UNS UNIVERSIDAD NACIONAL DEL SUR

2024
Bahía Blanca, Argentina

Tesis de Doctorado en Ingeniería

Circuitos Integrados de Bajo Consumo para Arquitecturas de Redes Neuronales Profundas

Nicolás Daniel Rodríguez

Director: Pedro Marcelo Julián / **Co-director:** Eduardo Emilio Paolini

Prefacio

Esta tesis se presenta como parte de los requisitos para optar al grado Académico de Doctor en Ingeniería, de la Universidad Nacional del Sur y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otra. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el ámbito del Departamento de Ingeniería Eléctrica y de Computadoras (DIEC) durante el período comprendido entre el 1 de Abril de 2018 y el 16 de Octubre de 2024, bajo la dirección del Dr. Pedro Marcelo Julian y del Ing. Eduardo Emilio Paolini, pertenecientes al Instituto de Investigaciones en Ingeniería Eléctrica "Alfredo Desages" (UNS-CONICET) y al Departamento de Ingeniería Eléctrica y de Computadoras de la Universidad Nacional del Sur.

Bahía Blanca, 16 de Octubre de 2024

Ing. Nicolás Daniel Rodríguez

UNIVERSIDAD NACIONAL DEL SUR
Subsecretaría de Posgrado

La presente tesis ha sido aprobada el ...05/03/2025....,
mereciendo la calificación de ...10 (..diez.....)

SAFE Martin Dario
Subsecretario de Posgrado
26.09.2025 12:43:47

Agradecimientos

Quisiera dedicar esta tesis a mi familia, amigos y colegas, quienes me brindaron su incondicional apoyo a lo largo de todos estos años. Agradezco profundamente a mis supervisores por su guía y apoyo durante este largo viaje, cuyo acompañamiento fue clave para la culminación de este trabajo. También quiero expresar mi gratitud a Silicon Austria Labs, ya que desde que me uní a la organización, mi investigación ha avanzado significativamente, gracias a los recursos disponibles y, sobre todo, a las incontables discusiones y sugerencias de mis colegas, cuyo profesionalismo y dedicación han sido una fuente constante de inspiración y motivación. Finalmente, deseo agradecer a mi madre, abuela y esposa, quienes son las personas más importantes e influyentes en mi vida. Sin su amor, fortaleza y sabiduría, este logro no habría sido posible.

Resumen

Esta tesis se enfoca en el desarrollo e implementación de aceleradores en circuitos integrados de uso específico (ASIC) para la ejecución eficiente de Redes Neuronales Profundas (DNN). Estas redes se caracterizan por involucrar una gran cantidad de datos, tanto de parámetros como de entradas, por lo que resulta imprescindible no solo un cómputo energéticamente eficiente, sino también un balance óptimo entre la transferencia de datos y el procesamiento. Para ello, en este trabajo se propone un algoritmo Simplicial Simétrico a Canales Separados (ChSymSim), que produce implementaciones de bajo consumo, y se optimiza una arquitectura que permite soportar la ejecución de distintos tipos de capas (diversas configuraciones de precisión, *kernel*, *stride* y *padding*) manteniendo la eficiencia energética.

Para poner en evidencia el impacto de la implementación, se realizaron dos prototipos preliminares con estructuras de prueba y evaluación (I/O de datos, buses, configuración, control) y se culminó con la fabricación de un sistema en *chip* (SoC) complejo de 9mm^2 en una tecnología de 65nm. Adicionalmente, se desarrollaron técnicas de entrenamiento con cuantización (QAT), optimizadas para funciones Simpliciales Simétricas en punto fijo.

Los experimentos realizados mostraron una eficiencia energética elevada, alcanzando valores promedios superiores a los 4 TOPS/W bajo diversas configuraciones, con un máximo de 12,12 TOPS/W. Estos resultados permiten asegurar que el acelerador ChSymSim propuesto es una solución viable para el procesamiento eficiente de algoritmos para aprendizaje de máquina (ML) en dispositivos embebidos de baja potencia.

Abstract

This thesis focuses on the development and implementation of accelerators in application-specific integrated circuits (ASIC) for the efficient execution of Deep Neural Networks (DNN). These networks involve a large amount of data, both in terms of parameters and inputs, making it essential to achieve not only energy-efficient processing but also an optimal balance between data transfer and computation. To this end, this work proposes a Channel-wise Symmetric Simplicial algorithm (ChSymSim), which yields low-power implementations, and optimizes an architecture that supports the execution of different types of layers (various configurations of precision, kernel, stride, and padding) while maintaining energy efficiency.

To highlight the impact of the implementation, two preliminary prototypes were developed with testing and evaluation structures (data I/O, buses, configuration, control), culminating in the fabrication of a complex system-on-chip (SoC), with an area of 9mm^2 in a 65nm technology. Additionally, quantization-aware training (QAT) techniques were developed, optimized for Symmetric Simplicial functions in fixed-point arithmetic.

The experiments conducted showed high energy efficiency, achieving average values above 4 TOPS/W under various configurations, with a maximum of 12,12 TOPS/W. These results confirm that the proposed ChSymSim accelerator is a viable solution for the efficient processing of machine learning (ML) algorithms in low-power embedded devices.

Índice general

Índice de figuras

Índice de tablas

Acrónimos

ML Aprendizaje de máquina - *Machine Learning* 31

NN Red neuronal - *Neural network* 31

FC *Fully-connected* 32

CNN Red neuronal convolucional - *Convolutional neural network* 32

GPU Unidad de procesamiento gráfico - *Graphics Processing Unit* 32

MAC Multiplicación y acumulación 33

DNN Red neuronal profunda - *Deep neural network* 33

IoT Internet de las cosas - *Internet of Things* 33

ASIC Circuito integrado de uso específico - *Application-Specific Integrated Circuit* 33

FPGA *Field-Programmable Gate Array* 33

PWM Modulación por ancho de pulsos - *Pulse-width modulation* 71

SymSim Simplicial Simétrica - *Symmetric Simplicial* 74

SE Elemento Estructurante - *Structuring Element* 85

Inf Valor Infinito 93

NaN *Not a Number* 93

AF Función de Activación - *Activation Function* 98

SM *Shape Mask* 120

MUX Multiplexor 144

QSPI *Quad Serial Peripheral Interface* 169

P&R *Place and Route* 171

GPIO *General Purpose Input/Output* 172

AMBA *Advanced Microcontroller Bus Architecture* 172

AHB *Advanced High-performance Bus* 172

API *Application Programming Interface* 194

SDK *Software Development Kit* 193

PLL *Phase-Locked Loop* 195

NoC *Network-on-Chip* 217

LSB Bit menos significativo - *Least Significant Bit* 258

MSB Bit más significativo - *Most Significant Bit* 258

SR *set/reset* 274

Capítulo 1

Introducción

1.1. Problemática y estado del arte

Durante las décadas pasadas, los algoritmos de Aprendizaje de Máquina (ML[1]) y las Redes Neuronales (NN[2]) se volvieron populares a la hora de resolver tareas complejas, tales como reconocimiento y clasificación de imágenes, segmentación, regresión, entre otras, impulsados por el crecimiento exponencial de las capacidades de las computadoras. Estos algoritmos proveen un modelo matemático para representar el comportamiento de neuronas cuando son excitadas por impulsos eléctricos a través de sus dendritas. Cada neurona se modela como un sumador de entradas pesadas, y una función no lineal de salida. El número promedio de entradas a una neurona real es de miles o decenas de miles [10]. Esto resulta en un número excesivo para implementar electrónicamente incluso utilizando tecnologías avanzadas de circuitos integrados, lo que ha producido diferentes enfoques para la implementación de NNs.

Uno de estos enfoques ha sido el de plantear arquitecturas compuestas por arreglos regulares de celdas de procesamiento, con un número bajo de entradas y solo conexiones locales a las neuronas vecinas [11], lo que permite procesar numerosos datos en paralelo y de forma rápida al reducir la necesidad de una comunicación global. Por otra parte, debido a la reducida actividad que producen, son en general esquemas de un consumo de energía inherentemente bajo, en contraste con las arquitecturas

[1] Aprendizaje de máquina - *Machine Learning*
[2] Red neuronal - *Neural network*

tradicionales en las cuales todos los datos son transmitidos a un procesador central que se ocupa de realizar los cálculos requeridos. Este enfoque se encuadra dentro de los métodos neuromórficos, es decir que está inspirado en la naturaleza biológica de las neuronas, y ha sido utilizado principalmente para el filtrado de imágenes y la realización de operaciones morfológicas [12]. Mediante extensiones para clasificación de imágenes y utilizando diversas capas de NNs, estos modelos han evolucionado hasta convertirse en las arquitecturas de Redes Neuronales Convolucionales (CNN[3]) como las conocemos hoy en día [13], con la identificación de dígitos escritos a mano como uno de los ejemplos más conocidos de aplicación.

En base a la disponibilidad de mayores recursos computacionales, en la forma de placas de GPU[4] o servidores dedicados, se plantearon estructuras más ambiciosas y con varias capas sucesivas de NNs [14], que generalmente se componen de numerosos filtros convolucionales seguidos de neuronas con conectividad completa (capas FC[5]). Este tipo de estructuras alcanza fácilmente decenas o centenas de millones de parámetros, que deben ser ajustados mediante algoritmos de optimización en base a una cierta cantidad de imágenes de entradas y sus salidas deseadas. Desde la primera red de estas características (AlexNet) [15], se han propuesto extensiones y modificaciones, las cuales han permitido mejorar notablemente las propiedades de clasificación [16–18]. Estas aplicaciones se han vuelto un elemento clave en el abanico de arquitecturas de procesamiento del tipo *Big Data*, donde grandes cantidades de datos deben ser procesados en tiempo real, generalmente mediante centros de servidores que tienen un consumo energético elevado. En la actualidad, los sistemas de mayor rendimiento [19] alcanzan el Exaflop/s o EFLOPS (10^{18} operaciones de punto flotante por segundo) pero con consumos superiores a los 20 MW, mientras que aquellos más enfocados a la eficiencia energética [20] (superando los 60 GFLOPS/W) aún presentan grandes consumos de varias decenas o cientos de kW. Estos consumos de potencia se traducen en costos de alrededor de 35 millones de dólares en electricidad anual para los sistemas de mayor rendimiento y cientos de miles de dólares para los de mayor eficiencia, con costos de operación y mantenimiento que ya supera largamente el costo de capital. De hecho, el grueso del costo mayormente re-

[3] Red neuronal convolucional - *Convolutional neural network*
[4] Unidad de procesamiento gráfico - *Graphics Processing Unit*
[5] *Fully-connected*

side en costos edilicios y de refrigeración. Son estas infraestructuras computacionales las que soportan muchas de las funciones "inteligentes" actualmente disponibles en los teléfonos celulares, cuyo ejemplo más clásico es la interpretación y traducción de voz. De manera que hay un interés creciente en el desarrollo de circuitos integrados más eficientes desde el punto de vista energético que sean capaces de llevar a cabo, al menos parcialmente, estas funciones.

Es debido a estos costos de procesamiento con grandes centros de datos, junto a la necesidad de minimizar las latencias presentes en la comunicación con estos, que surge el paradigma de aplicaciones *edge* y Internet de las Cosas (IoT[6]), entre los que se encuentran ejemplos tales como agricultura inteligente para riego óptimo [21], detección de anomalías en la salud de pacientes [22], navegación autónoma y semi-autónoma [23], entre otras. En estos entornos, se requiere agregar "inteligencia" a sistemas embebidos de manera que cálculos complejos como los de Redes Neuronales Profundas (DNN[7]) puedan ser realizados en los mismos dispositivos, minimizando así la comunicación a un intercambio de datos entre sensores y unidades de procesamiento conectadas a una red local. De esta forma, si se utiliza *hardware* de propósito general, como por ejemplo microcontroladores, se reduce el consumo energético pero se presentan extensos tiempos de cómputo, mientras que en el otro extremo, con GPUs embebidas se aceleran notablemente los tiempos de cómputos pero a costa de un consumo energético excesivo (decenas de *watts* que podrían drenar la batería del dispositivo en unos pocos minutos).

Es por esto que encontrar un balance entre la capacidad de cómputo y el consumo energético se vuelve crucial en este tipo de aplicaciones, lo que suele requerir de aceleradores dedicados en circuitos integrados (ASIC[8]), dado que una FPGA[9] posee gran versatilidad (ideal para la realización de prototipos) pero con consumos de potencia elevados. Existen múltiples ejemplos de esto en la literatura. En [24] se implementa un arreglo 2D de multiplicadores y acumuladores (MAC[10]), con precisión variable de 4 bits, 8 bits y 16 bits. El circuito logrado se fabricó en una tecnología

[6] Internet de las cosas - *Internet of Things*
[7] Red neuronal profunda - *Deep neural network*
[8] Circuito integrado de uso específico - *Application-Specific Integrated Circuit*
[9] *Field-Programmable Gate Array*
[10] Multiplicación y acumulación

FDSOI de 28nm en 1,87mm^2 y alcanza una eficiencia entre 0,26 y 10 TOPS/W. El sistema descrito en [25] puede implementar filtros de tamaño 5×5 y 7×7. El *chip* resultante (Origami) fue fabricado en una tecnología CMOS[11] UMC 65nm y utiliza tanto pesos como entradas de 12 bits. En la configuración de alta velocidad (500MHz) alcanza un pico de 196 GOPS, mientras que en términos de eficiencia los autores reportan 437 GOPS/W con 1,2V@500MHz y 803 GOPS/W para 0,8V@189MHz. El *chip* Eyeriss v2 [26] está fabricado en TSMC 65nm LP y utiliza una precisión de 8 bits para entradas/pesos y 20 bits para sumas de productos parciales. Las convoluciones se realizan por medio de 192 elementos de procesamiento que pueden ser configurados para computar distintos modelos, como por ejemplo AlexNet, con la que se reporta una eficiencia de 253,2 GOPS/W. [27] utiliza una estructura MAC con precisión de punto fijo de 24 bits. Los autores manifiestan lograr una precisión de reconocimiento similar a la lograda con 64 bits en punto flotante con doble-precisión. El *chip* fue fabricado en un *die* de 4×4mm^2 en 65nm, con una eficiencia de 1,42 TOPS/W (1,2V@125MHz) ante 16 bits de entradas y pesos. [28] implementa un procesador general en 65nm capaz de lograr 8,1 TOPS/W CNN o RNN (Recursive Neural Networks). Los autores muestran que una precisión dinámica con un mecanismo adaptativo y una longitud de palabra de 4 bits logra resultados similares a utilizar 32 bits en punto flotante. [29] presenta un acelerador fabricado en UMC 65nm y optimizado para CNN de pesos binarios que alcanza 1,5 TOPS a 1,2V. El circuito integrado presenta un área de 1,9 mm^2 y un consumo de potencia de 895μW. El motor convolucional soporta *kernels* de varios tamaños (entre 1×1 y 7×7).

1.2. Objetivo

Teniendo en cuenta esta problemática establecida en la Sección 1.1, el objetivo principal de esta tesis es explorar y desarrollar unidades de cómputo en *hardware*, para ser implementadas en circuitos integrados (IC[12]) eficientes (priorizando en términos de bajo consumo) y que sean capaces de resolver Redes Neuronales Profundas (DNN) en sistemas con recursos limitados. Las funciones simpliciales son buenas

[11] *Complementary Metal Oxide Semiconductor*
[12] Circuito integrado - *Integrated Circuit*

candidatas para resolver Redes Neuronales (NN) con un consumo reducido, ya que estas cuentan con estructuras de cómputo eficientes en *hardware* [30–37]. Sin embargo, tales implementaciones se han limitado hasta el presente a pequeños arreglos que procesaban filtros simples o determinadas operaciones morfológicas (sin capacidad de aprendizaje), agrupando entradas en vecindarios de a lo sumo 3×3 elementos. Dichas arquitecturas fueron pensadas para hacer un solo filtro o capa, en lugar de aplicaciones de mayor tamaño como podría ser el cómputo de NNs completas, por lo que resulta necesario modificar estas estructuras de cómputo, añadiendo funcionalidades requeridas para el cálculo de NNs, tales como tamaño de *kernel* configurable, *stride* y *padding*, pero evitando agregar complejidad en términos de *hardware* que anulen los beneficios en eficiencia que presentan estas estructuras.

Como punto de partida para esta mejora u optimización, se debe elegir alguna variante de las funciones simpliciales que pueda emplearse para el cómputo de DNNs. En principio, la función simplicial pura sería una buena opción dada su versatilidad a la hora de representar un gran abanico de funciones, siempre que sean lineales a tramos, pudiendo también representar a cualquier función lineal estándar en capas FC o convolucionales. Desafortunadamente, los parámetros o pesos requeridos para el cómputo de este tipo de operación escala exponencialmente con el número de parámetros. Las capas FC generalmente presentan un enorme número de entradas, mientras que las convencionales, que fueron ideadas para reducir la cantidad de entradas por operación, pueden utilizar filtros con *kernel* de dimensiones 5×5 o 7×7, lo que incrementa notablemente la cantidad de memoria que se requiere para guardar los parámetros de una función simplicial pura.

Considerando que la cantidad de memoria en chip es limitada y que las operaciones para introducir o sacar datos de memorias externas son las que producen un mayor consumo energético, la idea de utilizar a la función simplicial en su versión simétrica resulta más atractiva, ya que esta requiere de una cantidad de parámetros similar a las funciones lineales (FC o convolución) que tradicionalmente se usan en Redes Neuronales. En función de lo expuesto, para cumplir con el objetivo de generar *hardware* eficiente para el cómputo de DNNs, se utilizarán funciones simpliciales simétricas, explorando sus capacidades y limitaciones en entornos de ML, y realizando las modificaciones u optimizaciones que se requieran para dicha tarea.

1.3. Contribución

Una parte central del trabajo de esta tesis corresponde al análisis y evaluación de las ya conocidas funciones simpliciales y simpliciales simétricas, tanto algoritmos como unidades de procesamiento en *hardware*, para su uso en redes neuronales y entornos de ML, incluyendo comparaciones con algoritmos y operadores estándar tales como convoluciones y capas FC. Las principales contribuciones de esta tesis, clasificadas de acuerdo al nivel de abstracción, se describen a continuación.

Algoritmos y *software*:

- Desarrollo del cálculo de gradientes para las funciones simpliciales, diseñando así algoritmos de *backpropagation* que permiten su entrenamiento tanto en ambientes de simulación propios (para análisis preliminares), como entornos de ML y DNN preexistentes (como el caso de PyTorch). Debido a que el uso de capas simpliciales puramente simétricas presenta fuertes restricciones que limitan las capacidades de los modelos de DNN, se proponen mejoras al algoritmo que aún siendo de fácil implementación, logran aliviar y hasta eliminar dichas restricciones. Con estas mejoras se propone la variante simplicial simétrica a canales separados (ChSymSim[13]) que procesa cada canal de entrada con una función simplicial simétrica diferente, para luego sumar la contribución de todos los canales procesados.

- Desarrollo de librerías de código (Matlab® y Python) que permiten simular el comportamiento de estas funciones y operadores al verse afectados por cuantización en sus entradas y parámetros. Si bien existen librerías que permiten ejecutar DNNs con operaciones de enteros (punto fijo), esto sólo es posible con cuantizaciones estándar de 64, 32, 16 y como mínimo 8 bits. Para una mejor comprensión de los algoritmos y dada la ausencia de soporte para la ejecución de DNNs con cuantizaciones de fracciones de byte, se produjeron módulos, clases y funciones que replican tanto las funciones estándar de convolución y lineales (FC), como simpliciales simétricas, añadiendo la posibilidad de tener un número de bits de cuantización configurable para entradas, parámetros y

[13]Simplicial Simétrica a Canales Separados - *Channel-wise Symmetric Simplicial*

salidas. Estos códigos permiten el uso de rango estático o dinámico, dependiendo de si se entrena desde cero o se realiza un ajuste fino con factores de escala como potencias de 2, lo que permite implementar el re-escalado de las salidas en *hardware* con un simple *shift* aritmético.

Arquitecturas y modelos de alto nivel:

- Desarrollo de modelos de alto nivel en Python para bloques de *hardware* que, sin necesidad de calcular la operación en sí misma ni realizar simulaciones con código RTL[14], permiten estimar los tiempos de cómputo, área y consumo energético. Comparación entre operadores lineales estándar y simpliciales, observando los casos en los que resultan más eficientes.

- Análisis a nivel sistema del comportamiento del procesador simplicial al ejecutar capas particulares de redes neuronales. Aplicación de estas herramientas para la optimización de implementaciones en *hardware* del operador simplicial simétrico, evitando la repetición de cálculos y las transferencias de datos innecesarias.

Implementaciones en *hardware*:

- Implementaciones preliminares con una sola unidad de procesamiento para la función simplicial simétrica. Esto requiere el desarrollo de modelos en RTL que incluyan algunas de las mejoras propuestas por esta tesis, tales como tamaño de *kernel* variable, *bias*, producto de entradas por pesos binarios, cómputo por canal, etc. Se emplean optimizaciones como enmascarado y habilitación de señales que minimizan la actividad de los nodos del circuito y que, en conjunto con estrategias estándar como *clock-gating*, permiten reducir aún más el consumo energético del procesador simplicial simétrico.

- Fabricación de dos circuitos integrados utilizando estas versiones preliminares de un único elemento de procesamiento simplicial simétrico. Estos ICs son en esencia SoC[15]s que incorporan memorias SRAM[16], microprocesador y periféri-

[14] Abstracción a nivel de registros - *Register Transfer Level*
[15] Sistema en Chip - *System on Chip*
[16] Memoria de acceso aleatorio estático - *Static Random Access Memory*

cos, para poder configurar, ejecutar y evaluar al operador simplicial simétrico propuesto.

- Diseño de una arquitectura de procesamiento simplicial simétrico con numerosos elementos de procesamiento dispuestos en forma de arreglo "cuadrado". De esta forma, se aprovecha mejor la reutilización de recursos y se reducen las repeticiones de cómputos, en relación a las implementaciones como "columna" ya existentes. Para complementar el diseño de la arquitectura, al arreglo de PE[17]s se le sumaron otros módulos tales como controladores, registros de configuración, una unidad de escalado, redondeo y ReLU[18], entre otros bloques de hardware que ofrecen un mayor grado de optimización en cuanto a diversos modos de operación adaptados a configuraciones de capas específicas, que varían en tamaño de *kernel*, *stride* y *padding*.

- Fabricación de un SoC de tamaño y funcionalidad superior a sus dos versiones preliminares. Manteniendo el microprocesador, las memorias junto a los demás módulos y periféricos fueron expandidos y modificados para permitir mejores y más rápidas transferencias de datos, necesarias para la ejecución de DNNs. El procesador simplicial simétrico integrado en este SoC incluye comunicación directa con controladores de DMA[19], lo que permite operaciones más rápidas y eficientes, con el microprocesador principal limitándose a simplemente configurar los módulos necesarios para la operación.

- Exploración de tecnologías no convencionales, tales como lógica dinámica, para la implementación de módulos fundamentales de la arquitectura simplicial simétrica, como es el caso del comparador. Dicho comparador optimizado para bajo consumo representa el cuello de botella del circuito en cuanto a velocidad, por lo que se realizaron simulaciones de este bloque con lógica dinámica que mostraron mejoras en cuanto a consumo energético y tiempos de operación máximos, respecto a la implementación tradicional con lógica CMOS.

[17]Elemento de procesamiento - *Processing Element*
[18]Unidad de rectificación lineal - *Rectified Linear Unit*
[19]Acceso directo a memoria - *Direct Memory Access*

1.4. Organización

Este primer capítulo ofrece una introducción al tema de estudio, mediante una revisión bibliográfica que describe el contexto y la problemática sobre la cual se desarrolla esta tesis, a la vez que se plantea el objetivo general a abordar a lo largo de los siguientes capítulos. Además, se presenta una lista de los aportes realizados durante el desarrollo de este trabajo.

El Capítulo 2 introduce conceptos preliminares que apoyan el desarrollo de los contenidos de los siguientes capítulos, desde nociones básicas de procesamiento de imágenes, *Machine Learning* y Redes Neuronales, qué algoritmos presentan y cómo se entrenan, hasta métodos de cuantización y casos especiales como operadores morfológicos. Además, en dicho capítulo se introduce todo lo desarrollado hasta el momento sobre algoritmos simpliciales y simpliciales simétricos, lo que constituye el punto de partida de lo trabajado en la tesis.

El Capítulo 3 describe las mejoras y actualizaciones realizadas a los algoritmos simplicial y simplicial simétrico (desde un punto de vista de *software*), así como también su método de entrenamiento, con el fin de una correcta integración en entornos de Redes Neuronales. Al final de este Capítulo se detallan ejemplos de Redes Neuronales Profundas que hacen uso de capas simpliciales simétricas y que ilustran, además, la evolución de dichas capas para mejores resultados en clasificación de imágenes.

El Capítulo 4 presenta el modelado del *hardware* requerido para el procesamiento de los nuevos algoritmos simpliciales introducidos en el Capítulo 3. Se realizan, además, estimaciones de tiempos de procesamiento, área, consumos de potencia y energía, con el objetivo de hallar arquitecturas de procesamiento óptimas con las que implementar capas simpliciales en Redes Neuronales Profundas, concluyendo con la comparación con una arquitectura de cómputo lineal tradicional, tanto con modelos de alto nivel como con resultados de síntesis y simulaciones.

El Capítulo 5 describe las implementaciones en *hardware* de las arquitecturas simpliciales introducidas en el Capítulo 4, junto a los sistemas empleados para su evaluación y uso. A continuación, se muestran los distintos circuitos integrados (SoC) desarrollados como plataformas de evaluación de estos procesadores simpli-

ciales simétricos y los resultados experimentales obtenidos con los mismos.

Finalmente, el Capítulo 6 presenta las conclusiones generales y destaca los trabajos futuros necesarios para continuar con esta línea de investigación.

1.5. Metodología

La metodología utilizada durante el desarrollo de esta tesis consistió en:

- propuesta, evaluación y simulación de algoritmos de procesamiento para entornos de ML;
- definición de arquitecturas de procesamiento en base a estimaciones de tiempo, área y consumo de las implementaciones de los algoritmos propuestos;
- desarrollo de código RTL de las arquitecturas seleccionadas, simulación y verificación funcional;
- síntesis con librerías de celdas estándar y posterior implementación en VLSI[20] de las arquitecturas de procesamiento elegidas, incluyendo simulaciones posteriores a ambos pasos para confirmar funcionalidad y estimar consumo energético;
- fabricación, pruebas y mediciones de los ASIC desarrollados, haciendo foco en el desempeño de la arquitectura de procesamiento de DNNs propuesta.

1.6. Publicaciones

Empleando dicha metodología, los resultados obtenidos durante el desarrollo de la tesis dieron origen a las siguientes publicaciones:

- en [38] se presentaron los primeros experimentos del entrenamiento de DNNs con capas simpliciales simétricas para la aproximación de funciones genéricas y clasificación de imágenes en bases de datos sencillas como MNIST [39];

[20] Circuitos integrados de gran escala/complejidad - *Very Large Scale Integration*

- en [40] se realizó un análisis en cuanto a la aproximación de funciones genéricas con estructuras simpliciales simétricas, incluyendo la combinación de estas estructuras con rotaciones en el dominio de entrada para obtener mayores grados de libertad;
- en [9] se detalla el algoritmo de *backpropagation* y se muestran resultados de entrenar una red con capas simpliciales simétricas y conexiones residuales, para la clasificación de imágenes de GTSRB[21] [2];
- en [41] se exploran los errores producidos por la cuantización en capas simpliciales, presentando resultados en la clasificación de imágenes aéreas de UC Merced Land Use [6], comparando con estructuras de cómputo lineales tanto en cuanto a algoritmos como en términos de área y consumo de las unidades de procesamiento elementales;
- en [8] se profundizó en la clasificación de imágenes satelitales (en este caso de USTC SmokeRS [7]), realizando una comparación de desempeño entre los modelos ResNet con capas ChSymSim vs. capas convolucionales tradicionales;
- en [42] se introdujo un análisis de arquitecturas de procesamiento ChSymSim, con múltiples unidades de cómputo para la obtención de numerosas salidas en paralelo, considerando principalmente los tiempos de cómputo y retardos introducidos por la carga de datos;
- en [43] se presentó el primer SoC (DigineuronV1) fabricado como prototipo para la evaluación del procesador ChSymSim, que integra la primera prueba de concepto de este acelerador con una sola unidad de procesamiento;
- en [44] se mostraron los resultados experimentales obtenidos con el segundo *chip* prototipo (DigineuronV2), comparando con la versión anterior y analizando las mejoras y correcciones introducidas en el acelerador ChSymSim;
- en cuanto al SoC final de esta tesis (DigineuronV3a), tanto los detalles de su diseño como sus resultados experimentales se encuentran en proceso de publicación.

[21] *German Traffic Sign Recognition Benchmark*

Capítulo 2

Conceptos Preliminares

En este capítulo se presentan los conceptos fundamentales que establecen las bases para el desarrollo de algoritmos, la exploración de arquitecturas y el diseño de *hardware*, que se tratarán en los capítulos posteriores. Estos conceptos son esenciales no solo para implementar soluciones eficientes, sino que también ofrecen un marco teórico que permita comprender los desafíos técnicos abordados en este trabajo, a la vez que facilitan el entendimiento de las estrategias utilizadas y las contribuciones desarrolladas a lo largo de esta tesis.

De esta forma, la Sección 2.1 estará dedicada a proporcionar nociones básicas sobre el procesamiento de imágenes y el aprendizaje automático (*Machine Learning*), cubriendo aspectos como el perceptrón, las funciones de activación, el algoritmo de *backpropagation*, las funciones de convolución y normalización, así como los operadores morfológicos, entre otros. Estos conceptos sientan las bases para el funcionamiento de las Redes Neuronales modernas, lo que permite el procesamiento eficiente de datos complejos, especialmente en tareas de reconocimiento de patrones y clasificación de imágenes. Posteriormente, en la Sección 2.2, se detallarán las principales técnicas de cuantización, las cuales son cruciales para el diseño de circuitos integrados digitales y sistemas de bajo consumo energético. Finalmente, la Sección 2.3 se centrará en describir las funciones simpliciales y simpliciales simétricas, que representan los pilares fundamentales para el desarrollo de las arquitecturas de procesamiento eficientes propuestas en esta tesis.

2.1. Procesamiento de imágenes y *Machine Learning*

El procesamiento de imágenes no solo ayuda en la preparación de los datos (preprocesamiento y *data augmentation*), sino que también permite extraer y transformar información visual en representaciones que los modelos de aprendizaje de máquina (ML) puedan interpretar de manera efectiva. En particular, en las arquitecturas de Redes Neuronales Convolucionales (CNN), el procesamiento de imágenes actúa como un componente activo durante el entrenamiento, donde operaciones como la convolución y el agrupamiento (*pooling*) permiten que el modelo identifique patrones complejos, como bordes, texturas y formas en diferentes escalas y niveles de abstracción [45]. Este tipo de procesamiento, que se lleva a cabo de manera iterativa en las distintas capas de la red, es lo que permite a los modelos de ML extraer características relevantes de manera jerárquica y aprender representaciones profundas de las imágenes, adaptándose así a tareas complejas como la clasificación de imágenes [15]. De este modo, el procesamiento de imágenes constituye una parte central en la construcción y mejora del conocimiento que los modelos de Redes Neuronales adquieren sobre las características visuales, especialmente cuando se incrementa la "profundidad" de los mismos [14].

Entre las tareas más complejas relacionadas con el procesamiento de imágenes en el dominio de ML, se encuentran ejemplos como la detección y el reconocimiento de objetos, la segmentación, la clasificación y la regresión. En la detección y reconocimiento de objetos, no solo se debe identificar la presencia de un objeto en una imagen, sino también localizarlo y clasificarlo dentro de múltiples categorías, lo que es esencial para aplicaciones como la conducción autónoma [46] o la identificación de personas mediante cámaras de seguridad [47]. La segmentación lleva este análisis un paso más allá al dividir una imagen en múltiples regiones o "segmentos", asignando a cada píxel una etiqueta que indica a qué objeto o parte de la escena pertenece, siendo fundamental en campos como la medicina, donde se requiere identificar con precisión estructuras anatómicas en imágenes de diagnóstico [48]. Por su parte, la clasificación de imágenes asigna etiquetas a imágenes completas basadas en las características aprendidas por el modelo, lo que es útil en sistemas de recomendación

visual o identificación automática de elementos en grandes bases de datos [18]. Finalmente, la regresión aplicada al procesamiento de imágenes permite predecir valores continuos a partir de entradas visuales, como en el caso de la estimación de la profundidad en imágenes 2D [49] o la predicción de variables cuantitativas en análisis de materiales [50] o clima [51]. Estas tareas ilustran la capacidad del procesamiento de imágenes en combinación con estrategias de ML no solo para identificar patrones, sino también para interpretar información compleja y generar predicciones precisas.

2.1.1. Perceptrón y funciones de activación

En el contexto de ML y el procesamiento de imágenes, el perceptrón representa una de las estructuras de cómputo más importantes en el desarrollo de Redes Neuronales. Este modelo, introducido en 1957 por [52], se basa en un esquema simple de una sola neurona artificial, o varias de estas agrupadas, que realiza una tarea de clasificación binaria. Aunque el perceptrón original es limitado en su capacidad para resolver problemas no lineales, su concepto fundamental ha sido la base sobre la cual se han construido arquitecturas más complejas utilizadas en el procesamiento de imágenes y aprendizaje automático. La neurona artificial en este modelo (Fig. 2.1) opera mediante la combinación lineal de entradas ponderadas o, en otras palabras, la suma pesada de entradas, seguido de una función de activación, lo que le permite asignar una etiqueta a las entradas basadas en una función de umbral o activación definida.

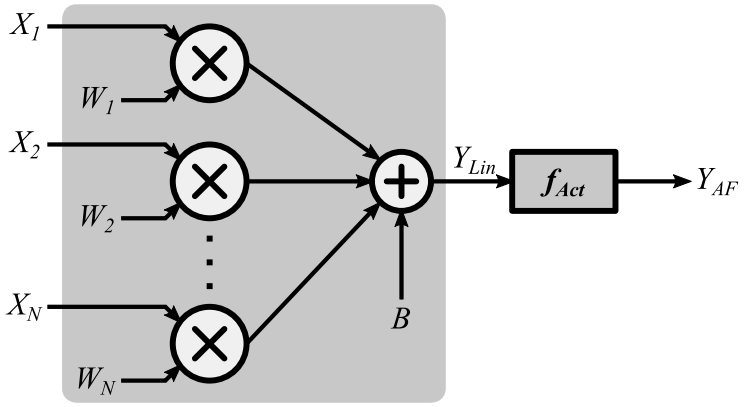


Figura 2.1: Diagrama en bloques del modelo de neurona artificial.

Por su parte, las funciones de activación introducen las no linealidades esencia-

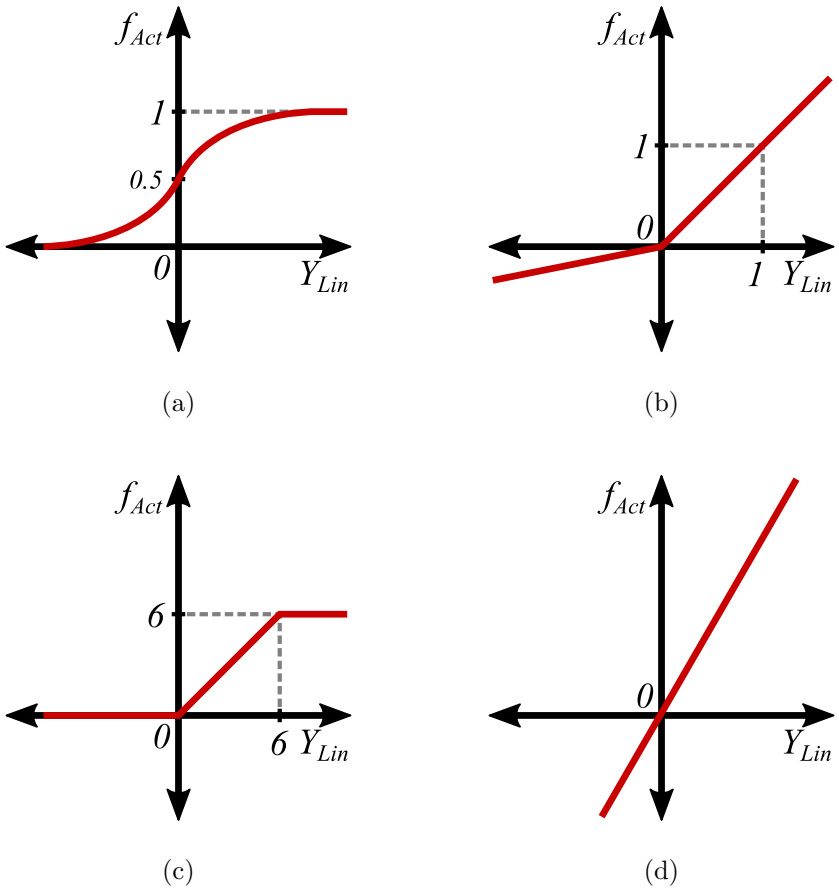


Figura 2.2: Ejemplos de funciones de activación: a) Sigmoide; b) PReLU; c) ReLU6; d) Lineal.

les en las Redes Neuronales, lo que permite que el modelo aprenda y represente relaciones complejas en los datos. Entre las funciones de activación más comunes se encuentran la función sigmoide, la tangente hiperbólica (*tanh*) y la familia de funciones rectificadas linealmente (ReLU). Cada una de estas funciones de activación, ilustradas en la Fig. 2.2, posee características específicas que influyen en el rendimiento y la convergencia del entrenamiento. Un ejemplo de esto es la función ReLU que simplemente anula los valores negativos obtenidos de la suma ponderada de las entradas a una neurona, lo que ha demostrado ser particularmente eficaz en DNNs debido a su capacidad para mitigar el problema de *vanishing gradient*[1] y acelerar la convergencia del modelo de red [53]. Otras variantes muy utilizadas de la función ReLU son las conocidas como PReLU[2] (Parametric ReLU) y ReLU6. Por

[1]Desvanecimiento de gradiente: el gradiente propagado es atenuado en cada capa, impidiendo que las primeras capas de la red ajusten sus parámetros.

[2]*Parametric Rectified Linear Unit*

un lado, la función PReLU introduce un parámetro adicional que multiplica a los valores negativos de la salida de una capa (en vez de anularlos como la activación ReLU tradicional) y que puede ajustarse durante el entrenamiento. Esto permite que la capa siguiente de la red sea capaz de utilizar (en cierta medida) el rango negativo de las salidas de la capa previa, lo que puede mejorar la capacidad de la red para aprender características más complejas [5]. Por otro lado, ReLU6 es un caso particular de saturación del rango positivo de las salidas de una capa, que limita la salida de la función ReLU a un máximo de 6, lo que puede ser beneficioso para mantener la estabilidad numérica en redes profundas y evitar problemas de explosión de gradientes [54]. En caso de no utilizarse función de activación alguna o aplicarse un simple factor de escala, la secuencia de capas de neuronas equivaldría simplemente a un gran mapa lineal. Es debido a esto que las funciones de activación son fundamentales para que los modelos de redes neuronales puedan realizar tareas complejas y aprender representaciones profundas a partir de datos visuales y otras formas de información.

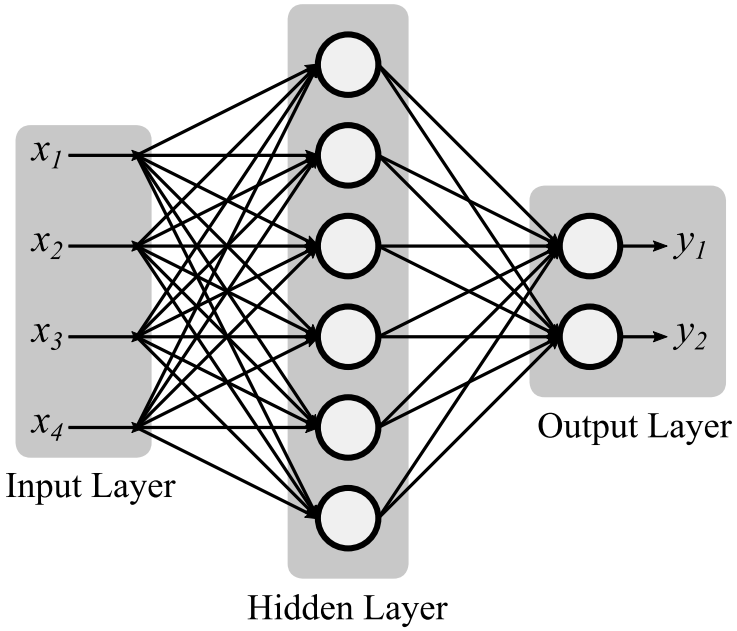


Figura 2.3: Red Neuronal poco profunda (*Shallow Network* o SLFN).

Las neuronas artificiales, como las empleadas en el perceptrón, pueden organizarse en capas para construir arquitecturas más complejas que resuelvan problemas más avanzados. Uno de los modelos más simples de estas redes es la Red Neuronal

Poco Profunda (*Shallow Network*), también conocida como SLFN[3]. Este tipo de red, representada en la Fig. 2.3, generalmente se compone de una capa de entrada (que recibe los datos directamente), una capa intermedia denominada capa "oculta" (*hidden layer*), y una capa de salida. Aunque las *Shallow Networks* pueden resolver una amplia gama de problemas, como la clasificación y la regresión, suelen requerir de un gran número de neuronas para alcanzar un rendimiento comparable a otros modelos más complejos y de múltiples capas. Esto se debe a que, en una red con solo una capa oculta, todas las transformaciones requeridas para modelar la complejidad de los datos de entrada deben realizarse en esa capa única, lo que conlleva a la necesidad de incrementar significativamente el número de neuronas a medida que aumenta la complejidad del problema a resolver. De esta forma, algunos modelos de *Shallow Networks* [55, 56] contienen cientos o miles de neuronas en la capa "oculta" incluso para casos de problemas sencillos como la clasificación de imágenes del *dataset* MNIST [39].

2.1.2. Entrenamiento de una neurona artificial

En el entrenamiento supervisado, para el aprendizaje de la Red Neuronal se utiliza el algoritmo de *backpropagation*, que ajusta los pesos de la red minimizando una cierta función de costo, que toma como entrada la salida (predicción) de la red y un valor objetivo para dicha salida. Este proceso implica calcular el gradiente de la función de costo respecto a los pesos, y luego actualizar los pesos en la dirección opuesta al gradiente usando métodos como el descenso de gradiente estocástico (SGD[4]). En el caso de una neurona individual, el proceso de *backpropagation* sigue los mismos principios generales, enfocándose en la actualización de los pesos asociados a dicha neurona. Como ya se mencionó anteriormente, durante la fase de propagación hacia adelante (*forward propagation*) se multiplican las entradas x_i de la neurona por sus respectivos pesos w_i, y luego se suman para producir el valor neto y_{Lin} que será

[3] *Single hidden Layer Feedforward neural Network*
[4] *Stochastic Gradient Descent*

afectado por una función de activación produciendo así su salida y_{AF}:

$$y_{Lin} = \left(\sum_i w_i x_i \right) + b \ ,$$
$$y_{AF} = f_{Act}\left(y\right) \ ,$$

con b como el sesgo o *bias* y f_{Act} la función de activación.

En la fase de *backpropagation*, se calcula primero el error cometido en la predicción de la salida de la neurona $y_{pred} = y$ en comparación con el valor objetivo y_{true}. Este error se cuantifica mediante una función de costo $L = f_{Loss}\left(\boldsymbol{y}_{pred}, \boldsymbol{y}_{true}\right)$. Dado que el objetivo es minimizar esta función de costo ajustando los pesos, se calcula el gradiente del error o función de costo L respecto a cada peso w_i usando la regla de la cadena:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y_{AF}} \frac{\partial y_{AF}}{\partial y_{Lin}} \frac{\partial y_{Lin}}{\partial w_i} \ ,$$

donde el término $\partial L / \partial y_{AF}$ representa la sensibilidad del error respecto a la salida, mientras que $\partial y_{AF} / \partial y_{Lin}$ es la derivada de la función de activación y $\partial y_{Lin} / \partial w_i = x_i$ es la derivada de la combinación lineal de las entradas. A partir de este gradiente, los pesos se actualizan de acuerdo a la regla de actualización:

$$w_i \leftarrow w_i - \eta \frac{\partial L}{\partial w_i} \ ,$$

con η como el factor de aprendizaje que controla el tamaño del ajuste. Este proceso se repite hasta que el error de la predicción sea lo suficientemente bajo o hasta que se alcance un número máximo de iteraciones, permitiendo así que la neurona aprenda o ajuste su respuesta en función de las entradas. En lugar de actualizar los pesos después de cada muestra, el método de SGD los actualiza en lotes (*batch*) de N_{batch} muestras aleatorias, promediando el efecto de los gradientes obtenidos en cada muestra y mejorando así la eficiencia del entrenamiento:

$$w_i \leftarrow w_i - \frac{\eta}{N_{batch}} \sum_{j=1}^{N_{batch}} \left(\frac{\partial L}{\partial w_i} \right)_j \ . \tag{2.1}$$

Para la actualización del *bias* se sigue un procedimiento similar, donde se tiene que:

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y_{AF}} \frac{\partial y_{AF}}{\partial y_{Lin}} \ ,$$

ya que la derivada parcial de la salida lineal con respecto al *bias* es $\partial y_{Lin}/\partial b = 1$, por ser este último una constante.

En el entrenamiento de redes neuronales, una de las primeras decisiones clave es la elección de una función de costo L. Entre las funciones de costo más utilizadas se encuentran el Error Cuadrático Medio (MSE[5]) y la Entropía Cruzada (CEL[6]). El MSE es común en problemas de regresión y se define como:

$$L = \frac{1}{N} \sum_{i=1}^{N} \left(y_{pred,i} - y_{true,i} \right)^2 \ , \tag{2.2}$$

donde N es el número de salidas o neuronas. Por otro lado, CEL se utiliza en problemas de clasificación y mide la diferencia entre dos distribuciones de probabilidad predicha $\boldsymbol{z}_{pred}$ y real $\boldsymbol{z}_{true}$, de manera tal que:

$$L = -\sum_{i=1}^{N} y_{true,i} \log \left(y_{pred,i} \right) \ , \tag{2.3}$$

con N representando también al número de clases.

Una vez definida la función de costo, es necesario utilizar un optimizador para ajustar los pesos de la red, minimizando dicha función y aplicando reglas de actualización que ajusten los pesos basándose en los gradientes calculados mediante *backpropagation*. El optimizador más sencillo es el ya mencionado SGD y descrito por la Ec. (2.1), que simplemente substrae del peso el gradiente obtenido y escalado por el factor de aprendizaje η. Sin embargo, existen optimizadores más avanzados, como *Adam* [57], que adaptan el tamaño del paso de actualización basándose en las

[5] *Mean Squared Error*
[6] *Cross-Entropy Loss*

primeras y segundas derivadas acumuladas de los gradientes:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial w_i} \ , \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\partial L}{\partial w_i}\right)^2 \ , \\ w_i &\leftarrow w_i - \frac{\eta}{\sqrt{v_t}+\epsilon} m_t \ , \end{aligned} \tag{2.4}$$

donde m_t y v_t son estimaciones de la media y la varianza del gradiente, β_1 y β_2 son parámetros de control, y ϵ es un pequeño valor para evitar divisiones por cero. Cabe destacar que, en las expresiones de la Ec. (2.4), se omitió por simplicidad el efecto de promediar los gradientes por grupos de muestras o *batch*, pero este debe tenerse en cuenta para mejorar la convergencia del algoritmo. De esta manera, *Adam* y otros optimizadores avanzados permiten un ajuste más rápido y eficiente de los pesos en comparación con SGD estándar. Otras técnicas de optimización se basan en modificar algunos hiperparámetros a lo largo del proceso de entrenamiento. Algunos ejemplos de esto son el aumento en el número de muestras por *batch* o el empleo de alguna función de decaimiento para el factor de aprendizaje η.

Habiendo seleccionado una función de costo y optimizador adecuados, se deben aplicar estas herramientas en un proceso de entrenamiento que permita ajustar los pesos del modelo, minimizando el error en las predicciones. Sin embargo, es crucial contar con una estrategia adecuada para evaluar el rendimiento del modelo a lo largo del entrenamiento y asegurar que este no sólo optimice la función de costo para los datos de entrenamiento, sino que también generalice adecuadamente la solución para nuevos datos. En este sentido, se vuelve fundamental dividir los datos en conjuntos de entrenamiento, validación y prueba (*test*) para evaluar el rendimiento de la red. El conjunto de entrenamiento (subdividido en grupos o *batch*) se utiliza para ajustar los pesos del modelo mediante el algoritmo de *backpropagation* y el optimizador elegido. El conjunto de validación, en cambio, consiste en datos nuevos que se emplean para monitorear el rendimiento del modelo durante el proceso de entrenamiento, computando las salidas de la red solamente en modo *forward*. Esto también permite ajustar hiperparámetros como el factor de aprendizaje η, el número de épocas o iteraciones de entrenamiento, la arquitectura de la red, entre otros. Por

último, el conjunto de prueba se reserva exclusivamente para evaluar la capacidad de generalización del modelo sobre datos no vistos al finalizar el entrenamiento.

Una técnica clave para mejorar la capacidad de generalización de la red es la validación cruzada (*k-fold cross validation*). En este caso, los datos originales (entrenamiento y validación) se dividen en k sub-grupos (*folds*), donde en cada iteración del proceso se seleccionan $k-1$ *folds* para el entrenamiento, mientras que el *fold* restante se usa para la validación. Este procedimiento se repite k veces, alternando los roles de entrenamiento y validación en cada *fold*, de manera que cada muestra se utilice tanto para entrenar como para validar en algún punto del proceso. El resultado final es un promedio de las métricas de rendimiento obtenidas en cada iteración, proporcionando así una evaluación más robusta de la capacidad del modelo para generalizar a datos no vistos. Este enfoque mitiga el riesgo de una partición desfavorable de los datos, donde una división particular podría sesgar los resultados si algunos *folds* contienen patrones inusuales o atípicos. Además, la validación cruzada es especialmente útil en conjuntos de datos pequeños, donde maximizar el uso de todas las muestras para entrenamiento y validación es crucial. Es importante señalar que, una vez finalizado el proceso de validación cruzada y ajustado el modelo de acuerdo a los hiperparámetros optimizados, el conjunto de prueba (*test*) debe ser utilizado únicamente para la evaluación final del modelo. Esto garantiza que las decisiones del modelo no hayan sido influenciadas por los datos de prueba, proporcionando una medida realista de su rendimiento sobre datos completamente nuevos. Otra técnica que ayuda a la generalización consiste en el método conocido como *data augmentation*. Esta estrategia se basa en modificar de forma aleatoria los datos de entrenamiento, en función de las características de los mismos, incluyendo así rotaciones, ligeros cambios de color, etc. Esto permite que el modelo se ajuste con más variaciones de los datos originales y extraiga así la información más relevante de los mismos, lo que es particularmente útil para conjuntos de datos pequeños.

2.1.3. Redes neuronales profundas

Las Redes Neuronales Profundas (DNNs) se distinguen por contar con múltiples capas "ocultas" situadas entre la capa de entrada y la capa de salida, lo que justifica

su denominación de "profundas". Un ejemplo fundamental de estas redes es el Perceptrón Multi-capa (MLP[7]), que se compone de varias capas de perceptrones con neuronas completamente conectadas (FC), donde en cada capa se realiza una transformación lineal seguida de una función de activación no lineal. El aumento en la profundidad permite que el modelo procese la información a través de diversas etapas, facilitando que cada capa intermedia extraiga características de nivel superior basadas en los conocimientos adquiridos por las capas anteriores. Esta estructura jerárquica y progresiva confiere a las DNNs la capacidad de aprender representaciones de datos en varios niveles de abstracción, permitiendo la detección de patrones complejos y una descomposición más detallada y eficiente del problema. Como resultado, las DNNs pueden lograr un rendimiento superior con un menor número total de neuronas en comparación con las Redes Neuronales Poco Profundas (*Shallow Networks* o SLFN).

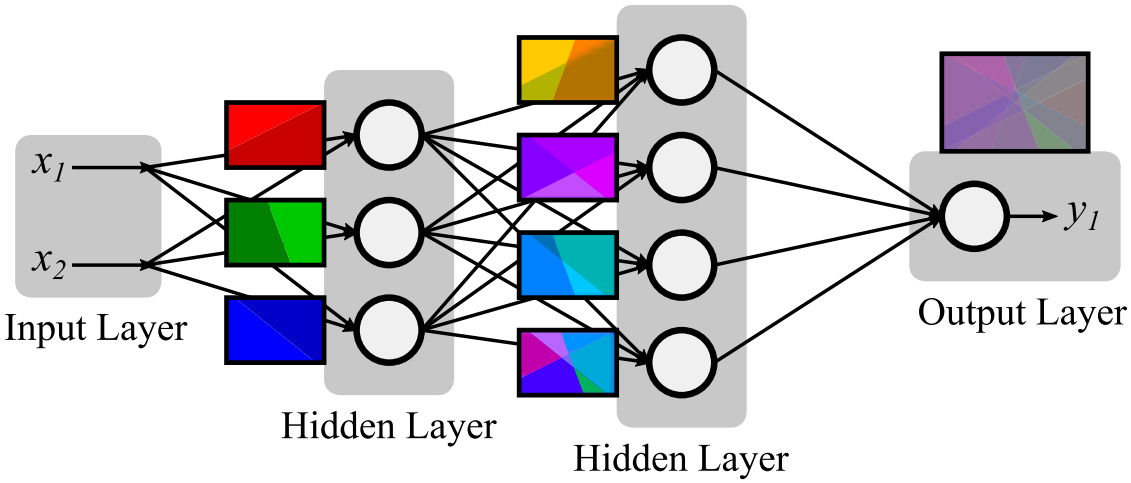


Figura 2.4: Ejemplo de Red Neuronal Profunda (MLP) y su generación de "regiones lineales" por capa.

El proceso de *forward propagation* en una DNN, ilustrado en la Fig. 2.4, se basa en transmitir los datos de entrada a través de cada capa de la red, aplicando transformaciones lineales y funciones de activación sucesivas hasta alcanzar la salida final. Durante este proceso, se crean "regiones lineales", donde cada región representa una zona en la que el modelo puede aproximar una función lineal sobre los datos de entrada a la capa. Estas regiones lineales se combinan para formar las fronteras

[7]Perceptrón multi-capa - *Multi-Layer Perceptron*

de decisión (*decision boundaries*) que el modelo utiliza para clasificar o hacer predicciones. La habilidad de una DNN para ajustar estas fronteras de decisión con precisión es crucial para capturar relaciones no lineales complejas entre las variables de entrada y realizar predicciones más sofisticadas.

El algoritmo de *backpropagation*, mencionado en la Sub-Sección 2.1.2, puede extenderse a las DNNs utilizando la regla de la cadena para calcular los gradientes de la función de costo respecto a cada peso en la red. Si se considera a $\boldsymbol{y}_L$ como la salida final de la red, $\boldsymbol{y}_l$ la salida de una capas intermedia "l", y $\boldsymbol{x}_{l+1} = \boldsymbol{y}_l$ la entrada a la siguiente capa, se tiene que:

$$\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{w}_l} &= \frac{\partial L}{\partial \boldsymbol{y}_l}\frac{\partial \boldsymbol{y}_l}{\partial \boldsymbol{w}_l} \ , \\
\frac{\partial L}{\partial \boldsymbol{w}_l} &= \left(\frac{\partial L}{\partial \boldsymbol{y}_L}\frac{\partial \boldsymbol{y}_L}{\partial \boldsymbol{y}_{L-1}} \cdots \frac{\partial \boldsymbol{y}_{l+1}}{\partial \boldsymbol{y}_l}\right)\frac{\partial \boldsymbol{y}_l}{\partial \boldsymbol{w}_l} \ , \\
\frac{\partial L}{\partial \boldsymbol{w}_l} &= \frac{\partial L}{\partial \boldsymbol{x}_{l+1}}\frac{\partial \boldsymbol{y}_l}{\partial \boldsymbol{w}_l} \ ,
\end{aligned}$$

por lo que es posible hallar los gradientes de los pesos $^{\partial L}/_{\partial \boldsymbol{w}_l}$ en base al gradiente de la entrada de la capa siguiente $^{\partial L}/_{\partial \boldsymbol{x}_{l+1}}$. De esta forma, partiendo de la función de costo L y la salida de la última capa $\boldsymbol{y}_L$, se computan capa por capa tanto los gradientes con respecto de los pesos como de las entradas, siendo estos últimos "propagados" hacia las capas previas, permitiendo así ajustar los pesos de manera eficiente incluso en redes con muchas capas. Cabe destacar que en las expresiones anteriores se incluye a la función de activación dentro de la capa, por lo que la derivada $^{\partial \boldsymbol{y}_l}/_{\partial \boldsymbol{w}_l}$ ya no es simplemente $\boldsymbol{x}_l$ para capas lineales, sino que dependerá de dicha función de activación. Dado que las capas de DNNs poseen varias neuronas, lo que implica que las salidas y entradas de cada capa son vectores (pudiendo escalar a matrices o tensores en modelos más complejos), con el fin de obtener expresiones para los gradientes de pesos/*bias* y entradas, se puede utilizar la expresión de la regla de la cadena para cada elemento del vector o matriz, sabiendo que:

$$\frac{\partial L}{\partial v_{i,j}} = \sum_{l=1}^{L}\sum_{k=1}^{K}\frac{\partial L}{\partial u_{k,l}}\frac{\partial u_{k,l}}{\partial v_{i,j}} \ , \tag{2.5}$$

donde se suman las derivadas parciales para todas las funciones de capa $u_{k,l}$ que

contengan al elemento $v_{i,j}$.

A medida que las DNNs aumentan en profundidad, surgen ciertos problemas inherentes a su entrenamiento, tales como el desvanecimiento de gradientes (*vanishing gradient*) y la explosión de gradientes (*exploding gradient*). En redes muy "profundas", los gradientes pueden volverse extremadamente pequeños en las capas más cercanas a la entrada (*vanishing gradient*), lo que impide que estas ajusten sus parámetros de manera efectiva, ralentizando o incluso deteniendo el aprendizaje. Por el contrario, cuando los gradientes se vuelven excesivamente grandes (*exploding gradient*), los pesos de la red pueden sufrir actualizaciones inestables, lo que lleva a oscilaciones abruptas en el proceso de entrenamiento o incluso a la divergencia del modelo. Estas dificultades han motivado el desarrollo de diversas técnicas, como la inicialización de pesos adecuada, la normalización por lotes (BatchNorm[8]) y el uso de funciones de activación como la ReLU, que mitigan parcialmente el problema del desvanecimiento del gradiente, facilitando el entrenamiento de redes más profundas de manera eficiente. En estos casos de redes muy "profundas", la función de activación ReLU con una saturación en los valores positivos pudiera parecer una buena alternativa a la ReLU tradicional, al prevenir que se presenten valores grandes a la salida. Estas activaciones pueden ser especialmente útiles cuando se requieren valores de entrada/salida acotados, ya sea porque las funciones en sí mismas tienen un rango de entrada definido o porque se desea implementar la cuantización de entradas y/o salidas (Sección 2.2). Sin embargo, la saturación de estos valores positivos puede traer problemas tales como pérdida de información y saturación de gradientes, ya sea por mantener el rango de las entradas en valores pequeños o por directamente anular los valores del gradiente en las posiciones donde la activación ReLU satura sus entradas. A pesar de esto, existen casos de Redes Neuronales donde se emplean activaciones del tipo ReLU saturadas, como por ejemplo MobileNet [54] que hace uso de la función ReLU6, con la que se encontró empíricamente un balance entre la saturación de los gradientes y las ventajas en cuanto a cuantización de acotar las salidas de la capa a un rango determinado.

[8]Normalización por grupos de muestras - *Batch Normalization*

2.1.4. Redes neuronales convolucionales

Las Redes Neuronales Convolucionales (CNNs) representan una clase especializada de DNNs, diseñada principalmente para trabajar con datos estructurados en forma de cuadrícula, como las imágenes. Este tipo de redes se basa en el uso de capas de convolución, que permiten a la red extraer de manera eficiente características locales, al aplicar un mismo conjunto de pesos o *kernel* a lo largo de toda la imagen o tensor de entrada, lo que reduce significativamente la cantidad de parámetros a entrenar. A diferencia de las capas FC, las capas convolucionales limitan las conexiones de una neurona a pequeñas regiones de la imagen o tensor, preservando así la estructura espacial de los datos. Este tipo de capas son esencialmente filtros espaciales, ya que se aplican a las regiones locales de la imagen, y en los que cada filtro es entrenado para reconocer una característica o *feature* particular, lo que permite identificar rasgos específicos como contornos, texturas o patrones geométricos. Estos algoritmos, junto con el uso de filtros especializados, hacen que las CNNs sean extremadamente eficientes y precisas en tareas complejas de ML, como las aplicaciones de *computer vision*.

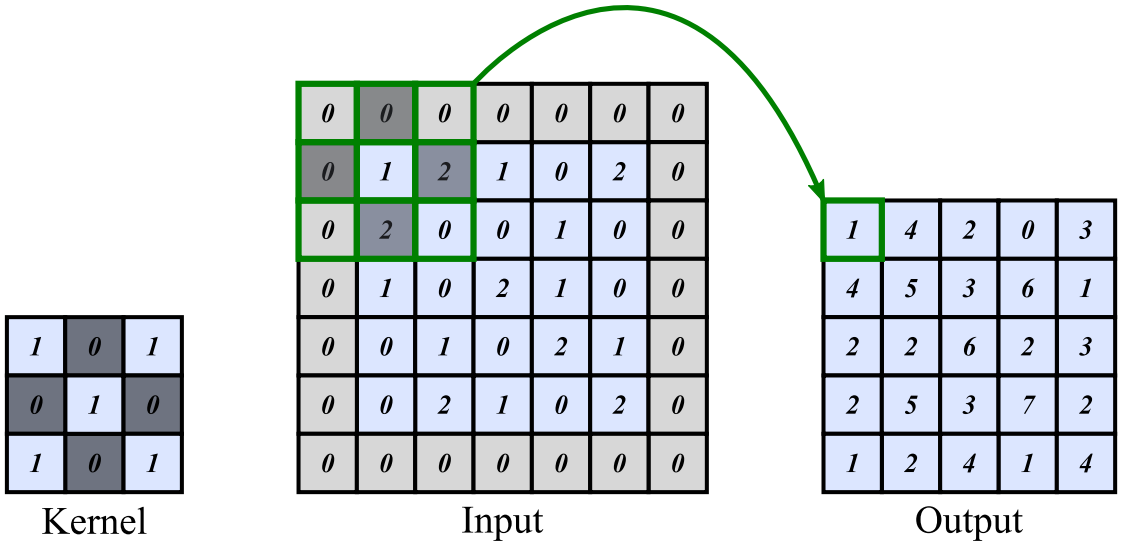


Figura 2.5: Ejemplo de convolución 2D, procesando un canal con *kernel* de 3×3, *stride* 1, *padding* 1 y sin *dilation*.

Una de las capas más utilizadas en las CNNs es la convolución bidimensional (Conv2D[9]), la cual se encarga de desplazar un filtro (*kernel*) a lo largo del alto y

[9]Convolución en dos dimensiones

ancho de la imagen de entrada, aplicando una suma ponderada entre las entradas que se solapan con los pesos de dicho *kernel*. Este proceso, ilustrado en la Fig. 2.5, genera un mapa de activación donde cada valor representa la respuesta del filtro a una región específica de la imagen original. En los casos donde se manejan varios canales de entrada, como en una imagen RGB o en los tensores de capas intermedias, se filtran los canales individuales con un conjunto de pesos diferente, para posteriormente generar el mapa de activación final como la suma (por píxel) de los canales filtrados. Esto permite que las CNNs combinen la información de múltiples características o colores en una única representación, sin perder la capacidad de detectar patrones específicos en cada canal. Este proceso se repite además con varios filtros diferentes, uno por *feature* o canal de salida que se desee extraer de la imagen o tensor original, añadiendo opcionalmente un valor de *bias*. De esta forma, la función de una capa Conv2D puede describirse con el Algoritmo 1, donde por simplicidad se omiten los índices correspondientes a la asignación de la variable intermedia $\boldsymbol{H}$ correspondiente a los canales filtrados. En dicho algoritmo se hace uso además de la función "im2col", que transforma a la imagen de entrada en columnas, con cada columna conteniendo a los píxeles de la entrada que se usan en cada paso de la convolución, lo que permite realizar la operación con simples productos vector-vector. Otro tipo de capa convolucional similar, utilizado en [54], es la *depth-wise separable convolution*, que consiste en la aplicación de filtros 2D para cada canal por separado, que se comparten para todos los *features* de salida, por lo que en una primera etapa se obtienen solamente los canales filtrados. Los *features* de salida se obtienen luego de realizar la convolución (Conv2D) de dichos canales procesados con *kernels* de tamaño 1×1, reduciendo así la cantidad de parámetros a entrenar a costa de reducir un poco el desempeño del modelo.

Para ajustar la resolución de la salida, se puede modificar el valor de *stride*, que define el número de píxeles en que se desplaza el *kernel* en cada paso, lo que también influye en la reducción de la dimensionalidad si este posee valores superiores a 1. Además, existen técnicas como el *padding*, que consiste en añadir ceros (u otro valor) alrededor de la imagen de entrada para controlar el tamaño del mapa de salida, asegurando que se mantengan las dimensiones o se minimice la pérdida de información en los bordes. Otra variación importante es el uso de *dilation*, que

Algoritmo 1 Función de convolución en dos dimensiones (Conv2D)

Entrada:

- Tensor de entrada $\boldsymbol{X} \in \mathbb{R}^{H_{in} \times W_{in} \times D_{in}}$, H_{in} y W_{in} el alto y ancho de $\boldsymbol{X}$, D_{in} el número de canales de entrada;
- Pesos $\boldsymbol{W} \in \mathbb{R}^{N \times D_{in} \times D_{out}}$, con $N = k_H \times k_W$ el tamaño del *kernel*;
- *bias* $\boldsymbol{b} \in \mathbb{R}^{D_{out}}$.

Salida:

- Tensor $\boldsymbol{Y} \in \mathbb{R}^{H_{out} \times W_{out} \times D_{out}}$.

for $i = 1 : D_{out}$ **do**
  $\boldsymbol{X}_c \leftarrow \text{im2col}(\boldsymbol{X}, (k_H, k_W))$
  for $j = 1 : (H_{out} \times W_{out})$ **do**
    for $k = 1 : D_{in}$ **do**
      $\boldsymbol{H} \leftarrow \boldsymbol{W}^T \boldsymbol{X}_c$
    end for
    $\boldsymbol{Y}_c[j, i] \leftarrow \left(\sum_{k=1}^{D_{in}} \boldsymbol{H}[j, k, i] \right) + \boldsymbol{b}[i]$
  end for
end for
$\boldsymbol{Y} \leftarrow \text{reshape}(\boldsymbol{Y}_c)$
return $\boldsymbol{Y}$, $\boldsymbol{\mu}$, $\boldsymbol{Ind}_s$

introduce espacios entre los elementos del *kernel*, expandiendo así su campo receptivo pero sin aumentar el número de parámetros, lo que permite capturar características en distintas escalas de la imagen. Considerando una entrada $\boldsymbol{X} \in \mathbb{R}^{H_{in} \times W_{in} \times D_{in}}$ y un *kernel* $\boldsymbol{W} \in \mathbb{R}^{k_H \times k_W \times D_{in} \times D_{out}}$, el tamaño (en alto H_{out} y ancho W_{out}) de la salida estará dado por:

$$H_{out}/W_{out} = \left\lfloor \frac{H_{in}/W_{in} + 2 \times P_{H/W} - D_{H/W} \times \left(K_{H/W} - 1\right) - 1}{S_{H/W}} + 1 \right\rfloor , \qquad (2.6)$$

donde $P_{H/W}$ corresponde al *padding*, $D_{H/W}$ al uso de *dilation* y $S_{H/W}$ al *stride*, según como son aplicados a las filas (H) y a las columnas (W). Mientras que el *padding* puede aplicarse a la imagen o tensor de entrada $\boldsymbol{X}$ en el Algoritmo 1, parámetros como *stride* y *dilation* afectan directamente la ejecución de la función "im2col", descartando algunas columnas al aumentar el valor de *stride* y modificando con el valor de *dilation* a la selección de pixeles que se ubican en cada columna.

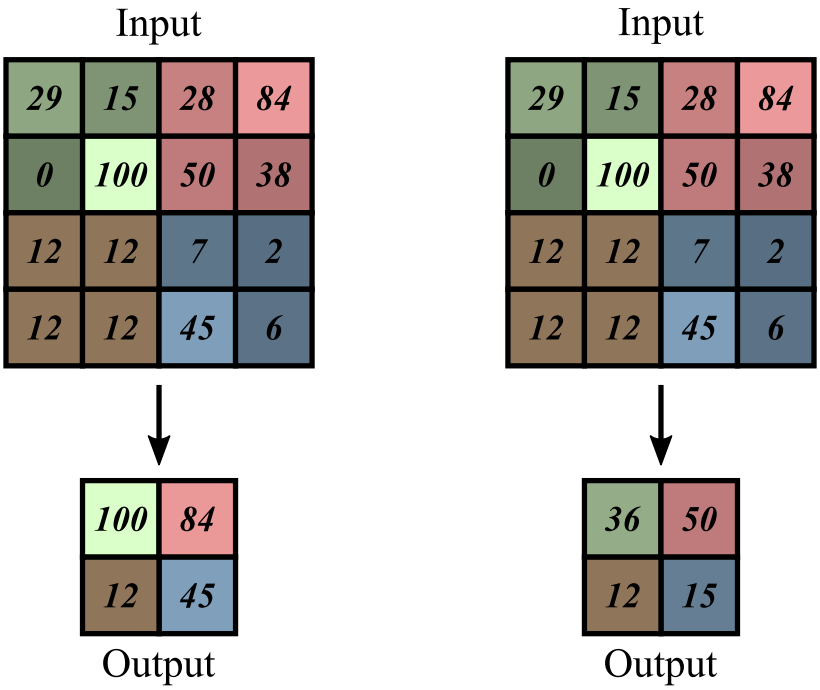


Figura 2.6: Ejemplos de *pooling*: a) MaxPool; b) AvgPool.

Además de las capas convolucionales, las CNNs típicamente incluyen funciones de *pooling*, que reducen las dimensiones de los datos de entrada al realizar un submuestreo de los mismos (*stride* mayor o igual a 2), manteniendo sus características más importantes. Uno de los métodos más comunes de esta clase es el *max pooling* (MaxPool[10]), mostrado en la Fig. 2.6a, que opera de forma similar a la convolución al recorrer la imagen o tensor con una ventana de un tamaño determinado, computando el valor máximo dentro del vecindario de píxeles que se superponen a la ventana deslizante. Este tipo de operación también es utilizada como un filtro espacial, en los casos donde el valor de *stride* es inferior al tamaño de la ventana elegida, como sucede con la familia de redes ResNet [18] que emplean un MaxPool con ventana de 3×3 y *stride* 2. Por otro lado, el *average pooling* o AvgPool[11] (Fig. 2.6b) opera de forma similar, pero calculando el promedio en lugar del máximo, siendo a su vez un filtro lineal como la convolución. De hecho, la operación de AvgPool puede resolverse como una convolución en la que todos los pesos del *kernel* son $1/N$, siendo N el tamaño del *kernel*.

[10]Filtrado por valor máximo
[11]Filtrado por valor medio

La normalización por lotes, también conocida como *Batch Normalization* o simplemente BatchNorm, es una técnica ampliamente utilizada en las DNNs y CNNs para mejorar la estabilidad y eficiencia del proceso de entrenamiento. Su principal objetivo es el de reducir la variación en la distribución de las activaciones (salidas) de cada capa, lo que permite que la red aprenda de manera más rápida y confiable. Durante el entrenamiento, la función de BatchNorm actúa normalizando las salidas de la capa previa utilizando la media y varianza calculadas a partir de los datos de cada lote de entrada. Dicha normalización ayuda a mitigar el problema de *vanishing gradient*, ya que al reducir la dispersión de las activaciones se evita que los gradientes disminuyan de manera drástica a medida que se propagan hacia atrás por la red. Como resultado, las actualizaciones de los parámetros se mantienen efectivas, incluso en redes profundas. Además, al estabilizar la distribución de las activaciones en cada capa, esta función permite el uso de tasas de aprendizaje más altas, acelerando así el proceso de entrenamiento y favoreciendo una convergencia más rápida. Una vez realizada la normalización en sí misma, se aplican dos parámetros adicionales: una factor de escala que multiplica a las salidas y la suma de un *bias*, los cuales pueden ser entrenados y permiten que la red recupere su capacidad de representar adecuadamente los datos. Considerando lo anteriormente mencionado, el algritmo de BatchNorm puede escribirse como:

$$\boldsymbol{Y}_{BN} = \gamma \frac{\boldsymbol{Y} - E[\boldsymbol{Y}]}{\sqrt{V[\boldsymbol{Y}] + \epsilon}} + \beta \ , \tag{2.7}$$

con $\boldsymbol{Y}$ como la salida de la capa a normalizar, $E[\boldsymbol{Y}]$ y $V[\boldsymbol{Y}]$ la esperanza (media) y varianza acumuladas, γ y β como el factor de escala y *bias* a entrenar, y finalmente ϵ como un valor pequeño para evitar la división por 0. Para capas como Conv2D, el algoritmo de BatchNorm descrito en la Ec. (2.7), se aplica a cada canal de salida de forma independiente, contando con diferentes parámetros γ y β por canal.

Durante el entrenamiento, el cálculo de la media y varianza para BatchNorm es computacionalmente intenso y suele requerir el uso de representaciones en punto flotante para manejar las magnitudes de forma adecuada. Sin embargo, durante la inferencia, que se da cuando el modelo se utiliza exclusivamente para realizar predicciones, se emplean los valores fijos de media y varianza, que fueron previamente

calculados a lo largo del proceso de entrenamiento. Este ajuste convierte el proceso de normalización en una operación que puede ser integrado como factores de escala directamente en los parámetros de la red, como los pesos y los *bias*, siempre que la operación de BatchNorm se realice inmediatamente después de la capa convolucional o FC. Para el caso de una capa lineal, la "fusión" de dicha capa con el algoritmo de BatchNorm puede efectuarse de la siguiente forma:

$$\boldsymbol{W}' = \boldsymbol{W}\frac{\gamma}{\sqrt{\sigma^2+\epsilon}} \tag{2.8}$$

$$\boldsymbol{B}' = \gamma\left(\frac{\boldsymbol{B}-\mu}{\sqrt{\sigma^2+\epsilon}}\right)+\beta\ , \tag{2.9}$$

por lo que la capa "fusionada" puede interpretarse como simplemente una capa lineal con los nuevos valores de pesos $\boldsymbol{W}'$ y *bias* $\boldsymbol{B}'$, obtenidos a partir de los parámetros entrenables γ y β, así como también los valores fijos de media μ y varianza σ^2. Gracias a esto, es posible eliminar los cómputos realizados por la capa de BatchNorm, reemplazando los valores de pesos y *bias* para las capas lineales por sus nuevas versiones de las Ecs. (2.8) y (2.9), que a su vez pueden ser representados utilizando punto fijo para un procesamiento (inferencia) más eficiente en *hardware*.

2.1.5. Operadores morfológicos

La morfología matemática es una técnica fundamental en el procesamiento de imágenes que se centra en la forma y estructura de los objetos presentes en una imagen. Esta metodología emplea operadores morfológicos (generalmente filtros no lineales) para transformar la imagen según las características geométricas de sus elementos, lo que los vuelve esenciales para diversas aplicaciones en el análisis y la mejora de imágenes, permitiendo una manipulación precisa de la información visual basada en la forma. Entre estos operadores se encuentran la erosión y la dilatación como los casos más básicos de morfología matemática. Por un lado, la erosión actúa reduciendo los objetos en la imagen, eliminando píxeles en los bordes de los objetos y, por ende, reduciendo su tamaño, mientras que la dilatación expande los objetos al añadir píxeles en los bordes, incrementando su tamaño y conectividad. Además, al combinarse estos dos casos en secuencias específicas, se pueden produ-

cir otros operadores morfológicos más complejos, tales como la apertura y el cierre. La apertura consiste en aplicar una erosión seguida de una dilatación, eliminando pequeños objetos y suavizando los bordes de los objetos más grandes. En contraste, el cierre aplica primero una dilatación y luego una erosión, cerrando pequeños huecos y conectando regiones discontinuas dentro de los objetos. Tanto los operadores morfológicos básicos de erosión y dilatación, como sus extensiones en apertura y cierre, son presentados en la Fig. 2.7, donde a partir de una imagen binaria se muestran los resultados de filtrar esta con los operadores mencionados. En casos similares al presentado en la Fig. 2.7, donde se tiene un fondo oscuro mientras que el objeto es blanco (o de color claro), la operación de erosión coincide con una capa de MinPool[12], mientras que la dilatación es equivalente a una capa de MaxPool.

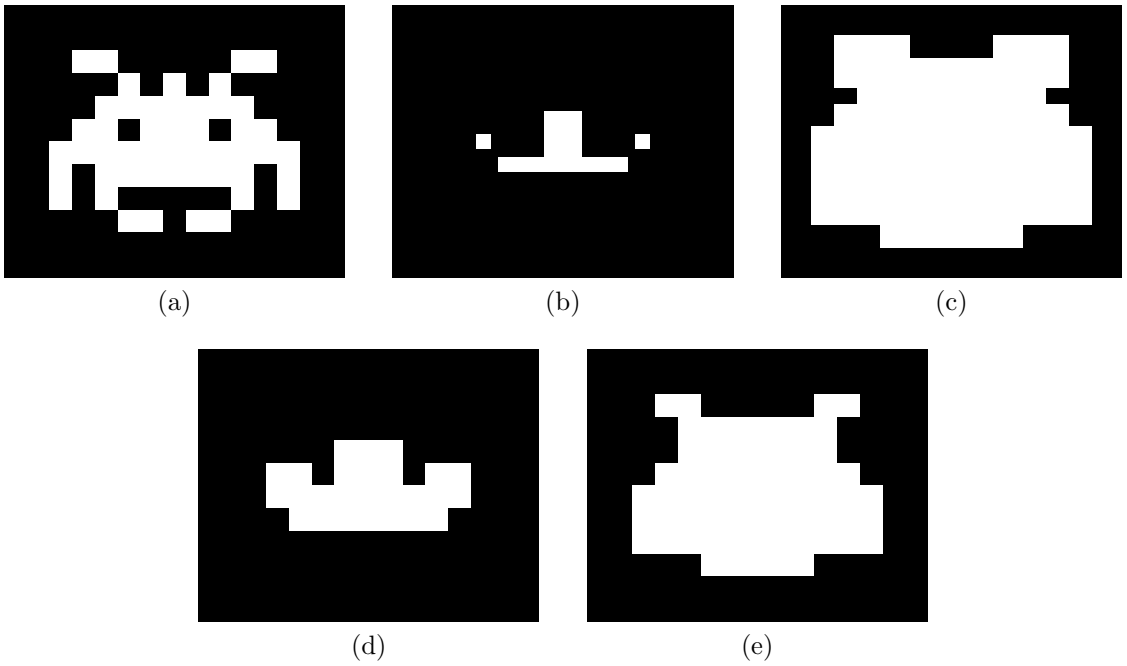
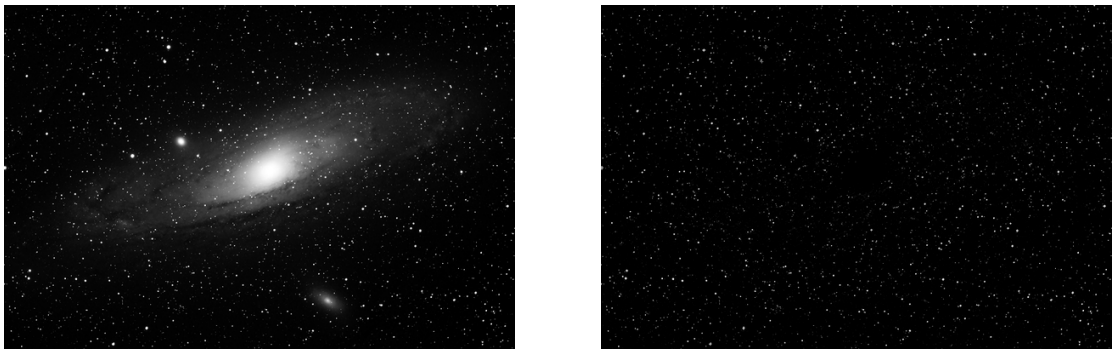


Figura 2.7: Ejemplos de operaciones morfológicas (binarias): a) imagen original; b) erosión; c) dilatación; d) apertura; e) cierre.

Además de los operadores mencionados anteriormente, existen técnicas morfológicas más avanzadas que permiten un análisis aún más detallado de las imágenes. Uno de estos casos es el conocido como *Top-Hat* (Fig. 2.8), que se utiliza para resaltar estructuras específicas que difieren del fondo de la imagen. Esta operación implica una apertura inicial que luego se substrae de la imagen original, logrando así extraer

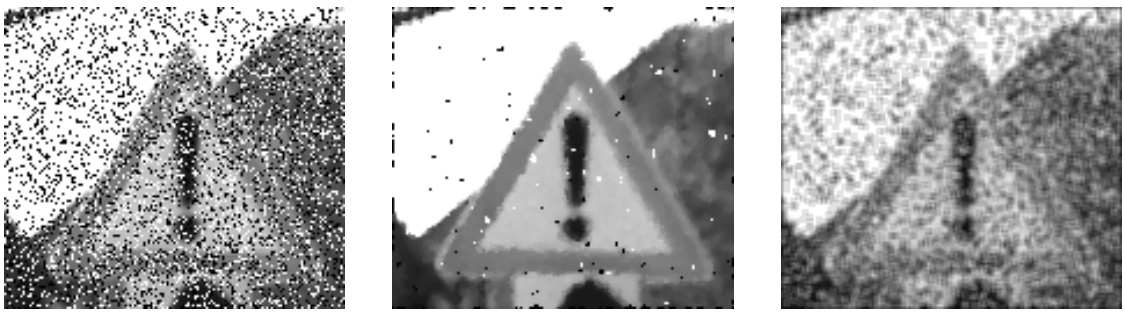
[12]Filtrado por valor mínimo

detalles finos (como pequeños objetos) y eliminar gradualmente elementos de fondo. Es por esto que la función *Top-Hat* resulta de gran utilidad en aplicaciones donde se busca detectar cambios sutiles en texturas o formas. Por otro lado, los filtros de mediana son herramientas eficaces para remover el ruido impulsivo, también conocido en el procesamiento de imágenes como *salt and pepper* (Fig. 2.9). A diferencia de los filtros lineales tradicionales (como media o AvgPool), el filtro de mediana es capaz de preservar los bordes de los objetos mientras suprime el ruido. De esta forma, filtros de mediana resultan especialmente útiles en la construcción de algoritmos más complejos para la reducción de ruido en imágenes, como por ejemplo en combinación con CNNs [58].



(a) (b)

Figura 2.8: Operación *Top-Hat* aplicada a una imagen de ejemplo: a) imagen original extraida de [1]; b) imagen procesada.



(a) (b) (c)

Figura 2.9: Reducción de ruido *salt and pepper* con operadores lineales y morfológicos: a) imagen original de [2], con ruido añadido; b) filtro de mediana; c) filtro AvgPool.

Las Redes Neuronales Morfológicas (MNN[13]) son una extensión de CNNs que

[13] *Morphological Neural Network*

combinan los conceptos de la morfología matemática con los de Redes Neuronales tradicionales, permitiendo la integración de operaciones morfológicas dentro del proceso de aprendizaje y obteniendo mejores resultados en, por ejemplo, la clasificación de escenarios. Estas redes utilizan dos estrategias principales: por un lado se utilizan operaciones morfológicas fijas pero con pesos entrenables que multiplican a las entradas durante previo a la operación morfológica, mientras que por otro lado se realiza una aproximación para los operadores MinPool y MaxPool con el fin de elegir uno de estos según las necesidades del problema. En esta segunda alternativa, la aproximación de dichos operadores morfológicos permite que posean un parámetro entrenable mediante técnicas estándar de *backpropagation*, que es el que distingue entre un operador y el otro. Algunos ejemplos de estas redes son las presentadas por [59], en donde utilizan redes inspiradas en LeNet [13] y Alexnet [15], pero reemplazando las capas convolucionales (Conv2D) por filtros morfológicos básicos (aproximados) que pueden ser entrenados. Otro ejemplo similar de MNN son las arquitecturas propuestas por [60], que hacen uso de filtros morfológicos y conexiones residuales (diferencia entre imagen original y procesada) para capturar las características o *features* de la imagen, que luego son procesados por capas FC estándar.

2.2. Cuantización

La cuantización es una técnica fundamental en la implementación de redes neuronales de forma eficiente, especialmente cuando se busca reducir la complejidad computacional y el consumo de memoria en dispositivos con recursos limitados, como son por ejemplo los sistemas embebidos. En términos de ML, este proceso implica reducir la precisión de los parámetros (pesos y *bias*) y activaciones (entradas/salidas) de las capas de una Red Neuronal, transformando valores que originalmente son en punto flotante (generalmente de 32 bits) a formatos más compactos, como números en punto fijo o enteros de menor precisión (8 bits o menos). Para llevar a cabo este proceso, es necesario hallar el rango de los valores que toman tanto los parámetros de las capas como sus activaciones. Esto se debe a que la cuantización distribuye un conjunto limitado de valores discretos a lo largo del rango continuo de los datos originales. Sin un rango adecuado, es posible que se pierda información

relevante o que el modelo no pueda representar correctamente las relaciones entre las entradas y las salidas de las capas de la red. Es por esto que el cálculo de dicho rango es indispensable para que se mantenga un equilibrio entre la reducción del tamaño de los datos y la precisión del modelo.

Para la obtención del rango de los datos (parámetros y activaciones), se utilizan principalmente dos estrategias de cuantización: la estática y la dinámica. La cuantización estática implica el uso de rangos fijos, que son obtenidos a partir de estadísticas calculadas con anterioridad, generalmente realizando la inferencia con modelo de red entrenado en punto flotante y utilizando un conjunto de datos de entrada representativos del problema a resolver. La cuantización dinámica, en cambio, ajusta el rango durante la ejecución de la red, permitiendo una mayor flexibilidad en escenarios donde los datos varían significativamente, pero a costa de una mayor complejidad computacional al añadirse el cálculo del rango de entrada/salida (parámetros fijos durante inferencias) a las operaciones de la capa.

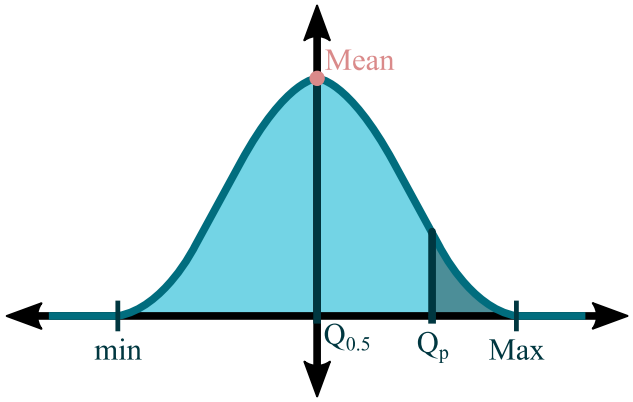


Figura 2.10: Obtención del rango de datos a partir de su distribución

El cálculo del rango de cuantización en sí mismo puede ser implementado utilizando diferentes estrategias. Una de las más comunes es el escalado min-max, donde se identifican los valores mínimos y máximos de los parámetros y/o activaciones, permitiendo que los valores extremos sean representados de manera precisa. Sin embargo, este método puede ser sensible a los valores atípicos (*outliers*), lo que podría llevar a una cuantización ineficiente si dichos valores extremos no son representativos para la mayoría de los datos a cuantizar. Para evitar este problema se suele recurrir al uso de cuantiles, que seleccionan un rango basado en un porcentaje determinado

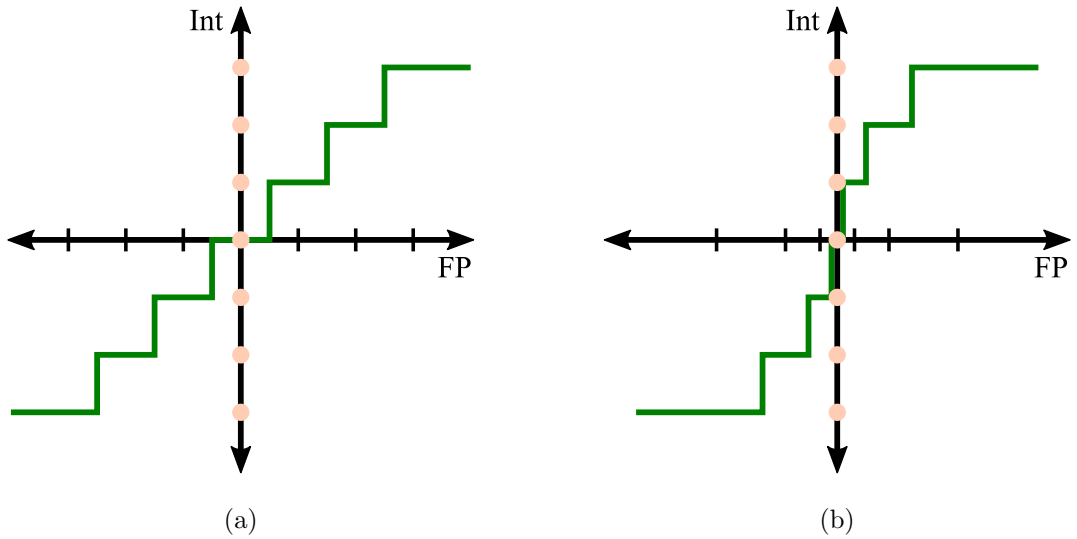
de la estadística de los datos, descartando así los valores atípicos y enfocándose en un rango que refleje mejor la distribución general de los parámetros o activaciones. Esta segunda técnica requiere, además, del uso de saturación para los valores que se encuentren por debajo o por encima de los rangos obtenidos por los cuantiles seleccionados, de manera que dichos *outliers* puedan ser mapeados a valores válidos y no generen errores durante el cómputo de la red. Ambas estrategias presentan un balance entre la simplicidad con el escalado min-max y la robustez con el cómputo de cuantiles, seleccionando uno entre estos dos métodos según el tipo de datos y el nivel de precisión que se busque mantener tras el proceso de cuantización.

La cuantización en sí misma puede dividirse en varios tipos según la forma en que los valores de los parámetros y activaciones se mapean a un rango discreto. Una de las clasificaciones más comunes es la que separa la cuantización entre uniforme y no-uniforme. En la cuantización uniforme, los valores continuos de los datos a cuantizar se distribuyen de manera equidistante a lo largo de un rango fijo, lo que simplifica las operaciones y el almacenamiento al poder representarse todos los valores con un mismo escalado lineal. Este enfoque es simple y eficiente, pero puede ser menos preciso, ya que algunos valores críticos podrían perderse al limitar la resolución en regiones específicas del rango donde se presenta una mayor densidad de datos. Por otro lado, la cuantización no-uniforme asigna más niveles de representación a las regiones del rango donde los datos se concentran, mejorando así la precisión en dichas zonas a costa de una mayor complejidad computacional.

Dentro del enfoque de cuantización uniforme, también se pueden distinguir dos variantes fundamentales: la cuantización simétrica y la cuantización asimétrica. En la cuantización simétrica, los valores se distribuyen de manera equitativa alrededor del cero, lo que resulta útil cuando los datos tienen una distribución centrada y aproximadamente balanceada entre valores positivos y negativos. De esta forma, el proceso de cuantización puede describirse como:

$$x_{int} = \text{Int}\left(x_{float} \times S\right) \ , \tag{2.10}$$

donde x_{int} es el valor cuantizado, x_{float} es el valor original, S es el factor de escalado que transforma los valores en punto flotante al rango entero deseado e $\text{Int}\left(\right)$ es la



(a) (b)

Figura 2.11: Modelos de cuantización: a) Uniforme; b) No-Uniforme.

función que produce el valor entero, eliminando la parte fraccionaria remanente luego de aplicarse el factor de escala. Ejemplos de esta función Int () son el redondeo, el truncado, el techo (*ceil*) y el piso (*floor*). En la cuantización asimétrica, en cambio, el rango de los valores cuantizados no está necesariamente centrado en cero, lo que permite un uso más eficiente de los valores de cuantización cuando los datos tienen un sesgo hacia los valores positivos o negativos. Esto resulta especialmente útil en escenarios donde las activaciones son predominantemente no negativas. La ecuación correspondiente en este caso es:

$$x_{int} = \text{Int}\left(x_{float} \times S\right) - z_{int} \ , \tag{2.11}$$

donde z_{int} es el valor de cero en punto fijo o entero (*offset*), que permite ajustar el rango de cuantización de forma más flexible. Este método ofrece un mayor control sobre el rango de cuantización, permitiendo una mejor representación de los valores originales en situaciones donde los datos no están distribuidos de manera simétrica. Cabe destacar que en caso de fijar el valor de cero en $z_{int} = 0$, como por ejemplo en la cuantización de valores estrictamente positivos, ambas Ecs. (2.10) y (2.11) son equivalentes.

Dado un rango definido por $x \in [m, M]$, con $m < M$ y $0 < M$, calculado ya sea mediante el método min-max o por cuantiles, y "q" como los bits de precisión

seleccionados para la cuantización, el factor de escala S puede hallarse como:

$$S = \frac{2^q - 1}{M} \ , \tag{2.12}$$

si los valores post-cuantización son magnitudes no-signadas (como por ejemplo activaciones luego de una ReLU), o

$$S = \frac{2^{q-1} - 1}{M - m} \ , \tag{2.13}$$

para mapear los datos a valores signados. En el caso de cuantización uniforme y valores signados, cabe mencionar que el rango de los datos se asume como $x \in [-\mu, \mu]$, con $\mu = max(|m|, M)$.

Por otra parte, para la implementación en *hardware* resulta conveniente el uso de factores de escala S como potencia de 2, lo que permite realizar la cuantización mediante simples desplazamientos de bits (*bit-shifts*). Para lograr esto, se puede calcular el factor de escala acorde a las Ec. (2.12) o (2.13) para luego llevar el valor resultante a la potencia de 2 más cercana (redondeo) o a dicha potencia que sea inferior o superior al valor de S, si se desea incluir los valores máximos al comprimir el rango (potencia de 2 menor a S) o si requiere de mayor precisión en los valores pequeños, expandiendo el rango y saturando los valores que se encuentren por encima del nuevo valor máximo (potencia de 2 mayor a S). Esta estrategia de utilizar potencias de 2 como factores de escala es especialmente útil en la implementación eficiente del re-escalado de las activaciones o salidas de una capa, ya que estas deben ser mapeadas a precisiones menores para ser utilizadas como entradas de capas posteriores en la red.

Finalmente, la cuantización puede clasificarse por granularidad o, en otras palabras, según el nivel de detalle con el que se aplican los rangos de cuantización a los datos de la Red Neuronal. La cuantización por tensor representa el enfoque más simple en esta categoría, debido a que se utiliza un rango único para todos los elementos del tensor a cuantizar, por lo que se comparten los mismos límites mínimos y máximos para todos los valores de dicho tensor. Si bien este método es computacionalmente eficiente, puede llevar a una pérdida de precisión en redes com-

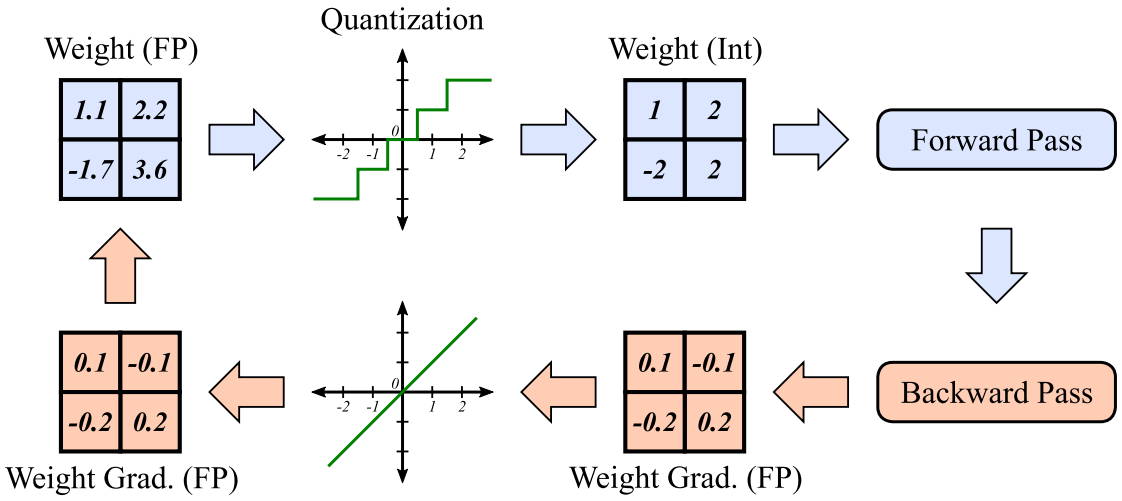


Figura 2.12: Descripción gráfica del entrenamiento con cuantización (QAT).

plejas donde los valores de los parámetros o activaciones varían significativamente entre diferentes canales. Por otra parte, la cuantización por canal ajusta los rangos de manera independiente para cada canal en una capa convolucional o para cada neurona (salida) en capas FC, lo que permite capturar mejor las características específicas de cada grupo de parámetros o activaciones. Este enfoque es más preciso y flexible, especialmente en casos donde los diferentes canales presentan distribuciones con rangos muy variados, pero introduce una mayor complejidad computacional al calcular diversos rangos y efectuar diferentes cuantizaciones y re-escalados.

En la mayoría de los casos, el desempeño de las Redes Neuronales se ve perjudicado al efectuarse la cuantización de sus parámetros y activaciones, debido a que esta se entrena con precisiones de datos superiores a los típicos en implementaciones de *hardware* embebido. Para solucionar este problema, una estrategia muy utilizada es la de realizar un ajuste fino a la red pre-entrenada en punto flotante, pero modelando en este caso los efectos de la cuantización de las variables involucradas. Este método conocido como QAT[14], que se traduciría como entrenamiento con cuantización, permite que los valores de los parámetros de la red se reajusten (re-entrenen) para compensar los efectos de la cuantización con precisiones bajas, de manera que el modelo pueda ser exportado y ejecutado sin problemas en *hardware* embebido. Sin embargo, dada la necesidad de mantener una precisión adecuada para el cómputo de los gradientes (que generalmente son valores pequeños) y la posterior actualización

[14] *Quantization Aware Training*

de los parámetros de la red, se opta por utilizar la cuantización solamente durante los cómputos relacionados a la inferencia, manteniendo una copia de los parámetros en punto flotante y actualizando estos en lugar de las versiones cuantizadas. Si por el contrario se cuantizaran tanto los gradientes como las operaciones de actualización (*update*), los cambios en los parámetros por cada iteración serían tan grandes que el modelo nunca convergería (o lo haría muy lentamente) a la solución deseada. Por otro lado, para la ejecución de este proceso de QAT, podría ser conveniente aprovechar las ventajas ofrecidas por los entornos de programación y entrenamiento de DNNs más populares (como por ejemplo PyTorch) que están diseñados/optimizados para operaciones en punto flotante. En este caso, los tensores cuantizados (valores enteros) pueden ser re-convertidos al dividir estos por el mismo factor de escala S que se utilizó para cuantizarlos, obteniendo así versiones en punto flotante (*fake quantized*) que mantienen los efectos de la representación discreta de sus valores y que pueden ser utilizados en el entorno de programación tradicional.

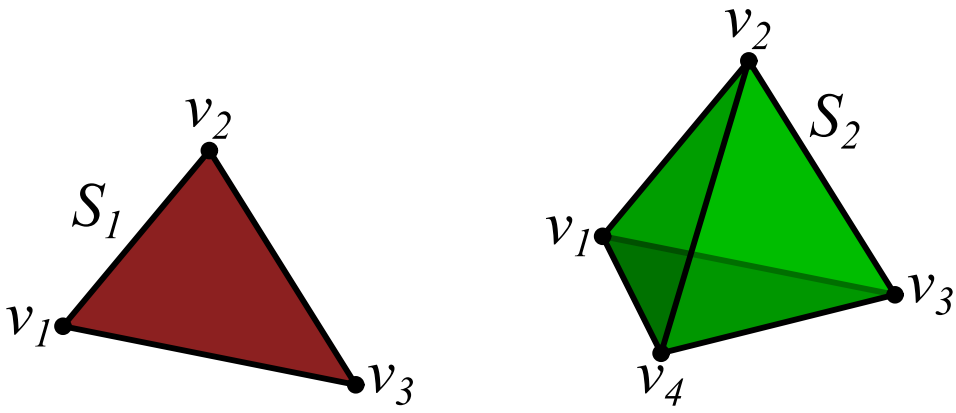
2.3. Funciones simpliciales

Las funciones simpliciales, introducidas en [61] y desarrolladas en [30], son esencialmente funciones lineales a tramos (PWL[15]), donde cada tramo lineal comprende una región denominada "símplice" y que se define como todo punto $\boldsymbol{x} \in \mathbb{R}^N$, $\boldsymbol{x} = [x_1\ x_2\ \cdots\ x_N]^T$ que es una combinación convexa de los $N+1$ vértices $\boldsymbol{v}_i \in \mathbb{R}^N$, de manera que:

$$S\left(\boldsymbol{v}_1, \boldsymbol{v}_2, \cdots, \boldsymbol{v}_{N+1}\right) = \left\{ \boldsymbol{x} \,\middle|\, \boldsymbol{x} = \sum_{i=1}^{N+1} \mu_i \boldsymbol{v}_i \right\} , \tag{2.14}$$

donde $0 \leqslant \mu_i \leqslant 1$ y $\sum_{i=1}^{N+1} \mu_i = 1$. Dicho de otra forma, un símplice es la región del espacio $\mathbb{R}^N$ que se encuentra delimitada (encerrada) por $N+1$ vértices $\boldsymbol{v}_i$, y donde la función simplicial es estrictamente lineal. En las Figuras 2.13a y 2.13b se ilustran dos ejemplos de símplices con vértices genéricos, en 2D y 3D respectivamente.

[15]Lineal a tramos - *Piece-wise linear*



(a) (b)

Figura 2.13: Ilustración de símplices genéricos en: a) dos dimensiones (2D); b) tres dimensiones (3D).

La función simplicial (dentro de un símplice) queda definida como:

$$f(\boldsymbol{x}) = \sum_{i=1}^{N+1} \mu_i f(\boldsymbol{v}_i) = \sum_{i=1}^{N+1} \mu_i c_i \ , \tag{2.15}$$

que equivale a la suma de los productos de los valores μ_i, que definen a los puntos del símplice de acuerdo a la Ecuación (2.14), con unos coeficientes $c_i = f(\boldsymbol{v}_i)$ que corresponden los valores de la función lineal (para dicho tramo) en los vértices $\boldsymbol{v}_i$. Si bien los valores μ_i dependen en cierta medida de los vértices $\boldsymbol{v}_i$, para el cálculo de la función simplicial de la Ecuación (2.15), $\boldsymbol{\mu} = [\mu_1 \ \mu_2 \ \cdots \ \mu_{N+1}]^T$ puede obtenerse conociendo simplemente los valores del punto $\boldsymbol{x}$ donde se desee calcular la función:

$$\boldsymbol{\mu} = \begin{bmatrix} x_{s_1} \\ x_{s_2} - x_{s_1} \\ x_{s_3} - x_{s_2} \\ \vdots \\ x_{s_N} - x_{s_{(N-1)}} \\ 1 - x_{s_N} \end{bmatrix} \ , \tag{2.16}$$

con x_{s_j} los elementos del vector ordenado $\boldsymbol{x}_s = \text{sort}(\boldsymbol{x})$ de menor a mayor, lo que implica que $x_{s_1} = \min(\boldsymbol{x})$ y $x_{s_N} = \max(\boldsymbol{x})$. Para el cálculo de los valores de $\boldsymbol{\mu}$ en la Ec. (2.16), se considera que los vértices $\boldsymbol{v}_i$ pertenecen a un hipercubo de

volumen unitario (con un total de 2^N vértices que poseen elementos $v_j \in \{0,1\}, \forall j$), por lo que los puntos interiores $\boldsymbol{x}$ de los símplice presentan elementos definidos como $x_j \in [0,1], \forall j$. Al ser los valores μ_i computados mediante las diferencias de los elementos ordenados, obteniendo así sus magnitudes relativas, se cumple con las condiciones de $0 \leqslant \mu_i \leqslant 1$ y $\sum_{i=1}^{N+1} \mu_i = 1$, de manera que estos representan "porciones" del rango de la entrada $\boldsymbol{x}$. Para el cómputo de la función en cualquier símplice definido dentro del dominio de entrada (hipercubo unitario), solamente es necesario cambiar el conjunto de coeficientes c_i por el que corresponda al símplice en el que se halla el vector de entrada $\boldsymbol{x}$.

En cuanto a su implementación en *hardware*, esta se realiza mediante la comparación de las entradas al operador con una rampa digital (contador), lo que genera una codificación en tiempo de las mismas mediante señales binarias cuya duración en estado "alto" o 1 lógico, en ciclos de reloj, corresponde al valor entero de la entrada correspondiente. Por cada ciclo de reloj que dure la rampa digital, estas entradas codificadas en tiempo (o señales del tipo PWM[16]), son utilizadas para producir una dirección con la cual seleccionar en memoria (o registros locales) un coeficiente particular y acumularlo. De esta forma, la duración en ciclos de reloj de cada dirección diferente corresponde a los valores μ_i descritos en la Ec. (2.16), mientras que la acumulación de un mismo coeficiente por los ciclos que dura su dirección representa el producto parcial $\mu_i c_i$. Este proceso se ilustra en la Fig. 2.14, en la que se tiene un operador simplicial de 3 entradas, lo que conlleva a una selección de coeficientes entre 8 posibles valores (desde c_1 hasta c_8). Una vez finalizada la rampa (habiendo alcanzado el valor máximo de la precisión elegida), el valor del final del acumulador de coeficientes da como resultado la salida de la función simplicial.

Una de las ventajas principales de esta arquitectura de procesamiento es que el producto del vector $\boldsymbol{\mu}$ por el de coeficientes se realiza como sumas en tiempo, por lo que se evita el uso de multiplicadores (para los productos parciales $\mu_i c_i$), que generalmente se caracterizan por ocupar un gran área en *chip* y presentar un consumo de potencia elevado. Por otra parte, la duración del cómputo depende de la resolución de la rampa (cantidad de pasos de esta) y que depende solamente de la precisión y rango de las entradas. Es por esto que, a menor precisión de

[16]Modulación por ancho de pulsos - *Pulse-width modulation*

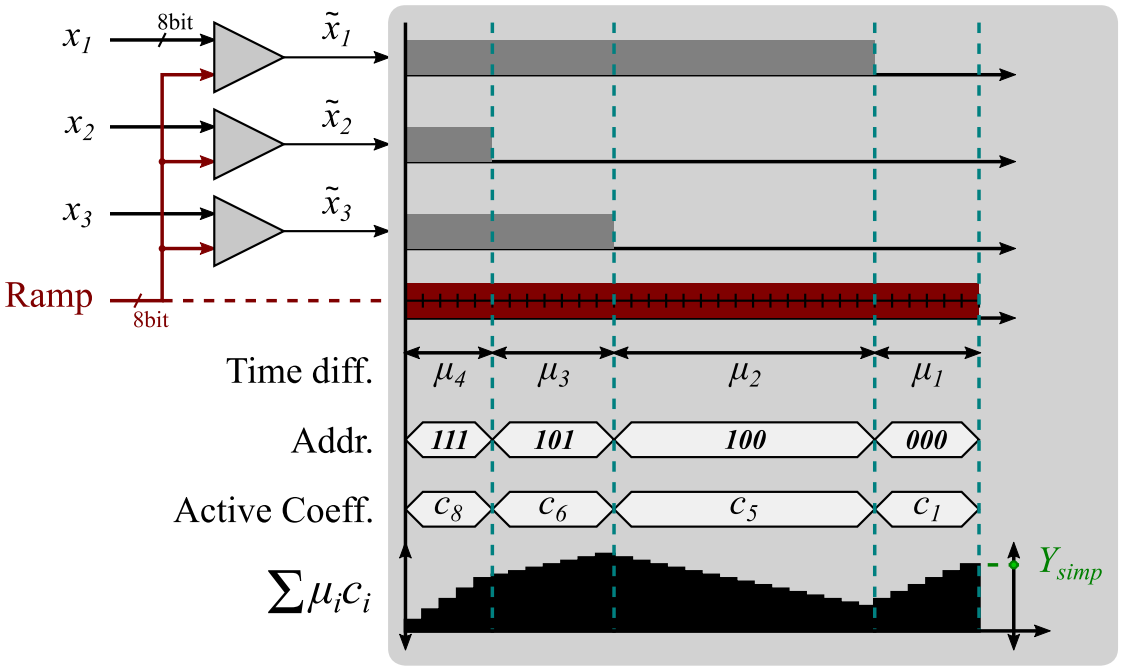
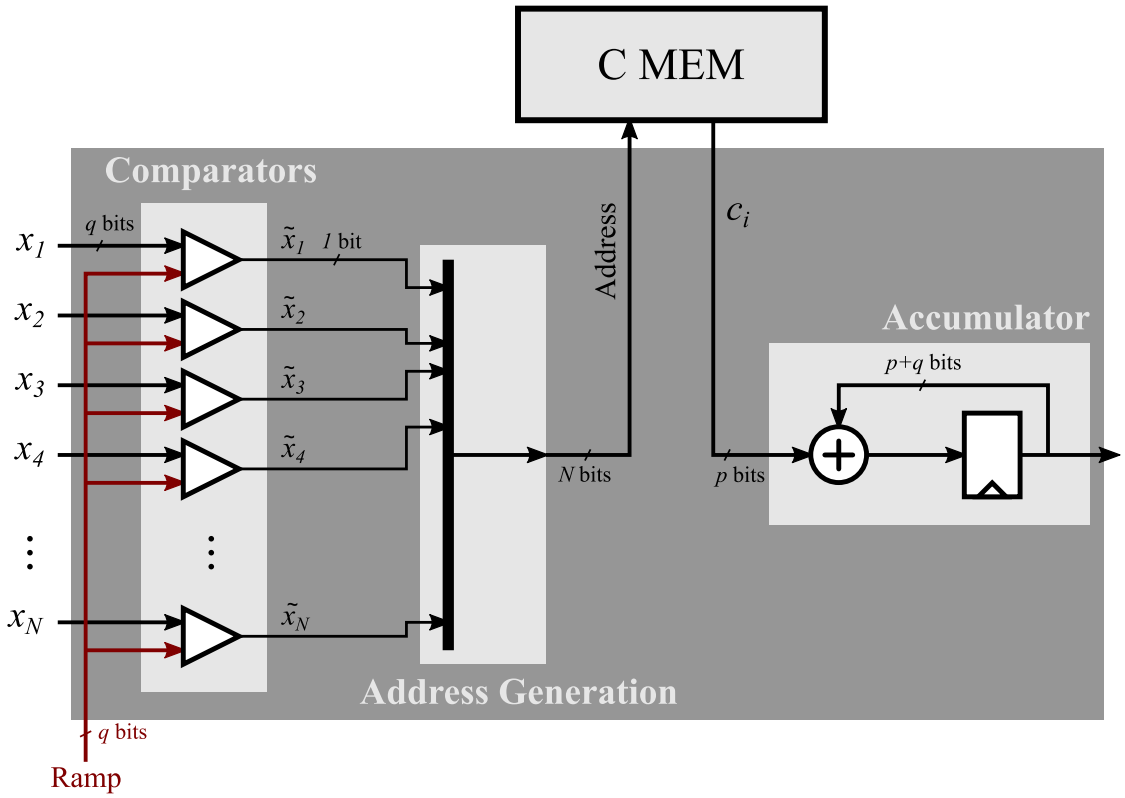


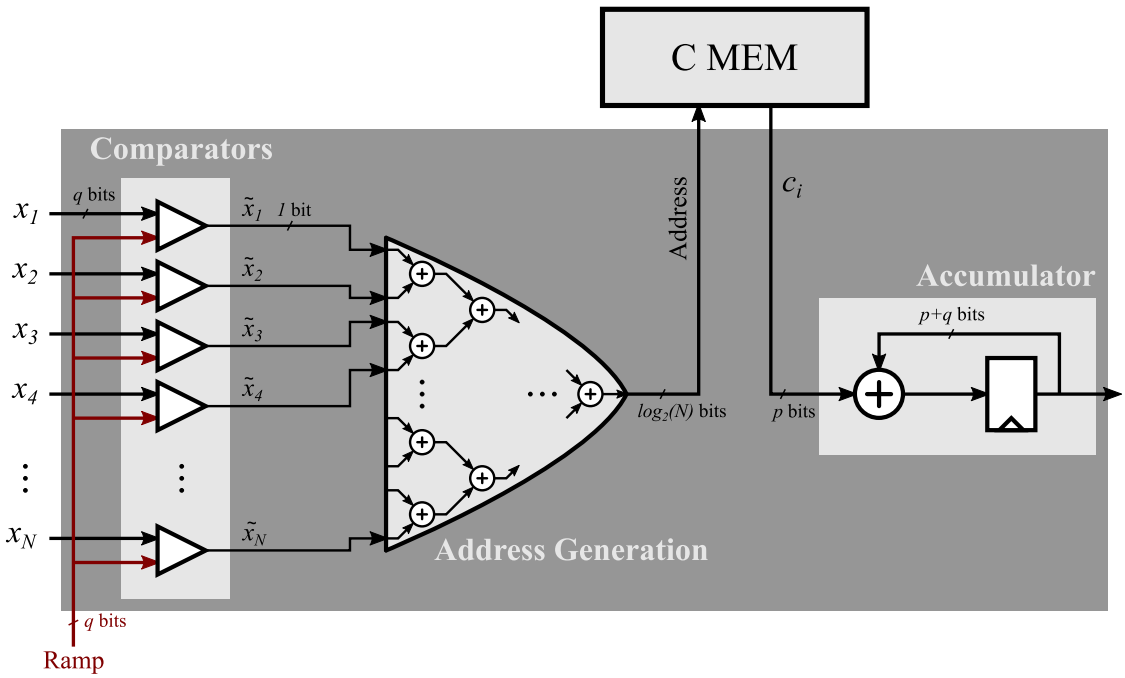
Figura 2.14: Estructura de cómputo simplicial.

entradas, menor tiempo de cómputo se requiere para que se realice el cálculo de la función simplicial. De esta forma, al estar el tiempo de cómputo ligado solamente a la precisión de las entradas, es independiente al número de las mismas, por lo que se vuelve más eficiente mientras mayor sea la cantidad de entradas al bloque. Dichas características hacen que este modelo de procesador simplicial sea de gran utilidad para la implementación de *hardware* eficiente con el cuál ejecutar Redes Neuronales en circuitos integrados. Esta arquitectura de cómputo se detalla en la Fig. 2.15a, donde las salidas binarias de los comparadores son combinadas para generar la dirección con la cual seleccionar al coeficiente c_i de memoria (C MEM) y sumarlo en el posterior acumulador.

Sin embargo, a medida que la cantidad de entradas N se incrementa, el número de coeficientes o pesos requeridos para el cómputo simplicial crece exponencialmente, específicamente en un factor de 2^N, lo que conlleva grandes requisitos de memoria para almacenar dichos parámetros. Esto se debe a que para efectuar el cómputo simplicial se deben definir los valores que toma la función en todos los símplices posibles (en todos los vértices del dominio de entrada), de manera que se pueda seleccionar el coeficiente adecuado sin importar a qué símplice pertenece el vector de entradas, lo que determinará la secuencia de direcciones de coeficientes a seleccionar. Como al-



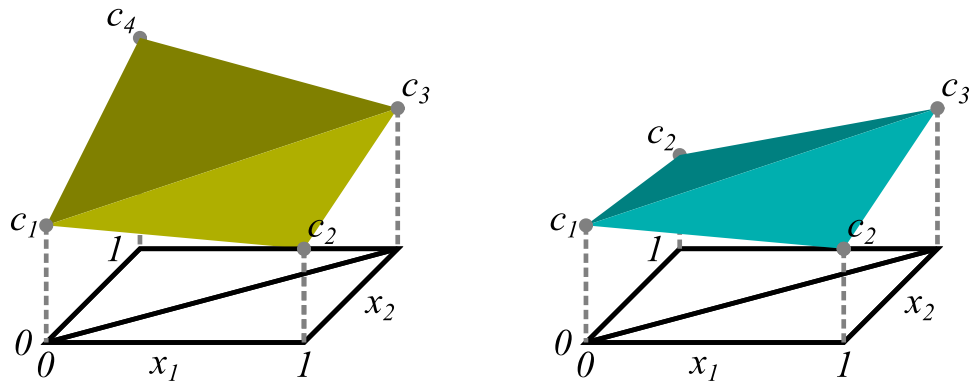
(a)



(b)

Figura 2.15: Arquitecturas de procesamiento para funciones simpliciales: a) simplicial pura; b) simplicial simétrica.

ternativa, [37] propone la formulación denominada simplicial simétrica (SymSim[17]), que emplea siempre el mismo conjunto de $N+1$ coeficientes, sin importar el símplice que corresponde a las entradas, lo que reduce significativamente los requerimientos de memoria del operador. Debido a que la única diferencia entre la función simplicial pura y esta variante radica en el uso de los coeficientes, las expresiones para el cálculo de la función simplicial simétrica son análogas a las ecuaciones anteriormente descritas en esta sección. En términos de *hardware*, esta variante simétrica (Fig. 2.15b) del operador simplicial implica la suma de las entradas convertidas en tiempo (señales PWM) para generar la dirección de coeficientes, en lugar de simplemente combinar dichas señales binarias, mientras que los demás bloques se mantienen de la versión simplicial pura (Fig. 2.15a). Esto se debe a que, al ser siempre los mismos $N+1$ coeficientes, su dirección (índice) depende de la cantidad de 1s que hay en las entradas PWM y no de sus posiciones. A pesar del agregado de un sumador (árbol de sumas) para generar la dirección de coeficientes, al tratarse de sumas de valores binarios, este bloque de *hardware* no representa un gran incremento tanto en área como en consumo, especialmente si se compara con la reducción en registros (o SRAM) de almacenamiento para coeficientes con respecto al operador simplicial puro.



(a) (b)

Figura 2.16: Ejemplos de funciones simpliciales en dos dimensiones (dos entradas): a) simplicial pura; b) simplicial simétrica.

En la Fig. 2.16 se ilustran dos ejemplos de funciones simpliciales de dos entradas: una simplicial pura (Fig. 2.16a) y otra simétrica (Fig. 2.16b), ambas definidas sobre

[17]Simplicial Simétrica - *Symmetric Simplicial*

un rango $x_1, x_2 \in [0, 1]$. En ambos casos, puede observarse que en dicho dominio de entrada se definen dos símplices, uno para $x_1 > x_2$ (con coeficientes c_1, c_2 y c_3) y otro para $x_1 < x_2$ (con c_1, c_3 y c_4), donde para el caso simplicial puro se describen dos funciones lineales completamente diferentes (teniendo como único requisito la continuidad en $x_1 = x_2$). Por el contrario, el caso simétrico cuenta con funciones lineales en uno de los símplices que son el reflejo de la función en el otro, produciendo un eje de simetría en los valores $x_1 = x_2$. Debido a que el uso de los coeficientes es siempre el mismo (en este caso c_1, c_2 y c_3), la función simplicial simétrica ignora el orden de las entradas, produciendo el mismo resultado tanto para el vector $\boldsymbol{x} = [x_1, x_2]$ como para $\boldsymbol{x} = [x_2, x_1]$, dando importancia solo a las magnitudes relativas de sus elementos. A pesar de la reducción en memoria de coeficientes, la simetría impuesta por esta variante restringe considerablemente el tipo de funciones que puede representar.

Sin embargo, tanto las operaciones morfológicas básicas (MinPool y MaxPool), como los filtros de mediana, mencionados en la Sub-Sección 2.1.5, son esencialmente simétricos, por lo que la arquitectura simplicial simétrica mostrada en la Fig. 2.15b es capaz que implementar este tipo de funciones al simplemente utilizar coeficientes o pesos simétricos específicos. En base al cálculo de los valores μ presentados en la Ec. (2.16), puede observarse que eligiendo los coeficientes $\boldsymbol{c} = [c_1, c_2, \cdots, c_N, c_{N+1}] = [1, 0, \cdots, 0, 0]$, es decir $c_1 = 1$ y 0 en el resto, el resultado de la función simplicial simétrica es x_{s_1}, que corresponde al valor mínimo. Por el contrario, si la selección de coeficientes es de $\boldsymbol{c} = [1, 1, \cdots, 1, 0]$, lo que equivale a $c_{N+1} = 0$ y 1 en los demás pesos simétricos, el resultado es igual a x_{s_N} que representa el valor máximo del vector de entrada $\boldsymbol{x}$. De forma similar puede hallarse la mediana, al utilizar la primera mitad de coeficientes con valor 1 (hasta $c_{\lceil N/2 \rceil}$), mientras que los valores siguientes se fijan en 0.

Con las estructuras de cómputo simpliciales puras (Fig. 2.15a) y simétricas (Fig. 2.15b) se han fabricado diversos circuitos integrados para el procesamiento de imágenes. Un ejemplo de esto es el *chip* presentado en [33], fabricado en 90nm, que dispone de un elemento de procesamiento (PE) simplicial puro con un comparador para una entrada de 6 bits, 1 bit de coeficientes y 5 bits de salida. La generación de direcciones (5 bits) para la selección de coeficientes se realiza a partir de la entrada comparada

(PWM) y las de 4 celdas vecinas. El *chip* fabricado en [33] implementa un arreglo de 64×64 celdas de procesamiento, con el que se reporta una eficiencia de 817,8 GOPS/W. Un caso más reciente de la implementación de operadores simpliciales (como el de la Fig. 2.15a) en ASICs es el fabricado por [35], llamado MORPHO8PWL y fabricado con la tecnología CMOS 55nm. Este *chip* cuenta con una columna de 64 PEs, cada uno con tres entradas de 8 bits, que son comparadas y utilizadas para la selección de uno entre 32 coeficientes de 3 bits por ciclo de reloj, produciendo una salida de 12 bits. Para la obtención de la dirección (5 bits) de coeficientes, se utilizan las tres entradas codificadas en PWM, junto a las propias de los PEs vecinos en la fila superior e inferior, logrando una configuración espacial en forma de "+" o "×". Con esta arquitectura, en [35] se reporta una eficiencia de 2,95 TOPS/W. Por otro lado, en [36] se presenta un procesador morfológico basado en funciones simpliciales simétricas de 1 bit de entradas y coeficientes. Este procesador denominado MORPHO1SYM y fabricado en 55nm, cuenta con un arreglo de 48×48 PEs similares a lo mostrado en la Fig. 2.15b, donde cada PE toma un vecindario de 9 entradas (3). Con este *chip*, en [36] se reporta una eficiencia de 293 TOPS/W. Naturalmente, estos valores de eficiencia energética corresponden a distintas definiciones de operación básica (OP), por lo que para la comparación con los SoCs fabricados en esta tesis, que se realizará en el Capítulo 5, dichos valores serán escalados a una definición común de operación elemental.

Debido a sus implementaciones eficientes en *hardware* que carecen de multiplicadores u otras estructuras de grandes dimensiones y consumo energético, así como también a su independencia del número de entradas para los tiempos de cómputo, las funciones simpliciales simétricas resultan prometedoras para su uso en cuanto a la ejecución o inferencia de DNNs. Por otra parte, estas estructuras de cómputo tienen la capacidad de representar funciones morfológicas sin aproximaciones y, al desarrollarse algoritmos de *backpropagation* tal y como se mostrará en el Capítulo 3, de aprenderlas mediante el simple ajuste de sus coeficientes. Esto vuelve a las arquitecturas simpliciales simétricas especialmente útiles para las aplicaciones de MNN mencionadas en la Sub-Sección 2.1.5.

Capítulo 3

Funciones simpliciales aplicadas a Redes Neuronales

Tal como se mencionó en la Sección 1.2, el objetivo principal de esta tesis es generar operadores en *hardware* eficientes para la ejecución de Redes Neuronales Profundas (DNN), las cuales generalmente presentan una extensa cantidad de capas y muchos parámetros, lo que requiere de un gran número de operaciones. Con esto en mente, esta tesis aprovecha algoritmos que ya cuentan con estructuras de cómputo (*hardware*) eficientes, en particular la familia de funciones simpliciales descritas en la Sección 2.3, debiendo efectuar ciertas modificaciones a dichos algoritmos para que estos puedan integrarse correctamente en entornos de cómputo de Aprendizaje de Máquina (ML).

En este capítulo se describirá entonces el uso de las funciones simpliciales (en especial simétricas) para aplicaciones de NN, específicamente en cuanto a algoritmos y *software*, junto con las modificaciones necesarias para su integración en DNN y CNN. En primera instancia, se desarrollarán en la Sección 3.1 los cálculos de gradientes en capas simpliciales necesarios para la ejecución del algoritmo de *Back-propagation* y el consecuente entrenamiento de sus parámetros, convirtiéndose en el mecanismo fundamental para su inclusión en Redes Neuronales. En la Sección 3.2 se presentarán modificaciones a los algoritmos simpliciales que permitan un mejor desempeño general de las redes con este tipo de capas, priorizando la eficiencia de su implementación en *hardware* con tales modificaciones o adiciones, concluyendo en la

función simplicial simétrica a canales separados (ChSymSim). Dada la importancia de una correcta inicialización de parámetros en DNNs, para una rápida y estable convergencia durante su entrenamiento, la Sección 3.3 mostrará las ecuaciones obtenidas para una óptima inicialización de los algoritmos simpliciales (particularmente ChSymSim), considerando tanto modo *forward* como *backward* y funciones de activación del tipo ReLU, junto con los resultados del entrenamiento de dos CNNs en comparación con un método de inicialización estándar. Finalmente, la Sección 3.4 detallará algunos de los experimentos realizados con DNNs basadas en funciones simpliciales simétricas, como ejemplos exitosos de la inclusión de dichas funciones, consiguiendo resultados cercanos o hasta superiores a los mismos modelos con capas convolucionales tradicionales.

3.1. *Backpropagation* en funciones simpliciales

Como ya se mencionó anteriormente, tanto la función simplicial pura como la simplicial simétrica (SymSim) ya presentan implementaciones eficientes en *hardware*, y fueron empleadas para el cálculo de filtros de una cantidad reducida de entradas, lo que representaría un precedente para su integración en DNNs. Sin embargo, estos casos de uso se realizaron con conjuntos de pesos/coeficientes fijos y conocidos, por lo que el desarrollo de sus gradientes es fundamental a la hora de poder introducirlas en Redes Neuronales, otorgándoles la capacidad de "entrenar" sus parámetros por medio del algoritmo de *backpropagation* junto al resto de capas que conforman cualquier DNN o CNN estándar. Sabiendo que para minimizar los requisitos de memoria en implementaciones de Redes Neuronales, resulta más conveniente utilizar la función simplicial simétrica, el cálculo de los gradientes será desarrollado particularmente para esta variante. Se comentarán además los pasos a seguir para obtener dichos gradientes en el caso de la función simplicial pura, dado que dicho desarrollo no difiere mucho del de su variante simétrica.

En las expresiones originales de la función simplicial mostradas en el capítulo anterior, Sección 2.3, el cálculo de los valores $\boldsymbol{\mu}$ se realizó considerando al vector de entrada $\boldsymbol{x}$ dentro de un hipercubo unitario, más precisamente $x_i \in [0, 1]\,,\ \forall i \in R^n$. Para este análisis, en cambio, se extiende y generaliza dicho rango introduciendo un

Algoritmo 2 Función simplicial simétrica (SymSim) en modo *forward*

Entrada:

- Vector de entrada $\boldsymbol{x} = [x_1\ x_2\ \cdots\ x_N]^T$, $x_i \in [m, M]$, con m y M como los valores mínimo y máximo del dominio de entrada, respectivamente;
- coeficientes (pesos) simétricos $\boldsymbol{C} \in \mathbb{R}^{(N+1)\times F}$.

Salida:

- Función SymSim $\boldsymbol{y} = f_s(\boldsymbol{x})$, $f_s : \mathbb{R}^N \to \mathbb{R}^F$;
- valores intermedios $\boldsymbol{\mu}$ requeridos para el cómputo de gradientes;
- índices $\boldsymbol{Ind}_s$ de la operación de ordenamiento (*sort*), los cuales también serán utilizados para el cálculo de los gradientes.

$[\boldsymbol{x}_s, \boldsymbol{Ind}_s] \leftarrow \text{sort}(\boldsymbol{x})$

$$\boldsymbol{\mu} \leftarrow \begin{bmatrix} \boldsymbol{x}_s \\ M \end{bmatrix} - \begin{bmatrix} m \\ \boldsymbol{x}_s \end{bmatrix}$$

$\boldsymbol{y} \leftarrow \boldsymbol{C}^T\boldsymbol{\mu}$

return $\boldsymbol{y}$, $\boldsymbol{Ind}_s$, $\boldsymbol{\mu}$

valor mínimo m y un máximo M, es decir $x_i \in [m, M]$ para cada una de las componentes, con valores arbitrarios m y M en $\mathbb{R}$, que pueden ser valores tanto positivos como negativos, siempre que respeten $m < M$. Con esta extensión, y expandiendo a múltiples salidas al utilizar F diferentes filtros simétricos, las ecuaciones de la función SymSim pueden compilarse en el Algoritmo 2, donde se puede observar que ahora el vector $\boldsymbol{\mu} \in [0, (M-m)]^{N+1}$, que además cumple con la condición de $\sum_{i=1}^{N+1} \mu_i = M - m$, se calcula como

$$\boldsymbol{\mu} = \begin{bmatrix} x_{s_1} - m \\ x_{s_2} - x_{s_1} \\ x_{s_3} - x_{s_2} \\ \vdots \\ x_{s_N} - x_{s_{(N-1)}} \\ M - x_{s_N} \end{bmatrix}. \tag{3.1}$$

Cabe mencionar que se puede emplear también el Algoritmo 2 para el cálculo de la

función simplicial pura, considerando que el grupo de coeficientes $\boldsymbol{C}$ (que proviene de calcular la función PWL en los vértices simpliciales) es diferente según el orden de las entradas, agregando otra dimensión a la matriz $\boldsymbol{C}$, y siendo seleccionados en base a los índices $\boldsymbol{Ind}_s$.

Para realizar el proceso de *backpropagation* se deben computar principalmente dos gradientes o derivadas parciales por capa: la derivada de la función de costo con respecto a los pesos ($\partial L/\partial \boldsymbol{C}$), la cual se usará para actualizar los pesos en cada paso del entrenamiento y la derivada con respecto a la entrada ($\partial L/\partial \boldsymbol{x}$), la cual se propagará a otras capas de la red. El primer caso de estas dos derivadas parciales ($\partial L/\partial \boldsymbol{C}$) resulta ser el más directo de calcular, al ser los coeficientes usados en un simple producto matricial. Del Algoritmo 2 se tiene que la salida está definida como:

$$\boldsymbol{y} = \boldsymbol{C}^T \boldsymbol{\mu} \ ,$$

y expresada en forma matricial se convierte en:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_F \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,F} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,F} \\ \vdots & \vdots & \ddots & \vdots \\ c_{N+1,1} & c_{N+1,2} & \cdots & c_{N+1,F} \end{bmatrix}^T \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_{N+1} \end{bmatrix} \ ,$$

Si se realiza el producto matricial (considerando $j = 1, 2, \cdots, F$) se tiene que

$$\begin{aligned} y_j &= \sum_{i=1}^{N+1} c_{i,j}\mu_i \\ y_j &= c_{1,j}\mu_1 + c_{2,j}\mu_2 + \cdots + c_{N+1,j}\mu_{N+1} \ . \end{aligned} \tag{3.2}$$

Debido a que la función de costo $L = f\left(\boldsymbol{y}_{pred}, \boldsymbol{y}_{true}\right)$ es un escalar, tal y como ocurre en los ejemplos de las Ecs. (2.2) y (2.3), el jacobiano con respecto al vector

de coeficientes tiene las mismas dimensiones que dicho vector:

$$\frac{\partial L}{\partial \boldsymbol{C}} = \begin{bmatrix} \frac{\partial L}{\partial c_{1,1}} & \frac{\partial L}{\partial c_{1,2}} & \cdots & \frac{\partial L}{\partial c_{1,F}} \\ \frac{\partial L}{\partial c_{2,1}} & \frac{\partial L}{\partial c_{2,2}} & \cdots & \frac{\partial L}{\partial c_{2,F}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial c_{N+1,1}} & \frac{\partial L}{\partial c_{N+1,2}} & \cdots & \frac{\partial L}{\partial c_{N+1,F}} \end{bmatrix}$$

Si se aplica la regla de la cadena que se muestra en la Ec. (2.5), esta derivada parcial se vuelve

$$\frac{\partial L}{\partial \boldsymbol{C}} = \begin{bmatrix} \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial c_{1,1}} & \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial c_{1,2}} & \cdots & \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial c_{1,F}} \\ \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial c_{2,1}} & \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial c_{2,2}} & \cdots & \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial c_{2,F}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial c_{N+1,1}} & \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial c_{N+1,2}} & \cdots & \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial c_{N+1,F}} \end{bmatrix}.$$

Si se despeja la ecuación anterior, considerando las expresiones de la Ec. (3.2), se obtiene

$$\frac{\partial L}{\partial \boldsymbol{C}} = \begin{bmatrix} \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial y_1}{\partial c_{1,1}} & \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial y_2}{\partial c_{1,2}} & \cdots & \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial y_F}{\partial c_{1,F}} \\ \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial y_1}{\partial c_{2,1}} & \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial y_2}{\partial c_{2,2}} & \cdots & \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial y_F}{\partial c_{2,F}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial y_1}{\partial c_{N+1,1}} & \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial y_2}{\partial c_{N+1,2}} & \cdots & \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial y_F}{\partial c_{N+1,F}} \end{bmatrix} = \begin{bmatrix} (\mu_1)\frac{\partial L}{\partial \boldsymbol{y}} & (\mu_1)\frac{\partial L}{\partial \boldsymbol{y}} & \cdots & (\mu_1)\frac{\partial L}{\partial \boldsymbol{y}} \\ (\mu_2)\frac{\partial L}{\partial \boldsymbol{y}} & (\mu_2)\frac{\partial L}{\partial \boldsymbol{y}} & \cdots & (\mu_2)\frac{\partial L}{\partial \boldsymbol{y}} \\ \vdots & \vdots & \ddots & \vdots \\ (\mu_{N+1})\frac{\partial L}{\partial \boldsymbol{y}} & (\mu_{N+1})\frac{\partial L}{\partial \boldsymbol{y}} & \cdots & (\mu_{N+1})\frac{\partial L}{\partial \boldsymbol{y}} \end{bmatrix},$$

por lo que finalmente, el gradiente de los pesos simpliciales simétricos equivale a

$$\frac{\partial L}{\partial \boldsymbol{C}} = \boldsymbol{\mu}\left(\frac{\partial L}{\partial \boldsymbol{y}}\right)^T. \tag{3.3}$$

Para el gradiente de los coeficientes (o pesos) en la función simplicial pura, se sabe que solo un grupo se utilizó dada una cierta entrada, por lo que el gradiente tiene valores no nulos solamente para ese grupo de pesos.

Siendo una función lineal a tramos, tal y como sucede con la activación ReLU, se puede calcular el gradiente del algoritmo simplicial simétrico con respecto a su entrada ($\partial L/\partial \boldsymbol{x}$) como el de una función lineal cualquiera, donde se usa un cierto grupo de coeficientes según a qué región o símplice pertenece la entrada. Sin embargo, resulta conveniente realizar una derivación del gradiente paso a paso, en base a los

cálculos expresados en el Algoritmo 2. De manera similar al gradiente de pesos, al ser la función de costo L un escalar, el jacobiano con respecto a los valores de $\boldsymbol{\mu}$ puede expresarse como

$$\frac{\partial L}{\partial \boldsymbol{\mu}} = \begin{bmatrix} \frac{\partial L}{\partial \mu_1} \\ \frac{\partial L}{\partial \mu_2} \\ \vdots \\ \frac{\partial L}{\partial \mu_{N+1}} \end{bmatrix},$$

que al despejar las derivadas parciales de sus elementos con la regla de la cadena de la Ec. (2.5) y la expresión de la salida simplicial simétrica de la Ec. (3.2), se obtiene

$$\frac{\partial L}{\partial \boldsymbol{\mu}} = \begin{bmatrix} \sum_{j=1}^{F} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial \mu_1} \\ \sum_{j=1}^{F} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial \mu_2} \\ \vdots \\ \sum_{j=1}^{F} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial \mu_{N+1}} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{F} (c_{1,j}) \frac{\partial L}{\partial y_j} \\ \sum_{j=1}^{F} (c_{2,j}) \frac{\partial L}{\partial y_j} \\ \vdots \\ \sum_{j=1}^{F} (c_{N+1,j}) \frac{\partial L}{\partial y_j} \end{bmatrix},$$

que en forma matricial puede expresarse como

$$\frac{\partial L}{\partial \boldsymbol{\mu}} = \boldsymbol{C} \frac{\partial L}{\partial \boldsymbol{y}} . \tag{3.4}$$

Tomando la definición del cálculo de $\boldsymbol{\mu}$ de la Ec. (3.1) y expresando el jacobiano del vector de entradas ordenadas $\boldsymbol{x}_s$, así como también aplicando la regla de la cadena de la Ec. (2.5), se obtiene

$$\frac{\partial L}{\partial \boldsymbol{x}_s} = \begin{bmatrix} \frac{\partial L}{\partial x_{s_1}} \\ \frac{\partial L}{\partial x_{s_2}} \\ \vdots \\ \frac{\partial L}{\partial x_{s_N}} \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial \mu_1} \frac{\partial \mu_1}{\partial x_{s_1}} + \frac{\partial L}{\partial \mu_2} \frac{\partial \mu_2}{\partial x_{s_1}} \\ \frac{\partial L}{\partial \mu_1} \frac{\partial \mu_2}{\partial x_{s_2}} + \frac{\partial L}{\partial \mu_3} \frac{\partial \mu_3}{\partial x_{s_2}} \\ \vdots \\ \frac{\partial L}{\partial \mu_N} \frac{\partial \mu_N}{\partial x_{s_N}} + \frac{\partial L}{\partial \mu_{N+1}} \frac{\partial \mu_{N+1}}{\partial x_{s_N}} \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial \mu_1} - \frac{\partial L}{\partial \mu_2} \\ \frac{\partial L}{\partial \mu_2} - \frac{\partial L}{\partial \mu_3} \\ \vdots \\ \frac{\partial L}{\partial \mu_N} - \frac{\partial L}{\partial \mu_{N+1}} \end{bmatrix},$$

que también puede expresarse como

$$\frac{\partial L}{\partial \boldsymbol{x}_s} = \begin{bmatrix} \frac{\partial L}{\partial \mu_1} \\ \frac{\partial L}{\partial \mu_2} \\ \vdots \\ \frac{\partial L}{\partial \mu_N} \end{bmatrix} - \begin{bmatrix} \frac{\partial L}{\partial \mu_2} \\ \frac{\partial L}{\partial \mu_3} \\ \vdots \\ \frac{\partial L}{\partial \mu_{N+1}} \end{bmatrix}. \tag{3.5}$$

Finalmente, para obtener el gradiente de la función SymSim con respecto a su entrada ($^{\partial L}/_{\partial \boldsymbol{x}}$), la no-linealidad del ordenamiento de las entradas, que también es una función lineal a tramos, puede resolverse simplemente reordenando el gradiente de la salida acorde a los índices producidos por la función de ordenamiento o *sort*, tal que

$$\frac{\partial L}{\partial \boldsymbol{x}} = \frac{\partial L}{\partial \boldsymbol{x}_s}\left(\boldsymbol{Ind}_{sb}\right) \ , \tag{3.6}$$

donde $\boldsymbol{Ind}_{sb}$ son los índices para revertir el ordenamiento efectuado por $\boldsymbol{x}_s = \text{sort}\left(\boldsymbol{x}\right)$.

Algoritmo 3 Función simplicial simétrica (SymSim) en modo *backward*

Entrada:

- Error propagado (derivada parcial) con respecto a la salida $^{\partial L}/_{\partial \boldsymbol{y}}$;
- conjunto de pesos simpliciales $\boldsymbol{C}$;
- resultado parcial $\boldsymbol{\mu}$ obtenido en el Algoritmo 2;
- conjunto de índices de ordenamiento $\boldsymbol{Ind}_s$ (también hallados en Alg. 2).

Salida:

- Derivada parcial (gradiente) $^{\partial L}/_{\partial \boldsymbol{C}}$, utilizada para la actualización de los pesos simpliciales;
- derivada parcial (gradiente) $^{\partial L}/_{\partial \boldsymbol{x}}$ empleada para el cálculo del error propagado hacia otras capas de la red.

$\frac{\partial L}{\partial \boldsymbol{C}} \leftarrow \boldsymbol{\mu}\left(\frac{\partial L}{\partial \boldsymbol{y}}\right)^T$

$\frac{\partial L}{\partial \boldsymbol{\mu}} \leftarrow \boldsymbol{C}\frac{\partial L}{\partial \boldsymbol{y}}$

$[\_, \boldsymbol{Ind}_{sb}] \leftarrow \text{sort}\left(\boldsymbol{Ind}_s\right)$

$\frac{\partial L}{\partial \boldsymbol{x}} \leftarrow \frac{\partial L}{\partial \boldsymbol{\mu}}\left(\boldsymbol{Ind}_{sb}\right) - \frac{\partial L}{\partial \boldsymbol{\mu}}\left(\boldsymbol{Ind}_{sb} + 1\right)$

return $\frac{\partial L}{\partial \boldsymbol{C}}, \frac{\partial L}{\partial \boldsymbol{x}}$

Con los resultados obtenidos por las Ecuaciones (3.3)-(3.6) se puede desarrollar el cómputo para el modo *backward* (cálculo de gradientes) para la función SymSim, el cual puede apreciarse en el Algoritmo 3. En dicho algoritmo se puede observar que los índices para revertir la función de ordenamiento (*sort*) se obtienen al ordenar los índices auxiliares obtenidos mediante el Algoritmo 2, que originalmente producían el vector de entradas ordenado $\boldsymbol{x}_s = \boldsymbol{x}\left(\boldsymbol{Ind}_s\right)$. A su vez, el Algoritmo 3 realiza en

un solo paso lo expresado por las Ecuaciones (3.5) y (3.6), cosa que en la mayoría de lenguajes de programación suele ser más eficiente.

Resulta importante destacar que durante el entrenamiento en punto flotante, debido a los problemas mencionados en la Sub-sección 2.1.1 con la saturación de la ReLU, resulta conveniente utilizar el cálculo dinámico de los rangos de la entrada para la función SymSim, en lugar de definirlos de antemano y saturar las salidas. Si bien esto podría presentar un problema a la hora de ser entrenadas con datos de entrada/salida que posean una gran dispersión en los rangos de sus valores, ya que las funciones serían muy distintas entre sí y resultaría complicado converger a una única solución, dicha variabilidad en los datos puede subsanarse con el uso de capas de BatchNorm. De esta forma se preserva toda la información de las entradas/salidas durante el entrenamiento y se deja que el algoritmo de SGD fije dichos valores del rango de entrada de la capa SymSim. Una vez que la red en punto flotante converge a un valor adecuado, dichos rangos se pueden fijar en base a las distribuciones de las entradas/salidas de cada capa SymSim.

Con solo tener los gradientes de la función simplicial simétrica, este algoritmo ya puede ser integrado en cualquier DNN o CNN, teniendo en cuenta que los Algoritmos 2 y 3 corresponden a capas FC simétricas, mientras que para capas de filtrado simétrico espacial (para reemplazar convoluciones) se deben tener algunas consideraciones adicionales. Sin embargo, el orden de las entradas a una capa cualquiera de la red suele tener importancia, efecto que se refleja en los pesos de las conexiones en capas lineales (o convolucionales) tradicionales. La función SymSim ignora completamente esta "espacialidad", lo que puede dar como resultado desempeños inferiores en una DNN que reemplace por completo capas convolucionales y/o FC por simpliciales simétricas estándar. Es por esta razón que se requiere modificar ligeramente el algoritmo SymSim original, teniendo en cuenta de alguna manera las posiciones de las entradas y no solo sus magnitudes relativas.

3.2. Algoritmo simplicial simétrico a canales separados

Si bien el algoritmo simplicial simétrico (SymSim) presenta implementaciones eficientes en *hardware*, lo que motivó a su uso en el desarrollo de esta tesis, este presenta limitaciones en cuanto a funcionalidad, forzando una simetría espacial e ignorando las posiciones de los elementos de entrada. De hecho, esta capacidad de interpretar la "espacialidad" de los datos de entrada o *features* intermedios es lo que ha permitido que las Redes Neuronales, particularmente CNNs, lograran tan buenos resultados en tareas complejas en el marco del procesamiento de imágenes. Siendo uno de los objetivos el de reemplazar capas tradicionales por algoritmos simpliciales simétricos, de modo que se aproveche su eficiencia a la hora de implementar Redes Neuronales completas en *hardware*, el carecer de estas aptitudes puede comprometer el desempeño de dichas redes cuando la tarea a realizar posee cierta complejidad.

De esta forma, una primera mejora que se podría realizar al algoritmo SymSim, inspirada en Redes Neuronales Morfológicas, es agregar pesos que multipliquen a las entradas, previo a la ejecución de la función Simétrica. Esto añadiría el efecto de "espacialidad" de la misma forma que ocurre con las capas lineales de una NN, pero con el costo de introducir cómputos adicionales, lo que elimina la principal ventaja de la implementación de la función simplicial que es la de prescindir de multiplicadores. Por otro lado, en términos de *hardware* también se debe considerar que estos productos parciales entre peso y entrada requerirá de un mayor número de bits para no perder precisión, lo que conlleva a la necesidad de agregar módulos para re-escalar dichos valores a la precisión original, o de lo contrario el tiempo de cómputo simplicial crecería exponencialmente. Como alternativa, dichos pesos pueden ser binarios, dando así un mínimo de efecto de "espacialidad" al seleccionar algunas entradas según su posición (sin hacer distinción entre estas) y anulando otras, simplificando considerablemente su posterior implementación en *hardware*. Estos pesos binarios constituyen una especie de elemento estructurante (SE[1]) y pueden ser entrenados utilizando técnicas de entrenamiento con cuantización (QAT). Si se introduce este elemento estructurante (preferentemente binario) al algoritmo

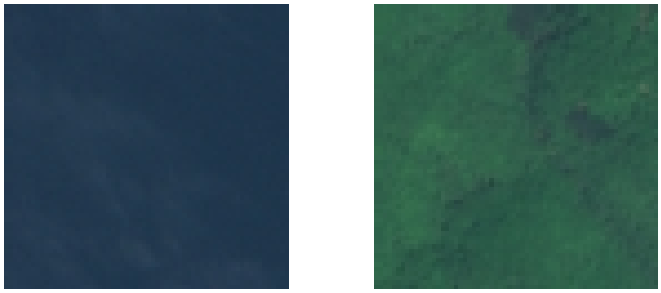
[1]Elemento Estructurante - *Structuring Element*

SymSim mostrado en las Secciones 2.3 y 3.1, resultan las siguientes ecuaciones:

$$\begin{aligned} \boldsymbol{z} &= \boldsymbol{x} \circ \boldsymbol{SE} \\ \boldsymbol{z}_s &= \text{sort}\,(\boldsymbol{z}) \\ \boldsymbol{\mu} &= \begin{bmatrix} \boldsymbol{z}_s \\ M \end{bmatrix} - \begin{bmatrix} m \\ \boldsymbol{z}_s \end{bmatrix} \\ \boldsymbol{y} &= \boldsymbol{C}^T \boldsymbol{\mu}\ . \end{aligned} \tag{3.7}$$

Cabe mencionar que este SE difiere del elemento estructurante en la definición de Morfología Matemática, ya que en operaciones morfológicas un SE consiste en una selección de entradas y no un producto. Con una selección de entradas se puede propagar un gradiente con respecto a la entrada; desafortunadamente, esto no es posible para la selección en si misma, por lo que no es "entrenable". Debido a esto, el SE como producto con peso binario es lo más cercano al caso de Morfología Matemática que puede ser entrenado mediante el método tradicional de *backpropagation*. Como una simplificación adicional se pueden utilizar SE compartidos para todos los *features* de salida (D_{out}), pasando a poseer dimensiones $N \times D_{in}$.

Si bien la inclusión del elemento estructurante (como producto por peso binario) permite tener en cuenta la posición de las entradas, esto puede no ser suficiente en algunos casos como el de las capas convolucionales (CNN), donde cada canal de entrada por separado aporta cierta información, pudiendo esta ser incluso más importante que lo relevante a la posición de los píxeles en 2D. Esto se debe a que cada canal representa un *feature* que fue extraído mediante un filtro (*kernel*) particular. Una idea intuitiva de este fenómeno podría ser un caso de clasificación de imágenes RGB, donde por un lado se tiene un escenario de una superficie de agua (Figura 3.1a), mientras que por otro lado se tiene una imagen aérea de un bosque (Figura 3.1b). Independientemente de la textura, es evidente que en la Figura 3.1a el color dominante es el azul, mientras que en la Figura 3.1b es el verde. El simple hecho de distinguir entre estos dos colores (por medio de los canales B y G) es suficiente para clasificar estas dos imágenes, pero las magnitudes relativas de dichos canales extraídas por la función SymSim, pudiera no aportar suficiente información en estos



(a) (b)

Figura 3.1: Imágenes del *dataset* EuroSAT de [3, 4]: a) SeaLake_1858; b) Forest_907.

casos, donde para píxeles con valores $[0\ , x\ , 0]$ y $[0\ , 0\ , x]$ el resultado de la función SymSim es idéntico.

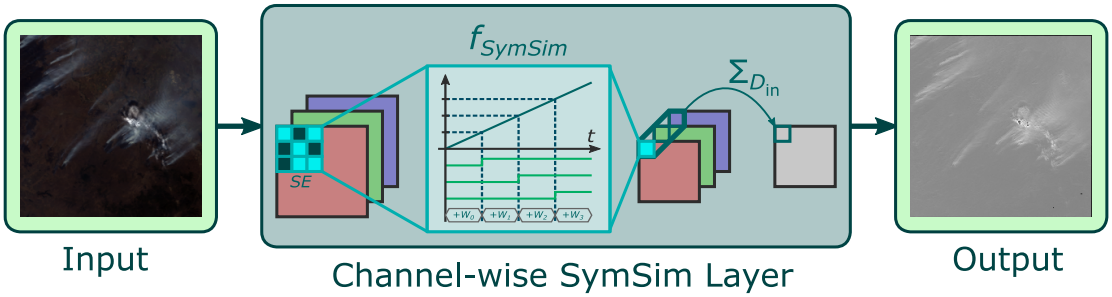


Figura 3.2: Ilustración de capa simplicial simétrica a canales separados (*ChSymSim*).

Agregar una distinción adicional entre dichos canales de entrada en una capa de filtrado simplicial simétrico (que opera como una convolución) resulta entonces fundamental, pudiendo ser implementada mediante un conjunto de coeficientes simpliciales diferente para cada canal de entrada y realizando el filtrado de un canal por vez, sumando los resultados de cada píxel de salida. A esta variante, que también incluye al elemento estructurante binario, se le puede denominar simplicial simétrica a canales separados, o en inglés *channel-wise symmetric simplicial* (ChSymSim), cuyo concepto general se encuentra ilustrado en la Fig. 3.2. Combinando la inclusión del elemento estructurante descrito en la Ec. (3.7) con el procesamiento a canales separados, se presenta en el Algoritmo 4 una descripción más detallada para la función ChSymSim, donde el cambio en la forma de la salida $\boldsymbol{Y} = \text{reshape}\,(\boldsymbol{Y}_c)$ se debe a que la forma 2D se encuentra "vectorizada" en la dimensión indicada por el sub-índice j de $\boldsymbol{Y}_c$, mientras la dimensión indicada por i corresponde a los canales (*features*) de salida D_{out}. Naturalmente, la función $\text{im2col}(\boldsymbol{X}, (k_H, k_W))$, que es esencial para

filtrados espaciales, también depende de otros parámetros tales como *stride*, *dilation* y *padding*, aunque este último puede aplicarse directamente a la entrada antes de ejecutar la función ChSymSim. Por simplicidad, tanto estos parámetros como los subíndices en las asignaciones de $\boldsymbol{H}$ y $\boldsymbol{\mu}$, fueron omitidos en el Algoritmo 4.

Algoritmo 4 Función simplicial simétrica a canales separados (ChSymSim)

Entrada:

- Tensor de entrada $\boldsymbol{X} \in \mathbb{R}^{H_{in} \times W_{in} \times D_{in}}$, $x_n \in [m, M]$, $n \in \{1, \cdots, H_{in} \times W_{in} \times D_{in}\}$, H_{in} y W_{in} el alto y ancho de $\boldsymbol{X}$, D_{in} el número de canales de entrada;
- elemento estructurante (binario) $\boldsymbol{SE} \in \mathbb{R}^{N \times D_{in}}$, $SE_t \in [0, 1]$, $t \in \{1, \cdots, N \times D_{in}\}$, $N = k_H \times k_W$ el tamaño del *kernel*;
- coeficientes (pesos) Simétricos $\boldsymbol{C} \in \mathbb{R}^{(N+1) \times D_{in} \times D_{out}}$;
- *bias* $\boldsymbol{b} \in \mathbb{R}^{D_{out}}$.

Salida:

- Tensor $\boldsymbol{Y} \in \mathbb{R}^{H_{out} \times W_{out} \times D_{out}}$, resultado de la función ChSymSim;
- valores intermedios $\boldsymbol{\mu}$;
- índices $\boldsymbol{Ind}_s$ de la operación de ordenamiento (*sort*).

for $i = 1 : D_{out}$ **do**
  $\boldsymbol{X}_c \leftarrow \text{im2col}(\boldsymbol{X}, (k_H, k_W))$
  for $j = 1 : (H_{out} \times W_{out})$ **do**
    for $k = 1 : D_{in}$ **do**
      $\boldsymbol{Z} \leftarrow \boldsymbol{X}_c \circ \boldsymbol{SE}$
      $[\boldsymbol{Z}_s, \boldsymbol{Ind}_s] \leftarrow \text{sort}(\boldsymbol{Z})$
      $\boldsymbol{\mu} \leftarrow \begin{bmatrix} \boldsymbol{Z}_s \\ M \end{bmatrix} - \begin{bmatrix} m \\ \boldsymbol{Z}_s \end{bmatrix}$
      $\boldsymbol{H} \leftarrow \boldsymbol{C}^T \boldsymbol{\mu}$
    end for
    $\boldsymbol{Y}_c[j, i] \leftarrow \left(\sum_{k=1}^{D_{in}} \boldsymbol{H}[j, k, i]\right) + \boldsymbol{b}[i]$
  end for
end for
$\boldsymbol{Y} \leftarrow \text{reshape}(\boldsymbol{Y}_c)$
return $\boldsymbol{Y}$, $\boldsymbol{\mu}$, $\boldsymbol{Ind}_s$

Para el caso de *backpropagation*, al ser la implementación del SE un simple

producto elemento a elemento con la entrada, los gradientes pueden hallarse como:

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{SE}} &= \boldsymbol{x} \circ \frac{\partial L}{\partial \boldsymbol{z}} \\ \frac{\partial L}{\partial \boldsymbol{x}} &= \boldsymbol{SE} \circ \frac{\partial L}{\partial \boldsymbol{z}} \ , \end{aligned} \tag{3.8}$$

donde la derivada parcial $^{\partial L}/_{\partial \boldsymbol{z}}$ se obtiene siguiendo el procedimiento para el gradiente con respecto a la entrada en el Algoritmo 3 (Sección 3.1). En cuanto al filtrado de canales por separados en sí mismo, este no requiere grandes modificaciones para el cálculo de gradientes en comparación con lo visto en la Sección 3.1, ya que los resultados parciales de la función SymSim simplemente se suman para hallar la salida final de la capa (*feature* de salida), de modo que resolviendo el jacobiano y aplicando regla de la cadena (de la misma forma que en la Sección 3.1), se tiene que

$$\frac{\partial L}{\partial \boldsymbol{H}} = \left\{ \frac{\partial L}{\partial \boldsymbol{H}\left[j,k,i\right]} \;\middle|\; \frac{\partial L}{\partial \boldsymbol{H}\left[j,k,i\right]} = \frac{\partial L}{\partial \boldsymbol{Y}_c\left[j,i\right]} \ , \forall k \right\} \ , \tag{3.9}$$

lo que se traduce en una expansión de la matriz derivada parcial $^{\partial L}/_{\partial \boldsymbol{Y}_c}$, añadiendo una dimensión intermedia y repitiendo los elementos $^{\partial L}/_{\partial y_c[j,i]}$ tantas veces como canales de entrada D_{in}. Al igual que lo que ocurre en capas convolucionales, la suma de un *bias* $\boldsymbol{b}$ resulta irrelevante para el cálculo del gradiente de entrada, al ser una suma de una constante. El gradiente de dichos *bias* se obtiene directamente a partir del gradiente de la salida:

$$\frac{\partial L}{\partial \boldsymbol{b}} = \sum_{j=1}^{H_{out} \times W_{out}} \frac{\partial L}{\partial \boldsymbol{Y}_c[j]} \ . \tag{3.10}$$

Con estos resultados mostrados en las Ecs. (3.8), (3.9) y (3.10), y considerando que el gradiente de un ordenamiento de los elementos de una matriz o tensor (cambio de forma) es el ordenamiento inverso de los gradientes propagados, se puede expandir el algoritmo de cálculo de gradientes para la función ChSymSim, tal y como se puede observar en el Algoritmo 5. Para el gradiente de pesos simpliciales $^{\partial L}/_{\partial \boldsymbol{C}}$ se realiza la suma de las contribuciones de todos los pasos de filtrado ($j = 1, \cdots, H_{out} \times W_{out}$), realizada mediante el producto matriz por vector $\boldsymbol{\mu}[:,:,k,i]^{\partial L}/_{\partial \boldsymbol{H}[:,k,i]}$, donde

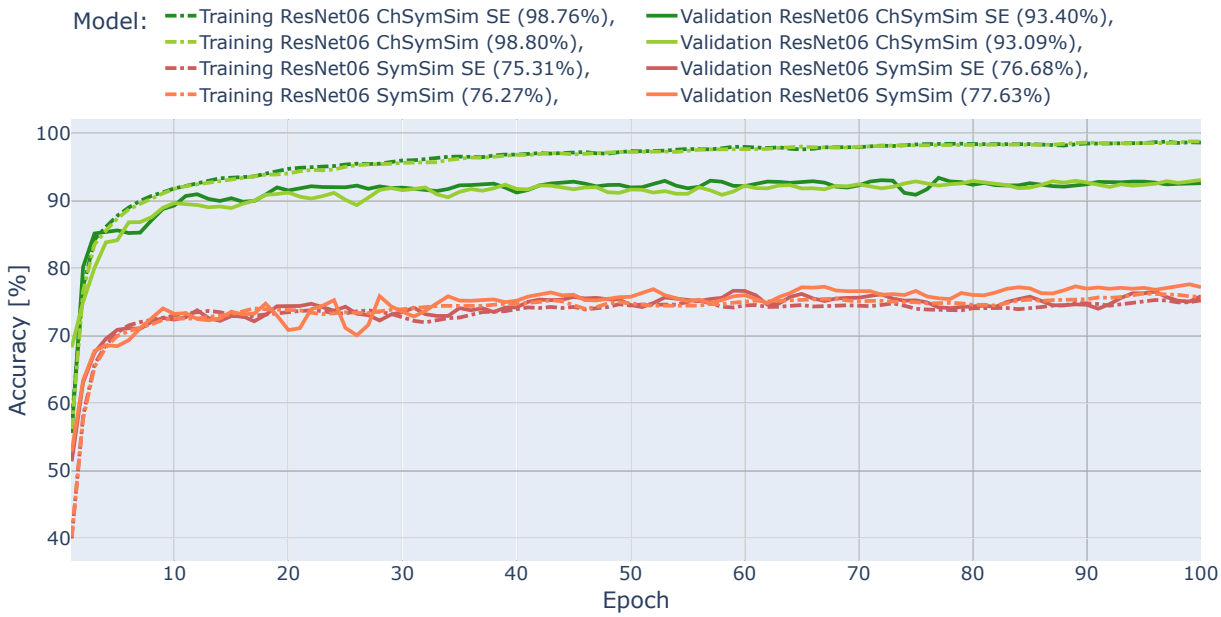


Figura 3.3: Comparación del desempeño para clasificar imágenes de EuroSAT ([3] y [4]) entre redes ResNet06 (Tabla D.4) con capas simpliciales simétricas y ChSymSim, con y sin SE.

se tiene $\boldsymbol{\mu} \in \mathbb{R}^{(N+1)\times(H_{out}\times W_{out})\times D_{in}}$, debido a que se utilizaron los mismos pesos para cada paso. Finalmente, se hace uso de la función col2im() para reconstruir la forma original de la imagen/tensor de entrada. Para esto, se toman las columnas del gradiente propagado, que son transformadas en porciones con la forma del *kernel* en 2D ($k_H \times k_W$), y ubicadas en el lugar correspondiente en una matriz/tensor con las mismas dimensiones de la entrada, realizando además la suma de los gradientes (propagados hasta ese momento) en los píxeles donde se produce un solapamiento.

Tanto para el Algoritmo 4 como para el Algoritmo 5, algunos índices fueron omitidos para simplificar sus expresiones. Un ejemplo de esto son los subíndices de las variables dentro de los ciclos *for* en el Algoritmo 4, donde para $\boldsymbol{\mu}$ se deben guardar los resultados en cada iteración para ser utilizados durante la etapa de *backpropagation*. De igual forma, se simplificaron las expresiones que involucran a la función de *sort* y las matrices de índices que éstos generan, donde se debe tener en cuenta que el ordenamiento ocurre con la dimensión que presenta el tamaño del *kernel* $N = k_H \times k_W$, conservando los resultados para cada paso del filtrado y canal de entrada D_{in}.

Para ilustrar los beneficios de la función simplicial simétrica a canales separa-

Algoritmo 5 Cálculo de gradientes para la función ChSymSim del Algoritmo 4

Entrada:

- Error propagado (derivada parcial) con respecto a la salida $\partial L/\partial \boldsymbol{Y}$;
- conjunto de pesos simpliciales $\boldsymbol{C}$;
- elementos estructurantes (binarios) $\boldsymbol{SE}$;
- resultado parcial $\boldsymbol{\mu}$;
- conjunto de índices de ordenamiento $\boldsymbol{Ind}_s$.

Salida:

- Derivada parcial (gradiente) $\partial L/\partial \boldsymbol{b}$, con respecto a los *bias*;
- derivada parcial (gradiente) $\partial L/\partial \boldsymbol{C}$, con respecto a los pesos simpliciales;
- derivada parcial (gradiente) $\partial L/\partial \boldsymbol{SE}$ con respecto a los elementos estructurantes;
- derivada parcial (gradiente) $\partial L/\partial \boldsymbol{X}$ con respecto a la entrada del Algoritmo 4.

$\frac{\partial L}{\partial \boldsymbol{Y}_c} \leftarrow \text{reshape}\left(\frac{\partial L}{\partial \boldsymbol{Y}}\right)$
$\frac{\partial L}{\partial \boldsymbol{b}} \leftarrow \sum_{j=1}^{(H_{out}\times W_{out})} \frac{\partial L}{\partial \boldsymbol{y}_c[j]}$
$\frac{\partial L}{\partial \boldsymbol{H}} \leftarrow \text{repeat}\left(\frac{\partial L}{\partial \boldsymbol{Y}_c}\right)$
for $k = 1 : D_{in}$ **do**
  for $i = 1 : D_{out}$ **do**
    $\frac{\partial L}{\partial \boldsymbol{C}[:,k,i]} \leftarrow \boldsymbol{\mu}[:,:,k]\frac{\partial L}{\partial \boldsymbol{H}[:,k,i]}$
  end for
  $\frac{\partial L}{\partial \boldsymbol{\mu}[:,:,k]} \leftarrow \sum_i^{D_{out}} \boldsymbol{C}[:,k,i]\left(\frac{\partial L}{\partial \boldsymbol{H}[:,k,i]}\right)^T$
end for
$[\_, \boldsymbol{Ind}_{sb}] \leftarrow \text{sort}\left(\boldsymbol{Ind}_s\right)$
$\frac{\partial L}{\partial \boldsymbol{Z}} \leftarrow \frac{\partial L}{\partial \boldsymbol{\mu}}\left(\boldsymbol{Ind}_{sb}\right) - \frac{\partial L}{\partial \boldsymbol{\mu}}\left(\boldsymbol{Ind}_{sb}+1\right)$
$\frac{\partial L}{\partial \boldsymbol{SE}} \leftarrow \sum_{j=1}^{(H_{out}\times W_{out})} \boldsymbol{X}_c[:,j,:] \circ \frac{\partial L}{\partial \boldsymbol{Z}[:,j,:]}$
$\frac{\partial L}{\partial \boldsymbol{X}_c} \leftarrow \boldsymbol{SE} \circ \frac{\partial L}{\partial \boldsymbol{Z}}$
$\frac{\partial L}{\partial \boldsymbol{X}} \leftarrow \text{col2im}\left(\frac{\partial L}{\partial \boldsymbol{X}_c}\right)$
return $\frac{\partial L}{\partial \boldsymbol{b}}, \frac{\partial L}{\partial \boldsymbol{C}}, \frac{\partial L}{\partial \boldsymbol{SE}}, \frac{\partial L}{\partial \boldsymbol{X}}$

dos presentada en los Algoritmos 4 y 5, respecto al uso de estructuras de cómputo SymSim puras, se entrenaron cuatro modelos de ResNet06 basados en el descrito en la Tab. D.4, reemplazando los filtros convolucionales por capas SymSim puras, ChSymSim, con y sin SE. Para los casos en los que no se utilizó SE en los mode-

los, simplemente se dejaron fijos todos los valores de los SE en 1, de modo que el producto resultase en el mismo valor que la entrada, y se eliminó la posibilidad de modificarlos (actualizarlos) durante el entrenamiento. Para más información sobre este tipo de arquitecturas de Redes Neuronales, se invita al lector a consultar el Apéndice D, donde se presentan más detalles sobre la estructura de sus capas y requisitos de memoria. Este experimento se realizó entrenando a los modelos mencionados para la clasificación de las imágenes satelitales del *dataset* EuroSAT ([3] y [4]), utilizando el mismo método de inicialización (Sección 3.3) y de entrenamiento que en el ejemplo que se presentará en la Subsección 3.4.3, con resultados que se muestran en la Fig. 3.3. En dicha figura se puede observar que el desempeño del uso de capas SymSim puras, que aplican la misma función Simétrica a todos los canales a la vez, es muy inferior al obtenido con la misma red pero con capas ChSymSim, donde se aplica una función Simétrica diferente a cada canal. La complejidad del modelo SymSim puro es, de hecho, insuficiente para poder "aprender" en base a los datos de entrenamiento, apenas superando el 75 % de los datos de entrenamiento clasificados correctamente. Debido a esta escasa complejidad en el modelo, los datos de validación fácilmente alcanzan resultados de clasificación similares a los de entrenamiento. De lo mostrado en la Fig. 3.3, es fácil notar las ventajas de la separación por canales del método ChSymSim sobre el simétrico puro, mientras que el uso de un SE (en este ejemplo) no parece aportar gran beneficio al algoritmo.

3.3. Inicialización de parámetros

La elección de valores iniciales "inadecuados" para los parámetros de cada capa en una DNN puede ocasionar que dicha DNN no sea capaz de converger a una solución cercana al mínimo error global o, en el mejor de los casos, que lo haga muy lentamente. Si se inicializaran todos los pesos en un mismo valor, por ejemplo, podría resultar en una simetría entre las neuronas de una misma capa que dificulte el aprendizaje de ciertos patrones de los datos de entrada. Parámetros con valores iguales limitan su respuesta a solo algunos de estos patrones de la entrada y a su vez producen gradientes similares que apenas modifican dichos parámetros, dejando a la red "atrapada" en una solución sub-óptima. Esto puede resolverse con una inicia-

lización aleatoria, especialmente cuando los valores pueden ser tanto positivos como negativos, rompiendo así esa simetría. Otros problemas que se pueden presentar en este tipo de redes son los llamados *vanishing gradients* y *exploding gradients*. Por un lado, el problema de *vanishing gradients* ocurre cuando los pesos de la red son tan pequeños que, al multiplicar al gradiente, lo van reduciendo hasta prácticamente hacerlo "desaparecer". Esto se vuelve peor cuanto más profunda sea la red, consiguiendo que las primeras capas apenas se muevan de sus valores iniciales. En el otro extremo están los *exploding gradients*, que ocurre cuando los parámetros son tan grandes que el error crece exponencialmente al multiplicarse en cada capa y termina por llevar a los pesos de las primeras capas a valores no representables como Inf[2] o incluso NaN[3], haciendo que el modelo completo de la red se "rompa" y retorne valores absurdos.

Es por esto que la inicialización de parámetros en una DNN se vuelve un factor crítico y de hacerse de forma eficiente, podría incluso acelerar el proceso de entrenamiento de forma notable. En general, se emplean distribuciones gaussianas de media cero para dicha inicialización ya que esta rompe con la simetría mencionada anteriormente ya sea por su aleatoriedad o por general tanto valores positivos como negativos. Es sin embargo en la elección de la varianza (o desviación estándar) donde se encuentra la clave para evitar tanto el problema de *vanishing gradients* como de *exploding gradients*. Cabe mencionar que incluso inicializando los parámetros con una distribución uniforme, al final del entrenamiento de la DNN, una vez que converge a un resultado con error mínimo, las distribuciones que se pueden observar en los parámetros son gaussianas de media cero y desviación estándar diverso. Ya que el objetivo de la inicialización de los parámetros es que estos se asemejen a sus valores finales luego del entrenamiento (para una convergencia más rápida), se prefiere el uso de distribuciones gaussianas para la generación de sus valores iniciales aleatorios.

En el Apéndice C se realiza un desarrollo de las expresiones de varianza para los pesos de una capa SymSim (y por extensión ChSymSim), siguiendo los lineamientos de [62] y [5], considerando diversos casos de activaciones tipo ReLU. Este análisis se

[2]Valor Infinito
[3]*Not a Number*

realizó con el fin de hallar una distribución inicial óptima para los pesos simétricos de manera que, al entrenar una Red Neuronal con capas SymSim o ChSymSim, las varianzas de las entradas/salidas de cada capa se mantengan estables. Debido a que las entradas/salidas de las capas tienen gran influencia en el cálculo de los gradientes, se vuelve fundamental el reducir su variabilidad mediante una correcta inicialización de los pesos (para el modo *forward*), consiguiendo así gradientes estables y acelerando el "aprendizaje" de la red. En este análisis también se consideró el efecto directo que tienen los pesos sobre el cálculo de los gradientes, hallando otras expresiones para la inicialización de los pesos simétricos para el modo *backward*.

En base a los resultados obtenidos en el Apéndice C, particularmente las Ecs. (C.9)-(C.11), se tiene que para el modo *forward* los pesos de las capas SymSim deberían inicializarse con distribuciones (si se eligen gaussianas) de media 0 y varianza $1/\gamma(N_l)$, donde γ y N_l se describen como:

$$\gamma\left(N_l\right) = \frac{3}{4}\left[1 + a_l^2 - \frac{\left(1-a_l\right)^2}{\pi}\right] N_l + \left[\frac{\left(1-a_l\right)^2}{4\pi} + 18\right]$$

$$N_l = \left(K_H \times K_W + 1\right) \times CH_{in} \ , \tag{3.11}$$

donde l es el índice de la capa, CH_{in} el número de canales de entrada de la capa, K_H y K_W los tamaños de alto y ancho del *kernel* respectivamente. Esta definición con $\gamma = \gamma_{PReLU}$ y parámetro a_l es una generalización y se calculó considerando activaciones del tipo PReLU. De esta expresión se pueden despejar los casos sin activación ($a_l = 1$) y con activaciones ReLU ($a_l = 0$), siendo esta última ligeramente diferente a lo obtenido en Eq. (C.9) pero sólo en el término independiente y que tiene poca influencia con valores grandes de N_l. Por otra parte, para el modo *backward*, las Ecuaciones (C.14) y (C.15) del Apéndice C sugieren una inicialización con varianza $1/\hat{\gamma}(\hat{N}_l)$, donde $\hat{\gamma}$ y $\hat{N}_l$ son:

$$\hat{\gamma}\left(\hat{N}_l\right) = \frac{3\left(1+a_l^2\right)}{4}\hat{N}_l$$

$$\hat{N}_l = K_H \times K_W \times CH_{out} \ , \tag{3.12}$$

repitiendo a K_H y K_W como alto y ancho del *kernel*, l el índice de la capa, a_l el parámetro de la activación PReLU, pero cambiando el número de canales de entrada por el de *features* de salida (CH_{out}).

Tanto γ como $\hat{\gamma}$ son constantes que solo dependen de las dimensiones de la capa SymSim y se calculan solamente al inicio del entrenamiento, en la definición misma de la arquitectura de la DNN, debiendo elegir una de estas dos constantes (o una combinación de las mismas) para la inicialización de pesos, según corresponda para el caso de Red Neuronal y los datos a entrenar. Cabe destacar que, generalmente en capas de filtrado espacial, el número de *features* de salida CH_{out} es mayor (doble o más) que la cantidad de canales de entrada CH_{in}, por lo que al inicializar los pesos simétricos con varianza $1/\hat{\gamma}(\hat{N}_l)$ suele resultar en valores más pequeños que la varianza $1/\gamma(N_l)$.

En base a estas expresiones se evaluaron diferentes inicializaciones para las capas ChSymSim, en comparación con el método de *Kaiming* [5], para dos redes y *datasets*: LeNet-5 con FashionMNIST y ResNet06 con EuroSAT. Para ambas redes se reemplazaron todas las capas convolucionales por capas ChSymSim de SE compartidos por *features* de salida, mientras que las activaciones empleadas fueron del tipo ReLU estándar. Los *bias* en ambos casos fueron inicializados en 0, debido a que estos serán fijados por las capas de BatchNorm, mientras que todos los elementos estructurantes SE fueron inicializados con 1 (como una función simplicial simétrica pura). El primer caso de estudio corresponde a las primeras 10 iteraciones del entrenamiento de una red LeNet-5 (basada en la presentada por [63]) para la clasificación de imágenes de FashionMNIST [64]. Los datos de estos entrenamientos se muestran en la Fig. 3.4, donde se observan que todos los casos de inicialización para el algoritmo ChSymSim, obtenidos a partir del análisis del Apéndice C, son similares o superiores a la inicialización de *Kaiming* [5] para activaciones ReLU, logrando los mejores resultados con la inicialización para el modo *Backward* que hace uso de la constante $\hat{\gamma}(\hat{N}_l)$. En particular, la inicialización *Forward* que utiliza la constante $\gamma(N_l)$ se comporta prácticamente igual a la de *Kaiming*, lo que podría deberse a que el factor dominante N_l es igual en ambos casos. El segundo caso, mostrado en la Fig. 3.5, es el de una red de arquitectura ResNet06 (basada en los modelos propuestos en [18]) que se entrena para clasificar las imágenes satelitales del

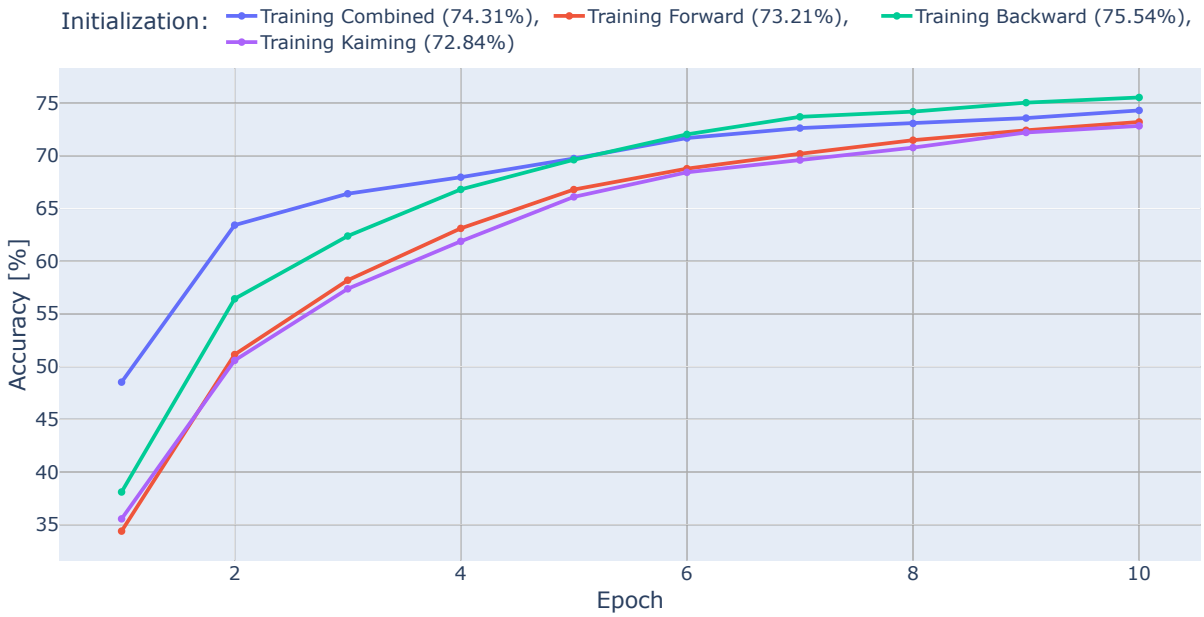


Figura 3.4: Primeras 10 épocas de entrenamiento de ChSymSim LeNet para clasificación del *dataset* FashionMNIST (Training), en base a diferentes inicializaciones: *Combined* (producto $\gamma\hat{\gamma}$), *Forward* (γ), *Backward* ($\hat{\gamma}$), *Kaiming* (normal) [5].

dataset EuroSAT [3, 4]. Este modelo de red se encuentra detallado en el Apéndice D (Tabla D.4), donde se mantienen los filtros convolucionales con *kernel* 1×1, los cuales son exclusivos para *down-sample* de la entrada al bloque básico para su uso en la conexión residual. Si bien los filtros 1×1 pueden ser reemplazados directamente por algoritmos ChSymSim (la función simétrica de una sola entrada es lineal), estas se mantienen convolucionales para facilitar su entrenamiento con los algoritmos estándar. Al igual que lo visto anteriormente (Fig. 3.4), las inicializaciones *Forward* y *Kaiming* resultan en comportamientos similares en cuanto al aprendizaje de la red durante las primeras 10 épocas del entrenamiento, viéndose nuevamente superadas por la inicialización *Backward*. Sin embargo, en este ejemplo es con el método *Combined* o "combinado", que inicializa los pesos simétricos con varianza $1/\gamma\hat{\gamma}$, con el que se obtienen los mejores resultados, muy superiores a los demás métodos de inicialización evaluados. Esto se debe a que generalmente los pesos de DNNs tienden a valores pequeños, especialmente con redes muy "profundas". Es por esto que al inicializar los pesos en valores muy pequeños, es más probable que la red converja rápidamente a la solución, en comparación con valores iniciales grandes que deban ajustarse progresivamente con pequeños gradientes. Por otra parte, mientras más

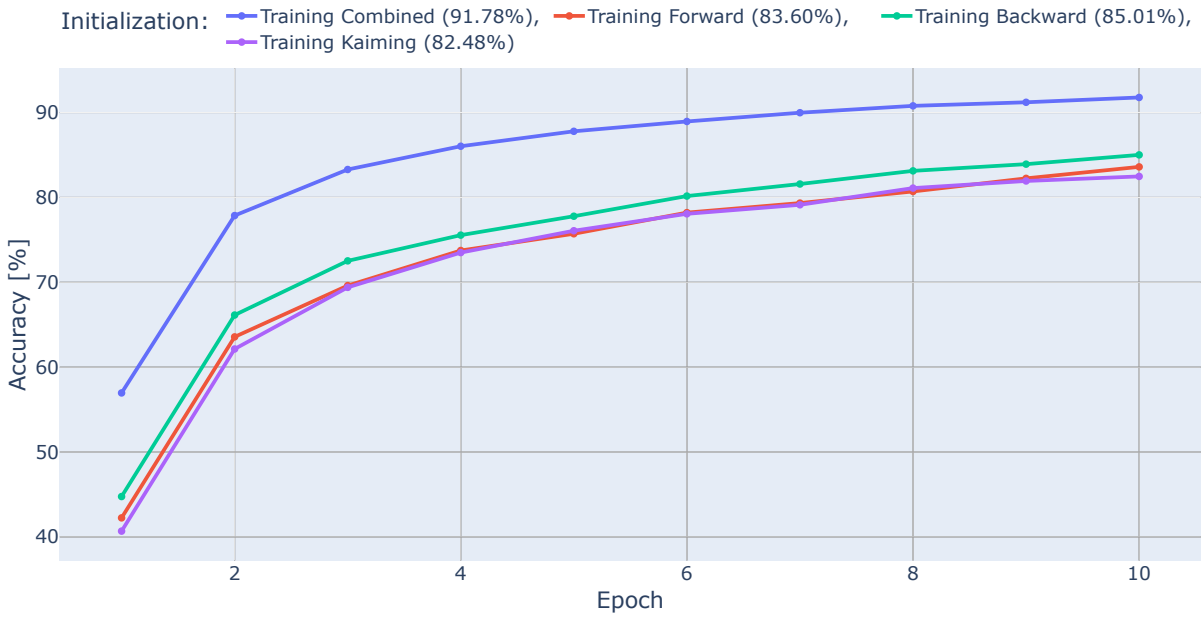


Figura 3.5: Primeras 10 épocas de entrenamiento de ChSymSim ResNet06 (Tabla D.4) para clasificación del *dataset* EuroSAT (Training), en base a diferentes inicializaciones: *Combined* (producto $\gamma\hat{\gamma}$), *Forward* (γ), *Backward* ($\hat{\gamma}$), *Kaiming* (normal) [5].

pequeños son los valores iniciales de los pesos, mayor es el riesgo de que se produzca el fenómeno de *vanishing gradient*. Dado que la red de la Fig. 3.5 es del tipo ResNet, su inicialización más conveniente es *Combined*, ya que este tipo de red presenta bloques llamados *skip-connection*[4] que "renuevan" los gradientes que se propagan hacia atrás en la red e impidiendo así que estos "desaparezcan", tal y como podría ocurrir con redes clásicas como la LeNet utilizada para la Fig. 3.4.

3.4. Redes Neuronales simpliciales simétricas

En esta sección se muestran ejemplos de Redes Neuronales Profundas (DNN) en las cuales se utilizaron capas simpliciales simétricas puras y a canales separados (ChSymSim). El objetivo de esta sección es el de ilustrar casos donde se logró entrenar dichas capas con algoritmos SymSim para la clasificación de diferentes bases de datos, alcanzando a clasificar correctamente más del 90 % de los datos de entrada incluso con modelos pequeños, y que en ciertos casos superan a lo obtenido con modelos convolucionales estándar. Los experimentos que se detallan a continuación

[4]Suma de la entrada con la salida procesada por una o más capas

muestran, además, la evolución de los algoritmos SymSim adaptados a entornos de Redes Neuronales, realizando optimizaciones a dichos algoritmos que simplifiquen una futura implementación en *hardware*, pero sin comprometer el desempeño de los modelos.

3.4.1. RSSNN

Uno de los primeros casos explorados con modelos de Redes Neuronales de capas SymSim es el publicado en [9], bajo el nombre de *Residual Symmetric Simplicial Neural Network* (RSSNN) y que se inspira en las Redes Neuronales Morfológicas de [60] y [59]. En dicho trabajo se entrenan arquitecturas de redes neuronales como la mostrada en la Fig. 3.7, para la clasificación del *dataset* GTSRB, el cual corresponde a 43 tipos diferentes de señales de tránsito alemanas (Fig. 3.6).

Figura 3.6: *Dataset* GTSRB (German Traffic Sign Recognition Benchmark), publicado en [2].

Estas arquitecturas de DNN se componen de dos capas con M filtros simpliciales simétricos (sin AF[5]), seguidos de una conexión residual y dos capas FC de N_1 y N_2 neuronas respectivamente con ReLU, concluyendo con una capa FC de clasificación con 43 neuronas (clases en el *dataset* GTSRB).

En este caso, la imagen de entrada (en escala de grises y tamaño 31×35) se procesa mediante M filtros simpliciales simétricos, con *kernel* de 3×3, *padding*

[5]Función de Activación - *Activation Function*

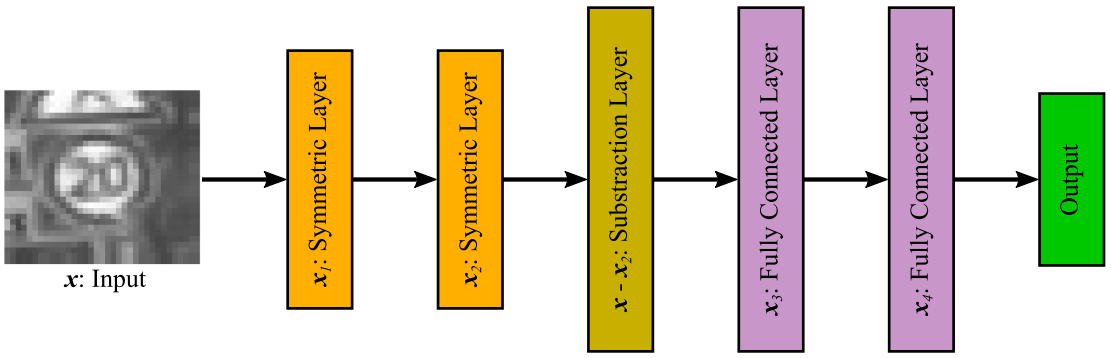


Figura 3.7: Residual Symmetric Simplicial Neural Network (RSSNN).

y *stride* 1, en los que las entradas son multiplicadas por pesos o SE no-binarios. La segunda capa SymSim opera de forma similar a la primera, aplicando un filtro diferente por canal. Cada uno de los M *features* producidos como resultado de los filtros Simétricos son sustraídos de la imagen original, generando así un arreglo de $31 \times 35 \times M$ que se emplea como entrada para la primer capa FC.

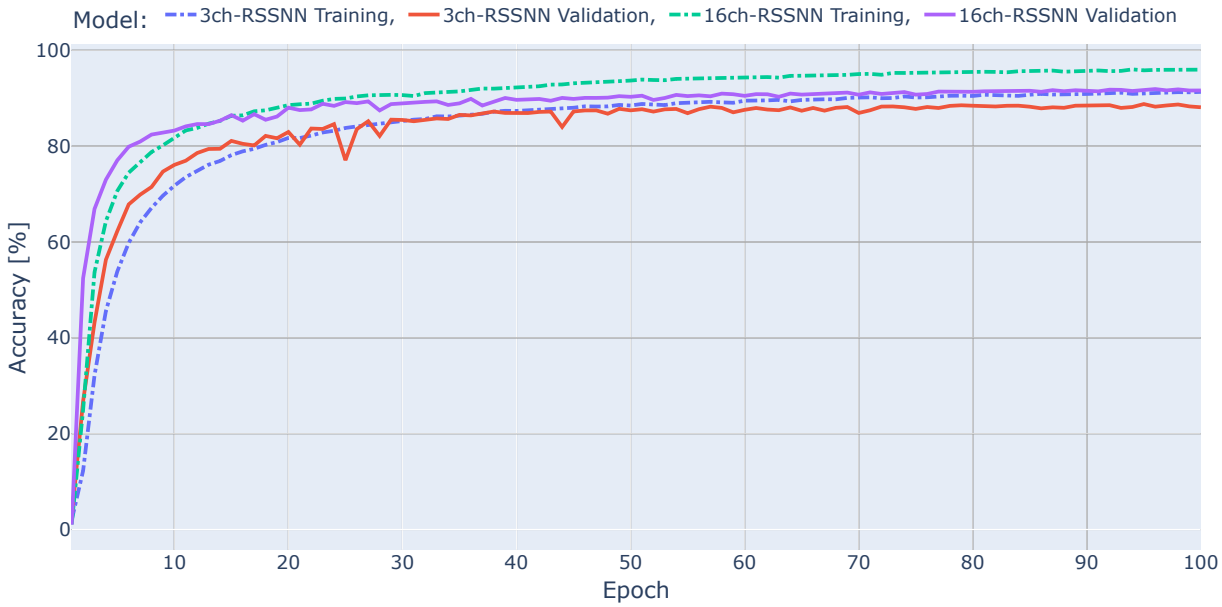


Figura 3.8: Resultado de entrenamientos de redes RSSNN y clasificación del *dataset* GTSRB.

Estas redes poseen pesos con valores iniciales que siguen una distribución normal (media 0,5 y desviación estándar $1/\sqrt{N}$, donde N es el tamaño del *kernel*) y fueron entrenadas durante 100 iteraciones del *dataset* GTSRB, con valores de $M = 3$ y 16, $N_1 = 256$ y $N_2 = 128$. En términos de capas FC, se eligió un modelo de Red Neuronal reducido en comparación a lo propuesto en [60] debido a limitaciones en los

Modelo RSSNN	Desempeño en entrenamiento	Desempeño en validación
$M = 3,\ N_1 = 256,\ N_2 = 128$	91,59 %	88,78 %
$M = 6,\ N_1 = 256,\ N_2 = 128$	92,83 %	89,74 %
$M = 6,\ N_1 = 512,\ N_2 = 256$	96,52 %	90,61 %
$M = 16,\ N_1 = 256,\ N_2 = 128$	96,08 %	91,9 %

Tabla 3.1: Clasificación del *dataset* GTSRB por parte de los modelos RSSNN, obtenidos en [9].

recursos computacionales disponibles. Tanto los pesos/coeficientes simétricos como SE se mantienen en valores entre 0 y 1 por lo que, luego de cada actualización de dichos parámetros, todo valor por encima de 1 es recortado a 1 y todo valor negativo es reemplazado por 0.

Los resultados de la clasificación del *dataset* GTSRB por parte de las redes RSSNN, a lo largo del entrenamiento de las mismas, pueden observarse en la Fig. 3.8, mientras que los valores finales de precisión se muestran en la Tabla 3.1. A pesar de poseer una inicialización de parámetros estándar (sin optimizaciones), de no presentar otras técnicas que aceleren el entrenamiento o la convergencia (como capas de BatchNorm) y de ser entrenadas con el método más sencillo de SGD, estos modelos logran una clasificación de alrededor del 90 %, mejorando a medida que se emplean más filtros simpliciales para la extracción de *features* en las primeras capas. Si bien los resultados de clasificación no alcanzan a lo reportado en [60] (97,48 % con $M = 16$ pero con capas FC considerablemente más grandes), este experimento sienta un precedente en el entrenamiento de capas SymSim en Redes Neuronales Profundas, con posibilidad de mejorar los resultados de clasificación si se emplean modelos con más neuronas y/o técnicas de optimización durante el entrenamiento. Resulta importante destacar que en este ejemplo se utilizaron y entrenaron SE no-binarios, lo que puede dar como resultado el "aprendizaje" de funciones morfológicas puras, convoluciones estándar (la suma en sí misma es una función simétrica), y otros algoritmos aún más complejos. Sin embargo, esta multiplicación de entradas implica complejas implementaciones en hardware que podrían comprometer los beneficios de la función SymSim, por lo que se optó por binarizar los SE para los ejemplos posteriores de Redes Neuronales con funciones Simétricas.

3.4.2. SymSim DMN

Otro ejemplo que ilustra el empleo de funciones SymSim en Redes Neuronales es el mostrado en [41]. En este trabajo se presentó una arquitectura de red neuronal denominada *Deep Morphological Network* (DMN) y representada en la Fig. 3.9, donde la primera capa morfológica es reemplazada por una función simplicial simétrica (denominada SymSim *Pointwise*) con SE binario, que posee 6 filtros de tamaño 11×11, *padding* 5 y *stride* de 4 en ambas direcciones, lo que produce una salida de 6 canales y de un cuarto del tamaño de la entrada original. Dicha capa SymSim *Pointwise* corresponde a un filtrado simplicial simétrico *depth-wise*[6], seguido de una convolución del tipo *point-wise*[7] y sin función de activación. Luego de la capa SymSim se realiza una operación de MaxPool, seguida de capas lineales (FC) con activaciones de tipo *leaky* ReLU o PReLU, con parámetro $1/16$ (de fácil implementación en hardware) para los valores negativos.

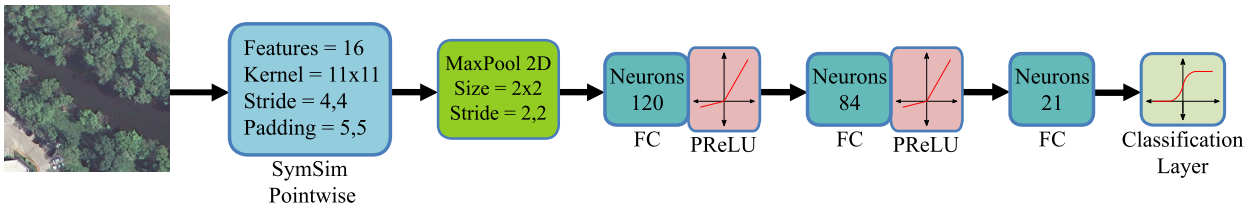


Figura 3.9: Deep Morphological Network (DMN) con primera capa utilizando algoritmo simplicial simétrico y convolución *point-wise*.

Para este caso, la red de la Fig. 3.9 fue entrenada durante 70 épocas para clasificar el *dataset* UC Merced Land Use (Fig. 3.10), que corresponde a imágenes aéreas con 21 clases de escenarios entre los que se encuentran ríos, bosques, playas, edificios, etc. Dicho entrenamiento se realizó mediante el algoritmo de SGD (sin capas de BatchNorm) con un tamaño de *batch* de 16 imágenes, con ratios de aprendizaje (*learn rates*) de 0,5 para parámetros simpliciales y 0,005 para pesos lineales, los cuales son afectados por un factor de decaimiento exponencial (*learn rate decay*) de 0,98 para ambos casos. Los coeficientes simétricos y SEs fueron inicializados con distribución uniforme $U(0, 1)$. Para este fin, todas las imágenes fueron reducidas a 224×224 pixeles, se realizaron *flips* horizontales y verticales aleatorios, se separaron en 4

[6]Mismos coeficientes/pesos para todos los canales de entrada
[7]Filtros de tamaño 1×1

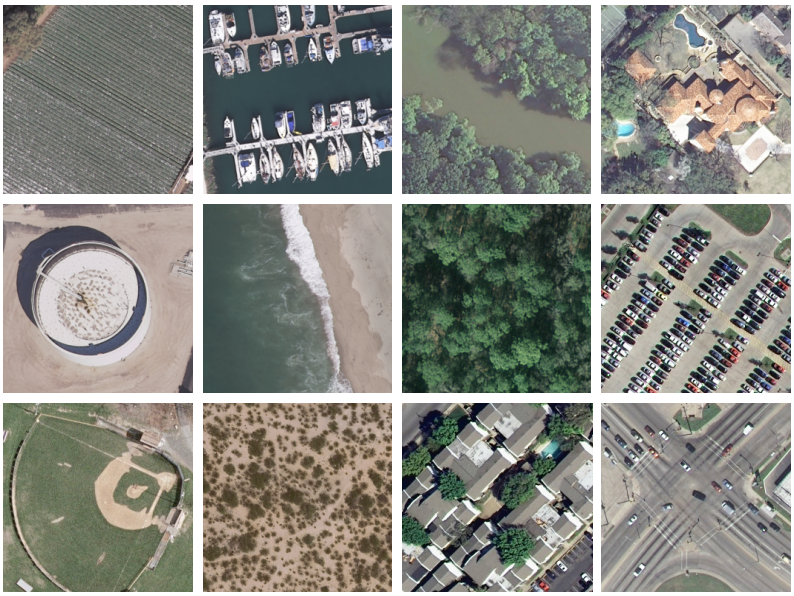


Figura 3.10: UC Merced Land Use *Dataset*, publicado en [6].

grupos de 420 para realizar *k-fold cross-validation*[8] , dando como resultado las curvas de desempeño mostradas en la Fig. 3.11. En dicha figura se puede observar que en base a la selección de los pesos iniciales y un método de entrenamiento relativamente sencillo (sin normalización), el desempeño del modelo en cuanto a la clasificación de los datos de validación resultan en promedio (línea sólida) notablemente inferiores respecto a los datos de entrenamiento. Además, en la Fig. 3.11 se presenta una gran dispersión para los valores de desempeño obtenidos, lo que se puede apreciar en los rangos máximos y mínimos cubiertos por el área en color alrededor de las curvas de desempeño promedio.

Se realizó además un análisis de los efectos de cuantización en la capa SymSim *Pointwise* de la red DMN, cuantizando solamente las entradas y pesos del algoritmo simplicial simétrico. Por un lado, la Fig. 3.12a muestra el error numérico de la salida de la capa SymSim, el cuál es obtenido a partir de la diferencia entre las salidas originales y las cuantizadas, aplicando norma 2 y dividiendo por el número de salidas totales. Este valor de error se halla, a su vez, promediando los errores para todas las imágenes del *dataset*, todos los filtros simpliciales y todos los canales de entrada. Por otra parte, la Fig. 3.12b muestra los resultados de clasificación bajo los efectos de cuantización de entradas y pesos en la capa SymSim, presentando

[8]Separación del *dataset* en grupos o *folds*, dejando uno para validar y el resto para entrenar, repitiendo el proceso cambiando el *fold* utilizado para validación.

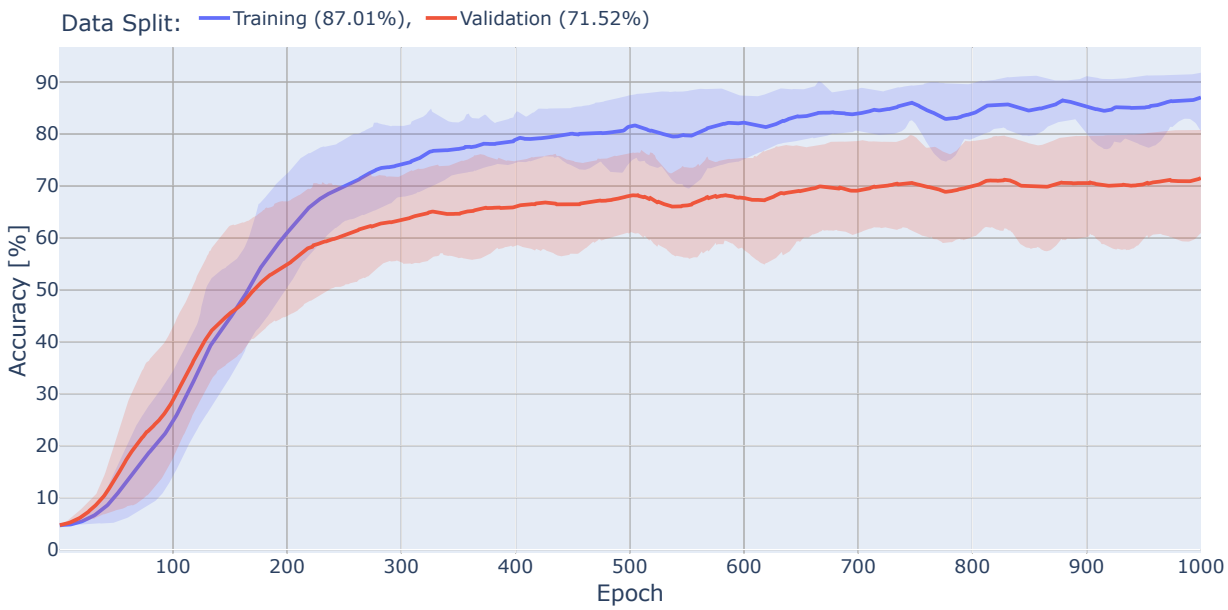


Figura 3.11: Resultados del entrenamiento de red SymSim DMN utilizando *k-fold cross validation* con el *dataset* UC Merced Land Use.

poca degradación en la clasificación de las imágenes de UC Merced Land Use con cuantizaciones entre 4 y 8 bits de las entradas, mientras que la red completa se muestra prácticamente insensible a la cuantización de los pesos Simétricos.

Para explorar aún más el potencial de las capas simpliciales simétricas, se entrenó una versión extendida de la red SymSim DMN (ver Fig. 3.13). Para esta versión, se agregó otra capa con filtros SymSim, convoluciones *point-wise* y MaxPool, con 32 filtros Simétricos de 7×7, *stride* 4 y *padding* 3, lo que reduce notablemente la cantidad de entradas de la primera capa FC ($3 \times 3 \times 32$ en lugar de $28 \times 28 \times 16$ de la red original mostrada en la Fig. 3.9). A su vez, se añadieron activaciones PReLU luego de las capas SymSim *Pointwise* y capas de BatchNorm, y se entrenó de manera similar a la anterior pero por 1500 épocas totales, obteniendo así los resultados de clasificación del *dataset* UC Merced Land Use, que se muestran en la Fig. 3.14. El aumento de complejidad en esta versión, dado por la adición de otra capa SymSim y activaciones PReLU, permite que este modelo obtenga mejores resultados de clasificación, llegando a superar el 90 % de imágenes clasificadas correctamente. Dada la importancia del rango de entrada en las funciones SymSim, los factores de escala del algoritmo de BatchNorm generan entradas/salidas con rangos más homogéneos, lo que estabiliza el aprendizaje (similar para todas las iteraciones de *k-fold cross-*

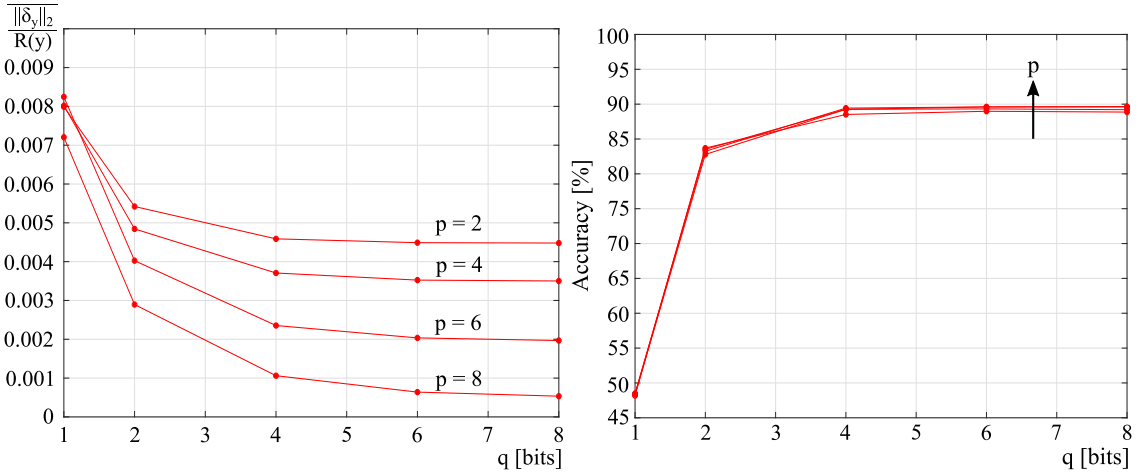


Figura 3.12: a) Error numérico de la salida de la capa simplicial simétrica (SymSim) de la red DMN, en función de la cuantización de entradas y pesos simpliciales; b) Clasificación de la red DMN para el *dataset* UC Merced Land Use (*training set*) al cuantizar las entradas y pesos de la capa SymSim de la red DMN. Para ambos gráficos q representa los bits de cuantización de las entradas y p para los pesos.

validation) y logra una convergencia más rápida. Gracias a esto se observa en la Fig. 3.14 un desempeño en la clasificación de los datos de validación muy cercanos a lo obtenido con los datos de entrenamiento, mientras que la variabilidad en los resultados de cada iteración (*k-fold*) se encuentra notablemente reducida en comparación con lo mostrado en la Fig. 3.11.

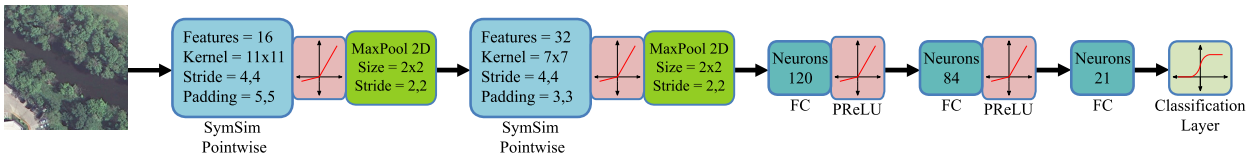


Figura 3.13: Deep Morphological Network (DMN), en su versión extendida, con capas SymSim y convoluciones "*point-wise*".

Este segundo caso de estudio no solo muestra resultados exitosos en la integración de funciones simpliciales simétricas en entornos de DNN, logrando clasificaciones cercanas y hasta superiores a lo reportado hasta el momento de publicación de dichos resultados en [41] (llegando a un 76,7 % en [59]), sino que además ilustra la tolerancia del algoritmo SymSim ante la cuantización de entradas, y especialmente de pesos, así como también los beneficios del uso de capas de BatchNorm para normalizar los rangos de entrada/salida de las capas SymSim y estabilizar/acelerar el proceso de entrenamiento de las mismas. En relación a futuras implementaciones en *hardware*

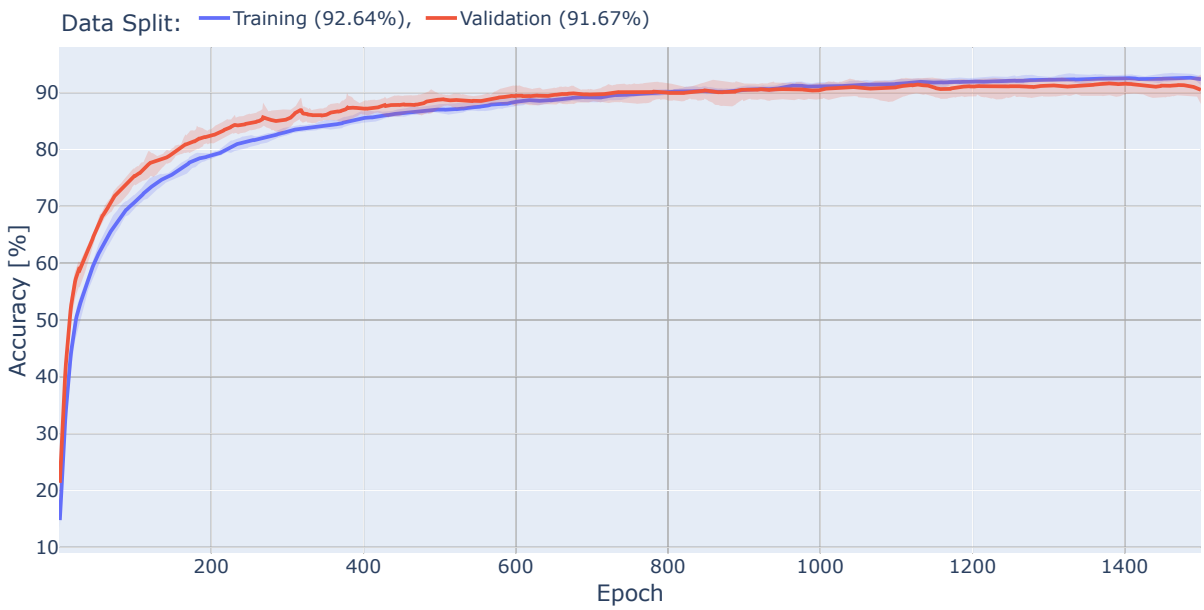


Figura 3.14: Resultados del entrenamiento de red SymSim DMN extendida, utilizando capas de BatchNorm y *k-fold cross validation* con el *dataset* UC Merced Land Use.

de estos modelos, es importante mencionar que el algoritmo de BatchNorm es solamente utilizado durante el entrenamiento de las redes, y que una vez definidos los parámetros del modelo, los escalados producidos por las capas de BatchNorm pueden "fusionarse" con los propios pesos y *bias* de las capas que los preceden, ya sean lineales o simpliciales. Por otra parte, el uso de la secuencia de SymSim *depth-wise* y convolución *point-wise* puede simplificarse al multiplicar los pesos simétricos por canal por su respectivo peso del filtro 1×1 de la convolución *point-wise*, incrementando así el número de parámetros pero procesando la capa entera solamente con el algoritmo ChSymSim.

3.4.3. ChSymSim ResNets

El primer ejemplo de uso del algoritmo simplicial simétrico a canales separados (ChSymSim) se encuentra en los modelos presentados en [8]. Esta publicación se centró en el procesamiento de imágenes satelitales y la observación *on-board*, donde es crucial lograr un equilibrio eficiente entre el rendimiento del modelo, el consumo de energía y los tiempos de cálculo. Dadas las limitaciones en la comunicación con sistemas satelitales, como la latencia y el ancho de banda en los enlaces de *up-link*, se

decidió utilizar versiones reducidas de las arquitecturas ResNet, con menos *features* y capas (ver Apéndice D). Los modelos ResNet son especialmente adecuados para la clasificación de imágenes, ya que sus conexiones residuales permiten entrenar redes más profundas de manera efectiva, mejorando el rendimiento sin aumentar significativamente el tamaño del modelo, lo cual es esencial en escenarios con restricciones de comunicación como el mencionado.

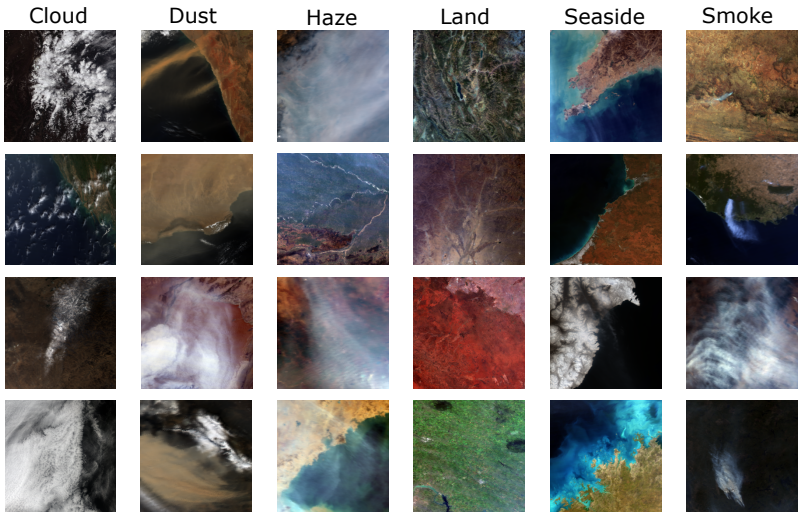


Figura 3.15: *Dataset* USTC SmokeRS, publicado en [7].

En dicho trabajo se eligió como ejemplo al *dataset* USTC SmokeRS [7], ilustrado en la Fig. 3.15, el cual consiste en 6225 imágenes satelitales RGB de tamaño 256×256, con una distancia de muestreo del suelo de 1 km por píxel y 6 categorías (clases) de escenarios en los que se incluyen nubes, polvo, niebla, terreno despejado, costa y humo. Para el caso presentado en [8], esta base de datos fue separada en un 80 % de las imágenes para entrenamiento y el 20 % restantes para validación.

Los modelos ResNet en su versión simplicial simétrica, escogidos para clasificar este *dataset*, se construyeron sustituyendo las capas convolucionales por capas ChSymSim, dejando los filtros 1×1 como lineales, mientras que las activaciones ReLU fueron removidas al no ser estrictamente necesarias (el algoritmo ChSymSim es de naturaleza no lineal). Tanto los modelos simpliciales simétricos como los convolucionales (a modo de comparación) fueron entrenados dentro del entorno de Pytorch, con *Cross-Entropy Loss* (CEL) como función de costo, 32 muestras por *batch*, optimizador Adam (configuración por defecto) y una tasa de aprendizaje (*learn-rate*) inicial de 0,001 con decaimiento exponencial ($\gamma = 0{,}990$). Se utilizaron además

técnicas de *data augmentation* tales como transformaciones con espejados aleatorios horizontales y verticales, así como recortes aleatorios con un tamaño de 224×224, que es el tamaño de entrada de los modelos. Los pesos y coeficientes para las capas ChSymSim se inicializaron utilizando la distribución uniforme de Kaiming [5] con el parámetro $a = \sqrt{5}$ (estándar para capas convolucionales en Pytorch). La motivación para elegir la distribución de Kaiming como inicialización para los coeficientes de ChSymSim es la similitud de las funciones simpliciales con las PReLU, ya que ambas son funciones lineales a tramos. Para el elemento estructurante, se eligió una distribución uniforme e inicializada con valores entre $-0{,}25$ y $1{,}25$ para producir picos en 0 y 1 en la distribución de los parámetros de precisión completa después del recorte en el primer ciclo de avance.

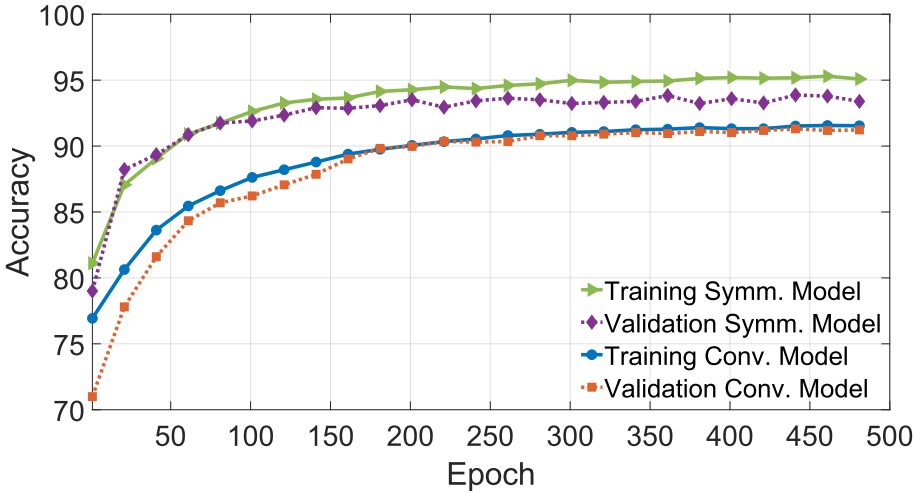
	Error CEL		**Precisión** [%]	
Modelo	**Entrenamiento**	**Validación**	**Entrenamiento**	**Validación**
ResNet18 Conv.	0,004	0,307	99,94	93,51
ResNet04 Conv.	0,238	0,258	91,48	91,75
ResNet04 ChSymSim	0,129	0,208	95,32	93,38
ResNet06 Conv.	0,072	0,222	97,61	93,51
ResNet06 ChSymSim	0,044	0,186	98,39	94,31
ResNet08 Conv.	0,071	0,235	97,57	92,87
ResNet08 ChSymSim	0,067	0,250	97,85	92,31

Tabla 3.2: Resultados de entrenamiento para la clasificación de USTC SmokeRS con los modelos ResNet luego de 500 épocas.

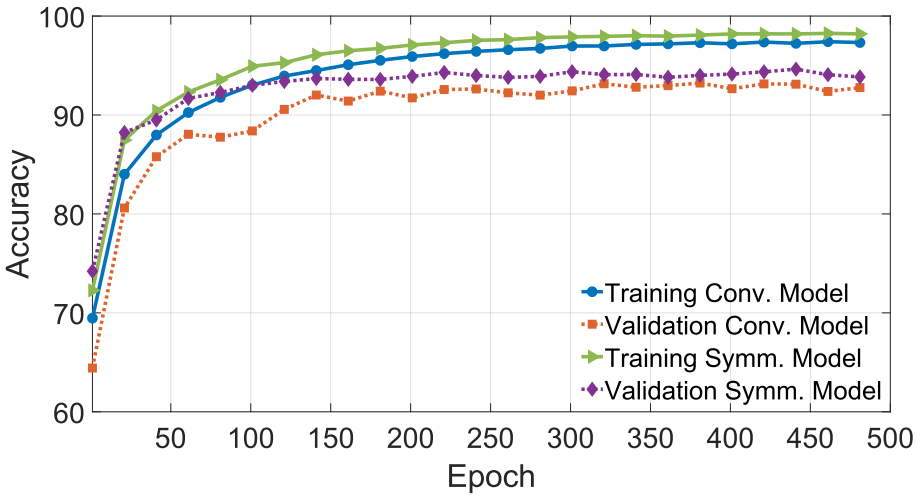
Los resultados del entrenamiento, luego de 500 iteraciones o épocas, se muestran en la Tabla 3.2. Se puede observar en estos resultados que la red ResNet18, a pesar de estar reducida en la cantidad de *features* en comparación con el modelo original de [18], sigue siendo muy complejo para esta tarea, teniendo una gran diferencia entre los valores obtenidos por los *sets* de entrenamiento (*Training*) y validación (*Validation*). En cambio, los modelos ChSymSim más pequeños como ResNet04 y ResNet06 logran un desempeño similar al obtenido con ResNet18 en los datos de validación, e incluso mejor que el promedio de 92,75 % (aunque para un *split* de 60/40 en lugar de 80/20 de este caso) reportado por [7].

Como casos especiales, las Figs. 3.16a y 3.16b muestran la precisión en la clasificación de las imágenes a lo largo del entrenamiento, para los modelos ResNet04 y ResNet06 respectivamente. De la Fig. 3.16a se puede observar que el modelo convo-

lucional obtiene una clasificación inferior a la versión simplicial Simétrica, tanto en entrenamiento como en validación. Por otra parte, ambos modelos ResNet06 tienen clasificaciones más cercanas, pero con la variante simplicial siendo la que alcanza mejores resultados, consiguiendo este modelo el mejor desempeño en validación respecto a los valores mostrados en la Tabla 3.2.



(a)



(b)

Figura 3.16: a) Resultados de clasificación del *dataset* USTC SmokeRS con modelos ResNet04 ChSymSim y convolucional; b) Resultados de clasificación del *dataset* USTC SmokeRS con modelos ResNet06 ChSymSim y convolucional. Resultados presentados en [8]

Como otro caso de estudio con redes simpliciales simétricas para clasificación de imágenes satelitales, se experimentó con el modelo ResNet06 (ya que este dio los mejores resultados en el caso anterior) para la clasificación de la base de datos conocida como EuroSAT [3, 4], ilustrada en la Figura 3.17. Para este ejemplo se

separó al *dataset* en 5 *folds* para entrenamiento y 2 *folds* para validación, aunque no se realizó *k-fold cross-validation*.

Figura 3.17: *Dataset* EuroSAT, publicado en [3, 4].

El entrenamiento se realizó por unas 100 épocas, con una tasa de aprendizaje (*learn-rate*) fijo en 0,001, optimizador NAdam con *weight decay* desacoplado y de valor 0,0005, con CEL como función de costo, obteniendo así los resultados mostrados por la Fig. 3.18. El modelo simplicial simétrico emplea activaciones ReLU luego de las capas ChSymSim, manteniendo aún las capas de BatchNorm antes de la activación, para poder realizar una "fusión" directa luego del entrenamiento. La inicialización de los parámetros en las capas ChSymSim en este experimento corresponden a la variante "combinada" o *Combined* (mostrada en la Sección 3.3) para los pesos simétricos, los *bias* son inicializados todos en 0 y los SE en 1, mientras que las demás capas emplean la inicialización estándar de PyTorch. Es importante destacar que a pesar de todos los métodos de optimización utilizados durante este entrenamiento, que están pensados originalmente para CNN, el modelo convolucional no logra generalizar bien los datos de validación, con una curva repleta de subidas y bajadas en la precisión de clasificación. Por otra parte, el modelo ChSymSim que representa mayor complejidad, dado que los datos de entrenamiento se clasifican con mayor precisión, alcanza una buena generalización ya desde las primeras 10 o 20 iteraciones del entrenamiento.

Estos dos experimentos con redes tipo ResNet muestran el potencial de las funciones simpliciales simétricas, particularmente el algoritmo ChSymSim desarrollado para esta tesis, para su integración en DNN y tareas como la clasificación de imáge-

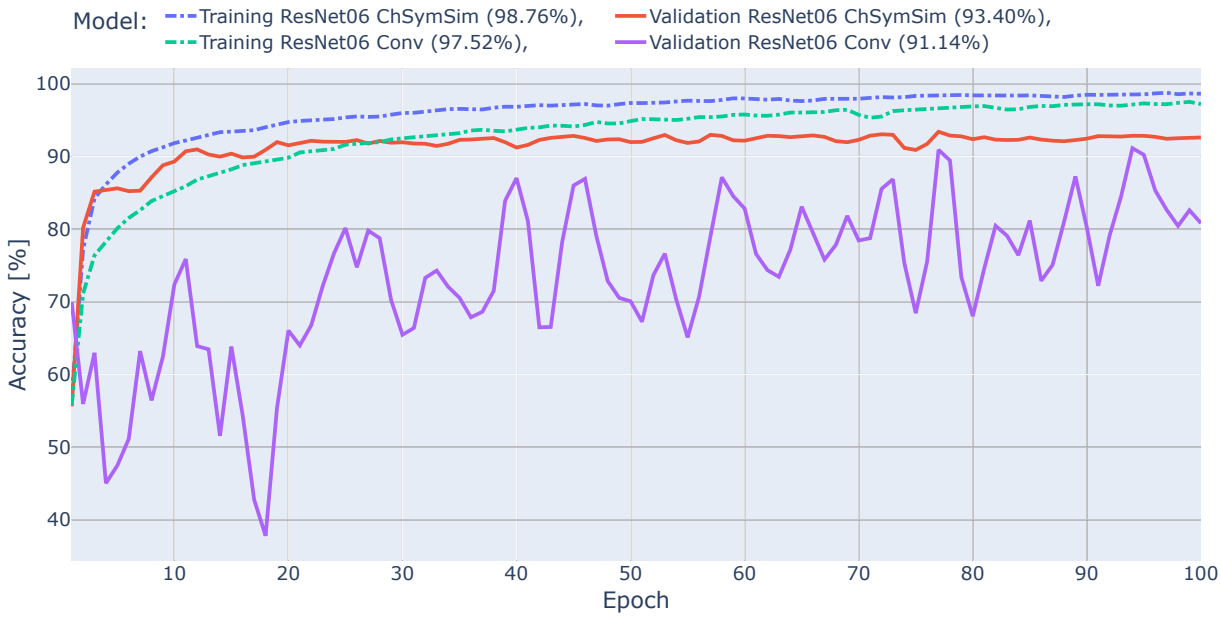


Figura 3.18: Resultados del entrenamiento de modelos ResNet06 ChSymSim (Tabla D.4) y convolucional para clasificación del *dataset* EuroSAT.

nes satelitales. Los resultados que se presentan en esta subsección ilustran dicho potencial al obtener desempeños superiores a los mismos modelos que solamente utilizan capas convolucionales. A pesar de que algunas de las técnicas de optimización utilizadas para este ejemplo fueron pensadas originalmente para redes con capas convolucionales y/o FC, el algoritmo ChSymSim aprovecha dichas estrategias de igual manera, logrando "aprender" y "generalizar" la información ofrecida por los datos de entrenamiento de forma rápida y relativamente estable. En comparación con los ejemplos de uso anteriores (Subsecciones 3.4.1 y 3.4.2), resulta evidente que una adecuada inicialización, así como técnicas de optimización y normalización, permiten a las capas ChSymSim explotar ese potencial, logrando "aprender" en unas pocas iteraciones y generalizando correctamente la información ofrecida por los datos con las que se entrenan. A su vez, se han obtenido resultados satisfactorios con el algoritmo ChSymSim y modelos relativamente pequeños, lo que constituye un gran avance en la dirección de implementaciones de Redes Neuronales para aplicaciones con recursos limitados.

Con el objetivo de comprobar el funcionamiento de estos modelos ChSymSim para implementaciones en *hardware*, se cuantizaron tanto las entradas/salidas como los parámetros de todas las capas de la red ResNet06 y se realizó un re-entrenamiento

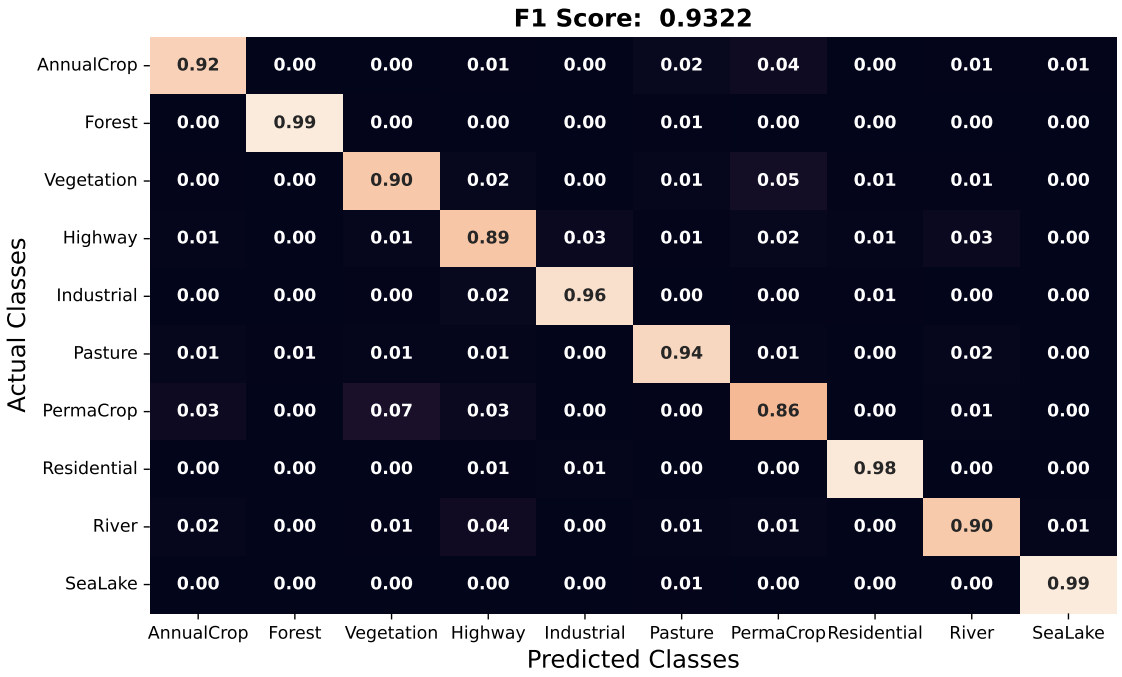


Figura 3.19: Matriz de confusión del modelo ResNet06 cuantizado, clasificando imágenes del *dataset* EuroSAT.

para compensar los errores introducidos. Para esto se desarrollaron librerías en PyTorch con las cuales simular los efectos que se producen al utilizar precisiones (en bits) de entradas y parámetros, alterando los valores en punto flotante que este entorno de programación utiliza como estándar. Modelos de cuantización para las salidas de las capas fueron también introducidos de manera que también se puedan observar los efectos que se presentan al escalar los resultados a la precisión de entrada de la siguiente capa de la red. Debido a la sensibilidad que presenta el algoritmo ChSymSim ante cambios en el rango de las entradas, el primer paso para este experimento consiste en computar (en base las imágenes de EuroSAT) y fijar el rango de entrada de todas las capas de la red (especialmente capas ChSymSim), cuantizando dichas entradas con 8 bits y realizando un re-ajuste (*fine-tuning*) aprovechando las capas de BatchNorm para acelerar el entrenamiento. Dado que los rangos de parámetros y salidas son afectados por los factores de escala introducidos por las capas de BatchNorm, antes de poder cuantizar estas variables es necesario realizar la fusión de estas con las capas ChSymSim o convolucionales/lineales correspondientes. Una vez realizada la cuantización de entradas y posterior fusión de capas BatchNorm, el

modelo ResNet06 utilizado en este caso fue afectado por la cuantización de parámetros (8 bits de pesos y 16 bits de *bias*) y salidas (8 bits al igual que las entradas), computando y fijando los rangos de dichas variables. Finalmente, en la Fig. 3.19 se muestran los resultados de clasificación, luego de un nuevo re-ajuste para considerar la cuantización completa de la red, consiguiendo un desempeño similar al del modelo ResNet06 con capas en punto flotante y precisión estándar (32 bits) presentado en la Fig. 3.18.

3.5. Conclusiones

En este capítulo se profundizó en las funciones simpliciales simétricas y se propuso una versión mejorada (ChSymSim), específicamente para la implementación de Redes Neuronales Profundas en aplicaciones *edge*. Para esto se desarrollaron algoritmos de *backpropagation* con los cuales entrenar/ajustar los parámetros simpliciales, acorde a las necesidades del problema a resolver. Se realizó además una exploración para obtener una inicialización adecuada para los parámetros de la función ChSymSim, de manera de estabilizar los gradientes durante el entrenamiento y acelerar así la convergencia del modelo a la solución deseada. Finalmente, se entrenaron pequeños modelos de redes con capas ChSymSim, para casos como la clasificación de imágenes aéreas/satelitales y señales de tránsito, que a su vez son ejemplos de aplicaciones *edge*. Para el entrenamiento de estos modelos se incorporaron todas las técnicas desarrolladas en este capítulo, junto con estrategias de cuantización, con las que se obtuvieron resultados satisfactorios en cuanto a la clasificación de imágenes, superando a los modelos convencionales en estos casos de uso.

Capítulo 4

Modelado de arquitecturas

Con el objetivo principal de diseñar estructuras de *hardware* eficiente para la ejecución de Redes Neuronales en aplicaciones con recursos limitados (*edge*), se seleccionó para esta tesis la arquitectura de cómputo simplicial, que ya cuenta con algunas implementaciones de operadores eficientes, y se mejoraron para lograr un mejor desempeño en modelos de Redes Neuronales más complejos. Sin embargo, previo a la implementación en ASIC de un procesador dedicado que utilice estos algoritmos y estructuras de cómputo, es necesario el desarrollo de modelos que permitan explorar en una primera instancia diferentes arquitecturas para dicho operador.

Este capítulo presenta un análisis con distintas dimensiones y estructuras para el acelerador ChSymSim, que en base a modelos simplificados de tiempo de cómputo, área y consumo energético, ayuda a tomar decisiones de diseño para optimizar la arquitectura en uno o más de estos aspectos. Aunque útiles para una evaluación inicial del operador, las estimaciones que estos modelos de alto nivel ofrecen son insuficientes para describir completamente las características del acelerador dentro de un sistema más complejo, por lo que se vuelve indispensable realizar tanto síntesis como simulaciones con las que se puedan obtener medidas de área y consumo de potencia más cercanas a una implementación real. Además, para poder contrastar las ventajas ofrecidas por el operador ChSymSim y en qué casos este es más eficiente, resulta necesario compararlo con arquitecturas de cómputo de DNNs estándar, tales como procesadores lineales o convolucionales.

Este capítulo está organizado de la siguiente manera. En la Sección 4.1, se

ofrece una descripción detallada de la estructura seleccionada para el acelerador ChSymSim, junto a sus bloques internos y esquemas de control. La elección de esta arquitectura de cómputo y sus dimensiones se encuentra ligada al análisis con modelos de alto nivel desarrollado en la Sección 4.2, donde se presentan métricas, estimaciones y criterios de optimización que orientan el diseño del operador a una implementación lo más eficiente posible. Finalmente, en la Sección 4.3 se realiza una comparación con un procesador convolucional, de arquitectura similar al acelerador ChSymSim propuesto, tanto con modelos de alto nivel como mediante la síntesis y posteriores simulaciones de ambos operadores, con lo que se obtiene una caracterización más "realista" de los mismos.

4.1. Descripción del operador ChSymSim

Si bien la descripción del algoritmo ChSymSim del Capítulo 3 puede ser suficiente para su implementación en lenguajes de programación y entornos de ML estándar, para el diseño de un acelerador en *hardware* se deben tener en cuenta no sólo la función ChSymSim en sí misma, sino que además se deben considerar operaciones adicionales tales como la transferencia de datos desde la memoria (*on-chip* o externa) al core, y viceversa, así como escalado, redondeo y activación (generalmente ReLU o PReLU). Un ejemplo de esto se muestra en la Fig. 4.1, donde se presenta un diagrama en bloques con la función ChSymSim y operaciones auxiliares, realizando en serie la carga de entradas, pesos y procesamiento por canal, para luego escalar/redondear el resultado de la suma de los canales procesados, aplicar una activación tipo ReLU (opcional) y escribir las salidas en memoria.

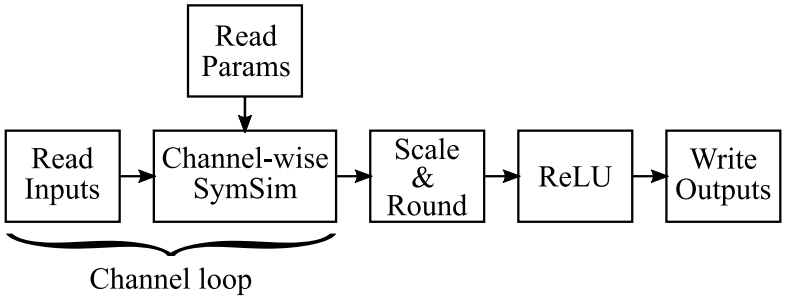


Figura 4.1: Descripción gráfica de la función ChSymSim, incluyendo operaciones auxiliares como carga de datos, escalado, ReLU y escritura de salidas.

La transferencia de datos en sí misma es necesaria para poder iniciar el procesamiento y, según las dimensiones del acelerador y la capa a procesar, esto puede representar un cuello de botella que produzca retardos en los tiempos de cómputo. Por esta razón, el diseño del acelerador y sus dimensiones de entrada/salida deberían tener en cuenta estos procesos de lectura/escritura de datos para poder optimizar el tiempo total de cómputo de la capa o red completa. A su vez, el uso de un registro interno y *latch* que hagan de *buffer* resulta conveniente para poder realizar un *pipeline* entre las lecturas/escrituras y los ciclos de procesamiento, de modo que no se impida o bloquee el cómputo debido a la carga de nuevos datos (pesos y entradas) al acelerador o la extracción de sus resultados. Si no se desean introducir errores durante el cómputo, tales como errores de truncado u *overflow*, entre otros,

es imprescindible contar con operaciones de escalado y redondeo. Estas operaciones permiten reducir la precisión de las salidas para su reutilización en futuros cálculos (las salidas de un bloque intermedio de cómputo es generalmente mucho mayor que la precisión requerida en la entrada de bloques posteriores).

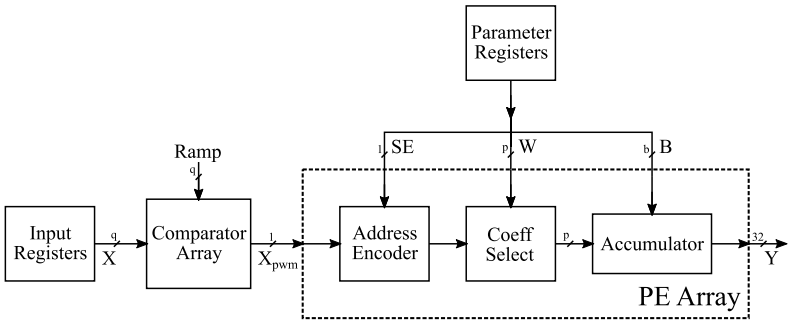
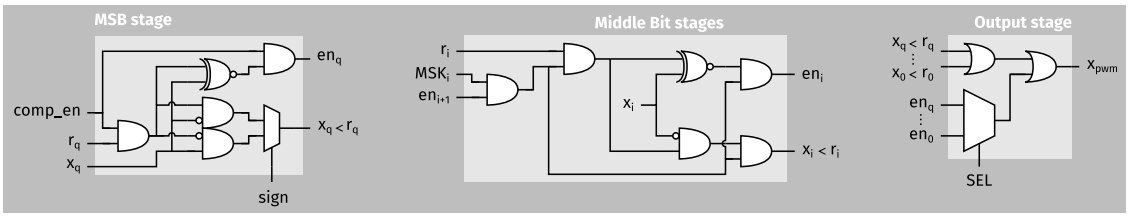


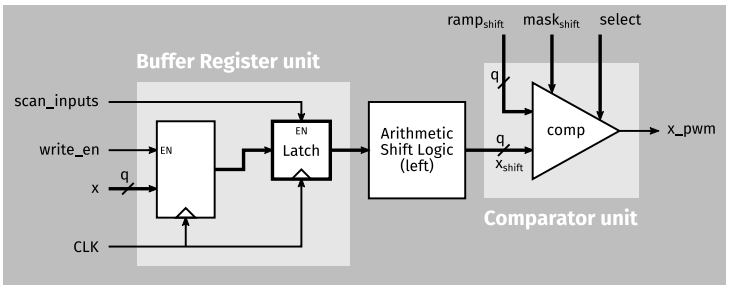
Figura 4.2: Diagrama en bloques (alto nivel) propuesto para el operador ChSymSim.

Para realizar un procesador capaz de computar el algoritmo ChSymSim en base a las implementaciones originales de la función simplicial y simplicial simétrica en *hardware*, presentadas en la Sección 2.3, se requiere de diversos bloques intermedios, que se muestran en la Fig. 4.2. A partir de los datos almacenados en los registros de entrada, un arreglo de comparadores compara dichos datos con una rampa digital (esencialmente un contador), obteniendo las señales tipo PWM que corresponden al valor de las entradas convertido al dominio tiempo. Para el caso del acelerador ChSymSim presentado en este capítulo, se diseñó un generador de rampa que consiste en un contador que puede ser configurado con un valor inicial, un valor final y paso unitario creciente o decreciente ($+1$ o -1), mientras que los comparadores utilizados son del tipo menor igual (entrada $\leqslant$ rampa). La duración de cada ciclo de procesamiento queda definida entonces por la cantidad de ciclos que le toma a la rampa en ir desde el valor inicial hasta el final (según su configuración).

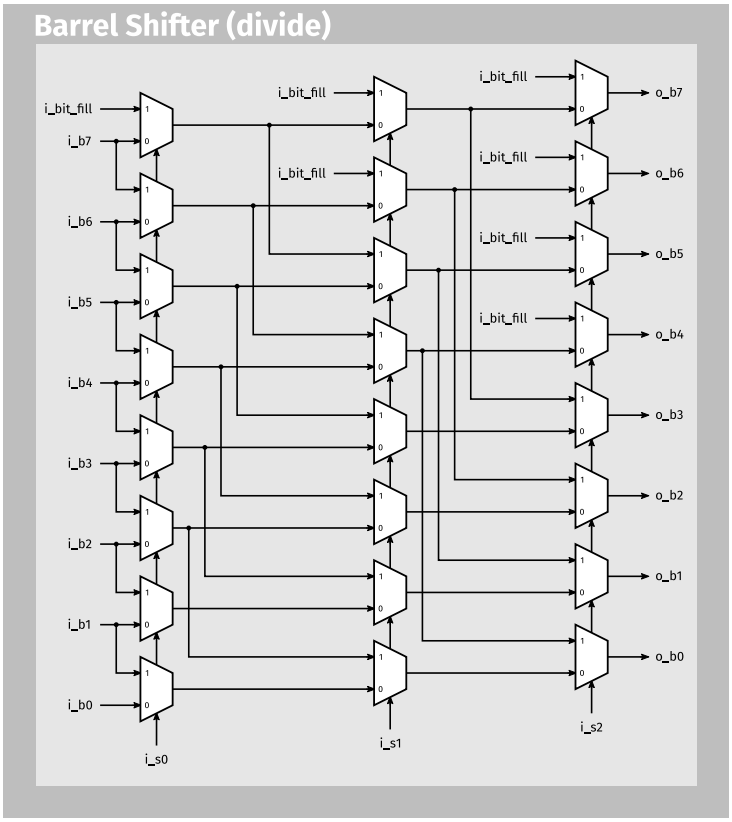
Detalles adicionales de los comparadores pueden observarse en la Fig. 4.3a, donde se tienen diferentes etapas o sub-bloques para computar la función "menor o igual" de cada bit. Este diseño de comparador aprovecha el hecho de que la comparación se realiza con una rampa digital (contador) para optimizar el consumo de potencia: los bits menos significativos de la rampa son los que presentan más cambios a lo largo del ciclo de procesamiento, pero la comparación con los mismos no se habilita hasta que los bits más significativos resuelvan la comparación por igual. De esta forma se



(a)



(b)



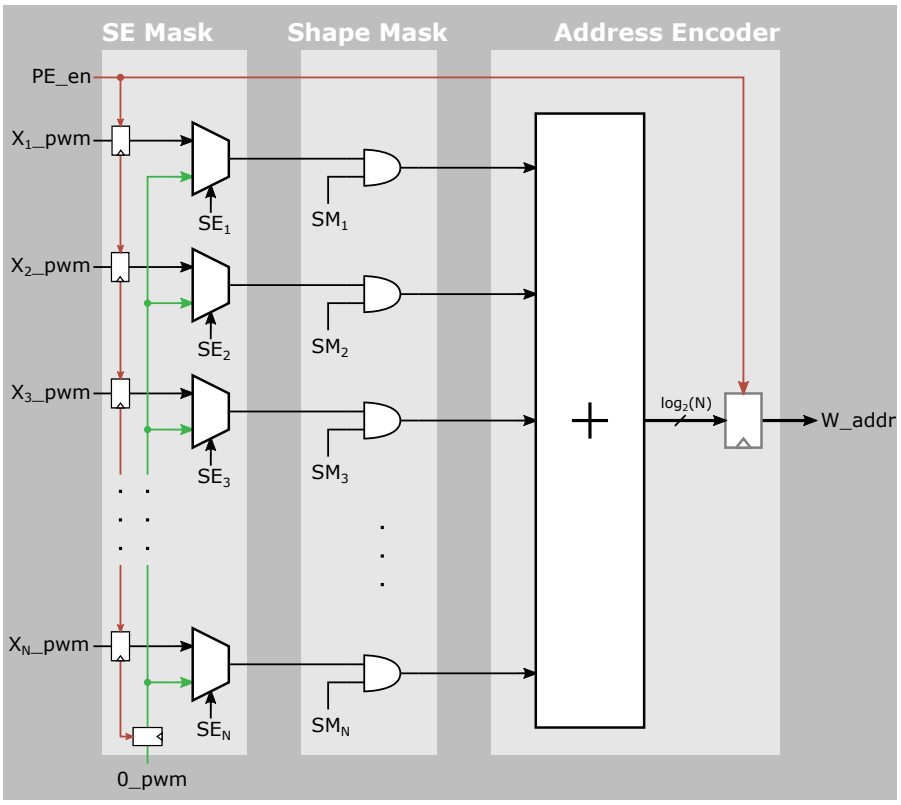
(c)

Figura 4.3: a) Circuito del comparador entre entrada x y rampa r, con salida igual a $x \leqslant r$; b) bloque de entrada y comparación del acelerador ChSymSim; c) *barrel shifter* para implementar un desplazamiento de bits a la izquierda.

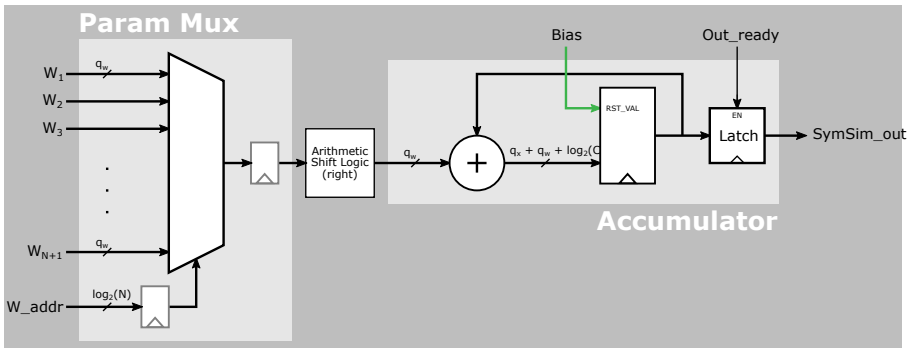
logra que cada nodo del circuito del comparador cambie una sola vez a lo largo del ciclo de procesamiento, ignorando los cambios irrelevantes en la rampa (la entrada es fija) y reduciendo así el consumo energético de forma significativa. El circuito del comparador presentado en la Fig. 4.3a también posee una máscara que deshabilita completamente la comparación de los bits menos significativos (bits no utilizados) cuando la precisión es menor, aunque esta funcionalidad requiere que tanto la entrada como la rampa sean afectadas por un *bit-shift* (a la izquierda) de manera de que se alineen con los sub-bloques de comparación de los bits más significativos. Dada una rampa con paso unitario, el circuito de comparación podría simplificarse considerablemente al considerar solamente la comparación por igual. Sin embargo, se optó por añadir la comparación por menor para dar mayor robustez al sistema en caso de que se introduzcan entradas menores o mayores a los valores extremos de la rampa. La comparación por menor también permite futuras implementaciones con escalado de entradas por medio del paso de la rampa, con incrementos (o decrementos) no unitarios en los valores del contador.

En la Fig. 4.3b se muestra el bloque de *buffer* y comparación para un sólo valor de entrada. Este consiste en un *buffer* para poder realizar el *pipeline* entre carga de datos y procesamiento, una etapa de lógica (*barrel shifter* de la Fig. 4.3c) para realizar el *shift* aritmético antes mencionado, y finalmente el comparador detallado en la Fig. 4.3a. Dado que el objetivo de esta tesis es el de implementar al acelerador ChSymSim en un circuito integrado, el *buffer* se compone de un registro para la cargar del valor de entrada, mientras que lo que mantiene el dato a procesar es un *latch* (mitad de área que un registro) que se vuelve transparente (se habilita) durante un solo ciclo de reloj, en el preciso momento en que todas las entradas están listas para su procesamiento y ya no se deban realizar más transferencias de datos. Este circuito de entrada y comparación mostrado en la Fig. 4.3b es repetido en forma de arreglo para poder almacenar y comparar una porción de la imagen (tensor) de entrada al acelerador.

Las entradas codificadas en tiempo (en forma de PWM) son dirigidas al elemento de procesamiento (PE), en grupos determinados por las dimensiones de dicho PE, donde por cada ciclo de reloj son multiplicadas por el elemento estructurante (SE) binario, enmascaradas según forma y tamaño del *kernel* a computar, y finalmente



(a)



(b)

Figura 4.4: Elemento de procesamiento del acelerador ChSymSim: a) Bloque *Address Encoder* para generación (codificación) de la dirección de pesos; b) Bloque *Mux. Accumulator* para selección de pesos y acumulación de los mismos.

sumadas (lo que da la simetría a la función) para generar la dirección del peso o coeficiente simétrico a seleccionar. Este proceso se obtiene como resultado del bloque denominado *Address Encoder* que se muestra en la Fig. 4.4a. Dicho bloque posee registros para sus entradas y su salida, los cuales son utilizados tanto a modo de *pipeline* para incrementar la frecuencia de reloj a la que el circuito puede operar, como para reducir el consumo de potencia del bloque (mientras los registros se hallen deshabilitados, no se producen cambios en los nodos del circuito y por lo tanto no hay disipación de potencia dinámica). Al ser los SE valores binarios, es decir 0 o 1, el producto por los mismos se resuelve con un simple multiplexor, seleccionando entre la entrada en formato PWM o la señal proveniente de codificar el valor 0 en tiempo. Esta selección entre señales de entrada o 0 en forma PWM se debe al hecho de que este diseño de acelerador permite tanto valores de entrada signados como no signados, teniendo diferentes formas de onda PWM del valor 0 para estos dos casos, y cambiando además según los valores inicial y final de la rampa. Por otra parte, la Máscara de Forma denominada SM[1] es implementada mediante simples compuertas AND que impiden la suma de algunas entradas PWM, dependiendo de la configuración del *kernel*. Esto permite que al computar *kernels* más pequeños, solo se deba cargar en el operador ChSymSim la cantidad mínima requerida de entradas y pesos, sin la necesidad de rellenar con ceros, lo que reduce los tiempos de escritura.

El último bloque que compone al PE, denominado *Mux. Accumulator* y mostrado en la Fig. 4.4b, es el encargado de seleccionar un peso y acumularlo (sumarlo) por cada ciclo de reloj. Dicho peso, determinado por la dirección provista por el *Address Encoder*, es seleccionado por el multiplexor del bloque y afectado por una unidad de *shift* aritmético, que realiza un desplazamiento de bits a la derecha (según la precisión deseada), a la vez que enmascara los bits no utilizados, extendiendo el signo ubicado en el bit más significativo del peso. Al inicio del cómputo del algoritmo ChSymSim, el registro de acumulación en este bloque se inicializa en el valor del *bias* y, una vez que todos los canales han sido procesados y sumados, el resultado del acumulador es copiado en la salida de un *latch*, lo que permite el procesamiento de nuevos datos a la vez que se realiza la lectura de las salidas.

Como parte del trabajo de esta tesis se realizaron algunas variaciones de acele-

[1] *Shape Mask*

rador ChSymSim de un solo PE, con ligeras variaciones con respecto a los circuitos finales presentados en esta sección. En el Capítulo 5 se profundizará sobre estas variaciones y sus correspondientes implementaciones. En capas de filtrado espacial, resulta conveniente utilizar un arreglo de PEs, que al tener algunas entradas en común y compartir todos los parámetros, pueden producir varias salidas en paralelo de forma más eficiente que serializando el cómputo con un único PE. Por esta razón, se realizó un diseño final de acelerador ChSymSim con un arreglo de PEs interconectados, realizando algunas optimizaciones basadas en los resultados del análisis de arquitectura que se introducirá en la Sección 4.2 y lo aprendido al evaluar los prototipos antes mencionados. Este diseño final de acelerador en forma de arreglo se muestra en la Fig. 4.5, el cual corresponde a una implementación del algoritmo ChSymSim que comparte elementos estructurantes entre *features* de salida. El banco de memoria de entrada en esta arquitectura corresponde a un arreglo de *buffers* que se conectan con los comparadores tal y como se mostró en la Fig. 4.3b. Los bancos de memoria de parámetros también son *buffers* implementados de la misma forma que los de entrada, es decir, por medio de un registro seguido de un *latch*, con diferentes secciones de memoria para los SE, coeficientes/pesos simétricos y *bias*. Por otra parte, los elementos de procesamiento se componen de los bloques y circuitos mostrados en las Figs. 4.4a y 4.4b.

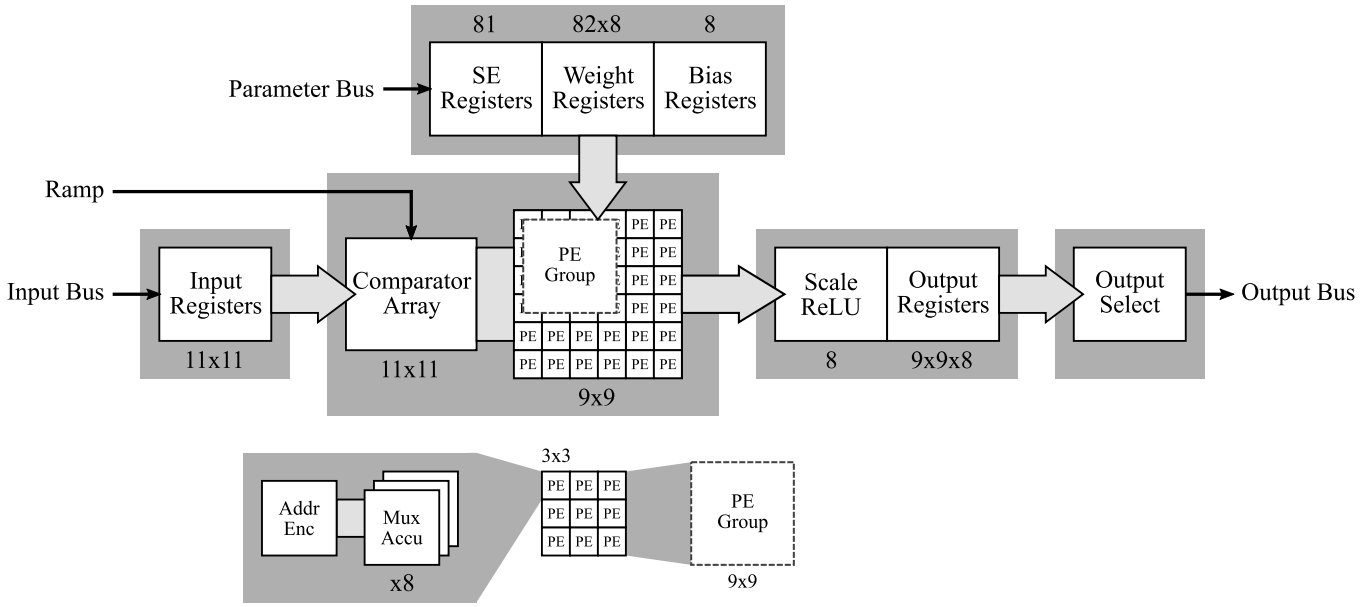


Figura 4.5: Diagrama en bloques del procesador ChSymSim.

Al compartir los mismos SE para todos los *features* de salida, solo se requiere de un bloque *Address Encoder* en cada PE para generar la dirección de pesos, mientras que el único bloque que debe replicarse para producir más salidas (*features*) por PE es el *Mux. Accumulator*. Con el fin de aprovechar el mayor número de entradas ya disponibles (y comparadas) en un ciclo de procesamiento, resulta conveniente ubicar tantos PE en el arreglo como sea necesario para que se puedan computar todas las salidas con *stride* 1 en función del número de entradas totales. Para que esto sea posible, se debe contar con un número de PEs por fila y/o columna de $(H/W)_{out} = (H/W)_{in} + K_{(H/W)} - 1$, donde $K_{(H/W)}$ son las dimensiones del PE relacionadas al tamaño de *kernel* que puede computar, mientras que H y W indican número de filas y columnas, respectivamente. De esta forma, PEs adyacentes comparten $K_H - 1$ filas y/o $K_W - 1$ columnas, según su ubicación relativa en el arreglo. En Redes Neuronales con filtrados espaciales, tales como CNNs, se suelen utilizar con mayor frecuencia capas con *kernel* pequeño (mayormente 3×3), teniendo muy pocos casos con *kernels* más grandes, por lo que pudiera ser conveniente que el tamaño de los PEs se elija teniendo como foco el acelerar estos cómputos con *kernels* de hasta 3×3 entradas.

En base a la separación del PE en dos bloques, uno para generación de una dirección con la que el otro seleccione al peso a acumular, es posible realizar cálculos de filtrados espaciales con *kernels* de mayor tamaño, sin necesidad de incorporar otro arreglo de PEs al sistema. Esto puede lograrse agrupando los PEs que no comparten entradas de la forma que se muestra en la Fig. 4.6. Cada grupo de entradas PWM es procesado por el correspondiente *Address Encoder* (dimensionado para *kernels* pequeños) y luego se suman las direcciones de pesos generadas. La dirección resultante es utilizada por un solo *Mux. Accumulator* (de mayor tamaño que el de los demás PE) para seleccionar el peso a acumular, con la capacidad de seleccionar entre un mayor número de pesos que compongan al *kernel* grande. De esta forma, se pueden computar *kernels* de mayor tamaño a los usuales 3×3 con simplemente agrandar el bloque *Mux. Accumulator* de un sólo PE por grupo, e incluyendo el sumador de direcciones de pesos en este PE extendido.

Los casos más usuales de operaciones de Redes Neuronales con precisiones de punto fijo tienden a utilizar 8 bits para representar tanto entradas como pesos, generalmente valores signados y en complemento a dos. A pesar de esto, existen casos

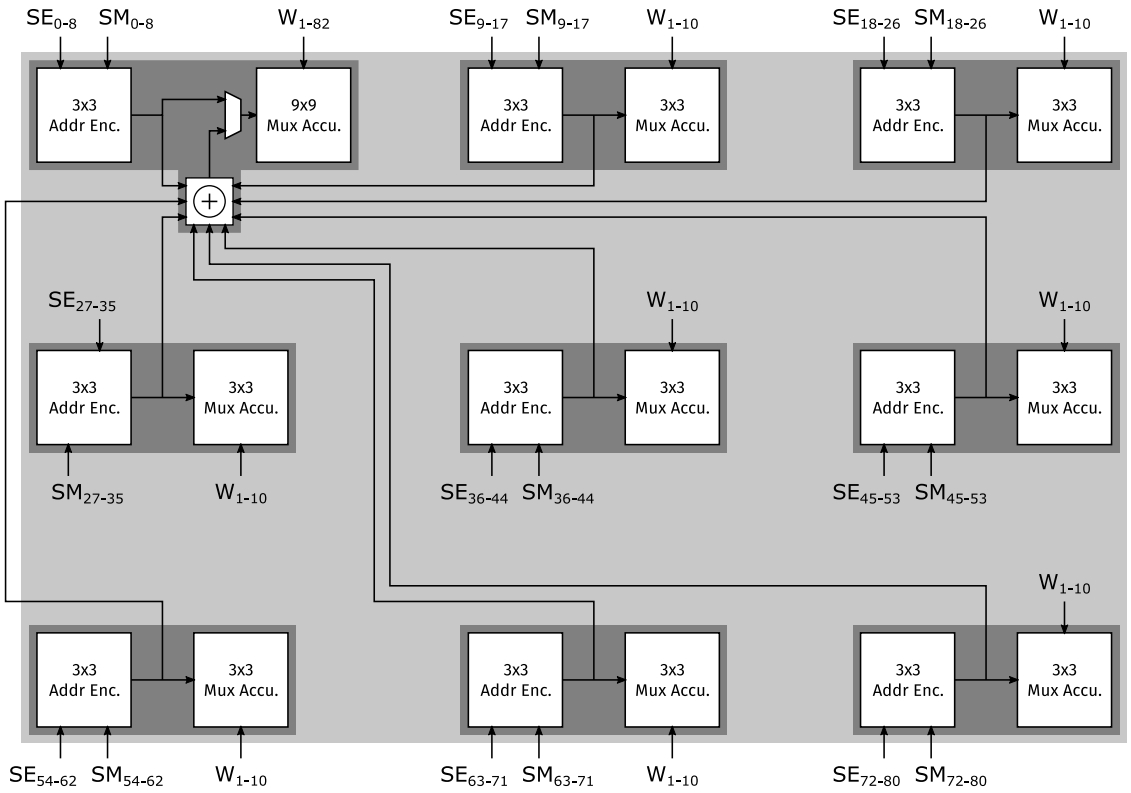


Figura 4.6: Agrupamiento de PEs (que no compartan entradas) para el cómputo de *kernels* grandes en el acelerador ChSymSim.

donde la precisión de entradas y pesos puede ser menor a 8 bits, donde el desempeño de la Red Neuronal prácticamente no se ve perjudicado pese a la reducción de precisión. Para las variantes de la función simplicial simétrica introducidas en esta tesis, en el ejemplo de la Sub-Sección 3.4.2 se mostró que los resultados de clasificación del *dataset* de entrenamiento apenas variaban con la cuantización de entradas entre 8 y 4 bits, además de no presentar cambios notables con la cuantización de parámetros. Para el caso particular de los parámetros de una capa SymSim o ChSymSim, los pesos simétricos requeridos para realizar operaciones morfológicas puras poseen valores 0 o 1, por lo que utilizar 1 bit de precisión para pesos no representa cambio alguno en estas funciones. En implementaciones tradicionales de aceleradores de Redes Neuronales, esta reducción de precisión pudiera no representar grandes cambios en los consumos energéticos, más allá del hecho de que al usar operadores de 8 bits, los bits (más significativos) que no son utilizados quedan fijos por la extensión de signo y se reducen las transiciones en algunos nodos del circuito del operador. Para las funciones simpliciales y simétricas, una reducción en los bits de las entradas tam-

bién representa un beneficio en cuanto a tiempos de cómputo, ya que este depende de la duración de la rampa, que con menor precisión debe recorrer menos valores.

Precisión	N° de canales comprimidos (en un byte)
1	8
2	4
$3-4$	2
$5-8$	1

Tabla 4.1: Compresión de canales posibles para el acelerador ChSymSim.

Además de lo anteriormente mencionado, es posible obtener otra ventaja de la reducción en la precisión de parámetros y entradas: si se realiza una compresión de los datos en la que se disponen más de una entrada, salida o peso en un mismo byte, se puede reducir la cantidad de transferencias totales durante el procesamiento de una capa, reduciendo así el consumo de potencia al ser las operaciones de lectura/escritura de datos generalmente demandantes desde un punto de vista energético. Dada la característica serial del cómputo por canales separados que tiene el acelerador ChSymSim, se decidió comprimir varios canales de entrada en un mismo byte, tanto para entradas como para pesos simétricos. De esta forma, cuando se tiene por ejemplo una precisión de 4 bits de entradas y/o pesos, se requieren datos nuevos una vez por cada dos ciclos de procesamiento (o canales de entrada). Para este caso se utilizan los 4 bits menos significativos para el primer ciclo y los 4 más significativos para el segundo, haciendo uso de los bloques de *shift* aritmético y máscaras para el manejo de la precisión, anteriormente mencionados en esta Sección. Debido a que la precisión por defecto es 8 bits (byte), para que al comprimir los canales estos puedan caber sin errores de truncado, el número de canales comprimidos en un byte se elige en base a la potencia de 2 más cercana, tal y como se muestra en la Tabla 4.1.

La Figura 4.7 ilustra el funcionamiento del acelerador ChSymSim mediante un diagrama de flujo con los estados de su controlador. El cómputo de la capa ChSymSim se comienza mediante la señal *start_proc*, mediante la que el controlador pasa del estado *IDLE* a uno de inicialización en la que se reinician los contadores de canal (dentro del controlador) y los acumuladores del acelerador. En este estado inicial (*START*) se fija también en 0 a la señal *symsim_ready*, indicando que el acelerador se encuentra ocupado procesando. Debido a que se requiere un único

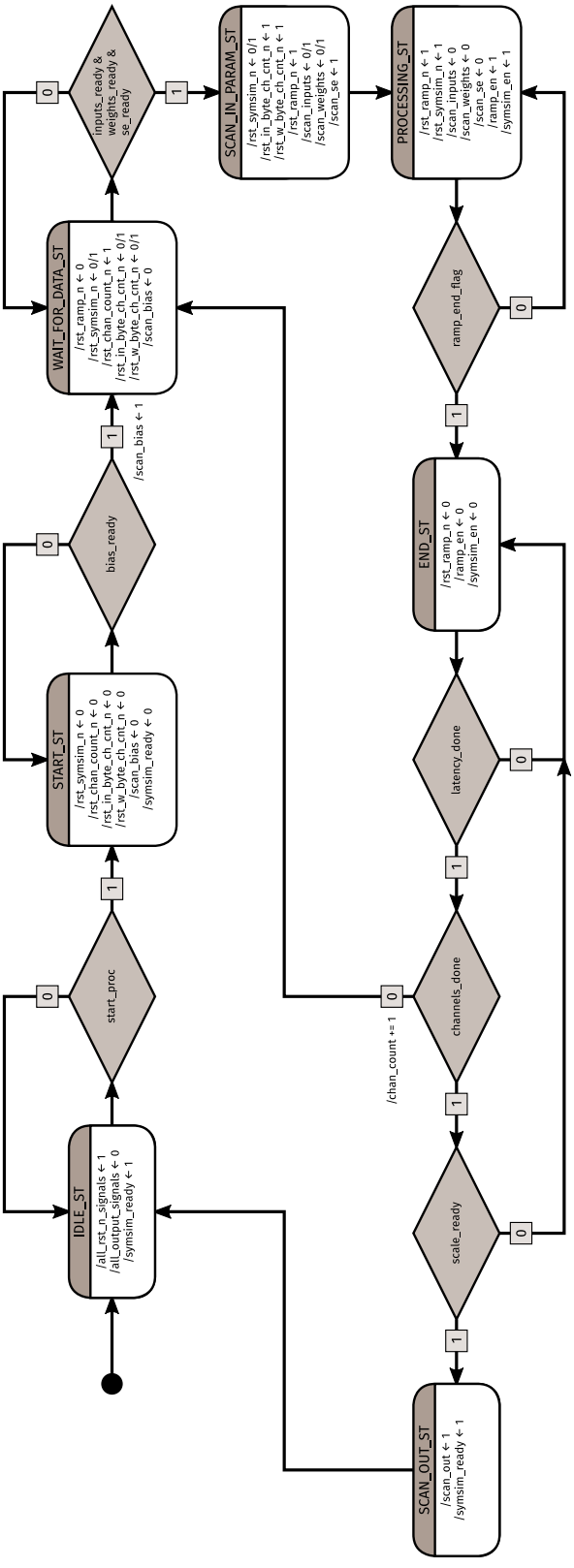


Figura 4.7: Diagrama de flujo para la máquina de estados (controlador) del acelerador ChSymSim.

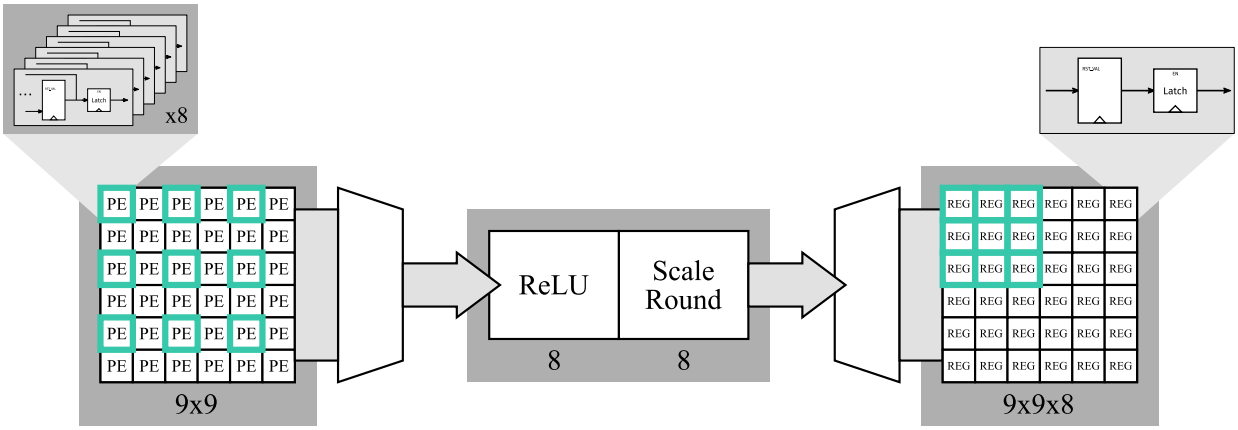
valor de *bias* para todos los canales de entrada y al ser "sumado" como un valor de *reset* del registro de acumulación del PE, el controlador espera a que todos los *bias* para todos los *features* de salida se hallen cargados en los bancos de memoria de parámetros, lo que se indica por medio de la señal *bias_ready*. Además de esperar a tener valores de *bias* válidos, el controlador también debe esperar (en el estado *WAIT_FOR_DATA*) a que el elemento estructurante, el canal de entrada y sus respectivos pesos simétricos, se hallen disponibles en los *buffers*, indicado por las señales "*ready*" correspondientes. Para los casos de menor precisión, en los que dos o más canales son escritos a la vez, se cuentan con contadores adicionales que ignoran las respectivas señales "*ready*" hasta que todos los canales por byte hayan sido procesados. Una vez que los datos a procesar se encuentran disponibles, se pasa al estado *SCAN_IN_PARAM* en el que se copia tanto al canal de entrada como a sus correspondientes pesos y SE, pasando a través del *latch* que posee cada *buffer*, para que inmediatamente después se inicie el cómputo al pasar al estado *PROCESSING*. Al finalizar el contador de la rampa, y por lo tanto la operación SymSim para el presente canal de entrada, se esperan unos ciclos de reloj adicionales (estado *END*) teniendo en cuenta las latencias del operador debido a sus registros de *pipeline*. Finalmente, si todos los canales fueron procesados y acumulados, se procede a su escalado/redondeo y posterior extracción de resultados, mientras que se vuelve al estado *WAIT_FOR_DATA* y se repite el procedimiento en caso de haber aún canales por procesar.

Sabiendo que los resultados provistos por la función ChSymSim poseen una precisión significativamente superior a las entradas y pesos, de modo de evitar errores por *overflow* o *underflow*, las salidas de los acumuladores en los PE deben primero ser re-escaladas a los 8 bits (o menos) originales para poder ser utilizadas en otra capa luego de ser extraídas del acelerador. Para esto, se utiliza el sub-módulo de escalado y redondeo, que también satura las salidas según la precisión deseada, mientras que el sub-módulo de ReLU implementa dicha función de activación con sus variantes *leaky* y saturada. Las unidades de escalado, redondeo y ReLU operan seleccionando una por una las salidas de los PEs del arreglo, escalando todos los *features* a la vez, y ubicando los resultados en *buffers* de salida (Fig. 4.8a). Esto se diseñó con el fin de incorporar el *stride* dentro del paso de selección de salidas de los PE, de manera que

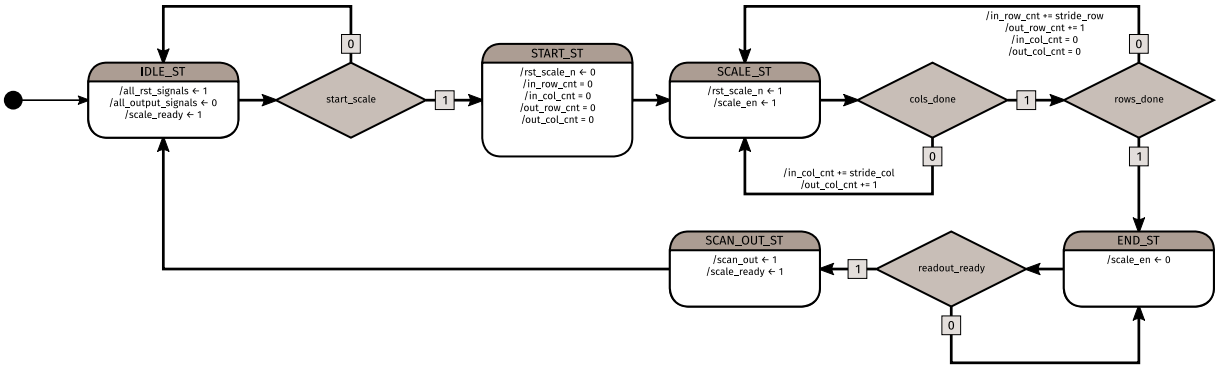
se ignoren las posiciones no relevantes cuando el *stride* es mayor que 1, mientras que las salidas escaladas se ubican de forma consecutiva (sin dejar huecos), permitiendo lecturas más sencillas y que puedan extraer más datos con una sola transferencia. Como una optimización adicional, este acelerado permite deshabilitar los PEs no utilizados cuando el *stride* es mayor a 1, por medio de las señales de habilitación de los registros internos en los bloques *Address Encoder* y *Mux. Accumulator* (Figs. 4.4a y 4.4b respectivamente), para que su consumo se reduzca considerablemente en estos modos de operación, volviéndose casi despreciable. El diagrama de estados del controlador para el arreglo de unidades de escalado, redondeo y ReLU se muestra en la Fig. 4.8b. De manera similar al esquema de control del operador ChSymSim, se pasa de estado *IDLE* a *START* mediante una señal externa, y se reinician los registros de los contadores internos, y donde luego de recorrer las filas y columnas correspondientes de los PEs (según la configuración de *stride*), se espera en un estado *END* a las latencias del sistema antes de escanear los resultados en los *latch* de salida. La información detallada sobre los bloques que componen a las unidades de escalado, redondeo y ReLU, y su funcionamiento, se incluye en el Apéndice E.

Este acelerador dispone, además, de dos opciones para realizar *padding*, tanto a la entrada como en la salida del operador. Para el caso de la entrada, es posible inicializar los registros correspondientes con el valor de *padding* deseado, para luego escribir las entradas originales en las direcciones adecuadas, debiendo agregar complejidad en las configuraciones de escritura de datos. Para el caso de *padding* a la salida, se cuenta con la posibilidad de desplazar (*shift*) los resultados de la función ChSymSim (luego del escalado, redondeo y activación), para incorporar los valores de *padding* deseados al bus de salida, simplificando así la lectura de datos del acelerador pero a costa de más memoria para almacenar los resultados. La incorporación de valores de *padding* en el bus de salida se realiza dentro del selector de salidas, que consiste en un multiplexor que selecciona los resultados del escalado, redondeo y ReLU según la dirección que se desee leer.

Para el el diseño de los selectores/multiplexores se realizó un análisis, incluido en el Apéndice F, basado en uno de los patrones de lectura de datos más usual dado por un incremento constante y unitario (+1) en la dirección de los datos seleccionados. Dicho análisis presenta dos técnicas de optimización: una que minimiza el



(a)



(b)

Figura 4.8: Escalado, redondeo y ReLU: a) arreglo de unidades de cómputo; b) diagrama de estados del controlador.

consumo de potencia simplemente con reordenar las entradas y bits de dirección del multiplexor; y otra que se enfoca en evitar replicar bloques de *hardware*, al separar el multiplexor en selectores más pequeños para cada *feature* de salida (máscara de compuertas AND y selección con celdas OR) y un *one-hot encoder* para generar la dirección de selección (uno sólo para todos los *features*). De esta forma, el acelerador ChSymSim presentado en esta Sección utiliza el modelo de selector *One-Hot* para los bloques de *Mux. Accumulator*, reduciendo así tanto área como parte del consumo energético. Para el resto de selectores, tales como los multiplexores de salida del acelerador, que conectan con sus respectivos *buses* a los resultados de los PEs, salidas escaladas, entradas, parámetros y datos de configuración, se empleó el modelo de multiplexor estándar, optimizado con entradas reordenadas.

La arquitectura de acelerador ChSymSim propuesta en la Fig. 4.5 presenta un arreglo de 9×9 PEs, con 8 *features* por PE, que se interconectan en grupos de

3×3 para el cálculo con *kernel* que supere las dimensiones del PE. Debido a que el tamaño de *kernel* máximo que un PE puede procesar es de 3×3 (elegido por ser el más usual en CNNs), el bloque de entrada con *buffers* y comparadores resulta en un tamaño de 11×11 elementos. Los bancos de memoria para SE, pesos simétricos y *bias* dependen exclusivamente del número de *features* y del tamaño máximo de *kernel* que se puede computar utilizando los grupos de PEs, siendo 9×9 para este caso. Estas dimensiones fueron seleccionadas en base a las restricciones de área para su integración en un circuito integrado (que se describirá en el Cap. 5), priorizando la capacidad de generar 8 *features* de salida en paralelo, para así producir todas las salidas cuando se usa la compresión de canales con 1 bit de precisión. Con un número de *features* de salida igual a 8 y con *kernel* de 3×3 en cada PE, las dimensiones de los grupos de PEs que se pudo implementar en base a las limitaciones de área para su fabricación en ASIC, fue de 3×3 PEs por grupo. No obstante, el código de RTL generado en esta tesis permite la elección de las dimensiones del acelerador mediante parámetros especiales, tales como tamaño de *kernel* en los PE, cantidad de PEs por grupo, número de *features* de salida en paralelo, máxima cantidad de canales de entrada, entre otros. También es posible modificar los parámetros del código RTL para expandir el arreglo de elementos de procesamiento sin que estos PEs adicionales se encuentren agrupados, de modo de priorizar la paralelización de los cómputos con *kernels* pequeños.

4.2. Modelos de alto nivel

El desarrollo de modelos de alto nivel es de gran importancia a la hora de explorar diversas arquitecturas de procesamiento y así poder seleccionar la que mejor se adapte a los condiciones, requisitos y/o limitaciones del caso de uso en cuestión. Si bien es posible generar código RTL con las distintas arquitecturas a probar, pudiendo estimar tiempos de cómputo (mediante simulaciones puras de funcionamiento lógico o simulaciones *behavioral*), como también área y consumo, mediante la síntesis con celdas estándar, esto puede requerir horas o incluso días. Es por esto que al producir modelos de "primer orden", aún cuando estos sean sencillos y se basen en el operador aislado, incluso ignorando ciertos efectos de las interacciones con el resto del sistema,

puede dar una primera idea para evaluar las arquitecturas más prometedoras (que luego puede ponerse a prueba en entornos más realistas como simulaciones post-síntesis). En esta Sección se muestran algunos de estos modelos de alto nivel con los que se exploraron arquitecturas para el acelerador ChSymSim, haciendo foco en optimizar el diseño para prevenir la repetición de operaciones y evitar replicar bloques de *hardware* tanto como sea posible, concluyendo finalmente en el diseño propuesto en la Sección 4.1.

4.2.1. Eficiencia de no-solapamiento

Cuando se desea paralelizar varias operaciones y así acelerar los cómputos de las mismas, aprovechando además que algunos datos de entrada son comunes a más de una operación, como en el caso de un filtrado espacial o convolución, generalmente se recurre a un diseño de acelerador en forma de arreglo, en el cual una porción de la entrada (vector, matriz o tensor) es procesada por varias unidades a la vez. Sin embargo, en ocasiones en que la entrada a procesar no cabe completamente en el arreglo, algunas porciones de esta entrada deben procesarse en siguientes iteraciones y, por lo tanto, los tiempos de carga de datos y cómputo deben repetirse. Esto puede crear regiones de superposición o solapamiento entre dos pasos consecutivos de filtrado espacial; esto depende de cuán grande sea la cantidad de entradas que el acelerador puede procesar en comparación con el tamaño completo del tensor de entrada. El tamaño más pequeño posible para la cantidad de entradas a procesar es el tamaño del *kernel*, lo que significa que solo se calcula una salida para cada paso de procesamiento, mientras que el tamaño más grande es el tamaño completo del tensor de entrada, donde todas las salidas de uno o más *features* se calculan en paralelo (en un solo paso). Otros factores que influyen en la superposición son el tamaño del *kernel*, que determina cuántas entradas se requieren para cada salida, y el *stride*, que representa cuántas entradas se saltean por paso de procesamiento.

La Figura 4.9 ilustra este problema, mostrando el caso de un filtrado en dos dimensiones de una imagen (o tensor 2D) de entrada con dimensiones $Im_H \times Im_W$, y destacando las regiones donde se produce el solapamiento de cada bloque de entradas (cada uno de tamaño $CA_H \times CA_W$). Las regiones de superposición, es decir los datos

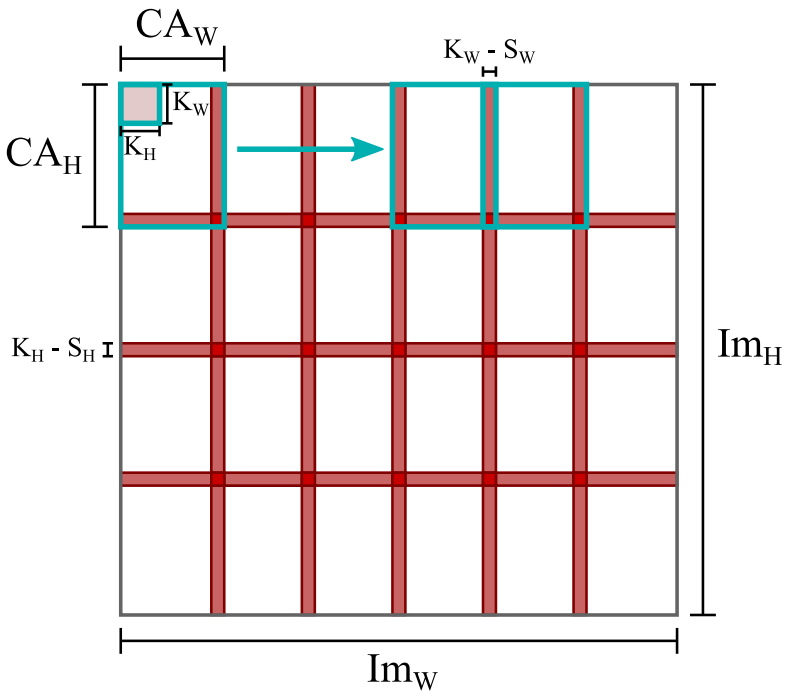


Figura 4.9: Regiones de solapamiento originadas por las entradas compartidas entre series de procesamiento consecutivos, a partir de una imagen o tensor de entrada de tamaño $Im_H \times Im_W$, tomando porciones de $CA_H \times CA_W$ en cada ciclo de procesamiento, con *kernel* de tamaño $k_H \times k_W$ y *stride* (S_H, S_W).

que deben usarse para pasos consecutivos de procesamiento, tienen un ancho y alto que está dado por la diferencia entre las dimensiones del *kernel* ($k_H \times k_W$) y el *stride* (S_H y S_W), mientras que la cantidad de regiones (filas y columnas) se da por la relación entre las dimensiones totales de la entrada en comparación con la porción que se toma por ciclo de procesamiento (denominado CA por corresponder al arreglo de comparadores en el operador ChSymSim). De esta forma, la cantidad de entradas en dichas regiones de solapamiento se pueden calcular como:

$$O_H = (K_H - S_H) \times \left(\left\lceil \frac{Im_H}{CA_H} \right\rceil - 1 \right) \times Im_W \tag{4.1}$$

$$O_W = (K_W - S_W) \times \left(\left\lceil \frac{Im_W}{CA_W} \right\rceil - 1 \right) \times Im_H \tag{4.2}$$

$$DO = (K_H - S_H) \times (K_W - S_W) \times \left(\left\lceil \frac{Im_H}{CA_H} \right\rceil - 1 \right) \times \left(\left\lceil \frac{Im_W}{CA_W} \right\rceil - 1 \right) , \tag{4.3}$$

donde O_H y O_W son las filas y columnas totales de datos repetidos en ciclos de procesamiento consecutivos, mientras que DO representa la cantidad de entradas que se usan en múltiples ciclos de procesamiento (intersección de 4 bloques de cómpu-

to), y que corresponde a la intersección de las regiones de solapamiento filas con columnas. Con los cálculos de las Ecs. (4.1), (4.2) y (4.3) se puede establecer una métrica de eficiencia que promueva la reducción de estas regiones de superposición o solapamiento de datos por ciclo de procesamiento, lo que resulta en:

$$\eta_{n.o.} = \frac{Im_H \times Im_W - (O_H + O_W - DO)}{Im_H \times Im_W} \ , \tag{4.4}$$

denominada "eficiencia de no-solapamiento" (en inglés *non-overlap efficiency*) y presentada en [42]. Al tratarse este ejemplo de un filtrado espacial en 2D, el número de canales en un tensor de entrada incrementa de igual forma la cantidad de datos que presentan solapamiento como los que no, por lo que no se incluye en la Ec. (4.4).

$CA_H \backslash CA_W$	8	16	32	64
8	0,1469	0,2651	0,3242	0,3579
16	0,2651	0,4784	0,5850	0,6459
32	0,3242	0,5850	0,7154	0,7899
64	0,3579	0,6459	0,7899	0,8722

Tabla 4.2: Eficiencia de No-Solapamiento para una capa con entrada de tamaño $3 \times 224 \times 224$ (RGB), kernel de 7×7, *stride* 2 y *padding* 3.

En base a esta métrica de eficiencia, se realizaron dos ejemplos con capas del modelo ResNet18 (descrito en el Apéndice D), asumiendo que el acelerador posee un arreglo de PEs con tamaño de *kernel* igual al que se desea computar en cada caso. El primero de estos ejemplos corresponde a la primera capa de la red (*Layer* 1), la cual realiza un filtrado espacial con *kernel* 7×7, *stride* 2 y *padding* 3 (para solo reducir el tamaño debido al *stride*), procesando una imagen RGB de tamaño 224×224. En la Tabla 4.2 se puede observar la eficiencia de no-solapamiento de acuerdo a la cantidad de entradas (filas CA_H y columnas CA_W) que se pueden utilizar en un solo ciclo de procesamiento. En este caso, al ser el tamaño de *kernel* tan grande en relación al *stride*, los valores de eficiencia resultan relativamente bajos, con un mínimo de aproximadamente 15 % en $CA_H = CA_W = 8$, que es prácticamente el tamaño del *kernel* a computar, por lo que la paralelización de cómputos es casi nula. Para poder obtener una mejor eficiencia con entradas y *kernels* de gran tamaño, se requiere entonces expandir el arreglo de PEs de forma considerable.

En el segundo ejemplo, y de forma similar al anterior, se computó la eficiencia

$CA_H \backslash CA_W$	8	16	32	64
8	0,7695	0,8310	0,8618	0,8772
16	0,8310	0,8975	0,9307	0,9474
32	0,8618	0,9307	0,9652	0,9824
64	0,8772	0,9474	0,9824	1,0000

Tabla 4.3: Eficiencia de No-Solapamiento para una capa con entrada de tamaño $16 \times 56 \times 56$, kernel de 3×3, *stride* 2 y *padding* 1.

de no-solapamiento dado por el cómputo de la capa *Layer* 3_1 del modelo ResNet18 mostrado en la Tabla D.1, el cual procesa una entrada de 56×56 (16 canales) con un *kernel* de 3×3, *stride* 2 y *padding* 1. Los resultados de dicho análisis pueden apreciarse en la Tabla 4.3. Para este segundo caso, al tener tanto entrada como *kernel* de menor tamaño, además de un paso o *stride* cercano a las dimensiones del *kernel*, se tienen eficiencias notablemente superiores. En particular, con $CA_H = CA_W = 64$ se puede procesar la entrada completa, por lo que la eficiencia es directamente la unidad, dado que no puede existir solapamiento entre ciclos de procesamiento si se necesita de un solo ciclo.

Independientemente de qué tamaño de *kernel* es más eficiente de computar, ambos ejemplos muestran que para imágenes o tensores de entrada "cuadrados" (cantidad de filas igual a columnas), y una cantidad de PEs constante, se debe dimensionar el arreglo con $CA_H = CA_W$ para maximizar la eficiencia de no-solapamiento. Esto puede verse claramente en las Tablas 4.2 y 4.3, donde al tomarse 32×32 entradas por ciclo de procesamiento, la eficiencia es superior que con 64×16 y 16×64, a pesar de que se tenga la misma cantidad de entradas totales.

4.2.2. Análisis y modelado temporal

El modelado de los tiempos de procesamiento resulta de suma importancia a la hora de decidir y dimensionar la arquitectura del operador. Una primera estimación de estos tiempos de cómputo mediante modelos simples se vuelve crucial a la hora de optimizar el *pipeline* entre la entrada/salida de datos al acelerador y el procesamiento propiamente dicho. De esta forma, se podrían predecir cuellos de botella y tiempos "muertos", para luego minimizarlos en base al diseño de una arquitectura con las dimensiones adecuadas.

Dado que la principal característica de la arquitectura de procesamiento simplicial es la de convertir las entradas al dominio tiempo, de modo de realizar los productos mediante sumas de un mismo peso por cada ciclo de reloj, es evidente que el tiempo de procesamiento en sí mismo depende de la duración de esta codificación en tiempo. Como ya se mencionó en la Sección 4.1, esta codificación en dominio tiempo se realiza mediante la comparación de las entradas con una rampa digital, para la cual es posible configurar su valor inicial y final. La duración en término de ciclos de reloj de esta rampa es el factor determinante del tiempo de procesamiento de una operación ChSymSim, para uno o más elementos de procesamiento en paralelo. Dada una rampa que se inicia en un valor r_{min} y finaliza en r_{max}, el tiempo de cómputo se define como:

$$T_{PE} = (r_{max} - r_{min}) + PE_{latency} \ . \tag{4.5}$$

En muchos casos de Redes Neuronales, a menos que se trate de la capa de entrada o se disponga de activaciones con saturaciones, no se tiene conocimiento de estos rangos o valores iniciales y finales de la rampa (r_{min} y r_{max}), por lo que resultaría más conveniente estimar este tiempo de cómputo en base al rango máximo, que está dado por el número de bits de precisión de la entrada q_{in}, de modo que la Ec. (4.5) puede reescribirse como:

$$T_{PE} = 2^{q_{in}} + PE_{latency} \ . \tag{4.6}$$

En ambas Ecs. (4.5) y (4.6) se tiene un término adicional, denominado $PE_{latency}$, que se utiliza para considerar los tiempos de latencia del operador (por registros de *pipeline*) y otros retardos como estados de espera o lectura del controlador del acelerador ChSymSim.

Naturalmente, este tiempo de cómputo de los PEs del operador ChSymSim no es suficiente para describir el tiempo de cómputo total, al ser necesaria una carga previa tanto de entradas como de parámetros para poder realizar la operación. Teniendo en cuenta estos tiempos de carga de datos, la naturaleza secuencial del procesamiento de canales y su acumulación en el algoritmo ChSymSim, el tiempo de cómputo de

un canal de entrada está dado por:

$$T_{CH} = T_{fill} + T_{PE} \ , \tag{4.7}$$

donde se considera una secuencia de carga de datos de entradas y parámetros (T_{fill}) y procesamiento de los mismos (T_{PE}). Por otra parte, si se tienen memorias internas de entrada y parámetros con *buffers*, como los mostrados en la Sección 4.1, es posible cargar nuevos datos al mismo tiempo que los anteriores son procesados, por lo que el tiempo de cómputo de un canal de entrada queda definido por el proceso de mayor duración, de manera tal que:

$$T_{CH} = \max\left(T_{fill},\ T_{PE}\right) \ . \tag{4.8}$$

El tiempo de carga de datos total T_{fill} de las Ecs. (4.7) y (4.8) contiene tanto a la escritura de los registros internos de entradas como los que corresponden a los parámetros tales como *bias*, pesos simétricos y SE. En el caso de las entradas del acelerador, este tiempo de carga se halla fuertemente influenciado por las dimensiones del arreglo de *buffers* y comparadores ($CA_H \times CA_W$), de manera que para llenar el arreglo completo se tiene:

$$T_{inputs} = T_{setup} + T_{write} \times CA_W \times \left\lceil CA_H \times \frac{q_{in}}{BW} \right\rceil + T_{latency} \ , \tag{4.9}$$

que describe al modo "filas", mientras que para el modo "columnas" se tiene:

$$T_{inputs} = T_{setup} + T_{write} \times CA_H \times \left\lceil CA_W \times \frac{q_{in}}{BW} \right\rceil + T_{latency} \ . \tag{4.10}$$

La elección entre las Ecs. (4.9) y (4.10) está dada por la arquitectura misma del acelerador, dependiendo si internamente el *bus* (de ancho BW en término de bits) alimenta a varias columnas o filas a la vez. Cualquiera que sea la configuración del *bus*, el tiempo de escritura de entradas también se ve afectado por el tiempo T_{write} que le toma a la unidad encargada de escribir dichas entradas en presentar al bus completo con nuevos datos, ya sea el microprocesador (unidad de control del sistema) o un controlador DMA dedicado. Además de lo anteriormente mencionado,

se añaden el tiempo T_{setup} para modelar la configuración de la transferencia en sí misma, junto con $T_{latency}$ para considerar las latencias de la unidad que escribe los datos en el acelerador.

Para la estimación de los tiempos de carga de parámetros al operador ChSymSim, los factores dominantes son el tamaño del *kernel* a computar ($k_H \times k_W$), que determina el número de SE y pesos simétricos a enviar, junto con la cantidad de *features* de salida a producir en paralelo (OF), obteniendo una expresión de la forma:

$$\begin{aligned} T_{params} = T_{setup} + T_{write} \times \Bigg\{ & \left\lceil (k_H \times k_W) \times \frac{q_{SE}}{BW} \right\rceil + \cdots \\ & + \left\lceil (k_H \times k_W + 1) \times \frac{q_w}{BW} \right\rceil \times OF + \cdots \\ & + \left\lceil OF \times \frac{q_b}{BW} \right\rceil \Bigg\} + T_{latency} \ , \end{aligned} \tag{4.11}$$

en la que se vuelven a utilizar los tiempos T_{setup}, T_{write} y $T_{latency}$, para modelar los retardos adicionales que presenta la unidad encargada de escribir los parámetros, independientes de la cantidad de datos a escribir. En la Ec. (4.11) también se consideran los bits de precisión para los elementos estructurantes q_{SE}, pesos simétricos q_w, *bias* q_b y el ancho del bus de parámetros BW. Considerando el cómputo en serie de los canales de entrada, la expresión completa de la Ec. (4.11) debe ser utilizada una sola vez por ciclo de *features* de salida, mientras que para cada ciclo de cómputo de canal de entrada se ignora el término relacionado al *bias*, resultando en:

$$\begin{aligned} T_{params} = T_{setup} + T_{write} \times \Bigg\{ & \left\lceil (k_H \times k_W) \times \frac{q_{SE}}{BW} \right\rceil + \cdots \\ & + \left\lceil (k_H \times k_W + 1) \times \frac{q_w}{BW} \right\rceil \times OF \Bigg\} + T_{latency} \ . \end{aligned} \tag{4.12}$$

El tiempo de carga total queda definido de forma similar a lo visto en las Ecs. (4.7) y (4.8), donde la elección entre la suma o el máximo entre los tiempos de carga de entradas y parámetros depende de si se posee o no la capacidad de paralelizar ambas transferencias de datos, en otras palabras:

$$T_{fill} = T_{inputs} + T_{params} \ , \tag{4.13}$$

cuando una misma unidad escribe entradas y parámetros en serie, mientras que si tanto entradas como parámetros son escritos en el acelerador en paralelo, se tiene que:

$$T_{fill} = \max\left(T_{inputs},\ T_{params}\right)\ , \tag{4.14}$$

para lo que se requiere no solo de dos unidades de transferencia de datos, sino que además es necesario que el acelerador posea *buses* independientes, y preferentemente memorias SRAM diferentes para almacenar estos datos.

Finalmente, para obtener una estimación del tiempo total de la capa ChSymSim a computar, se debe multiplicar el valor T_{CH} obtenido en la Ec. (4.7) o (4.8) por el número de iteraciones sobre los canales de entrada N_{CH}, sobre *features* de salida y la cantidad de pasos de filtrado espacial en 2D, obteniendo así:

$$T_{layer} = T_{CH} \times N_{CH} \times \left\lceil \frac{OF_{layer}}{OF_{core}} \right\rceil \times \left\lceil \frac{Im_H}{CA_H} \right\rceil \times \left\lceil \frac{Im_W}{CA_W} \right\rceil\ . \tag{4.15}$$

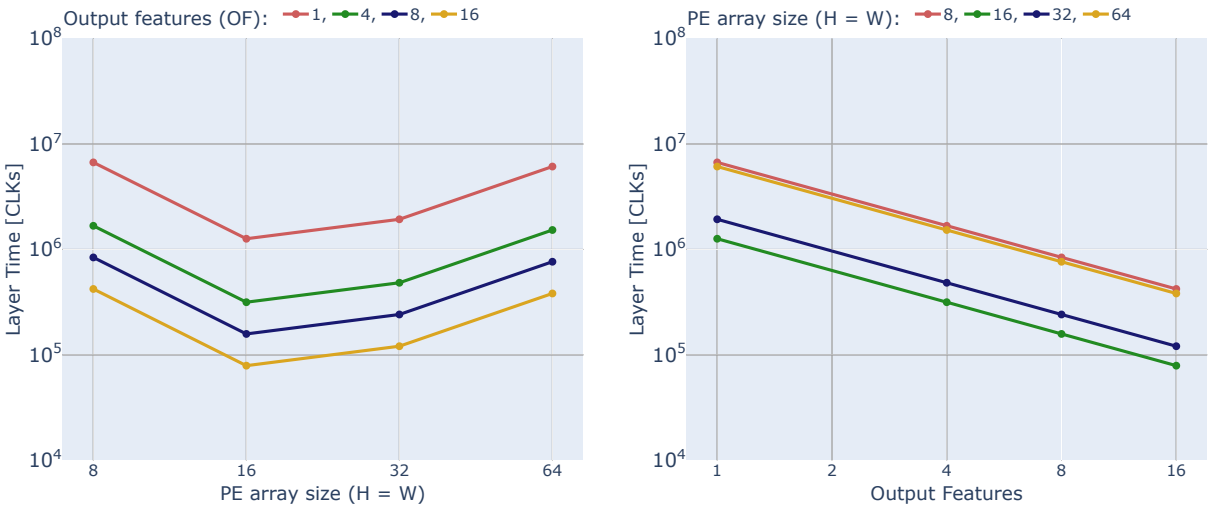
Naturalmente, este tiempo total de capa T_{layer} es una estimación pesimista, dado que se modela (mediante las funciones techo) que la cantidad de *features* y pasos de filtrado de la capa son múltiplos enteros de las dimensiones del acelerador, a pesar de que la última iteración de los *features* y pasos de filtrado pueden dar como resultado un tiempo de cómputo más corto que lo obtenido en la Ec. (4.15), debido a que el número de datos requeridos para procesar es menor y para la última de todas las iteraciones ya no se requiere de datos nuevos por lo que el tiempo de cómputo está dado por solamente el procesamiento en sí mismo.

Por otra parte, los tiempos de escalado/ReLU y salidas suelen ser despreciables en capas de filtrado espacial 2D tradicionales, al igual que sucede en el caso del algoritmo ChSymSim, ya que para poder realizar estas operaciones es necesario haber finalizado todos los ciclos de procesamiento de canales de entrada, que generalmente tiene una duración considerablemente superior. Este tiempo de escalado y salidas es incluso menos influyente en el tiempo total de la capa cuando se cuenta con un *bus* independiente y *buffers* que permitan la paralelización de estas operaciones y el cálculo de la capa ChSymSim propiamente dicha. Si en cambio se computan capas de filtrado del tipo *depth-wise*, podría representar un cuello de botella en el tiempo de cómputo total, al requerirse un escalado/ReLU y la posterior extracción de

resultados por cada ciclo de procesamiento de canal de entrada. Los tiempos de configuración del acelerador (incluyendo la máscara para forma del *kernel* SM) también son ignorados en este análisis, debido a que estos ocurren solamente al principio del cómputo de la capa y no es necesario repetirlo hasta que el procesamiento completo haya finalizado y se desee computar una capa diferente de la Red Neuronal.

En base a los modelos temporales expresados anteriormente, se muestra en las Figs. 4.10 una estimación del tiempo total de cómputo (en ciclos de reloj) de la capa Layer3_1 de la red ResNet18 ChSymSim propuesta en el Apéndice D, en relación al tamaño del acelerador en cuanto al arreglo de entrada (Fig. 4.10a) y *features* de salida en paralelo (Fig. 4.10b). Esta capa presenta una entrada de tamaño 56×56 con 16 canales y se computan 32 *features* a partir de *kernels* 3×3 (*stride* 2 y *padding* 1). Para la estimación del tiempo que demora el acelerador en procesar la capa entera, se asume que tanto las entradas (de 8 bits) como los parámetros (SE con 1 bit, 8 bits para pesos y *bias* de 16 bits) pueden cargarse en paralelo, mediante DMAs con $T_{setup} = 12$, $T_{write} = 1$ y $T_{latency} = 6$. Además, se considera que dicha escritura de datos en el acelerador se produce mientras este se encuentra procesando los datos previos (con $T_{PE} = 263$), empleando entonces las Ec. (4.8) y (4.14). En este caso en particular, se utiliza el modo "columna" para la carga de entradas, haciendo uso de la Ec. (4.10), aunque tratándose de un arreglo de PEs y tensor de entrada cuadrados, ambas Ecs. (4.9) y (4.10) generan el mismo resultado.

En la Fig. 4.10a se puede apreciar un punto de quiebre, cuando el tamaño del arreglo de PEs es 16×16, correspondiendo al tiempo total de capa mínimo, mientras que en la Fig. 4.10b se observa una tendencia monótonamente decreciente al aumentar la cantidad de *features* de salida que se computan en paralelo. A partir de 16×16 PEs y a medida que se incrementa el tamaño del arreglo, el tiempo de escritura de entradas T_{inputs} se vuelve superior a los tiempos de carga de parámetros T_{param} y de procesamiento T_{PE} (Fig. 4.11), produciendo tiempos de espera en los que no se puede realizar ningún cómputo, e incurriendo como consecuencia en tiempos totales de capa superiores. Tal y como se observa en la Fig. 4.11, con un tamaño de *kernel* y *features* de salida fijos, el tiempo de carga de pesos simétricos es constante, cosa que no ocurre con el SE debido a la necesidad de replicarlo para cada PE en el arreglo (Fig. 4.6). En base a esto, se puede concluir que las dimensiones del arreglo



(a) (b)

Figura 4.10: Estimación de tiempo total para procesar la capa Layer3$_1$ de ResNet18 ChSymSim (Tabla D.1), al computar los 16 canales de entrada en serie: a) Tiempo total vs. tamaño del arreglo de PEs; b) Tiempo total vs. cantidad de *features* computados en paralelo.

de PEs más eficientes en relación al tiempo total del cómputo de este tipo de capa (*kernels* de 3×3) se encuentran alrededor de 16 PEs por fila y por columna, mientras que resulta conveniente expandir el acelerador en cuanto al cómputo en paralelo de *features* de salida tanto como sea posible. Esto es uno de los motivos por el cual, para el diseño final del acelerador mostrado en la Sección 4.1 y ante limitaciones de área, se opta por priorizar la cantidad de *features* generados por cada PE en lugar de expandir el arreglo en alto y ancho, eliminando la posibilidad de computar *kernels* más grandes que 9×9.

Para el caso de la Fig. 4.12, se presentan estimaciones de tiempo (en ciclos de reloj) para de la misma capa que en el ejemplo de la Fig. 4.10, pero con la diferencia de realizar el procesamiento de los 16 canales de entrada en paralelo, lo que implica que en la Ec. (4.15) la cantidad de ciclos de canal es $N_{CH} = \lceil {CH_{layer}}/{CH_{core}} \rceil = 1$. En la Fig. 4.12a puede observarse que el cómputo de canales de entrada en paralelo conlleva a tiempos totales de capa superiores en comparación con su procesamiento en serie (Fig. 4.10a), a pesar de que se elimina el ciclo de canales de la Ec. (4.15). Esto ocurre al tener tiempos de carga de entradas y parámetros superan por mucho al tiempo de procesamiento en sí mismo (Fig. 4.12b), lo que produce nuevamente

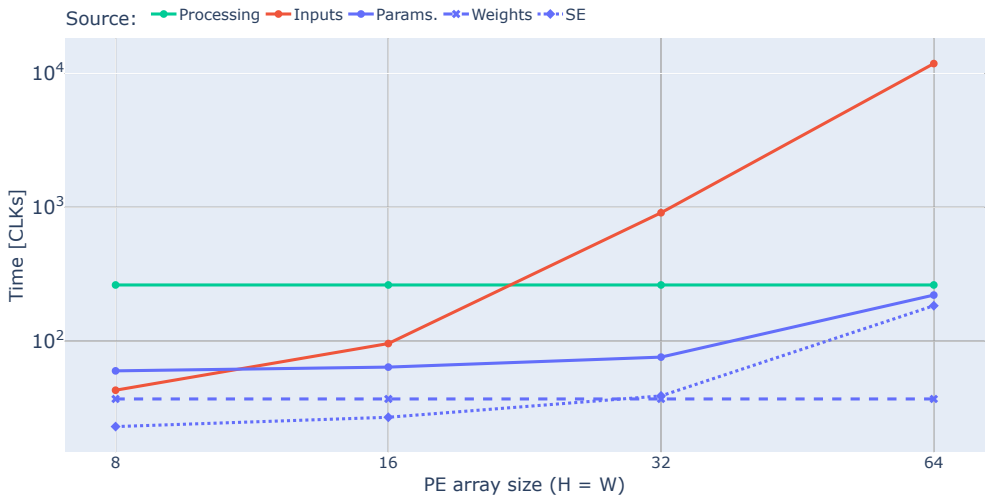
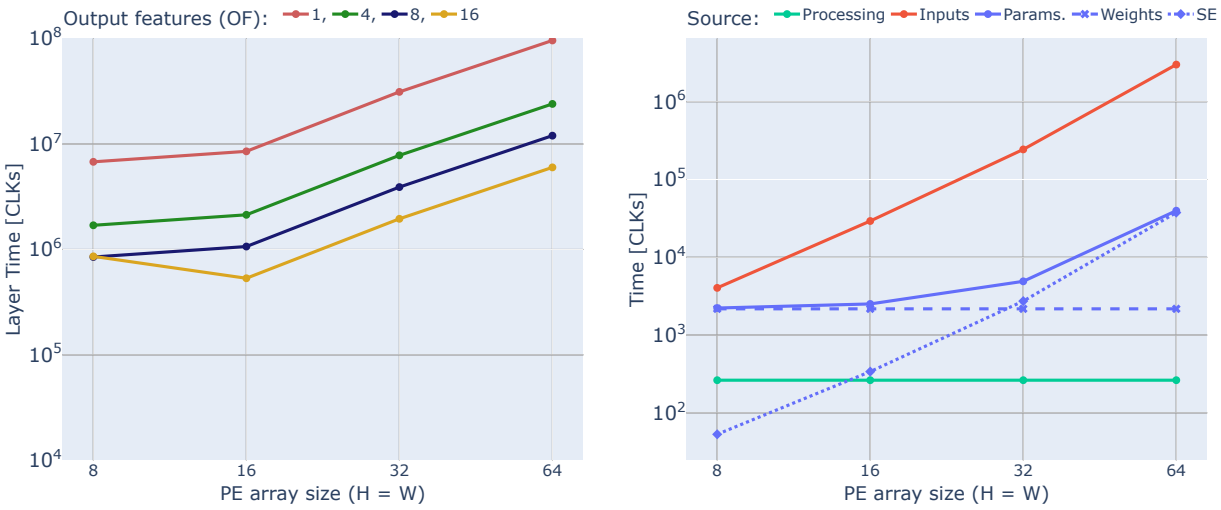


Figura 4.11: Influencia en el tiempo total de la capa Layer3_1 de ResNet18 ChSymSim (Tabla D.1), por parte del procesamiento en sí mismo, carga de entradas y escritura de parámetros, para 8 *features* de salida.

tiempos de espera sin cómputo alguno, aunque ahora acrecentados por el retardo en cargar tantos canales de entradas y parámetros, empeorando el tiempo total incluso con un pequeño arreglo de PEs. El tiempo de escritura de entradas en el acelerador es a su vez escalado fuertemente con el aumento del tamaño (alto y ancho) del arreglo de PEs, lo que conlleva a que este sea el tiempo dominante en la Fig. 4.12b. De este análisis surge la decisión de implementar la arquitectura de operador ChSymSim (Sección 4.1) que procese los canales de entrada en serie, aprovechando además esta característica para realizar algunas optimizaciones adicionales, tales como la compresión de canales para reducir ciclos de lectura/escritura al tener precisiones menores a 8 bits de entradas y/o pesos simétricos.

4.2.3. Estimación de área

El cálculo del área de silicio que puede ocupar la implementación de un diseño en particular resulta de suma importancia, incluso más que el análisis temporal previamente realizado en la Sub-sección 4.2.2. El hecho de que el diseño final "quepa" en el área disponible es un factor determinante para su realización, debiendo fabricar una versión sub-óptima si la implementación deseada no llega a satisfacer las restricciones de área impuestas por condiciones de fabricación (tecnología, relación de aspecto, etc.) y/o limitaciones de presupuesto. Es por esto que una primera esti-



(a) (b)

Figura 4.12: Estimación de tiempos para procesar la capa Layer3_1 de ResNet18 ChSymSim (Tabla D.1), al computar los 16 canales de entrada en paralelo: a) tiempo total; b) tiempos de cómputo de PEs, carga de entradas y escritura de parámetros.

mación del área que ocupa el diseño en cuestión, así como qué bloques contribuyen más a ese área total, resulta de gran ayuda para plantear qué módulos se pueden reducir, reutilizar u optimizar, de manera que la arquitectura final sea lo más pequeña posible.

Compuerta lógica	Escala ($Gate_{scale}$)
NOT	0
NAND(2)/NOR(2)	1
AND(2)	**1**
OR(2)	1
XOR(2)/XNOR(2)	1,5
MUX(2:1)	2
ADDH	2
ADDF	5
DLATCH	2
DFFQ	8

Tabla 4.4: Factores de escala de compuertas lógicas (aproximación) para estimación de área.

Con el fin de realizar una evaluación inicial del área que ocuparía el acelerador ChSymSim, se realiza un conteo de "compuertas lógicas" en los bloques que componen al operador, según se describe en la Sección 4.1. Para lograr esto, todas las compuertas lógicas del circuito se expresan en función de una referencia (compuerta

AND de dos entradas), aplicando un factor de escala tal y como se muestra en la Tabla 4.4. En este análisis, se considera a las compuertas NAND y NOR de dos entradas del mismo tamaño que una AND, a pesar de que poseen una implementación más pequeña. De forma similar, algunas implementaciones de compuertas OR pueden ocupar menos espacio que una AND. A su vez, todo registro se considera con habilitación y *reset* (ocupando 8 veces el área de una AND), a pesar de que en la realidad se tienen registros sin estas funcionalidades o bien el bloque de habilitación es común a varios registros (*clock-gating*). Estos errores de área que introducen un factor "pesimista" al considerar algunas compuertas más grandes de lo usual, son sin embargo compensados por el hecho de que todos los inversores (NOT) son ignorados. Por otra parte, cuando se tiene lógica con varias entradas (operadas en un solo ciclo de reloj), las operaciones se descomponen como árboles con la compuerta lógica de dos entradas equivalente, lo que al presentar un número N de entradas, produce un conteo de $N-1$ compuertas básicas (multiplicadas por la escala correspondiente de la Tabla 4.4). Finalmente, para estimar el área total que ocupa el circuito (generalmente en μm^2 o mm^2), se multiplica al conteo total de compuertas por el área de la compuerta de referencia, en base a las especificaciones en hojas de datos de la librería de celdas estándar de la tecnología deseada, de manera que

$$Area = \sum_{i=1}^{N_{gates}} Gate_{area} = \sum_{i=1}^{N_{gates}} Gate_H \times Gate_W \ , \tag{4.16}$$

donde N_{gates} es la cantidad de compuertas, $Gate_H$ es el alto de las celdas estándar (*pitch*) y $Gate_W$ el ancho de la compuerta de referencia.

Al tener en cuenta los bloques que componen al acelerador ChSymSim, propuesto en la Sección 4.1, se realizó un análisis en cuanto al área que ocuparía el operador ChSymSim, considerando esencialmente el arreglo de comparadores (con *barrel-shifter*) de la Fig. 4.3 y el de PEs (Fig. 4.4), omitiendo lógica de control, *buses* de salida y bancos de memoria de tanto para entradas como para parámetros. En los modelos utilizados para dicho análisis, el número de PEs en el arreglo se incrementa con el tamaño del *kernel* para reutilizar las entradas (sin tener que volverlas a cargar) con el objetivo de optimizar el cómputo con *stride* 1. En el caso de presentar grupos de PEs, se añade el sumador (y selector) de direcciones provenientes

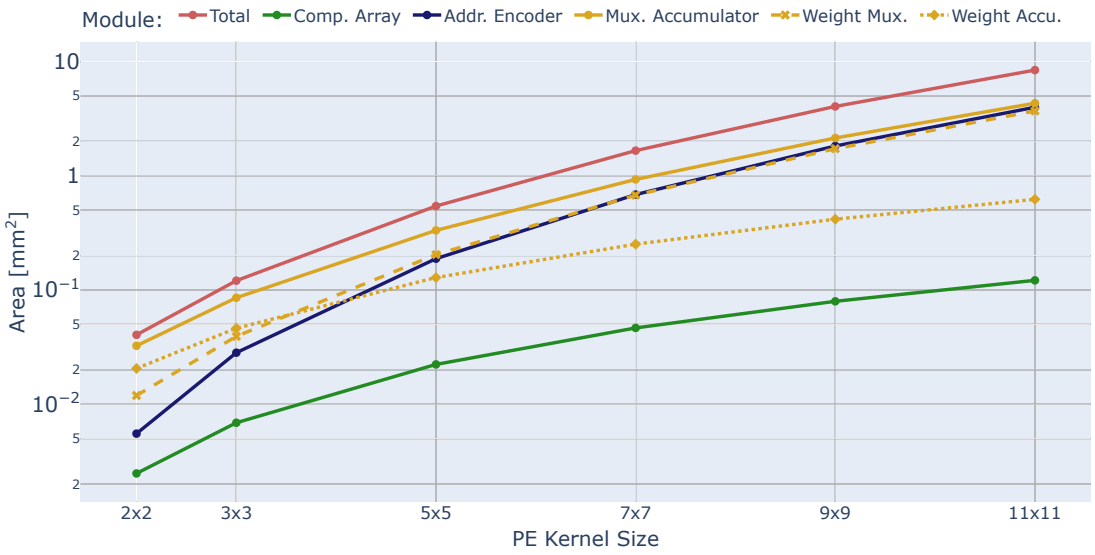


Figura 4.13: Área del operador ChSymSim y sus módulos, en función del tamaño del kernel, para un solo grupo de PEs y 8 *features* de salida con diferente SE para cada uno.

de las unidades *Addr. Encoder* y se expande el multiplexor del *Mux. Accumulator* del PE "lider" del grupo, tal y como se mostró en la Fig. 4.6. Los resultados de una primera estimación del área del acelerador se encuentra en la Fig. 4.13, en la que se presenta como escala dicho área según el tamaño de *kernel* del PE, contando 8 *features* de salida calculados en paralelo. Este caso en particular utiliza diferentes SE por *features* de salida, lo que implica que tanto las unidades de *Addr. Encoder* como *Mux. Accumulator* se encuentran repetidos 8 veces en cada PE.

Si bien el módulo *Addr. Encoder* no representa una gran contribución al área total para tamaños de *kernel* pequeños, al aumentar tanto la cantidad como el tamaño de los PEs, este se vuelve un factor a tener en cuenta. Con esto en mente, se optó por utilizar una versión compartida de los SE para todos los *features* de salida, repitiendo solamente las unidades de *Mux. Accumulator*, y cuya área estimada se muestra en la Fig. 4.14. Dado que el SE no representa un gran impacto en los resultados de clasificación de Redes Neuronales con capas ChSymSim, tal y como se mostró en la Sección 3.2, con este método se logra reducir un poco el área total del acelerador sin perjudicar su desempeño, pasando ahora a ser el bloque de *Addr. Encoder* el que menos contribuye al área total.

En las Figs. 4.13 y 4.14 se puede apreciar que el mayor impacto en el área lo

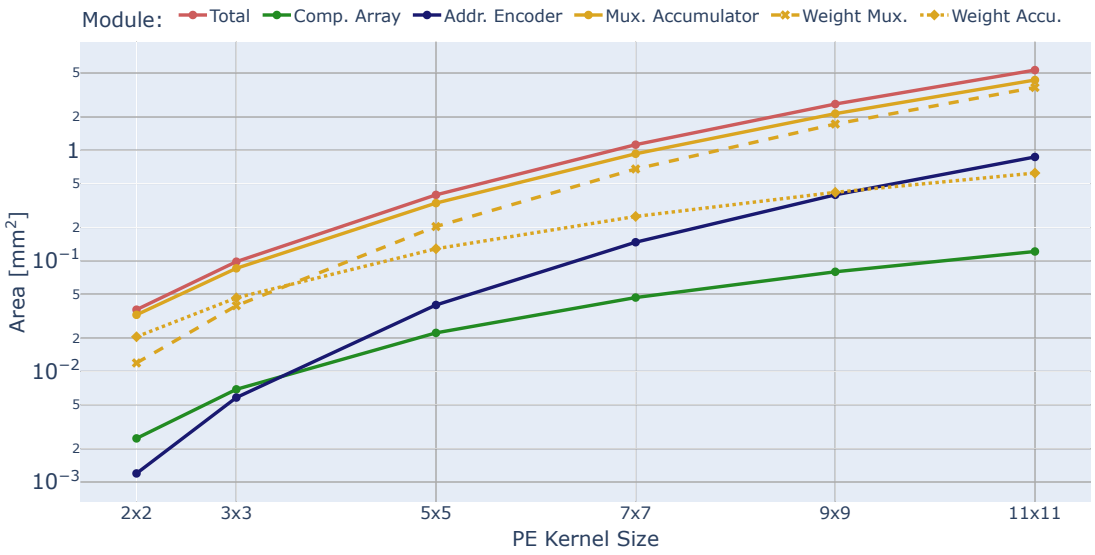


Figura 4.14: Área del core ChSymSim y sus módulos, en función del tamaño del kernel, para un solo grupo de PEs y 8 *features* de salida.

tienen el bloque de *Mux. Accumulator*, el cual es la suma de las contribuciones de área de los selectores de pesos (*Weight Mux.*) y de los acumuladores (*Weight Accu.*). A medida que el tamaño del *kernel* se incrementa, los multiplexores de pesos en el *Mux. Accumulator* pasan a ser los que representan la mayor contribución al área total, por lo que se vuelve crucial realizar alguna optimización para estos selectores. Del análisis desarrollado en el Apéndice F, en el que se evaluaron diferentes arquitecturas de multiplexores, la versión de selector *One-Hot* es la más apropiada en cuanto a reducción de área, al dividir el selector en dos sub-módulos: un *one-hot encoder* que traduce la dirección del valor a seleccionar en un 1 en la posición correcta y 0 en el resto, más otro bloque con compuertas AND y OR para enmascarar los valores no seleccionados. De esta forma, cuando se comparten las entradas y los SE para todos los *features* de salida, se puede poner un único *one-hot encoder* que codifique la dirección general del peso producida por el *Addr-Encoder*, mientras que el hardware para la selección del peso que se halla dentro de los bloques *Mux-Accumulator* se reduce a simplemente el grupo de compuertas AND y OR, minimizando así la parte del MUX[2] que se repite al incrementar el número de PEs. Desafortunadamente, al estar el acumulador de pesos compuesto por un simple *buffer* (registro seguido de *latch*) y un sumador, no es posible realizar más optimizaciones para el bloque *Mux-*

[2]Multiplexor

Accumulator en cuanto a área se refiere, siendo una reducción en la precisón (bits de salida) la única forma de minimizar el tamaño del acumulador de pesos, lo que limitaría la cantidad de canales de entrada que se pueden procesar sin la necesidad de extraer resultados parciales.

Como ya se mostró en la Sub-Sección 4.2.2, el procesamiento de canales en paralelo para la función ChSymSim no resulta eficiente en términos de tiempo total de cómputo. Con respecto al área total del operador, es evidente que toda paralelización de cómputos requiere de la repetición de bloques de *hardware*, en los que se incluyen tanto bancos de memoria como unidades de procesamiento. En el caso del acelerador ChSymSim, para realizar el cómputo de canales de entrada en paralelo se debe incorporar *hardware* adicional que pueda realizar las sumas de los resultados del procesamiento de cada canal, por lo que no se recomienda esta alternativa de diseño al no representar beneficio alguno.

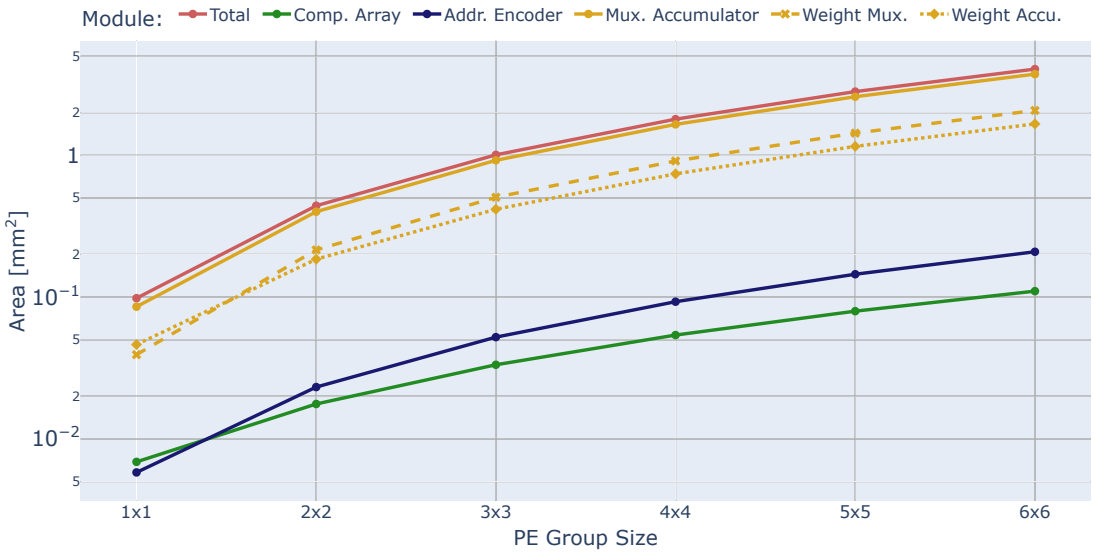


Figura 4.15: Área del core ChSymSim y sus módulos, en función del tamaño de grupo de PEs, con *kernel* de tamaño 3×3 y 8 *features* de salida por PE.

Tanto la optimización de compartir *SE*s para todos los *features* de salida, así como también el uso de selector *One-Hot*, son estrategias que surgen de la necesidad de reducir el área cuando se incrementa el tamaño del *kernel* que un sólo PE puede computar. A pesar de esto, ambas técnicas son también efectivas cuando se tienen *kernels* pequeños. Debido a esto, y teniendo en cuenta que la eficiencia de no-solapamiento (Sub-Sección 4.2.1) con arreglos de PEs limitados se reduce con

menor intensidad para *kernels* pequeños, se decidió dimensionar cada PE para procesar *kernels* de tamaño 3×3. En base a los resultados obtenidos en la Fig. 4.15, priorizando además la cantidad de *features* de salida que se pueden procesar en paralelo (de al menos 8), el tamaño de grupo de PEs escogido para el acelerador es de 3×3. Con este tamaño de grupo de PEs se pueden computar *kernels* de tamaño hasta 9×9, mientras que el área total se mantiene alrededor del 1 mm^2.

4.2.4. Consumo de potencia y energía

Para la estimación de potencia del operador se pueden utilizar los mismos modelos producidos para el análisis de área, con los que se consigue el conteo de compuertas lógicas por bloque. Al conocer el consumo de potencia dinámico ($Gate_{dyn}$) y por *leakage* ($Gate_{leak}$) de la celda de referencia, en base a las hojas de datos de la librería de celdas estándar, el consumo de potencia total se obtiene como:

$$Power_{total} = f_{CLK} \times \sum_{i=1}^{N_{gates}} Gate_{dyn} \times \alpha + \sum_{i=1}^{N_{gates}} Gate_{leak} \ , \tag{4.17}$$

con N_{gates} nuevamente como la cantidad de compuertas lógicas del circuito, f_{CLK} la frecuencia de reloj objetivo y α como un factor de actividad que tiene valores entre 0 y 1, según el porcentaje de cambios en los nodos del circuito a lo largo de un ciclo de procesamiento. Algunos de los bloques del acelerador ChSymSim mostrados en la Sección 4.1 fueron diseñados con este factor de actividad en mente. Un ejemplo de esto es el comparador en sí mismo (Fig. 4.3a), que al habilitar las etapas de comparación sólo cuando son requeridas, se logra limitar la actividad a prácticamente una única transición por nodo del circuito en todo el ciclo de procesamiento. Por simplicidad, para este análisis de consumo de potencia y energía, se considera la duración de un ciclo de procesamiento como el tiempo de cómputo intrínseco del arreglo de PEs o, en otras palabras, a lo expresado por la Ec. (4.6). Además, dependiendo de la configuración de la capa ChSymSim que se desea computar, algunos comparadores y PEs pueden desactivarse por medio de señales de habilitación de registros internos, reduciendo idealmente su actividad a 0.

Considerando estas actividades, se obtuvo una estimación inicial del consumo de

potencia del operador ChSymSim, cuyos resultados se muestran en la Fig. 4.16, en la cual se puede observar el comportamiento del arreglo de PEs y sus sub-bloques en base a la precisión de la entrada y distintos tamaños de *kernel*. Para este análisis se utilizó el modelo de acelerador ChSymSim resultante de las estimaciones de tiempo de cómputo y área: arquitectura descrita en la Sección 4.1 (Fig. 4.5), con 9×9 PEs, agrupados de a 3×3 y pudiendo computar cada uno 8 *features* (con SE compartido) a partir de *kernels* de hasta 3×3. Al igual que para las estimaciones de área y tiempo, sólo se tuvo en cuenta a los arreglos de comparadores (con sus correspondientes *barrel-shifters*) y PEs.

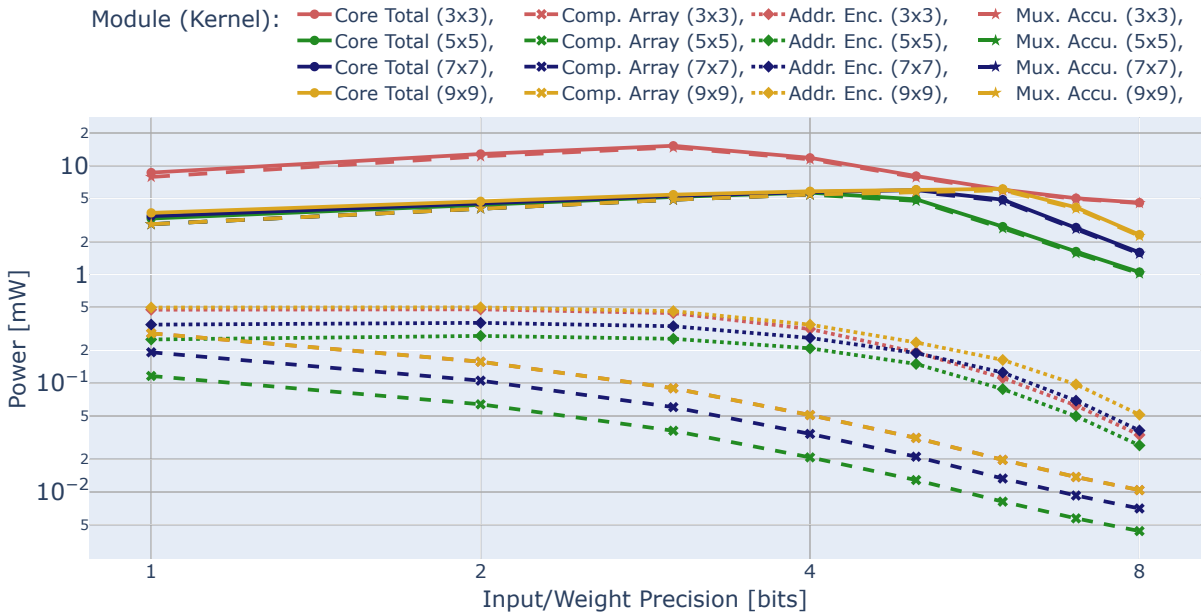


Figura 4.16: Consumo de potencia del operador ChSymSim y sus módulos, en función del tamaño del kernel y la precisión.

En la Fig. 4.16 se puede apreciar que el consumo de potencia en general decrece a medida que la precisión aumenta, debido a que el tiempo de cómputo se incrementa mientras que los nodos del circuito a partir de cierto punto ya no presentan más transiciones, produciendo así un factor de actividad menor. Un claro ejemplo de esto son los comparadores que, como se mencionó anteriormente, fueron optimizados para reducir su consumo energético, teniendo una sola transición por nodo durante el ciclo de procesamiento, lo que consigue un factor de actividad de $1/T_{PE}$. En el caso del *barrel-shifter* también se tiene una actividad de, en el peor caso, todos los nodos realizando una transición por ciclo de procesamiento, debido a que las entradas

experimentan un *bit-shift* según su precisión (y compresión de canales) solamente al ser leídas y se mantienen constantes durante el resto de su procesamiento. Si bien en la Figs. 4.14 y 4.15 se mostró que el arreglo de comparadores no es el bloque más pequeño, su reducida actividad convierte a este módulo en el de menor consumo de potencia, teniendo los mismos valores para 3×3 y 9×9 (máxima utilización del arreglo), mientras que para 7×7 y 5×5 no todas las entradas son necesarias por lo que algunos comparadores pueden "apagarse" mediante máscaras en registros de configuración.

Por otra parte, el módulo *Addr. Encoder* de los PEs tiene como principales elementos la aplicación de máscaras (mediante compuertas MUX y AND) a las señales PWM provenientes del comparador, seguidas de un sumador para generar la dirección de pesos. Al poseer entradas PWM, resulta evidente que la aplicación de las máscaras tendrán como mucho una transición por ciclo de procesamiento (cuando la señal PWM pasa de 0 a 1 o viceversa). Sin embargo, para el sumador se debe realizar un análisis estadístico, promediando la actividad ante distintos casos de datos aleatorios, y para todos los distintos valores de precisión entre 1 y 8 bits. En la Fig. 4.16 se puede observar que debido a la actividad del sumador, el consumo de potencia de este módulo es un poco más elevado que el de los comparadores, a pesar de representar una menor área (menor número de compuertas lógicas). De manera similar a los resultados de área, las unidades *Mux. Accumulator* obtienen valores de potencia muy superiores en relación a los comparadores y bloques *Addr. Encoder*, constituyendo prácticamente todo el consumo de potencia del acelerador. Las curvas de potencia estimada para el módulo *Mux. Accumulator* muestran un punto de quiebre donde el tiempo de cómputo (2^{prec}) es mayor que el tamaño del *kernel* a computar, a partir del cual la potencia decrece considerablemente con el aumento en la precisión. Esto se debe a que la cantidad de pesos a elegir es inferior a la duración del ciclo de procesamiento, por lo que durante algunos ciclos de reloj el multiplexor no presenta cambios, reduciendo así el factor de actividad. Además de lo anteriormente mencionado, mientras que para ambos casos de *kernel* pequeños (3×3) y grandes (5×5, 7×7 y 9×9) se utilizan todos las unidades *Addr. Encoder*, cuando el *kernel* a computar supera el tamaño que un solo PE puede computar, para el agrupamiento de PEs se necesita solamente una unidad *Mux. Accumulator*,

mientras que se deshabilitan las pertenecientes a los demás PEs en el grupo. Esto produce una disminución en el consumo total para el cómputo de *kernels* grandes, a costa de producir considerablemente menos salidas. Para los valores más pequeños de precisión, por ejemplo 1 y 2 bits, el tiempo de latencia del circuito domina sobre el tiempo de cómputo en sí mismo, lo que implica una reducción en la actividad y por lo tanto un consumo ligeramente menor en los bloques *Mux. Accumulator*.

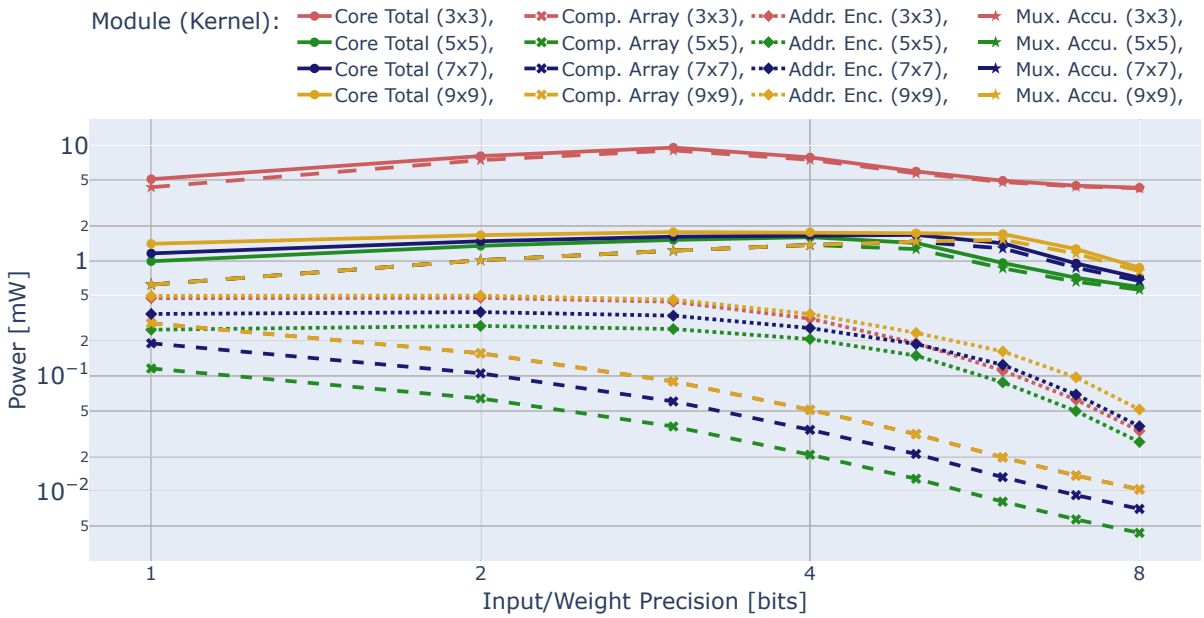


Figura 4.17: Consumo de potencia del operador ChSymSim y sus módulos, en función del tamaño del kernel y la precisión, utilizando multiplexores optimizados (entradas reordenadas).

Debido a que la mayor disipación de potencia se encuentra en el bloque *Mux. Accumulator*, resulta conveniente optimizar dicho módulo para reducir su consumo energético. En el Apéndice F se muestra una optimización para multiplexores en términos de potencia, simplemente reordenando las entradas y señal de dirección de manera que se aprovechen los pasos en los valores de dirección estrictamente crecientes (o decrecientes), y minimizando así la actividad de los nodos del multiplexor. Empleando esta técnica en los modelos de estimación de potencia, siguiendo las expresiones del Alg. 6, se presentan los resultados con el multiplexor optimizado en la Fig. 4.17. Tal y como se puede observar en esta nueva estimación de potencia, con esta simple estrategia de reordenar entradas en el multiplexor de pesos, se logra reducir el consumo total, especialmente para los casos de cómputo con *ker-*

nel grandes (mayores que 3×3) donde se deben seleccionar más pesos por ciclo de procesamiento.

En la implementación final del acelerador ChSymSim, para la selección de pesos se utiliza la versión *One-Hot* (Fig. F.2), debido a que esta variante también reduce el consumo energético. A pesar de los resultados de síntesis y simulación de selectores en el Apéndice F, que mostraron una mayor eficiencia en el reordenamiento de entradas de un MUX tradicional frente a la arquitectura *One-Hot*, el uso de esta última para minimizar el área mostrado en la Sub-Sección 4.2.3 también resulta beneficioso en términos de potencia.

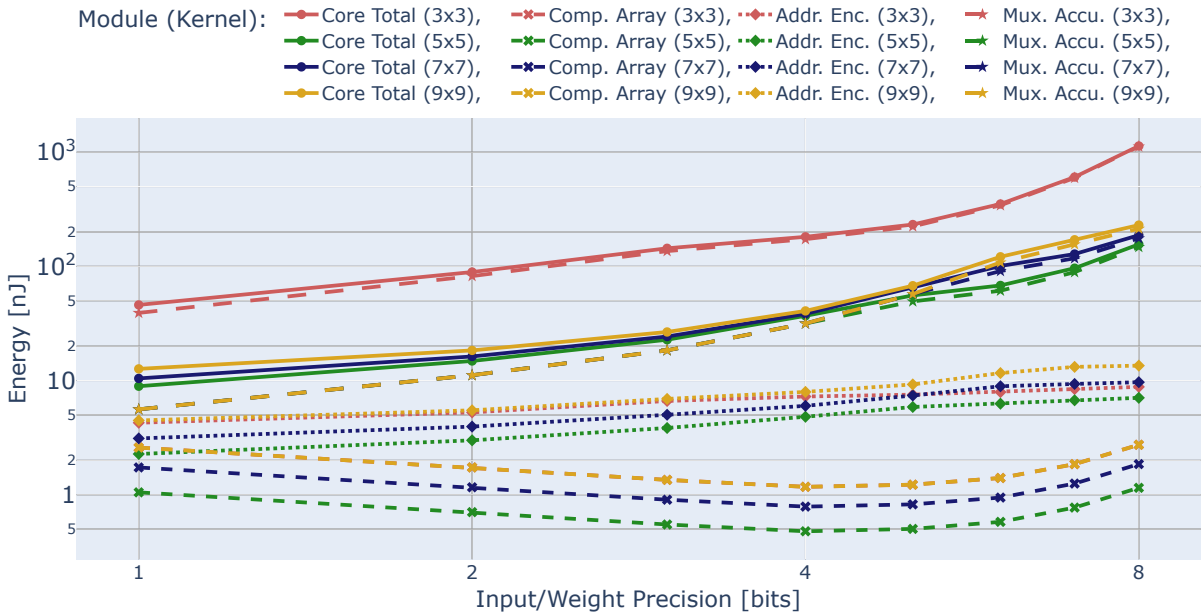


Figura 4.18: Consumo energético del operador ChSymSim y sus módulos, en función del tamaño del kernel y la precisión, utilizando multiplexores optimizados (entradas reordenadas).

A modo de concluir esta Sub-Sección de modelos de alto nivel para acelerador ChSymSim se realiza una estimación de energía en relación a la precisión de entrada (factor determinante en el tiempo de procesamiento) y distintos tamaños de *kernel* a computar, presentando los resultados en la Fig. 4.18. Para el cálculo del consumo energético de la arquitectura resultante de las decisiones y optimizaciones en esta Sección, simplemente se multiplica la potencia estimada, que en este caso corresponde a la de la Fig. 4.17, por la duración del procesamiento en segundos, según la frecuencia de reloj utilizada para dicha estimación de potencia. De esta forma,

se puede observar en la Fig. 4.18 que, a pesar de la disminución de la potencia en los bloques del acelerador por su reducida actividad, el consumo energético total se dispara cuando mayor es la precisión de las entradas, compensando esta disminución de potencia con tiempos de procesamiento considerablemente más largos.

4.3. Operador ChSymSim vs. lineal

Una vez establecida la estructura final del acelerador ChSymSim, con sus dimensiones obtenidas a partir de modelos de alto nivel, resulta de interés compararlo con las arquitecturas de cómputo tradicional, para confirmar los beneficios del cómputo simplicial en DNNs. En este sentido, los modelos de alto nivel son convenientes, dado que permiten una exploración del espacio de diseño, comparando tiempos, área y consumo, sin la necesidad de implementar/sintetizar cada una de ellas, lo cual requeriría de mucho tiempo de cómputo. Sin embargo, para poder contrastar el desempeño del acelerador propuesto en un entorno más realista, en esta sección se lleva a cabo la comparación entre ambas alternativas con base a resultados de síntesis lógica y simulaciones post-síntesis.

4.3.1. Comparación con modelos de alto nivel

Dado que se diseñó el operador ChSymSim pensando específicamente en capas de filtrado espacial, la estructura de cómputo estándar (lineal) más relevante para comparar el acelerador diseñado en esta tesis es el correspondiente a una convolución. Con el objetivo de realizar la comparación de la forma más justa posible, se diseñó un operador convolucional con una estructura similar al acelerador ChSymSim, cuya arquitectura se presenta en la Fig. 4.19. Para este operador convolucional se aprovecharon algunas de las estrategias utilizadas para la versión ChSymSim, entre las que se encuentran:

- uso de registros *buffer* para *pipeline* entre escritura/lectura de datos y procesamiento;
- cómputo de canales en serie, con compresión de los mismos para reducir el número de transferencias de datos totales cuando la precisión es menor a 8

bits;

- multiplexores optimizados para selecciones con dirección monótonamente creciente o decreciente;
- arreglo de PEs con la mayoría dimensionados para *kernels* pequeños (hasta 3×3) mientras que unos pocos se expanden para permitir el cómputo de *kernels* grandes;

entre otras técnicas de optimización exploradas a lo largo del modelado de alto nivel de la Sección 4.2.

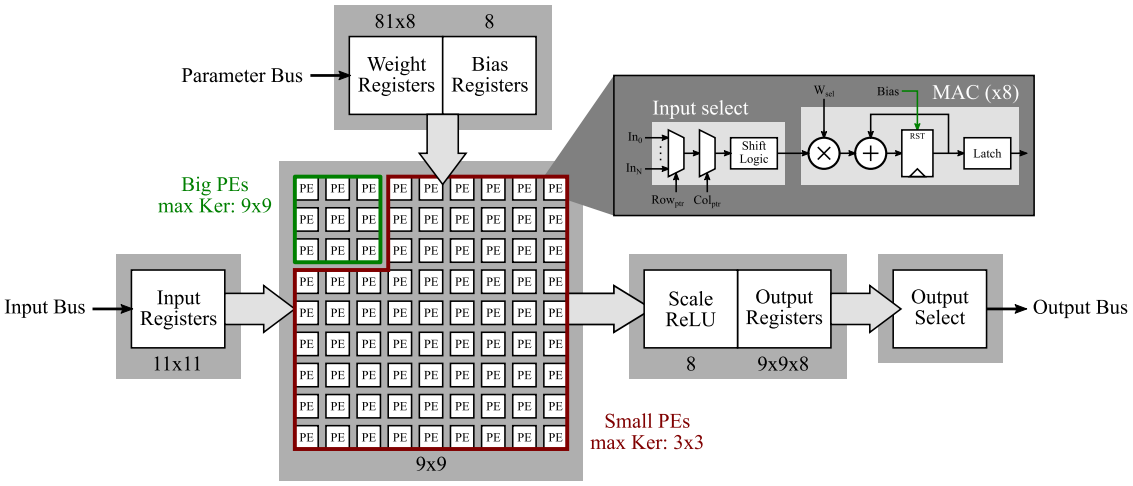
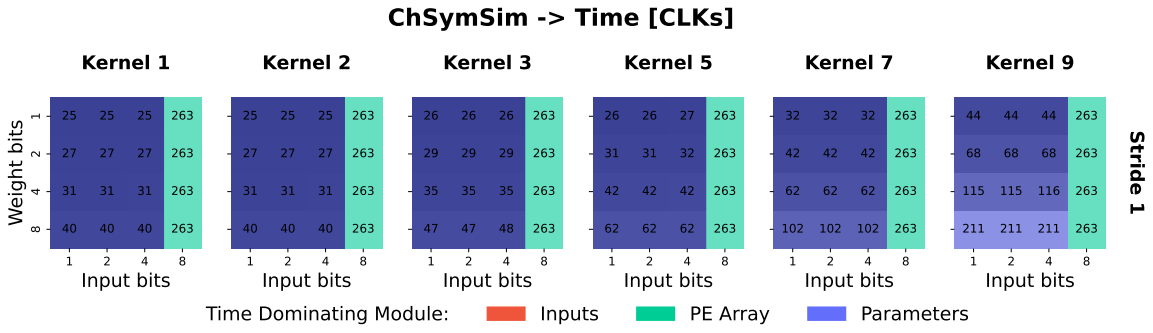


Figura 4.19: Diagrama en bloques del operador convolucional.

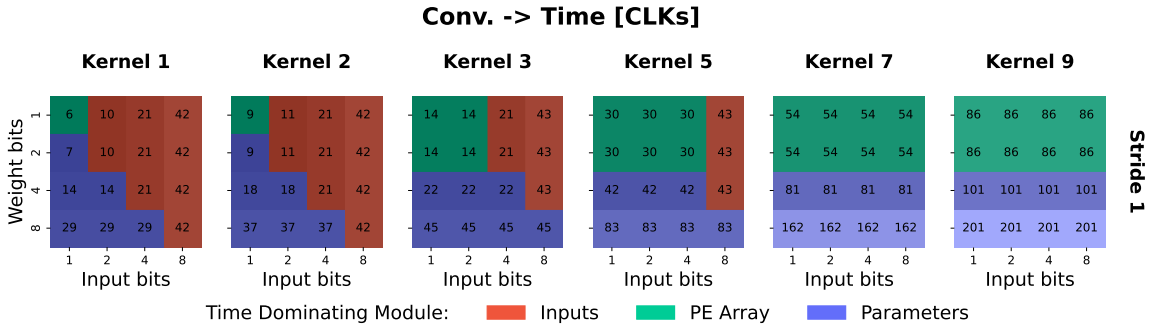
Este diseño de operador convolucional, mostrado en la Fig. 4.19, cuenta con un arreglo de 9×9 PEs, donde cada PE consiste en una parte de selección de entradas y lógica de *bit-shift* para ajuste de precisión y selección de canales comprimidos, seguida de 8 unidades MAC para computar *features* de salida en paralelo. Para realizar un ciclo de procesamiento con este acelerador se emplea un mecanismo de control similar al de las unidades de escalado, redondeo y ReLU, en el que se utilizan contadores para manejar los punteros a filas y columnas del *kernel*, con los cuales se seleccionan la entrada y el peso correspondiente para multiplicar y acumular en cada PE. De esta forma, el tiempo de procesamiento de un canal de entrada en un PE del acelerador convolucional es de

$$T_{PE} = (k_H \times k_W) + PE_{latency} \ , \tag{4.18}$$

con $k_H \times k_W$ como el tamaño del *kernel* a computar. Dado que el conteo se realiza primero por columnas (cambiando el puntero a cada ciclo) y luego por filas (cambiando el puntero una vez por ciclo de columnas), con el fin de reducir el consumo de potencia se dispone como multiplexor de entradas al selector de filas, mientras que la selección de columnas se realiza mediante la etapa de MUX siguiente. Debido a que en una convolución el peso seleccionado para todos los PEs es el mismo y solo difiere para *features* de salida, como optimización adicional se comparten los multiplexores de pesos (y bloques de *bit-shift* aritmético), replicando esta estructura solamente para el número de *features* de salida en paralelo. Al igual que con el operador ChSymSim, el *bias* es cargado en el registro de acumulación al inicio del primer ciclo de procesamiento (una sola vez para todos los canales de entrada), como un valor de *reset* del registro.



(a)



(b)

Figura 4.20: Tiempos de procesamiento de un canal de entrada, considerando carga de entradas y parámetros en *pipeline* con *buses* de 64 bits, ignorando carga inicial para: a) acelerador ChSymSim; b) operador convolucional.

Una primera comparación que puede realizarse entre ambos aceleradores es el tiempo que les lleva realizar el procesamiento de un canal de entrada. Para esto se

utilizaron las ecuaciones desarrolladas en la Sub-Sección 4.2.2, considerando completa paralelización entre carga de entradas, pesos y procesamiento propiamente dicho. Los resultados de esta comparación, para distintos valores de precisión de entradas/pesos y tamaños de *kernel*, se presentan en la Fig. 4.20, donde los mapas de calor mostrados en la Fig. 4.20a corresponden al operador ChSymSim, mientras que en la Fig. 4.20b corresponden al acelerador convolucional. En ambos casos, cuando se requieren menos transferencias por la configuración de precisión, los tiempos de carga de entradas y/o parámetros son divididos por la cantidad de canales almacenados en un byte, mostrando con un color diferente cuál es el tiempo dominante en la duración total del ciclo de procesamiento de cada canal de entrada. De ambas figuras es posible concluir que para *kernels* pequeños (menores o iguales a 3×3), dado los limitados pasos de selección de entradas y pesos del operador convolucional, su tiempo de cómputo es inferior al del acelerador ChSymSim. Esto se invierte con el cómputo de *kernels* grandes (5×5 o mayores) donde, al no importar la cantidad de entradas para el cómputo simplicial, se obtienen menores tiempos de cómputo con el operador ChSymSim al bajar la precisión de entrada.

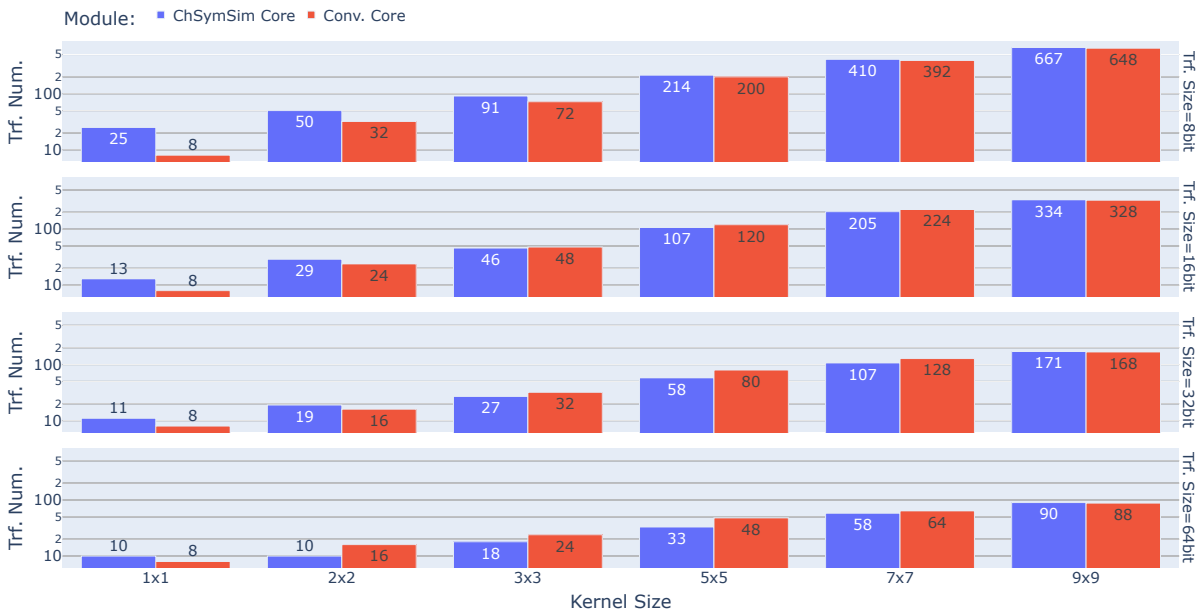
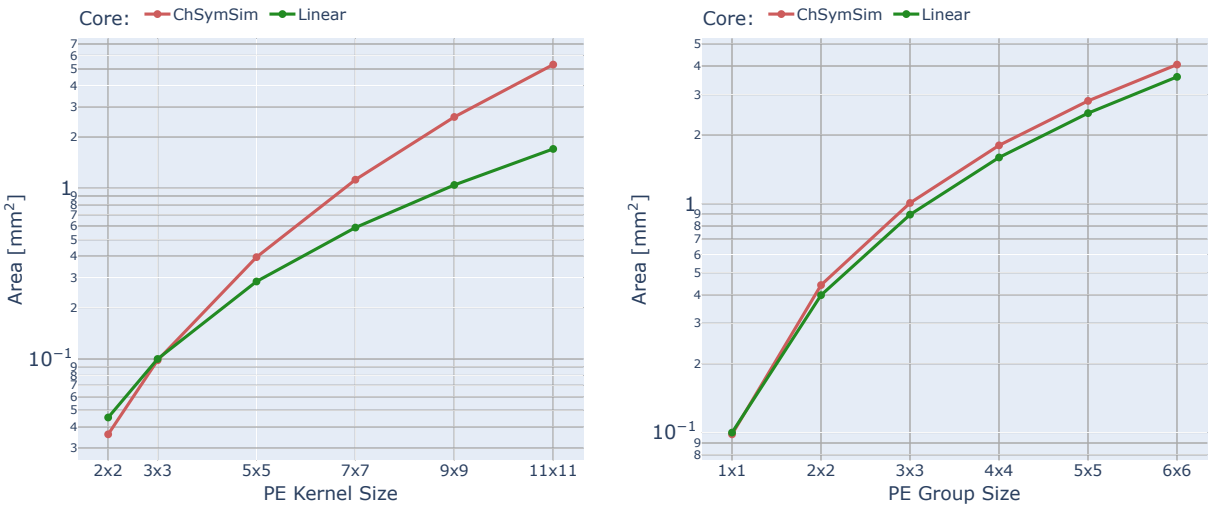


Figura 4.21: Número de transacciones para la carga de parámetros, requeridas por los procesadores convolucional y ChSymSim.

En la mayoría de los casos de *kernel* y precisión en la Fig. 4.20, se tiene como tiempo dominante a la duración de la transferencia de parámetros, definiendo así el

tiempo total de procesamiento del canal. Cuando se da esta situación, la diferencia fundamental entre los operadores ChSymSim y convolucional es el número de parámetros (o transferencias) requeridos para el cómputo. La Figura 4.21 muestra el número de transferencias necesarias para ambos aceleradores en base al tamaño de dichas transacciones. Para el caso ChSymSim, si bien se tiene el agregado del SE, lo que implicaría más datos a ser enviados al operador, debido a que las posiciones de las entradas son irrelevantes para este tipo de cómputo, los pesos simétricos del *kernel* pueden ser posicionados en la memoria en forma de arreglo de una sola dimensión, por lo que sólo se requiere enviar la cantidad exacta correspondiente al tamaño del *kernel*. Para el acelerador convolucional, al acceder la memoria de pesos en base a punteros de filas y columnas, con *kernels* que no llenen dicha memoria se debe realizar un *padding* de 0s para que al transferir los datos, estos se posicionen en las ubicaciones adecuadas, requiriendo de esta forma más datos que el operador ChSymSim. Una alternativa a esto sería contar con *hardware* adicional para combinar ambos punteros de fila y columna en una sola dirección en 1D (aumentando el área/potencia), o bien realizar la transferencia de pesos con diferentes configuraciones (requiriendo de más memoria). Para tamaños de *kernel* pequeños, las latencias de la unidad que transfiere los parámetros a los aceleradores (controlador DMA) incrementan los tiempos de escritura de datos. Debido a ello, para el caso ChSymSim se requiere ligeramente más tiempo incluso si necesitara menos datos totales, al tener el DMA que leer dos configuraciones (pesos y SE). Por otra parte, con tamaños de *kernel* grandes que no llenen la memoria interna de pesos (5×5 y 7×7), el tiempo de carga de parámetros es significativamente inferior para el acelerador ChSymSim, observado en la Fig. 4.20 con 8 bits de pesos, que es el tiempo de transferencia puro y sin ser escalado por el número de canales por byte como en los demás casos.

En relación al comportamiento del área total de los operadores ChSymSim (arreglo de comparadores y PE) y convolucional (solo arreglo de PE) según las dimensiones de los mismos, la Fig. 4.22 muestra que para lograr una menor área con el acelerador ChSymSim, se debe elegir un tamaño de *kernel* de 3×3, manteniendo un sólo grupo de PEs de manera que 3×3 también sea el tamaño máximo de *kernel* (Fig. 4.22a). Para el cómputo de *kernels* de mayor tamaño, en términos de área resulta más conveniente incrementar el número de PEs por grupo (Fig. 4.22b), aún



(a) (b)

Figura 4.22: Comparación de área entre operadores ChSymSim y convolucional, con 8 *features* de salida por PE: a) en función de la cantidad del tamaño de *kernel* por PE, para un solo grupo de PEs; b) en función del tamaño de grupo de PEs, con tamaño de *kernel* por PE de 3×3.

si el operador ChSymSim ocupa más espacio que el convolucional, ya que el área total escala en menor medida que con el tamaño de *kernel* por PE.

La Figura 4.23 presenta una estimación para la energía de los procesadores ChSymSim y convolucional, en función de la precisión de las entradas y para tamaños de *kernel* típicos de 3×3, 5×5, 7×7 y un máximo de 9×9.

Para el cálculo de energía se consideran los modelos de aceleradores utilizados para el cálculo de área de la Fig. 4.22, con un tamaño de *kernel* por PE de 3×3 y grupos de 3×3 PEs, y las ecuaciones de tiempos intrínsecos de los operadores (es decir, el tiempo que le toma al acelerador procesar un canal de entrada sin considerar lectura/escritura de datos). Aquí es posible observar que para *kernels* grandes, utilizando el agrupamiento de PE, el consumo de energía del acelerador ChSymSim es igual o inferior al del operador convolucional siempre y cuando el tiempo de cómputo simplicial (que depende de la precisión de la rampa) sea menor al tiempo de procesamiento del acelerador convolucional. Para los casos con tamaño de *kernel* más grandes, tales como 7×7 y 9×9, las intersecciones entre los consumos energéticos de los operadores se da alrededor de los puntos donde el tiempo de procesamiento intrínseco de los aceleradores es similar, o en otras palabras $2^{q_{in}} \approx$

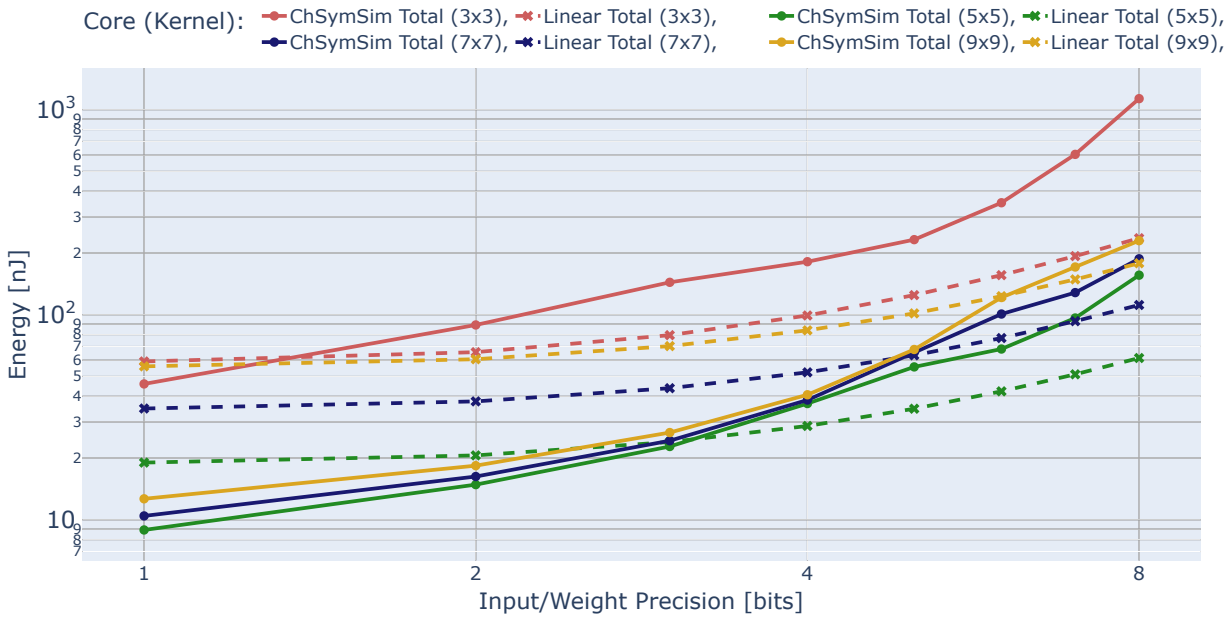


Figura 4.23: Energía estimada para los aceleradores ChSymSim y convolucional, en función de la precisión de las entradas. Procesadores con tamaño de *kernel* mínimo de 3×3, grupos de PE de 3×3 y 8 *features* de salida en paralelo.

$K_H \times K_W$. A medida que el tamaño de *kernel* disminuye, esta tendencia desaparece y los consumos del operador ChSymSim se vuelven inferiores al del convolucional solamente con precisiones de entradas reducidas. En general, se puede concluir que la arquitectura ChSymSim presenta un menor consumo que una implementación tradicional cuando el tamaño de *kernel* a computar es grande, y la precisión de las entradas es baja.

Como otro punto de comparación, el operador Lineal/Convolucional suele tener multiplicadores que solo operan con entradas signadas (en la mayoría de los casos) o no-signadas, por lo que para poder usar ambos tipos se necesita de *hardware* adicional. El procesador ChSymSim, en cambio, puede cambiar entre el uso de valores de entrada signadas y no-signadas simplemente con los rangos de la rampa con la que son comparadas. Además de esto, el acelerador ChSymSim resulta más eficiente cuando se toma como entradas a las salidas de una activación ReLU. Luego de este tipo de activaciones, todos los valores negativos son anulados, de manera que si se usan magnitudes signadas (como en el caso usual de la convolución) se pierde un bit de precisión (el bit de signo se vuelve innecesario). Dado que el operador ChSymSim puede utilizar valores no-signados de forma nativa, es posible aprovechar todos los

bits de entrada para mejorar la precisión del cómputo. Si por el contrario la precisión no es tan importante, el acelerador Convolucional puede reducir un poco su consumo energético al no haber transiciones en los nodos del circuito que corresponden al bit más significativo (bit de signo siempre 0). Por otra parte, al tener solamente entradas positivas pero signadas, el procesador ChSymSim reduce a la mitad la duración de la rampa ya que no es necesario realizar la comparación con el rango completo de la precisión, lo que implica una reducción considerable tanto en tiempo de procesamiento como en consumo energético.

4.3.2. Síntesis y resultados de simulación

Los modelos de alto nivel son de gran utilidad para tener un modelo de primer orden sobre el comportamiento del área, potencia y tiempos de cómputo del acelerador. Si bien con esta información es posible realizar algunas optimizaciones a nivel de diseño, previo a su implementación, sigue siendo necesario realizar una síntesis de la arquitectura final y posteriores simulaciones para contrastar las estimaciones de alto nivel con modelos más complejos y cercanos a la realidad. Por esta razón, se realizó la síntesis de los aceleradores completos ChSymSim y convolucional, utilizando las herramientas de *Synopsys Design Compiler* con celdas estándar mixtas (hVt, rVt y lVt) en tecnología TSMC 64nm LP y con especificaciones (*constraints*) de diseño para una frecuencia de reloj máxima de 50 MHz. Los operadores sintetizados poseen las mismas dimensiones que sus homólogos modelados y comparados en la Sub-Sección 4.3.1, con tamaño de *kernel* de 3×3, 8 *features* de salida y la capacidad de acumular hasta 256 canales de entrada por cada PE. Para el cómputo de *kernels* de mayor tamaño, el acelerador ChSymSim suma las direcciones de pesos de hasta 3×3 PEs, lo que permite computar *kernels* hasta un máximo de 9×9. Por otra parte, el operador convolucional simplemente expande los multiplexores en algunos PEs para poder seleccionar las 9×9 entradas y pesos.

Los valores de área reportados por la herramienta de síntesis para los procesadores ChSymSim y convolucional, incluyendo sus diversos módulos, se detallan en la Tabla 4.5 y se ilustran en la Fig. 4.24. Las áreas de cada bloque (o sub-bloque), representadas en la Fig. 4.24, están dadas tanto por el color (valor en mm^2) como

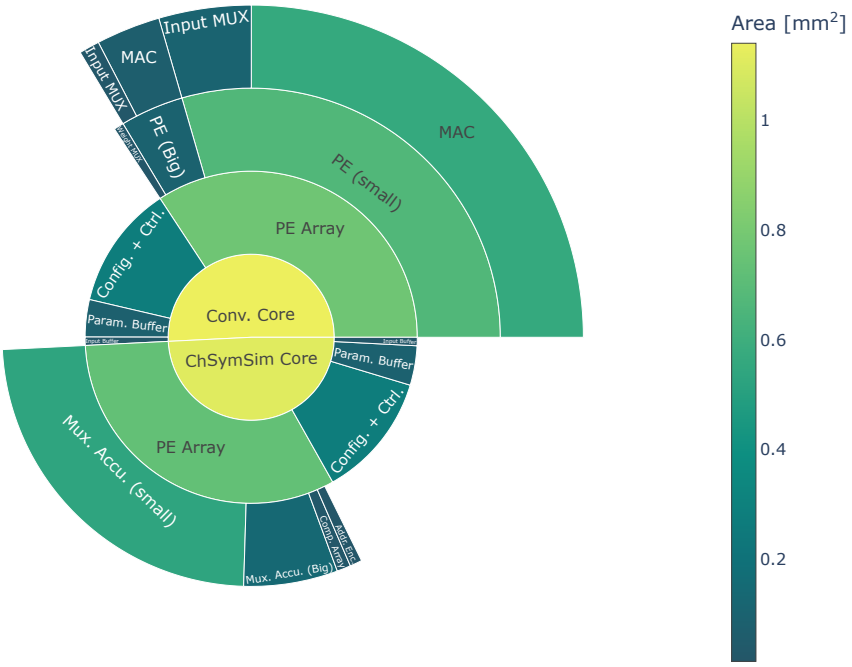


Figura 4.24: Comparación de área (post-síntesis) de los operadores ChSymSim y convolucional, junto a todos los módulos que los componen.

por la fracción del módulo de mayor jerarquía que representan. Un ejemplo de esto son los aceleradores propiamente dichos, que según su valor de área ocupan una parte del círculo central (área total). Un primer detalle a destacar en los resultados de área post-síntesis es que, al contrario a los resultados del modelado de alto nivel con las dimensiones elegidas, el operador ChSymSim posee un área ligeramente inferior al convolucional. Esto se debe a que, además de la reducción de área por *clock-gating* que no se consideró en los modelos de alto nivel, la herramienta de síntesis realiza optimizaciones de área y consumo adicionales. Estas optimizaciones pueden combinar algunas compuertas lógicas en otras equivalentes de la librería de celdas estándar que son más complejas, pero a la vez más pequeñas, que las compuertas originales separadas. Por el contrario, si se observan los reportes de área previos a las optimizaciones (pre-opt.), detallados en la Tabla 4.5, se puede apreciar que el procesador ChSymSim completo es ligeramente más grande que el convolucional, en una relación similar a lo predicho para sus arreglos de PEs en los modelos de alto nivel (Fig. 4.22b con grupo de PEs de tamaño 3×3).

En cuanto a los módulos que componen los aceleradores, tanto en la Fig. 4.24

Módulo	ChSymSim	Conv.
Total (pre-opt.)	1,5977	1,5076
Total (post-opt.)	1,1048	1,1401
Arreglo de PEs	0,7269	0,7688
Configuración y control	0,2734	0,2717
Buffer de parámetros	0,0862	0,0813
Buffer de entradas	0,0183	0,0183

Tabla 4.5: Resultados de área (en mm^2) de los diseños post-síntesis para los aceleradores ChSymSim y convolucional y sus bloques principales.

como en los detalles mostrados en la Tabla 4.5 se puede observar que el bloque que mayor área presenta es el arreglo de PEs, representando alrededor de un tercio del área total de los procesadores ChSymSim y convolucional. Como era de esperarse, el área del banco de memoria (*buffer*) de entradas es idéntica en ambos casos. En cuanto a los parámetros, el operador ChSymSim es mas grande ya que requiere de pesos adicionales (8 más que en el operador convolucional) y además tiene los registros para SE y la máscara del *kernel* (SM). Los bloques de configuración y control resultan ligeramente más grandes en el procesador ChSymSim, dado que este requiere de algunos bloques adicionales (como el generador de rampa) y algunos registros de configuración extra para señales que no están presentes en el acelerador convolucional, aunque dicha diferencia es casi imperceptible (menos del 1 % de su área relativa).

Por otra parte, en la Fig. 4.25 se muestran los resultados de potencia obtenidos en simulaciones pos-síntesis de ambos aceleradores ChSymSim y convolucional, realizadas con una frecuencia de reloj de 25 MHz (mitad de la frecuencia límite) para asegurar un correcto funcionamiento de los operadores. Para estas simulaciones se utilizaron datos aleatorios durante la ejecución de los bloques de convolución y ChSymSim (llenando las 11×11 entradas de ambos aceleradores) y con 32 canales de entrada (32 ciclos de procesamiento). Debido a complicaciones en los modelos de simulación de celdas de *clock gating*, se ignoraron los retardos temporales de las librerías de celdas estándar, por lo que se obtiene una actividad "ideal" de los nodos de los circuitos (*netlist*) sintetizados (no se presentan *glitches*). Si bien las curvas presentadas en la Fig. 4.25 corresponden la potencia de los aceleradores completos, incluyendo *buffers* de entradas/parámetros y lógica de control, los resultados para

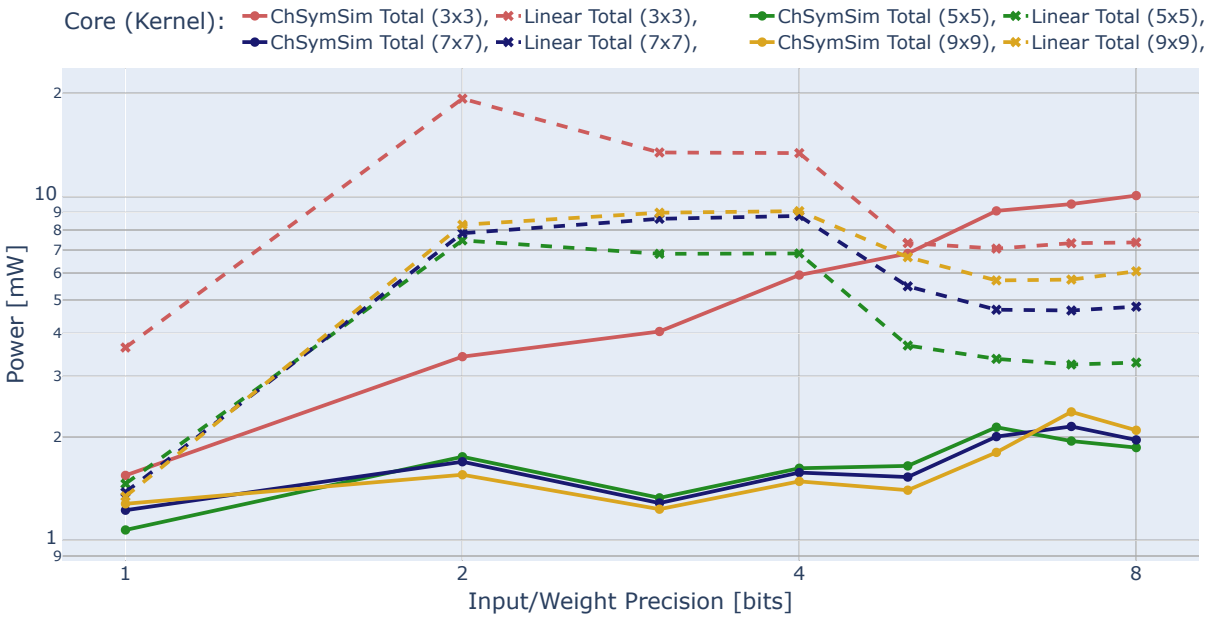
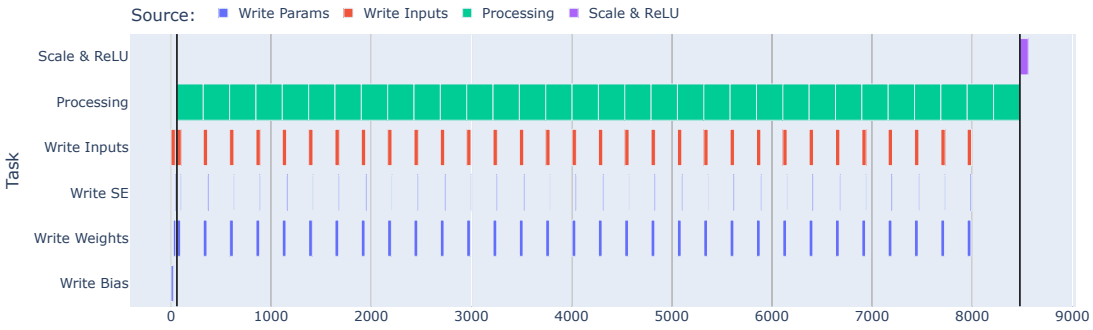


Figura 4.25: Resultados de consumo de potencia total en los aceleradores ChSymSim y convolucional (lineal), extraídos de la actividad ("switching activity") de simulaciones post-síntesis ideales.

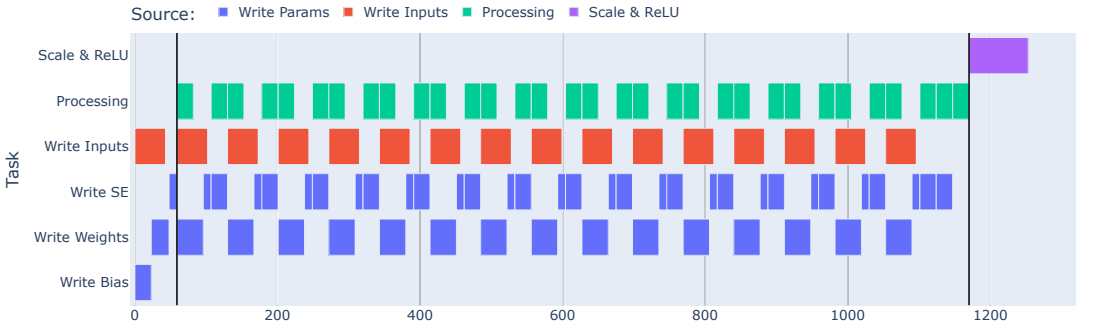
únicamente los arreglos de PEs (junto a los comparadores en el caso ChSymSim) es casi idéntico (salvo un factor de escala).

Al diferencia de los modelos de alto nivel presentados en la Sección 4.2, las simulaciones post-síntesis tienen en cuenta los tiempos de carga de datos y ciclos de espera adicionales en los controladores de los aceleradores, por lo que el comportamiento del consumo de potencia presentado en la Fig. 4.25 difiere de lo estimado en dicha sección. Para tener una mejor comprensión de por qué ocurre esta diferencia entre potencia estimada y "real", es necesario observar con más detalle el *pipeline* entre el procesamiento y la carga de datos en el acelerador. Con esto en mente, en la Fig. 4.26 se muestran algunos ejemplos con diagramas temporales del cómputo completo de los 32 canales de entrada (con *kernel* de 3×3) por parte del procesador ChSymSim, incluyendo carga inicial de datos, junto al escalado, redondeo y activación ReLU de los resultados.

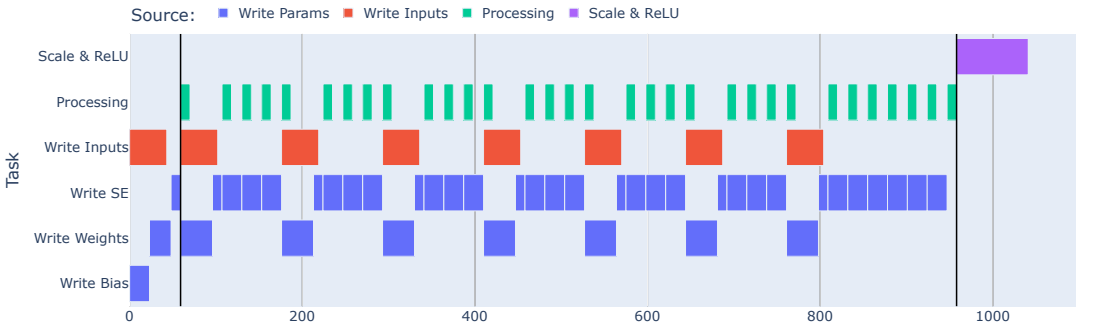
Teniendo una cuantización de entradas de 8 bits (Fig. 4.26a), el procesamiento simplicial domina por completo al tiempo de procesamiento total, por lo que el factor de actividad α que multiplica al consumo de potencia dinámico de las compuertas lógicas en la Ec. (4.17), es básicamente 1 para los PEs. Sin embargo, a medida que



(a)



(b)



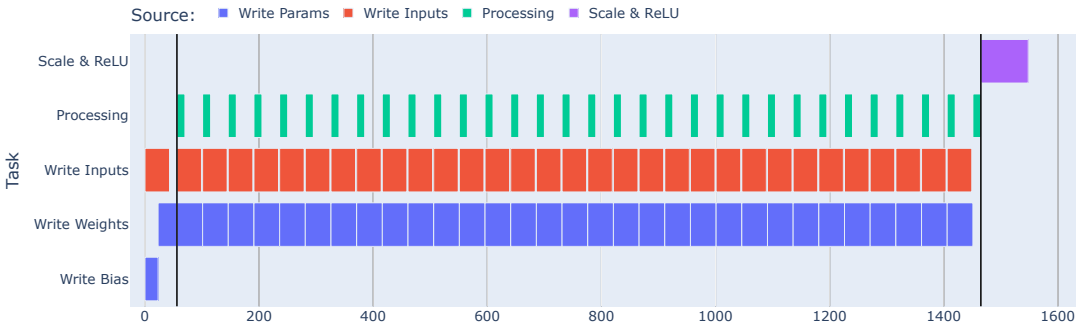
(c)

Figura 4.26: Tiempo de cómputo total (incluyendo carga de datos con DMA) para el operador ChSymSim con *kernel* 3×3, 32 canales, 8 *features* de salida y precisión de entrada/pesos de: a) 8 bits; b) 4 bits; c) 2 bits.

los bits de precisión de entrada se reducen (Figs. 4.26b y 4.26c), los tiempos de transferencia de datos (especialmente la carga de parámetros) comienzan a dominar el tiempo de procesamiento. Debido a esto, el arreglo de PEs presenta tiempos "muertos" (prácticamente sin actividad) en los que el acelerador se encuentra a la espera de nuevos datos para poder iniciar la siguiente iteración del cómputo de la función ChSymSim. Esta ausencia de actividad en los circuitos de los PEs implica una reducción de potencia que las estimaciones "ideales" desarrolladas en la Sección 4.2 no tuvieron en cuenta, explicando así el comportamiento de las curvas de potencia del acelerador ChSymSim, observados en la Figs. 4.25.

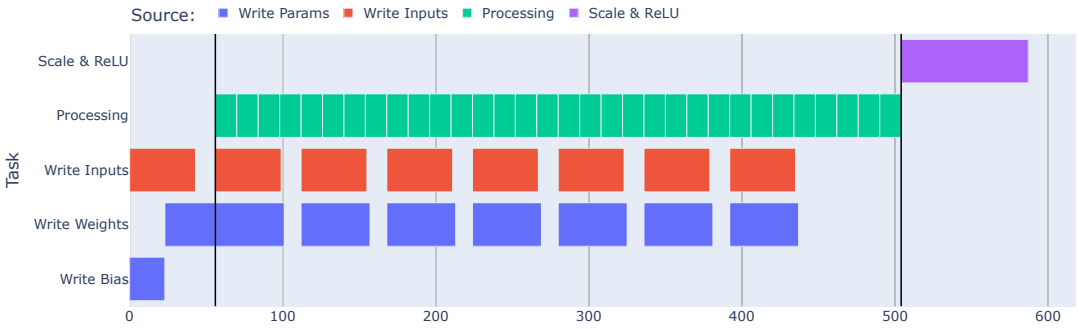
Un efecto similar ocurre con el cómputo del acelerador convolucional, donde a pesar de que el tiempo de procesamiento no esté definido por la precisión de la entrada como sucede con la arquitectura ChSymSim, también se presentan tiempos de espera cuando la carga de entradas o parámetros supera al tiempo de cómputo. Para ilustrar lo anteriormente mencionado, la Fig. 4.27 presenta los diagramas de tiempo para el procesamiento de los 32 canales con el operador convolucional y *kernel* de tamaño 3×3, mostrando los casos con precisiones de entradas y pesos de 8, 4 y 2 bits. Para 8 bits de entrada/pesos (Fig. 4.27a), los tiempos de procesamiento resultan considerablemente menores en comparación a cómo escala la escritura de parámetros al tener que cargar pesos para todos los *features* de salida. Nuevamente, esta diferencia en los tiempos de procesamiento y carga de datos produce tiempos de espera en el arreglo de PEs que reducen la actividad. Esto deja de ocurrir a partir de que el número de canales comprimidos (en un byte) permite los suficientes ciclos de cómputo, tales que la suma de sus tiempos de procesamiento supere a los de carga de datos (Fig. 4.27c), teniendo así actividad constante en los PEs hasta finalizar la operación.

Para considerar el efecto de estos tiempos "muertos" en los consumos de potencia de los aceleradores, resulta conveniente computar un factor de actividad global en los arreglos de PEs, dado por las condiciones del sistema y la configuración de capa a computar. Dicho factor de actividad, mostrado en la Fig. 4.28, se obtiene calculando la relación de tiempo de cómputo/procesamiento de los aceleradores sobre el tiempo total del bloque (acumulación de canales procesados). El procesador convolucional presenta disminuciones en la actividad (y por lo tanto en la potencia) para 3-4 bits



(a)

(b)



(c)

Figura 4.27: Tiempo de cómputo total (incluyendo carga de datos con DMA) para el acelerador convolucional con *kernel* 3×3, 32 canales, 8 "features" de salida y precisión de entrada/pesos de: a) 8 bits; b) 4 bits; c) 2 bits.

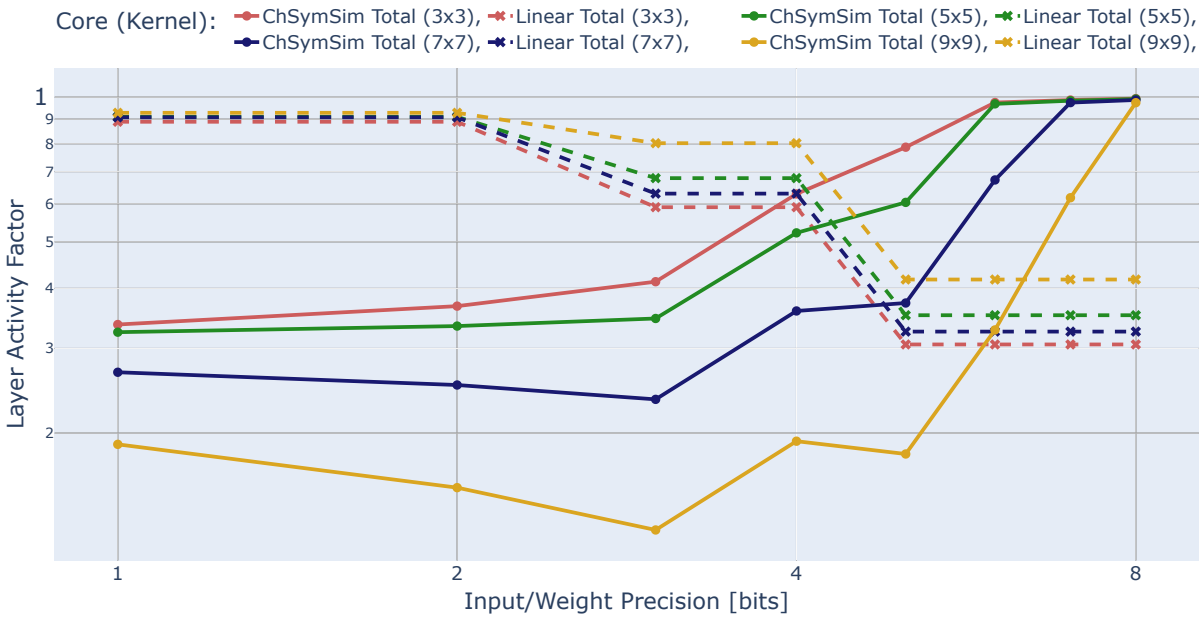


Figura 4.28: Actividad total de los aceleradores ChSymSim y convolucional, calculada a partir de los ciclos de procesamiento y espera de datos (con 32 canales de entrada).

de precisión (2 canales por byte), llegando a un mínimo de actividad para 5-8 bits (1 canal por byte). Como se dijo anteriormente, para el caso convolucional, a mayor cantidad de canales por byte más ciclos de procesamiento (de duración fija según tamaño de *kernel*) se pueden realizar sin necesidad de más datos. Es por esto que con 1-2 bits de precisión, la serie de ciclos de procesamiento es superior a una carga de datos (pesos y/o entradas). Sin embargo, para el caso específico de 1 bit de entradas y pesos, el consumo de potencia mostrado en las Figs. 4.25 disminuye a pesar de que el factor de actividad global se mantenga en su máximo (igual a 1). Esto se debe a que al ser datos *unsigned* y carecer de extensión de signo en los multiplicadores y acumuladores, una buena porción del *hardware* queda fija en 0 y se elimina así la actividad de tales nodos.

Tal y como se predijo al observar los diagramas de tiempo de la Fig. 4.26, para el operador ChSymSim los factores de actividad globales máximos se dan con mayores bits de precisión. Es a partir de que la precisión de las entradas produce tiempos de procesamiento inferiores a la carga de datos (según el tamaño del *kernel*), que la actividad disminuye al quedarse el acelerador ChSymSim a la espera de nuevos datos que procesar. Este comportamiento casi inverso a la actividad del operador

convolucional se debe no solamente a que el tiempo de cómputo simplicial varía según la precisión de las entradas, sino que además es influenciado por la escritura del SE. Al requerirse de un SE por cada canal y este no presentar compresión alguna (a diferencia de los pesos simétricos), no es posible encadenar ciclos de procesamiento de canales sucesivos como en el caso convolucional. Cuando la precisión de las entradas se reduce, y como consecuencia también lo hace el tiempo de procesamiento, la carga del SE se vuelve un factor dominante en el tiempo de cómputo total y los tiempos de espera entre procesamientos se incrementan, reduciendo notablemente la actividad de los PEs.

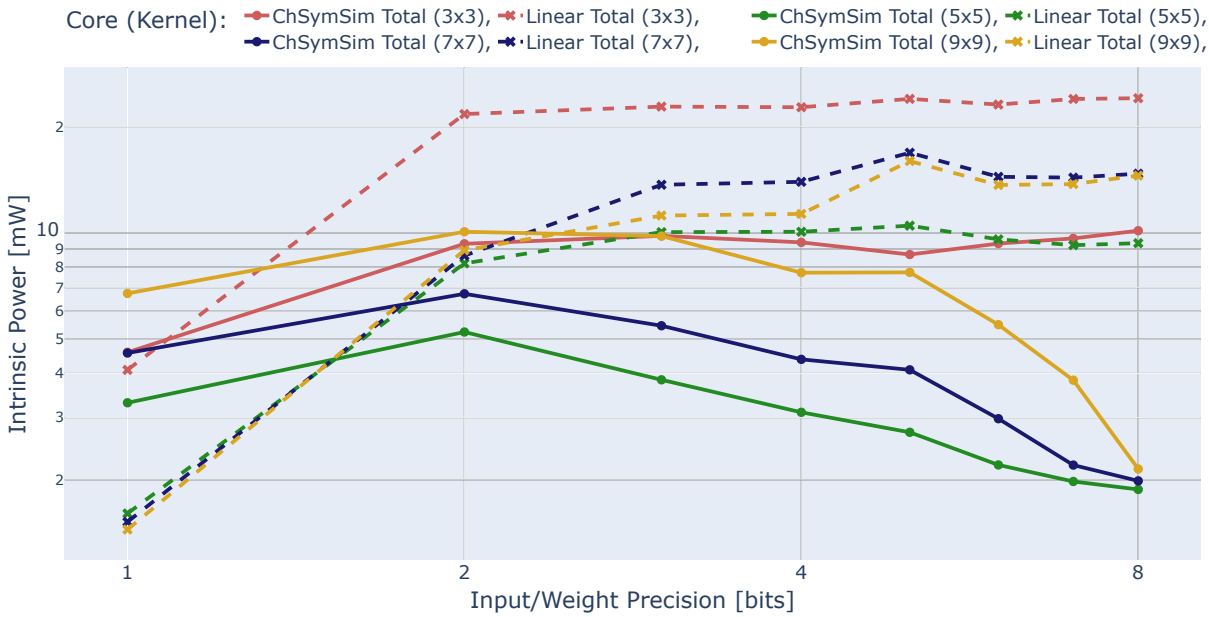


Figura 4.29: Resultados de consumo de potencia intrínseca (ignorando carga de datos y tiempos de espera) de los procesadores ChSymSim y convolucional (simulaciones post-síntesis ideales).

Si se extrae este factor de actividad global (Fig. 4.28) de los resultados de potencia presentados por la Fig. 4.25, se obtiene un consumo de potencia intrínseco de los aceleradores o, en otras palabras, el consumo debido al procesamiento continuo (sin tiempos de espera). Este consumo de potencia intrínseco puede observarse en la Fig. 4.29, donde para el acelerador convolucional se ve reducido al usar 1 bit de entradas/pesos (multiplicaciones y sumas de sólo 1 o 0, sin extensión de signo), mientras que para precisiones de 2 o más bits, el consumo se incrementa poco a poco. En el caso del acelerador ChSymSim, la reducción de consumo para 1 bit es

inferior a la de la convolución, dado que las porciones de *hardware* con nodos fijos en 0 se limitan al acumulador. Por otra parte, para tamaños grandes de *kernel*, el consumo del acelerador se reduce al incrementar la precisión, tal y como se predijo con los modelos de alto nivel. A pesar de presentarse algunas diferencias debido a las optimizaciones realizadas por la herramienta de síntesis, el comportamiento de las curvas de potencia intrínsecos al procesamiento del acelerador ChSymSim, mostrados en la Fig. 4.29, resultan similares a lo observado en las estimaciones en la Sección 4.2.

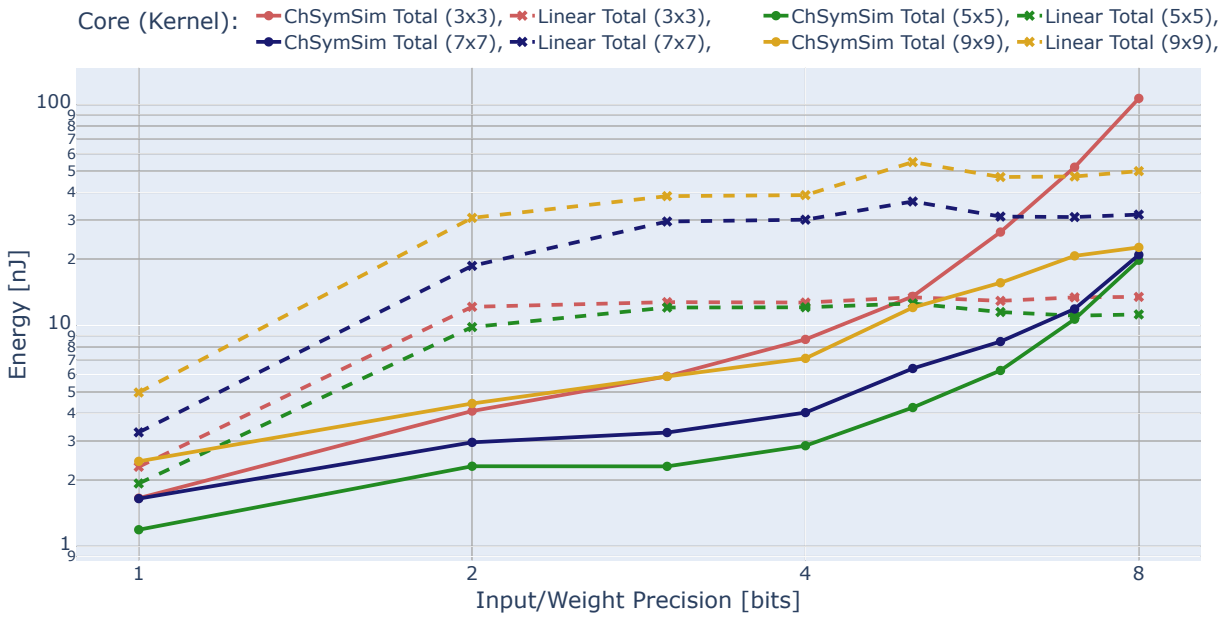


Figura 4.30: Resultados de energía de los aceleradores ChSymSim y convolucional, a partir del consumo de potencia total (simulaciones post-síntesis ideales) y tiempo de cómputo incluyendo ciclos de espera.

Finalmente, en la Fig. 4.30 se muestra una comparación de la energía en los aceleradores, calculada a partir de los datos de potencia hallados en las simulaciones post-síntesis y el tiempo de cómputo total, que incluye los efectos de la carga de datos para procesar. En base a estos resultados de energía más "realistas" se puede concluir que la arquitectura de procesamiento ChSymSim propuesta en esta tesis es efectivamente más eficiente para precisiones de entradas bajas y cómputo de *kernel* grandes. En comparación con el procesador convolucional, el acelerador ChSymSim consume menos energía cuando se computan *kernels* pequeños (como el caso de 3×3) cuando la precisión es igual o inferior a 4 bits, mientras que para *kernels*

grandes se tiene un menor consumo energético para prácticamente todos los valores de precisión de entradas.

4.4. Conclusiones

En este capítulo se introdujo el operador ChSymSim junto a sus bloques básicos, el cual fue diseñado para acelerar las funciones del algoritmo simplicial simétrico mejorado. Se desarrollaron modelos de alto nivel (relativamente ideales) con el objetivo de seleccionar una de las arquitecturas de mayor eficiencia, para luego realizar una síntesis y simulaciones post-síntesis con las cuales verificar el funcionamiento de la arquitectura seleccionada bajo condiciones más "realistas". Asimismo, para el diseño de los bloques que conforman dicho acelerador se planteó como objetivo minimizar la replicación de *hardware* al paralelizar operaciones (aprovechando recursos compartidos), "apagar" módulos no utilizados (al menos con técnicas *clock-gating*) usando señales de habilitación apropiadas, técnicas de *pipeline* para evitar tiempos muertos entre operaciones, compresión de datos para reducir transferencias, etc.

A pesar de su simplicidad, los modelos de alto nivel fueron de gran utilidad para identificar los bloques de *hardware* más demandantes en términos de tiempo, área y consumo, así como las variantes de la arquitectura y las dimensiones más adecuadas para lograr un buen balance entre el procesamiento de Redes Neuronales y las transferencias de datos necesarias para tal fin. Esto permitió que, en combinación con las técnicas anteriormente mencionadas, se pueda optimizar el diseño mediante la aplicación de estrategias de cómputo eficiente, haciendo foco en las partes más relevantes del acelerador. Los resultados de simulación mostraron una eficiencia energética superior para el procesador ChSymSim, en comparación con aceleradores tradicionales como el convolucional, ante tamaños de *kernel* pequeños y precisión baja, o para grandes dimensiones de *kernel* con cualquier precisión típica del cómputo embebido.

Capítulo 5

Implementaciones en circuitos integrados

En este capítulo se muestra la implementación en circuitos integrados (ASIC) del procesador ChSymSim, previamente analizado y optimizado para cómputos de capas en Redes Neuronales. Para esto se diseñaron y fabricaron tres Sistemas en Chip (SoC) como plataformas de evaluación del acelerador, que integran módulos auxiliares para control y transferencia de datos. Los primeros dos *chips* fueron concebidos como prototipos para la verificación de la unidad de procesamiento principal, de manera de depurar y analizar su funcionamiento previo a la implementación final. El tercer *chip* presenta la versión final del acelerador ChSymSim, con los ajustes y optimizaciones correspondientes a lo observado en las versiones prototipo. Además, este último SoC incluye mejoras y ampliaciones en los módulos de transferencia de datos, tales como controladores de DMA e interfaces QSPI[1] múltiples, adaptando el sistema a las necesidades del operador ChSymSim para mejorar su eficiencia. La tecnología seleccionada para la fabricación de los ASIC fue TSMC 65nm LP, debido a su accesibilidad y su buena relación costo-área para la realización de proyectos académicos.

De esta forma, en la Sección 5.1 se presentan las versiones preliminares del acelerador ChSymSim, utilizadas como prueba de concepto a lo largo de los distintos *chips* fabricados durante el transcurso de esta tesis. En la Sección 5.2, en cambio, se

[1] *Quad Serial Peripheral Interface*

detalla la implementación final en ASIC de la arquitectura de procesador ChSymSim descrita en el Capítulo 4, considerando tanto los aspectos de fabricación (tecnología, distribución de potencia y reloj, etc.) como el sistema y módulos con los que interactúa el acelerador durante su operación. Finalmente, en la Sección 5.3 se muestran los resultados experimentales en cuanto a funcionalidad y desempeño de los SoCs fabricados, haciendo foco en la implementación final del acelerador y su operación con diferentes configuraciones de capa ChSymSim.

5.1. Versiones preliminares

Dada la complejidad que representa la fabricación de circuitos integrados, antes de poder implementar el diseño completo del procesador ChSymSim propuesto en esta tesis, es necesario realizar antes algunos prototipos y versiones preliminares. Estos prototipos ayudan, en una primera instancia, al desarrollo de un flujo de diseño digital con el que diseñan, evalúan y posteriormente se fabrican los SoCs desarrollados. De esta forma, se puede "poner a punto" y generar *scripts* para las herramientas de síntesis y P&R[2], a la vez que se conectan con las herramientas de simulación para realizar evaluaciones en todas las etapas del diseño. Por otra parte, las pruebas realizadas en estos prototipos permiten analizar y mejorar el comportamiento del acelerador ChSymSim ante interacciones con sistemas más complejos, en los que se incluyen bloques de control y transferencia de datos, que ante simulaciones del operador "aislado" no se tienen en cuenta.

En la Sub-Sección 5.1.1 se presenta la primera iteración de SoC prototipo, denominada DigineuronV1, en la que se integra y se evalúa una versión preliminar del procesador ChSymSim con una sola unidad de procesamiento. Por otra parte, en la Sub-Sección 5.1.2 se detalla la segunda versión del SoC prototipo (DigineuronV2) que, además de reemplazar el microprocesador por otro de uso libre e incluir un controlador de DMA para transferencias de datos más eficientes, contiene un acelerador ChSymSim con diversas mejoras respecto de su variante en DigineuronV1. Tanto el SoC como el procesador ChSymSim presentados en la Sub-Sección 5.1.2 sientan las bases para el sistema integrado en el *chip* final de esta tesis.

5.1.1. DigineuronV1

DigineuronV1 [43] es la primera iteración de un SoC fabricado durante el transcurso de esta tesis, que sirve como una plataforma de prueba para desarrollar y evaluar aceleradores de Redes Neuronales de bajo consumo. El objetivo de este diseño es producir una plataforma flexible compuesta por infraestructura de control y comunicación (procesador, *bus* interno, memoria, interfaces de entrada/salida, etc.), donde se puedan insertar fácilmente diferentes bloques de procesamiento para cómputos

[2] *Place and Route*

de Redes Neuronales. Este primer circuito integrado, de tamaño $1{,}25mm \times 1{,}25mm$ y mostrado en la Fig. 5.1, fue fabricado con los esfuerzos conjuntos del grupo de investigación utilizando la tecnología TSMC $65nm$, a través del programa *mini@sic* de Europractice [65].

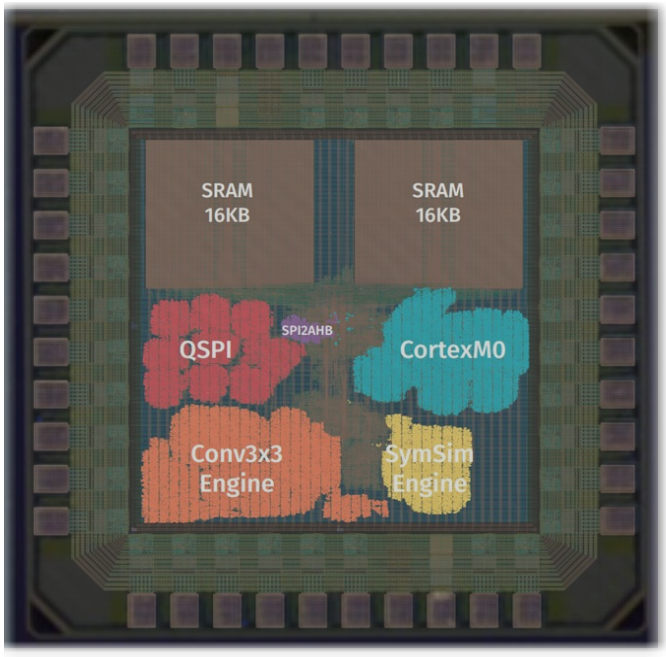


Figura 5.1: DigineuronV1 SoC, *layout* superpuesto sobre el *chip* fabricado, marcando los bloques más importantes del mismo.

Para la implementación del *chip* DigineuronV1, se diseñó un sistema basado en Arm® AMBA[3]® (específicamente AHB[4]-Lite), el cual se muestra en la Fig. 5.2. Este sistema está compuesto principalmente por un procesador Arm® Cortex® M0 como unidad de control principal, una interfaz SPI2AHB que también funciona como maestro en el sistema (para carga de programas y *debug*), registros de control y dos bancos de SRAM de $16KB$ (programa y datos). También se cuenta con un bloque Quad SPI y puertos GPIO[5] para transferencias de datos fuera del *chip*. La decisión de utilizar un *bus* de la clase AHB (de 32 bits) radica en que este proporciona una comunicación de gran ancho de banda entre los distintos componentes del sistema, permitiendo transferencias de datos eficientes (*pipeline*) y simplificando la integración de varios bloques de procesamiento y periféricos.

[3] *Advanced Microcontroller Bus Architecture*
[4] *Advanced High-performance Bus*
[5] *General Purpose Input/Output*

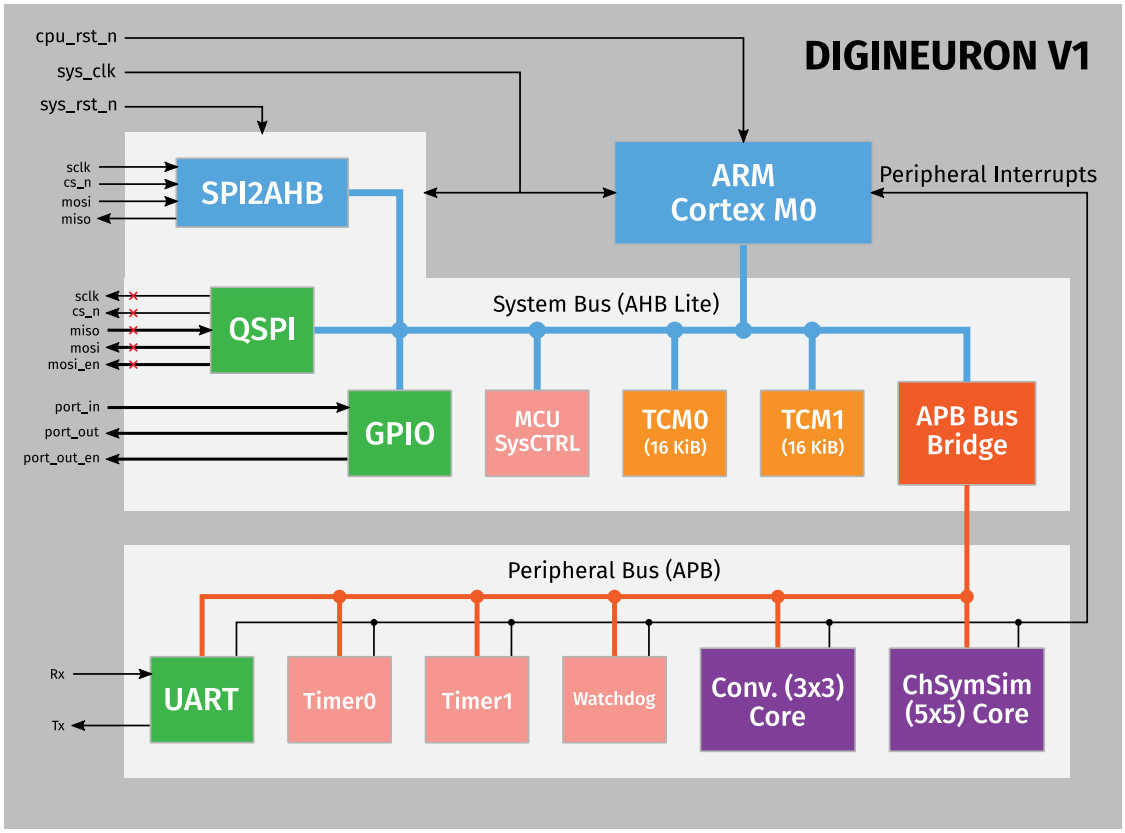


Figura 5.2: Diagrama en bloques con los módulos de DigineuronV1 SoC.

Por otro lado, si bien el *bus* dedicado para periféricos (APB[6], también de 32 bits) es más lento, su simplicidad ofrece una "barrera de seguridad" adicional. Esto se debe a que, incluso si los bloques conectados a este no están bien implementados, se pueden evitar ciertos tipos de errores potenciales o condiciones de carrera (*race conditions*) relacionados con transferencias de datos de alta velocidad y arbitrajes complejos. Es por esto que los aceleradores de prueba (ChSymSim y convolucional estándar) fueron conectados por medio de la interfaz APB, de manera que no afecten al resto del sistema en caso de presentar fallos. Conectados mediante APB, se encuentran también otros módulos con bajos requerimentos de velocidad de transferencia de datos, tales como interfaz UART[7], temporizadores y *watchdog*.

El procesador ChSymSim implementado como prueba de concepto en este *chip* se muestra en la Fig. 5.3. Este operador consiste en una serie de registros de configuración y señales de control (*flags*), junto con un controlador dedicado y un generador

[6] *Advanced Peripheral Bus*
[7] *Universal Asynchronous Receiver Transmitter*

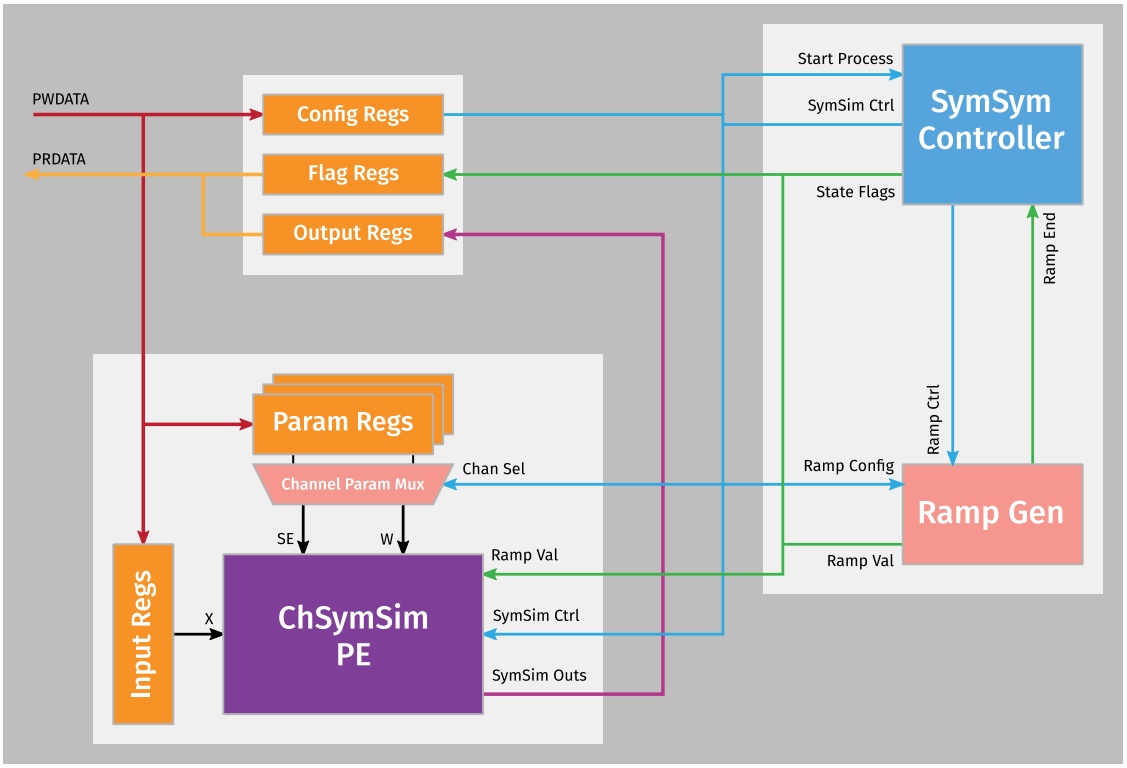


Figura 5.3: Diagrama en bloques del acelerador ChSymSim implementado en DigiNeuronV1.

de rampa. Dichos registros de configuración contienen información relevante para la operación, como por ejemplo qué parámetros de canal deben seleccionarse para la operación, los valores de inicio, fin e incremento/decremento para la rampa digital, y los valores de reinicio manual y habilitaciones de registros internos. Todas las señales que salen del controlador y los valores de la rampa se escriben en los registros de control (*flags*) durante el cómputo del canal de la función ChSymSim. Estos datos pueden ser leídos por el microprocesador para informar al sistema cuándo leer las salidas y/o comenzar a procesar el siguiente canal, o bien para *debug* del ciclo de procesamiento del operador. Como bloque principal, este acelerador cuenta con un único PE capaz de computar filtros ChSymSim de tamaño hasta 5×5, que a su vez es alimentado por 25 registros *buffer* de entrada (X) de 8 bits y 3 canales de registros de parámetros que contienen SE binarios y 26 pesos W (también de 8 bits). El *buffer* de parámetros para almacenar hasta 3 canales se diseñó considerando el cómputo de la primera capa de una CNN, que generalmente consiste en entradas RGB. Sin embargo, en base al análisis realizado en el Capítulo 4, esto se descartó

para futuras implementaciones. Los resultados obtenidos por el PE se conectan finalmente a registros mapeados en memoria para su extracción por medio del *bus* APB.

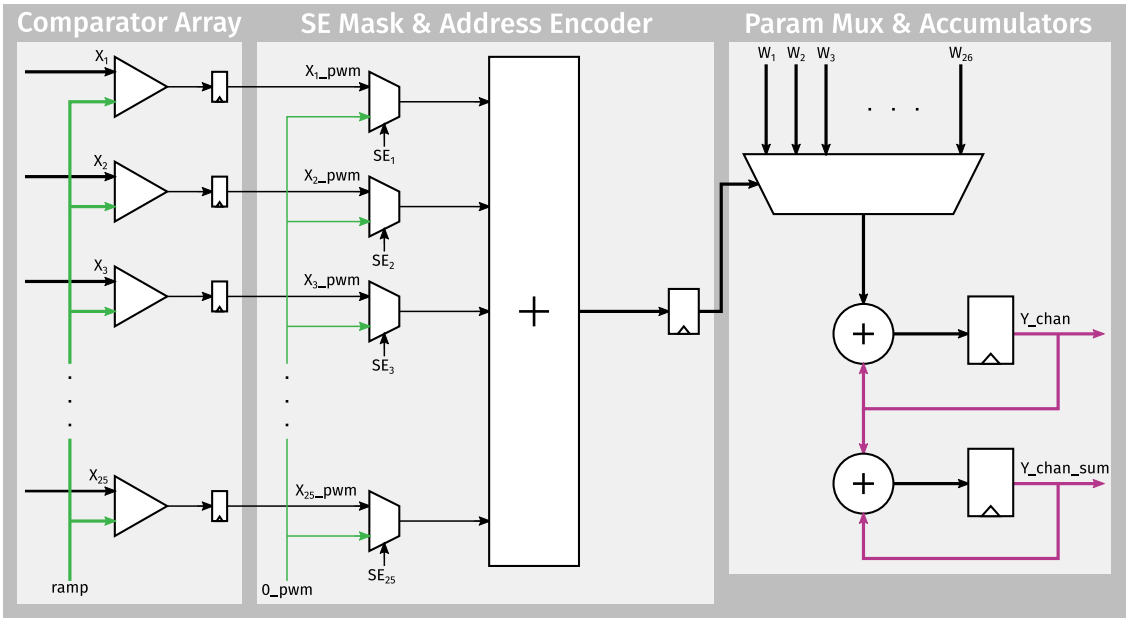


Figura 5.4: Diagrama en bloques del PE perteneciente al operador ChSymSim implementado en DigineuronV1.

El elemento de procesamiento diseñado para este *chip* se muestra en la Fig. 5.4 y su funcionamiento es similar al PE descrito en la Sección 4.1. En este PE, las entradas de 8 bits (X_1 a X_{25}) a ser comparadas con la rampa se consideran siempre signadas, y no se cuenta con máscaras o *bit-shift* aritméticos para manejar la precisión de las mismas. De esta forma, la precisión puede alterarse igualmente con los valores inicial y final de la rampa, pero para evitar errores es necesario realizar las extensiones de signo correspondientes previas a cargar los datos de entrada en el acelerador. A diferencia de la implementación del *Address Encoder* propuesta en el Capítulo 4, el bloque de generación de dirección de pesos en el PE mostrado en la Fig. 5.4 no cuenta con Máscara de Forma para el *kernel* (*Shape Mask*). Debido a esto, para poder ejecutar *kernels* de tamaños menores a 5×5 es necesario que se agregue *padding* de 0s al cargarse las entradas en el acelerador. Otra alternativa es usar el valor de *reset* de los registros y llenarlos de manera adecuada, evitando cargar datos en las posiciones no ocupadas por el *kernel*. Sin embargo, en cualquiera de estos dos casos se puede incurrir en tiempos de carga superiores, reduciendo así la

eficiencia del *pipeline*. Es por esto que en las siguientes versiones se decidió utilizar esta máscara para cancelar las entradas innecesarias en *kernels* más pequeños que el tamaño nativo del PE. En esta versión de PE se utilizaron dos acumuladores: uno para acumular los coeficientes seleccionados y otro para sumar los canales ya procesados. Esto se hizo a modo de verificación, ya que de esta forma se pueden extraer también cada canal Y_chan (16 bits) procesado de forma independiente, en lugar de solo el resultado final (o sumas parciales) Y_chan_sum, de 18 bits.

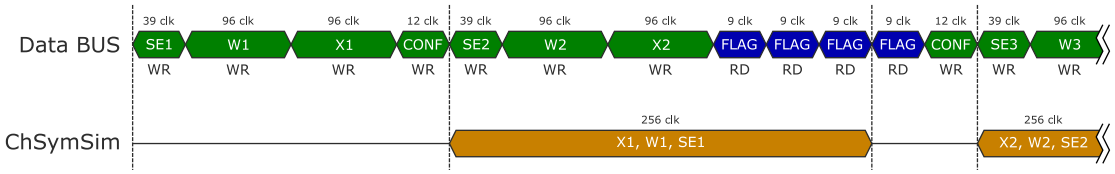


Figura 5.5: *Pipeline* de procesamiento del acelerador ChSymSim integrado en DigineuronV1.

En la Fig. 5.5 se ilustra la operación en *pipeline* del acelerador ChSymSim implementado en DigineuronV1. Antes de que comience la operación, los registros de configuración, las entradas y los parámetros para procesar el primer canal, deben ser previamente establecidos. Una vez que las entradas y los parámetros del canal a procesar (SE y pesos) se encuentran escritos en los registros internos del acelerador, se escribe uno de los registros de configuración para dar inicio al procesamiento, provocando que el controlador escanee las entradas y los pesos de sus respectivos *buffers*, reinicie y habilite el generador de rampa junto a los registros dentro del PE. Luego de iniciado el procesamiento, el controlador espera a que la rampa alcance el valor final para registrar y acumular la salida del canal. Debido a que esta implementación del acelerador ChSymSim ya cuenta con registros *buffer* para entradas y pesos, la siguiente iteración de entradas y parámetros puede cargarse a la vez que se realiza el procesamiento, aunque al contarse con un solo *bus*, tanto entradas como parámetros deben ser escritos en serie. Este *pipeline* entre procesamiento y carga de datos, junto con el hecho de que ambas duraciones resultan similares (Fig 5.5), constituyen los primeros indicadores de que la presencia de *buffers* para varios canales de entrada es innecesaria. Después de que la rampa finaliza, el controlador habilita los registros de salida para escribir tanto el resultado de procesar el canal actual como la suma parcial (o total) de canales procesados.

Si bien DigineuronV1 fue diseñado principalmente como entorno de pruebas y para generar un flujo inicial de diseño digital para la fabricación de circuitos integrados, el haber incorporado un pequeño operador ChSymSim resultó de gran utilidad para esta tesis. A pesar de ser una versión temprana, la posibilidad de evaluar el diseño del acelerador permitió identificar los errores de diseño en la interfaz QSPI y en el acelerador ChSymSim que se presentan al interactuar con un sistema real. Esto ayudó al análisis y desarrollo de optimizaciones, así como mejoras en funcionalidad, que se implementaron en las versiones posteriores.

5.1.2. DigineuronV2

DigineuronV2 [44] es un SoC diseñado como una plataforma para prototipar y probar diferentes aceleradores de Redes Neuronales. Al igual que su iteración anterior (Sub-sección 5.1.1), este *chip* ha sido fabricado utilizando la tecnología TSMC de 65nm (*mini@sic* program [65]), con un tamaño de 1,25mm ×1,25mm, y puede funcionar con una frecuencia de reloj de hasta 100MHz, con una tensión de alimentación de 1,2V. En la Fig. 5.6 se puede observar el *layout* de DigineuronV2 con sus bloques más relevantes, superpuesto a una fotografía del ASIC fabricado.

En cuanto a control del sistema AHB, este SoC presenta dos diferencias fundamentales en relación a su versión anterior (DigineuronV1): reemplazo del microprocesador y adición de controlador de Acceso Directo a Memoria (DMA). Para reducir el uso de bloques IP[8] pertenecientes a ARM® , se reemplaza al Cortex® M0 por un μP[9] de uso libre denominado CV32E40P, perteneciente a OpenHW Group [66]. Este nuevo procesador está basado en la arquitectura (ISA[10]) RISC-V RV32IMC, presenta un *pipeline* de 4 etapas y cuenta con dos puertos de 32 bits: uno para obtener las instrucciones desde SRAM y el otro para operaciones de lectura/escritura de datos. Debido a que estos dos puertos utilizan el *bus* OBI[11], para comunicarse con el resto del sistema se debieron utilizar módulos de interfaz que traduzcan las transacciones OBI de 32 bits a AMBA® AHB de 64 bits. Por otra parte, el controlador de DMA personalizado, permite el acceso directo y transferencias más rápidas

[8] Propiedad intelectual - *Intellectual Property*
[9] Microprocesador
[10] *Instruction Set Architecture*
[11] *Open Bus Interface*

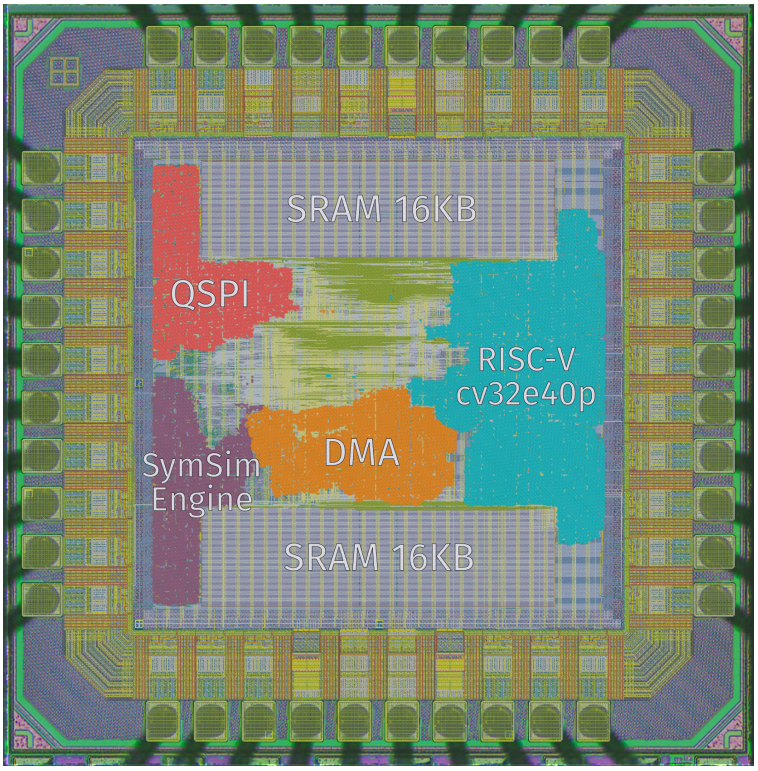


Figura 5.6: DigineuronV2 SoC, *layout* superpuesto sobre el *die*, marcando los bloques más importantes del mismo.

que con el μP. Este controlador cuenta con dos interfaces AHB, una para lectura y otra para escritura, con las que se pueden enviar datos a otros bloques del sistema a la vez que se leen los siguientes valores a transferir. Para realizar las transferencias de datos, este DMA hace uso de configuraciones alojadas en SRAM, con pasos y direcciones configurables en $1D$, $2D$ o $3D$.

Como se puede ver en el diagrama de bloques de la Fig. 5.7, este SoC consiste principalmente en un *bus* AMBA® AHB-Lite de 64 bits para los módulos que requieren de transferencias de datos rápidas y un *bus* AMBA® APB de 32 bits para los periféricos e interfaces más lentos, con un módulo intermediario o "puente" que hace la traducción de las transacciones de un *bus* a otro. En el *bus* principal del sistema (AHB-Lite) se encuentran dos bancos de memoria SRAM (TCM0 y TCM1) de 16KB cada uno, así como registros de control/configuración general del sistema (MCU SysCTRL). Dichos registros de control general almacenan los pesos/prioridades para los árbitros en la matriz AHB, datos de *debug* de las SRAMs (BIST[12]),

[12] *Built-in Self-Test*

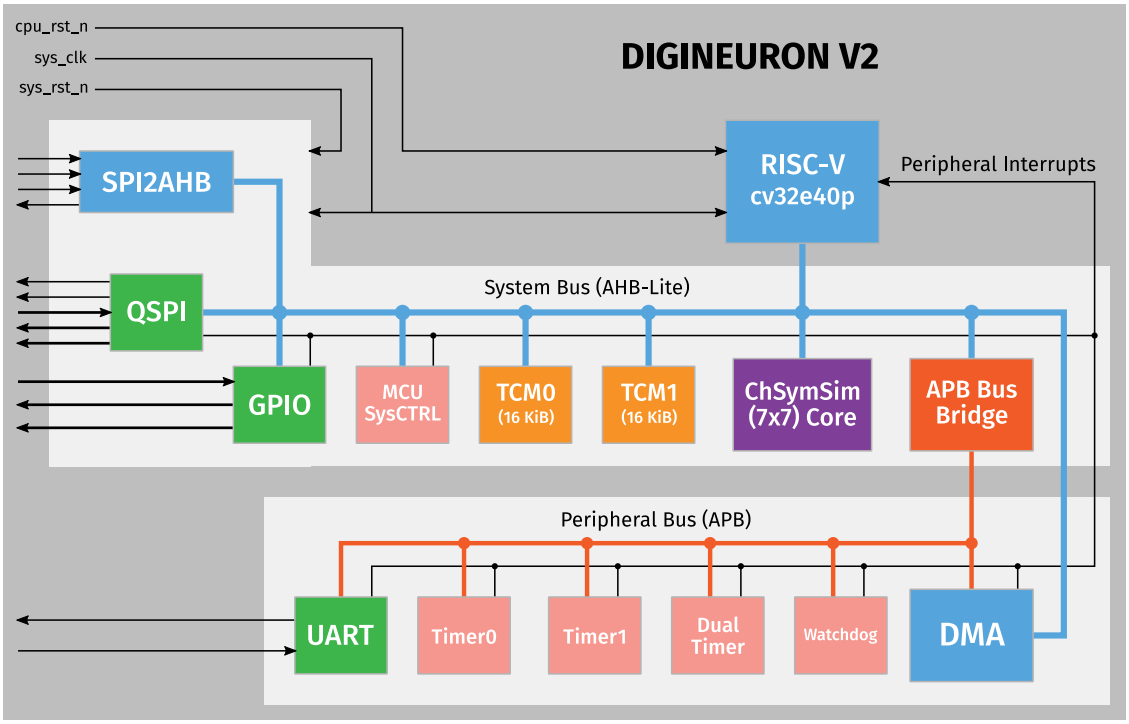


Figura 5.7: Diagrama en bloques con los módulos de DigineuronV2 SoC.

la dirección (puntero) de inicio del microprocesador, máscaras de interrupción, etc. DigineuronV2 cuenta con un solo acelerador prototipo para Redes Neuronales, que corresponde a una versión actualizada del operador ChSymSim de un solo PE, trasladado al *bus* AHB para permitir transferencias de datos más rápidas con un consumo de energía similar, mejorando así su eficiencia. Como interfaces con el "exterior", este SoC dispone de dos módulos conectados al *bus* AHB: un puerto GPIO de 16 bits y una unidad QSPI (con algunas correcciones respecto a la fabricada en DigineuronV1). Finalmente, los periféricos que requieren transferencias de datos menos frecuentes, como los bloques UART, *watchdog* y temporizadores, se encuentran conectados al *bus* APB. Dado que la configuración general del DMA no requiere de mucha velocidad, el módulo completo se encuentra en APB, mientras que sus puertos (*master*) para manejo de transacciones se encuentran conectados directamente a la matriz AHB.

La versión del acelerador ChSymSim implementada en DigineuronV2 amplía el *kernel* que el PE es capaz de procesar, computando hasta un tamaño máximo de 7×7 entradas. El PE en este caso es prácticamente el mismo que en el diseño final, contando con una unidad de *Address Encoder* igual a la presentada en el Capítu-

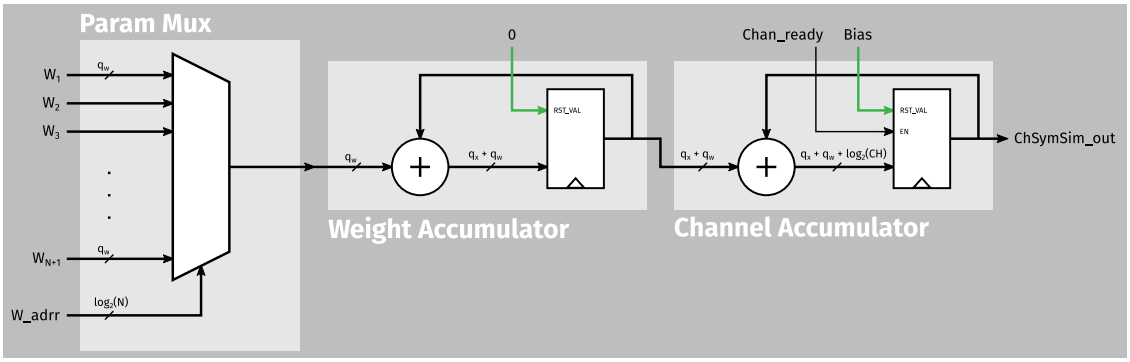


Figura 5.8: Sub-bloque *Mux. Accumulator* del PE en el acelerador ChSymSim fabricado en DigineuronV2.

lo 4, haciendo uso de la Máscara de Forma del kernel (*Shape Mask* o ***SM***) para implementar fácilmente *kernels* más pequeños. Respecto al bloque *Mux. Accumulator*, se mantiene el de DigineuronV1 (Sub-sección 5.1.1), contando con dos etapas de acumulación con motivos de validación y *debug*. La incorporación de un *bias* es otra de las mejoras con respecto a la versión anterior, y se implementa simplemente reiniciando el registro del acumulador de canales (*Channel Accumulator*), tal y como se muestra en la Fig. 5.8. En particular, el operador ChSymSim fabricado en DigineuronV1 cuenta con un error por el cual la última adición del peso por canal no ocurre, lo que debe solucionarse mediante *software*. En la implementación de DigineuronV2, este problema fue resuelto al prolongar la habilitación del registro de acumulación por un ciclo de reloj adicional.

Esta segunda iteración resultó de suma utilidad para pulir el flujo de diseño digital, a la vez que sirvió para mejorar el diseño de la plataforma de pruebas para aceleradores de Redes Neuronales, al añadir otros bloques para incrementar la eficiencia en cuanto a transferencia de datos. Por el lado del operador ChSymSim, este *chip* permitió realizar y verificar un prototipo de PE que, al arreglar el problema de la suma del último peso por canal y añadir mejoras/funcionalidades, realiza efectivamente la misma operación que su contraparte en *software*. Esto hace posible el integrar un acelerador para las funciones ChSymSim mediante un arreglo mucho más avanzado de PEs, capaces de computar varias salidas en paralelo y hacer un mejor uso de los *buses* y módulos de transferencia de datos.

5.2. Versión final: DigineuronV3a

DigineuronV3a es el SoC que concluye esta tesis, en el cual se integraron todos los conocimientos adquiridos al evaluar los prototipos previamente fabricados, así como todas las técnicas de diseño, optimizaciones y mejoras desarrolladas en el Capítulo 4. La principal diferencia con los prototipos fabricados anteriormente (DigineuronV1 y V2) radica en la mayor complejidad tanto en cuanto al procesador ChSymSim como en el sistema en general. En el caso del acelerador ChSymSim, este ya no solo consiste en una unidad de procesamiento básica con control y registros, sino que se ha expandido a un arreglo con varios PEs interconectados para una mayor paralelización y eficiencia de cómputo. Esta nueva versión de procesador ChSymSim incluye unidades de escalado, redondeo y ReLU con controlador dedicado que, en conjunto con el multiplexor de salida, permite la implementación de *stride* y *padding* así como una fácil extracción y reutilización de los resultados del acelerador. En cuanto al sistema presente en este *chip*, los módulos dedicados a la transferencia de datos, tales como el controlador de DMA y QSPI, fueron expandidos en funcionalidad para satisfacer las necesidades del operador ChSymSim. Con el fin de establecer un *pipeline* entre procesamiento de entradas y lectura de salidas, DigineuronV3a integra dos unidades de cada uno de los módulos mencionados, en lugar de solo una como en los *chips* anteriores. A continuación se describirá el SoC DigineuronV3a tanto en sus aspectos de fabricación como en el sistema que contiene, sus módulos e interacciones con la versión final del procesador ChSymSim.

Para la fabricación de este ASIC se utilizó la misma tecnología que con los *chips* prototipo, es decir TSMC 65nm, lo que permite centrarse casi por completo en el diseño del acelerador ChSymSim y en el sistema que lo rodea. Esto se debe a que el flujo digital ya fue diseñado y probado en el transcurso del trabajo para esta tesis, por medio de la fabricación de los dos SoC prototipo, por lo que se debieron realizar muy pocas modificaciones para adaptarlo a este diseño en particular. Tanto el *layout* de DigineuronV3a como el *chip* en sí mismo pueden observarse en la Fig. 5.9, donde se puede observar un aspecto diferente al de las dos iteraciones previas (Sección 5.1). Esto se debe a que por la cantidad de bloques y tamaños de los mismos, se requirió de $9mm^2$ de área total, pero debido a limitaciones dadas por el fabricante,

fue necesario dimensionar el *chip* con un tamaño de 2mm ×4,5mm, siendo 2mm el alto máximo disponible. Esto limitó considerablemente la frecuencia máxima de operación del ASIC, ya que muchos bloques o celdas han debido ser ubicados en extremos alejados y los retardos/latencias debido a esto hacen que el proceso de P&R para cerrar correctamente tiempos de *setup* sea más complejo. Con el fin de aliviar los análisis y optimizaciones de tiempo en algunos circuitos del diseño, fue necesario emplear configuraciones de *multicycle path*, sabiendo que dichos casos pueden tolerar retardos mayores a un período del reloj.

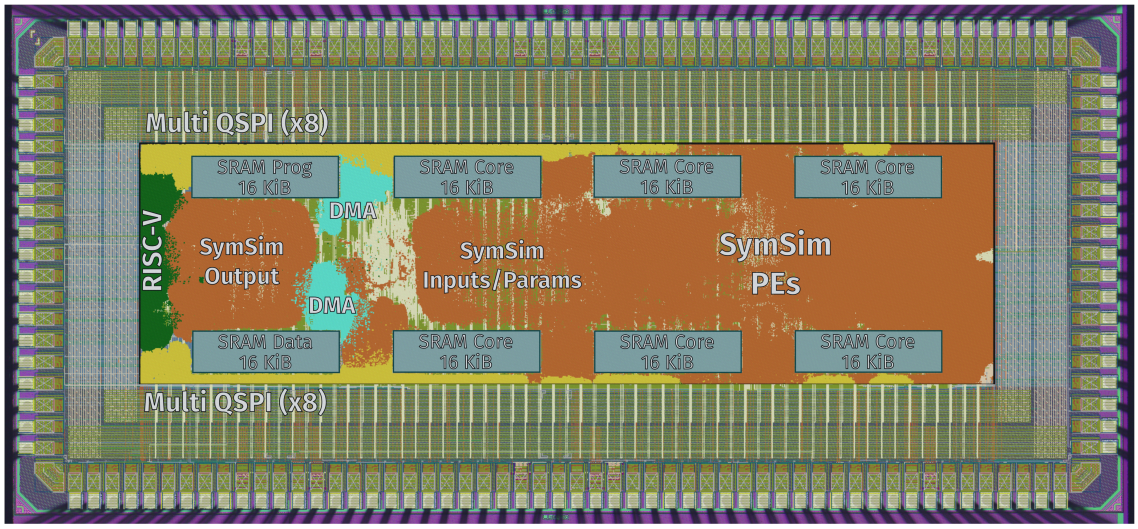


Figura 5.9: DigineuronV3a SoC, *layout* superpuesto sobre el *die*, donde se marcan los bloques más importantes del mismo.

Para la fabricación de este ASIC se utilizó el proceso TSMC 65nm CMOS LP MS/RF, lo que permite fabricar dispositivos de bajo consumo (LP) y tiene soporte para circuitos *Mixed-Signal* (MS) y de radiofrecuencia (RF), aunque solamente se le sacó provecho al apartado de bajo consumo ya que el diseño en sí mismo es puramente digital y de frecuencia estándar. La metalización empleada fue 1P9M_6X1Z1U, que cuenta con una sola capa de polisilicio y 9 capas de metal, de las cuales las capas M2 a M7 son más gruesas y especializadas para distribución de potencia y señales que requieren baja resistividad, mientras que las líneas de metal M8 son de un mayor espesor (para una mejor distribución de potencia) y la capa M9 está optimizada para ruteo de circuitos en RF[13]. Naturalmente, al tratarse de un diseño puramente digital, las capas de metales utilizadas para P&R fueron desde M1 hasta

[13]Radiofrecuencia

M8. En cuanto a celdas estándar, se contó con librerías de ARM® con tensiones de umbral mixtas, con una mayoría de celdas de alto umbral (hVt) para bajo consumo, mientras que unas pocas celdas con tensiones de umbral regular y baja (rVt y lVt respectivamente) fueron elegidas por la herramienta de síntesis para satisfacer tiempos de *setup* en las partes más demandantes del circuito.

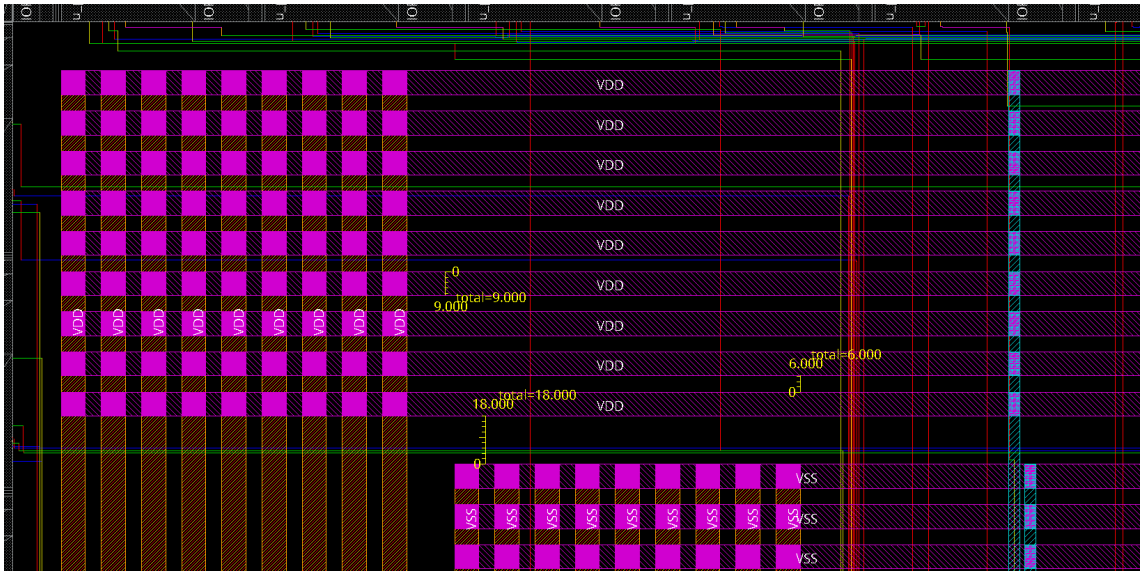


Figura 5.10: Anillo de metales para distribución de potencia en el SoC DigineuronV3a.

El proceso seleccionado soporta tensiones de alimentación nominales de 1,2V o 2,5V, con tensiones de entrada/salida de 3,3V, aunque según el diseño del circuito y distribución de potencia, el circuito integrado puede funcionar con otras tensiones de I/O[14] (como por ejemplo 1,8V) y tensiones de alimentación menores o mayores a los valores nominales. En la Fig. 5.9 también se pueden apreciar los anillos para la distribución de potencia (*power rings*) diseñados con múltiples capas y líneas de metal de manera que el SoC pueda soportar corrientes de alrededor de 243mA, o el equivalente a una potencia de 290mW con una tensión de alimentación de 1,2V. Estos anillos se diseñaron con una aproximación de 1mA soportado por cada 1μm de ancho de línea, de manera que se utilizaron 3 capas de anillos ($M2$-$M3$, $M4$-$M5$ y $M6$-$M7$), con 9 líneas de 9μm de ancho por dominio de potencia (V_{DD} y V_{SS}). El anillo para distribución de la tensión de alimentación V_{DD}, junto con las medidas de ancho de pista y espaciado, se muestra en la Fig. 5.10. Tanto el espaciado entre

[14]Entrada/Salida - *Input/Output*

líneas de $6\mu m$ como el *offset* entre dominios de potencia ($18\mu m$) fueron elegidos, además, para satisfacer las reglas de diseño de espaciado y densidad de metales. Esto reduce considerablemente el espacio disponible para el diseño (*core area*), pero al combinarse con una buena distribución de potencia (cantidad adecuada de *pads* de alimentación y líneas de distribución desde los anillos), permite que el diseño funcione a frecuencias superiores al incrementar la tensión de alimentación (*overclocking*), sin apenas aumentar la temperatura. Además de lo anteriormente mencionado, se utilizó la capa M8 para las líneas de distribución de potencia verticales (de *ring* a *core*), aprovechando la baja resistividad que este metal ofrece.

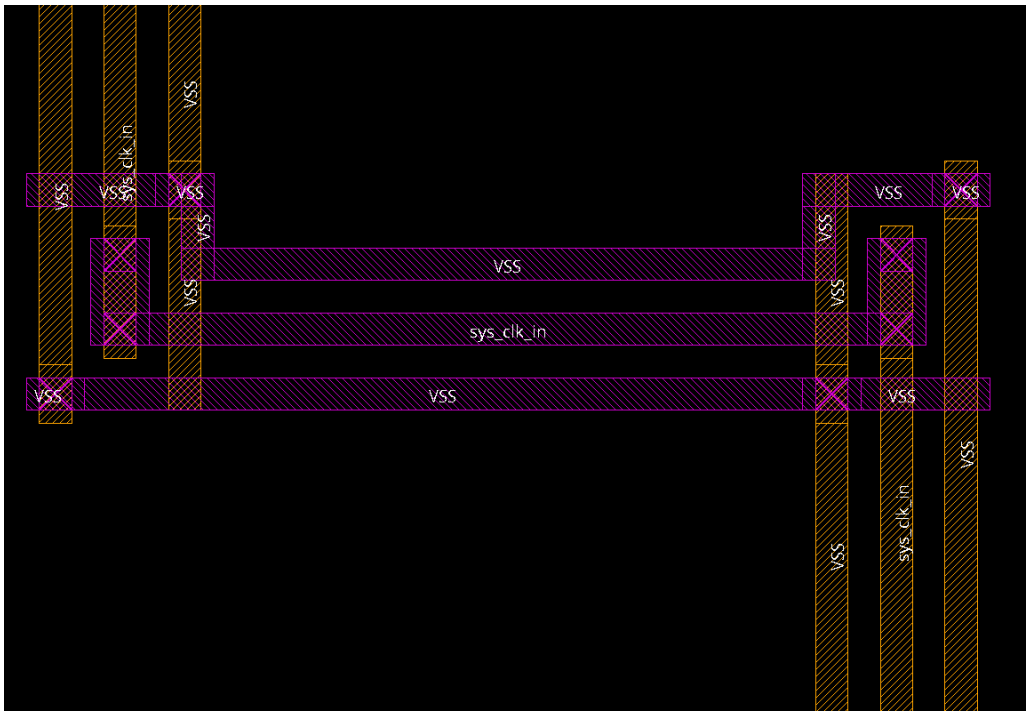


Figura 5.11: Línea de distribución de la señal de reloj (de *pads* a *core*).

Dada la relación de aspecto del *chip* que se observa en la Fig. 5.9, para realizar una correcta distribución del reloj a las celdas del circuito, se ubicó el correspondiente *pad* en el centro (parte superior) del ASIC. Además de esto, se cubrió dicho *pad* con los propios de V_{DD} y V_{SS} (GND) con el fin de utilizar estos planos de tensión continua para proteger la señal del reloj de ruido y *cross-talk*, a la vez que se mantiene la integridad de la señal y se asegura una impedancia controlada para proporcionar un entorno operativo estable. En la generación del árbol de reloj, se realizó una protección adicional de sus señales internas en el *chip*, mediante la extensión de

pistas de metal con V_{SS} alrededor de las líneas que distribuyen al reloj (ilustrado en la Fig. 5.11), hasta las capas de metal M4 y M5.

Al igual que sus versiones anteriores, DigineuronV3a es un SoC con *buses* e interfaces de comunicación basadas en el protocolo AMBA® AHB para el sistema central y APB para algunos periféricos que no requieren de transferencias de datos tan rápidas. En la Fig. 5.12 se ilustra el sistema general de DigineuronV3a, donde se cuenta con tres tipos de bloques que actúan como *masters* (microprocesador, SPI2AHB y DMAs), mientras que como *slaves* del sistema AHB se tienen al acelerador ChSymSim (introducido en el Capítulo 4), SRAMs, dos interfaces QSPI múltiples y el sub-sistema APB: temporizadores, GPIO, registros de control y configuración del sistema, etc. La interconexión entre módulos *master* y los periféricos (*slaves*) del sistema se realiza mediante una "matriz AHB" que contiene varios árbitros y decodificadores-multiplexores para manejar las transacciones del protocolo AMBA. En comparación con los *chips* anteriores (Sección 5.1), DigineuronV3a posee muchos más bloques de SRAM, aumentando la memoria total disponible a $128KiB$. Si bien esta implementación necesita de una matriz AHB más compleja (más bloques que distinguir según su dirección), esto facilita el acceso de datos (lectura/escritura) en paralelo, siempre que los mismos se encuentren distribuidos en diferentes bloques de memoria. De esta forma, mediante una adecuada distribución de los datos, se tiene menos probabilidad de que dos o más *masters* realicen transacciones con la misma SRAM y se incurran en tiempos de espera por arbitrajes (según las prioridades definidas en los registros de control del sistema). Para más información sobre el sistema AHB y su mapa de memoria en el *chip* DigineuronV3a, se recomienda al lector dirigirse al Apéndice H.

Como unidad de control y configuración principal, este *chip* hace uso del mismo microprocesador de uso libre CV32E40P [66] que DigineuronV2 (Sección 5.1), con las conversiones requeridas para sus dos interfaces OBI de 32 bits (instrucciones y datos) al *bus* central AHB de 64 bits. De forma similar a la elección de la tecnología, se decidió utilizar este procesador debido a que ya fue probado en un *chip* preliminar, permitiendo así centrar los esfuerzos en la implementación del acelerador ChSymSim que representa el elemento central en esta tesis. Además del microprocesador, se hace uso de un módulo SPI2AHB tanto para control como para *debug* del sistema, el

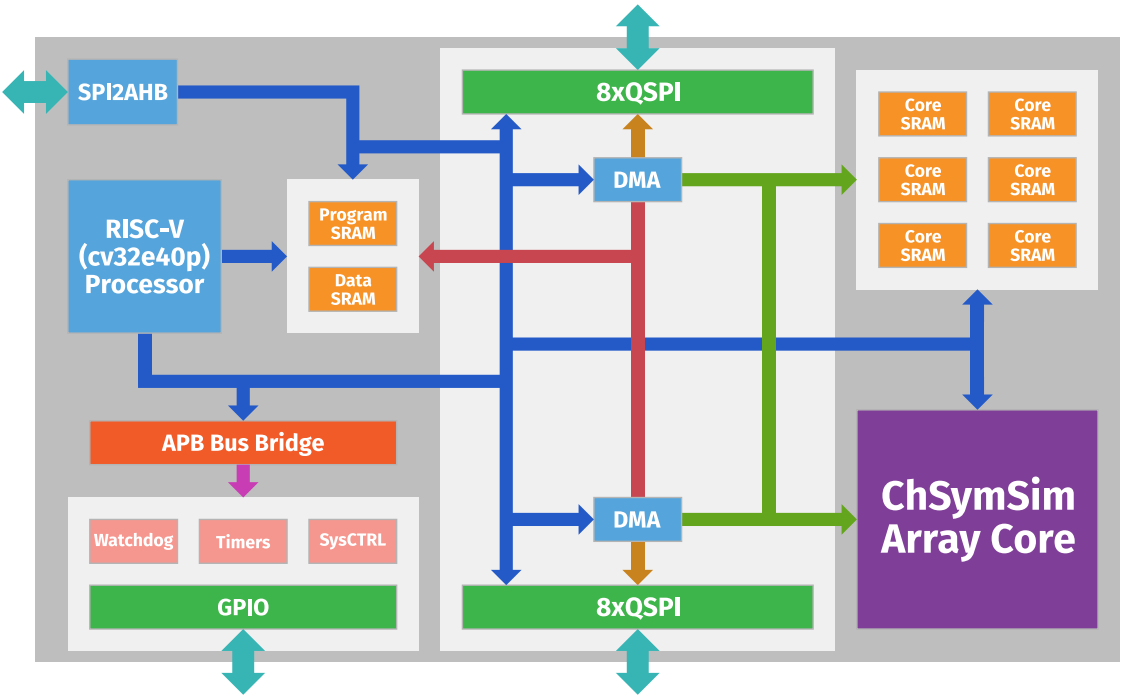


Figura 5.12: Diagrama en bloques con los módulos de DigineuronV3a SoC.

cual obtiene las instrucciones para las transferencias en AHB desde un controlador externo (específicamente computadora y FPGA) por medio de su interfaz SPI[15]. De esta forma, cuando el bloque SPI2AHB es activado por medio de su señal *spi_cs_n* y recibe en su puerto *spi_mosi* la información referente a modo de transacción, dirección dentro del *chip* y eventualmente datos (en caso de tratarse de una escritura), el controlador interno decodifica esta información y, una vez finalizada la transferencia serie, se presenta en el puerto AHB la transacción correspondiente. En caso de una lectura, el controlador realiza la transacción AHB inmediatamente después de recibir modo y dirección, para luego enviar los datos leídos en serie por el puerto *spi_miso*. Todas las transferencias así como máquina de estados y registros vinculados al protocolo SPI en este módulo, son controlados por medio de la señal de reloj externa *spi_sclk*, que debe estar en polaridad CPOL = 0 (reloj inicia y finaliza en valor 0 lógico) y fase CPHA = 1 (datos presentados en el flanco ascendente del reloj y leídos durante el flanco descendente del mismo).

Por otra parte, los controladores de DMA (inspirados en Arm® PrimeCell® μDMA Controller) tienen dos interfaces *master* de AHB (64 bits), dedicadas a rea-

[15] *Serial Peripheral Interface*

lizar operaciones de lectura y escritura simultáneamente. Gracias a esto, es posible que cada segmento de datos sea escrito inmediatamente después de haberse leído desde otra ubicación, a costa de agregar complejidad a la matriz AHB (cada DMA equivale a dos *masters* de AMBA). Al realizarse el pedido de datos a uno de los DMAs, ya sea por medio de señales externas o al escribir sus registros internos, este controlador busca en la memoria SRAM correspondiente la información respecto a la transferencia que va a realizar, lo que le toma alrededor de 4 ciclos de reloj. Esta configuración de transferencia contiene datos tales como dirección de origen de los datos a transferir, la dirección de destino, cantidad de datos a leer/escribir, saltos de dirección en dos o tres dimensiones, entre otros. Cada DMA posee 8 canales con prioridades definidas por el usuario, lo que les permite utilizar hasta 8 configuraciones de transferencia diferentes más otras 8 configuraciones alternativas adicionales. Si se reciben el pedido de datos por más de un canal a la vez, se tienen unos ciclos de reloj como retardos adicionales en los que el DMA revisa las prioridades y selecciona qué transferencia (canal) realizar primero. Por otro lado, los registros internos de cada DMA se acceden a través del *bus* APB, y contienen la dirección a la primera configuración de transferencia (canal 0), la dirección a la configuración alternativa del mismo canal, prioridades de los canales de transferencia, etc. Estos registros se encuentran localizados en el sub-sistema APB ya que, a diferencia de las configuraciones de transferencias, no se necesita modificar sus valores con tanta frecuencia. En el Apéndice H se encuentran más detalles sobre el mapa de memoria de los registros internos de los DMAs, al igual que los tipos de configuración de transferencias.

Una vez que el DMA finaliza una transferencia completa de datos, este sobre-escribe la configuración del canal correspondiente en SRAM, generalmente guardando la última dirección de origen y destino, a la vez que se invalida la configuración. Dado que este comportamiento no resulta eficiente para su uso con el acelerador ChSymSim, debiendo esperar por cada ciclo de procesamiento a que se modifiquen y validen estas configuraciones, se debieron realizar algunas modificaciones en el diseño de los controladores de DMA. Dichas modificaciones consisten principalmente en incorporar registros internos que, según su valor, eviten que el DMA sobre-escriba ciertos datos de configuración de la transferencia anterior. Además, los registros de

entradas, parámetros y salidas del operador ChSymSim se encuentran siempre en el mismo bloque de direcciones, por lo que se añadieron banderas (*flags*) en la configuración de transferencia para mantener estas direcciones iniciales de origen o destino según corresponda. De esta forma, se puede realizar la escritura y procesamiento de múltiples canales de entrada en serie, sin necesidad de alterar las configuraciones en SRAM, lo que reduce los ciclos de espera y el tiempo de procesamiento total.

Otro bloque que fue modificado según las necesidades del operador ChSymSim es la interfaz QSPI. El módulo de QSPI implementado en los *chips* anteriores (Sección 5.1) cuenta con dos memorias FIFO[16], una para lectura y otra para escritura, a las que se accede por medio del *bus* AHB. Para realizar una escritura, se cargan los datos en la FIFO correspondiente, que separa los bytes a ser enviados, y en base a la configuración almacenada en los registros internos del QSPI, se escriben las señales de reloj, *chip select* y datos (1, 2 o los 4 puertos) que correspondan a dicha transacción. De forma similar, para una lectura se escriben algunos datos en la FIFO de escritura (modo, dirección, etc.) y se inicializa la transferencia por medio de registros internos en el QSPI. En este caso, los datos leídos por la interfaz se almacenan en la FIFO de lectura por byte recibido, y son concatenados para llenar el *bus* AHB que cuenta con 64 bits. Dada la cantidad de entradas y parámetros (por canal) requeridos en cada ciclo de procesamiento, si estos datos deben ser obtenidos de memorias externas al *chip*, un sólo bloque QSPI resulta insuficiente para proporcionar todas las entradas o pesos necesarios en un tiempo cercano o menor que el procesamiento en sí mismo, lo que representa un cuello de botella en el tiempo de cómputo total del acelerador. Por esta razón, el módulo de QSPI fue expandido para poder leer/escribir más datos en un menor tiempo, utilizando múltiples líneas de datos. Para evitar replicar *hardware* al expandir dicho bloque, esta nueva versión denominada MQSPI[17] simplemente incrementa el número de FIFOs de lectura y escritura en paralelo, mientras que se comparte la lógica de control y configuración. En base al ancho del *bus* AHB, se optó por utilizar 8 memorias FIFO (16 en total contando las de lectura y las de escritura), que se conectan al *bus* contribuyendo con un byte cada una, lo que equivale a tener 8 particiones QSPI que se acceden

[16] *First Input First Output*
[17] *Multi-Quad Serial Peripheral Interface*

en paralelo. Además, se agregó lógica adicional que permite realizar transferencias con solo algunas particiones QSPI, en caso de que se conecten diferentes módulos a esta interfaz y/o se requiera que algunas de estas particiones funcionen de forma separada. En DigineuronV3a se integraron dos módulos MQSPI para poder realizar tanto lecturas de entradas como de parámetros en paralelo, o bien leer datos a la vez que se escriben los resultados del acelerador ChSymSim en una memoria externa.

El procesador ChSymSim integrado en este *chip* corresponde al acelerador con arreglo de PEs descrito en el Capítulo 4. Este implementación posee cuatro interfaces de AMBA® AHB con *buses* o puertos independientes, separando entre datos de configuración, entradas, parámetros y salidas. En comparación con las versiones prototipo en los *chips* anteriores (Sección 5.1), esto resulta en una gran mejora en cuanto a velocidad de cómputo, ya que se minimizan los tiempos de espera y ciclos muertos del operador ChSymSim al poderse realizar lecturas y/o escrituras de datos en paralelo. Por medio de los dos controladores de DMA es posible escribir entradas y parámetros al mismo tiempo, o escribir datos a la vez que se leen resultados de cómputos anteriores, mientras que el microprocesador puede cambiar algunas configuraciones en caso de ser necesario. Desde el puerto de configuración se pueden acceder a diversos registros de control, con los que se pueden reiniciar los valores de las memorias de entradas, parámetros y registros internos de los PEs, así como también definir los modos de operación del acelerador y habilitar/deshabilitar ciertas funcionalidades. Algunos ejemplos de estas configuraciones son la selección de entradas y salidas signadas o no signadas, selección del modo de operación con PEs independientes (para *kernels* pequeños) o PEs compartidos (*kernels* grandes), precisión de entradas, pesos y salidas, habilitación de escalado automático de resultados, entre otras. En general, estos registros de configuración se disponen de forma consecutiva, dejando algunos espacios vacíos para que el acceso a ciertas variables se pueda realizar de forma directa e independiente, sin necesidad de leer datos previos y aplicar máscaras al escribir el valor nuevo en el registro.

Los *buffers* de entrada se disponen en forma de arreglo en dos dimensiones (Tabla 5.1), con las columnas en direcciones contiguas, por lo que con una transacción del *bus* completo (64 bits) se pueden escribir hasta 8 columnas del arreglo de entrada. En cambio, las filas de este arreglo de entradas se acceden mediante saltos de dirección,

	Col. 1	Col. 2	Col. 3	Col. 4	Col. 5	Col. 6	Col. 7	Col. 8	Col. 9	Col. 10	Col. 11
Fila 1	0x00			···				0x07	0x08	···	0x0A
Fila 2	0x10			···				0x17	0x18	···	0x1A
Fila 3	0x20			···				0x27	0x28	···	0x2A
···				···						···	
Fila 11	0xA0			···				0xA7	0xA8	···	0xAA

Tabla 5.1: Mapa de memoria resumido del arreglo de entradas al acelerador ChSymSim.

	SM	*SE*	Pesos	*Bias*
Inicio	0x000	0x400	0x800	0xC00
Fin	0x00A	0x40A	0xAB9	0xC0F

Tabla 5.2: Mapa de memoria resumido para los parámetros del acelerador ChSymSim.

según la cantidad de *double-words* que ocupan las columnas. Por otra parte, las memorias de parámetros están divididas en cuatro porciones de 1024 bytes (Tabla 5.2), donde cada una aloja todos los *features* correspondientes a la Máscara de Forma del *kernel* (SM), SE, pesos simétricos y *bias*. El tamaño de dichas porciones de memoria se eligió en base al bloque que más espacio necesita, que en este caso son los pesos (requieren un total de 656 bytes), alineando el primer valor de un nuevo *feature* en el primer byte del *bus* (dirección inicial como múltiplo de 8), y simplificando el decodificador al elegir los primeros bits no utilizados de la dirección de parámetros (bits [12:11]). En el caso particular del *bus* de salida, al conectarse tanto resultados finales como datos parciales (con motivos de *debug*) se utilizaron registros de *pipeline* para aumentar la frecuencia máxima de operación del circuito de salida. El uso de estos registros de *pipeline* requirió la implementación de una máquina de estados que añada ciclos de espera para la lectura de dicho *bus*. En base a la dirección seleccionada en el *bus* de salida, se pueden leer las salidas escaladas (resultados finales de hasta 8 bits) con o sin *padding*, los resultados de los PEs sin escalar (24 bits), las direcciones de pesos generadas por los módulos *Address Encoder* y las señales PWM producidas por los comparadores. La arquitectura del procesador ChSymSim y sus interfaces se ilustra en la Fig. 5.13, mientras que los mapas de memoria completos de configuración, entradas, parámetros y salidas, junto a descripciones detalladas de las señales internas del acelerador, se encuentran en el

Apéndice I.

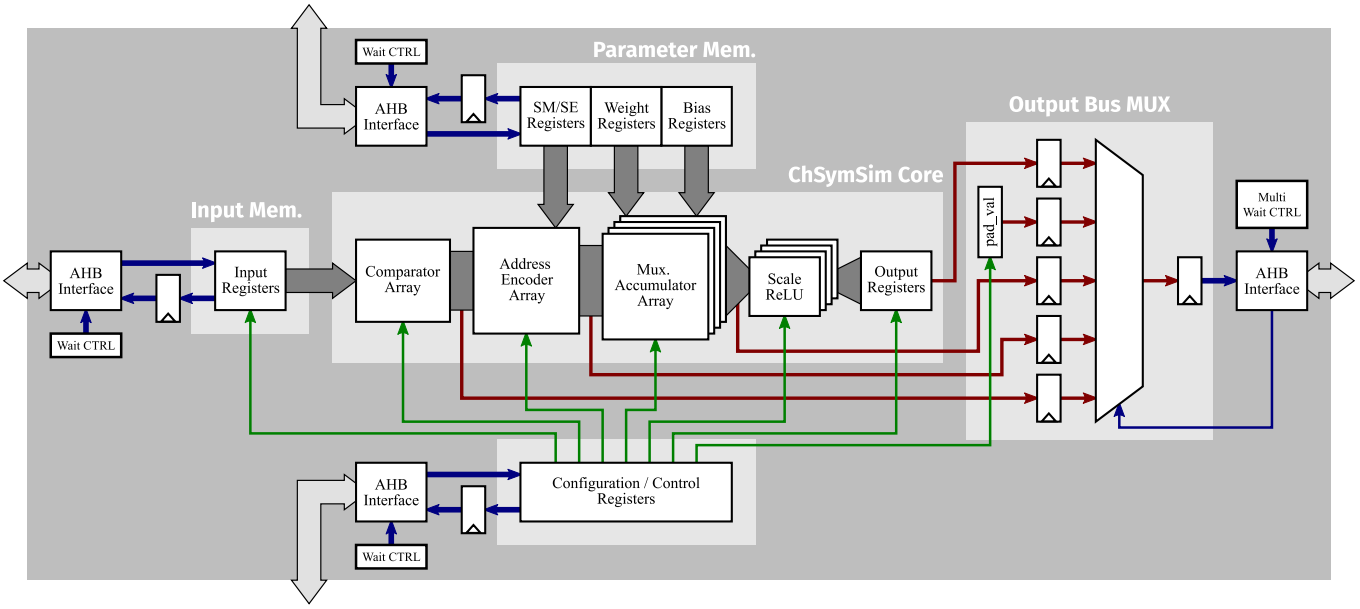


Figura 5.13: Diagrama en bloques con el acelerador ChSymSim y las interfaces con el *bus* AHB.

Además de los controladores (automáticos) del operador ChSymSim y unidades de escalado y ReLU, se implementó un modo manual o *debug* en el que las señales de habilitación y *reset* provienen de registros de configuración *set-reset* internos, que vuelven su estado original al siguiente ciclo de reloj. También se cuenta con registros especiales que contienen los valores de contadores necesarios para el control de algunos bloques, tales como contador de canal por byte (en modo de canales comprimidos con precisiones menores a 5 bits), punteros a salidas para escalar, valor de rampa para comparar las entradas, etc. Esto resulta de gran utilidad cuando se desea leer resultados parciales tales como las direcciones de pesos o las entradas codificadas en PWM, dado que en el modo de operación automático (utilizando los controladores dedicados) estas salidas normalmente cambian en cada ciclo de reloj, lo que dificulta la verificación del funcionamiento del circuito.

Para realizar cómputos de forma más eficiente (minimizando ciclos de espera), se implementó una conexión directa entre el procesador ChSymSim y los DMAs, utilizando las señales *request* y *done* de estos últimos. Cada vez que se inicia el procesamiento o un ciclo de canal, el acelerador ChSymSim realiza el pedido de los datos a los DMAs directamente, mediante las señales de *request* que se activan por

un solo ciclo de reloj, para lo que deben estar habilitadas en la configuración general del acelerador. Una vez se terminan de transferir los datos, los controladores de DMA dan aviso al operador ChSymSim de que los datos se encuentran disponibles para su uso (señal *done*) que, en caso de ser habilitadas en los registros de configuración, activan las señales *ready* para que el controlador ChSymSim prosiga con su cómputo. Esto permite que el acelerador se comunique directamente con los controladores de DMA para la transferencia de entradas y parámetros de cada canal, así como la salida escalada, sin necesidad de que el microprocesador intervenga durante el ciclo entero de procesamiento (siempre que el escalado sea automático).

En conclusión, DigineuronV3a representa una gran mejora en comparación con sus dos versiones anteriores (Sección 5.1). Este nuevo SoC no solo integra la versión final del operador ChSymSim, presentado en el Capítulo 4, que expande tanto sus dimensiones como sus funcionalidades, sino que además optimiza las interfaces y periféricos del sistema en base a las necesidades de dicho acelerador. Este *chip* también aprovecha algunas conexiones especiales de los controladores de DMA, con las que se puede realizar ciclos de procesamiento y transferencia de datos con tiempos de espera tan cortos como sea posible, reduciendo estos a solo sus latencias intrínsecas. De esta forma, solo se requiere el uso del μP al principio del procesamiento, tanto para generar las configuraciones de transferencia de los DMAs (según tamaño de *kernel*, *stride*, etc.), como para inicializar el controlador del acelerador. Por otro lado, si bien el diseño del ASIC está pensado para bajo consumo y bajas frecuencias de reloj, las distribuciones de potencia y reloj internas permiten que el circuito pueda operar a mayores frecuencias, sacrificando consumo energético al aumentar la tensión de alimentación, pero sin experimentar incrementos notables en la temperatura.

5.3. Resultados experimentales

En esta sección se presentan los resultados experimentales en cuanto a consumo, rendimiento (*throughput*) y eficiencia energética del SoC DigineuronV3a, junto a una comparación con las mediciones de los *chips* prototipo DigineuronV1 y DigineuronV2, presentados en [43] y [44] respectivamente.

Previo a la obtención de valores de consumo y desempeño de los *chips* fabrica-

dos, es fundamental la evaluación y verificación funcional de los bloques integrados en estos. En la Sub-Sección 5.3.1 se presenta el entorno de pruebas utilizado para este fin, donde se detalla la plataforma (*software* y *hardware*) empleada para la comunicación con los SoCs, así como el diseño del circuito de evaluación producido por el grupo de investigación. La Sub-Sección 5.3.1 también describe la metodología de evaluación empleada para comprobar el correcto funcionamiento de los distintos bloques del *chip*, y presenta algunos de los resultados obtenidos. En el caso particular del acelerador ChSymSim integrado en DigineuronV3a, se realiza además un experimento computando algunas capas de la red ResNet06 entrenada y cuantizada en el Capítulo 3, con datos de entrada "reales" y comparando las salidas del *chip* con las correspondientes a los modelos objetivo en PyTorch.

Luego de las pruebas y verificaciones de funcionalidad, en la Sub-Sección 5.3.2 se muestran los resultados de mediciones de potencia y cálculos de desempeño para los aceleradores ChSymSim de los SoCs fabricados en esta tesis. Esta sub-sección se enfoca en DigineuronV3a y su versión final del procesador ChSymSim, para lo que presenta los datos medidos y calculados que resultan del cómputo de dicho acelerador ante diversas configuraciones de capa (tamaño de *kernel*, *stride* y precisión) y condiciones de tensión de alimentación/frecuencia de reloj. Finalmente, la Sub-Sección 5.3.2 incluye una comparación de los valores de consumo, rendimiento y eficiencia obtenidos con las versiones prototipo y otros *chips* que representan el estado del arte.

5.3.1. Evaluación de los SoCs fabricados

Para la evaluación de los SoCs fabricados en esta tesis se utilizó un entorno de pruebas que consiste principalmente en una computadora con la cual se envían estímulos al *chip* en cuestión mediante su interfaz USB[18] 3.0, y que se conecta a una placa de desarrollo Opal Kelly® XEM7310 (con una FPGA Xilinx® Artix-7) que implementa los circuitos necesarios para traducir los comandos de USB 3.0 a las interfaces del SoC, tal y como se muestra en la Fig. 5.14. Debido a que las placas Opal Kelly® presentan un entorno de desarrollo (FrontPanel® SDK[19]) que consiste en

[18] *Universal Serial Bus*
[19] *Software Development Kit*

bloques IP en *hardware* para interfaces de comunicación computadora-FPGA, junto con una API[20] en diferentes lenguajes de programación (como Python en este caso), plataformas como XEM7310 se vuelven herramientas útiles para la implementación de circuitos con los que evaluar SoCs. De esta forma, los datos enviados por la interfaz USB de la computadora son recibidos por un microcontrolador en la placa de desarrollo, que se conecta directamente a un bloque *host* en la FPGA. El módulo *host* es el encargado de manejar la comunicación con los diversos *endpoints*, los cuales se conectan a este por un *bus* compartido y proveen/reciben datos de los bloques del circuito diseñado para evaluar los SoCs.

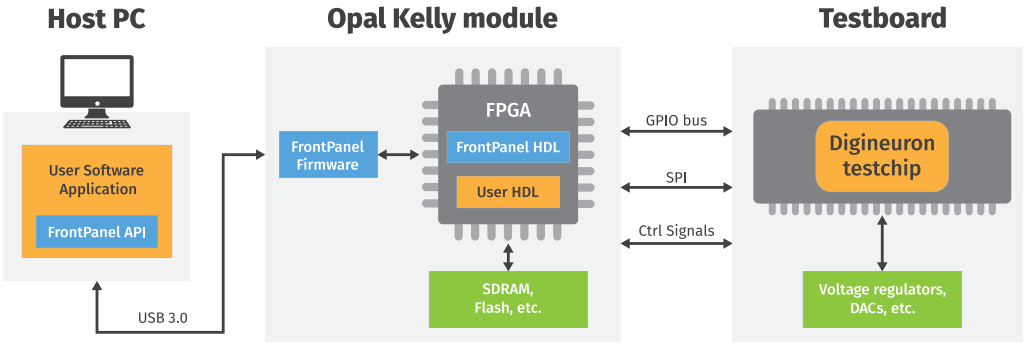


Figura 5.14: Entorno de pruebas para la evaluación de los SoCs fabricados.

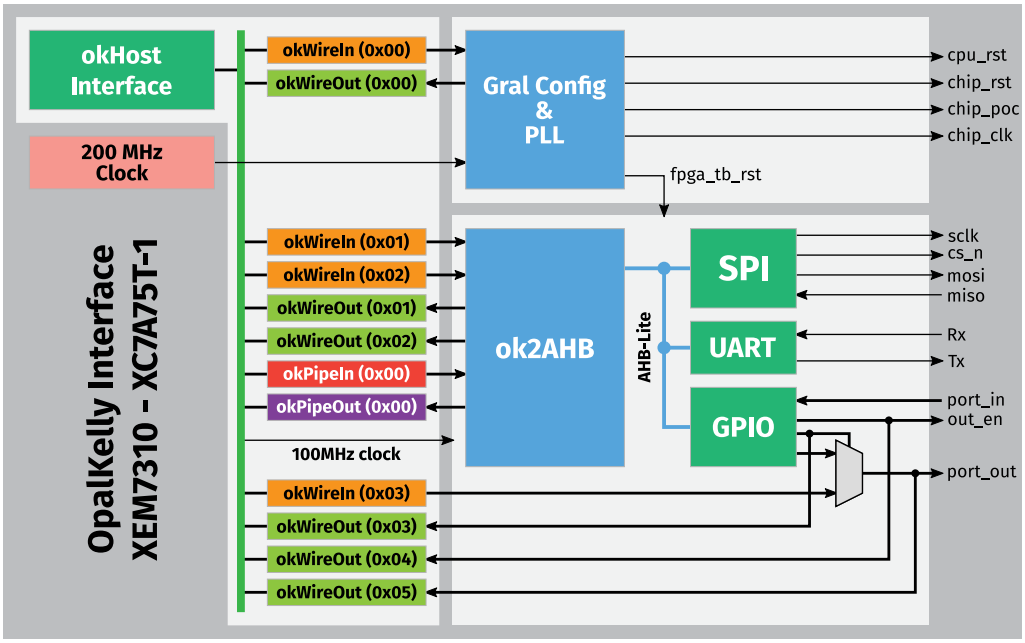
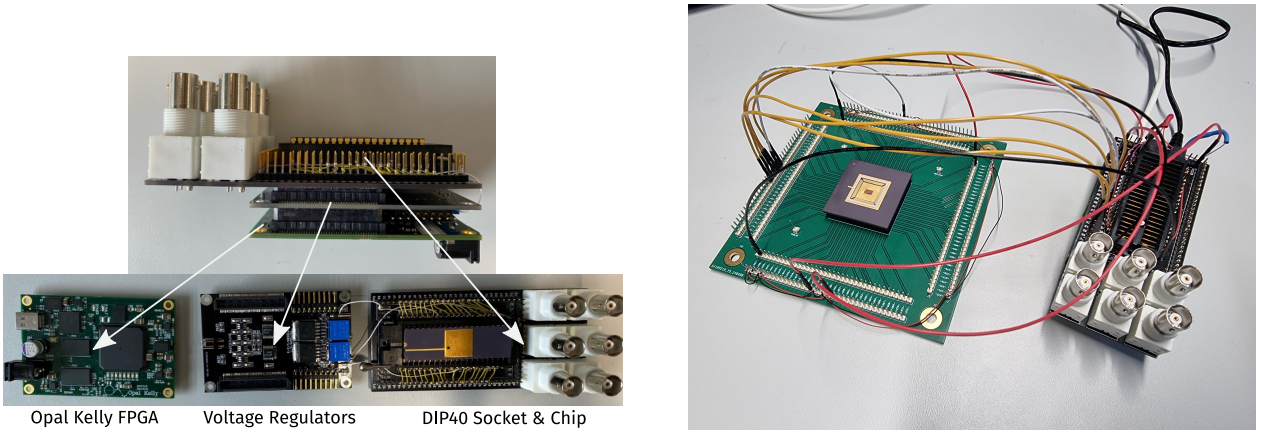


Figura 5.15: Diagrama de bloques de la plataforma de pruebas (*testbech*) en FPGA.

El circuito de pruebas en FPGA, desarrollado por el grupo de investigación para

[20] *Application Programming Interface*

la evaluación de los SoCs fabricados, se ilustra en la Fig. 5.15. Este circuito presenta un módulo SPI *Master* que controla directamente a la unidad SPI2AHB del *chip* a ser evaluado, lo que facilita el acceso directo a las memorias y registros internos del SoC, a la vez que permite ejecutar y depurar los periféricos y aceleradores integrados. También se implementaron interfaces GPIO y UART para comunicarse con las propias del *chip* y que, junto con el módulo SPI *Master*, pueden ser controladas utilizando los *endpoints* (IPs) de Opal Kelly® a través de un bloque ok2AHB. Los tipos de *endpoint* utilizados en este diseño son okWireIn/okWireOut para configuración, mientras que los bloques okPipeIn/okPipeOut se usan para transferencia de datos en serie. Por otra parte, tanto los *resets* del *chip* completo y el microprocesador integrado en este, como el correspondiente al sub-sistema AHB en la FPGA, se controlan mediante un módulo de configuración general, que también presenta una unidad PLL[21] para generar el reloj que alimenta al SoC.



(a) (b)

Figura 5.16: Plataformas de evaluación para los SoC: a) DigineuronV1 y V2; b) DigineuronV3a.

En conjunto con la computadora y la placa Opal Kelly® XEM7310, para la evaluación de los *chips* se utilizó una placa con reguladores de tensión para convertir las señales que provienen de la FPGA en los 3,3V requeridos para la tensión de I/O de los SoCs. Esta placa intermedia también suministra a los *chips* con los 1,2V de tensión *core* nominal indicada por la tecnología con la que estos fueron fabricados. Finalmente, se hace uso de una placa con zócalo para el SoC a evaluar y pines con

[21] *Phase-Locked Loop*

los que conectar sondas de un osciloscopio u analizador lógico para la observación de las señales que entran y salen del *chip*. Esta placa hace uso de dos líneas de pines independientes a cada lado del encapsulado, una línea que se conecta con los puertos del *chip* y la otra para las salidas/entradas de la FPGA. Esto requiere de un cableado adicional para la conexión FPGA-SoC que, si bien no es eficiente en cuanto a integridad de señal se refiere, permite el uso de la misma placa para evaluar varios prototipos con el mismo encapsulado pero con configuraciones de puertos diferentes (tal y como ocurre con DigineuronV1 y V2). Debido a las dimensiones y cantidad de puertos disponibles en el SoC DigineuronV3a, el encapsulado utilizado para éste difiere del de las dos versiones anteriores (CPGA144 en lugar de DIL40), por lo que se debió diseñar y fabricar una placa adicional con el zócalo y el número de pines necesarios para este *chip*. En la Figura 5.16a se muestra una foto con la plataforma de pruebas compuesta por las placas anteriormente mencionadas (FPGA, reguladores de tensión y zócalo para *chip*) y su conexión en forma de "sandwich". Por otro lado, en la Fig. 5.16b se presenta una foto con la misma plataforma de pruebas, a la que se añade la placa específica para el SoC DigineuronV3a con el *chip* incluido.

```
SPI2AHB + SRAM Test:
Data Written to 'data_mem' (Addr 0x0)    ->      | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0a | 0b | 0c | 0d | 0e | 0f | 10 |
Data Read from 'data_mem' (Addr 0x0)     ->      | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0a | 0b | 0c | 0d | 0e | 0f | 10 |
```

Figura 5.17: Patrón de datos escrito y leído del *chip*, por medio de la interfaz SPI2AHB, como prueba de "vida" del sistema.

Antes de poder evaluar el desempeño de los aceleradores, es necesario comprobar el correcto funcionamiento de los *chips* y sus bloques internos, para lo que se utilizaron diversas configuraciones y programas de prueba. Un primer análisis consiste en probar el bloque SPI2AHB, ya que esta es la interfaz fundamental que permite acceder y depurar a los demás bloques del sistema. Para esto, se pueden utilizar los registros internos del módulo, que solo requieren las señales propias del protocolo SPI, por lo que la correcta escritura y lectura de estos registros representa la primera "prueba de vida" del *chip*. Como evaluación del sistema AHB, se envían y leen datos hacia los bloques de SRAM internos por medio de dicha interfaz, para luego observar si los datos leídos corresponden a los enviados, tal y como se ilustra en la Fig. 5.17. De esta forma no solo se evalúa al módulo SPI2AHB, sino que además se verifica el

correcto funcionamiento de las SRAMs y la Matriz AHB. Una vez probada la interfaz de *debug* principal (SPI2AHB) y sus conexiones internas, se procede a evaluar al microprocesador integrado en los SoCs. De esta forma se escribe un programa en C que instruya al microprocesador a generar y escribir en memoria datos con patrones específicos. Este programa es compilado en la computadora y enviado al *chip* por medio de su interfaz SPI2AHB. Una vez el programa a ejecutar se encuentra cargado en las memorias SRAM, se desactiva el *reset* del microprocesador y se espera el tiempo necesario para que finalice la ejecución del programa, para finalmente leer los datos generados por el μP y corroborar que estos son los valores esperados. De igual manera se comprueba el correcto funcionamiento de los demás módulos y periféricos integrados en los *chips*, generando y transfiriendo datos con patrones determinados que pueden ser contrastados al finalizar la operación, ya sea por medio de lectura y escritura de datos directa con el módulo SPI2AHB (con instrucciones directas desde la API en Python) o por medio de código en C ejecutado por el microprocesador.

En el caso particular de los procesadores ChSymSim implementados, su funcionamiento se comprueba generando datos de entradas y pesos aleatorios, que se cargan en las memorias SRAM del *chip* (para luego ser transferidos por el μP y/o DMAs) o directamente en los registros del acelerador utilizando la interfaz SPI2AHB. Una vez el procesamiento de estos datos aleatorios finaliza, los resultados deben ser extraídos del *chip* para finalmente compararlos con los obtenidos mediante el modelo en *software* (Python), ya que estos representan las salidas esperadas de la función ChSymSim. Para una verificación con datos "realistas", en el caso del acelerador final en el SoC DigineuronV3a, se ejecutaron algunas capas la red ResNet06 ChSymSim (Apéndice D). Dicha red fue entrenada con imágenes satelitales de la base de datos EuroSat [3]- [4], cuyos resultados de clasificación se mostraron en el Capítulo 3. Con el fin de obtener un modelo reproducible en el SoC, se aplicó cuantización de entradas, salidas y parámetros a las capas de dicha red, para luego realizar un re-entrenamiento de la misma (*fine-tunning*), logrando un desempeño en clasificación similar a la versión en punto flotante. La cuantización elegida para este modelo corresponde a los valores de precisión máxima del acelerador ChSymSim, es decir 16 bits de *bias*, 8 bits para entradas, salidas y pesos simétricos, y finalmente SE binarios.

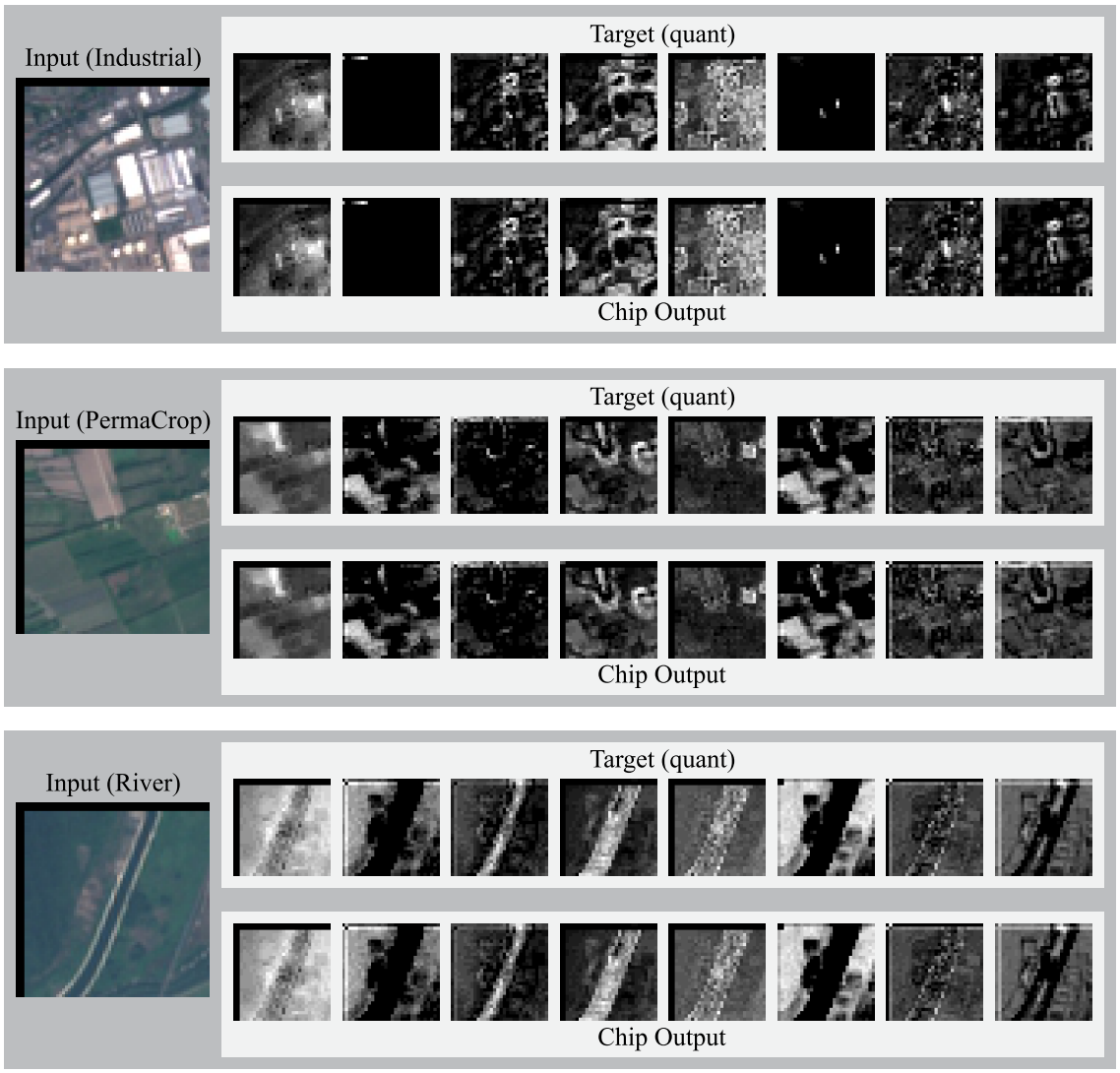


Figura 5.18: Primeros 8 *features* de la primera capa de ResNet06 (Apéndice D) obtenidas por el acelerador ChSymSim integrado en DigineuronV3a.

En la Fig. 5.18 se muestran los resultados de la primera capa de la red al procesar algunas imágenes de EuroSat con el SoC DigineuronV3a, donde se opera con un filtro ChSymSim de 7×7 entradas, *stride* 2 y 16 *features* de salida. Para simplificar el experimento, las imágenes de entrada (con *padding* 3 previamente aplicado) son subdividas en porciones de 11×11 píxeles (tamaño completo del arreglo de entradas en el acelerador ChSymSim) y se realiza el procesamiento de a una porción por vez. De esta forma, por cada iteración del cómputo de la capa, se envían las 11×11 entradas al *chip* y, una vez concluida la operación, se extraen los $2 \times 2 \times 8$ resultados con los que se reconstruyen (en la computadora) los primeros 8 *features* de salida de dicha capa, que coinciden con lo observado al procesar la entrada completa con la

capa cuantizada en PyTorch (función objetivo). Debido a que la segmentación de la imagen de entrada se realiza en bloques completos de 11×11 píxeles, se descartan las últimas filas y columnas que no llegan a completar el arreglo de entradas del procesador ChSymSim, lo que simplifica la configuración del mismo para esta prueba de funcionamiento. El efecto de este descarte de elementos de la imagen de entrada puede observarse en la Fig. 5.18 donde solo están presentes los *padding* a izquierda y superior, tanto en la entrada como los efectos producidos (marcos) por estos en los *features* de salida. Finalmente, en la Fig. 5.19 se presentan algunos *features* de salida obtenidos al procesar una imagen de EuroSat con el segundo filtro de la segunda capa de ResNet06 (*kernel* 3×3, *stride* 1, *padding* 1), tanto en sus versiones en punto flotante (*float32*) y cuantizada (PyTorch) como con el acelerador ChSymSim del *chip* DigineuronV3a. Nuevamente, las salidas del modelo cuantizado en *software* y del procesador ChSymSim del *chip* resultan idénticas.

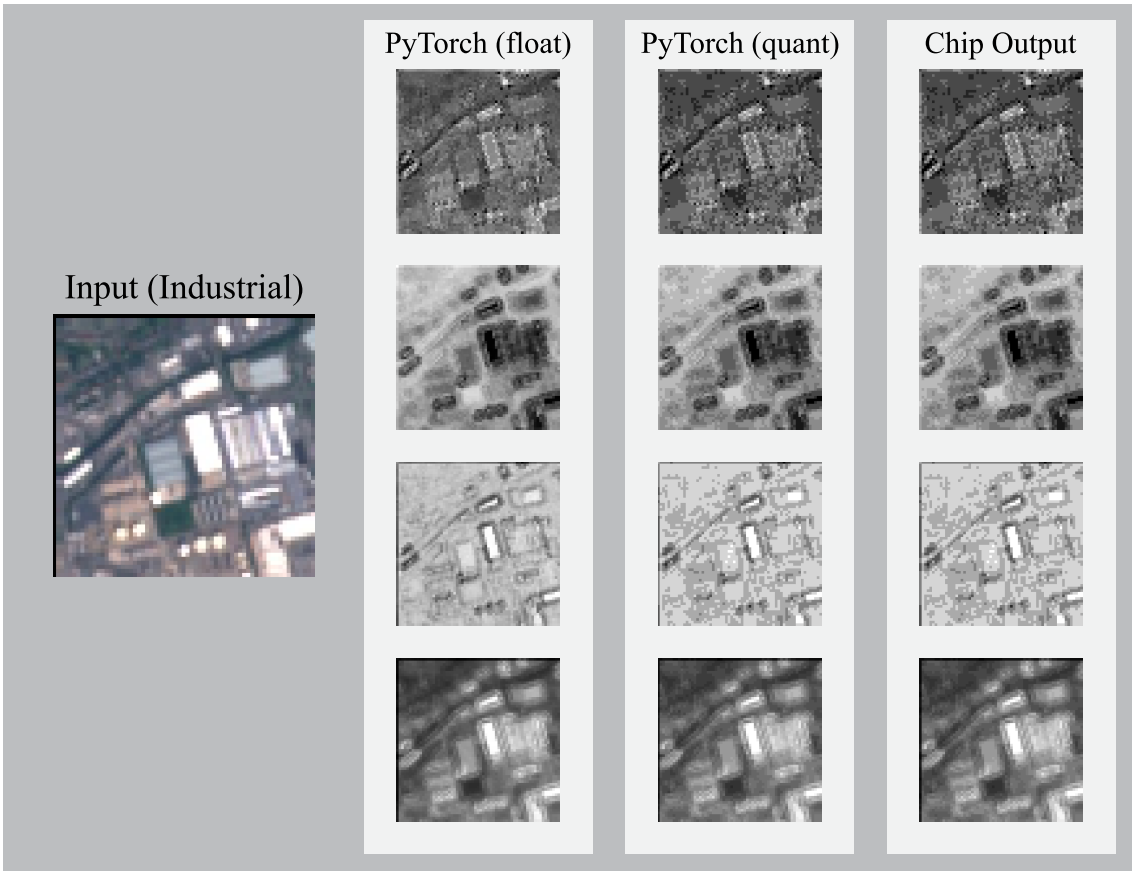


Figura 5.19: *Features* del segundo filtro de capa 2 en ResNet06 (Apéndice D), con imagen de entrada de EuroSat [3, 4], procesada con el acelerador ChSymSim de DigineuronV3a.

5.3.2. Mediciones y desempeño

Para una evaluación funcional del procesador ChSymSim, es posible realizar el cómputo haciendo uso exclusivo de la interfaz SPI2AHB para el control del sistema, sin intervención alguna del microprocesador. Para la obtención de mediciones de consumo energético promedio, en cambio, es necesario que el acelerador procese los datos de forma continua, por lo que si se utiliza solamente la interfaz SPI2AHB, que introduce retardos por transferencias de datos lentas, se contaría con numerosos ciclos con el operador en estado *IDLE* y las mediciones de consumo resultantes serían mucho menores a los valores reales durante el cómputo. Para evitar esta situación, conviene utilizar al μP para que se reinicie el procesamiento en el acelerador apenas este finalice el cómputo previo, estando en *IDLE* solamente durante los ciclos en los que el μP lee las banderas (*flags*) de estado y escribe en los registros de configuración para dar inicio a otro ciclo de procesamiento. Además, en los casos donde se cuenta con controladores de DMA, es recomendable que estos sean los encargados de transferir los datos de entradas, de parámetros (por canal) y de salidas, ya que estos permiten transacciones más rápidas y eficientes que con el microprocesador. Debido a que en los SoCs fabricados no se cuenta con dominios (puertos) de alimentación exclusivos para los aceleradores, la obtención de consumos de potencia en estos debe realizarse mediante medición indirecta, es decir, por medio de la diferencia entre una medición con el sistema completo dedicado al cómputo de la función ChSymSim y otra con los periféricos y μP realizando las mismas tareas, pero con el acelerador siempre en estado de *IDLE*. Los resultados experimentales de consumo de potencia en los aceleradores obtenidos para los SoCs DigineuronV1 y V2 (presentados en [43] y [44] respectivamente), provienen de experimentos con datos aleatorios y medición indirecta, tal y como se describió anteriormente. Para estos casos se utilizaron tanto la tensión como la frecuencia nominales de los *chips*: 3,3V para I/O, 1,2V para *core* y reloj de 100MHz. Debido a que los procesadores ChSymSim en ambos casos son de un solo PE y *feature* de salida, ambos registros de entrada y parámetros son lo suficientemente pequeños como para ser llenados por el μP con datos nuevos, antes de que terminen de procesarse los datos previos.

En el caso de las mediciones de potencia para el acelerador ChSymSim inte-

grado en DigineuronV3a, se realizó un procedimiento similar, con el μP dedicado exclusivamente a reiniciar los cómputos en bucle y dejando que la transferencia de datos se realice mediante las interacciones directas entre el operador y los DMAs (señales *request* y *done*). Para lograr que el μP se dedique solamente al reinicio del procesamiento, se optó por configurar los DMAs para que cada vez que lean los datos aleatorios desde los bloques de SRAM, sea siempre a partir de las mismas direcciones de origen iniciales (siempre el mismo canal de entrada). De esta forma se evita introducir retardos adicionales por modificar las configuraciones de los DMAs en memoria, o que estos intenten acceder a direcciones de memoria "ilegales" en algún punto del bucle infinito y comprometan el correcto funcionamiento mientras se realizan las mediciones. Para evaluar el desempeño de las estrategias de *clock-gating* y habilitación de PEs, se realizaron experimentos con diferentes tamaños de *kernel*, valores de *stride* y precisión de entradas y parámetros, considerando el procesamiento de 64 canales consecutivos. En dichos experimentos se ignoró el *padding*, dado que este solo altera algunos valores de entrada/salida pero no tiene efecto en la habilitación de los PEs (baja influencia en el consumo del acelerador). Los resultados del consumo de potencia del SoC para dichos experimentos se resumen en la Tabla 5.3, contando con *kernel* de 3×3 (8 bits de entradas y pesos) para el consumo máximo, *kernel* de 5×5 (8 bits de entradas y 1 bit de pesos) para el consumo mínimo, una alimentación de 1,8V para I/O (más utilizada que 3,3V en FPGAs más modernas), y combinaciones de tensión de *chip* (V_{DD} *core*) y frecuencias de reloj. Entre estas combinaciones de tensión/frecuencia se cuenta con valores nominales de 1,2V@25MHz, mientras que para minimizar los consumos en frecuencias de reloj bajas se utilizó el rango de 0,81V@8MHz. También se aumentó la tensión de alimentación V_{DD} *core* (a 2,1V) para hacer funcionar el acelerador a frecuencias elevadas (100MHz), superiores a las del diseño original. El valor de frecuencia nominal de 25MHz fue elegido para asegurar el correcto funcionamiento del ASIC ante problemas en el diseño del circuito de pruebas en FPGA, que causa *resets* indeseados en el sistema cuando se generan relojes con ciertos rangos de frecuencia (en este caso $50\text{MHz} \leqslant f_{CLK} < 25\text{MHz}$). Dado que esta frecuencia nominal es inferior a la del diseño del SoC, se realizaron mediciones con la tensión mínima de funcionamiento a dicha frecuencia (1,07V) para observar la disminución en el consumo energético

para este caso de operación. El instrumento de medición utilizado para obtener los resultados de potencia del *chip* fue el analizador de potencia R&S® HMC8015 que, si bien no presenta una resolución temporal elevada (máximo $100ms$ por muestra), dicho dispositivo cuenta con una API en Python, por lo que la adquisición de las mediciones puede realizarse con el mismo *script* que controla al *chip*, casi inmediatamente después de iniciarse el procesamiento.

	Reset	Sleep	SoC Max (3×3)			SoC min (5×5)		
			$q = 8$, $p = 8$	$q = 4$, $p = 4$		$q = 8$, $p = 1$		
			stride 1	*stride* 2	*stride* 3	*stride* 1	*stride* 2	*stride* 3
2,10V@100MHz	66,69	45,55	212,76	128,96	109,99	85,72	77,99	73,59
1,20V@25MHz	4,59	2,88	16,62	9,66	8,04	6,05	5,46	5,12
1,07V@25MHz	3,60	2,24	13,27	7,56	6,31	4,73	4,27	4,00
0,81V@8MHz	0,93	0,58	2,46	1,39	1,15	0,87	0,78	0,73

Tabla 5.3: Mediciones de potencia en [mW] del SoC DigineuronV3a para diferentes valores de frecuencia y tensión de alimentación, para cómputos de capas ChSymSim con entradas de "q" bits y pesos "p" (64 canales y 10 repeticiones).

$V_{DD}@f_{clk}$ \ *kernel*	1×1		2×2		3×3		5×5		7×7		9×9	
	Max	min	Max	min	Max	min	Max	min	Max	min	Max	min
2,10V@100MHz	70,73	65,60	70,83	65,62	71,74	65,81	74,96	66,32	81,35	67,07	89,61	67,98
1,20V@25MHz	4,85	4,46	4,86	4,46	4,94	4,48	5,18	4,52	5,66	4,57	6,28	4,64
1,07V@25MHz	3,80	3,49	3,80	3,49	3,86	3,50	4,07	3,53	4,45	3,58	4,96	3,64
0,81V@8MHz	0,70	0,65	0,70	0,65	0,71	0,65	0,75	0,65	0,82	0,66	0,92	0,67

Tabla 5.4: Mediciones de potencia en [mW] del SoC DigineuronV3a para diferentes valores de frecuencia y tensión de alimentación, realizando solamente control y transferencia de datos (64 canales y 10 repeticiones).

Para las mediciones en el modo de operación que excluye al procesador ChSymSim, necesarias para el cálculo del consumo del acelerador en sí mismo, se cuenta con el μP para reactivar las transferencias de datos con los DMAs, esperando el tiempo que le tomaría al operador ChSymSim procesarlos antes de pasar al siguiente ciclo de cómputo. Mediante esta estrategia se logra un comportamiento similar al que presentan el μP y otros periféricos cuando se procesa con el acelerador. Los resultados con este modo de operación se resumen en la Tabla 5.4, donde se considera solamente el caso con valor de *stride* igual a 1, ya que este no afecta a las transferencias de datos (entradas y parámetros) y cualquier diferencia (casi nula) en los resultados de potencia al variar dicho valor se debe simplemente a la aleatoriedad de

los datos a leer/escribir. En dicha tabla se presentan los valores máximos y mínimos de consumo de potencia por control y transferencia de datos, donde el valor mínimo ocurre con los casos de 1 bit de entrada y parámetros (una transferencia cada 8 canales procesados). El valor máximo, en cambio, se obtiene para *kernel* pequeños (1×1 a 3×3) en los casos de 8 bits de entradas para ambos pesos y parámetros, mientras que para *kernels* grandes este máximo se halla con 8 bits de parámetros y 2 bits de pesos. Este aumento en el consumo de potencia para transferencias de *kernel* grandes y precisión de entrada reducida se debe a dos factores fundamentales. Por un lado se tiene a la relación entre el tiempo de *setup* de los DMAs y la de transferencia de datos propiamente dicha, donde la cantidad de parámetros a enviar al acelerador para *kernels* grandes es tal que la Matriz AHB se encuentra casi siempre activa, con tiempos de *setup* y latencias de DMA despreciables en comparación con las constantes lecturas y escrituras de datos. Este efecto se combina con un mínimo tiempo de espera por procesamiento dado por la baja precisión de entradas, lo que promueve esta constante actividad en los DMAs y la Matriz AHB en dichos casos.

Para obtener un resultado de consumo de potencia más preciso, tanto en el caso de la potencia total (Tabla 5.3) como en el consumo de las transferencias de datos (Tabla 5.4), se repitió el procedimiento 10 veces, generando nuevos datos aleatorios y promediando al final todas las mediciones en el modo de operación seleccionado (*kernel*, *stride* y bits de precisión). Al procesar siempre el mismo canal, debido a que la configuración de los DMAs ocasiona transferencias de datos de las mismas regiones de memoria, la actividad resultante en el acelerador presenta un patrón fijo para todos los ciclos de procesamiento. Mediante la realización de numerosos experimentos con diferentes datos aleatorios de entradas y pesos, se logra variar dichos patrones de comportamiento en la actividad del circuito del operador ChSymSim, por lo que en promedio se tiene un valor prácticamente igual al de procesar diferentes canales.

Como se mencionó anteriormente, ninguno de los SoCs fabricados para esta tesis cuenta con puertos de alimentación individuales para los procesadores ChSymSim, sino que la alimentación general (V_{DD} *core*) es compartida por todos los bloques del *chip*. Es por esto que, para estimar (medición indirecta) el consumo del acelerador en DigineuronV3a por sí solo, se utiliza la diferencia entre el consumo de potencia total

del SoC (Tabla 5.3) y el que se obtiene por control y transferencia de datos (Tabla 5.4). Estos resultados de medición indirecta de potencia del acelerador se resumen en la Tabla 5.5, donde se muestran tanto los valores de consumo máximos y mínimos, como los casos de operación donde estos ocurren. Dichos casos de operación de consumo máximo y mínimo coinciden con lo visto para las mediciones de potencia del *chip* completo, ya que por las dimensiones del operador ChSymSim y cantidad de cómputos en paralelo que realiza, éste representa generalmente el mayor consumo del SoC. Todos los resultados de medición, tanto directos como indirectos, con las que se construyeron las Tablas 5.3, 5.4 y 5.5, se encuentran en el Apéndice J, donde se cuenta con los valores de potencia para distintos casos de *kernel*, *stride* y precisión de entradas/pesos.

	ChSymSim Max (3×3)			ChSymSim min (5×5)		
	$q = 8$, $p = 8$	$q = 4$, $p = 4$		$q = 8$, $p = 1$		
	stride 1	*stride* 2	*stride* 3	*stride* 1	*stride* 2	*stride* 3
2,10V@100MHz	141,14	59,92	41,01	15,70	8,16	3,78
1,20V@25MHz	11,70	4,92	3,30	1,25	0,66	0,32
1,07V@25MHz	9,42	3,84	2,58	0,97	0,51	0,23
0,81V@8MHz	1,75	0,71	0,47	0,18	0,09	0,04

Tabla 5.5: Estimación de potencia en [mW] del acelerador ChSymSim en DigineuronV3a, para diferentes valores de frecuencia y tensión de alimentación, realizando cómputos con entradas de "q" bits y pesos de "p" (64 canales y 10 repeticiones).

En base a los datos presentados en la Tabla 5.5 se puede observar que el consumo máximo del procesador ChSymSim se da en todos los casos para *kernel* de tamaño 3×3, pero con dos condiciones de precisión diferentes: 8 bits de entradas y pesos para el modo de operación con *stride* 1, mientras que el máximo consumo se obtiene con entradas y pesos de 4 bits cuando se opera con *stride* 2 y 3. Con 4 bits de entradas y pesos, el tiempo de cómputo se vuelve cercano al tamaño del *kernel* a computar, por lo que la actividad en los nodos del circuito (generación de dirección de pesos, selectores y acumuladores) es mayor, mientras que los tiempos de espera por transferencia de datos no producen los suficientes ciclos *IDLE* para compensar tal actividad. Por otro lado, con *stride* 1 se encuentran activos todos los bloques *Mux. Accumulator*, siendo estos los que representan el mayor consumo energético

y, a pesar de seleccionar pocos pesos diferentes a lo largo del procesamiento, la actividad constante en el acumulador produce un notable incremento en los valores medidos (indirectamente) de potencia.

En cuanto al valor mínimo de consumo medido para el acelerador, este ocurre con *kernel* 5×5 (8 bits de entradas y 1 de parámetros, para todos los casos de *stride*). Esto se debe a que, en primer lugar, al utilizar el modo de operación para *kernel* grandes, la mayoría de los bloques *Mux. Accumulator* en los PE se encuentran "apagados" (*clock-gating*). En segundo lugar, la disminución del consumo para este caso es ocasionada por la baja actividad de los nodos del circuito con respecto al tiempo de procesamiento: con 8 bit de entradas se requieren de al menos 256 ciclos de reloj (sin contar latencias) mientras que la selección de pesos es de, como máximo, 25 pesos (máxima actividad de $^{25}/_{256}$), por lo que la mayoría del tiempo se mantiene el mismo peso seleccionado. Esta configuración de capa también reduce el consumo del acelerador ya que, al ser pesos de 1 bit, no hay extensiones de signo y el único cambio posible en el peso seleccionado es del bit menos significativo, quedando los demás bits fijos en valor 0. Esto también influye en el acumulador ya que solamente se suma 1 o se mantiene el valor de acumulación, minimizando el consumo en dicho sub-bloque.

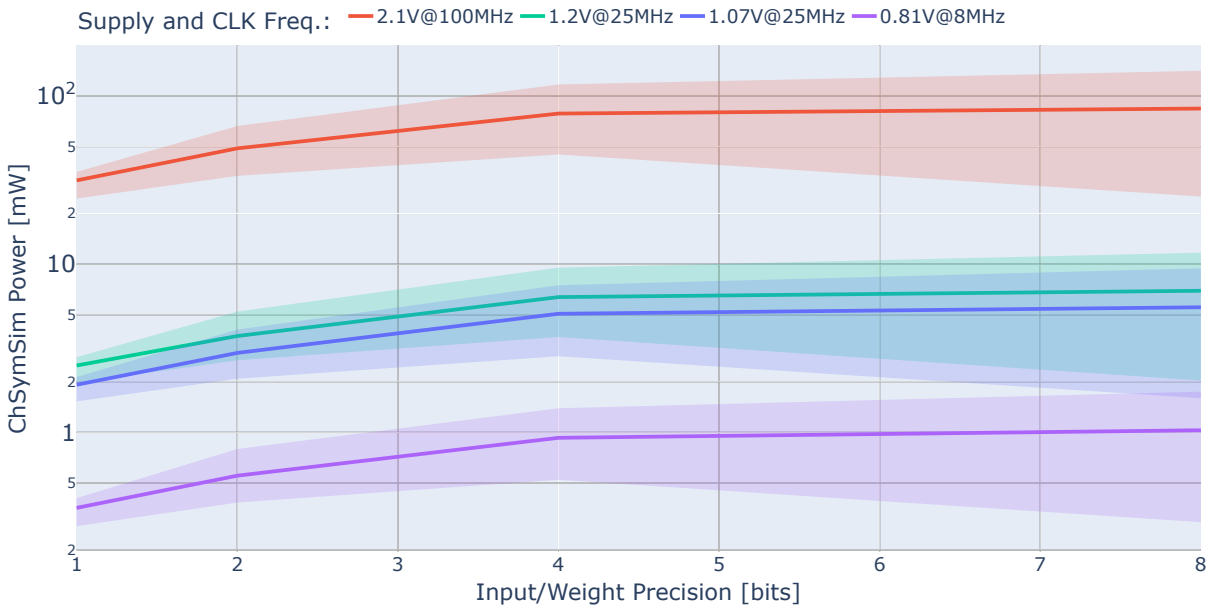
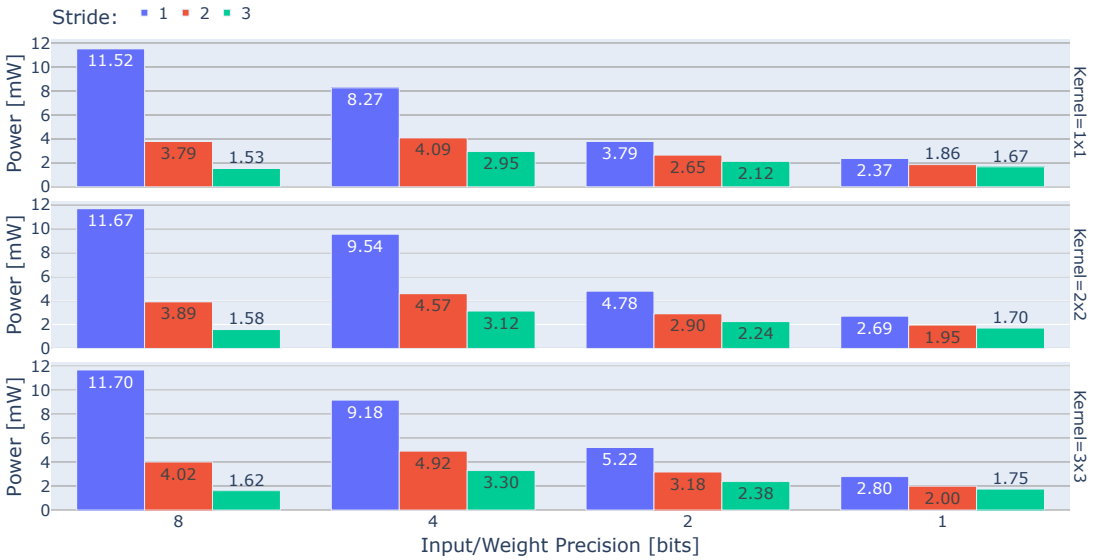


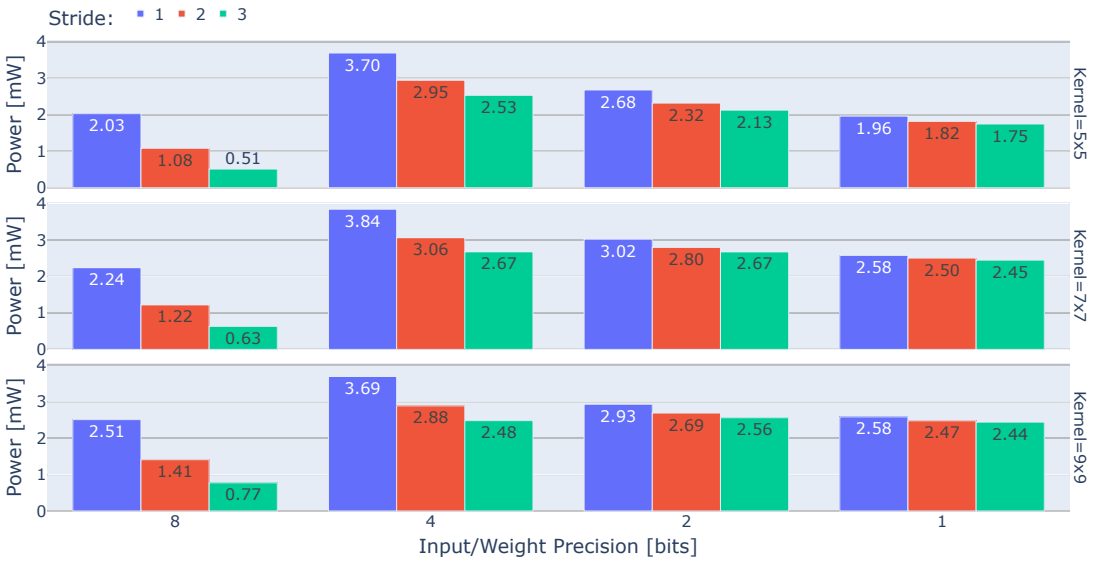
Figura 5.20: Consumo de potencia promedio del acelerador ChSymSim integrado en DigineuronV3a, con *stride* 1 y diversos tamaños de *kernel*.

En la Fig. 5.20 se presenta el consumo de potencia promedio del procesador ChSymSim integrado en el SoC DigineuronV3a, en base a distintos valores de precisión (la misma para pesos y entradas), tensión de alimentación y frecuencia de reloj, para los cuales se realizaron las mediciones presentadas en la Tabla 5.5. En estas curvas de consumo medio, que incluyen los rangos máximos y mínimos calculados en todos los tamaños de *kernel* evaluados (con *stride* 1), se puede observar que el comportamiento del consumo al variar la precisión de entradas y pesos se mantiene para todos los casos de tensión/frecuencia (que solamente difieren en un factor de escala). A partir de 4 bits de entradas y parámetros, este consumo medio se mantiene casi constante, mientras que con 8 bits el rango mínimo decrece, debido a la reducción en la actividad mencionada anteriormente: procesamiento con muchos ciclos de reloj pero pocas variaciones en los nodos del circuito. En cuanto a los efectos en el consumo de potencia en el acelerador ante variaciones de *stride*, ilustrado en la Fig. 5.21, se observa una notable reducción en dicho consumo al aumentar el valor de *stride*. Esto está relacionado al hecho de que con *stride* 1 todos los elementos de los PEs se encuentran activos, obteniendo así el mayor consumo de potencia, mientras que con *stride* 2 la mitad de los PEs son deshabilitados y con *stride* 3 sólo un tercio de los PEs son utilizados para la ejecución de la función ChSymSim. Este efecto adquiere más notoriedad con precisiones de 8 y 4 bits, donde el acelerador se encuentra realizando cómputos en casi todos los ciclos de reloj y por lo tanto el consumo es mayor. Para el caso particular de *kernel* grandes, en la Fig. 5.21b se puede apreciar que esta variación en el consumo debido al valor de *stride* es menor. Este modo de operación, aún para *stride* 1, cuenta con un número de PEs habilitados considerablemente inferior al de el cómputo de *kernels* pequeños (Fig. 5.21a), por lo que "apagar" unos pocos PEs al aumentar el valor de *stride* no produce cambios tan significativos en el consumo energético.

Considerando tensión/frecuencia nominal (1,2V@25MHz) y cómputos de la función ChSymSim con *stride* 1, las mediciones indirectas de potencia y posteriores cálculos de energía para el acelerador integrado en DigineuronV3a se muestran en las Figs. 5.22 y 5.23, respectivamente. Los resultados de energía del procesador ChSymSim se hallan utilizando el tiempo que toma el procesamiento de un canal de entrada, que se multiplica por el valor de consumo de potencia correspondiente.



(a)



(b)

Figura 5.21: Consumo de potencia del acelerador ChSymSim integrado en DigineuronV3a, para tensión/frecuencia nominales de 1,2V@25MHz, con diversos valores de *stride* y *kernel*: a) 1×1 a 3×3; b) 5×5 a 9×9.

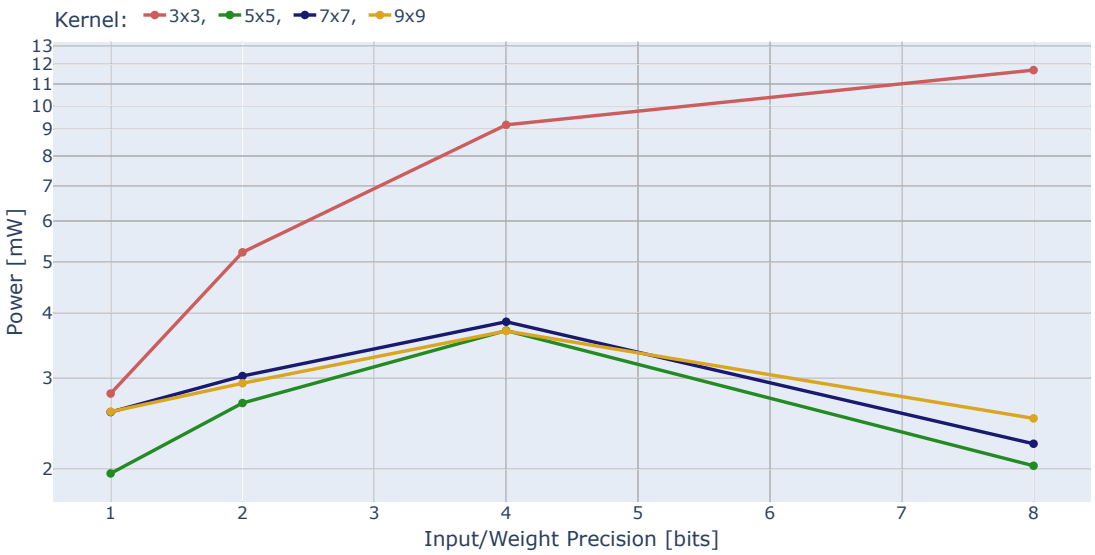


Figura 5.22: Consumo de potencia del acelerador ChSymSim integrado en DigineuronV3a, para tensión/frecuencia nominales de 1,2V@25MHz y *stride* 1.

Para esto se promedia el tiempo de cómputo total (incluyendo carga de datos), es decir, el tiempo hasta obtener la salida final de la función ChSymSim ante una entrada de 11×11 elementos (tamaño del arreglo de entradas), dividiendo dicho valor por el número de canales de entrada procesados (64 en este caso). En ambas Figs. 5.22 y 5.23 pueden observarse comportamientos de potencia y energía en función de la precisión de entradas, que resultan similares a los predichos en el Capítulo 4 (especialmente en las simulaciones pos-síntesis), aunque con valores ligeramente superiores a lo obtenido por simulación. Dichas diferencias en los valores de potencia y energía medidos/calculados (en promedio 1,78mW y 5,40nJ mayores) con respecto a lo simulado se deben principalmente a que el modelo post-síntesis simulado carece de ubicación para las compuertas lógicas y líneas de metal que las conecten (*wires*), ignorando además las celdas *buffer* añadidas durante el proceso de fabricación para satisfacer integridad de señal y tiempos de *hold*.

A partir de las mediciones de potencia presentadas anteriormente se calculan otras métricas de desempeño, tales como rendimiento (*throughput*) y eficiencia energética, las cuales se hallan resumidas en Tabla K.1, a la vez que se detallan en el Apéndice K para las distintas variaciones de tensión/frecuencia, tamaño de *kernel*, *stride* y precisión. Para estos cálculos de desempeño del procesador ChSymSim en DigineuronV3a, la operación general (OP) se define como la cantidad de sumas de

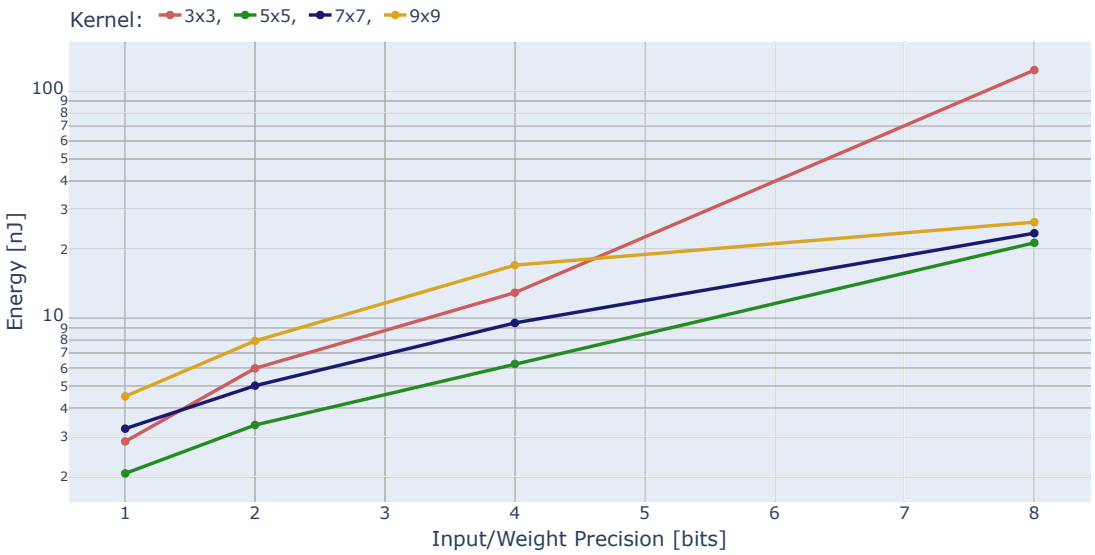


Figura 5.23: Consumo energético del acelerador ChSymSim integrado en DigineuronV3a, para tensión/frecuencia nominales de 1,2V@25MHz y *stride* 1.

valores con 8 bits de precisión, realizadas en un ciclo de procesamiento de canal. Dada la complejidad de la función ChSymSim, el cálculo de OP se realiza mediante la conversión de todas las operaciones internas, desde la aplicación de máscara a las entradas comparadas (AND por bit), pasando por el ordenamiento (*sort*), las diferencias de los valores ordenados (valores μ), multiplicación por pesos simétricos (o coeficientes c) y finalmente la suma de los productos parciales ($\mu \times c$). Si bien la cantidad de OP para estos tipos de cómputos pueden ser normalizadas a número de sumas con cierta facilidad, generalmente como el producto de la cantidad de entradas por la precisión correspondiente (escalada a 8 bits), la operación de ordenamiento resulta más compleja. Es por esto que la función *sort* se aproxima como la longitud (cota superior o *upper bound*) de las redes de ordenamiento, considerando los resultados de [67] para 1 a 17 entradas, lo obtenido por el software SorterHunter de [68] para 18 a 32 entradas, y finalmente realizando una aproximación lineal de estos comportamientos para número de entradas superiores. Por otra parte, las multiplicaciones y acumulaciones (MAC) equivalentes son normalizadas a cómputos con 8 bits de entradas y pesos, por lo que una MAC de 4 bits (tanto para entradas como para pesos) corresponde a $^1/_4$ de una MAC de 8 bits, mientras que una MAC de 2 bits representa $^1/_{16}$. En ambos casos de número de OP y MAC equivalentes, estos se computan para un ciclo de procesamiento de canal, en función del número

de entradas/pesos y precisión de los mismos, por lo que tanto la acumulación de los canales procesados como la suma del *bias* no están consideradas en estos cálculos. De igual forma que para el cálculo de energía, para obtener los valores de rendimiento y eficiencia, se escalan el número de OP y MAC por el correspondiente tiempo de procesamiento de canal, según la precisión de las entradas y considerando los retardos introducidos por carga de datos al acelerador.

Métrica \ V_{DD}@f_{clk}	2,1V@100MHz		1,2V@25MHz		1,07V@25MHz		0,81V@8MHz	
	min	Max	min	Max	min	Max	min	Max
Energía [nJ]	6,44	371,19	2,06	123,09	1,61	99,38	0,91	57,38
Energía por OP [pJ]	0,58	46,30	0,19	14,72	0,14	11,84	0,08	6,77
Rendimiento [GOPS]	1,72	104,80	0,43	26,20	0,43	26,20	0,14	8,38
Rendimiento [MMACS]	30,80	6124,84	7,70	1531,21	7,70	1531,21	2,46	489,99
Eficiencia [TOPS/W]	0,02	1,73	0,07	5,31	0,08	6,90	0,15	12,12
Eficiencia [GMACS/W]	0,39	78,00	1,21	226,33	1,51	288,97	2,64	498,50

Tabla 5.6: Medidas de energía, desempeño y eficiencia (resumidas) del acelerador ChSymSim en el SoC DigineuronV3a, considerando *stride* 1.

	V3a		V2	V1
Tecnología (TSMC) [nm]	65		65	65
Área [mm^2]	$2{,}00 \times 4{,}50$		$1{,}25 \times 1{,}25$	$1{,}25 \times 1{,}25$
V_{DD} *core* [V]	1,2		1,2	1,2
Frecuencia [MHz]	25		100	100
Tamaño máximo de *kernel*	9×9	3×3	7×7	5×5
Consumo SoC [mW]	7,91	16,62	10,41	14,89
Consumo ChSymSim [mW]	2,51	11,70	0,73	0,68
Energía ChSymSim por OP [pJ]	0,24	1,32	0,27	0,49
Rendimiento [GOPS]	10,44	8,87	2,68	1,38
Rendimiento [MMACS]	554,37	554,37	19,53	10,16
Eficiencia ChSymSim [TOPS/W]	4,17	0,76	3,66	2,03
Eficiencia ChSymSim [GMACS/W]	221,23	47,38	26,72	14,94
Precisión [bits]	8-1		8	8

Tabla 5.7: Comparativa entre los SoCs fabricados: DigineuronV1, V2 y V3a, donde "OPs" están normalizadas a sumas de 8 bits.

Con el fin de realizar una comparación entre las diferentes versiones del procesador ChSymSim (PE de prueba/prototipo y el acelerador final como arreglo de PEs) integrados en los SoCs fabricados en esta tesis, en la Tabla 5.7 se muestra un resumen de los datos de consumo de potencia (medidos) y otras métricas de desempeño

calculadas a partir de estos. Para todos los casos, se midió el consumo de potencia de los *chips* (y acelerador de forma indirecta) al procesar (en bucle) datos aleatorios de 8 bits (tanto entradas como pesos), con tensión de alimentación y frecuencias nominales. Tanto los aceleradores de DigineuronV1 como V2 fueron medidos utilizando al máximo sus recursos, es decir, computando una cantidad de entradas igual al tamaño máximo de *kernel* que soportan. Por esta razón, y para realizar una comparación más justa, las mediciones del acelerador en DigineuronV3a se tomaron de los experimentos con *stride* 1 y *kernel* máximos: 3×3 y 9×9 para modos de *kernel* pequeños y grandes, respectivamente. De esta forma, al computar *kernels* de tamaño 9×9, el procesador ChSymSim integrado en el SoC DigineuronV3a representa una mejora sustancial en cuanto a rendimiento y eficiencia energética en relación a las versiones anteriores de V1 y V2. Por otra parte, la operación de *kernels* de tamaño 3×3 en el acelerador integrado en el *chip* V3a también presenta un rendimiento superior a sus versiones anteriores, pero con menor eficiencia energética en cuanto al cómputo simplicial (OP) dada la reducida cantidad de operaciones que representa el ordenamiento de sólo 3×3 entradas, a pesar de generarse muchos resultados en paralelo. En cuanto al cómputo de operaciones MAC equivalentes con *kernels* de 3×3 entradas, el operador ChSymSim en V3a mantiene su superioridad respecto a los *chips* V1 y V2, aunque sin llegar a ser tan eficiente como con el cómputo de *kernel* grandes (9×9).

En la Tabla 5.8 se muestra una comparación del consumo y desempeño promedio del procesador ChSymSim en DigineuronV3a, con otros aceleradores de CNNs del estado del arte integrados en *chips* de la misma tecnología. Por un lado, para realizar la comparación de los SoCs se recurre al uso de unidades para el número de operaciones como las MACs (de 8 bits) equivalentes, ya que las redes tradicionales usan simples productos y sumas. Sin embargo, debido a que las funciones ChSymSim son notablemente más complejas que MACs tradicionales, pudiendo obtener mejores desempeños que las convoluciones en los ejemplos de redes mostradas en el Capítulo 3, la Tabla 5.8 también incluye una comparación en cuanto a OPS para el rendimiento y OPS/W para la eficiencia de los aceleradores. De esta forma, el número de operaciones (OP) se normaliza como sumas de 8 bits, realizando los cálculos correspondientes a partir de las MACs en los procesadores del estado del arte.

	DigineuronV3a		Eyeriss v2 [26]	DNPU [28]
Tecnología	65nm		65nm	65nm
Área [mm^2]	2,00 × 4,50		N/A	4,00 × 4,00
V_{DD} *core* [V]	1,07	0,81	N/A	0,77
Frecuencia [MHz]	25	8	200	50
Consumo [mW]	3,05	0,57	617,49	30,25
Rendimiento [MMACS]	232,70	74,46	68520,28	32417,88
Eficiencia [GMACS/W]	84,81	145,85	111,77	1066,92
Rendimiento [GOPS]	7,14	2,29	685,20	389,01
Eficiencia [TOPS/W]	2,72	4,69	1,12	12,80
Precisión (entradas/pesos) [bits]	8-1		8	4

Tabla 5.8: Comparación de consumo y desempeño (promedio) de aceleradores para Redes Neuronales en SoCs.

A partir de los valores presentados en la Tabla 5.8, se observa que el diseño de acelerador propuesto por esta tesis (integrado en DigineuronV3a) posee un consumo significativamente inferior al de otros *chips* en la misma tecnología. A pesar de que el rendimiento general del procesador ChSymSim es muy inferior al de los demás SoCs, este extremadamente bajo consumo le permite ser más eficiente que otros aceleradores del estado del arte, superando a *chips* como Eyeriss v2 [26]. En el caso de DNPU [28], su eficiencia es considerablemente mayor debido a que su unidad dedicada a CNNs sólo puede realizar operaciones con una precisión de 4 bits (entradas y pesos), por lo que su diseño es mucho más simple. Si por el contrario se realizan convoluciones de 8 bits, la eficiencia general del *chip* DNPU disminuiría notablemente al forzar el uso del acelerador MLP (diseñado específicamente para capas FC de 16 bits). En términos de OPS y OPS/W, la complejidad de la función ChSymSim incrementa aún más la eficiencia del acelerador propuesto por esta tesis, pudiendo acercarse a la de DNPU en su valor máximo de 12,12 TOPS/W con 0,81V@8MHz (Tabla K.1).

En la Tabla 5.9 se presenta una comparación del consumo y desempeño (promedio) del acelerador ChSymSim en DigineuronV3a con otros diseños de procesadores simpliciales simétricos, desarrollados por el grupo de investigación previo al inicio del desarrollo de esta tesis. Para realizar esta comparación, los valores de consumo y eficiencia fueron escalados a la tecnología de 65nm (manteniendo valores originales

entre paréntesis), ya que ambos estos diseños previos fueron integrados en 55nm, por medio de un factor de escala de $(65/55)^2 \times 65/55$ para tener en cuenta tanto el consumo dinámico (capacidad del circuito) y el consumo por *leakage*. Por otro lado, los valores de rendimiento y, por lo tanto, eficiencia reportados por MORPHO1SYM [36] fueron hallados considerando operaciones lógicas de 1 bit, por lo que estos también fueron escalados (factor de 1/8) a operaciones de 8 bits, para poder ser comparables con DigineuronV3a y MORPHO8PWL [35]. En base a los valores mostrados en la Tabla 5.9 se puede observar que el diseño presentado por esta tesis (integrado en DigineuronV3a) supera ampliamente el desempeño logrado por el procesador simplicial simétrico de 8 bits en MORPHO8PWL [35], que representa un diseño más cercano al propuesto en esta tesis. Por el contrario, el procesador de 1 bit en MORPHO1SYM [36] consigue un desempeño y eficiencia superiores, debido a que al estar limitado a mínimas precisiones, los circuitos implementados son considerablemente más sencillos, por lo que estos pueden además ser ejecutados con mayores frecuencias de reloj y a menores tensiones de alimentación. De esta forma, el diseño de acelerador ChSymSim propuesto por esta tesis sacrifica parte de su desempeño para lograr una mayor variedad de funciones y configuraciones que éste puede ejecutar, ubicándose en un punto medio en cuanto a eficiencia y rendimiento, en comparación con diseños previos que implementan funciones simpliciales simétricas.

	DigineuronV3a		MORPHO8PWL [35]	MORPHO1SYM [36]
Tecnología	65nm		55nm	55nm
Área [mm^2]	2,00 × 4,50		0,76 × 1,13	0,85 × 0,65
V_{DD} *core* [V]	1,07	0,81	0,5	0,6
Frecuencia [MHz]	25	8	1	75
Consumo [μW]	3049,57	567,86	12,18 (7,38)	4951,92 (3000)
Rendimiento [MOPS]	7143,57	2285,94	21,80	110160
Eficiencia [TOPS/W]	2,72	4,69	1,79 (2,95)	22,25 (36,72)
Precisión (entradas) [bits]	8-1		8	1
Precisión (pesos) [bits]	8-1		3	1

Tabla 5.9: Comparación de consumo y desempeño (promedio) de aceleradores simpliciales y simétricos.

5.4. Conclusiones

En este capítulo se presentaron los circuitos integrados fabricados para evaluar el diseño del acelerador ChSymSim, con las dos primeras versiones como prototipos (DigineuronV1 y V2) para analizar principalmente el sistema y las estructuras de prueba. Luego de evaluar el comportamiento de los PEs simpliciales simétricos integrados en los prototipos, se fabricó un SoC mucho más complejo (DigineuronV3a), que incluye la versión definitiva del operador ChSymSim y módulos auxiliares de transferencia de datos, optimizados para su uso conjunto con el acelerador. La funcionalidad de los *chips* y sus bloques fueron comprobadas, al mostrar finalmente los resultados de la ejecución de algunas capas de una red neuronal, en los que se obtienen los mismos valores tanto con los modelos de PyTorch entrenables como con el acelerador ChSymSim en DigineuronV3a.

A partir de los resultados experimentales obtenidos al evaluar los distintos circuitos integrados fabricados durante el trabajo de esta tesis, se pudo comprobar el desempeño del procesador ChSymSim, tanto en su versión final integrada en DigineuronV3a como en sus versiones de prueba con un solo PE en los *chips* prototipos. Los resultados obtenidos en las pruebas y mediciones muestran un comportamiento similar a lo predicho por los modelos y simulaciones post-síntesis realizadas en el Capítulo 4, con valores de consumo en el orden de unos pocos mW en condiciones nominales y ligeramente más elevados, debido a la inclusión de lógica adicional (*buffers*) y líneas de metal durante el proceso de P&R.

En cuanto al desempeño de los aceleradores ChSymSim integrados, su bajo consumo, en combinación con la cantidad de operaciones por segundo que pueden realizar, consiguen eficiencias energéticas del orden de TOPS/W (o GMACS/W equivalentes), lo cual mejora considerablemente con bajas tensiones de alimentación a costa de reducir la frecuencia de operación. Esto es especialmente evidente en la versión final del acelerador integrado en DigineuronV3a que, gracias a una inteligente expansión del arreglo de PEs y las optimizaciones implementadas en el diseño de los mismos, consigue un buen compromiso entre eficiencia energética y capacidad de realizar operaciones ChSymSim con variadas configuraciones de *kernel*, *stride* y *padding*, sin necesidad de añadir otras estructuras de cómputo.

Capítulo 6

Conclusiones

En esta tesis se ha profundizado sobre la familia de funciones simpliciales, en especial las simétricas, por su reducción en la cantidad de memoria necesaria para almacenar parámetros. Estas funciones han sido utilizadas previamente como una alternativa para la ejecución eficiente de aceleradores de baja complejidad, particularmente implementaciones de filtros espaciales [33, 35, 36]. Debido a las limitaciones en funcionalidad que las implementaciones simétricas presentan, la primera parte del trabajo de esta tesis consistió en evaluar y mejorar los algoritmos simpliciales, teniendo en cuenta su posterior implementación en circuitos integrados, obteniendo de esta manera la función Simplicial Simétrica a Canales Separados (ChSymSim). Se desarrollaron las funciones de *backpropagation* necesarias para poder ajustar los parámetros de las capas ChSymSim, pudiendo integrarlas y entrenarlas en entornos de ML estándar como PyTorch, con lo que se lograron desempeños similares (o superiores en algunos casos) a los mismos modelos de DNNs con capas convolucionales tradicionales. Además de lo anteriormente mencionado, se aplicaron modelos de cuantización y re-entrenamiento para obtener Redes Neuronales con capas ChSymSim que puedan ser implementadas directamente en *hardware*, reduciendo la degradación de su desempeño por el uso de precisiones reducidas.

A partir del algoritmo ChSymSim desarrollado en esta tesis, se diseñó una arquitectura de procesador como arreglo de múltiples Elementos de Procesamiento (PE), explorando diferentes tamaños y configuraciones de cómputo en paralelo. Mediante modelos de alto nivel se pudo optimizar la arquitectura propuesta para el cómputo

de capas en CNNs, haciendo un uso más eficiente de sus recursos en *hardware*, para diversas configuraciones de *kernel*, *stride* y *padding*. Con dichos modelos se logró un desarrollo más eficiente en cuanto micro-arquitectura, al poder identificar y optimizar los bloques de procesamiento simplicial que representan un mayor impacto en área y consumo energético. La interconexión entre PEs para la reutilización de bloques de cómputo en diferentes modos de operación, junto con un apropiado manejo de sus habilitaciones y un uso/transferencia de datos inteligente, representan los factores fundamentales para lograr una mayor eficiencia energética. En base al diseño y optimizaciones analizadas en esta tesis, se comparó el desempeño del diseño de operador ChSymSim propuesto con una arquitectura similar pero con unidades de procesamiento lineales convencionales (multiplicadores y acumuladores). Los resultados de los modelos de alto nivel y simulaciones post-síntesis de ambos diseños mostraron que, para configuraciones de *kernel* grandes y precisión de entradas baja, el acelerador ChSymSim resulta superior al convolucional.

Finalmente, se realizaron dos prototipos de sistemas en circuitos integrados (SoC), fabricados con tecnología CMOS 65nm, en los que se incluyeron las versiones preliminares de las unidades de procesamiento ChSymSim y con los que se pudo depurar/analizar el desempeño de la implementación del PE al interactuar con sistemas reales. La fabricación de *chips* en esta tesis concluyó con una versión final de SoC (también en 65nm), integrando la arquitectura completa del acelerador ChSymSim con múltiples unidades de procesamiento e implementando todas las mejoras resultantes de los análisis con los modelos de alto nivel, simulaciones post-síntesis y pruebas con los *chips* prototipos. Mientras que los aceleradores de prueba del procesador ChSymSim fueron evaluados con datos aleatorios, la arquitectura final integrada en el último SoC fue verificada además con datos de entrada y capas de una Red Neuronal cuantizada, obteniendo resultados idénticos a los modelos en *software* que fueron entrenados en PyTorch. En cuanto a consumo de potencia y desempeño, el *chip* que concluye esta tesis presentó un consumo de unas decenas de mW (en condiciones nominales) y eficiencias energéticas del orden de TOPS/W para cómputos simpliciales de 8 bits de entradas y parámetros, lo que representa un buen compromiso entre eficiencia energética y versatilidad del acelerador para el cómputo de diferentes configuraciones de capas. Comparado con otros *chips* del

estado del arte, el acelerador ChSymSim propuesto supera en eficiencia a los diseños que fueron desarrollados para operaciones tradicionales de 8 bits, mientras que se encuentra un poco por debajo de aquellos procesadores que por su simplicidad están limitados a precisiones muy bajas.

A partir de los resultados y conclusiones obtenidos, se pueden vislumbrar dos direcciones posibles para continuar con la línea de investigación. La exploración de arquitecturas de cómputo realizada se enfocó en mayor medida en la optimización de la micro-arquitectura del operador ChSymSim, empleando interconexiones entre PE y flujos de datos (*dataflows*) tradicionales. A pesar de que para algunas decisiones de diseño se tuvieron en cuenta interacciones a nivel de sistema (dimensiones de capa, compresión de datos de entrada/salida, interacción con controladores externos, etc.), el trabajo de esta tesis deja abierta la posibilidad para la exploración de estrategias de optimización en cuanto a macro-arquitectura más avanzadas, como por ejemplo el uso de redes-en-*chip* (NoC[1]) para mejorar la comunicación entre PEs, operaciones con múltiples núcleos de procesamiento (cómputo de capas de la red en paralelo), consideración de datos de entrada *sparse*, entre otras. Por otro lado, la combinación de unidades de procesamiento lineal (multiplicadores) y posteriores PEs simpliciales simétricos permitiría la implementación de algoritmos más complejos. Debido a que la función simétrica incluye la suma como una opción posible, además de otras operaciones típicas de CNNs como máximo, mínimo, mediana, etc., el agregar la multiplicación por pesos no-binarios a las entradas (con su correspondiente escalado para mantener la precisión) posibilita el diseño de un acelerador mucho más versátil que lo desarrollado en esta tesis.

[1] *Network-on-Chip*

Bibliografía

[1] GeeksforGeeks. (2023) Top hat and black hat transform using python-opencv. [Online]. Available: https://www.geeksforgeeks.org/top-hat-and-black-hat-transform-using-python-opencv/

[2] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "The German Traffic Sign Recognition Benchmark: A multi-class classification competition," in *IEEE International Joint Conference on Neural Networks*, 2011, pp. 1453–1460.

[3] P. Helber, B. Bischke, A. Dengel, and D. Borth, "Introducing EuroSAT: A novel dataset and deep learning benchmark for land use and land cover classification," in *IGARSS 2018-2018 IEEE International Geoscience and Remote Sensing Symposium*. IEEE, 2018, pp. 204–207.

[4] ——, "Eurosat: A novel dataset and deep learning benchmark for land use and land cover classification," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 2019.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034.

[6] Y. Yang and S. Newsam, "Bag-of-visual-words and spatial extensions for land-use classification," in *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. GIS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 270–279. [Online]. Available: https://doi.org/10.1145/1869790.1869829

[7] R. Ba, C. Chen, J. Yuan, W. Song, and S. Lo, "SmokeNet: Satellite smoke scene detection using convolutional neural network with spatial and channel-wise attention," *Remote Sensing*, vol. 11, no. 14, 2019. [Online]. Available: https://www.mdpi.com/2072-4292/11/14/1702

[8] N. Rodriguez, L. Ratschbacher, C. Xu, and P. Julian, "Exploration of deep neural networks with symmetric simplicial layers for on-satellite earth observation processing," in *2022 Argentine Conference on Electronics (CAE)*, 2022, pp. 31–36.

[9] N. Rodriguez, P. Julian, and M. Villemur, "Symmetric simplicial neural networks," in *2021 55th Annual Conference on Information Sciences and Systems (CISS)*, 2021, pp. 1–6.

[10] M. Megías, Z. Emri, T. Freund, and A. Gulyás, "Total number and distribution of inhibitory and excitatory synapses on hippocampal CA1 pyramidal cells," *Neuroscience*, vol. 102, no. 3, pp. 527–540, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306452200004966

[11] L. Chua and T. Roska, "The CNN paradigm," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 40, no. 3, pp. 147–156, 1993.

[12] L. O. Chua, "CNN: Twenty years later," in *2008 International Conference on Communications, Circuits and Systems*, 2008, pp. 13–14.

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

[14] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv e-prints*, p. arXiv:1409.1556, Sep. 2014.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, may 2017. [Online]. Available: https://doi.org/10.1145/3065386

[16] L. Deng and D. Yu, *Deep Learning: Methods and Applications*. Now Foundations and Trends, 2014.

[17] E. Oyallon, E. Belilovsky, and S. Zagoruyko, "Scaling the scattering transform: Deep hybrid networks," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 5619–5628.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[19] TOP500, "TOP500 list - june 2024," 2024, accessed: 2024-09-12. [Online]. Available: https://www.top500.org/lists/top500/2024/06/

[20] ——, "GREEN500 list - june 2024," 2024, accessed: 2024-09-12. [Online]. Available: https://www.top500.org/lists/green500/2024/06/

[21] S. Wolfert, L. Ge, C. Verdouw, and M.-J. Bogaardt, "Big data in smart farming – a review," *Agricultural Systems*, vol. 153, pp. 69–80, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0308521X16303754

[22] A.-M. Rahmani, N. K. Thanigaivelan, T. N. Gia, J. Granados, B. Negash, P. Liljeberg, and H. Tenhunen, "Smart e-health gateway: Bringing intelligence to internet-of-things based ubiquitous healthcare systems," in *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, 2015, pp. 826–834.

[23] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to End Learning for Self-Driving Cars," *arXiv*, Apr. 2016.

[24] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 246–247.

[25] L. Cavigelli and L. Benini, "Origami: A 803-gop/s/w convolutional network accelerator," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 11, pp. 2461–2475, 2017.

[26] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[27] J. Sim, J.-S. Park, M. Kim, D. Bae, Y. Choi, and L.-S. Kim, "14.6 a 1.42tops/w deep convolutional neural network recognition processor for intelligent ioe systems," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, 2016, pp. 264–265.

[28] D. Shin, J. Lee, J. Lee, J. Lee, and H.-J. Yoo, "Dnpu: An energy-efficient deep-learning processor with heterogeneous multi-core architecture," *IEEE Micro*, vol. 38, no. 5, pp. 85–93, 2018.

[29] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultralow power binary-weight cnn acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 48–60, 2018.

[30] P. Julian, A. Desages, and O. Agamennoni, "High-level canonical piecewise linear representation using a simplicial partition," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 46, no. 4, pp. 463–480, 1999.

[31] P. Julian, R. Dogaru, and L. Chua, "A piecewise-linear simplicial coupling cell for CNN gray-level image processing," in *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)*, vol. 3, 2001, pp. 109–112 vol. 2.

[32] P. Mandolesi, P. Julian, and A. Andreou, "A scalable and programmable simplicial CNN digital pixel processor architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 5, pp. 988–996, 2004.

[33] M. Di Federico, P. Julián, and P. S. Mandolesi, "SCDVP: A simplicial CNN digital visual processor," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 7, pp. 1962–1969, 2014.

[34] M. Di Federico, P. Julián, A. G. Andreou, and P. S. Mandolesi, "Fully functional fine-grain vertically integrated 3d focal plane neuromorphic processor," in *2014 SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, 2014, pp. 1–2.

[35] M. Villemur, P. Julian, T. Figliola, and A. G. Andreou, "Neuromorphic cellular neural network processor for intelligent internet-of-things," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–4.

[36] M. Villemur, P. Julian, and A. G. Andreou, "Energy aware simplicial processor for embedded morphological visual processing in intelligent internet of things," *Electronics Letters*, vol. 54, no. 7, pp. 420–422, 2018. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/el.2017.4738

[37] M. Villemur, "Estructuras de procesamiento neuromórfico de bajo consumo para sistemas de visión en internet de las cosas," PhD dissertation, Universidad Nacional del Sur, 2019. [Online]. Available: https://repositoriodigital.uns.edu.ar/handle/123456789/4634

[38] N. Rodriguez, P. Julian, and E. Paolini, "A simplicial piecewise linear approach for efficient hardware realization of neural networks : (invited presentation)," in *2019 53rd Annual Conference on Information Sciences and Systems (CISS)*, 2019, pp. 1–3.

[39] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[40] N. Rodriguez, P. Julian, and E. Paolini, "Function approximation using symmetric simplicial piecewise-linear functions," in *2019 XVIII Workshop on Information Processing and Control (RPIC)*, 2019, pp. 292–297.

[41] P. Julian, A. G. Andreou, M. Villemur, and N. Rodriguez, "Simplicial computation: A methodology to compute vector–vector multiplications

with reduced complexity," *International Journal of Circuit Theory and Applications*, vol. 49, no. 11, pp. 3766–3788, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cta.3128

[42] N. Rodriguez, M. Villemur, and P. Julian, "Architecture analysis for symmetric simplicial deep neural networks on chip," in *2023 57th Annual Conference on Information Sciences and Systems (CISS)*, 2023, pp. 1–6.

[43] N. Rodriguez, M. Villemur, D. Klepatsch, D. G. Ivanovich, and P. Julián, "System on chip testbed for deep neuromorphic neural networks," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023, pp. 1–5.

[44] N. Rodriguez, D. G. Ivanovich, M. Villemur, and P. Julian, "RISC-V based SoC platform for neural network acceleration," in *2024 Argentine Conference on Electronics (CAE)*, 2024, pp. 142–147.

[45] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[46] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3354–3361.

[47] S. Xu, Y. Cheng, K. Gu, Y. Yang, S. Chang, and P. Zhou, "Jointly Attentive Spatial-Temporal Pooling Networks for Video-based Person Re-Identification," *arXiv e-prints*, p. arXiv:1708.02286, Aug. 2017.

[48] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds. Cham: Springer International Publishing, 2015, pp. 234–241.

[49] D. Eigen and R. Fergus, "Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 2650–2658.

[50] T.-S. Vu, M.-Q. Ha, D.-N. Nguyen, V.-C. Nguyen, Y. Abe, T. Tran, H. Tran, H. Kino, T. Miyake, K. Tsuda, and H.-C. Dam, "Towards understanding structure–property relations in materials with interpretable deep learning," *npj Computational Materials*, vol. 9, no. 1, p. 215, Dec 2023. [Online]. Available: https://doi.org/10.1038/s41524-023-01163-9

[51] M. Reichstein, G. Camps-Valls, B. Stevens, M. Jung, J. Denzler, N. Carvalhais, and Prabhat, "Deep learning and process understanding for data-driven earth system science," *Nature*, vol. 566, no. 7743, pp. 195–204, Feb 2019. [Online]. Available: https://doi.org/10.1038/s41586-019-0912-1

[52] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65 6, pp. 386–408, 1958. [Online]. Available: https://api.semanticscholar.org/CorpusID:12781225

[53] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ser. ICML'10. Madison, WI, USA: Omnipress, 2010, p. 807–814.

[54] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018. [Online]. Available: http://arxiv.org/abs/1801.04381

[55] E. Phaisangittisagul, "An analysis of the regularization between l2 and dropout in single hidden layer neural network," in *2016 7th International Conference on Intelligent Systems, Modelling and Simulation (ISMS)*, 2016, pp. 174–179.

[56] M. D. Tissera and M. D. McDonnell, "Modular expansion of the hidden layer in single layer feedforward neural networks," in *2016 International Joint Conference on Neural Networks (IJCNN)*, 2016, pp. 2939–2945.

[57] D. P. Kingma, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[58] A. Noor, Y. Zhao, R. Khan, L. Wu, and F. Y. Abdalla, "Median filters combined with denoising convolutional neural network for gaussian and impulse noises," *Multimedia Tools and Applications*, vol. 79, no. 25, pp. 18 553–18 568, Jul 2020. [Online]. Available: https://doi.org/10.1007/s11042-020-08657-4

[59] K. Nogueira, J. Chanussot, M. Dalla Mura, and J. A. dos Santos, "An Introduction to Deep Morphological Networks," *arXiv e-prints*, p. arXiv:1906.01751, Jun. 2019.

[60] Y. Shen, X. Zhong, and F. Y. Shih, "Deep Morphological Neural Networks," *arXiv e-prints*, p. arXiv:1909.01532, Sep. 2019.

[61] M.-J. Chien and E. Kuh, "Solving nonlinear resistive networks using piecewise-linear analysis and simplicial subdivision," *IEEE Transactions on Circuits and Systems*, vol. 24, no. 6, pp. 305–317, 1977.

[62] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Journal of Machine Learning Research - Proceedings Track*, vol. 9, pp. 249–256, 01 2010.

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

[64] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms.

[65] C. Das and J. McLean, "The EUROPRACTICE mini@sic program," in *Proceedings 2003 IEEE International Conference on Microelectronic Systems Education. MSE'03*, 2003, pp. 45–46.

[66] O. Group, "CORE-V CV32E40P RISC-V IP," 2023. [Online]. Available: https://github.com/openhwgroup/cv32e40p

[67] M. Codish, L. Cruz-Filipe, M. Frank, and P. Schneider-Kamp, "Twenty-Five Comparators is Optimal when Sorting Nine Inputs (and Twenty-Nine for Ten)," *arXiv e-prints*, p. arXiv:1405.5754, May 2014.

[68] B. Dobbelaere, "SorterHunter," 2022. [Online]. Available: https://github.com/bertdobbelaere/SorterHunter

[69] N. Weiss, P. Holmes, and M. Hardy, *A Course in Probability*. Pearson Addison Wesley, 2005.

Apéndice A

Propiedades útiles de Esperanza y Varianza

Esperanza

La esperanza (o media) de una variable aleatoria discreta se define como

$$\mathrm{E}\left(X\right) = \sum_{x} x\mathrm{P}\left(X = x\right), \tag{A.1}$$

mientras que para una variable continua se calcula como

$$\mathrm{E}\left(X\right) = \int_{-\infty}^{\infty} x f_{dist}\left(x\right) dx \tag{A.2}$$

Propiedades de Esperanza:

$$\mathrm{E}\left(a\right) = a,\ a = \mathrm{cte} \tag{A.3}$$

$$\mathrm{E}\left(aX\right) = a\mathrm{E}\left(X\right),\ a = \mathrm{cte} \tag{A.4}$$

$$\mathrm{E}\left(X + Y\right) = \mathrm{E}\left(X\right) + \mathrm{E}\left(Y\right) \tag{A.5}$$

La esperanza (media) del producto de dos variables aleatorias independientes es:

$$\mathrm{E}\left(XY\right) = \mathrm{E}\left(X\right)\mathrm{E}\left(Y\right) \tag{A.6}$$

Varianza

La varianza de una variable aleatoria puede hallarse como

$$\mathrm{Var}\left(X\right) = \left[x - \mathrm{E}\left(X\right)\right]^2 \mathrm{P}\left(X = x\right)$$

$$\mathrm{Var}\left(X\right) = \mathrm{E}\left(X^2\right) - \mathrm{E}\left(X\right)^2 \tag{A.7}$$

$$\mathrm{Var}\left(aX\right) = a^2 \mathrm{Var}\left(X\right) \tag{A.8}$$

La varianza de la suma de dos variables aleatorias es:

$$\mathrm{Var}\left(X + Y\right) = \mathrm{Var}\left(X\right) + 2\mathrm{Cov}\left(X, Y\right) + \mathrm{Var}\left(Y\right), \tag{A.9}$$

donde si estas son independientes $\mathrm{Cov}\left(X, Y\right) = 0$, por lo que la ecuación anterior resulta

$$\mathrm{Var}\left(X + Y\right) = \mathrm{Var}\left(X\right) + \mathrm{Var}\left(Y\right) \tag{A.10}$$

En cuanto al producto de dos variables aleatorias independientes, la varianza se calcula como

$$\mathrm{Var}\left(XY\right) = \mathrm{E}\left(X^2\right)\mathrm{E}\left(Y^2\right) - \left[\mathrm{E}\left(X\right)\mathrm{E}\left(Y\right)\right]^2 \tag{A.11}$$

Apéndice B

ReLU en variables aleatorias

B.1. Capas ReLU tradicionales

La ecuación de una función de activación de tipo ReLU es una de las no linealidades más utilizadas en redes neuronales. En su forma más simple puede escribirse como

$$f_{ReLU}(y) = \max(0, y) = \begin{cases} y & , \quad y \geq 0 \\ 0 & , \quad y < 0 \end{cases} .$$

Dada una variable aleatoria Y, que se asume con distibución normal de media cero y varianza $\mathrm{Var}(Y) = \sigma_y^2$, la media de la distribución resultante de $X = f_{ReLU}(Y)$ se obtiene como

$$\begin{aligned} \mathrm{E}(X) &= \int_{-\infty}^{\infty} x f_{dist}(x)\, dx \\ &= \int_{-\infty}^{0} 0 f_{dist}(y)\, dy + \int_{0}^{\infty} y f_{dist}(y)\, dy \\ &= \int_{0}^{\infty} \frac{y}{\sigma_y \sqrt{2\pi}} e^{-y^2/2\sigma_y^2} dy. \end{aligned}$$

Realizando el cambio de variable

$$\begin{aligned}\frac{y^2}{2\sigma_y^2} &= u \\ \frac{y}{\sigma_y^2}dy &= du \\ \frac{y}{\sigma_y}dy &= \sigma_y du,\end{aligned}$$

por lo que la integral anterior se resuelve como

$$\begin{aligned}\mathrm{E}\left(X\right) &= \int_0^\infty \frac{\sigma_y}{\sqrt{2\pi}} e^{-u} du \\ &= \frac{-\sigma_y}{\sqrt{2\pi}} e^{-u}\Big|_0^\infty \\ &= \frac{\sigma_y}{\sqrt{2\pi}} \left(e^0 - e^{-\infty}\right),\end{aligned}$$

donde finalmente el resultado de la esperanza de una variable aleatoria con distribución normal, con media cero y varianza σ_y^2, afectada por una ReLU, resulta

$$\mathrm{E}\left(X\right) \approx \frac{\sigma_y}{\sqrt{2\pi}}. \tag{B.1}$$

La esperanza del cuadrado, por otra parte, se calcula como

$$\begin{aligned}\mathrm{E}\left(X^2\right) &= \int_{-\infty}^{\infty} x^2 f_{dist}\left(x\right) dx \\ &= \int_0^\infty y^2 f_{dist}\left(y\right) dy \\ &= \int_0^\infty \frac{y^2}{\sigma_y\sqrt{2\pi}} e^{-y^2/2\sigma_y^2} dy,\end{aligned}$$

donde realizando los siguientes reemplazos de variables

$$\begin{aligned} u &= \frac{y}{\sigma_y\sqrt{2}} \\ 2\sigma_y^2 u^2 &= y^2 \\ du &= \frac{dy}{\sigma_y\sqrt{2}} \end{aligned}$$

$$\begin{aligned} v &= u^2 \\ v^{\frac{1}{2}} &= u \\ dv &= 2u\ du \end{aligned}$$

la integral anterior resulta

$$\begin{aligned} \mathrm{E}\left(X^2\right) &= \frac{2\sigma_y^2}{\sqrt{\pi}} \int_0^\infty u^2 e^{-u^2} du \\ &= \frac{\sigma_y^2}{\sqrt{\pi}} \int_0^\infty v^{\frac{1}{2}} e^{-v} dv \\ &= \frac{\sigma_y^2}{\sqrt{\pi}} \int_0^\infty v^{\frac{3}{2}-1} e^{-v} dv \\ &= \frac{\sigma_y^2}{\sqrt{\pi}}\ \Gamma\left(\frac{3}{2}\right) = \frac{\sigma_y^2}{\sqrt{\pi}} \frac{\sqrt{\pi}}{2}. \end{aligned}$$

Para este último resultado se utilizó la función gamma $\Gamma(t) = \int_0^\infty v^{t-1} e^{-v} dv$, obteniendo así

$$\mathrm{E}\left(X^2\right) = \frac{\sigma_y^2}{2} = \frac{1}{2}\mathrm{Var}\left(y\right). \tag{B.2}$$

Por último, la varianza de la salida de una activación ReLU, en función de la varianza de la entrada, se halla como

$$\begin{aligned} \mathrm{Var}\left(X\right) &= \mathrm{E}\left(X^2\right) - \mathrm{E}\left(X\right)^2 \\ \mathrm{Var}\left(X\right) &= \frac{\sigma_y^2}{2} - \frac{\sigma_y^2}{2\pi} \\ \mathrm{Var}\left(X\right) &= \left(1 - \frac{1}{\pi}\right) \frac{\sigma_y^2}{2} \end{aligned} \tag{B.3}$$

B.2. Capas PReLU

La función PReLU se diferencia de la ReLU tradicional por no anular los valores negativos de la entrada, sino que los escala con valor generalmente pequeño a, meintras que la parte positiva de la entrada se mantiene igual, tal y como ocurre con la función ReLU tradicional. La función PReLU puede definirse entonces como

$$f_{PReLU}(y) = \begin{cases} y & , \ y \geq 0 \\ ay & , \ y < 0 \end{cases},$$

donde se puede observar que la función ReLU es un caso particular de la PReLU con el parámetro $a = 0$. El cálculo de la media y varianza de la salida de una capa PReLU $X = f_{PReLU}(Y)$, dada una variable aleatoria Y con distribución normal de media 0 y varianza $\mathrm{Var}(Y) = \sigma_y^2$, pueden calcularse de forma análoga a la función ReLU, considerando que ahora el término de la integral entre $[-\infty, 0]$ no es nula, de manera que

$$\begin{aligned} \mathrm{E}(X) &= \int_{-\infty}^{0} x f_{dist}(x)\, dx + \int_{0}^{\infty} x f_{dist}(x)\, dx \\ &= \int_{-\infty}^{0} a y f_{dist}(y)\, dy + \int_{0}^{\infty} y f_{dist}(y)\, dy \\ &= -\int_{0}^{\infty} a y f_{dist}(y)\, dy + \int_{0}^{\infty} y f_{dist}(y)\, dy \\ &= (1-a)\int_{0}^{\infty} y f_{dist}(y)\, dy, \end{aligned}$$

que resulta ser el mismo cálculo que con la función ReLU, pero escalado en $1 - a$, por lo que la esperanza (media) producida por la función PReLU es

$$\mathrm{E}(X) \approx \frac{(1-a)}{\sqrt{2\pi}}\sigma_y. \tag{B.4}$$

De manera similar, para la esperanza del cuadrado $\mathrm{E}\left(X^2\right)$ se tiene

$$\begin{aligned}\mathrm{E}\left(X^2\right) &= \int_{-\infty}^{0} a^2 y^2 f_{dist}\left(y\right) dy + \int_{0}^{\infty} y^2 f_{dist}\left(y\right) dy \\ &= \int_{0}^{\infty} a^2 (-y)^2 f_{dist}\left(y\right) dy + \int_{0}^{\infty} y^2 f_{dist}\left(y\right) dy \\ &= (1+a^2) \int_{0}^{\infty} y f_{dist}\left(y\right) dy,\end{aligned}$$

que al repetir el procedimiento desarrollado para la función ReLU tradicional se obtiene

$$\mathrm{E}\left(X^2\right) = \frac{(1+a^2)}{2}\sigma_y^2.$$

La varianza producida por una capa PReLU, combinando Ecuaciones (B.4) y (B.2), resulta

$$\begin{aligned}\mathrm{Var}\left(X\right) &= \frac{(1+a^2)}{2}\sigma_y^2 - \left[\frac{(1-a)}{\sqrt{2\pi}}\sigma_y\right]^2 \\ &= \frac{\sigma_y^2}{2} + \frac{a^2\sigma_y^2}{2} - \frac{\sigma_y^2}{2\pi} + \frac{2a\sigma_y^2}{2\pi} - \frac{a^2\sigma_y^2}{2\pi} \\ &= \frac{1}{2}\left[1 + a^2 - \frac{(1-a)^2}{\pi}\right]\sigma_y^2 \\ &= \frac{1}{2}\left(1 + a^2 - \frac{1}{\pi} + \frac{2a}{\pi} - \frac{a^2}{\pi}\right)\sigma_y^2\end{aligned}$$

donde finalmente se tiene que

$$\mathrm{Var}\left(X\right) = \frac{1}{2}\left[\left(1 - \frac{1}{\pi}\right)a^2 + \frac{2}{\pi}a + \left(1 - \frac{1}{\pi}\right)\right]\sigma_y^2 \tag{B.5}$$

B.3. Capas ReLU con saturación

En el caso de una capa ReLU con saturación en los valores positivos, con un valor de saturación M, la expresión de la salida en función de una entrada Y puede expresarse como

$$f_{ReLUsat}\left(Y\right) = \min\left(\max\left(0, y\right), M\right) = \begin{cases} M & , \quad y > M \\ y & , \quad 0 \leq y \leq M \\ 0 & , \quad y < 0 \end{cases}.$$

Una vez más, se puede realizar un análisis de media (esperanza) y varianza de la salida de una capa ReLU con saturación, ante una determinada entrada Y con distribución normal de media cero y varianza σ_y^2, de forma análoga a los dos casos anteriores (ReLU estándar y PReLU). Comenzando por la media se tiene que

$$\begin{aligned}\mathrm{E}(X) &= \int_{-\infty}^{\infty} x f_{dist}(x)\, dx \\ &= \int_0^M \frac{y}{\sigma_y\sqrt{2\pi}} e^{-y^2/2\sigma_y^2} dy \\ &= \frac{\sigma_y}{\sqrt{2\pi}} \int_0^M e^{-u^2} du = -\frac{\sigma_y}{\sqrt{2\pi}} e^{-u}\Big|_0^M ,\end{aligned}$$

donde se realizó una vez más la sustitución $u = {}^{y^2}/_{2\sigma_y^2}$, obteniendo así

$$\mathrm{E}(X) = \frac{\sigma_y}{\sqrt{2\pi}} \left(1 - e^{-M}\right). \tag{B.6}$$

En cuanto a la esperanza del cuadrado $\mathrm{E}(X^2)$, esta puede calcularse como

$$\begin{aligned}\mathrm{E}\left(X^2\right) &= \int_{-\infty}^{\infty} x^2 f_{dist}(x)\, dx \\ &= \int_0^M \frac{y^2}{\sigma_y\sqrt{2\pi}} e^{-y^2/2\sigma_y^2} dy \\ &= \frac{2\sigma_y^2}{\sqrt{\pi}} \int_0^M u^2 e^{-u^2} du \\ &= \frac{2\sigma_y^2}{\sqrt{\pi}} \int_0^M u \left(u e^{-u^2}\right) du,\end{aligned}$$

en donde la sustitución realizada es $y^2 = 2\sigma_y^2 u^2$. Para poder resolver esta integral es necesario utilizar una integración por partes, considerando $g(u) = u$ y $h'(u) = ue^{-u^2}$, de modo que $h(u) = ({}^1/_2)e^{-u^2}$. Por medio de esta integración por partes

resulta entonces

$$\begin{aligned}
\mathrm{E}\left(X^2\right) &= \int_0^M g\left(u\right) h'\left(u\right) du \\
&= g\left(u\right) h\left(u\right) \Big|_0^M - \int_0^M g'\left(u\right) h\left(u\right) du \\
&= \frac{2\sigma_y^2}{\sqrt{\pi}} \left[\left(\frac{u}{2} e^{-u^2} \right) \Big|_0^M - \frac{1}{2} \int_0^M e^{-u^2} du \right] \\
&= \frac{2\sigma_y^2}{\sqrt{\pi}} \left[M e^{-M^2} - M - \frac{\sqrt{\pi}}{2} \mathrm{erf}\left(M\right) \right],
\end{aligned}$$

siendo $\mathrm{erf}\left(M\right) = {}^{2}/_{\sqrt{\pi}} \int_0^M e^{-u^2} du$ la Función Error, que no tiene una solución cerrada sino que se requiere de aproximaciones.

Apéndice C

Inicialización de capas simpliciales simétricas

En este apéndice se realiza un análisis estadístico para la función SymSim, tanto en modo *forward* como *backward*, con el fin de obtener distribuciones óptimas con las cuales iniciar sus parámetros. Este análisis se basa en lo propuesto por [62] y [5], donde la idea principal es mantener las varianzas de dichos parámetros constante a lo largo de la DNN, evitando así tener tanto *vanishing gradients* como *exploding gradients*. En el caso de [62] se busca optimizar dichas varianzas de pesos iniciales en capas lineales y considerando activaciones del tipo tangente o sigmoide, mientras que [5] lo hace para capas lineales con activaciones de tipo ReLU o su variante paramétrica PReLU o *leaky* ReLU.

C.1. Modo *forward*

En base a lo descrito en el Capítulo 3, se tiene que la función simplicial simétrica (con elemento estructurante) en modo *forward* está dada por:

$$
\begin{aligned}
\boldsymbol{y} &= \boldsymbol{c}^T \boldsymbol{\mu} + \boldsymbol{b} \\
\boldsymbol{\mu} &= \left[\; z_{s(1)} - m \quad , \cdots , \quad z_{s(i)} - z_{s(i-1)} \quad , \cdots , \quad M - z_{s(N)} \; \right] \\
\boldsymbol{z}_s &= \text{sort}(\boldsymbol{z}) \\
\boldsymbol{z} &= \boldsymbol{x} \cdot \boldsymbol{SE} \; ,
\end{aligned}
$$

donde $\boldsymbol{y}_l$ son las salidas de la capa SymSim, $\boldsymbol{c}_l$ los coeficientes/pesos, $\boldsymbol{b}_l$ los bias, m y M los valores mínimos y máximos donde la función SymSim está definida (con $m < M$), $\boldsymbol{SE}$ los elementos estructurantes y $\boldsymbol{x}$ las entradas. Para este análisis se asume que todas las entradas/parámetros en una capa de la red tienen la misma distribución y son independientes entre sí, de modo que $\prod \mathrm{Var}\left(x_i\right) = n \times \mathrm{Var}\left(x\right)$. También se asume que los bias $\boldsymbol{b}_l$ en cada capa se inicializan todos en cero, siendo que este parámetro en particular no influye en los gradientes y por tanto no presenta ninguno de los problemas conocidos por inicialización "inadecuada" anteriormente mencionados.

Comenzando por las ecuaciones de la función simplicial simétrica en modo *forward*, se tiene que para una salida cualquiera de una capa SymSim puede expresarse como:

$$
\begin{aligned}
y_l &= \sum_{i=1}^{N_l+1} \mu_{l(i)} c_{l(i)} + b_l \\
\mathrm{Var}\left(y_l\right) &= \mathrm{Var}\left(\sum_{i=1}^{N_l+1} \mu_{l(i)} c_{l(i)} + b_l\right) \\
&= \mathrm{Var}\left(\sum_{i=1}^{N_l+1} \mu_{l(i)} c_{l(i)}\right) + \mathrm{Var}\left(b_l\right) \\
&= \mathrm{Var}\left(\sum_{i=1}^{N_l+1} \mu_{l(i)} c_{l(i)}\right) \\
\mathrm{Var}\left(y_l\right) &= \sum_{i=1}^{N_l+1} \mathrm{Var}\left(\mu_{l(i)} c_{l(i)}\right) \ , \qquad \text{(C.1)}
\end{aligned}
$$

donde la varianza del bias se anula al ser inicializado en una constante ($\mathrm{Var}\left(b_l\right) = 0$). A su vez, la varianza de la sumatoria es la sumatoria de las varianzas al tener los pesos $c_{l(i)}$ y valores $\mu_{l(i)}$ distribuciones independientes entre sí. En este caso, el producto vectorial $\boldsymbol{\mu c}$ fue re-escrito como una suma de N_l términos, con N_l como la cantidad de entradas necesarias para computar una sola salida, o en otras palabras, definido como $(K_H \times K_W + 1) \times CH_{in}$, siendo $K_H \times K_W$ el tamaño del kernel en 2D y CH_{in} la cantidad de canales de la entrada. Al no poderse asumir que los valores de $\mu_{l(i)}$ pertenecen a una distribución con media cero, la varianza del producto en

la Ecuación (C.1) e vuelve

$$\begin{aligned}\mathrm{Var}\left(y_l\right) &= \sum_{i=1}^{N_l+1} \mathrm{E}\left(\mu_{l(i)}^2\right) \mathrm{E}\left(c_{l(i)}^2\right) - \left[\mathrm{E}\left(\mu_{l(i)}\right) \mathrm{E}\left(c_{l(i)}\right)\right]^2 \\ &= \sum_{i=1}^{N_l+1} \mathrm{E}\left(\mu_{l(i)}^2\right) \mathrm{Var}\left(c_{l(i)}\right) \\ \mathrm{Var}\left(y_l\right) &= \mathrm{Var}\left(c_l\right) \sum_{i=1}^{N_l+1} \mathrm{E}\left(\mu_{l(i)}^2\right) \ , \end{aligned} \tag{C.2}$$

donde al tener pesos $c_{l(i)}$ con la misma distribuciones, independientes entre sí y media cero, el producto de esperanzas (medias) al cuadrado se anula, se obtiene la varianza de los pesos como $\mathrm{E}\left(c_{l(i)}^2\right) = \mathrm{Var}\left(c_{l(i)}\right)$ y se puede sacar $\mathrm{Var}\left(c_{l(i)}\right) = \mathrm{Var}\left(c_l\right)$ como factor común. Para continuar el análisis de la varianza de la capa SymSim, resulta conveniente separar la sumatoria de la Ecuación (C.2) a partir de los valores de $\mu_{l(i)}$, resultando en tres tipos de términos diferentes:

$$\begin{aligned}\mathrm{E}\left(\mu_{l(1)}^2\right) &= \mathrm{E}\left[\left(z_{s_{l(1)}} - m_l\right)^2\right] = \mathrm{E}\left(z_{s_{l(1)}}^2\right) - 2m_l\mathrm{E}\left(z_{s_{l(1)}}\right) + m_l^2 \\ \mathrm{E}\left(\mu_{l(j)}^2\right) &= \mathrm{E}\left[\left(z_{s_{l(j)}} - z_{s_{l(j-1)}}\right)^2\right] = \mathrm{E}\left(z_{s_{l(j)}}^2\right) - 2\mathrm{E}\left(z_{s_{l(j)}}\right) \mathrm{E}\left(z_{s_{l(j-1)}}\right) + \mathrm{E}\left(z_{s_{l(j-1)}}^2\right) \\ \mathrm{E}\left(\mu_{l(N+1)}^2\right) &= \mathrm{E}\left[\left(M_l - z_{s(N)}\right)^2\right] = \mathrm{E}\left(z_{s_{l(N)}}^2\right) - 2M_l\mathrm{E}\left(z_{s_{l(N)}}\right) + M_l^2 \ , \end{aligned}$$

con m_l y M_l los rangos mínimo y máximo (respectivamente) de la función SymSim para la capa l. Para obtener dichos términos se tuvo en cuenta varias propiedades de la esperanza, mencionadas en el Apéndice A, tales como esperanza de la suma, esperanza de constantes, etc. Considerando además que todas las variables $z_{s_{l(i)}}$ tienen la misma distribución, se obtiene

$$\begin{aligned}\mathrm{E}\left(\mu_{l(1)}^2\right) &= \mathrm{E}\left(z_{s_l}^2\right) - 2m_l\mathrm{E}\left(z_{s_l}\right) + m_l^2 \\ \mathrm{E}\left(\mu_{l(j)}^2\right) &= 2\mathrm{E}\left(z_{s_l}^2\right) - 2\mathrm{E}\left(z_{s_l}\right)^2 = 2\mathrm{Var}\left(z_{s_l}\right) \\ \mathrm{E}\left(\mu_{l(N+1)}^2\right) &= \mathrm{E}\left(z_{s_l}^2\right) - 2M_l\mathrm{E}\left(z_{s_l}\right) + M_l^2 \ . \end{aligned}$$

Reescribiendo la expresión para la varianza de la salida de la capa SymSim y_l de la Ecuación (C.2), expandiendo los términos de las medias $\mathrm{E}\left(\mu_l^2\right)$ anteriormente

calculados, resulta

$$\begin{aligned}\mathrm{Var}\left(y_l\right) &= \mathrm{Var}\left(c_l\right)\left\{\mathrm{E}\left(\mu_{l(1)}^2\right) + \left[\sum_{i=2}^{N_l}\mathrm{E}\left(\mu_{l(j)}^2\right)\right] + \mathrm{E}\left(\mu_{l(N+1)}^2\right)\right\} \\ &= \mathrm{Var}\left(c_l\right)\left\{2\mathrm{E}\left(z_{s_l}^2\right) + 2\left(N_l-1\right)\mathrm{Var}\left(z_{s_l}\right) - 2(m_l+M_l)\mathrm{E}\left(z_{s_l}\right) + m^2 + M^2\right\} \\ \mathrm{Var}\left(y_l\right) &= \mathrm{Var}\left(c_l\right)\left\{2\mathrm{E}\left(z_l^2\right) + 2\left(N_l-1\right)\mathrm{Var}\left(z_l\right) - 2(m_l+M_l)\mathrm{E}\left(z_l\right) + m^2 + M^2\right\} \ ,\end{aligned} \tag{C.3}$$

donde al ser $\boldsymbol{z}_{s_l}$ un re-ordenamiento de $\boldsymbol{z}_l$ y por lo tanto tener sus elementos la misma distribución, tanto la media (incluyendo la esperanza del cuadrado) como la varianza son iguales, es decir $\mathrm{E}\left(z_{s_l}\right) = \mathrm{E}\left(z_l\right)$, $\mathrm{E}\left(z_{s_l}^2\right) = \mathrm{E}\left(z_l^2\right)$ y $\mathrm{Var}\left(z_{s_l}\right) = \mathrm{Var}\left(z_l\right)$.

Para finalmente expresar la varianza de la salida de una capa SymSim respecto de la entrada, hace falta despejar los términos con z_l sabiendo que estos se computan mediante el producto de la entrada y el elemento estructurante, es decir $z_l = x_l SE_l$, donde los valores del SE_l son binarios y siguen la siguiente distribución:

$$SE_l = \begin{cases} 1 & , \quad P\left(SE_l = 1\right) = 1/2 \\ 0 & , \quad P\left(SE_l = 0\right) = 1/2 \end{cases} \ ,$$

de manera que la distribución de z_l resulta

$$z_l = \begin{cases} x_l & , \quad SE_l = 1 \\ 0 & , \quad SE_l = 0 \end{cases} \ .$$

La varianza $\mathrm{Var}\left(z_l\right)$ puede obtenerse usando la ley de varianza total [69] (en inglés *law of total variance*) que para este caso establece:

$$\mathrm{Var}\left(z_l\right) = \mathrm{E}\left[\mathrm{Var}\left(z_l|SE_l\right)\right] + \mathrm{Var}\left[\mathrm{E}\left(z_l|SE_l\right)\right] \ .$$

El primer término puede hallarse utilizando la definición de esperanza en variables

discretas (Ec. A.1) como

$$\begin{aligned}
\mathrm{E}\left[\mathrm{Var}\left(z_l|SE_l\right)\right] &= \mathrm{Var}\left(z_l=0|SE_l\right)\mathrm{P}\left(z_l=0|SE_l\right)+\mathrm{Var}\left(z_l=x_l|SE_l\right)\mathrm{P}\left(z_l=x_l|SE_l\right)\\
&= \mathrm{Var}\left(0\right)\mathrm{P}\left(SE_l=0\right)+\mathrm{Var}\left(x_l\right)\mathrm{P}\left(SE_l=1\right)\\
&= \mathrm{Var}\left(x_l\right)\mathrm{P}\left(SE_l=1\right)\\
\mathrm{E}\left[\mathrm{Var}\left(z_l|SE_l\right)\right] &= \frac{1}{2}\mathrm{Var}\left(x_l\right)\ ,
\end{aligned}$$

donde al ser z_l directamente dependiente de SE_l, las probabilidades condicionales son directamente la probabilidad del elemento estructurante $\mathrm{P}\left(z_l=x_l|SE_l\right)=\mathrm{P}\left(SE_l=1\right)$ y $\mathrm{P}\left(z_l=0|SE_l\right)=\mathrm{P}\left(SE_l=0\right)$. De manera similar, el segundo término de la ley de varianza total puede obtenerse como

$$\begin{aligned}
\mathrm{E}\left(z_l|SE_l\right) &= x_l\mathrm{P}\left(z_l=x_l|SE_l\right)+0\mathrm{P}\left(z_l=0|SE_l\right)\\
&= x_l\mathrm{P}\left(z_l=x_l|SE_l\right)\\
\mathrm{E}\left(z_l|SE_l\right) &= \left(1/2\right)x_l
\end{aligned}$$

$$\begin{aligned}
\mathrm{Var}\left[\mathrm{E}\left(z_l|SE_l\right)\right] &= \mathrm{Var}\left[\left(1/2\right)x_l\right]\\
\mathrm{Var}\left[\mathrm{E}\left(z_l|SE_l\right)\right] &= \frac{1}{4}\mathrm{Var}\left(x_l\right)\ .
\end{aligned}$$

La varianza de esta variable intermedia z_l, expresada en términos de la entrada x_l resulta entonces

$$\begin{aligned}
\mathrm{Var}\left(z_l\right) &= \mathrm{E}\left[\mathrm{Var}\left(z_l|SE_l\right)\right]+\mathrm{Var}\left[\mathrm{E}\left(z_l|SE_l\right)\right]\\
&= \left(1/2\right)\mathrm{Var}\left(x_l\right)+\left(1/4\right)\mathrm{Var}\left(x_l\right)\\
\mathrm{Var}\left(z_l\right) &= \frac{3}{4}\mathrm{Var}\left(x_l\right)\ . \qquad \text{(C.4)}
\end{aligned}$$

La esperanza o media de z_l puede obtenerse en cambio por medio de la ley de

esperanza total [69] (del inglés *law of total expectation*) que en este caso implica

$$\begin{aligned} \mathrm{E}\left(z_l\right) &= \mathrm{E}\left[\mathrm{E}\left(z_l|SE_l\right)\right] \\ &= \mathrm{E}\left[\left(1/2\right)x_l\right] \\ \mathrm{E}\left(z_l\right) &= \frac{1}{2}\mathrm{E}\left(x_l\right) \ , \end{aligned} \tag{C.5}$$

con $\mathrm{E}\left(z_l|SE_l\right) = \left(1/2\right)x_l$ calculado anteriormente. Por otra parte, la esperanza de z_l^2 puede calcularse de la definición de varianza a partir de

$$\begin{aligned} \mathrm{Var}\left(z_l\right) &= \mathrm{E}\left(z_l^2\right) - \mathrm{E}\left(z_l\right)^2 \\ \mathrm{E}\left(z_l^2\right) &= \mathrm{Var}\left(z_l\right) + \mathrm{E}\left(z_l\right)^2 \\ \mathrm{E}\left(z_l^2\right) &= \left(3/4\right)\mathrm{Var}\left(x_l\right) + \left(1/4\right)\mathrm{E}\left(x_l\right)^2 \ . \end{aligned} \tag{C.6}$$

Finalmente, reemplazando Ecuaciones (C.4) - (C.6) en la Ecuación (C.3) se obtiene

$$\begin{aligned} \mathrm{Var}\left(y_l\right) &= \mathrm{Var}\left(c_l\right)\left[2\mathrm{E}\left(z_l^2\right) + 2\left(N_l - 1\right)\mathrm{Var}\left(z_l\right) - 2(m_l + M_l)\mathrm{E}\left(z_l\right) + m_l^2 + M_l^2\right] \\ \mathrm{Var}\left(y_l\right) &= \mathrm{Var}\left(c_l\right)\left[\left(3/2\right)\mathrm{Var}\left(x_l\right) + \left(1/2\right)\mathrm{E}\left(x_l\right)^2 + \left(3/2\right)\left(N_l - 1\right)\mathrm{Var}\left(x_l\right)\right. \\ &\quad \left. + \left(m_l + M_l\right)\mathrm{E}\left(x_l\right) + m_l^2 + M_l^2\right] \\ \mathrm{Var}\left(y_l\right) &= \mathrm{Var}\left(c_l\right)\left[\frac{3}{2}N_l\mathrm{Var}\left(x_l\right) + \frac{1}{2}\mathrm{E}\left(x_l\right)^2 + \left(m_l + M_l\right)\mathrm{E}\left(x_l\right) + m_l^2 + M_l^2\right] \ . \end{aligned} \tag{C.7}$$

Una vez obtenida la varianza de la salida y_l de una capa SymSim, con l como el número de capa, es necesario expresarla en función de la salida de la capa anterior y_{l-1}, con el fin de hallar la varianza con la cual definir la distribución que inicialice los pesos c_l de manera que se mantenga la varianza entrada-salida constante a lo largo de toda la red. La forma en la que se desarrolle la Ecuación (C.7) para hallar esta relación de varianzas de la salida de una capa con la salida de la capa anterior depende de la función de activación.

Una de las funciones de activación más utilizadas en DNN es la ReLU, que anula todos los valores negativos, o dicho de otra manera $f_{ReLU}\left(y\right) = \max\left(0, y\right)$. De esta forma, se fijan los rangos mínimos de entrada de todas las capas como

$m_l = 0$, mientras que el rango máximo depende de las salidas de las capas anteriores (previo a computarse la ReLU). Si se asume que dichas salidas de la capa anterior y_{l-1} pertenecen a una distribución normal de media cero y varianza $\mathrm{Var}\left(y_{l-1}\right)$, se puede aproximar dicho valor del rango máximo como $M_l \approx 3\sigma_{y_{l-1}}$ (conteniendo alrededor del 99,73 % de la distribución), donde $\sigma_{y_{l-1}}$ es el desviación estándar de la distribución de y_{l-1} y $\sigma^2_{y_{l-1}} = \mathrm{Var}\left(y_{l-1}\right)$. Considerando además los resultados mostrados en el Apéndice B, específicamente Ecuaciones (B.1)-(B.3), la Ecuación (C.7) puede expresarse como

$$\begin{aligned}
\mathrm{Var}\left(y_l\right) &\approx \mathrm{Var}\left(c_l\right)\left[\frac{3}{4}N_l\left(1-\frac{1}{\pi}\right)\sigma^2_{y_{l-1}} + \frac{1}{4\pi}\sigma^2_{y_{l-1}} + \left(3\sigma_{y_{l-1}}\right)\frac{\sigma_{y_{l-1}}}{\sqrt{2\pi}} + 9\sigma^2_{y_{l-1}}\right] \\
\mathrm{Var}\left(y_l\right) &\approx \mathrm{Var}\left(c_l\right)\left[\frac{3}{4}\left(1-\frac{1}{\pi}\right)N_l + \frac{1}{4\pi} + \frac{3}{\sqrt{2\pi}} + 9\right]\sigma^2_{y_{l-1}} \\
\mathrm{Var}\left(y_l\right) &\approx \mathrm{Var}\left(c_l\right)\gamma_{ReLU}\left(N_l\right)\mathrm{Var}\left(y_{l-1}\right) \ , \qquad \text{(C.8)}
\end{aligned}$$

donde la constante

$$\gamma_{ReLU}\left(N_l\right) = \frac{3}{4}\left(1-\frac{1}{\pi}\right)N_l + \left(\frac{1}{4\pi} + \frac{3}{\sqrt{2\pi}} + 9\right) \approx \frac{N_l}{2} + 11 \qquad \text{(C.9)}$$

escala de forma lineal con N_l.

Se puede observar que para que la varianza se mantenga constante a lo largo de la red, el término $\mathrm{Var}\left(c_l\right)\gamma\left(N_l\right)$ debe ser igual a 1, por lo que la varianza con la cual inicializar los pesos c_l de la capa SymSim resulta

$$\mathrm{Var}\left(c_l\right) = \frac{1}{\gamma\left(N_l\right)} \ , \qquad \text{(C.10)}$$

donde se recuerda que $N_l = \left(K_H \times K_W + 1\right) \times CH_{in}$, con $K_H \times K_W$ como el tamaño del kernel en 2D y CH_{in} la cantidad de canales de entrada, ambas variables correspondientes a la capa l.

Este mismo análisis se puede realizar para otro tipo de funciones de activación, obteniendo resultados similares a las Ecuaciones (C.8) y (C.10), pero con diferentes constantes $\gamma\left(N_l\right)$. Un ejemplo de esto sería la función PReLU con parámetro a_l, donde para resolver la Ecuación (C.7) se consideran los rangos $m_l \approx -3\sigma_{y_{l-1}}$ y $M_l \approx 3\sigma_{y_{l-1}}$, debido a que ahora sí hay valores negativos presentes. Si bien los valores

negativos van a ser escalados por un número pequeño y por lo tanto $-3\sigma_{y_{l-1}}$ puede resultar mucho menor al valor mínimo real, para este análisis resulta conveniente mantener el rango de entrada simétrico. De esta forma se puede reescribir la Ecuación (C.7), utilizando los resultados del Apéndice B, como

$$\begin{aligned}
\mathrm{Var}\left(y_l\right) &\approx \mathrm{Var}\left(c_l\right)\left\{\frac{3}{4}N_l\left[1+a_l^2-\frac{\left(1-a_l\right)^2}{\pi}\right]\sigma_{y_{l-1}}^2+\frac{\left(1-a_l\right)^2}{4\pi}\sigma_{y_{l-1}}^2+9\sigma_{y_{l-1}}^2+9\sigma_{y_{l-1}}^2\right\} \\
\mathrm{Var}\left(y_l\right) &\approx \mathrm{Var}\left(c_l\right)\left\{\frac{3}{4}\left[1+a_l^2-\frac{\left(1-a_l\right)^2}{\pi}\right]N_l+\frac{\left(1-a_l\right)^2}{4\pi}+18\right\}\sigma_{y_{l-1}}^2 \\
\mathrm{Var}\left(y_l\right) &\approx \mathrm{Var}\left(c_l\right)\gamma_{PReLU}\left(N_l\right)\mathrm{Var}\left(y_{l-1}\right) \ ,
\end{aligned}$$

de donde se obtiene que para llegar a la misma conclusión que la Ecuación (C.10), la constante $\gamma_{PReLU}\left(N_l\right)$ debe calcularse como

$$\gamma_{PReLU}\left(N_l\right)=\frac{3}{4}\left[1+a_l^2-\frac{\left(1-a_l\right)^2}{\pi}\right]N_l+\left[\frac{\left(1-a_l\right)^2}{4\pi}+18\right] \ . \tag{C.11}$$

A partir de la Ecuación (C.11) y utilizando $a_l=0$, se puede hallar una expresión para la constante $\gamma\left(N_l\right)$ en el caso de una activación del tipo ReLU tradicional, pero teniendo en cuenta un rango de entrada simétrico para la función SymSim, es decir $m_l\approx-3\sigma_{y_{l-1}}$ y $M_l\approx 3\sigma_{y_{l-1}}$. La constante $\gamma\left(N_l\right)$ para esta función de activación y rango resulta

$$\gamma_{ReLU}\left(N_l\right)=\frac{3}{4}\left(1-\frac{1}{\pi}\right)N_l+\left(\frac{1}{4\pi}+18\right)\approx\frac{N_l}{2}+18 \tag{C.12}$$

que es muy similar a la Ecuación (C.9), dado que también crece de forma lineal con N_l (misma pendiente) y varía solamente en el término de la ordenada al origen. De la función PReLU no solamente se puede obtener la función ReLU tradicional, sino que también se puede eliminar la función de activación al utilizar $a_l=1$. Debido a que la función SymSim es de por sí no lineal, y por lo tanto puede prescindir de una función de activación, resulta interesante mencionar cuál sería la inicialización óptima de los pesos para este caso. La constante $\gamma_{NoAC}\left(N_l\right)$ puede hallarse entonces a partir de la Ecuación (C.11) reemplazando $a_l=1$, debido a que la elección de m_l

y M_l es la misma, teniendo así

$$\gamma_{NoAC}\left(N_l\right) = \frac{3}{2}N_l + 18\ . \tag{C.13}$$

En el caso de una ReLU con saturación en los valores positivos, como por ejemplo el valor máximo M del rango de entrada de la función SymSim, se tiene que debido a los resultados de la esperanza $\mathrm{E}\left(X\right)$ y esperanza del cuadrado $\mathrm{E}\left(X_l^2\right)$ (para la obtención de la varianza) halladas en el Apéndice B, al despejar la Ecuación (C.7) se tendrían términos del tipo $e^{-M_l^2}$ y $\mathrm{erf}\left(M_l\right)$. Teniendo que cuenta que M_l^2 puede aproximarse proporcionalmente a $\sigma_{y_{l-1}}^2 = \mathrm{Var}\left(Y_{l-1}\right)$, el despeje de la Ecuación (C.7) escala drásticamente en complejidad comparado con los ejemplos anteriores de función de activación, por lo que no se incluirá a la función de activación ReLU saturada en este análisis. Si se desea utilizar este tipo de activación, como por ejemplo la función ReLU6 empleada en redes tipo MobileNet [54], se pueden usar los resultados con los casos de ReLU no saturadas para una inicialización que, aunque no completamente óptima, al menos considere las propiedades de las funciones SymSim.

C.2. Modo *backward*

Para el caso del modo backward, se puede realizar un análisis similar al anterior para varianzas de la función SymSim, calculando la distribución de los pesos/coeficientes que mantenga una varianza de gradiente entrada/salida constante durante la ejecución del algoritmo de *backpropagation*. En este caso, el único gradiente relevante es el de una entrada ($^{\partial L}/_{\partial x}$), que se calcula como

$$\begin{aligned}
\frac{\partial L}{\partial x} &= \sum_{i=1}^{\hat{N}} \frac{\partial L}{\partial z_i} SE_i \\
\frac{\partial L}{\partial \boldsymbol{z}} &= \mathrm{arrange}\left(\frac{\partial L}{\partial \boldsymbol{z}_s}, \boldsymbol{Ind}_{sb}\right) \\
\frac{\partial L}{\partial \boldsymbol{z}_s} &= \left[\frac{\partial L}{\partial \mu_1} - \frac{\partial L}{\partial \mu_2}, \cdots, \frac{\partial L}{\partial \mu_N} - \frac{\partial L}{\partial \mu_{N+1}}\right] \\
\frac{\partial L}{\partial \boldsymbol{\mu}} &= \boldsymbol{c}\frac{\partial L}{\partial \boldsymbol{y}}\ ,
\end{aligned}$$

donde $\hat{N}$ es la cantidad de salidas donde aparece dicha entrada, coincidiendo con $K_H \times K_W \times CH_{out}$ para la mayoría de las entradas, SE es el Elemento estructurante binario, $\boldsymbol{c}$ los pesos de la función SymSim y la operación "arrange $(\partial L/\partial \boldsymbol{z}_s, \boldsymbol{Ind}_{sb})$" reorganiza los elementos de $\partial L/\partial \boldsymbol{z}_s$ según los índices $\boldsymbol{Ind}_{sb}$ (para revertir el ordenamiento del modo *forward*).

Calculando entonces la varianza del gradiente respecto a una entrada de un capa l, se tiene que

$$\begin{aligned}\mathrm{Var}\left(\frac{\partial L}{\partial x_l}\right) &= \mathrm{Var}\left(\sum_{i=1}^{\hat{N}_l} \frac{\partial L}{\partial z_{l(i)}} SE_{l(i)}\right) \\ &= \sum_{i=1}^{\hat{N}_l} \mathrm{Var}\left(\frac{\partial L}{\partial z_{l(i)}} SE_{l(i)}\right) \\ &= \hat{N}_l \mathrm{Var}\left(\frac{\partial L}{\partial z_l} SE_l\right) \ ,\end{aligned}$$

con la varianza de la suma como la suma de las varianzas por ser variables independientes entre sí y poseer la misma distribución. Repitiendo el uso de la Ley de varianza total [69], recordando que al ser SE_l una variable aleatoria binaria, y que por lo tanto el producto por el elemento estructurante SE tiene la siguiente probabilidad condicional

$$\frac{\partial L}{\partial z SE} = \begin{cases} \partial L/\partial z & , \quad SE = 1 \\ 0 & , \quad SE = 0 \end{cases} \ ,$$

se puede continuar despejando la varianza del gradiente de la entrada como

$$\begin{aligned}
\mathrm{Var}\left(\frac{\partial L}{\partial x_l}\right) &= \hat{N}_l \frac{3}{4} \mathrm{Var}\left(\frac{\partial L}{\partial z_l}\right) \\
&= \frac{3}{4} \hat{N}_l \mathrm{Var}\left(\frac{\partial L}{\partial z_{s_l}}\right) \\
&= \frac{3}{4} \hat{N}_l \mathrm{Var}\left(\frac{\partial L}{\partial \mu_{l_j}} - \frac{\partial L}{\partial \mu_{l_{j+1}}}\right) \\
&= \frac{3}{4} \hat{N}_l \left[\mathrm{Var}\left(\frac{\partial L}{\partial \mu_l}\right) + \mathrm{Var}\left(-\frac{\partial L}{\partial \mu_l}\right)\right] \\
&= \frac{3}{4} \hat{N}_l \left[2\mathrm{Var}\left(\frac{\partial L}{\partial \mu_l}\right)\right] \\
&= \frac{3}{2} \hat{N}_l \mathrm{Var}\left(\frac{\partial L}{\partial \mu_l}\right) \ ,
\end{aligned}$$

donde se recuerda que la varianza de elementos re-ordenados es igual a la varianza de los elementos originales, es decir $\mathrm{Var}\left({}^{\partial L}/_{\partial z_l}\right) = \mathrm{Var}\left({}^{\partial L}/_{\partial z_{s_l}}\right)$, y que por tener los valores de ${}^{\partial L}/_{\partial \mu_l}$ la misma distribución pero siendo independientes entre sí, la varianza de la suma es la suma de sus varianzas. Eligiendo a los pesos de la función SymSim con distribución normal de media cero, la varianza del gradiente de la entrada ${}^{\partial L}/_{\partial x_l}$ resulta

$$\begin{aligned}
\mathrm{Var}\left(\frac{\partial L}{\partial x_l}\right) &= \frac{3}{2} \hat{N}_l \mathrm{Var}\left(c_l \frac{\partial L}{\partial y_l}\right) \\
&= \frac{3}{2} \hat{N}_l \left\{ \mathrm{E}\left(c_l^2\right) \mathrm{E}\left[\left(\frac{\partial L}{\partial y_l}\right)^2\right] - \left[\mathrm{E}\left(c_l\right) \mathrm{E}\left(\frac{\partial L}{\partial y_l}\right)\right]^2 \right\} \\
&= \frac{3}{2} \hat{N}_l \mathrm{Var}\left(c_l\right) \mathrm{E}\left[\left(\frac{\partial L}{\partial y_l}\right)^2\right] \ ,
\end{aligned}$$

en donde al reemplazar la esperanza del cuadrado por lo obtenido en el Apéndice B para la activación PReLU con parámetro a_l, finalmente resulta

$$\begin{aligned}
\mathrm{Var}\left(\frac{\partial L}{\partial x_l}\right) &= \frac{3}{2} \hat{N}_l \mathrm{Var}\left(c_l\right) \frac{\left(1 + a_l^2\right)}{2} \mathrm{Var}\left(\frac{\partial L}{\partial x_{l+1}}\right) \\
&= \mathrm{Var}\left(c_l\right) \frac{3\left(1 + a_l^2\right)}{4} \hat{N}_l \mathrm{Var}\left(\frac{\partial L}{\partial x_{l+1}}\right) \\
&= \mathrm{Var}\left(c_l\right) \hat{\gamma}\left(\hat{N}_l\right) \mathrm{Var}\left(\frac{\partial L}{\partial x_{l+1}}\right) \ .
\end{aligned}$$

Para que la varianza del gradiente de la entrada de una capa SymSim se mantenga constante resulta conveniente que, de forma similar al modo *forward*, los pesos c_l tengan una varianza que compense los efectos de la constante $\hat{\gamma}\left(\hat{N}_l\right)$, teniendo entonces

$$\mathrm{Var}\left(c_l\right) = \frac{1}{\hat{\gamma}\left(\hat{N}_l\right)} \ , \tag{C.14}$$

con la constante $\hat{\gamma}\left(\hat{N}_l\right)$ se calcula como

$$\hat{\gamma}\left(\hat{N}_l\right) = \frac{3\left(1+a_l^2\right)}{4}\hat{N}_l \ . \tag{C.15}$$

Cuando en la función SymSim, se comparten los mismo elementos estructurantes para todos los canales de salida, lo que beneficia la implementación en Hardware, algunas de las derivadas parciales para la obtención del gradiente de la entrada cambian, teniendo entonces

$$\begin{aligned}
\frac{\partial L}{\partial x} &= \sum_{i=1}^{K_H \times K_W} \frac{\partial L}{\partial z_i} SE_i \\
\frac{\partial L}{\partial \boldsymbol{z}} &= \mathrm{arrange}\left(\frac{\partial L}{\partial \boldsymbol{z}_s}, Ind\right) \\
\frac{\partial L}{\partial \boldsymbol{z}_s} &= \left[\frac{\partial L}{\partial \mu_1} - \frac{\partial L}{\partial \mu_2}, \cdots, \frac{\partial L}{\partial \mu_N} - \frac{\partial L}{\partial \mu_{N+1}}\right] \\
\frac{\partial L}{\partial \boldsymbol{\mu}} &= \sum_{j=1}^{CH_{out}} \boldsymbol{c}_j \frac{\partial L}{\partial \boldsymbol{y}_j} \ .
\end{aligned}$$

A pesar de este cambio en las derivadas parciales, siguiendo el mismo análisis anterior, manteniendo las consideraciones de distribuciones independientes entre las variables, se puede llegar al mismo resultado que las Ecuaciones (C.14) y (C.15).

Apéndice D

Arquitecturas ResNet

Este apéndice mustra los bloques convencionales que componen las arquitecturas de Resdes Neuronales denominadas ResNet#, donde el caracter "#" es reemplazado por la cantidad de capas convolucionales y capas FC, publicadas originalmente en [18]. Dicho tipo de arquitectura se basa en el uso de "bloques residuales" que permiten entrenar Redes Neuronales muy profundas mediante el uso de conexiones residuales (suma de la entrada al bloque con el resultado de su procesamiento) que facilitan el flujo de gradientes a través de la red, mitigando así el problema de la degradación de la precisión y *vanishing gradient* en redes profundas.

Uno de los modelos más utilizados de ResNet, por su reducido número de parámetros (el más pequeño de los propuestos por [18]) y buen desempeño, es el denominado ResNet18, cuya estructura se muestra en la Tabla D.1, teniendo un cuarto de los *features* por capa que la red ResNet18 original, tal y como se propuso en [8]. Esta se compone de una primera capa con un filtro de tamaño 7×7 y *stride* 2, seguido de un MaxPool que no sólo realiza un *down-sample* (por poseer *stride* 2) sino que al tener una ventana de 3×3 se podría decir que también realiza un filtrado morfológico (dilatación). A partir de la segunda capa, esta arquitectura presenta sucesivos bloques (denominados *Basic Block*) con secuencias de filtros 3×3 y su característica *skip-connection*[1] con una activación ReLU al final, los cuales son mostrados en la Figura D.1. Ambos tipos de "bloques básicos" en ResNet18 poseen dos filtros en los que el primero puede tener *stride* 1 (Fig. D.1b) o 2 (Fig. D.1a) para *down-sampling*.

[1] Conexión residual (entrada original sumada a entrada post-procesada).

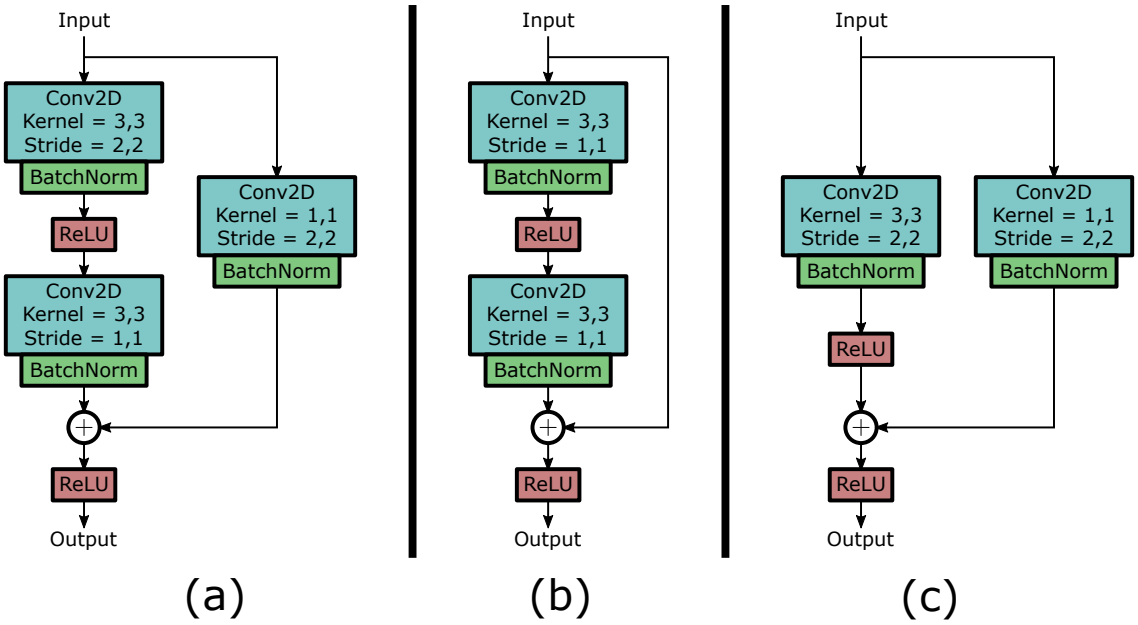


Figura D.1: Bloques básicos (Basic Blocks) de las arquitecturas ResNet: a) Bloque básico con *down-sample*; b) Bloque Básico; c) Bloque Básico reducido y con *down-sample*.

En caso de realizar *down-sampling*, se emplea un filtro del tipo *point-wise* (1×1) y sin activación para también efectuar *down-sampling* en la entrada original y que los tamaños de las entradas a la *skip-connection* coincidan. Todas las capas de filtrado (incluyendo MaxPool) tienen el *padding* necesario para no modificar el tamaño (alto y ancho) de la salida, por lo que la reducción en este aspecto se produce solamente por el *stride*. Finalmente, los *features* extraídos la sucesión de capas de filtrado son reducidos a un vector (un sólo elemento por canal) mediante un AvgPool (adaptable al tamaño de su entrada para comprimirla en un arreglo $1 \times 1 \times$ *features*) y procesados por una capa FC con tantas neuronas de salida como clases en el *dataset* que se desee clasificar.

Aún siendo ResNet18 la de menor cantidad de capas/parámetros de la familia ResNet propuesta originalmente en [18], esta sigue siendo relativamente grande (alrededor de 700 KB) para aplicaciones en sistemas con recursos limitados, tales como sistemas embebidos, *edge processing*, IoT, etc. Por esta razón, en [8] se presentaron variantes que no solo reducen la cantidad de *features* de salida de las capas, sino que además eliminan algunos de los *Basic Block* para reducir tanto la memoria requerida como el número de operaciones. De este "recorte" de capas surgen tres variantes con el nombre de ResNet04, ResNet06 y ResNet08, cuyas estructuras se

Layer	***Type***	***Kernel***	***Stride***	***Features***
Layer 1	Filter + BatchNorm (+ ReLU)	7×7	2	16
	MaxPool	3×3	2	-
Layer 2_1	Basic Block	3×3	1	16
		3×3	1	16
Layer 2_2	Basic Block	3×3	1	16
		3×3	1	16
Layer 3_1	Basic Block Downsample	3×3	2	32
		3×3	1	32
		1×1	2	32
Layer 3_2	Basic Block	3×3	1	32
		3×3	1	32
Layer 4_1	Basic Block Downsample	3×3	2	64
		3×3	1	64
		1×1	2	64
Layer 4_2	Basic Block	3×3	1	64
		3×3	1	64
Layer 5_1	Basic Block Downsample	3×3	2	128
		3×3	1	128
		1×1	2	128
Layer 5_2	Basic Block	3×3	1	128
		3×3	1	128
Layer 6	Adaptive AvgPool	7×7	-	-
	Fully Connected	128	-	# Classes

Tabla D.1: Arquitectura de ResNet18.

muestran en las Tablas D.2, D.3 y D.5, respectivamente. ResNet04 en introduce un nuevo tipo de "bloque básico" mostrado en la Figura D.1-c, que para mantener al menos dos de estos bloques, elimina el segundo filtro que compone al *Basic Block* original, manteniendo solamente el que se utiliza para *down-sample* (*stride* 2).

Layer	***Type***	***Kernel***	***Stride***	***Features***
Layer 1	Filter + BatchNorm (+ ReLU)	7×7	4	16
	MaxPool	3×3	2	-
Layer 2	Basic Block (Reduced)	3×3	2	32
	Downsample	1×1	2	32
Layer 3	Basic Block (Reduced)	3×3	2	64
	Downsample	1×1	2	64
Layer 4	Adaptive AvgPool	7×7	-	-
	Fully Connected	64	-	# Classes

Tabla D.2: Arquitectura de ResNet04.

En este Apéndice también se muestra una comparación de la memoria (estimada) necesaria para almacenar los parámetros de los modelos ResNet y sus variantes con funciones ChSymSim aquí propuestos (ver Tabla D.6), considerando a las capas

Layer	***Type***	***Kernel***	***Stride***	***Features***
Layer 1	Filter + BatchNorm (+ ReLU)	7×7	4	16
	MaxPool	3×3	2	-
Layer 2	Basic Block Downsample	3×3	2	32
		3×3	1	32
		1×1	2	32
Layer 3	Basic Block Downsample	3×3	2	64
		3×3	1	64
		1×1	2	64
Layer 4	Adaptive AvgPool	7×7	-	-
	Fully Connected	64	-	# Classes

Tabla D.3: Arquitectura de ResNet06.

Layer	***Type***	***Kernel***	***Stride***	***Features***
Layer 1	Filter + BatchNorm (+ ReLU)	7×7	2	16
	MaxPool	3×3	2	-
Layer 2	Basic Block Downsample	3×3	2	32
		3×3	1	32
		1×1	2	32
Layer 3	Basic Block Downsample	3×3	2	64
		3×3	1	64
		1×1	2	64
Layer 4	Adaptive AvgPool	4×4	-	-
	Fully Connected	64	-	# Classes

Tabla D.4: Arquitectura de ResNet06 para entrada de tamaño 64×64.

Layer	***Type***	***Kernel***	***Stride***	***Features***
Layer 1	Filter + BatchNorm (+ ReLU)	7×7	4	16
	MaxPool	3×3	2	-
Layer 2	Basic Block	3×3	1	16
		3×3	1	16
Layer 3	Basic Block Downsample	3×3	2	32
		3×3	1	32
		1×1	2	32
Layer 4	Basic Block Downsample	3×3	2	64
		3×3	1	64
		1×1	2	64
Layer 5	Adaptive AvgPool	7×7	-	-
	Fully Connected	64	-	# Classes

Tabla D.5: Arquitectura de ResNet08.

de BatchNorm ya fusionadas con los filtros correspondientes e ignorando capas de MaxPool y AvgPool. Al exportar estas redes para su implementación, es posible que parámetros adicionales tales como configuraciones de capa (tamaño de kernel, *stride*, *padding*, etc.) puedan incurrir en valores de memoria superiores a los estimados en la Tabla D.6, pero manteniéndose en el mismo orden de magnitud, por lo que tales estimaciones siguen siendo de utilidad a la hora de inferir la memoria necesaria para almacenar la red.

	float32		***float16***		***int8***	
Model	**Conv.**	**ChSymSim**	**Conv.**	**ChSymSim**	**Conv.**	**ChSymSim**
ResNet18	2,807	-	1,403	-	0,703	-
ResNet04	0,114	0,128	0,057	0,066	0,029	0,035
ResNet06	0,299	0,339	0,150	0,174	0,075	0,092
ResNet08	0,318	0,360	0,159	0,185	0,080	0,098

Tabla D.6: Requisitos de memoria en [MB] para la ejecución de algunas arquitecturas ResNet.

Apéndice E

Unidades de escalado, redondeo y ReLU

Al efectuar diferentes tipos de cómputo, como por ejemplo las multiplicaciones y sumas en las capas tradicionales en Redes Neuronales, la precisión de los resultados de dichas operaciones se vuele significativamente superior al de las entradas y parámetros, debido a que se requieren mayor cantidad de bits para representar estos valores sin perder precisión o incurrir en errores de *overflow* o *underflow*. Algunos ejemplos de esto son las sumas, que requieren de $\log_2(N)$ bits adicionales para representar el resultado de sumar N números, o la multiplicación, que requiere $p+q$ bits totales para representar el producto de dos números (uno con p bits y el otro con q bits). Es debido a esto que, al ser estas operaciones parte de una serie de cómputos en un algoritmo más complejo, como las capas de una DNN, se vuelve indispensable el re-convertir o re-escalar las salidas de estos cómputos a la misma precisión (bits) de las entradas, y que puedan entonces ser utilizados para cálculos posteriores.

En base a esto se diseñó una unidad de escalado y redondeo como la mostrada en la Fig. E.1, que posee tres secciones principales: escalado, redondeo y *clip*. La primera de estas secciones se ocupa de aplicar el escalado propiamente dicho al valor de entrada del bloque (salida de los PEs del acelerador). Esto se realiza extendiendo dicho valor en sus bits menos significativos con 0, consiguiendo así un número con el doble de bits en los que la mitad más significativa es la entrada original y la mitad menos significativa son 0, para finalmente aplicar el factor de escala (potencia de

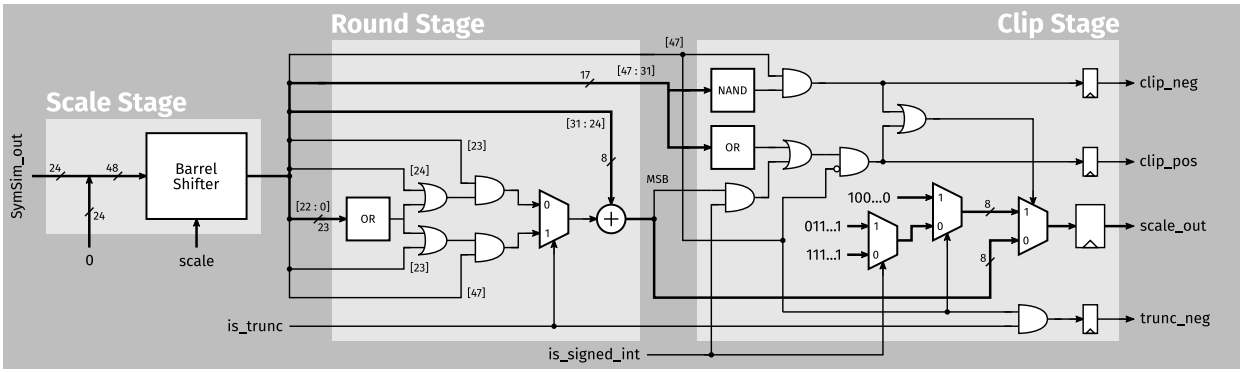


Figura E.1: Diseño de la unidad en *hardware* para escalado y redondeo.

2) mediante una operación de *shift* aritmético a derecha (división) usando un *barrel shifter* similar al de la Fig. 4.3c. La segunda sección es la encargada de implementar el redondeo según la función "round()" en Python, para lo que se inicialmente se asume que la mitad de bits más significativos son la parte entera mientras que la mitad menos significativa es la parte fraccionaria, y se aplica la siguiente ecuación:

$$\text{round}(\text{int\_bits}, \text{frac\_bits}) = \begin{cases} \text{int\_bits} + 1\ , & \text{frac\_bits} > 0{,}5 \\ \text{int\_bits} + 1\ , & \text{frac\_bits} = 0{,}5\ \&\ \text{int\_bits}\,[0] \\ \text{int\_bits}\ , & \text{frac\_bits} < 0{,}5 \end{cases} , \quad \text{(E.1)}$$

donde "frac_bits" es la parte fraccionaria (mitad menos significativa), de manera que $0 \leqslant \text{frac\_bits} < 1$, "int_bits" es la parte entera (mitad más significativa), con "int_bits [0]" como el bit menos significativo e indicando si la parte entera es un valor par o impar (0 o 1 respectivamente). Este bloque de redondeo en la Fig. E.1 también puede realizar la operación de truncado, seleccionando solamente la parte entera (int_bits) cuando el valor es positivo y "int_bits + 1" cuando el valor a truncar es negativo. La última etapa del circuito de escalado y redondeo lleva a cabo una saturación o *clip* del valor redondeado (o truncado) en base a los rangos de cuantización (configurables), comparando por mayor y menor con los rangos máximo y mínimo, respectivamente. Al final de este bloque, la salida escalada, redondeada (o truncada) y saturada (según corresponda) es registrada para introducir un *pipeline* con los bloques posteriores, además de producir señales (*flags*) que dan aviso si se truncó un valor negativo, o si se produjo la saturación (*clip*) por caso positivo (rango máximo) o negativo (rango mínimo).

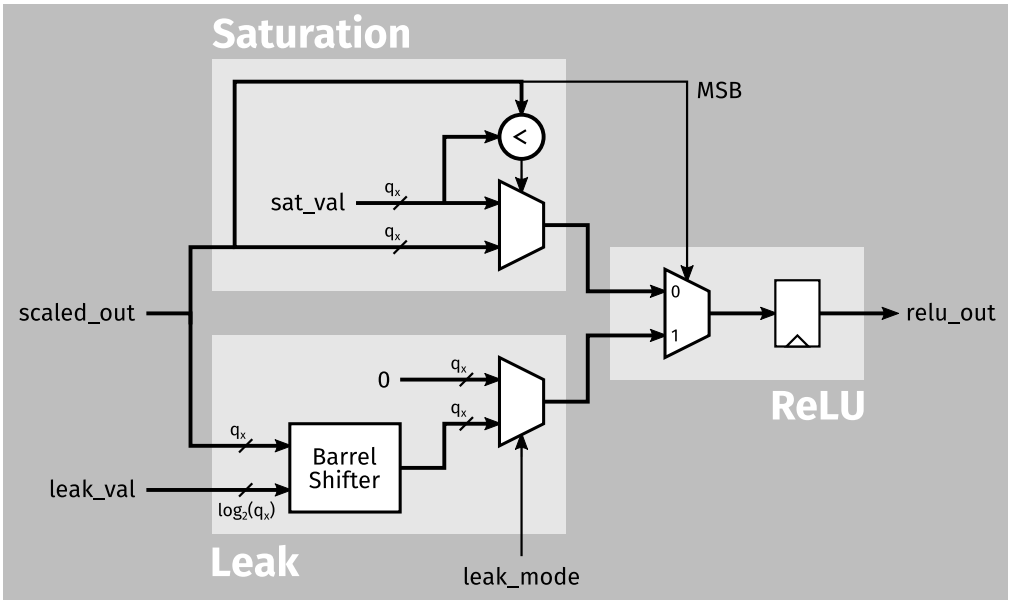


Figura E.2: Arquitectura en *hardware* para el cómputo de activaciones ReLU.

En Redes Neuronales, las capas más usadas son FC y convolucionales, que son operaciones intrínsecamente lineales. Es por esto que, para obtener el efecto de "neurona artificial" y capas, se emplean funciones de activación (no lineales), con la operación ReLU y sus variantes como las activaciones más populares. Si bien la función ChSymSim propuesta en esta tesis ya es de carácter no lineal, la baja complejidad en la implementación de un módulo que ejecute una activación PReLU, en contraste con algunas ventajas que esta activación ofrece, tales como acotar rangos de entrada/salida y mejorar el rendimiento general de la red (modelos más complejos), es motivo suficiente para diseñar una unidad de procesamiento para la función PReLU y añadirla al sistema del acelerador ChSymSim. Para poder realizar este tipo de activaciones, se diseñó un bloque en *hardware* como el presentado en la Fig. E.2, que posee dos bloques en paralelo: *leakage* para valores negativos y saturación para valores positivos, seleccionando al final entre la rama positiva o la negativa según el bit de signo (bit más significativo). Por un lado, la saturación de valores positivos se implementa mediante la comparación con el valor máximo permitido (configurable), seleccionando entre entrada original o valor de saturación según el resultado de la comparación. Por otra parte, el *leakage* se realiza con una división por un factor de escala utilizando un *shift* aritmético a derecha. Finalmente se selecciona entre este valor escalado o 0, según se desee realizar una ReLU tradicional o PReLU (*leaky*

ReLU).

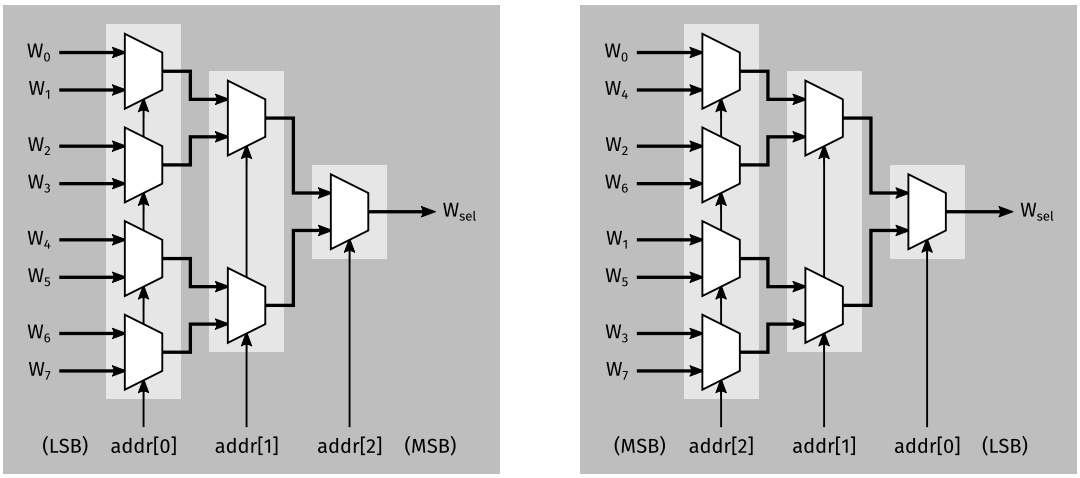
Debido a que el efecto de *leakage* es una división (por potencia de 2), es necesario extender la precisión a la salida para no introducir errores de truncado adicionales (no implementado en la Fig. E.2), por lo que resultaría conveniente el primero realizar la activación para posteriormente escalar y redondear la salida. Además de esta consideración, debido a la similitud del *hardware* para computar ambos bloques, para la implementación final se combinaron ambos bloques (Figs. E.1 y E.2) en una sola unidad de de escalado, redondeo y ReLU. Para esto se aplica el efecto de *leakage* de la función PReLU como un factor de escala que multiplica al original, sumando el número de bits a desplazar solo si la entrada es un valor negativo, re-utilizando así el *barrel shifter* de la etapa de escalado. La saturación de valores positivos en la ReLU, en cambio, se realiza por medio de la comparación por rango mayor, original de la sección de *clip* en la unidad de escalado y redondeo, de manera que no se añade hardware adicional respecto de la implementación del escalado y redondeo de la Fig. E.1. De esta forma se consigue una unidad completa que implementa ambos bloques con casi la misma complejidad que el módulo de escalado y redondeo de la Fig. E.1, añadiendo algunas compuertas adicionales para la implementación del *leakage* en la PReLU, tales como un sumador para el factor de escala (bits a desplazar) y algunos selectores (MUX).

Apéndice F

Optimizaciones de selectores en *hardware*

En la mayoría de las implementaciones de operadores de Redes Neuronales, ya sean para el cómputo de funciones lineales (FC o convolucionales), como las propuestas por esta tesis para el algoritmo ChSymSim, se cuentan con registros internos de entradas/parámetros para los que deben seleccionarse una salida por ciclo de reloj, generalmente mediante multiplexores. Debido a las grandes dimensiones de estos multiplexores y que generalmente son accedidos de forma continua con una dirección diferente, pueden representar un consumo de potencia elevado. Es por esto que resulta importante el optimizar el consumo energético de estos bloques, aprovechando el conocimiento que se posee a cerca de los patrones de acceso a los mismos.

En casi todos los casos de acceso de datos por medio de un multiplexor, dichos datos en memoria o registros internos se acceden de forma secuencial, con direcciones que provienen de un contador, por lo que estas direcciones presentan incrementos (o decrementos) constantes. Incluso si se realizan saltos de memoria en las lecturas, se mantiene la tendencia monótonamente creciente (o decreciente) en los valores de la dirección del dato a seleccionar. La Figura F.1a muestra un ejemplo de multiplexor (MUX) estándar de 8 entradas (pesos W), con implementaciones en forma de árbol de multiplexores 2:1 (caso usual), donde los datos de entrada se encuentran dispuestos de forma ordenada. De esta forma, para obtener el dato correspondiente a la



(a) (b)

Figura F.1: Selectores de pesos: a) Selector tradicional (entradas ordenadas); b) Selector optimizado para lecturas con dirección en incrementos.

dirección dada, la primera etapa del árbol de multiplexores se conecta al LSB[1] que se encuentra en la posición 0, conectando los demás bits a las siguientes etapas del árbol, terminando con el MSB[2] para la selección final en la última etapa con un solo MUX 2:1. Teniendo generalmente incrementos unitarios (+1) en los valores de la dirección, resulta evidente que los valores seleccionados en la primera etapa del multiplexor cambiarán para todos los ciclos de reloj, debido a las contínuas transiciones en el bit "addr [0]" de 0 a 1 y viceversa. Estos cambios en los valores seleccionados en la primera etapa del árbol se propaga a las etapas posteriores, teniendo así cambios (transiciones) en prácticamente todos los nodos del circuito por cada ciclo de reloj (factor de actividad de 1), lo que produce un notable consumo de potencia.

Sin embargo, al conocer este efecto de cambios constantes en el LSB de la dirección (bits de selección), se puede realizar una optimización en términos de consumo energético con simplemente cambiar las conexiones entre los bits de la dirección y las etapas del multiplexor, tal y como se muestra en la Fig. F.1b. Para lograr esta implementación del multiplexor, los bits de la dirección deben ser espejados, al igual que los bits de los índices que representan las posiciones de las entradas. Considerando este ejemplo de 8 entradas, se alternan las entradas en las posiciones $1 = (001)_b$ con $4 = (100)_b$ y $3 = (011)_b$ con $6 = (110)_b$, mientras que se mantienen las mismas

[1]Bit menos significativo - *Least Significant Bit*
[2]Bit más significativo - *Most Significant Bit*

entradas en las posiciones $0 = (000)_b$, $2 = (010)_b$, $5 = (101)_b$ y $7 = (111)_b$, ya que sus índices son los mismos a pesar del espejado de bits.

Algoritmo 6 Cálculo de la actividad en un multiplexor optimizado

Entrada:

- Dimensiones del multiplexor en forma de árbol de MUX 2:1 (cantidad de entradas in_num).

Salida:

- Actividad teórica del multiplexor (mux_act), el cual fue optimizado para lecturas con direcciones incrementales.

stage_num $\leftarrow \lceil \log_2 (\text{in\_num}) \rceil$
mux_act $\leftarrow 0$
mux_nodes $\leftarrow 0$
for stage $= 0 : (\text{stage\_num} - 1)$ **do**
  node_num $\leftarrow 2^{\text{stage}}$
  toggle_act $\leftarrow {}^{1}/_{\text{node\_num}}$
  if $[\text{stage} = (\text{stage\_num} - 1)]$ & $[(\text{in\_num} \,\%\, 2^{\text{stage}}) > 0]$ **then**
    node_num $\leftarrow \text{in\_num} \,\%\, 2^{\text{stage}}$
  end if
  mux_nodes $\leftarrow$ mux_nodes + node_num
  mux_act $\leftarrow$ mux_act + node_num $\times$ toggle_act
end for
return mux_act

Al tener el MSB de la dirección (que menos cambios presenta) en la primera etapa, los valores seleccionados en dicha etapa se mantienen constantes por aproximadamente la mitad del ciclo de lectura, en caso de que todos los valores sean leídos. Propagando los cambios en los nodos del circuito mostrado en la Fig. F.1b a lo largo de un ciclo completo de lectura, con incrementos unitarios en los valores de la dirección, se tiene como resultado una actividad teórica que puede estimarse por medio del Algoritmo 6. Para dicho algoritmo, se calcula la cantidad de etapas que tiene el multiplexor, y se suma la cantidad de nodos por etapa con su actividad (según los cambios en el bit de la dirección que corresponda), para finalmente realizar un promedio entre todos los nodos del multiplexor. Cuando se tiene un número

de entradas que no es potencia de 2, la cantidad de nodos de la última etapa es menor y se calcula como el resto de la división entre el número de entradas "in_num" y la cantidad de nodos máxima para dicha etapa (2^{stage}).

En el caso de 8 entradas, mostrado en la Fig. F.1b, esta actividad es de aproximadamente 0,43, lo que resulta considerablemente menor al caso del multiplexor clásico de la Fig. F.1a. Este factor de actividad se reduce considerablemente a medida que las dimensiones de entrada del multiplexor aumentan. Estos patrones de dirección incremental también aplica a lectura de los resultados u otros datos del acelerador, tales como entradas, pesos o configuraciones, debiendo seleccionar varios datos consecutivos a la vez para llenar el *bus* de salida, por lo que esta optimización del multiplexor por un simple ordenamiento de entradas es útil para más casos además de la selección de pesos/entradas para los PEs del acelerador.

Para el caso particular del acelerador ChSymSim propuesto en esta tesis (Sección 4.1), se tiene una separación de los PE en dos módulos: uno para la generación de la dirección del peso (*Address Encoder*) y otro para su selección y acumulación (*Mux. Accumulator*). Aprovechando que el bloque de generación de dirección de pesos es compartido por todos los *features*, se puede utilizar otro tipo de implementación de selector como el mostrado en la Fig. F.2. En este tipo de selector, la dirección es codificada en modo *one-hot*, es decir 1 en la posición seleccionada y 0 en el resto, que luego se usa como máscara (compuertas AND) para anular las entradas no deseadas, obteniendo así la salida objetivo por medio de la operación OR entre todas las entradas enmascaradas, que generalmente es implementada como árbol de compuertas OR. Con este modelo, se puede utilizar el sub-bloque de codificación *one-hot* directamente a la salida de los módulos *Address Encoder*, mientras que se simplifican los multiplexores de los *Mux. Accumulator* (repetidos por cada *feature* de salida) a simplemente la máscara con compuertas AND y el selector como árbol de compuertas OR. Esto también representa un consumo de potencia inferior respecto al selector tradicional con multiplexores de la Fig. F.1a, debido a que por cada ciclo de selección de entradas, solamente se producen transiciones en dos de las ramas del árbol de selección (compuertas OR): la nueva entrada seleccionada se activa mientras que se anula la selección anterior, con todas las otras ramas en 0 debido a las máscara de compuertas AND.

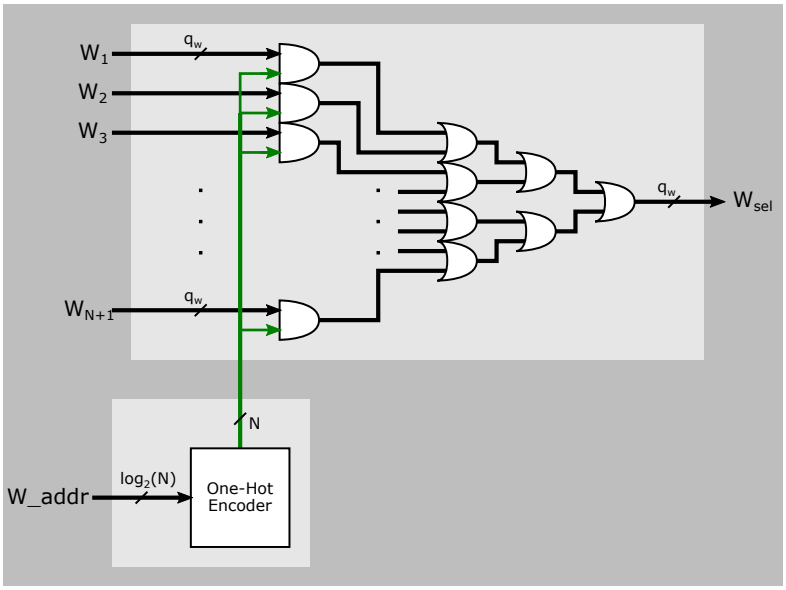


Figura F.2: Selector de pesos *One-Hot*, separado en codificación de dirección y selector propiamente dicho (compuertas AND y OR).

Para concluir este análisis de diseño de selectores, se realizó la síntesis (con celdas de tecnología CMOS 65 nm) y simulación de tres selectores diferentes: multiplexor tradicional, multiplexor optimizado mediante re-ordenamiento de entradas y bits de la dirección, y finalmente la variante con codificación de dirección *one-hot*. Los resultados en cuanto al consumo de potencia de estos selectores puede observarse en la Tabla F.1, para la que se sintetizaron todos los selectores para 25 entradas de 8 bits y se simuló la selección de todas las entradas de forma secuencial, por medio de 10 iteraciones diferentes con nuevos conjuntos de entradas aleatorias para cada iteración. De estos resultados se puede apreciar que, si bien el factor de reducción de potencia en el multiplexor optimizado (Fig. F.1b) no es el estimado por el Algoritmo 6, donde para 25 entradas se tiene una actividad de alrededor de 0,19, se sigue obteniendo menor consumo de potencia en comparación con la implementación estándar. Esta diferencia puede deberse entre la actividad predicha y el valor simulado de potencia puede deberse a que la herramienta de síntesis realiza optimizaciones adicionales, por lo que los circuitos resultantes pueden diferir de los equivalentes a las Figs. F.1a y F.1b. Como era de esperarse, la implementación *One-Hot* consigue menor potencia que el diseño de multiplexor estándar, aunque sigue siendo ligeramente superior a la versión optimizada por medio del cambio en las posiciones de las entradas.

En base a los resultados de la simulación post-síntesis que se muestran en la Tabla

Tipo de MUX	Potencia [μW]
Estándar	33,98
Optimizado	15,46
One-Hot	16,90

Tabla F.1: Consumo de potencia (simulación post-síntesis) de multiplexores con 25 entradas de 8 bits, con 10 iteraciones de entradas aleatorias.

F.1, se recomienda utilizar alguna de estas técnicas de implementación de selectores, siempre que esto sea posible. Para casos en los que se seleccionan varios datos a la vez, utilizando una misma dirección, se le puede sacar más provecho a la implementación *One-Hot*, que además de tener generalmente menor consumo, se repite una sub-bloque de menor complejidad en *hardware* (máscara y árbol con compuertas AND y OR) en lugar de repetir el multiplexor completo. En situaciones donde la versión *One-Hot* no puede aprovecharse, como por ejemplo si tanto direcciones como datos son diferentes, al re-ordenar las entradas acorde a lo mostrado en la Fig. F.1b, se puede obtener menores consumos energéticos que con el multiplexor tradicional. El diseño y simulación de un modelo de selector que combine ambas técnicas, es decir *One-Hot* y re-ordenamiento de entradas/máscara, queda pendiente como trabajo posterior a esta tesis.

Apéndice G

Conexionado de los SoC fabricados

En este apéndice se presentan los diagramas de conexión de señales a los *pads* de los circuitos integrados fabricados durante el trabajo de esta tesis (DigineuronV1, V2 y V3a). En esta sección se muestran también los pines y las correspondientes señales asociadas en los encapsulados que contienen los *chips* ya mencionados.

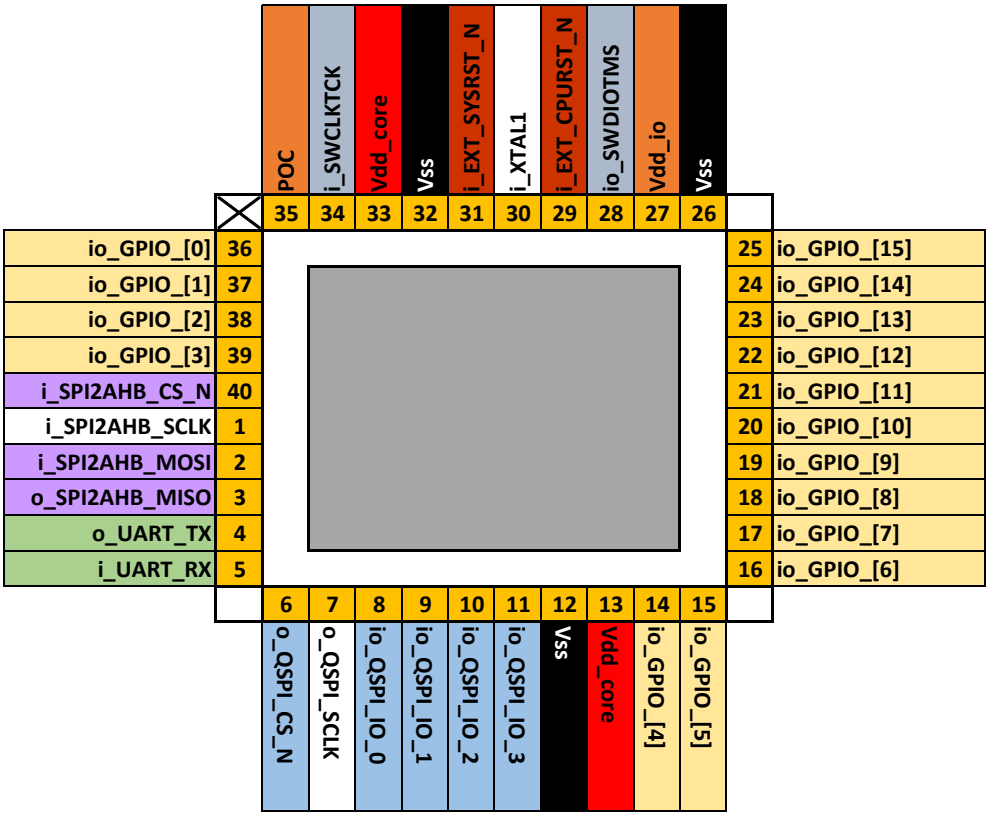


Figura G.1: Ubicación de las señales en el SoC DigineuronV1: esquema de *pads*.

i_SPI2AHB_SCLK	1		40	i_SPI2AHB_CS_N
i_SPI2AHB_MOSI	2		39	io_GPIO_[3]
o_SPI2AHB_MISO	3		38	io_GPIO_[2]
o_UART_TX	4		37	io_GPIO_[1]
i_UART_RX	5		36	io_GPIO_[0]
o_QSPI_CS_N	6		35	POC
o_QSPI_SCLK	7		34	i_SWCLKTCK
io_QSPI_IO_0	8		33	Vdd_core
io_QSPI_IO_1	9		32	Vss
io_QSPI_IO_2	10		31	i_EXT_SYSRST_N
io_QSPI_IO_3	11		30	i_XTAL1
Vss	12		29	i_EXT_CPURST_N
Vdd_core	13		28	io_SWDIOTMS
io_GPIO_[4]	14		27	Vdd_io
io_GPIO_[5]	15		26	Vss
io_GPIO_[6]	16		25	io_GPIO_[15]
io_GPIO_[7]	17		24	io_GPIO_[14]
io_GPIO_[8]	18		23	io_GPIO_[13]
io_GPIO_[9]	19		22	io_GPIO_[12]
io_GPIO_[10]	20		21	io_GPIO_[11]

Figura G.2: Ubicación de las señales en el SoC DigineuronV1: diagrama de encapsulado DIL40.

POC	1		40	Vdd_core
Vss	2		39	Vss
Vdd_io	3		38	io_GPIO_[15]
o_QSPI_CS_N	4		37	io_GPIO_[14]
o_QSPI_SCLK	5		36	io_GPIO_[13]
io_QSPI_IO_0	6		35	io_GPIO_[12]
io_QSPI_IO_1	7		34	io_GPIO_[11]
io_QSPI_IO_2	8		33	io_GPIO_[10]
io_QSPI_IO_3	9		32	io_GPIO_[9]
i_EXT_SYSRST_N	10		31	io_GPIO_[8]
i_XTAL1	11		30	io_GPIO_[7]
o_UART_TX	12		29	io_GPIO_[6]
i_UART_RX	13		28	io_GPIO_[5]
i_SPI2AHB_MOSI	14		27	io_GPIO_[4]
o_SPI2AHB_MISO	15		26	io_GPIO_[3]
i_SPI2AHB_SCLK	16		25	io_GPIO_[2]
i_SPI2AHB_CS_N	17		24	io_GPIO_[1]
i_EXT_CPURST_N	18		23	io_GPIO_[0]
Vdd_core	19		22	Vdd_core
Vss	20		21	Vss

Figura G.3: Ubicación de las señales en el SoC DigineuronV2: diagrama de encapsulado DIL40.

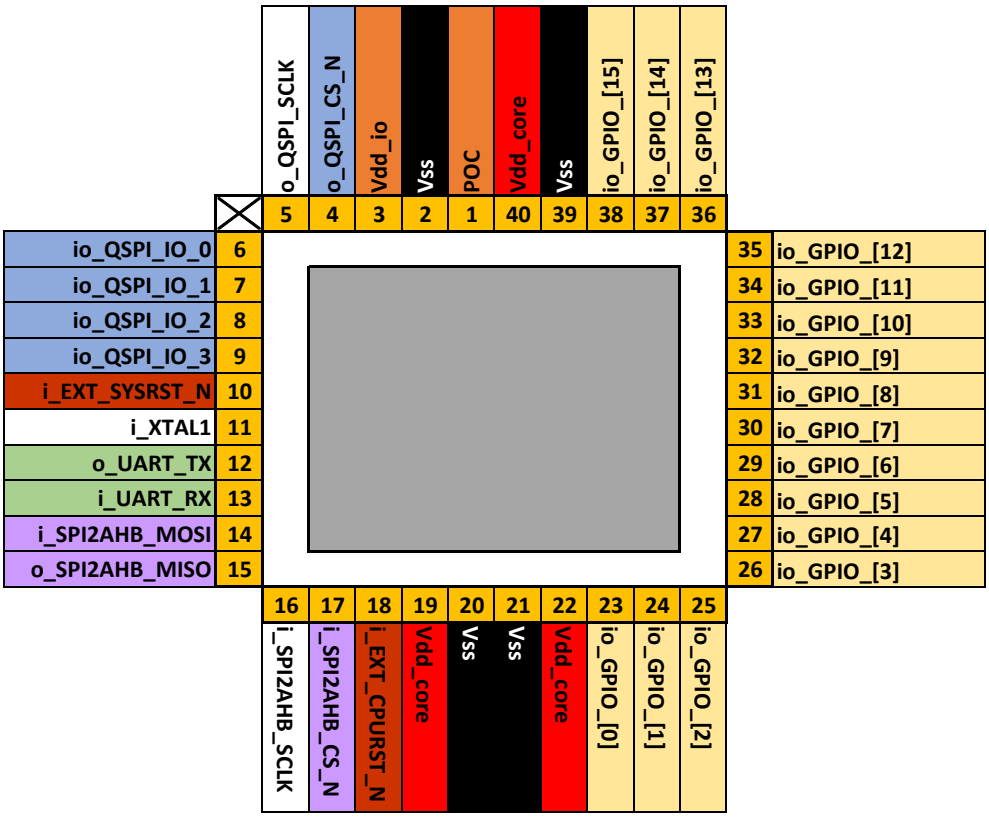


Figura G.4: Ubicación de las señales en el SoC DigineuronV2: esquema de *pads*.

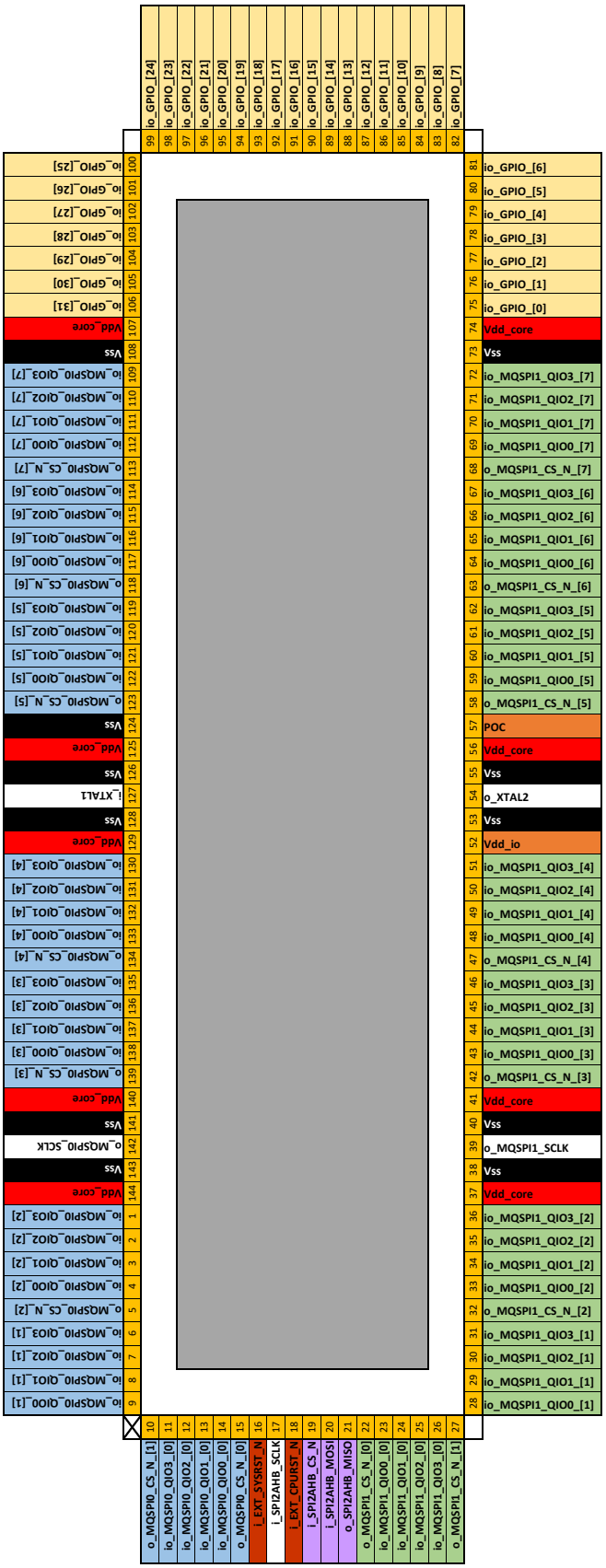


Figura G.5: Ubicación de las señales en los *pads* del SoC DigineuronV3a

Figura G.6: Ubicación de las señales en los *pins* de DigineuronV3a, con encapsulado CPGA144.

Apéndice H

DigineuronV3a: Mapas de memoria del SoC

En este apéndice se introducen los mapas de memoria de los módulos integrados en el SoC DigineuronV3a.

En la Tabla H.1 se presentan las direcciones en las cuales se ubican los bloques subordinados (*slaves*) del sistema AHB, mientras que en la Tabla H.2 se ilustra el acceso permitido a estos por parte de las unidades de control (*masters*). A pesar de que algunos registros o memorias son inferiores a los $16KiB$, se eligió este valor para el salto de memoria mínimo entre *slaves* para simplificar el *hardware* (*decomux*) de la matriz AHB. En dicha tabla se especifica el tipo de árbitro que le corresponde a cada *slave* del sistema, contando con prioridad simple (SP[1]) o "Round-Robin" (WRR[2]) con pesos que se especifican en los registros de configuración GCONF (subsistema APB).

En la Tabla H.3 se presenta el mapa de memoria del sub-sistema APB dedicado a periféricos que requieren de menor velocidad de transferencia de datos. De igual forma que con el *bus* AHB, para simplificar el *hardware* de direccionamiento, se utilizan saltos de memoria de $64KiB$ para las direcciones de todos los módulos en el *bus* APB, aún si estos tienen considerablemente menos registros.

Los registros de configuración del sistema se encuentras separados en dos módulos: GCONF y SysCTRL (o *sys_ctrl* en Tabla H.1). Por un lado, el banco de registros

[1] *Simple Priority*
[2] *Weighted Round Robin*

Nombre	Dirección	Árbitro	Descripción
program_mem	0x00000000	SP	Memoria SRAM para el programa del microprocesador
data_mem	0x00004000	SP	Memoria SRAM para el almacenamiento general de datos
core_mem_0	0x00008000	SP	Memoria SRAM para datos del acelerador (entrada, parámetros y/o salidas)
core_mem_1	0x0000C000	SP	Memoria SRAM para datos del acelerador (entrada, parámetros y/o salidas)
core_mem_2	0x00010000	SP	Memoria SRAM para datos del acelerador (entrada, parámetros y/o salidas)
core_mem_3	0x00014000	SP	Memoria SRAM para datos del acelerador (entrada, parámetros y/o salidas)
core_mem_4	0x00018000	SP	Memoria SRAM para datos del acelerador (entrada, parámetros y/o salidas)
core_mem_5	0x0001C000	SP	Memoria SRAM para datos del acelerador (entrada, parámetros y/o salidas)
symsim_conf	0x10000000	WRR	Registros de configuración del acelerador ChSymSim
symsim_params	0x10010000	WRR	Registros de parámetros del acelerador ChSymSim (SM, SE, pesos simétricos, *bias*)
symsim_inputs	0x10020000	WRR	Registros de entradas (1 canal) del acelerador ChSymSim
symsim_outputs	0x10030000	WRR	Registros de salidas del acelerador ChSymSim (resultados escalados, *padding*, resultados sin escalar, dirección de pesos, entradas codificadas en PWM)
apb_subsys	0x20000000	WRR	Sub-sistema conectado con interfaz APB, contiene módulos tales como DMAs (configuración), temporizadores y GPIO
multi_qspi_0	0x30000000	WRR	Registros de configuración y memorias FIFO (entrada/salida) del bloque MQSPI-0
multi_qspi_1	0x30004000	WRR	Registros de configuración y memorias FIFO (entrada/salida) del bloque MQSPI-1

Tabla H.1: Mapa de memoria del *bus* AHB.

*Slave**Master*	*spi2ahb*	*dma0_rd*	*dma0_wr*	*dma1_rd*	*dma1_wr*	*μP_inst*	*μP_data*
program_mem	✓	✓	✓	✓	✓	✓	✓
data_mem	✓	✓	✓	✓	✓	✓	✓
core_mem_0	✓	✓	✓	✓	✓	✓	✓
core_mem_1	✓	✓	✓	✓	✓	✓	✓
core_mem_2	✓	✓	✓	✓	✓	✓	✓
core_mem_3	✓	✓	✓	✓	✓	✓	✓
core_mem_4	✓	✓	✓	✓	✓	✓	✓
core_mem_5	✓	✓	✓	✓	✓	✓	✓
symsim_conf	✓						✓
symsim_params	✓	✓	✓	✓	✓		✓
symsim_inputs	✓	✓	✓	✓	✓		✓
symsim_outputs	✓	✓	✓	✓	✓		✓
apb_subsys	✓						✓
multi_qspi_0	✓	✓	✓				✓
multi_qspi_1	✓			✓	✓		✓

Tabla H.2: Mapa de acceso del *bus* AHB.

Nombre	Dirección	Descripción
dma0	`0x00000000`	Registros de configuración del controlador de acceso directo a memoria DMA-0
dma1	`0x00010000`	Registros de configuración del controlador de acceso directo a memoria DMA-1
gconf	`0x00020000`	Registros de configuración generales del sistema: Prioridades para los árbitros de la matriz AHB y registros BIST de las memorias SRAM
sys_ctrl	`0x00030000`	Registros de control del sistema (configuración del microprocesador e interrupciones)
timer0	`0x00040000`	Temporizador con contador (decreciente) de 32 bit, capaz de generar interrupciones al microprocesador cuando la cuenta llega a 0
timer1	`0x00050000`	Temporizador con contador (decreciente) de 32 bit, capaz de generar interrupciones al microprocesador cuando la cuenta llega a 0
watchdog	`0x00060000`	Temporizador *watchdog* con contador (decreciente) de 32 bit. Genera una interrupción al contar hasta 0 y se re-inicia la cuenta. Si no se realiza un *clear* en el registro de interrupción, al volver a contar hasta 0 se activa la señal de *reset*
gpio	`0x00070000`	Puertos de entrada/salida (32 bits) de propósito general, capaz de generar interrupciones ante cambios (*toggle*, *level* o *edge*) en dichos puertos

Tabla H.3: Mapa de memoria del *bus* APB, con dirección base `0x20000000`.

denominado GCONF contiene las prioridades de los *slaves* del sistema AHB que los árbitros utilizan cuando se dan transacciones simultáneas, así como también los registros BIST para depuración de las memorias SRAM. Por otra parte, el bloque SysCTRL (Tabla H.4) contiene registros de configuración y control del microprocesador y su selección de interrupciones.

Nombre	Dirección	Modo	Descripción
hart_id	0x000	R/W	*Hardware Thread ID* para la identificación del núcleo de procesamiento. Utilizado mayormente en sistemas *multi-core*
boot_addr	0x004	R/W	Dirección base donde se inicial el programa cuando el μP sale del estado de *reset*
mtvec_addr	0x008	R/W	Dirección base donde se ubican las instrucciones (*Machine Trap-Vector*) a ser ejecutadas ante interrupciones/excepciones en el μP
dm_halt_addr	0x00C	R/W	Dirección base a donde el μP accede cuando se inicia el modo *Debug*, quedando suspendido a la espera de comandos por parte del *Debugger*
dm_except_addr	0x010	R/W	Dirección base a donde el μP accede cuando se produce una interrupción o excepción, estando en modo *Debug*
irq_mode	0x014	R/W	Modo de operación de interrupciones: selección en multiplexor de interrupciones
irq_mask	0x018	R/W	Máscara de habilitación de interrupciones
software_irq	0x01C	R/W	Interrupciones por software, la interrupción combinada debe ser habilitada con *irq_mask*[3]
last_irq_id	0x020	R/	ID de la última interrupción atendida por el μP

Tabla H.4: Mapa de memoria del módulo SysCTRL, con dirección base 0x20030000.

En la Tabla H.5 se incluye un mapa de las interrupciones al microprocesador que pueden utilizar los *slaves* del sistema AHB. El μP cv32e40p [66] tiene un vector de 32 bits de interrupciones, con la máxima prioridad con la interrupción en el bit 31 y disminuyendo en orden descendente (mínima prioridad en bit 3). Algunas de las 32 interrupciones posibles se encuentran reservadas y su acceso se halla deshabilitado por *hardware*, con valor fijo en 0 lógico a la entrada del vector de interrupciones. Los bits de interrupciones disponibles son entonces los que se ubican desde la posición 31 a la 16 que corresponden a las interrupciones rápidas (F-IRQ^3), la posición 11

que se define como interrupción externa al *chip* (MEI[4]), la posición 7 dedicada a las interrupciones por temporizadores (MIT[5]) y la posición 3 descrita como interrupción por *software* (MSI[6]).

Posición \ *Modo*	0	1	2	3	Descripción
31	*watchdog*	*watchdog*	*watchdog*	*watchdog*	F-IRQ 15
30	*symsim_scale*	*symsim_scale*	*symsim_comb*	*symsim_comb*	F-IRQ 14
29	*symsim_outputs*	*symsim_outputs*	*dma_0*[0]	*dma_0*[4]	F-IRQ 13
28	*symsim_bias*	*symsim_bias*	*dma_0*[1]	*dma_0*[5]	F-IRQ 12
27	*symsim_se*	*symsim_se*	*dma_0*[2]	*dma_0*[6]	F-IRQ 11
26	*symsim_weights*	*symsim_weights*	*dma_0*[3]	*dma_0*[7]	F-IRQ 10
25	*symsim_inputs*	*symsim_inputs*	*dma_1*[4]	*dma_1*[0]	F-IRQ 9
24	*mqspi_0*[0]	*mqspi_1*[0]	*dma_1*[5]	*dma_1*[1]	F-IRQ 8
23	*mqspi_0*[1]	*mqspi_1*[1]	*dma_1*[6]	*dma_1*[2]	F-IRQ 7
22	*mqspi_0*[2]	*mqspi_1*[2]	*dma_1*[7]	*dma_1*[3]	F-IRQ 6
21	*mqspi_0*[3]	*mqspi_1*[3]	*mqspi_0_comb*	*mqspi_0_comb*	F-IRQ 5
20	*mqspi_1_comb*	*mqspi_0_comb*	*mqspi_1_comb*	*mqspi_1_comb*	F-IRQ 4
19	*dma_0_comb*	*dma_0_comb*	*dma_0_comb*	*dma_0_comb*	F-IRQ 3
18	*dma_1_comb*	*dma_1_comb*	*dma_1_comb*	*dma_1_comb*	F-IRQ 2
17	*timer_0*	*timer_0*	*timer_0*	*timer_0*	F-IRQ 1
16	*timer_1*	*timer_1*	*timer_1*	*timer_1*	F-IRQ 0
15-12	0	0	0	0	Reservadas
11	*gpio_comb*	*gpio_comb*	*gpio_comb*	*gpio_comb*	MEI
10-8	0	0	0	0	Reservadas
7	*timer_comb*	*timer_comb*	*timer_comb*	*timer_comb*	MIT
6-4	0	0	0	0	Reservadas
3	*software_comb*	*software_comb*	*software_comb*	*software_comb*	MSI
2-0	0	0	0	0	Reservadas

Tabla H.5: Mapa interrupciones del microprocesador: selección según el modo de operación.

En el caso de las interrupciones que presentan el sufijo "'*comb*", estas se refieren las generadas por la combinación (OR) de todas las interrupciones del módulo indicado en el nombre. Esto se realiza para dar mayor foco a otras interrupciones (ejecución directa), mientras que las "combinadas", al no ser tan relevantes, requieren que se lea el registro de interrupción del módulo en cuestión para poder proceder con la ejecución del código de excepción que corresponda. De forma similar, la interrupción por *software* es implementada como la combinación de los bits del registro "*software_irq*" en el bloque SysCTRL, por lo que una vez esta interrupción se activa, es necesario obtener primero el valor del registro "*software_irq*" para ejecutar la rutina que corresponda.

[3] *Fast Interrupt Request*
[4] *Machine External Interrupt*
[5] *Machine Timer Interrupt*
[6] *Machine Software Interrupt*

Apéndice I

Mapas de memoria del acelerador ChSymSim

En este apéndice se encuentran los mapas de memoria de los registros exclusivos del procesador ChSymSim, diseñado para esta tesis e integrado en el SoC DigineuronV3a. Dichos registros internos son utilizados tanto para el almacenamiento de valores de configuración y control, como para la copia local de las entradas y parámetros necesarios para realizar la operación.

De esta forma, en las Tablas I.1-I.13 se presentan las ubicaciones, descripciones y modos de acceso (esrcitura y/o lectura) de los registros de configuración del operador ChSymSim, que se ubican en la dirección base `0x10000000` del sistema AHB. Debido a que se tienen señales de configuración con longitudes mayores a 8 bits, pero menores a 16 bits, el mapa de memoria para escribir/leer dichas señales fue diseñado para transacciones de *half-words*. Además de esto, algunas direcciones de memoria se encuentran "vacias" de manera que algunas variables puedan ser accedidas de forma directa e independiente, con transacciones de como mínimo 8 bits (Byte), de acuerdo con el protocolo AMBA. Un ejemplo de esto son las direcciones de *set* y *clear* con saltos de 8 Bytes (tamaño del *bus*), lo que impide que se puedan seleccionar ambas en una sola transacción. Otro ejemplo es la separación de señales de control como *resets* de otras como configuración generales, permitiendo que dichas configuraciones puedan cambiarse sin riesgo a activar por error el *reset* de algún bloque. Es importante destacar que para las direcciones de señales del tipo *set* y

clear, durante la lectura de ambas direcciones se obtiene el mismo valor de salida del registro SR[1] al cual se dirigen dichas señales de escritura.

Nombre	Dirección	Bits	Modo	Descripción
sm_rst_n	`0x000` (*set*) `0x008` (*clr*)	`0`	$^{R}/_{W}$	*Reset* de los registros de Máscara de Forma (SM)
se_rst_n	`0x000` (*set*) `0x008` (*clr*)	`1`	$^{R}/_{W}$	*Reset* de los registros de Elemento Estructurante (SE)
w_rst_n	`0x000` (*set*) `0x008` (*clr*)	`2`	$^{R}/_{W}$	*Reset* de los registros de pesos simétricos
b_rst_n	`0x000` (*set*) `0x008` (*clr*)	`3`	$^{R}/_{W}$	*Reset* de los registros de *bias*
in_rst_n	`0x000` (*set*) `0x008` (*clr*)	`4`	$^{R}/_{W}$	*Reset* de los registros de entradas del acelerador
symsim_ctrl_rst_n	`0x000` (*set*) `0x008` (*clr*)	`5`	$^{R}/_{W}$	*Reset* del controlador del operador ChSymSim
scale_ctrl_rst_n	`0x000` (*set*) `0x008` (*clr*)	`6`	$^{R}/_{W}$	*Reset* del controlador del bloque de escalado, redondeo y ReLU.
ramp_rst_n	`0x000` (*set*) `0x008` (*clr*)	`8`	$^{R}/_{W}$	*Reset* del generador de rampa.
symsim_rst_n	`0x000` (*set*) `0x008` (*clr*)	`9`	$^{R}/_{W}$	*Reset* del arreglo de PEs y comparadores del operador ChSymSim
scale_rst_n	`0x000` (*set*) `0x008` (*clr*)	`10`	$^{R}/_{W}$	*Reset* de los registros de las unidades de escalado, redondeo y ReLU

Tabla I.1: Mapa de memoria para configuración del acelerador ChSymSim: *resets* de registros internos, controladores y bancos de memorias (*buffers*).

En particular, en la Tabla I.6 se muestran las máscaras de habilitación de los distintos sub-bloques que componen a los PEs del acelerador. Para simplificar el *hardware* de control y mantener la fácil escalabilidad del diseño (código RTL parametrizado), se optó por la generación por *software* de dichas señales. De esta forma, debe tenerse en cuenta que en el modo de operación "*big_kernel*", las unidades *Mux. Accumulator* útiles corresponden a la de los primeros pasos del filtrado, que en el caso del diseño implementado en DigineuronV3a se encuentran en los primeros 3×3 PEs. Por lo tanto, en el modo de operación "*big_kernel*", las máscaras "*mux_accu_row_mask*" y "*mux_accu_col_mask*" deberán contener el valor `0b000000111` (con *stride* 1), mientras que en el modo "*small_kernel*" dichas máscaras

[1] *set/reset*

Nombre	Dirección	Bits	Modo	Descripción
input_is_signed	0x010	0	$^R/_W$	Selector de presición (signo) de las entradas al acelerador ChSymSim: ▪ 0 : *unsigned* ▪ 1 : *signed*
output_is_signed	0x010	1	$^R/_W$	Selector de presición (signo) de las salidas escaladas del acelerador ChSymSim: ▪ 0 : *unsigned* ▪ 1 : *signed*
use_big_kernel	0x010	2	$^R/_W$	Selector para modo de *kernel* (tamaño): ▪ 0 : Modo *small* (*Addr. Encoder* independientes) ▪ 1 : Modo *big* (*Addr. Encoder* compartidos)
req_after_last_chan	0x010	3	$^R/_W$	Habilitación para pedido de datos (*request*) durante el procesamiento del último canal de entrada
extra_req_sel	0x010	10:8	$^R/_W$	Selección de la señal de *request* de datos adicional: ▪ 1 : *bias* ▪ 2 : SE ▪ 3 : pesos ▪ 4 : entradas ▪ 5 : salidas ▪ 6 : escalado ▪ default : 0 (sin *request*)
chan_num	0x012	9:0	$^R/_W$	Configuración del número de canales de entrada a computar.

Tabla I.2: Mapa de memoria (configuración general) del acelerador ChSymSim.

Nombre	Dirección	Bits	Modo	Descripción
ext_bias_done_sel	0x014	0	$^R/_W$	Habilitación de señal externa para dar aviso al acelerador de que los *bias* se encuentran disponibles para su uso (*bias_ready*)
ext_se_done_sel	0x014	1	$^R/_W$	Habilitación de señal externa para dar aviso al acelerador de que el SE (canal de entrada) se encuentran disponibles para su uso (*se_ready*)
ext_weights_done_sel	0x014	2	$^R/_W$	Habilitación de señal externa para dar aviso al acelerador de que los pesos simétricos (canal de entrada) se encuentran disponibles para su uso (*weights_ready*)
ext_inputs_done_sel	0x014	3	$^R/_W$	Habilitación de señal externa para dar aviso al acelerador de que las entradas (canal) se encuentran listas para su procesamiento (*inputs_ready*)
ext_readout_done_sel	0x014	4	$^R/_W$	Habilitación de señal externa para dar aviso al acelerador de que los resultados anteriores han sido leídos (*readout_ready*)

Tabla I.3: Mapa de memoria para configuración del acelerador ChSymSim: habilitación de señales "*transfer_done*" externas.

Nombre	Dirección	Bits	Modo	Descripción
in_prec	0x018	3:0	$^R/_W$	Configuración de la precisión de entradas: precisión general (8-1 bits)
w_prec	0x018	11:8	$^R/_W$	Configuración de la precisión de pesos simétricos: precisión general (8-1 bits)
ch_per_in_byte_prec	0x01A	3:0	$^R/_W$	Configuración de la precisión de entradas: canales por Byte (8, 4, 2 u 1 bits)
ch_per_w_byte_prec	0x01A	11:8	$^R/_W$	Configuración de la precisión de pesos simétricos: canales por Byte (8, 4, 2 u 1 bits)
ch_per_in_byte_num	0x01C	3:0	$^R/_W$	Configuración del número de canales de entradas por Byte (1, 2, 4 u 8 canales)
ch_per_w_byte_num	0x01C	11:8	$^R/_W$	Configuración del número de canales de pesos simétricos por Byte (1, 2, 4 u 8 canales)
ch_out_mode	0x01E	1:0	$^R/_W$	Configuración de *features* de salida por Byte: ▪ 0 : 1 *feature* (8-5 bits) ▪ 1 : 2 *features* (4-3 bits) ▪ 2 : 4 *features* (2 bits) ▪ 3 : 8 *features* (1 bit)

Tabla I.4: Mapa de memoria para configuración del acelerador ChSymSim: precisión de entradas y pesos simétricos.

Nombre	Dirección	Bits	Modo	Descripción
ramp_start_val	0x020	7:0	$^{R}/_{W}$	Valor de inicio de la rampa. Generalmente rango mínimo de la entrada (rampa creciente) o rango máximo (rampa decreciente)
ramp_end_val	0x020	15:8	$^{R}/_{W}$	Valor final de la rampa. Generalmente rango máximo de la entrada (rampa creciente) o rango mínimo (rampa decreciente)
ramp_is_decr	0x022	0	$^{R}/_{W}$	Selección de rampa (monótonamente) creciente o decreciente: ▪ 0 : rampa creciente ▪ 1 : rampa decreciente

Tabla I.5: Mapa de memoria para configuración del acelerador ChSymSim: generador de rampa.

de habilitación serán 0b111111111. Por otra parte, todas las unidades *Address Encoder* son utilizadas en ambos modos, requiriendo de máscaras "*addr_enc_row_mask*" y "*addr_enc_col_mask*" con valores 0b111111111, con algunas excepciones en casos con *kernels* de tamaños 4 a 6 que no utilizan las últimas unidades del arreglo de PEs, requiriendo valores como 0b000111111. En casos donde se opera con valores de *stride* mayores a 1, los PEs del acelerador correspondientes deben ser apagados para ahorrar energía, requiriendo máscaras similares a 0b101010101 y 0b001001001 para *stride* 2 y 3, respectivamente.

En la Tabla I.9 se presenta el mapa de memoria de las señales de control manual del acelerador ChSymSim. En estos casos, las señales de habilitación (control) que van al generador de rampa y al arreglo de PEs, afectadas además por las máscaras mostradas en la Tabla I.6, presentan una función de *clear* automático para que sean activadas solo una vez por ciclo de reloj. De esta forma es posible controlar y evaluar cada paso del procesamiento del circuito incluso con sistemas lentos que requieren de varios ciclos de reloj para leer datos luego de escribir dichos registros de control.

El mapa de memoria de los registros locales para el almacenamiento local de parámetros, ubicado en la dirección base 0x10010000, se encuentra en la Tabla I.14. En esta, se muestran tanto la dirección inicial como la final de cada *feature* por parámetro en cuestión, contando con un solo *feature* para variables como SE y

Nombre	Dirección	Bits	Modo	Descripción
comp_row_mask	0x028	10:0	$^{R}/_{W}$	Máscara de comparadores (por fila) para habilitar/deshabilitar comparaciones según el tamaño del bloque de entradas necesarias para el cómputo
comp_col_mask	0x02A	10:0	$^{R}/_{W}$	Máscara de comparadores (por columna) para habilitar/deshabilitar comparaciones según el tamaño del bloque de entradas necesarias para el cómputo
out_feat_mask	0x02C	7:0	$^{R}/_{W}$	Máscara para habilitar/deshabilitar bloques *Mux. Accumulator* por *feature* de salida
addr_enc_row_mask	0x030	8:0	$^{R}/_{W}$	Máscara (por fila) para habilitar/deshabilitar los bloques de *Address Encoder* en el arreglo de PEs, según modo de *kernel* (filtrado espacial o FC) y *stride*
addr_enc_col_mask	0x032	8:0	$^{R}/_{W}$	Máscara (por columna) para habilitar/-deshabilitar los bloques de *Address Encoder* en el arreglo de PEs, según modo de *kernel* (filtrado espacial o FC) y *stride*
mux_accu_row_mask	0x034	8:0	$^{R}/_{W}$	Máscara (por fila) para habilitar/deshabilitar los bloques completos de *Mux. Accumulator* en el arreglo de PEs, según modo de *kernel* (filtrado espacial *small*, *big*, o FC) y *stride*
mux_accu_col_mask	0x036	8:0	$^{R}/_{W}$	Máscara (por columna) para habilitar/deshabilitar los bloques completos de *Mux. Accumulator* en el arreglo de PEs, según modo de *kernel* (filtrado espacial *small*, *big*, o FC) y *stride*

Tabla I.6: Mapa de memoria para configuración del acelerador ChSymSim: máscaras de habilitación de comparadores y bloques en PEs.

Nombre	Dirección	Bits	Modo	Descripción
sat_neg_val	0x038	7:0	$^R/_W$	Valor mínimo de saturación para salidas escaladas (y post ReLU)
sat_pos_val	0x038	15:8	$^R/_W$	Valor máximo de saturación para salidas escaladas (y post ReLU)
scale_shift_down	0x03A	4:0	$^R/_W$	Factor de escala (*bit shift*) para función de escalado
relu_leak_shift	0x03A	12:8	$^R/_W$	Factor de escala (división con *bit shift*) para función *leak* en activación PReLU
scale_stop_row	0x03C	3:0	$^R/_W$	Valor final del puntero a filas de salida para escalar
scale_stop_col	0x03C	11:8	$^R/_W$	Valor final del puntero a columnas de salida para escalar
scale_is_trunc	0x03E	0	$^R/_W$	Selección entre truncado o redondeo de salidas escaladas: ▪ 0 : operación de redondeo ▪ 1 : operación de truncado
scale_auto_start_en	0x03E	1	$^R/_W$	Habilitación para escalado automático de salidas del arreglo de PEs
pad_in_val	0x040	7:0	$^R/_W$	Valor para *padding* de entradas (valor de *reset* de los registros de entrada)
pad_out_val	0x040	15:8	$^R/_W$	Valor para *padding* de salidas (conectado directamente al multiplexor de salidas)
padding_row	0x042	2:0	$^R/_W$	Cantidad de filas de salida añadidas como *padding* (no utilizado)
padding_col	0x042	10:8	$^R/_W$	Cantidad de columnas de salida añadidas como *padding*: valor de *shift* de Bytes en el bus de salida, que se llenan con "*pad_out_val*"
stride_row	0x048	1:0	$^R/_W$	Configuración de *stride* por filas.
stride_col	0x048	9:8	$^R/_W$	Configuración de *stride* por columnas

Tabla I.7: Mapa de memoria para configuración del acelerador ChSymSim: configuración de escalado de salidas, *padding* y *stride*.

Nombre	Dirección	Bits	Modo	Descripción
start_symsim	0x050 (*set*) 0x058 (*clr*)	0	$^{R}/_{W}$ $^{R}/$	Da inicio al procesamiento ChSymSim
start_scale	0x050 (*set*) 0x058 (*clr*)	1	$^{R}/_{W}$ $^{R}/$	Da inicio al escalado de las salidas de los PEs
bias_ready	0x050 (*set*) 0x058 (*clr*)	2	$^{R}/_{W}$ $^{R}/$	Indica que los *bias* son válidos
se_ready	0x050 (*set*) 0x058 (*clr*)	3	$^{R}/_{W}$ $^{R}/$	Indica que los SE son válidos
w_ready	0x050 (*set*) 0x058 (*clr*)	4	$^{R}/_{W}$ $^{R}/$	Indica que los pesos simétricos son válidos
in_ready	0x050 (*set*) 0x058 (*clr*)	5	$^{R}/_{W}$ $^{R}/$	Indica que las entradas son válidas
symsim_ready	0x050 (*set*) 0x058 (*clr*)	6	$^{R}/$ $^{R}/$	Indica que finalizó el procesamiento de la función ChSymSim
scale_ready	0x050 (*set*) 0x058 (*clr*)	7	$^{R}/$ $^{R}/$	Indica que finalizó el escalado de las salidas
readout_ready	0x050 (*set*) 0x058 (*clr*)	8	$^{R}/_{W}$ $^{R}/$	Indica que las salidas (escaladas) fueron leídas
ramp_end_flag	0x050 (*set*) 0x058 (*clr*)	9	$^{R}/$ $^{R}/$	Indica que las rampa alcanzó su valor final
in_error_prec_flag	0x050 (*set*) 0x058 (*clr*)	10	$^{R}/$ $^{R}/$	Indica que hay un error en la precisión de las entradas
in_error_count_flag	0x050 (*set*) 0x058 (*clr*)	11	$^{R}/$ $^{R}/$	Indica que hay un error en la selección del canal en un Byte de entrada
w_error_prec_flag	0x050 (*set*) 0x058 (*clr*)	12	$^{R}/$ $^{R}/$	Indica que hay un error en la precisión de los pesos simétricos
w_error_count_flag	0x050 (*set*) 0x058 (*clr*)	13	$^{R}/$ $^{R}/$	Indica que hay un error en la selección del canal en un Byte de pesos

Tabla I.8: Mapa de memoria para configuración del acelerador ChSymSim: *flags*.

Nombre	Dirección	Bits	Modo	Descripción
manual_symsim	0x060 (*set*) 0x068 (*clr*)	0	$^{R}/_{W}$ $^{R}/_{W}$	Modo manual para control del operador ChSymSim (comparadores y PEs)
manual_ramp	0x060 (*set*) 0x068 (*clr*)	1	$^{R}/_{W}$ $^{R}/_{W}$	Modo manual para control de la rampa digital en el acelerador ChSymSim
manual_scale	0x060 (*set*) 0x068 (*clr*)	2	$^{R}/_{W}$ $^{R}/_{W}$	Modo manual para control de las unidades de escalado, redondeo y ReLU
manual_ramp_en	0x062 (*set*) Auto (*clr*)	0	/W -	Habilitación del contador en el generador de rampa digital
manual_symsim_comp_en	0x062 (*set*) Auto (*clr*)	1	/W -	Habilitación de los comparadores en el operador ChSymSim
manual_symsim_addr_enc_pwm_en	0x062 (*set*) Auto (*clr*)	2	/W -	Habilitación para la lectura de entradas PWM en los PEs (unidades *Address Encoder*)
manual_symsim_addr_enc_add_en	0x062 (*set*) Auto (*clr*)	3	/W -	Habilitación para la generación de dirección de pesos en los PEs (unidades *Address Encoder*)
manual_symsim_w_addr_en	0x062 (*set*) Auto (*clr*)	4	/W -	Habilitación para la lectura de dirección de pesos en los PEs (unidades *Mux. Accumulator*)
manual_symsim_w_sel_en	0x062 (*set*) Auto (*clr*)	5	/W -	Habilitación para la selección de pesos en los PEs (unidades *Mux. Accumulator*)
manual_symsim_accu_en	0x062 (*set*) Auto (*clr*)	6	/W -	Habilitación para la acumulación de pesos en los PEs (unidades *Mux. Accumulator*)
manual_scale_en	0x062 (*set*) Auto (*clr*)	8	/W -	Habilitación para las unidades de escalado, redondeo y ReLU
manual_scale_wr_out_en	0x062 (*set*) Auto (*clr*)	9	/W -	Habilitación para los registros de salida de las unidades de escalado, redondeo y ReLU
manual_se_scan_en	0x064 (*set*) Auto (*clr*)	0	/W -	Habilitación para cada *latch* interno que copia los SE en las memorias de parámetros
manual_w_scan_en	0x064 (*set*) Auto (*clr*)	1	/W -	Habilitación para cada *latch* interno que copia los pesos simétricos en las memorias de parámetros
manual_b_scan_en	0x064 (*set*) Auto (*clr*)	2	/W -	Habilitación para cada *latch* interno que copia los *bias* en las memorias de parámetros
manual_in_scan_en	0x064 (*set*) Auto (*clr*)	3	/W -	Habilitación para cada *latch* interno que copia las entradas al operador ChSymSim
manual_symsim_out_scan_en	0x064 (*set*) Auto (*clr*)	4	/W -	Habilitación para cada *latch* interno que copia las salidas (24 bits) del operador ChSymSim
manual_scale_out_scan_en	0x064 (*set*) Auto (*clr*)	5	/W -	Habilitación para cada *latch* interno que copia las salidas escaladas del operador ChSymSim

Tabla I.9: Mapa de memoria para configuración del acelerador ChSymSim: Modo Manual (*Debug*).

Nombre	Dirección	Bits	Modo	Descripción
manual_ramp_val	0x070	7:0	$^{R}/_{W}$	Valor de la rampa digital, definido por usuario para el modo de operación manual
manual_chan_cnt	0x072	9:0	$^{R}/_{W}$	Valor del contador de canales de entrada, definido por usuario para el modo de operación manual
manual_ch_per_in_byte_cnt	0x074	3:0	$^{R}/_{W}$	Valor del contador de canales por Byte de entrada, definido por usuario para el modo de operación manual
manual_in_bit_shift	0x074	7:4	$^{R}/_{W}$	Cantidad de bits a desplazar los Bytes de entrada, definido por usuario para el modo de operación manual
manual_ch_per_w_byte_cnt	0x074	11:8	$^{R}/_{W}$	Valor del contador de canales por Byte de peso simétrico, definido por usuario para el modo de operación manual
manual_w_bit_shift	0x074	15:12	$^{R}/_{W}$	Cantidad de bits a desplazar los Bytes de pesos simétricos, definido por usuario para el modo de operación manual
manual_scale_row_in_sel	0x076	3:0	$^{R}/_{W}$	Puntero de selección de filas para la lectura de entradas en las unidades de escalado (con redondeo) y ReLU, definido por usuario para el modo de operación manual
manual_scale_col_in_sel	0x076	7:4	$^{R}/_{W}$	Puntero de selección de columnas para la lectura de entradas en las unidades de escalado (con redondeo) y ReLU, definido por usuario para el modo de operación manual
manual_scale_row_out_sel	0x076	11:8	$^{R}/_{W}$	Puntero de selección de filas para la escritura de salidas en las unidades de escalado (con redondeo) y ReLU, definido por usuario para el modo de operación manual
manual_scale_col_out_sel	0x076	15:12	$^{R}/_{W}$	Puntero de selección de columnas para la escritura de salidas en las unidades de escalado (con redondeo) y ReLU, definido por usuario para el modo de operación manual

Tabla I.10: Mapa de memoria para configuración del acelerador ChSymSim: Modo Manual (*Debug*).

Nombre	Dirección	Bits	Modo	Descripción
ctrl_ramp_val	0x078	7:0	R/	Valor de la rampa digital, producido por el generador de rampa
ctrl_chan_cnt	0x07A	9:0	R/	Valor del contador de canales de entrada, producido por el controlador del operador ChSymSim
ctrl_ch_per_in_byte_cnt	0x07C	3:0	R/	Valor del contador de canales por Byte de entrada, producido por el controlador del operador ChSymSim
ctrl_in_bit_shift	0x07C	7:4	R/	Cantidad de bits a desplazar los Bytes de entrada, producido por el controlador del operador ChSymSim
ctrl_ch_per_w_byte_cnt	0x07C	11:8	R/	Valor del contador de canales por Byte de peso simétrico, producido por el controlador del operador ChSymSim
ctrl_w_bit_shift	0x07C	15:12	R/	Cantidad de bits a desplazar los Bytes de pesos simétricos, producido por el controlador del operador ChSymSim
ctrl_scale_row_in_sel	0x07E	3:0	R/	Puntero de selección de filas para la lectura de entradas a escalar, producido por el controlador de las unidades de escalado, redondeo y ReLU
ctrl_scale_col_in_sel	0x07E	7:4	R/	Puntero de selección de columnas para la lectura de entradas a escalar, producido por el controlador de las unidades de escalado, redondeo y ReLU
ctrl_scale_row_out_sel	0x07E	11:8	R/	Puntero de selección de filas para la escritura de salidas escaladas, producido por el controlador de las unidades de escalado, redondeo y ReLU
ctrl_scale_col_out_sel	0x07E	15:12	R/	Puntero de selección de columnas para la escritura de salidas escaladas, producido por el controlador de las unidades de escalado, redondeo y ReLU

Tabla I.11: Mapa de memoria para configuración del acelerador ChSymSim: Señales de control (*Debug*).

Nombre	Dirección	Bits	Modo	Descripción
bias_irq_en	0x080 (*set*) 0x088 (*clr*)	0	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación de la interrupción al μP para la escritura de *bias*
se_irq_en	0x080 (*set*) 0x088 (*clr*)	1	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación de la interrupción al μP para la escritura de SE (por canal de entrada)
weight_irq_en	0x080 (*set*) 0x088 (*clr*)	2	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación de la interrupción al μP para la escritura de pesos simétricos (por canal de entrada)
inputs_irq_en	0x080 (*set*) 0x088 (*clr*)	3	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación de la interrupción al μP para la escritura de las entradas (canal) a procesar
readout_irq_en	0x080 (*set*) 0x088 (*clr*)	4	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación de la interrupción al μP para realizar la lectura de los resultados escalados
scale_irq_en	0x080 (*set*) 0x088 (*clr*)	5	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación de la interrupción al μP para realizar escalado de las salidas de los PEs
bias_irq	0x080 (*set*) 0x088 (*clr*)	8	$^{R}/$ $^{R}/_{W}$	Registro de interrupción para la escritura de *bias*
se_irq	0x080 (*set*) 0x088 (*clr*)	9	$^{R}/$ $^{R}/_{W}$	Registro de interrupción para la escritura de SE (por canal de entrada)
weight_irq	0x080 (*set*) 0x088 (*clr*)	10	$^{R}/$ $^{R}/_{W}$	Registro de interrupción para la escritura de pesos simétricos (por canal de entrada)
inputs_irq	0x080 (*set*) 0x088 (*clr*)	11	$^{R}/$ $^{R}/_{W}$	Registro de interrupción para la escritura de las entradas (canal) a procesar
readout_irq	0x080 (*set*) 0x088 (*clr*)	12	$^{R}/$ $^{R}/_{W}$	Registro de interrupción para la lectura de los resultados escalados
scale_irq	0x080 (*set*) 0x088 (*clr*)	13	$^{R}/$ $^{R}/_{W}$	Registro de interrupción para el escalado de las salidas de los PEs

Tabla I.12: Mapa de memoria para configuración del acelerador ChSymSim: Interrupciones para transferencia de datos con el μP.

Nombre	Dirección	Bits	Modo	Descripción
bias_req_en	0x082 (*set*) 0x08A (*clr*)	0	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación para la señal de *request* de *bias*
se_req_en	0x082 (*set*) 0x08A (*clr*)	1	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación para la señal de *request* de SE (por canal de entrada)
weight_req_en	0x082 (*set*) 0x08A (*clr*)	2	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación para la señal de *request* de pesos simétricos (por canal de entrada)
inputs_req_en	0x082 (*set*) 0x08A (*clr*)	3	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación para la señal de *request* del canal de entrada)
readout_req_en	0x082 (*set*) 0x08A (*clr*)	4	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación para la señal de *request* de la lectura de salidas (escaladas)
extra_req_en	0x082 (*set*) 0x08A (*clr*)	5	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación para la señal de *request* adicional (según registro "*extra_req_sel*")
ext_bias_done_en	0x082 (*set*) 0x08A (*clr*)	8	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación para la señal *done* de escritura de *bias*, proveniente de los DMAs
ext_se_done_en	0x082 (*set*) 0x08A (*clr*)	9	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación para la señal *done* de escritura de SE, proveniente de los DMAs
ext_weights_done_en	0x082 (*set*) 0x08A (*clr*)	10	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación para la señal *done* de escritura de pesos simétricos, proveniente de los DMAs
ext_inputs_done_en	0x082 (*set*) 0x08A (*clr*)	11	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación para la señal *done* de escritura de entradas, proveniente de los DMAs
ext_readout_done_en	0x082 (*set*) 0x08A (*clr*)	12	$^{R}/_{W}$ $^{R}/_{W}$	Habilitación para la señal *done* de lectura de resultados, proveniente de los DMAs

Tabla I.13: Mapa de memoria para configuración del acelerador ChSymSim: Habilitaciones de señales *request* y *done* para transferencia de datos con los DMAs.

SM (con 11 Bytes para cada uno de estos parámetros), mientras que los registros de pesos simétricos *W* (82 Bytes por *feature*) y de *bias B* (2 Bytes por *feature*) contienen hasta 8 *features*.

		Feat. 1	*Feat.* 2	*Feat.* 3	*Feat.* 4	*Feat.* 5	*Feat.* 6	*Feat.* 7	*Feat.* 8	Bytes por *Feat.*
SM	0x000	0x000	-	-	-	-	-	-	-	11
	0x00A	0x00A	-	-	-	-	-	-	-	
SE	0x400	0x400	-	-	-	-	-	-	-	11
	0x40A	0x40A	-	-	-	-	-	-	-	
W	0x800	0x800	0x858	0x8B0	0x908	0x960	0x9B8	0xA10	0xA68	82
	0xAB9	0x851	0x8A9	0x901	0x959	0x9B1	0xA09	0xA61	0xAB9	
B	0xC00	0xC00	0xC02	0xC04	0xC06	0xC08	0xC0A	0xC0C	0xC0E	2
	0xC0F	0xC01	0xC03	0xC05	0xC07	0xC09	0xC0B	0xC0D	0xC0F	

Tabla I.14: Mapa de memoria de parámetros para la función ChSymSim: Máscara de Forma de *kernel* (*SM*), Elementos Estructurantes (*SE*), pesos simétricos (*W*) y *bias* (*B*).

En la Tabla I.15 se presentan las direcciones del arreglo de entradas del acelerador ChSymSim (por Byte), accedidas desde la dirección base `0x10020000` en el sistema AHB.

	Col. 1	Col. 2	Col. 3	Col. 4	Col. 5	Col. 6	Col. 7	Col. 8	Col. 9	Col. 10	Col. 11
Fila 1	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A
Fila 2	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1A
Fila 3	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27	0x28	0x29	0x2A
Fila 4	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37	0x38	0x39	0x3A
Fila 5	0x40	0x41	0x42	0x43	0x44	0x45	0x46	0x47	0x48	0x49	0x4A
Fila 6	0x50	0x51	0x52	0x53	0x54	0x55	0x56	0x57	0x58	0x59	0x5A
Fila 7	0x60	0x61	0x62	0x63	0x64	0x65	0x66	0x67	0x68	0x69	0x6A
Fila 8	0x70	0x71	0x72	0x73	0x74	0x75	0x76	0x77	0x78	0x79	0x7A
Fila 9	0x80	0x81	0x82	0x83	0x84	0x85	0x86	0x87	0x88	0x89	0x8A
Fila 10	0x90	0x91	0x92	0x93	0x94	0x95	0x96	0x97	0x98	0x99	0x9A
Fila 11	0xA0	0xA1	0xA2	0xA3	0xA4	0xA5	0xA6	0xA7	0xA8	0xA9	0xAA

Tabla I.15: Mapa de memoria completo del arreglo de entradas al acelerador ChSymSim.

Finalmente, la Tabla I.16 contiene las direcciones de salidas del procesador ChSymSim, que se acceden por medio de la dirección base `0x10030000` (AHB). Para la obtención de resultados escalados a 8 bits, incluyendo píxeles con *padding*, se realizan mediante la lectura directa del *bus* de salida, agrupando como máximo 8 columnas por transacción. En este caso, el uso de *padding* se traduce a un desplazamiento de los valores de salida hacia la derecha, rellenando con el valor de *padding*

las ubicaciones iniciales que quedan libres luego de este *shift* de datos. En el caso de leerse datos por fuera del arreglo de salidas, los Bytes correspondientes en el *bus* son rellenados con el valor de *padding*. Si se desea obtener el *bus* completo con valores de *padding*, lo que correspondería a las primeras o últimas filas de la imagen/tensor objetivo, el acelerador ChSymSim cuenta con una dirección de salida específica, ubicada en `0x0480`.

Por otra parte, por motivos de evaluación y depuración (*debug*), es posible acceder a los resultados parciales tales como las entradas codificadas en señales PWM, las direcciones de peso a seleccionar durante el procesamiento, y los resultados en precisión completa (24 bits). Para las salidas de los PEs (resultados en 24 bits) se cuenta con la capacidad de extraer hasta un máximo de 2 valores por transacción (una sola fila y dos columnas de datos), debido a que dichos valores son extendidos a 32 bits. En cambio, las entradas codificadas como PWM son leídas por fila, por lo que se cuenta con 11 *double-words* (uno por fila), donde en cada uno se disponen las 11 columnas en los bits menos significativos y el resto de los bits se rellena con 0s. Las direcciones de pesos generadas por las unidades *Address Encoder* son extendidas a 8 bits, por lo que pueden ser leídas de forma similar a los resultados escalados, rellenando con 0s para ubicaciones de columnas que excedan el tamaño del arreglo de PEs.

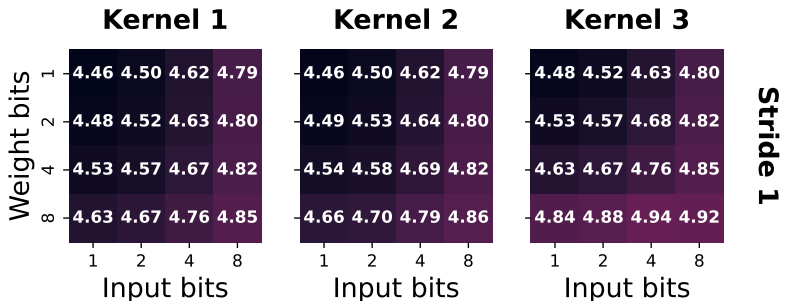
Salidas	Inicial	Final	Salto por Fila
Resultados escalados (*feature* 1)	0x0000	0x008F	0x0010
Resultados escalados (*feature* 2)	0x0090	0x011F	
Resultados escalados (*feature* 3)	0x0120	0x01AF	
Resultados escalados (*feature* 4)	0x01B0	0x023F	
Resultados escalados (*feature* 5)	0x0240	0x02CF	
Resultados escalados (*feature* 6)	0x02D0	0x035F	
Resultados escalados (*feature* 7)	0x0360	0x03EF	
Resultados escalados (*feature* 8)	0x03F0	0x047F	
Padding Fila (*bus* completo)	0x0480	0x0487	-
Entradas Codificadas (PWM)	0x1000	0x1057	0x0008
Dirección de Pesos	0x2000	0x208F	0x0010
Resultados en 24 bits (*feature* 1)	0x3000	0x3167	0x0028
Resultados en 24 bits (*feature* 2)	0x3168	0x32CF	
Resultados en 24 bits (*feature* 3)	0x32D0	0x3437	
Resultados en 24 bits (*feature* 4)	0x3438	0x359F	
Resultados en 24 bits (*feature* 5)	0x35A0	0x3707	
Resultados en 24 bits (*feature* 6)	0x3708	0x386F	
Resultados en 24 bits (*feature* 7)	0x3870	0x39D7	
Resultados en 24 bits (*feature* 8)	0x39D8	0x3B3F	

Tabla I.16: Mapa de direcciones (resumido) para las salidas del acelerador ChSym-Sim.

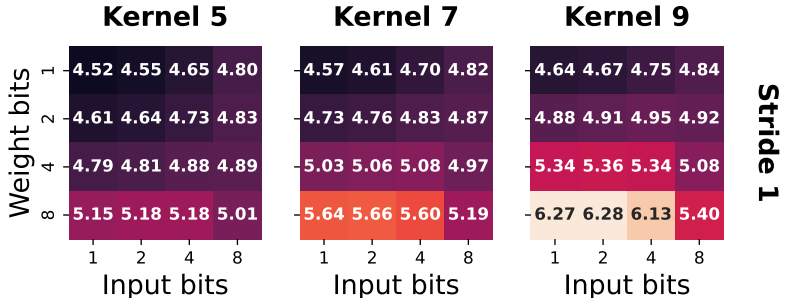
Apéndice J

DigineuronV3a: mediciones de consumo

En este apéndice se presentan las mediciones, tanto directas como indirectas, de los consumos del acelerador ChSymSim y transferencia de datos en el SoC DigineuronV3a.

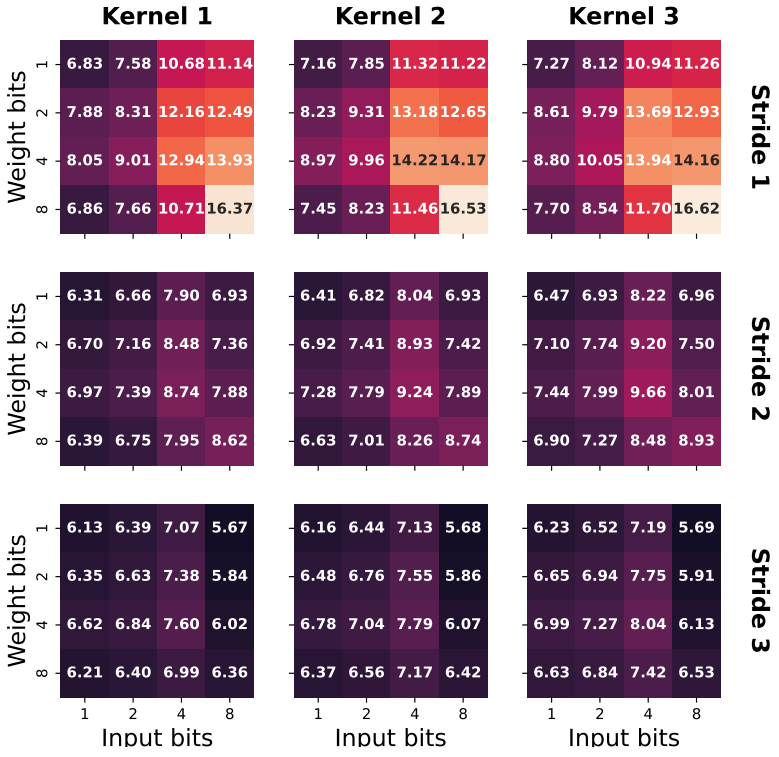


(a)

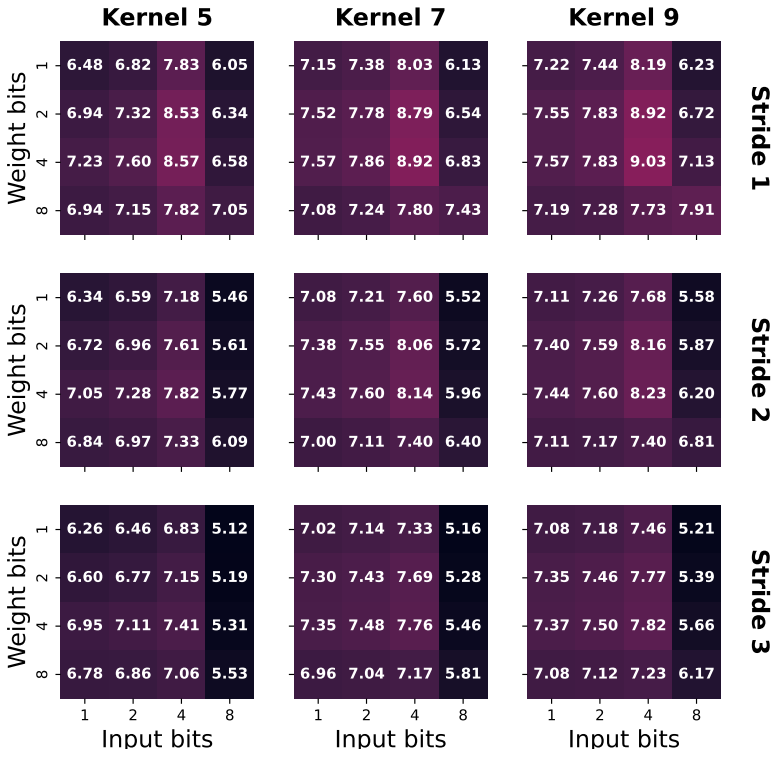


(b)

Figura J.1: Consumo de potencia en [mW] medido (1,2V@25MHz) del SoC DigineuronV3a, realizando solo transferencias de datos (*mu*P y DMAs), con diferentes configuraciones de ChSymSim: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

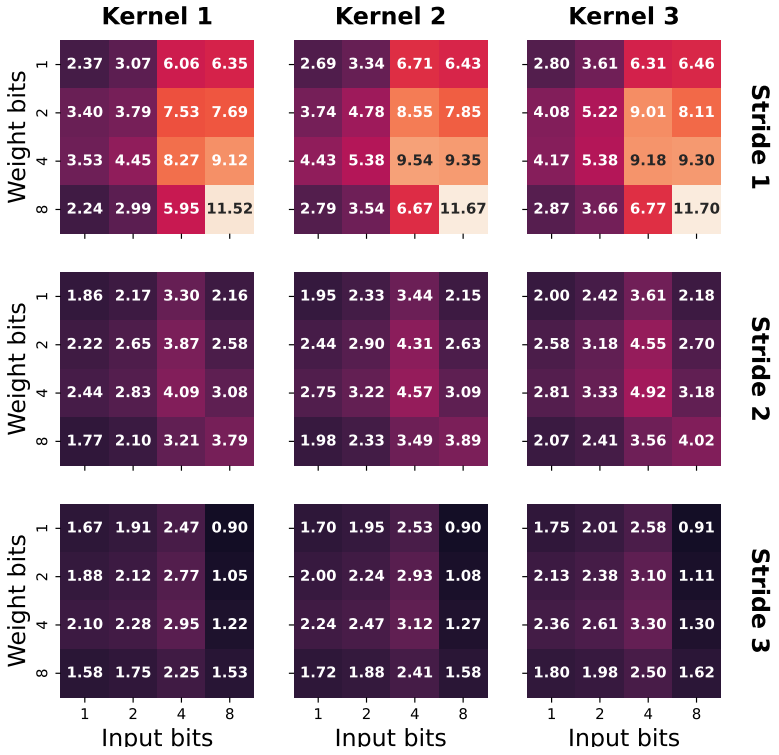


(a)

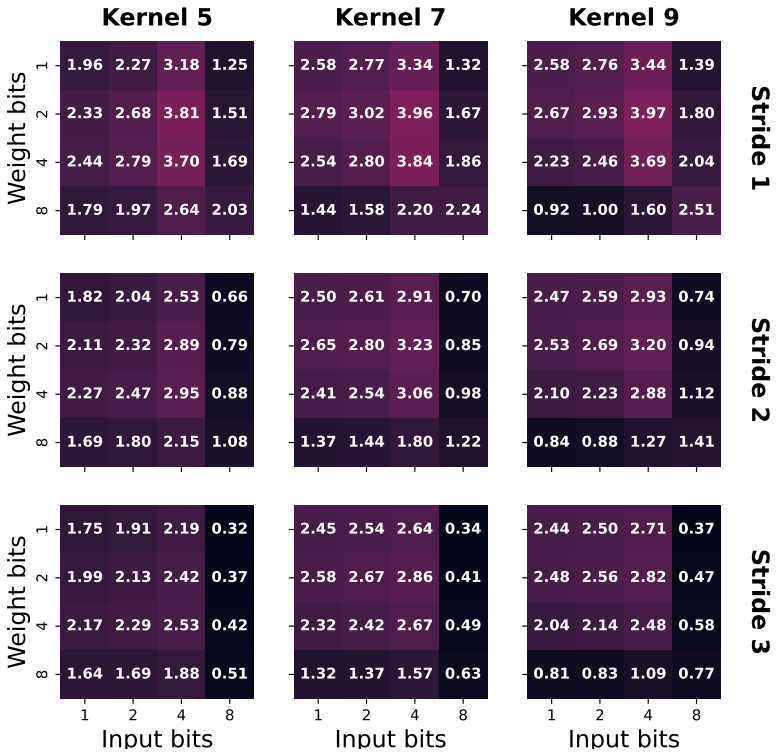


(b)

Figura J.2: Consumo de potencia en [mW] medido (1,2V@25MHz) del SoC DigineuronV3a, ejecutando diferentes configuraciones de ChSymSim: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

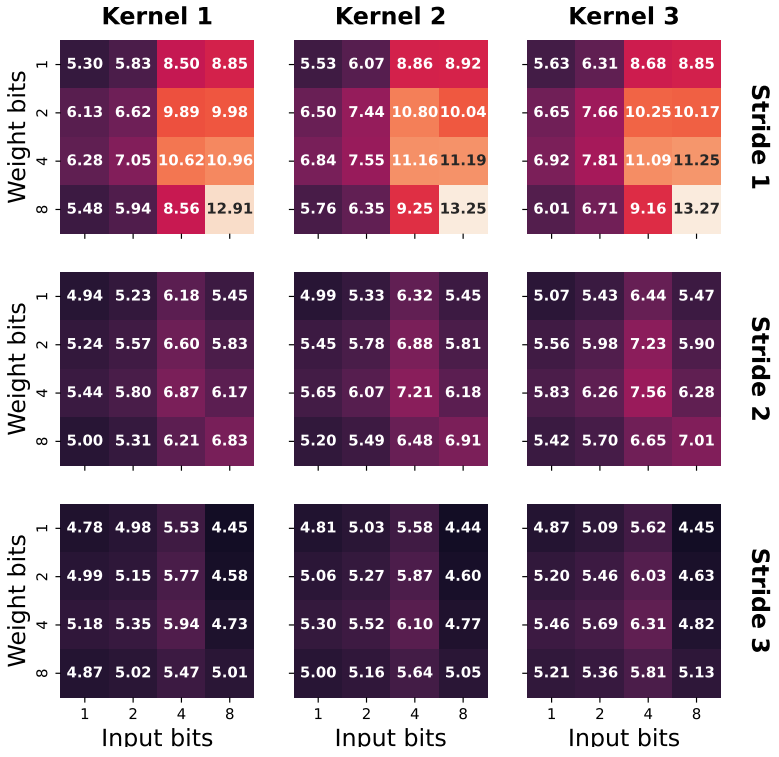


(a)

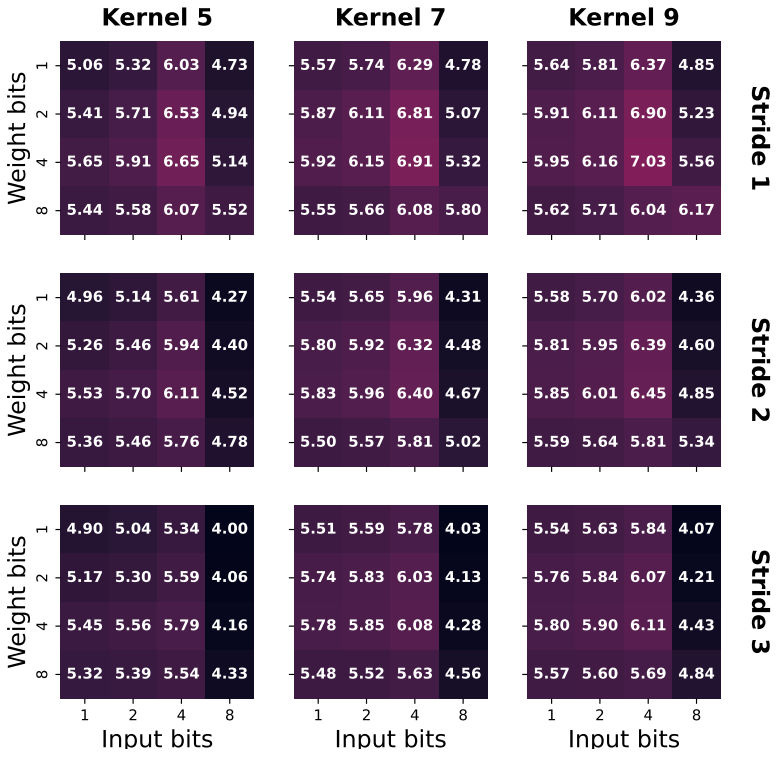


(b)

Figura J.3: Consumo de potencia en [mW] calculado (1,2V@25MHz) del acelerador ChSymSim en DigineuronV3a, ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

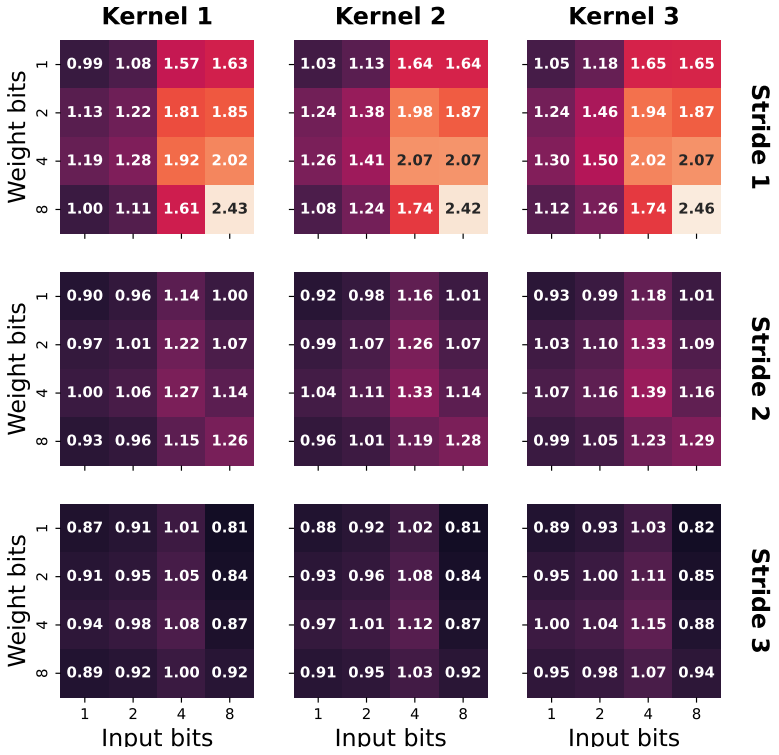


(a)

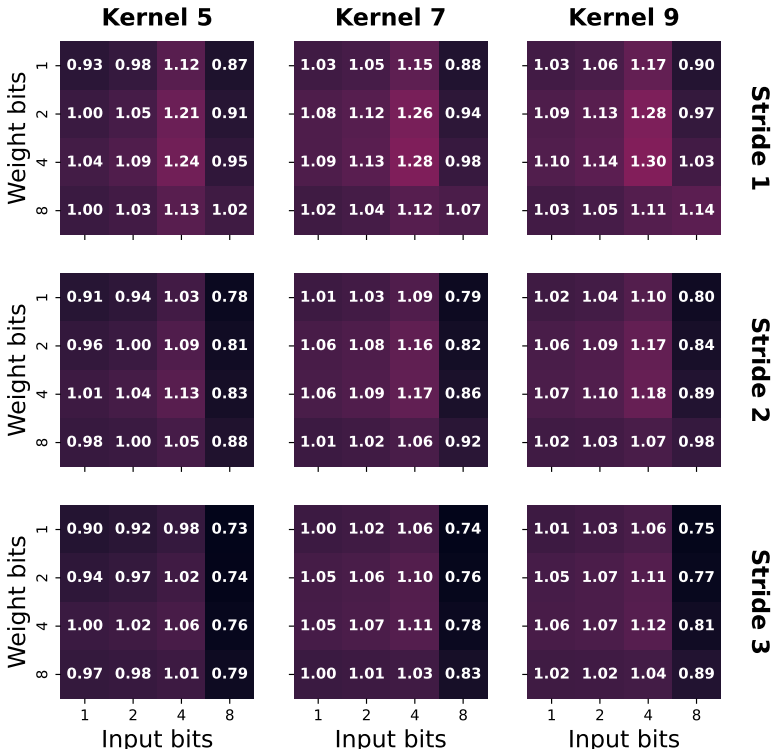


(b)

Figura J.4: Consumo de potencia en [mW] medido (1,07V@25MHz) del SoC DigineuronV3a, ejecutando diferentes configuraciones de ChSymSim: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

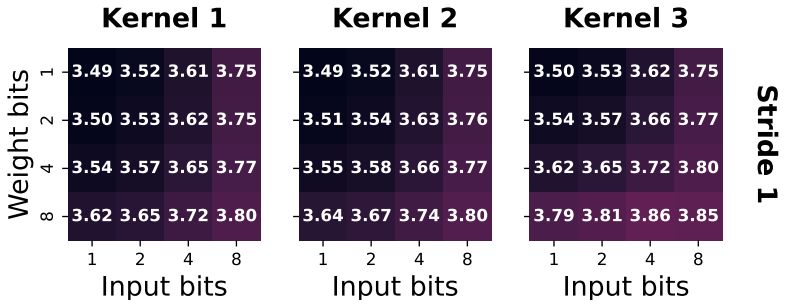


(a)

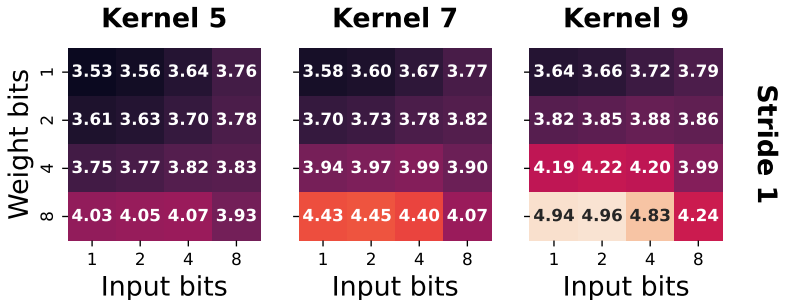


(b)

Figura J.5: Consumo de potencia en [mW] calculado (0,81V@8MHz) del SoC DigineuronV3a, ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

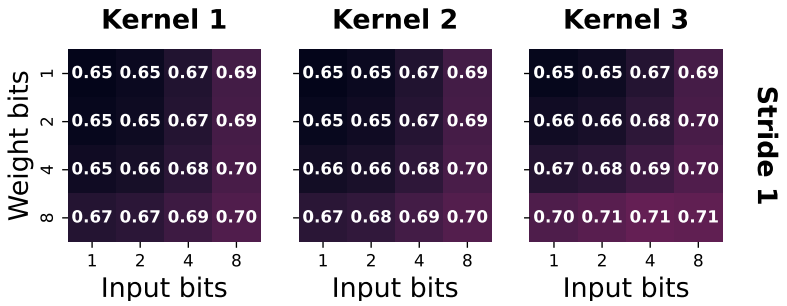


(a)

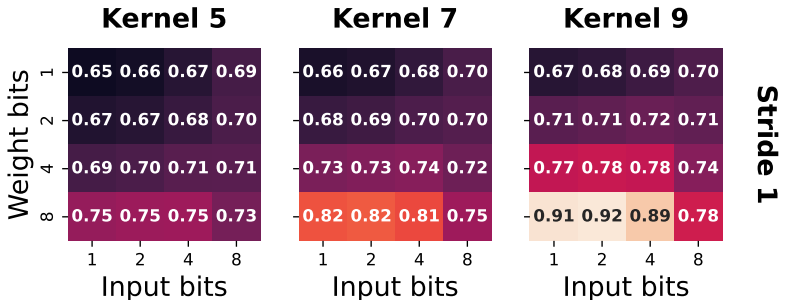


(b)

Figura J.6: Consumo de potencia en [mW] medido (1,07V@25MHz) del SoC DigineuronV3a, realizando solo transferencias de datos (*mu*P y DMAs), con diferentes configuraciones de ChSymSim: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

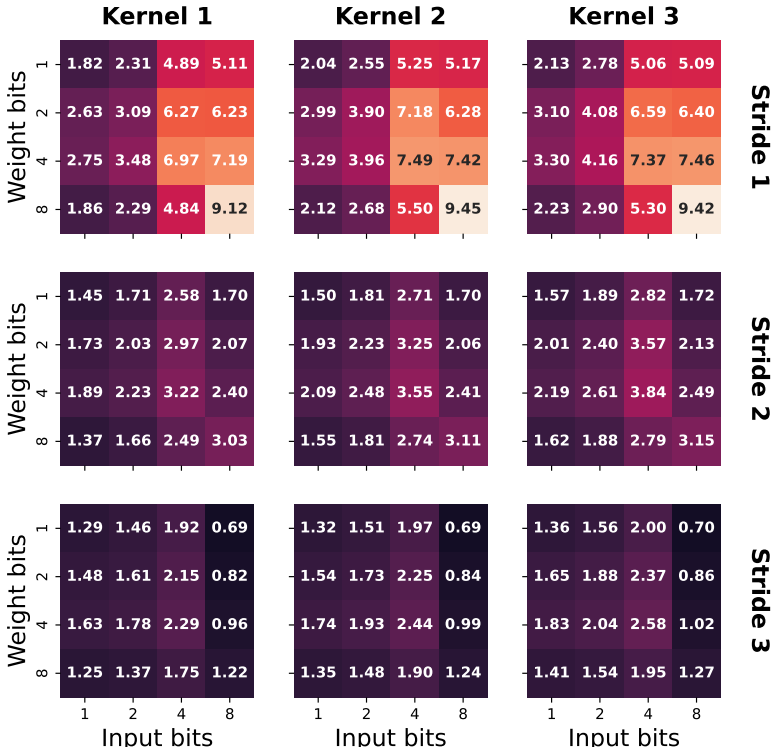


(a)

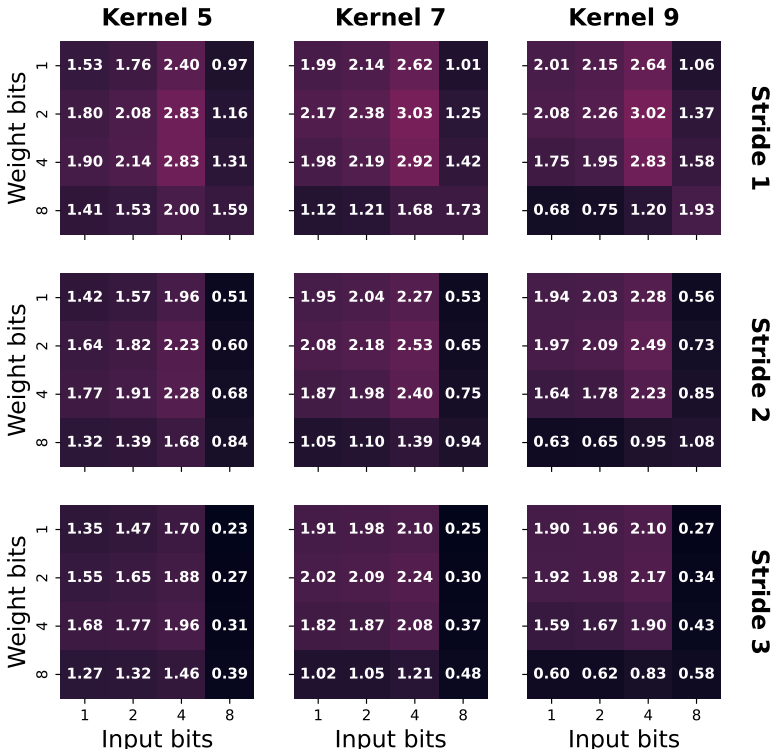


(b)

Figura J.7: Consumo de potencia en [mW] medido (0,81V@8MHz) del SoC DigineuronV3a, realizando solo transferencias de datos (*mu*P y DMAs), con diferentes configuraciones de ChSymSim: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

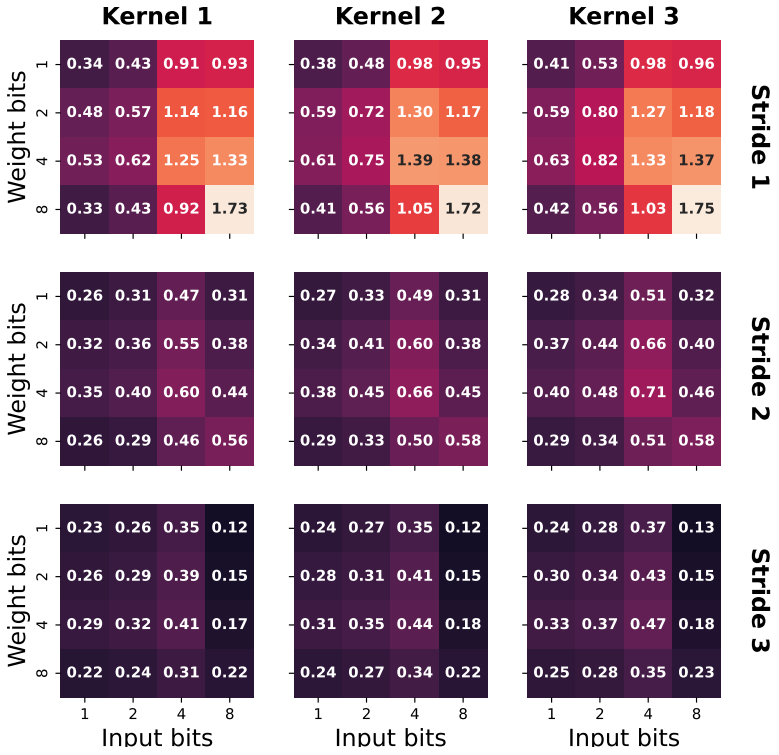


(a)

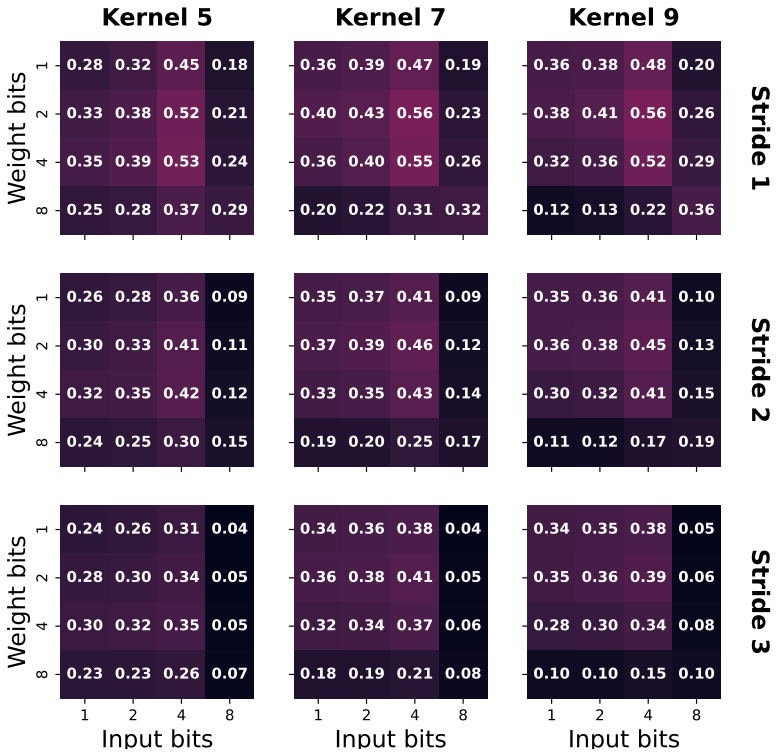


(b)

Figura J.8: Consumo de potencia en [mW] calculado (1,07V@25MHz) del acelerador ChSymSim en DigineuronV3a, ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

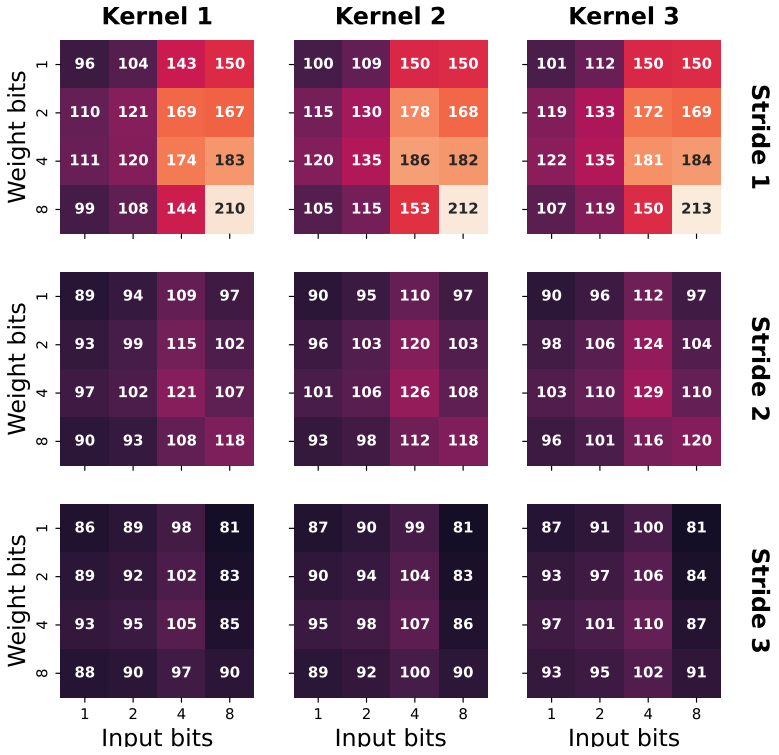


(a)

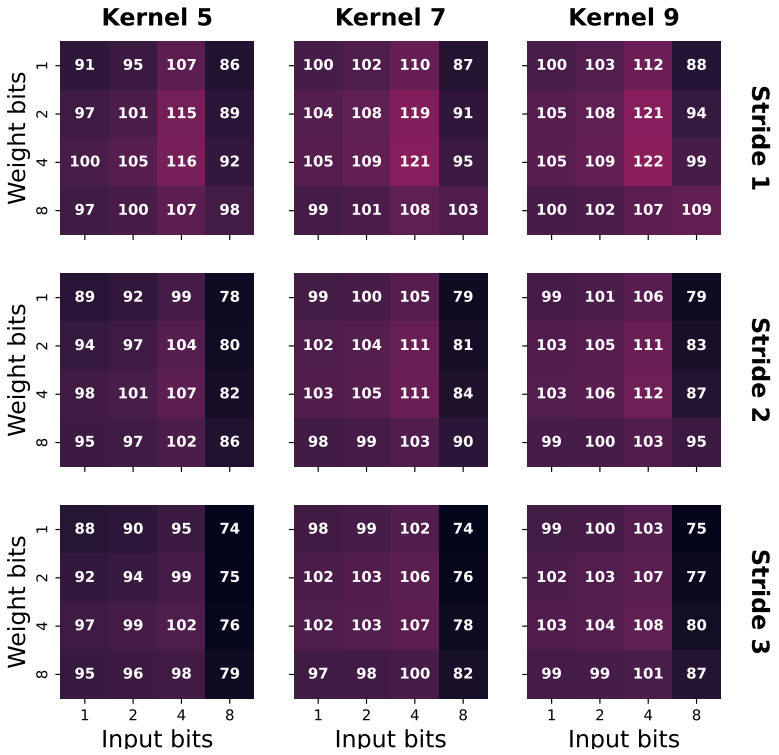


(b)

Figura J.9: Consumo de potencia en [mW] calculado (0,81V@8MHz) del acelerador ChSymSim en DigineuronV3a, ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

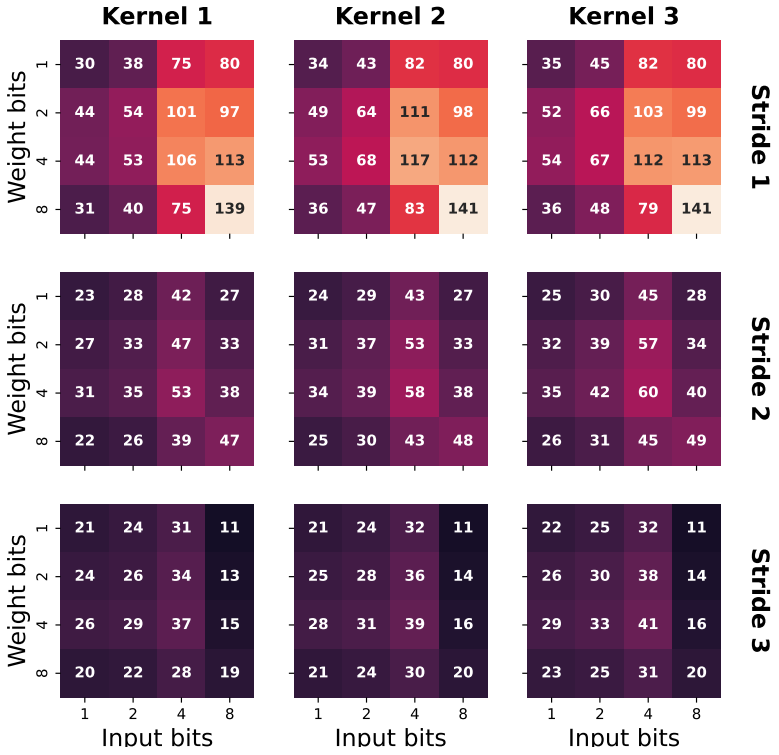


(a)

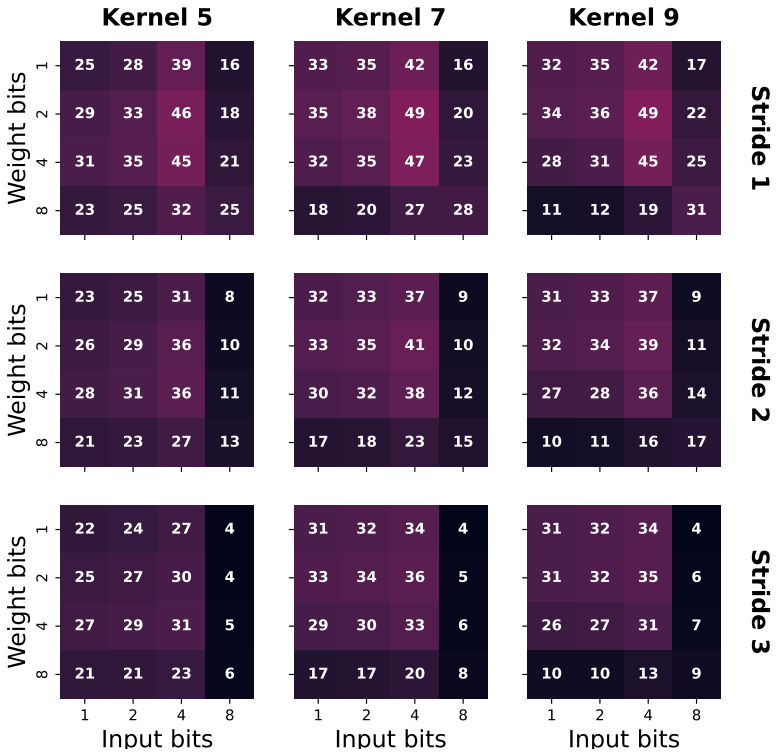


(b)

Figura J.10: Consumo de potencia en [mW] medido (2,1V@100MHz) del SoC DigineuronV3a, ejecutando diferentes configuraciones de ChSymSim: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

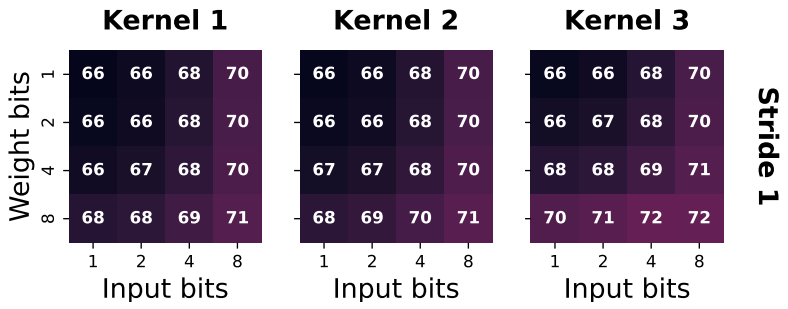


(a)

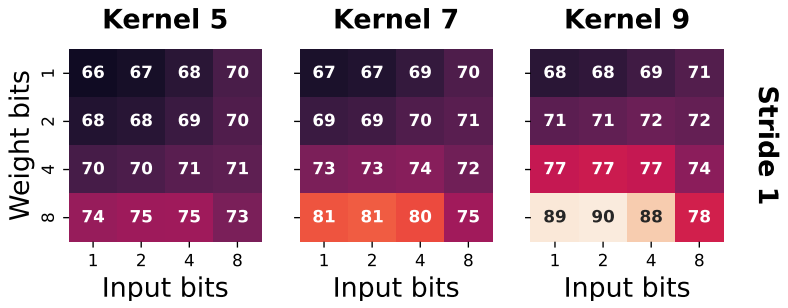


(b)

Figura J.11: Consumo de potencia en [mW] calculado (2,1V@100MHz) del acelerador ChSymSim en DigineuronV3a, ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).



(a)



(b)

Figura J.12: Consumo de potencia en [mW] medido (2,1V@100MHz) del SoC DigineuronV3a, realizando solo transferencias de datos (*mu*P y DMAs), con diferentes configuraciones de ChSymSim: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

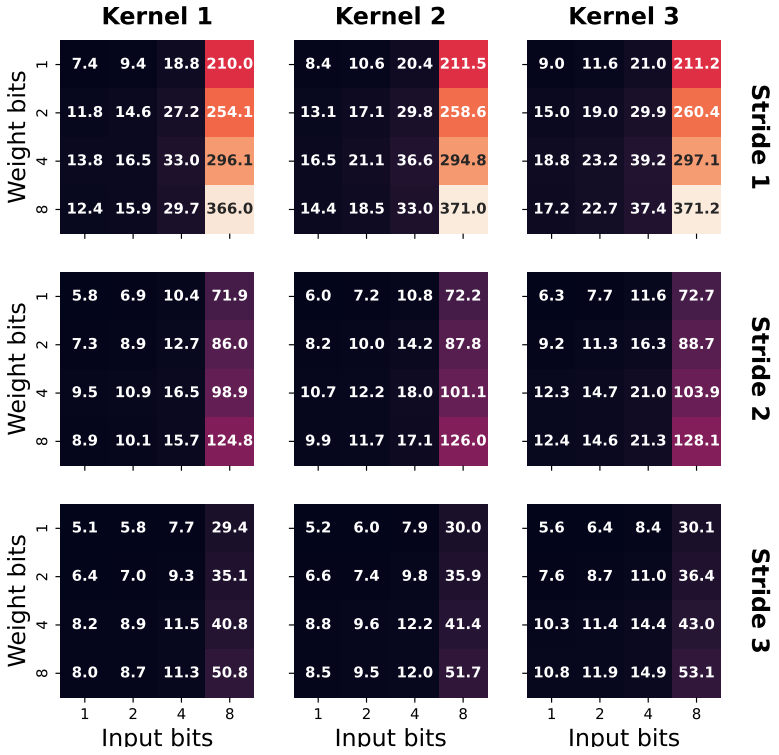
Apéndice K

DigineuronV3a: desempeño y eficiencia

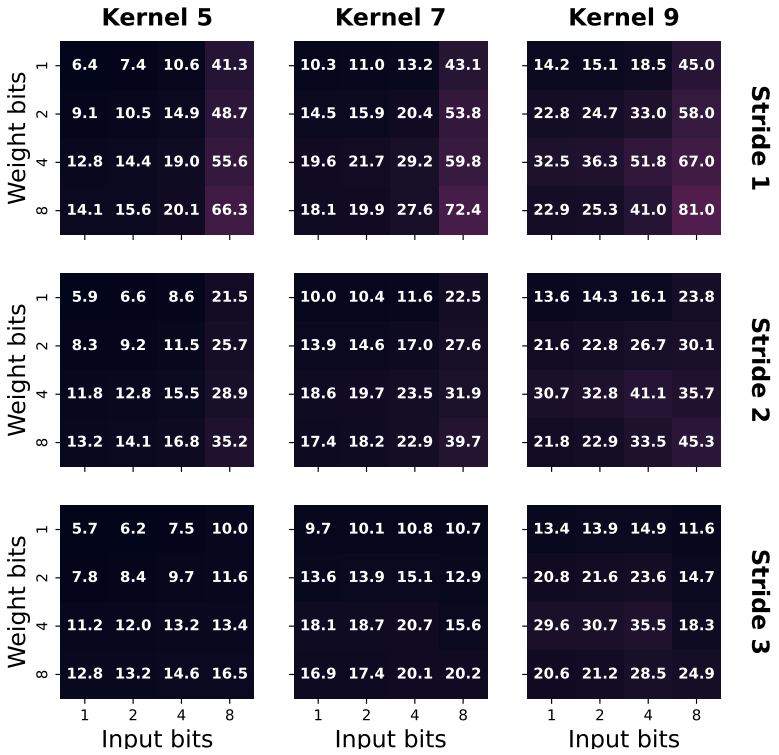
En este apéndice se presentan los valores calculados de energía, rendimiento (*throughput*) y eficiencia del acelerador ChSymSim integrado en el SoC DigineuronV3a, los cuales fueron hallados a partir de las mediciones de potencia del Apéndice J. Estos resultados se encuentran resumidos en la Tabla K.1, mientras que se detallan en las Figs. K.1-K.22, mostrando todos los casos analizados de tamaño de *kernel*, *stride*, precisión de entrada y pesos simétricos.

Métrica \ V_{DD}@f_{clk}	2,1V@100MHz		1,2V@25MHz		1,07V@25MHz		0,81V@8MHz	
	min	Max	min	Max	min	Max	min	Max
Energía [nJ]	6,44	371,19	2,06	123,09	1,61	99,38	0,91	57,38
Energía por OP [pJ]	0,58	46,30	0,19	14,72	0,14	11,84	0,08	6,77
Rendimiento [GOPS]	1,72	104,80	0,43	26,20	0,43	26,20	0,14	8,38
Rendimiento [MMACS]	30,80	6124,84	7,70	1531,21	7,70	1531,21	2,46	489,99
Eficiencia [TOPS/W]	0,02	1,73	0,07	5,31	0,08	6,90	0,15	12,12
Eficiencia [GMACS/W]	0,39	78,00	1,21	226,33	1,51	288,97	2,64	498,50

Tabla K.1: Medidas de energía, desempeño y eficiencia (resumidas) del acelerador ChSymSim en el SoC DigineuronV3a, considerando *stride* 1.

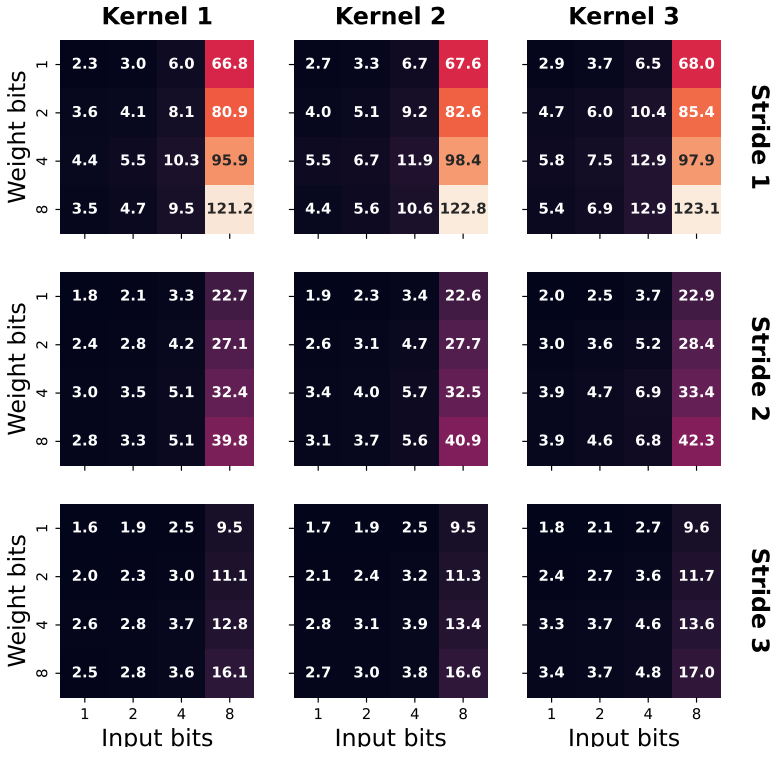


(a)

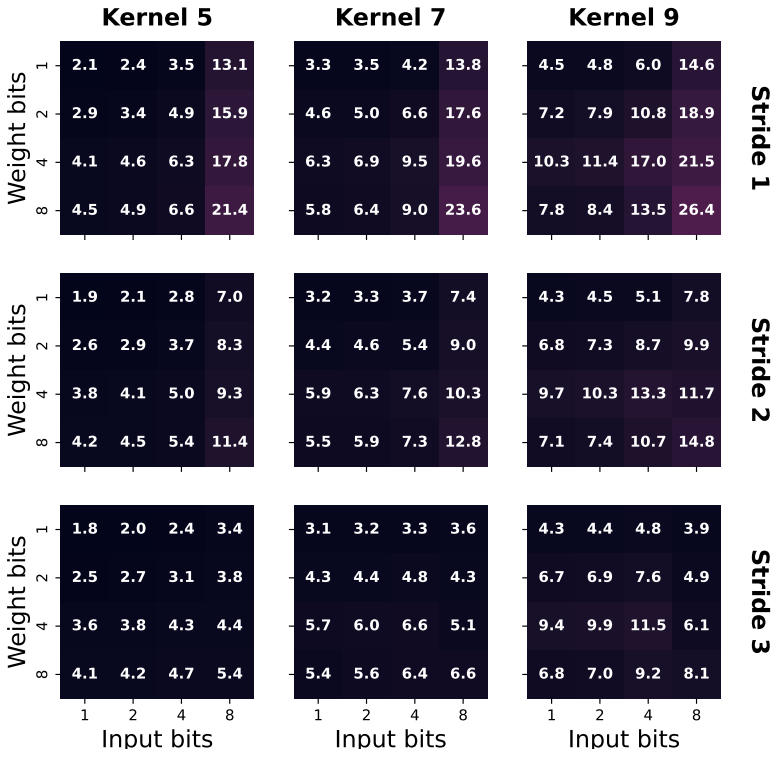


(b)

Figura K.1: Energía media en [nJ] del acelerador ChSymSim (2,1V@100MHz), calculada por ciclo de procesamiento y ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

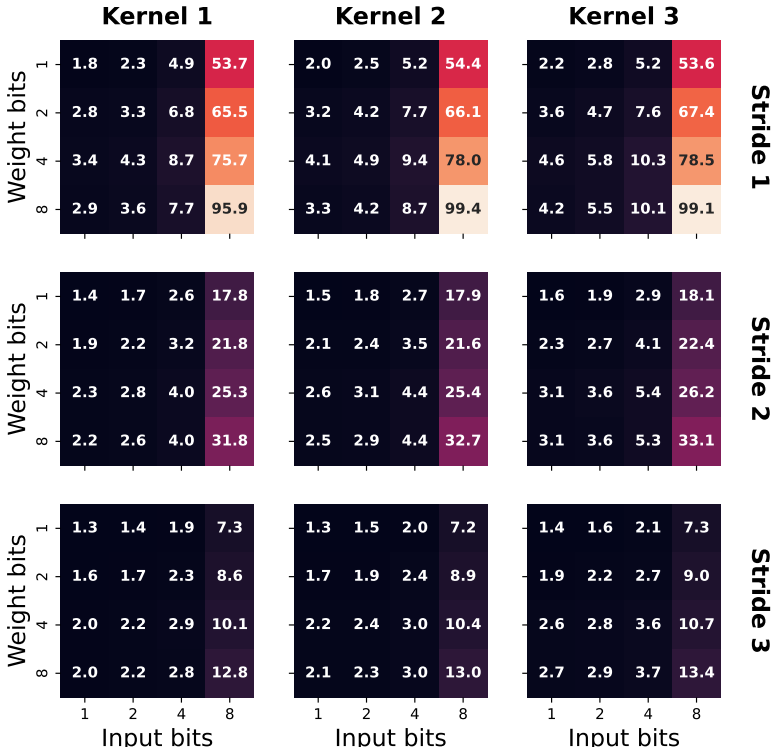


(a)

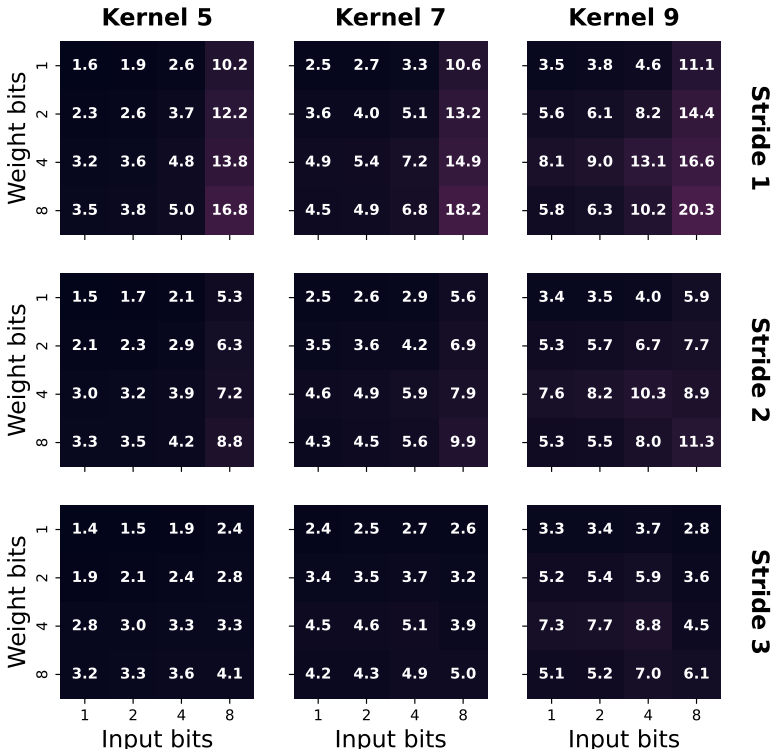


(b)

Figura K.2: Energía media en [nJ] del acelerador ChSymSim (1,2V@25MHz), calculada por ciclo de procesamiento y ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

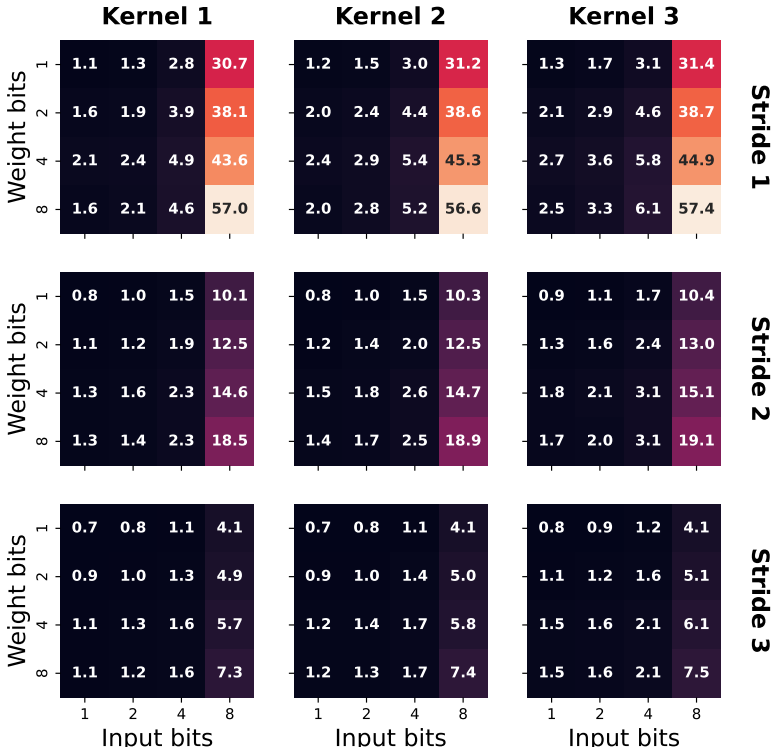


(a)

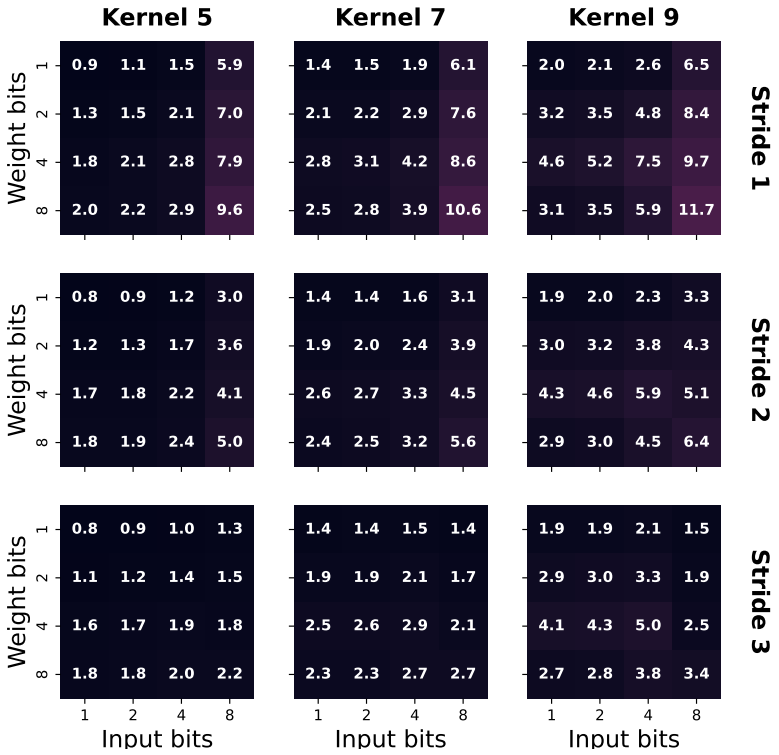


(b)

Figura K.3: Energía media en [nJ] del acelerador ChSymSim (1,07V@25MHz), calculada por ciclo de procesamiento y ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

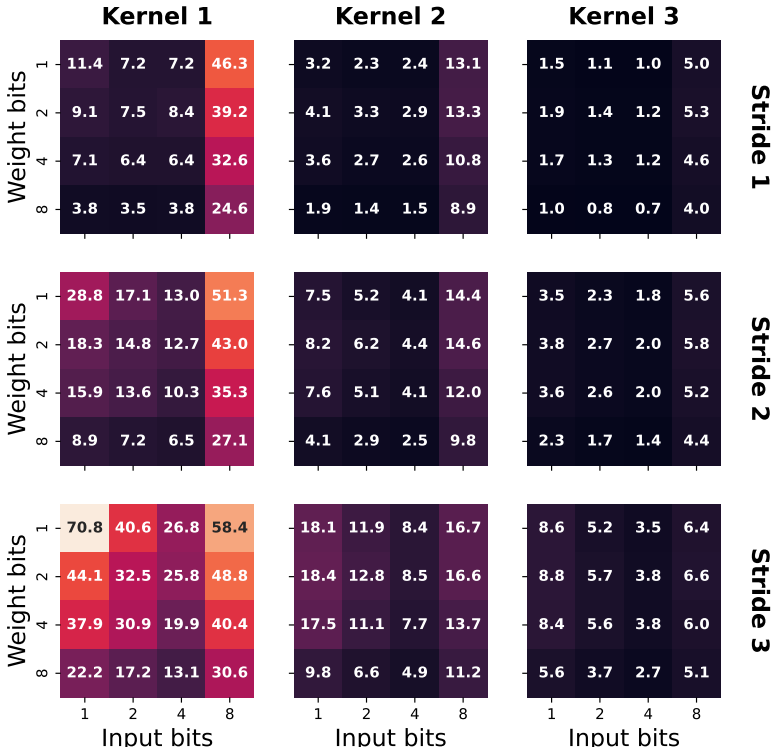


(a)

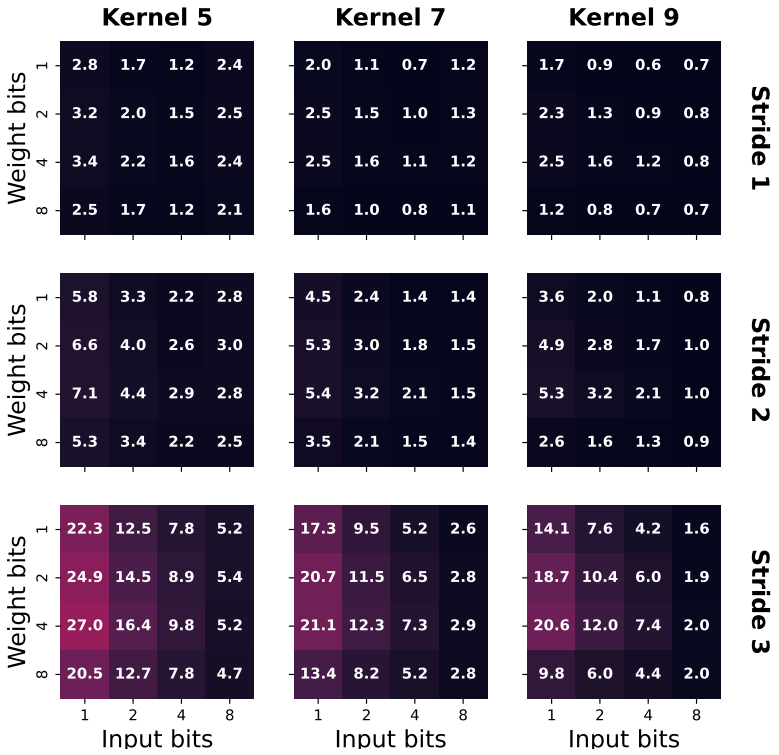


(b)

Figura K.4: Energía media en [nJ] del acelerador ChSymSim (0,81V@8MHz), calculada por ciclo de procesamiento y ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

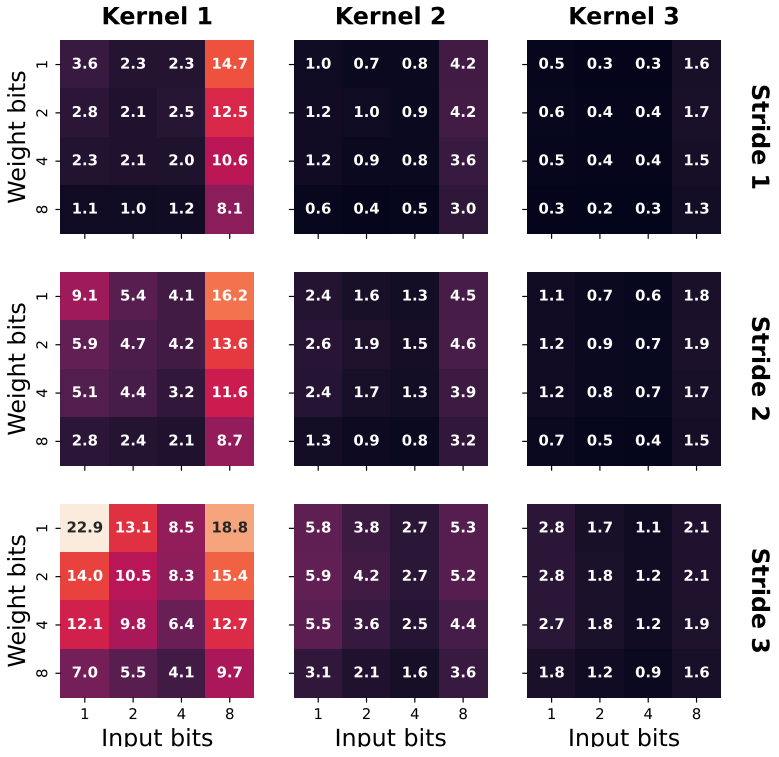


(a)

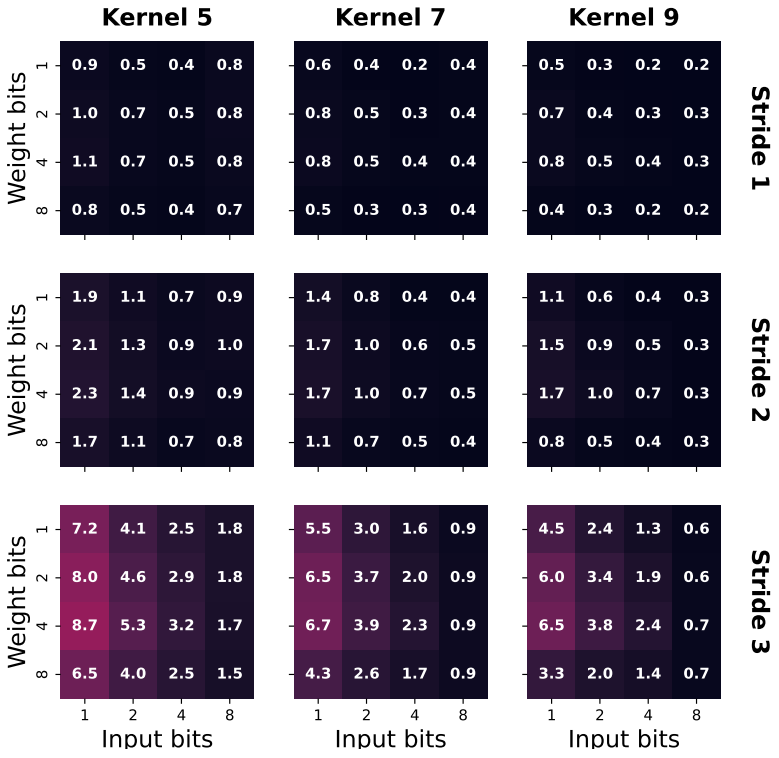


(b)

Figura K.5: Energía por operación en [pJ] del acelerador ChSymSim (2,1V@100MHz), con operaciones de 8 bits y ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

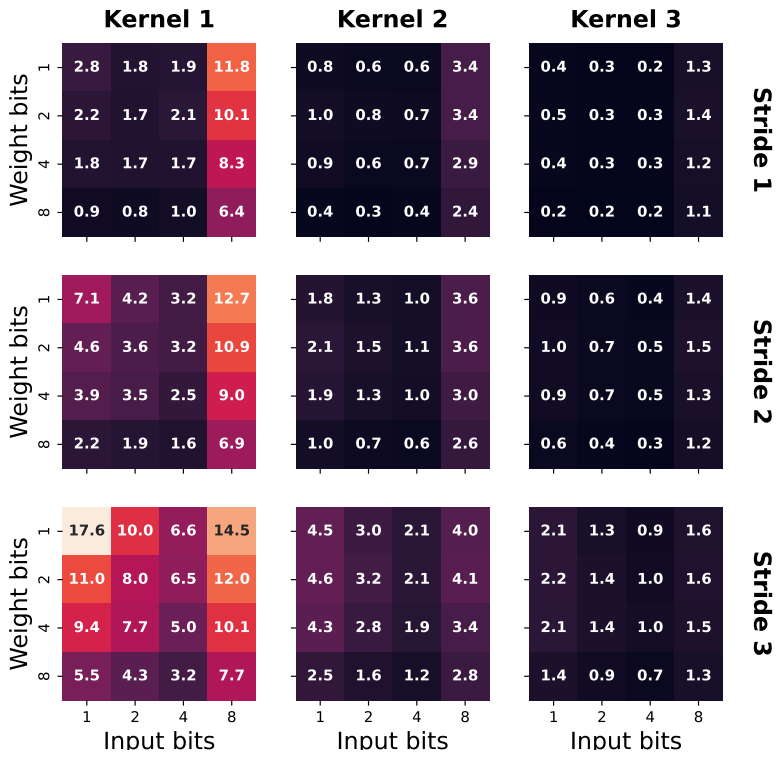


(a)

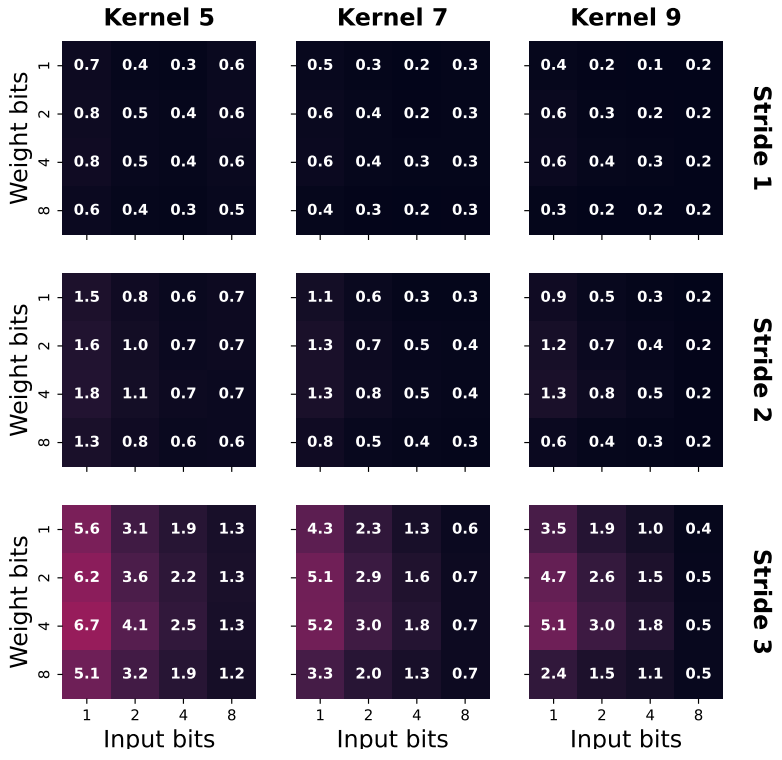


(b)

Figura K.6: Energía por operación en [pJ] del acelerador ChSymSim (1,2V@25MHz), con operaciones de 8 bits y ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

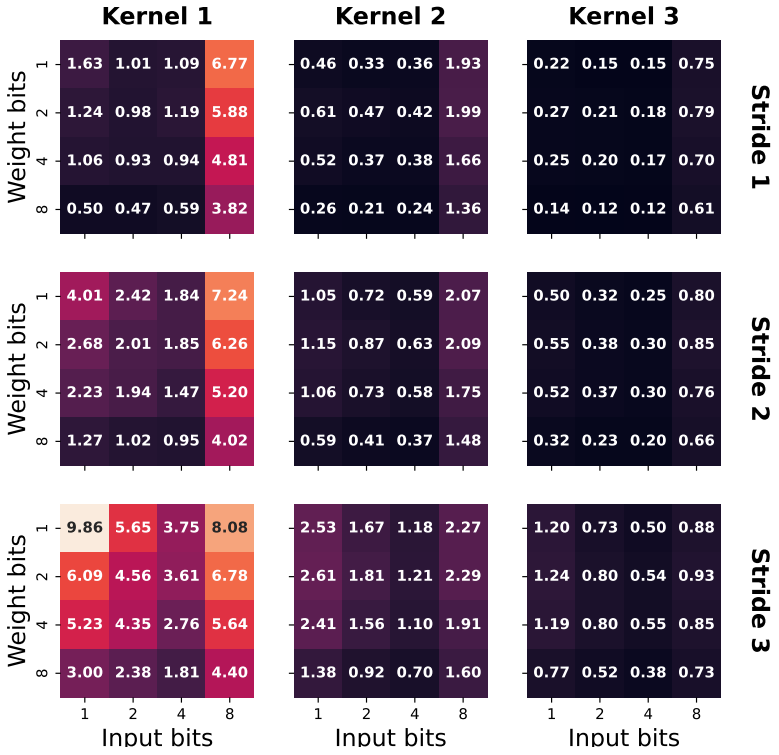


(a)

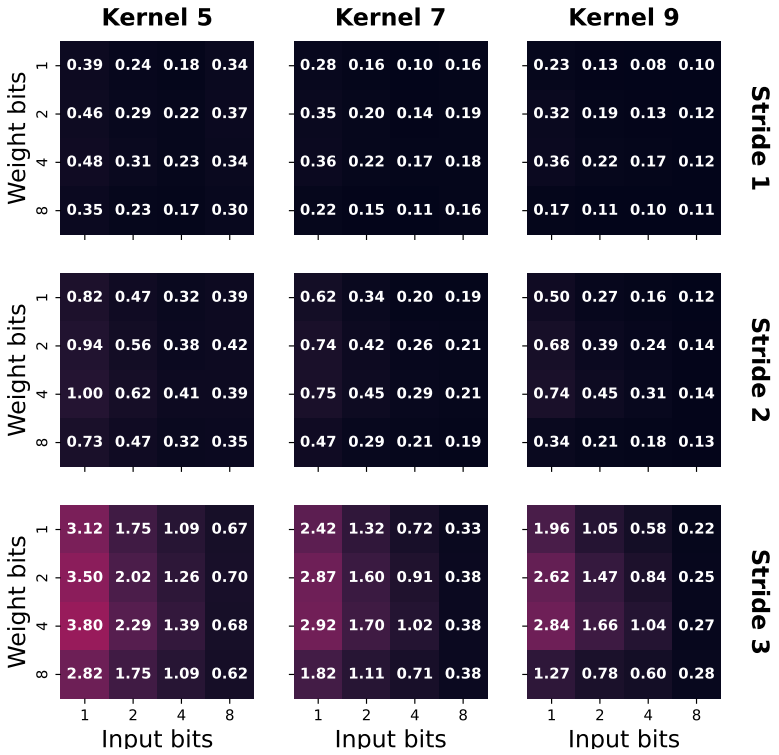


(b)

Figura K.7: Energía por operación en [pJ] del acelerador ChSymSim (1,07V@25MHz), con operaciones de 8 bits y ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

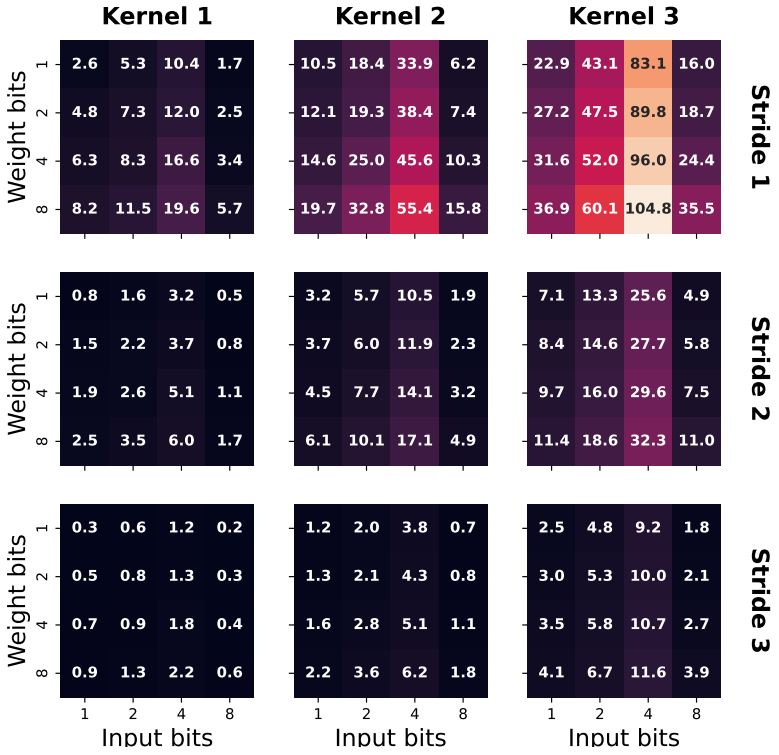


(a)

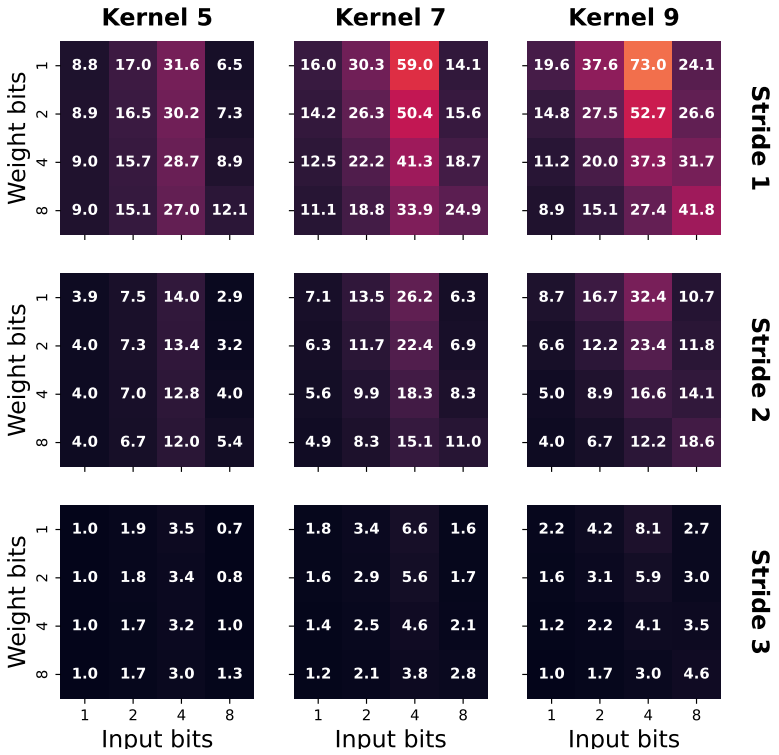


(b)

Figura K.8: Energía por operación en [pJ] del acelerador ChSymSim (0,81V@8MHz), con operaciones de 8 bits y ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

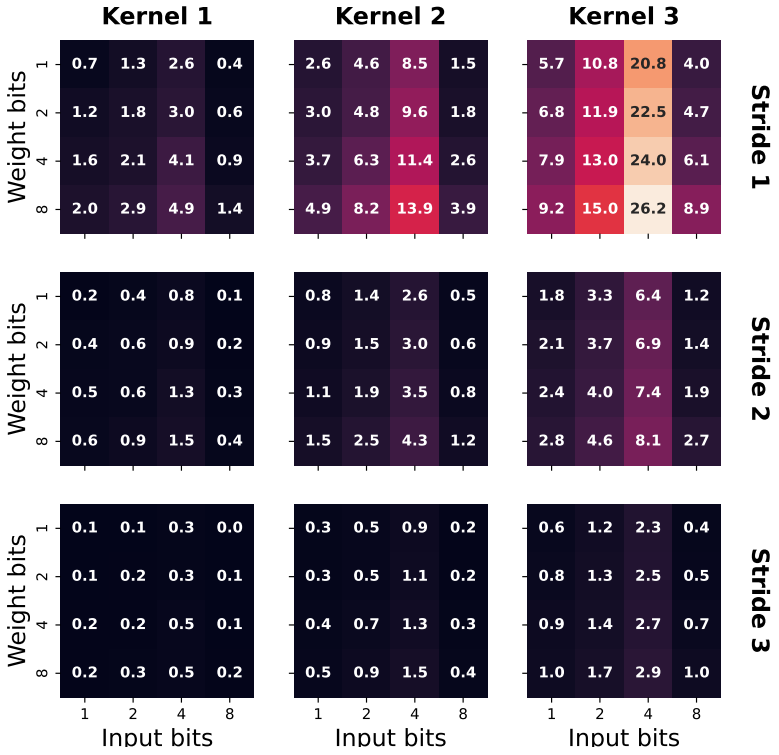


(a)

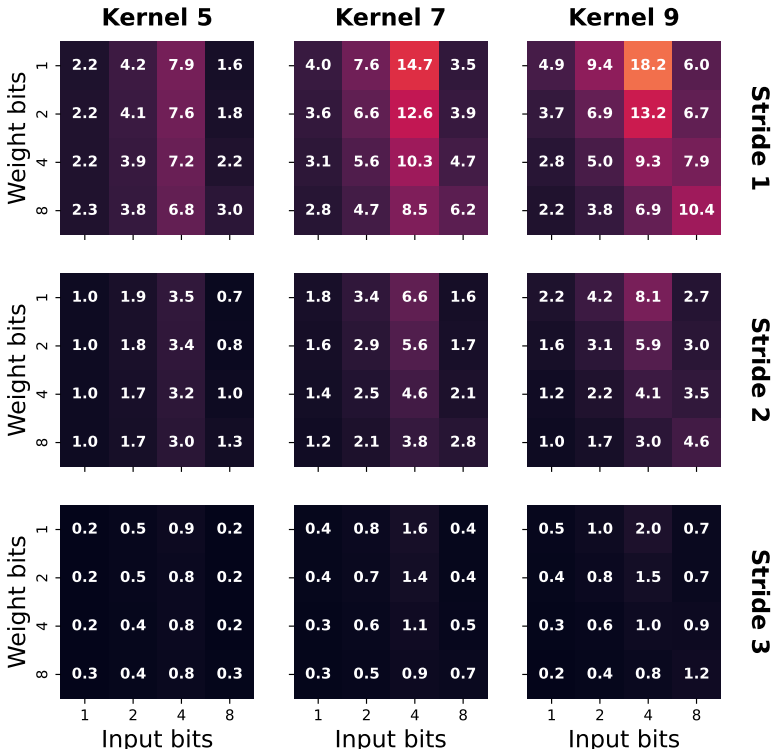


(b)

Figura K.9: Rendimiento en [GOPS] del acelerador ChSymSim (operaciones de 8 bits @100MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

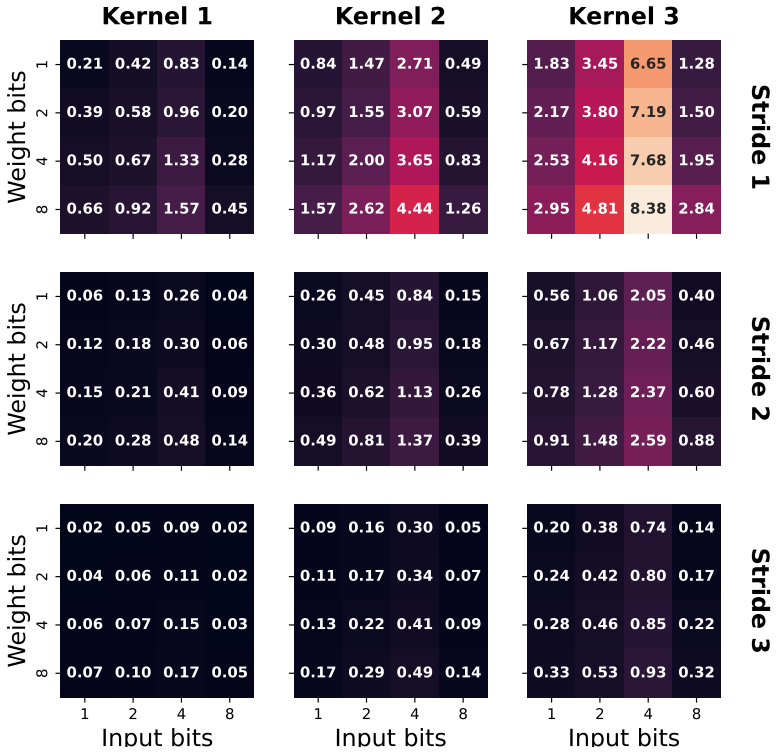


(a)

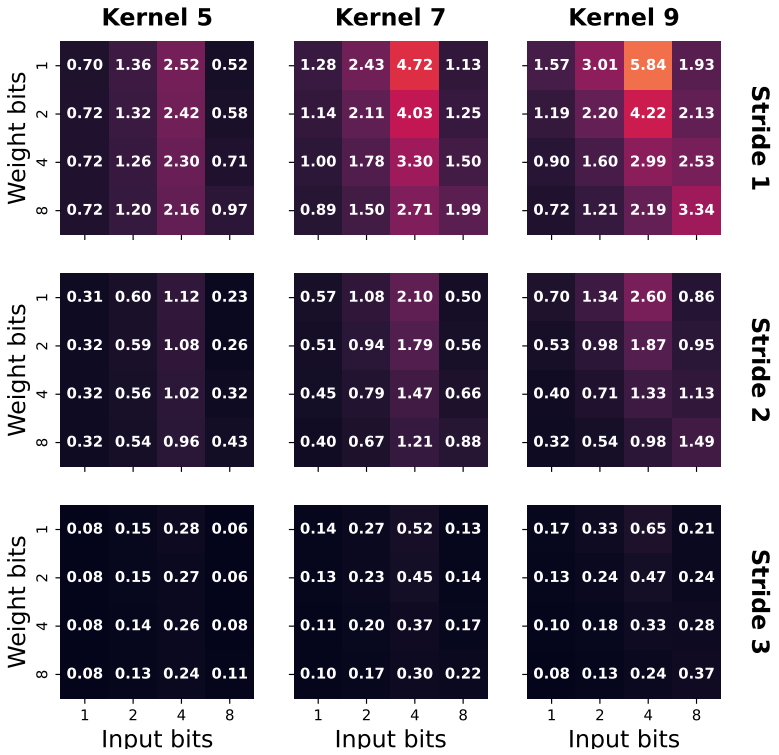


(b)

Figura K.10: Rendimiento en [GOPS] del acelerador ChSymSim (operaciones de 8 bits @25MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

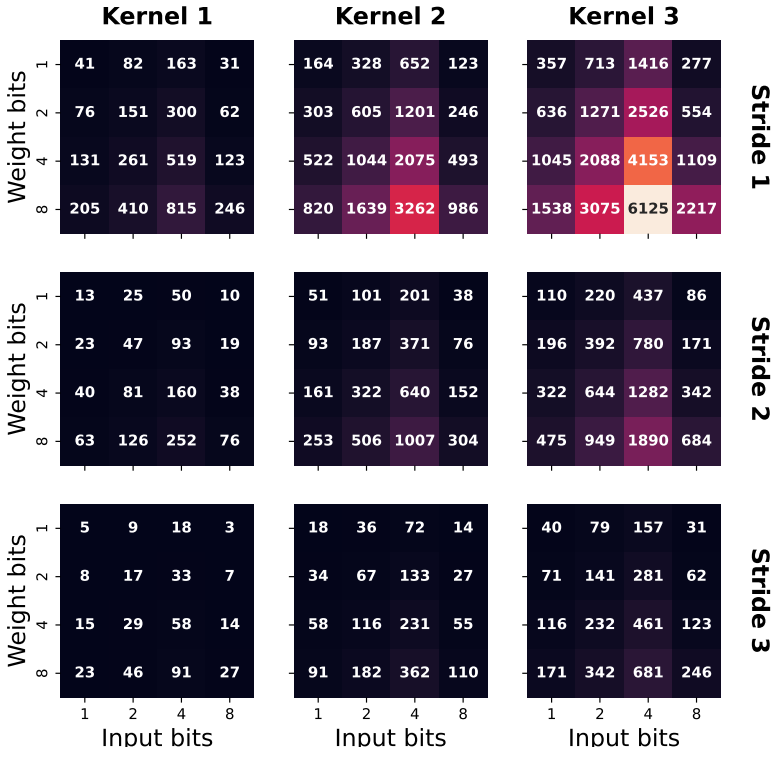


(a)

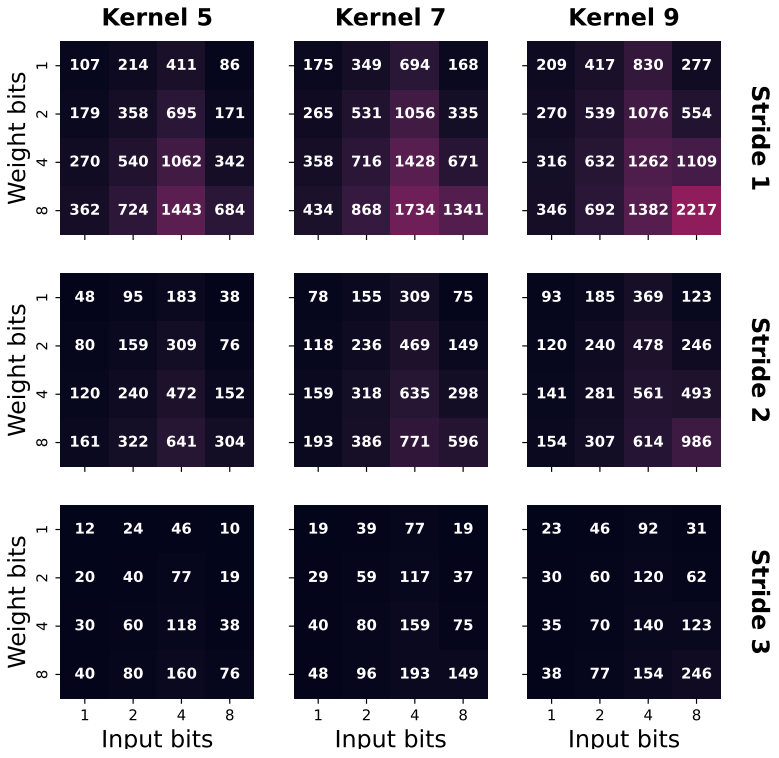


(b)

Figura K.11: Rendimiento en [GOPS] del acelerador ChSymSim (operaciones de 8 bits @8MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

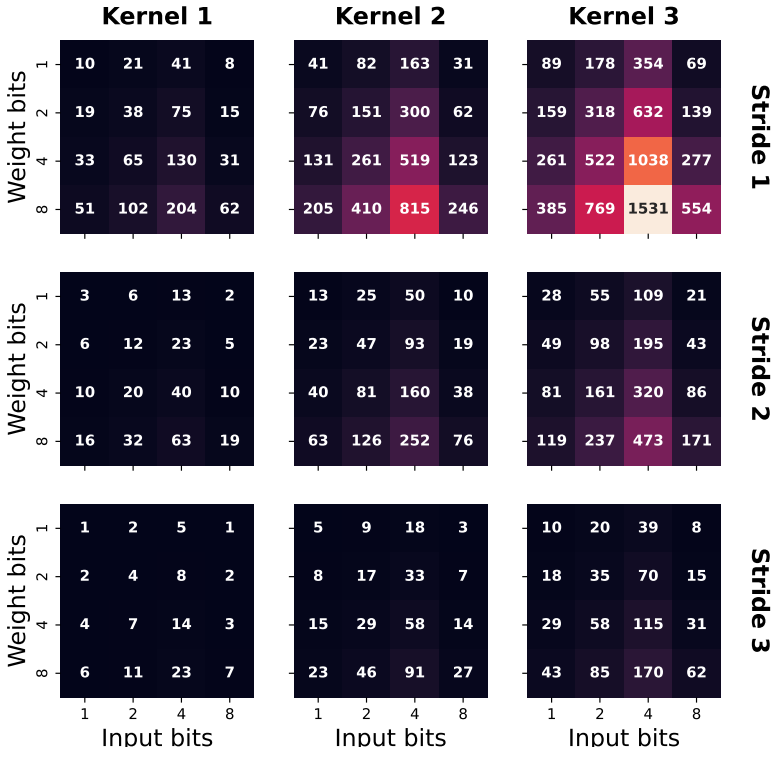


(a)

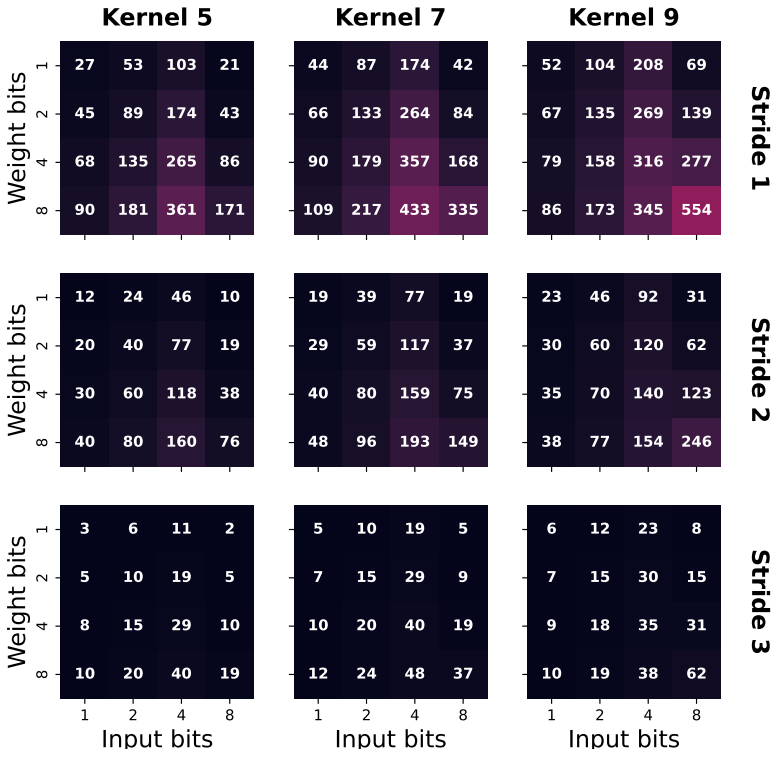


(b)

Figura K.12: Rendimiento en [MMACS] del acelerador ChSymSim (operaciones de 8 bits @100MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

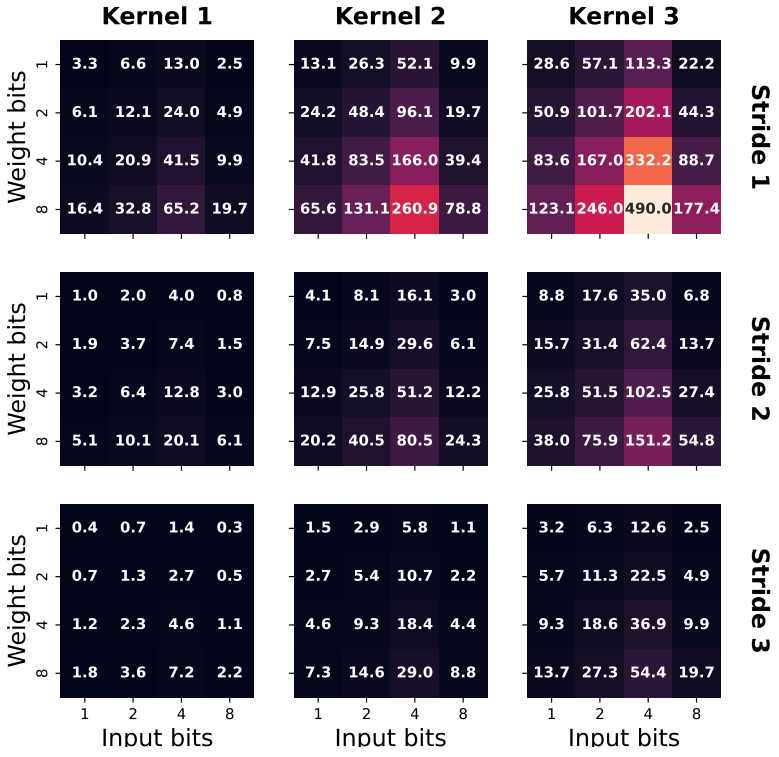


(a)

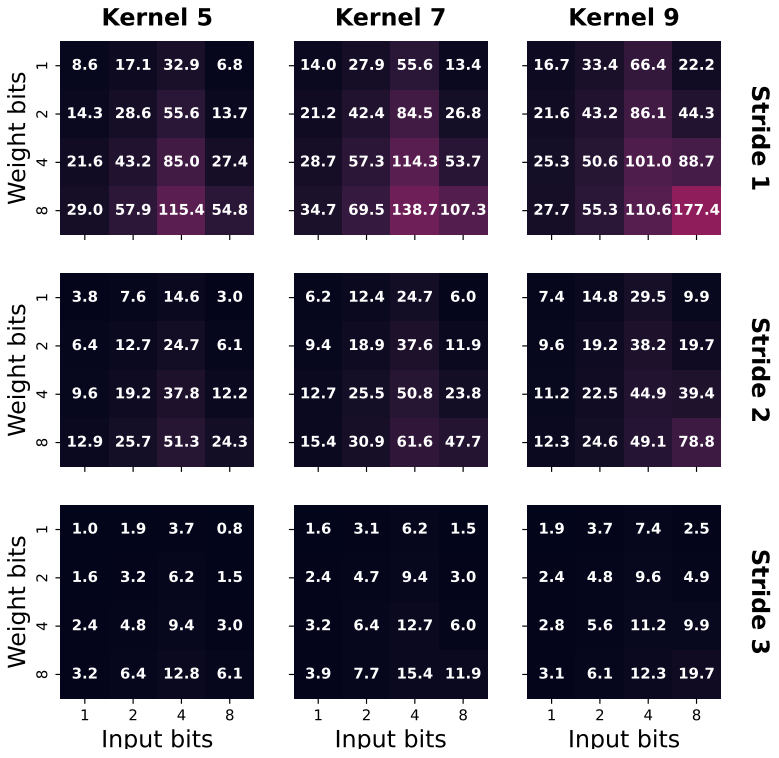


(b)

Figura K.13: Rendimiento en [MMACS] del acelerador ChSymSim (operaciones de 8 bits @25MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

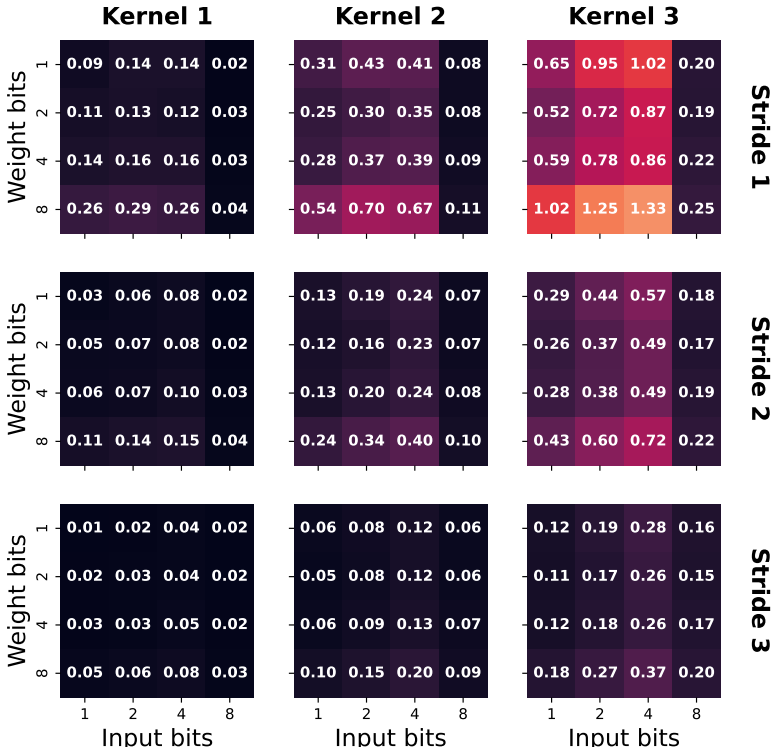


(a)

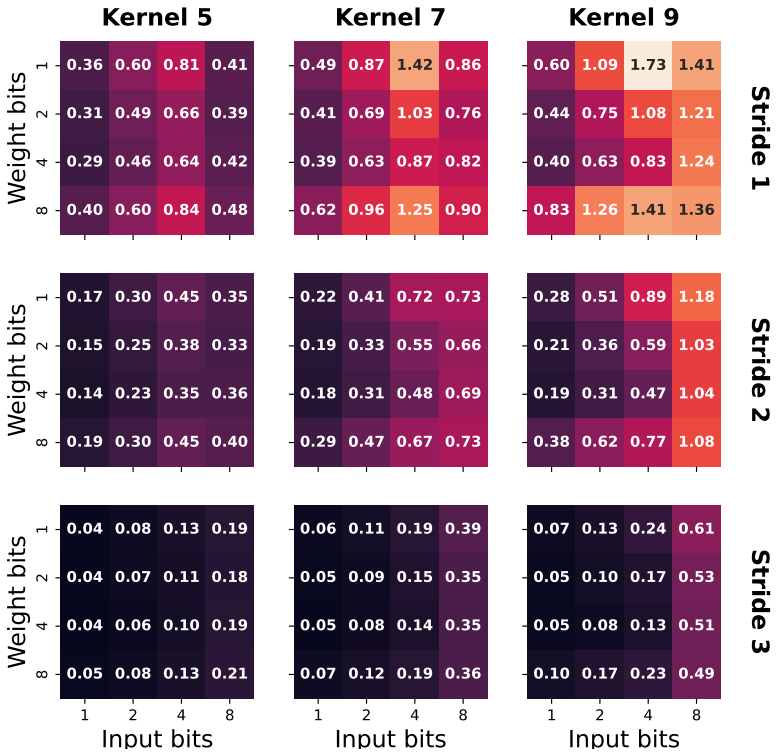


(b)

Figura K.14: Rendimiento en [MMACS] del acelerador ChSymSim (operaciones de 8 bits @8MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

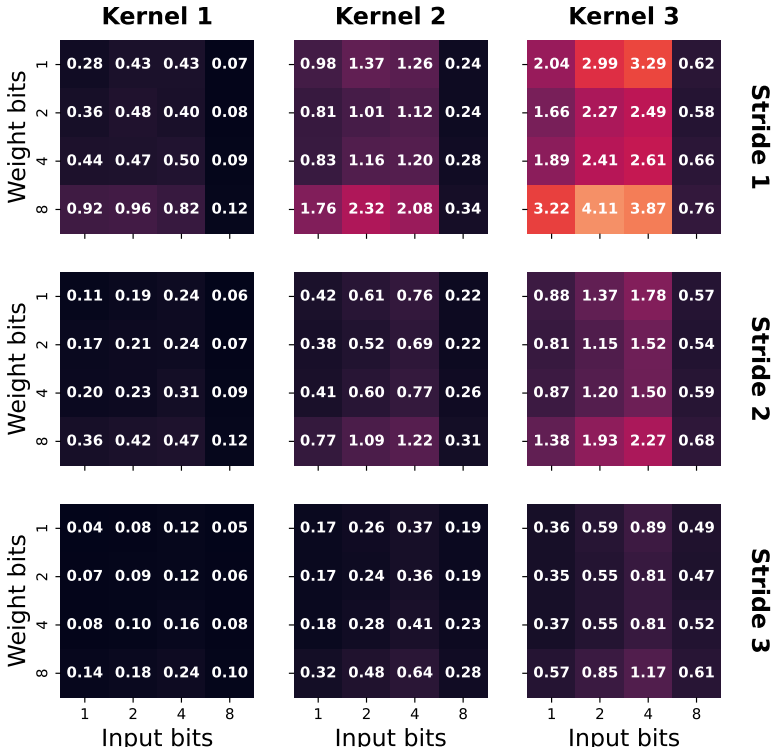


(a)

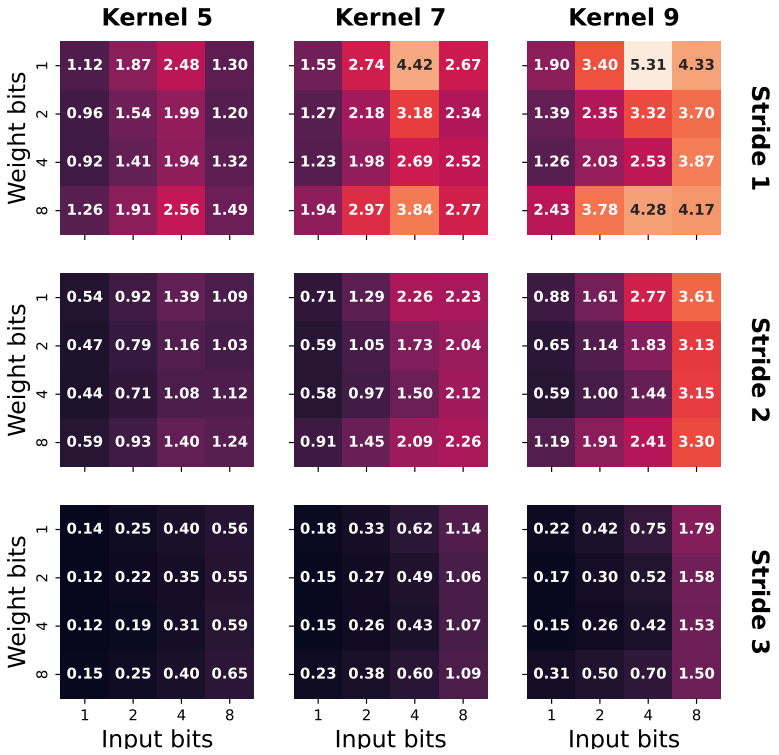


(b)

Figura K.15: Eficiencia en [TOPS/W] del acelerador ChSymSim (operaciones de 8 bits y 2,1V@100MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

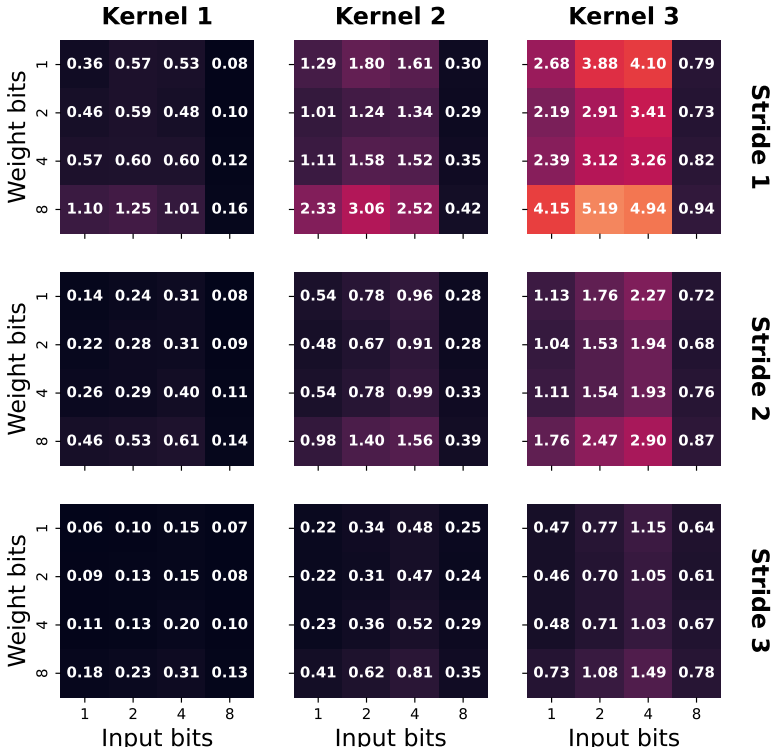


(a)

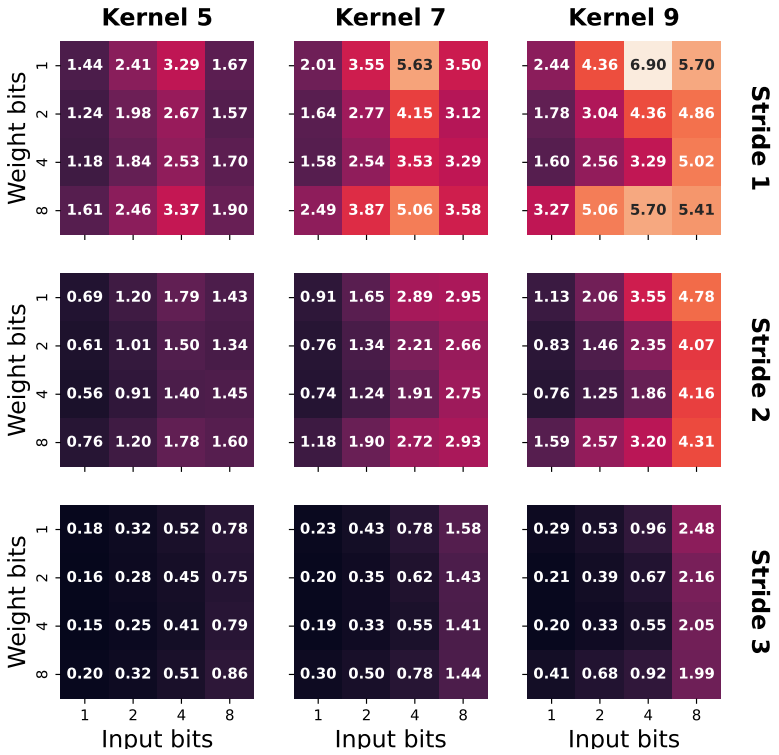


(b)

Figura K.16: Eficiencia en [TOPS/W] del acelerador ChSymSim (operaciones de 8 bits y 1,2V@25MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

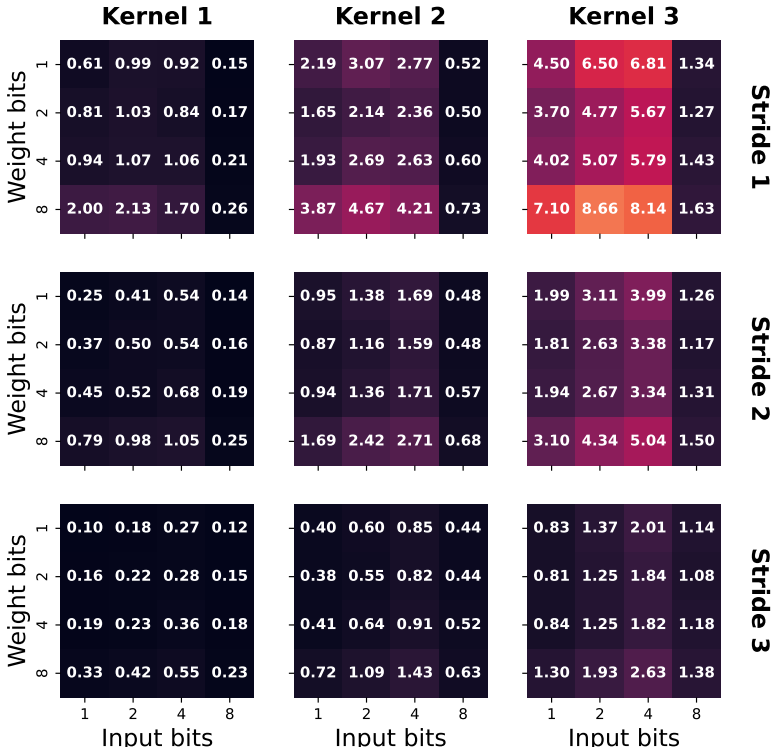


(a)

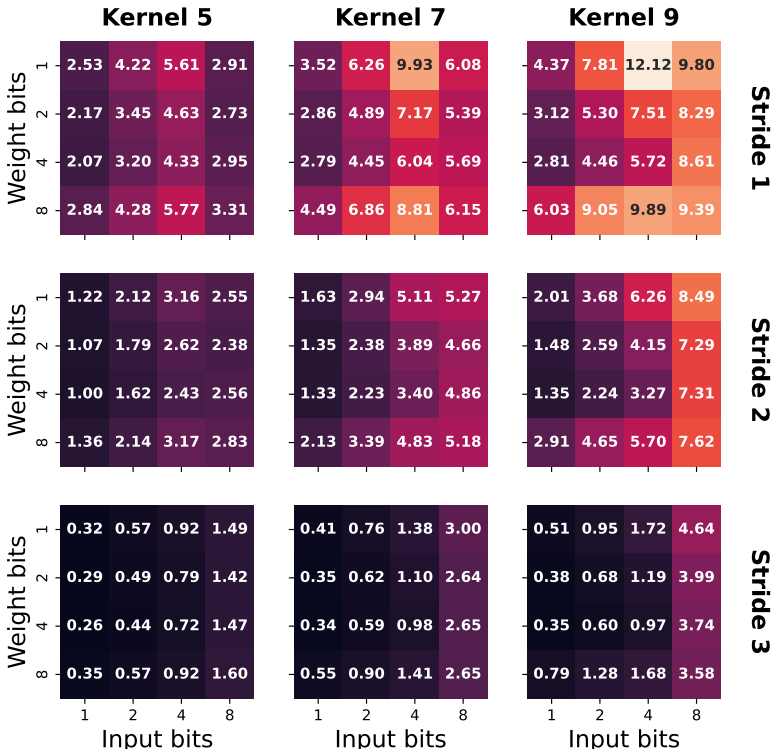


(b)

Figura K.17: Eficiencia en [TOPS/W] del acelerador ChSymSim (operaciones de 8 bits y 1,07V@25MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

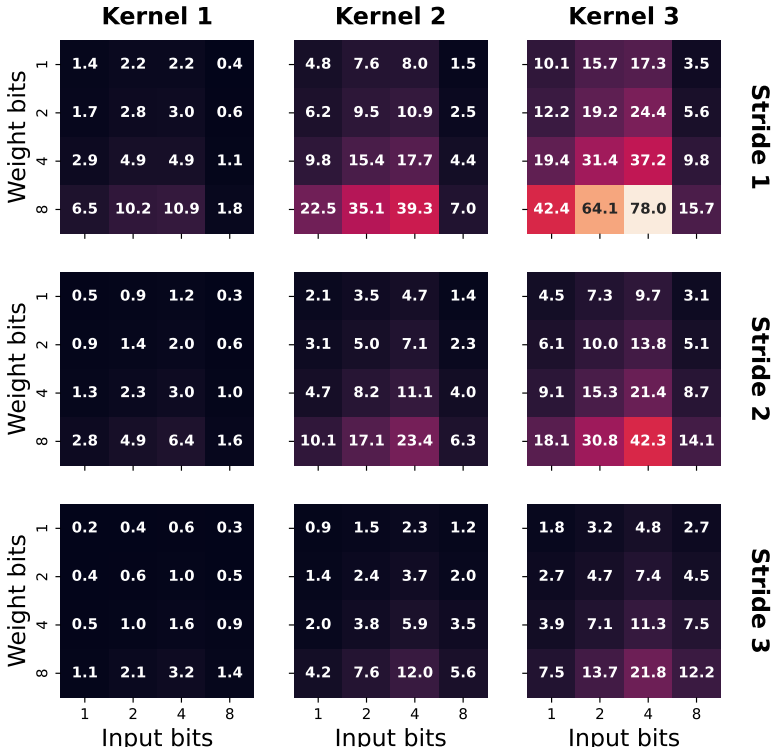


(a)

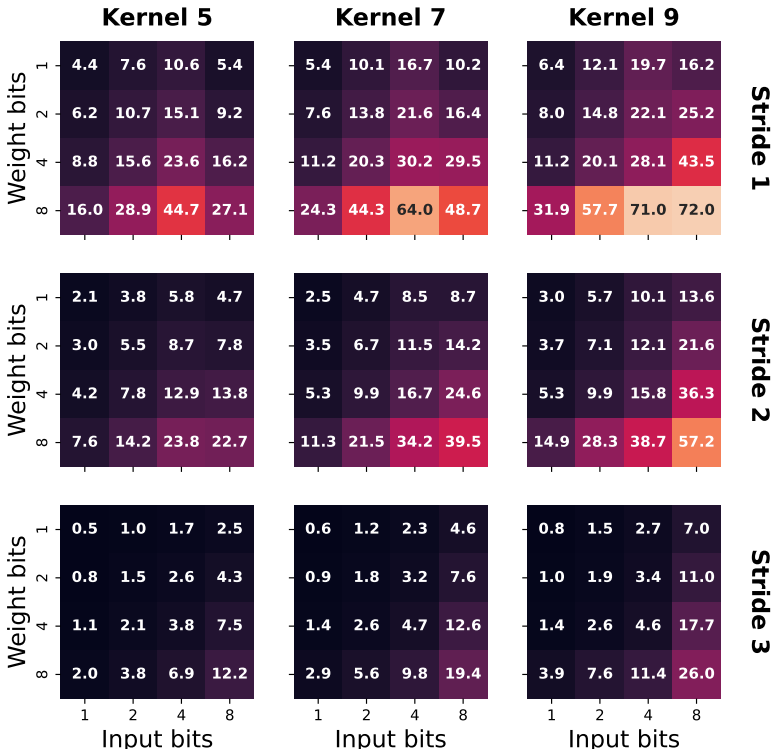


(b)

Figura K.18: Eficiencia en [TOPS/W] del acelerador ChSymSim (operaciones de 8 bits y 0,81V@8MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

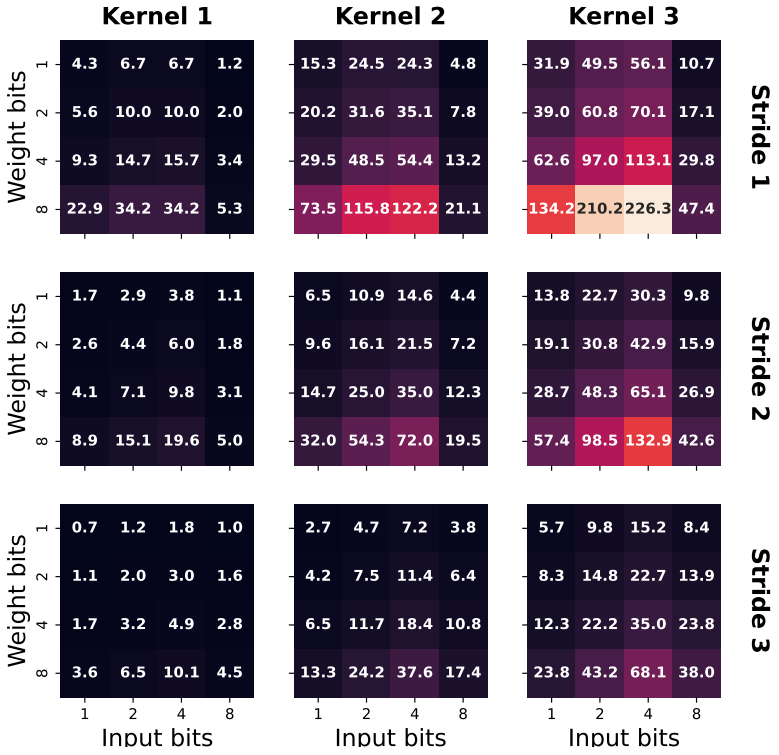


(a)

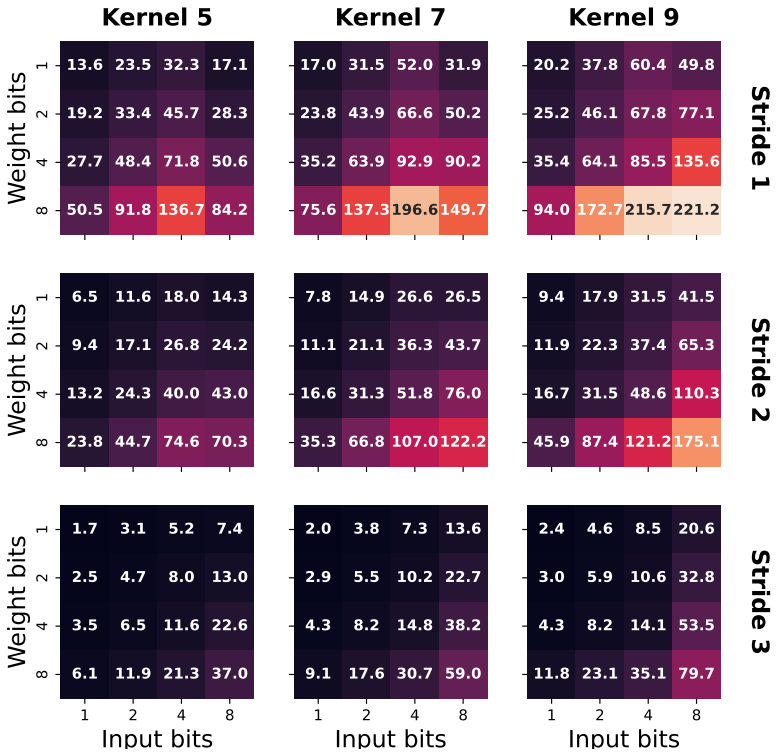


(b)

Figura K.19: Eficiencia en [GMACS/W] del acelerador ChSymSim (operaciones de 8 bits y 2,1V@100MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

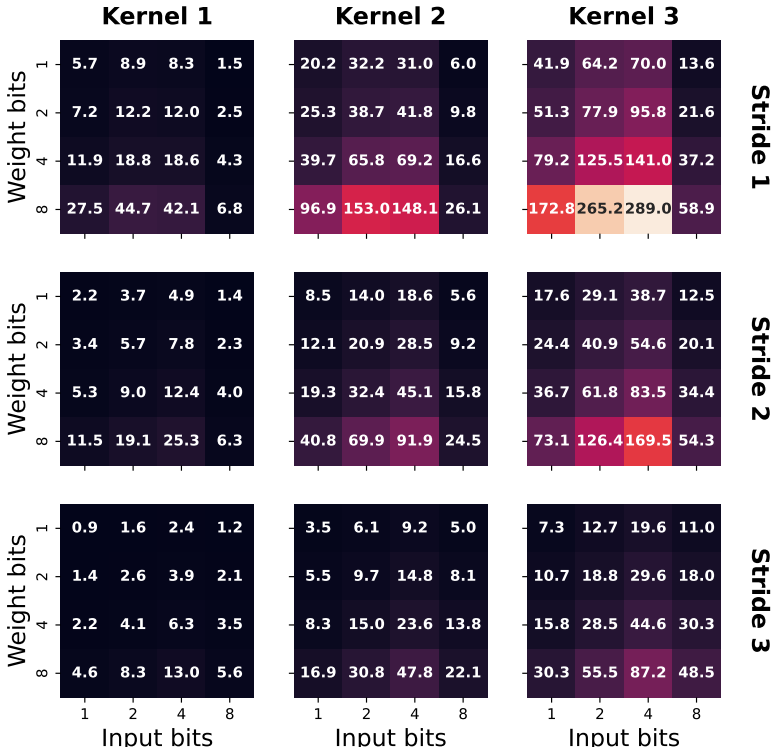


(a)

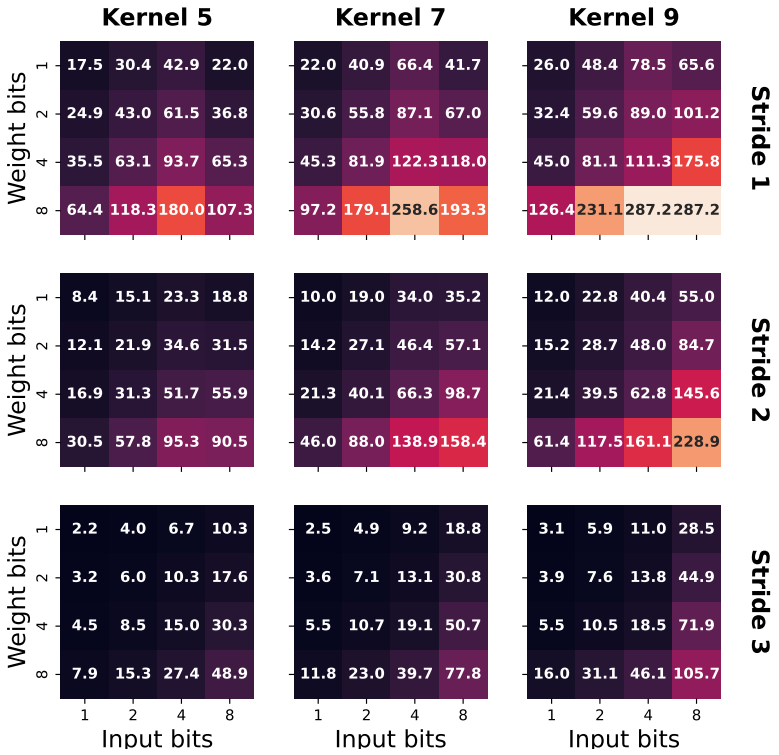


(b)

Figura K.20: Eficiencia en [GMACS/W] del acelerador ChSymSim (operaciones de 8 bits y 1,2V@25MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).

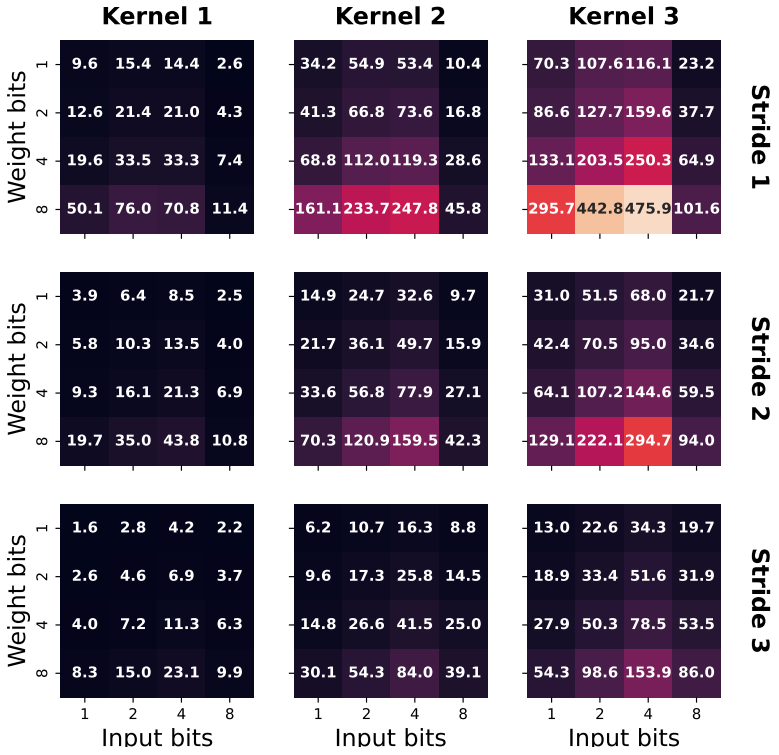


(a)

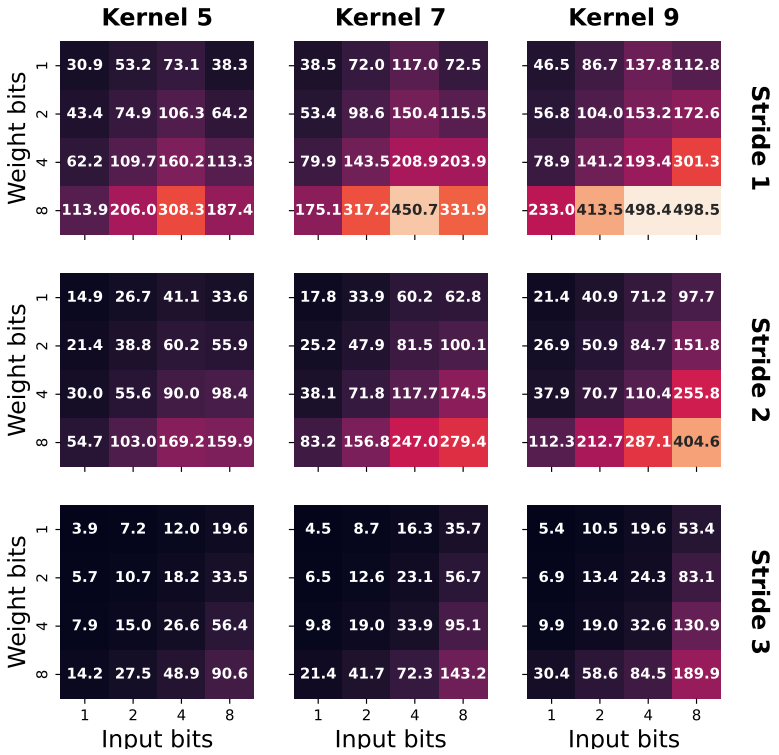


(b)

Figura K.21: Eficiencia en [GMACS/W] del acelerador ChSymSim (operaciones de 8 bits y 1,07V@25MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).



(a)



(b)

Figura K.22: Eficiencia en [GMACS/W] del acelerador ChSymSim (operaciones de 8 bits y 0,81V@8MHz), ejecutando diferentes configuraciones: a) kernel pequeños (1×1 hasta 3×3); b) kernel grandes (hasta 9×9).