figura 3.2, la estructura genérica y más común de un CLB está formada por registros biestables (*flip-flop*), las *lookup-tables* que almacenan unas salidas pre-establecidas para una combinación de entradas determinada, los *multiplexores* que permiten seleccionar entre 2 o más entradas, y finalmente algunos CLBs también implementan un *full-adder* encargado de realizar las operaciones de suma y resta.

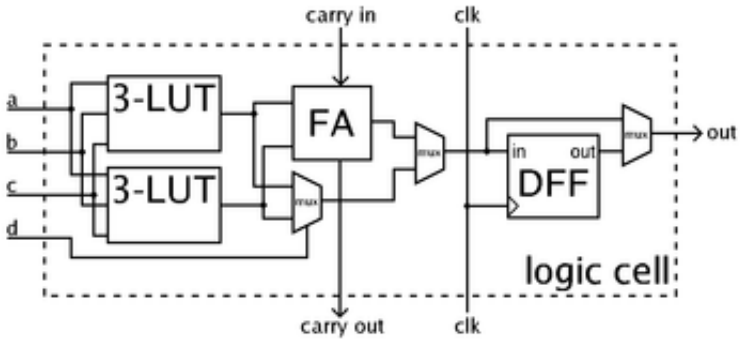

Figura 3.2: Arquitectura de un CLB

Flip-Flop

Los biestables (flip-flop) son circuitos que tienen dos estados estables los cuáles representan un único bit. Cada biestable presente en un CLB es un registro binario que permite guardar estados lógicos entre ciclos de reloj en un circuito sobre la FPGA. Esta unidad es la representación básica de almacenaje en una FPGA.

LUTs

Una LUT (*Look-up Table*) es un bloque lógico que actúa como una memoria ROM. Este bloque lógico se encarga de implementar las cuatro funciones lógicas básicas (AND, OR, XOR, NOT). Una vez programada la FPGA, cada LUT se configura para realizar una función lógica determinada. En resumen, una LUT es el conjunto de conexiones y circuitos presentes para la obtención de operaciones lógicas combinacionales. Mediante distintas configuraciones y en determinados tipos de FPGA, una LUT también puede implementar colas FIFO, muy usadas en este proyecto.

Multiplexor

Este componente es un bloque combinacional lógico formado por dos o más entradas, que tendrá como salida una de estas entradas según la señal de selección introducida.

Full-Adder

Circuito capaz de realizar operaciones de suma y resta para un número determinado de bits, según configuración. Cabe destacar que este componente no se encuentra incorporado en la mayoria de fabricantes. En algunos casos estas operaciones se implementan en dispositivos como los **DSP** que realizan operaciones de multiplicaciones y divisiones, las cuáles requieren de varios ciclos de ejecución, y en algunos casos de suma y resta.

Memorias

Cada FPGA realiza distintas implementaciones de memorias. Las memorias más comunes presentes en una FPGA son las **BRAM** (Block RAM), pequeñas RAMs con un tamaño medio que oscila entre los 4Kb y 32 Kb según implementaciones y necesidades, por otro lado están las **URAM** (ultra RAM), son iguales que las BRAM pero éstas últimas disponen de dos puertos de entrada y de salida, lo que permite hacer dos lecturas o dos escrituras a la memoria concurrentemente. Otro componente que sirve también para implementar las memorias, son las colas FIFOs.

3.1.2. Xilinx Alveo U-200

Durante todo el transcurso del trabajo se ha hecho uso de la FPGA *Xilinx Alveo U-200* [6], para realizar distintas ejecuciones y pruebas de uso. La FPGA ha sido proporcionada por el **GAP** (Grupo de Arquitecturas Paralelas, UPV).

Este tipo de FPGA es una tarjeta aceleradora, muy usada en centros de datos, su principal objetivo es realizar tareas de aceleración de algoritmos. Este dispositivo esta dividido en tres regiones lógicas distintas (*SLRs, Super Logic Region*), en este caso, cada una de ellas tiene un número distinto de componentes, según se aprecia en la imagen de la tabla de especificaciones 3.3.

Resource	SLR 0	SLR 1	SLR 2
CLB LUT	355K	160K	355K
CLB register	723K	331K	723K
Block RAM tile	638	326	638
URAM	320	160	320
DSP	2265	1317	2265

Figura 3.3: Especificaciones Alveo U-200 por región lógica (SLR)

Para conseguir una menor latencia en las comunicaciones al realizar las operaciones, el objetivo es usar el mínimo espacio posible de las regiones. Además cuándo más SLRs se usen, mayor será el consumo energético del dispositivo, debido a que la FPGA tendrá que activar las regiones correspondientes para realizar las operaciones.

3.2 Programación de una FPGA

La implementación de todo el proceso de inferencia durante el desarrollo del trabajo se ha realizado sobre el lenguaje de programación C. Se ha optado por este lenguaje debido a que ofrece un control total sobre la memoria, permitiendo así que el algoritmo sea lo más óptimo posible. Primero se ha realizado una primera aproximación de todos y cada uno de los algoritmos en C, y después se ha adaptado esta implementación a la FPGA.

Para la implementación sobre la FPGA también se ha usado C, mediante *High Level Synthesis* que permite a partir de una implementación de código de alto nivel, cómo es el caso de C, obtener una descripción del código y así implementarlo mediante determinadas directivas directamente sobre la FPGA, de una manera óptima y eficiente.

3.2.1. Lenguaje de descripción de hardware

Normalmente la forma más común de programar una FPGA es mediante un lenguaje de descripción de hardware, cómo VHDL o Verilog, el cuál mediante determinadas instrucciones describen el comportamiento y funcionamiento de un circuito digital.

En este ejemplo se ha realizado una implementación de un multiplexor de 32 bits de 2 entradas a 1 salida mediante Verilog:

```
module MUX2_TO_1 (
    input[31:0] a,
    input[31:0] b,
    input sel,
    output[31:0] out);

    always @ (a or b or sel) begin
        case (sel)
            1'b0: out <= a;
            1'b1: out <= b;
        endcase;
```

```
12      end;
13  endmodule;
```

Listing 3.1: Implementación de un multiplexor de 2 entradas a 1 salida con Verilog

Los bloques *always* @ como el del código anterior, permiten describir acciones y establecer asignaciones en determinadas condiciones. Por ejemplo, en el código anterior el bloque se ejecutará cada vez que se produzca un cambio en cualquiera de las entradas o en la entrada de selección.

Los lenguajes de descripción de hardware permiten realizar una configuración de la FPGA casi completa, pues permiten describir al detalle el comportamiento final del circuito a implementar. En cambio, programar la FPGA mediante un lenguaje de descripción de hardware resulta en una implementación más costosa tanto en tiempo cómo en coste de recursos.

Es por ello, que surgió la herramienta *High Level Synthesis* que permite reducir tiempos de implementación al ofrecer una considerable capacidad de abstracción al realizar la implementación de la FPGA en un lenguaje de alto nivel.

3.2.2. High Level Synthesis

El entorno de desarrollo Vivado junto a Vitis HLS es una herramienta desarrollada por Xilinx que permite realizar diseño de hardware de una FPGA desde lenguajes de alto nivel (C y C++). Para una determinada aplicación específica desarrollada en un lenguaje de alto nivel, esta herramienta mediante operaciones de síntesis y descripciones de comportamiento, es capaz de realizar y aplicar un buen diseño de hardware que imita a la aplicación original. La herramienta HLS ofrece una serie de ventajas y beneficios considerable:

- Gran capacidad de abstracción del hardware al programador, lo que permite un aumento en la productividad al realizar la implementación ya que permite realizar un diseño de hardware sin usar un lenguaje de descripción de hardware.
- Reducir tiempo de desarrollo.
- Verificación de la correcta funcionalidad del algoritmo que se quiere diseñar e implementar en un lenguaje de alto nivel.
- Permite realizar tareas de depuración de manera más rápida y eficiente que mediante un lenguaje de descripción de hardware.
- Recompilar el código original en alto nivel rápidamente e implementarlo en otros dispositivos sin realizar una implementación específica para una determinada plataforma.

La herramienta de síntesis HLS está formada por la ejecución de una serie de etapas distintas: [7]

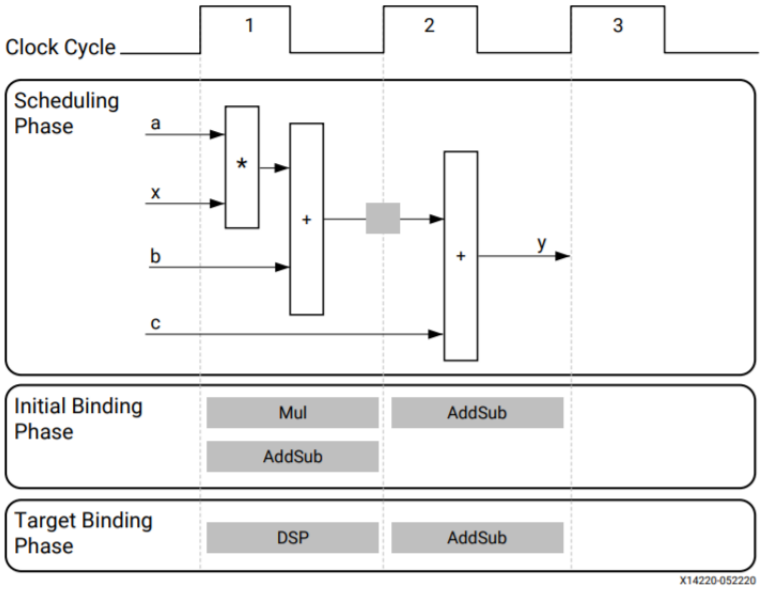

Figura 3.4: Ejemplo de planificación y enlazado de operaciones

Planificación de operaciones

En esta etapa, la herramienta realiza un análisis para obtener las operaciones que se van a realizar durante cada ciclo de reloj. Para ello obtiene las dependencias de las operaciones, mide la latencia del ciclo en cuestión y tiempo que tardaría en realizarse la operación sobre el dispositivo objetivo, se obtiene el uso de recursos disponibles y por último se aplican las directivas que ha requerido el diseñador.

Enlazado y asignación

La etapa de enlazado es la encargada de asignar los recursos de hardware necesarios para cada una de las operaciones descritas e identificadas en la etapa superior.

```
1 int foo(char x, char a, char b, char c) {
2     char y;
3     y = x*a+b+c;
4     return y;
5 }
```

Dado el código de ejemplo anterior y la figura 3.4 que representa las dos etapas descritas anteriormente. Primero se realiza la planificación de las operaciones, de lo cuál se obtiene que la herramienta de síntesis planifica las operaciones por ciclo de reloj:

- Durante el primer ciclo de reloj se realiza la multiplicación y la primera suma.
- En el segundo y último ciclo de reloj se realiza la última suma, si el resultado de la anterior operación está guardada en el registro (región marcada en gris en la figura entre el primer y segundo ciclo de reloj). Los registros son

necesarios sólo si se precisa que el resultado de una operación se mantenga entre ciclos de ejecución.

En el ejemplo anterior, todas las variables son de tipo `char`, por lo que los puertos de los datos de entrada en la implementación final en hardware tendrán un ancho de 8 bits. Y por otra parte, la variable de retorno es de tipo `int`, por lo que en este caso se precisan 32 bits de ancho del puerto de salida.

Extracción de la lógica de control

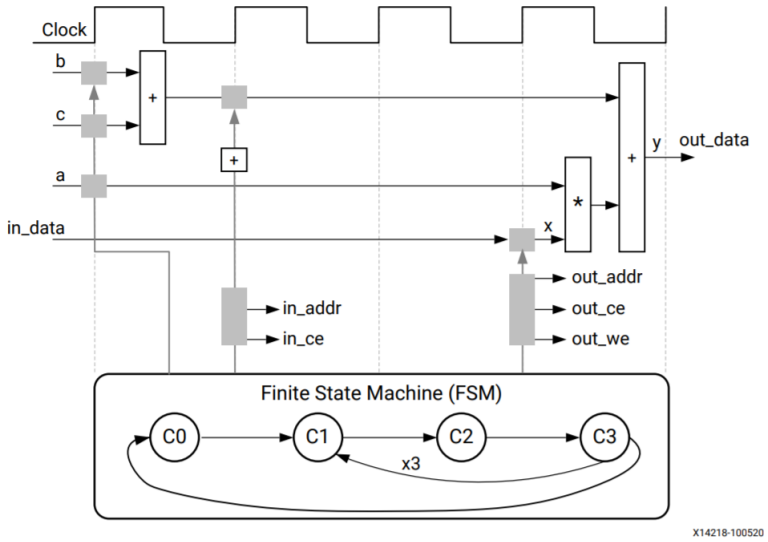

Figura 3.5: Ejemplo para la etapa de extracción de la lógica de control

Finalmente, durante esta etapa se crea una máquina de estados finitos (FSM) que secuencia y ordena las operaciones en el diseño final según la planificación realizada anteriormente por la herramienta.

```
1 void foo(int in[3], char a, char b, char c, int out[3]) {
2     int x,y;
3     for(int i = 0; i < 3; i++) {
4         x = in[i];
5         y = a*x + b + c;
6         out[i] = y;
7     }
8 }
```

Este código realiza la misma operación que el código anterior. En cambio, en este caso realiza las operaciones dentro de un bucle `for` y dos de sus argumentos son arreglos. Durante esta etapa la herramienta obtendrá un FSM como el mostrado en la figura 3.5, la cuál describe una operación compuesta por cuatro ciclos de ejecución, los cuáles se repetirán tres veces, excepto el primer ciclo.

3.2.3. Directivas de optimización de HLS

Para lograr una implementación eficiente y útil sobre la FPGA usando HLS, no basta con escribir el código fuente sobre C y luego compilar mediante las herramientas de HLS, hay que añadir unas determinadas directivas al código en C. Las directivas de C indican al preprocesador del compilador determinadas acciones que debe aplicar antes de la compilación, en este caso, las directivas específicas de HLS, vienen dadas por *#pragma HLS <directiva>*, establecen el comportamiento que deberá tener el algoritmo a implementar según las necesidades del diseño.

Vamos a describir las directivas de optimización de HLS que se han usado para la implementación final del proceso de inferencia:

PIPELINE

Con esta directiva, se permite la ejecución concurrente de operaciones sobre un bucle o función. Un bucle al cuál se ha aplicado esta directiva, permite procesar nuevos datos cada N ciclos de reloj, también conocido como **Intervalo de Iniciación** (II). Por ejemplo, con un `II=1`, un bucle optimizado mediante la directiva *PIPELINE* permitirá procesar un nuevo dato de entrada en cada ciclo de reloj.

En el ejemplo de la figura 3.6, se aprecia cómo en un bucle con tres operaciones diferenciadas, se reduce el número de ciclos necesarios para realizar la ejecución, pasando de 8 ciclos con la solución sin la directiva, a finalmente 4 con ella.

La directiva *PIPELINE* ha resultado muy útil en el desarrollo de este trabajo, pues nos ha permitido que cuando una etapa finalize el procesamiento sobre un píxel, pueda seguir la ejecución con la siguiente etapa con el mismo píxel.

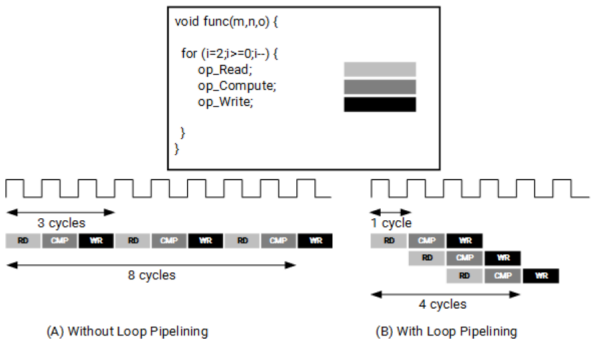

Figura 3.6: Directiva HLS PIPELINE

UNROLL

La técnica *unrolling* es una técnica de optmización de bucles muy usada en varios ámbitos del desarrollo, tanto de hardware como de software. Esta técnica permite reducir el coste de ejecución de las iteraciones de un bucle, a costa de aumentar el número de instrucciones dentro del mismo.

En el ámbito de las FPGAs, aplicar *unrolling* permite para un bucle dado de `N` iteraciones y con un *unrolling* completo, ejecutar el bucle de manera totalmente

concurrente, reduciendo así el tiempo de ejecución considerablemente, al realizarse sólo una iteración. La desventaja de aplicar esta técnica es que incrementa considerablemente el uso de recusros de la FPGA, el espacio requerido para realizar una iteración se multiplica por el factor de *unrolling* aplicado.

DATAFLOW

Mediante ésta directiva, se consigue realizar un pipelining entre las distintas funciones involucradas en el proceso. En el ejemplo de la figura 3.7 , se aprecia como para una ejecución de tres funciones, después de aplicar ésta directiva, la ejecución se reduce de ocho ciclos a un total de cinco, debido a aplicar el pipelining entre las distintas etapas.

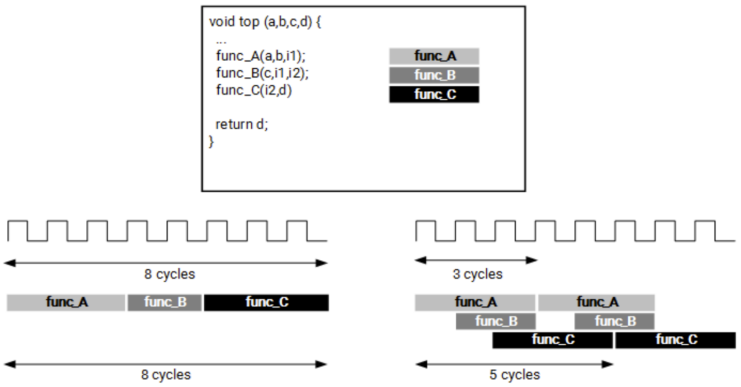

Figura 3.7: Directiva HLS DATAFLOW

ARRAY PARTITION

Realiza un particionado de una matriz (o vector) en varios elementos. Este particionado, en la implementación final sobre hardware, resulta en múltiples memorias pequeñas o registros, en vez de una única memoria grande.

Además, el particionado permite que se puedan producir más lecturas y escrituras concurrentes sobre la matriz o vector. La desventaja de aplicar esta directiva, es que en el caso de una matriz de tamaño considerable, puede hacer un uso excesivo de registros.

DATA PACK

Empaqueta y agrupa los elementos de una estructura (`struct`) en un elemento escalar del tamaño de esa estructura.

Por ejemplo, si se aplica la directiva sobre un elemento que implementa la estructura definida en 3.2, resultará en la implementación final de hardware, en un único elemento de tamaño `32 bits * 3 = 96 bits`.

```
1 struct data_t {
2     int elements[3];
3 }
```

Listing 3.2: Ejemplo struct con data_pack

STREAM

Por defecto, la herramienta de síntesis implementa los vectores o matrices en memoria RAM. Si la lectura o escritura se produce de manera secuencial, es decir no se requiere acceder concurrentemente a distintas posiciones de memoria, esta matriz se puede implementar mediante la directiva *STREAM*, que usará colas FIFO (*First In - First Out*) para la implementación de las matrices, las cuáles el valor de salida de la cola será el primer valor que se ha escrito.

CAPÍTULO 4
Implementación con HLS

En este capítulo se mostrará la implementación sobre HLS del proceso de inferencia realizado durante el proyecto.

La implementación se ha abordado de manera que al realizarse la ejecución de un proceso de inferencia determinado se pudiera conocer el tiempo de ejecución del mismo sabiendo únicamente el tamaño de la imagen a aplicar, independientemente del número de canales de ésta, pues el objetivo de este trabajo es conseguir una paralelización de la imagen a nivel de canales (CPI). Es decir, el tiempo de ejecución completo del proceso de inferencia vendrá determinado únicamente por el tamaño de la imagen (ancho y largo), conociendo para ello la frecuencia de reloj de la FPGA objetivo.

4.1 Estructura del código

Vamos a pasar a detallar cómo se ha estructurado los archivos de todo el código para la implementación final.

Por un lado, en la carpeta **raíz** del proyecto tenemos:

- **Makefile**

  Archivo que establece las reglas y acciones a realizar por el programa `make`. Con la ejecución se lanza la compilación del programa con las reglas establecidas en el archivo.

- **setenv.sh**

  Script encargado de establecer las variables de entorno necesarias para acceder a todas las utilidades de la herramienta HLS.

Por otro lado, en la carpeta **fuente** del proyecto nos encontramos con:

- **net.cpp**

  Archivo principal y único del kernel del proceso de inferencia entero, que implementa todas las capas necesarias.

- **test_net.cpp**

Archivo lanzadera el cuál genera datos aleatorios según la configuración deseada, transfiere los datos a la FPGA y lo ejecuta. Finalmente, el test compara los resultados obtenidos por la máquina *host* con los obtenidos por la FPGA.

Y finalmente, en el directorio ***include*** se disponen de todas las archivos cabeceras que definen los parámetros constantes y tipos de datos:

- **parameters.h**

  En este archivo se definen los parámetros estáticos y específicos para la implementación final sobre la FPGA.

- **types.h**

  Definición genérica de los tipos de datos a usar para el proceso de inferencia. Permite seleccionar un tipo de datos concreto sin hacer ningún cambio en todo el código.

- **utils.h**

  Archivo con definiciones de utilidades precisadas para algunas operaciones.

4.2 Tipos de datos

En el transcurso del proyecto, se han hecho pruebas de uso con distintos tipos de datos. Primero se optó por el uso de los tipos primitivos de C:

- `int`. Tipo primitivo de enteros de 32 bits.
- `float`. Tipo primitivo de números en coma flotante de 32 bits.

Se descartó el uso del tipo `double` debido a su gran tamaño (64 bits), y teniendo en cuenta además que en una FPGA las operaciones de precisión de coma flotante afectan mucho al rendimiento final.

4.2.1. Tipos de datos de Precisión Arbitraria

Librería perteneciente a la herramienta HLS, la cuál permite establecer tipos de datos de tamaño arbitario. Por ejemplo, en el caso de que se requiera una implementación de un multiplicador de 18 bits, mediante esta librería se puede especificar un tipo de datos específico de 18 bits, evitando así ser forzados a usar un multiplicador de 32 bits con los tipos primitivos de C.

Se han usado tres tipos de datos distintos de precisión arbitraria, con distintas configuraciones:

- **AP Integer**

  Tipo de precisión arbitraria entero. Permite establecer el tamaño del tipo de datos a cualquier tamaño según necesidades. En este proyecto, se ha usado este tipo con un tamaño de 8 bits, dando así números entre `-127 y 128`.

```
#ifdef API_TYPE
#include "ap_int.h"
#define DATA_WIDTH 8
typedef ap_int<DATA_WIDTH> data_type;
#define MIN_VALUE -127
#endif
```

- **AP Fixed**

  Tipo de precisión arbitraria de coma fija. Al igual que con el anterior tipo, permite determinar su tamaño, pero en este caso tanto de la parte entera como de la coma flotante.

```
#ifdef APF_TYPE
#include "ap_fixed.h"
typedef ap_fixed<16, 8> data_type;
#define MIN_VALUE -999
#endif
```

- **AP Unsigned Integer**

  Tipo de precisión arbitraria entero sin signo. Este tipo es como el `unsigned` primitivo de C. En este caso, a diferencia del primer tipo de precisión arbitraria mencionado, no tiene signo, es decir, este tipo solo puede representar números positivos. En el ejemplo siguiente usado en el proyecto, puede comprender números con valores comprendidos entre 0 y 255.

```
#ifdef APU_TYPE
#include "ap_int.h"
#define DATA_WIDTH 8
typedef ap_uint<DATA_WIDTH> data_type;
#define MIN_VALUE 0
#endif
```

4.2.2. Definición de los tipos de datos usados

Para la representación de un píxel de la imagen se ha creado un nuevo tipo de datos, mediante la palabra reservada `typedef` de C que permite crear un tipo de datos a partir de otro ya existente. Durante este proyecto se han creado varios tipos de datos según necesidades en las distintas etapas. Vamos a destacar el tipo de datos creado para la imagen de entrada y el tipo creado para la imagen de salida:

```
typedef struct input_t {
    data_type pixel[CPI]; /* Input channels */
} Input_t;

typedef struct output_t {
  data_type pixel[CPO]; /* Output channels */
} Output_t;
```

Por lo tanto, para una imagen de entrada de tamaño `N` y `CPI`, conseguimos una matriz únicamente de `N` elementos de tipo `input_t` que internamente tiene

agrupados los píxeles pertenecientes a todos los canales. El uso de este tipo de datos, facilita la transferencia de datos y su manipulación al mantener los datos agrupados mediante estructuras, representados en C como `struct`.

4.3 Estructura de la implementación

Vamos a describir ahora cómo se ha centrado la implementación del proceso de inferencia, las problemáticas que se han encontrado y cómo se han abordado.

Principalmente todo el proceso de cómputo de la capa convolucional se ha dividido en cinco partes diferenciadas cómo se aprecia en la figura 4.1, las cuáles realizan acciones y operaciones distintas.

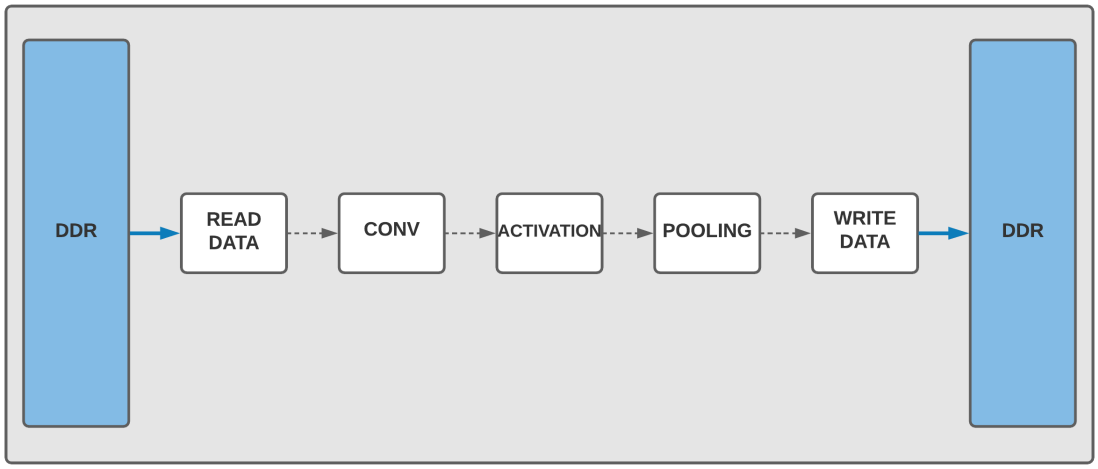

Figura 4.1: Estructura del proceso de inferencia

El proceso de inferencia de una capa convolucional desarrollado empieza con la transferencia de una imagen desde la máquina *host* a la FPGA, una vez la imagen esté almacenada en la memoria RAM **DDR** de la FPGA, comienza el proceso de inferencia sobre la FPGA de forma autónoma.

En la figura 4.1 se muestran dos tipos de flechas, las flechas de color azul representan una lectura o escritura directa de los datos desde memoria, mientras que las otras flechas representan accesos a las colas FIFO.

4.3.1. Módulo *READ_DATA*

Módulo encargado de la lectura de los datos de la imagen de entrada. Para la lectura, se lee de memoria píxel por píxel de la imagen y se transfiere a una cola **FIFO** implementada en HLS mediante la directiva *HLS STREAM*. Esta cola será la que servirá de entrada al siguiente módulo.

```
1 static void read_data(uint HI, uint WI, Input_t *ptr_in, hls::stream<
    Input_t> &out)
2 {
3     int size = HI * WI;
```

```
4       for (int i=0; i < size; i++) {
5           #pragma HLS PIPELINE II=1
6           out << ptr_in[i];
7       }
8   }
```

Listing 4.1: Módulo de lectura

Para una imagen de tamaño `HI x WI`, este módulo realizará mediante un bucle de `HI x WI` iteraciones la lectura de la imagen en memoria, la cuál para cada iteración se leerá de memoria y se pasará a través de la cola FIFO para ser usada por el siguiente módulo.

Y finalmente, en la línea 6 del anterior código se realiza la lectura desde memoria (`ptr_in[i]`) y se le pasa el valor a la cola FIFO de salida.

4.3.2. Módulo *WRITE_DATA*

Implementación parecida al módulo anterior, con la única diferencia de que este módulo se encarga de hacer la escritura en memoria de la imagen resultante del proceso de inferencia, leyendo directamente de la cola FIFO de entrada.

```
1   static void write_output(uint HO, uint WO, hls::stream<Output_t> &in,
        Output_t *output)
2   {
3       int size = HO * WO;
4       for (int i=0; i < size; i++) {
5           #pragma HLS PIPELINE II=1
6           output[i] = in.read();
7       }
8   }
```

Listing 4.2: Módulo de escritura

La función `in.read()`, lee el elemento del *stream* posicionado para su lectura, y mediante la asignación típica de C se escribe en memoria (`output[i]`).

4.4 Implementación de la capa convolucional

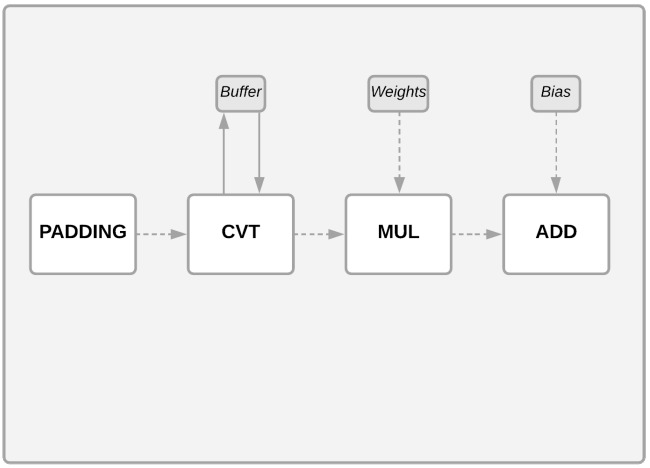

Figura 4.2: Flujo de la capa convolucional

En esta sección se detalla la implementación de la capa convolucional, describiendo cada uno de los módulos mostrados en la figura 4.2.

La implementación de esta capa de la red se ha dividido en cuatro módulos con la técnica del *dataflow*. Como se aprecia en la imagen anterior, cada módulo lee los datos del anterior, se procesan y se sacan hacia la siguiente etapa.

En el ejemplo anterior de la figura 2.5, la operación se divide en:

- Primero, se aplica el ***padding*** correspondiente, en el ejemplo anterior se añade una fila y columna de ceros, a los dos lados horizontales y verticales, respectivamente.
- Después para operar sobre las regiones marcadas en la figura, se aplica el módulo ***CVT***, encargado de recibir del módulo anterior los píxeles de la imagen.
- Por otro lado, para obtener el resultado del producto de cada filtro por el de la matriz de pesos correspondiente, se usa el módulo ***MUL***.
- Y por último, el módulo ***ADD*** añade todos los valores obtenidos anteriormente por cada uno de los canales de la imagen y se le añade el modificador de pesos (*bias*).

Cómo se aprecia en el esquema anterior, hay dos tipos de flechas, por un lado las flechas sólidas que representan transferencias de datos desde memoria, bien sea implementada con *BlockRAMs* o con una memoria RAM DDR. Y por otra parte, las flechas punteadas representan transferencias de datos mediante colas FIFO.

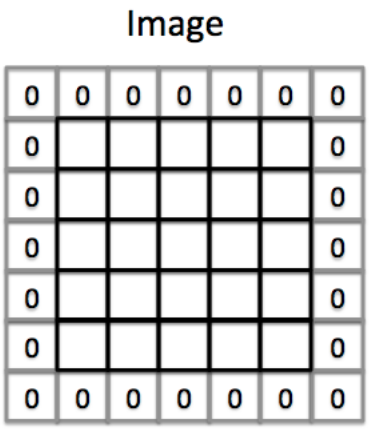

Figura 4.3: Ejemplo de *padding*

4.4.1. Módulo *PADDING*

En este módulo se reciben los datos obtenidos a partir de la lectura de la memoria, mediante el módulo descrito en la anterior sección `read_data()`, y finalmente, se aplican los valores de *padding* establecidos al lanzar la ejecución del proceso de inferencia.

La técnica de *padding* consiste en agregar una cantidad determinada de píxeles con valor a cero a todos los lados de la imagen que se esté procesando. La cantidad de píxeles que se agrega a los lados puede ser configurable. Por ejemplo, en la figura 4.3, se tiene una imagen original de tamaño `5 x 5` a la cuál se le ha añadido un píxel a todos los lados de la imagen, con lo que la imagen resultante es de tamaño `7 x 7`.

Normalmente, el padding se usa cuando se quiere conservar el tamaño original de la imagen, pues al finalizar la operación convolucional, el tamaño final de la imagen puede verse reducido pudiendo extraerse menos características de las que se desean durante el proceso de inferencia.

Vamos a pasar al detalle de la implementación de esta operación:

```
1 static void padding_conv(uint HI, uint WI, uint PH, uint PW, hls::
    stream<Input_t> &in, hls::stream<Input_t> &out)
2 {
3     Input_t pix_zero;
4     Input_t pix_out;
```

Por un lado, la función recibe como datos de entrada el tamaño de la imagen, mediante las expresiones `uint HI` y `uint WI`. Además, recibe también el tamaño del *padding* a aplicar, en este caso se permite un valor distinto para los píxeles del ancho de la imagen y otro para los del largo de la misma.

Después, por otra parte, en el siguiente fragmento de código, se define el píxel con valor a cero, que será el cuál se usará para añadir los píxeles de *padding* a los lados de la imagen.

```
1     for (int cpi=0; cpi < CPI; cpi++) {
2         #pragma HLS UNROLL
3         pix_zero.pixel[cpi] = 0.f;
```

```
    }
```

Finalmente, en este último fragmento de código, se aplica el *padding* en si. Se realizan las operaciones adecuadas para aplicar los píxeles de *padding* en las posiciones correctas. Durante cada iteración de los bucles, se tiene que sacar un píxel de la imagen para que la siguiente etapa pueda realizar el procesamiento oportuno sobre ese píxel. Para ello, se calcula, teniendo en cuenta el tamaño resultante de la imagen al aplicar el *padding*, si el píxel de salida será de la imagen actual o propio del *padding*.

```
    for (int hi=0; hi < HI; hi++) {
        for (int wi=0; wi < WI; wi++) {
            #pragma HLS PIPELINE II=1
            int check_ph = PH - 1;
            int check_pw = PW - 1;

            int img_col = (hi > (check_ph)) && (hi < (HI - PH));
            int img_row = (wi > (check_pw)) && (wi < (WI - PW));

            if (img_col && img_row) pix_out = in.read();
            else pix_out = pix_zero;

            out << pix_out;
        }
    }
}
```

4.4.2. Módulo *CVT*

Una vez se ejecuta la etapa anterior, el módulo *Convert (Cvt)*, recibe el píxel a procesar y lo almacena en un *buffer* temporal en memoria, representado mediante un arreglo:

```
    Input_t pix_buff[CONV_KH];
    #pragma HLS ARRAY_PARTITION variable=pix_buff complete
    Input_t pixels[CONV_KH * CONV_KW];
    #pragma HLS ARRAY_PARTITION variable=pixels complete
    Input_t buffer[CONV_KH][MAX_WIDTH];
    #pragma HLS ARRAY_PARTITION variable=buffer dim=1
```

El *buffer* se ha requerido para mantener guardados de forma temporal los valores que encajarían dentro del tamaño de filtro de procesamiento de la capa convolucional, debido a que en cada iteración se lee un píxel diferente, y se sobreescribe el valor leído en cada iteración.

A este arreglo se le ha aplicado la directiva `ARRAY_PARTITION`, para dividir en pequeñas partes el *buffer*, debido a que en un mismo ciclo del proceso se estará accediendo de manera concurrente a memoria para leer del *buffer*, cómo veremos más adelante.

Los otros dos arreglos, `pix_buff` y `pixels`, se usan para operaciones intermedias, y nótese el uso también de la directiva `ARRAY_PARTITION`, que en este caso se usa con la connotación `complete`, la cuál realiza el particionado completo evitando así el uso de memorias para implementarlo mediante registros.

```
for (int hi=0; hi < HI; hi++) {
    for (int wi=0; wi < WI; wi++) {
        #pragma HLS PIPELINE II=1

        pixel = input.read();
```

En el fragmento de código superior, se lee el píxel de la imagen, el cuál acaba de ser procesado en el ciclo de ejecución anterior por la etapa *padding*, y se almacena en la variable `pixel` implementada en forma de registro sobre el diseño final de hardware. Después, esta operación se repite, en la siguiente iteración del bucle anidado.

```
        int row_buffer = (hi %CONV_KH);
        buffer[row_buffer][wi] = pixel;

        int shift_frame = (hi > (CONV_KH - 2)) & (wi > (CONV_KW - 1));
        send_frame = (hi > (CONV_KH - 2)) & (wi > (CONV_KW - 2));
        send_frame = send_frame & ((hi %CONV_SH == (CONV_SH - 1)) && (
            wi %CONV_SW == (CONV_SW - 1)));
```

En el anterior fragmento de código, se realiza dos de los cálculos más importantes de toda la ejecución de ésta etapa:

- Por una parte, en la variable `shift_frame` se realiza el cálculo para saber si se tiene que cambiar la posición del píxel de la imagen en el búffer.
- Por otro lado, mediante el registro `send_frame` se establecerá si en la iteración en ejecución se ha recibido la región entera sobre la cuál se debe realizar la convolución.

```
        for (int i=0; i < CONV_KH; i++) {
            #pragma HLS UNROLL
            pix_buff[i] = buffer[i][wi];
        }
```

En el bucle anterior, se asigna a un registro el valor del píxel almacenado en memoria. Se ha planteado esta implementación debido a que posteriormente el proceso deberá obtener de forma concurrente todos los píxeles del filtro sobre el que se ésta realizando el proceso, y así se evita romper la técnica de *pipelining* que obligaría a incrementar el intervalo de iniciación para igualarse a los ciclos de lectura de memoria, que sólo permite una única lectura por ciclo de reloj.

```
        for (int kh=0; kh < CONV_KH; kh++) {
            #pragma HLS UNROLL
            int row = (hi + kh) %CONV_KH;

            for (int kw=0; kw < (CONV_KW - 1); kw++) {
                #pragma HLS UNROLL
                int index = kh + kw;

                if (shift_frame) pixels[index] = pixels[index + 1];
                else if (wi==kw) pixels[index] = pix_buff[row];
            }
            int addr = (kh+1) * CONV_KH - 1;
            if (row_buffer == kh) pixels[addr] = pix_buff[kh];
```

```
14          }
15
16          if (send_frame) {
17              for (int i=0; i < (CONV_KH * CONV_KW); i++) {
18                  #pragma HLS UNROLL
19                  frame.kernel[i] = pixels[i];
20              }
21              output << frame;
22          }
23      }
24  }
```

En el anterior y último fragmento de código del módulo, se obtienen los píxeles de la imagen de la región correspondiente sobre el que se está aplicando el filtro de la convolución en su correcta posición. Los píxeles se van asignando a los distinto registros (representados en C mediante arreglos), para ello, se va haciendo el cambio si es necesario en la iteración sobre la que se está realizando el proceso, según el valor obtenido anteriormente mediante la variable `shift_frame`.

Finalmente, el módulo si ya ha obtenido todos los píxeles de la región sobre la que se está ejecutando el proceso, indicado por la variable `send_frame`, le asigna los valores a la cola FIFO de salida mediante la última instrucción, que será usada por el siguiente módulo.

4.4.3. Módulo *MUL*

Este módulo es el encargado de realizar el producto de cada uno de los píxeles del filtro recibido por el anterior módulo por la matriz de pesos correspondiente al canal de la imagen en cuestión.

El siguiente fragmento de código crea un arreglo para almacenar el valor resultante de la operación descrita anteriormente para un píxel determinado. Para ello, se ha aplicado la directiva `ARRAY_PARTITION`, para hacer un particionado completo, y así evitar que se implemente el arreglo en memoria, evitando así romper la técnica de *pipelining* aplicada durante todo el proceso de inferencia, pues el acceso de una memoria se realiza de manera síncrona, permitiendo únicamente solo una lectura por ciclo de reloj.

```
1       data_type sum[CPO];
2       #pragma HLS ARRAY_PARTITION variable=sum complete
3       for (int cpo=0; cpo < CPO; cpo++) {
4           #pragma HLS UNROLL
5           sum[cpo] = 0.0;
6       }
7
8       weight = weights_in.read();
```

Posteriormente, se lee la matriz de pesos desde la cola FIFO creada explícitamente para leer de memoria los pesos y implementarlos mediante registros en la FPGA.

Luego, en esta parte del proceso de la convolución, se recibe por cada iteración de ejecución, la región de la imagen sobre la que se tiene que aplicar la convolución. Es por ello, que en este módulo el número de iteraciones es diferente al de

los módulos anteriores, puesto que sólo se obtiene del módulo `CVT` el filtro cuando se ha leído este mismo entero.

Finalmente, al final de cada iteración se vuelve a inicializar el arreglo `sum`, para así no alterar el resultado de las operaciones en iteraciones posteriores. Y por último, se escribe en la cola FIFO de salida, el resultado de la convolución sobre la región de la imagen en ejecución.

```
    for (int ho=0; ho < HO; ho++) {
        for (int wo=0; wo < WO; wo++) {
            #pragma HLS PIPELINE

            data_in = cvt_in.read();

            for (int cpi=0; cpi < CPI; cpi++) {
                #pragma HLS UNROLL
                for (int k=0; k < CONV_KW * CONV_KH; k++) {
                    #pragma HLS UNROLL
                    for (int cpo=0; cpo < CPO; cpo++) {
                        #pragma HLS UNROLL
                        sum[cpo] += data_in.kernel[k].pixel[cpi] *
                            weight.kernel[k].pixel[cpo];
                    }
                }
            }

            for (int cpo=0; cpo < CPO; cpo++) {
                #pragma HLS UNROLL
                out_pix.pixel[cpo] = sum[cpo];
                sum[cpo] = 0.0;
            }
            out << out_pix;
        }
    }
```

4.4.4. Módulo *ADD*

Y por último, en la última etapa integrada en el proceso de la convolución se añade el *bias*. El *bias* es un valor constante para cada canal de la imagen de salida. Por ejemplo, para una convolución de una imagen resultante con ocho canales de salida, se requiere un total de ocho *bias* distintos, uno por cada canal de la imagen.

```
    bias = bias_in.read();
    for (int hi=0; hi < HO; hi++) {
        for (int wi=0; wi < WO; wi++) {
            #pragma HLS PIPELINE II=1
            data_in = in.read();

            for (int cpo=0; cpo < CPO; cpo++) {
                #pragma HLS UNROLL
                data_out.pixel[cpo] = bias.pixel[cpo] + data_in.pixel[
                    cpo];
            }
            out << data_out;
        }
    }
```

Cómo se aprecia en el código anterior, primero se lee desde la cola el *bias* que se ha pasado desde la máquina *host*, se añade a cada canal de la imagen su *bias* correspondiente, y finalmente se le pasa a la cola de salida el valor final de la convolución.

4.5 Implementación de la capa de activación

En esta capa intermedia entre la convolucional y la capa de submuestreo, se realiza la función de activación, la cuál sirve para poder normalizar los datos y establecer unos determinados umbrales, con el fin de asegurar que las características de la imagen se mantienen dentro de las necesidades del proceso de inferencia diseñado.

En el diseño del proceso de inferencia realizado en el trabajo se han aplicado las dos funciones de activación más comunes:

- **ReLU (*Rectified Linear Unit)***

```
1    static data_type fn_relu(data_type x) {
2        if (x < 0) return (data_type) 0;
3        return x;
4    }
```

Esta función es simple debido en parte a que la propia función de activación lo es, si el parámetro de la función es menor que cero, esta retorna cero, en caso contrario devuelve el mismo valor.

- **Sigmoide**

```
1    #ifdef AP_FIXED
2        return data_type{1.0f / (1.0f + hls::exp(-x.to_float()))};
3    #endif
4    #ifdef FLOAT_TYPE
5        return 1 / (1 + hls::exp(-x));
6    #else
7        return 1 / (1 + hls::exp(-x.to_float()));
8    #endif
```

La implementación de esta función ya tiene un aumento en la complejidad, debido a las distintas implementaciones que se han tenido que hacer para cada tipo de datos usado. En esta función de activación se usa la librería de operaciones matemáticas de HLS, la cual ofrece una optimización considerable en la operación matemática exponencial.

4.6 Implementación de la capa de muestreo

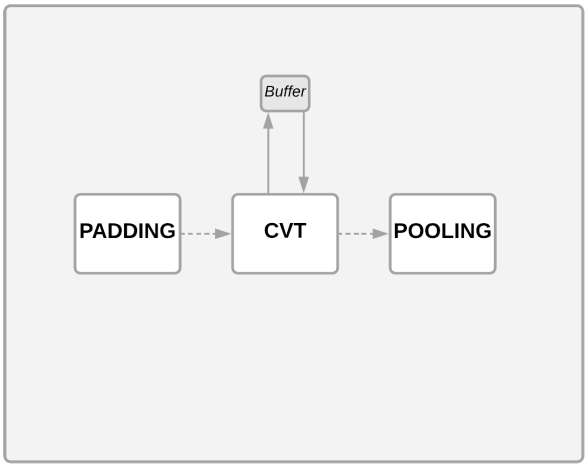

Figura 4.4: Flujo de la capa de muestreo

Por otra parte, en esta sección se va a detallar la implementación de la capa de muestreo, la cuál acepta tanto el muestreo de máximos (*maxpooling*) como el muestreo de medias (*avgpooling*).

Esta capa de la red neuronal está formada por tres etapas implementadas mediante la técnica de *pipelining*. Tal como se aprecia en la figura 4.4, se leen los datos que se han obtenido de la anterior capa y se procesan por cada una de las siguientes etapas:

- `PADDING`. Recibe los datos que se han obtenido a partir de la capa convolucional y la función de activación, y aplica los valores de *padding* establecidos por la configuración. Este módulo es el mismo que el descrito para la capa convolucional pero con los tipos de datos y parámetros específicos para la capa de la *pooling*.

- `CVT`. Una vez se ha ejecutado la etapa anterior, esta etapa recibe el píxel y lo almacena en un buffer temporal en memoria. Hechas las operaciones oportunas y una vez se obtenga la ventana de filtro del tamaño establecido en la configuración, se pasa esta a la siguiente etapa encargada de realizar la operación de *pooling*. Al igual, que la anterior etapa, esta tiene el mismo comportamiento que el descrito en la capa convolucional pero con los parámetros necesarios para la *pooling*.

- `POOLING`. La etapa final, recibe por la cola FIFO de entrada ventanas de filtro para así aplicar el muestreo por máximos o por medias, según se haya establecido al lanzar la ejecución. Vamos a pasar a describirla con detalle en el siguiente apartado.

4.6.1. Módulo *POOLING*

En este último módulo del proceso de inferencia implementado, se reciben los datos de los módulos *padding* y *cvt* de la capa de muestreo.

En el módulo de la capa de submuestreo se ha implementado de forma genérica, permite un proceso de inferencia con la capa de submuestreo por máximos (*maxpooling*) y por medias (*avgpooling*). Además, está diseñado para aceptar cualquier tamaño de imagen, limitado en primer lugar por la configuración del módulo *CVT* y en segundo lugar por las propias limitaciones de espacio de la FPGA.

```
static void pooling(uint HO, uint WO, int enable_maxpooling, int
    enable_avgpooling, hls::stream<Pool_t> &input, hls::stream<Output_t
    > &output) {
    int size_out, size_pool;
    Pool_t frame;
    Output_t pix;

    size_out = HO * WO;
    size_pool = POOL_KH * POOL_KW;

    for (int i=0; i < size_out; i++) {
        #pragma HLS PIPELINE II=1

        frame = input.read();
```

En el fragmento de código superior, por una parte se realizan `N` iteraciones, siendo `N` el número de píxeles de la imagen resultante del proceso de inferencia y se lee la región de la imagen sobre la que se aplicará la operación de submuestreo.

```
        for (int cpo=0; cpo < CPO; cpo++) {
            #pragma HLS UNROLL
            data_type maxpool_value = MIN_VALUE;
            data_type avgpool_value = 0;
            for (int k=0; k < size_pool; k++) {
                #pragma HLS UNROLL

                data_type value = frame.pool[k].pixel[cpo];

                if (value > maxpool_value) maxpool_value = value;
                avgpool_value += value;
            }
            avgpool_value /= size_pool;

            pix.pixel[cpo] = enable_maxpooling ? maxpool_value :
                enable_avgpooling ? avgpool_value : frame.pool[0].pixel
                [cpo];
        }
        output << pix;
    }
}
```

Finalmente, en la última parte del módulo se calculan tanto el muestreo por máximos como el muestreo por medias, y será al final de cada iteración donde se comprueba que tipo de muestreo se aplica y se obtiene su respectivo valor.

Se ha abordado esta solución debido a que resultaba más práctico realizar pruebas de varias ejecuciones sin la necesidad de compilar para cambiar el tipo de muestreo, reduciendo así el tiempo de diseño y experimentación. Aunque cabe destacar que resulta más óptimo en cuánto a uso de espacio de FPGA realizar una implementación más específica que una más genérica (cómo la descrita en este capítulo con un muestreo por máximos y otro por medias), puesto que la herramienta HLS debe aplicar distintas operaciones intermedias que aumentan el uso de recursos requerido.

CAPÍTULO 5
Resultados y análisis de uso de recursos

5.1 Resultados de ejecución

En este capítulo se mostrarán los tiempos de ejecución del proceso de inferencia diseñado, con distintos tamaños de imagen, para demostrar el principal objetivo del trabajo: conseguir tiempos de ejecución similares independientmente de la dimensionalidad de la red neuronal. Todas las pruebas de ejecución se han realizado con un tamaño fijo de imagen de `256 x 256`.

La toma de tiempos mostrada en la tabla 5.1 se han obtenido a partir de la media resultante de cinco ejecuciones totales.

En la gráfica 5.1, se observa cómo se ha conseguido el principal objetivo del trabajo, pues se consiguen tiempos de ejecución muy parecidos independientemente de la dimensionaliadad de la red, con la excepción cuándo la red dispone de 16 canales de entrada y salida, que aumenta ligeramente el tiempo de ejecución al aumentar el tamaño de la red. Este incremento viene dado principalmente en la transferencia de datos desde la máquina host a la FPGA.

5.2 Análisis de uso

Para la toma de uso de espacio del proceso de inferencia diseñado, se ha tomado la implementación con todas las capas descritas anteriormente. Se han realizado toma de recursos con dos tipos de datos distintos, por una parte el tipo de precisión arbitraria de coma fija (*AP Fixed*) y por otra parte el tipo de precisión

CPI	CPO	ap_fixed	ap_int
1	1	0.44	0.44
4	4	0.46	0.43
8	8	0.42	0.47
16	16	0.7	0.58

Tabla 5.1: Tiempos de ejecución expresado en milisegundos

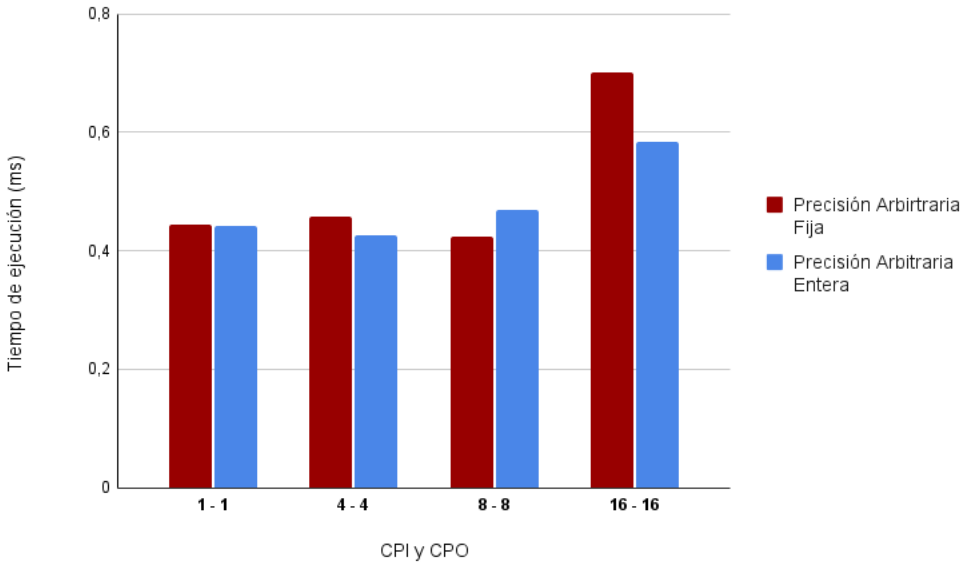

Figura 5.1: Tiempos de ejecución

arbitraria entero (*AP Int*), descartándose el tipo primitivo de C `float` debido a su gran tamaño con el considerable aumento de recursos que eso conlleva.

Primero vamos a mostrar para distintos canales de imagen de entrada y salida con el tipo de precisión arbitraria de coma fija, detallando el uso de cada uno de los componentes de la FPGA:

5.2.1. Uso con precisión arbitraria fija

La siguiente tabla (5.2), muestra el número de componentes usados de la FPGA para distintos valores de canales de imagen:

CPI	CPO	LUTs	FFs	BRAM	DSP
1	1	22310	23361	9	49
4	8	44485	52439	45	328
8	16	74998	83818	88	1192
16	16	101591	106649	112	2344

Tabla 5.2: Uso de recursos con Precisión Arbitraria Fija

En la siguiente tabla (5.3) se muestran los distintos porcentajes de uso de recursos de la *SLR-1* (la región lógica usada por defecto). Cómo se aprecia, en el diseño realizado se hace un buen uso de los componentes con la única excepción de los *DSPs* que incluso superan el número máximo de estos en la región lógica, precisando de un mayor número requiriendo así activar otra región lógica para ello.

CPI	CPO	LUTs	FFs	BRAM	DSP
1	1	6.3	3.2	1.4	2.2
4	8	12.2	6.5	5.1	8.1
8	16	19.7	10.5	10.1	27.2
16	16	28.6	14.7	17.6	103.5

Tabla 5.3: Porcentaje de uso de recursos con Precisión Arbitraria Fija

5.2.2. Uso con precisión arbitraria entera

La siguiente tabla (5.4), muestra el número de componentes usados de la FPGA para distintos valores de canales de imagen:

CPI	CPO	LUTs	FFs	BRAM	DSP
1	1	20509	20588	9	45
4	8	39661	34935	36	200
8	16	78207	51899	72	680
16	16	118484	64489	96	1320

Tabla 5.4: Uso de recursos con Precisión Arbitraria Entera

Por otro ladao, la siguiente tabla (5.5) muestra los distintos porcentajes de uso de recursos de la *SLR-1* (la región lógica usada por defecto). Cómo se aprecia, en el diseño realizado con precisión arbitraria entera se reducen el número de registros y de memorias, debido a la reducción del tamaño del tipo de dato, igual ocurre con el uso de los *DSPs*. Por otro lado, el uso de *LUTs* se ve incrementado debido a un mayor número de operaciones intermedias que se deben realizar en este tipo de datos.

CPI	CPO	LUTs	FFs	BRAM	DSP
1	1	5.8	2.9	1.4	2
4	8	10.1	4.5	4.1	5.3
8	16	17.3	6.6	8.2	15.9
16	16	33.4	8.9	15.1	58.3

Tabla 5.5: Procentaje de uso de recursos con Precisión Arbitraria Entera

La gráfica 5.2 muestra la comparativa de uso de las *Lookup Tables* con el tipo de precisión arbitraria fija de 16 bits frente al tipo de precisión arbitraria entera de 8 bits. Se observa como a partir de 16 canales de imagen, el tipo entero necesita de un mayor uso de LUTs, debido a las operaciones intermedias que se deben realizar para mantener la consistencia, debido a que el tipo entero implementado es de únicamente 8 bits.

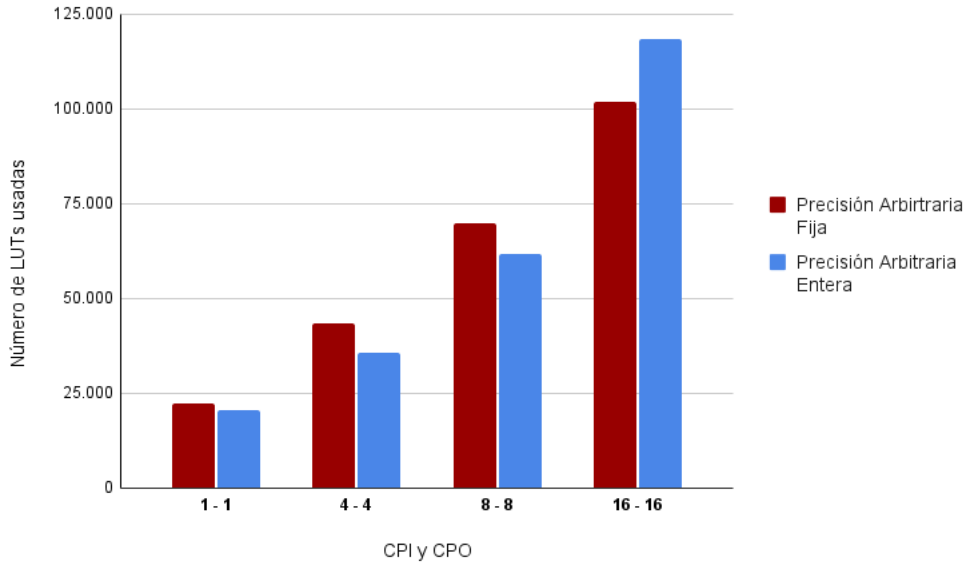

Figura 5.2: Uso de LUTs

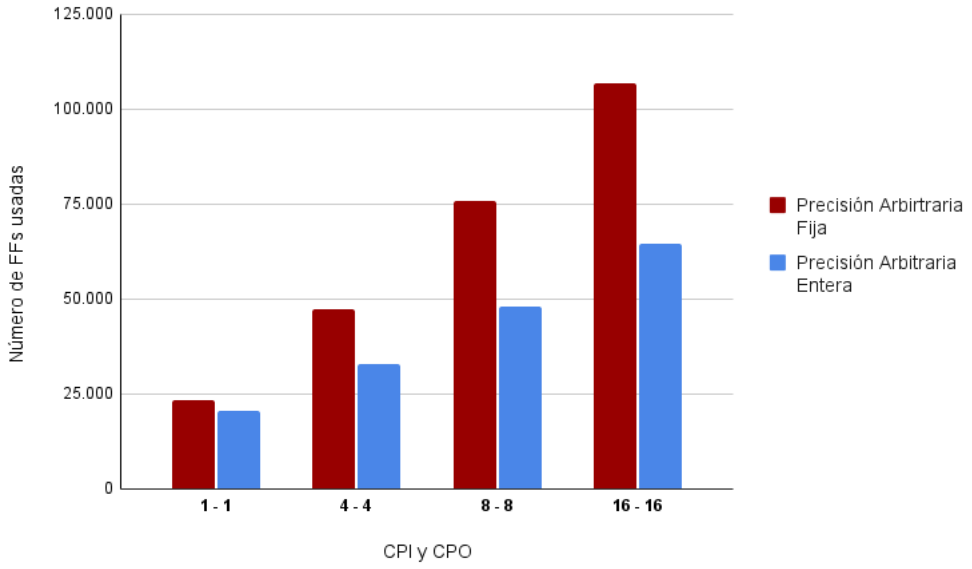

Figura 5.3: Uso de registros biestables

Por otro lado en la gráfica 5.3 se muestra la comparativa en el uso de registros de la FPGA. Como se observa, el diseño con el tipo arbritario fijo requiere de un mayor uso de registros biestables, debido al tamaño del tipo de dato (16 bits frente a los 8 bits del tipo arbritario entero).

En el caso de las BRAM, como se observa en la gráfica 5.4 se requiere un mayor uso de las *Block RAMs* por parte del tipo arbritario fijo, aunque no es demasiada la diferencia de uso entre las dos implementaciones.

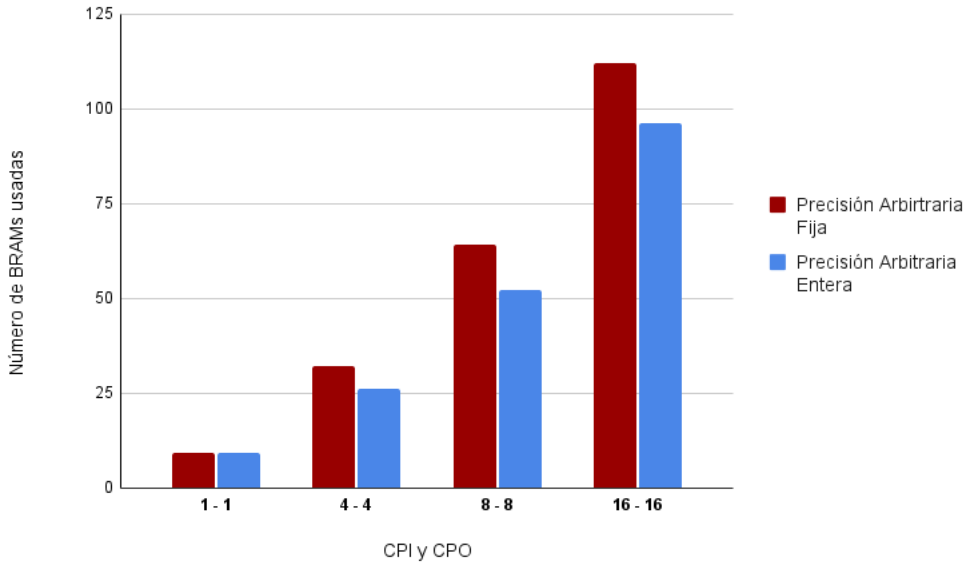

Figura 5.4: Uso de BRAMs

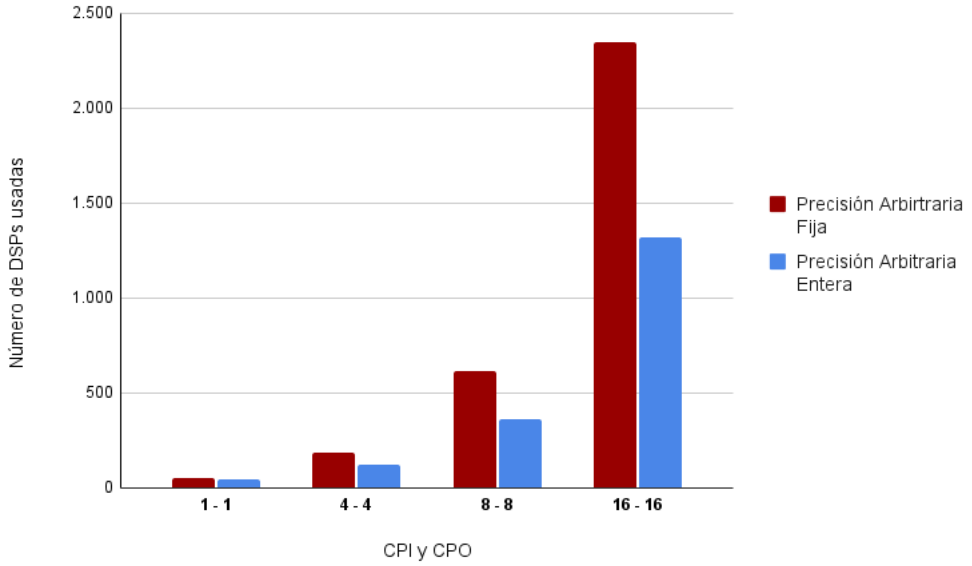

Figura 5.5: Uso de DSPs

Y por último en la gráfica 5.5 se muestra el uso de los DSPs, usados en su gran mayoría como multiplicadores para realizar las operaciones requeridas en el módulo *MUL* de la capa convolucional. Añadir que el uso de los DSPs en el tipo arbitrario fijo ha superado al disponible en una SLR, por lo que la FPGA necesitaría activar otra SLR incrementando así el consumo de la misma.

CAPÍTULO 6
Conclusiones y trabajo futuro

6.1 Conclusiones

Gracias al desarrollo de este proyecto se ha aprendido y profundizado en el funcionamiento de las redes neuronales de manera más directa, sin abstracciones, y los distintos problemas que se encuentran al trabajar en algoritmos de perceptrón multicapa.

Se ha conseguido desarrollar una capa de las redes neuronales convolucionales, como la capa de submuestreo, aplicando distintas técnicas de programación mediante el uso de directivas de optimización.

Por otro lado, se ha aprendido a trabajar en FPGAs comprendiendo su funcionamiento y aprendiendo a realizar la programación de las mismas, cómo se programa y realizar experimentaciones con la misma. Además, también se ha aprendido a aplicar distintas herramientas y tecnologías que permiten el uso de la FPGA desde otra perspectiva.

6.2 Trabajo futuro

Alguno de los puntos a destacar en la continuación de este proyecto son los siguientes:

- Aplicar otro tipo de aproximación para el tipo de datos usado en el proceso de inferencia, que permita así poder implementar sobre la FPGA grandes redes neuronales, como por ejemplo para la segmentación de imagenes, caracterizadas por requerir grandes redes neuronales (del orden de hasta 512 canales por imagen).
- Aunque en este proyecto se han aplicado las funciones de activación más comunes, hay otras muy usadas en otros ámbitos, que se podría explorar el uso de ellas sobre la FPGA.
- Añadir al proceso de inferencia la capacidad de poder ejecutar varias imágenes simultáneamente, pudiendo así lograr el ancho de banda capaz de lograr la FPGA como se ha mencionado durante el trabajo.

Bibliografía

[1] Bora Tar. *Digital System Design with FPGA Implementation Using Verilog and VHDL.* McGraw-Hill Education, primera edición, 2017

[2] David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, quinta edición, 2011

[3] Explicación intuitiva de las redes neuronales convolucionales (CNN). https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

[4] Aprende Deep Learning junto a Keras en 10 minutos y crea tu primera Convolutional Neural Network https://medium.com/dreamlearning/aprende-deep-learning-en-10-minutos-e4e9e8950cd8

[5] *Get Moving with Alveo.* https://github.com/xilinx/get_moving_with_alveo

[6] Alveo U200 and U250 Data Center Accelerator Cards Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf

[7] High-Level Synthesis (HLS). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf

[8] Directivas HLS. https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/okr1504034364623-1.html

[9] Tipos de datos de precisión arbitraria - HLS. https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/ogi1585574074269.html

[10] Objetivos de desarrollo sostenible - Naciones Unidas. https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/

[11] *Experimental framework to explore deep neural network inference.* https://finn-hlslib.readthedocs.io/en/latest/index.html

[12] GIT https://git-scm.com/about

[13] GAP - Grupo de Arquitecturas Paralelas http://www.gap.upv.es/

[14] Xilinx - *Floorplanning Method* https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/Floorplanning_Methodology_Guide.pdf

[15] *Forward Propagation, Backward Propagation, and Computational Graphs* https://d2l.ai/chapter_multilayer-perceptrons/backprop.html

[16] Recurrent Neural Networks. Remembering what's important https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce

APÉNDICE A
Objetivos de desarrollo sostenible

Los objetivos de desarrollo sostenible (ODS) son un marco de adopción de objetivos globales que se aprobó en consenso por la ONU. [10] El principal objetivo de este marco es el de conseguir erradicar la probreza, proteger el planeta y asegurar la prosperidad. En total, existen un total de 17 objetivos comunes, cada uno de ellos tiene metas específicas a cumplir en un plazo máximo de 15 años:

1. Fin de la pobreza
2. Hambre Cero
3. Salud y Bienestar
4. Educación de calidad
5. Igualdad de género
6. Agua limpia y saneamiento
7. Energía asequible y no contaminante
8. Trabajo decente y crecimiento económico
9. Reducción de las desigualdades
10. Ciudades y comunidades sostenibles
11. Producción y consumo responsables
12. Acción por el clima
13. Vida submarina
14. Vida de ecosistemas terrestres
15. Paz, justicia e instituciones sólidas
16. Alianzas para lograr los objetivos

Este trabajo de fin de grado tiene relación directa con los siguiente objetivos de desarrollo sostenible:

ODS-9. Industria, Innovación e Infraestructura La optimización de los algoritmos desarrollados y con una mejora de los mismos en el futuro, permite mejorar la industria actual, reduciendo costes en todo el sector y ofreciendo una mayor eficiencia.

ODS-11. Ciudades y comunidades sostenibles Mediante el uso de algoritmos optimizados aplicados a las FPGA se consigue una reducción en el consumo eléctrico, permitiendo una infraestructura más sostenible.

ODS-13. Acción por el clima Al igual que hemos mencionado anteriormente, una reducción del consumo eléctrico resulta más beneficio para el medio ambiente, ofreciendo así unas herramientas que apliquen estos algoritmos más sostenibles para el medioambiente.

Figura A.1: Objetivos de Desarrollo Sostenible

APÉNDICE B
Código net.cpp

```cpp
#include <hls_stream.h>
#include <hls_math.h>
#include <stdio.h>
#include <stdlib.h>

#include "types.h"
#include "parameters.h"
#include "utils.h"

extern "C" {

/*
* Reads input data ---> to the stream.
* @param HI Height of input image
* @param WI Width of input image
* @param ptr_in Pointer to the input image
* @param out Output stream
*/
static void read_data(uint HI, uint WI, Input_t *ptr_in, hls::stream<
    Input_t> &out) {

    int size = HI * WI;
    for (int i=0; i < size; i++) {
        #pragma HLS PIPELINE II=1
        out << ptr_in[i];
    }
}

/*
* CONV LAYER
*/

static void read_bias(data_type *ptr_bias, hls::stream<Output_t> &out)
    {
    Output_t bias;
    for (int i=0; i < CPO; i++) {
        #pragma HLS UNROLL

        bias.pixel[i] = ptr_bias[i];
    }
    out << bias;
}

```

```
42 static void read_weights(data_type *ptr_weights, hls::stream<Conv_out_t
       > &out) {
43     int k, cpo=0;
44     Conv_out_t weight;
45 
46     int size = CONV_KH * CONV_KW * CPO * CPI;
47 
48     for (int i=0; i < size; i++) {
49         #pragma HLS UNROLL
50 
51         weight.kernel[k++].pixel[cpo] = ptr_weights[i];
52 
53         if (k == (CONV_KH * CONV_KH)) { k=0; cpo++; }
54     }
55 }
56 
57 static void padding_conv(uint HI, uint WI, uint PH, uint PW, hls::
       stream<Input_t> &in, hls::stream<Input_t> &out) {
58     Input_t pix_zero;
59     Input_t pix_out;
60     int cols;
61     int size_row;
62 
63     for (int cpi=0; cpi < CPI; cpi++) pix_zero.pixel[cpi] = 0.f; // HLS
           automatically applies UNROLL
64 
65     for (int hi=0; hi < HI; hi++) {
66         for (int wi=0; wi < WI; wi++) {
67             int check_ph = PH - 1;
68             int check_pw = PW - 1;
69 
70             int img_col = (hi > (check_ph)) && (hi < (HI - PH));
71             int img_row = (wi > (check_pw)) && (wi < (WI - PW));
72 
73             if (img_col && img_row) pix_out = in.read();
74             else pix_out = pix_zero;
75 
76             out << pix_out;
77         }
78     }
79 }
80 
81 static void cvt_conv(uint HI, uint WI, hls::stream<Input_t> &input, hls
       ::stream<Conv_in_t> &output) {
82     int send_frame;
83     Conv_in_t frame;
84     Input_t pixel;
85 
86     Input_t pix_buff[CONV_KH];
87     #pragma HLS ARRAY_PARTITION variable=pix_buff complete
88     Input_t pixels[CONV_KH * CONV_KW];
89     #pragma HLS ARRAY_PARTITION variable=pixels complete
90     Input_t buffer[CONV_KH][MAX_WIDTH];
91     #pragma HLS ARRAY_PARTITION variable=buffer dim=1
92 
93     for (int hi=0; hi < HI; hi++) {
94         for (int wi=0; wi < WI; wi++) {
95             #pragma HLS PIPELINE II=1
96 
```

```
97              pixel = input.read();
98
99              int row_buffer = (hi %CONV_KH);
100             buffer[row_buffer][wi] = pixel;
101
102             int shift_frame = (hi > (CONV_KH - 2)) & (wi > (CONV_KW -
                    1));
103             send_frame = (hi > (CONV_KH - 2)) & (wi > (CONV_KW - 2));
104             send_frame = send_frame & ((hi%CONV_SH == (CONV_SH - 1)) &&
                     (wi%CONV_SW == (CONV_SW - 1)));
105
106             for (int i=0; i < CONV_KH; i++) pix_buff[i] = buffer[i][wi
                    ];
107
108             for (int kh=0; kh < CONV_KH; kh++) {
109                 #pragma HLS UNROLL
110                 int row = (hi + kh) %CONV_KH;
111
112                 for (int kw=0; kw < (CONV_KW - 1); kw++) {
113                     #pragma HLS UNROLL
114                     int index = kh + kw;
115
116                     if (shift_frame) pixels[index] = pixels[index + 1];
117                     else if (wi==kw) pixels[index] = pix_buff[row];
118                 }
119                 int addr = (kh+1) * CONV_KH - 1;
120                 if (row_buffer == kh) pixels[addr] = pix_buff[kh];
121             }
122
123             if (send_frame) {
124                 for (int i=0; i < (CONV_KH * CONV_KW); i++) frame.
                        kernel[i] = pixels[i];
125
126                 output << frame;
127             }
128         }
129     }
130 }
131
132 static void mul(uint HO, uint WO, hls::stream<Conv_in_t> &cvt_in, hls::
        stream<Conv_out_t> &weights_in, hls::stream<Output_t> &out) {
133     Conv_out_t weight;
134     Conv_in_t      data_in;
135     Output_t       out_pix;
136     data_type      sum[CPO];
137
138     #pragma HLS ARRAY_PARTITION variable=sum complete
139     #pragma HLS ARRAY_PARTITION variable=weight dim=0 complete
140
141     for (int cpo=0; cpo < CPO; cpo++) {
142         #pragma HLS UNROLL
143         sum[cpo] = 0.0;
144     }
145
146     weight = weights_in.read();
147
148     for (int ho=0; ho < HO; ho++) {
149         for (int wo=0; wo < WO; wo++) {
150             #pragma HLS PIPELINE
```

```
151
152             data_in = cvt_in.read();
153
154             for (int cpi=0; cpi < CPI; cpi++) {
155                 #pragma HLS UNROLL
156                 for (int k=0; k < CONV_KW * CONV_KH; k++) {
157                     #pragma HLS UNROLL
158                     for (int cpo=0; cpo < CPO; cpo++) {
159                         #pragma HLS UNROLL
160                         sum[cpo] += data_in.kernel[k].pixel[cpi] *
                                weight.kernel[k].pixel[cpo];
161                     }
162                 }
163             }
164
165             for (int cpo=0; cpo < CPO; cpo++) {
166                 #pragma HLS UNROLL
167                 out_pix.pixel[cpo] = sum[cpo];
168                 sum[cpo] = 0.0;
169             }
170             out << out_pix;
171         }
172     }
173 }
174
175 static void add(uint HO, uint WO, hls::stream<Output_t> &in, hls::
    stream<Output_t> &bias_in, hls::stream<Output_t> &out) {
176     Output_t bias;
177     Output_t data_in;
178     Output_t data_out;
179
180     bias = bias_in.read();
181
182     for (int hi=0; hi < HO; hi++) {
183         for (int wi=0; wi < WO; wi++) {
184             #pragma HLS PIPELINE II=1
185             data_in = in.read();
186
187             for (int cpo=0; cpo < CPO; cpo++) {
188                 #pragma HLS UNROLL
189                 data_out.pixel[cpo] = bias.pixel[cpo] + data_in.pixel[
                        cpo];
190             }
191             out << data_out;
192         }
193     }
194 }
195
196 /*
197 * ACTIVATION
198 */
199
200 static data_type fn_sigmoid(data_type x) {
201     #ifdef AP_FIXED
202         return data_type{1.0f / (1.0f + hls::exp(-x.to_float()))};
203     #endif
204     #ifdef FLOAT_TYPE
205         return 1 / (1 + hls::exp(-x));
206     #else
```

```
207         return 1 / (1 + hls::exp(-x.to_float()));
208     #endif
209 }
210 
211 static data_type fn_relu(data_type x) {
212     if (x < 0) return (data_type) 0;
213     return x;
214 }
215 
216 static void activation(uint HI, uint WI, hls::stream<Output_t> &in, hls
    ::stream<Output_t> &out) {
217     Output_t data_in;
218     Output_t data_out;
219 
220     for (int hi=0; hi < HI; hi++) {
221         for (int wi=0; wi < WI; wi++) {
222             #pragma HLS PIPELINE II=1
223 
224             data_in = in.read();
225             for (int cpo = 0; cpo < CPO; cpo++) {
226                 #pragma HLS UNROLL
227 
228                 #ifdef ENABLE_RELU
229                     data_out.pixel[cpo] = fn_relu(data_in.pixel[cpo]);
230                 #endif
231 
232                 #ifdef ENABLE_SIGMOID
233                     data_out.pixel[cpo] = fn_sigmoid(data_in.pixel[cpo
                        ]);
234                 #endif
235             }
236 
237             out << data_out;
238         }
239     }
240 }
241 
242 
243 /*
244 * POOLING LAYER
245 */
246 
247 static void padding_pool(uint HI, uint WI, uint PH, uint PW, hls::
    stream<Output_t> &in, hls::stream<Output_t> &out) {
248 
249     Output_t pix_zero;
250     Output_t pix_out;
251     int cols;
252     int size_row;
253 
254     #pragma HLS ARRAY_PARTITION variable=pix_zero dim=0 complete
255 
256     for (int cpi=0; cpi < CPO; cpi++) pix_zero.pixel[cpi] = 0.f; // HLS
        automatically applies UNROLL
257 
258     for (int hi=0; hi < HI; hi++) {
259         for (int wi=0; wi < WI; wi++) {
260             int check_ph = PH - 1;
261             int check_pw = PW - 1;
```

```
262
263             int img_col = (hi > (check_ph)) && (hi < (HI - PH));
264             int img_row = (wi > (check_pw)) && (wi < (WI - PW));
265
266             if (img_col && img_row) pix_out = in.read();
267             else pix_out = pix_zero;
268
269             out << pix_out;
270         }
271     }
272 }
273
274 static void cvt_pool(uint HI, uint WI, hls::stream<Output_t> &input,
    hls::stream<Pool_t> &output) {
275     int send_frame;
276     Pool_t frame;
277     Output_t pixel;
278
279     Output_t pix_buff[POOL_KH];
280     #pragma HLS ARRAY_PARTITION variable=pix_buff complete
281     Output_t pixels[POOL_KH * POOL_KW];
282     #pragma HLS ARRAY_PARTITION variable=pixels complete
283     Output_t buffer[POOL_KH][MAX_WIDTH];
284     #pragma HLS ARRAY_PARTITION variable=buffer dim=1
285
286     for (int hi=0; hi < HI; hi++) {
287         for (int wi=0; wi < WI; wi++) {
288             #pragma HLS PIPELINE II=1
289
290             pixel = input.read();
291
292             int row_buffer = (hi %POOL_KH);
293             buffer[row_buffer][wi] = pixel;
294
295             int shift_frame = (hi > (POOL_KH - 2)) & (wi > (POOL_KW -
                1));
296             send_frame = (hi > (POOL_KH - 2)) & (wi > (POOL_KW - 2));
297             send_frame = send_frame & ((hi%POOL_SH == (POOL_SH - 1)) &&
                 (wi%POOL_SW == (POOL_SW - 1)));
298
299             for (int i=0; i < POOL_KH; i++) pix_buff[i] = buffer[i][wi
                ];
300
301             for (int kh=0; kh < POOL_KH; kh++) {
302                 #pragma HLS UNROLL
303                 int row = (hi + kh) %POOL_KH;
304
305                 for (int kw=0; kw < (POOL_KW - 1); kw++) {
306                     #pragma HLS UNROLL
307                     int index = kh + kw;
308
309                     if (shift_frame) pixels[index] = pixels[index + 1];
310                     else if (wi==kw) pixels[index] = pix_buff[row];
311                 }
312                 int addr = (kh+1) * POOL_KH - 1;
313                 if (row_buffer == kh) pixels[addr] = pix_buff[kh];
314             }
315
316             if (send_frame) {
```

```
317                 for (int i=0; i < (POOL_KH * POOL_KW); i++) frame.pool[
                        i] = pixels[i];
318
319                 output << frame;
320             }
321         }
322     }
323 }
324
325 static void pooling(uint HO, uint WO, int enable_maxpooling, int
    enable_avgpooling, hls::stream<Pool_t> &input, hls::stream<Output_t
    > &output) {
326     int size_out, size_pool;
327     Pool_t frame;
328     Output_t pix;
329
330     size_out = HO * WO;
331     size_pool = POOL_KH * POOL_KW;
332
333     for (int i=0; i < size_out; i++) {
334         #pragma HLS PIPELINE II=1
335
336         frame = input.read();
337         for (int cpo=0; cpo < CPO; cpo++) {
338             #pragma HLS UNROLL
339
340             data_type maxpool_value = MIN_VALUE;
341             data_type avgpool_value = 0;
342             for (int k=0; k < size_pool; k++) {
343                 #pragma HLS UNROLL
344
345                 data_type value = frame.pool[k].pixel[cpo];
346
347                 if (value > maxpool_value) maxpool_value = value;
348                 avgpool_value += value;
349             }
350             avgpool_value /= size_pool;
351
352             pix.pixel[cpo] = enable_maxpooling ? maxpool_value :
                    enable_avgpooling ? avgpool_value : frame.pool[0].pixel
                    [cpo];
353         }
354         output << pix;
355     }
356 }
357
358 static void write_data(uint HO, uint WO, hls::stream<Output_t> &in,
    Output_t *output) {
359     int size = HO * WO;
360     for (int i=0; i < size; i++) {
361         #pragma HLS PIPELINE II=1
362
363         output[i] = in.read();
364     }
365 }
366
367 void k_net(uint HI, uint WI, uint PH, uint PW, uint enable_maxpooling,
    uint enable_avgpooling, Input_t *input, data_type *ptr_bias,
    data_type *ptr_weights, Output_t *output) {
```

```
368     #pragma HLS INTERFACE s_axilite port=WI bundle=control
369     #pragma HLS INTERFACE s_axilite port=HI bundle=control
370     #pragma HLS INTERFACE s_axilite port=PH bundle=control
371     #pragma HLS INTERFACE s_axilite port=PW bundle=control
372     #pragma HLS INTERFACE s_axilite port=enable_maxpooling bundle=
            control
373     #pragma HLS INTERFACE s_axilite port=enable_avgpooling bundle=
            control
374     #pragma HLS INTERFACE m_axi port=input offset=slave bundle=gmem
375     #pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem
376     #pragma HLS INTERFACE m_axi port=ptr_bias offset=slave bundle=gmem1
377     #pragma HLS INTERFACE m_axi port=ptr_weights offset=slave bundle=
            gmem2
378     #pragma HLS INTERFACE s_axilite port=return bundle=control
379
380     uint HI_PADD = (HI + 2 * PH);
381     uint WI_PADD = (WI + 2 * PW);
382
383     uint HO_CONV = (HI - CONV_KH + 2 * PH) / CONV_SH + 1;
384     uint WO_CONV = (WI - CONV_KW + 2 * PW) / CONV_SW + 1;
385
386     uint HI_PADD_POOL = (HO_CONV + 2 * PH);
387     uint WI_PADD_POOL = (WO_CONV + 2 * PW);
388
389     uint HO = (HO_CONV - POOL_KH + 2 * PH) / POOL_SH + 1;
390     uint WO = (WO_CONV - POOL_KW + 2 * PW) / POOL_SW + 1;
391
392     #pragma HLS data_pack variable=input
393     #pragma HLS data_pack variable=ptr_bias
394     #pragma HLS data_pack variable=ptr_weights
395     #pragma HLS data_pack variable=output
396
397     static hls::stream<Input_t> stream_in ("str_in");
398     static hls::stream<Output_t> stream_bias ("str_bias");
399     static hls::stream<Conv_out_t> stream_weights ("str_weights");
400     static hls::stream<Input_t> stream_padd_conv ("str_padd_conv");
401     static hls::stream<Conv_in_t> stream_cvt_conv ("str_cvt_conv");
402     static hls::stream<Output_t> stream_mul_conv ("str_mul_conv");
403     static hls::stream<Output_t> stream_conv_out ("str_conv_out");
404     static hls::stream<Output_t> stream_activation ("str_activation");
405     static hls::stream<Output_t> stream_padd_pool ("str_padd_pool");
406     static hls::stream<Pool_t> stream_cvt_pool ("str_pool_out");
407     static hls::stream<Output_t> stream_out ("str_conv_out");
408     #pragma HLS STREAM variable=stream_in depth=32
409     #pragma HLS STREAM variable=stream_bias depth=32
410     #pragma HLS STREAM variable=stream_weights depth=32
411     #pragma HLS STREAM variable=stream_padd_conv depth=32
412     #pragma HLS STREAM variable=stream_cvt_conv depth=32
413     #pragma HLS STREAM variable=stream_mul_conv depth=32
414     #pragma HLS STREAM variable=stream_conv_out depth=32
415     #pragma HLS STREAM variable=stream_activation depth=32
416     #pragma HLS STREAM variable=stream_padd_pool depth=32
417     #pragma HLS STREAM variable=stream_cvt_pool depth=32
418     #pragma HLS STREAM variable=stream_out depth=32
419
420
421     #pragma HLS DATAFLOW
422     read_data(HI, WI, input, stream_in);
423     read_bias(ptr_bias, stream_bias);
```

```
424         read_weights(ptr_weights, stream_weights);
425         padding_conv(HI_PADD, WI_PADD, PH, PW, stream_in, stream_padd_conv)
                ;
426         cvt_conv(HI_PADD, WI_PADD, stream_padd_conv, stream_cvt_conv);
427         mul(HO_CONV, WO_CONV, stream_cvt_conv, stream_weights,
                stream_mul_conv);
428         add(HO_CONV, WO_CONV, stream_mul_conv, stream_bias, stream_conv_out
                );
429         activation(HO_CONV, WO_CONV, stream_conv_out, stream_activation);
430         padding_pool(HI_PADD_POOL, WI_PADD_POOL, PH, PW, stream_activation,
                 stream_padd_pool);
431         cvt_pool(HI_PADD_POOL, WI_PADD_POOL, stream_padd_pool,
                stream_cvt_pool);
432         pooling(HO, WO, 1, 0, stream_cvt_pool, stream_out);
433         write_data(HO, WO, stream_out, output);
434     }
435 
436 }
```

APÉNDICE C
Código `test_net.cpp`

```
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <random>
#include <xcl2.hpp>
#include <vector>

#include "math.h"

#include "utils.h"
#include "parameters.h"

cl::Buffer buf;
cl::Context context;
cl::CommandQueue q;
cl::Program program;

int data_in_size;
int data_out_size;
int bias_size;
int weights_size;

int WIDTH_IN;
int HEIGHT_IN;
int WIDTH_CONV_OUT;
int HEIGHT_CONV_OUT;
int WIDTH_OUT;
int HEIGHT_OUT;
int enable_maxpool;
int enable_avgpool;
int PW;
int PH;

static void CPU_conv(data_type *input, data_type *output, data_type *
    bias, data_type *weights) {
    int size_out, index_h, index_w, index_k;
    int addr, addr_out;

    int w_offset = -PW;
    int h_offset = -PH;

    size_out = HEIGHT_CONV_OUT * WIDTH_CONV_OUT * CPO;
```

```
43      for (int i=0; i < size_out; i++) output[i] = 0.0f;
44
45      for (int cpi=0; cpi < CPI; cpi++) {
46          for (int cpo=0; cpo < CPO; cpo++) {
47              addr_out = 0;
48              for (int ho=0; ho < HEIGHT_CONV_OUT; ho++) {
49                  for (int wo=0; wo < WIDTH_CONV_OUT; wo++) {
50                      addr_out = cpo + (wo + ho * WIDTH_CONV_OUT);
51
52                      for (int kh=0; kh < CONV_KH; kh++) {
53                          for (int kw=0; kw < CONV_KW; kw++) {
54                              index_h = h_offset + (ho * CONV_SH) + kh;
55                              index_w = w_offset + (wo * CONV_SW) + kw;
56                              addr = cpi + (index_w + index_h * WIDTH_IN)
                                    ;
57                              int valid = (index_h >= 0 && index_h <
                                    HEIGHT_IN &&
58                                          index_w >= 0 && index_w <
                                              WIDTH_IN);
59
60                              data_type value = 0;
61                              if (valid) value = in[addr];
62
63                              index_k = kh + kw + (cpo * CONV_KH *
                                    CONV_KW) + (cpi * CONV_KH * CONV_KW);
64
65                              output[addr_out] += input[addr] * weights[
                                    index_k];
66                          }
67                      }
68                      output[addr_out] += bias[cpo];
69                  }
70              }
71          }
72      }
73  }
74
75  static void CPU_activation(data_type *input, data_type *output) {
76      int size_out; /* Input size is equal to output size */
77      size_out = HEIGHT_CONV_OUT * WIDTH_CONV_OUT * CPO;
78
79      for (int i=0; i < size_out; i++) {
80          #ifdef ENABLE_RELU
81              output[i] = input[i];
82              if (output[i] < 0) output[i] = 0;
83          #endif
84          #ifdef ENABLE_SIGMOID
85              #ifdef APF_TYPE
86                  output[i] = data_type{1.0f / (1.0f + exp(-input[i].
                        to_float()))};
87              #else
88                  output[i] = data_type{1.0f / (1.0f + exp(-input[i].
                        to_float()))};
89              #endif
90          #endif
91      }
92  }
93
```

```
94 static void CPU_pooling(data_type *input, data_type *output, int
       enable_maxpooling, int enable_avgpooling) {
95     int index_out, index_h, index_w;
96     int addr;
97     data_type maxpool_value = 0;
98     data_type avgpool_value = 0;
99 
100    int w_offset = -PW;
101    int h_offset = -PH;
102 
103    if ((enable_avgpooling ^ enable_avgpooling)) { fprintf(stderr, "[
           HOST | CPU_pooling()] No pooling layer was selected!\n"); exit
           (-1); }
104 
105    for (int cpi=0; cpi < CPO; cpi++)
106        for (int ho=0; ho < HEIGHT_OUT; ho++)
107            for (int wo=0; wo < WIDTH_OUT; wo++) {
108                index_out = (wo + (ho * WIDTH_OUT)) * CPO + cpi;
109 
110                for (int kw=0; kw < POOL_KW; kw++) {
111                    for (int kh=0; kh < POOL_KH; kh++) {
112                        index_h = h_offset + (ho * POOL_SH) + kh;
113                        index_w = w_offset + (wo * POOL_SW) + kw;
114                        addr = (index_w + (WIDTH_IN * index_h)) * CPI +
                               cpi;
115                        int valid = (index_h >= 0 && index_h <
                               HEIGHT_CONV_OUT &&
116                                     index_w >= 0 && index_w <
                                        WIDTH_CONV_OUT);
117 
118                        data_type value = -INFINITY;
119                        if (valid) value = in[addr];
120 
121                        if (value > maxpool_value) maxpool_value =
                               value;
122                        avgpool_value += value;
123                    }
124                }
125                avgpool_value /= (POOL_KW * POOL_KH);
126 
127                output[index_out] = enable_maxpooling ? maxpool_value :
                       enable_avgpooling ? avgpool_value : data_type{0.0f
                       };
128            }
129 }
130 
131 void net_host(data_type *input, data_type *output, data_type *bias,
       data_type *weights, int enable_relu, int enable_sigmoid, int
       enable_maxpooling, int enable_avgpooling) {
132 
133    int size_out = HEIGHT_CONV_OUT * WIDTH_CONV_OUT * CPO;
134    std::vector<data_type, aligned_allocator<data_type>> out_conv(
           size_out);
135 
136    CPU_conv(input, out_conv.data(), bias, weights);
137    CPU_activation(out_conv.data(), out_conv.data());
138    CPU_pooling(out_conv.data(), output, enable_maxpooling,
           enable_avgpooling);
139 }
```

```
140
141
142
143 uint64_t get_duration_ns(const cl::Event &event) {
144     cl_int err;
145     uint64_t nstimestart, nstimeend;
146     OCL_CHECK(err,
147               err = event.getProfilingInfo<uint64_t>(
                      CL_PROFILING_COMMAND_START,
148                                                      &nstimestart));
149     OCL_CHECK(err,
150               err = event.getProfilingInfo<uint64_t>(
                      CL_PROFILING_COMMAND_END,
151                                                      &nstimeend));
152     return (nstimeend - nstimestart);
153 }
154
155 void event_cb(cl_event event1, cl_int cmd_status, void *data) {
156         cl_int err;
157         cl_command_type command;
158         cl::Event event(event1, true);
159         OCL_CHECK(err, err = event.getInfo(CL_EVENT_COMMAND_TYPE, &
                    command));
160         cl_int status;
161         OCL_CHECK(err,
162                                                 err = event.getInfo(
                                                        CL_EVENT_COMMAND_EXECUTION_STATUS
                                                        , &status));
163         const char *command_str;
164         const char *status_str;
165         switch (command) {
166         case CL_COMMAND_READ_BUFFER:
167                 command_str = "buffer read";
168                 break;
169         case CL_COMMAND_WRITE_BUFFER:
170                 command_str = "buffer write";
171                 break;
172         case CL_COMMAND_NDRANGE_KERNEL:
173                 command_str = "kernel";
174                 break;
175         case CL_COMMAND_MAP_BUFFER:
176                 command_str = "kernel";
177                 break;
178         case CL_COMMAND_COPY_BUFFER:
179                 command_str = "kernel";
180                 break;
181         case CL_COMMAND_MIGRATE_MEM_OBJECTS:
182                 command_str = "buffer migrate";
183                 break;
184         default:
185                 command_str = "unknown";
186         }
187         switch (status) {
188         case CL_QUEUED:
189                 status_str = "Queued";
190                 break;
191         case CL_SUBMITTED:
192                 status_str = "Submitted";
193                 break;
```

```
194         case CL_RUNNING:
195                 status_str = "Executing";
196                 break;
197         case CL_COMPLETE:
198                 status_str = "Completed";
199                 break;
200         }
201         printf("[%s]: %s %s\n", reinterpret_cast<char *>(data),
               status_str,
202                                 command_str);
203         fflush(stdout);
204 }
205
206
207 void set_callback(cl::Event event, const char *queue_name) {
208   cl_int err;
209   OCL_CHECK(err, err = event.setCallback(CL_COMPLETE, event_cb, (void
          *) queue_name));
210 }
211
212 uint64_t net_fpga(
213     std::vector<data_type, aligned_allocator<data_type>> &input,
214     std::vector<data_type, aligned_allocator<data_type>> &output,
215         std::vector<data_type, aligned_allocator<data_type>> &bias,
216         std::vector<data_type, aligned_allocator<data_type>> &weights,
217     std::string &binaryFile
218 ) {
219   cl_int err;
220
221   int size_in  = data_in_size  * sizeof(data_type);
222   int size_out = data_out_size * sizeof(data_type);
223
224   auto devices = xcl::get_xil_devices();
225     auto device = devices[0];
226
227   OCL_CHECK(err, cl::Context context(device, NULL, NULL, NULL, &err));
228     OCL_CHECK(err, cl::CommandQueue q(context, device,
           CL_QUEUE_PROFILING_ENABLE, &err));
229     auto device_name = device.getInfo<CL_DEVICE_NAME>();
230
231   auto fileBuf = xcl::read_binary_file(binaryFile);
232     cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};
233     devices.resize(1);
234     OCL_CHECK(err, cl::Program program(context, devices, bins, NULL, &
          err));
235
236     OCL_CHECK(err, cl::Kernel kernel(program, KERNEL_NAME, &err));
237
238   OCL_CHECK(err,
239               cl::Buffer buffer_in(context,
240                                    CL_MEM_USE_HOST_PTR |
                                         CL_MEM_READ_ONLY,
241                                    size_in,
242                                    input.data(),
243                                    &err));
244     OCL_CHECK(err,
245               cl::Buffer buffer_out(context,
246                                    CL_MEM_USE_HOST_PTR |
                                         CL_MEM_WRITE_ONLY,
```

```
247                                     size_out,
248                                     output.data(),
249                                     &err));
250
251     int size_bias = bias_size * sizeof(data_type);
252     OCL_CHECK(err,
253               cl::Buffer buffer_bias(context,
254                                      CL_MEM_USE_HOST_PTR |
                                           CL_MEM_READ_ONLY,
255                                      size_bias,
256                                      bias.data(),
257                                      &err));
258
259     int size_weights = weights_size * sizeof(data_type);
260     OCL_CHECK(err,
261               cl::Buffer buffer_weights(context,
262                                         CL_MEM_USE_HOST_PTR |
                                              CL_MEM_READ_ONLY,
263                                         size_weights,
264                                         weights.data(),
265                                         &err));
266
267
268   //Set the kernel arguments
269     int narg = 0;
270     OCL_CHECK(err, err = kernel.setArg(narg++, HEIGHT_IN));
271     OCL_CHECK(err, err = kernel.setArg(narg++, WIDTH_IN));
272     OCL_CHECK(err, err = kernel.setArg(narg++, PH));
273     OCL_CHECK(err, err = kernel.setArg(narg++, PW));
274     OCL_CHECK(err, err = kernel.setArg(narg++, enable_maxpool));
275     OCL_CHECK(err, err = kernel.setArg(narg++, enable_avgpool));
276     OCL_CHECK(err, err = kernel.setArg(narg++, buffer_in));
277     OCL_CHECK(err, err = kernel.setArg(narg++, buffer_bias));
278     OCL_CHECK(err, err = kernel.setArg(narg++, buffer_weights));
279     OCL_CHECK(err, err = kernel.setArg(narg++, buffer_out));
280
281     //These commands will load the source_in1 and source_in2 vectors
           from the host
282     //application into the buffer_in1 and buffer_in2 cl::Buffer objects
           . The data
283     //will be be transferred from system memory over PCIe to the FPGA
           on-board
284     //DDR memory.
285     OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_in,
           buffer_bias, buffer_weights}, 0 ));
286
287     cl::Event event;
288     uint64_t kernel_duration = 0;
289
290     //Launch the kernel
291     OCL_CHECK(err, err = q.enqueueTask(kernel, NULL, &event));
292
293     //The result of the previous kernel execution will need to be
           retrieved in
294     //order to view the results. This call will write the data from the
295     //buffer_output cl_mem object to the source_fpga_results vector
296     OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_out},
           CL_MIGRATE_MEM_OBJECT_HOST));
297     OCL_CHECK(err, err = q.finish());
```

```
298
299     kernel_duration = get_duration_ns(event);
300
301     return kernel_duration;
302
303 }
304
305 void fill_data(data_type *data_in, data_type *bias, data_type *weights)
        {
306   int i=0;
307     srand(time(0));
308
309     /* Fill the input data */
310   for (int h=0; h < HEIGHT_IN; h++)
311     for (int w=0; w < WIDTH_IN; w++)
312       for (int c=0; c < CPI; c++) {
313         data_in[i] = i;
314         i++;
315       }
316
317     /* The fill of the bias and the kernel
318     *  is only valid for the first layer */
319
320     /* Fill the bias */
321     for (i=0; i < bias_size; i++) {
322         bias[i] = i + rand() % 9;
323         printf("[HOST | fill_data()] %2.2f\n", bias[i]);
324     }
325
326     /* Fill the weights */
327     for (i=0; i < weights_size; i++) {
328         weights[i] = i + rand() % 9;
329     }
330
331 }
332
333 void show_net_params() {
334     printf("\nNET parameters\n");
335
336     printf("\t CPI -> %2d\n", CPI);
337     printf("\t CPO -> %2d\n\n", CPO);
338     printf("\t WI  -> %2d\n", WIDTH_IN);
339     printf("\t HI  -> %2d\n\n", HEIGHT_IN);
340     printf("\t WO  -> %2d\n", WIDTH_OUT);
341     printf("\t HO  -> %2d\n\n", HEIGHT_OUT);
342
343     printf("\n");
344 }
345
346 /*
347 * Define el tama o de entrada y salida de la net.
348 * Nota: Nose porque solo funciona as  .
349 */
350 void allocate() {
351
352     data_in_size = CPI * WIDTH_IN * HEIGHT_IN;
353     data_out_size = CPO * HEIGHT_OUT * WIDTH_OUT;
354     bias_size = CPO;
355     weights_size = CONV_KW * CONV_KH * CPO * CPI;
```

```
356 }
357
358 int main(int argc, char **argv) {
359     if (argc != 8) {
360         std::cout << "Usage: " << argv[0] << " <XCLBIN File> <WI> <HI>
              <PW> <PH> <max_pool> <avg_pool>" << std::endl;
361         return EXIT_FAILURE;
362     }
363
364     std::string binaryFile = argv[1];
365
366     WIDTH_IN = atoi(argv[2]);
367     HEIGHT_IN = atoi(argv[3]);
368     PW = atoi(argv[4]);
369     PH = atoi(argv[5]);
370     enable_maxpool = atoi(argv[6]);
371     enable_avgpool = atoi(argv[7]);
372
373     WIDTH_CONV_OUT = (WIDTH_IN - CONV_KW + 2 * PW) / CONV_SW + 1;
374     HEIGHT_CONV_OUT = (HEIGHT_IN - CONV_KW + 2 * PH) / CONV_SH + 1;
375
376     WIDTH_OUT = (WIDTH_CONV_OUT - POOL_KW + 2 * PW) / POOL_SW + 1;
377     HEIGHT_OUT = (HEIGHT_CONV_OUT - POOL_KH + 2 * PH) / POOL_SH + 1;
378
379     show_net_params();
380     allocate();
381
382     uint64_t kernel_duration;
383
384     printf("[HOST] TEST NETWORK\n");
385
386     std::vector<data_type, aligned_allocator<data_type>> data_in(
          data_in_size);
387     std::vector<data_type, aligned_allocator<data_type>> out_host(
          data_out_size);
388     std::vector<data_type, aligned_allocator<data_type>> out_fpga(
          data_out_size);
389     std::vector<data_type, aligned_allocator<data_type>> bias(bias_size
          );
390     std::vector<data_type, aligned_allocator<data_type>> weights(
          weights_size);
391
392   fill_data(data_in.data(), bias.data(), weights.data());
393
394     printf("[HOST] Computing the net...\0");
395     net_host(data_in.data(), out_host.data(), bias.data(), weights.data
          (), 1, 0, 1, 0); /* Compute the net in the host */
396     printf(" OK\n");
397
398     kernel_duration = net_fpga(data_in, out_fpga, bias, weights,
          binaryFile); /* Compute the net in the fpga */
399
400     int match = 1;
401     for (int i=0; i < data_out_size; i++) {
402         if (out_host[i] != out_fpga[i]) {
403             printf("Mismatch Result | i = %d, CPU=%2.2f, FPGA=%2.2f\n",
                  i, out_host[i], out_fpga[i]);
404             match = 0;
405             break;
```

```
406             }
407         }
408
409         if (match) printf("[HOST] All results OK!\n");
410
411         std::cout << "[HOST] Wall Clock Time: " << kernel_duration << std::
                endl;
412
413         return 0;
414     }
```