Universidad de Concepción
Dirección de Postgrado
Facultad de Ingeniería - Programa de Magister en Ciencias de la Ingeniería con mención en Ingeniería Eléctrica

ACELERACIÓN HARDWARE PARA INFERENCIA EN REDES NEURONALES CONVOLUCIONALES

Tesis presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al grado de Magíster en Ciencias de la Ingeniería con mención en Ingeniería Eléctrica

POR: IGNACIO JESÚS PÉREZ CERDEIRA

Profesor Guía: Dr. Miguel Ernesto Figueroa Toro

Profesor Co-guía Externo: Dr. Gonzalo Andrés Carvajal Barrera

enero, 2021
Concepción, Chile

Universidad de Concepción
Facultad de Ingeniería
Departamento de Ingeniería Eléctrica

Profesor Guía:
Dr. Miguel Ernesto Figueroa Toro
Profesor Co-guía Externo:
Dr. Gonzalo Andrés Carvajal Barrera

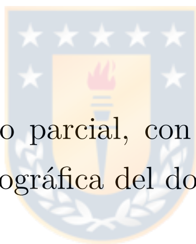
ACELERACIÓN HARDWARE PARA INFERENCIA EN REDES NEURONALES CONVOLUCIONALES

Ignacio Jesús Pérez Cerdeira

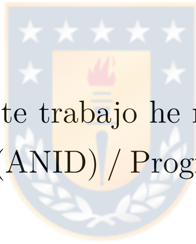
Informe de tesis para optar al grado de:
"Magíster en Ciencias de la Ingeniería con mención en Ingeniería Eléctrica"

Comisión:
Dr. Mario Medina
Dr. Julio Aracena

Concepción, Chile, enero 2021



Se autoriza la reproducción total o parcial, con fines académicos, por cualquier medio o procedimiento, incluyendo la cita bibliográfica del documento.



Durante las diferentes etapas de este trabajo he recibido el apoyo financiero de la Agencia Nacional de Investigación y Desarrollo (ANID) / Programa de Becas / MAGISTER NACIONAL 22180733.

Resumen

Gracias a su alta precisión para clasificar objetos en imágenes, las redes neuronales convolucionales (CNN) son una herramienta muy relevante en la visión computacional. Este tipo de red neuronal utiliza capas convolucionales para extraer características de las imágenes y las clasifica por medio de capas de clasificación. Típicamente, el proceso de reconocimiento, llamado inferencia, es aplicado en unidades centrales de procesamiento (CPU) o unidades de procesamiento gráfico (GPU), pero debido al alto paralelismo de estas últimas, las GPUs muestran un mejor desempeño. Aun así, su alto consumo de potencia dificulta la implementación de estas plataformas en dispositivos móviles. Por esto, una alternativa para solucionar este problema es diseñar arquitecturas hardware en sistemas dedicados, como arreglos de compuertas programables (FPGA), que permiten reducir el consumo de potencia y acelerar la inferencia.

Debido a esto, en este trabajo diseñamos una arquitectura heterogénea para realizar la inferencia de la CNN MobileNet V2 sobre hardware dedicado. Esta arquitectura utiliza una CPU y memoria embebida para controlar y almacenar datos del proceso, y un acelerador hardware sintetizado en la lógica programable de un FPGA para disminuir los tiempos de inferencia. Para reducir la cantidad de operaciones y datos en la FPGA, utilizamos técnicas de loop tiling, pruning y cuantización.

Implementamos la arquitectura sobre la plataforma Xilinx Zynq Ultrascale+, utilizando la CPU ARM Cortex-A53 como controlador, una memoria DDR4 de 2GB para almacenar datos y la FPGA XCZU7EV para sintetizar cuatro elementos de procesamiento que permiten la inferencia en paralelo. Nuestra implementación puede inferir una imagen de la base de datos ImageNet de 224×224 píxeles en 220ms, utilizando 532 bloques de RAM (BRAM), 24 RAMs de UltraScale (URAM) y 340 procesadores digitales de señales (DSP) del FPGA, y consumiendo 7.34W de potencia. Comparada con una implementación software sobre una GPU y CPU, nuestro diseño es 10.18 veces más lento que la GPU y tiene un frame-rate similar a la CPU, pero consume 29.23 y 12.93 veces menos potencia que estos dispositivos respectivamente.

Agradecimientos

En primer lugar, agradezco a mi familia, en especial a mi abuelo, hermana y padres, que han sido un gran apoyo en estos años de trabajo. Gracias por todas sus enseñanzas, que por ellas, me he convertido en la persona que soy hoy en día.

A los profesores Miguel Figueroa y Gonzalo Carvajal, por ser un guía durante el desarrollo de este trabajo, además de creer en mis capacidades y potenciarlas. A mis compañeros de laboratorio, en especial a Wladimir y Javier, por el grato ambiente y todas sus enseñanzas, que sin ellas, este trabajo no hubiera sido posible.

Por último, quiero agradecer a mi polola, Javiera, por su apoyo incondicional y los maravillosos momentos que hemos pasado juntos.

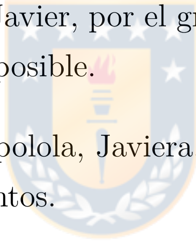


Tabla de contenidos

Resumen I

Agradecimientos II

Índice de figuras VI

Índice de tablas VIII

Abreviaciones X

1. Introducción **1**

1.1. Introducción general . . . 1

1.2. Hipótesis . . . 5

1.3. Objetivos . . . 5

1.3.1. Objetivo general . . . 5

1.3.2. Objetivos específicos . . . 5

1.4. Temario . . . 6

2. Revisión bibliográfica **7**

2.1. Conceptos claves . . . 7

2.2. Redes neuronales convolucionales . . . 8

2.3. Métodos de aceleración para redes neuronales convolucionales . . . 9

2.3.1. Métodos de optimización de ciclos `for` . . . 10

2.3.1.1. Loop tiling . . . 11

2.3.1.2. Loop unrolling . . . 11

2.3.2. Métodos de compresión de redes . . . 12

2.3.2.1. Cuantización de datos . . . 12

2.3.2.2. Descomposición por valores singulares . . . 15

2.3.2.3. Redes neuronales convolucionales hashing . . . 16

2.3.2.4. Pruning . . . 18
2.4. Aceleradores hardware . . . 20
2.5. Diseño de arquitectura con HLS . . . 22
2.5.1. Beneficios y limitaciones de HLS . . . 23
2.6. Discusión . . . 24

3. Arquitectura de MobileNet V2 **27**
3.1. MobileNet V2 . . . 27
3.2. Capas convolucionales . . . 27
3.2.1. Convoluciones . . . 28
3.2.2. Normalización . . . 31
3.2.3. Activación . . . 32
3.2.4. Módulos *bottleneck* . . . 32
3.3. Capas de clasificación . . . 34
3.3.1. Aplanamiento . . . 34
3.3.2. Componentes totalmente conectados . . . 35
3.3.3. Función clasificadora . . . 36
3.4. Organización de las capas . . . 36

4. Reducción de complejidad en MobileNet V2 **38**
4.1. Fusión de normalización 2D en convoluciones . . . 38
4.2. Loop tiling en MobileNet V2 . . . 39
4.3. Pruning en MobileNet V2 . . . 40
4.4. Cuantización en MobileNet V2 . . . 41

5. Arquitectura de acelerador hardware **43**
5.1. Sistema general . . . 43
5.2. Sección de control . . . 44
5.3. Sección de procesamiento . . . 45
5.3.1. Etapa de decodificación . . . 45
5.3.2. Etapa de procesamiento . . . 45
5.3.2.1. Acelerador de convolución clásicas y *depthwise* . . . 46
5.3.2.2. Acelerador de convolución *expansion* y *projection* . . . 48
5.3.2.3. Acelerador de aplanamiento . . . 49
5.3.2.4. Acelerador de componentes totalmente conectados . . . 50
5.3.3. Organización de datos en los elementos de procesamiento . . . 51

6. Implementación en hardware **52**

6.1. Descripción del hardware 52

6.2. Software utilizado para implementar la arquitectura 53

6.3. Consideraciones de implementación 53

6.3.1. Consideraciones para MobileNet V2 53

6.3.2. Consideraciones de diseño en HLS 56

6.3.2.1. Directiva HLS PIPELINE 56

6.3.2.2. Directiva HLS UNROLL 57

6.3.2.3. Directiva HLS ARRAY PARTITION 59

6.3.2.4. Directiva HLS RESOURCE 60

6.3.2.5. Directiva HLS INTERFACE 60

6.4. Comparación de implementación punto fijo y punto flotante 61

7. Resultados **64**

7.1. Reportes de la implementación en FPGA 64

7.1.1. Recursos utilizados 64

7.1.2. Latencia 66

7.1.3. Frecuencia de reloj y camino crítico 69

7.1.4. Consumo de potencia 69

7.1.5. Tiempo de inferencia 71

7.2. Escalabilidad 72

7.3. Análisis de resultados 73

7.3.1. Comparación con otros trabajos 73

7.3.2. Comparación con software 75

7.3.3. Discusión 75

8. Conclusión **78**

9. Anexo **82**

9.1. Repositorio 82

Bibliografía **87**

Índice de figuras

1.1. Funcionamiento de redes neuronales convolucionales. 2

2.1. Distribución de cómputo y parámetros en redes neuronales. 10

2.2. Ciclo `for` normal y con loop tiling. 11

2.3. Ciclo `for` normal y con loop unrolling x2. 12

2.4. Flujo de cuantización con ajuste fino. 14

2.5. Cuantización por centroides y clusters. 16

2.6. Arquitectura red neuronal convolucional hashing [1]. 17

2.7. Arquitectura red neuronal convolucional hashing [2]. 18

2.8. Pruning en red neuronal. 19

2.9. Comparación de metodos de pruning. 20

3.1. Extracción de características en imágenes mediante convolución con paso 2. . . . 28

3.2. Tipos de convolución en MobileNet V2. 30

3.3. Módulos *bottleneck*. 33

3.4. Modelo de componentes totalmente conectados. 35

4.1. Loop tiling aplicado sobre mapas y pesos de MobileNet V2. 40

4.2. Pruning por bloques aplicado en MobileNet V2. 40

5.1. Sistema general de la arquitectura hardware. 44

5.2. Arquitecturas de aceleradores de convoluciones clásicas y *depthwise*. 46

5.3. Funcionamiento buffers de línea y banco de registros. 47

5.4. Arquitectura de acelerador de convoluciones *expansion* y *projection*. 48

5.5. Arquitectura de acelerador de aplanamiento. 50

5.6. Arquitectura de acelerador de componentes totalmente conectados. 50

5.7. División espacial de las activaciones y pesos en hardware. 51

6.1. Plataforma ZCU104. 54

6.2. Desempeño de ciclo `for` con y sin pipeline. 56

6.3. Comportamiento de ciclo principal de sección de procesamiento al aplicar o no loop unrolling. 58

6.4. Loop unrolling aplicado en convoluciones *expansion/projection*. 59

Índice de tablas

2.1. Compresión de pruning en diferentes modelos de redes neuronales. 19

2.2. Comparación de implementaciones de redes neuronales convolucionales en hardware dedicado. 22

3.1. Organización de capas en MobileNet V2. 37

6.1. Características de plataforma ZCU104. 53

6.2. Tiempos de inferencia al variar factores de loop tiling. 54

6.3. Resultados de pruning en MobileNet V2. 55

6.4. Resultados cuantización lineal en MobileNet V2. 55

6.5. Ciclos de procesamiento de cada acelerador al aplicar o no pipeline. 57

6.6. Ciclos de procesamiento con diferentes factores de loop unrolling en convoluciones *expansion/projection*. 58

6.7. Porcentaje de utilización de recursos en arquitecturas de punto fijo y punto flotante. 62

6.8. Tiempos de inferencia y consumo de potencia en arquitecturas de punto fijo y punto flotante. 62

7.1. Uso de recursos por cada PE. 65

7.2. Uso de recursos totales. 66

7.3. Uso de recursos de la arquitectura diseñada. 66

7.4. Latencia y ciclos de procesamiento en aceleradores de cada PE. 67

7.5. Potencia consumida por cada PE. 70

7.6. Potencia consumida por recurso. 70

7.7. Potencia consumida de la arquitectura diseñada. 71

7.8. Tiempos de inferencia en cada capa de MobileNet V2. 72

7.9. Porcentaje de utilización de recursos al variar la cantidad de PEs en la FPGA XCZU19EG. 73

7.10. Tiempos de inferencia al variar la cantidad de PEs en la FPGA XCZU19EG. . . 73

Abreviaciones

ASIC circuito integrado de aplicación específica (del inglés *Application-Specific Integrated Circuit)*

BRAM bloque de RAM (del inglés *Block RAM)*

CNN red neuronal convolucional (del inglés *Convolutional Neural Network)*

CPU unidad central de procesamiento (del inglés *Central Processing Unit)*

DMA acceso directo a memoria (del inglés *Direct Memory Access)*

DPU unidad de procesamiento de aprendizaje profundo (del inglés *Deep Learning Processing Unit)*

DSP procesador de señales digitales (del inglés *Digital Signal Processor)*

FC capa completamente conectada (del inglés *Fully Connected Layer)*

FIFO primero en entrar-primero en salir (del inglés *First In-First Out)*

FPGA arreglo de compuertas programables (del inglés *Field Programmable Gate Array)*

fps cuadros por segundo (del inglés *frame per second)*

GOPS giga operaciones por segundo (del inglés *Giga Operations per Second)*

GPU unidad de procesamiento gráfico (del inglés *Graphic Processing Unit)*

HLS síntesis de alto nivel (del inglés *High-Level Synthesis)*

LUT tabla de búsqueda (del inglés *Look Up Table)*

PE elemento de procesamiento (del inglés *Processing Element)*

PL lógica programable (del inglés *Programmable Logic)*

PS subsistema de procesamiento (del inglés *Processing Subsystem)*

RAM memoria de acceso aleatorio (del inglés *Random Access Memory)*

ReLU unidad lineal rectificada (del inglés *Rectified Linear unit)*

RGB rojo-verde-azul (del inglés *Red Green Blue)*

RTL nivel de transferencia de registro (del inglés *Register-Transfer Level)*

SVD descomposición por valores singulares (del inglés *Singular Value Decomposition)*

URAM RAM de UltraScale (del inglés *Ultrascale RAM)*

VLSI circuito de transistores integrados de gran escala (del inglés *Very Large Scale Integration)*

Capítulo 1

Introducción

1.1. Introducción general

En la actualidad, las redes neuronales artificiales son frecuentemente utilizadas para varias aplicaciones debido a su versatilidad. Dentro de estas podemos destacar el procesamiento de lenguaje, robótica, programación automática, procesamiento de imágenes, entre otras [3]. Respecto al procesamiento de imágenes, éste es un campo que se ha visto beneficiado por los avances en computación e inteligencia artificial, permitiendo realizar funciones que en años pasados eran muy complejas de llevar a cabo, como el reconocimiento e identificación de objetos según su tipo. Gracias a esto, han aparecido soluciones a aplicaciones computacionales como el reconocimiento de rostros para identificación de personas [4], detección de obstáculos para manejo automático [5], restauración de calidad en imágenes de súper resolución [6], entre otras. Esto se debe a la aparición de nuevos algoritmos que han aumentado las tasas de acierto en el reconocimiento de objetos. El principal artífice de esto son las CNNs (del inglés *Convolutional Neural Network*s, redes neuronales convolucionales) [7], que gracias a trabajos como [8, 9, 10] han permitido alcanzar precisiones de reconocimiento superiores al 70 % para grupos de imágenes de 1000 clases.

Las CNNs son un tipo de red neuronal artificial que reemplazan las multiplicaciones y pesos unidimensionales por convoluciones y máscaras de filtros. El funcionamiento de éstas es mostrado en la Figura 1.1. Cada CNN está dividida en capas convolucionales y capas de clasificación. Las primeras realizan la convolución de las imágenes por N filtros para extraer patrones, donde N es la profundidad o número de canales de la capa, seguidas de una función de activación que elimina datos poco relevantes, obteniendo mapas de características o activaciones. Además de

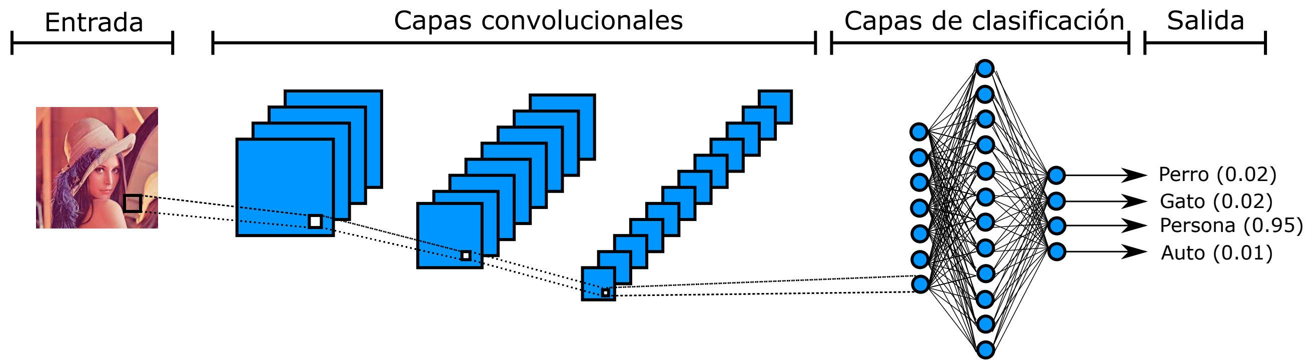


Fig. 1.1: Funcionamiento de redes neuronales convolucionales.

la convolución y activación de datos, algunas de estas capas incorporan funciones de reducción, que disminuyen el tamaño de la imagen para simplificar el cómputo de la red, como *max pooling*, que reduce la imagen eliminando datos en una ventana exceptuando el de mayor valor. Por su parte, las capas de clasificación están encargadas de identificar qué tipo de objeto es el más probable que esté contenido en la imagen de entrada. Para esto, estas capas unen todas las salidas de las neuronas de la última capa convolucional entre sí, multiplicando cada una por un peso. Este proceso es repetido una cierta cantidad de veces hasta que la red entrega una cantidad C de datos de salida, donde C corresponde a la cantidad de clases u objetos posibles a identificar. Finalmente, la red utiliza una función clasificadora, típicamente SoftMax [11], para identificar cual es la clase más probable que se corresponda con la imagen de entrada. A todo este proceso se le conoce como inferencia.

Para calcular los parámetros de la red, es necesario aplicar un proceso de entrenamiento denominado *backpropagation* [12]. Éste consiste en realizar la inferencia a un grupo de imágenes, denominado grupo de entrenamiento, cuyos resultados esperados son conocidos por el proceso. Cuando la inferencia es completada, el algoritmo calcula el error entre las salidas obtenidas y las correctas. Este error es propagado desde la capa de salida hasta la capa de entrada, con el fin recalcular cada peso en función de la contribución que tuvo cada uno de ellos en los resultados de la inferencia. Este proceso es repetido en varias iteraciones, también conocidas como *epochs*, hasta que la CNN alcanza un error aceptable para la aplicación. Cabe destacar que el entrenamiento utiliza un hiperparámetro denominado tasa de aprendizaje, que limita variaciones abruptas de los parámetro entre cada *epoch*.

La inferencia de CNNs puede ser realizada sobre GPUs (del inglés *Graphic Processing Unit*s, unidades de procesamiento gráfico) o CPUs (del inglés *Central Processing Unit*s, unidades centrales de procesamiento), pero debido al alto paralelismo propio de las imágenes, las GPUs tienen la ventaja de inferir en un menor tiempo. Esto se debe a que las GPUs están optimizadas

para procesar cargas de trabajo en paralelo, debido a que poseen una gran cantidad de núcleos en comparación con las CPUs, que si bien poseen núcleos de mayor tamaño, la cantidad de éstos es menor ya que están enfocadas para trabajos secuenciales. Aun así, las GPUs presentan el inconveniente de tener un alto consumo de potencia para inferir imágenes, llegando a los 250W para las más modernas [13]. Sin embargo, GPU embebidas, como NVIDIA Jetson usando el entorno de trabajo NVIDIA TensorRT [14], permiten optimizar la inferencia de imágenes utilizando un bajo consumo de potencia en comparación con las GPUs tradicionales. Aun así, su consumo de potencia sigue siendo alto para ser utilizadas en aplicaciones donde se necesita un bajo consumo de potencia por sobre velocidad de inferencia.

Un ejemplo de este tipo de aplicaciones es el reconocimiento de rostros en dispositivos móviles, usado para seguridad o propósitos sociales, que con el fin de maximizar la energía en las baterías, minimizan el consumo de potencia a costa del frame-rate alcanzado, siendo 1 fps (del inglés *frame per second*, cuadros por segundo) suficientemente rápido para detectar una cara [15]. Otra aplicación que privilegia el consumo de potencia es la clasificación de imágenes en drones, donde debido a que gran parte de la energía es utilizada por los motores del dispositivo, es imperioso disminuir el consumo de potencia del resto del hardware [16]. Por esto, una solución al alto consumo de potencia del hardware tradicional, es realizar la inferencia de imágenes en plataformas dedicadas, como FPGAs (del inglés *Field Programmable Gate Array*s, arreglos de compuestas programables) o ASICs (del inglés *Application-Specific Integrated Circuit*s, circuitos integrados de aplicación específica), ya que permiten desarrollar hardware específico, reduciendo el uso de recursos, maximizando el paralelismo y manteniendo el desempeño de otros dispositivos hardware, reduciendo la potencia consumida. [17].

Si bien los ASICs y FPGAs permiten utilizar menos recursos y consumir menos potencia, existen limitaciones para implementar las CNNs en estos dispositivos. Uno de ellos es el alto costo computacional de las CNNs, ya que en general los modelos se han vuelto más complejos para alcanzar tasas de acierto más elevadas, provocando que aumente el número de operaciones por imagen a inferir. Por ejemplo, AlexNet (2012) [8] utiliza 1.4 GOPS (del inglés *Giga Operations per Second*, giga operaciones por segundo), pero alcanza una tasa de acierto del 57.2 %, precisión baja en comparación con CNNs más modernas, como ResNet (2015) [18], que utiliza 22.6 GOPS pero alcanza 78.5 % de precisión, por lo que CNNs con arquitecturas similares a AlexNet no son muy usadas actualmente. Esta cantidad de operaciones es un problema en las FPGAs, ya que los recursos aritmético/lógicos son limitados.

Otra limitación para implementar CNNs en ASICs y FPGAs es la cantidad de parámetros

que tienen los diferentes modelos de este tipo de redes. Por ejemplo, GoogLeNet (2015) [10] tiene aproximadamente 7 millones, mientras que VGG (2014) [9] tiene cerca de 140 millones de parámetros. Esto genera un conflicto tanto en ASICs y FPGAs, ya que a diferencia de las CPUs y GPUs, donde disponemos de memorias en el orden de los GBs para almacenar datos, la cantidad de memoria en el chip de los dispositivos de hardware dedicado es limitada, siendo del orden de los MBs, por lo que se debe recurrir a almacenar datos en memoria externa [19]. Esto afecta negativamente los tiempos de inferencia, ya que la memoria externa en las FPGAs tiene un ancho de banda limitado comparado con otros dispositivos hardware, como GPUs [20].

Por esto, investigaciones han abordado diferentes formas de solucionar estos inconvenientes. La primera es realizar una compresión de la red por medio de técnicas como pruning [21], descomposición por valores singulares [13], clasificación por hash [22] o cuantización de datos [23], permitiendo que las redes usen menos memoria y a su vez reduzcan la cantidad de operaciones por inferencia. Además, técnicas como loop tiling [19] son utilizadas para disminuir los accesos a memoria y así poder utilizar de mejor forma el ancho de banda disponible.

Aun así, es necesario mezclar estas estrategias de implementación con una CNN que tenga baja complejidad de cómputo y pocos parámetros con el fin de reducir el consumo de potencia, el uso de recursos y lograr una tasa de cuadros por segundo aceptable para aplicaciones como reconocimiento de rostros en dispositivos móviles. MobileNet V2 [24] (cuya arquitectura es explicada en detalle en el Capítulo 3) presenta estas características, ya que reemplaza las capas convolucionales clásicas por módulos *bottleneck*, que combinan convoluciones *expansion* que aumentan la profundidad aplicando N convoluciones 1×1 por canal, convoluciones *depthwise* que extraen características al aplicar una sola convolución 3×3 por canal, y convolución *projection* que disminuyen la profundidad aplicando N convoluciones 1×1 por canal. Esto reduce la complejidad y la cantidad de parámetros de MobileNet V2 al compararla con otras CNNs [24]. Además, este modelo utiliza capas residuales, capas que utilizan mapas de características anteriores para obtener resultados más precisos, permitiendo aumentar el número de capas y a su vez mejorar la precisión de la red.

Debido a esto, en este trabajo de tesis diseñamos una arquitectura hardware para acelerar la inferencia de MobileNet V2, pero manteniendo un bajo uso de recursos y consumo de potencia. Para esto, aplicamos sobre la CNN diferentes técnicas para optimizar el uso del ancho de banda y reducción de parámetros, como loop tiling, pruning y cuantización de datos. La arquitectura diseñada está basada en un sistema heterogéneo que combina la lógica programable de un FPGA para sintetizar un acelerador hardware para la inferencia, sumado a una CPU y memoria externa

para controlar el proceso y almacenar los pesos y activaciones respectivamente. Cabe destacar que en función de disminuir el tiempo de diseño de la arquitectura respecto a métodos de diseño de RTL (del inglés *Register-Transfer Level*, nivel de transferencia de registro) tradicionales con Verilog y VHDL, utilizamos la herramienta HLS (del inglés *High-Level Synthesis*, síntesis de alto nivel), proceso de diseño automático que interpreta un algoritmo codificado en un lenguaje de alto nivel (como C o C++) y lo convierte en un RTL.

1.2. Hipótesis

Mediante técnicas de aceleración, reducción y cuantización de redes es posible diseñar una arquitectura hardware capaz de aplicar inferencia de la red MobileNet V2 para reconocimiento de imágenes con una baja utilización de recursos y consumo de potencia.

1.3. Objetivos

1.3.1. Objetivo general

Diseñar una arquitectura hardware sobre un FPGA para acelerar la inferencia de MobileNet V2, garantizando un bajo uso de recursos y consumo de potencia.

1.3.2. Objetivos específicos

Los objetivos específicos son los siguientes:

- Utilizar técnicas de pruning y cuantización de datos para reducir los parámetros de MobileNet V2.
- Definir técnicas para acelerar el cómputo y maximizar la reutilización de datos en la arquitectura a diseñar.
- Diseñar e implementar la arquitectura sobre un FPGA.

- Evaluar la precisión de clasificación y velocidad de inferencia para una imagen en la arquitectura hardware respecto a su versión en hardware programable.
- Comparar el uso de recursos y consumo de potencia de la arquitectura diseñada respecto a otros dispositivos hardware.

1.4. Temario

El siguiente trabajo está organizado de la siguiente forma:

- Capítulo 2: Presenta la revisión bibliográfica de este trabajo, detallando diferentes modelos de CNN, técnicas para la aceleración de la inferencia y los resultados de diferentes trabajos que han implementado este tipo de redes sobre hardware dedicado.
- Capítulo 3: Detalla el modelo de la CNN MobileNet V2, mostrando los tipos de capas convolucionales, la capa de clasificación y la organización de estas.
- Capítulo 4: Muestra las diferentes técnicas que aplicamos al modelo original de MobileNet V2 para reducir su complejidad, cantidad de pesos y bits, además de maximizar el ancho de banda de memorias.
- Capítulo 5: Describe en profundidad la arquitectura diseñada para acelerar la inferencia de MobileNet V2 sobre un FPGA.
- Capítulo 6: Detalla la plataforma FPGA y el software utilizado para implementar la arquitectura, además de mostrar las consideraciones de loop tiling, pruning y cuantización de datos utilizados en la red. También se muestran las directivas HLS utilizadas y una comparación de desempeño de la arquitectura punto fijo y punto flotante.
- Capítulo 7: Muestra los resultados obtenidos al implementar la arquitectura, como el uso de recursos, consumo de potencia, tiempo de inferencia y la comparación de estos con software y otros trabajos.
- Capítulo 8: Resume el trabajo realizado, mostrando las conclusiones y el trabajo futuro a realizar.

Capítulo 2

Revisión bibliográfica

En este capítulo mostramos los trabajos relacionados con respecto a la implementación en hardware dedicado de las CNNs. En específico, detallamos algunos modelos de CNNs, los principales métodos de aceleración de inferencia, aceleradores hardware diseñados para este tipo de redes y una descripción de la herramienta de diseño de hardware HLS.

2.1. Conceptos claves

Si bien el enfoque de este capítulo son las arquitecturas de CNNs y los diferentes métodos para implementarlas sobre hardware dedicado, antes debemos introducir algunos conceptos que son necesarios para entender el funcionamiento de estas redes. Aquí destacamos la base de datos ImageNet y las métricas que se utilizan para medir las tasas de precisión en las CNNs.

Para reconocimiento de imágenes en CNNs típicamente se utiliza la base de datos ImageNet [25]. Este grupo de imágenes es utilizada en el concurso ILSVRC challenge. ImageNet tiene un poco más de 1.3 millones de imágenes RGB (del inglés *Red Green Blue*, rojo-verde-azul) de alta definición divididas en 1000 clases. Estas imágenes están agrupadas en un grupo de entrenamiento, para entrenar la red, grupo de validación, para validar los parámetros de la CNN en entrenamiento e inferencia, y grupo de testeo, cuyas imágenes no contienen clasificación y son utilizadas en el concurso para evaluar la precisión de las redes neuronales. El grupo de entrenamiento cuenta con 1.3 millones de imágenes, mientras que el conjunto de validación agrupa 50000 imágenes, 50 por clase.

Para medir la precisión en esta base de datos, los investigadores utilizan las métricas de top-1 y top-5. Estas indican si la clase o tipo objeto presente en la imagen de entrada (resultado esperado) corresponde a la clase que tiene el resultado con mayor probabilidad al terminar la inferencia para el caso de top-1 o si se encuentra entre las 5 más probables para top-5.

2.2. Redes neuronales convolucionales

Los primeros estudios sobre las CNNs aparecen con [26]. En este trabajo, los investigadores diseñaron un modelo de red neuronal inspirado en la corteza visual de los seres vivos, en el cual las neuronas están divididas en capas que están conectadas unas con otras para identificar información cada vez más compleja y propagarla entre ellas. Más tarde, con [7] el modelo mejoró al agregar entrenamiento con propagación hacia atrás, permitiendo una precisión más elevada al reconocer imágenes.

En la última década, los modelos de CNN han evolucionado de ser simples, sólo presentando capas de convolución y clasificación, pero con una gran cantidad de pesos, superando muchas veces los 50 millones, a ser más complejos al tener diferentes tipos de etapas dentro de cada capa, como etapas de normalización, múltiples convoluciones, aplicación de residuales, etc, pero logrando disminuir considerablemente la cantidad de parámetros en la red, siendo típicamente menor a los 10 millones de pesos. Dentro del primer grupo de modelos descritos podemos incluir a AlexNet [8] (2012), ZeilerNet [11] (2014) y VGG-16 [9] (2014). AlexNet y ZeilerNet presentan una arquitectura similar, con 5 capas de convolución y 3 capas de clasificación, con cerca de 60 millones de pesos y alcanzado una precisión top-1 en ImageNet de 57.2 % y 62.5 % respectivamente. Por su parte, VGG-16 presenta una mayor cantidad de capas, 13 convolucionales y 3 de clasificación, aumentando la cantidad de pesos a 138 millones. Aun así, la precisión de esta red alcanza un 68.5 % para top-1 sobre ImageNet.

Respecto a los modelos más complejos, podemos destacar a GoogLeNet [10] (2014), ResNet [18] (2015), SqueezeNet [27] (2016) y MobileNet [28] (2017). GoogLeNet utiliza módulos *inception* en cada capa, los cuales aplican en paralelo convoluciones 1x1, 3x3 5x5 y una etapa de reducción, presentando cerca de 6 millones de parámetros y una precisión del 77.45 % en top-1 en ImageNet. ResNet utiliza capas convolucionales residuales, que consisten en utilizar mapas de características de capas anteriores, permitiendo aumentar el numero de capas y a su vez la precisión, logrando un 78.47 % en ImageNet. El uso de residuales es fundamental para aumentar el número de capas, ya que sin estas, si la cantidad de capas es muy grande, el gradiente de

entrenamiento recibido por las neuronas internas es prácticamente cero, por lo que conexiones extras ayudan a compensar este problema.

SqueezeNet es un modelo reducido de AlexNet que sólo utiliza 1.2 millones de pesos y alcanza una precisión de 57.5 % en ImageNet. La ventaja de este modelo respecto a AlexNet es que utiliza módulos *fire*, que aplican en una primera instancia una convolución 1x1 seguidas de convoluciones 1x1 y 3x3 en paralelo, además de la ausencia de capas de clasificación en la red, siendo reemplazadas por una capa de convolución tradicional. MobileNet reemplaza las convoluciones normales por convoluciones en profundidad. Para esto, esta red utiliza convoluciones *depthwise*, que aplican una sola convolución $K \times K$ $(K > 1)$ por canal de la imagen, y convoluciones *pointwise*, que aplican N convoluciones 1x1 por cada canal. Gracias a esto, la red utiliza 4.2 millones de parámetros y alcanza una precisión de 70.6 % para top-1 sobre ImageNet.

En 2018, [24] realiza mejoras a [28] obteniendo MobileNet V2. Aquí, los autores modifican las capas *depthwise* y *pointwise* de MobileNet aplicando el grupo de capas denominado *bottleneck*. Cada *bottleneck* aplica las siguientes capas: (1) la capa *expansion*, que realiza convoluciones 1x1 y aumenta el número de canales del mapa de entrada, (2) capas *depthwise* de MobileNet con filtros 3x3, para finalmente aplicar (3) la capa *projection* que realiza convoluciones 1x1 y disminuye el número de canales al original. Adicionalmente, el algoritmo utiliza residuales sumando el mapa de entrada y salida de algunos *bottleneck*, además de etapas de normalización para mejorar los resultados. Gracias a *bottleneck* y residuales, MobileNet V2 utiliza menos pesos que su predecesora, 3.4 millones para una resolución de 224x224 píxeles. Esta red alcanza una precisión sobre ImageNet top-1 y top-5 de 71.8 % y 91.0 % respectivamente.

2.3. Métodos de aceleración para redes neuronales convolucionales

Como hemos mencionado en el Capítulo 1, las principales dificultades para implementar CNNs sobre hardware es la cantidad de cómputo necesario y el tamaño de las redes producto de la gran cantidad de pesos. Tanto el cómputo como los parámetros no están equitativamente distribuidos en los diferentes tipos de capas, sino que cada una de ellas presenta una mayor o menor cantidad de dichas variables, tal como muestra la Figura 2.1. Específicamente, en la capa de convolución, la red realiza una gran parte de los cálculos debido a la cantidad de convoluciones por capa y a la complejidad de esta operación [19], mientras que como en la capa de clasificación

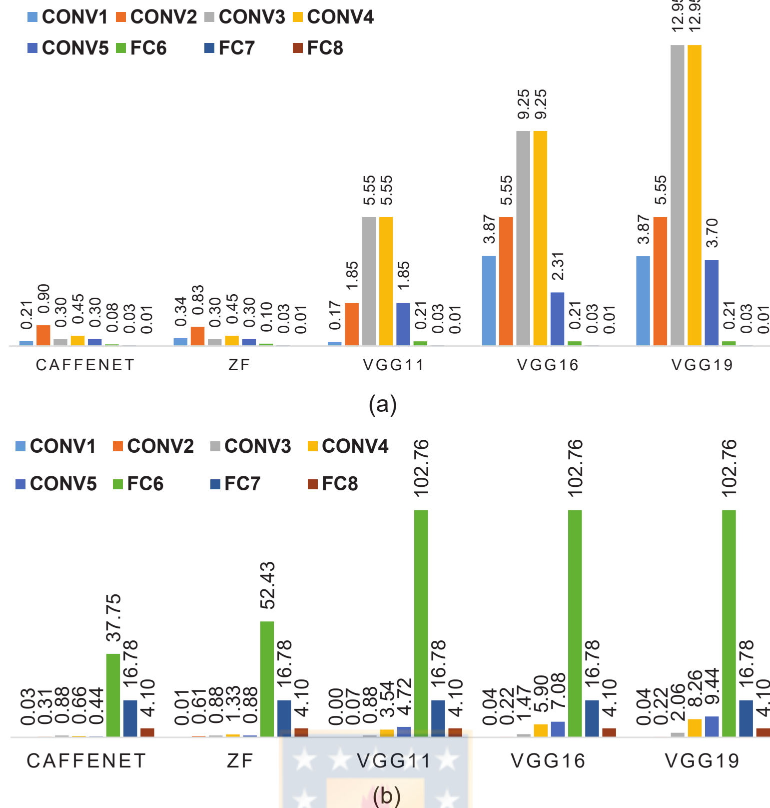


Fig. 2.1: Distribución de cómputo y parámetros en redes neuronales.
(a) Operaciones necesarias en diferentes capas (GOP). (b) Cantidad de pesos en diferentes capas (Millones). (Fuente: [13]).

las neuronas deben conectarse entre sí, la red necesita una gran cantidad de pesos, aumentando la cantidad de datos y memoria a utilizar [23, 13].

Para solucionar estos dos inconvenientes, existen diferentes técnicas que permiten facilitar y mejorar el rendimiento de las CNNs en hardware. Éstas las podemos dividir en dos tipos: (1) métodos de optimización de ciclos `for` y (2) métodos de compresión de redes.

2.3.1. Métodos de optimización de ciclos `for`

Uno de los problemas de las CNNs en hardware es la cantidad de pesos necesarios para realizar inferencia. Una alternativa que ayuda a reducir los tiempos de cómputo es reutilizar la mayor cantidad de datos antes de volver a leer desde memoria, optimizando el uso del ancho de banda. Algunas técnicas que ayudan en este propósito son loop tiling y loop unrolling.

Ciclo for normal	Ciclo for con loop tiling
```for(int i=0; i<N; i++){ for(int j=0; j<M; j++){ ... } }```	```for(int i=0; i<N; i+=A){ for(int j=0; j<M; j+=B){ for(int k=i; k<min(N, i+A); k++){ for(int l=j; l<min(M, j+B); l++){ ... } } } }```

Fig. 2.2: Ciclo `for` normal y con loop tiling.

2.3.1.1. Loop tiling

Loop tiling es una técnica utilizada en los ciclos `for` para maximizar la reutilización de datos, reduciendo los accesos a memoria. La Figura 2.2 muestra un ciclo `for` normal y con loop tiling, donde en este último se subdividen los ciclos `for` en bloques más pequeños, volviendo acceder a memoria sólo cuando todas las operaciones fueron realizadas con los datos ya leídos.

En aceleradores hardware para CNNs, loop tiling es una técnica comúnmente utilizada para reducir los accesos a memoria al realizar las convoluciones de la red, permitiendo utilizar ancho de banda del hardware en otros procesos. En [19] los investigadores utilizan loop tiling sobre los ciclos `for` con un mayor número de iteraciones. Los ciclos con loop tiling son los ciclos del mapa de características (filas y columnas) y las profundidades de las capas de entrada y salida (cantidad de filtros). La arquitectura propuesta en este trabajo realiza el loop tiling de estos 4 ciclos en el chip del dispositivo, guardado los datos de estos sub-bloques en la memoria interna del dispositivo, mientras que utiliza memoria externa sólo para leer los datos de entrada de cada loop tiling, reduciendo los accesos a memoria y acelerando el cómputo.

Al igual que en [19], en [13] utilizan loop tiling sobre los ciclos de columnas y filas del mapa de características y las profundidades de las capa de entrada y salida de la red. Además, en esta arquitectura los investigadores utilizan loop tiling sobre las capas de clasificación de la red, específicamente sobre la matriz de pesos (filas y columnas) y el largo del vector de datos de cada capa, reduciendo aún más los accesos a memoria.

2.3.1.2. Loop unrolling

Loop unrolling es una técnica utilizada en los ciclos `for` para reducir los tiempos de espera o burbujas en el procesador. La Figura 2.3 muestra un ciclo `for` normal y con loop unrolling,

Ciclo for normal	Ciclo for con loop unrolling x2

`for(int i=0; i<N; i++){`
`function (i);`
`}`
	`for(int i=0; i<N; i+=2){`
	`function (i);`
	`function (i+1);`
	`}`

Fig. 2.3: Ciclo `for` normal y con loop unrolling x2.

donde en este último se aumenta el tamaño del ciclo en un factor dos, agregando dos iteraciones más en el mismo bucle para eliminar burbujas y así agilizar los tiempos de ejecución del ciclo.

En [19] utilizan loop unrolling para maximizar la cantidad de recursos de la FPGA que realizan cómputo, es decir, evitar que existan burbujas en las unidades de procesamiento del sistema. En este trabajo, los autores realizaron loop unrolling sobre los ciclos tiling de las profundidades de las capas de entrada y salida de la red. Si bien determinaron que para cada capa la cantidad de loop unrolling debe ser diferente para minimizar las burbujas, esto es muy complejo de realizar, por lo que aplican loop unrolling x7 y x64 en los ciclos de profundidades de entrada y salida respectivamente. Combinando ambas técnicas, [19] logra una aceleración de un 11.73 % respecto a trabajos que no utilizan loop tiling ni loop unrolling [29].

2.3.2. Métodos de compresión de redes

Si bien la reutilización de datos y pesos con loop tiling y loop unrolling disminuye los accesos a memoria y aceleran la inferencia, la cantidad de parámetros de la red sigue siendo grande. Por esto, existen diferentes técnicas que reducen la cantidad de pesos de las CNNs, conocidas como compresión de redes. Dentro de estas podemos encontrar las técnicas de cuantización de datos, hashing, descomposición por valores singulares y pruning.

2.3.2.1. Cuantización de datos

Debido a que las CNNs utilizan millones de parámetros para inferir, sumado al hecho de que los pesos son valores punto flotante, la cantidad de memoria para almacenarlos puede superar los 20 MB, complicando el almacenamiento de todos los pesos en la memoria interna de los FPGAs. Una forma de reducir la cantidad de memoria utilizada y simplificar el cómputo es

convertir los números punto flotante a punto fijo.

En [13] utilizan cuantización lineal de punto fijo para guardar los pesos y mapas de características de la red. Sin embargo, a diferencia de trabajos anteriores, donde la cantidad de bits fraccionarios es estática en todas las capas de la CNN, los autores utilizaron cuantización dinámica, que cambia la posición del punto fijo en cada capa, con el propósito de minimizar el error por truncar los pesos y mapas de características.

La ecuación (2.1) muestra la cantidad de bits fraccionarios para cada capa que reducen el error de precisión utilizada en [13]. Aquí, fb y db representan la cantidad de bits de la parte fraccionaria y entera de punto fijo respectivamente, mientras que X es el valor a cuantizar (peso o dato del mapa).

$$fb = \arg\min_{x} \sum |X_{float} - X(db, fb)| \tag{2.1}$$

Los autores obtuvieron los siguientes resultados al aplicar el método estático y dinámico con diferente cantidad de bits fraccionarios sobre VGG-16. Para el caso de punto fijo estático, usando 16 bits de cuantización la precisión de la red cae un 0.06 %, mientras que para 8 bits cae un 38.34 %. Usando cuantización dinámica, los investigadores usaron 8 bits para los pesos y datos de los mapas, decreciendo la precisión en un 0.62 %, mientras que utilizando 8 bits para los datos y 4 u 8 para los pesos dependiendo de la capa, la precisión cae un 0.4 %, lo que demuestra un mejor desempeño de la cuantización dinámica. Cabe destacar que este trabajo no compara el uso de recursos y consumo de potencia entre los dos tipos de cuantización.

En [23] utilizan cuantización logarítmica de punto fijo dinámico para calcular la cantidad de bits que representa la parte fraccionaria en cada capa, siguiendo el esquema mostrado en la Figura 2.4. En este trabajo los autores realizan un histograma con los valores logarítmicos de los pesos y datos de cada capa para elegir la posición del punto fijo de los pesos y datos. En caso de overflow, dichos valores toman el máximo de la capa, mientras que para parámetros con underflow, son reemplazados por cero. Una vez elegida la posición del punto fijo, realizan un test de precisión para minimizar el error que introduce esta representación. Si el error es muy alto, el sistema selecciona una nueva posición para el punto fraccionario, repitiendo el proceso hasta encontrar el error mínimo, para luego continuar con las siguientes capas.

Adicionalmente, los autores añaden al flujo de cuantización el método de ajuste fino, que consiste en recalcular el valor de pesos y datos en base a la posición del punto fijo. Para esto,

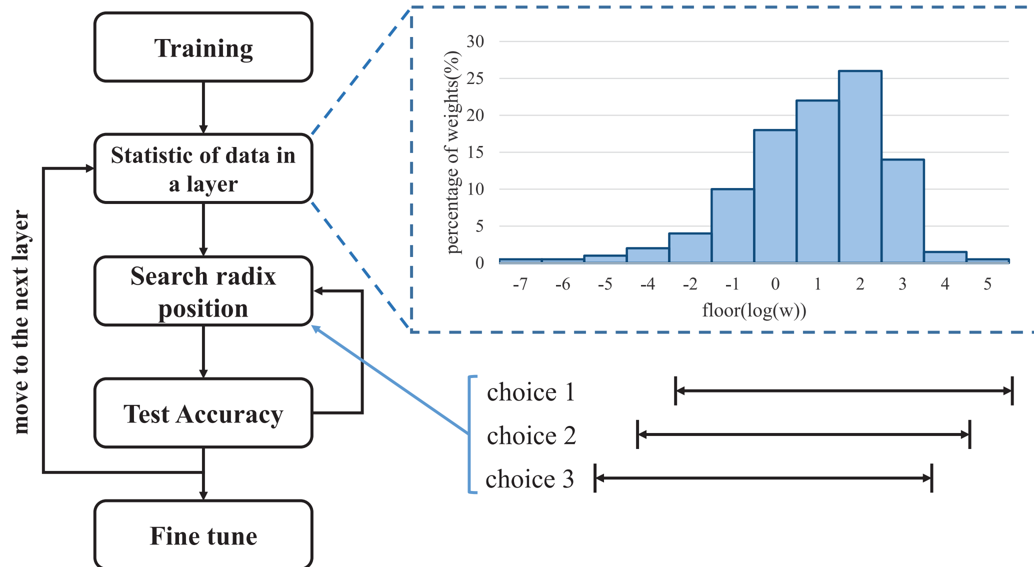


Fig. 2.4: Flujo de cuantización con ajuste fino.
(Fuente: [23]).

consideran los gradientes, pesos y datos como punto flotante, lo que permite variar el valor de los nuevos pesos, mejorando la precisión de la red punto fijo.

Los autores comparan la precisión obtenida al usar métodos de cuantización sin reentrenamiento (como [13]) con la de ajuste fino. Al usar 8 bits de cuantización sin ajuste fino, la precisión cae un 1.76 %, mientras que con este método, la precisión decae tan sólo un 0.08 %. Al disminuir la cantidad de bits a 6, la precisión decae 18 % y 7 % para cuantización sin y con ajuste fino respectivamente.

En [30] usan la ecuación (2.2) para cuantizar de forma precisa los pesos punto flotante a punto fijo en el rango $[-1, 1]$. Aquí, $tanh(w)$ lleva los pesos al rango $[-1, 1]$, mientras que al dividir por $2max|tanh(w)|$ y sumar 0.5 al resultado, el rango es normalizado a $[0, 1]$. La función $Q_k(\cdot)$ cuantiza los valores obtenidos a k bits punto fijo mediante $\frac{i}{2^k-1}$, con $i = 0, 1, 2, ..., 2^{k-1}$. Posteriormente, los resultados de la cuantización son multiplicados por 2 y restados por 1 para obtener pesos en el rango $[-1, 1]$.

$$w_k = 2 \cdot Q_k(\frac{\tanh(w)}{2 \cdot \max|\tanh(w)|} + 0.5) - 1 \tag{2.2}$$

Para el caso de los datos, [30] utiliza la función PACT$(\cdot)$ de la ecuación (2.3). Esta función normaliza los datos x de cada capa de la red al rango $[0, |\alpha|]$, con $|\alpha|$ calculado mediante entrenamiento. Luego, usando la ecuación (2.4) los datos son cuantizados a k bits punto fijo.

Los autores recomiendan reemplazar el factor multiplicativo $|\alpha|$ de la ecuación (2.4) por $|s|$, obteniendo la ecuación (2.5). $|s|$ es un valor común en todas las capas, lo que reduce el cómputo en la CNN. Utilizando este método, los investigadores logran cuantizar los pesos y datos a 1 y 4 bits respectivamente en las capas convolucionales y a 8 bits para los pesos de las capas de clasificación.

$$y = \mathrm{PACT}(x) = \frac{|x| - |x - |\alpha|| + |\alpha|}{2} \tag{2.3}$$

$$y = Q_k(y/|\alpha|) \cdot |\alpha| \tag{2.4}$$

$$y = Q_k(y/|\alpha|) \cdot |s| \tag{2.5}$$

En [21] utilizan centroides y clusters para cuantizar los pesos y datos de la red, tal como muestra la Figura 2.5. Esto consiste en dividir los parámetros en N posibles valores, asignando a cada peso el valor más cercano. La división de los centroides puede ser lineal o en base a la distribución de los parámetros, pero con esta última la red mantiene una precisión similar a la punto flotante. Luego, el flujo de cuantización aplica ajuste fino a los centroides, actualizando los pesos con el gradiente de reentrenamiento.

Adicionalmente, [21] aplica la codificación de Huffman [31] para comprimir los centroides ajustados. Este método consiste en representar con más bits los centroides con menos presencia en la red y con menos bits los con más frecuencia. Al aplicar esta codificación sobre VGG-16, los investigadores lograron reducir la cantidad de bits a 8 para las capas convolucionales y a 5 para las de clasificación. Luego, en el hardware, el sistema utiliza una LUT (del inglés *Look Up Table*, tabla de búsqueda) para decodificar los valores de Huffman a punto fijo de 16 bits.

2.3.2.2. Descomposición por valores singulares

La SVD (del inglés *Singular Value Decomposition*, descomposición por valores singulares) [32] es un método para factorizar matrices utilizando los vectores y valores propios de éstas. Esto consiste en descomponer una matriz A en una multiplicación de 3 matrices, tal como la ecuación (2.6) muestra. Aquí, A es descompuesta en U, una matriz ortogonal formada por las

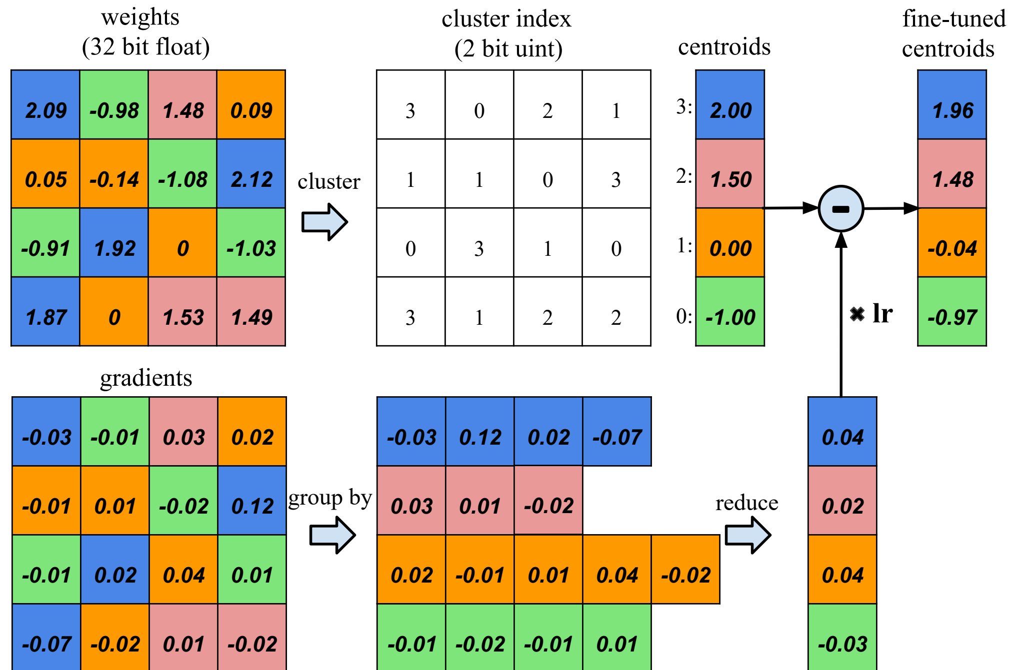


Fig. 2.5: Cuantización por centroides y clusters.
(Fuente: [21]).

columnas de $\frac{Av_x}{\sigma_x}$ con σ_x la raíz cuadrada del valor propio x de A^TA (valor singular de A) y v_x el vector propio de A^TA; en $\sum$, matriz diagonal de los valores singulares de A; y V, matriz de vectores propios de A^TA.

$$A = U \cdot \sum \cdot V^T \tag{2.6}$$

Este método de factorización es aplicado en [13] para reducir la cantidad de parámetros en las capas de clasificación. Además, ya que la cantidad de pesos de estas capas es reducida, las operaciones a realizar también disminuyen. Gracias a SVD la red VGG-16 es reducida de 140 millones a cerca de 50 millones de pesos, una tasa de compresión cercana a 2.8 veces el tamaño original, perdiendo sólo un 0.04 % de precisión.

2.3.2.3. Redes neuronales convolucionales hashing

CNN hashing [33] es un método que utiliza CNNs y hashing para codificar pares de imágenes según su similitud. El método está dividido en dos partes: (1) las capas convolucionales de la CNN son utilizadas para extraer características en las imágenes de entrada y (2) las capas clasificadoras son entrenadas para codificar las características extraídas según su similitud, mediante

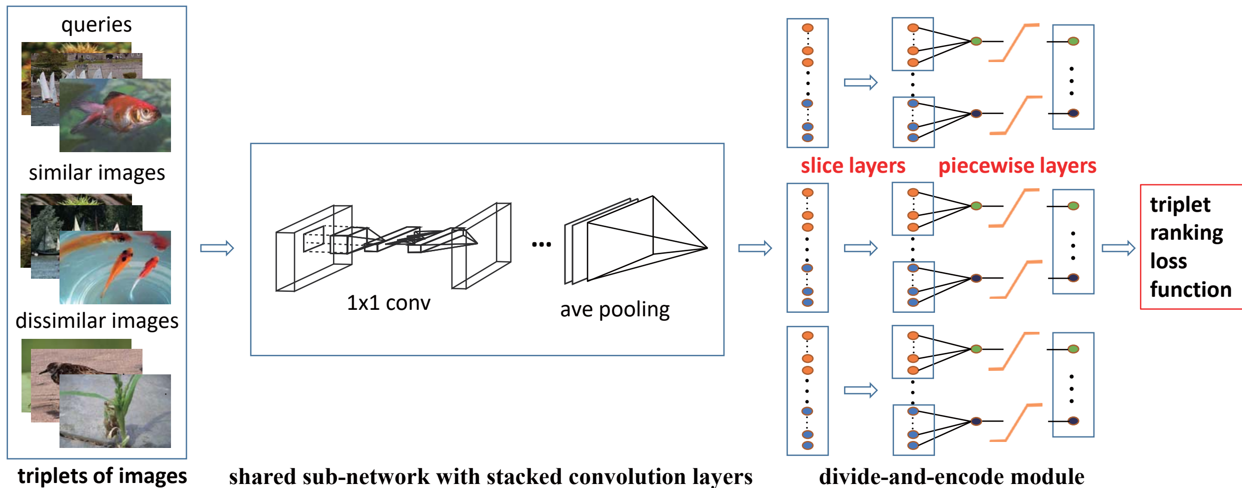


Fig. 2.6: Arquitectura red neuronal convolucional hashing [1].
(Fuente: [1]).

funciones hash.

Las funciones hash son funciones que pueden mapear un conjunto de datos en otros, tal como lo muestra la ecuación (2.7). Aquí, H es la función hash y U es el conjunto a mapear en el conjunto M. La idea de las funciones hash es que el nuevo conjunto M sea más pequeño que U, permitiendo una representación compacta. Debido a esto, cuando la red aprende el comportamiento de la función hash, el código generado para la imagen de entrada es de un menor tamaño, manteniendo las características de esta gracias a las capas convolucionales.

$$H : U \rightarrow M \tag{2.7}$$

En [1] los autores utilizan este método para generar códigos binarios que representan imágenes según su similitud. Tal como muestra la Figura 2.6, el algoritmo usa capas convolucionales 1x1 para extraer características de las imágenes. Además, utiliza un triple ranking de perdidas para encontrar la función hash H que asemeje los códigos cuando las imágenes sean similares y los aleje cuando sean diferentes.

Uno de los principales problemas de precisión que tienen las CNNs hashing es que al utilizar las funciones hash para codificar una imagen, éstas pasan del espacio euclidiano a Hamming, por lo que existen pérdidas de características. Por esto, en [2] utilizan la arquitectura mostrada en la Figura 2.7, donde usan funciones de perdidas que minimizan el error que introduce la transformación de espacio, permitiendo mantener la semántica o características de las imágenes en los códigos binarios. Además, para mejorar aún más la precisión, los autores utilizan una

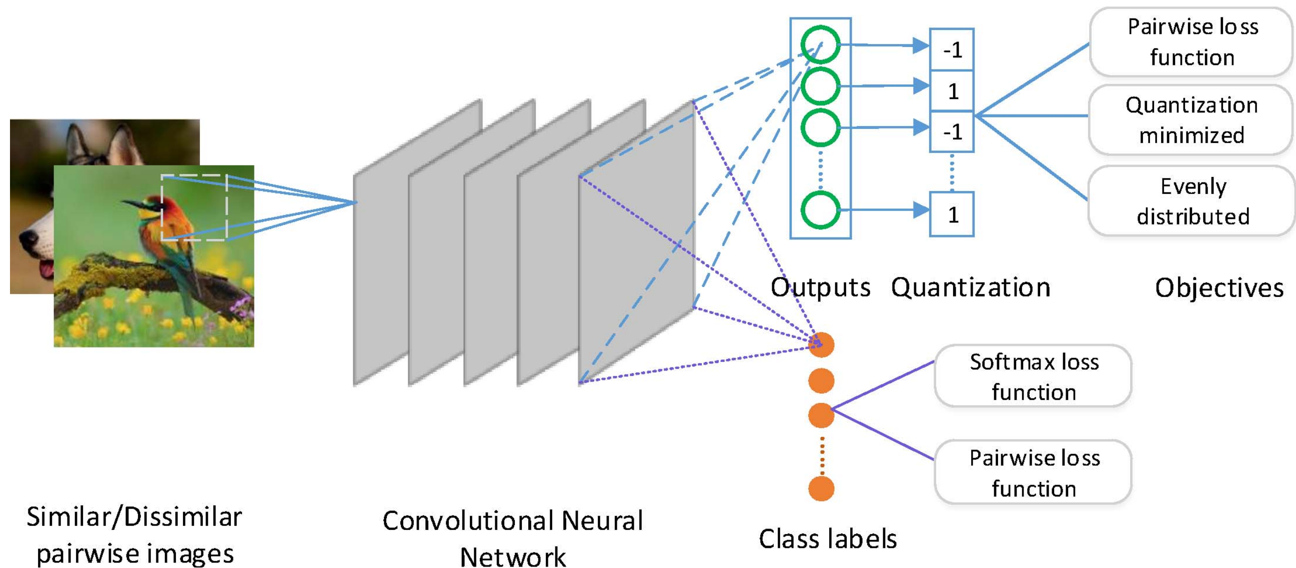


Fig. 2.7: Arquitectura red neuronal convolucional hashing [2].
(Fuente: [2]).

función de clasificación para identificar la clase de la imagen, añadiendo un factor extra para generar el código. Notar que esto es posible ya que aquí utilizan GoogLeNet para extraer características, red que tiene diferentes niveles de capas de clasificación, por lo que es posible extraer la clase antes de terminar de procesar la imagen.

2.3.2.4. Pruning

Pruning es una técnica utilizada para eliminar conexiones en las redes neuronales, dejando las más relevantes y así reducir el tamaño de la red, tal como muestra la Figura 2.8. Esta técnica consiste en que, en base a un modelo ya entrenado, el algoritmo ordena los pesos por capa, reemplazando por cero los que están bajo un umbral. Luego, como al eliminar conexiones la precisión es afectada, el algoritmo debe reentrenar la red para actualizar los pesos no ceros y así aumentar la precisión. Este proceso puede ser aplicado en varias iteraciones para mejorar la compresión de la red.

Además de reducir la cantidad de memoria necesaria para almacenar los pesos, la ausencia de una parte de estos también reduce la cantidad de operaciones por capa, acelerando el cómputo. Debido a esto, la arquitectura puede reducir el uso de recursos, disminuyendo a su vez la potencia consumida.

En [21] aplican la técnica de pruning sobre diferentes CNNs, obteniendo las tasas de reducción que muestra la Tabla 2.1. Por ejemplo, para el caso de VGG-16, la red es reducida 12

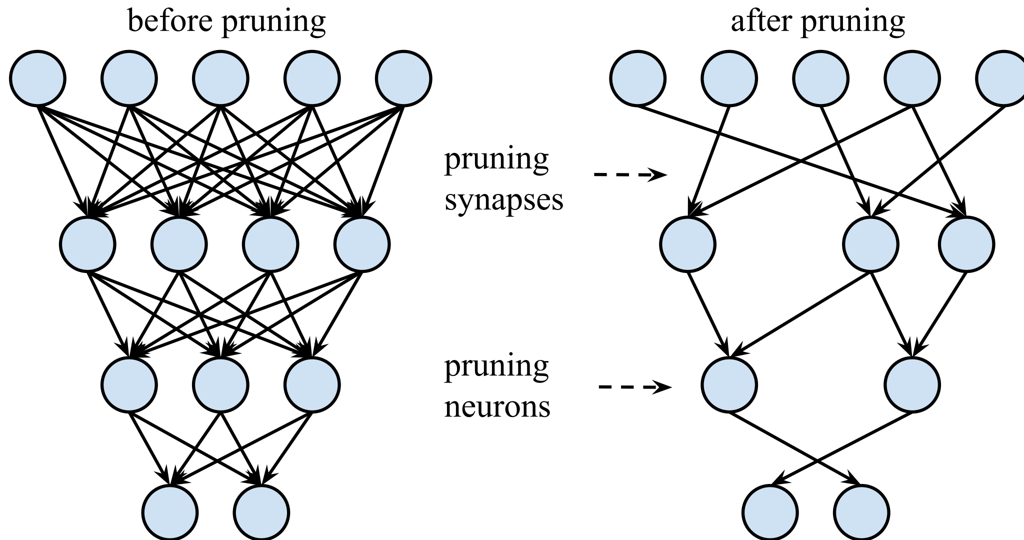


Fig. 2.8: Pruning en red neuronal.
(Fuente: [21]).

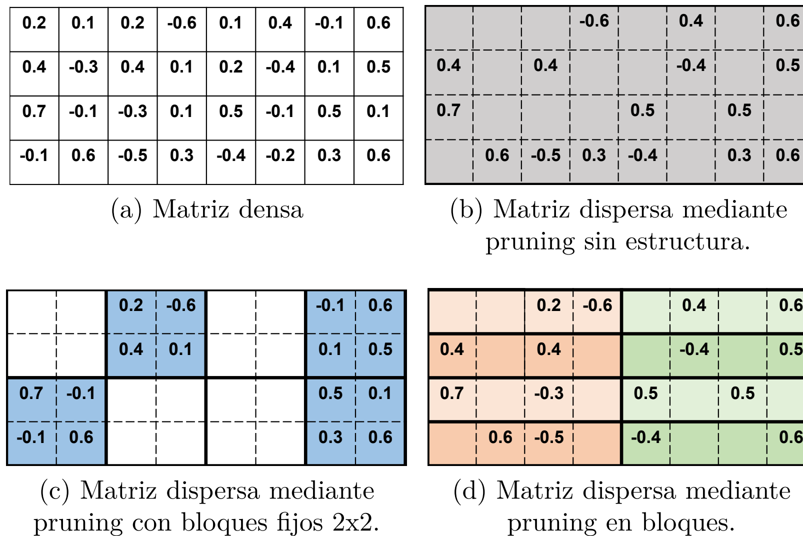
Modelo CNN	Factor de compresión	Reducción de operaciones
AlexNet	9.0	3.0
VGG-16	12.0	5.0
GoogLeNet	3.4	4.5
ResNet-50	3.4	6.3
SquezeNet	3.2	3.5

Tabla 2.1: Compresión de pruning en diferentes modelos de redes neuronales.
(Fuente: [21]).

veces su tamaño original, pasando de 138 millones a sólo 11.5 millones de pesos. Además, las operaciones en la red disminuyen a un quinto del total, sin reducir la precisión de la red debido al reentrenamiento.

Si bien pruning reduce la cantidad de pesos y operaciones, su implementación en hardware se vuelve más compleja. Esto se debe a que como hay pesos que no son considerados, la carga operacional de los PEs (del inglés *Processing Element*s, elementos de procesamiento) puede estar desbalanceada. Para solucionar esto, existen algunas técnicas para balancear la carga en cada PE del hardware, en las que podemos destacar pruning por bloques, tal como muestra la Figura 2.9.

En [21, 34] proponen un balanceo de carga basado en la subdivisión de las matrices de cada capa en bloques más pequeños, donde el pruning es aplicado a cada sub-matriz de forma independiente, lo que permite que cada bloque tenga la misma cantidad de elementos no ceros,



(a) Matriz densa

(b) Matriz dispersa mediante pruning sin estructura.

(c) Matriz dispersa mediante pruning con bloques fijos 2x2.

(d) Matriz dispersa mediante pruning en bloques.

Fig. 2.9: Comparación de metodos de pruning.
(Fuente: [34]).

balanceando la carga en los PEs. Una segunda alternativa a pruning por bloques es propuesta por [35]. Aquí los autores dividen la matriz en bloques de tamaño fijo, eliminando los bloques que en promedio tienen menor incidencia en la inferencia y manteniendo los que la afectan más.

En [34] compararon las tres alternativas a pruning (sin estructura, por bloques y en bloques fijos), donde utilizando una dispersión del 90 %, pruning sin estructura puede eliminar el total del 90 % de los pesos de la red sin afectar considerablemente la precisión. Del mismo modo, pruning por bloques es capaz de eliminar cerca del 85 % de los pesos menos influyentes en inferencia, pero manteniendo la carga balanceada en las unidades de procesamiento. Por otro lado, pruning en bloques fijos puede eliminar cerca del 30 % de los pesos sin afectar la precisión.

2.4. Aceleradores hardware

Diversos trabajos han implementado diferentes CNNs en hardware, tanto sobre FPGAs, GPUs y ASICs, los cuales son resumidos en la Tabla 2.2. En [19] diseñan una arquitectura implementada sobre un FPGA utilizando el método de *roofline* para maximizar el ancho de banda, utilizando técnicas como loop tiling y loop unrolling para acelerar la inferencia. En [13] utilizan el método SVD para reducir la red VGG-16. Sumado a esto, utilizan loop tiling para reutilizar datos de memoria y cuantización dinámica de punto fijo, disminuyendo aún más la cantidad de memoria utilizada para almacenar los pesos. En [21] diseñaron un arquitectura en ASIC. Ésta implementa los métodos de pruning para reducir la cantidad de parámetros en la

red, y cuantización basada en centroide y codificación de Huffman para reducir la cantidad de bits de los pesos en memoria.

En [23] implementan una arquitectura programable capaz de mapear diferentes CNNs en hardware. Para esto, el flujo de trabajo está dividido en (1) cuantizar la red por medio de cuantización dinámica de punto fijo, agregando una etapa de ajuste fino para mejorar la precisión, (2) compilar el grupo de instrucciones y pesos de la red, (3) traspasar archivos al hardware para inferir. En [36] utilizan pruning por bloques para reducir MobileNet y mapearla sobre una FPGA. A diferencia de otros trabajos, debido al pruning y la cuantización de pesos y datos, los mapas de la red pueden ser almacenados en la memoria interna del FPGA, utilizando memoria externa para leer y enviar las imágenes de entrada y salida respectivamente. Para esto, los autores diseñaron una arquitectura con tres memorias internas, dos para guardar los mapas de características de entrada y salida, y una para guardar los pesos de la red, utilizando loop tiling para acelerar el procesamiento del sistema.

En [37] implementan la red MobileNet V2 en un FPGA Arria 10 SoC logrando inferir imágenes a 266 fps. Esta velocidad de inferencia se debe a la gran cantidad de recursos disponibles en la FPGA utilizada, ya que pueden utilizar 4 núcleos de procesamiento en paralelo donde cada uno puede procesar 32 canales de la red a la vez. Sumado a esto, gracias a la cantidad de memoria disponible, los autores pueden almacenar todos los mapas de características en las memorias internas del dispositivo, evitando recurrir a dispositivos externos del FPGA para procesar la inferencia, como CPU o memoria externa. Aun así, es necesario probar el desempeño cuando disponemos de plataformas con recursos limitados. En [30] diseñan una CNN llamada DiracDeltaNet basada en ShuffleNet [38] que divide la extracción de características en ramas, aplicando diferentes filtros. DiracDeltaNet cambia algunos aspectos de ShuffleNet, especialmente reemplazando las convoluciones 3x3 por módulos llamados *shifters* que basan su funcionamiento en la concatenación de la información de las neuronas vecinas en la central, ahorrando recursos y tiempo de procesamiento. Además, este trabajo cuantiza los pesos y datos utilizando punto fijo con la función PACT$(\cdot)$ de la ecuación (2.3), junto con loop unrolling para acelerar la inferencia de resultados.

En [16] utilizan la plataforma embebida NVIDIA Jetson Xavier para implementar una versión reducida de MobileNet V2 de 12 clases, con el objetivo de clasificar objetos viales mediante drones de bajo consumo de potencia. Para esto, modifican la capa de clasificación de MobileNet V2 tradicional para que sólo reconozca las 12 categorías, para luego reentrenar la nueva red bajo estas condiciones. Implementada sobre la GPU, la red puede inferir una imagen en 15.38ms

	Zhang et al [19]	Qiu et al [13]	Han [21]	Guo et al [23]	Su et al [36]	Bai te al [37]	Yang et al [30]	Mohan et al [16]
Año	2015	2016	2017	2018	2018	2018	2019	2020
Hardware	FPGA	FPGA	ASIC	FPGA	FPGA	FPGA	FPGA	GPU
Modelo	VX485T	XC7Z045	45nm	XC7Z020	XCZU9EG	Arria 10	XCZU3EG	J. Xavier
Frec. (MHz)	100	150	800	214	150	133	250	ND
CNN	AlexNet	VGG-16	VGG-16	VGG-16	MobileNet	MobileNetV2	DiracDeltaNet	MobileNetV2
Reducción	NA	SVD	Pruning	NA	Pruning	NA	NA	NA
Cuantización	Flotante	Dinámica	C/H	Ajuste fino	Punto fijo	Punto fijo	PACT	Flotante
Bits	32	16	4 a 16	8	8-4	16	1-4	32
LUT	186251	182616	-	29867	139000	163506	24130	-
FF	205704	127653	-	35489	55000	ND	29867	-
BRAM	1024	486	-	86	1729	1844	170	-
DSP	2240	780	-	190	1452	1278	37	-
Potencia (W)	18.61	9.63	0.59	3.5	ND	ND	5.5	14.39
Des. (GOP/s)	61.62	136.97	102	84.3	91.2	170.6	47.09	ND
fps	ND	4.45	ND	2.75	127	266	58.7	65.02
Precisión top-1	ND	ND	68.83 %	67.72 %	64.6 %	71.8 %	70.1 %	ND
Precisión top-5	ND	86.66 %	89.09 %	88.06 %	84.5 %	91.0 %	88.2 %	ND

Tabla 2.2: Comparación de implementaciones de redes neuronales convolucionales en hardware dedicado.
NA: No aplicada, ND: No disponible, C/H: Centroide/Huffman.

consumiendo 14.39W de potencia.

2.5. Diseño de arquitectura con HLS

HLS es un proceso de diseño automatizado que interpreta y convierte un algoritmo en una arquitectura hardware digital, con el objetivo de facilitar y acelerar el diseño de hardware en comparación con la descripción directa del RTL. Para esto, el diseñador entrega a la herramienta un código en lenguaje de alto nivel, como C o C++, que describe el algoritmo. A partir de este código, la herramienta analiza y convierte la descripción del algoritmo a un diseño de RTL en un lenguaje de descripción de hardware, como Verilog o VHDL. Aun así, para optimizar el desempeño de la arquitectura, el diseñador debe agregar directivas que especifiquen como debe ser sintetizado el hardware, como dominios de reloj, pipeline, loop unrolling, latencias, especificar los tipos de recursos a utilizar, etc.

2.5.1. Beneficios y limitaciones de HLS

La principal ventaja de diseñar hardware mediante HLS en vez de RTL es que el tiempo de diseño disminuye radicalmente, pudiendo pasar de meses de trabajo a sólo semanas, ya que algoritmos complejos son difíciles de ser descritos mediante lenguajes de descripción de hardware [39]. Otra ventaja es que como el algoritmo es descrito mediante código de alto nivel, sólo es necesario resintetizar para utilizar el hardware en FPGAs con arquitecturas diferentes, mientras que en RTL, típicamente hay que reescribir el código [40]. Otro punto a favor es la fácil implementación con arquitecturas heterogéneas, que combinan FPGA y CPU. Esto se debe a que en HLS sólo basta con agregar directivas para definir los protocolos de comunicación, a diferencia de RTL, que es necesario describirlas mediante código [40].

Sin embargo, HLS tiene desventajas en ciertas aplicaciones. Debido a que en códigos de alto nivel se obvian las rutas de control, los diseñaron no tienen un conocimiento total de como se están sintetizando dichas rutas, por lo que en aplicaciones donde es necesario conocer en detalle la arquitectura de control, es mejor utilizar directamente una descripción con RTL. Otra limitación es que HLS tiene problemas cuando se utilizan punteros y recursividad en el código de alto nivel [40]. En software, los punteros representan la dirección de memoria de una variable. Sin embargo, en hardware dedicado, las memorias internas se distribuye en bloques pequeños o registros que son independientes entre sí y tienen direcciones diferentes, por lo que es costoso y poco transparente para el diseñador emular el comportamiento de los punteros con HLS. En el caso de recursividad, en software se utiliza una pila para almacenar las variables de cada llamada a función. Sin embargo, en las FPGAs no existe una pila de almacenamiento, ya que cada variable es almacenada en registros que son definidos antes de sintetizar el hardware, por lo que el diseñador debe especificar una cantidad máxima de llamadas a función antes de convertir el código a RTL, lo que limita la recursividad del proceso.

Tal como menciona [40], el procesamiento de imágenes es un tipo de aplicación que presenta beneficios al sintetizar el hardware mediante HLS. Esto se debe a que las herramientas de HLS pueden sintetizar eficientemente los buffers de línea y las ventanas de registros para procesar las imágenes como un stream de datos, además de disminuir los tiempos de diseño, ya que el diseño en RTL de estos algoritmos no es trivial. Por esto, trabajos previos que han implementado CNNs en hardware dedicado como [19, 36] han utilizado HLS para sintetizar sus arquitecturas.

2.6. Discusión

La literatura estudiada nos demuestra que ya se han diseñado e implementado diferentes CNNs en diferentes plataformas hardware, logrando un menor consumo de potencia en dispositivos embebidos. Dentro de estos trabajos podemos dividir las implementaciones hardware en dos grupo. Primero, las arquitecturas para redes de gran tamaño [23, 13, 19, 21], como AlexNet y VGG-16, que en general, debido a la gran cantidad de parámetros (superando los 50 millones) y todas las operaciones a realizar, los tiempos de inferencia son bajos además de utilizar una mayor cantidad de recursos, sobre todo en memorias internas. El segundo grupo está compuesto por las arquitecturas que trabajan con redes más pequeñas [36, 30], MobileNet y DiracDeltaNet, que debido a la poca cantidad de parámetros, operaciones y capas, incluso evitando las capas de clasificación, los tiempos de inferencia mejoran.

Esto se debe a que el tamaño y complejidad de la red repercute directamente en el desempeño que podremos alcanzar con la implementación sobre hardware, por lo que la elección de la red es fundamental para lograr buenos resultados. MobileNet V2 es una CNN que tiene ciertas ventajas sobre otros modelos estudiados. Específicamente, MobileNet V2 utiliza convoluciones *expansion* y *projection*, que expanden y reducen la cantidad de canales de la red aplicando filtros 1x1, mientras que por medio de convoluciones *depthwise*, la red sólo aplica un filtro 3x3 por canal generado, disminuyendo la cantidad de parámetros para extraer características. Otro punto a favor de la red es el uso de capas residuales, que permiten aplicar más capas al modelo sin afectar los resultados, incrementando la precisión respecto a MobileNet V1.

Para implementar las CNNs en hardware dedicado existen dos inconvenientes. Estos son el cómputo necesario para trabajar en las capas convolucionales de la red y la cantidad de pesos especialmente en las capas de clasificación, por lo que en redes más grandes, estos problemas influyen más en el desempeño.

Para abordar estos inconvenientes, diferentes trabajos proponen métodos como loop tiling, loop unrolling y compresión de redes. Respecto a las que muestran un mejor desempeño según la literatura, loop tiling y loop unrolling son buenas alternativas para implementar CNNs en hardware dedicado, ya que permiten disminuir los accesos a memoria. Cuantización de datos también es una técnica esencial en la implementación en hardware dedicado, ya que al utilizar punto fijo con una cantidad reducida de bits, las redes usan menos memoria y las operaciones, principalmente las sumas de productos en las convoluciones, son más sencillas de diseñar.

Si bien las funciones PACT(·) de [30] son las que generan parámetros con el menor número de bits punto fijo, la obtención del hiperparámetro $|\alpha|$ mediante entrenamiento es compleja, ya que es necesario medir la precisión de la red en muchas iteraciones, complicando la obtención del valor correcto. Por esto, cuantización dinámica y por centroide parecen ser las mejores alternativas para cuantizar pesos y datos, las que combinadas con algoritmos de codificación como es realizado en [21], permiten reducir la cantidad de bits de los pesos en memoria, recuperando el número de bits mediante la decodificación utilizando LUTs.

Un punto importante a considerar en la implementación de una CNN en hardware dedicado es reducir al máximo la cantidad de datos que contiene la red. Para esto, la reducción de CNNs es una herramienta fundamental. Una técnica estudiada fue SVD, pero tal como menciona [21], la descomposición de las matrices no puede reducir tanto la red sin pérdidas cuantiosas de precisión, y la ausencia de reentrenamiento en este método hace imposible compensarlo. CNN hashing reduce las capas clasificadoras de la red reemplazándolas por hashing, pero existen dos problemas con esta técnica que no la hacen una buena candidata para comprimir redes para hardware dedicado. La primera es que las técnicas de hashing en CNN no reducen las capas convolucionales de la red, ya que sólo están enfocadas en las capas de clasificación, por lo que si las convoluciones son de gran tamaño, la red seguirá siendo compleja. La segunda es que si bien la red es reducida al reemplazar las capas de clasificación, la CNN aún sigue siendo grande, ya que las funciones hash son aprendidas por la red por medio de pesos, los cuales son una gran cantidad al conectar todas las neuronas entre sí y tener un peso por cada bit a codificar.

Una técnica que puede comprimir cuantiosamente la red sin pérdidas grandes de precisión es pruning. La ventaja de esta técnica respecto a las anteriores es que una vez realizada la reducción de pesos, el reentrenamiento de la red compensa la precisión actualizando los pesos no ceros. Gracias a esto, según lo planteado en [21], con esta técnica es posible reducir las redes casi hasta el 10 % de su tamaño. Aun así, el principal problema de pruning es la no simetría de las matrices generadas, por lo que trabajos como [21, 34] recomiendan utilizar pruning por bloques que mantienen la misma cantidad de elementos no ceros en cada PE, balanceando la carga y así acelerando la ejecución en hardware.

En conclusión, modelos de CNNs de baja complejidad y de pocos parámetros, sumado a técnicas de pruning por bloques y cuantización de datos, facilitan la implementación en hardware dedicado de este tipo de redes neuronales. Además, técnicas como loop tiling y loop unrolling disminuyen los accesos a memoria, lo que repercute favorablemente en el uso de recursos, potencia consumida y en la aceleración de inferencia. Por esto, estas técnicas son las más propicias

para implementar la red MobileNet V2 en hardware dedicado. Además, para disminuir los tiempos de diseño y a que HLS entrega más beneficios que desventajas en la síntesis de hardware para procesamiento de imágenes, utilizamos esta herramienta para diseñar la arquitectura de inferencia de la red neuronal.

Capítulo 3

Arquitectura de MobileNet V2

En este capítulo mostramos la composición de la arquitectura de la red MobileNet V2, detallando tanto el funcionamiento de las capas convolucionales como las capas de clasificación, además de la distribución de parámetros en estos dos tipos de capas de la red.

3.1. MobileNet V2

Como ya mencionamos en el Capítulo 2, la ventaja que entrega MobileNet V2 para ser implementada en hardware dedicado es su poca cantidad de pesos y su alta precisión en comparación a otras redes sobre ImageNet. En específico, esta red tiene cerca de 3.4 millones de pesos entre las capas de convolución y clasificación, alcanzando una precisión top-1 y top-5 de 71.8 % y 91.0 % sobre ImageNet respectivamente.

3.2. Capas convolucionales

Las capas convolucionales de MobileNet V2 están compuestas por etapas de convoluciones, normalización y activación. Estas etapas están agrupadas en módulos denominados *bottleneck*.

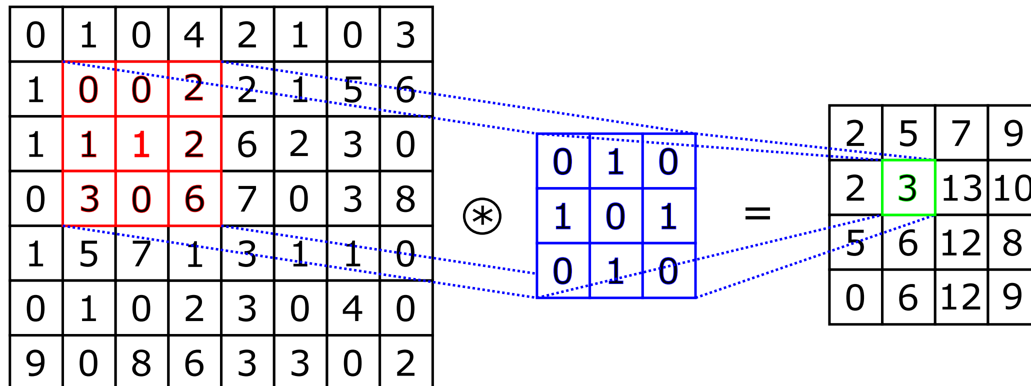


Fig. 3.1: Extracción de características en imágenes mediante convolución con paso 2.

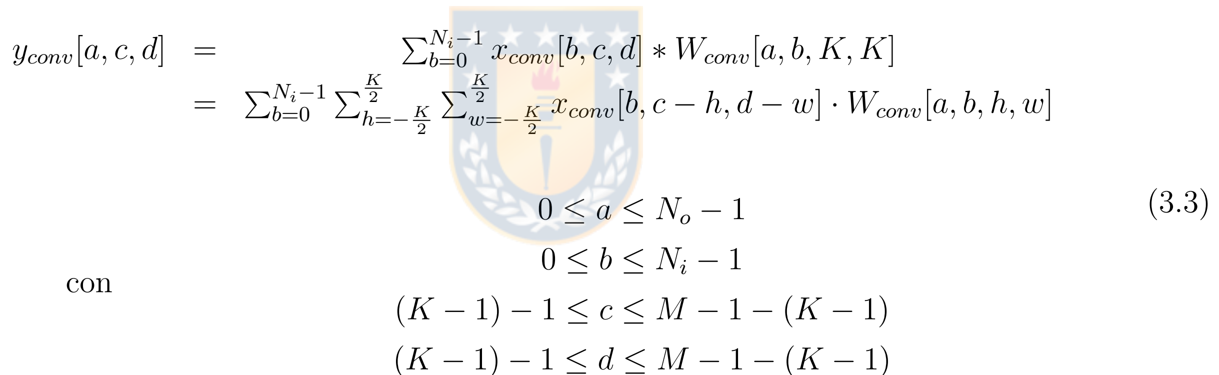
3.2.1. Convoluciones

La operación de convolución es una función matemática descrita en la ecuación (3.1) que transforma dos funciones f y g en una tercera que representa la superposición de f y la traslación de g. Las CNNs utilizan esta operación para extraer características de las imágenes, tales como bordes o formas, por lo que como procesan variables en dos dimensiones, es necesario realizar convoluciones en 2D, cuya fórmula es mostrada en la ecuación (3.2), donde x corresponde a la imagen de entrada, z al filtro convolucional, y a la imagen de salida o mapa de característica, H y W al alto y ancho de la imagen respectivamente y K_i y K_j al alto y ancho del filtro respectivamente. La Figura 3.1 ilustra este proceso. Aquí, un filtro aplica la operación de convolución a una vecindad de tamaño $K = 3$, con K el tamaño del filtro, obteniendo como resultado un píxel de salida del mapa de características. Luego, el filtro se desplaza por todos los píxeles de la imagen hasta completar el mapa de características, pudiendo repetir este proceso si la imagen tiene más canales. Cabe destacar que la convolución también puede ser realizada por pasos, es decir, ir saltándose filas y/o columnas para así reducir el tamaño de las imágenes de salida. En el caso de la Figura 3.1, el paso de la convolución es dos, ya que el filtro se salta una fila o una columna para aplicar la convolución.

$$f[n] * g[n] = \sum_{m} f[m]g[n-m] \tag{3.1}$$

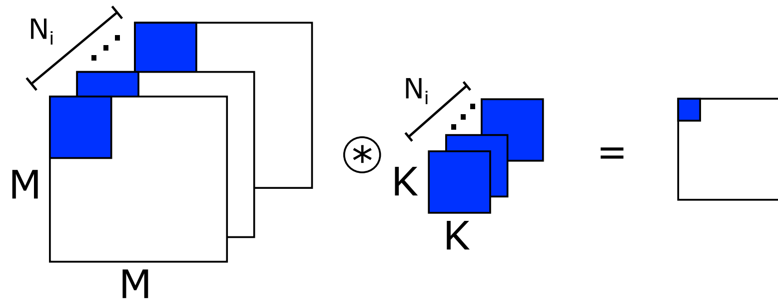
$$y[c,d] = \sum_{k=-\frac{K_i}{2}}^{\frac{K_i}{2}} \sum_{l=-\frac{K_j}{2}}^{\frac{K_j}{2}} x[c-k, d-l] \cdot z[k,l] \quad \text{con} \quad \begin{matrix} (K_i - 1) - 1 \leq c \leq H - 1 - (K_i - 1) \\ (K_j - 1) - 1 \leq d \leq W - 1 - (K_j - 1) \end{matrix} \tag{3.2}$$

MobileNet V2 utiliza 4 tipos de convoluciones, las cuales son mostradas en la Figura 3.2. La convolución clásica o normal mostrada en la Figura 3.2a es la convolución utilizada típicamente en otras CNNs, como AlexNet, VGG, GoogLeNet, etc, cuya formula es mostrada en la ecuación (3.3), donde x_{conv} e y_{conv} son los mapas de características de entrada y salida respectivamente, W_{conv} son los filtros convolucionales, N_o y N_i corresponden la cantidad de canales o profundidad de salida y entrada de los mapas de características, M es el tamaño de los mapas y K el tamaño de los filtros, que en el caso de MobileNet V2 $K = 3$. Este tipo de convolución consiste en realizar la convolución entre una cantidad N_i de filtros de tamaño $K \times K$ con los N_i canales del mapa de características de entrada, obteniendo un canal del mapa de características de salida al sumar todos los resultados anteriores. Luego, este proceso es repetido N_o veces hasta completar los N_o canales del mapa de características de salida. En MobileNet V2, esta convolución es utilizada en la primera capa para extraer las características base de la imagen de entrada.

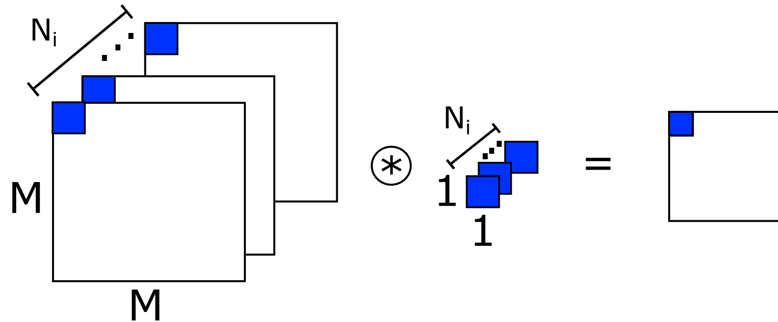


$$
\begin{aligned}
y_{conv}[a,c,d] &= \sum_{b=0}^{N_i-1} x_{conv}[b,c,d] * W_{conv}[a,b,K,K] \\
&= \sum_{b=0}^{N_i-1} \sum_{h=-\frac{K}{2}}^{\frac{K}{2}} \sum_{w=-\frac{K}{2}}^{\frac{K}{2}} x_{conv}[b,c-h,d-w] \cdot W_{conv}[a,b,h,w] \\
\text{con} \quad & 0 \le a \le N_o - 1 \\
& 0 \le b \le N_i - 1 \\
& (K-1) - 1 \le c \le M - 1 - (K-1) \\
& (K-1) - 1 \le d \le M - 1 - (K-1)
\end{aligned}
\tag{3.3}
$$

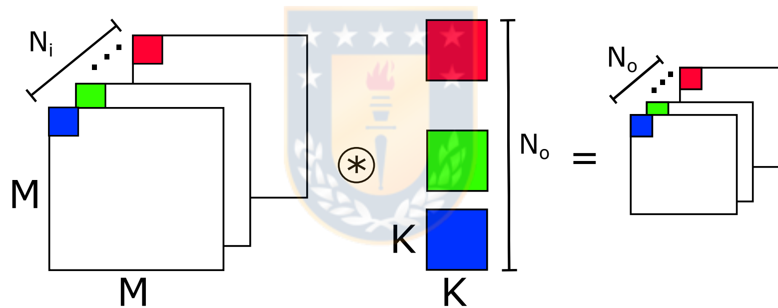
Las convoluciones *expansion* y *projection* mostradas en la figura Figura 3.2b son muy similares a las convoluciones clásicas, pero a diferencia de estas, realizan la convolución por filtros de tamaño 1x1 en vez de KxK, convirtiendo la operación de convolución en sólo multiplicar la imagen por un escalar y luego sumar los resultados de cada canal de entrada para obtener un canal de salida, proceso detallado en la ecuación (3.4). La diferencia entre convoluciones *expansion* y *projection* es que en las primeras, la cantidad de canales de salida N_o aumentan respecto a la cantidad de canales de entrada N_i, mientras que en las segundas, pasa lo contrario.



(a) Convolución normal.



(b) Convolución *expansion* y *projection*.



(c) Convolución *depthwise*.

Fig. 3.2: Tipos de convolución en MobileNet V2.

$$y_{conv}[a,c,d] = \sum_{b=0}^{N_i-1} x_{conv}[b,c,d] \cdot W_{conv}[a,b,1,1]$$

$$\text{con} \quad \begin{array}{c} 0 \leq a \leq N_o - 1 \\ 0 \leq b \leq N_i - 1 \\ 0 \leq c \leq M - 1 \\ 0 \leq d \leq M - 1 \end{array} \tag{3.4}$$

Las convoluciones *depthwise* mostradas en la Figura 3.2c y en la ecuación (3.5) consisten en realizar una sola convolución de tamaño $K \times K = 3 \times 3$ a cada canal de entrada para obtener un canal de salida, permitiendo disminuir la computación en comparación con las convoluciones normales, en las cuales es necesario realizar N_i convoluciones para obtener un canal de salida.

Cabe destacar, que en este tipo de convoluciones, el número de canales de entrada N_i es igual al número de canales de salida N_o. En MobileNet V2, las capas *depthwise* son usadas para extraer las características de cada mapa.

$$\begin{aligned} y_{conv}[a,c,d] &= x_{conv}[b,c,d] * W_{conv}[a,b,K,K] \\ &= \sum_{h=-\frac{K}{2}}^{\frac{K}{2}} \sum_{w=-\frac{K}{2}}^{\frac{K}{2}} x_{conv}[b,c-h,d-w] \cdot W_{conv}[a,b,h,w] \end{aligned}$$

$$\text{con} \quad \begin{aligned} 0 \leq a \leq N_o - 1 \\ 0 \leq b \leq N_i - 1 \\ (K-1)-1 \leq c \leq M-1-(K-1) \\ (K-1)-1 \leq d \leq M-1-(K-1) \end{aligned} \tag{3.5}$$

3.2.2. Normalización

La normalización de *batch* [41] es una técnica utilizada para mejorar la velocidad de entrenamiento, desempeño y estabilidad de las redes neuronales. En el caso de MobileNet V2, esta es utilizada sobre cada mapa de características de la red para evitar que los valores se estos se disparen y así estabilizar la red. Para aplicar la normalización 2D, se utiliza la ecuación (3.6), donde y_{conv} corresponde a la entrada, la cual es la salida de la etapa de convolución, y_{norm} a la salida de la etapa de normalización, $\mathrm{E}[y_{conv}]$ a la media de y_{conv}, $\mathrm{Var}[y_{conv}]$ a la varianza de y_{conv}, γ y β a parámetros multiplicativos y aditivos respectivamente y ϵ a un coeficiente de estabilidad para evitar que la división no esté definida. Cabe destacar que γ y ϵ son calculados en el proceso de entrenamiento, ϵ es típicamente utilizado como 0.00001 y si bien $\mathrm{E}[y_{conv}]$ y $\mathrm{Var}[y_{conv}]$ pueden ser calculadas mediante la entrada en el proceso de inferencia, sus valores son calculados en entrenamiento utilizando todo el grupo de imágenes para sólo ser aplicados en inferencia mediante dos parámetros fijos.

$$y_{norm} = \frac{y_{conv} - \mathrm{E}[y_{conv}]}{\sqrt{\mathrm{Var}[y_{conv}] + \epsilon}} \cdot \gamma + \beta \tag{3.6}$$

3.2.3. Activación

Si bien todos los pesos de la red tienen impacto en el resultado, algunas convoluciones entregan valores de píxeles que son poco relevantes. Para esto, las CNNs, al igual que las neuronas de visión de los seres vivos, inhiben y excitan ciertas señales, por medio de un proceso denominado activación. Esto consiste en eliminar valores de los mapas de características que son poco relevantes y dejar pasar los demás. Para esto, las CNNs típicamente utilizan la función rectificador conocida como ReLU (del inglés *Rectified Linear unit*, unidad lineal rectificada), mostrada en la ecuación (3.7), aproximando a cero los pesos negativos y manteniendo iguales los demás.

$$\mathrm{ReLU}(y_{norm}) = \max(0, y_{norm}) \tag{3.7}$$

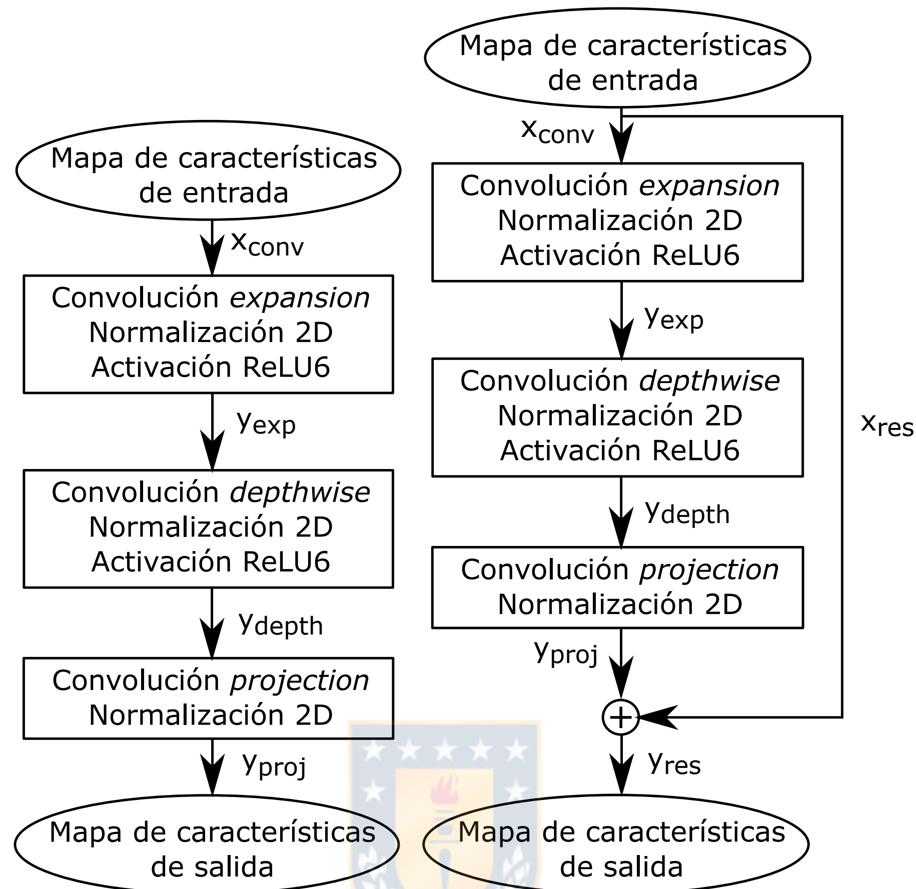
En el caso de MobileNet V2, la red utiliza la función ReLU6, que a diferencia de la función ReLU tradicional, ReLU6 trunca a 6 los valores que son mayores a éste, tal cual como se muestra en la ecuación (3.8). Esto se realiza para evitar que los mapas de características disparen su valores, dándole estabilidad a la red y facilitando la aplicación de punto fijo.

$$\mathrm{ReLU6}(y_{norm}) = \min(\max(0, y_{norm}), 6) \tag{3.8}$$

3.2.4. Módulos *bottleneck*

MobileNet V2 combina la etapa de convolución, normalización y activación en módulos llamados *bottleneck* formando una capa de la red. Estos módulos buscan aumentar la cantidad de canales para así usar menos pesos y operaciones en la extracción de características mediante las convoluciones *depthwise*, para finalmente reducir la cantidad de canales a su número original. Además, algunas capas utilizan residuales para mejorar la precisión de la red.

La arquitectura interna de estos módulos es mostrada en la Figura 3.3. Los módulos *bottleneck* reciben como entrada el mapa de características x_{conv} obtenido por la capa anterior, al cual aplican la convolución *expansion* para aumentar el número de canales, normalización 2D y activación ReLU6 obteniendo el mapa y_{exp}. Una vez aumentada la cantidad de canales del mapa de entrada, los módulos utilizan convoluciones *depthwise* para extraer las característi-



(a) Módulo *bottleneck* sin residual. (b) Módulo *bottleneck* con residual.

Fig. 3.3: Módulos *bottleneck*.

cas, seguidos de la etapa de normalización 2D y activación ReLU6, obteniendo como salida el mapa de características y_{depth}. Finalmente, el módulo reduce la cantidad de canales mediante convoluciones *projection* y normalización 2D, obteniendo y_{proj}. Además, tal como muestra la Figura 3.3b, algunos *bottleneck* utilizan el concepto de residuales de ResNet sumando el mapa de entrada con la salida de la etapa de convoluciones *projection*, obteniendo como salida del módulo y_{res}. Notar que los mapas de residuales serán denotados como x_{res}, tal como muestra la Figura 3.3.

Las ecuaciones 3.9 y (3.10) muestran el factor de reducción de las operaciones y pesos entre los módulos *bottleneck* y las convoluciones clásicas. Para el caso del número de operaciones, como los módulos *bottleneck* combinan las convoluciones *expansión/projection* y *depthwise*, que utilizan $N_o \times N_i \times M \times M$ y $N_i \times M \times M \times K \times K$ operaciones respectivamente, la cantidad de operaciones en estos módulos es de $2 \times N_o \times N_i \times M \times M + N_i \times M \times M \times K \times K$. Por su parte, las convoluciones clásicas utilizan $N_i \times N_o \times M \times M \times K \times K$ operaciones. Bajo el mismo concepto anterior, la cantidad de pesos en los módulos *bottleneck* y las convoluciones

clásicas es de $2 \times N_o \times N_i + N_i \times K \times K$ y $N_i \times N_o \times K \times K$ respectivamente. Al calcular el factor de reducción para las operaciones y pesos entre ambos tipos de capa, obtenemos $\frac{2}{K^2} + \frac{1}{N_o}$ para ambos casos, es decir, mientras $K > \sqrt{2}$ y $N_o > 1$, los módulos *bottleneck* utilizan menos operaciones y parámetros que las convoluciones clásicas. Como en MobileNet V2 $K = 3$ y N_o siempre es mayor a uno, la agrupación de las convoluciones *expansion/projection* y *depthwise* es favorable para reducir pesos y operaciones.

$$F_o = \frac{O_{bottleneck}}{O_{clásicas}} = \frac{2 \times N_o \times N_i \times M \times M + N_i \times M \times M \times K \times K}{N_i \times N_o \times M \times M \times K \times K} = \frac{2}{K^2} + \frac{1}{N_o} \tag{3.9}$$

$$F_w = \frac{W_{bottleneck}}{W_{clásicas}} = \frac{2 \times N_o \times N_i + N_i \times K \times K}{N_i \times N_o \times K \times K} = \frac{2}{K^2} + \frac{1}{N_o} \tag{3.10}$$

3.3. Capas de clasificación

Las capas de clasificación de MobileNet V2 están compuestas por una etapa de aplanamiento de mapas, una etapa de componentes totalmente conectados y una función clasificadora.

3.3.1. Aplanamiento

La etapa de aplanamiento se encarga de convertir los mapas de características a un vector de datos. Esto se realiza como etapa previa a la etapa de componentes totalmente conectados, ya que en esta se determina a que clase pertenece la imagen de entrada en función de las características extraídas por las etapas de convolución, proceso realizado al multiplicar una matriz de pesos por un vector datos que corresponden a la información comprimida de cada canal del mapa de características de la última capa convolucional de la red. En el caso de MobileNet V2, la red utiliza un promediador denominado *average pooling*, detallado en la ecuación (3.11), donde y_{avg} es el vector de datos, y_{layer} es el mapa de características obtenido por la última capa convolucional y N_o con M corresponden a la profundidad y tamaño de y_{layer} respectivamente. Esta operación consiste en promediar los píxeles de cada canal del mapa de características para así obtener una componente del vector de datos, formando un arreglo unidimensional del mismo tamaño que la cantidad de canales del último mapa de características.

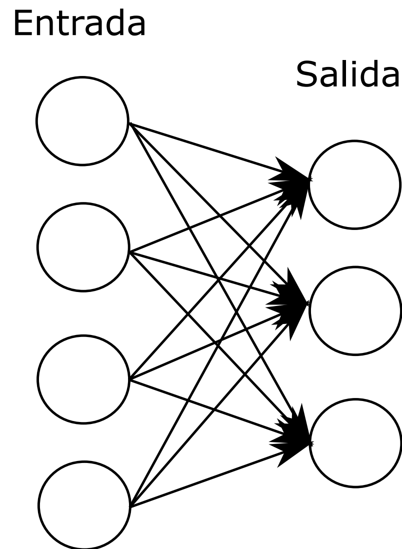


Fig. 3.4: Modelo de componentes totalmente conectados.

$$y_{avg}[a] = \frac{\sum_{c=0}^{M-1} \sum_{d=0}^{M-1} y_{layer}[a,c,d]}{M \cdot M} \quad \text{con} \quad 0 \leq a \leq N_o - 1 \tag{3.11}$$

3.3.2. Componentes totalmente conectados

Las CNNs, incluida MobileNet V2, utilizan el modelo de componentes totalmente conectados o FC (del inglés *Fully Connected Layer*, capa completamente conectada) para determinar la clase a la cual pertenecen las características extraídas de la imagen. Este modelo esta basado en el perceptrón multicapa [42], el cual permite resolver problemas donde las entradas no son lineamente separables, como es el caso de las CNNs.

La Figura 3.4 muestra el funcionamiento de este modelo de clasificación. Aquí, cada entrada a la capa se asocia linealmente a las demás entradas para obtener cada salida, formando una conexión total entre todas las entradas y salida del sistema. Esto es descrito matemáticamente en la ecuación (3.12), donde las N_{class} componentes de salida de la capa de componentes conectados y_{fc}, con N_{class} el número de clases que puede detectar la red, son calculadas como una combinación lineal entre cada entrada y_{avg} y la matriz de peso W_{fc}, además de sumar una constante b_{fc}.

$$y_{fc}[i] = b_{fc}[j] + \sum_{j=0}^{N_o-1} W_{fc}[i,j] \cdot y_{avg}[j] \quad \text{con} \quad 0 \leq i \leq N_{class} \tag{3.12}$$

3.3.3. Función clasificadora

Si bien los componentes conectados ya entregan valores de que clase es la que domina la imagen de entrada, es complicado apreciar cual es la probabilidad exacta que tiene cada clase de pertenecer a la imagen. Para esto, las CNNs utilizan funciones clasificadores que convierten los valores entregados por las capas de clasificación en un valor probabilístico entre 0 y 1.

MobileNet V2 utiliza la función SoftMax que normaliza un vector de tamaño K en el rango de $[0,1]$. La ecuación (3.13) muestra el funcionamiento de la función SoftMax. Aquí, la componente i-ésima del vector de salida y es calculado mediante la exponencial de la i-ésima componente del vector de entrada x menos el valor máximo de dicho arreglo, cuyo resultado es dividido por la sumatoria las exponenciales de cada elemento de x. Una vez obtenido el vector de salida, la red ya es capaz de entregar la probabilidad de que la imagen de entrada pertenezca a cada clase de la base de datos, completando el proceso de inferencia.

$$y_{softmax}[i] = \frac{e^{y_{fc}[i]-\max(y_{fc})}}{\sum_{j=1}^{N_{class}} e^{y_{fc}[j]}} \quad \text{con} \qquad 0 \leq i \leq N_{class} \tag{3.13}$$

3.4. Organización de las capas

Tal como hemos mencionado en este capítulo, MobileNet V2 esta formada por 9 capas convolucionales, destacando una capa convolucional normal, una capa de convolución *expansion* y 7 módulos *bottleneck*; además de una capa de aplanamiento, una capa de clasificación de componentes totalmente conectados y una función clasificadora SoftMax. La tabla 3.1 muestra el detalle de todas las capas de MobileNet V2. Cabe destacar que cuando el paso es dos en los módulos *bottleneck*, sólo se aplica en la primera repetición. Del mismo modo, cuando hay residual en módulos *bottleneck*, no se aplica en la primera repetición.

Entrada ($MxMxN_i$)	Salida ($MxMxN_o$)	Capa	Número de repeticiones	Factor de expansión	Paso	Residual	Miles de parámetros
224x224x3	112x112x32	Convolución clásica	1	-	2	No	0.8
112x112x32	112x112x16	Módulo *bottleneck*	1	1	1	No	0.7
112x112x16	56x56x24	Módulo *bottleneck*	2	6	2	Si	12.9
56x56x24	28x28x32	Módulo *bottleneck*	3	6	2	Si	37.3
28x28x32	14x14x64	Módulo *bottleneck*	4	6	2	Si	177.9
14x14x64	14x14x96	Módulo *bottleneck*	3	6	1	Si	296.0
14x14x96	7x7x160	Módulo *bottleneck*	3	6	2	Si	784.2
7x7x160	7x7x320	Módulo *bottleneck*	1	6	1	No	469.4
7x7x320	7x7x1280	Convolución *expansion*	1	-	1	No	409.6
7x7x1280	1x1x1280	Aplanamiento	1	-	-	-	0.0
1x1x1280	1x1x1000	Componentes conectados	1	-	-	-	1280.0
1x1x1000	1x1x1000	SoftMax	1	-	-	-	0.0

Tabla 3.1: Organización de capas en MobileNet V2.

Capítulo 4

Reducción de complejidad en MobileNet V2

Este capítulo muestra y detalla la forma en que aplicamos los métodos descritos en el Capítulo 2, además de simplificaciones al modelo de MobileNet V2 descrito en el Capítulo 3, con el fin de reducir la cantidad de operaciones, parámetros y bits de la CNN para ser implementada en hardware. Específicamente, aquí detallamos la fusión de la etapa de normalización 2D con las convoluciones en las capas convolucionales y la aplicación de las técnicas de loop tiling, pruning para eliminar pesos y cuantización de bits.

4.1. Fusión de normalización 2D en convoluciones

Tal como mencionamos en el Capítulo 3 de este trabajo, la red MobileNet V2 utiliza normalización 2D para estabilizar las activaciones y mejorar el entrenamiento. Sin embargo, al momento de inferir, esta etapa es un problema ya que agrega 5 operaciones, lo que aumenta en un mínimo de 5 ciclos la obtención de resultados. Esto se ve agravado ya que dos de estas operaciones son una división y una raíz cuadrada, cálculos que son complejos de implementar en hardware, aumentando la cantidad de recursos utilizados y los tiempos de procesamiento.

Por esto, según lo detallado en [43], fusionamos la etapa de normalización 2D en el cálculo de las convoluciones de la red. Esto consiste en mezclar los parámetros $\mathrm{E}[x]$, $\mathrm{Var}[x]$, γ y β con los pesos de cada convolución de la red, permitiendo reducir la complejidad de cálculos, parámetros y ciclos de operación. Para realizar esto, reescribimos la ecuación (3.6) en la ecuación (4.1),

donde se separaron los parámetros que multiplican al mapa de salida de la etapa de convolución y_{conv}, denotados ahora como $W_{norm} = \frac{\gamma}{\sqrt{\mathrm{Var}[y_{conv}]+\epsilon}}$, con el resto que sólo se suman al resultado, denotados ahora como $b_{norm} = \beta - \frac{\mathrm{E}[y_{conv}]}{\sqrt{\mathrm{Var}[y_{conv}]+\epsilon}} \cdot \gamma$. Así, como W_{norm} y b_{norm} son constantes, pueden ser combinados matemáticamente a los filtros convolucionales, como es mostrado en la ecuación (4.2), donde W_{norm} es una constante multiplicativa a los pesos convolucionales W_{conv}, mientras que b_{norm} se agrega como un bias. Cabe destacar que denotaremos la multiplicación de W_{conv} y W_{norm} como W_{cn}.

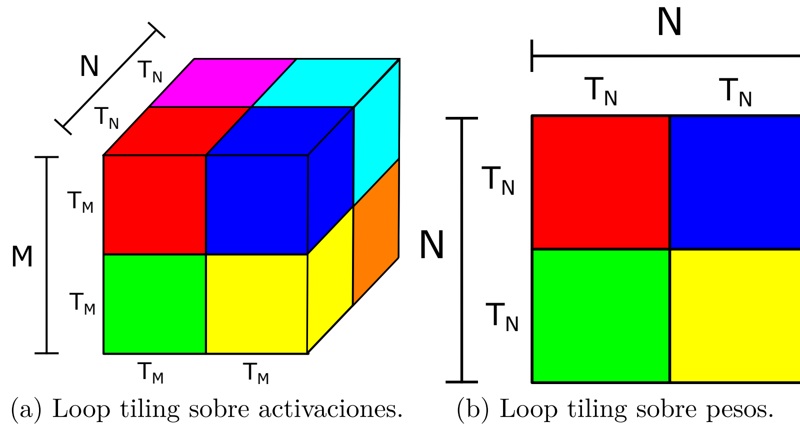
$$y_{norm} = \frac{\gamma}{\sqrt{\mathrm{Var}[y_{conv}] + \epsilon}} \cdot y_{conv} + (\beta - \frac{\mathrm{E}[y_{conv}]}{\sqrt{\mathrm{Var}[y_{conv}] + \epsilon}} \cdot \gamma) = W_{norm} \cdot y_{conv} + b_{norm} \tag{4.1}$$

$$\begin{aligned} y_{norm} &= (W_{norm} \cdot W_{conv}) * x_{conv} + b_{norm} \\ y_{norm} &= W_{cn} * x_{conv} + b_{norm} \end{aligned} \tag{4.2}$$

Gracias a esto, es posible reducir la complejidad de los cálculos al evitar divisiones y raíces cuadradas, reduciendo la cantidad de operaciones de 5 a 1, sólo agregando una suma al final de cada convolución. Además, ya que no se están utilizando los 4 cuatro parámetros de la etapa de normalización, la cantidad de pesos en la red se reduce en un 0.5 %, eliminando cerca de 17000 parámetros. Cabe destacar que como sólo se está aplicando una transformación matemática a la arquitectura de la red, ésta no repercute en la precisión de clasificación, por lo que es posible utilizarla tanto en FPGAs como en otros dispositivos hardware.

4.2. Loop tiling en MobileNet V2

Como explicamos en el Capítulo 2, la técnica de loop tiling es fundamental para reutilizar datos y así reducir los accesos a la memoria externa. La Figura 4.1 muestra la división de las activaciones y pesos que realizamos sobre MobileNet V2 para aplicar loop tiling. Para el caso de las activaciones, dividimos los mapas en bloques de tamaño $T_M \times T_M \times T_N$, con T_M y T_N los factores de tiling aplicados sobre el tamaño de los mapas y la cantidad de canales de estos respectivamente. Por su parte, como los pesos en general son matrices de dos dimensiones, los dividimos en bloques de tamaño $T_N \times T_N$. Cabe destacar, que para el caso de las convoluciones *depthwise*, los pesos están agrupados en arreglos de 3 dimensiones de tamaño $N \times K \times K$,



(a) Loop tiling sobre activaciones. (b) Loop tiling sobre pesos.

Fig. 4.1: Loop tiling aplicado sobre mapas y pesos de MobileNet V2. .

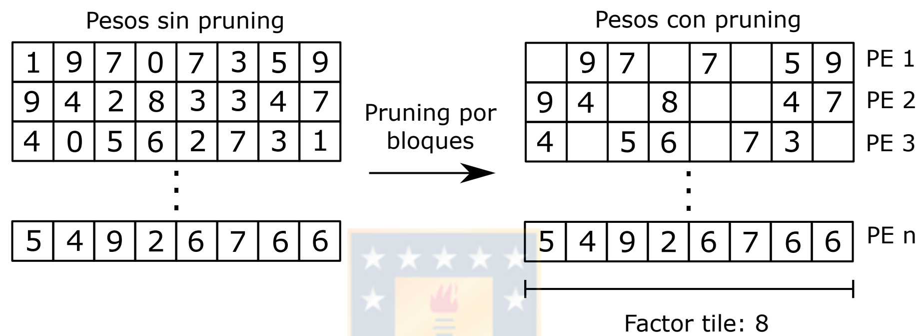


Fig. 4.2: Pruning por bloques aplicado en MobileNet V2.

pero como $K = 3$, no aplicamos loop tiling sobre estas dimensiones, por lo que los bloques de parámetros para este tipo de convoluciones son de tamaño $T_N \times K \times K$.

4.3. Pruning en MobileNet V2

Como mencionamos en el Capítulo 2, es necesario aplicar pruning estructurado a los pesos de la red para balancear las cargas en los PEs. Para esto, realizamos pruning en bloques [34], tal como muestra la Figura 4.2. Aquí, dividimos los bloques en función del factor de tiling de la profundidad de entrada de cada capa para eliminar una cantidad fija de pesos, permitiendo que todos los PEs del hardware dedicado realicen la misma cantidad de multiplicaciones.

Cabe destacar que como al aplicar pruning se obtiene una matriz dispersa, se pierde la posición del peso. Para esto, utilizamos un índice que indica cuantos pesos fueron eliminados entre un peso no cero y el siguiente. Por ejemplo, de la Figura 4.2 podemos ver que en el tercer PE, el peso de valor 4 tiene un índice uno, ya que hay un sólo peso eliminado entre este y el siguiente (5), mientras que el índice en el peso de valor 5 es cero ya que no hay pesos eliminados

entre este y el peso siguiente.

Es importante mencionar que sólo aplicamos pruning a los pesos de las convoluciones *expansion* y *projection* y de la capa de clasificación. No reducimos la cantidad de pesos de las convoluciones *depthwise*, ya que además de conocer la posición de los pesos dentro de la capa, es necesario conocer la posición del peso dentro del filtro 3×3, complicando aún más la inferencia. Sumado a esto, como las convoluciones *depthwise* se encargan de extraer las características de la imagen, eliminar pesos afecta considerablemente la precisión de la red, ya que descartando el 10 % de los pesos de todas las convoluciones obtenemos una precisión sin reentrenamiento de un 55.21 %, aumentando el error en un 16.59 %, mientras que si sólo eliminamos el 10 % de los pesos de las convoluciones *expansion* y *projection*, la precisión sin reentrenamiento alcanza un 71.04 %, sólo un 0.76 % menor que la precisión sin pruning. Aun así, si bien no aplicar pruning a las capas *depthwise* aumenta la cantidad de pesos de la red, estas capas sólo contienen aproximadamente 65000 pesos, representando sólo un 3 % de los pesos convolucionales totales de red, por lo que no aplicar pruning en ellas no repercute significantemente en el tamaño de la CNN.

4.4. Cuantización en MobileNet V2

Tal como mencionamos en el Capítulo 2, además de utilizar pruning para reducir la cantidad de pesos, también es necesario disminuir la cantidad de bits tanto en los pesos de la red como en mapas de características, lo cual nos permite almacenar una mayor cantidad de datos en la memoria del hardware y a su vez reducir la complejidad de cálculo. Para esto, aplicamos cuantización lineal dinámica, como en [13], tanto para los pesos como activaciones de cada capa de la red.

La cuantización lineal está dada por la ecuación (4.3), donde y_{quant} e y_{float} representan a los valores quantizados y flotantes respectivamente, mientras Δ es la escala que cuantiza los datos. Para determinar la escala Δ es necesario conocer la cantidad de bits que serán utilizados para cuantizar cada parámetro. Específicamente, la escala es calculada mediante la ecuación (4.4), donde se divide la resta entre el máximo y mínimo valor del dato flotante por el máximo valor posible según la cantidad de bits utilizada (2^{bits}).

$$y_{quant} = \text{redondeo}(\frac{y_{float}}{\Delta}) \tag{4.3}$$

$$\Delta = \frac{\max(y_{float}) - \min(y_{float})}{2^{bits}} \tag{4.4}$$

Capítulo 5

Arquitectura de acelerador hardware

En este capítulo mostramos la arquitectura hardware diseñada para realizar la inferencia de MobileNet V2. Aquí detallamos en el sistema general de nuestro diseño, en las secciones de control y procesamiento, y cómo nuestra arquitectura realiza la inferencia.

5.1. Sistema general

La Figura 5.1 muestra el sistema general de la arquitectura diseñada para la inferencia de MobileNet V2 en hardware dedicado. La arquitectura está basada en un sistema heterogéneo que combina la lógica programable del FPGA, denominada como PL (del inglés *Programmable Logic*, lógica programable), junto con todos los dispositivos externos a esta, como CPU, GPU y memorias externas, las que son conocidas como PS (del inglés *Processing Subsystem*, subsistema de procesamiento). En nuestra arquitectura, utilizamos la PL para diseñar aceleradores hardware dedicados para los diferentes tipos de capas de la CNN, agrupados en PEs, que permiten la inferencia de MobileNet V2 en paralelo, por lo que la llamaremos sección de procesamiento. Por otro lado, el PS, que llamaremos sección de control, usa la CPU para controlar el proceso de inferencia y la memoria RAM (del inglés *Random Access Memory*, memoria de acceso aleatorio) externa para almacenar las activaciones y pesos que no pueden ser guardados en las memorias internas de cada PE por limitaciones del hardware.

La inferencia de MobileNet V2 en la arquitectura funciona de la siguiente manera: (1) la CPU envía las activaciones y el residual a una interfaz DMA (del inglés *Direct Memory Access*, acceso directo a memoria), que permite un stream de datos entre la memoria externa y los

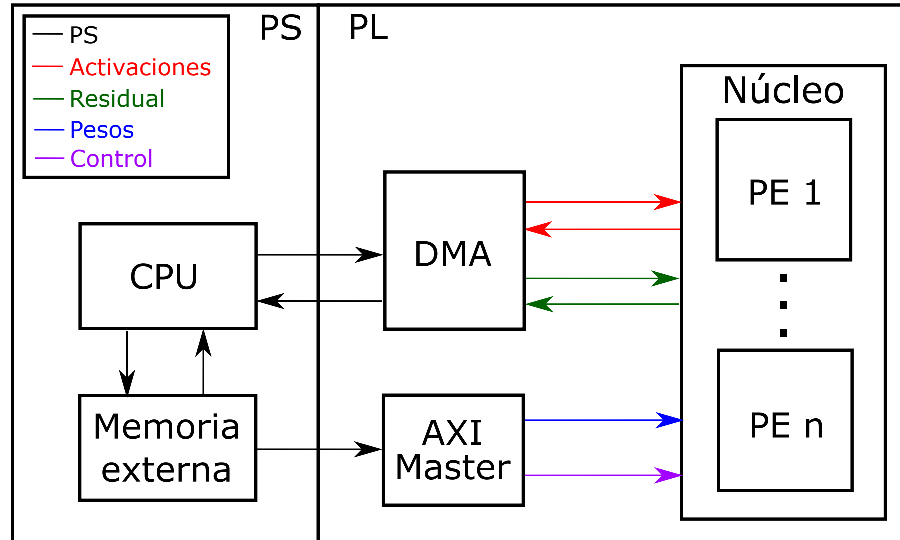


Fig. 5.1: Sistema general de la arquitectura hardware.

PEs, y conecta los pesos e información de control almacenados en la memoria externa con la lógica programable, por medio del protocolo de comunicación AXI Master, (2) cada PE recibe la información de control y la almacena en un buffer, y (3) los PEs reciben las activaciones de entrada y pesos, procesan la capa según la información de control y envían los resultados de vuelta a la memoria externa por medio de la interfaz DMA. Cabe destacar que este proceso es repetido varias veces en cada capa para completar el procesamiento.

5.2. Sección de control

Tal como mencionamos en la sección 5.1, la sección de control está constituida por una CPU, que se encarga de controlar el proceso de inferencia, y una memoria externa que almacena las activaciones y pesos de la red. Específicamente, la CPU indica a cada PE de la PL qué capa se procesará, enviando la información de control a la sección de procesamiento. Luego, la CPU inicia el protocolo de comunicación AXI Master y la interfaz DMA para que cada PE pueda acceder directamente a la memoria externa para leer las activaciones, residuales y pesos según corresponda. Mientras los datos son procesados, la CPU queda ociosa hasta que la interfaz DMA activa la señal de termino de transferencia, indicando que los PEs ya han enviado todos las activaciones de salida de la capa. Con esto, la CPU ordena los datos recibidos en la memoria externa y repite el proceso con la siguiente capa. Cabe destacar, que la capa *softmax* es procesada en la CPU debido a la complejidad de sus operaciones (divisiones y exponencial) y a que sólo procesa los 1000 datos que entrega la capa de clasificación.

5.3. Sección de procesamiento

Como ya mencionamos en el sistema general, la sección de procesamiento es utilizada para acelerar la inferencia de MobileNet V2. Esta sección del hardware está constituida por n PEs que procesan cada capa de la red en paralelo. Cada PE lee la información de control enviada por el PS, lo que denominaremos como etapa de decodificación, y procesa la capa de MobileNet V2, la que llamaremos etapa de procesamiento.

5.3.1. Etapa de decodificación

En la etapa de decodificación, cada PE recibe la información necesaria para procesar cada capa de MobileNet V2, almacenándola en buffers para ser leídos por cada acelerador. Esta información está dividida en tres grupos:

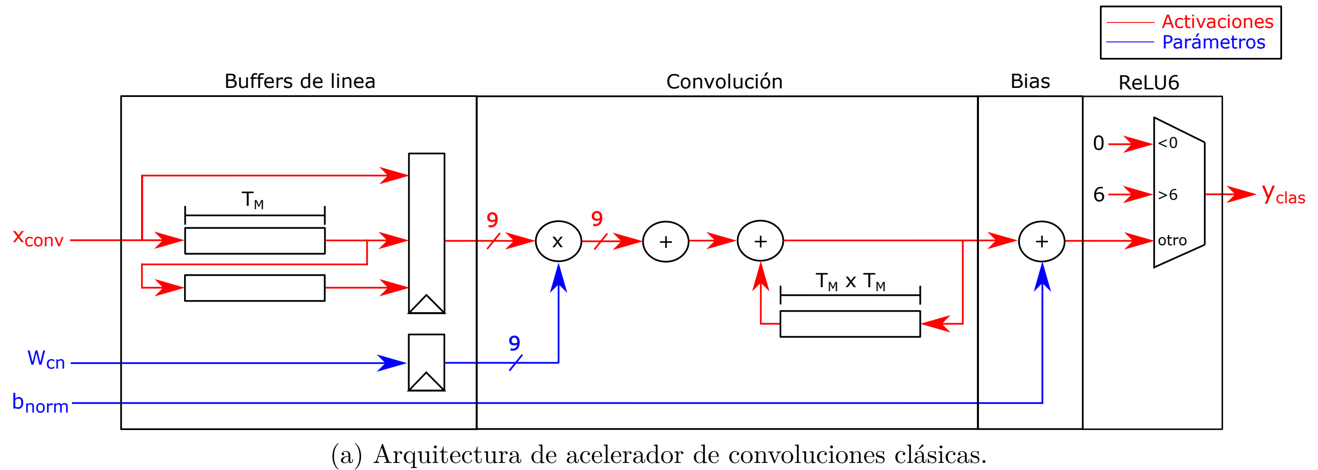
Información de capa: Estos datos están compuestos por el tipo de capa a procesar (capa convolucional normal, *depthwise*, *expansion/projection*, aplanamiento o de componentes conectados) y el tamaño y profundidad de las activaciones de entrada, salida y pesos.

Información de cuantización: Datos compuestos por la posición del punto fijo necesario para operar la red. Entre estos se encuentran los puntos fijos de las activaciones, residual, parámetros y ReLU6. Cabe destacar que si bien estos dos últimos pueden ser calculados mediante los puntos fijos de las activaciones y pesos, son entregados como información para disminuir el uso de recursos y el tiempo de procesamiento.

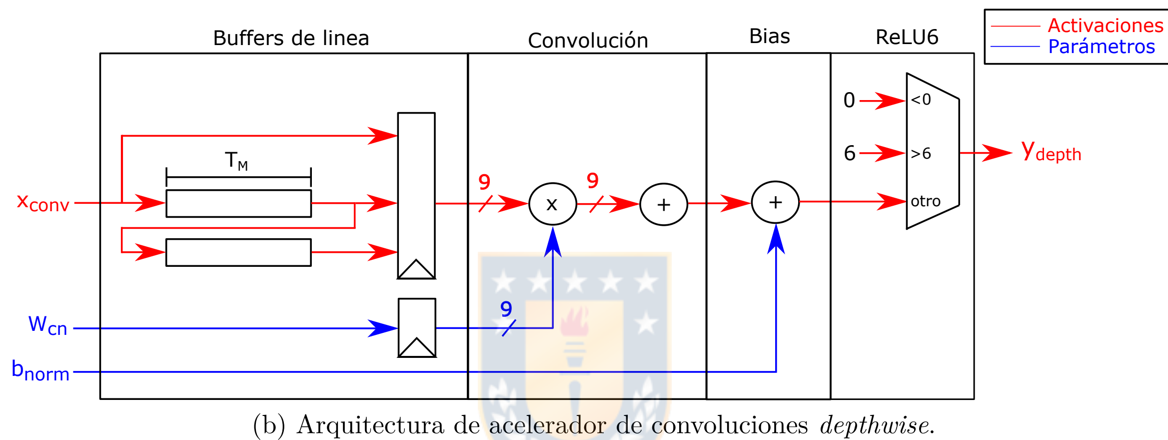
Información de loop tiling: Esta información corresponde al tamaño de los bloques a procesar. Específicamente, estos datos son los factores de loop tiling de profundidad de entrada, salida y de tamaño de las activaciones y pesos de la capa.

5.3.2. Etapa de procesamiento

La etapa de procesamiento, tal como dice su nombre, se encarga de procesar los datos recibidos desde la sección de control para lograr la inferencia de la red. Debido a que MobileNet V2 posee 5 tipos de capas descritas en el Capítulo 3, cada PE contiene un acelerador para cada una, seleccionándolas por medio de un demultiplexor controlado por la información de control.



(a) Arquitectura de acelerador de convoluciones clásicas.



(b) Arquitectura de acelerador de convoluciones *depthwise*.

Fig. 5.2: Arquitecturas de aceleradores de convoluciones clásicas y *depthwise*.

5.3.2.1. Acelerador de convolución clásicas y *depthwise*

Ya que el funcionamiento del acelerador de convoluciones clásicas es similar al de convoluciones *depthwise*, en esta sección detallamos a los dos mostrando las diferencias según corresponda. Aun así, es importante destacar que ambas arquitecturas son totalmente independientes entre sí. La Figura 5.2 muestra las arquitecturas de los aceleradores para las convoluciones clásicas y *depthwise* de MobileNet V2. El funcionamiento de estos aceleradores se puede dividir en la etapa de buffers de línea, ejecución de la operación de convolución, adición de bias y función de activación.

En la primera etapa, ya que cada PE recibe un píxel del mapa de características de entrada por ciclo, y como las convoluciones clásicas y *depthwise* en MobileNet V2 utilizan máscaras de 3×3, los aceleradores utilizan dos buffers de linea configurados como memorias FIFO (del inglés *First In-First Out*, primero en entrar-primero en salir) de tamaño T_M para almacenar una fila del bloque de entrada conectadas en cascada y un banco de nueve registros para generar

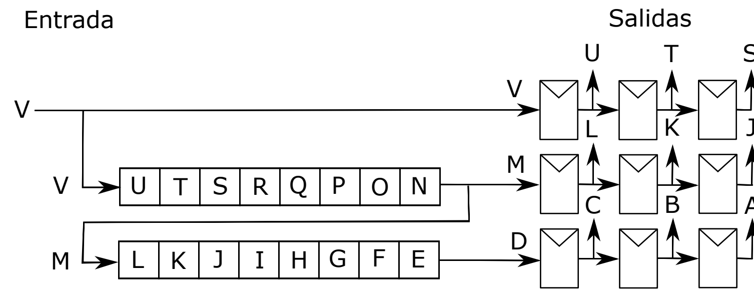


Fig. 5.3: Funcionamiento buffers de línea y banco de registros.

la ventana 3×3. La Figura 5.3 detalla el funcionamiento de esta etapa. Aquí, a medida que el acelerador va leyendo los píxeles, se van almacenando en el primer buffer de línea. Cuando este se llena, el primer píxel que entra sale de la memoria FIFO y es almacenado en el segundo buffer de línea. Cuando el segundo buffer de línea está completo, el primer dato que se almacenó sale de la memoria FIFO, lo que permite obtener 3 datos por ciclo. Aun así, para generar la ventana 3×3, los datos de salida de los buffers de línea y la entrada DMA al acelerador son almacenadas en un banco de nueve registros, que al ir desplazando los valores almacenados al siguiente, permite que la arquitectura puede leer y así procesar nueve píxeles a la vez. Cabe destacar que de la misma forma, el acelerador lee los nueve pesos del filtro 3×3 correspondiente y los almacena en registros para realizar la convolución.

Con la ventana de nueve píxeles del mapa de entrada y el filtro convolucional ya disponibles, los aceleradores ya pueden realizar la convolución. Para esto, estas arquitecturas utilizan 9 multiplicadores en paralelo para multiplicar el mapa de entrada con el filtro convolucional, sumando los resultados para obtener un píxel según las ecuaciones (3.3) y (3.5) respectivamente. A diferencia de las convoluciones *depthwise*, la convolución clásica necesita sumar las resultados de las operaciones de convolución de todos los canales del mapa de entrada para generar un canal del mapa de salida de la red, tal como detalla la ecuación (3.3). Para esto, este acelerador agrega una etapa extra agregando un sumador para sumar el resultado del canal que se esta procesando con las sumas parciales de los canales anteriores. Ya que el resultado de cada píxel del canal de salida es diferente, agregamos un buffer de tamaño $T_M \times T_M$ para almacena las sumas parciales de todo el canal de salida.

Una vez que se obtiene un píxel del mapa de características de salida, ya sea al realizar la operación de convolución en el caso de las convolución *depthwise* o al realizar la suma de todos los resultados de los canales de entrada en el caso de las convoluciones clásicas, el acelerador suma a cada píxel el bias de normalización de la ecuación (4.2). Este bias es leído desde la memoria externa y es almacenado en un registro para no perderlo. Finalmente, los aceleradores

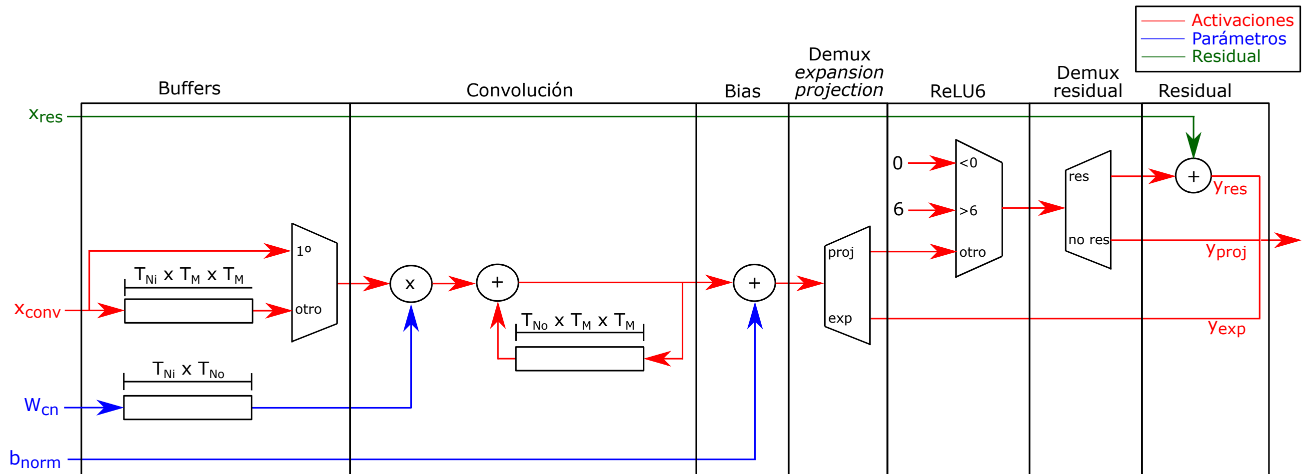


Fig. 5.4: Arquitectura de acelerador de convoluciones *expansion* y *projection*.

utilizan la función ReLU6 como un multiplexor para saturar en cero o seis las salidas menores y mayores respectivamente, enviando las activaciones de vuelta a la memoria externa por medio de la interfaz DMA como un stream de datos.

5.3.2.2. Acelerador de convolución *expansion* y *projection*

La Figura 5.4 muestra la arquitectura del acelerador para convoluciones *expansion/projection*. El funcionamiento de esta arquitectura puede ser dividido en las etapas de almacenamiento de mapas y pesos de entrada en buffers, multiplicación, adición de bias y selección de tipo de capa (*expansion* o *projection*).

A diferencia de las convoluciones clásicas y *depthwise*, las capas *expansion/projection* utilizan una operación de convolución de una dimensión, es decir, una multiplicación tal como muestra la ecuación (3.4), por lo que no hay que generar una ventana 3×3. Aun así, ya que estas convoluciones utilizan todos los canales del mapa de entrada para generar un canal de salida, y como la CPU sólo envía una vez cada canal, es necesario almacenar el bloque completo de la activación de entrada para que los datos no se pierdan. Por esto, la primera etapa del acelerador consiste en almacenar este bloque en un buffer de tamaño $T_{Ni} \times T_M \times T_M$. Para acelerar la inferencia, el acelerador va llenando el buffer al mismo tiempo que se procesa el primer canal de salida de la capa, leyendo directamente el dato desde la interfaz DMA. Una vez que el buffer ya esta lleno, se leen las activaciones desde éste. Cabe destacar que también se almacenan en un buffer de tamaño $T_{Ni} \times T_{No}$ los pesos convolucionales, con el objetivo de acelerar la inferencia por pruning de la capa.

La siguiente etapa del acelerador consiste en aplicar la ecuación (3.4). Para esto, se multiplica el píxel entregado por la etapa de almacenamiento con el peso unitario. Luego, al igual que en las convoluciones clásicas, cada canal de salida de las convoluciones *expansion/projection* se obtienen al sumar todos los productos de los canales de entradas y pesos, por lo que el acelerador incorpora un sumador que suma el producto del canal de entrada que se está procesando con las sumas parciales de los productos de los canales anteriores, almacenados en un buffer de tamaño $T_{No} \times T_M \times T_M$. Una vez que el acelerador suma todos los productos, se agrega el bias de normalización de la ecuación (4.2), leído desde la memoria externa y almacenado en un registro para no perderlo.

La última etapa del acelerador determina si la capa es *expansion* o *projection*. Si la capa utiliza una convolución *expansion*, la suma de la etapa de bias es transmitida por la interfaz DMA a la memoria externa. En caso contrario, el acelerador aplica la función de activación ReLU6 por medio de un multiplexor para saturar en cero o seis los datos que son menores o mayores respectivamente, dejando pasar los intermedios. Luego, se utiliza un demultiplexor que determina si la capa utiliza o no residual. Si la capa utiliza residuales, el acelerador suma la activación de la etapa ReLU6 con el residual leído desde la interfaz DMA. Finalmente, los resultados son transmitidos a la memoria externa por medio de la interfaz DMA como un stream de datos.

5.3.2.3. Acelerador de aplanamiento

La Figura 5.5 muestra la arquitectura diseñada para acelerar la capa de aplanamiento. A diferencia de los aceleradores anteriores, ya que la CPU envía los píxeles del mapa de entrada de forma continua y el tamaño del última mapa convolucional es 7×7 píxeles por canal, no es necesario utilizar memorias para almacenar las activaciones, por lo que el acelerador lee directamente los datos desde la interfaz DMA. Luego, se realiza la suma del píxel leído con la suma parcial de los píxeles anteriores del canal. Cuando el acelerador suma los 49 píxeles de un canal de entrada, se divide el resultado por 49 para calcular su promedio, tal como detallamos en la ecuación (3.11), enviando el resultado a la memoria externa por la interfaz DMA. Cabe destacar, que como se está dividiendo por una constante, el acelerador utiliza LUTs para implementar la división.

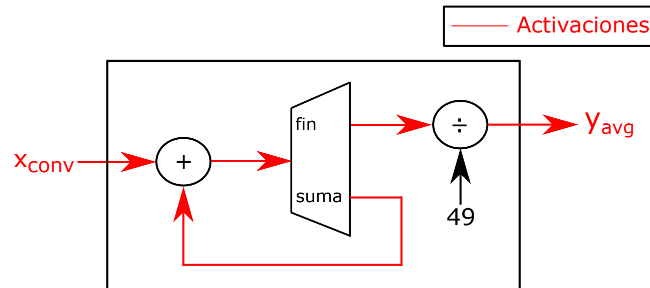


Fig. 5.5: Arquitectura de acelerador de aplanamiento.

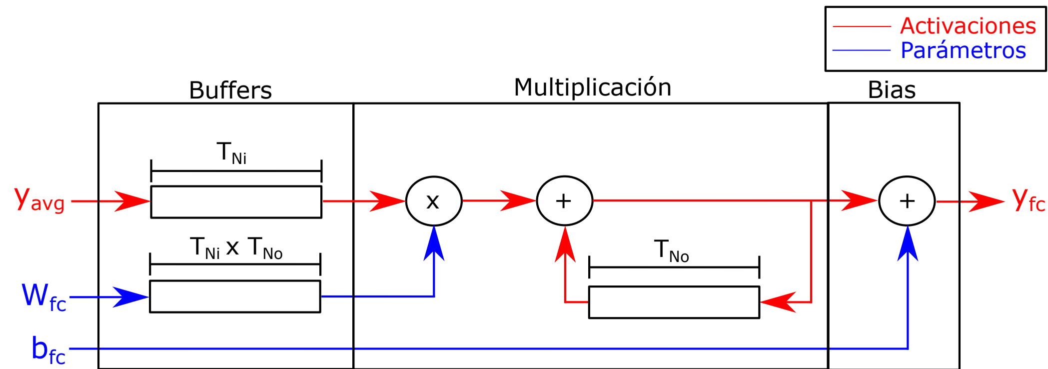


Fig. 5.6: Arquitectura de acelerador de componentes totalmente conectados.

5.3.2.4. Acelerador de componentes totalmente conectados

La Figura 5.6 muestra la arquitectura del acelerador para la capa de componentes totalmente conectados. El funcionamiento del acelerador puede ser dividido en las etapas de almacenamiento de activaciones en la memoria interna, multiplicación de activaciones por pesos y adición de bias.

Al igual que el acelerador de convoluciones *expansion/projection*, es necesario almacenar las activaciones, ya que la CPU sólo envía una vez cada dato y la capa necesita utilizarlos varias veces para obtener el arreglo de salida. Para esto, el acelerador lee los datos desde la memoria externa por medio del protocolo de comunicación correspondiente, y los almacena en un buffer de tamaño T_{Ni} y $T_{Ni} \times T_{No}$ para activaciones y parámetros respectivamente.

Una vez que se llenan los buffers, el acelerador multiplica las activaciones por los pesos, tal cual como detalla la ecuación (3.12). Luego, se suma el producto obtenido con las sumas parciales de los productos de las multiplicaciones anteriores, que son almacenadas en un buffer de tamaño T_{No}. Finalmente, una vez que se han sumando todos los productos de las activaciones de entrada, el acelerador suma al resultado el bias b_{fc}, enviando el componente del arreglo de salida hacia la memoria externa como un stream de datos por medio de la interfaz DMA.

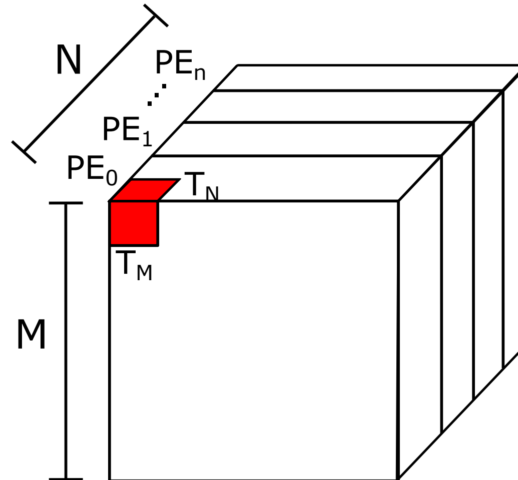


Fig. 5.7: División espacial de las activaciones y pesos en hardware.

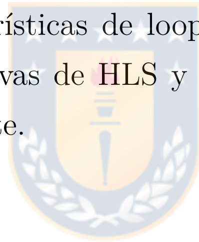
5.3.3. Organización de datos en los elementos de procesamiento

Como nuestro diseño utiliza n PEs para procesar en paralelo, realizamos una división espacial de la red para que cada uno procese la misma cantidad de datos y así mejorar el balance de cargas por pruning. La Figura 5.7 muestra la división espacial sobre las activaciones y pesos. Para esto, dividimos la profundidad de las activaciones y pesos (N) por la cantidad de PEs que utiliza nuestra arquitectura (n). Así, aplicando el factor de loop tiling de profundidad, cada PE procesará $\frac{N}{n \cdot T_N}$ bloques, balanceando la carga sobre cada uno.

Capítulo 6

Implementación en hardware

En este capítulo mostramos la plataforma FPGA y las consideraciones para implementar la arquitectura. Específicamente, detallamos las características del FPGA, las plataformas para sintetizar la arquitectura, las características de loop tiling, pruning y cuantización aplicadas a MobileNet V2, el uso de las directivas de HLS y una comparación del desempeño entre la arquitectura punto fijo y punto flotante.



6.1. Descripción del hardware

Para implementar la arquitectura descrita en el Capítulo 5 utilizamos la plataforma Xilinx Zynq UltraScale+ MPSoC, en específico el modelo ZCU104 mostrado en la Figura 6.1. Esta plataforma incorpora la FPGA XCZU7EV con las CPUs ARM Cortex-A53 y ARM Cortex-R5 y la GPU ARM Mali-400. La tabla 6.1 muestra las características de la plataforma ZCU104.

Cabe destacar que además de memorias BRAM (del inglés *Block RAM*, bloque de RAM), la FPGA XCZU7EV tiene un tipo de memoria llamada URAM (del inglés *Ultrascale RAM*, RAM de UltraScale). Las memorias URAM tienen dos puertos, permitiendo la lectura y escritura sincrónica tal como ocurre con las memorias BRAM, pero utilizan bloques de tamaño fijo de 4K x 72 bits, mientras que en las BRAM el ancho de estos bloques es variable. Debido a esto, al almacenar datos de menos de 72 bits en la memorias URAM, siempre se utilizará la misma cantidad de bloques.

Hardware	Descripción
FPGA XCZU7EV	504.000 celdas lógicas 461.000 flip-flops 11Mb de BRAM 27Mb de URAM 1728 unidades DSP
CPU ARM Cortex-A53	CPU quad-core o dual-core Arquitectura de 64 o 32 bits Frecuencia de reloj de 1.5 GHz Dos caches nivel 1 de 32KB y una cache nivel 2 de 1MB
CPU ARM Cortex-R5	Arquitectura de 32 bits Frecuencia de reloj de 600MHz Cache nivel de 32KB y cache nivel 2 de 128 KB
GPU ARM Mali-400	Frecuencia de 667MHz Soporta OpenGL y OpenVG Cache de 64KB
Otros	Memoria RAM DDR4 de 2GB en el PS Puerto USB-UART Slot Micro SD

Tabla 6.1: Características de plataforma ZCU104.

6.2. Software utilizado para implementar la arquitectura

Para diseñar el acelerador hardware utilizamos el proceso de diseño HLS, convirtiendo el código a RTL mediante el software Xilinx Vivado HLS 2019.2. Para manejar la comunicación entre la CPU, FPGA y memoria utilizamos el software Xilinx Vitis IDE 2019.2. Sintetizamos el diseño RTL mediante el software Xilinx Vivado 2019.2 design suite.

6.3. Consideraciones de implementación

6.3.1. Consideraciones para MobileNet V2

Tal como mencionamos en el Capítulo 4, utilizamos las técnicas loop tiling, pruning y cuantización para reducir y acelerar la inferencia de MobileNet V2 en hardware dedicado. En el caso de loop tiling, lo aplicamos sobre el tamaño de los mapas y en la cantidad de canales de

Fig. 6.1: Plataforma ZCU104.

Factor de loop tiling		Tiempo de inferencia (ms)
Activaciones	Pesos	
14	16	252.8
14	32	239.7
14	64	241.6
28	16	230.2
28	32	220.5

Tabla 6.2: Tiempos de inferencia al variar factores de loop tiling.

activaciones y pesos. La tabla 6.2 muestra los tiempos de inferencia al variar los factores de loop tiling tanto para activaciones como pesos. De la tabla podemos apreciar que a medida que los factores de tiling son mayores, menor es el tiempo de inferencia, ya que disminuyen los accesos a la memoria externa. Debido a esto, escogimos factores de tiling de 28 y 32 para el tamaño de los mapas T_M y profundidad T_N respectivamente. Cabe destacar que factores de tiling mayores provocan una sobreutilización de las memorias internas del FPGA.

En el caso del pruning, aplicamos diferentes tasas de dispersión (d) sobre las capas *expansion/projection* y de componentes totalmente conectadas bajo las condiciones de loop tiling mencionadas. La tabla 6.3 muestra las precisiones top-1 alcanzadas bajo estas condiciones, reentrenando la red con una tasa de aprendizaje de 0.001 y 30 *epochs*. Con estos resultados, podemos ver que las capas convolucionales *expansion/projection* tienen un mayor efecto en la

Tasas de dispersión (d)		Precisión top-1
Expansion/projection	Componentes conectados	
0.1	0.3	70.68
0.1	0.7	69.77
0.3	0.7	65.78
0.5	0.7	63.55

Tabla 6.3: Resultados de pruning en MobileNet V2.

Variables cuantizadas	Precisión top-1 según número de bits (%)				
	16 bits	12 bits	10 bits	8 bits	6 bits
Pesos capa convolución	71.88	71.87	71.79	69.82	21.64
Pesos capa clasificación	71.88	71.88	71.86	71.84	71.75
Activaciones capa convolución	71.89	71.85	71.46	60.51	0.46
Activaciones clasificación	71.88	71.87	71.86	71.73	69.25

Tabla 6.4: Resultados cuantización lineal en MobileNet V2.

precisión alcanzada por MobileNet V2, por lo que escogimos una tasa de dispersión del 30 % para las convoluciones *expansion/projection* y de un 70 % para la capa de clasificación, ya que permiten una mayor reducción de la red sin provocar un gran aumento en el error de precisión.

En el caso de la cuantización, la aplicamos con diferentes cantidades de bits tanto para las activaciones como pesos, independiente del pruning aplicado, obteniendo las precisiones mostradas en la tabla 6.4. A partir de estos resultados y bajo los criterios de seleccionar la menor cantidad de bits con una buena precisión, escogimos cuantizar los pesos de las capas de convolución y clasificación con 12 y 6 bits respectivamente, mientras que para las activaciones, utilizamos 12 bits en convoluciones y 10 bits al clasificar. Cabe destacar que debido a la cuantización dinámica, cada capa de la red posee un punto fijo diferente, variando entre 5 y 10 bits fraccionarios para las capas convolucionales y de 1 bit para la capa de clasificación.

Finalmente, combinando las tres técnicas, obtuvimos una precisión de inferencia top-1 y top-5 de 65.62 % y 87.03 % respectivamente, aumentando el error en un 6.26 % y 3.26 % comparado con las precisiones top-1 y top-5 de la red sin modificar. Cabe destacar que debido a la cantidad de recursos disponibles, nuestra implementación utiliza cuatro PEs en paralelo.

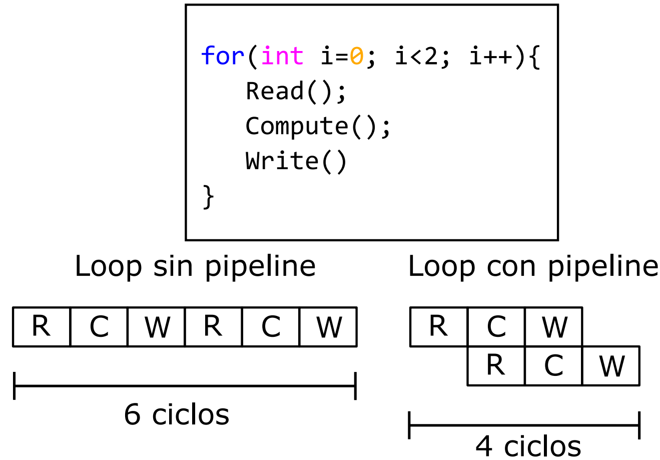


Fig. 6.2: Desempeño de ciclo `for` con y sin pipeline.

6.3.2. Consideraciones de diseño en HLS

Para lograr una óptima síntesis de la arquitectura propuesta en el Capítulo 5 utilizamos diferentes directivas en HLS.

6.3.2.1. Directiva HLS PIPELINE

Pipeline es una técnica para implementar en simultaneo instrucciones dentro del hardware, disminuyendo el tiempo de procesamiento. La Figura 6.2 muestra el desempeño en un loop al aplicar o no pipeline. Como podemos apreciar de la figura, el uso de pipeline disminuye el tiempo total de procesamiento de 6 a 4 ciclos, además de permitir la obtención de un resultado por ciclos en vez de cada tres. Por esto, utilizamos la directiva HLS PIPELINE en cada arquitectura de procesamiento de nuestra implementación, para que los píxeles de salida sean enviados como un stream de datos, disminuyendo el tiempo de inferencia. Además utilizamos las directivas de dependencias de datos para asegurar un correcto funcionamiento del pipeline.

La tabla 6.5 muestra la comparación de la cantidad de ciclos de procesamiento al usar o no la directiva HLS PIPELINE. Como es de esperar, el uso de pipeline en los aceleradores reduce drásticamente la cantidad de ciclos de procesamiento, ya que al no usar la directiva de HLS, no se realizan operaciones independientes en paralelo. A modo de ejemplo, si denominamos a L como la latencia para procesar un dato, en la arquitectura sin pipeline cada resultado se obtiene cada L ciclos, mientras que con pipeline, sólo el primer resultado se procesa en esta cantidad de tiempo, entregando los siguientes cada 1 ciclo de reloj. Cabe destacar que las latencias y ciclos de procesamiento son explicados en detalle en el Capítulo 7.

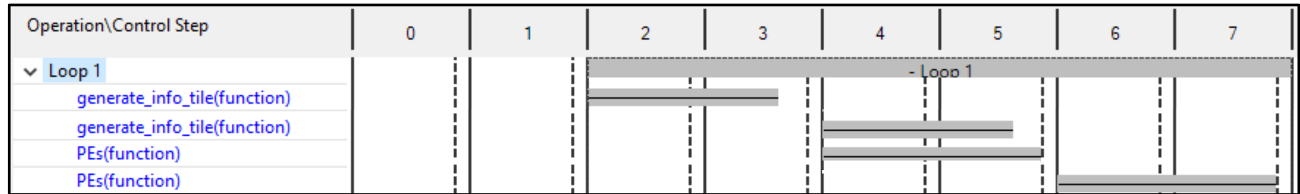
Acelerador	Etapa	Ciclos con pipeline		Ciclos sin pipeline	
		Mínima	Máxima	Mínima	Máxima
Convoluciones clásicas	Lectura de pesos	16	16	90	90
	Buffers de línea	60	60	60	60
	Ventana de registros	5	5	12	12
	Procesamiento	1020	1020	42360	42360
Convoluciones *depthwise*	Lectura de pesos	17	17	99	153
	Buffers de línea	18	60	18	60
	Ventana de registros	5	5	12	12
	Procesamiento	89	908	2106	47700
Convoluciones *exp/proj*	Lectura de pesos	104	1032	288	11264
	Procesamiento	53	788	406	3192
Aplanamiento	Procesamiento	1632	1632	3136	3136
Componentes conectados	Lectura de pesos	2056	2056	22528	22528
	Procesamiento	4098	4098	12288	12288

Tabla 6.5: Ciclos de procesamiento de cada acelerador al aplicar o no pipeline.

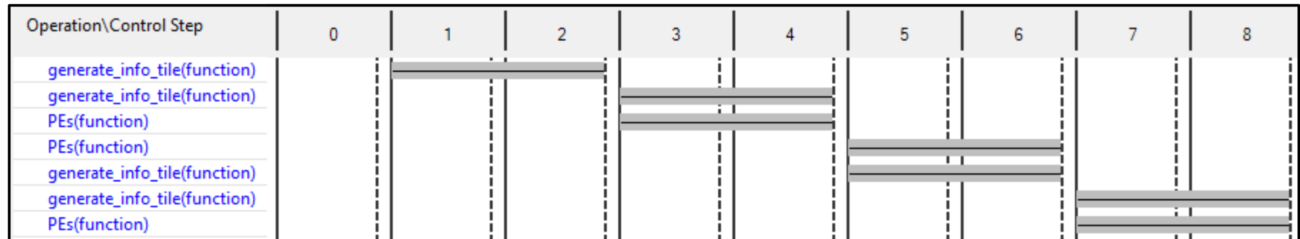
6.3.2.2. Directiva HLS UNROLL

Tal como mencionamos en el Capítulo 2, la técnica de loop unrolling permite reducir las burbujas de procesamiento, mejorando el desempeño del algoritmo. Por esto, aplicamos la directiva HLS UNROLL para desenrollar completamente el ciclo principal de la sección de procesamiento. Este ciclo se encarga de decodificar y procesar cada capa de MobileNet V2, tomando varias iteraciones debido al uso de loop tiling, . Además, para lograr un efecto de pipeline entre cada etapa, repetimos dos veces los módulos de decodificación y procesamiento. La Figura 6.3 muestra el desempeño del ciclo principal de la etapa de procesamiento al aplicar o no la directiva de unrolling. Como podemos apreciar, al no desenrollar el ciclo, dos iteraciones son procesada en 6 bloques de tiempo, mientras que al utilizar la directiva, se eliminan las burbujas que permiten optimizar el pipeline, por lo que dos iteraciones toman sólo 4 bloques de tiempo en ser procesada, reduciendo el tiempo de inferencia.

Otra ventaja del loop unrolling es que si se combina con pipeline permite el procesamiento en paralelo. Si bien esto idealmente permite paralelizar todas las etapas de procesamiento de cada PE, el uso de la interfaz DMA lo limita, ya que no se dispone de más de una entrada por ciclo. Sin embargo, como en la arquitectura de convoluciones *expansion/projection* se leen los datos desde la memoria externa sólo al procesar el primer canal de salida, almacenando los mapas en un



(a) Ciclo principal de procesamiento sin unroll.



(b) Ciclo principal de procesamiento con unroll.

Fig. 6.3: Comportamiento de ciclo principal de sección de procesamiento al aplicar o no loop unrolling.

Factor de unrolling	Ciclos de procesamiento	
	Mínima	Máxima
Sin unrolling	53	788
Unrolling x7	30	114
Unrolling x14	15	57
Unrolling x28	8	29

Tabla 6.6: Ciclos de procesamiento con diferentes factores de loop unrolling en convoluciones *expansion/projection*.

buffer para ser reutilizados con los siguientes, es posible paralelizar la operación de convolución de los píxeles de los demás canales de salida. Por esto, aplicamos la directiva de loop unrolling sobre uno de los ciclos de tamaño de mapas de las convoluciones *expansion/projection*, tal como muestra la Figura 6.4. No aplicamos esta técnica sobre los dos ciclos de tamaño de mapas ya que se sobreutilizan recursos al sintetizar la arquitectura. La tabla 6.6 muestra la cantidad de ciclos de procesamiento al aplicar diferentes factores de loop unrolling sobre el ciclo. Como podemos apreciar, mientras mayor es el factor de unrroling aplicado, se realiza la convolución de más píxeles en paralelo, por lo que se reduce el tiempo de procesamiento. Debido a esto, utilizamos un factor de loop unrolling de 28 para paralelizar las convoluciones *expansion/projection* que no utilizan la interfaz DMA. Cabe destacar que no hay variaciones de tiempo al seguir aumentando el factor de unrolling, ya que la cantidad de datos almacenados en los buffers esta limitada por el factor de loop tiling escogido para las activaciones, que coincide con 28.

```
for (int i=0; i<TNo; i++){
 for (int j=0; j<TNi; j++){
  for (int k=0; k<TM; k++){
   for (int l=0; l<TM; l++){
    #pragma HLS UNROLL
    convolution();
   }
  }
 }
}
```

Fig. 6.4: Loop unrolling aplicado en convoluciones *expansion/projection*.

6.3.2.3. Directiva HLS ARRAY PARTITION

Cuando se define un arreglo en HLS, la herramienta lo sintetiza como una BRAM, por lo que sólo se puede leer y escribir un dato por ciclo. Para incrementar el número de lecturas y escrituras, es necesario dividir el arreglo en bloques más pequeños, de forma que cada uno sea sintetizado en BRAMs independientes, mejorando el desempeño a costa de aumentar la utilización de este recurso. Para esto, la herramienta incorpora la directiva HLS ARRAY PARTITION, que puede dividir los arreglos en bloques con tamaño fijo, de forma cíclica o completamente independientes.

En los aceleradores de convoluciones clásicas y *depthwise*, utilizamos esta directiva para separar las filas de los buffers de línea con el propósito de que puedan ser leídas y escritas en un mismo ciclo. También separamos completamente las ventanas de registros que almacenan las activaciones y los filtros 3×3 para garantizar que se realicen las multiplicaciones en paralelo. Si no aplicamos una partición completa, no se pueden leer datos al mismo tiempo, por lo que los aceleradores se comportan como si no tuvieran pipeline, aumentando los tiempos de procesamiento a los mostrados en la tabla 6.5. En el caso del acelerador de convoluciones clásicas, utilizamos un buffer para almacenar las sumas parciales. Ya que debido al pipeline sólo se accede a este buffer una vez por ciclo, no es necesario aplicar la directiva para dividirlo.

Para el acelerador de convoluciones *expansion/projection*, utilizamos buffers para almacenar las activaciones de entrada, los pesos y las sumas parciales de las convoluciones. Si bien el uso de la interfaz DMA provoca que sólo se acceda a estas memorias una vez por ciclo, el uso de loop unrolling con factor 28, hace necesario dividir completamente los buffers de activaciones de entrada y sumas parciales en una de las componentes de tamaño de mapas para garantizar

el paralelismo. Si dividimos estos buffers en bloques de menor tamaño, el acelerador no podría leer y escribir los 28 píxeles en paralelo, obteniendo los tiempos de procesamiento mostrados en la tabla 6.6. Para el buffer de pesos, no es necesario aplicar la directiva de HLS, ya que se utiliza el mismo peso por canal de entrada, por lo que sólo se accede a esta memoria una vez por ciclo.

Ya que el acelerador de aplanamiento no utiliza memorias, no utilizamos la directiva HLS ARRAY PARTITION. En el caso del acelerador de componentes totalmente conectados, utilizamos buffers para almacenar las activaciones de entrada, los pesos y las sumas parciales. Ya que debido al uso de la interfaz DMA se accede a cada memoria sólo una vez por ciclo, no hay paralelismo dentro del acelerador. Debido a esto, no es necesario dividir ninguno de los buffers mencionados.

6.3.2.4. Directiva HLS RESOURCE

La directiva HLS RESOURCE es utilizada para especificar con que recurso se sintetizará una variable, arreglo, operación aritmética o argumento de función. Si esta directiva no es especificada, HLS determina el recurso a utilizar, lo que puede comprometer el desempeño. En nuestra implementación, utilizamos esta directiva para especificar que los buffers que almacenan los pesos en las arquitecturas de convoluciones *expansion/projection* y de componentes conectados, sean sintetizadas como URAMs en vez de como BRAMs, recurso utilizado por defecto. Realizamos esto para no sobreutilizar las BRAMs al implementar el diseño.

6.3.2.5. Directiva HLS INTERFACE

La directiva HLS INTERFACE define el protocolo de comunicación de las entradas y salidas del diseño. En nuestra implementación, utilizamos esta directiva para definir los protocolos de AXI-Stream para las activaciones y residual y de AXI Master para los pesos y la información de decodificación.

6.4. Comparación de implementación punto fijo y punto flotante

Al realizar un diseño RTL tradicional, típicamente se asume que las implementaciones con punto fijo tienen un mejor desempeño en cuanto a tiempo de procesamiento, uso de recursos y consumo de potencia respecto a las de punto flotante. Sin embargo, al utilizar HLS, esto no siempre se cumple, ya que esta herramienta realiza optimizaciones para aprovechar al máximo los bloques DSP (del inglés *Digital Signal Processor*, procesador de señales digitales), lo que puede llevar a mejorar el desempeño respecto a las implementaciones de punto fijo. Debido a esto, realizamos una comparación de la arquitectura diseñada al utilizar punto fijo y punto flotante en la inferencia de MobileNet V2.

El primer punto a destacar, es que al utilizar la misma configuración de factor de tiling en el diseño punto flotante (28 para las activaciones y 32 para los pesos), Vivado no puede sintetizar el diseño, debido a que existe una gran utilización de LUTs y BRAMs al usar 4 PEs en paralelo, produciéndose superposiciones de caminos que el software no puede eliminar. Si bien esto ya demuestra que la implementación punto flotante utiliza más recursos que la punto fijo, no comprueba que realice la inferencia en un tiempo mayor. Por esto, para realizar una correcta comparación, redujimos el factor de tiling a 14 en las activaciones, sólo para realizar esta prueba.

La tabla 6.7 muestra la comparación del uso de recursos en las implementaciones de punto fijo y punto flotante. Tal como comentamos, la arquitectura punto flotante utiliza una mayor cantidad de recursos respecto a la punto fijo. Esto se debe a que la implementación punto flotante utiliza datos de 32 bits, divididos en 23 bits para el coeficiente o mantisa, 8 bits para el exponente y un bit de signo, mientras que la implementación punto fijo utiliza 12 y 10 bits para las activaciones en las capas convolucionales y de clasificación respectivamente, y 12 y 6 bits para los pesos de dichas capas. Debido a esto, la implementación punto flotante necesita más BRAMs para almacenar las activaciones, una mayor cantidad de DSPs para procesar multiplicaciones y sumas, registros de más bits para almacenar las variables entre etapas del pipeline, y por consecuente, una mayor cantidad de LUTs para controlar datos de mayor tamaño. Respecto al uso de URAMs, ambas implementaciones utilizan un 25 % del total, debido a que el tamaño de estas memorias es fijo en 72 bits.

La tabla 6.8 muestra el tiempo de inferencia y el consumo de potencia de ambas imple-

Arquitectura	LUT	Registros	BRAM	URAM	DSP
Punto fijo	51.23	28.28	39.10	25.00	16.44
Punto flotante	76.53	48.89	58.97	25.00	41.44

Tabla 6.7: Porcentaje de utilización de recursos en arquitecturas de punto fijo y punto flotante.

Arquitectura	Tiempo de inferencia (ms)	Tasa de cuadros por segundo (fps)	Potencia consumida (W)
Punto fijo	239.7	4.17	6.92
Punto flotante	389.1	2.57	9.42

Tabla 6.8: Tiempos de inferencia y consumo de potencia en arquitecturas de punto fijo y punto flotante.

mentaciones. Respecto al consumo de potencia, la implementación punto flotante consume 1.36 veces más potencia que la punto fijo. Esto está directamente relacionado con el uso de recursos, ya que mientras más grande es el hardware, es necesaria más energía para alimentar el chip. Al comparar los tiempos de inferencia, la arquitectura punto flotante es 1.62 veces más lenta que la punto fijo. Esto se debe a que en operaciones de multiplicaciones y suma, como $a \times b + c$, que son la base de las convoluciones, la arquitectura punto flotante realiza la multiplicación de los coeficientes de 23 bits y suma los exponentes de 8 bits para obtener el producto de $a \times b$, mientras que para realizar la suma, hay que desplazar el coeficiente de uno de los sumandos para que coincidan los exponentes, para luego sumar los coeficientes de c y el producto de $a \times b$. Por otro lado, para realizar la multiplicación de $a \times b$, la arquitectura punto fijo realiza una multiplicación de 12 bits y un desplazamiento para mantener el tamaño de los datos de entrada, mientras que para sumar, sólo suma los 12 bits de c y el producto de $a \times b$. Cabe destacar que no es necesario desplazar uno de los sumandos porque debido a la cuantización utilizada, la cantidad de bits fraccionarios es constante dentro de cada capa. Ya que la arquitectura punto flotante realiza una multiplicación de más bits que la punto fijo, sumado a que agrega una etapa más para obtener el resultado de la suma, aumentan los ciclos de latencia, y por consecuente, aumenta el tiempo de inferencia.

A partir de los datos obtenidos, podemos asegurar que la implementación punto fijo tiene un mejor desempeño que la punto flotante en la arquitectura diseñada. Aun así, esto no demuestra que el punto fijo es siempre más beneficioso que las implementaciones en punto flotante. Por ejemplo, si es que la cuantización utilizada no garantizara que la posición de la coma decimal es

constante dentro de cada capa o si la cantidad de bits punto fijo es mayor a 23, utilizar punto flotante puede ser una mejor opción para el diseño en HLS.

Capítulo 7

Resultados

En este capítulo mostramos los resultados de la implementación del acelerador hardware sobre un FPGA. Específicamente, detallamos la utilización de recursos, el consumo de potencia y los tiempo de inferencia de la arquitectura. Además, comparamos los resultados con implementaciones en otros dispositivos hardware.

7.1. Reportes de la implementación en FPGA

A continuación mostramos el reporte de uso de recursos, consumo de potencia y tiempo de inferencia de la implementación en hardware. Estos resultados fueron obtenidos en post-implementación desde el software Xilinx Vivado 2019.2 design suite.

7.1.1. Recursos utilizados

Para un análisis más detallado, dividimos el uso de recursos para cada PE y para la arquitectura completa. La tabla 7.1 muestra el uso de recursos para cada PE de la arquitectura diseñada. La tabla separa la utilización de recursos para cada acelerador, módulo y protocolos en el PE. Como podemos apreciar, cada PE de la arquitectura utiliza 125 BRAMs, 6 URAMs y 85 DSP. La mayoría de las memorias BRAM es utilizada en el acelerador de convoluciones *expansion/projection*, específicamente en los buffers que almacenan las activaciones de entrada en la etapa de almacenamiento y las sumas parciales en la etapa de convolución. El resto de las memorias BRAM son utilizadas en buffers en los aceleradores de convoluciones clásicas, com-

Acelerador	LUT	Registros	BRAM	URAM	DSP
Convoluciones clásicas	2378	1737	6	0	21
Convoluciones *depthwise*	1904	1747	0	0	18
Convoluciones *expansion/projection*	3668	1335	112	3	40
Aplanamiento	363	209	0	0	1
Componentes conectados	775	578	1	3	4
Lectura de información de control	3215	2211	6	0	1
Protocolos de comunicación	3069	5514	0	0	0
Total	15372	13331	125	6	85
Porcentaje	6.67 %	2.89 %	20.03 %	6.25 %	4.92 %

Tabla 7.1: Uso de recursos por cada PE.

ponentes conectados y en el módulo de lectura de información de control para almacenar las sumas parciales, las activaciones de entrada y la información de control respectivamente. Cabe destacar que, ya que nuestro diseño utiliza un factor de tiling para los tamaños de los mapas T_M de 28, la arquitectura utiliza LUT y registros configuradas como FIFOs para los buffers de linea de los aceleradores de convoluciones clásicas y *depthwise*. Además de BRAM, los acelerados de convoluciones *expansion/projection* y de componentes conectadas utiliza URAM para almacenar los pesos de la capa.

La tabla 7.2 muestra los recursos utilizados por la arquitectura completa. El recursos más utilizado es la BRAM, ya que cada PE las utiliza para sintetizar los buffers que almacenan las activaciones e información de control, mientras que los pesos son almacenados en memorias URAMs. Otro recursos muy utilizado son las LUTs, que nuestra arquitectura utiliza para crear las maquinas de estado que controlan la inferencia en cada PE y para implementar algunas operaciones aritmético/lógicas. El acelerador utiliza DSPs para sintetizar el resto de las operaciones matemáticas, especialmente multiplicaciones y sumas, usando casi un 20 % del total de los recursos disponibles. Además, nuestro diseño utiliza sólo un BUFG para generar la frecuencia de reloj, ya que sólo se usa un reloj para toda la PL.

La tabla 7.3 muestra el uso de recursos utilizados por toda la arquitectura diseñada, pero separada en los módulos de comunicación PS-PL y en el núcleo de procesamiento. De la tabla podemos apreciar que la arquitectura utiliza 532 BRAMs, 24 URAMs y 340 DSPs. En específico, el núcleo de procesamiento utiliza todas las memorias URAMs y bloques DSPs, ya que en esta sección del hardware es donde se almacenan los pesos y se procesa la red. Por otro lado, la comunicación PS-PL utiliza 32 BRAMs, que el AXI Master y las interfaces DMAs utilizan como buffers de entrada y salida para almacenar algunos datos y así acelerar la transmisión de

Recurso	Utilizado	Disponible	Porcentaje (%)
LUT	118233	230400	51.32
LUTRAM	12825	101760	14.57
FF	128614	460800	27.91
BRAM	532	624	85.26
URAM	24	94	25.00
DSP	340	1728	19.68
BUFG	1	544	0.18

Tabla 7.2: Uso de recursos totales.

Módulo	LUT	Registros	BRAM	URAM	DSP
Comunicación PS-PL	57072	75322	32	0	0
Núcleo de procesamiento (PEs)	61161	53292	500	24	340
Total	118233	128614	532	24	340
Porcentaje	51.32 %	27.91 %	85.26 %	25.00 %	19.68 %

Tabla 7.3: Uso de recursos de la arquitectura diseñada.

estos.

7.1.2. Latencia

La tabla 7.4 muestra las latencias y ciclos de procesamiento de cada etapa en los aceleradores de las capas de MobileNet V2. Los ciclos de cada etapa de los aceleradores de las capas de convolución muestran la cantidad de ciclos en procesar cada bloque de tiling de canal de entrada, mientras que el total corresponde al tiempo para procesar todo un bloque de tiling de canal de salida. En el caso de los aceleradores de aplanamiento y FC, la cantidad de ciclos muestra el tiempo en que se procesa cada bloque de tiling de salida. De la tabla podemos apreciar que la cantidad ciclos mínimos y máximos en los aceleradores de convoluciones clásicas, aplanamiento y componentes conectados son los mismos. Esto se debe a que como MobileNet V2 sólo utiliza estas capa una vez, la cantidad de datos a procesar en cada PE es siempre la misma, por lo que no hay variaciones en la cantidad de ciclos de procesamiento de cada etapa.

Para los aceleradores de convoluciones clásicas y *depthwise*, la lectura de pesos tiene una latencia de 9 y 10 ciclos respectivamente, completando el proceso en 16 y 17 ciclos en cada arquitectura. Esto se debe a que como se están leyendo los pesos por medio de un AXI Master,

Acelerador	Etapa	Latencia	Ciclos de procesamiento Mínima	Ciclos de procesamiento Máxima
Convoluciones clásicas	Lectura de pesos	9	16	16
	Buffers de línea	1	60	60
	Ventana de registros	2	5	5
	Procesamiento	3	1020	1020
Total				**26712**
Convoluciones *depthwise*	Lectura de pesos	10	17	17
	Buffers de línea	1	18	60
	Ventana de registros	2	5	5
	Procesamiento	10	89	908
Total				**(140 a 1001)** $\times \boldsymbol{f_o}$
Convoluciones *exp/proj*	Lectura de pesos	10	104	1032
	Procesamiento	4	53	788
Total	**(65 a 800)** $\times \boldsymbol{f_i} \times \boldsymbol{f_o} \times \boldsymbol{d} +$ **(104 a 1032)** $\times \boldsymbol{d}$			
Aplanamiento	Procesamiento	51	1632	1632
Total				**1632**
Componentes conectados	Lectura de pesos	10	2056	2056
	Procesamiento	5	4098	4098
Total				**6167** $\times \boldsymbol{d}$

Tabla 7.4: Latencia y ciclos de procesamiento en aceleradores de cada PE.

es necesario especificar la dirección de lectura desde la memoria externa, lo que agrega ciclos de latencia y aumenta el tiempo de procesamiento. El tiempo de lectura en las convoluciones *depthwise* toma un ciclo extra debido a la lectura del bias, proceso que en las convoluciones clásicas sólo es realizado en el último canal de entrada. Para las etapas de buffers de línea y ventana de registros, los tiempos concuerdan con el uso de pipeline, ya que para llenar los dos buffers deben transcurrir $2 \cdot P + 4$, con P la cantidad de píxeles a procesar en la iteración, valor que varía de 7 (tamaño mínimo de los mapas según la arquitectura de MobileNet V2) a 28 (factor de tiling) píxeles dependiendo de la capa. A este tiempo se agregan 4 ciclos para añadir los píxeles del borde del bloque. La etapa de procesamiento es completada en 1020 ciclos para las convoluciones clásicas y en 89 a 908 ciclos en las convoluciones *depthwise*. Si bien esta etapa debería tomar $P \times P$ ciclos, con tiempos teóricos de 784 ciclos para las convoluciones clásicas y de 49 y 784 ciclos en las *depthwise*, se agregan ciclos de procesamiento para comprobar el paso de la convolución, desplazar la ventana de registros, modificar la dirección de escritura en los buffers de mapas de salida, y en el caso de las convoluciones clásicas, realizar la sumatoria con las convoluciones de los canales de entrada anteriores. Por último, obtener un bloque del mapa

de salida de $28 \times 28 \times 8 = 6272$ píxeles en la convolución clásica toma 26712 ciclos, agregando ciclos extras debido a las latencias de las etapas de lectura de pesos y de procesamiento descritas, además de que cada canal de salida es el resultado de la suma de las convoluciones de los canales de entrada, como es descrito en la ecuación (3.3), teniendo que realizar tres convoluciones por los canales RGB de la imagen antes de realizar la suma. Del mismo modo, el tiempo de procesamiento de los bloques de convoluciones *depthwise* varían entre $140 \times f_o$ a $1001 \times f_o$, con f_o la cantidad de canales de salida a procesar en cada bloque, que varían de 8 a 32 según la organización de datos en cada PE.

Para el acelerador de convoluciones *expansion/projection*, la lectura de pesos tiene una latencia de 10 ciclos, que al igual que en las convoluciones clásicas y *depthwise*, se debe a la especificación de la dirección de lectura desde la memoria externa. Debido a que la arquitectura necesita conocer todos los pesos e índices del bloque de tiling para no procesar los canales con parámetros cero, la etapa de lectura de pesos toma de 104 a 1032 ciclos. Ya que estos valores que concuerdan con el tamaño de los bloques, que varían de $12 \times 8 = 96$ en las primeras capas a $32 \times 32 = 1024$ en las últimas, podemos asegurar que la directiva HLS PIPELINE esta funcionando correctamente. La etapa de procesamiento completa la convolución de un canal de salida en 53 a 788 ciclos, valores que coinciden con los tamaños de los mapas $7 \times 7 = 49$ a $28 \times 28 = 784$ respectivamente, por lo que la directiva de pipeline funciona correctamente. Cabe que estos valores son promedios entre los canales que utilizan la interfaz DMA para acceder a la memoria externa y los que no, que paralelizan el proceso. Finalmente, procesar un bloque del mapa de salida varía entre $65 \times f_i \times f_o \times d + 104 \times d$ y $800 \times f_i \times f_o \times d + 1032 \times d$ ciclos. Las variables f_i y f_o representan la cantidad de canales de entrada y salida a procesar en cada bloque, mientras que d representa la tasa de dispersión utilizada en el pruning, que en nuestra implementación tiene un valor de 0.3 en las capas de convoluciones *expansion/projection*.

En el caso del acelerador de la capa de aplanamiento, la latencia para obtener un valor de salida es de 51 ciclos, ya que como se está promediando todo un canal del mapa de características de tamaño 7×7 para obtener una componente del arreglo unidimensional, deben ser sumados 49 píxeles, proceso que toma 49 ciclos gracias al pipeline implementado. A este tiempo se agregan dos ciclos extras para leer el primer píxel y realizar la división del aplanamiento. Así, el procesamiento de un bloque de tamaño $7 \times 7 \times 32$ toma 1632 ciclos. Para el acelerador de componentes totalmente conectados, como debemos almacenar todos los parámetros para garantizar un correcto funcionamiento de los índices del pruning, la etapa de lectura de pesos almacena un bloque de 32×64 parámetros en 2056 ciclos, con una latencia inicial de 10 ciclos para especificar la dirección de lectura desde la memoria externa por la comunicación AXI Master.

En la etapa de procesamiento, el procesamiento de un bloque de 32×64 activaciones toma 4098 ciclos, tiempo que es el doble de lo esperado. Esto se debe a que HLS considera que la variable de distancia del peso no cero siguiente en la inferencia puede actualizarse en el mismo ciclo que se esta leyendo el índice del pruning, ignorando las directivas de dependencia y aumentando el intervalo de salida del pipeline de uno a dos ciclos para solucionarlo. Aun así, como la capa de componentes conectados no tiene un tiempo de procesamiento relevante en la inferencia comparada con las capas convolucionales, esta limitación no repercute significativamente en el tiempo de inferencia de la arquitectura. En total, procesar un bloque de 32×64 activaciones toma $6167 \times d$ ciclos, con d la tasa de dispersión del pruning, que para esta capa tiene un valor de 0.7.

7.1.3. Frecuencia de reloj y camino crítico

Nuestra implementación utiliza una frecuencia de reloj de 200MHz para la PL, la cual corresponde a la máxima frecuencia con la que puede operar el sistema, resultado informado en el reporte de tiempo de post-implementación de la herramienta Vivado 2019.2 design suite. El camino crítico que limita esta frecuencia se encuentra en el acelerador de convoluciones clásicas, específicamente en uno de los DSP que realiza la multiplicación de la convolución 3×3 y almacena el producto en un registro. A frecuencias mayores, la arquitectura no logra almacenar el producto antes de terminar un ciclo de reloj, por lo que se obtienen resultados erróneos al terminar la inferencia. Si bien es posible dividir la multiplicación en dos ciclos con la directiva HLS LATENCY de la herramienta Vivado HLS 2019.2, provocaría un aumento en la latencia del acelerador, afectando negativamente el tiempo de inferencia de la arquitectura, incluso aumentando la frecuencia de reloj.

7.1.4. Consumo de potencia

La tabla 7.5 muestra la potencia consumida por cada acelerador, módulo y protocolos en los PE. El acelerador de convoluciones *expansion/projection* es la sección de hardware que consume más potencia por PE, ya que es el módulo que utiliza más recursos, necesitando más señales de control por el pruning y las máquinas de estado, aumentando la potencia. Por su parte, los aceleradores de convoluciones clásicas y *depthwise* utilizan 114mW y 49mW respectivamente, ya que la máquina de estado es más pequeña, sobre todo en las convoluciones *depthwise*, que sólo aplican una convolución por canal. Del mismo modo, ya que los aceleradores de aplanamiento y

Acelerador	Potencia (W)
Convoluciones clásicas	0.114
Convoluciones *depthwise*	0.049
Convoluciones *expansion/projection*	0.203
Aplanamiento	0.004
Componentes conectados	0.027
Lectura de información de control	0.066
Protocolos de comunicación	0.113
Total	0.576

Tabla 7.5: Potencia consumida por cada PE.

Recurso	Potencia (W)
Reloj	0.569
Señales	1.180
Lógica	1.441
BRAM	0.445
URAM	0.060
DSP	0.245
PS	2.676
Total	6.616

Tabla 7.6: Potencia consumida por recurso.

de componentes conectados tienen una arquitectura más simple, sólo consumen 4mW y 27mW respectivamente. El resto de la potencia es consumida por el módulo de lectura de información de control con 66mW, y por los protocolos de comunicación, que debido al uso de señales de estado para controlar cada transferencia, consume 113mW. Así, en total, cada PE consume 576mW.

La tabla 7.6 muestra la potencia consumida por cada recurso en la arquitectura. Como podemos apreciar, el mayor consumo de potencia está en las señales de control de las máquinas de estado del hardware, en la lógica programable, como LUT, LUTRAM y registros, y en la CPU y memoria externa del PS, con 1.180W, 1.441W y 2.676W respectivamente. El resto de la potencia es consumida por el reloj, las memorias BRAM y URAM, y los bloques DSP. Así, los recursos consumen un total de 6.616W.

Finalmente, la tabla 7.7 muestra la potencia total, dinámica y estática de cada sección de hardware de nuestra arquitectura. De la tabla podemos ver que nuestro diseño consume en total

Módulo	Total (W)	Dinámica (W)	Estática (W)
Comunicación PS-PL	2.314	1.999	0.315
Núcleo de procesamiento (PEs)	2.246	1.941	0.305
CPU y memoria externa (PS)	2.779	2.676	0.103
Total	7.339	6.616	0.723

Tabla 7.7: Potencia consumida de la arquitectura diseñada.

7.339W, con 6.616W y 0.723W de potencia dinámica y estática. Específicamente, la sección del PS consume 2.779W (2.676W y 0.103W de potencia dinámica y estática), mientras que la sección de PL consume 4.560W (4.245W y 0.620W de potencia dinámica y estática). De estos resultados podemos deducir que la mayor cantidad de potencia es consumida por la lógica programable de nuestro diseño, pero no en el núcleo de procesamiento, sino que en las interfaces DMAs. Esto se debe a que en estos módulos se necesitan varias señales para iniciar, mantener, finalizar y resetear la transmisión y recepción de datos entre PS y PL. También es importante destacar que una parte importante de la potencia es consumida por la CPU para manejar y enviar las activaciones y pesos de MobileNet V2.

7.1.5. Tiempo de inferencia

Al medir el tiempo de inferencia para la implementación sobre el FPGA, nuestra arquitectura puede procesar una imagen de 224×224 píxeles a 4.54 fps (220.5ms). Este tiempo está repartido en 57.4ms en el PS para manejar y enviar los datos, y en 163.1ms en la PL para procesar la inferencia.

La tabla 7.8 muestra los tiempos de PS y PL de cada capa de MobileNet V2. De la tabla podemos apreciar que los tiempos de PS son mayores en las primeras capas de la red. Esto se debe a que mientras mayor sea el tamaño de los mapas comparado con su factor de tiling $T_M = 28$, la CPU deberá ordenar más bloques para ser enviados a los PEs, por lo que los tiempos aumentan. Esto se comprueba en el segundo módulo *bottleneck*, donde debido al factor de expansión 6 como se muestra en la tabla 3.1, las activaciones suman un total de $112 \times 112 \times 96 = 1204224$ datos, la mayor cantidad en toda la red, teniendo que enviar 16 bloques de $28 \times 28 \times 24$ píxeles a los cuatro PEs (Notar que debido a la organización espacial de los PEs, cada PE procesa un cuarto de la cantidad de canales de la capa, en este caso, $96/4 = 24$).

Respecto a los tiempos de la PL, los tiempos son mayores mientras mayor sea la cantidad de

Entrada ($MxMxN_i$)	Salida ($MxMxN_o$)	Capa	Tiempo PS (ms)	Tiempo PL (ms)	Tiempo total (ms)
224x224x3	112x112x32	Convolución clásicas	1.5	18.1	19.6
112x112x32	112x112x16	Módulo *bottleneck*	5.7	7.2	12.9
112x112x16	56x56x24	Módulo *bottleneck*	29.0	29.9	58.9
56x56x24	28x28x32	Módulo *bottleneck*	9.2	12.7	21.9
28x28x32	14x14x64	Módulo *bottleneck*	4.0	12.9	16.9
14x14x64	14x14x96	Módulo *bottleneck*	3.4	18.7	22.1
14x14x96	7x7x160	Módulo *bottleneck*	3.3	30.1	33.4
7x7x160	7x7x320	Módulo *bottleneck*	0.7	14.6	15.3
7x7x320	7x7x1280	Convolución *expansion*	0.2	12.9	13.1
7x7x1280	1x1x1280	Aplanamiento	0.2	0.3	0.5
1x1x1280	1x1x1000	Componentes conectados	0.2	5.6	5.8
1x1x1000	1x1x1000	SoftMax	0.1	0.0	0.1
MobileNet V2			57.4	163.1	220.5

Tabla 7.8: Tiempos de inferencia en cada capa de MobileNet V2.

datos de la activación a procesar, como en las primeras capas, o mientras mayor sea la cantidad de pesos, como en las últimas capas convolucionales. Para la cantidad de datos, el aumento de tiempo se debe a que mientras más datos se procesen, más ciclos se demorará la capa en entregar resultados y a que aumenta la latencia para llenar los buffers de activaciones. En el caso de la cantidad de pesos, los tiempos aumentan principalmente en las convoluciones *expansion/projection* para llenar los buffers que los almacenan antes de comenzar el procesamiento de la capa.

7.2. Escalabilidad

Si bien utilizamos 4 PEs en la implementación sobre la plataforma Zynq UltraScale+ MPSoC, esta cantidad de PEs está limitada por la cantidad de recursos disponibles, por lo que en FPGAs de mayor tamaño podríamos utilizar más PEs en paralelo y almacenar todos los datos en la memoria interna del dispositivo, disminuyendo el tiempo de inferencia. Para probar la escalabilidad de nuestro diseño, implementamos diferentes cantidades de PEs en la FPGA XCZU19EG de la familia Zynq UltraScale+ MPSoC, que dispone de 522720 LUTs, 1045440 registros, 984 BRAMs, 128 URAMs y 1968 DSPs. Debido a que en el laboratorio de VLSI de la Universidad de Concepción no disponemos de esta FPGAs, probamos el comportamiento de la escalabilidad mediante simulación.

La tabla 7.9 muestra el porcentaje de uso de recursos en la FPGA XCZU19EG al variar la cantidad de PEs. Como podemos apreciar, las LUTs y BRAMs son los recursos que limitan la

Cantidad de PEs	LUT	Registros	BRAM	URAM	DSP
4	23.28	12.15	26.63	18.75	17.28
8	46.04	23.94	53.25	37.50	34.55
12	70.58	37.45	79.88	56.25	51.83

Tabla 7.9: Porcentaje de utilización de recursos al variar la cantidad de PEs en la FPGA XCZU19EG.

Cantidad de PEs	Tiempo de inferencia (ms)	Tasa de cuadros por segundo (fps)
4	126.91	7.87
8	63.45	15.76
12	40.65	24.60

Tabla 7.10: Tiempos de inferencia al variar la cantidad de PEs en la FPGA XCZU19EG.

cantidad de PEs en la FPGA, ya que no es posible implementar más de 12 PE sin sobreutilizar estos recursos. La tabla 7.10 muestra los tiempos de inferencia con diferentes cantidades de PEs en la FPGA XCZU19EG. Como es de esperar, mientras mayor es la cantidad de PEs, menor es el tiempo de inferencia, ya que hay más procesamiento paralelo en la arquitectura. Por esto, si es que implementamos nuestra arquitectura en FPGAs de mayor tamaño, podemos alcanzar tiempos de inferencia mejores a los obtenidos con la FPGA XCZU7EV.

7.3. Análisis de resultados

7.3.1. Comparación con otros trabajos

Al comparar los resultados de utilización de recursos, potencia y tiempo de inferencia de nuestra arquitectura con los trabajos descritos en la tabla 2.2 podemos sacar diversas conclusiones. Respecto a los trabajos [36] y [37] que implementan las CNNs MobileNet V1 y V2, alcanzan tiempos de inferencia de 127 fps y 266 fps respectivamente, es decir, 28.04 y 58.72 veces más rápidos que nuestro diseño. Esto se debe a que estas arquitecturas almacenan todas las activaciones y pesos en la memoria interna del FPGA, ya que disponen de dispositivos con la suficiente cantidad de estos recursos. Gracias a esto, no requieren utilizar una CPU externa para el procesamiento, disminuyendo los tiempos de PS y la latencia en la PL al no tener que utilizar

ciclos para llenar buffers de activaciones y pesos. Además, debido a la disponibilidad de LUTs y DSPs, estas arquitecturas pueden utilizar más PEs en paralelo para acelerar la inferencia. En específico, [36] utiliza 32 PEs en paralelo, mientras que [37] utiliza 4 PEs donde cada uno de estos puede procesar 32 canales a la vez. Esto se ve reflejado al comparar la cantidad de recursos utilizados, ya que nuestra arquitectura sólo utiliza un 30 % y 28 % de las BRAMs y un 23 % y 27 % de los bloques DSPs que utilizan estos dos trabajos.

Comparado con los trabajos que implementan las CNNs AlexNet [19] y VGG-16 [13, 23] sobre FPGA, nuestra arquitectura tiene un frame-rate similar. En particular, comparado con [19], nuestro diseño utiliza un 51 % y 15 % de las BRAMs y DSPs utilizados por este trabajo, lo que se ve reflejado en el consumo de potencia, que es un 40 % menor en nuestra implementación. Sumado a esto, como en este trabajo implementan la CNN AlexNet, nuestra implementación de MobileNet V2 tiene una precisión de inferencia un 8.42 % mejor. Cabe destacar que [19] no reporta la velocidad de inferencia, por lo que no es posible comparar con nuestra implementación. Comparado con [13], nuestro diseño utiliza un 7 % más de BRAMs pero menos de la mitad de los bloques DSPs reportados por los autores. Además, este trabajo consume 2.28W más de potencia que nuestro trabajo. Respecto a [23], nuestro acelerador puede inferir dos fps más rápido, pero utiliza 438.5 y 150 BRAMs y DSPs más respectivamente, además de consumir 3.85W más de potencia. Es importante mencionar que si bien los trabajo [13, 23] alcanza 4.45 y 2.75 fps respectivamente, las arquitecturas fueron implementadas sobre los dispositivos XC7Z045 y XC7Z020 respectivamente, FPGAs de la familia Zynq-7000, de arquitectura anterior a la de la familia Zynq UltraScale+ utilizada en nuestro trabajo, por lo que si son implementadas sobre FPGAs nuevas, es posible que alcancen un mejor tiempo de inferencia, pudiendo superar la tasa de cuadros de nuestra implementación. Aun así, esto se debe a que los autores diseñaron las arquitecturas RTL manualmente en vez de usar HLS, lo que da ventajas de diseño, principalmente en uso de recursos y latencias, al tener un control absoluto de la arquitectura, por sobre tiempo de diseño que es la gran ventaja de HLS.

Comparando con [30], este trabajo puede inferir 12.9 veces más rápido que nuestra implementación, utilizando y consumiendo menos recursos y potencia. Esto se debe a que en este trabajo, los autores diseñaron específicamente una CNN para ser implementada sobre FPGAs, que reemplaza las multiplicaciones por desplazamientos de bits utilizando menos recursos lógico/aritméticos, y aplicando cuantización con funciones PACT$(\cdot)$ permitiendo utilizar datos de 1 a 4 bits. En comparación con la inferencia sobre ASIC [21], el consumo de potencia es considerablemente menor a todas las demás implementaciones, sólo consumiendo 590mW. Esto se debe a que como la arquitectura fue implementada en VLSI (del inglés *Very Large Scale Integration*,

circuito de transistores integrados de gran escala), los autores tienen mayor libertad para diseñar un hardware totalmente personalizado, eliminando recursos y protocolos que no puede omitirse en las FPGAs. Respecto al trabajo sobre la GPU NVIDIA Jetson Xavier [16], tiene un tiempo de inferencia 14.35 veces menor a nuestra implementación, lo cual se explica a que la GPU opera a una frecuencia de reloj más alta y a que utiliza a una versión de MobileNet V2 reducida, que sólo clasifica 12 clases en vez de las 1000 que utilizamos en nuestro trabajo. Aun así, la potencia consumida en esta plataforma es 1.96 veces mayor a la de nuestra implementación.

7.3.2. Comparación con software

Además de comparar con otros trabajos sobre hardware dedicado, comparamos el desempeño de nuestra arquitectura con versiones de MobileNet V2 en software usando el ambiente PyTorch con CUDA 10.2. Para esto, probamos la inferencia de la CNN sobre una GPU NVIDIA RTX 2080 y una CPU Intel i9-9900K. La inferencia en estos dispositivos toma 21.66ms (46.17 fps) y 222.1ms (4.50 fps) para la GPU y CPU respectivamente. Comparado con los tiempos de nuestro diseño, la GPU es 10.2 veces más rápida, mientras que la CPU tiene un frame-rate similar al nuestro. La ventaja que tiene nuestra arquitectura es el consumo de potencia, ya que estas plataformas consumen 215W y 95W, 29.25 y 12.93 veces más energía que la que utiliza nuestro acelerador. Esto se debe a que si bien la GPU tiene más núcleos que permiten acelerar la inferencia, estos dispositivos hardware no están diseñados solamente para este tipo de operaciones, por lo que deben pasar por más etapas de procesamiento para inferir una imagen, aumentando el uso de recursos y consumo de potencia. Debido a esto y tal como mencionamos en el Capítulo 1, GPUs y CPUs tradicionales no son utilizadas para aplicaciones sobre dispositivos embebidos, optando por inferir en FPGAs y GPUs de la familia Jetson, gracias a sus ventajas energéticas por sobre la velocidad de inferencia.

7.3.3. Discusión

De los resultados al implementar nuestra arquitectura sobre el FPGA y según las comparaciones con otros trabajos y software, podemos obtener diferentes conclusiones. Respecto a la velocidad de inferencia, nuestro diseño alcanza un frame-rate similar a las implementaciones de las CNNs AlexNet y VGG sobre FPGA y a la versión software de MobileNet V2 usando una CPU, pero más lento que la versión en GPU y las implementaciones de MobileNet V1 y V2 en hardware dedicado. Esto se debe a que como utilizamos una FPGA con recursos limitados,

nuestra arquitectura utiliza menos PEs que los trabajos [36, 37], además de almacenar las activaciones y pesos en memoria externa, teniendo que usar CPU para controlar las transferencias de memoria.

La limitante de esto es que si bien nuestra implementación puede procesar cada capa de la red en 4 PEs en paralelo, este paralelismo se ve limitado por la comunicación entre memoria externa y la PL, ya que no todos los PEs pueden acceder a ella al mismo tiempo, produciéndose un cuello de botella en la arquitectura. Aun así, si implementáramos nuestra arquitectura en un FPGA con más recursos, podríamos aumentar la cantidad de PEs en paralelo y almacenar todos los datos en la memoria interna del dispositivo, eliminando los tiempos de PS, transferencia PS-PL y latencia para almacenar las activaciones y pesos en los buffers de cada PE, disminuyendo el tiempo de inferencia.

Sin embargo, la principal ventaja de nuestro diseño tal como fue implementado es la cantidad de recursos utilizados y la potencia consumida en comparación con otras implementaciones. Específicamente, nuestro acelerador utiliza cerca de un 30 % y 25 % de las BRAMs y bloques DSPs que los trabajos [36, 37] que implementan las CNNs MobileNet V1 y V2 respectivamente. Esto es una ventaja ya que no siempre se disponen dispositivos con grandes cantidades de recursos, por lo que nuestra implementación permite realizar la inferencia en plataformas más pequeñas que las de estos trabajos.

Respecto a la potencia consumida, si bien los trabajos [36, 37] no reportan esta información, podemos comparar nuestro consumo de potencia con la de las versiones software de MobileNet V2 implementadas sobre la CPU y GPU tradicionales y la GPU Nvidia Jetson Xavier utilizada en [16]. Al realizar esta comparación, nuestro diseño presenta un consumo de potencia de 12 y 29 veces menos que el hardware tradicional, y de 1.96 veces menos que la GPU embebida. Esto permite que nuestra arquitectura pueda ser implementada en plataformas que necesiten un bajo consumo de energía, como robótica, teléfonos móviles o integrada en cámaras para la clasificación de objetos.

En síntesis, si bien nuestra arquitectura no alcanza los tiempos de inferencia de otras plataformas, como presentan una menor cantidad de recursos utilizados y consumo de potencia, es posible implementarla en dispositivos limitados en recursos hardware, garantizando un bajo consumo de potencia. Aun así, 4.5 fps pueden ser suficientes para detectar y clasificar objetos en algunas aplicaciones que no necesiten tanta velocidad, como por ejemplo clasificación de caras en dispositivos de seguridad, ya que una misma cara puede estar presente en el video por varios segundos. Por otro lado, ya que nuestra arquitectura tiene la capacidad de ser escalable,

si es que es implementada en dispositivos de mayor tamaño, los tiempos de inferencia mejoran, permitiendo que nuestro diseño pueda ser utilizado en aplicaciones que requieran una mayor tasa de fps.

Capítulo 8

Conclusión

En este trabajo diseñamos una arquitectura heterogénea de hardware dedicado tanto para la aceleración de tiempo de inferencia, como la reducción de uso de recursos y consumo de potencia en la inferencia de la CNN MobileNet V2. Nuestra arquitectura utiliza una CPU como controlador del proceso, una memoria RAM para almacenar los datos y la lógica programable de un dispositivo FPGA para implementar elementos de procesamiento que permiten la inferencia en paralelo. Cada elemento de procesamiento contiene aceleradores hardware para cada tipo de capa que utiliza MobileNet V2, como convoluciones clásicas, *depthwise*, *expansion/projection*, etapa de aplanamiento y capa de componentes totalmente conectados. La comunicación entre la CPU y cada elemento de procesamiento es realizada mediante interfaces DMA y el protocolo de comunicación AXI Master.

Para acelerar la inferencia, utilizamos diferentes técnicas que permiten reutilizar datos y reducir el tamaño de la red. Específicamente, utilizamos loop tiling para disminuir la cantidad de activaciones y pesos en la memoria interna del FPGA, además de una división espacial de la profundidad de los mapas de características para balancear la carga sobre cada elemento de procesamiento. Utilizamos la técnica de pruning con reentrenamiento sobre las capas de convolución *expansion/projection* y de componentes totalmente conectadas para eliminar pesos sin afectar considerablemente la precisión de MobileNet V2. Para reducir la complejidad de operación, combinamos la etapa de normalización con la de convolución sólo agregando un bias al proceso pero eliminando divisiones, sumas y multiplicaciones. También utilizamos la técnica de cuantización lineal para transformar los datos punto flotante a punto fijo y reducir la cantidad de bits de estos.

Implementamos la arquitectura diseñada sobre el sistema heterogéneo Xilinx Zynq Ultrascale+ MPSoC, utilizando la CPU ARM Cortex-A53 como controlador, la FPGA XCZU7EV para sintetizar cuatro elementos de procesamiento y una memoria externa DDR4 de 2GB para almacenar pesos y activaciones. Utilizamos el lenguaje de descripción de hardware de alto nivel HLS para diseñar los aceleradores y la plataforma Vivado 2019.2 para sintetizar, implementar y analizar la arquitectura sobre el FPGA. Nuestro diseño puede inferir una imagen de 224×224 píxeles a una tasa de 4.53 fps (220.5ms), consumiendo 7.34W de potencia y utilizando 532 BRAMs, 24 URAMs y 340 DSPs. Comparado con implementaciones en software usando el ambiente de desarrollo PyTorch con CUDA 10.2 sobre una GPU NVIDIA RTX 2080 y una CPU Intel i9-9900K, nuestra arquitectura tiene un frame-rate 10.18 veces más lenta que la GPU y similar a la CPU.

Comparado con otros trabajos que implementan las CNNs MobileNet V1 y V2, la velocidad de inferencia de nuestro diseño es 28.04 y 58.72 veces menor respectivamente. Esto se debe a que nuestro diseño utiliza una CPU y memoria externa para manejar, enviar y almacenar las activaciones y pesos de los núcleos de procesamiento. Nuestra implementación almacena estos datos en la memoria externa ya que la FPGA utilizada no dispone de la suficiente cantidad de recursos para almacenarlos en la memoria interna de la PL. Específicamente, la FPGA utilizada contiene 11Mb de BRAM y 27Mb de URAM, mientras que el mapa de características de mayor tamaño ($112 \times 112 \times 96$ píxeles) utiliza 14.5Mb, y que con el pruning aplicado, los pesos usan 28.2Mb. Esto provoca que el tiempo de inferencia aumente debido a que se añade tiempo para manejar las instrucciones en la CPU, para transferir los datos de la sección de PS a la PL y latencias para almacenar dichos datos en buffers en cada núcleo de procesamiento. Sin embargo, como nuestra arquitectura tiene la capacidad de ser escalable, puede ser implementada en dispositivos con más recursos, pudiendo almacenar todas las activaciones y pesos en la memoria interna, eliminando los tiempos de CPU, transferencia y latencia de escritura, además de aumentar la cantidad de PEs para procesar en paralelo. Bajo estas condiciones, si utilizamos 12 PEs en paralelo, nuestro diseño podría inferir una imagen a 24.60 fps (40.65ms), disminuyendo el tiempo de inferencia respecto a la implementación realizada en este trabajo.

Aun así, el punto fuerte de nuestra arquitectura es la cantidad de recursos utilizados y la potencia consumida. Comparado con los trabajos sobre MobileNet V1 y V2, nuestro diseño sólo utiliza un 30 % de los recursos reportados, mientras que al comparar con las versiones software implementadas sobre una GPU y CPU tradicionales, consume 29.23 y 12.93 menos potencia respectivamente. Al comparar con una GPU embebida, nuestra arquitectura mantiene un consumo de potencia menor, equivalente a 1.96 veces menos. Esto permite que nuestra arquitectura

pueda ser implementada en sistemas más pequeño, que dispongan de menos recursos y que necesiten de un bajo consumo de potencia. Además, inferir a 4.53 fps puede ser suficiente en ciertas aplicaciones, principalmente debido a que una imagen normalmente esta presente en video por varios cuadros por segundo, lo que permitiría que nuestra arquitectura pueda procesar cerca de cuatro imágenes diferentes en un segundo. Algunas de estas aplicaciones son el reconocimiento de rostros en dispositivos móviles o la clasificación de objetos en drones.

Como trabajo futuro y a modo de mejorar los resultados obtenidos, proponemos usar el protocolo Scatter-Gather de las interfaz DMA para mejorar los tiempos de transferencia de datos entre PS y PL. Esto se debe a que este protocolo utiliza memoria interna en vez de la memoria cache de la CPU para contener las instrucciones de transmisión y recepción de datos, lo que permite paralelizar la comunicación, disminuyendo los tiempos de inferencia. También proponemos utilizar un método de cuantización más agresivo, que permita disminuir aún más la cantidad de bits de activaciones y pesos, permitiendo almacenar más datos en la memoria interna. Uno de estos métodos es el uso de funciones PACT(·), que utilizan entrenamiento para encontrar un factor de cuantización para reducir significativamente el número de bits de los datos. Por último, proponemos probar la arquitectura con otro tipo de redes para medir la precisión y aceleración de estas. Para esto, sólo es necesario reordenar las capas de procesamiento, ya que cada PE contiene aceleradores para procesar varios tipos de capas que son utilizadas por otras CNN. También proponemos utilizar otras bases de datos para medir el desempeño de nuestra arquitectura.

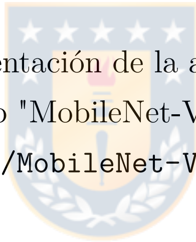
Finalmente, es importante mencionar que en el transcurso de la elaboración de esta tesis, han sido desarrolladas nuevas herramientas para diseñar e implementar la inferencia de CNNs sobre FPGAs. El ambiente de desarrollo Vitis AI de Xilinx permite utilizar frameworks como Caffe, Pytorch o TensorFlow para mapear arquitecturas de CNNs en hardware dedicado, cuantizando, aplicando pruning y optimizando cada capa de la red, y así sintetizar un DPU (del inglés *Deep Learning Processing Unit*, unidad de procesamiento de aprendizaje profundo) para ser implementado sobre FPGAs. Al comparar el desempeño de implementaciones de MobileNet V2 en Vitis AI sobre la plataforma ZCU102 [44] con nuestra arquitectura, la herramienta logra inferir una imagen a 216 fps, 47.63 veces más rápido que nuestra implementación, con un consumo de potencia similar al nuestro. Al tener estos resultados, la herramienta Vitis AI muestra un mejor desempeño que nuestra arquitectura, por lo que nuestra ventaja de consumo de potencia se pierde. Aun así, esta tesis aporta un flujo de diseño detallado para sintetizar CNNs específicas en HLS, además técnicas como la fusión de convoluciones con normalización que no son aplicadas por Vitis AI, por lo que futuros diseñadores pueden utilizar este trabajo para

desarrollar arquitecturas en HLS que puedan tener mejores resultados que los de la herramienta de Xilinx. Esto se debe a que como Vitis AI utiliza un DPU general y lo modifica para inferir en la red, se pueden añadir latencias que puedan ser descartadas por implementaciones manuales. Además, tal como ocurre al comparar el diseño de RTL tradicional con HLS, diseñar a más bajo nivel permite que el diseñador tenga un mayor control de la arquitectura.

Capítulo 9

Anexo

9.1. Repositorio



Los códigos fuente para la implementación de la arquitectura descrita en este trabajo están disponibles en el repositorio de GitHub "MobileNet-V2-inference-HLS", cuya dirección URL es: `https://github.com/ignperez-udec/MobileNet-V2-inference-HLS.git`

Bibliografía

[1] H. Lai, Y. Pan, Y. Liu, and S. Yan, "Simultaneous feature learning and hash coding with deep neural networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[2] C. Yan, H. Xie, D. Yang, J. Yin, Y. Zhang, and Q. Dai, "Supervised hash coding with deep neural network for environment perception of intelligent vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 1, pp. 284–295, Jan 2018.

[3] N. J. Nilsson, *Principles of artificial intelligence.* Morgan Kaufmann, 2014.

[4] J. Chen, V. M. Patel, and R. Chellappa, "Unconstrained face verification using deep CNN features," in *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, March 2016, pp. 1–9.

[5] M. Teichmann, M. Weber, M. Zöllner, R. Cipolla, and R. Urtasun, "Multinet: Real-time joint semantic reasoning for autonomous driving," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, June 2018, pp. 1013–1020.

[6] C. Dong, C. C. Loy, and X. Tang, "Accelerating the super-resolution convolutional neural network," in *European conference on computer vision.* Springer, 2016, pp. 391–407.

[7] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25 (NIPS)*, 2012.

[9] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations*, May 2015.

[10] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[11] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision – ECCV 2014*, 2014.

[12] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[13] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[14] NVIDIA Developer. NVIDIA TensorRT. [Online]. Available: https://developer.nvidia.com/tensorrt [Accessed: Sep. 23, 2019].

[15] S. Kim, J. Lee, S. Kang, J. Lee, and H. Yoo, "A power-efficient CNN accelerator with similar feature skipping for face recognition in mobile devices," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 4, pp. 1181–1193, 2020.

[16] S. Mohan, O. Shoghli, A. Burde, and H. Tabkhi, "Low-power drone-mountable real-time artificial intelligence framework for road asset classification," *Transportation Research Record*, 2020.

[17] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "OPU: An FPGA-based overlay processor for convolutional neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 35–47, 2020.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[19] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[20] Z. Jin and H. Finkel, "Population count on Intel® CPU, GPU and FPGA," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 432–439.

[21] S. Han, "Efficient methods and hardware for deep learning," Ph.D. dissertation, Stanford University, Stanford, 2017. [Online]. Available: https://purl.stanford.edu/qf934gh3708

[22] C. Yan, H. Xie, D. Yang, J. Yin, Y. Zhang, and Q. Dai, "Supervised hash coding with deep neural network for environment perception of intelligent vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 1, pp. 284–295, Jan 2018.

[23] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, Jan 2018.

[24] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[25] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[26] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[27] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," *arXiv e-prints*, p. arXiv:1602.07360, Feb 2016.

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

[29] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for convolutional networks," in *2009 International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 32–37.

[30] Y. Yang, Q. Huang, B. Wu, T. Zhang, L. Ma, G. Gambardella, M. Blott, L. Lavagno, K. Vissers, J. Wawrzynek *et al.*, "Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019, pp. 23–32.

[31] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[32] G. H. Golub and C. F. Van Loan, *Matrix computations.* JHU press, 2012, vol. 3.

[33] R. Xia, Y. Pan, H. Lai, C. Liu, and S. Yan, "Supervised hashing for image retrieval via image representation learning," in *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[34] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, 2019, pp. 63–72.

[35] S. Narang, E. Undersander, and G. Diamos, "Block-sparse recurrent neural networks," *arXiv preprint arXiv:1711.02782*, 2017.

[36] J. Su, J. Faraone, J. Liu, Y. Zhao, D. B. Thomas, P. H. Leong, and P. Y. Cheung, "Redundancy-reduced mobilenet acceleration on reconfigurable logic for imagenet classification," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications: 14th International Symposium, ARC 2018, Santorini, Greece, May 2-4, 2018, Proceedings 14.* Springer, 2018, pp. 16–28.

[37] L. Bai, Y. Zhao, and X. Huang, "A CNN accelerator on FPGAclock using depthwise separable convolution," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 10, pp. 1415–1419, Oct 2018.

[38] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6848–6856.

[39] D. Gajski, T. Austin, and S. Svoboda, "What input-language is the best choice for high level synthesis (HLS)?" in *Design Automation Conference*, 2010, pp. 857–858.

[40] D. G. Bailey, "The advantages and limitations of high level synthesis for FPGA based image processing," in *Proceedings of the 9th International Conference on Distributed Smart Cameras*, 2015, pp. 134–139.

[41] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, vol. 37, July 2015, pp. 448–456.

[42] S. K. Pal and S. Mitra, "Multilayer perceptron, fuzzy sets, and classification," *IEEE Transactions on Neural Networks*, vol. 3, no. 5, pp. 683–697, Sep. 1992.

[43] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv preprint arXiv:1806.08342*, 2018.

[44] M. Qasaimeh, K. Denolf, A. Khodamoradi, M. Blott, J. Lo, L. Halder, K. Vissers, J. Zambreno, and P. H. Jones, "Benchmarking vision kernels and neural network inference accelerators on embedded platforms," *Journal of Systems Architecture*, 2020.