Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Programa de Licenciatura en Ingeniería Electrónica

Desarrollo de un acelerador de función softmax con soporte para datos de punto fijo y punto flotante de precisión arbitraria para la ejecución de redes neuronales.

Informe de Trabajo Final de Graduación para optar por el título de
Ingeniero en Electrónica

con el grado académico de
Licenciatura

Anthony Leiva Valverde

Cartago, 26 de noviembre de 2025

Declaro que el presente documento de tesis ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos y resultados experimentales propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de tesis realizado y por el contenido del presente documento.

Anthony Leiva Valverde

Cartago, 26 de noviembre de 2025
Céd: 1-1794-0918

This work is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 4.0 International" license.

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Trabajo Final de Graduación
Acta de Aprobación

Defensa de Trabajo Final de Graduación
Requisito para optar por el título de Ingeniero en Electrónica
Grado Académico de Licenciatura
Instituto Tecnológico de Costa Rica

El Tribunal Evaluador aprueba la defensa del Trabajo Final de Graduación denominado *Desarrollo de un acelerador de función softmax con soporte para datos de punto fijo y punto flotante de precisión arbitraria para la ejecución de redes neuronales.*, realizado por el señor Anthony Leiva Valverde, y hace constar que cumple con las normas establecidas por la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal Evaluador

Dr.-Ing. Juan José Montero Rodríguez
Profesor Lector

Dr.-Ing William Quirós Solano
Profesor Lector

Dr.-Ing. Jorge Castro-Godínez
Profesor Asesor

Cartago, 26 de noviembre de 2022

Resumen

La función *softmax* es ampliamente utilizada como etapa final de clasificación en redes neuronales, donde transforma valores arbitrarios en distribuciones de probabilidad. Sin embargo, su implementación eficiente en hardware resulta desafiante debido al elevado costo computacional de operaciones como exponenciales, sumatorias acumulativas y divisiones. En este trabajo se evalúan diversas técnicas de computación aproximada para acelerar la función *softmax* empleando *High-Level Synthesis* (HLS), considerando aproximaciones mediante Series de Taylor e interpolación lineal basada en tablas de búsqueda (LUTs). El diseño soporta aritmética en punto fijo y punto flotante de precisión arbitraria, permitiendo explorar el compromiso entre precisión numérica, latencia y utilización de recursos.

Los resultados muestran que, para anchos de palabra reducidos, las aproximaciones basadas en punto flotante mantienen la estabilidad numérica necesaria para su integración en redes neuronales, obteniendo errores inferiores al 1 % con polinomios de Taylor de tercer orden. Por otro lado, las aproximaciones en punto fijo presentan limitaciones asociadas al rango dinámico, lo que afecta la correcta normalización de los valores de salida. En términos de rendimiento y eficiencia, las aproximaciones de Taylor de primer y segundo orden proporcionan un balance favorable entre reducción de recursos, baja latencia y precisión aceptable, mientras que las LUTs permiten una mejora moderada cuando se incrementa el número de muestras.

Palabras clave: redes neuronales, computación aproximada, punto flotante, interpolación, Series de Taylor.

Abstract

The *softmax* function is widely used in the output layers of neural networks to convert raw activation values into normalized probability distributions. Despite its conceptual simplicity, implementing softmax efficiently in hardware is challenging due to the computational cost of exponential and division operations. This work evaluates approximate computing strategies for softmax acceleration using High-Level Synthesis (HLS), focusing on Taylor series expansions and lookup table (LUT) based linear interpolation. The proposed architecture supports both fixed-point and arbitrary-precision floating-point arithmetic, enabling a detailed exploration of the trade-offs among accuracy, latency, and hardware resource utilization.

Experimental results show that, for reduced bit-width configurations, floating-point approximations preserve the numerical stability required for neural network inference, achieving errors below 1 % with third-order Taylor polynomials. In contrast, fixed-point implementations exhibit noticeable degradation due to limited dynamic range, which restricts the accurate representation of small probability values. From a performance perspective, first- and second-order Taylor approximations offer a favorable balance between resource reduction, numerical accuracy, and execution time, while LUT-based methods provide additional improvements when increasing the sampling resolution.

Keywords: Neural Networks, Approximate Computing, Floating Point, Interpolation, Taylor Series.

a mi amada familia...

Agradecimientos

El resultado de este trabajo no hubiese sido posible sin conocimiento y el apoyo del M.Sc. Luis Leon Vega, quien me acompañó durante el desarrollo del proyecto, y quien siempre tuvo la confianza para lograrlo. Quiero agradecer tambien al profesor Dr. Jorge Castro-Godínez, el cuál me brindó el seguimiento y apoyo necesario para lograrlo. Además, quiero agradecer a mi madre, Elizabeth Valverde Ramírez, por el apoyo incondicional que siempre me brinda, a mi novia, Keichel Elena Zamora Huertas, quien fue un pilar importante para lograr este objetivo. A los profesores que me ayudaron a convertirme en el profesional que soy, a mi familia y a mis amigos que estuvieron a mi lado en este proceso.

Anthony Leiva Valverde

Cartago, 26 de noviembre de 2025

Índice general

Índice de figuras III

Índice de tablas V

1. Introducción 1

1.1. Implementación de funciones no lineales en *FPGAs* . . . 1
1.2. Desarrollo de aceleradores de función Softmax . . . 3
1.3. Objetivos y estructura del documento . . . 4

2. Antecedentes y trabajo relacionado 6

2.1. Antecedentes . . . 6
2.1.1. Computación aproximada . . . 6
2.1.2. Series de Taylor . . . 7
2.1.3. Interpolación lineal por tramos basada en tablas de búsqueda (LUT) 8
2.1.4. High-Level Synthesis . . . 9
2.1.5. Redes neuronales profundas . . . 10
2.1.6. CuFP: An HLS Library for Customized Floating-Point Operators . 12
2.1.7. Template specialization y metaprogramación . . . 14

3. Desarrollo de acelerador de la función Softmax con soporte de datos punto fijo y punto flotante de precisión arbitraria 15

3.1. Implementación base en C++ y porteo a HLS . . . 15
3.2. Aproximación de exponencial por medio de Series de Taylor . . . 18
3.3. Aproximación de exponencial por medio de Interpolación Lineal por LUT . 19
3.4. Implementación de soporte de datos para punto flotante de precisión arbitraria . . . 21
3.5. Generación del banco de pruebas . . . 23

4. Resultados y análisis 30

4.1. Comparativa de recursos y latencia . . . 30
4.2. Análisis de la aproximación por Series de Taylor . . . 33
4.3. Análisis de la aproximación por LUTs . . . 34
4.4. Análisis del impacto de la frecuencia de Reloj (F_{clk}) . . . 35
4.5. Análisis del impacto del tamaño del vector de entrada . . . 37
4.6. Análisis de precisión con LeNet-5 (12 bits) . . . 39

4.7. Análisis de precisión con estándar *IEEE 754 half-precision* (16 bits) 40

5. Conclusiones y trabajo futuro **42**
5.1. Conclusiones 42
5.2. Trabajo futuro 43

Bibliografía **44**

Índice de figuras

1.1. Rol de la función *Softmax* dentro de una red neuronal 2
1.2. Diagrama de flujo general de la solución propuesta. 4

2.1. Interpolación lineal por LUT para la función exponencial. 8
2.2. Red neuronal profunda 11
2.3. Diagrama de bloques de la multiplicación 12
2.4. Diagrama de bloques de la suma/resta 13

3.1. Diagrama de flujo para la determinación del tipo de dato en tiempo de compilación. 22

4.1. Comparativa de utilización de recursos entre la versión exacta, aproximación por Series de Taylor y aproximación por LUTs (Punto flotante). . . . 31
4.2. Comparativa de utilización de recursos entre la versión exacta, aproximación por Series de Taylor y aproximación por LUTs (Punto fijo). 31
4.3. Comparación de latencia (tiempo de ejecución) para las implementaciones en Punto Flotante vs versión exacta (punto fijo). 32
4.4. Comparación de latencia (tiempo de ejecución) para las implementaciones en Punto Fijo. 32
4.5. Impacto del orden del polinomio de Taylor en los recursos y el tiempo de ejecución (Punto Flotante). 33
4.6. Impacto del orden del polinomio de Taylor en los recursos y el tiempo de ejecución (Punto Fijo). 33
4.7. Análisis de consumo de recursos y tiempo al variar la cantidad de muestras (Samples) en Punto Flotante. 34
4.8. Análisis de consumo de recursos y tiempo al variar la cantidad de muestras (Samples) en Punto Fijo. 34
4.9. Efecto de la frecuencia de reloj sobre el desempeño de la aproximación de Taylor en Punto Flotante. 35
4.10. Efecto de la frecuencia de reloj sobre el desempeño de la aproximación de Taylor en Punto Fijo. 36
4.11. Efecto de la frecuencia de reloj sobre el desempeño de la aproximación por LUTs en Punto Flotante. 36
4.12. Efecto de la frecuencia de reloj sobre el desempeño de la aproximación por LUTs en Punto Fijo. 37

4.13. Comportamiento de la arquitectura de Taylor (Punto Flotante) respecto al tamaño del vector de entrada. 37
4.14. Comportamiento de la arquitectura de Taylor (Punto Fijo) respecto al tamaño del vector de entrada. 38
4.15. Comportamiento de la arquitectura de LUTs (Punto Flotante) respecto al tamaño del vector de entrada. 38
4.16. Comportamiento de la arquitectura de LUTs (Punto Fijo) respecto al tamaño del vector de entrada. 39

Índice de tablas

3.8. Configuración del Banco de Pruebas para la evaluación del núcleo de Punto Flotante (CuFP) utilizando aproximación por Series de Taylor. 24
3.9. Configuración del Banco de Pruebas para la evaluación del núcleo de Punto Flotante (CuFP) utilizando aproximación basada en LUTs. 25
3.10. Configuración del Banco de Pruebas para la evaluación de la aproximación mediante Series de Taylor en aritmética de Punto Fijo. 26
3.11. Configuración del Banco de Pruebas para la evaluación de la aproximación mediante LUTs en aritmética de Punto Fijo. 27
3.12. Configuración del Banco de Pruebas para la evaluación de la implementación Exacta (`hls::exp()`). 28
3.13. Configuraciones para LeNet (12 bits, punto fijo) 28
3.14. Configuraciones para LeNet (12 bits, punto flotante) 29
3.15. Configuraciones para IEEE Half (16 bits, punto fijo) 29
3.16. Configuraciones para IEEE Half (16 bits, punto flotante) 29

4.1. Comparación de calidad de resultados para configuración LeNet-5 (12-bit) 40
4.2. Comparación de calidad de resultados para estándar IEEE Half-Precision (16-bit) 41

Capítulo 1

Introducción

1.1. Implementación de funciones no lineales en *FPGAs*

El uso de redes neuronales convolucionales, por sus siglas en inglés *Convolutional Neural Networks* (CNN), ha demostrado ser altamente efectivo en tareas de procesamiento de imágenes, como la clasificación, la detección de objetos y el reconocimiento de caracteres. Estas redes aprovechan operaciones de convolución para extraer características espaciales relevantes, lo que las hace especialmente adecuadas para analizar datos visuales. Además, las CNN resultan más eficientes en términos de memoria y complejidad computacional que las redes neuronales tradicionales [1].

En el ámbito de los sistemas computacionales, las CPU se emplean como principal recurso de procesamiento. Sin embargo, a medida que las aplicaciones se vuelven más complejas y los requisitos de rendimiento aumentan, aparecen limitaciones que afectan la eficiencia y el rendimiento global de los sistemas que utilizan CPU. Algunas de estas limitaciones se manifiestan en forma de cuellos de botella, ineficiencias energéticas y latencias de procesamiento, lo que obliga a los desarrolladores a considerar alternativas más especializadas, como las FPGA, para solventar estas barreras [2].

Uno de los principales desafíos al implementar redes neuronales en *hardware*, especialmente en arquitecturas basadas en FPGA, radica en la ejecución de funciones no lineales, como exponenciales, logaritmos o sigmoides. Estas funciones, al no ser representables mediante operaciones aritméticas simples, demandan un uso intensivo de recursos lógicos y de memoria [3].

En las capas de salida de las redes neuronales se emplea comúnmente un método de clasificación basado en la función *Softmax*. Esta función transforma el vector de salidas preestablecidas del modelo en una distribución de probabilidad, en la que la suma total de los valores es igual a uno. Así, cada componente del vector representa la probabilidad de que la entrada pertenezca a una clase específica, lo cual permite interpretar los resultados

de forma clara y cuantificable. La Figura 1.1 muestra el modelo gráfico del rol de la función *Softmax* en una red neuronal [4].

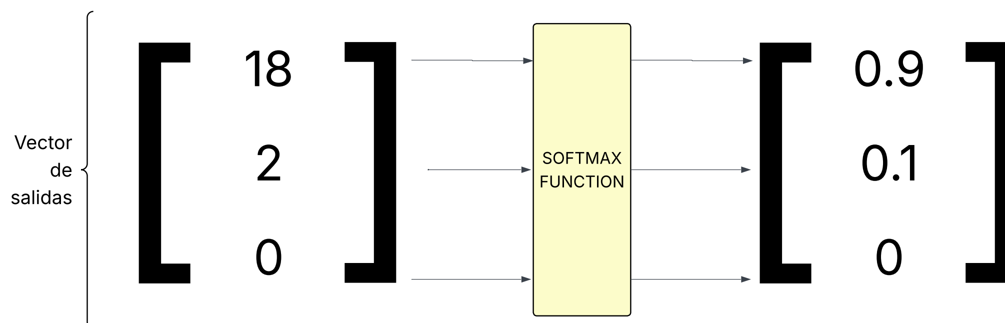

Figura 1.1: Rol de la función *Softmax* dentro de una red neuronal. Adaptado de [5].

La función *Softmax* constituye un componente esencial en las redes neuronales modernas, ya que se encarga de transformar el vector de salidas de una capa en una distribución de probabilidad interpretable. Su cálculo depende de operaciones exponenciales y divisiones, las cuales resultan especialmente costosas cuando se implementan directamente en *hardware*, como en FPGAs. Este alto costo computacional puede provocar cuellos de botella, limitar la escalabilidad del sistema y restringir su uso en plataformas con recursos limitados o con requerimientos estrictos de latencia y consumo energético. Debido a esta problemática, la computación aproximada se presenta como una alternativa prometedora, al permitir representar estas funciones mediante modelos de menor complejidad, reduciendo el consumo de recursos a cambio de una pérdida de precisión generalmente tolerable. Por este motivo, el presente proyecto plantea la implementación de al menos dos aproximaciones de la función *Softmax*, con el fin de evaluar su eficiencia y exactitud en entornos de cómputo reconfigurable [3], [6].

La función *Softmax* está dada por:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{1.1}$$

Uno de los principales retos actuales es el diseño de versiones aproximadas de funciones no lineales que reduzcan significativamente el uso de recursos, manteniendo una precisión aceptable para las tareas designadas. Las implementaciones específicas de funciones no lineales en FPGAs siguen siendo limitadas y, en algunos casos, presentan una complejidad excesiva en su diseño o en el uso de recursos lógicos, dificultando su integración en sistemas embebidos.

Otro aspecto crucial en el diseño de una implementación eficiente de la función *Softmax* es la selección del tipo de representación numérica. El uso de aritmética basada en punto fijo suele resultar más eficiente en términos de consumo de recursos y latencia; sin embargo,

esta eficiencia implica una posible pérdida de precisión que puede volverse significativa dependiendo del rango dinámico y la variabilidad de los datos de entrada. Por su parte, la representación en punto flotante proporciona una mayor precisión y estabilidad numérica, aunque a costa de un incremento considerable en la complejidad del *hardware* y en la utilización de recursos lógicos [7].

El presente proyecto se desarrolla en el ECAS Lab (*Efficient Computing Across the Stack Lab*) del Instituto Tecnológico de Costa Rica, un grupo de investigación enfocado en la exploración de soluciones de cómputo eficiente tanto en el *edge* como en entornos en la nube. Su trabajo abarca desde el diseño de circuitos hasta el desarrollo de aplicaciones a nivel de software, promoviendo la integración de técnicas de alto rendimiento y eficiencia energética en sistemas embebidos [8].

En el ECAS Lab se encuentra en desarrollo una biblioteca de código abierto denominada *Flexible Accelerator Library* (FAL), compuesta por aceleradores descritos en HLS que pueden ser sintetizados en FPGAs. Esta biblioteca se caracteriza por ser parametrizable en aspectos como la representación numérica, el nivel de precisión y el número de unidades de procesamiento, y permite ejecutar operaciones como multiplicaciones matriciales, operaciones elemento a elemento y convoluciones. Con ello, se busca facilitar el diseño y la exploración de aceleradores para la inferencia de redes neuronales en FPGAs de gama baja [9].

De esta manera, el presente proyecto aprovecha al máximo los recursos disponibles para implementar y optimizar la función *Softmax* mediante técnicas de computación aproximada, con el fin de ampliar el *framework* existente (FAL) y aumentar su aplicabilidad en diferentes escenarios. En particular, se plantea el desarrollo de una versión aproximada de la función *Softmax* optimizada para *hardware*, capaz de ejecutarse en FPGAs y alcanzar un balance adecuado entre precisión, uso de recursos y latencia. Esta propuesta permite la integración de redes neuronales eficientes en dispositivos de borde, posibilitando aplicaciones más rápidas, energéticamente eficientes y económicamente viables en entornos con recursos limitados.

1.2. Desarrollo de aceleradores de función Softmax

Para el desarrollo del acelerador se estableció un flujo de trabajo orientado al diseño e implementación de versiones aproximadas de la función *Softmax*, optimizadas para su ejecución en plataformas FPGA. El proceso se estructuró en varias etapas, iniciando con el análisis matemático de la función y la selección de técnicas de computación aproximada adecuadas para reducir la complejidad computacional, manteniendo una precisión aceptable para tareas de inferencia en redes neuronales.

Posteriormente, las aproximaciones seleccionadas fueron la aproximación por Series de Taylor y la interpolación por tramos basada en tablas de búsqueda (LUT). Ambas aproximaciones fueron descritas en lenguaje C++ y sintetizadas mediante la herramienta *Vitis*

HLS (*High-Level Synthesis*) de AMD. Cada diseño fue parametrizado para soportar diferentes anchos de datos y tipos de datos tanto de punto fijo como de punto flotante, permitiendo evaluar su impacto en el uso de recursos lógicos, la latencia y el rendimiento general del sistema.

La Figura 1.2 presenta el diagrama de flujo general de la solución propuesta, en el cual se ilustra la secuencia de etapas involucradas en el desarrollo del acelerador: desde la definición matemática de la función, pasando por su modelado en software y síntesis en hardware, hasta la validación de resultados y su posterior integración en la biblioteca FAL.

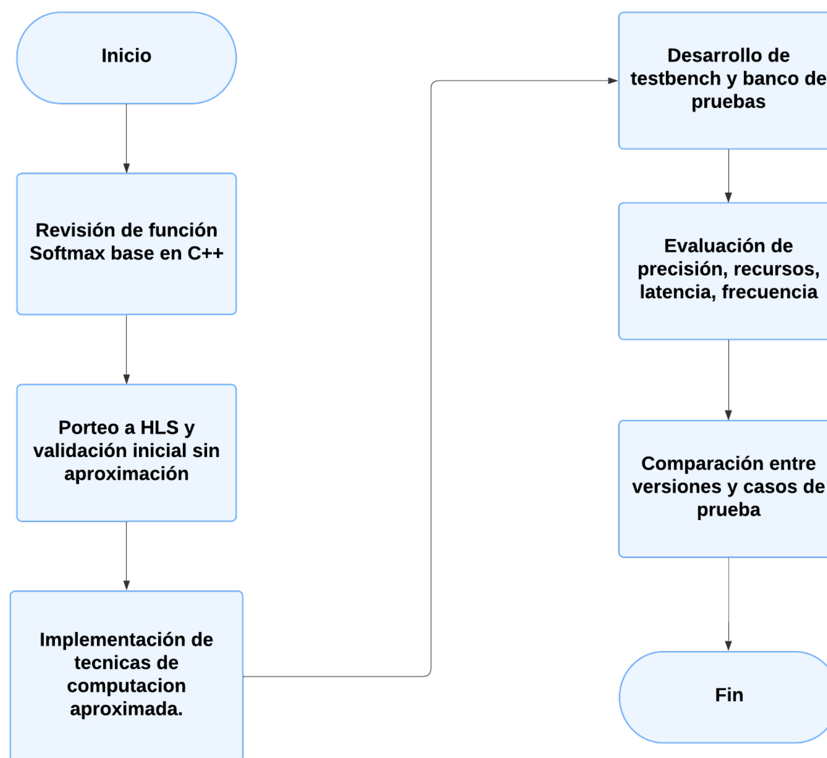

Figura 1.2: Diagrama de flujo general de la solución propuesta.

1.3. Objetivos y estructura del documento

El objetivo principal de este proyecto consiste en diseñar un acelerador para la función *Softmax* utilizando técnicas de computación aproximada, en conjunto con las bibliotecas de la herramienta *Vitis HLS*, con soporte tanto para punto fijo como para punto flotante de precisión arbitraria. Con ello, se busca desarrollar un diseño eficiente que optimice el uso de recursos lógicos en plataformas FPGA, manteniendo un nivel de precisión adecuado para tareas de inferencia en redes neuronales.

Para cumplir con este propósito, se planteó la implementación de al menos dos métodos de computación aproximada orientados al cálculo de la función *Softmax*, con el fin de comparar su rendimiento, consumo de recursos y calidad de resultados. Posteriormente, se diseñó un entorno de validación en C++/HLS, en conjunto con un banco de pruebas destinado a verificar la funcionalidad, precisión y estabilidad numérica de las versiones aproximadas desarrolladas. A continuación, se evaluaron distintas configuraciones del acelerador bajo diversas condiciones de prueba, considerando métricas como la precisión de salida, el uso de recursos lógicos, la latencia y la frecuencia de operación alcanzada.

El presente documento se organiza de la siguiente manera: en el **Capítulo 2** se exponen los fundamentos teóricos sobre el funcionamiento de la función *Softmax*, los principios de la computación aproximada y las bases del diseño en alto nivel mediante *Vitis HLS*. En el **Capítulo 3** se describe la metodología seguida para el desarrollo del acelerador, incluyendo las técnicas de aproximación empleadas, la descripción del flujo de diseño y los procesos de validación. Posteriormente, en el **Capítulo 4** se presentan los resultados experimentales, destacando las comparaciones en rendimiento, precisión y uso de recursos. Finalmente, el **Capítulo 5** expone las conclusiones generales del trabajo y plantea posibles líneas de investigación futura.

Capítulo 2

Antecedentes y trabajo relacionado

2.1. Antecedentes

2.1.1. Computación aproximada

La computación aproximada es un enfoque de diseño que busca reducir el consumo de recursos como energía, área o tiempo de cómputo, permitiendo cierto margen de error en los resultados siempre que este no comprometa la funcionalidad general del sistema. Este enfoque resulta especialmente útil en dominios como el aprendizaje automático, donde pequeñas imprecisiones no afectan de manera significativa la calidad percibida o la utilidad del resultado [6].

El objetivo es intercambiar precisión por mejoras sustanciales en eficiencia energética, velocidad y reducción de la complejidad del hardware [10]. Esto puede lograrse en distintos niveles: a nivel de hardware, mediante el diseño de circuitos optimizados; a nivel de software, mediante la simplificación de cálculos o el uso de tipos de datos de menor precisión; y a nivel de sistema, identificando componentes que pueden ser aproximados sin comprometer la funcionalidad global [11].

No obstante, este paradigma también conlleva desafíos importantes. Entre ellos destaca la necesidad de establecer mecanismos rigurosos para medir, acotar y predecir el error introducido por las aproximaciones, con el fin de garantizar que la calidad del resultado permanezca dentro de límites aceptables para la aplicación objetivo. A pesar de estas dificultades, la computación aproximada se perfila como una estrategia prometedora para sustituir o complementar circuitos convencionales en escenarios que demandan altos niveles de eficiencia y rendimiento.

2.1.2. Series de Taylor

En el ámbito del análisis matemático y sus aplicaciones en ingeniería surge la necesidad de simplificar funciones complejas. Muchas funciones que describen fenómenos naturales (como funciones trigonométricas, exponenciales o logarítmicas) son costosas de evaluar computacionalmente. El objetivo principal es aproximar estas funciones utilizando expresiones más simples [12].

Los polinomios son funciones especialmente adecuadas para estos fines, ya que requieren únicamente operaciones aritméticas básicas para su evaluación. Por ello, las series de Taylor constituyen una herramienta fundamental para aproximar funciones complejas mediante polinomios.

La idea central es que si se conoce el valor de una función $f(x)$ y sus derivadas en un punto a, es posible reconstruir la función en torno a dicho punto [13]. La serie de Taylor de $f(x)$ centrada en a se define como:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{2.1}$$

donde $f^{(n)}(a)$ es la n-ésima derivada de f evaluada en a, y $n!$ es el factorial de n.

Expandida, la serie toma la forma:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \dots \tag{2.2}$$

Un caso particular de la serie de Taylor, pero de suma importancia práctica y teórica, es la Serie de Maclaurin. Esta se define simplemente como una serie de Taylor centrada en el punto $a = 0$.

La fórmula de la Serie de Maclaurin es:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!}x^n \tag{2.3}$$

Las series de Maclaurin son fundamentales para poder aproximar funciones trascendentales básicas como e^x, $\sin(x)$ y $\cos(x)$. A continuación se muestra cómo se vería el desarrollo de las funciones mencionadas mediante series de Maclaurin:

- Exponencial: $e^x = 1 + x + \frac{x^2}{2!} + \dots$
- Seno: $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$
- Coseno: $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots$

2.1.3. Interpolación lineal por tramos basada en tablas de búsqueda (LUT)

La interpolación lineal basada en tablas de búsqueda (*Lookup Tables*, LUT) es un método ampliamente utilizado para aproximar funciones matemáticas con alto costo computacional. El procedimiento consiste en muestrear la función y almacenar valores precalculados. Durante la ejecución, el sistema selecciona los puntos más cercanos y realiza interpolación lineal para obtener el valor aproximado [14].

El método almacena puntos equiespaciados de la función y calcula polinomios por tramos. La Figura 2.1 muestra un ejemplo de la aproximación de e^x con ocho muestras dentro del dominio de interés.

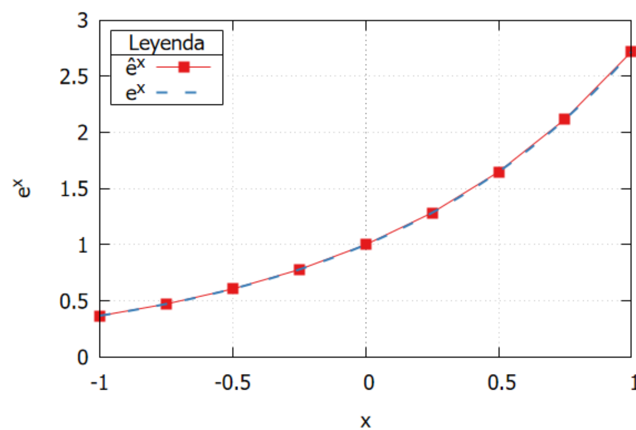

Figura 2.1: Interpolación lineal por LUT para la función exponencial.

Los segmentos de la función por tramos pueden calcularse en tiempo de ejecución (*runtime*) o precalcularse en tiempo de síntesis (o compilación). En una implementación en tiempo de ejecución, la pendiente y la intersección del segmento correspondiente se obtienen a partir de dos puntos consecutivos, según:

$$m_p = \frac{y_{p1} - y_{p0}}{x_{p1} - x_{p0}}, \qquad b_p = y_{p1} - m_p x_{p1} \tag{2.4}$$

y la aproximación se expresa como:

$$f_p(x) = m_p x + b_p, \qquad x_{p0} \leq x \leq x_{p1} \tag{2.5}$$

Este enfoque ofrece un equilibrio favorable entre precisión y eficiencia, ya que reduce la complejidad de cómputo al reemplazar operaciones no lineales por accesos a memoria y cálculos aritméticos básicos. En sistemas embebidos y aceleradores basados en FPGA, el uso de LUTs resulta especialmente ventajoso debido a la disponibilidad de memoria distribuida y la capacidad de realizar operaciones paralelas de manera eficiente. De esta forma, la interpolación lineal sobre LUT se posiciona como una técnica fundamental

en la implementación de funciones aproximadas, incluyendo exponenciales, logaritmos y funciones de activación utilizadas en algoritmos de inteligencia artificial y cómputo aproximado.

Un aspecto fundamental en el diseño de LUTs es el compromiso entre memoria y precisión. Incrementar el número de muestras reduce el error de aproximación, pero aumenta proporcionalmente el tamaño de la LUT, el consumo de memoria BRAM/URAM y la cantidad de accesos requeridos. Por el contrario, LUTs pequeñas reducen significativamente el uso de recursos, aunque introducen errores mayores, especialmente en funciones con alta curvatura.

En arquitecturas para FPGA es común emplear LUTs optimizadas donde, en lugar de almacenar los valores absolutos de cada punto, se almacenan las pendientes e intersecciones de los segmentos, reduciendo el ancho de palabra y, por ende, el uso de memoria. Este enfoque permite equilibrar precisión y área, convirtiendo a las LUTs en una técnica eficaz para aproximar funciones como exponenciales, logaritmos y funciones de activación.

2.1.4. High-Level Synthesis

La síntesis de alto nivel o *High-Level Synthesis* (HLS) se presenta como una metodología de diseño que permite generar hardware digital a partir de descripciones funcionales escritas en lenguajes de alto nivel como C, C++ u OpenCL. A diferencia del diseño tradicional basado en lenguajes de descripción de hardware (HDL) como Verilog o VHDL, HLS introduce un nivel de abstracción más elevado, lo que reduce significativamente la complejidad asociada al diseño de sistemas sobre FPGAs y acorta los tiempos de desarrollo. En lugar de especificar el comportamiento ciclo a ciclo del hardware, el diseñador define la lógica en términos algorítmicos; posteriormente, la herramienta de HLS se encarga de generar una arquitectura equivalente optimizada, incluyendo decisiones sobre paralelismo, planificación y asignación de recursos [15].

Dentro de este paradigma destaca la biblioteca *Vitis HLS* de AMD, una de las plataformas más utilizadas en el ámbito industrial y académico. Vitis HLS permite describir algoritmos en lenguajes de programación convencionales y transformarlos automáticamente en módulos sintetizables para FPGAs. A través de directivas de optimización (pragmas), los desarrolladores pueden guiar el proceso de síntesis para ajustar el rendimiento, el uso de recursos y la latencia, mientras que la herramienta proporciona informes detallados que facilitan un análisis detallado para mejoras en el diseño [16].

El uso de HLS resulta especialmente atractivo en aplicaciones intensivas en cómputo, donde la paralelización a nivel de hardware puede ofrecer importantes incrementos de rendimiento. En particular, algoritmos complejos como los empleados en la estabilización de imágenes, el procesamiento digital de señales o la visión por computador pueden beneficiarse de la capacidad de generar arquitecturas específicas y altamente optimizadas sin necesidad de que el diseñador implemente manualmente cada detalle a nivel de registros. En este sentido, Vitis HLS se posiciona como una herramienta clave para el desarrollo

rápido y eficiente de soluciones escalables basadas en FPGAs, permitiendo alcanzar altos niveles de rendimiento mientras se mantiene un flujo de trabajo accesible para desarrolladores con experiencia previa en programación de alto nivel [17].

Una de las ventajas principales de HLS es la posibilidad de aplicar directivas que transforman significativamente el rendimiento del hardware generado; entre las principales directivas se encuentran las siguientes:

- **Pipeline**: Permite iniciar nuevas iteraciones del algoritmo en cada ciclo de reloj, reduciendo la latencia y maximizando el throughput.
- **Unroll**: Replica hardware interno para procesar múltiples elementos en paralelo. Aumenta el paralelismo pero también el uso de recursos lógicos.
- **Dataflow**: Habilita la ejecución paralela entre funciones o etapas, permitiendo arquitecturas tipo *pipeline* para algoritmos complejos.
- **Interfaces AXI**: HLS permite generar interfaces como AXI4-Lite para control, AXI4-MM para acceso a memoria y AXI-Stream para flujos continuos de datos, facilitando la integración del IP dentro de un sistema más amplio.

2.1.5. Redes neuronales profundas

Las redes neuronales profundas buscan modelar ciertos aspectos del comportamiento del cerebro humano [18]. Están compuestas por neuronas artificiales. Cada neurona recibe entradas $x[n]$ ponderadas por pesos $w[n]$, suma un sesgo b y aplica una función de activación. Como se observa a continuación:

$$y(x) = \sigma\left(\sum_{n=0}^{N-1} x[n]\, w[n] + b\right) \tag{2.6}$$

Las redes neuronales operan en dos fases principales: la fase de entrenamiento y la fase de inferencia. En la fase de entrenamiento, y mediante el uso de un conjunto de datos y una arquitectura previamente definida, se determina el conjunto óptimo de pesos y sesgos mediante algoritmos de optimización. Por otro lado, durante la fase de inferencia o fase de predicción, los parámetros aprendidos se emplean para procesar nuevas entradas y generar resultados. En esta etapa, es común aplicar técnicas como la cuantización, cuyo objetivo es transformar valores originalmente representados en punto flotante hacia formatos enteros más eficientes, sin afectar de manera significativa el desempeño del modelo. De esta forma, la red neuronal utiliza los parámetros ajustados durante el entrenamiento para resolver problemas reales que no han sido previamente vistos [19], [20].

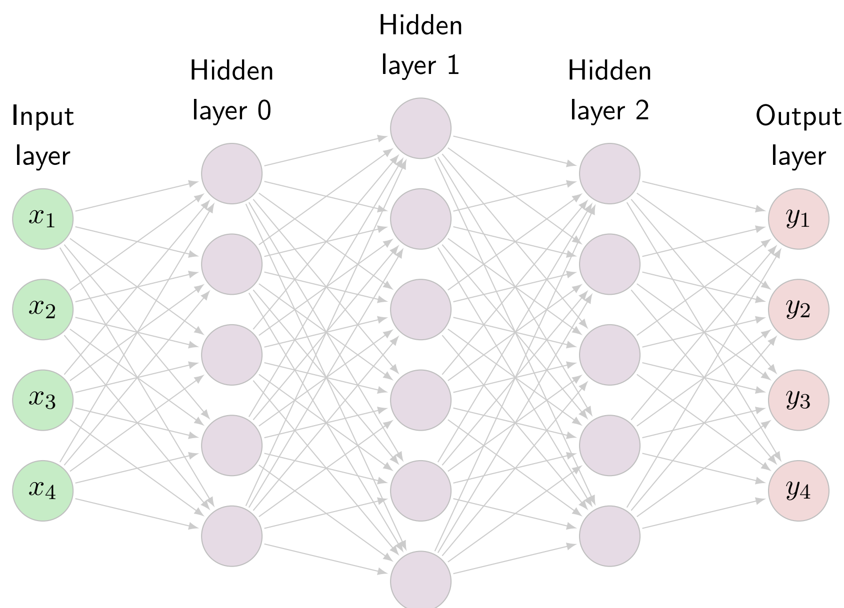

Figura 2.2: Red neuronal profunda. Adaptado de [5].

A partir de la Figura 2.2 se aprecia que esta arquitectura, al igual que una red neuronal, está compuesta por una capa de entrada, capas ocultas y una capa de salida. Los datos son suministrados por un agente externo y llegan primero a la capa de entrada, para luego ser procesados en las capas ocultas o intermedias. Cabe destacar que, cuando una red neuronal incorpora un número elevado de capas ocultas (más de tres capas), se clasifica como una DNN (*Deep Neural Network*).

Dentro de las arquitecturas neuronales, la función Softmax ocupa un papel fundamental como etapa final de clasificación. Sin embargo, su implementación directa presenta varios desafíos relevantes en hardware:

- **Costo computacional elevado**: la función requiere evaluar exponenciales, sumatorias acumulativas y divisiones, operaciones costosas en FPGA.
- **Riesgo de desbordamiento numérico**: para valores grandes de entrada, e^x puede crecer rápidamente y saturar el rango disponible dependiendo del tipo de dato que se utilice o también de la precisión que ofrezca el tipo de dato utilizado para representar esta función.
- **Sensibilidad a precisión**: pequeños errores en los exponenciales o en la sumatoria pueden degradar significativamente la distribución final.

Por ello, la función *softmax* es una candidata natural para técnicas de computación aproximada y evaluación cuantitativa ante distintos escenarios [4], [21].

2.1.6. CuFP: An HLS Library for Customized Floating-Point Operators

Existe una librería llamada *CuFP: An HLS Library for Customized Floating-Point Operators* que implementa operaciones aritméticas en punto flotante utilizando formatos personalizados. En esta librería se encuentran algunas operaciones, como es el ejemplo de la suma y de la multiplicación; estas siguen un conjunto bien definido de etapas. En la suma, primero se alinean los exponentes desplazando la mantisa del operando con exponente menor, para luego sumar o restar las mantisas ya alineadas. En la multiplicación, las mantisas se multiplican directamente y los exponentes se suman, obteniendo un valor preliminar que requiere normalización.

Dentro de esta librería se implementan, entre otras, dos funciones de interés, `mul()` y `sum()`, que operan sobre una clase de punto flotante personalizado denominada `CustomFloat` [22].

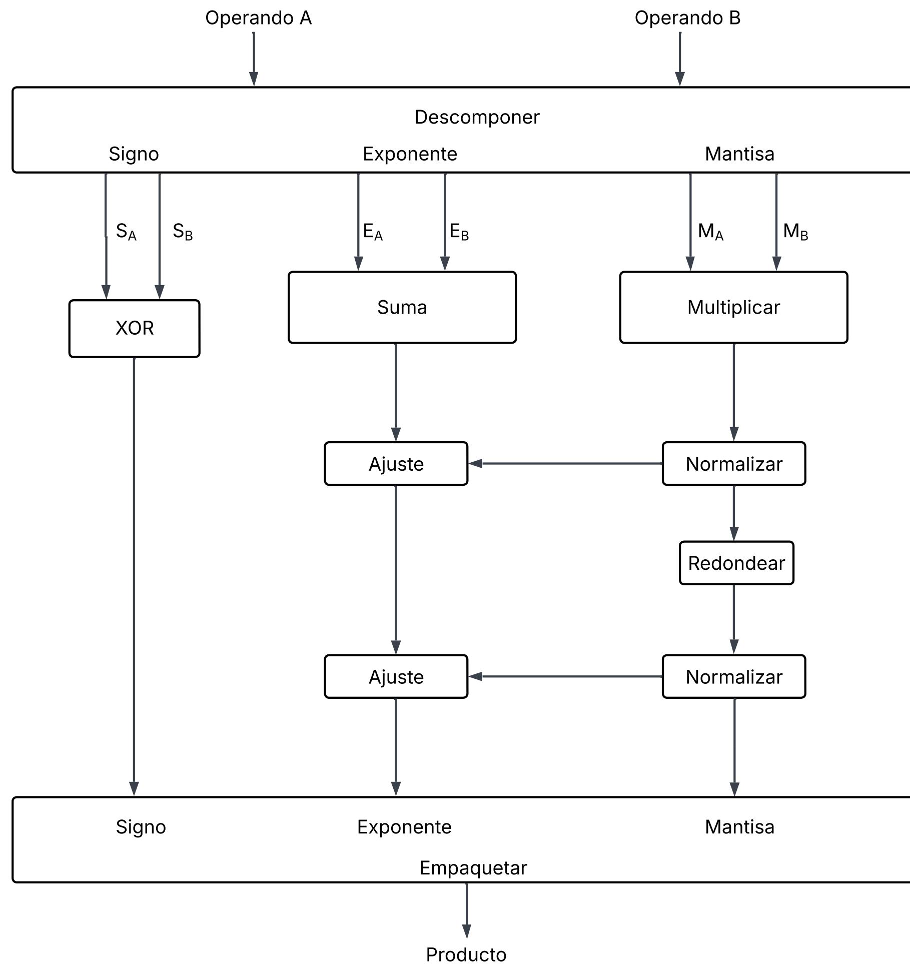

Figura 2.3: Diagrama de bloques de la multiplicación. Adaptado de [22].

En la operación `mul()` inicia multiplicando las mantisas de los operandos, generando

un resultado intermedio de mayor tamaño. Luego aplica un proceso de normalización y redondeo, ajusta el exponente sumando los exponentes originales y cualquier bit de desbordamiento, determina el signo mediante una operación XOR y finalmente construye un nuevo objeto `CustomFloat`. Un diagrama general de este proceso se muestra en la Figura 2.3.

De igual manera, la función `sum()` desarrolla la operación de suma para este formato personalizado. El procedimiento comienza identificando el exponente mayor, que se adopta como referencia, y alineando la mantisa del operando con exponente menor mediante desplazamientos. Posteriormente se extienden ambas mantisas a un mismo ancho, se aplican los signos correspondientes y se realiza la suma o resta. Tras ello, el resultado se normaliza localizando el bit más significativo o gestionando un posible desbordamiento, ajustando el exponente según el desplazamiento aplicado y realizando el redondeo si está habilitado. Finalmente, se reconstruye el número resultante en formato `CustomFloat`. Un diagrama general de este proceso se muestra en la Figura 2.4.

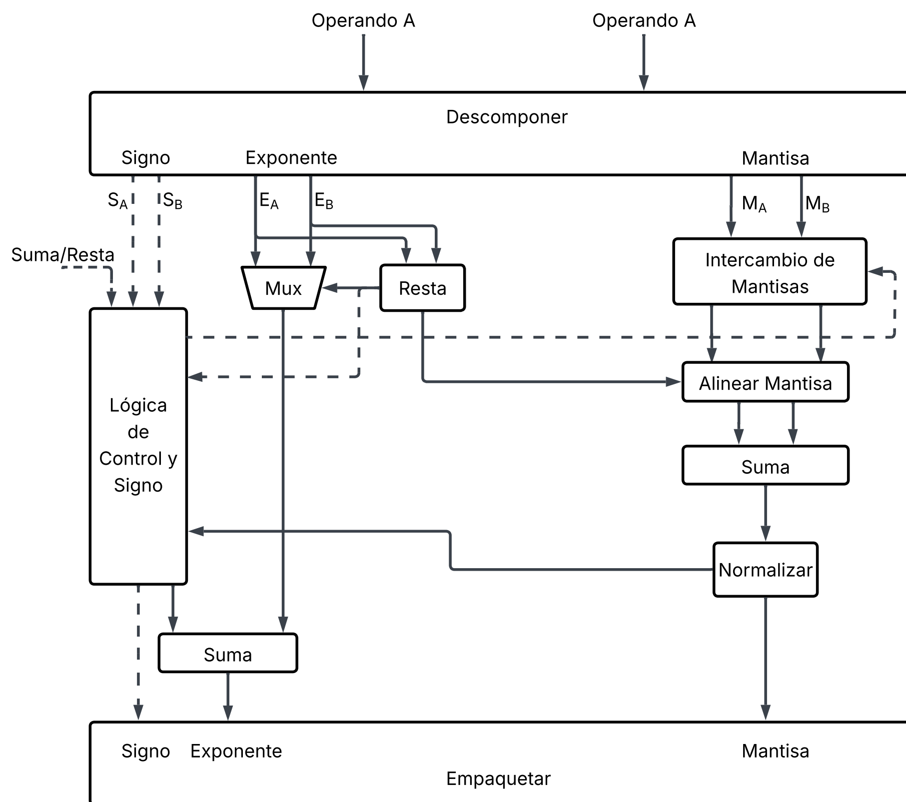

Figura 2.4: Diagrama de bloques de la suma/resta. Adaptado de [22].

En dicha librería se destaca que, a diferencia del punto fijo, los números en punto flotante ofrecen un rango dinámico considerablemente mayor, ya que permiten desplazar la posición del punto binario según el valor representado. Sin embargo, también se enfatiza que esta flexibilidad incrementa la complejidad de las operaciones, pues la mayoría requiere alinear exponentes, desempaquetar los operandos en sus valores internos antes del cálculo

y normalizar el resultado antes de almacenarlo nuevamente [22].

2.1.7. Template specialization y metaprogramación

La metaprogramación es una técnica de programación que permite trasladar parte de la lógica de un programa desde la etapa de ejecución hacia la etapa de compilación. Este enfoque resulta especialmente útil en aplicaciones donde el rendimiento es crítico, ya que posibilita generar código especializado y optimizado antes de que el programa sea ejecutado. En C++, este tipo de programación se basa principalmente en el uso de plantillas, las cuales ofrecen un mecanismo poderoso para construir componentes genéricos capaces de adaptarse a distintos tipos de datos sin sacrificar eficiencia. Al tomar decisiones estructurales en tiempo de compilación, la metaprogramación contribuye a reducir tiempos de ejecución y mejorar la calidad del código generado [23].

Un ejemplo representativo de metaprogramación es la especialización de plantillas (*template specialization*). De forma análoga al concepto de herencia en la programación orientada a objetos, que permite la creación de clases derivadas especializadas, la especialización de plantillas habilita la definición de versiones concretas de una plantilla genérica basadas en patrones de tipos o valores estáticos. En el momento de instanciación, el compilador selecciona automáticamente la versión más adecuada, ya sea la implementación primaria o una de sus especializaciones. Esto posibilita la definición de comportamientos completamente diferenciados sin necesidad de estructuras condicionales en tiempo de ejecución.

Listado 2.1 Estructura de Especialización de Plantillas

```
1:  // 1. Plantilla Base (Caso por defecto: Error)
2:  template <class T, bool decision
3:  class className : ErrorSpecializationNotFound<T>{};
4:  // 2. Especialización para primer caso(true)
5:  template <class T>
6:  class className<T, TRUE>{
7:      [...] ▷ Código para el caso 1
8:  };
9:  // 3. Especialización para segundo caso(false)
10: template <class T>
11: class className<T,FALSE>{
12:     [...] ▷ Código para el caso 2
13: };
```

En la estructura mostrada en el Listado 2.1 se aprecia claramente este mecanismo. Primero se declara una plantilla base que actúa como caso por defecto cuando no existe una especialización válida. A continuación, se incluyen dos especializaciones distintas, cada una asociada a un valor estático que determina la selección de la implementación correspondiente. Este diseño permite implementar funcionalidades específicas para distintos tipos o condiciones en tiempo de compilación, manteniendo un código eficiente.

Capítulo 3

Desarrollo de acelerador de la función Softmax con soporte de datos punto fijo y punto flotante de precisión arbitraria

En este capítulo se describe la metodología empleada para el diseño e implementación de un acelerador de la función *softmax* con soporte para datos en punto fijo y punto flotante de precisión arbitraria. Se detalla el flujo completo seguido durante el proceso de desarrollo, iniciando con una implementación base en `C++`, su posterior porteo hacia *High-Level Synthesis* (HLS), y finalmente la incorporación de métodos de computación aproximada para el cálculo eficiente de la exponencial [21].

La estructura de este capítulo se organiza de la siguiente manera: primero se expone la implementación base en `C++` de la ecuación (3.1), necesaria para entender el comportamiento matemático de la función. Luego, se detalla la metodología para migrar esta implementación hacia una arquitectura sintetizable en HLS. Posteriormente, se desarrollan dos aproximaciones para el cálculo de la exponencial: una basada en Series de Taylor y otra en interpolación lineal mediante tablas de búsqueda (*Lookup Tables*, LUT). Finalmente, se explican las configuraciones empleadas en *Vivado HLS* para evaluar rendimiento, latencia y utilización de recursos (LUTs, FFs, BRAMs y DSPs).

3.1. Implementación base en C++ y porteo a HLS

Para iniciar con el diseño del acelerador, primeramente, se desarrolló la función softmax en `C++`, para lo cual se utilizó la siguiente ecuación [18]:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{3.1}$$

De esta ecuación se extrae el siguiente algoritmo necesario para obtener los resultados:

Algoritmo 3.1 Algoritmo para la función Softmax

```
1: def softmax(x, n):
2:    sum = 0.0                          ▷ Inicializar suma
3:    Para (i = 0; i <n; i = i + 1):     ▷ 1. Calcular el denominador (suma)
4:      sum = sum + exp(x[i])
5:    S = nuevo_vector(n)                ▷ Crear vector de resultados
6:    Para (i = 0; i <n; i = i + 1):     ▷ 2. Calcular valores finales
7:      S[i] = exp(x[i]) / sum
8:    return S
```

Las funciones no lineales como la exponencial representan una alta complejidad al momento de realizar circuitos que hagan sus cálculos; por ello se buscan otras alternativas que reduzcan la complejidad computacional que estas representan [5].

Para realizar el porteo a HLS primeramente se realiza un análisis de entradas y salidas que tiene el acelerador; posteriormente se divide el acelerador en 3 funciones principales:

1. **Módulo de entrada:** Responsable de leer datos desde memoria y almacenarlos en un `hls::stream`.
2. **Módulo principal de cómputo:** Encargado de calcular la función *softmax*.
3. **Módulo de salida:** Escribe los resultados nuevamente en memoria externa.

Esta división permite aplicar pragmas de optimización como *pipeline*, *dataflow*, *unroll* y la configuración de interfaces AXI para maximizar paralelismo y reducir latencia [16].

Para la etapa de entrada se presenta el Algoritmo 3.2, en el cual se realiza la lectura del vector de entrada y se almacena en un `hls::stream`, el cual almacena los bits de los datos de entrada para posteriormente ser procesados en la función de softmax.

Algoritmo 3.2 Carga de Datos de Memoria a Stream

```
1: def load_input(in_vector, stream, size):
2:    Para (iter = 0; iter <2; iter++):
3:      Para (i = 0; i <size; i++):
4:        dato = in_vector[i]
5:        write(stream, dato)
6:    fin Para
7: return
```

El Algoritmo 3.3 muestra cómo fue diseñada la función de softmax en HLS. Dicho algoritmo se compone de dos ciclos principales: el primero para la lectura de datos y el cálculo de la sumatoria de las exponenciales, donde también se calcula un factor de escala correspondiente al recíproco ($scale = 1/(Sum)$) de la sumatoria total de las exponenciales; y

finalmente, en el segundo ciclo, se calculan los valores de salida calculando la exponencial de cada elemento del vector de entrada y multiplicándola por el resultado del recíproco de la sumatoria. Esto se realiza con el fin de minimizar operaciones como la división, ya que esta es muy demandante computacionalmente, lo que ocasionaría problemas como cuellos de botella o un incremento en la latencia o área del acelerador [4].

Algoritmo 3.3 Algoritmo de Aproximación y Normalización

```
1:  def compute_aprox(in_stream, out_stream, size):
2:     sum = 0.0
3:
4:     // Fase 1: Acumulación
5:     Para (i = 0; i <size; i++):
6:       val = read(in_stream)
7:       exp_val = exp(val)
8:       sum += exp_val
9:
10:    // Fase 2: Factor de escala
11:    scale = 1.0 / sum
12:
13:    // Fase 3: Normalización
14:    Para (i = 0; i <size; i++):
15:      val = read(in_stream)
16:      res = exp(val) * scale
17:      write(out_stream, res)
18: return
```

El Algoritmo 3.4 muestra el módulo de escritura del `hls::stream` a la memoria donde se almacenarán los resultados de salida obtenidos. En este algoritmo se leen los elementos que salen de la función de cómputo y se almacenan en el arreglo de salida del acelerador.

Algoritmo 3.4 Escritura de Stream a Memoria

```
1: def store_result(out_array, stream, size):
2:    Para (i = 0; i <num_elements; i++):
3:      dato = read(stream)
4:      out_array[i] = dato
5:    fin Para
6: return
```

3.2. Aproximación de exponencial por medio de Series de Taylor

Como se describe en el Algoritmo 3.3, el cálculo de la función *softmax* depende directamente de la evaluación de la función exponencial, la cual representa una de las operaciones más exigentes en términos de recursos dentro de una arquitectura digital. Esto se debe a la necesidad de realizar múltiples multiplicaciones, divisiones y, en algunos casos, utilizar hardware especializado para funciones trascendentales. Ante esta complejidad, se exploraron técnicas de aproximación que permitieran reproducir el comportamiento de la función exponencial con una precisión adecuada, pero con una reducción significativa en latencia y consumo de recursos [12].

Las Series de Taylor se presentan como una solución ideal, ya que estas permiten representar funciones no lineales mediante polinomios, lo cual resulta especialmente ventajoso en entornos de diseño con *High-Level Synthesis* (HLS), ya que los polinomios pueden implementarse utilizando operaciones aritméticas básicas como sumas y multiplicaciones, sin requerir bloques funcionales complejos. En particular, se emplea la expansión en serie de Maclaurin, que corresponde al caso especial de la Serie de Taylor centrada en $x_0 = 0$, y se expresa como:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

Esta elección se justifica por dos razones principales: primero, los coeficientes de la serie son constantes e independientes del punto de evaluación, lo que simplifica su implementación; segundo, el rango de entrada del acelerador puede cubrirse adecuadamente con un orden de serie suficientemente alto, sin necesidad de realizar desplazamientos o reescalamientos adicionales.

Para llevar esta aproximación al hardware, se tomó como referencia la implementación disponible en la biblioteca *FAL* [9], la cual utiliza una versión iterativa de la Serie de Taylor. Esta estrategia permite evaluar el polinomio de forma incremental, reutilizando resultados intermedios y evitando cálculos redundantes, lo que se traduce en una implementación más eficiente en términos de área y ciclos de reloj.

El Algoritmo 3.5, extraído de *FAL*, muestra cómo se calcula la aproximación polinomial de orden O para e^x. Su eficiencia radica en dos aspectos clave: el cálculo progresivo de las potencias de x y la actualización recursiva del factorial, lo que evita recomputaciones innecesarias y reduce el uso de recursos lógicos.

El parámetro O, que define el orden de la serie, tiene un impacto directo en la precisión del resultado. A mayor orden, mayor exactitud, pero también mayor complejidad computacional. En hardware, esto se traduce en un incremento tanto de la latencia como del consumo de recursos. Por tanto, la elección del orden óptimo implica un balance entre precisión y eficiencia.

Algoritmo 3.5 Series de Taylor para e^x

```
1: def taylor_series(x, O):
2:     sum = 1.0
3:     num = 1.0
4:     den = 1.0
5:     Para (i = 1; i <= O; i++):
6:         num = num * x          ▷ x^k
7:         den = den / i          ▷ Factorial k!
8:         sum = sum + (num * den)
9:     return sum
```

En etapas posteriores de este trabajo, se adaptará la implementación para soportar distintos formatos numéricos, incluyendo representaciones en punto fijo y punto flotante con precisión arbitraria. Esto permitirá analizar cómo la elección del tipo de dato afecta la precisión de la aproximación, la estabilidad del cálculo y el uso de recursos.

3.3. Aproximación de exponencial por medio de Interpolación Lineal por LUT

Otro método empleado para aproximar la función exponencial dentro del acelerador es la Interpolación Lineal mediante Tablas de Búsqueda (LUTs). Al igual que en la aproximación por Series de Taylor, este método busca reducir la complejidad computacional y el uso de recursos al sustituir operaciones costosas como la evaluación directa de la función exponencial por consultas a una tabla precomputada y operaciones lineales simples [14].

En la librería *FAL* [9] se dispone de una implementación base de este enfoque, la cual fue utilizada como punto de partida en el diseño del acelerador. A diferencia de la aproximación mediante Series de Taylor, donde la precisión depende del orden O de la serie, en la aproximación por LUT la exactitud está determinada por el número de puntos almacenados y el tamaño del paso entre ellos. De esta forma, al incrementar la resolución de la tabla se mejora la precisión a costa de aumentar el uso de memoria.

Con el fin de optimizar la latencia en hardware y reducir la complejidad aritmética durante la ejecución, en este trabajo se propone almacenar durante la fase de síntesis datos como las pendientes e intersecciones asociadas a cada intervalo de la tabla. Esta estrategia elimina la necesidad de realizar divisiones adicionales y reduce el número de multiplicaciones requeridas, acortando el tiempo entre la operación de búsqueda y el cómputo del valor aproximado.

A continuación, se muestra el Algoritmo 3.6, basado en la implementación de la librería *FAL*, que describe el proceso de aproximación de la función exponencial mediante Interpolación Lineal.

Algoritmo 3.6 Interpolación Lineal para aproximación de e^x mediante LUT

```
1: def linear_interpolation(x):
2:     // — 1. Parámetros de la LUT —
3:     begin = LUT::Minimum ▷ Valor mínimo representado en la tabla
4:     step = LUT::Step ▷ Tamaño del incremento entre puntos
5:     kPoints = LUT::Points ▷ Número total de puntos en la tabla
6:     lut[] = generar_lut() ▷ Tabla de valores precomputados
7:
8:     // 2. Cálculo del índice base
9:     i_lower = (x - begin) / step ▷ Índice del punto inferior
10:
11:     // 3. Verificación de límites
12:     si (i_lower <= 0):
13:       i_lower = 0
14:     si (i_lower >= kPoints - 1):
15:       i_lower = kPoints - 2
16:     i_upper = i_lower + 1 ▷ Índice superior correspondiente
17:
18:     // 4. Obtención de valores de la LUT
19:     x_lower = begin + (i_lower * step) ▷ Valor x1 del intervalo
20:     y_lower = lut[i_lower] ▷ Valor y1
21:     y_upper = lut[i_upper] ▷ Valor y2
22:
23:     // 5. Interpolación Lineal
24:     resultado = y_lower + ((x - x_lower) / step) * (y_upper - y_lower)
▷ y1 + (x − x1)(y2−y1)/step
25:     return resultado
```

El Algoritmo 3.6 recibe como entrada el valor a evaluar y utiliza los parámetros de configuración de la tabla, tales como el valor mínimo, el tamaño del paso y el número de puntos, para determinar el segmento al que pertenece dicho valor. Posteriormente, aplica una operación de interpolación lineal entre los dos valores más cercanos de la tabla, produciendo así una aproximación eficiente de la función exponencial.

El Algoritmo 3.7 muestra el procedimiento utilizado para calcular los valores de la Tabla de Búsqueda (LUT) durante la fase de compilación. La generación de esta tabla se realiza de manera completamente estática con el objetivo de evitar cualquier costo computacional asociado a la evaluación de la función exponencial en tiempo de ejecución. Para ello, se utiliza la función estándar `std::exp()`, la cual proporciona una evaluación precisa y determinista de e^x para cada uno de los puntos definidos dentro del dominio de aproximación.

El proceso de inicialización de la LUT depende directamente de tres parámetros fundamentales. El primero es la cantidad total de puntos almacenados en la tabla, que determina

Algoritmo 3.7 Inicialización de Tabla de Búsqueda (LUT)

```
1: def init_lut():
2:     LUT = array[N]
3:     Para (i = 0; i <N; i++):
4:         val_x = MIN_VAL + (i * STEP)
5:         LUT[i] = exp(val_x)
6:     fin Para
7: return LUT
```

la resolución de la aproximación. El segundo es el intervalo o rango del dominio en el cual se desea aproximar la función, el cual se divide de manera uniforme según el número de puntos establecidos. Por último, el tercer parámetro corresponde al tamaño del paso (*step*) entre valores consecutivos, el cual se calcula a partir del dominio e influye de manera directa tanto en la precisión de la aproximación como en el tamaño final de la LUT.

En etapas posteriores del desarrollo, este algoritmo será adaptado para operar tanto con datos en punto fijo como punto flotante, con el fin de evaluar su impacto en la precisión, latencia y uso de recursos del acelerador.

3.4. Implementación de soporte de datos para punto flotante de precisión arbitraria

Con el objetivo de brindar al acelerador una mayor flexibilidad y adaptabilidad frente a distintos escenarios de aplicación, se incorporó soporte para operaciones en punto flotante. Para ello, se tomó como base la biblioteca *CuFP: An HLS Library for Customized Floating-Point Operators*, la cual proporciona una implementación versátil de operadores en punto flotante con precisión configurable [22]. Esta biblioteca, desarrollada para entornos de diseño mediante *High-Level Synthesis* (HLS), incluye un conjunto de operaciones fundamentales como suma, resta, multiplicación de matrices y productos escalares, que fueron utilizadas para el desarrollo del acelerador.

El código fuente de *CuFP* sirvió como punto de partida para extender el acelerador de la función *softmax* con soporte para punto flotante. Para lograr una implementación eficiente, se recurrió al uso de metaprogramación en C++. En particular, se empleó la utilidad `std::conditional` para determinar el tipo de dato en tiempo de compilación [23], como se ilustra en la Figura 3.1. Esta técnica permite seleccionar entre una representación en punto flotante o punto fijo de manera automática.

Para lograr esta flexibilidad, se introdujo una variable de tipo booleana denominada `is_floating_point`, la cual determina el tipo de dato a utilizar. Cuando esta variable toma el valor verdadero, las variables internas de los algoritmos se definen en punto flotante; en caso contrario, se emplean tipos en punto fijo. A partir de esta definición,

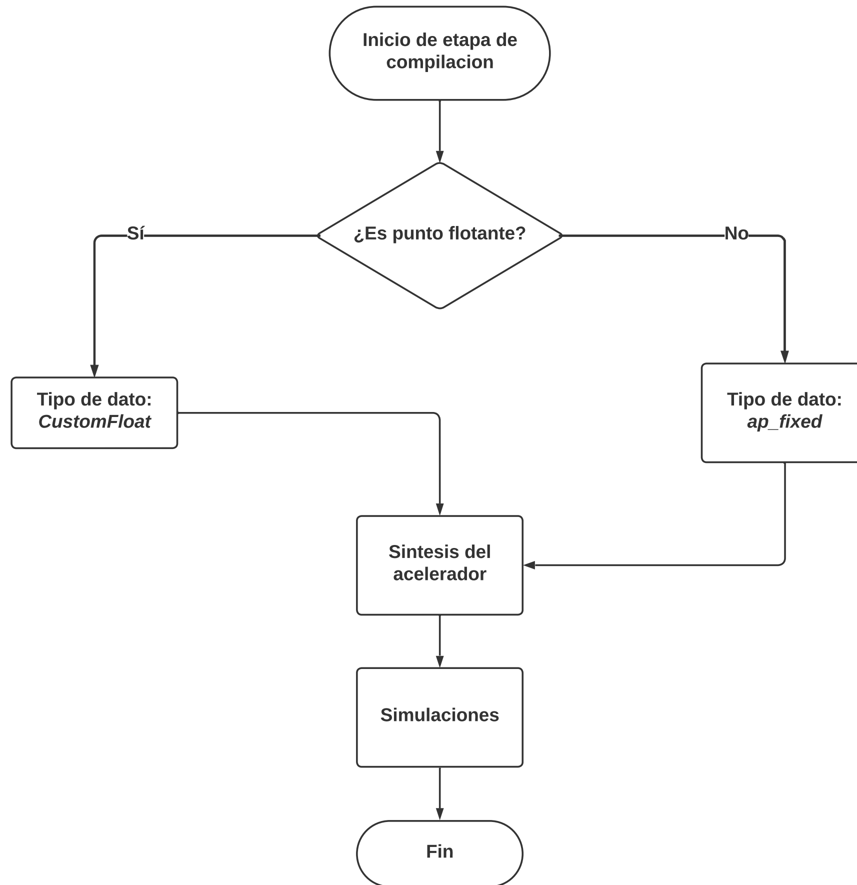

Figura 3.1: Diagrama de flujo para la determinación del tipo de dato en tiempo de compilación.

los algoritmos fueron configurados utilizando la técnica de *template specialization*, permitiendo adaptar automáticamente su comportamiento al tipo de dato seleccionado sin necesidad de modificar manualmente el código fuente o hacerlo en tiempo de ejecución.

En el caso del Algoritmo 3.5, utilizado para la aproximación de la función exponencial mediante Series de Taylor, originalmente se soportaban únicamente tipos de datos en punto fijo. Con la incorporación de esta estrategia, el algoritmo fue extendido para aceptar tanto punto fijo como punto flotante, ajustando internamente las variables y operaciones al tipo de dato especificado en tiempo de compilación mediante *template specialization*.

De forma complementaria, se extendió también el Algoritmo 3.6, encargado de realizar interpolaciones lineales sobre una tabla de búsqueda (LUT). Originalmente, este algoritmo recibía como parámetros la cantidad de puntos de la LUT, así como los valores de inicio y fin del dominio de aproximación. Con la incorporación de *template specialization*, se añadió como nuevo parámetro de entrada la variable *is_floating_point*, permitiendo adaptar dinámicamente el comportamiento del algoritmo; cuando su valor es verdadero, las variables internas se definen en punto flotante; en caso contrario, se utilizan representacio-

nes en punto fijo. A partir de esta configuración, el algoritmo se ajusta automáticamente al tipo de dato requerido, mejorando la reutilización del código y facilitando la exploración de configuraciones de precisión en función de los requisitos específicos de cada aplicación.

3.5. Generación del banco de pruebas

Con el fin de evaluar de manera exhaustiva el comportamiento del acelerador desarrollado, se diseñó un banco de pruebas que abarca una amplia variedad de configuraciones representativas de distintos escenarios operativos. Este conjunto de evaluaciones considera parámetros fundamentales del diseño, tales como el ancho del dato, el orden de las aproximaciones utilizadas para Series de Taylor, la cantidad de muestras en las *LookUp Tables*, las dimensiones del vector procesado y la frecuencia de operación.

La selección de estos parámetros permite observar el efecto de cambios arquitectónicos y variaciones en la carga de trabajo sobre el desempeño del acelerador, garantizando un análisis completo tanto para los métodos basados en Series de Taylor como para la aproximación mediante tablas de búsqueda (LUTs), en formatos de punto flotante y punto fijo. A continuación, se presentan las tablas que resumen las configuraciones empleadas para cada uno de los métodos evaluados.

Para la evaluación de la aproximación mediante Series de Taylor utilizando la configuración de punto flotante (*CuFP*), se estableció un conjunto exhaustivo de configuraciones orientadas a caracterizar y analizar el acelerador bajo distintas condiciones de precisión y carga computacional. La Tabla 3.8 sintetiza dichas combinaciones, abarcando variaciones en el ancho del dato, la longitud de la mantisa, el orden de la serie de Taylor, el tamaño del vector de entrada y la frecuencia de operación. Esta metodología permite analizar de manera exhaustiva la relación entre precisión numérica, estabilidad del cálculo y rendimiento hardware, permitiendo así un análisis detallado del acelerador.

Tabla 3.8: Configuración del Banco de Pruebas para la evaluación del núcleo de Punto Flotante (CuFP) utilizando aproximación por Series de Taylor.

Análisis	**Ancho** (W)	**Mantisa** (M)	**Orden Taylor**	**Vector** (N)	F_{clk} **(MHz)**
Precisión	4	1	1	1024	200
	6	3	1	1024	200
	8	3	1	1024	200
	10	4	1	1024	200
	12	6	1	1024	200
	14	8	1	1024	200
	16	10	1	1024	200
	32	23	1	1024	200
Orden de Taylor	16	10	1	1024	200
	16	10	2	1024	200
	16	10	3	1024	200
Frecuencia	16	10	2	512	100
	16	10	2	512	150
	16	10	2	512	200
	16	10	2	512	250
	16	10	2	512	300
Dimensión del vector	16	10	2	128	200
	16	10	2	256	200
	16	10	2	512	200
	16	10	2	1024	200

Para el método de aproximación basado en tablas de búsqueda (LUTs) aplicando aritmética de punto flotante, se definió un conjunto estructurado de configuraciones orientado a estudiar el impacto de la densidad de muestreo (*samples* o número de puntos), el formato del dato y la carga de procesamiento. La Tabla 3.9 resume estas combinaciones, incluyendo variaciones en el ancho del dato, la mantisa, el número de puntos almacenados en la tabla, el tamaño del vector y la frecuencia objetivo. Este conjunto experimental permite analizar el funcionamiento del núcleo bajo diferentes niveles de demanda computacional y condiciones de operación.

Tabla 3.9: Configuración del Banco de Pruebas para la evaluación del núcleo de Punto Flotante (CuFP) utilizando aproximación basada en LUTs.

Análisis	**Ancho (W)**	**Mantisa (M)**	**Puntos / Samples**	**Vector (N)**	F_{clk} **(MHz)**
Precisión	4	1	8	1024	200
	6	2	4	1024	200
	8	3	4	1024	200
	10	4	8	1024	200
	12	6	8	1024	200
	14	8	8	1024	200
	16	10	8	1024	200
	32	23	8	1024	200
Número de puntos	16	10	8	512	200
	16	10	16	512	200
	16	10	32	512	200
Frecuencia	16	10	8	512	100
	16	10	8	512	150
	16	10	8	512	200
	16	10	8	512	250
	16	10	8	512	300
Dimensión del vector	16	10	8	128	200
	16	10	8	256	200
	16	10	8	512	200
	16	10	8	1024	200

En el método basado en Series de Taylor implementado en aritmética de punto fijo se definió un conjunto de configuraciones resumidas en la Tabla 3.10. Al igual que en las demás configuraciones, se consideran variaciones en el ancho total del dato, los bits asignados a la parte entera, el orden de la serie, la dimensión del vector y la frecuencia de reloj. Este banco de pruebas permite analizar el desempeño del núcleo en múltiples condiciones operativas.

Tabla 3.10: Configuración del Banco de Pruebas para la evaluación de la aproximación mediante Series de Taylor en aritmética de Punto Fijo.

Análisis	**Ancho** (W)	**Bits Enteros** (I)	**Orden Taylor**	**Vector** (N)	F_{clk} **(MHz)**
Precisión	4	2	3	1024	200
	6	2	3	1024	200
	8	4	3	1024	200
	10	4	3	1024	200
	12	6	3	1024	200
	14	6	3	1024	200
	16	6	3	1024	200
	32	16	3	1024	200
Orden de Taylor	16	6	1	512	200
	16	6	2	512	200
	16	6	3	512	200
Frecuencia	16	6	2	512	100
	16	6	2	512	150
	16	6	2	512	200
	16	6	2	512	250
	16	6	2	512	300
Dimensión del vector	16	6	2	128	200
	16	6	2	256	200
	16	6	2	512	200
	16	6	2	1024	200

La Tabla 3.11 presenta las combinaciones seleccionadas para la aproximación mediante tablas de búsqueda en punto fijo, que incluyen variaciones en el ancho total, los bits enteros, el número de puntos de la LUT, la dimensión del vector y la frecuencia de operación.

Tabla 3.11: Configuración del Banco de Pruebas para la evaluación de la aproximación mediante LUTs en aritmética de Punto Fijo.

Análisis	**Ancho Total** (W)	**Bits Enteros** (I)	**Puntos / Samples**	**Vector** (N)	F_{clk} **(MHz)**
Precisión	4	2	8	1024	200
	6	2	8	1024	200
	8	4	8	1024	200
	10	4	8	1024	200
	12	6	8	1024	200
	14	6	8	1024	200
	16	6	8	1024	200
	32	8	8	1024	200
Cantidad de puntos	16	6	8	512	200
	16	6	16	512	200
	16	6	32	512	200
Frecuencia	16	6	8	512	100
	16	6	8	512	150
	16	6	8	512	200
	16	6	8	512	250
	16	6	8	512	300
Dimensión del vector	16	6	8	128	200
	16	6	8	256	200
	16	6	8	512	200
	16	6	8	1024	200

Con el fin de realizar comparaciones de precisión y analizar el tiempo de ejecución entre las aproximaciones y la implementación exacta basada en `hls::exp()`, se definió un conjunto de configuraciones para distintos anchos de datos. La Tabla 3.12 resume estas combinaciones, manteniendo constante el tamaño del vector y la frecuencia de operación para facilitar la comparación directa entre escenarios.

Tabla 3.12: Configuración del Banco de Pruebas para la evaluación de la implementación Exacta (`hls::exp()`).

Análisis	**Ancho (W)**	**Bits Enteros (I)**	**Vector (N)**	F_{clk} **(MHz)**
Precisión	4	2	1024	200
	6	2	1024	200
	8	4	1024	200
	10	4	1024	200
	12	6	1024	200
	14	6	1024	200
	16	6	1024	200
	32	16	1024	200

Finalmente, con el fin de realizar comparaciones de calidad de resultados, se propone un caso de análisis basado en un escenario representativo para el acelerador. Este consiste en utilizar un tipo de dato cuyo ancho sea compatible con una red neuronal de baja precisión, como LeNet-5, la cual opera con un formato de 12 bits [20]. Bajo esta configuración, se realizaron pruebas tanto para las versiones en punto fijo como en punto flotante de cada acelerador y, posteriormente, sus resultados fueron comparados con la versión exacta de la función *Softmax*. Las configuraciones consideradas para este conjunto de experimentos se muestran en las Tablas 3.13 y 3.14.

Tabla 3.13: Configuraciones para LeNet (12 bits, punto fijo)

Configuración	BW	Parte entera	Vector	Orden Taylor	Puntos LUT
Taylor orden 1	12	6	10	1	–
Taylor orden 2	12	6	10	2	–
Taylor orden 3	12	6	10	3	–
LUT (8 pts)	12	6	10	–	8
LUT (16 pts)	12	6	10	–	16
LUT (32 pts)	12	6	10	–	32
Versión exacta	12	6	10	–	–

Tabla 3.14: Configuraciones para LeNet (12 bits, punto flotante)

Configuración	BW	Mantisa	Vector	Orden Taylor	Puntos LUT
Taylor orden 1	12	6	10	1	–
Taylor orden 2	12	6	10	2	–
Taylor orden 3	12	6	10	3	–
LUT (8 pts)	12	6	10	–	8
LUT (16 pts)	12	6	10	–	16
LUT (32 pts)	12	6	10	–	32
Versión exacta	12	6	10	–	–

Además del caso anterior, se propone estudiar el comportamiento del acelerador bajo el estándar IEEE 754 *Half-Precision* de 16 bits. Este formato es ampliamente utilizado en redes neuronales modernas gracias a su balance entre precisión y costo computacional [7]. Para este caso, también se realiza un análisis de calidad de resultados y del error asociado para cada aproximación, comparando los métodos de Taylor, LUTs y la versión exacta de la función Softmax. Las configuraciones utilizadas para este estudio se resumen en las Tablas 3.15 y 3.16.

Tabla 3.15: Configuraciones para IEEE Half (16 bits, punto fijo)

Configuración	BW	Parte entera	Vector	Orden Taylor	Puntos LUT
Taylor orden 1	16	10	10	1	–
Taylor orden 2	16	10	10	2	–
Taylor orden 3	16	10	10	3	–
LUT (8 pts)	16	10	10	–	8
LUT (16 pts)	16	10	10	–	16
LUT (32 pts)	16	10	10	–	32
Versión exacta	16	10	10	–	–

Tabla 3.16: Configuraciones para IEEE Half (16 bits, punto flotante)

Configuración	BW	Mantisa	Vector	Orden Taylor	Puntos LUT
Taylor orden 1	16	10	10	1	–
Taylor orden 2	16	10	10	2	–
Taylor orden 3	16	10	10	3	–
LUT (8 pts)	16	10	10	–	8
LUT (16 pts)	16	10	10	–	16
LUT (32 pts)	16	10	10	–	32
Versión exacta	16	10	10	–	–

Capítulo 4

Resultados y análisis

En este capítulo se presentan los resultados obtenidos durante la evaluación del acelerador de la función *softmax*, considerando distintos métodos de aproximación, configuraciones de ancho de palabra y arquitecturas numéricas tanto en punto fijo como en punto flotante. El análisis se realiza tomando como referencia los recursos disponibles de una tarjeta *AMD Kria KV260* [24].

El propósito de este capítulo es comparar el comportamiento de las distintas arquitecturas propuestas en términos de utilización de recursos, latencia, impacto del tamaño de vector, sensibilidad a la frecuencia de reloj y calidad numérica de los resultados. De esta manera es posible identificar las configuraciones que ofrecen un equilibrio favorable entre precisión, eficiencia y consumo de hardware.

Las secciones siguientes detallan cada una de las pruebas realizadas, acompañadas de resultados que resumen el comportamiento observado para las diferentes aproximaciones evaluadas.

4.1. Comparativa de recursos y latencia

En la Figura 4.1 se muestra una comparativa global en cuanto a recursos para las versiones del acelerador de punto flotante. En ella se observa que para los casos de Series de Taylor y LUTs se mantienen los recursos de DSPs, FF y LUTs constantes o con un ligero incremento cuando se aumenta el ancho de los datos, mientras que para la versión exacta con *hls::exp()* se presenta un mayor crecimiento en cuanto a FFs cuando este se configura para 32 bits. En cuanto a BRAMs, para los 3 casos de estudio se presenta un incremento sustancial cuanto mayor es el ancho de los datos, llegando en el caso de la versión exacta a ocupar la totalidad de los recursos presentes en la tarjeta que se utiliza como referencia.

Para el caso de punto fijo, se observa un consumo de recursos similar en cuanto a DSPs y FFs entre las versiones aproximadas y la exacta, a excepción del caso en el que el ancho de bits es igual a 4. Sin embargo, las versiones aproximadas presentan un mayor consumo

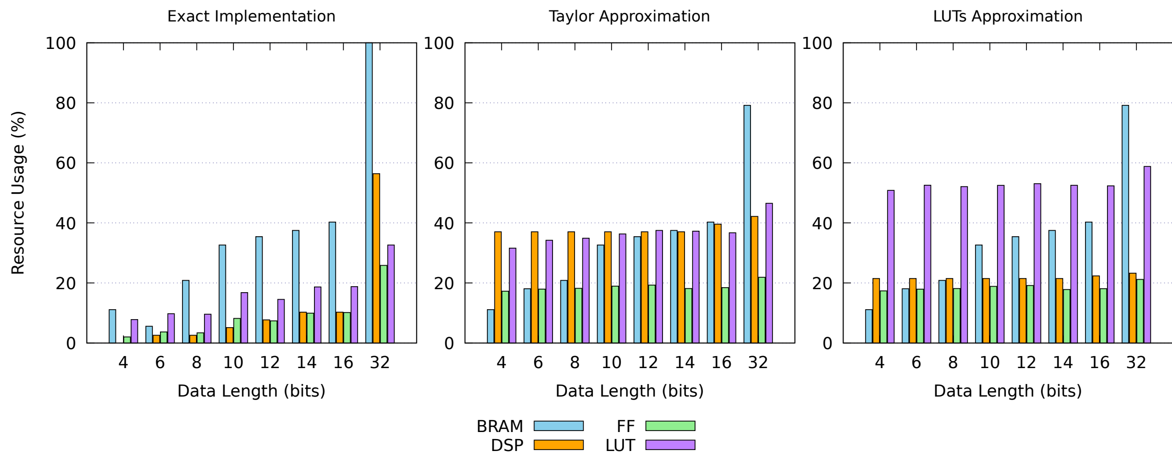

Figura 4.1: Comparativa de utilización de recursos entre la versión exacta, aproximación por Series de Taylor y aproximación por LUTs (Punto flotante).

de recursos en cuanto a BRAMs para anchos de datos pequeños, al igual que los FFs, y las 3 versiones necesitan una gran cantidad de BRAMs cuanto mayor es la precisión configurada. Esto es debido a que se necesita más memoria para almacenar las variables a procesar y las variables intermedias.

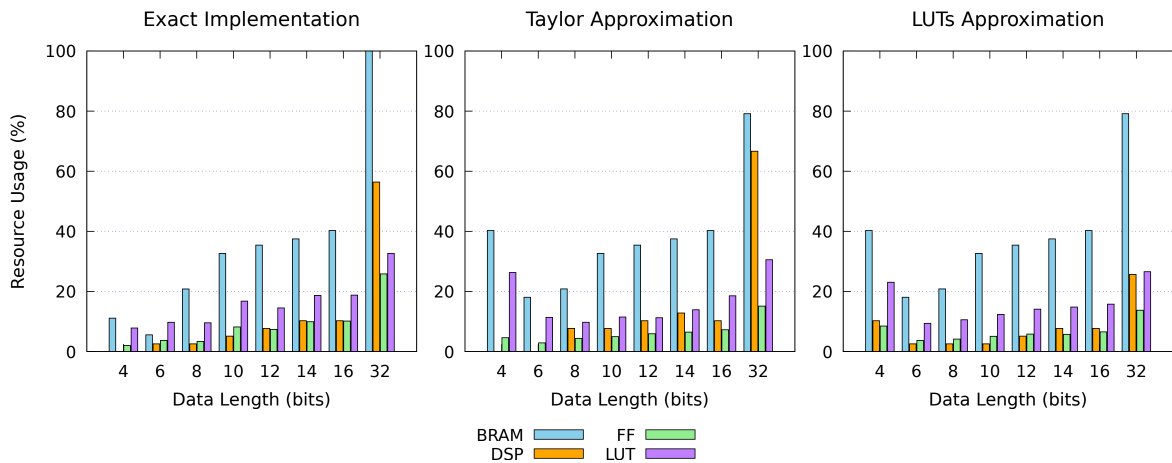

Figura 4.2: Comparativa de utilización de recursos entre la versión exacta, aproximación por Series de Taylor y aproximación por LUTs (Punto fijo).

Para el caso de las aproximaciones implementadas en punto flotante, se observa en la Figura 4.3 que el tiempo de ejecución se mantiene prácticamente constante entre los distintos casos evaluados, siempre que se operen bajo las mismas condiciones de frecuencia y tamaños del vector de entrada. No obstante, la versión exacta presenta un tiempo de ejecución inferior al de las aproximaciones en punto flotante. Esta diferencia se debe a que las implementaciones aproximadas en punto flotante no se encuentran completamente optimizadas, lo que introduce una mayor complejidad en el cómputo y, en consecuencia, incrementa la latencia respecto a la versión exacta.

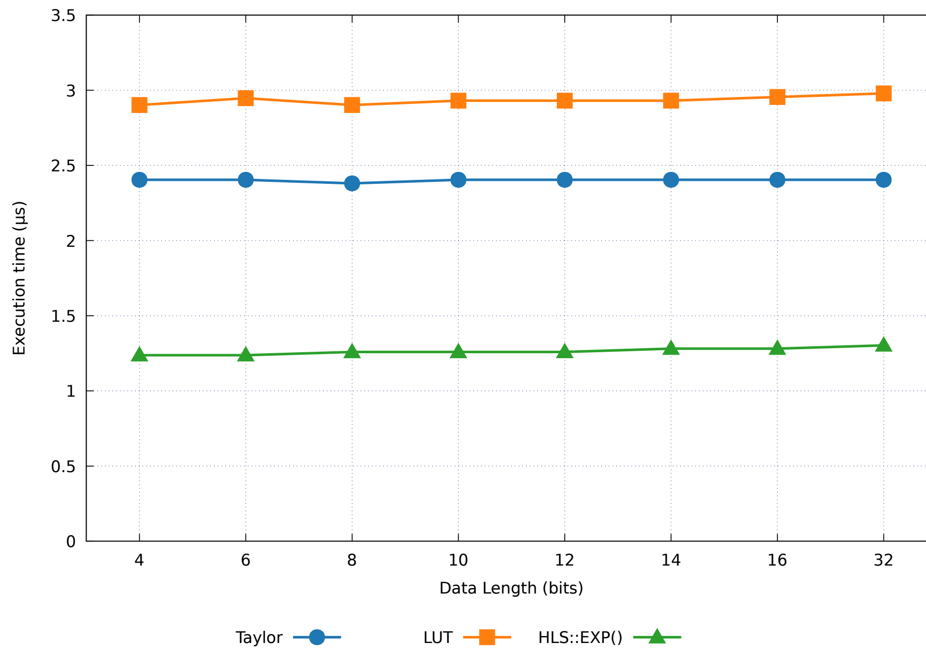

Figura 4.3: Comparación de latencia (tiempo de ejecución) para las implementaciones en Punto Flotante vs versión exacta (punto fijo).

Para las implementaciones de punto fijo, se observa en la Figura 4.4 que las 3 versiones presentan un tiempo de ejecución similar; las aproximaciones por series de Taylor y por LUTs presentan una mejoría notable en tiempo de ejecución para la configuración de 4 bits, mientras que para anchos de datos mayores presentan valores muy similares a la versión exacta.

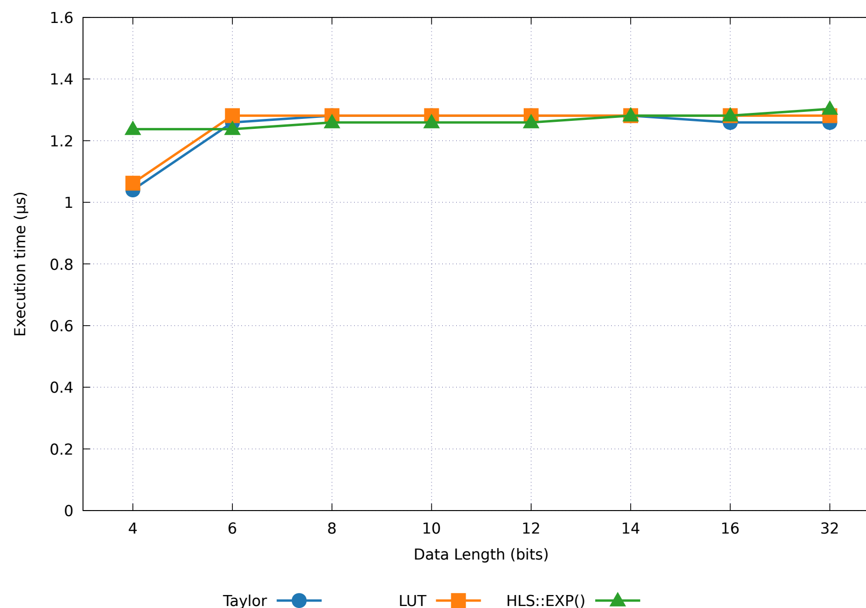

Figura 4.4: Comparación de latencia (tiempo de ejecución) para las implementaciones en Punto Fijo.

4.2. Análisis de la aproximación por Series de Taylor

A continuación, en las Figuras 4.5 y 4.6 se observa el efecto de utilizar distintos órdenes para el polinomio de las Series de Taylor. Se observa un incremento de recursos en términos de DSPs, FFs y LUTs cuanto mayor es el orden de la Serie para ambos tipos de datos. Además, se observa un incremento en tiempo de ejecución cuanto mayor es el orden para el caso de la representación en punto flotante, mientras que para la versión en punto fijo, se observa un incremento en tiempo de ejecución solamente entre los casos de orden 1 y orden 2; las series de orden 2 y orden 3 mantienen un mismo tiempo de ejecución, solamente variando en términos de recursos.

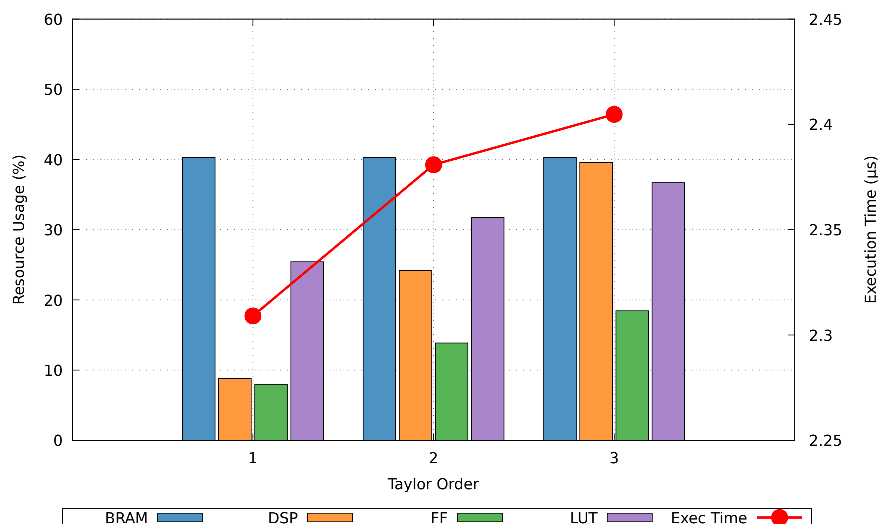

Figura 4.5: Impacto del orden del polinomio de Taylor en los recursos y el tiempo de ejecución (Punto Flotante).

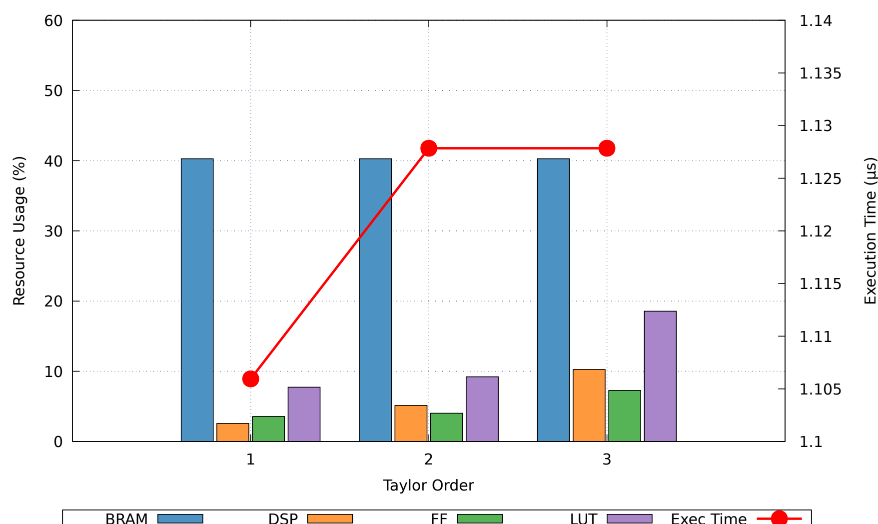

Figura 4.6: Impacto del orden del polinomio de Taylor en los recursos y el tiempo de ejecución (Punto Fijo).

4.3. Análisis de la aproximación por LUTs

Se observa en la Figura 4.7 que para la aproximación de *softmax* mediante LUTs, el número de muestras de la aproximación tiende a aumentar el tiempo de ejecución; así también se muestra un aumento de consumo de recursos en cuanto a LUTs, mientras que las BRAM, DSPs y FFs permanecen en valores constantes o muy similares. Esto es debido a que se necesitan más recursos para representar las LUTs de las aproximaciones, así como también para realizar las consultas necesarias para los cálculos del acelerador.

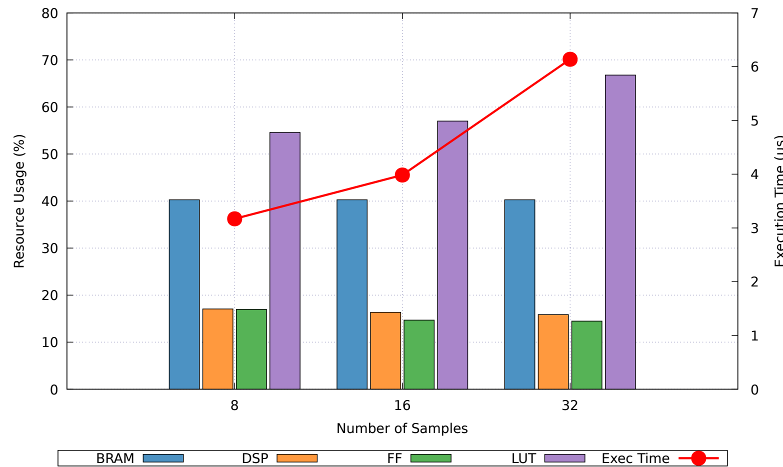

Figura 4.7: Análisis de consumo de recursos y tiempo al variar la cantidad de muestras (Samples) en Punto Flotante.

En lo que respecta a la implementación en punto fijo, la Figura 4.8 evidencia una notable estabilidad en las métricas de desempeño; tanto el consumo de recursos como los tiempos de ejecución se mantienen prácticamente invariables, independientemente de la configuración evaluada.

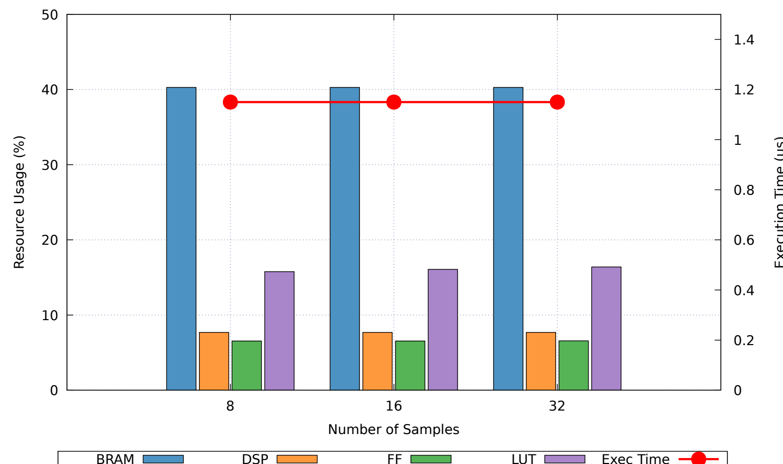

Figura 4.8: Análisis de consumo de recursos y tiempo al variar la cantidad de muestras (Samples) en Punto Fijo.

4.4. Análisis del impacto de la frecuencia de Reloj (F_{clk})

En la Figura 4.9 se observa el efecto de la frecuencia objetivo para el acelerador de *softmax* con aritmética de punto flotante. En la gráfica se muestra poca variabilidad en cuanto a recursos, por lo que el comportamiento se puede decir que es estable en términos de BRAM, DSPs, FFs y LUTs utilizadas a lo largo del barrido de frecuencia, mientras que a partir de 150 MHz se muestra cómo disminuye el tiempo de ejecución cuanto mayor es la frecuencia.

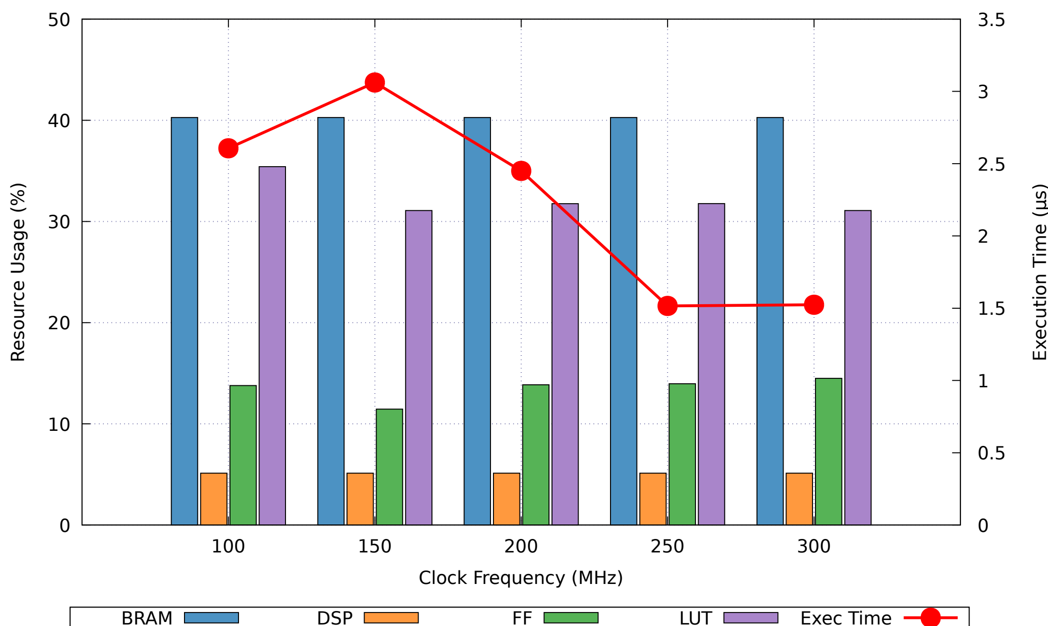

Figura 4.9: Efecto de la frecuencia de reloj sobre el desempeño de la aproximación de Taylor en Punto Flotante.

En la Figura 4.10 se muestra el resultado de realizar un barrido de frecuencia en el acelerador. Se observa que los recursos se mantienen constantes para los distintos casos de estudio, mientras que se presenta una disminución en términos de tiempo de ejecución cuanto mayor es la frecuencia.

En la Figura 4.11 se muestra el efecto de aumentar la frecuencia de reloj para la aproximación por LUTs en punto flotante. Se observa que el tiempo de ejecución es menor en frecuencias altas que en las bajas, mientras que se presenta una estabilidad en cuanto a recursos a lo largo del barrido.

En la Figura 4.12 se observa el efecto de la frecuencia de reloj para el caso de la aproximación por LUTs en punto fijo. En la gráfica se observa una tendencia a la baja en cuanto a tiempo de ejecución, salvo para el caso de 300 MHz, en donde presenta un alza en el tiempo de ejecución. En términos de recursos, se observa que se mantienen relativamente constantes, salvo en 300 MHz en donde aumentan ligeramente las LUTs requeridas.

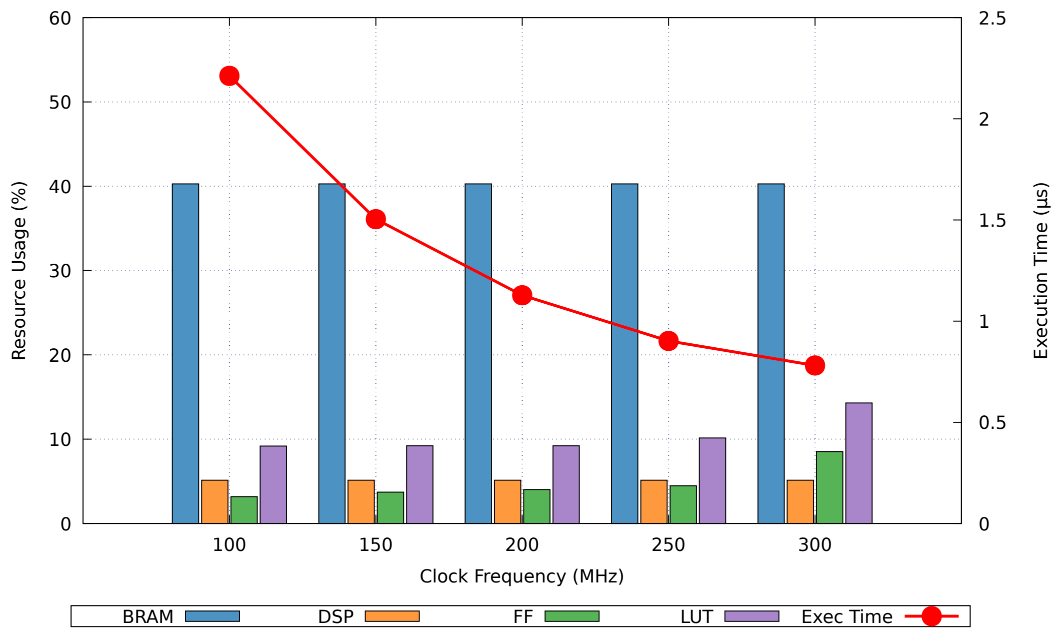

Figura 4.10: Efecto de la frecuencia de reloj sobre el desempeño de la aproximación de Taylor en Punto Fijo.

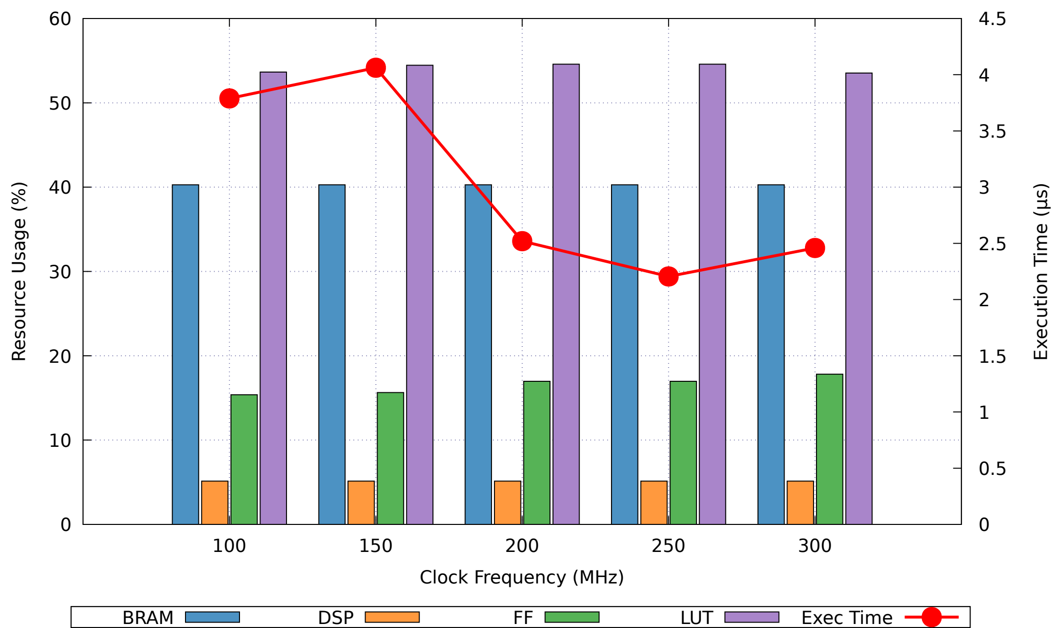

Figura 4.11: Efecto de la frecuencia de reloj sobre el desempeño de la aproximación por LUTs en Punto Flotante.

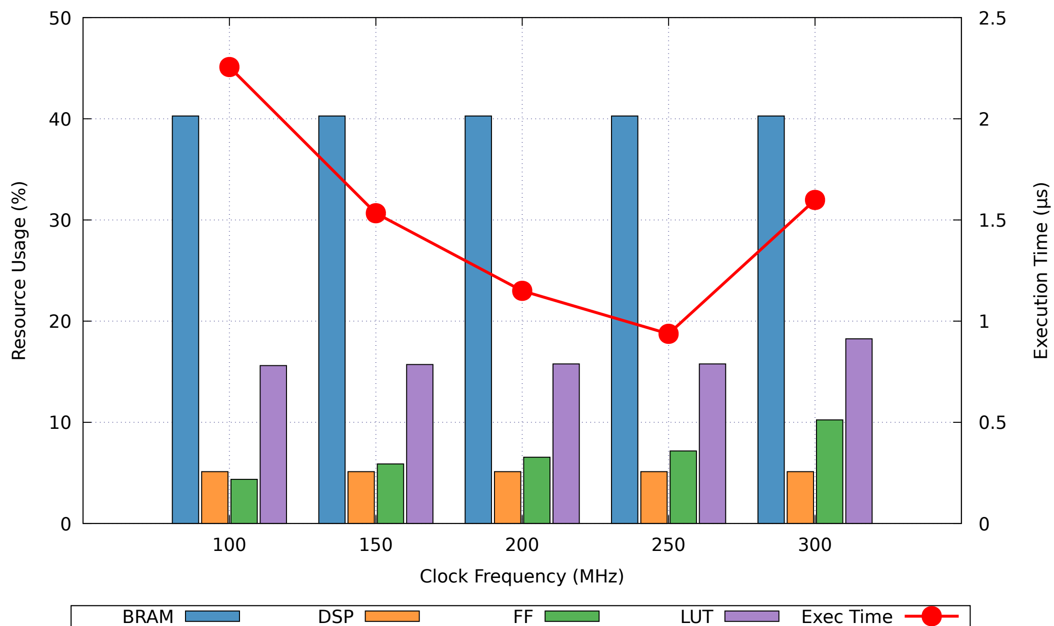

Figura 4.12: Efecto de la frecuencia de reloj sobre el desempeño de la aproximación por LUTs en Punto Fijo.

4.5. Análisis del impacto del tamaño del vector de entrada

En las Figuras 4.13, 4.14, 4.15 y 4.16 se observa un comportamiento consistente en todas las arquitecturas evaluadas. El uso de recursos se mantiene prácticamente constante para cada configuración y no depende del tamaño del vector de entrada. El único parámetro que varía de manera apreciable es el tiempo de ejecución, el cual aumenta de forma proporcional al número de elementos procesados.

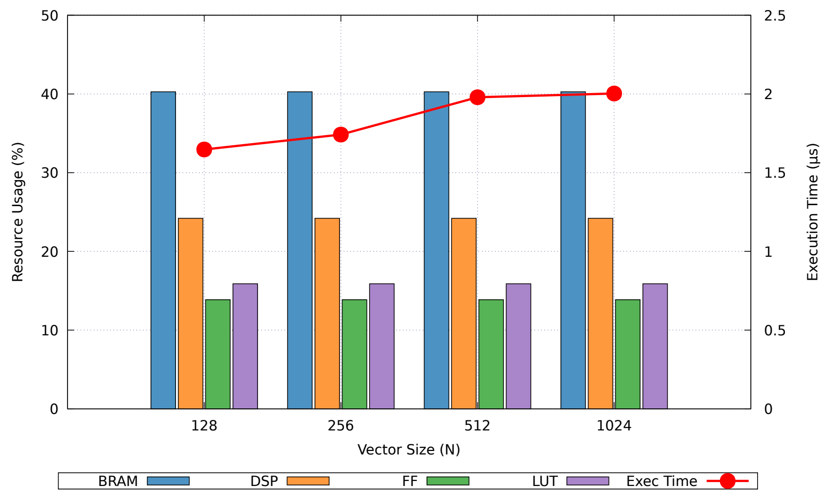

Figura 4.13: Comportamiento de la arquitectura de Taylor (Punto Flotante) respecto al tamaño del vector de entrada.

Este patrón confirma que las arquitecturas están diseñadas con un paralelismo fijo y que el crecimiento del vector afecta únicamente la cantidad de iteraciones realizadas. Como resultado, el costo computacional se incrementa de manera lineal, mientras que el uso de LUTs, FFs, DSPs y BRAM permanece constante.

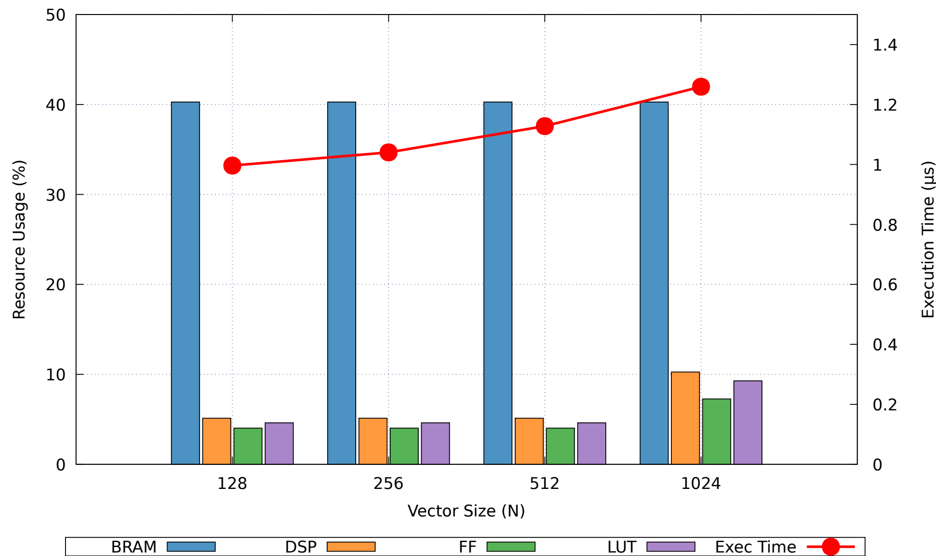

Figura 4.14: Comportamiento de la arquitectura de Taylor (Punto Fijo) respecto al tamaño del vector de entrada.

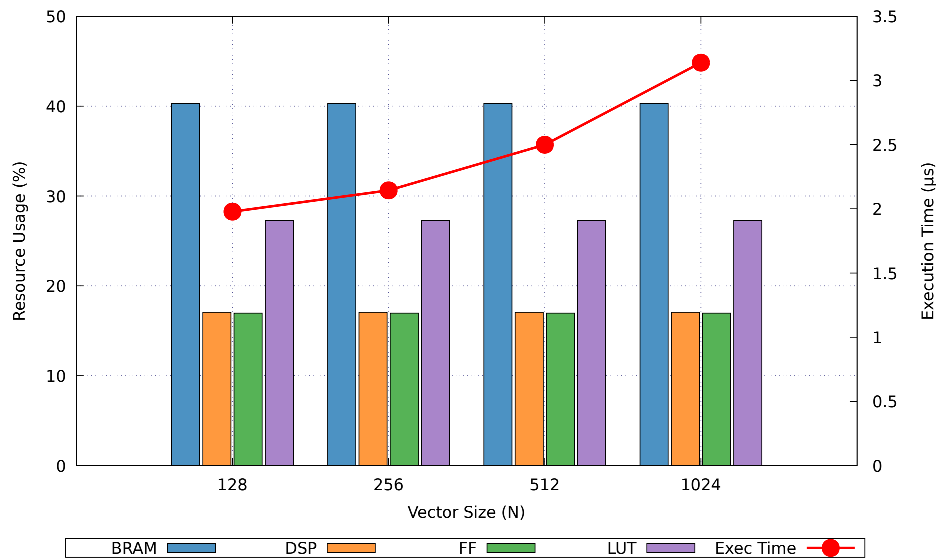

Figura 4.15: Comportamiento de la arquitectura de LUTs (Punto Flotante) respecto al tamaño del vector de entrada.

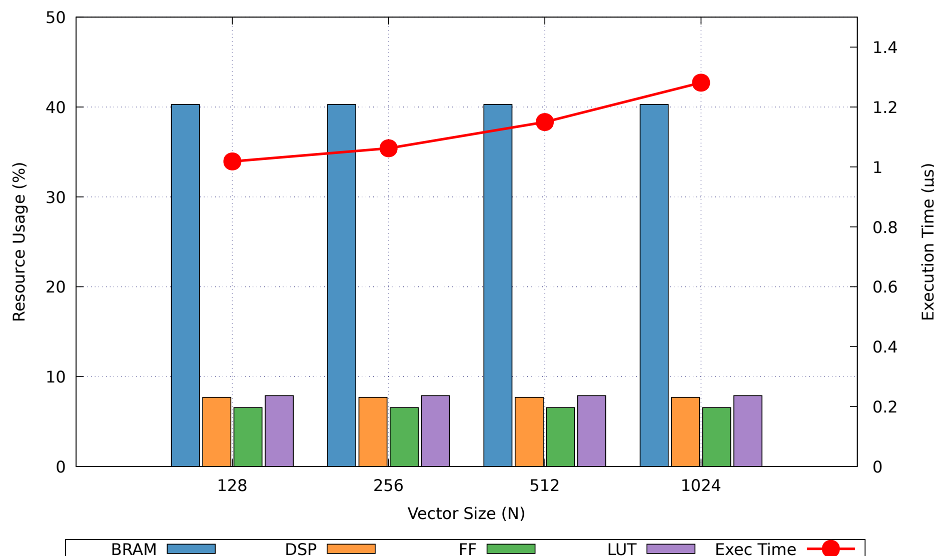

Figura 4.16: Comportamiento de la arquitectura de LUTs (Punto Fijo) respecto al tamaño del vector de entrada.

4.6. Análisis de precisión con LeNet-5 (12 bits)

Los resultados presentados en la Tabla 4.1 muestran las limitaciones importantes que introduce el uso de un ancho de palabra de 12 bits. Idealmente, la sumatoria de las probabilidades de salida debería ser igual a 1.0.

En la versión exacta se observa un comportamiento crítico, ya que la sumatoria alcanza solo 0.78125. Esto representa un error aproximado del 21.875 % y evidencia que el formato de 12 bits no dispone del rango dinámico ni de la precisión necesaria para representar valores muy pequeños de la distribución. En estos casos, los valores tienden a truncarse a cero, lo cual reduce la contribución total y evita que la suma alcance el valor esperado.

En las implementaciones de punto flotante, la configuración basada en *CustomFloat* con aproximación de Taylor de segundo orden ofrece el mejor desempeño global, con una sumatoria de 1.009 y un error cercano al 0.977 %. Esto sugiere que el rango dinámico del punto flotante compensa parcialmente la restricción de bits y permite representar valores pequeños que contribuyen de manera efectiva a la salida final. Al aumentar la complejidad, como ocurre con la aproximación mediante LUT de 32 puntos, se observa un incremento significativo del error, posiblemente debido a la acumulación de errores de redondeo en operaciones sucesivas dentro de una mantisa reducida.

En las implementaciones de punto fijo, la sumatoria se mantiene consistentemente por debajo de 1.0, con valores entre 0.75 y 0.84. Este comportamiento es propio de la saturación y de la limitada precisión fraccionaria del formato. Sin embargo, la aproximación por LUT de 16 puntos presenta el resultado más favorable dentro de este grupo, con un error cercano al 15.625 %, incluso superior al obtenido con la versión exacta. Esto puede explicarse por el hecho de que los valores precalculados en la LUT reducen el error asocia-

do al cálculo en tiempo real y permiten una representación más estable dentro del rango limitado del formato de 12 bits.

Tabla 4.1: Comparación de calidad de resultados para configuración LeNet-5 (12-bit)

Configuración	Suma Total FP	Suma Total Fixed	% Error FP	% Error Fixed
Taylor Orden 1	0.78125	0.75000	21.875	25.000
Taylor Orden 2	1.00977	0.76562	0.977	23.437
Taylor Orden 3	1.01367	0.78125	1.367	21.875
LUT (8 pts)	1.04297	0.81250	4.297	18.750
LUT (16 pts)	0.80664	0.84375	19.336	15.625
LUT (32 pts)	0.67188	0.78125	32.813	21.875
Exact Version	0.78125	0.78125	21.875	21.875

4.7. Análisis de precisión con estándar *IEEE 754 half-precision* (16 bits)

Al analizar los resultados bajo el estándar IEEE 754 Half-Precision (16 bits) presentados en la Tabla 4.2, se observa un contraste notable entre el desempeño del punto flotante y el punto fijo.

En el caso del punto flotante, el formato de 16 bits proporciona el rango dinámico necesario para la función Softmax. Todas las aproximaciones alcanzan una sumatoria muy cercana a la unidad ideal. Destaca la aproximación de Taylor de orden 3, que logra un error residual de apenas 0.0061 %. Esto confirma que la asignación de bits del estándar, entre signo, exponente y mantisa, permite manejar adecuadamente las variaciones características de la función exponencial.

Por otro lado, la implementación en punto fijo de 16 bits no presenta mejoras significativas respecto a la versión de 12 bits. Los errores se mantienen en rangos elevados, entre 15 % y 25 %, sin converger al valor esperado de 1.0. Esto sugiere que el problema no depende únicamente del número total de bits, sino del rango dinámico limitado del punto fijo, donde las probabilidades pequeñas continúan experimentando pérdidas por *underflow*. Este efecto persiste incluso al emplear métodos de aproximación como series de Taylor o tablas de búsqueda.

En conclusión, para arquitecturas basadas en 16 bits, el uso de punto flotante resulta necesario para conservar la consistencia matemática de la capa Softmax, mientras que el punto fijo podría utilizarse para aplicaciones donde las pérdidas de precisión no sean tan significativas.

Tabla 4.2: Comparación de calidad de resultados para estándar IEEE Half-Precision (16-bit)

Configuración	**Suma Total FP**	**Suma Total Fixed**	**% Error FP**	**% Error Fixed**
Taylor Orden 1	1.00024	0.75000	0.0240	25.000
Taylor Orden 2	0.99969	0.76562	0.0305	23.437
Taylor Orden 3	0.99994	0.78125	0.0061	21.875
LUT (8 pts)	1.00012	0.81250	0.0120	18.750
LUT (16 pts)	1.00049	0.84375	0.0490	15.625
LUT (32 pts)	1.00024	0.78125	0.0240	21.875
Exact Version	0.78125	0.78125	21.875	21.875

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

En primer lugar, se logró diseñar e implementar un acelerador funcional utilizando dos métodos principales de computación aproximada: series de Taylor e interpolación lineal mediante tablas de búsqueda (LUTs). Dicha unidad se implementó por medio de lenguaje de programación C++ y es completamente adaptable a la herramienta *Vitis HLS*.

Como segundo aporte, se logró optimizar el acelerador haciendo uso de técnicas de metaprogramación (como *template specialization*), lo que permite generar una arquitectura flexible capaz de alternar entre aritmética de punto fijo y punto flotante en tiempo de compilación.

Con el banco de pruebas se observa que los recursos utilizados por datos en punto flotante son mayores que los de punto fijo para el mismo escenario. Además, en tiempos de ejecución, la arquitectura basada en punto fijo presenta un menor tiempo de ejecución, superando incluso a la versión exacta en el caso de 4 bits. Asimismo, se evidenció que al aumentar el orden de la serie de Taylor, así como la cantidad de muestras para la interpolación mediante LUTs, se produce un incremento en los recursos y tiempos de ejecución para las versiones en punto flotante, mientras que en las versiones en punto fijo estos valores se mantienen más constantes.

También se analizó el efecto de aumentar la frecuencia de operación sobre las distintas versiones del acelerador, observándose una disminución en los tiempos de ejecución, salvo en el caso de la interpolación por LUTs para punto fijo, donde se aprecia un incremento a 300 MHz. Este comportamiento podría originarse por el impacto de la frecuencia sobre una ruta crítica o por alguna violación de temporización. Por otro lado, los recursos se mantienen constantes para todas las frecuencias evaluadas.

Finalmente, realizando el analisis de precisión para los casos de prueba de Lenet-5 (12 bits) e IEEE (*half precision*) se determina que las implementaciones aproximadas en punto flotante constituyen una alternativa precisa para el cálculo de softmax en FPGA,

obteniendo resultados de hasta 0,0061 % para las series de Taylor de Orden 3 en el caso de IEEE *half precision* y un 0,977 % para el caso de la Lenet-5, mientras que las versiones en punto fijo podrían emplearse únicamente en escenarios donde la pérdida de información no comprometa el desempeño del sistema final, ya que las versiones basadas en este tipo de dato reflejaron un alto porcentaje de error, teniendo como error minimo un 15,625 % para el caso de LUT de 16 muestras. Adicionalmente, durante el desarrollo de este proyecto se produjo el siguiente articulo [21].

5.2. Trabajo futuro

A partir de las limitaciones y hallazgos de este trabajo, se proponen las siguientes líneas de investigación para extender la funcionalidad del acelerador:

- **Soporte para reconfigurabilidad dinámica:** Evolucionar la arquitectura para permitir el ajuste de los parámetros de aproximación (como el orden de Taylor o el modo de precisión) en tiempo de ejecución mediante registros de configuración AXI4-Lite. Esto habilitaría la creación de sistemas de *cómputo aproximado dinámico*, capaces de sacrificar precisión para ahorrar energía automáticamente cuando el nivel de batería del dispositivo sea crítico.
- **Optimización de la jerarquía de memoria:** Implementar técnicas de compresión para las tablas de búsqueda (LUTs) o estrategias de particionamiento de arreglos más agresivas en el código fuente. Esto permitiría escalar la solución para utilizar tablas de mayor resolución sin saturar los bloques BRAM disponibles en la FPGA.
- **Integración con otros sistemas:** Escalar el diseño para integrarlo como una capa final dentro de una red neuronal convolucional completa (CNN) desplegada en hardware. Esto permitiría medir el impacto real de la aproximación de la función *Softmax* sobre la precisión final de tareas de clasificación complejas en un entorno de producción.

Bibliografía

[1] A. Upreti, «Convolutional Neural Network (CNN): A comprehensive overview,» *International Journal of Multidisciplinary Research and Growth Evaluation*, vol. 03, págs. 488-493, 2022. DOI: https://doi.org/10.54660/anfo.2022.3.4.18.

[2] J. Fowers, G. Brown, P. Cooke y G. Stitt, «A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications,» 2012, págs. 47-56. DOI: 10.1145/2145694.2145704.

[3] P. Kumar Meher, «An optimized lookup-table for the evaluation of sigmoid function for artificial neural networks,» en *2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip*, 2010, págs. 91-95. DOI: 10.1109/VLSISOC.2010.5642617.

[4] Z. Li, H. Li, X. Jiang, B. Chen, Y. Zhang y G. Du, «Efficient FPGA Implementation of Softmax Function for DNN Applications,» en *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, 2018, págs. 212-216. DOI: 10.1109/ICASID.2018.8693206.

[5] Y. Zhang et al., «Base-2 Softmax Function: Suitability for Training and Efficient Hardware Implementation,» *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, n.º 9, págs. 3605-3618, 2022. DOI: 10.1109/TCSI.2022.3175534.

[6] J. Han y M. Orshansky, «Approximate computing: An emerging paradigm for energy-efficient design,» en *2013 18th IEEE European Test Symposium (ETS)*, 2013, págs. 1-6. DOI: 10.1109/ETS.2013.6569370.

[7] Y. Martín-Hernando, L. F. Rodríguez-Ramos y M. R. Garcia-Talavera, «Fixed-point vs. floating-point arithmetic comparison for adaptive optics real-time control computation,» en *Adaptive Optics Systems*, vol. 7015, SPIE, 2008, 70152Z. DOI: 10.1117/12.787408.

[8] E. O. Fonseca, *Efficient Computing Across the Stack - Instituto Tecnológico de Costa Rica*, 2024. dirección: https://github.com/ECASLab.

[9] L. G. Leon-Vega, D. C. Chavarria, F. E. Fernandez y J. Castro-Godinez, *Flexible Accelerator Library: Approximate Math Operators*, ver. v0.2.0, 2023. DOI: 10.5281/zenodo.8184190.

[10] H. Saadat y S. Parameswaran, «Special session: hardware approximate computing: how, why, when and where?» En *2017 International Conference on Compilers, Architectures and Synthesis For Embedded Systems (CASES)*, 2017, págs. 1-2. DOI: 10.1145/3125501.3125518.

[11] G. Zervakis, K. Tsoumanis, S. Xydis, N. Axelos y K. Pekmestzi, «Approximate Multiplier Architectures Through Partial Product Perforation: Power-Area Tradeoffs Analysis,» en *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, 2015, págs. 229-232. DOI: 10.1145/2742060.2742109.

[12] K. Chen, Y. Gao, H. Waris, W. Liu y F. Lombardi, «Approximate Softmax Functions for Energy-Efficient Deep Neural Networks,» *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, n.º 1, págs. 4-16, 2023. DOI: 10.1109/TVLSI.2022.3224011.

[13] L. Zhao, Z. Yao y S. Wang, «Stability and accuracy analysis for Taylor series numerical method,» *Tsinghua Science and Technology*, vol. 9, n.º 1, págs. 51-56, 2004.

[14] Y. Chen, H. Huang y J. Ma, «Look-Up Table Based Linear Interpolation Algorithm for BPSK Signal Compensation,» en *2025 4th International Symposium on Semiconductor and Electronic Technology (ISSET)*, 2025, págs. 259-264. DOI: 10.1109/ISSET66828.2025.11184891.

[15] M. M. P. Coussy D. Gajski, «An Introduction to High-Level Synthesis,» *Design Test of Computers, IEEE*, págs. 8-17, 2009. DOI: 10.1109/MDT.2009.69.

[16] A. M. Devices, *Vitis Accelerated Libraries*, s.f. dirección: https://www.xilinx.com/products/design-tools/vitis/vitis-libraries.html.

[17] Ó. Lucía, E. Monmasson, D. Navarro, L. A. Barragán, I. Urriza y J. I. Artigas, «Chapter 29 - Modern Control Architectures and Implementation,» en *Control of Power Electronic Converters and Systems*, F. Blaabjerg, ed., Academic Press, 2018, págs. 477-502. DOI: https://doi.org/10.1016/B978-0-12-816136-4.00030-0.

[18] E. Alpaydin, «4 NEURAL NETWORKS AND DEEP LEARNING,» en *Machine Learning*. 2021, págs. 105-141.

[19] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera y M. Martina, «An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks,» *Future Internet*, vol. 12, n.º 7, 2020. DOI: 10.3390/fi12070113.

[20] T. Liang, J. Glossner, L. Wang, S. Shi y X. Zhang, *Pruning and Quantization for Deep Neural Network Acceleration: A Survey*, 2021. arXiv: 2101.09671 [cs.CV].

[21] A. Leiva-Valverde, F. Elizondo-Fernández, L. G. León-Vega, C. Meinhardt y J. Castro-Godínez, *A Quantitative Evaluation of Approximate Softmax Functions for Deep Neural Networks*, 2025. arXiv: 2501.13379 [cs.AR]. dirección: https://arxiv.org/abs/2501.13379.

[22] F. Hajizadeh, T. Ould-Bachir y J. P. David, «CuFP: An HLS Library for Customized Floating-Point Operators,» *Electronics*, vol. 13, n.º 14, 2024. DOI: 10.3390/electronics13142838.

[23] B. Bachelet, A. Mahul y L. Yon, «Template Metaprogramming Techniques for Concept-Based Specialization,» *Scientific Programming*, vol. 21, n.º 1-2, 2013. DOI: https://doi.org/10.3233/SPR-130362.

[24] AMD, *AMD Kria™ System-on-Modules*, s.f. dirección: https://www.amd.com/es/products/system-on-modules/kria.html.