Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Generación de un Módulo Optimizado de Inferencia en FPGAs con HLS

TRABAJO FIN DE MÁSTER

Máster Universitario en Computación en la Nube y de Altas Prestaciones

Autor: Medina Chaveli, Laura

Tutor: Flich Cardo, José
Robles Martínez, Antonio
Hernández Luz, Carles

Curso 2020-2021

# Resum

Les FPGAs (field-programmable gate array) poden ser utilitzades per a la inferència de models de Xarxes Neuronals en sistemes encastats, donat que aquest tipus de dispositiu presenten una alta eficiència energètica i un alt rendiment. Així mateix, la possibilitat de dissenyar maquinari mitjançant High-level synthesis (HLS) ha disminuït la quantitat d'esforç necessari per al desenvolupament de codi per a PGAs.

D'altra banda, hi ha escenaris en els quals no es posible realitzar la inferència completa sobre FPGAs, necessitant un dispositiu CPU per a l'execució de les parts no soportades.

En aquest treball, s'ha utilitzat l'accelerador HLSinf. HLSinf és una implementació en HLS de codi obert d'acceleradors personalitzats per a processos d'inferència de Xarxes Neuronals sobre dispositius FPGAs. A més, en aquest projecte, s'han desenvolupat nous mòduls dins d'aquest accelerador. Així mateix, s'ha integrat l'accelerador amb la llibreria EDDL (European Distributed Deep Learning library), la qual permet l'execució de models sobre diversos dispositius.

Paraules clau: FPGA, High-Level Synthesis, Xarxes Neuronals

Resumen

Las FPGAs (*field-programmable gate array*) pueden ser utilizadas para la inferencia de modelos de Redes Neuronales en sistemas embebidos, dado que este tipo de dispositivo presenta una alta eficiencia energética y un alto rendimiento. Asimismo, la posibilidad de diseñar *hardware* mediante High-level synthesis (HLS) ha disminuido la cantidad de esfuerzo necesario para el desarrollo de código para FPGAs.

Por otra parte, existen escenarios en los cuales no se pueda realizar la inferencia completa sobre FPGAs, necesitando un dispositivo CPU para la ejecución de las partes no soportadas.

En este trabajo, se ha utilizado el acelerador HLSinf. HLSinf es una implementación de HLS de código abierto de aceleradores personalizados para procesos de inferencia de Redes Neuronales sobre dispositivos FPGA. Además, en este proyecto, se han desarrollado nuevos módulos dentro de este acelerador. Asimismo, se ha integrado el acelerador con la libreria EDDL (*European Distributed Deep Learning library*), la cual permite la ejecución de modelos sobre varios dispositivos.

Palabras clave: FPGA, High-Level Synthesis, Redes Neuronales

Abstract

FPGAs (field-programmable gate array) can be used for the inference of Neural Networks models in embedded systems since this type of device presents high energy efficiency and high performance. Moreover, the ability to design hardware using High-level synthesis (HLS) has decreased the amount of effort required to develop code for FPGAs.

On the other hand, there are scenarios in which the complete inference on FPGAs cannot be performed, requiring a CPU device to execute the unsupported parts.

In this work, the accelerator HLSinf has been used. HLSinf is an open-source HLS implementation of custom accelerators for DeepLearning inference processes. on FPGA devices. Furthermore, in this project, new modules have been developed within this

accelerator. Moreover, the accelerator has been integrated with the EDDL (European Distributed Deep Learning library) library, which allows the execution of models on various devices.

Key words: FPGA, High-Level Synthesis, Neuronal Networks

Índice general

Índice general v
Índice de figuras vii
Índice de tablas viii

1 Introducción 1
1.1 Motivación . . . 1
1.2 Objetivos . . . 3
1.3 Estructura de la memoria . . . 3
2 Contexto 5
2.1 Convoluciones . . . 5
2.2 Pooling . . . 7
2.3 Funciones de Activación . . . 10
2.3.1 ReLU . . . 10
2.3.2 Softplus . . . 11
2.3.3 Tangente hiperbólica . . . 11
2.4 High-Level Synthesis . . . 11
2.4.1 Pragmas . . . 14
2.5 OpenCL . . . 17
2.6 HLSinf . . . 18
2.7 EDDL . . . 20
3 Estado del Arte 23
3.1 Aceleradores en FPGA . . . 23
3.1.1 FINN *framework* . . . 23
3.1.2 Deep Learning Processor Unit . . . 24
3.1.3 Intel FPGA Deep Learning Acceleration Suite . . . 25
3.1.4 Otros . . . 26
3.2 Redes Neuronales . . . 26
3.3 Reconocimiento de imágenes . . . 27
4 Optimizaciones en HLSinf 29
4.1 Modelo YOLOv4 . . . 29
4.2 Modelo VGG16 . . . 30
4.3 Capa STM . . . 30
4.3.1 Optimizaciones HLS . . . 31
4.4 Capa Add . . . 33
4.5 Capa LeakyReLU . . . 34
4.6 Soporte a otros tipos de convolución . . . 34
4.6.1 Modificación *Padding* para soporte multivalor . . . 35
4.6.2 Modificación CVT para soporte de stride multivalor . . . 35
5 Adaptación EDDL 37
5.1 Introducción . . . 37
5.2 Adaptación de formatos de datos . . . 37
5.3 Capas fusionadas . . . 38

5.4 Clase HLSinf . . . 39
5.4.1 Soporte multikernel . . . 40
5.5 Generación del modelo . . . 41
5.5.1 Búsqueda de capas fusionadas . . . 42
5.5.2 Gestión de capas padre . . . 42
5.5.3 Adaptación de los parámetros . . . 45
5.5.4 Soporte para convoluciones con tamaño de filtro de uno por uno . 45
6 Evaluación 47
6.1 Recursos y prestaciones de los nuevos módulos en el acelerador HLSinf . 47
6.1.1 Módulos STM y *Add* . . . 47
6.1.2 Estimación de recurso. . . . 48
6.2 Prestaciones de la solución final . . . 49
6.2.1 VGG16 . . . 49
6.3 YOLOv4 . . . 51
7 Conclusiones 55
8 Trabajo futuro 57
8.1 Mejora de la gestión de memoria entre la FPGA y la CPU . . . 57
8.2 Mejora del acelerador HLSinf mediante técnicas de cuantificación . . . 57
8.3 Mejora del acelerador HLSinf mediante técnicas de dispersión . . . 58
8.4 Evaluación exhaustiva de consumo y prestaciones . . . 58

Apéndice

A Modelo VGG16 utilizado 63

Índice de figuras

1.1 Arquitectura básica de una FPGA[3]. 2
1.2 Arquitectura de SELENE Soc. 2
1.3 Arquitectura básica de una FPGA[3]. 3

2.1 Aplicación de una convolución sobre unos datos de entrada. 6
2.2 Aplicación de una convolución sobre unos datos de entrada con *padding*. . 7
2.3 Funcionamiento de una max-pooling. 8
2.4 Funcionamiento de avg-pooling. 9
2.5 Comparación de los métodos max-pooling y avg-pooling [12]. 9
2.6 Función de activación ReLU. 10
2.7 Función de activación Softplus. 11
2.8 Función de activación Tanh. 11
2.9 Fases de High-Level synthesis [25]. 13
2.10 Ejemplo de un proceso de *scheduling* y *binding* [26]. 14
2.11 Efecto de la directiva *dataflow* [29]. 15
2.12 Efecto del *pipeline* [29]. 16
2.13 Efecto de la directiva *unroll* [29]. 17
2.14 Arquitectúra básica de módulos de HLSinf. 18
2.15 Definición del acelerador HLSinf. 20

3.1 Flujo del compilador FINN [42]. 23
3.2 Arquitectura de una DPU[43]. 24
3.3 OpenVINO Toolkit [45]. 25
3.4 Flujo de una aplicación de Redes Neuronales. 26
3.5 Funcionamiento de YOLO[55]. 28

4.1 Nuevas capas de YOLOv4. 29
4.2 Esquema del módulo STM. 30
4.3 Arquitectura de HLSinf con el módulo STM. 31
4.4 Definición del acelerador HLSinf. 32
4.5 Arquitectura del acelerador HLSinf con el nuevo módulo Add 33

5.1 Efecto de la capa Transform con CPI igual a cuatro. 38
5.2 Generación de modelos de capas fusionadas en la FPGA. 41
5.3 Generación de modelos de capas fusionadas en la FPGA. 43
5.4 Funcionamiento de la estructura associated_layers. 44
5.5 Soporte para convoluciones de filtro con tamaño de uno por uno. 46

6.1 Estimación de recursos necesarios globales para la implementación de HLSinf con tamaño CPI/CPO 4/4. 48
6.2 Estimación de recursos. 49
6.3 Modelo VGG16 con capas fusionadas. 50
6.4 Ejecución del modelo VGG16 en varios dispositivos. 51
6.5 Ejecución del modelo YOLOv4 en varios dispositivos. 52

6.6 Tiempo de ejecución de varias capas en YOLOv4. 53
A.1 Versión modificada del modelo VGG16. 63

Índice de tablas

2.1 Puertos asociados a los argumentos de entrada del acelerador HLSinf. . . 20
4.1 Nuevo puerto AXI asociado al argumento de entrada del módulo Add. . . 34
4.2 Parámetros de las convoluciones del modelo YOLOv4. 35
5.1 Atributos de interés de la clase HLSinf. 39

CAPÍTULO 1

Introducción

1.1 Motivación

El uso de las Redes Neuronales en las aplicaciones está teniendo cada vez más impacto en diversas áreas, sobretodo se ha observado una mayor notoriedad en los campos de reconocimiento de imagen, sonido y vídeo. Incluso, en los últimos años, se han realizado varios estudios sobre la gran utilidad de este tipo de aplicaciones en el campo médico, donde se están desarrollando diversos modelos de Redes Neuronales orientados a mejorar el diagnóstico de múltiples enfermedades como el cáncer[1], Parkinson[2], entre otras.

No obstante, la gran carga computacional de este tipo de aplicaciones dificulta su ejecución sobre dispositivos CPU (*central processing unit*), dado que estos no cuentan con la suficiente capacidad computacional para satisfacer los requisitos de estas aplicaciones. Sin embargo, los dispositivos GPU se han convertido hoy en día en una buena herramienta para afrontar grandes cargas computacionales, dada su arquitectura y la existencia de plataformas que facilitan la programación de este tipo de dispositivos.

Aunque las GPUs son una buena opción para el proceso de entrenamiento de las Redes Neuronales, estas presentan un elevado consumo energético, lo que provoca que este tipo de dispositivos no sean buenos para procesos de inferencia de Redes Neuronales en las cuales prevalezca la eficiencia energética.

Un dispositivo FPGA es un tipo de circuito integrado que puede ser programado para diferentes algoritmos. Las FPGAs modernas están compuestas por millones de puertas lógicas, las cuales pueden ser configuradas para la ejecución de diferentes funciones. Estos dispositivos se han convertido en un foco de investigación para la ejecución de aplicaciones basadas en inferencia, dado que presentan una baja latencia así como una elevada eficiencia energética. Dado que para sistemas embebidos es necesario implementar *hardware* cuyo consumo energético sea bajo, las FPGAs son una buena opción para este tipo de sistemas.

En la figura 1.1 se puede observar el diseño básico de una FPGA, las cuales están compuestas por múltiples bloques lógicos configurables (del inglés *Configurable Logic Blocks* o CLB).

Previamente a HLS (High-Level Synthesis) se utilizaban únicamente lenguajes de descripción *hardware*, los cuales requerían altos periodos de desarrollo y un elevado grado de conocimiento de circuitos electrónicos. No obstante, con la llegada de nuevas herramientas de síntesis de alto rendimiento, se ha conseguido elevar el nivel de abstracción mediante el uso de lenguajes de alto nivel como C o C++, los cuales permiten desarrollar aplicaciones para FPGAs en periodos más cortos.

Figura 1.1: Arquitectura básica de una FPGA[3].

Los sistemas críticos de seguridad (*safety-critical systems*) son sistemas que, en caso de fallo, podría conllevar la pérdida de vidas o daños importantes. Un ejemplo de este tipo de sistemas podría ser los dispositivos de control de vuelo o sistemas nucleares.

Las plataformas existentes para sistemas críticos de seguridad (*safety-critical systems*) presentan una falta de flexibilidad debido al uso de componentes de licencia cerrada. SELENE [4] tiene como objetivo cubrir esta brecha proponiendo una nueva familia de plataformas críticas para la seguridad, basadas en componentes de código abierto, consiguiendo así construir una plataforma que facilite la certificación de aplicaciones críticas de seguridad complejas [5].

En la figura 1.2 se puede observar la arquitectura de SELENE SoC (el *system on chip* utilizado en el proyecto de SELENE). De esta figura se remarca el acelerador HLSinf, el cual se encuentra conectado a la agrupación de núcleos y a la memoria.

Figura 1.2: Arquitectura de SELENE Soc.

En SELENE, los aceleradores se utilizan con la librería EDDL (*European Distributed Deep Learning*)[6] una librería en desarrollo de código abierto orientada para la ejecución distribuida de aplicaciones de Redes Neuronales. La librería EDDL forma parte del pro-

yecto DeepHealth[7], un proyecto europeo que tiene como objetivo ofrecer un *framework* unificado y adaptado para explotar arquitecturas heterogéneas de computación de alto rendimiento.

Finalmente, en la figura 1.3, se puede observar la configuración final de la EDDL y el acelerador HLSinf dentro de SELENE SoC. Además, en esta figura se representa de manera gráfica con un cuadro verde la localización exacta donde se realizará una mejora de tanto la librería EDDL como del acelerador HLSinf.

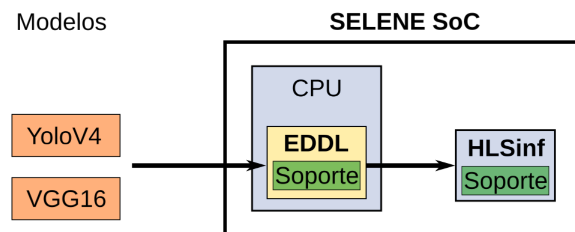

Figura 1.3: Arquitectura básica de una FPGA[3].

1.2 Objetivos

El objetivo principal de este trabajo es el desarrollo y adaptación de soporte para inferencia de modelos de Redes Neuronales en el contexto de los proyectos mencionados anteriormente (SELENE y DeepHealth). Para ello, utilizaremos el acelerador HLSinf e identificaremos carencias y módulos necesarios a ser incorporados para el soporte de procesos específicos. También tendremos en cuenta la librería EDDL y como esta deberá ser modificada para una integración adecuada con HLSinf. Este objetivo lo podemos descompone en la siguientes objetivos parciales:

- Desarrollo de soporte para inferencia en sistemas embebidos
- Desarrollo de módulos para la ejecución de capas de Redes Neuronales en HLS para FPGAs.
- Integración del acelerador HLSinf en la librería EDDL.
- Identificación de características del modelo YOLOv4.
- Identificación de carencias en HLSinf para el modelo YOLOv4.
- Desarrollo de soporte para la ejecución del modelo YOLOv4.
- Evaluación de la solución y búsqueda de cuellos de botella.

1.3 Estructura de la memoria

El presente trabajo se encuentra dividido en los siguientes capítulos:

- En el capítulo 1 se encuentra la introducción del proyecto, donde se describe la motivación y los objetivos del mismo.
- En el capítulo 2 se describen los conceptos básicos sobre los que se centra este proyecto. Concretamente, se explican las bases de algunas de las capas utilizadas en Redes Neuronales, así como el funcionamiento de HLS y OpenCL. Finalmente, se describen los conceptos importante del acelerador HLSinf y de la librería EDDL.

- En el capítulo 3 se exponen los principales aceleradores de FPGA así como las etapas involucradas en una aplicación de Redes Neuronales.
- En el capítulo 4 se detallan los módulos desarrollados en HLS y su integración con el acelerador HLSinf.
- En el capítulo 5 se explica el proceso de integración del acelerador HLSinf en la librería EDDL.
- En el capítulo 6 se muestra la evaluación realizada sobre la solución implementada.
- En el capítulo 7 se exponen las conclusiones extraídas.
- En el capítulo 8 se exponen un conjunto de nuevos pasos a seguir para la mejora del estado actual del proyecto.

CAPÍTULO 2
Contexto

En este capítulo se describen los conceptos básicos en los que se centra todo el proyecto. Concretamente, se describe el funcionamiento y la importancia de algunas funciones presentes en las Redes Neuronales como las convoluciones, las funciones de *pooling* y algunas capas de activación. Sin embargo, los principios básicos de funcionamiento de las Redes Neuronales se describirán el la sección 3.2. En este capitulo también se realizará una breve descripción sobre el funcionamiento de *High-Level Synthesis* (HLS) y OpenCL y, finalmente, se realizará una descripción sobre las aplicaciones centrales de este proyecto: HLSinf (*High-Level Synthesis inference*) y EDDL (*European Distributed Deep Learning*).

2.1 Convoluciones

La convolución es un tipo especializado de operación matemática utilizada para extraer características de una conjunto de datos de entrada utilizando para ello los llamados filtros (o *kernels*). Esta operación es una de las más importantes para las Redes Neuronales ya que permite reconocer patrones entre diferentes imágenes de entrada. Además, también es común su uso en otros contextos científicos como en el procesamiento de señal[8] o el procesado de lenguaje natural[9], entre otros. En esta sección y en todo el proyecto nos centramos en la convolución aplicada al tratamiento de imágenes.

Existen tres conjuntos de datos que componen una convolución, los cuales son:

- **Imágenes de entrada**: datos de entrada sobre los cuales se desea realizar la convolución. Su dimensión viene dada por HI (altura de las imágenes de entrada), WI (amplitud de las imágenes de entrada) y CI (número de canales de las imágenes de entrada).
- **Filtro**: datos de entrada que componen el elemento clave de la convolución. Según los valores del filtro se pueden extraer diferentes características de las imágenes de entrada. Las dimensiones del filtro se componen por KH (altura de filtro) y KW (amplitud del filtro). Típicamente estos tamaños suelen ser de 3 x 3, 5 x 5 ó 7 x 7.
- **Imágenes de salida**: resultado de la convolución donde la dimensionalidad se compone por HO (altura de las imágenes de salida), WO (amplitud de las imágenes de salida) y CO (número de canales de las imágenes de salida). Estas dimensiones varían en función a diferentes parámetros de la convolución.

Las convoluciones se basan en realizar productos vectoriales entre el filtro y los datos de entrada hasta obtener todos los valores de salida. Para conseguir esto, se realizan diversas iteraciones donde en cada una de ellas se multiplica el valor de cada elemento

del filtro por un subconjunto de elementos de la imagen de entrada (comúnmente conocido como ventana) para, posteriormente, realizar la suma de los productos calculados, obteniendo así un único elemento de salida por cada iteración. Para elegir el siguiente subconjunto de datos de la imagen de entrada se realiza un desplazamiento de la ventana anterior de manera que los elementos de la nueva ventana coincidan con los elementos del filtro. Este desplazamiento es conocido como *stride*[10].

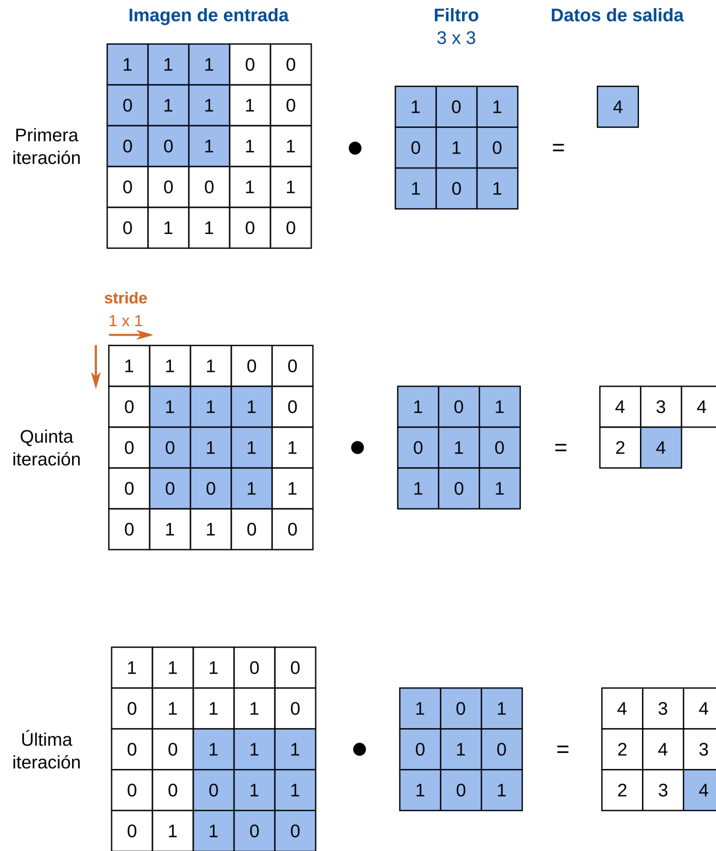

Figura 2.1: Aplicación de una convolución sobre unos datos de entrada.

En la figura 2.1 se plasma el funcionamiento de una convolución aplicada a unos datos de entrada con un único canal de dimensión 5 x 5 utilizando también un filtro de tamaño 3 x 3. Como se observa, para realizar la convolución se agrupan los datos de entrada en una ventana para definir un subconjunto de elementos, la cual se encuentra representada en la figura como el conjunto de elementos de entrada con color azul. El tamaño de la ventana viene definida por el tamaño del filtro. Posteriormente, se realiza el producto vectorial entre la ventana y el filtro y, seguidamente, se desplaza la ventana una cantidad de elementos igual al *stride* definido para llevar a cabo la misma operación con un subconjunto diferente de datos de entrada. Finalmente, se obtienen unos datos de salida de dimensión 3 x 3.

Como se ha podido observar en la figura 2.1, en una convolución es común que las dimensiones de los datos de salida sean menores a las dimensiones de entrada, dado el tamaño de filtro y el *stride* definido. Esto, en el contexto de las Redes Neuronales, puede ser un problema dado que es común el uso de una gran cantidad de convoluciones por lo que esto puede conllevar una reducción significativa de los datos de entrada. El *padding* es la técnica utilizada para resolver este problema. Esta técnica consiste en añadir a los datos de entrada valores nulos o ceros en los márgenes para conseguir así una mayor dimensionalidad de los datos de salida.

En la figura 2.2 se representa el funcionamiento del *padding*. Como se puede observar, al introducir elementos nulos en los márgenes de la imagen (representados en color verde) se obtiene una dimensionalidad de los datos de entrada iguales a la de los datos de entrada.

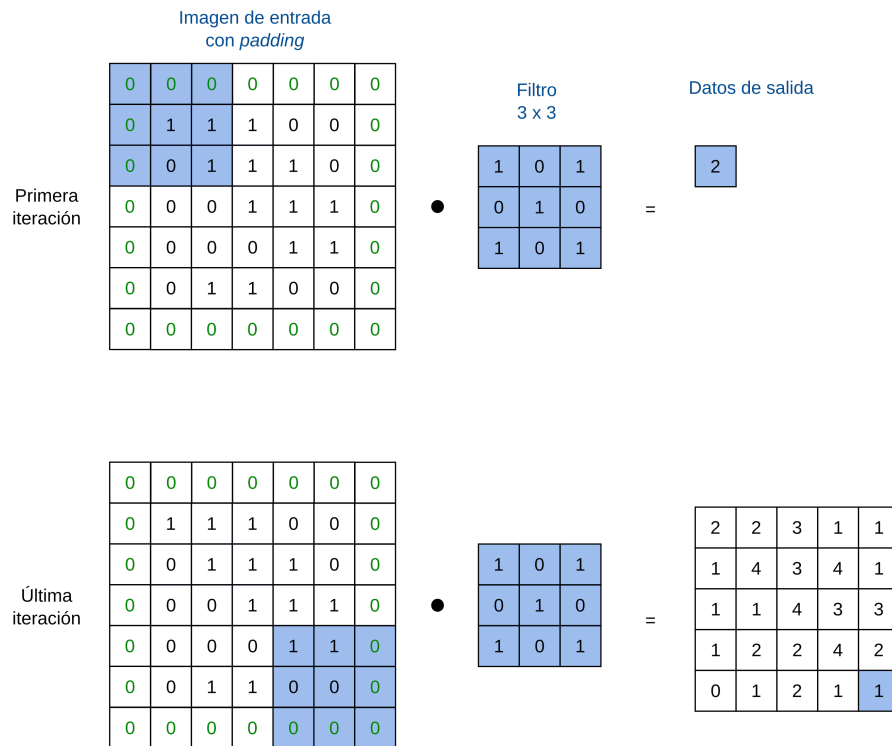

Figura 2.2: Aplicación de una convolución sobre unos datos de entrada con *padding*.

Por lo tanto, podemos calcular las dimensiones de los datos de salida a partir de los diferentes parámetros explicados anteriormente:

$$dim(O) = (\frac{HI+2PH-KH}{SH} + 1, \frac{WI+2PW-KW}{SW} + 1)$$

En la anterior ecuación O corresponde a las imágenes de salida, PH y PW corresponden a al *padding* y SH y SW al *stride* [11].

Otro parámetro importante para las Redes Neuronales es el bias, un parámetro definido durante la etapa de entrenamiento que permite proporcionar flexibilidad al modelo y determina qué neuronas permanecen activas. Para ello, al realizar la función de activación se suma el valor del bias al valor pasado a la función (el cual será el cálculo de los valores de la neuronas por su correspondiente peso).

En las Redes Neuronales es común utilizar múltiples convoluciones para obtener diversas características de las imágenes de entrada utilizando para ello un conjunto de filtros con diferentes propiedades. En este proyecto, se utilizan FPGAs para realizar estas operaciones utilizando la implementación propuesta en HLSinf.

2.2 Pooling

Existen diferentes métodos propuestos para implementar la operación de *Pooling* como *max-pooling*, *average-pooling*, *mixed-pooling*, entre otros. Sin embargo, los métodos *max-*

pooling y *average-pooling* destacan entre la industria de las Redes Neuronales debido a su simplicidad[12].

Los diferentes conjuntos de datos que intervienen en una operación *pooling* son las siguientes:

- **Imágenes de entrada**: datos de entrada sobre los cuales se desea realizar la operación *pooling* para reducir su dimensionalidad. Su dimensión viene dada por HI (altura de las imágenes de entrada), WI (amplitud de las imágenes de entrada) y CI (número de canales de las imágenes de entrada).

- **Imágenes de salida**: resultado de la operación donde la dimensionalidad viene definida por HO (altura de las imágenes de salida), WO (amplitud de las imágenes de salida) y CO (número de canales de las imágenes de salida). Estas dimensiones, como veremos a continuación, varían en función a diferentes parámetros de la operación.

Las operaciones de *pooling* se basan en calcular el elemento representativo de cada subconjunto obteniendo un elemento de salida por iteración, donde cada subconjunto viene definido por los parámetros *stride* y tamaño de filtro. El tamaño de filtro indica la cantidad de elementos de la matriz de entrada que formará el subconjunto de elementos. Por otra parte, al igual que en las convoluciones, el *stride* marca la cantidad de elementos sobre los cuales se desplaza la ventana de elementos. Típicamente, en las Redes Neuronales se suelen utilizar tamaños de filtros de 2 x 2 y *strides* de 2 X 2.

Además, los parámetros descritos anteriormente definen las dimensiones de la imagen de salida de la siguiente manera:

$$dim(O) = (\frac{HI-KH}{SH} + 1, \frac{WI-KW}{SW} + 1, CI)$$

En la anterior ecuación O corresponde a las imágenes de salida y SH y SW al *stride* [12].

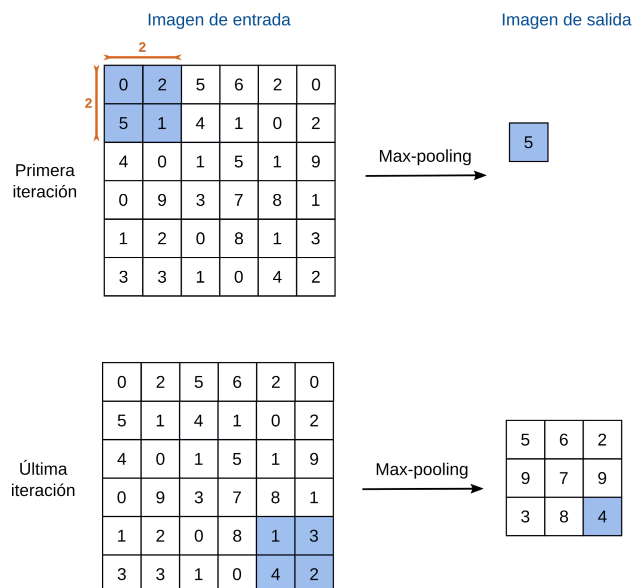

Figura 2.3: Funcionamiento de una max-pooling.

En las figuras 2.3 y 2.4 se puede observar el funcionamiento de las operaciones de max-pooling y avg-pooling de una imagen de entrada de dimensión 3 x 3 x 1, un *stride*

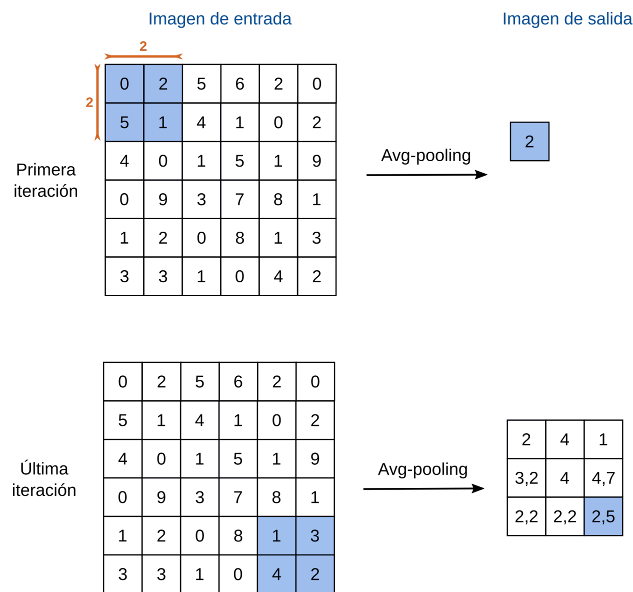

Figura 2.4: Funcionamiento de avg-pooling.

de 2 x 2 y un tamaño de filtro de 2 x 2. Como resultado, se obtiene una imagen de salida de dimensión 3 x 3 x 1.

Como se puede observar en las anteriores figuras, diferentes implementaciones del método *pooling* pueden conllevar a resultados dispares. Por esto, es importante decidir el método *pooling* que pueda extraer las características relevantes para nuestro caso de uso según los datos de entrada y los filtros. Esta problemática queda reflejada en la figura 2.5, donde se muestra el resultado de aplicar las operaciones de max-pooling y avg-pooling a dos imágenes. En estas imágenes, los píxeles blancos representan el valor más alto y los píxeles negros el valor más bajo. En la imagen de la izquierda se muestra como el método max-pooling consigue una imagen totalmente blanca eliminando la figura existente. Esto es debido a que, al elegir el máximo de cada subconjunto de píxeles de la imagen, el mayor de ellos siempre es un píxel de color blanco, por lo que aquellos representativos no se ven reflejados en la imagen de salida. Por otra parte, la aplicación del método avg-pooling consigue mantener la figura aunque consigue un efecto borroso sobre ella.

En cambio, en la imagen de la derecha de la figura se muestra como el método max-pooling consigue un submuestreo de la imagen manteniendo la figura y sin aplicar un efecto borroso, lo cual sigue ocurriendo en el caso del método avg-pooling.

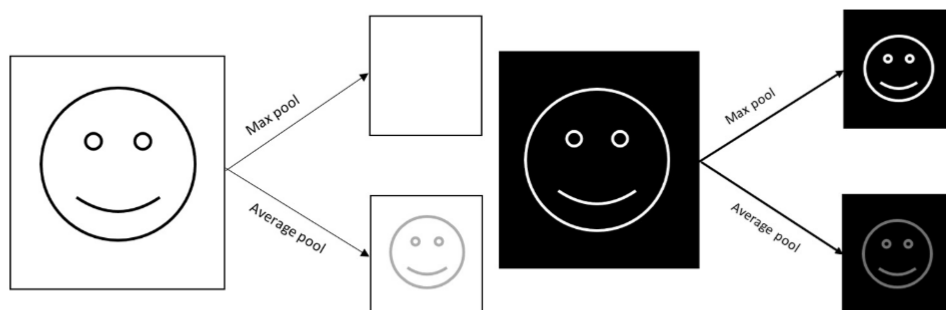

Figura 2.5: Comparación de los métodos max-pooling y avg-pooling [12].

Como se ha comentado anteriormente en las Redes Neuronales es común alternar las capas *pooling* con las capas convolucionales. En este proyecto, también se utilizan FPGAs para realizar estas operaciones utilizando la implementación propuesta en HLSinf.

2.3 Funciones de Activación

Las funciones de activación se utilizan en las Redes Neuronales para transformar una señal de entrada en una de salida aplicando para ello la función correspondiente.

Existen numerosas funciones de Activación utilizadas en el contexto de las Redes Neuronales y es de vital importancia elegir correctamente el tipo de función ya que este tipo de operaciones tienen un gran impacto en la precisión en la red neuronal [13]. Algunas de las funciones utilizadas e implementadas en este proyecto son las funciones ReLU (Rectified Linear Unit)[14], Softplus[15] y Tangente hiperbólica[16].

2.3.1. ReLU

Esta función es la más ampliamente utilizada en las Redes Neuronales dado que contribuye a la mejora del rendimiento de las Redes Neuronales. Esto es debido a que esta función de activación es fácil de calcular y, con ella, se logra obtener una gran velocidad de convergencia[17].

Esta función se encuentra definida como $f(x) = max(0, x)$ donde x corresponde al valor de entrada de la función. Como resultado se obtiene un conjunto de señales donde si el valor es menor al valor de umbral definido se obtendrá el mínimo como valor de salida. En cambio, si la señal de entrada es mayor que el valor de umbral (en esta caso cero) el valor de salida será igual al valor de entrada. En la figura 2.6 se representa el comportamiento de la función ReLU.

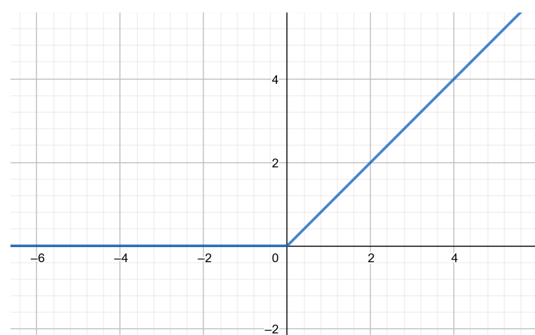

Figura 2.6: Función de activación ReLU.

Como se ha comentado anteriormente, la función ReLU anula los elementos negativos. Sin embargo, estos elementos descartados pueden contener información relevante. Este problema se resuelve con una variante de la función ReLU conocida como LeakyReLU. Esta variante de la función ReLU se encuentra definida como: $f(x) = max(\alpha n, x)$, donde α es un parámetro predefinido y suele estar definido entre los valores 0 y 1. Como resultado, la función LeakyReLU comprime los valores negativos, por lo que la información con valores negativos se mantiene, evitando en gran medida la dispersión de la red[18].

2.3.2. Softplus

Otra de las funciones de activación utilizadas en este proyecto es la función Softplus. Esta función se define como: $f(x) = log(1 + exp(x))$ y tiene un comportamiento similar a la función ReLU dado que todos los elementos de salida que proporciona son positivos. En la figura 2.7 se puede observar el comportamiento de esta función, donde se muestra que, a diferencia de las funciones de activación anteriores, esta no proporciona linealidad al sistema [15].

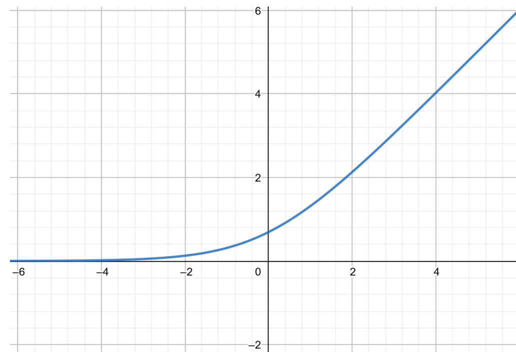

Figura 2.7: Función de activación Softplus.

2.3.3. Tangente hiperbólica

La tangente hiperbólica es otra de las funciones de activación ampliamente utilizada en Redes Neuronales. Esta función se define como:

$$f(x) = \frac{sinh(x)}{cosh(x)}$$

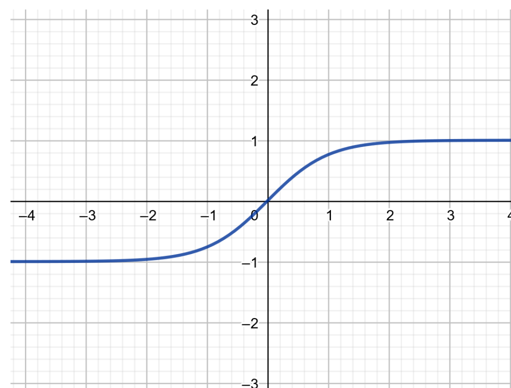

Figura 2.8: Función de activación Tanh.

La curva de esta función se observa en la figura 2.8, donde se puede ver que se trata de una función simétrica centrada en el elemento cero. Además, los elementos de salida de esta función varían entre un rango acotado de 1 y -1.

2.4 High-Level Synthesis

En los 80 emergieron nuevos lenguajes de descripción *hardware* (*Hardware Description Language* o HDL) como Verilog y VHDL utilizados para describir el diseño de los circuitos

electrónicos. Esto provocó que por más de una década las aplicaciones para FPGA se desarrollaran mediante HDL. Sin embargo, el uso de estos lenguajes requieren largos periodos de desarrollo y un alto conocimiento en circuitos electrónicos. Además, con el incremento de la complexidad de las tecnologías y de las aplicaciones modernas aumentó la necesidad de elevar el nivel de abstracción y acelerar la automatización de los procesos de síntesis y verificación de los procesos de diseño del *hardware*.

Es por eso que en los 2000 empezaron a emerger las herramientas de síntesis de alto nivel (High-Level Synthesis o HLS) lo cual ofreció la posibilidad de programar aplicaciones para las FPGAs utilizando lenguajes de alto nivel como C o C++[19]. HLS eleva el nivel de abstracción, permitiendo reducir el tiempo para el desarrollo y verificación de los programas. Además, también permite reducir el tiempo de depuración y permite la creación de código más legible.

Existen algunas librerías desarrolladas para HLS. Entre ellas, encontramos algunas de código abierto como HLS LIBS[20] y otras de vendedores conocidos como HLS IP Libraries[21] de Xilinx[22]. De esta última destacamos Vitis HLS Math Library[23], la cual ofrece diversas funciones matemáticas utilizadas en este proyecto.

HLS transforma las especificaciones de alto nivel aportadas por el desarrollador en una implementación *hardware* eficiente mediante un proceso automático o semiautomático. La arquitectura generada se describe en RTL (Register Transfer Level)[24] y contiene las rutas de datos (conocido como *datapaths*) requeridos por el diseño. En la figura 2.9 se pueden observar las fases que se llevan a cabo para realizar dicha transformación, donde se encuentran las siguientes tareas las cuales se explicarán con más detalle a continuación:

1. Compilación de la especificación.
2. Asignación de recursos *hardware* (conocido como *allocation*).
3. Planificación de las operaciones en ciclos de reloj (conocido como *scheduling*).
4. Vinculación de las operaciones a las unidades funcionales (conocido como *binding*).
5. Generación de la arquitectura RTL.

En primer lugar, se realiza la fase de compilación donde, a partir del programa descrito normalmente en C o C++ se crea la representación formal de dicho programa. Este modelo formal representa las dependencias de control y de datos existentes en el programa original. Además, en esta fase se suelen llevar a cabo optimizaciones del código como eliminación de código muerto.

En la fase de asignación de recursos se define el tipo y la cantidad de recursos hardware necesarios para implementar el diseño de entrada [25].

Una de las fases más importantes es el *scheduling*. En esta fase se determinan qué operaciones se llevan a cabo en cada ciclo de reloj. Para esto se tienen en cuenta los siguientes factores [26]:

- Dependencias entre operaciones.
- Duración de un ciclo de reloj y la frecuencia de reloj.
- Tiempo que tarda en completarse la operación.
- Recursos disponibles.
- Directivas de optimización especificadas por el usuario.

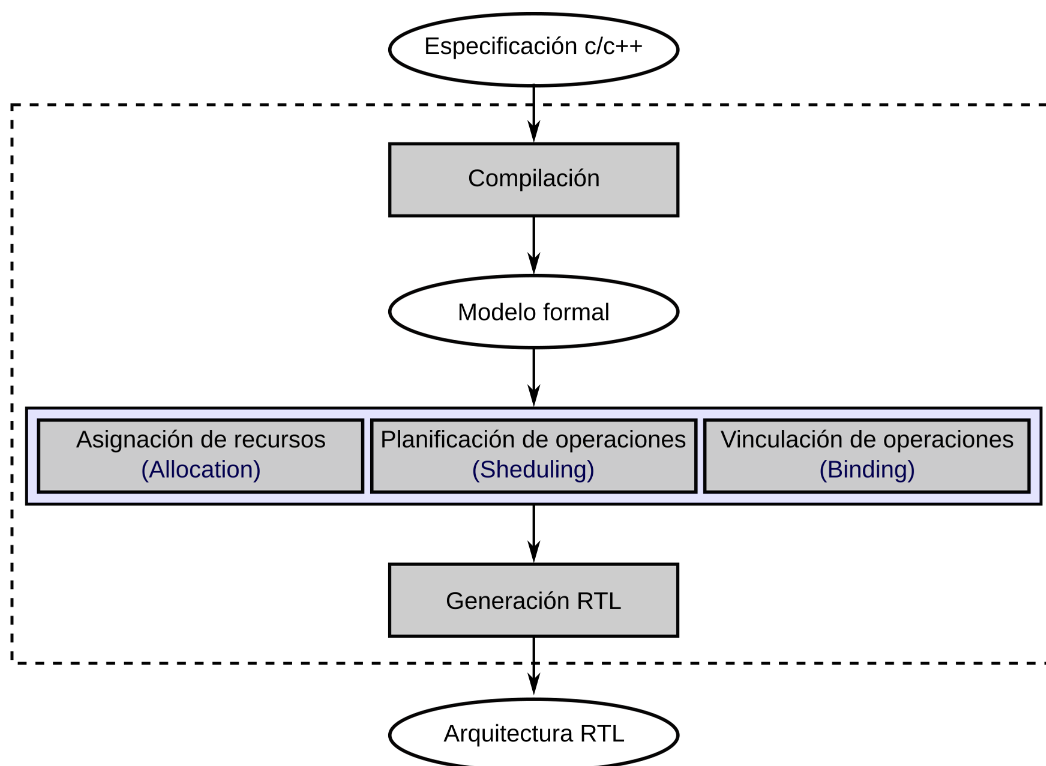

Figura 2.9: Fases de High-Level synthesis [25].

En la fase de *binding* cada operación en el modelo de especificación se implementa en alguna unidad funcional capaz de llevar a cabo dicha operación.

Finalmente, con las anteriores fases completadas se genera el modelo RTL aplicando todas las decisiones de diseño anteriores.

En la figura 2.10 se representa un ejemplo del proceso de *scheduling* y *binding* sobre la operación $y = x * a + b + c$. Primeramente, en la fase de *scheduling* se determinan las operaciones que se realizaran en cada ciclo de reloj, indicando que las dos primeras operaciones se han de realizar en el primer ciclo y la última en el segundo ciclo. Posteriormente, en la fase inicial *binding*, se implementan las operaciones utilizando multiplicadores (Mul) y sumadores (AddSub). En la segunda fase de *binding* (*target binding*) se implementan estos elementos sobre recursos hardware como, en este caso, módulos DSPs (módulos de cómputo presentes en las arquitecturas FPGAs que presentan un alto rendimiento y una alta eficiencia de implementación) [26].

Existen distintas herramientas orientadas a la programación en lenguaje de alto nivel (HLS) como Intel High Level Synthesis Compiler [27] o Vitis HLS[28] de Xilinx. Para este proyecto se ha utilizado Vitis HLS, ya que Xilinx ofrece un gran conjunto de herramientas para la programación en HLS como sistemas de depuración y genera gran cantidad de información relevante para el programador sobre los diseños desarrollados. También permite la integración de módulos descritos en HLS con módulos descritos en HDL de manera simple. Además, esta compañía ofrece un conjunto de documentación clara orientada al aprendizaje HLS. Estas herramientas proporcionan además pragmas orientados a la optimización del diseño y, en la siguiente subsección, se describirán aquellos más relevantes para este proyecto.

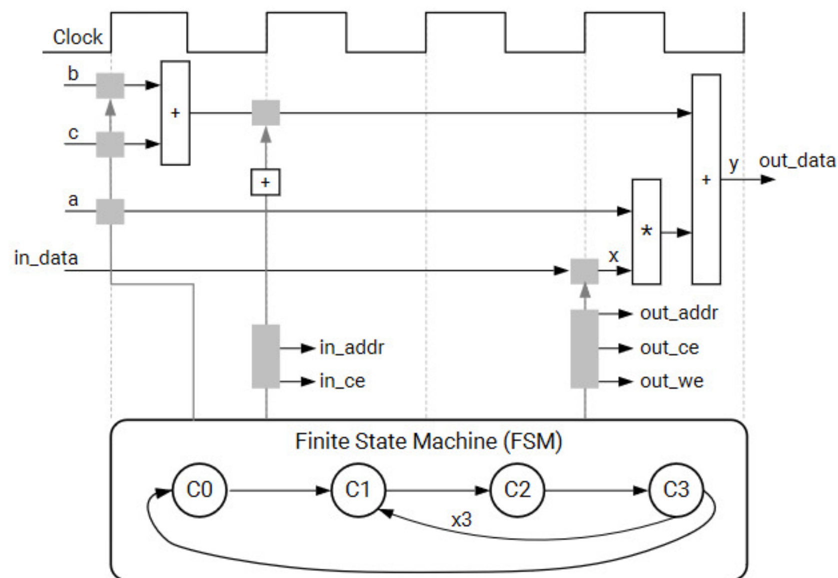

Figura 2.10: Ejemplo de un proceso de *scheduling* y *binding* [26].

2.4.1. Pragmas

Los pragmas de HLS permiten reducir la latencia e incrementar el rendimiento utilizando la menor cantidad de área libre en el dispositivo posible, reduciendo al máximo los recursos del dispositivo en el esquema RTL generado pero implementando la opción más eficiente[29]. Estos pragmas se añaden directamente al código fuente.

Aunque existen una gran cantidad de pragmas, se describirán brevemente los más utilizados en este proyecto: *array_partition*, *dataflow*, *pipeline*, *stream*, *unroll* e *interface*.

Pragma HLS array_partition

Por defecto, los datos se almacenan en las memorias BRAM (*Block Random Access Memory*). Este tipo de memoria tiene típicamente un puerto de escritura y un puerto de lectura, lo que permite realizar una lectura y una escritura por cada ciclo[30]. Esto puede ocasionar una limitación de eficiencia si se desea realizar más de una lectura o escritura en paralelo sobre datos almacenados en la misma BRAM.

Este pragma permite organizar subconjuntos de una matriz de datos en múltiples memorias o registros, incrementando la cantidad de posibles accesos por ciclo. Sin embargo, se utilizará más área del dispositivo.

Pragma HLS dataflow

Este pragma permite ejecutar varias funciones concurrentemente. Para esto, este pragma analiza las dependencias de datos entre funciones secuenciales y crea canales FIFO los cuales permiten ejecutar una función antes de que se generen todos los datos necesarios de la función anterior.

En la figura 2.11 se puede observar el efecto que tiene este pragma sobre un programa. Este programa pretende ejecutar tres funciones (func_A, func_B y func_C) donde la función func_B tiene una dependencia de datos con la func_A y la función func_C tiene una dependencia de datos con la función func_B. Las funciones tiene una latencia de 3,2 y 3 ciclos respectivamente. La opción A (izquierda) representa una ejecución totalmen-

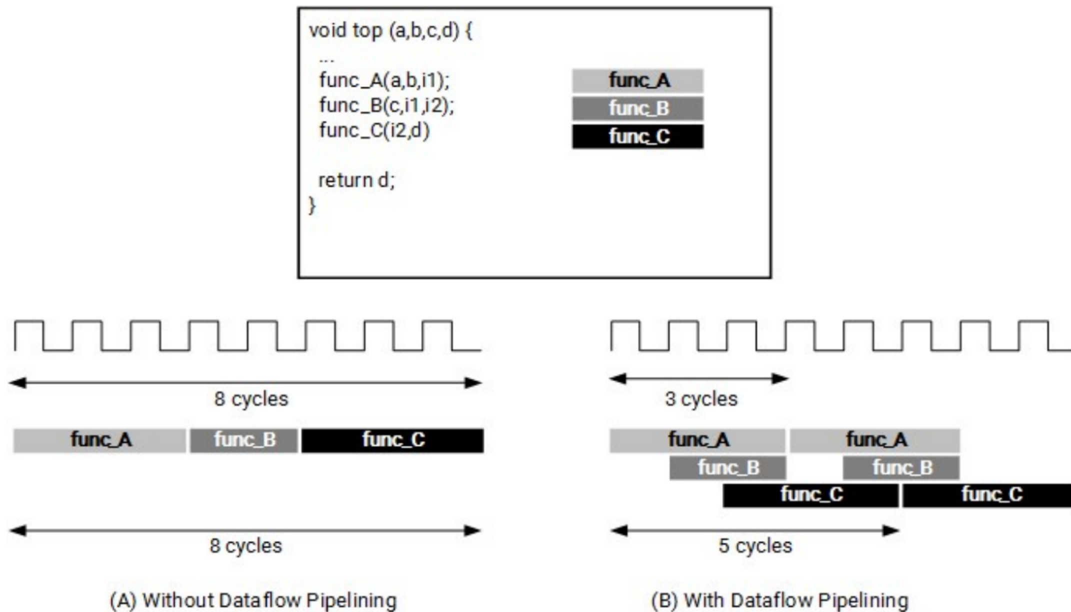

Figura 2.11: Efecto de la directiva *dataflow* [29].

te secuencial de este programa lo cual conlleva 8 ciclos ejecutar completamente las tres funciones.

La opción B (derecha) corresponde a una ejecución en la cual se ha utilizado la directiva *dataflow*. En este caso, la función func_B puede empezar a ejecutarse el segundo ciclo ya que la func_A ha generado en el primer ciclo los datos necesarios para llevar a cabo el primer ciclo de la función func_B, lo cual ocurre también en el tercer ciclo entre las funciones func_B y func_C. Como resultado, conlleva solo 5 ciclos ejecutar completamente las tres funciones.

Pragma HLS pipeline

Este pragma permite reducir el intervalo de iniciación(II) de un bucle consiguiendo procesar nuevas iteraciones cada N ciclos sin que necesariamiente la iteración anterior se haya procesado completamente, donde N corresponde al II.

Aunque es tarea del programador resolver las dependencias entre las iteraciones, si Vitis HLS no puede crear un modelo con un intervalo de iniciación igual al especificado o por defecto mostrará una advertencia y generará el diseño con el menor II posible[31].

En la figura 2.12 se observa el efecto de este pragma sobre un bucle. Este bucle ejecuta un total de tres iteraciones (con una latencia de 3 ciclos cada una) donde las operaciones que se realizan no son dependientes entre ellas. En la opción A (izquierda) se ejecuta este bucle sin *pipelining* lo que provoca que exista una latencia de 3 ciclos entre cada procesamiento de entrada de datos de cada iteración (II = 3). Esto provoca una latencia de 8 ciclos antes de que la última escritura se efectúe. Por otra parte, en el caso B (derecha) se muestra la ejecución del bucle con *pipeline* y un II de uno, ya que se procesan nuevos datos de entrada cada ciclo. De esta manera, se obtiene una latencia de 5 ciclos.

Pragma HLS stream

Por defecto, las matrices de datos se almacenan en memorias BRAM. Este tipo de memorias permiten realizar varios accesos de memoria sobre la misma dirección ya que

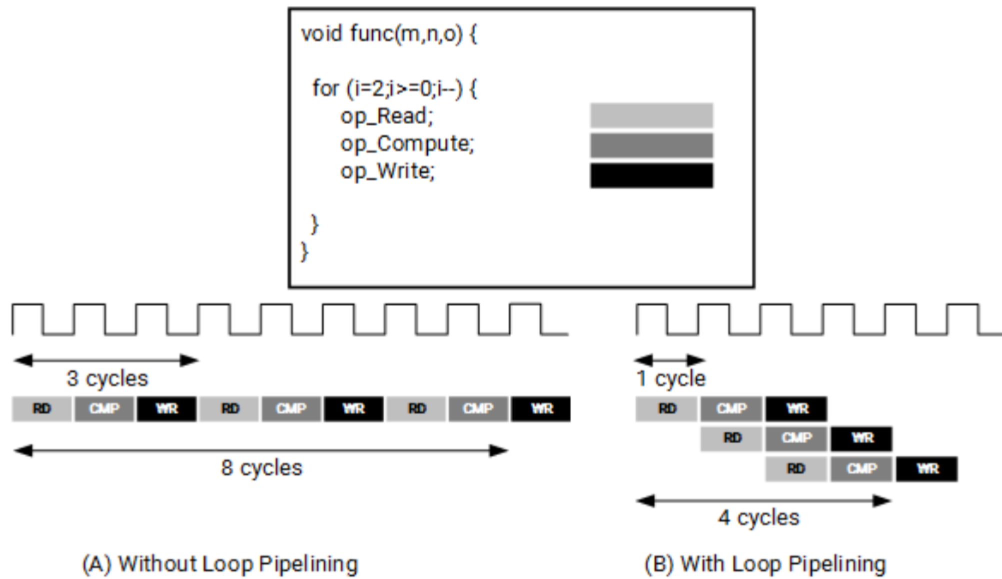

Figura 2.12: Efecto del *pipeline* [29].

permiten acceder a cualquier dirección de memoria. Sin embargo, para casos de uso en los que se requiere procesar los datos en el orden en el que los datos son creados es más adecuado que los datos se almacenen en FIFOs (*First In, First Out*), los cuales actúan como colas. Mediante este pragma, se indica que los datos se almacenen en FIFOs donde, además, se puede indicar el tamaño del FIFO.

Pragma HLS unroll

Esta directiva permite transformar el bucle original para ejecutar una agrupación de iteraciones como un conjunto de operaciones independientes. Realiza múltiples copias del cuerpo del bucle en el diseño RTL permitiendo ejecutar varias iteraciones concurrentemente desde el *Hardware*. En caso de no desear un *unrolling* completo puede indicar el factor de *unrolling*, donde este factor determinará las copias que se realizarán en el modelo RTL y los recursos del dispositivo a utilizar.

En la figura 2.13 se puede observar el efecto de la directiva *unroll* sobre un bucle, el cual realiza la suma de los elementos de un vector y un escalar. El caso A (izquierda) corresponde a la ejecución del bucle sin la directiva. En el se puede observar como únicamente se utiliza un sumador para todas las iteraciones, lo que provoca una latencia de cuatro ciclos y una ejecución secuencial. El caso B (derecha) representa la ejecución del bucle con el uso de esta directiva. Como consecuencia, se utilizan el mismo número de sumatorios que iteraciones para realizar todas las sumas en el mismo ciclo. En este caso, dado que las cuatro sumas se pueden realizar en el mismo ciclo, la ejecución del bucle tarda únicamente un ciclo en ejecutarse.

Como se verá más adelante, estas directivas se han utilizado para desarrollar las nuevas funcionalidades añadidas al acelerador HLSinf para poder así generar el modelo RTL más eficiente posible.

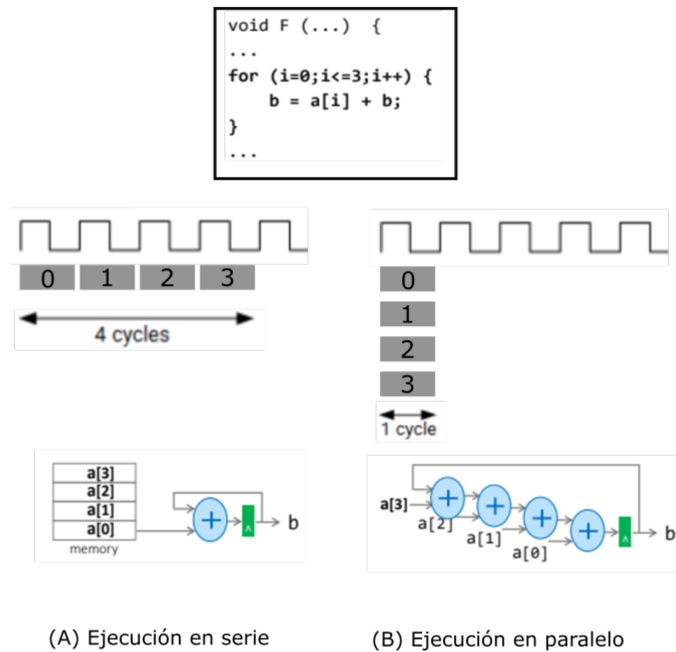

Figura 2.13: Efecto de la directiva *unroll* [29].

Pragma HLS interface

En lenguaje C, todas las operaciones de entrada y salida se realizan a partir de argumentos de entrada a las funciones. Sin embargo, en un diseño RTL, estas mismas operaciones de entrada y salida deben realizarse a través de un puerto en la interfaz diseñada. Dichos puertos operan siguiendo un protocolo de entrada/salida específico. En HLS esto es posible gracias a la directiva *interface*, la cual especifica como se crean los puertos RTL y como se asocian a los argumentos de entrada de la función.

Como se explicará más adelante, esta directiva es utilizada en este proyecto para asignar un argumento de entrada del acelerador HLSinf a su puerto AXI (del inglés *Advanced eXtensible Interface*) correspondiente.

2.5 OpenCL

OpenCL [32] es un estándar de código abierto orientado a la programación para sistemas heterogéneos. Con OpenCL una aplicación se divide en dos partes: *host* y *device*. La parte *host* es aquella encargada de interactuar con la API de OpenCL. Por otra parte, el *device* es el dispositivo destino en el que se quiere ejecutar el *kernel*. Un *kernel* es un fragmento de código donde se define una funcionalidad con el fin de ser ejecutada en la parte *device*. OpenCL permite escribir la parte del host en lenguaje C o C++ y, además, permite ser compilado por un compilador convencional para su ejecución en una CPU host. Asimismo, este estándar proporciona un lenguaje de alto nivel y abstracciones de hardware, lo cual permite a los desarrolladores acelerar las aplicaciones de sistemas heterogéneos simplificando la comunicación entre diferentes arquitecturas *hardware* que pueden componer las partes *host* y *device*[33].

El conjunto de dispositivos OpenCL generalmente no comparten la misma región de memoria y tienen un diferente set de instrucciones. Sin embargo, OpenCL asume toda esta heterogeneidad entre los dispositivos de manera transparente para el desarrollador. Además, proporciona funciones para enumerar los dispositivos destino (*devices*) disponibles, como CPUs, GPUs (*graphics processing unit*) y FPGAs y gestionar los llamados

contextos para administrar la memoria, realizar transferencias de datos entre las partes *host* y *device*, llevar a cabo la ejecución de *kernels*, consultar el progreso de ejecución del *device* y la verificación de errores. Cabe destacar que, aunque OpenCL garantiza la correcta portabilidad de *kernels* entre diferentes arquitecturas, no garantiza que este ofrezca el mismo rendimiento entre ellas.

En este trabajo OpenCL se utiliza para la comunicación y la gestión de memoria entre la FPGA y la CPU en HLSinf y EDDL gracias a las funciones que ofrece su API así como la sincronización entre ambos dispositivos.

2.6 HLSinf

HLSinf (High-Level Synthesis inference) es un acelerador en desarrollo de código abierto[34]. Este acelerador está siendo adaptado para otros proyectos financiados por la Unión Europea como SELENE [4] o Deep Health [7].

Este acelerador está siendo desarrollado en High-Level Synthesis (HLS), lo que permite elevar el nivel de abstracción de su implementación sin requerir la necesidad de tener un alto conocimiento en circuitos electrónicos.

HLSinf tiene como objetivo proporcionar una gran flexibilidad y configurabilidad para procesos de inferencia de modelos de Redes Neuronales basados en convoluciones utilizando para ello FPGAs. Puede ser adaptado a diferentes plataformas FPGAs y permite configurar diversos aspectos importantes como el tipo de dato, el paralelismo interno y el tipo de convolución o módulos a instanciar.

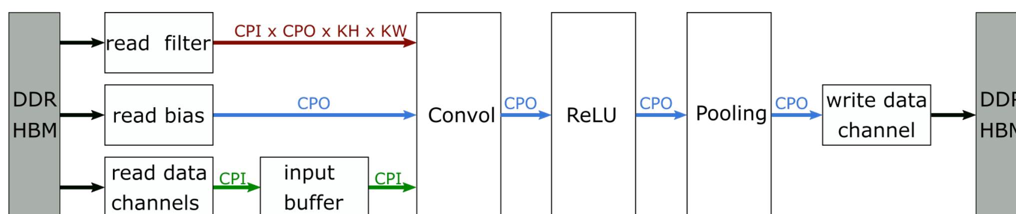

Figura 2.14: Arquitectúra básica de módulos de HLSinf.

El diseño del acelerador sigue el modelo *dataflow* (explicado en la sección 2.4.1) donde diversos módulos se encuentran interconectados mediante *streams* de datos. Esto puede observarse en la figura 2.14, donde encontramos diferentes tipos de módulos: módulos encargados del procesado de datos y módulos de operaciones aritméticas básicas de Redes Neuronales.

En primer lugar, los módulos de procesado de datos están en contacto con la memoria de la FPGA (DDR o HBM) y tienen como objetivo leer o escribir los datos almacenados en estas memorias. En cuanto a los módulos de lectura nos encontramos los módulos "*read bias*"(lee el bias para las operaciones convolucionales), "*read filter*"(lee los filtros para las operaciones convolucionales) y *read data channel* (lee las imágenes entrada). Estos módulos obtienen como parámetro de entrada de las funciones los datos de la memoria de la FPGA y, después de preparar y adaptar los datos según el modelo que se explicará a continuación, transportan los datos a otros módulos internos del acelerador mediante los llamados *streams* de datos. Cuando todos los módulos de operaciones aritméticas han terminado su ejecución los datos obtenidos llegan al último módulo del acelerador nombrado "*write data channel*", el cual escribe los datos resultantes en la memoria DDR o HBM de la FPGA.

En segundo lugar, encontramos los módulos de operaciones aritméticas utilizadas para llevar poder llevar a cabo procesos de inferencia de diferentes modelos de Redes Neuronales. Si se sigue observando la figura 2.14 se puede ver que los módulos de lectura de datos están conectados al módulo de convoluciones. Seguidamente, este se conecta con el módulo ReLU el cual está conectado a la vez con el módulo Pooling. Actualmente, el módulo Pooling puede ser configurado para realizar las operaciones de Maxpooling o Avgpooling. Finalmente, este está conectado con el módulo encargado de la escritura de los datos en la memoria de la FPGA.

Cada uno de los módulos presentes en la arquitectura HLSinf tiene un parámetro de activación, a excepción del módulo responsable de la ejecución de convoluciones (representado como Convol en la figura 2.14), el cual se ejecuta en cada llamada a HLSinf. De esta manera, HLSinf ofrece un segundo grado de flexibilidad ya que, además de indicar los módulos a instanciar en la FPGA, también se pueden indicar a posteriori las operaciones a realizar en cada ejecución.

La funcionalidad del acelerador puede ser extendida añadiendo nuevos módulos a su arquitectura básica. Esto se verá más adelante con los nuevos módulos desarrollados en este proyecto para la adaptación del modelo YOLOv4[35], un sistema de detección de objetos a tiempo real, donde se han desarrollado e integrado soporte a nuevas capas para su ejecución sobre dispositivos FPGA.

Los datos son almacenados y tratados en agrupaciones de un conjunto de canales. De esta manera, varios canales son leídos concurrentemente, donde el número de canales paralelos viene definido por el parámetro CPI (del inglés *channels per input*). De la misma manera, también se puede definir la cantidad de canales que se escriben en la salida mediante el parámetro CPO (del inglés *channels per output*). Estos dos parámetros definen el paralelismo básico del acelerador y permiten adaptarlo a los recursos de cualquier tarjeta FPGA. Además, los filtros también poseen un formato atípico definido por Go x Gi x CPO x CPI x KH x KW, donde Go se define como $Go = O/CPO$ y Gi se define como $Gi = I/CPI$. Además, I corresponde a canales de entrada y O corresponde a canales de salida.

Otra configurabilidad posible del acelerador es el tipo de convolución. Actualmente HLSinf soporta las operaciones de convoluciones directas, Winograd y DeepWise Separable (DWS), lo que permite adaptar la funcionalidad, eficiencia y recursos del acelerador a cada caso de uso particular.

El acelerador permite configurar específicos tipos de datos. De esta manera, los módulos instanciados asumirán el tipo de datos indicados. Actualmente, el acelerador soporta tipos de datos FP32, FP16, n-bit fixed-point, y enteros de n-bits.

HLSinf presenta un sistema mediante el uso de directivas *define* con el cual se pueden indicar los módulos a instanciar en la FPGA. De esta manera, el acelerador puede adaptarse a cada contexto. Esto se ejemplifica en figura 2.15, donde se está definiendo un sistema con los módulos ReLU y Pooling y el tipo de convolución directa. Como se puede observar, mediante este sistema también se define el tipo de datos u otros aspectos importantes para el acelerador como el CPI y el CPO. Mediante este sistema, el usuario es capaz de seleccionar únicamente aquellos módulos que desea implementar en la FPGA.

Finalmente, cabe mencionar que en HLS los argumentos de entrada de la función deben asociarse a un puerto AXI. Esto, como se ha explicado en la sección 2.4.1 es posible gracias a la directiva *interface*. En la tabla 2.1 se observa el sistema de puertos implementados, donde cada estructura de datos tiene asociado un puerto de datos AXI diferente.

```
#ifdef CONF_ALVEO_U280_8x8_DIRECT_FP32
#define ALVEO_U280
#define DIRECT_CONV
#define FP32_DATA_TYPE
#define USE_RELU
#define USE_POOLING
#define CPI              8
#define CPO              8
#define LOG2_CPO         3
#define WMAX           256
#define HMAX           256
#define READ_BURST_SIZE  2
#define STREAMS_DEPTH    2
#define INPUT_BUFFER_SIZE  65536
#endif
```

Figura 2.15: Definición del acelerador HLSinf.

Argumento de la función	Descripción del argumento	Puerto AXI
ptr_data	Datos de entrada sobre los cuales se ejecuta la convolución	gmem0
ptr_kernel	Datos de los filtros	gmem1
ptr_bias	Datos del bias	gmem2
ptr_out	Datos de salida	gmem3

Tabla 2.1: Puertos asociados a los argumentos de entrada del acelerador HLSinf.

2.7 EDDL

La librería EDDL (*European Distributed Deep Learning*) es una librería de código abierto utilizada para el aprendizaje profundo orientado a casos médicos. Este proyecto se encuentra dentro del proyecto H2020 DeepHealth[7] y dispone de una amplia documentación, guías de utilización [36] y ejemplos de uso y su código se encuentra disponible en [6].

Está desarrollada en lenguaje C++ y cuenta con soporte para diferentes arquitecturas y aceleradores, permitiendo la ejecución eficiente de los modelos de Redes Neuronales sobre las plataformas CPU, GPU y FPGA. En este proyecto, nos centraremos en el soporte y posterior estudio de prestaciones desarrollado para ejecuciones sobre arquitecturas FPGA[37].

La librería EDDL permite definir Redes Neuronales personalizadas, donde el usuario concatena capas de Redes Neuronales con el fin de crear un modelo desde cero. Asimismo, la EDDL también permite importar y exportar modelos en formato *Open Neural Network Exchange* (ONNX)[38], un formato utilizado para representar modelos de aprendizaje automático.

La API de EDDL se centra sobre los conceptos de tensores (matrices multidimensionales con un tipo de datos uniforme) y modelos de Redes Neuronales. La clase Tensor, además de agrupar aquello que concierne a tensores, también tiene el rol de actuar como la capa de abstracción de hardware (del inglés *Hardware Abstraction Layer* o HAL) donde los modelos y los tensores son creados e inicializados en la arquitectura especificado por el usuario mediante el objeto *computing service* (CS). Asimismo, todas las operaciones posteriores que involucren tensores se realizarán en el dispositivo especificado de manera transparente para el usuario[39].

CAPÍTULO 3
Estado del Arte

En este capítulo se expondrán los principales aceleradores de Redes Neuronales en FPGAs existentes. También se enumerarán las principales plataformas *software* orientadas al desarrollo de aplicaciones de Redes Neuronales. Finalmente, se describirán los modelos de Redes Neuronales orientado al reconocimiento de imágenes más influyentes actualmente.

3.1 Aceleradores en FPGA

Como se ha explicado anteriormente, la gran ventaja del acelerador HLSinf es aportar un gran grado de configurabilidad para crear diseños arquitectónicos sobre diferentes plataformas FPGAs adaptándose a sus recursos y necesidades. Sin embargo, es importante realizar un estudio para conocer los principales aceleradores de Redes Neuronales en FPGAs existentes. A continuación, se detallan algunos de ellos.

3.1.1. FINN *framework*

FINN es un *framework* experimental de Xilinx Research Labs orientado al proceso de inferencia de aceleradores de Redes Neuronales para FPGAs centrados en arquitecturas estilo flujo de datos (*dataflow*) explicado anteriormente[40].

FINN permite al usuario generar arquitecturas especializadas para modelos específicos de Redes Neuronales. Este *framework* proporciona el *frontend* donde se suele utilizar Brevitas[41], las etapas de optimizaciones y transformaciones y, finalmente, múltiples *backends* orientados a explorar los recursos y el rendimiento del diseño generado.

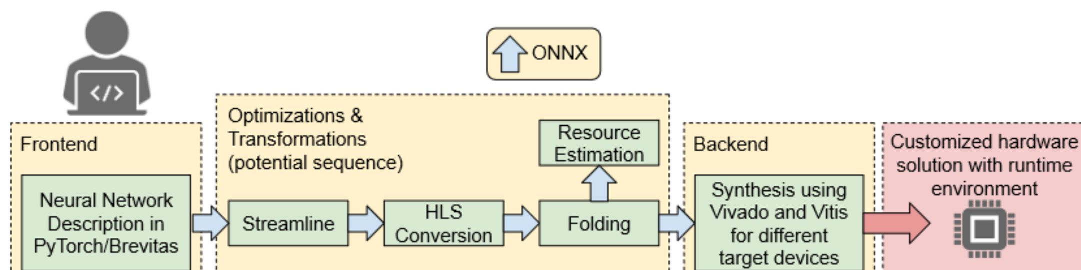

Figura 3.1: Flujo del compilador FINN [42].

En la figura 3.1 se puede observar el flujo de compilador FINN. En primer lugar, se llevan a cabo las tareas del *Frontend*. En este, se obtiene la descripción de un modelo

de Redes Neuronales y se exporta en formato ONNX. A partir de este modelo ONNX se genera un grafo para, posteriormente, generar transformaciones y optimizaciones sobre el mismo. Seguidamente, las operaciones de alto nivel del grafo se transforman a implementaciones existentes en la librería FINN, lo que da lugar a un grafo con capas instanciables en bloques hardware equivalentes, donde cada nodo del grafo corresponde a una llamada de función de VivadoHLS. Como resultado de todo lo anterior, se obtiene una representación y, a partir de esta, se genera el código necesario para el desarrollo del acelerador hardware. Finalmente, en la fase de *backend* se obtiene el código generado para realizar la síntesis del acelerador[42]. Este *framework* contiene un repositorio llamado FINN-HLS, el cual contiene descripciones de múltiples capas en lenguaje C++, las cuales hacen uso de la arquitectura de flujo de datos (*dataflow*) explicada anteriormente. FINN-HLS forma parte del backend de FINN y se integra con las herramientas de Xilinx.

3.1.2. Deep Learning Processor Unit

Xilinx ha desarrollado otro mecanismo para mejorar el rendimiento para el proceso de inferencia de Redes Neuronales en FPGAs mediante el desarrollo de un IP core (una unidad reutilizable de diseño lógico) conocido como Deep Learning Processor Unit (DPU).

La unidad DPU permite la implementación eficiente de Redes Neuronales Convolucionales entre los que destacamos los modelos VGG, YOLOv3 y ResNet. Además, contiene algunos parámetros configurables por el usuario para adaptar el uso de los recursos en función de la cantidad de recursos lógicos disponibles y personalizar diferentes características como el tamaño del canal u opciones para algunas funciones como la convolución o la operación softmax[43]. Asimismo, puede ser integrada al diseño deseado mediante el entorno Xilinx Vivado Design Suite.

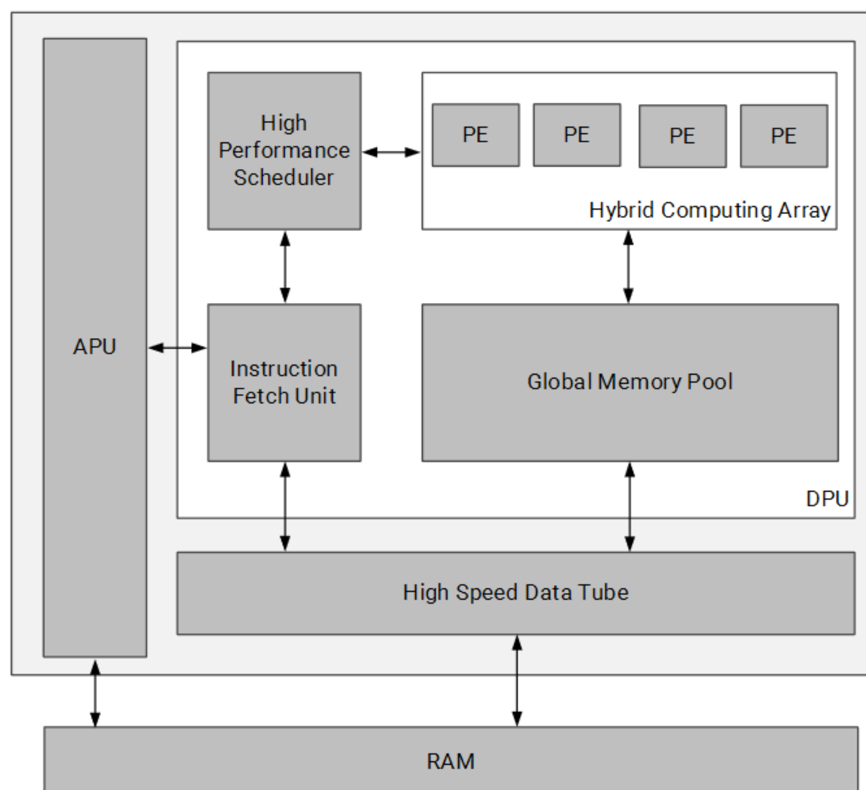

Figura 3.2: Arquitectura de una DPU[43].

En la figura 3.2 se puede observar las diferentes partes que componen la DPU. Podemos encontrar un módulo programador de alto rendimiento (*high performance scheduler module*), un módulo de matriz de cómputo híbrido (*hybrid computing array module*), un módulo de unidad de búsqueda de instrucciones (*instruction fetch unit module*) y un módulo de grupo de memoria global (*global memory pool module)*).

Sin embargo, aunque la DPU posee parámetros modificables por el usuario para alguna de sus funciones, ofrece una cantidad limitada de capas soportadas y, además, una solución cerrada e inflexible de las mismas. Esto se puede observar, por ejemplo, en el soporte que presenta la DPU para la capa LeakyReLU, donde el coeficiente está definido a 0.1 y no es modificable por el usuario. Además, la cantidad limitada e inflexible de capas soportadas limita los modelos soportados como es el caso de YOLOv4, el cual contiene funciones de activaciones como softplus o tangente hiperbólica no soportadas en la DPU.

3.1.3. Intel FPGA Deep Learning Acceleration Suite

La suite de Intel (FPGA Deep Learning Acceleration) sirve para realizar inferencia de modelos de Redes neuronales de manera simple en las arquitecturas FPGAs. Intel FPGA DL Acceleration Suite forma parte de OpenVINO toolkit (Open Visual Inference and Neural network Optimization), un conjunto de herramientas y librerías para dispositivos Intel orientadas al desarrollo de diferentes tipos de aplicaciones como reconocimiento automático de voz o el procesamiento del lenguaje natural[44]. Además, OpenVINO también permite la inferencia de Redes Neuronales Convolucionales, entre otros aspectos[44].

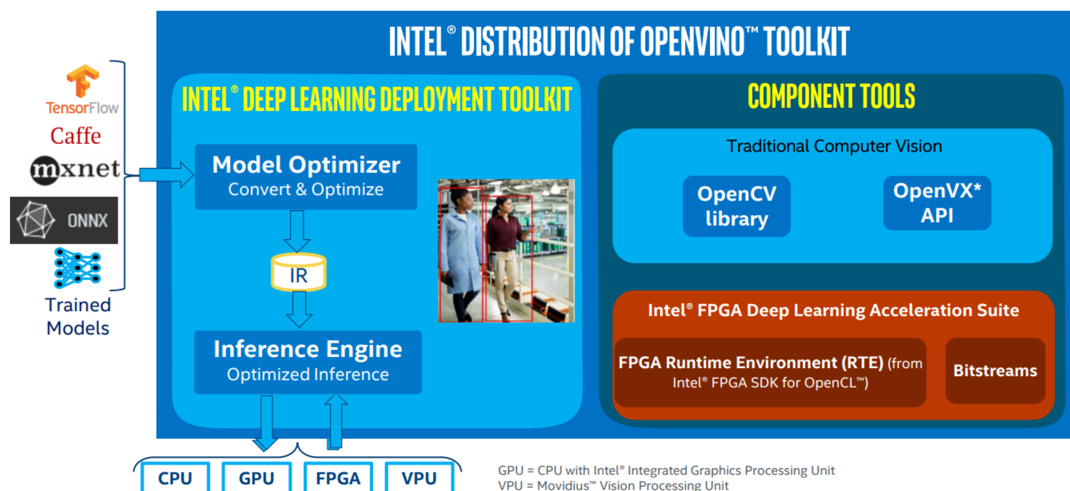

Figura 3.3: OpenVINO Toolkit [45].

En la figura 3.3 podemos observar las partes que componen OpenVINO, donde destacamos Intel Deep Learning Deployment Toolkit (conocido como DL/DT) y FPGA Deep Learning Acceleration Suite. En primer lugar DL/DT se utiliza para implementar Redes de Aprendizaje profundo a través de llamadas a funciones de alto nivel en C++. Podemos observar dos componentes:

- Model Optimizer: optimiza el modelo aportado y lo transforma en una representación intermedia (IR) de modelos de Redes Neuronales.
- Inference Engine: es un conjunto de bibliotecas C++ que proporcionan una API común para la inferencia sobre plataformas Intel. Es el encargado de leer la repre-

sentación intermedia (IR), establecer los formatos de entrada y salida y ejecutar el modelo en los dispositivos.

Por otra parte, Intel FPGA Deep Learning Acceleration Suite se encarga de programar la FPGA mediante Bitstreams[45].

3.1.4. Otros

Además de los aceleradores y las herramientas explicadas anteriormente, existen numerosos trabajos orientadas a implementaciones de aceleradores de Redes Neuronales en diferentes arquitecturas FPGAs. De los más recientes destacamos FP-DNN[46], un *framework* que genera una implementación hardware mediante RTL-HLS a partir de modelos descritos mediante TensorFlow[47]. También encontramos Caffeine, definida como una librería hardware/software reutilizable co-diseñada para acelerar eficientemente las Redes Neuronales Convolucionales (del inglés *convolutional neural network* o CNN) y Redes Neuronales Profundas (del inglés *deep neural network* o DNN) en FPGAs[48].

Todas estas plataformas son válidas para nuestro propósito en este trabajo. Ahora bien, HLSinf es un proyecto del grupo de investigación y, por lo tanto, tiene un mejor soporte para los miembros del grupo, así como una mejor predisposición para ser ampliado con nuevas funcionalidades como, por ejemplo, el soporte para *sparcity* (gestión eficiente de valores a cero) en futuros trabajos de investigación. Además, HLSinf obtiene las mismas prestaciones que las plataformas anteriores, superándolo a la capacidad de la FPGA en recursos.

3.2 Redes Neuronales

En los últimos años ha habido un indiscutible auge del uso de las Redes Neuronales en aplicaciones modernas. Mediante este tipo de tecnología se pueden desarrollar aplicaciones con funcionalidades muy diversas como el reconocimiento de imágenes, reconocimiento de voz e incluso son utilizadas para crear las recomendaciones efectivas de vídeos a los usuarios de la plataforma Youtube[49].

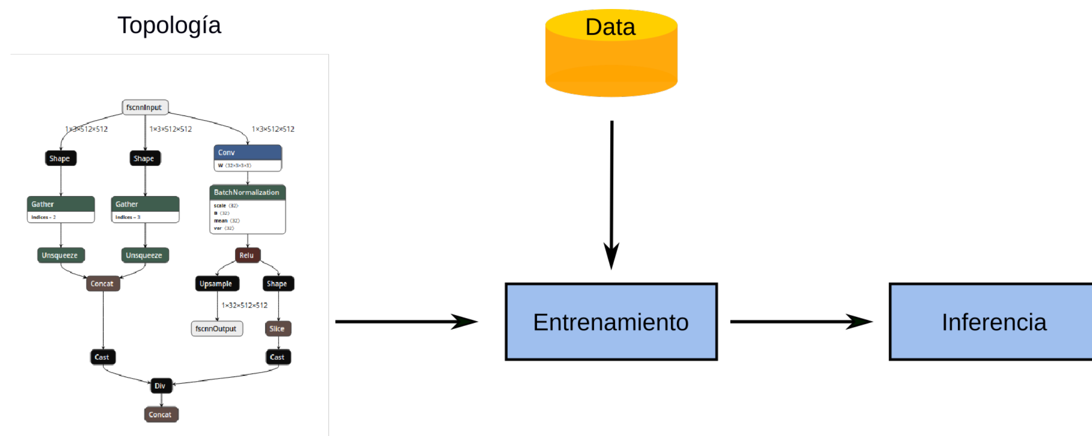

Figura 3.4: Flujo de una aplicación de Redes Neuronales.

Para desarrollar una aplicación de Redes Neuronales se ha de seguir unos pasos, donde cada uno de ellos puede y debe llevarse a cabo en diferentes plataformas hardware especializadas para ello. El conjunto de pasos puede verse en la figura 3.4. En primer

lugar, se ha de elegir la topología o modelo de Red Neuronal (por ejemplo VGG16, YOLO o GoogleNet). Esto se puede llevar a cabo mediante diferentes plataformas software existentes como Caffe [50], TensorFlow [47], Keras[51] o, como veremos en este proyecto, mediante la librería EDDL. Cabe mencionar que estos modelos pueden representarse en diferentes formatos como ONNX[38], Caffe[50], entre otros. En este proyecto utilizaremos la librería EDDL y modelos de Redes Neuronales en formato ONNX.

Si seguimos observando la figura anterior podemos observar que el segundo paso es entrenar la Red Neuronal utilizando para ello la máxima cantidad de datos posibles. Por ejemplo, para el entrenamiento de MNIST (una base de datos utilizada para el reconocimiento de números manuscritos) se suelen utilizar 60.000 imágenes para realizar el entrenamiento del modelo[52]. El proceso de entrenamiento es fundamental para la aplicación y suele requerir grandes cantidades de cómputo así como la necesidad de utilizar operaciones de coma flotante para aumentar la precisión de los modelos. Sin embargo, este paso no exige llevarse a cabo en tiempo real. Por estas razones, para este proceso, se suelen emplear GPUs de alto rendimiento, dado que este tipo de dispositivos poseen una gran cantidad de núcleos. Actualmente, existen múltiples GPUs las cuales alcanzan una gran tasa de FLOPS (*floating point operations per second*) donde destan las del fabricante NVIDIA. Actualmente, NVIDIA ha sacado al mercado GPUs con tecnología Tensor Core, las cuales pueden llegar a 100 teraFLOPS para aplicaciones de Redes Neuronales[53].

Finalmente, pasamos a la etapa de inferencia, la cual tiene como objetivo clasificar los nuevos datos de entrada que llegan a la aplicación utilizando el modelo ya entrenado con los datos generados en el paso de entrenamiento. Este paso, en algunos casos de uso, suele conllevar la necesidad de resolución en tiempo real y baja latencia, a diferencia del proceso de entrenamiento y, además, el proceso de inferencia no necesita operaciones de coma flotante para llevarse a cabo. Aunque las GPUs pueden usarse para la inferencia, presentan un alto consumo, llegando a 250W. Por lo que, en casos de uso en los que se debe priorizar un bajo consumo, como es el caso de SELENE, se debe optar por otro tipo de dispositivos *hardware*.

Todo lo expuesto anteriormente genera que los dispositivos FPGAs sean buenos para la etapa de inferencia, ya que presentan una gran configurabilidad, una mejor eficiencia energética y una menor latencia en comparación a las GPUs.

En este proyecto utilizaremos FPGAs para la ejecución de la inferencia de modelos de Redes Neuronales dado que disponemos de una red previamente entrenada.

3.3 Reconocimiento de imágenes

El reconocimiento de imágenes es una de las funcionalidades más populares dentro del campo de la Inteligencia Artificial. Este tipo de aplicaciones se usan en una gran cantidad de áreas tecnológicas como asistencia en la conducción o conducción autónoma, entre otros. Este tipo de aplicaciones tiene como objetivo determinar si existe cualquier instancia de un objeto dada una categoría (como animales, coches, señales) en una imagen devolviendo una representación espacial de dicho objeto si existe.

Los primeros modelos de Redes Neuronales orientados al reconocimiento de imágenes utilizaban el algoritmo de ventana deslizante. Este modelo se basa en obtener una región de la imagen y realizar la clasificación de dicha región para, posteriormente, mover la región y volver a realizar la clasificación hasta llegar al final de la imagen. Para obtener el objeto existente en la imagen se elige la predicción con la probabilidad más alta dentro del grupo de cientos o miles predicciones resultantes. Aunque este sistema funciona, se necesita una gran cantidad de cómputo para llevar a cabo el algoritmo.

Más tarde, surgió otro modelo llamado R-CNN (*Region-based Convolutional Network*)[54] el cual para evitar clasificar todas las regiones de la imagen, realiza una búsqueda con el fin de encontrar la región donde se encuentra el objeto a clasificar. Posteriormente, solo se clasifica dicha región, conllevando una mejora significativa en cuanto a cómputo y recursos.

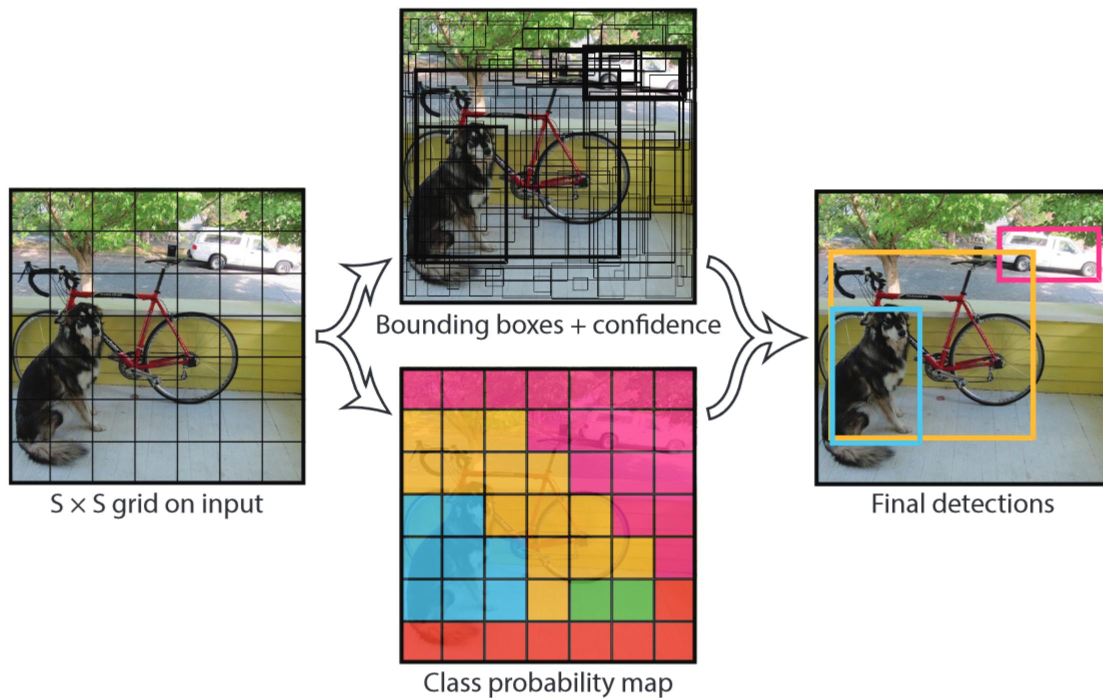

Figura 3.5: Funcionamiento de YOLO[55].

Sin embargo, en 2015, surge el modelo YOLO(*you only look once*) [55], el cual sigue un procedimiento diferente al anterior. YOLO es un modelo orientado a la detección de imágenes en tiempo real para predecir qué objetos se encuentran en la imagen y la posición de los mismos.

En la figura 3.5 podemos ver un esquema de su funcionamiento. En primer lugar, YOLO divide la imagen una malla de celdas y, para cada una de ellas, calcula los cuadros delimitadores (*bounding boxes*) donde se encuentra un objeto así como un puntuaje de confianza que refleja qué tan seguro está el modelo de que dentro del delimitador exista un objeto (*confidence*). Además, el modelo también predice cual es la clase a la que pertenece el objeto posicionado dentro de la celda (*class probability map*). A continuación, el puntuaje de confianza para los cuadros delimitadores se junta con las predicciones de clase de cada celda, generando probabilidades que indican la probabilidad de que exista un objeto de una clase específica dentro del cuadro delimitador. Finalmente, el modelo únicamente se queda con los delimitadores que tienen un alto puntuaje de confianza.

En este trabajo, utilizaremos una versión mejorada del modelo YOLO original conocida como YOLOv4[35] para el reconocimiento de imágenes.

CAPÍTULO 4
Optimizaciones en HLSinf

En este capítulo se describirá el desarrollo e integración de nuevas capas realizadas en el acelerador HLSinf. Concretamente, se explicarán los módulos STM y ADD y su importancia dentro del modelo de YOLOv4. Además, también se explicarán las modificaciones realizadas sobre otros aspectos del acelerador con el fin de aportar más flexibilidad para la ejecución de convoluciones de diferentes formatos.

4.1 Modelo YOLOv4

El modelo de YOLOv4 está compuesto por cientos de capas y tiene la característica de presentar una gran cantidad de repeticiones de los mismos patrones de aparición de capas en varios puntos de la arquitectura del modelo.

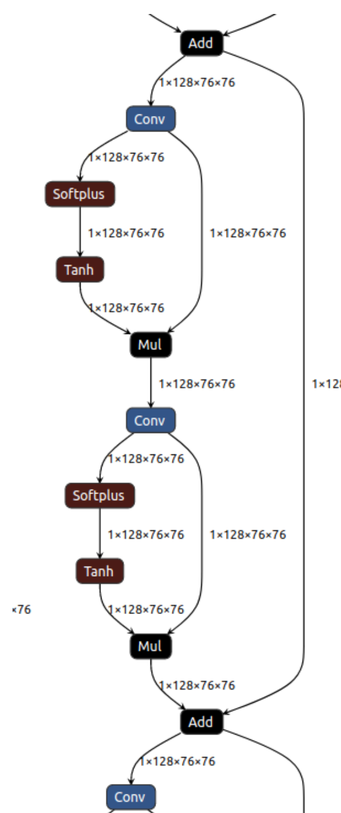

Figura 4.1: Nuevas capas de YOLOv4.

En la figura 4.1 se puede ver uno de estos patrones, el cual conforma el conjunto de operaciones más repetidas dentro de YOLOv4. Podemos observar que este modelo suele ejecutar una operación de convolución, seguida por una operación de softplus, otra operación de tangente hiperbólica para, finalmente, realizar una multiplicación elemento a elemento entre la salida del resultado de la tangente hiperbólica y la convolución previamente calculada.

Aunque el acelerador presenta actualmente soporte para múltiples capas, estas tres últimas operaciones no se encuentran soportadas actualmente. Por lo tanto, y dado que estas capas son las más repetidas dentro del modelo, se desarrollará e integrará un módulo orientado al soporte de este conjunto de capas, el cual finalmente será nombrado como módulo STM (nombrado así por las capas softplus, tangente hiperbólica y multiplicación).

Si seguimos observando la figura podemos ver otra capa representada como Add. Esta capa realiza la suma elemento a elemento entre las salidas de dos capas y, al igual que en el caso anterior, esta capa se puede observar durante todo el modelo YOLOv4, especialmente después del conjunto de capas STM. Para soportar esta capa, se desarrollará otro módulo independiente al cual llamaremos módulo Add.

4.2 Modelo VGG16

Otro modelo que se utilizará en este proyecto es una modificación del modelo VGG16 cuya arquitectura se encuentra soportada actualmente en el modelo HLSinf. El modelo completo puede observase en el apéndice A y está compuesto por 13 capas convolucionales, 16 capas de activación ReLU y 5 capas *max-pooling*, entre otros.

4.3 Capa STM

El primero módulo desarrollado se llama STM y se encarga de realizar un conjunto de operaciones orientadas a la inferencia de Redes Neuronales. Concretamente, está orientado a resolver las siguientes operaciones:

1. Función de activación **softplus**.
2. Función de activación **tangente hiperbólica**.
3. **Multiplicación elemento a elemento**.

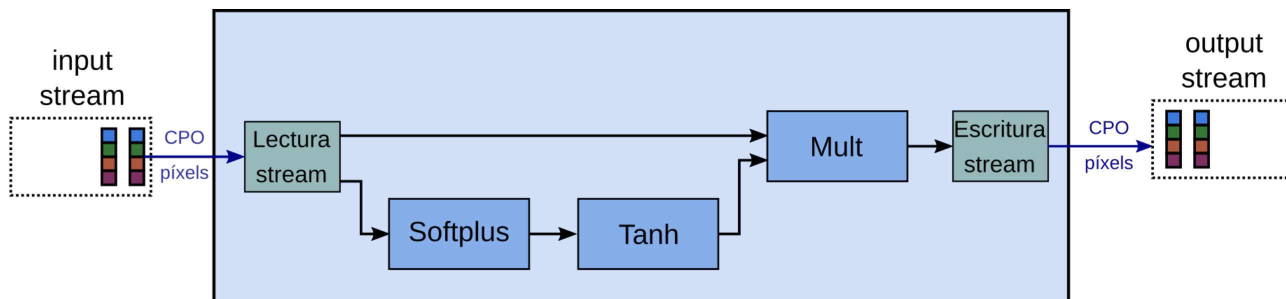

Figura 4.2: Esquema del módulo STM.

En la figura 4.2 puede observarse el flujo de estas tres funciones dentro del módulo STM. Como se puede observar, en primer lugar, se realiza una lectura del *stream* de datos

de entrada. Seguidamente, se realiza la función softplus. A partir del resultado de la función softplus se llevará a cabo la función de activación tangente hiperbólica (representada como Tanh). A continuación, la función de multiplicación (representada como mult) multiplicará elemento a elemento los datos resultantes de la operación Tanh juntamente con los mismos datos que han alimentado previamente la función softplus que, típicamente, serán los datos resultantes de la operación de convolución. Finalmente, se guardará el resultado en el *stream* de datos de salida.

La razón por la cual se ha decidido agrupar las tres capas en un único módulo es para obtener como resultado un módulo completamente especializado para el modelo YOLOv4, ya que la eficiencia que podría observarse en comparación a la utilización de tres módulos independientes es alta. Esto es debido a que, con nuestra implementación, solo es necesario recorrer los datos una única vez para la ejecución de los tres algoritmos. Además, como hemos mencionado anteriormente, estas tres capas se encuentran múltiples veces repetidas en el mismo orden, lo que hace que esta implementación sea posible y conveniente.

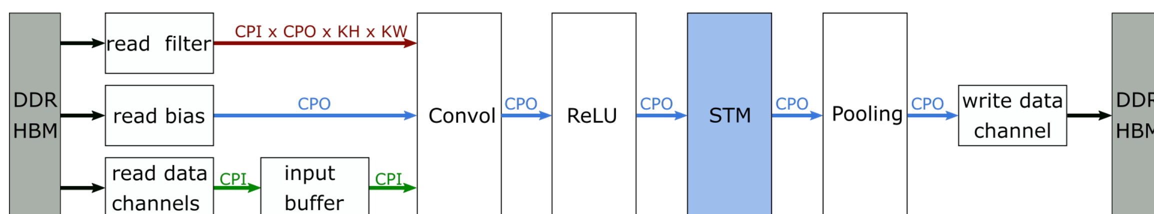

Figura 4.3: Arquitectura de HLSinf con el módulo STM.

Este módulo se ha diseñado para poder ser ejecutado tras el módulo de activación ReLU y antes del módulo Pooling descritos en la sección . En la figura 4.3 se puede observar la integración de este módulo en la arquitectura anterior descrita en 2.6. Como se puede observar, este nuevo módulo se posiciona entre los módulos ReLu y Pooling. Además, los datos de entrada del modelo STM vienen mediante el *stream* de datos resultante del módulo ReLu y, a la vez, los datos de salida alimentan el módulo Pooling mediante otro *stream* de datos.

Al igual que en los módulos preexistentes, el módulo STM se conecta a los módulos vecinos mediante las estructuras de datos *streams*, siguiendo la flujo de datos descritos en la sección 2.6.

Como se ha explicado en la sección 2.3 tanto las capas softplus como tangente hiperbólica están definidas mediante funciones matemáticas. Estas funciones, aunque pueden ser implementadas manualmente, se han implementado mediante las funciones que ofrece la librería Vitis HLS Math Library, dado su facilidad de uso. Concretamente, para las funciones se han utilizado las versiones de simple precisión.

Como hemos explicado en 2.6, HLSinf presenta un sistema mediante *defines* para elegir los módulos deseados en la implementación en la FPGA. Como se puede observar en la figura 4.4, el módulo STM se ha añadido a este diseño para permitir la integración de este nuevo módulo dentro de la arquitectura existente.

4.3.1. Optimizaciones HLS

En el siguiente pseudocódigo 4.1 se refleja la estructura seguida para el desarrollo de este módulo donde se encuentran los pragmas HLS utilizados con el fin de conseguir una implementación eficiente. Concretamente, se observan los pragmas *array partion*, *pipeline* y *unroll* explicados en la sección 2.4.1.

```
#ifdef CONF_ALVEO_U200_4x4_DIRECT_STM_FP32
#define ALVEO_U200
#define DIRECT_CONV
#define FP32_DATA_TYPE
#define USE_STM
#define USE_RELU
#define USE_POOLING
#define CPI                    4
#define CPO                    4
#define LOG2_CPO               2
#define WMAX                1024
#define HMAX                 256
#define READ_BURST_SIZE        4
#define STREAMS_DEPTH          4
#define INPUT_BUFFER_SIZE  65536
#endif
```

Figura 4.4: Definición del acelerador HLSinf.

```
1  Pragma ARRAY PARTITION de variable auxiliar data_in
2  Pragma ARRAY PARTITION de variable auxiliar data_out
3  Desde 0 hasta H * W
4    Pragma PIPELINE con II=1
5    Leemos el grupo de píxeles actual con tamaño CPO y lo almacenamos en la
         variable auxiliar data_in
6    Desde 0 hasta CPO
7      Pragma UNROLL
8      Leemos el pixel de data_in
9      Si el parámetro de activación del modulo STM está activado
10       Función softplus
11       Función Tanh al resultado de la ejecuón de softplus
12       Multiplicamos el píxel vírgen de entrada por el resultado de tanh
13       Escribimos el resultado en la variable auxiliar data_out
14     Sino
15       Escribimos directamente el píxel de entrada vírgen en la variable
             auxiliar data_out
16     Fin Si
17   Fin Desde
18   Escribimos el resultado en el stream de datos data_out para generar la salida
19 Fin Desde
```

Listing 4.1: Pseudocódigo del módulo STM.

Pipeline

En este módulo, este pragma se encuentra en la linea 3 del código 4.1 y se ha especificado para reducir el intervalo de iniciación a uno independientemente de la duración de cada iteración, consiguiendo un procesamiento de iteraciones cada un ciclo. Por lo tanto, con este pragma podemos aumentar considerablemente las prestaciones.

Unroll

Otro pragma de gran interés utilizado en este módulo es el *unroll*. Este pragma se encuentra en la linea 7 del código 4.1 y se encuentra dentro del segundo bucle. Tiene el objetivo de desenrollar el bucle completamente con el fin de crear CPO copias del código

en RTL para permitir la ejecución de las funciones en paralelo y, por tanto, procesar CPO píxeles a la vez.

Array partition

Como se ha explicado anteriormente, este pragma sirve para almacenar los datos en múltiples memorias o registros con el objetivo de realizar más de una lectura o escritura simultáneamente. En esta función, estos pragmas se utilizan sobre las variables auxiliares data_in y data_out, las cuales tienen un número de elementos igual a CPO que pueden ser procesados paralelamente. Concretamente, data_in contiene los datos de entrada necesarios para llevar a cabo toda la funcionalidad requerida.

Cabe destacar que, dado que el bucle interno ha sido desenrollado gracias al pragma *unroll* expuesto anteriormente, si no se utiliza el pragma *array partition* no se podrán realizar CPO lecturas a la vez y no será posible realizar más de una iteración del bucle simultáneamente, debido a que en todas las iteraciones se necesita realizar una lectura a la variable data_in (aunque en diferentes posiciones).

4.4 Capa Add

El móludo Add es el segundo módulo añadido al acelerador y, como se puede observar en la figura 4.5. Este módulo se posiciona entre el módulo Pooling y el de la función *write data channel* y se encarga de realizar la suma elemento a elemento entre las salidas de dos capas. Concretamente, realiza la suma entre la salida resultante de la ejecución del módulo anterior y la salida de otra capa. Los datos de esta última capa se leerán mediante la nueva función representada en la figura como *read input add*. Esta función sigue una estructura similar a la función preexistente *read data channels* la cual, a partir de un conjunto de datos pasados al acelerador, crea un nuevo *stream* de datos con la intención de alimentar al módulo correspondiente.

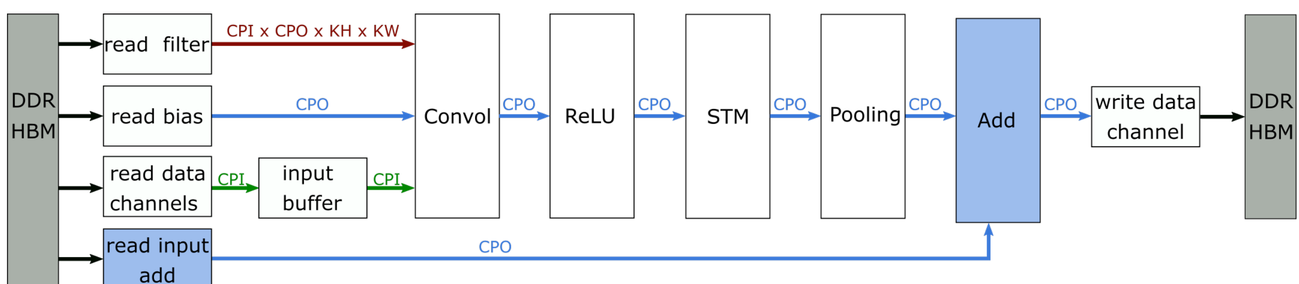

Figura 4.5: Arquitectura del acelerador HLSinf con el nuevo módulo Add

De la misma manera que los módulos anteriores y el módulo STM, el módulo Add también se encuentra conectado a los módulos vecinos mediante las estructuras de datos *streams*. Además, en este módulo se utilizan de igual manera las directivas HLS empleadas en el módulo STM.

Además, se ha creado un nuevo argumento de entrada para la lectura de los datos provenientes de la capa externa, es decir, aquella sobre la cual no se aplica la operación de convolución ni ejecuta ningún otro módulo. No obstante, esto crea la necesidad de añadir una nueva directiva HLS *interface* con el fin de crear y asociar un nuevo puerto AXI a este nuevo argumento de entrada. Como resultado, se puede observar la configuración final de los puertos en la tabla 4.1, donde se puede ver que se ha asociado el puerto gmem5 a los nuevos datos de entrada del módulo Add (*ptr_data_add*).

Argumento de la función	Descripción del argumento	Puerto AXI
ptr_data	Datos de entrada sobre los cuales se ejecuta la convolución	gmem0
ptr_kernel	Datos de los filtros	gmem1
ptr_bias	Datos del bias	gmem2
ptr_out	Datos de salida	gmem3
ptr_data_add	Datos de entrada de la segunda capa para la función de suma elemento a elemento	gmem5

Tabla 4.1: Nuevo puerto AXI asociado al argumento de entrada del módulo Add.

A diferencia del módulo ReLU, STM y Pooling, el módulo Add se instancia en todas las implementaciones dado que se ha excluido del sistema de *defines* anteriormente descrito. No obstante, si que cuenta con un parámetro en el que se indica la activación de este módulo y la lectura de datos de la segunda capa (mediante el módulo read input add) en cada ejecución.

4.5 Capa LeakyReLU

Además de las capas anteriormente descritas, YOLOv4 también tiene presente en su modelo múltiples capas de activación LeakyRelu (variante de la función de activación ReLU explicada en la sección 2.3.1). Dentro de este modelo, estas capas suelen estar posicionadas a continuación de una capa convolución, generando la posibilidad de agruparlas para crear una capa fusionada. Aunque el acelerador no tiene soporte inicial completo para esta función de activación, sí presenta soporte para la función de activación ReLU, lo cual facilita en gran medida la implementación de la función LeakyReLu.

Es por esto que se ha decidido modificar el módulo ReLU con el fin de crear un único módulo polivalente, el cual sea capaz de ejecutar tanto la función ReLU como la función LeakyReLU. Esto se ha llevado a cabo mediante la incorporación de un nuevo parámetro a la función ReLU, el cual indica el factor por el cual se multiplicarán los píxeles con valores menores a cero. De esta manera, si se desea ejecutar la función ReLU, este factor deberá de definirse con valor cero. Por lo contrario, si se desea ejecutar la función LeakyRelu, se deberá definir el factor con el valor requerido.

4.6 Soporte a otros tipos de convolución

Como hemos mencionado anteriormente, el acelerador HLSinf incluye varios parámetros que pueden ser definidos por el usuario y definirán el tipo convolución. Sin embargo, esto no ocurre con el *stride* y el *padding* cuyo valor se encuentra predefinido dentro del código por lo que es fijo e invariable. Esto provoca que solo se puedan ejecutar convoluciones con tamaño de *stride* de 3 x 3 y *padding* de uno en cada extremo.

En la tabla 4.2 se muestran los *paddings* y *strides* definidos para las convoluciones existentes dentro del modelo de YOLOv4. Como se puede observar, la limitación descrita anteriormente afecta en gran medida al modelo YOLOv4 ya que el cual posee múltiples convoluciones con tamaños de *stride* y *padding* variados. Por lo tanto, con el soporte

existente hasta ahora, solo podrían ejecutarse menos del 40 % de las convoluciones en el acelerador HLSinf. Por esta razón, se ha decidido aumentar el grado de flexibilidad del acelerador y permitir al usuario definir los parámetros de *stride* y *padding* que conforman una convolución.

Paddings	Stride	Número de apariciones
0x0x0x0	1x1	5
1x1x1x1	1x1	37
1x1x1x1	2x2	7
0x2x0x2	1x1	61

Tabla 4.2: Parámetros de las convoluciones del modelo YOLOv4.

4.6.1. Modificación *Padding* para soporte multivalor

Para permitir la ejecución de convoluciones con tamaño de *padding* variable, se han implementado un sistema en el cual, mediante varios argumentos de entrada, se indica el valor del *padding* de arriba (PT), abajo (PB), izquierdo (PL) y derecho (PD). De esta manera, el usuario no solo indica si desea activar el padding en cualquier de los extremos sino que también es capaz de indicar el valor de cada uno de ellos.

```
 h = 0;
 w = 0;
Desde 0 hasta numero de píxeles
  int realizar_padding_arriba  = h < PT;
  int realizar_padding_abajo   = h >= H + PT;
  int realizar_padding_derecha = w < PL;
  int realizar_padding_derecha = w >= (W + PL);
  Si algun realizar_padding activado
    data = zero;
  Sino
    data = in.read();
  Fin Si
  Escritura del dato "data" en el stream de datos
  w = w+1;
  Si w == W + PL + PR
    w = 0;
    h = h + 1;
    Si h == H + PT + PB
      h = 0;
    Fin Si
  Fin Si
Fin Desde
```

Listing 4.2: Pseudocódigo de la función de *padding*.

En el pseudocódigo 4.2 se puede observar la estrategia seguida para la gestión del *padding*, donde no se han incluido las directivas de HLS dado contiene las mismas directivas que el módulo STM.

4.6.2. Modificación CVT para soporte de stride multivalor

Para la ejecución de convoluciones con tamaño de *stride* variable, se han implementado, al igual que en el caso anterior, un conjunto de argumentos de entradas los cuales permiten indicar el valor de ambos tamaños de *stride*. De esta manera, se ha creado, en

primer lugar, el parámetro *stride* de altura definido como SH y, en segundo lugar, el *stride* de amplitud definido como (SW).

Posteriormente, se ha adaptado la función responsable de realizar el *stride* sobre los datos de entrada de la convolución para que pueda realzar un deslizamiento de la ventana variable. Como resultado, el usuario es capaz de definir ambas dimensiones del *stride* (SH y SW) permitiendo la ejecución de nuevos tipos de convoluciones.

CAPÍTULO 5
Adaptación EDDL

En este capítulo se exponen y explican las modificaciones llevadas a cabo en la librería EDDL. Concretamente, se describirán los cambios realizados para la portabilidad de los nuevos módulos y el desarrollo implementado para hacer posible la ejecución de capas sobre el acelerador HLSinf.

5.1 Introducción

Previamente a este trabajo, la EDDL ya presentaba un soporte básico de gestión de memoria para las arquitecturas FPGAs. Para ello, cada vez que se define el modelo de Red Neuronal debe indicarse el dispositivo *hardware* mediante el parámetro *compute service*. De esta manera, la EDDL gestionará la memoria del modelo sobre el dispositivo *hardware* indicado.

Aunque en este proyecto se han soportado las capas con más carga computacional de las Redes Neuronales así como las más utilizadas dentro del modelo YOLOv4, actualmente HLSinf no presenta soporte para todos los tipos de capas que componen este modelo. Por esta razón, para la ejecución de las capas no soportadas en la FPGA se utilizará, por defecto, la CPU.

Sin embargo, existen otros aspectos necesarios para la integración del acelerador HLSinf con la librería EDDL, los cuales se abordan en este proyecto. En primer lugar, el formato de datos utilizado en el acelerador difiere del formato de datos utilizado en la librería EDDL para la ejecución sobre la CPU, lo cual crea la necesidad de desarrollar un sistema de transformación de formatos en ambos sentidos. Además, el acelerador HLSinf está formado mediante la unión de diversos módulos. Esto fuerza la ejecución de múltiples capas en la misma llamada. Todo esto será resuelto más adelante mediante la solución propuesta e implementada en este proyecto.

5.2 Adaptación de formatos de datos

El primer aspecto a tener en cuenta para la generación del modelo en FPGAs es el formato de datos utilizados en el acelerador HLSinf y en la librería EDDL. Esta librería crea los modelos en la CPU con un formato de datos típico, donde cada imagen está compuesta por C x H x W. Como resultado, los datos de un canal se encuentran contiguos en memoria de manera secuencial. Sin embargo, y como se ha explicado anteriormente, el acelerador HLSinf tiene una organización de memoria basada en grupos de canales (CPI)

donde los datos de CPI canales se intercalan en la memoria, consiguiendo una imagen con dimensiones G x H x W x CPI, donde G se define como $G = C/CPI$.

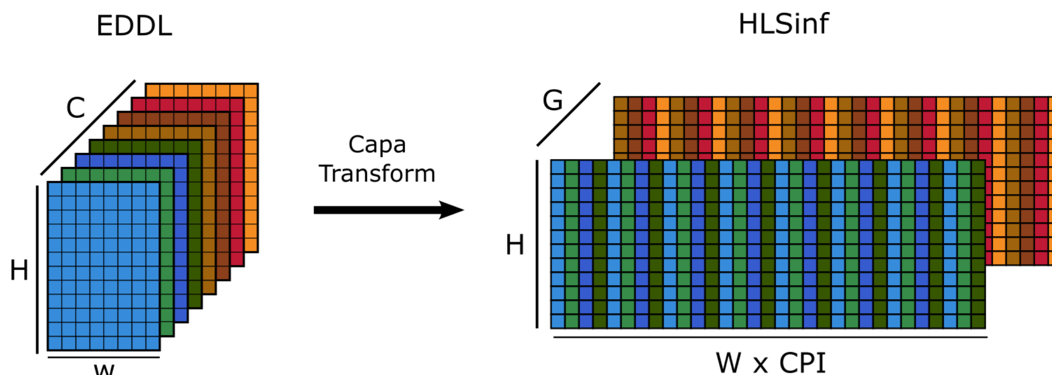

Figura 5.1: Efecto de la capa Transform con CPI igual a cuatro.

Para soportar esta variedad de formatos se ha creado la capa *Transform*. Esta capa tiene el objetivo de leer la imagen en formato C x H x W y transformar los datos al formato G x H x W x CPI o viceversa. Además, esta capa se encarga de asegurar que el número de canales sea múltiplo de CPI, dado que este es un requisito para ejecutar correctamente las capas en el acelerador.

El efecto de la capa *Transform* se encuentra representado en la figura 5.1, donde cada canal se ha representado con un color diferente. En esta figura se plasma una imagen de ocho canales y se asume un CPI igual a cuatro. A partir de una imagen en formato C x H x W (donde la memoria se encuentra organizada de manera que los datos de un canal se encuentran contiguos en memoria), los datos de la imagen se formatean en formato G x H x W x CPI. De esta manera, se puede observar como se generan dos grupos de datos, donde los datos de CPI canales quedan alternados, y donde el primer grupo contiene los datos de los primeros CPI canales y el segundo grupo contiene los datos de los segundos CPI canales.

Esta capa será necesaria cuando los formatos de datos entre dos capas no coincidan. Típicamente, esto se dará cuando el modelo tenga una capa no soportada en la FPGA y, por lo tanto, deba de ser ejecutada en la CPU con formato de datos C x H x W seguida de otra capa sñi soportada en FPGAs, la cual necesita los datos en formato G x H x W x CPI o viceversa. Por lo tanto, esta capa ha de posicionarse en medio de dos capas con distinto formato de datos para transformar los datos de la primera capa al formato de la segunda capa.

5.3 Capas fusionadas

El acelerador HLSinf sigue un modelo de capas fusionadas en el cual en cada llamada el acelerador puede ejecutar hasta cinco capas (Convolucional + ReLU/LeakyReLU + STM + Pooling + Add). Esto conlleva la necesidad de soporte, hasta ahora inexistente, para capas fusionadas dentro de la EDDL. Además, es importante que la utilización de capas fusionadas para la creación de modelos sea transparentes hacia el usuario, de manera que este únicamente describa las capas que conforman el modelo de manera no fusionada y, en post-procesado, la librería sea capaz de agrupar las capas e instanciar unas nuevas capas fusionadas. De esta manera, el usuario no tiene que preocuparse de qué capas fusionadas se encuentran soportadas y, además, la definición de un mismo modelo puede ser reutilizado en otro dispositivo sin soporte de capas fusionadas.

Por esta razón, se ha desarrollado una función que tiene como objetivo leer un modelo sin las capas fusionadas y generar un nuevo modelo para arquitecturas FPGAs con las mismas capas fusionadas. Finalmente, en este proyecto se ha desarrollado soporte para las siguientes capas fusionadas:

- Convolucional + ReLU.
- Convolucional + LeakyReLU.
- Convolucional + Maxpool.
- Convolucional + ReLU + Maxpool.
- Convolucional + Softplus + Tangente hiperbólica + Multiplicación elemento a elemento.
- Convolucional + Softplus + Tangente hiperbólica + Multiplicación elemento a elemento + Suma elemento a elemento.

5.4 Clase HLSinf

Para la creación y gestión de capas fusionadas en el acelerador HLSinf se ha implementado una clase llamada HLSinf. De esta manera, cada vez que se requiera la creación de una capa fusionada para su ejecución sobre el acelerador, se creará una objeto HLSinf cuyos atributos definirán tanto el tipo de convolución como las capas soportadas a ejecutar (Convolución, ReLU, *Pooling*, STM o Add). En la tabla 5.1 se describen los argumentos de interés para este proyecto de la clase HLSinf. Por lo tanto, mediante este sistema de activación de variables, se crea un registro de las capas que conformarán la capa fusionada para, posteriormente, indicar al acelerador qué módulos debe ejecutar.

Atributo	Descripción
H, W	Geometría de los datos de entrada
Ichannels	Canales de entrada
Ochannels	Canales de salida
KH, KW	Dimensiones del filtro
SH, SW	Dimensiones del *stride*
PT, PB, PL, PR	Dimensiones del *padding*
enable_relu	Capa ReLU activada
relu_factor	Factor de la capa ReLU
enable_maxp	Capa Maxpooling activada
enable_add	Capa Add activada
enable_stm	Capa STM activada

Tabla 5.1: Atributos de interés de la clase HLSinf.

Como se ha mencionado anteriormente en la sección 2.5, el acelerador se ejecuta sobre un dispositivo FPGA. Esto crea la necesidad de implementar una comunicación entre la FPGA y la EDDL. En este proyecto, esta comunicación se ha llevado a cabo gracias a OpenCL, ya que permite la comunicación en sistemas heterogéneos.

Por lo tanto, se ha implementado un sistema de funciones en la clase HLSinf la cual permite la ejecución de las capas fusionadas sobre el acelerador. Concretamente, estas funciones se encargan de lo siguiente:

1. Crear *buffers* de memoria OpenCL.
2. Soporte *multikernel*.
3. Establecer los argumentos de un kernel mediante OpenCL.
4. Lanzar la ejecución.

La creación de *buffers* de memoria OpenCL se realiza para las estructuras de datos involucradas en el proceso de convolución. Concretamente para los datos de la imagen, del filtro y el bias. Además, si se requiere la ejecución de la caapa Add, también se creará un *buffer* para los datos provenientes de la capa padre de la capa Add.

Por otra parte, el soporte *multikernel* se ha implementado para balancear la carga computacional de una ejecución en el acelerador HLSinf entre más de un *kernel*, si se encontrara más de uno instanciado en la FPGA. Este soporte se explica más detalladamente a continuación.

5.4.1. Soporte multikernel

Dado que el acelerador HLSinf se ha diseñado alrededor de concepto de canales por entrada/salida (CPI/CPO), los cuales suelen estar definidos con valores de 4, 8 o 16, no es de extrañar que, en la mayoría de convoluciones ordinarias, se deban de realizar varias iteraciones para calcular en su totalidad el resultado, ya que el número de canales suele sobrepasar en gran medida estos valores. La cantidad de iteraciones a realizar, dentro del contexto del acelerador HLSinf, se conocen como O_ITER y se calculan como $O_ITER = Ochannels/CPO$.

HLSinf tiene implementado un sistema en el cual se puede indicar el número de iteraciones a realizar por parámetro de entrada, donde la ejecución de cada iteración calculará una parte del resultado. Por lo tanto, gracias a este sistema, se pueden dividir las iteraciones entre varios *kernels*, los cuales pueden ejecutarse en paralelo con la finalidad de mejorar el rendimiento (*throughput*) total de la ejecución. Además, se han utilizado directivas OpenMP[56] para permitir la ejecución en paralelo de ambos *kernels*.

```
O_ITER = Ochannels / CPO
Si núm_kernels es igual a 1 o O_ITER < núm_kernel
  primera_iteración = 0
  última_iteración = (Ochannels / CPO) - 1
  Ejecución en HLSinf desde primera_iteración hasta última_iteración
Sino
  o_iter_per_kernel = O_ITER / núm_kernel
  Si O_ITER > (o_iter_por_kernel * núm_kernel)
    extra_iter = O_ITER - o_iter_per_kernel * núm_kernel;
  Fin Si
  #pragma omp parallel for
  Desde 0 hasta núm_kernel
    Si primer kernel
      primera_iteración = 0
      última_iteración = o_iter_por_kernel + extra_iter - 1
    Sino
      primera_iteración = o_iter_per_kernel * num_kernel_actual + extra_iter
      última_iteración = primera_iteración + o_iter_per_kernel - 1;
    Fin Si
    Ejecución en HLSinf desde primera_iteración hasta última_iteración
    Fin Desde
  Sino
Sino
```

Listing 5.1: Pseudocódigo de la gestión multikernel.

En el pseudocódigo 5.1 se puede observar la estrategia desarrollada para la gestión de iteraciones para varios *kernels*.

5.5 Generación del modelo

Dadas las dificultades expuestas anteriormente, se ha desarrollado un procedimiento para solventarlas, el cual consiste en el desarrollo de una función (llamada *model_for_fpga*) que tiene como objetivo leer un modelo sin capas fusionadas creado en la CPU, crear uno nuevo con capas fusionadas en la FPGA y devolverlo.

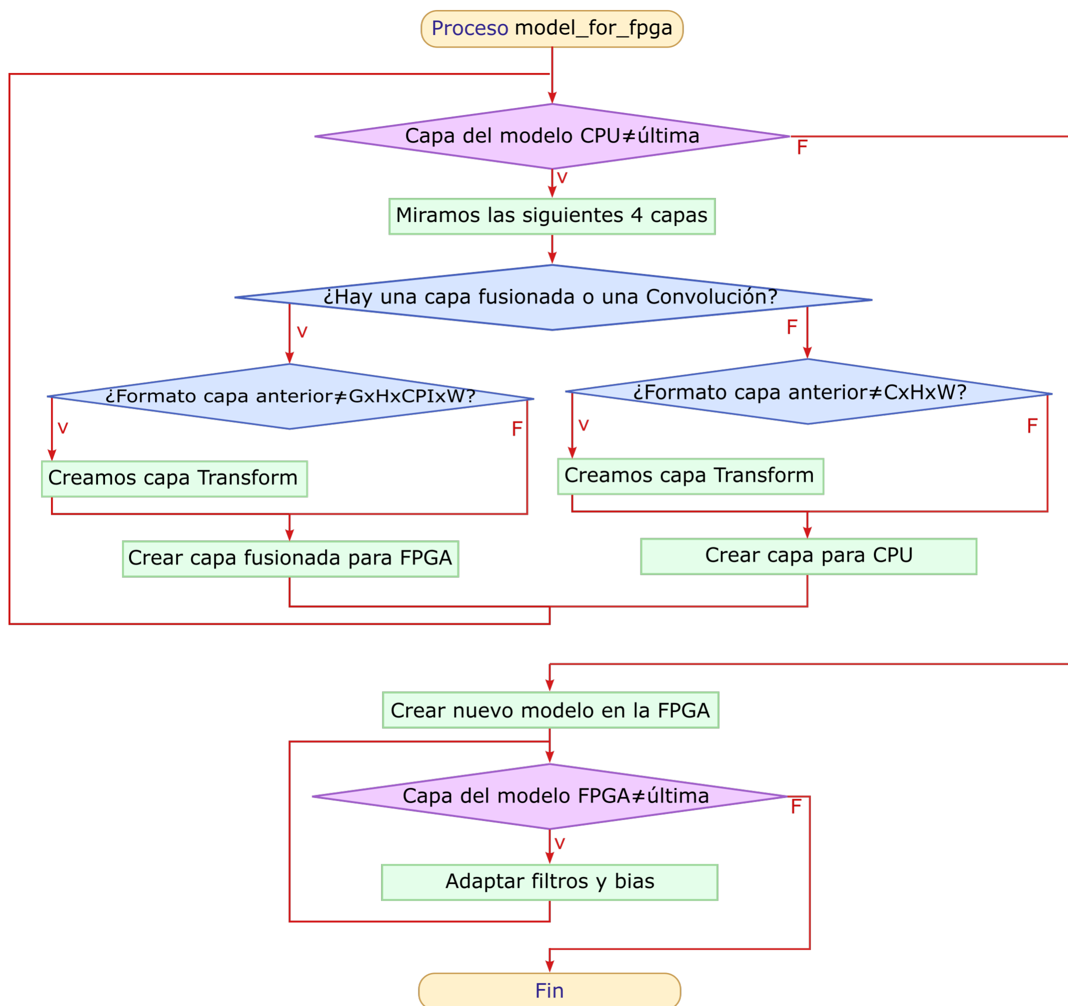

Figura 5.2: Generación de modelos de capas fusionadas en la FPGA.

Los pasos de esta función pueden observarse en la figura 5.2. En primer lugar, esta función se encarga de leer las capas que conforman el modelo de entrada buscando patrones de capas con el fin de fusionarlas. Esto se encuentra representado en dicha figura por los pasos 1 y 2 y será explicado con mayor detalle en la sección 5.5.1.

En caso de sí encontrar un conjunto de capas a fusionar y, dado que todas las capas fusionadas tienen como primera capa una convolucional, se comprueba si la capa padre a la capa convolucional en el modelo de la FPGA tiene el mismo formato de datos que la capa convolucional (paso 4). En caso de diferir el formato de datos, esta función crea una capa *Transform*, la cual transformará los datos de formato CxHxW a formato GxHxWxCPI (paso 6). Esta nueva capa se intercalará entre la capa fusionada y la capa padre de la misma. Finalmente, crea la capa fusionada en la FPGA. Además, en caso de que la función

no encuentre ninguna capa a fusionar, pero sí encuentre una sola capa convolucional, el procedimiento es el mismo, con la diferencia de que no se creará una capa fusionada, sino una capa convolucional con el formato y los parámetros adaptados para su ejecución en la FPGA. Cabe mencionar que el proceso de encontrar la capa padre de una capa en el modelo es un proceso complejo que se explicará en la sección 5.5.2.

Por lo contrario, si no encuentra ninguna capa a fusionar o una capa convolucional, esta función creará una capa adaptada para ser ejecutada en la CPU. Esta adaptación consiste en comprobar si la capa padre del modelo de la FPGA tiene el mismo formato de datos que la capa a instanciar (paso 5). Si, por lo contrario, tiene un formato adaptado a la FPGA (GxHxCPIxW), la función creará una capa *Transform* la cual transformará los datos de formato GxHxCPIxW a formato CxHxW y se posicionará entre la capa padre y la nueva capa a crear (paso 7 y 9).

Finalmente, con todas las capas del modelo de entrada procesadas, se crea el modelo de la FPGA a partir de todas las nuevas capas instanciadas (paso 10). Sin embargo, es necesario realizar una adaptación de los filtros y el bias (pasos 11 y 12). Cabe mencionar que para definir los parámetros que componen cada capa se toman como referencia los parámetros de las capas del modelo original. Estos son, por ejemplo, las dimensiones que definen la imagen de entrada como el número de canales. No obstante, existen algunos casos en los cuales estos parámetros han de ser recalculados con el fin de satisfacer los requisitos del acelerador HLSinf. Esto se explicará con más detalle en la sección 5.5.3.

5.5.1. Búsqueda de capas fusionadas

La EDDL tiene implementado un sistema que permite identificar el tipo de cada capa. De esta manera, se puede saber si la capa es una capa de entrada (*input*), una capa de activación o una capa convolucional, entre otros. Además, dado que existen múltiples tipos de capas de activación, estos tipos de capas tiene implementado un parámetro que permite determinar de qué tipo de capa de activación se trata la capa de interés. Mediante este mecanismo, la función *model_for_fpga* es capaz de reconocer cada capa con el fin de fusionar aquellas posibles.

Para determinar qué capas se pueden fusionar, esta función recorre todo el modelo posicionándose en una capa para, en la misma iteración, identificar tanto la capa en la que se encuentra posicionado como las siguientes cuatro capas. El número de capas a identificar en cada iteración viene definido por la capa fusionada con mayor número de capas que la conforman, la cual actualmente se trata de capa fusionada Conv+STM+Add, una capa formada por las capas convolucional, softplus, tangente hiperbólica, multiplicación elemento a elemento y suma elemento a elemento.

Cabe destacar que esta función siempre creará la capa fusionada con el mayor número posible de capas. Esto significa que si nos encontramos ante un modelo que contiene las capas convolucional, ReLU y *max-pooling*, la función *model_for_fpga* agrupará las tres capas y generará una única capa aunque exista la opción de generar una capa fusionada con una menor cantidad de capas compuesta solo por las capas convolucional y ReLU.

5.5.2. Gestión de capas padre

Como hemos mencionado en la descripción de la función de generación de modelos de capas fusionadas, uno de los pasos consiste en buscar e identificar la capa padre. Este proceso, consta de gran complejidad por varias razones.

En primer lugar, aunque existen algunos modelos formados por capas posicionadas de manera que crean una malla de una dimensión (como VGG16), existen otros modelos

cuyas capas forman una malla de dos dimensiones. Esto ocurre, por ejemplo, en algunos puntos de la arquitectura del modelo YOLOv4, representado en la figura 5.3, en el cual existen capas con multitud de padres.

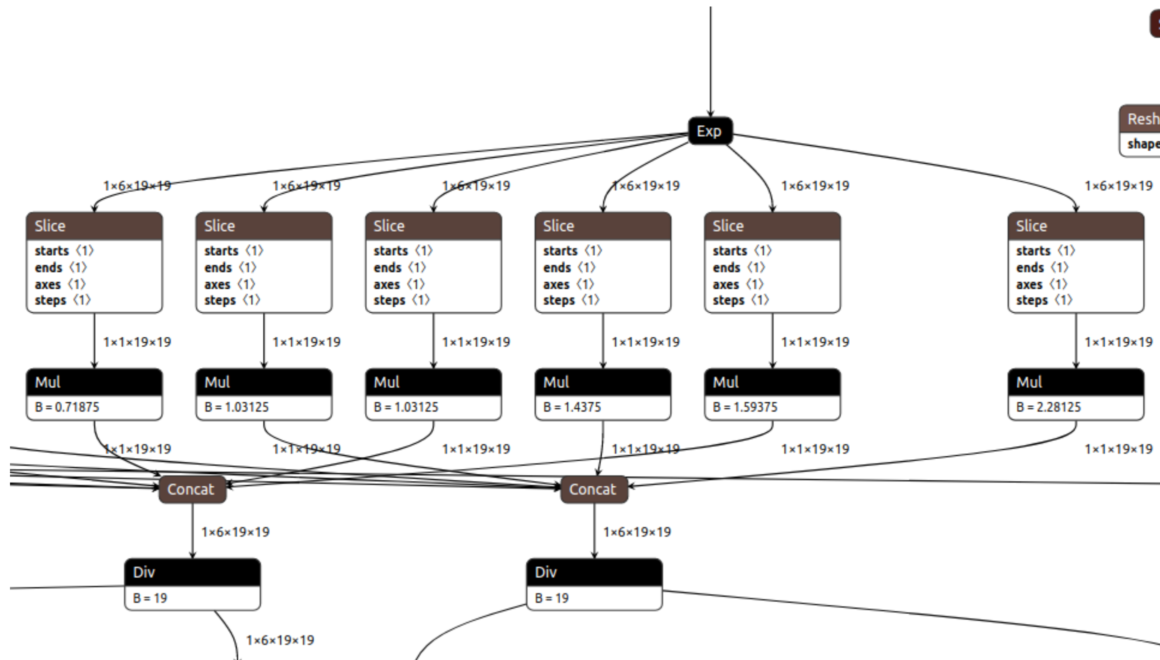

Figura 5.3: Generación de modelos de capas fusionadas en la FPGA.

Normalmente, la EDDL realiza la lectura e identificación de las capas de un modelo definido por un fichero ONNX. Para generar el modelo, la EDDL realiza un recorrido en profundidad (en inglés DFS o *Depth First Search*) de todo el modelo definido en ONNX para crear las capas y unirlas con sus respectivos padres. Por lo tanto, para todas las capas de un modelo con estructura de una dimensión el padre siempre será la capa anterior. Sin embargo, la existencia de modelos organizados en mallas de dos dimensiones complica la identificación de la capa padre, dado que la capa padre no será siempre la capa anterior.

Esta problemática es de gran relevancia para la función *model_for_fpga*, dado que el objetivo de esta esta función es crear sucesivas capas, las cuales han de estar asociadas a un padre. Dado que YOLOv4 tiene un modelo de dos dimensiones, la identificación del padre no es trivial y se ha de realizar mediante el modelo origen.

Además, también puede darse el caso en el cual una capa padre tenga dos hijos con diferentes tipos de datos. Esto puede dificultar la gestión de capas *Transform* en modelos de dos dimensiones, ya que la función ha de recordar qué capas *Transform* se han añadido al nuevo modelo.

Nuestra solución se basa en la estructura de datos representada en el código 5.2. Mediante esta estructura, cada vez que se crea una nueva capa para la FPGA se crea un número de entradas en la estructura *associated_layers* igual al número de capas fusionadas. En cada una de estas entradas, se asigna el campo *src* con la referencia a la capa origen del modelo original. Además, al mismo tiempo, dependiendo del formato de tipo de datos de la capa destino creada, también se asigna el campo *dst_ghwc* o *dst_chw* y su parámetro correspondiente *layer_id_ghwc* o *layer_id_chw*.

```
1 struct {
2   Layer *src;          // Capa del modelo origen
3   Layer *dst_ghwc;     // Capa del modelo destino cuyos datos
4                        //   se encuentran en formato GxHxWxCPI.
5   Layer *dst_chw;      // Capa del modelo destino cuyos
6                        //   datos se encuentran en formato CxHxW.
```

```
7    int    layer_id_ghwc;// Identificador de la capa destino con formato GxHxWxC
8    int    layer_id_chw; // Identificador de la capa destino con formato CxHxW
9  } associated_layers[MAX_ASSOCIATED_LAYERS];
```

Listing 5.2: Estructura de datos para gestión de capas.

Esto puede observarse en la figura 5.4, la cual muestra la gestión de capas mediante la estructura anteriormente descrita. Las capas cuyos datos se encuentran en formato GxHxWxCPI se encuentran representadas en color azul. Las capas con datos en formato CxHxW se encuentran representadas en color verde. Finalmente, las capas de transformación de datos en color marrón.

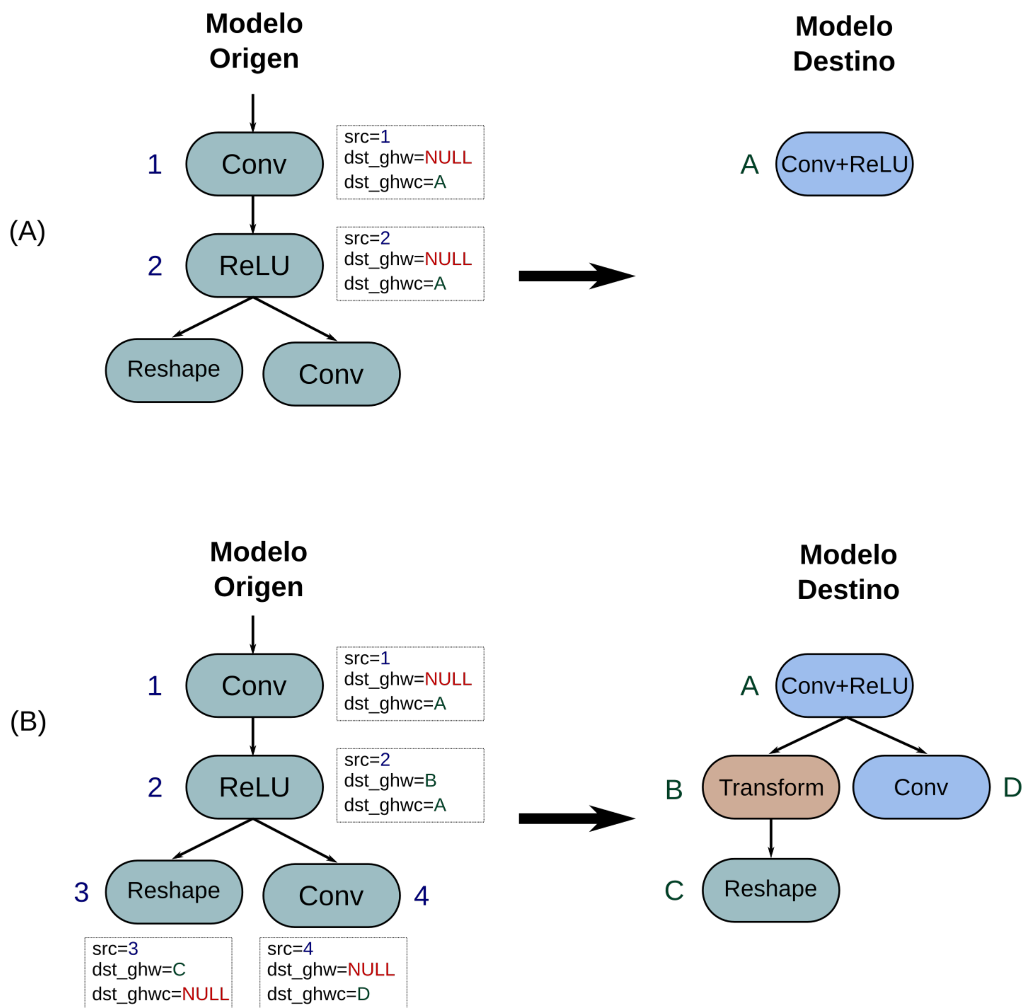

Figura 5.4: Funcionamiento de la estructura associated_layers.

En esta figura se plasma la creación de nuevas entradas a medida que se generan nuevas capas en el modelo destino. En primer lugar, se observa el paso A (arriba) donde, a partir de la capa Conv (capa 1) y ReLU (capa 2), se crea una nueva capa fusionada Conv+ReLU (A). Esto conlleva la creación de una entrada en la estructura de datos por cada capa fusionada (capas 1 y 2). En cada entrada se asigna, en primer lugar, la capa src a la capa origen correspondiente (capas 1 o 2). En segundo lugar, en ambas entradas, no se asigna ninguna capa destino con formato de datos GxHxW, dado que la capa destino creada tiene un formato GxHxWxCPI. Finalmente, también en ambas capas, se asigna como capa de destino con formato de datos GxHxWxCPI la nueva capa fusionada (capa A).

Si se sigue observando la figura anterior, se puede observar el paso B (abajo), donde se plasma el efecto de la capa *Transform* sobre la estructura nombrada. En primer lugar, la capa *Reshape* (la cual necesita un formatos de datos CxHxW) busca la entrada de su capa padre ReLU (capa 2) en la estructura associated_layers. A continuación, busca la capa asociada al parámetro dst_chw, ya que este contiene la referencia a la capa destino con el formato de datos que necesita la capa *Reshape*. Dado que en ese momento no existe (paso A), la función crea una capa *Transform* (capa B del modelo destino) y lo asocia a la entrada correspondiente, que en este caso es la entrada de la capa ReLU (capa 2). Seguidamente, se crea la capa *Reshape* y se estable como padre la capa *Transform*.

En siguiente lugar, se lee la capa 4 del modelo origen, la cual corresponde a una capa convolucional y, por lo tanto, puede ser ejecutada en la FPGA con formato GxHxWxCPI. Al buscar la capa padre en formato GxHxWxCPI dentro de la entrada de la estructura associated_layers se encuentra la capa fusionada creada anteriormente. Por esta razon, la capa D se asocia a la capa A sin problemas.

Por lo tanto, la estructura descrita anteriormente permite solucionar y simplificar los problemas descritos anteriormente. En primer lugar, mediante funciones auxiliares de búsqueda creadas sobre la estructura mostrada, es posible detectar las capas padres de cualquier capa de manera simple así como identificar el formato de las mismas.

5.5.3. Adaptación de los parámetros

El ultimo paso de la función *model_for_fpga*, cuyo esquema se plasma en la figura 5.2, consta del paso 12, en el cual se copian y adaptan los parámetros filtro y bias a las capas del nuevo modelo. Sin embargo, previamente a la copia de estos parámetros se necesita llevar a cabo un conjunto de adaptaciones.

En primer lugar, cabe mencionar que las capas con formato de datos GxHxWxCPI deben de tener un número de canales múltiples de CPI, lo cual puede no darse en el modelo origen. Esto conlleva la necesidad de dimensionar los canales en caso de que no ocurra este requisito, pudiendo provocar un tamaño de los parámetros bias y filtros diferente entre la capa original y la capa destino.

Por otra parte, al igual que los datos de las imágenes, los filtros deben de estar almacenados en el formato de datos correspondiente a cada capa según el dispositivo que vaya a ejecutarla. Cabe recordar, que una estructura típica de almacenamiento en memoria de los filtros viene definida por I x O x KH x KW, mientras que el acelerador HLSinf almacena los datos de los filtros en formato Go x Gi x CPO x CPI x KH x KW. Para estos problemas, se han diseñado un conjunto de funciones que realizan la transformación del formato del filtro y, además, dichas funciones realizan la copia de los parámetros asumiendo las discrepancia de las dimensiones.

5.5.4. Soporte para convoluciones con tamaño de filtro de uno por uno

En la sección sección 5.5.3 se han explicado las adaptaciones realizadas al acelerador HLSinf para el soporte de múltiples tamaños de *padding* y *stride*. No obstante, el tamaño de la ventana del filtro sigue siendo fijo y su dimensión viene definida por KH y KW, donde ambos parámetros adoptan el valor de 3. Como resultado, el acelerador HLSinf solo soporta tamaños de filtro de 3 x 3 inflexiblemente.

Sin embargo, en el modelo YOLOv4, encontramos varias convoluciones donde el tamaño del filtro está compuesto por únicamente un elemento (tamaño de 1x1), lo cual difiere del tamaño del filtro soportado por el acelerador. Afortunadamente, este tipo de convoluciones en el modelo YOLOv4 siempre viene definido con un *stride* de 1 x 1 con lo

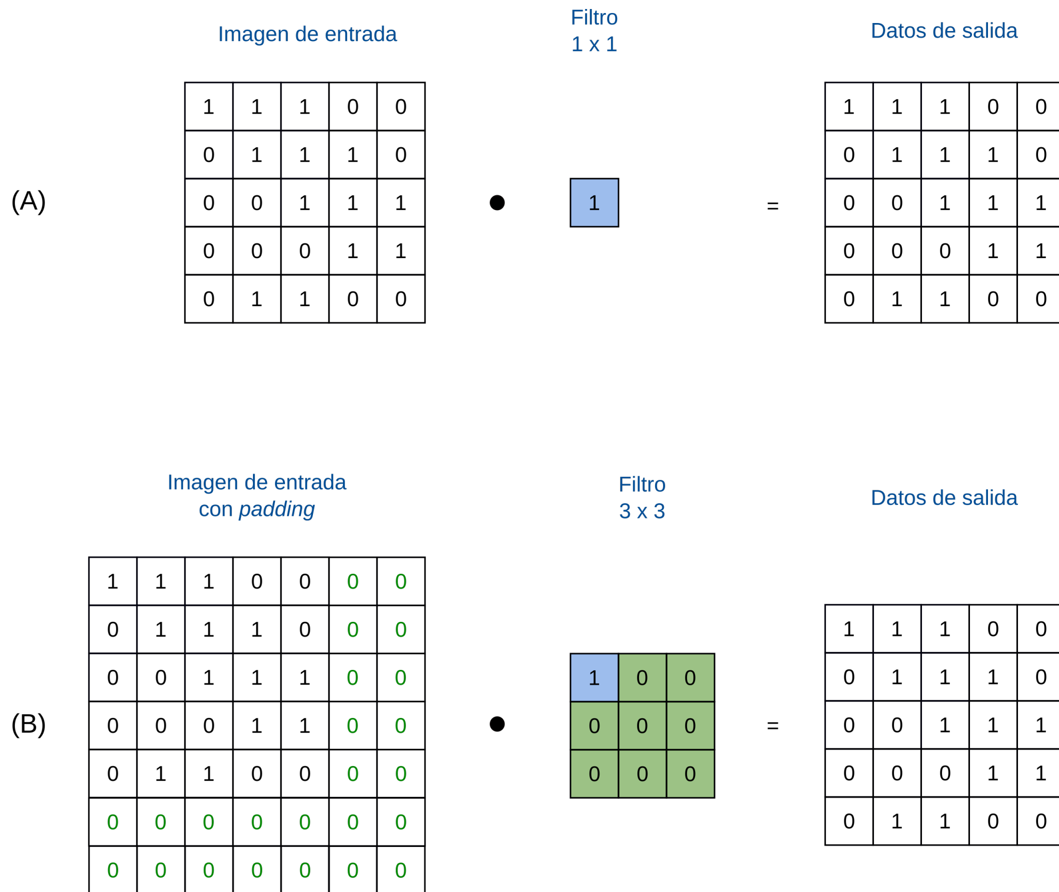

Figura 5.5: Soporte para convoluciones de filtro con tamaño de uno por uno.

cual, las convoluciones con *stride* de 1 x 1 y tamaños de filtro de 1 x 1 pueden ser tratadas como convoluciones con tamaño de filtro de 3 x 3 añadiendo un padding de 2 en cada extremo derecho e inferior y manteniendo el *stride* con valores de 1 x 1.

Esta transformación se plasma en la figura 5.5. En A (arriba) se observa el cálculo de una convolución con filtro de 1 x 1. En B (abajo), se observa la trasformación realizada donde, en al imagen de entrada, se han añadido dos columnas de *padding* a la derecha y dos filas de *padding* en la parte inferior. Además, en B, se ha aumentado la dimensionalidad del filtro rellenando con valores nulos los nuevos elementos. Como se observa, en ambos casos de obtiene el mismo resultado.

Por lo tanto, dentro de la EDDL, se ha creado una funcionalidad la cual consiste en identificar este convoluciones con un tamaño de filtro de 1 x 1 para así modificar los parámetros descritos con el fin de posibilitar su suporte en el acelerador HLSinf.

CAPÍTULO 6
Evaluación

En este capítulo se describen la etapa de análisis y evaluación realizadas sobre las implementaciones realizadas en este trabajo. Cabe mencionar que, para todos los resultados obtenidos sobre FPGA se ha utlizado la Alveo U200[57], la cual cuenta con 3 SLRs, 4 DDRs y 892.000 LUTs, . Por otra parte, para su comparación con la CPU, se ha utilizado el procesador Intel Core i7-7800X[58], el cual cuenta con 6 *cores* y 12 hilos (*threads*).

6.1 Recursos y prestaciones de los nuevos módulos en el acelerador HLSinf

En primer lugar, tanto para la evaluación de las prestaciones del acelerador HLSinf como para la evaluación de su integración con la librería EDDL, es necesario realizar una implementación del acelerador HLSinf en la FPGA.

Como se ha expuesto anteriormente, el acelerador tiene implementado un sistema que permite parametrizar el grado de paralelismo del acelerador mediante los valores de CPI y CPO, donde un mayor valor indicará un mayor grado de paralelismo y su valor mínimo es de 4. Sin embargo, estos parámetros, junto con el tipo de datos, también indican la cantidad de recursos que utilizará el acelerador dentro de la FPGA, pudiendo llegar a sobrepasar dichos recursos.

La relación entre paralelismo y recursos crea la necesidad de realizar un estudio de la escalabilidad de estos parámetros en relación al grado de paralelismo que ofrecen y a la cantidad de recursos que necesitan. Este estudio puede simplificarse gracias a una de las herramientas expuestas anteriormente: Vitis HLS. Esta herramienta, permite realizar un proceso de síntesis (proceso de conversión de diseño lógico FPGA de alto nivel en puertas lógicas), el cual muestra una aproximación fiable de la cantidad de recursos que necesitará el diseño implementado.

A continuación, se describirá el estudio de recursos realizado sobre varios diseños donde se variará el tipo de datos así como los valores de CPI y CPO. Además, también se verán los recursos necesarios para instanciar los nuevos módulos desarrollados.

6.1.1. Módulos STM y *Add*

En la figura 6.1 se puede observar la estimación de recursos para la implementación del acelerador HLSinf con tamaños de CPI y CPO de 4 cada uno respecto a la FPGA. En esta figura, se resalta el porcentaje de recursos totales requeridos para los nuevos módulos implementados y descritos en este proyecto.

Modules & Loops	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
k_conv2D	66764	2,220E5	-	66765	1	14	9	10	5
dataflow_parent_loop_proc	66723	2,220E5	-	66723	~0	14	8	9	5
dataflow_in_loop_o_iter_loop	66720	2,220E5	-	66606	~0	14	8	9	5
direct_conv	66649	2,220E5	-	66606	~0	10	4	4	2
stm	65691	2,190E5	-	65691	~0	3	2	2	0
pooling	65582	2,180E5	-	65544	0	~0	~0	~0	~0
read_kernel	222	739.000	-	222	0	0	~0	~0	0
relu	65581	2,180E5	-	65581	0	~0	~0	~0	0
add_data	65546	2,180E5	-	65546	0	~0	~0	~0	0
read_data_channels_gihwcpi	65613	2,180E5	-	65613	0	0	~0	~0	0
input_buffer	65543	2,180E5	-	65543	0	0	~0	~0	2
read_data_channels_gihwcpi_1	65612	2,180E5	-	65612	0	0	~0	~0	0
write_data_channels_gihwcpi	65610	2,180E5	-	65610	0	0	~0	~0	0
read_bias	73	243.000	-	73	0	0	~0	~0	0
dataflow_in_loop_o_iter_loop_entry376	0	0.0	-	0	0	0	~0	~0	0
o_iter_loop	66722	2,220E5	66722	-	-	-	-	-	-

Figura 6.1: Estimación de recursos necesarios globales para la implementación de HLSinf con tamaño CPI/CPO 4/4.

En primer lugar, resaltado con un cuadro rojo, se puede observar los recursos necesarios para la implementación del módulo STM. De estos recursos, se puede observar que el porcentaje de DSP necesarios es mayor en comparación a otros módulos . Esto es debido a que este módulo implementa varias funciones no triviales como funciones logarítmicas o exponenciales.

Por otra parte, representado con un cuadro verde, se puede observar los recursos demandados para la implementación del módulo *Add*. Dado que su funcionalidad es bastante simple (únicamente se realiza la suma elemento a elemento de dos estructuras de datos) no se exige una gran cantidad de recursos.

6.1.2. Estimación de recurso.

En segundo lugar, se ha realizado un estudio del efecto de los parámetros CPI, CPO y del tipo de datos utilizado para la estimación de recursos. Este estudio se encuentra representado en su totalidad en la figura 6.2 donde se comparan la estimación de recursos sobre las siguientes implementaciones:

- 4x4 FP32: CPI/CPO de 4 respectivamente y tipo de datos de coma flotante.
- 4x4 API8: CPI/CPO de 4 respectivamente y tipo de datos tipo de datos fijo.
- 8x8 FP32: CPI/CPO de 8 respectivamente y tipo de datos de coma flotante.
- 4x4 API8: CPI/CPO de 8 respectivamente y tipo de datos tipo de datos fijo.
- 16x16 API16: CPI/CPO de 16 respectivamente y tipo de datos tipo de datos fijo.

En este estudio destacamos, en primer lugar, la diferencia clara de recursos necesarios para la implementación de cualquier acelerador con tipo de datos de coma flotante y tipo de datos fijo. Por otra parte, podemos ver que, si se requiere instanciar un acelerador de coma flotante, solo será posible instanciar el acelerador con el mínimo tamaño de CPI/CPO soportado ya que, como se observa la figura 6.2, en C se puede ver que, para tamaños mayores de CPI/CPO igual a 4 no existen suficientes recursos. Por ejemplo, se puede observar el caso de la implementación con CPI/CPO igual a 8, donde se necesitan un 150 % de los DSPs existentes en un SLR de la FPGA.

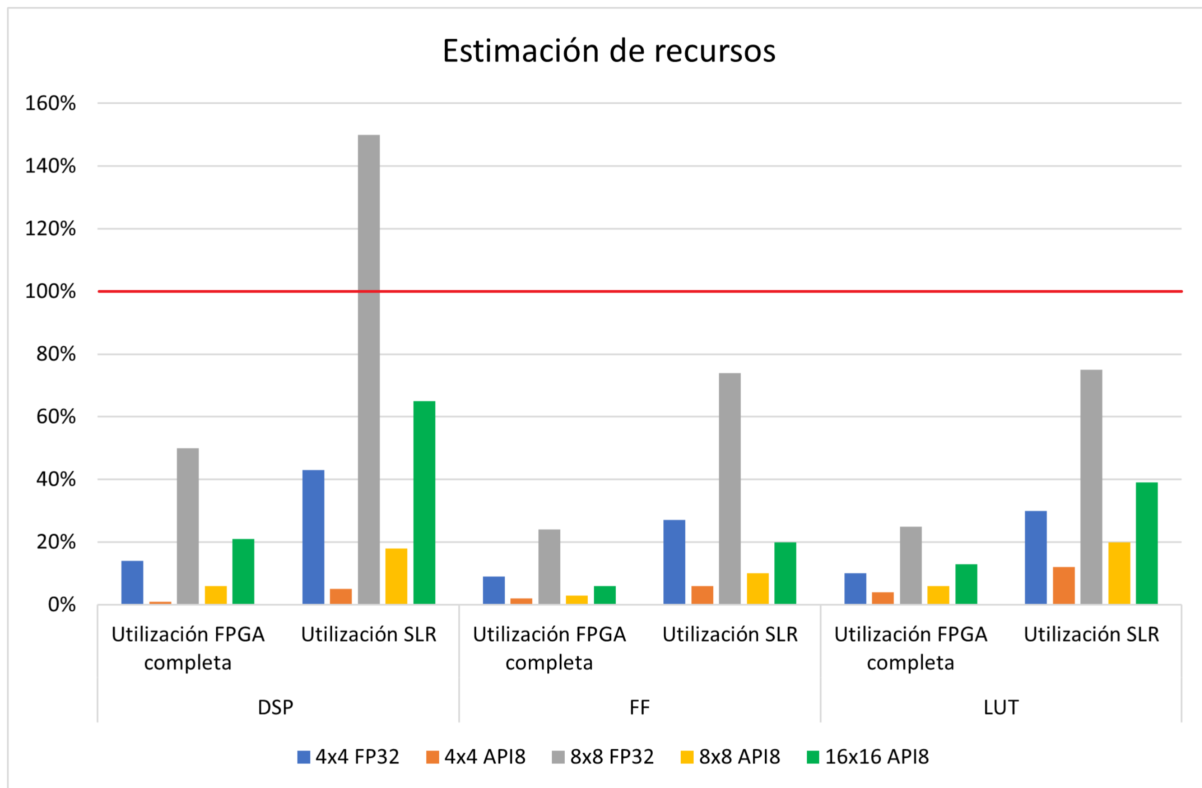

Figura 6.2: Estimación de recursos.

Sin embargo, y como se ha mencionado antes, las FPGAs en inferencia son utilizadas con datos de coma fija. Esto significa que la solución propuesta es inestancable, dado que ocupa relativamente poco.

6.2 Prestaciones de la solución final

Se ha desarrollado un estudio de prestaciones de la integración realizada entre la librería EDDL y el acelerador HLSinf. Para esto, se ha realizado la inferencia de imágenes utilizando para ello varios modelos de Redes Neuronales. Concretamente re ha utilizado una versión modificada del modelo VGG16 y, además, el modelo YOLOv4 descrito en este proyecto. De esta manera, y gracias a la EDDL, se cubre todas las funciones necesarias para la creación, gestión y ejecución del modelo. Por otra parte, las capas soportadas por el acelerador HLSinf se ejecutaran sobre la FPGA. A continuación analizaremos los resultados obtenidos.

6.2.1. VGG16

En primer lugar, se ha realizado un estudio de la ejecución con datos de tipo coma flotante de una versión modificada del modelo VGG16 cuya topología puede observarse detalladamente en el apéndice A. Este modelo ha sido elegido dada su simplicidad y la cantidad de capas fusionadas que ofrece, dado que su topología está formada por 39 capas, donde se encuentran 13 capas convolucionales, 15 capas ReLU y 5 capas *Max-Pooling*, entre otras. Además, casi todas estas capas pueden fusionarse, dando como resultado un modelo formado por un total de 23 capas, el cual se encuentra representado en la figura 6.3. En esta figura también se puede observar la creación de varias capas *Transform*, las cuales se posicionan entre la capa input (la cual tiene un formato de datos de IxHxW)

y la primera capa fusionada (con formato GxHxWxCPI) así como entre la última capa fusionada y la capa Average-pooling, la cual se ejecuta en la CPU (con formato IxHxW).

```
input1                   | (3, 224, 224)      =>  (3, 224, 224)
Transform                | (3, 224, 224)      =>  (4, 224, 224)
Conv_ReLU(HLSinf)        | (4, 224, 224)      =>  (16, 224, 224)
Conv_ReLU_Maxp(HLSinf)   | (16, 224, 224)     =>  (24, 112, 112)
Conv_ReLU(HLSinf)        | (24, 112, 112)     =>  (84, 112, 112)
Conv_ReLU_Maxp(HLSinf)   | (84, 112, 112)     =>  (40, 56, 56)
Conv_ReLU(HLSinf)        | (40, 56, 56)       =>  (188, 56, 56)
Conv_ReLU(HLSinf)        | (188, 56, 56)      =>  (176, 56, 56)
Conv_ReLU_Maxp(HLSinf)   | (176, 56, 56)      =>  (148, 28, 28)
Conv_ReLU(HLSinf)        | (148, 28, 28)      =>  (296, 28, 28)
Conv_ReLU(HLSinf)        | (296, 28, 28)      =>  (288, 28, 28)
Conv_ReLU_Maxp(HLSinf)   | (288, 28, 28)      =>  (224, 14, 14)
Conv_ReLU(HLSinf)        | (224, 14, 14)      =>  (104, 14, 14)
Conv_ReLU(HLSinf)        | (104, 14, 14)      =>  (308, 14, 14)
Conv_ReLU_Maxp(HLSinf)   | (308, 14, 14)      =>  (284, 7, 7)
Transform                | (284, 7, 7)        =>  (284, 7, 7)
Avgpool2                 | (284, 7, 7)        =>  (284, 7, 7)
Reshape1                 | (284, 7, 7)        =>  (13916)
Dense1                   | (13916)            =>  (83)
ReLU                     | (83)               =>  (83)
dense2                   | (83)               =>  (20)
relu2                    | (20)               =>  (20)
dense3                   | (20)               =>  (8)
```

Figura 6.3: Modelo VGG16 con capas fusionadas.

En la figura 6.4 se representa gráficamente el tiempo (en milisegundos) de la inferencia de una imagen en varios dispositivos. Cabe mencionar que se ha utilizado la misma imagen en todos los dispositivos. Como se ha mencionado anteriormente, se ha realizado la inferencia sobre un dispositivo CPU Intel Core i7-7800X y un dispositivo FPGA Alveo U200.

Para la ejecución sobre la FPGA se ha implementado un acelerador de coma flotante con valor de CPI y CPO igual a 4. Además, se han generado dos implementaciones donde, en el primer caso, únicamente se utilizado una instancia del acelerador, dando como resultado un único *kernel*. En segundo lugar, para observar su escalabilidad, se han utilizado dos instancias del acelerador, dando como resultado dos *kernels*.

Si seguimos observando la figura, podemos ver como la CPU alcanza el peor tiempo de ejecución, posicionándose en más de 5.000 milisegundos. Por debajo, nos encontramos la FPGA con un *kernel*, cuya ejecución tiene una latencia de más de 1.500 segundos. Esto, en comparación a la CPU de Intel, supone una aceleración (*speed-up*) de más de un factor de 3. Finalmente, podemos observar el resultado de la ejecución de la FPGA con 2 *kernels*, la cual tarda un poco más de 1.000 milisegundos. Como resultado, se obtiene una aceleración respecto a la CPU de un factor de 5.

Cabe remarcar que, aunque teóricamente se debería observar una mejora de un factor de dos entre la versión de 1 kernel de la FPGA y la versión *multikernel*, esto no se observa en su totalidad ya que la ejecución de las capas fusionadas suponen un 90 % de la ejecución total de la inferencia.

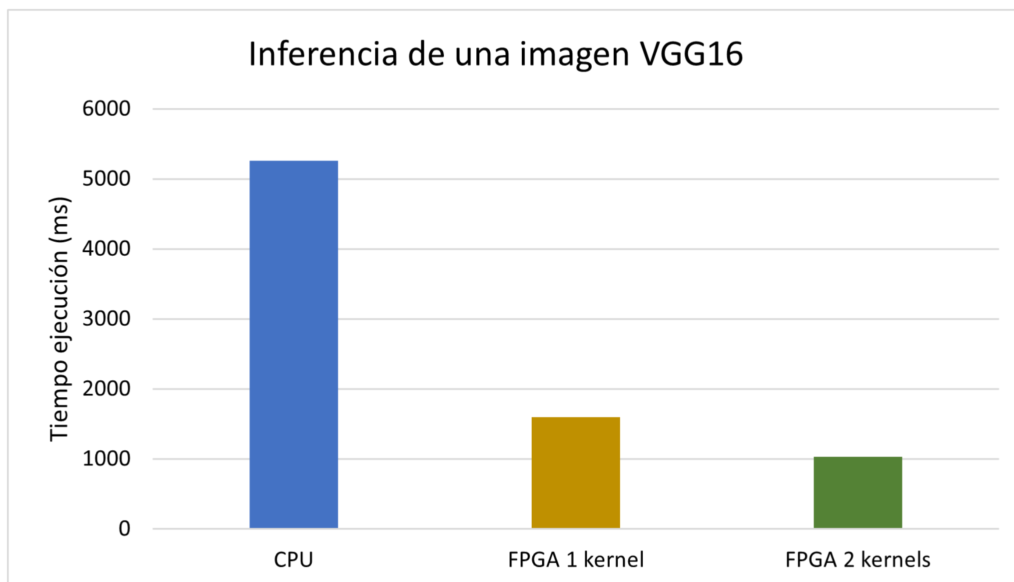

Figura 6.4: Ejecución del modelo VGG16 en varios dispositivos.

6.3 YOLOv4

En segundo lugar, se ha realizado un estudio de la ejecución del modelo YOLOv4 con datos de tipo coma flotante. Este modelo, originalmente, está formado por 642 capas, de las cuales 110 son convoluciones. De estas 110 convoluciones, 72 vienen seguidas por el patrón de capas STM (softplus, tangente hiperbólica y multiplicación).

Se ha utilizado la función *model_for_fpga* descrita anteriormente para transformar el modelo original construido sobre el dispositivo CPU a un nuevo modelo adaptado para FPGAs cuyo número de capas a descendido hasta 504. Entre estas capas, se destaca la generación de 72 capas *Transform* y 101 capas HLSinf. Además, cabe recordar que este nuevo modelo se ha generado fusionando todas las capas posibles.

Actualmente, el acelerador HLSinf solo soporta convoluciones donde las dimensiones H y W de datos de entrada son menores a 256. En el modelo YOLOv4, existen 9 convoluciones con una dimensión de H y W mayor a la soportada, lo cual provoca la necesidad de ser ejecutadas sobre la CPU. Como trabajo futuro nos planteamos el soporte de estas convoluciones.

En la figura 6.5 se puede observar el tiempo total (en milisegundos) de la inferencia de una imagen así como la suma del tiempo total de la ejecución de las capas convolucionales, STM (*softplus*, tangente hiperbólica, multiplicación) y *Add* de los diferentes dispositivos. Además, al igual que en el caso anterior, para el estudio de la ejecución sobre la FPGA se ha realizado el estudio sobre la versión de un *kernel* y la versión *multikernel*.

Cabe mencionar que, dado que existen 9 capas que no se soportan en la FPGA y, por lo tanto, no se reflejan en el tiempo de ejecución de las capas fusionadas, se han eliminado 9 capas convolucionales del cálculo de la ejecución total de las capas en el caso de CPU, para conseguir así reflejar de manera justa la misma carga computacional en todos los dispositivos.

Si seguimos observando la figura 6.5 se puede observar, en primer lugar, el tiempo de ejecución de las capas convolucionales, STM y *Add* representado con barras azules. Para el dispositivo CPU se observa un tiempo de total de 5.832 milisegundos. Seguidamente, se observa el tiempo total de la ejecución de las mismas capas en el caso del dispositivo FPGA con un *kernel*, el cual es de 4.875 milisegundo. Esto representa una aceleración

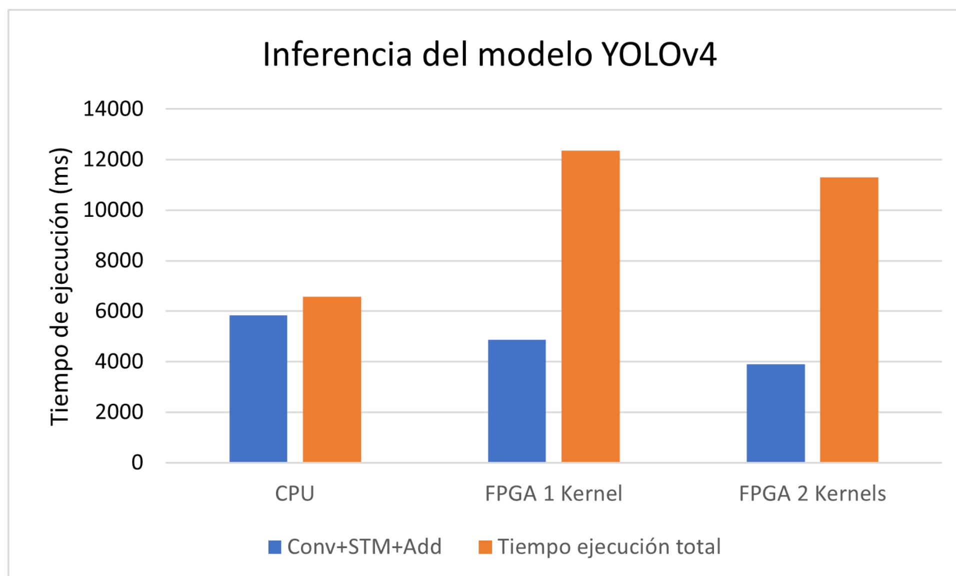

Figura 6.5: Ejecución del modelo YOLOv4 en varios dispositivos.

de un factor de 1,2 respecto a la CPU. En último lugar, se puede ver que el tiempo de ejecución de la FPGA con versión *multikernel* es de 3.898 milisegundos y una aceleración de un factor de 1,5.

En segundo lugar, en la misma figura, se observa el tiempo total de inferencia del modelo YOLOv4, representado con barras naranjas. En este caso, se puede observar como el tiempo en ambas versiones de la FPGA supera en gran medida el tiempo de ejecución de la CPU. Además, se puede observar como, en el caso de la CPU, la ejecución de las operaciones más costosas computacionalmente suponen más del 80 % del tiempo total de la ejecución. Sin embargo, en el caso de la FPGA, este porcentaje es mucho menor, pues en el caso de la ejecución sobre un *kernel*,la ejecución de estas capas solo suponen un 38 % de la ejecución total. Además, para el caso de la ejecución sobre la versión *multikernel*, este porcentaje baja hasta el 35 %.

Estos resultados son debidos a varias razones. En primer lugar, en la gráfica 6.6 se muestra el tiempo de media (en microsdegundos) por ejecución de una capa sin soporte para FPGA y, por lo tanto, ejecutada en todos los casos sobre CPU. Esta gráfica ilustra el problema que se encuentra para la ejecución de capas CPU en modelos adaptados para FPGA: estas presentan un tiempo de ejecución por llamada muy elevado en comparación a la ejecución con un modelo creado para CPU. Esto es debido a que, originalmente, para la ejecución de un modelo en FPGA, se diseñó un sistema en el cual, si una capa no estaba soportada en la FPGA, se realizaba una copia de datos entre la FPGA y la CPU de los datos de entrada para, posteriormente, ejecutar la función en la CPU y, a continuación, volver a copiar el resultado en la FPGA. Esto, ralentiza todo el proceso de inferencia ya que, si existe un conjunto seguido de capas a ejecutar en la CPU, en todas ellas se realizará la copia de datos, aunque no sea necesario.

En segundo lugar, otro factor que ralentiza la ejecución en la FPGA, es la generación de múltiples capas *Transform*. Aunque la ejecución de las mismas únicamente suponga un 4 % del tiempo total de ejecución, llama la atención la gran cantidad de capas que se han generado para la gestión de la diversidad de formatos. Esto, podría solucionarse mediante el desarrollo de nuevos módulos en el acelerador HLSinf, para reducir el número de capas a realizar en la CPU.

Finalmente, aunque el tiempo de ejecución de las capas fusionadas en la FPGA es menor al tiempo de ejecución de las mismas capas sin fusionar en la CPU, no se observa

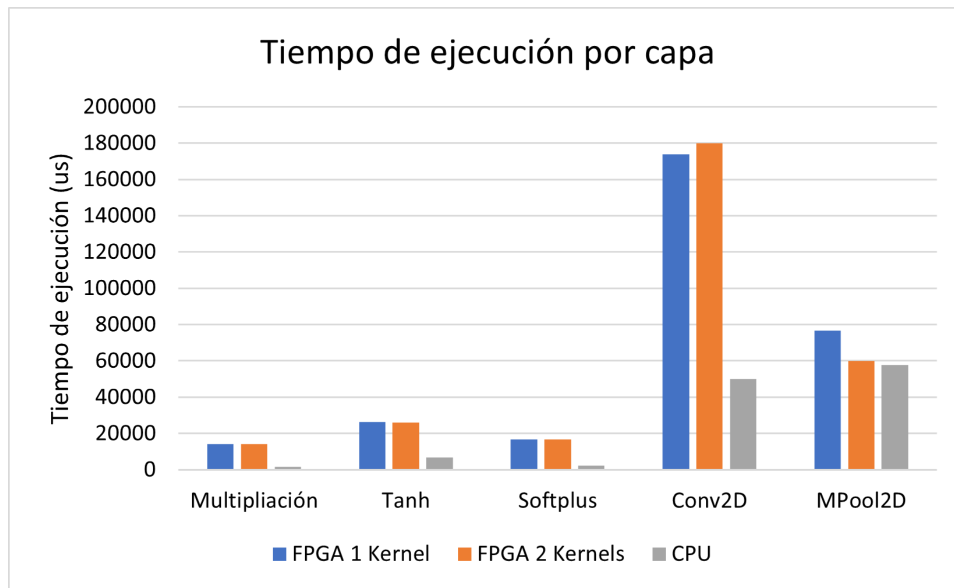

Figura 6.6: Tiempo de ejecución de varias capas en YOLOv4.

la misma aceleración que en el caso del modelo VGG16. Esto plantea la necesidad de realizar un análisis más intensivo implementando, por nuevos aceleradores con un CPI más grande.

Finalmente, cabe remarcar que para la ejecución de ambos modelos se ha utilizado datos de tipo coma flotante, lo cual ralentiza el proceso de inferencia y, como se ha visto en la sección 6.1, limita la dimensión de CPI/CPO. Sin embargo, y como se ha mencionado anteriormente, para los procesos de inferencia este tipo de datos no son necesarios. Por lo tanto, cambiar el tipo de datos podría traer grandes mejoras para la FPGA, dado que se podrían implementar aceleradores con una dimensión de CPI/CPO de 16, provocando una mejora considerable de las prestaciones del acelerador.

Todos estos aspectos y limitaciones están siendo desarrollados en la actualidad y son parte del trabajo futuro de este trabajo.

CAPÍTULO 7

Conclusiones

En este trabajo, se ha fijado como objetivo la identificación de características del modelo YOLOv4 así como la identificación de carencias del acelerador HLSinf para la ejecución de este modelo. Además, otro objetivo propuesto ha sido el desarrollo de nuevos módulos dentro del acelerador mediante HLS.

En este proyecto, se ha realizado un análisis de la arquitectura del modelo YOLOv4 donde, durante toda su topología, pueden observar patrones de ciertas capas no soportadas previamente en HLSinf. Posteriormente, se han desarrollado varios módulos en HLS dentro del acelerador HLSinf orientados a la ejecución de estos patrones de capas en FPGAs. Primeramente, se ha desarrollado el módulo STM, el cual permite ejecutar las capas Softplus, tangente hiperbólica y multiplicación elemento a elemento. En segundo lugar, se ha desarrollado el módulo Add y, en conjunto, otro módulo para la lectura en formato de datos G x H x W x CPI de la estructura de datos de entrada para este módulo. El módulo Add permite ejecutar la capa *Add*, la cual se basa en la suma elemento a elemento de dos capas.

Estos módulos se han implementado siguiendo la estructura de la arquitectura del acelerador preexistente. Esto significa que el desarrollo de los módulos se ha realizado siguiendo el modelo de flujo de datos (*dataflow*) y el modelo de gestión de datos mediante *streams* en la que se basa el acelerador.

Otro de los objetivos fijados para este proyecto, ha sido la integración del acelerador HLSinf en la librería EDDL. Esto se ha conseguido de manera transparente para el usuario. Para ello, dentro de la librería EDDL, se ha desarrollado la capa *Transform*, la cual realiza una transformación de los datos a formato CxHxW o GxHxWxCPI, cuando proceda.

Para esta integración, también se ha generado una nueva clase llamada HLSinf dentro de la libreria EDDL. Esta nueva clase tiene como objetivo crear y gestionar las capas fusionadas y adaptar los parámetros a las necesidades del acelerador HLSinf.

Otra funcionalidad añadida a la librería EDDL es el soporte *multikernel*. Mediante este soporte, esta librería es capaz de ejecutar las capas sobre más de un *kernel*, pudiendo mejorar de manera significativa las prestaciones de la ejecución.

Además, en esta librería, se ha generado una función para la adaptación de modelos de Redes Neuronales para dispositivos FPGA. Para esto, realiza la lectura y análisis de un modelo generado para CPU y genera un nuevo modelo con soporte para FPGAs. Además, esta función realiza un proceso de lectura para, posteriormente, generar capas fusionadas si se diera el caso. Además, también genera nuevas capas *Transform* en caso que sea necesario. Por último, esta función se encarga de transformar todos los filtros y bias al formato de datos correspondiente para el acelerador HLSinf.

Mediante todo esto, se ha conseguido dar un soporte completo y funcional de un dispositivo totalmente nuevo (HLSinf) dentro de la librería EDDL, la cual tiene una arquitectura interna compleja dada la cantidad de funcionalidades que posee. Asimismo, se ha conseguido dar soporte al modelo YOLOv4, logrando ejecutar la inferencia de una imagen sobre este modelo en FPGAs, gracias a la librería EDDL y el acelerador HLSinf.

Todo esto ha sido puesto en prueba mediante varias ejecuciones. En primer lugar, se ha realizado un estudio del modelo VGG16 sobre la solución propuesta. Para ello, se ha realizado la inferencia de una imagen sobre una FPGA Alveo U200 y sobre una CPU Intel Core i7-7800X, donde la versión *multikernel* presenta una aceleración de un factor de 5.

También se ha podido realizar la inferencia de una imagen en el modelo YOLOv4 satisfactoriamente. Desafortunadamente, y aunque la suma de la ejecución de las capas convolucionales ofrecen buenos resultados en la FPGA, los resultados han mostrado un tiempo de ejecución total más elevado en la FPGA. Como se ha explicado, esto es debido a la gestión de memoria entre dispositivos existentes actualmente en la librería EDDL. Esto, y junto a otras posibles mejoras, se desarrollan con más detalle en el capítulo final.

Finalmente, en este trabajo se ha conseguido desarrollar una solución para la ejecución eficiente de Redes Neuronales dentro del proyecto SELENE. Además, también se ha aumentando la funcionalidad del acelerador HLSinf y, por consiguiente, del proyecto DeepHealth.

CAPÍTULO 8
Trabajo futuro

Aunque se ha realizado un gran esfuerzo en este proyecto, consiguiendo buenos resultados para algunos casos de uso, existen algunas futuras mejoras posibles para tanto el acelerador HLSinf como para el soporte de FPGAs dentro de la librería EDDL. Estas mejoras, expuestas brevemente en el capitulo 6 serán explicadas a continuación con mayor detalle.

8.1 Mejora de la gestión de memoria entre la FPGA y la CPU

Como se ha observado en el capítulo de resultados, la gran cantidad de transferencias de memoria entre los dispositivos CPU y FPGA implica un mayor tiempo de ejecución de las capas sin soporte en la FPGA dentro de los modelos adaptados para FPGA.

Para minimizar el número de transferencias y conseguir una gestión de memoria eficiente utilizar y adaptar la capa *Transform*. Recordemos que esta capa marca el punto de unión entre dos capas de distinto formato, dado que todas las capas que se ejecutan en la CPU necesitan un formato de datos diferente al formato de datos requerido por la FPGA. Por lo tanto, se puede modificar la librería EDDL para que únicamente se realicen las transferencias en estas capas, disminuyendo así el tiempo total de ejecución.

Sin embargo, también se podría intentar soportar todas las capas del modelo dentro del acelerador, lo cual causaría una disminución significativa de el número transferencias entre la CPU y FPGA y de la generación de capas *Transform*.

8.2 Mejora del acelerador HLSinf mediante técnicas de cuantificación

Como hemos mencionado anteriormente, los modelos de han ejecutado con un tipo de datos de coma flotante. Sin embargo, se puede reducir la carga computacional mediante la conversión del tipo de datos de estos modelos de coma flotante a coma fija.

Para esto, se puede utilizar el método método de cuantificación de datos de precisión dinámica (*dynamic-precision data quantization method*), el cual propone cambiar el tipo de datos de las capas minimizando el error que pueda generar esta conversión[59].

Además, existen múltiples métodos orientados a la compresión de Redes Neuronales. Uno de estos es la descomposición por valores singulares (del inglés *singular value descomposition* o SVD)[60], el cual es un método que sirve para la compresión de datos que puede utilizarse para la reducción del número de parámetros de las capas[59].

Por otra parte, Xilinx ha puesto a disposición varios modelos de Redes Neuronales ya cuantificados los cuales podrían ser utilizados en lugar de los modelos de coma flotante utilizados en este proyecto.

8.3 Mejora del acelerador HLSinf mediante técnicas de dispersión

Los modelos de Redes Neuronales pueden presentar un alto grado de dispersión debido a varios factores. Por ejemplo, el uso de la función de activación ReLU provoca que los valores negativos se transformen a valores cero, pudiendo generando una dispersión de más del 40 % en modelos de Redes Neuronales Convolucionales[61].

Por lo tanto, para la aceleración de las Redes Neuronales se pueden implementar técnicas de dispersión (*sparcity*). El objetivo de estas técnicas es identificar un conjunto valores idénticos o cero, y comprimir los datos de manera que se evite el cálculo sobre estos elementos. De esta manera, se pretende reducir considerablemente la carga computacional, así como la comunicación y el almacenamiento de datos[61].

Por lo tanto, en un futuro se puede realizar un estudio de las técnicas de dispersión para, posteriormente, adaptarlas al acelerador HLSinf.

8.4 Evaluación exhaustiva de consumo y prestaciones

Si bien es cierto se ha realizado un estudio de prestaciones de la solución desarrollada, se puede realizar una evaluación más exhaustiva de las prestaciones de la FPGA con otros dispositivos orientados a la computación heterogénea como las GPUs. Donde, además, seria interesante que esta evaluación reflejara el consumo de los dispositivos a comparar.

Bibliografía

[1] Konstantina Kourou y col. “Machine learning applications in cancer prognosis and prediction”. En: *Computational and Structural Biotechnology Journal* 13 (2015), págs. 8-17. ISSN: 2001-0370. DOI: https://doi.org/10.1016/j.csbj.2014.11.005. URL: https://www.sciencedirect.com/science/article/pii/S2001037014000464.

[2] David Gil y Devadoss Johnson Manuel. “Diagnosing Parkinson by using artificial neural networks and support vector machines”. En: *Global Journal of Computer Science and Technology* 9.4 (2009).

[3] Xilinx INC. *Introduction to FPGA Design with Vivado High-Level Synthesis*. URL: https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf. (acceso: 28.06.2020).

[4] SELENE. *Self-monitored Dependable platform for High-Performance Safety-Critical Systems*. URL: https://www.selene-project.eu/. (acceso: 5.07.2020).

[5] Carles Hernàndez y col. “SELENE: Self-Monitored Dependable Platform for High-Performance Safety-Critical Systems”. En: (2020), págs. 370-377. DOI: 10.1109/DSD51259.2020.00066.

[6] EDDL GitHub. *European Distributed Deep Learning*. URL: https://github.com/deephealthproject/eddl. (acceso: 5.07.2020).

[7] Deep Health. *Deep-Learning and HPC to Boost Biomedical Applications for Health*. URL: https://deephealth-project.eu/. (acceso: 5.07.2020).

[8] Ph.D. By Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Pub; 1st edición, 1997. ISBN: 0966017633.

[9] Jun Suzuki, Hideki Isozaki y Eisaku Maeda. “Convolution Kernels with Feature Selection for Natural Language Processing Tasks.” En: (ene. de 2004), págs. 119-126. DOI: 10.3115/1218955.1218971.

[10] Rikiya Yamashita y col. “Convolutional neural networks: an overview and application in radiology”. En: *Insights into Imaging* 9 (jun. de 2018). DOI: 10.1007/s13244-018-0639-9.

[11] Vincent Dumoulin y Francesco Visin. “A guide to convolution arithmetic for deep learning”. En: (mar. de 2016).

[12] Pravendra Singh, Prem Raj y Vinay Namboodiri. “EDS pooling layer”. En: *Image and Vision Computing* 98 (abr. de 2020), pág. 103923. DOI: 10.1016/j.imavis.2020.103923.

[13] Siddharth Sharma, Simone Sharma y Anidhya Athaiya. “ACTIVATION FUNCTIONS IN NEURAL NETWORKS”. En: *International Journal of Engineering Applied Sciences and Technology* 04 (mayo de 2020), págs. 310-316. DOI: 10.33564/IJEAST.2020.v04i12.054.

[14] Vinod Nair, Geoffrey E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. En: *Proceedings of the 27th International Conference on International Conference on Machine Learnin* (2010), págs. 807-814.

[15] Andrew Senior y Xin Lei. “Fine context, low-rank, softplus deep neural networks for mobile speech recognition”. En: (mayo de 2014), págs. 7644-7648. DOI: 10.1109/ICASSP.2014.6855087.

[16] Ali Hamidoğlu. “On general form of the Tanh method and its application to nonlinear partial differential equations”. En: *Numerical Algebra, Control and Optimization* 6 (jun. de 2016), págs. 175-181. DOI: 10.3934/naco.2016007.

[17] Guifang Lin y Wei Shen. “Research on convolutional neural network based on improved Relu piecewise activation function”. En: *Procedia Computer Science* 131 (ene. de 2018), págs. 977-984. DOI: 10.1016/j.procs.2018.04.239.

[18] Xiaohu Zhang, Yuexian Zou y Wei Shi. “Dilated convolution neural network with LeakyReLU for environmental sound classification”. En: (2017), págs. 1-5. DOI: 10.1109/ICDSP.2017.8096153.

[19] Roberto Millon, Fernando Frati y Enzo Rucci. “A Comparative Study between HLS and HDL on SoC for Image Processing Applications”. En: (dic. de 2020).

[20] Siemens Digital Industries Software. *HLS LIBS*. URL: https://hlslibs.org/. (acceso: 30.06.2020).

[21] Xilinx Inc. *HLS IP Libraries*. URL: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/hls_ip_libraries.html. (acceso: 30.06.2020).

[22] Xilinx Inc. *Xilinx*. URL: https://www.xilinx.com/. (acceso: 30.06.2020).

[23] Xilinx Inc. *Vitis HLS Math Library*. URL: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_math_library.html#vfp1582649168418. (acceso: 30.06.2020).

[24] Frank Vahid. *Digital Design with RTL Design, Verilog and VHDL (2nd ed.)* John Wiley y Sons, 2010, pág. 247. ISBN: 978-0-470-53108-2.

[25] Philippe Coussy y col. “An Introduction to High-Level Synthesis”. En: *IEEE Design Test of Computers* 26.4 (2009), págs. 8-17. DOI: 10.1109/MDT.2009.69.

[26] Xilinx Inc. *Vitis High-Level Synthesis User Guide*. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf. (acceso: 30.06.2020).

[27] Intel Corporation. *Intel® High Level Synthesis Compiler*. URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html. (acceso: 30.06.2020).

[28] Xilinx Inc. *Vitis 2020.2 - HLS*. URL: https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0090-vitis-hls-hub.html. (acceso: 30.06.2020).

[29] Xilinx Inc. *HLS Pragmas*. URL: https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html#uyd1504034366571. (acceso: 30.06.2020).

[30] Xilinx Inc. *BRAM and Other Memories*. URL: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/jbt1504034294480.html. (acceso: 30.06.2020).

[31] Xilinx Inc. *Vitis HLS Migration Guide*. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1391-vitis-hls-migration-guide.pdf. (acceso: 30.06.2020).

[32] The Khronos Group Inc. *OpenCL*. URL: https://www.khronos.org/opencl/. (acceso: 30.06.2020).

[33] John E. Stone, David Gohara y Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". En: *Computing in Science Engineering* 12.3 (2010), págs. 66-73. DOI: 10.1109/MCSE.2010.69.

[34] HLSinf GitHub. *High-Level Synthesis inference accelerator for FPGAs*. URL: https://github.com/PEAK-UPV/HLSinf/tree/main. (acceso: 5.07.2020).

[35] Alexey Bochkovskiy, Chien-Yao Wang y Hong-Yuan Mark Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection". En: (2020). arXiv: 2004.10934 [cs.CV].

[36] EDDL. *Welcome to EDDL's documentation!* URL: https://deephealthproject.github.io/eddl/. (acceso: 26.07.2020).

[37] Eduardo Quinones Jose Flich Carles Hernandez y Roberto Paredes. "Distributed Training on a Highly Heterogeneous HPC System". En: (2020), págs. 359-370.

[38] The Linux Foundation. *Open Neural Network Exchange*. URL: https://onnx.ai. (acceso: 26.07.2020).

[39] Michele Cancilla y col. "The DeepHealth Toolkit: A Unified Framework to Boost Biomedical Applications". En: (2021), págs. 9881-9888. DOI: 10.1109/ICPR48806.2021.9411954.

[40] Xilinx. *FINN*. URL: https://finn.readthedocs.io/en/latest/. (acceso: 26.07.2020).

[41] Xilinx. *Brevitas*. URL: https://github.com/Xilinx/brevitas. (acceso: 31.08.2020).

[42] TOBIAS ALONSO y col. "Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning". En: (2021). URL: https://inaccel.com/wp-content/uploads/ACM_TRETS_DC_2020.pdf.

[43] Xilinx. *DPU Configuration*. URL: https://www.xilinx.com/html_docs/xilinx2019_2/vitis_doc/dpu_config.html#ariaid-title7. (acceso: 26.07.2020).

[44] Intel Corporation. *OpenVINO™ Toolkit Overview*. URL: https://docs.openvinotoolkit.org/latest/index.html. (acceso: 26.07.2020).

[45] Intel Corporation. *Deep Learning Inference with Intel® FPGAs*. URL: https://software.intel.com/content/www/us/en/develop/training/course-deep-learning-inference-fpga.html. (acceso: 26.07.2020).

[46] Yijin Guan y col. "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates". En: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2017), págs. 152-159. DOI: 10.1109/FCCM.2017.25.

[47] Google Brain team. *Tensorflow*. URL: https://www.tensorflow.org/. (acceso: 26.07.2020).

[48] Chen Zhang y col. "Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks". En: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.11 (2019), págs. 2072-2085. DOI: 10.1109/TCAD.2017.2785257.

[49] Zhe Zhao y col. *Recommending What Video to Watch next: A Multitask Ranking System*. Copenhagen, Denmark, 2019. DOI: 10.1145/3298689.3346997. URL: https://doi.org/10.1145/3298689.3346997.

[50] Caffe. *Caffe*. URL: https://caffe.berkeleyvision.org/. (acceso: 26.08.2020).

[51] Keras. *Keras*. URL: https://keras.io/. (acceso: 26.08.2020).

[52] Li Deng. "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]". En: *IEEE Signal Processing Magazine* 29.6 (2012), págs. 141-142. DOI: 10.1109/MSP.2012.2211477.

[53] NVIDIA. *NVIDIA V100 TENSOR CORE GPU*. URL: https://www.nvidia.com/en-us/data-center/v100/. (acceso: 03.09.2020).

[54] Ross Girshick. "Fast R-CNN". En: (2015). arXiv: 1504.08083 [cs.CV].

[55] Joseph Redmon y col. "You Only Look Once: Unified, Real-Time Object Detection". En: (jun. de 2016), págs. 779-788. DOI: 10.1109/CVPR.2016.91.

[56] OpenMP. *OpenMP*. URL: https://www.openmp.org. (acceso: 03.09.2020).

[57] Xilinx. *Alveo U200 Data Center Accelerator Card*. URL: https://www.xilinx.com/products/boards-and-kits/alveo/u200.html#specifications. (acceso: 03.09.2020).

[58] Intel. *Intel Core™ i7-7800X X-series Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/123589/intel-core-i77800x-xseries-processor-8-25m-cache-up-to-4-00-ghz.html. (acceso: 03.09.2020).

[59] Jiantao Qiu y col. "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network". En: FPGA '16 (2016), págs. 26-35. DOI: 10.1145/2847263.2847265. URL: https://doi.org/10.1145/2847263.2847265.

[60] MathWorks. *Singular value decomposition*. URL: https://es.mathworks.com/help/matlab/ref/double.svd.html. (acceso: 03.09.2020).

[61] Shail Dave y col. "Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights". En: *Proceedings of the IEEE* (2021), págs. 1-47. DOI: 10.1109/JPROC.2021.3098483.

APÉNDICE A
Modelo VGG16 utilizado

En la siguiente imagen puede verse la versión modificada del modelo VGG16 utilizada para la comparación de prestaciones entre la FPGA y la CPU.

```
1   input         | (3, 224, 224)     =>  (3, 224, 224)
2   Convolutional | (3, 224, 224)     =>  (13, 224, 224)
3   ReLU          | (13, 224, 224)    =>  (13, 224, 224)
4   Convolutional | (13, 224, 224)    =>  (24, 224, 224)
5   ReLU          | (24, 224, 224)    =>  (24, 224, 224)
6   MaxPooling    | (24, 224, 224)    =>  (24, 112, 112)
7   Convolutional | (24, 112, 112)    =>  (83, 112, 112)
8   ReLU          | (83, 112, 112)    =>  (83, 112, 112)
9   Convolutional | (83, 112, 112)    =>  (40, 112, 112)
10  ReLU          | (40, 112, 112)    =>  (40, 112, 112)
11  MaxPooling    | (40, 112, 112)    =>  (40, 56, 56)
12  Convolutional | (40, 56, 56)      =>  (185, 56, 56)
13  ReLU          | (185, 56, 56)     =>  (185, 56, 56)
14  Convolutional | (185, 56, 56)     =>  (174, 56, 56)
15  ReLU          | (174, 56, 56)     =>  (174, 56, 56)
16  Convolutional | (174, 56, 56)     =>  (147, 56, 56)
17  ReLU          | (147, 56, 56)     =>  (147, 56, 56)
18  MaxPooling    | (147, 56, 56)     =>  (147, 28, 28)
19  Convolutional | (147, 28, 28)     =>  (293, 28, 28)
20  ReLU          | (293, 28, 28)     =>  (293, 28, 28)
21  Convolutional | (293, 28, 28)     =>  (285, 28, 28)
22  ReLU          | (285, 28, 28)     =>  (285, 28, 28)
23  Convolutional | (285, 28, 28)     =>  (223, 28, 28)
24  ReLU          | (223, 28, 28)     =>  (223, 28, 28)
25  MaxPooling    | (223, 28, 28)     =>  (223, 14, 14)
26  Convolutional | (223, 14, 14)     =>  (104, 14, 14)
27  ReLU          | (104, 14, 14)     =>  (104, 14, 14)
28  Convolutional | (104, 14, 14)     =>  (308, 14, 14)
29  ReLU          | (308, 14, 14)     =>  (308, 14, 14)
30  Convolutional | (308, 14, 14)     =>  (283, 14, 14)
31  ReLU          | (283, 14, 14)     =>  (283, 14, 14)
32  MaxPooling    | (283, 14, 14)     =>  (283, 7, 7)
33  AveragePool   | (283, 7, 7)       =>  (283, 7, 7)
34  Flatten       | (283, 7, 7)       =>  (13867)
35  Gemm          | (13867)           =>  (83)
36  ReLU          | (83)              =>  (83)
37  Gemm          | (83)              =>  (20)
38  ReLU          | (20)              =>  (20)
39  Gemm          | (20)              =>  (8)
```

Figura A.1: Versión modificada del modelo VGG16.