

**Instituto Tecnológico de Costa Rica**

**Área Académica de Ingeniería en Computadores**

(Computer Engineering Academic Area)

**Programa de Licenciatura en Ingeniería en Computadores**

(Licentiate Degree Program in Computer Engineering)



**Diseño y optimización de una estrategia de mejora para una  
Red Neuronal Biológicamente Precisa, basada en el modelo  
Hodgkin Huxley extendido**

(Design and Optimization of an Improvement Strategy for a Biologically Precise Neural  
Network, based on the extended Hodgkin–Huxley Model)

**Informe de Trabajo de Graduación para optar por el título de Ingeniero en  
Computadores con grado académico de Licenciatura**

(Report of Graduation Work in fulfillment of the requirements for the degree of Licentiate  
in Computer Engineering)

Javier Espinoza González

**Cartago, junio de 2018**

(Cartago, June 2018)



TEC – Área Ingeniería en Computadores (CE)  
Acta de Aprobación de Trabajo de Graduación

Con fundamento en lo que establece el "Reglamento de Trabajos Finales de Graduación del Instituto Tecnológico de Costa Rica", el Tribunal Examinador del Trabajo Final de Graduación, nombrado con el propósito de evaluar el proyecto final de graduación.

**Diseño y optimización de una estrategia de mejora para una  
Red Neuronal Biológicamente Precisa, basada en el  
Modelo Hodgkin Huxley Extendido.**

Habiendo analizado el resultado general del trabajo presentado por los estudiantes:

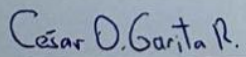
Primer Apellido	Segundo Apellido	Nombre	No. De carné
Espinoza	González	Javier Francisco	201019380


Emite el siguiente dictamen:

<p>APROBADO</p> <p>CALIFICACION: <u>95</u> puntos.</p>	<p><input type="radio"/> REPROBADO</p> <p><input type="radio"/> SE RECOMIENDA      <input type="radio"/> NO SE RECOMIENDA</p> <p>Brindarle una nueva oportunidad para la DEFENSA PUBLICA de su Trabajo Final</p> <p>NUEVA FECHA: _____</p>
--	--

Dando fe de lo aquí expuesto firmamos

  
Dr. Alfonso Chacón Rodríguez  
Profesor Asesor

  
Dr. César Garita Rodríguez  
Profesor Lector

  
Dr. Pablo Alvarado Moya  
Profesor Lector

14 de Junio de 2018

*A mi querida Madre*

## Agradecimientos

Quisiera agradecer a mis amigos, familiares, profesores, compañeros y todas aquellas personas que me ayudaron durante este largo trayecto de la universidad. Pero quiero brindar un especial agradecimiento a los ingenieros Carlos Salazar, Alfonso Chacón y Kaled Alfaro por su apoyo, conocimientos y paciencia brindada durante la realización de este trabajo de graduación.

De igual manera, quiero agradecer a los compañeros del laboratorio HPC de la escuela de Ingeniería en Electrónica por su dedicación y esfuerzo en las tareas que les fueron asignadas para poder llevar a cabo el proyecto de redes neuronales.

Asimismo agradezco a mis dos amigos incondicionales Leandro y German que me han apoyado y orientado como verdaderos hermanos durante todos estos años.

Por último, pero no menos importante, quiero agradecer también a mi hermana Jessica, a mi padre Francisco, pero, especialmente, a mi madre, Mayela González, porque sin ella, sin sus sacrificios y su amor nada de este sueño hubiera sido posible.

Javier Espinoza González  
Cartago, junio

## *Resumen*

### **Diseño y optimización de una estrategia de mejora para una Red Neuronal Biológicamente Precisa, basada en el modelo Hodgkin Huxley extendido**

by Javier Espinoza González

Se propone una optimización para un sistema multi-FPGA SoC con el fin de lograr un sistema heterogéneo, escalable y eficiente, basado en la plataforma Zynq SoC de Xilinx y, en consecuencia, acelerar el cálculo del modelo neuronal biológico de Hodgkin Huxley extendido (eHH). El procesador ARM de la placa SoC se encarga de ejecutar los tres compartimentos computacionales de una sola neurona según se define en el modelo eHH, cada uno con una complejidad computacional  $O(N)$ , mientras que el cálculo de las interacciones de uniones Gap para cada celda se implementa en el FPGA para afrontar su complejidad  $O(N^2)$ , explotando las técnicas de *hardware* de cómputo paralelo. Tal partición *hardware-software* del modelo eHH y su integración en un sistema SoC permite reducir los tiempos de las uniones Gap en un orden cuadrático. Los resultados experimentales muestran que la arquitectura de sistema multi-FPGA propuesta es capaz de manejar hasta 11000 neuronas sobre una sola placa FPGA.

Palabras Clave: GJ, IP, ION, paralelismo, *pipeline*, bloques, FIFO

## *Abstract*

This project proposes an optimization for a multi-FPGA SoC system in order to achieve a heterogeneous, scalable and efficient system, based on Xilinx's Zynq SoC platform, and therefore accelerate the calculation of the extended Hodgkin Huxley(eHH) biological neural model. The ARM processor of the SoC board is responsible for executing the three computational compartments of a single neuron as defined in the eHH model, each with a computational complexity of  $O(N)$ , while the calculation of the Gap junction interactions for each cell is implemented in the FPGA, to confront its complexity  $O(N^2)$  exploiting the techniques of parallel computing hardware. Such a hardware-software partition of the eHH model, and its integration in a SoC system allows to reduce the times of the Gap junctions in a quadratic order. Experimental results show that the proposed multi-FPGA system architecture is capable of handling up to 11,000 neurons on a single FPGA board.

Index Terms: GJ, IP, ION, parallelism, pipeline, blocks, FIFO

# Índice general

<b>Dedicatoria</b>	<b>II</b>
<b>Agradecimientos</b>	<b>III</b>
<b>Resumen</b>	<b>IV</b>
<b>Abstract</b>	<b>V</b>
<b>Lista de abreviaciones</b>	<b>XIII</b>
<b>1. Introducción</b>	<b>1</b>
1.1. Descripción general de la investigación . . . . .	1
1.2. Antecedentes . . . . .	2
1.2.1. Laboratorio en donde se desarrolló la investigación . . . . .	2
1.2.2. Trabajos similares . . . . .	2
1.3. Planteamiento del problema . . . . .	3
1.3.1. Contexto del problema . . . . .	3
1.3.2. Descripción de la situación problemática . . . . .	4
1.3.3. Definición concreta del problema . . . . .	4
1.4. Objetivos de la investigación . . . . .	4
1.4.1. Objetivo general . . . . .	4
1.4.2. Objetivos específicos . . . . .	5
1.5. Alcances, entregables y limitaciones de la investigación . . . . .	5
<b>2. Marco teórico</b>	<b>6</b>
2.1. Modelo ION . . . . .	6
2.2. Modelo Hodgkin–Huxley extendido . . . . .	7
2.3. Paralelismo . . . . .	7
2.3.1. <i>Pipeline</i> . . . . .	8
2.3.2. Hilos . . . . .	9
2.4. Trabajos similares . . . . .	9
2.5. FPGA . . . . .	12
2.6. Plataforma embebida Zynq Zedboard . . . . .	13
2.7. Herramientas de Xilinx . . . . .	13
2.7.1. Vivado HLS . . . . .	14
2.7.2. Vivado IDE . . . . .	14
2.8. Directivas . . . . .	15
2.8.1. <code>pragma HLS dataflow</code> . . . . .	15
2.8.2. <code>pragma HLS pipeline</code> . . . . .	16
2.8.3. <code>pragma HLS array_partition</code> . . . . .	16
2.8.4. <code>pragma HLS interface</code> . . . . .	16
2.9. AXI . . . . .	17
2.9.1. AXI4 . . . . .	17

2.9.2. AXI4-Lite . . . . .	18
2.9.3. AXI4-Stream . . . . .	18
2.10. Biblioteca HLS <i>Stream</i> . . . . .	18
2.11. Complejidad algorítmica . . . . .	19
<b>3. Marco metodológico</b>	<b>20</b>
3.1. Metodología de diseño . . . . .	20
3.2. Secuencia de pasos . . . . .	20
3.3. Estrategias de validación . . . . .	21
3.4. Herramientas y técnicas . . . . .	21
<b>4. Resultados de la investigación</b>	<b>23</b>
4.1. Análisis y medición de las partes del modelo ION en el algoritmo eHH . . . . .	23
4.2. Desarrollo de una estrategia FIFO para la separación del código GJ . . . . .	25
4.2.1. Implementación protocolo AXIS en un IP Core . . . . .	28
4.2.2. Desarrollo de algoritmo GJ para subdivisión de tareas . . . . .	32
4.3. Implementación de mejoras en el algoritmo GJ mediante técnicas <i>pipeline</i> . . . . .	38
4.4. Desarrollo e integración del GJ IP Core dentro de la plataforma de prueba ZedBoard . . . . .	47
4.4.1. Creación del archivo contenedor del código HDL mediante la herramienta Vivado . . . . .	47
4.4.2. Implementación del código HDL en la ZedBoard, mediante la creación de una imagen plantada en una SD . . . . .	49
4.5. Implementación conjunta de las partes del modelo ION . . . . .	52
4.6. Comparación de tiempos entre implementaciones del modelo ION . . . . .	55
4.6.1. Resultados obtenidos . . . . .	55
4.6.2. Análisis de resultados . . . . .	57
<b>5. Conclusiones y recomendaciones</b>	<b>61</b>
5.1. Conclusiones . . . . .	61
5.2. Recomendaciones . . . . .	62
<b>Bibliografía</b>	<b>63</b>



# Índice de cuadros

4.1. Tabla de complejidad de las instrucciones para el algoritmo GJ, según los accesos a memoria y operaciones algebraicas. . . . .	28
4.2. Tabla de tiempos de ejecución para las implementaciones sin <i>pipeline</i> , <i>pipeline</i> simple y <i>pipeline</i> por fila. Nótese la reducción de tiempos de ejecución para cada implementación. . . . .	46
4.3. Tabla de tiempos de ejecución para 10000 pasos de simulación o 500 ms de actividad cerebral. Nótese que se tabula el peor tiempo y mejor tiempo de ejecución para los 10000 pasos de simulación. Los tiempos son obtenidos de la optimización basada en el paralelismo de las GJ con bloques de ejecución de 8 x 8. . . . .	55
4.4. Tabla de tiempos de ejecución para el modelo neuronal eHH de este proyecto. Además, se especifica el tiempo total dividido en dos secciones: la primera el tiempo consumido por el algoritmo GJ; la segunda el tiempo consumido por las componentes de soma + axón + dendrita + cálculo de bloques. Nótese que el tiempo para el algoritmo GJ es el elemento que conlleva el peso de la simulación. . . . .	58
4.5. Tabla de tiempo estimado según la ecuación 4.4 vs. el tiempo medido. Nótese que la estimación es bastante certera. Claramente, puede notarse que la tasa de transferencia será de 864 MB/s, lo cual ratifica el problema existente en la transmisión. . . . .	60

# Índice de figuras

2.1. Representación gráfica del modelo ION, con especificación de las componentes soma, axón y dendrita (a) red de 8 neuronas (b) modelo de neurona única en detalle [6]. . . . .	6
2.2. Código fuente de la representación en C de una GJ realista para el modelo neuronal eHH [6]. . . . .	7
2.3. Estructura ideal de un <i>pipeline</i> por etapas, en la que se ejemplifica el estado actual de cada una de las instrucciones que están siendo paralelizadas [8]. . . . .	8
2.4. Perfiles de <i>software</i> del tiempo de cálculo porcentual tomado por los diversos compartimentos en el modelo de célula ION. Tomado de la implementación del trabajo [9]. . . . .	10
2.5. Perfiles de <i>software</i> de las operaciones aritméticas en el modelo ION para una red de 96 celdas totalmente interconectadas. Tomado del trabajo realizado en [9]. . . . .	10
2.6. Tiempo de ejecución del modelo ION en una placa de desarrollo Virtex-7. Desarrollado en el trabajo [9]. . . . .	11
2.7. Gráfica de la tendencia de crecimiento del tiempo de ejecución por medio de aproximación de curvas de mejor ajuste. El comportamiento del tiempo de ejecución crece de manera cuadrática de acuerdo con el crecimiento de la cantidad de neuronas para el trabajo descrito en [5]. .	11
2.8. Diagrama interno de la arquitectura de una FPGA que muestra en detalle la matriz de bloques lógicos y las interconexiones programables. Tomado de [11]. . . . .	12
2.9. Diagrama interno de la arquitectura Zynq-7000 que muestra las secciones de PL, PS y periféricos de entrada/salida. Tomado de la plataforma de desarrollo Vivado. . . . .	13
2.10. Diagrama que muestra el flujo de funcionamiento para la creación de un IP Core en la herramienta Vivado HLS [14]. . . . .	14
2.11. Ejemplo de la funcionalidad para una instrucción <code>pragma HLS dataflow</code> . 15	
2.12. Ejemplo de la funcionalidad para una instrucción <code>pragma HLS pipeline</code> [16]. . . . .	16
2.13. Diagrama de la arquitectura del canal de Lectura (A) y canal de Escritura(B) para el protocolo AXI4 [17]. . . . .	17
2.14. Diagrama de la arquitectura del canal de Lectura y Escritura para el protocolo AXI4-Stream. Adaptado de [17]. . . . .	18
2.15. Diagrama conceptual del funcionamiento de la biblioteca <code>HLS Stream</code> [18]. . . . .	19
4.1. Gráfica de comparación de tiempos de ejecución para el modelo ION del proyecto ZedBrain, en una arquitectura x86 y una arquitectura ARM. 24	
4.2. Gráfica de comparación de tiempos de ejecución para las componentes del modelo ION del proyecto ZedBrain, en una arquitectura x86. .	24

4.3.	Gráfica de comparación de tiempos de ejecución para las componentes del modelo ION del proyecto ZedBrain, en una arquitectura ARM.	25
4.4.	Gráfica de comparación de tiempos de ejecución para las componentes del modelo ION del proyecto ZedBrain, con GJ diferenciado, en una arquitectura <i>software</i> .	26
4.5.	Código en detalle de la comunicación ejercida en las GJ para el modelo ION, tomado del proyecto ZedBrain.	27
4.6.	Diagrama del mapa de memoria en el SoC donde se sitúa la interfaz AXI4_Lite en el IP. Se muestra además el código C++ con los pragmas necesarios para generar esta interconexión dentro del SoC . Figura adaptada de [5].	29
4.7.	Diagrama de bloques sobre el manejo del IP con la interfaz AXI4_Stream y controlado por el DMA, mediante la biblioteca HLS_Stream.	30
4.8.	Código que ejemplifica el uso de la interfaz AXI4_Stream en código C++, en un ambiente Vivado HLS, mediante la utilización de la biblioteca HLS_Stream.	32
4.9.	Estudio de las señales de control para el protocolo AXI4_Stream mediante el uso de la herramienta de visualización de formas de onda en el ambiente Vivado HLS. Nótese la línea de color rojo que muestra la activación de la señal TLAST, así como las señales TREADY, de color morado y TVALID de color verde, las cuales se activan para indicar el flujo de recepción de datos y para la indicación de los datos válidos, respectivamente. En TDATA vemos cada dato escrito al bus, para cada escritura realizada.	33
4.10.	Diagrama representativo de la funcionalidad del algoritmo GJ, el cual realiza un recorrido matricial por filas, compartiendo información con cada elemento de la fila.	33
4.11.	Diagrama representativo de la estrategia de subdivisión de tareas llevada a cabo en el algoritmo GJ, para ser compatible con un comportamiento FIFO que permita una mayor escalabilidad y eficiencia en el proyecto.	34
4.12.	Diagrama de bloques de el algoritmo GJ modificado para una división por tareas, para ser compatible con una estructura de datos FIFO(productor-consumidor).	36
4.13.	Gráfica porcentual del uso de los recursos para la implementación del algoritmo GJ, con bloques de tamaño 2 x 2, 4 x 4 y 8 x 8. Se puede observar que, en principio, se mantiene una similitud bastante alta para cada uno de los bloques utilizados.	37
4.14.	Gráfica comparativa de los tiempos ejecución requeridos para la implementación del algoritmo GJ con bloques de tamaño 2 x 2, 4 x 4 y 8 x 8. Nótese que el comportamiento para el bloque 8 x 8 mantiene un mejor rendimiento que los bloques 2 x 2 y 4 x 4.	37
4.15.	Representación de un riesgo de datos en el algoritmo cuando se lee un dato del arreglo <i>savedData</i> antes de que se haya terminado su escritura en memoria. Este fue encontrado durante el análisis de detección de riesgos durante la aplicación de la técnica de paralelismo de tuberías.	39
4.16.	Representación de la directiva en código C++ para la introducción de la técnica pipeline en el algoritmo GJ.	39

4.17. Gráfica comparativa de los tiempos ejecución requeridos para la implementación del algoritmo GJ con bloques de tamaño $2 \times 2$ , $4 \times 4$ y $8 \times 8$ , aplicando la técnica de paralelismo <i>pipeline</i> . Nótese que, al igual que la figura 4.14, el bloque $8 \times 8$ muestra mayor eficiencia conforme aumenta el número de neuronas para los tiempos de ejecución. . . . .	40
4.18. Gráfica porcentual del uso de los recursos para la implementación del algoritmo GJ, con bloques de tamaño $2 \times 2$ , $4 \times 4$ y $8 \times 8$ , utilizando una técnica <i>pipeline</i> . Se puede distinguir un aumento de los recursos, comparado con los mostrados en la figura 4.13. El aumento claramente es debido a la implementación del <i>pipeline</i> . . . . .	41
4.19. Análisis de formas de ondas sobre el algoritmo GJ utilizando una técnica de paralelismo de <i>pipeline</i> . Nótese una pérdida de tiempo de 980ns para la lectura entre los datos que son controlados por la bandera TDATA. . . . .	42
4.20. Representación de la directiva en código C++ para la introducción de la directiva <i>dataflow</i> en el IP del algoritmo GJ. Esta se utiliza para la optimización de la estrategia FIFO. . . . .	42
4.21. Gráfica porcentual del uso de los recursos para la implementación del algoritmo GJ, con bloques de tamaño $2 \times 2$ , $4 \times 4$ y $8 \times 8$ , utilizando una técnica <i>pipeline</i> y la directiva <i>dataflow</i> . . . . .	43
4.22. Gráfica comparativa de los tiempos ejecución requeridos para la implementación del algoritmo GJ con bloques de tamaño $2 \times 2$ , $4 \times 4$ y $8 \times 8$ , aplicando la técnica de paralelismo <i>pipeline</i> y la directiva <i>dataflow</i> . . . . .	43
4.23. Gráfica de la eficiencia para la implementación del algoritmo GJ con bloques de tamaño $2 \times 2$ , $4 \times 4$ y $8 \times 8$ , aplicando la técnica de paralelismo <i>pipeline</i> y la directiva <i>dataflow</i> . . . . .	44
4.24. Gráfica comparativa de los tiempos ejecución para un bloque de $8 \times 8$ con una implementación del algoritmo GJ previa a la utilización de <i>pipeline</i> vs. la implementación llevada a cabo con las optimizaciones mediante <i>pipeline</i> . Obsérvese que la implementación sin <i>pipeline</i> posee una línea cuadrática mucho más pronunciada comparada contra su homologas implementadas con <i>pipeline</i> . . . . .	45
4.25. Gráfica comparativa de los tiempos ejecución para un bloque de $8 \times 8$ con una implementación del algoritmo GJ con la utilización de <i>pipeline</i> simple vs. la implementación con <i>pipeline</i> por filas. Nótese que en simulaciones pequeñas los tiempos de ejecución del <i>pipeline</i> doble son más altos que el <i>pipeline</i> simple; sin embargo, al aumentar el número de neuronas existe una clara ventaja sobre el <i>pipeline</i> por filas. . . . .	46
4.26. Gráfica porcentual de las mejoras de tiempos de ejecución para las implementaciones del algoritmo GJ con <i>pipeline</i> simple y <i>pipeline</i> por filas, contra la implementación sin <i>pipeline</i> . Nótese como la implementación con <i>pipeline</i> por filas incrementa la mejora conforme aumenta el número de neuronas. . . . .	47
4.27. Módulo <code>Simulate_Hardware</code> que contiene el IP con el algoritmo de GJ modificado, visto desde el entorno de desarrollo Vivado. . . . .	48
4.28. Diagrama de bloques sobre la implementación final hacia la placa ZedBoard, que detalla la interconexión entre las IP del algoritmo GJ y el controlador de DMA. . . . .	48
4.29. Diagrama del contenido requerido en una tarjeta SD para la creación de un disco de arranque mediante una distribución Linux en la Zed-board. Imagen tomada de [5]. . . . .	49

4.30. Porción de código que representa la implementación del árbol de dispositivos, se puede apreciar la interacción con los IP del GJ y el IP del DMA. . . . .	51
4.31. Vista gráfica del directorio <i>/home</i> , ubicada en el sistema de archivos para la partición ROOTFS, la cual contiene los archivos necesarios para la sincronización de todas las componentes del modelo ION con el IP para GJ. . . . .	52
4.32. Diagrama del flujo que sigue el código implementado para el modelo ION, con el uso de la biblioteca <i>pThreads</i> . . . . .	53
4.33. Porción de código que establece conexión mediante el protocolo AXI_Lite para iniciar el IP o enviar parámetros necesarios en el mismo. . . . .	54
4.34. Diagrama representativo de la función para formar los bloques de 8 x 8 ejecutados en la IP de GJ modificada. . . . .	54
4.35. Gráfica de tiempos de ejecución finales para el modelo neuronal eHH construido en este proyecto. Nótese que la curva de los tiempos de ejecución tiene una tendencia cuadrática mostrada en la ecuación de la figura. . . . .	56
4.36. Gráfica comparativa de tiempos de los ejecución para el modelo neuronal eHH implementado previamente, contra el implementado en este proyecto. Nótese que los tiempos de este proyecto son hasta un 47 % peores que el proyecto previo. . . . .	57
4.37. Gráfica de porcentaje sobre el tiempo consumido para el algoritmo GJ modificado en promedio, contra el tiempo requerido por el promedio de la suma para las componentes soma, axón y dendrita sin GJ. . . . .	58
4.38. Gráfica de tiempos para la componente de el algoritmo GJ modificado en un modelo de co-simulación, con un bloque de 8 x 8, contra los tiempos de la componente dendrita con GJ analizados como parte del problema a solucionar del proyecto previo, ZedBrain. Nótese como el bloque IP del GJ actual sobrepasa enormemente en términos de velocidad de procesamiento al bloque dendrita con GJ integrado. . . . .	59

## Lista de abreviaciones

**AMBA** (Advanced-Microcontroller **B**us **A**rchitecture)

**ASIC** (Application-Specific Integrated **C**ircuit)

**ARM** (Advanced **RISC** **M**achine)

**AXI** (Advanced e**X**tensible **I**nterface)

**CPU** (Central **P**rocessing **U**nit)

**BRAM** (Block **R**andom **A**ccess **M**emory)

**DCILab** (Design **C**ircuits **I**ntegrate **L**aboratory)

**DDR3** (Double**D**ata **R**ate type three)

**DMA** (Direct **M**emory **A**ccess)

**DSP** (Digilant **S**ignal **P**rocessing)

**eHH** (extended **H**odgkin **H**uxley)

**EX** (Execute)

**FF** (Flip **F**lop)

**FIFO** (First **I**n **F**irst **O**ut)

**FPGA** (textbfField **P**rogrammable **G**ate **A**rrays)

**FSBL** (First **S**tate **B**oot **L**oader)

**GJ** (Gap **J**unction)

**GPGPU** (General-Purpose computation on **G**raphics **P**rocessing **U**nits)

**HDL** (Hardware **D**escription **L**anguage)

**HLS** (High **L**evel **S**ynthesis)

**HPC** (High **P**erformance **C**omputer)

**ID** (Instruction Decoding)

**IDE** (Integrated Deveploment Enviroment)

**IF** (Instruction Fetch)

**II** (Initiation Inteval)

**ILP** (Instruction Level Parallelism)

**ION** (Inferior Olivary Nucleus)

**IP** (Intellectual Property)

**LUT** (Look-Up Table)

**MEM** (Memory access)

**MPI** (Message Passing Interface)

**PL** (Programmable Logic)

**PS** (Processing System)

**RAM** (Random Access Memory)

**RAW** (Read After Write)

**RISC** (Reduced Instruction Set Computer)

**RTL** (Register-Transfer Level)

**SD** (Secure Digital)

**SoC** (System of Chip)

**SRAM** (Static Random Access Memory)

**UART** (Universal Asynchronous Receiver-Transmitter)

**USB** (Universal Serial Bus)

**UTP** (Unshielded Twisted Pair)

**WAR** (Write After Read)

**WAW** (Write After Write)

**WB** (Write Back)

## Capítulo 1

# Introducción

### 1.1. Descripción general de la investigación

El BrainFrame Project es una iniciativa internacional que busca afrontar uno de los grandes retos de la ingeniería: la simulación del cerebro y sus funciones. La generación de múltiples modelos de representación de neuronas con complejidades variables ha impuesto un reto a las áreas de tecnologías de información y ha demostrado ser un campo multidisciplinario.

Con el objeto de buscar mejorar la comprensión del comportamiento cerebral humano, se ha recurrido a la computación de alto rendimiento; sin embargo, esta ha encontrado grandes limitaciones en los métodos tradicionales de resolución de problemas, por lo que la búsqueda de nuevas tecnologías ha generado una emergente corriente de computación concurrente basada en dispositivos FPGA que han demostrado resultados promisorios en estudios iniciales. Para este proyecto, se pretende demostrar y aprovechar el alto rendimiento aportado por estos dispositivos, optimizando al máximo el proyecto previo con la utilización de diferentes técnicas de HLS y paralelismo que permitan un alto aprovechamiento de los recursos del sistema.

Este documento describe un proyecto que se encuentra inscrito en una colaboración entre el Laboratorio HPC de la Escuela de Ingeniería Electrónica del Tecnológico de Costa Rica y el Erasmus Brain Project del Erasmus Medical Center, en Rotterdam, Países Bajos.

Este proyecto está contextualizado como una continuación de una serie de proyectos que se han llevado a cabo en diferentes etapas. Para este proyecto en específico, se realiza un análisis sobre el modelo neuronal eHH con el fin de identificar los problemas de rendimiento que afecten la simulación de una red neuronal. Asimismo, desarrolla técnicas de paralelismo como la implementación de un *pipeline* por instrucciones dentro de un IP Core que, entre otras cosas, permite mejorar mediante ejecución paralela los tiempos de respuestas ante un determinado algoritmo.

Por último, este trabajo abarca una exploración en tecnologías que permitan desarrollar una estrategia de mejora ante problemas específicos como la escalabilidad, flexibilidad y eficiencia del proyecto previo. De la misma manera dedica una importante parte para el análisis de las tecnologías aplicadas o los resultados que se obtengan durante lo largo del proceso.



## 1.2. Antecedentes

### 1.2.1. Laboratorio en donde se desarrolló la investigación

La investigación fue llevada a cabo en el Laboratorio HPC, ubicado en la Escuela de Electrónica del Tecnológico de Costa Rica, pertenece a un laboratorio de investigación, conformado especialmente por estudiantes y profesores de las carreras de Ingeniería de Computadores e Ingeniería Electrónica. Dicho laboratorio se gestó como necesidad de continuar proyectos iniciados en el laboratorio DCILab.

En este laboratorio, trabajan los profesores Dr. Alfonso Chacón Rodríguez y el Ing. Carlos Salazar García. Este último se encuentra desarrollando su tesis doctoral en este laboratorio, en un proyecto conjunto con una sociedad multi-disciplinaria de departamentos de universidades a nivel global, compañías de investigación e institutos que tienen como objetivo común proveer diferentes herramientas que permitan avanzar en el conocimiento del cerebro humano y, en consecuencia, ofrecer soluciones a problemas conocidos a nivel neuronal.

Dicha sociedad es liderada por el Departamento de Neurociencia del Centro Médico Erasmus (Erasmus Medical Center), el Instituto Holandés de Neurociencia y el Instituto de Comunicaciones y Sistemas Computacionales. Asimismo, cuenta con participación de diferentes entidades universitarias tales como universidades de Grecia, Suecia, Colombia, Holanda, entre otras.

Se puede observar, por lo tanto, que el laboratorio HPC atiende un mercado poco conocido e investigaciones, relacionados específicamente con simulaciones neuronales, llevadas a cabo en diferentes equipos, tales como FPGA, que permiten probar y obtener resultados de computación de alto rendimiento a un bajo costo económico.

### 1.2.2. Trabajos similares

En [1] se menciona el creciente interés de explorar alternativas como las arquitecturas ARM, específicamente diseños híbridos como ARM-FPGA en HPC, ya que pueden ser catalizadores potenciales; de igual manera, P. Moorthy y N. Kapre, autores principales del artículo mencionado antes, coinciden en que las FPGA ofrecen la promesa de mejorar el rendimiento de las aplicaciones junto con el consumo de energía, esto comparados para una ejecución en paralelo con un equipo que utiliza un procesador X86.

Con base en dicha teoría, estos autores se plantean ejecutar algunas técnicas de rendimiento con el fin de medir la aceleración en un clúster creado con placas Zync SoC, aprovechando el paralelismo que ofrece dichas placas, asimismo establecieron comunicación mediante MPI para acelerar el problema de los gráficos dispersos, que suelen representar un embotellamiento de memoria en los sistemas x86 tradicionales. Según se muestra en los resultados y mediciones, se prueba que el clúster ZedWulf implementado puede ser casi 2 veces más eficiente energéticamente que los sistemas tradicionales basados en x86.

Otro documento que analiza un entorno similar es [2], en el que se plantea la caracterización de la simulación de redes neuronales en tiempo real a gran escala,

utilizando el paralelismo de las FPGA. Este proyecto posee un enfoque en neurociencia y tiene como fin el concretar un análisis de por qué las FPGA son mucho mejores que las GPGPU actuales y las CPU básicas. En este artículo, se describe la construcción de Bluehive, una máquina basada en FPGA, que aprovecha el paralelismo de dichas plataformas. Los resultados concretos mostrados se ofrecen sobre una implementación de el modelo Izhikevich, un modelo neuronal biológicamente preciso, con el cual se resaltan las capacidades de un diseño sobre las FPGA personalizadas, así como su alta escalabilidad, una utilización baja del ancho de banda y una baja latencia, lo que evidentemente mejora el rendimiento y permite una mayor utilización de los recursos.

## 1.3. Planteamiento del problema

### 1.3.1. Contexto del problema

Tal como se indicó en la sección de “Antecedentes”, el proyecto se desarrolla en el marco del Erasmus Brain Project y, en particular, sobre su subtema BrainFrame. Este proyecto busca crear una infraestructura capaz de proveer simulaciones de redes neuronales altamente detalladas en tiempo real. Tiene la finalidad de abrir un espacio de investigación para estudiar el órgano más complejo del ser humano, el cerebro, así como poder desarrollar secciones artificiales de este que sustituyan zonas dañadas por contusiones o accidentes cerebrovasculares, según explica el Ing. Carlos Salazar, en uno de los artículos publicados por el Tecnológico de Costa Rica para la divulgación de este proyecto [3].

El modelo a utilizar simula el núcleo olivar inferior [4], una de las secciones del cerebro humano que se encarga del manejo y control de las actividades sensoromotoras del cuerpo. Las neuronas allí establecidas se ven afectadas por la actividad eléctrica de la zona y cada neurona se ve influida por millares de condiciones externas. Como resultado de esa interacción, para generar una salida, la neurona recibe una corriente de entrada desde el núcleo cerebral y, a partir de la misma, se produce una corriente de salida como efecto a la reacción electro-química que se origina en su interior. Por ello, cada neurona, en conjunto con sus neuronas vecinas, genera una red neuronal compleja, en la que cada celda o neurona en su red se constituyen, a su vez, de tres compartimentos: soma, dendrita y axón; la dendrita es la etapa de entrada que conecta el cuerpo de la neurona con sus neuronas colindantes; el soma es el núcleo en el que se produce la reacción electro-química a partir de las entradas; el axón es la etapa de salida, la cual se conecta con el entorno neuronal (neuronas colindantes).

Específicamente para este proyecto de graduación, se cuenta con una implementación de una red neuronal de 4 nodos físicos en placas de desarrollo ZedBoards, llamada ZedBrain. ZedBrain utiliza el modelo neuronal extendido de Hodgkin Huxley, que a su vez contempla las neuronas con sus tres partes. El proyecto efectúa su comunicación mediante una conexión de red, Ethernet 100-BaseT sobre un conmutador 1000 Base T. Se utilizan protocolos MPI para su comunicación paralela. Existen, no obstante, limitaciones a la hora de la escalabilidad, lo que restringe a un número finito la cantidad de neuronas posibles a simular y establece un tope en el rendimiento del proyecto.

La utilización de nuevas técnicas que aceleren su rendimiento será el fundamento del proyecto con el empleo de herramientas como Vivado para aplicar HLS. El equipo a utilizar se mantiene en placas de desarrollo de *hardware* ZedBoard, lo que conlleva una limitante, ya que el proyecto debe desarrollarse estrictamente en el laboratorio HPC.

### 1.3.2. Descripción de la situación problemática

La problemática del proyecto consiste en la necesidad de optimizar y depurar la red neuronal albergada en el proyecto ZedBrain, la cual actualmente se encuentra implementada mediante el modelo neuronal extendido de Hodgkin Huxley en una red de 4 nodos de *hardware*, representados por las FPGA/ZedBoard. Estos nodos se comunican vía red; sin embargo, existe una limitante en la expansión a nuevos nodos, lo cual también imposibilita un aumento en el número posible de neuronas simuladas. El problema radica, entonces, en tres factores principales: escalabilidad, tiempo de ejecución y flexibilidad del sistema.

De [5], se toma la figura 2.7, la cual resume la problemática y establece los tiempos de ejecución del proyecto base, en el que para una sola ZedBoard, una simulación con trescientas neuronas conlleva un tiempo de procesamiento de alrededor 350 ms. Igualmente, en la figura 2.7, se concluye que el sistema tiene un comportamiento cuadrático en los tiempos de ejecución conforme aumenta el número de neuronas, por lo que en este evento es posible buscar una optimización determinante.

De igual manera, tal como se mencionó anteriormente, la escalabilidad se ve severamente limitada, ya que al implementar el número de las FPGA actuales, los tiempos de ejecución crecen enormemente. Lo anterior sucede ya que cada ZedBoard conectada al sistema contiene parámetros de entrada con arreglos de tamaño 1024. Estos arreglos se ven replicados en cada ZedBoard. Al replicarse se exceden las capacidades técnicas, limitando y afectando la escalabilidad y efectividad del sistema.

Se propone como solución el uso de un FIFO en la implementación actual, combinado con técnicas HLS, que se estima minimizará el problema y reducirá los tiempos de respuesta considerablemente. Por ende, para la resolución de este proyecto se deben involucrar aspectos en computación de alto rendimiento, así como tener un dominio clave en temas de control de equipos como las FPGA.

### 1.3.3. Definición concreta del problema

La necesidad de optimizar y depurar la red neuronal albergada en el proyecto ZedBrain, con la realización de una mejora en tres factores principales: escalabilidad, tiempo de ejecución y flexibilidad del sistema.

## 1.4. Objetivos de la investigación

### 1.4.1. Objetivo general

Optimizar el rendimiento de una red neuronal biológicamente precisa implementada en un sistema multi-FPGA con la disminución del tiempo de ejecución y la mejora de la comunicación entre los distintos nodos.

### 1.4.2. Objetivos específicos

- Incrementar el número máximo de neuronas que se pueden simular en el sistema neuronal mediante una reestructuración en los bloques.
  - **Indicador:** El sistema neuronal previo permite una simulación de 8000 neuronas en total y 2048 neuronas por cada FPGA.
- Disminuir el tiempo de ejecución para el algoritmo extendido Hodgkin Huxley, aprovechando el paralelismo de la FPGA.
  - **Indicador:** El sistema neuronal previo, para una simulación de 1200 neuronas y 4 FPGA, permite alcanzar tiempos de ejecución de entre 100 y 150 ms.
- Mejorar la flexibilidad del sistema neuronal para que permita un valor arbitrario e independiente en los estímulos de entrada del número de FPGA utilizadas.
  - **Indicador:** En el sistema neuronal previo los valores de los estímulos de entrada deben ser múltiplos del número de FPGA utilizadas.

## 1.5. Alcances, entregables y limitaciones de la investigación

Dentro de los entregables se encuentran:

- Anteproyecto
- Tutoriales o guías de descripción de los procesos llevados a cabo en los diferentes objetivos
- Resultados experimentales y análisis de implementaciones
- Proyecto finalizado
- Documento final de proyecto

Algunas de las limitaciones afrontadas fueron :

- Cantidad de equipos FPGA disponibles en el laboratorio, ya que la capacidad de miembros del equipo supera el número de dispositivos disponibles.
- Factor tiempo, ya que el tiempo de desarrollo del proyecto es restringido, por lo que es necesario una buena delimitación de los alcances.
- Recursos de cómputo disponible, debido a que el laboratorio no cuenta con suficiente equipo.

Los alcances logrados fueron:

- Sistema Neuronal con una optimización eficiente y funcional, ejecutado en un sistema FPGA.
- Trazo de una línea clara del camino a seguir para futuros proyectos.

## Capítulo 2

# Marco teórico

### 2.1. Modelo ION

El núcleo olivar inferior es una parte esencial del cerebro humano. En este proyecto, se utiliza un modelo matemático eHH que describe esta región con precisión biofísica. Según [6], el núcleo olivar inferior forma una parte intrincada del sistema olivo-cerebeloso, el cual es una de las regiones cerebrales más densas y desempeña un papel importante en el control sensoriomotor. Lo anterior provoca que las neuronas del olivar inferior sean especiales es su interconexión densa a través de conexiones eléctricas llamadas uniones de brecha (GJ, del inglés *gap junctions*) que difieren de las sinápsis típicas en que son puramente eléctricas. Las GJ facilitan el comportamiento de sincronización entre las neuronas olivar y, posteriormente, influyen en la sincronización y las propiedades de aprendizaje de todo el sistema olivo-cerebeloso.

En [6], se describe una neurona basada en tres compartimentos distintos: la dendrita (ubicación de las GJ), el soma y el axón. Esta distribución se aprecia con más detalle en la figura 2.1.

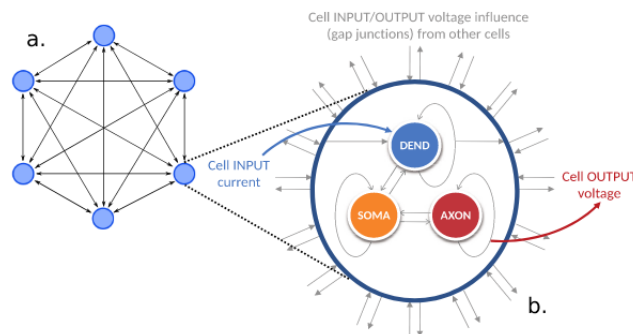


FIGURA 2.1: Representación gráfica del modelo ION, con especificación de las componentes soma, axón y dendrita (a) red de 8 neuronas (b) modelo de neurona única en detalle [6].

- Dendrita: se encarga de recibir las señales eléctricas de estímulo provenientes de sus neuronas vecinas y transferirla al soma [5].
- Soma: procesa el estímulo recibido y genera una acción de respuesta. Las interacciones generadas entre neuronas se llaman sinápsis, las cuales se observan de manera de pulsos en la respuesta del soma [5].
- Axón: que toma la salida del soma y este la adapta para transmitir la señal a otras neuronas [5].

## 2.2. Modelo Hodgkin–Huxley extendido

Este modelo numérico consiste en un conjunto de ecuaciones diferenciales no lineales que se utiliza para aproximar las características eléctricas de la célula (neurona). Es un modelo de tiempo continuo que describe el comportamiento de las células a lo largo del tiempo, en el que, en cada paso de tiempo, calcula un conjunto de parámetros para cada neurona. Para el cálculo de un único parámetro, es necesario llevar a cabo una función exponencial y varias multiplicaciones y divisiones. Además, para alcanzar la máxima precisión y una representación realista de la señal, el uso de la aritmética de coma flotante es esencial. Estos requisitos hacen que la complejidad computacional del modelo sea muy alta [7].

El modelo utilizado en este proyecto se encuentra expuesto en [6] y es una extensión del modelo Hodgkin Huxley, el que hace la separación entre las partes de la célula neuronal, soma, axón y dendrita. Además, define un tiempo de 50 micro segundos de ejecución por paso, un tiempo relacionado con la convergencia del método de Euler utilizado para resolver numéricamente las ecuaciones diferenciales (ver [6]). En consecuencia, a lo largo de este proyecto, será la meta ideal en cuanto a rendimiento temporal.

Específicamente, en conjunto, con la separación descrita, el modelo utiliza una representación realista en las GJ. Los GJ aquí se implementan como una representación muy específica del núcleo biológico (figura 2.2). Cada célula C en una población de  $N$  células acumula la influencia de una célula interconectada a ella (a través de un GJ) al restar su propio voltaje dendrítico ( $prevVdend$ ) del voltaje dendrítico de esa celda ( $neighVdend[i]$ ). De este modo, se acumula la influencia del voltaje resultante en una corriente agregada  $Ic$ , la cual factoriza el peso de la conexión GJ respectiva ( $C[i]$ ) [6].

---

```

1: for i = 0; i < InfOli_N_INPUT; i++ do
2:     V = prevVdend - neighVdend[i];
3:     f = 0.8 * V * exp(-1 * V * V / 100) + 0.2;
4:     Ic = Ic + (C[i] * f * V);
5: end for
6: return Ic;

```

---

FIGURA 2.2: Código fuente de la representación en C de una GJ realista para el modelo neuronal eHH [6].

## 2.3. Paralelismo

El paralelismo representa una técnica de programación e implementación en la que se pretende realizar operaciones simultáneamente con el fin de reducir tiempos de ejecución en un procesador. Existen diferentes tipos de paralelismo, entre los cuales se encuentra el paralelismo a nivel de datos, el paralelismo a nivel de hilos y el paralelismo a nivel de instrucciones [8].

El paralelismo a nivel de instrucciones (ILP) consiste en una técnica basada en la ejecución simultánea de instrucciones, entre los tipos de ILP más populares se encuentra la segmentación por tuberías o *pipeline* [8].

### 2.3.1. Pipeline

El *pipeline* es una técnica utilizada en el diseño de CPU para aumentar el rendimiento, mediante la separación de las etapas en el proceso de ejecución de una instrucción [8]. Entre los supuestos que existen para poder aplicar una técnica de *pipeline* se encuentran [8]:

- Todas las instrucciones pasan por todas las etapas.
- Las etapas no comparten recursos de *hardware* entre sí.
- El tiempo de propagación entre las etapas es el mismo.
- Las instrucciones son independientes entre sí.
- Las etapas se puede aislar temporalmente.

Entre las etapas de separación durante el proceso de ejecución de la instrucción se encuentran [8]:

- Búsqueda de instrucción (IF): su función principal es traer una nueva instrucción de la memoria de instrucciones.
- De-codificación de instrucciones(ID): se dice que el ID es el traductor de instrucciones además de realizar la lectura de los registros operados.
- Ejecución(EX): etapa que realiza las operaciones en la unidad lógica matemática.
- Acceso a memoria (MEM): encargada de realizar las instrucciones de carga y almacenaje en la memoria de datos.
- Escritura a registros (WB): escritura de resultados al banco de registros

En la figura 2.3, se muestra el proceso que seguiría un *pipeline* ideal, en el que se observa un bucle con cinco instrucciones que están separadas por un intervalo de iniciación de un ciclo de reloj. En este, se ve cada etapa de las instrucciones: IF, ID, EX, MEM y WB a lo largo de los 9 ciclos de reloj ejecutadas paralelamente.

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

FIGURA 2.3: Estructura ideal de un *pipeline* por etapas, en la que se ejemplifica el estado actual de cada una de las instrucciones que están siendo paralelizadas [8].

Las ventajas que posee un *pipeline* consiste en que provoca un aumento en el rendimiento del CPU, además de que brinda determinismo en la ejecución de las instrucciones. Las desventajas son que etapas o instrucciones lentas pueden afectar severamente el rendimiento, además que implica una mayor complejidad en *hardware*. Existe un ligero aumento en la latencia del programa, así como puede provocar riesgos de ejecución [8].

Un riesgo es una situación que previene que la siguiente instrucción pueda ser ejecutada en el ciclo de reloj correspondiente. Para este proyecto, se desea enfatizar los riesgos de datos que son las dependencias reales que existen en datos de instrucciones. Un riesgo de datos ocurre cuando en el *pipeline* se cambia el orden de acceso a lectura/escritura de operandos, de forma que el orden difiere de la ejecución secuencial en un procesador sin *pipeline* [8].

### 2.3.2. Hilos

Hilos (*threads* en inglés) es una técnica de que permite aprovechar el paralelismo por medio de la división de la ejecución de instrucciones en paquetes, llamados hilos [8].

Los hilos, según [4], son múltiples instancias de ejecución dentro de un proceso que comparten el mismo espacio de memoria y descriptores para el acceso a los recursos del sistema. En Linux, los hilos se crean dentro de un proceso a través de la llamada al sistema, aunque en la mayoría de los casos los programadores usan la biblioteca `pthread` para crear y gestionar los hilos. Estos fomentan el paralelismo, además de los recursos compartidos dentro de la ejecución de una aplicación.

## 2.4. Trabajos similares

Existen trabajos similares y trabajos previos los cuales han realizado estudios e investigaciones sobre la misma sección del cerebro, el ION. Lo anterior se debe a la relevancia que la misma tiene sobre el cuerpo humano. Sin embargo, las simulaciones se han vuelto rápidamente inviables debido al crecimiento significativo de los datos a procesar. Algunas de los trabajos ya realizados al respecto son [5] y [9].

En [9], se trabaja el modelo ION perfilado en código C, el cual es ejecutado mediante Matlab en una FPGA, posteriormente se traduce a código HLS para ser implementado en una tarjeta Vitex-7 en la que se logra mejorar hasta por 700 su predecesor Matlab.

Asimismo, en [9], se describe la aplicación del modelo ION, en la que se trabaja las neuronas por sus tres componentes: soma, axón y dendrita. Se perfila el modelo en código C mediante el gráfico que se muestra en la figura 2.4, el cual describe el tiempo de cálculo de las tres componentes y ejecuta por separado las GJ para enfatizar la importancia y el costo en tiempo de operación que requiere el modelo ION en un ambiente de *hardware*.



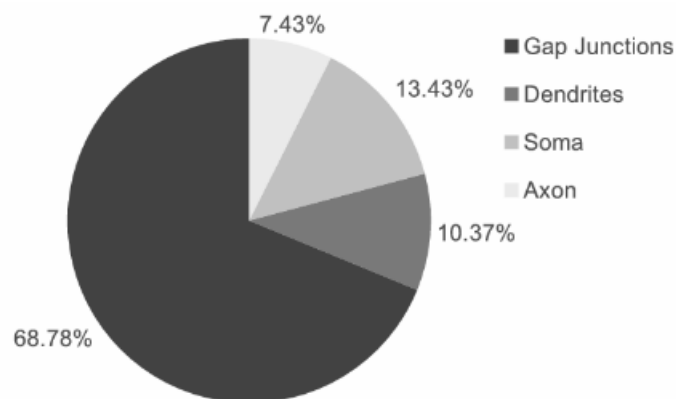


FIGURA 2.4: Perfiles de *software* del tiempo de cálculo porcentual tomado por los diversos compartimentos en el modelo de célula ION. Tomado de la implementación del trabajo [9].

En conjunto, con la figura 2.4, se encuentra en [9] la figura 2.5 en la que se puede observar que las GJ son responsables de un gran número de operaciones aritméticas a la hora de la ejecución del código que representa el modelo ION. La figura 2.5 describe que para el total de operaciones de suma, resta, división y multiplicación en el modelo ION, la mayor cantidad corresponde a la función GJ, mientras que las restantes pertenecen a los demás componentes del modelo, el soma, dendrita y axón.

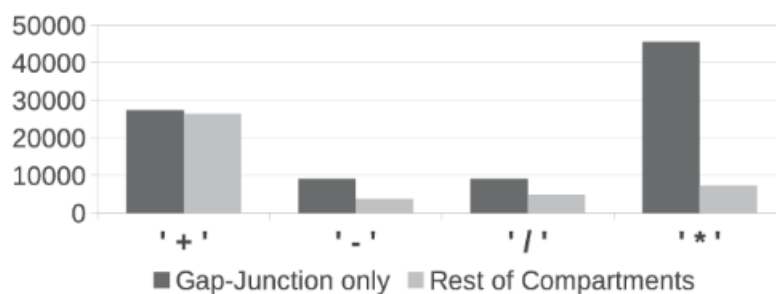


FIGURA 2.5: Perfiles de *software* de las operaciones aritméticas en el modelo ION para una red de 96 celdas totalmente interconectadas. Tomado del trabajo realizado en [9].

La figura 2.6 traza el tiempo de ejecución del diseño planteado en [9] para diferentes tamaños de red. Se puede observar que el tiempo de ejecución se escala con el tamaño de la red, debido a los cálculos de GJ que elevan exponencialmente el costo del rendimiento para una red interconectada total.

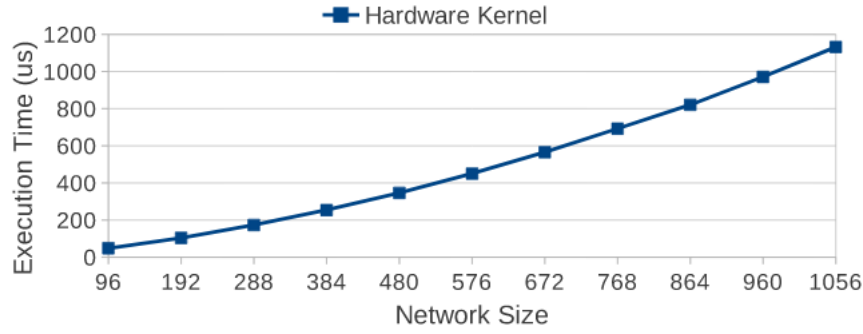


FIGURA 2.6: Tiempo de ejecución del modelo ION en una placa de desarrollo Virtex-7. Desarrollado en el trabajo [9].

Otro trabajo similar corresponde al planteado en [5], el cual utiliza el modelo eHH, este representa el proyecto predecesor para esta investigación o proyecto previo como se le llamará a lo largo de este trabajo. En la figura 2.7, se observa el tiempo de ejecución llevado a cabo, con un tope de 1200 neuronas para las diferentes ejecuciones con una, dos o hasta cuatro placas en paralelo. Asimismo, se puede distinguir que la línea de tendencia obtiene una alineación cuadrática de acuerdo al incremento de la red neuronal.

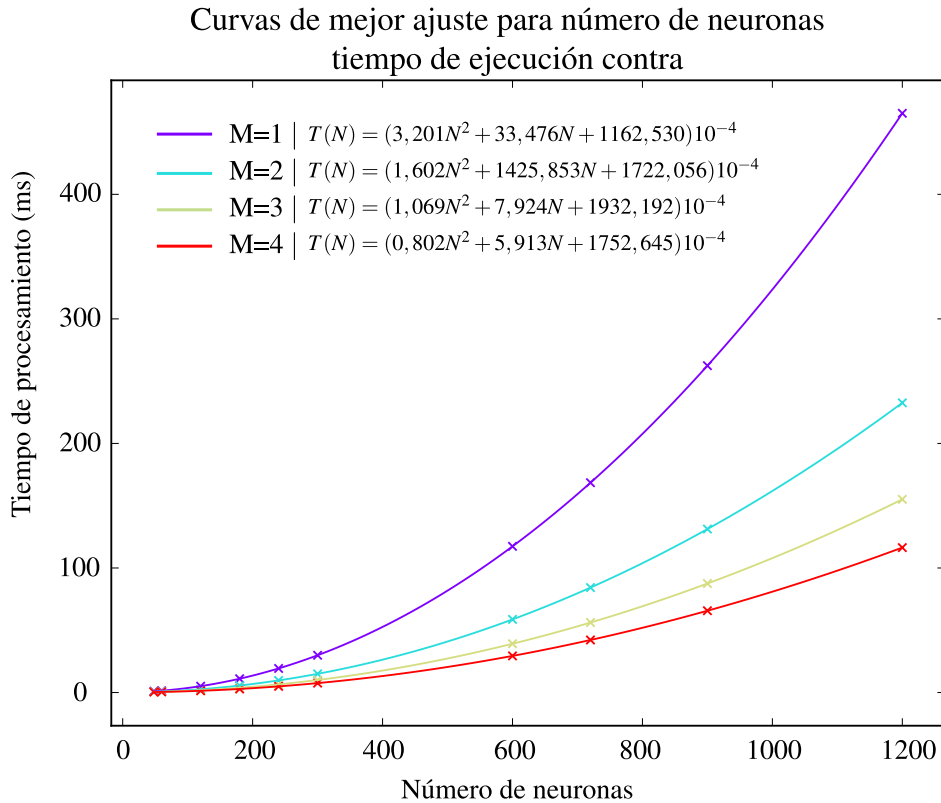


FIGURA 2.7: Gráfica de la tendencia de crecimiento del tiempo de ejecución por medio de aproximación de curvas de mejor ajuste. El comportamiento del tiempo de ejecución crece de manera cuadrática de acuerdo con el crecimiento de la cantidad de neuronas para el trabajo descrito en [5].

## 2.5. FPGA

Las FPGA son dispositivos semiconductores que se basan en una matriz de bloques lógicos configurables conectados a través de interconexiones programables. Las FPGA se pueden reprogramar según la aplicación deseada o los requisitos de funcionalidad después de la fabricación. Esta característica distingue y diferencia a las FPGA de los Circuitos Integrados de Aplicación Específica (ASIC), las cuales se fabrican a medida para tareas de diseño únicas. Aunque existen de igual manera las FPGA programables por una única vez, las FPGA dominantes o con mayor mercado se basan en SRAM, ya que se pueden reprogramar a medida que el diseño evoluciona [10].

Para programar una FPGA, a diferencia de los procesadores, en los cuales se escriben subrutinas de instrucciones con antelación. Para la FPGA se modifica una matriz de conexiones. Los bloques individuales de dicha matriz están constituidos por elementos que les permiten adoptar distintas funciones de transferencia. Juntos, los distintos bloques programados hacen que físicamente se constituya un circuito electrónico, de forma similar a un ASIC [11].

En la figura 2.8, se muestra un diagrama de la arquitectura interna de una FPGA, en donde se puede apreciar con más detalle la matriz de bloques lógicos y las interconexiones programables.

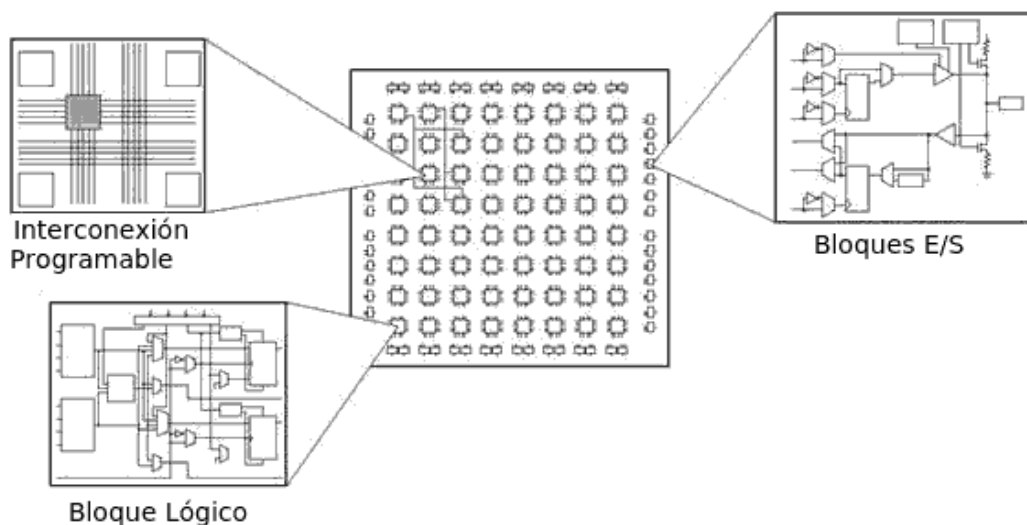


FIGURA 2.8: Diagrama interno de la arquitectura de una FPGA que muestra en detalle la matriz de bloques lógicos y las interconexiones programables. Tomado de [11].

Debido a su naturaleza programable, las FPGA son ideales para muchos mercados diferentes. Entre las principales aplicaciones se encuentran [10]:

- **Prototipos ASIC:** creación de prototipos ASIC con FPGA, lo cual permite realizar el modelado, una verificación rápida y precisa del sistema SoC del *software* integrado.
- **Medicina :** para aplicaciones de diagnóstico, monitoreo y terapia.
- **Computación de alto rendimiento y almacenamiento de datos**

## 2.6. Plataforma embebida Zynq Zedboard

ZedBoard es una placa de evaluación y desarrollo basada en Xilinx Zynq-7000 con todos los SoC programables y es la tarjeta de desarrollo que se utiliza en este proyecto. La placa combina dos secciones: un Dual Cortex-A9, el cual es un procesador ARM con periféricos integrados, que se identifica como sistema de procesamiento (PS). La otra sección es la lógica programable (PL) con 85000 celdas Series-7 [12].

Entre algunas de sus características más relevantes para este proyecto, se encuentran que, la placa posee 512 MB de memoria RAM DDR3, USB UART, 10/100/1000 Ethernet, entrada 12V DC y compartimiento para tarjeta SD [12].

El Zynq-7000 se puede orientar para un uso amplio en muchas aplicaciones. La sólida combinación de la ZedBoard con periféricos incorporados y capacidades de expansión la convierten en una plataforma ideal tanto para diseñadores principiantes como experimentados [12].

En la figura 2.9, se puede observar el diagrama interno de la Zynq-7000, en este se aprecia en detalle las secciones PL y PS, además de los periféricos de entrada y salida disponibles.

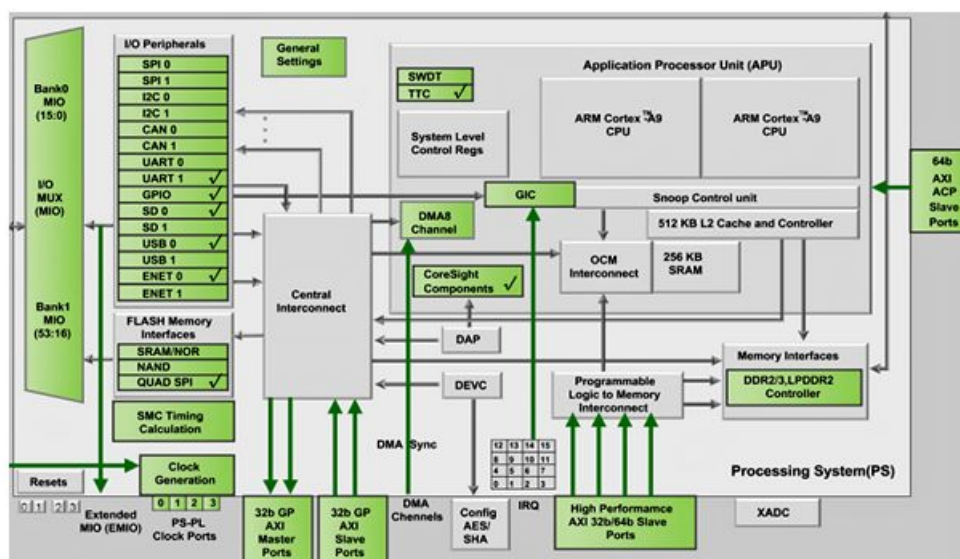


FIGURA 2.9: Diagrama interno de la arquitectura Zynq-7000 que muestra las secciones de PL, PS y periféricos de entrada/salida. Tomado de la plataforma de desarrollo Vivado.

## 2.7. Herramientas de Xilinx

HLx es la representación de una nueva era de las soluciones de programación de Xilinx para el desarrollo de sistemas inteligentes, conectados y diferenciados que aprovechan el *hardware* personalizado y optimizado para productos finales utilizando todos los dispositivos programables [13]. Entre sus herramientas, se describen las

que se presentan a continuación, las cuales fueron utilizadas en el desarrollo de este proyecto.

### 2.7.1. Vivado HLS

La herramienta Xilinx Vivado HLS sintetiza una función C en un bloque de IP Core que puede integrarse en un sistema de *hardware*. Está estrechamente integrado con el resto de las herramientas de diseño de Xilinx y proporciona un soporte integral de lenguaje y características, con el fin de crear una implementación óptima para un algoritmo C [14].

El proceso que sigue la herramienta se muestra en la figura 2.10 en donde Xilinx Vivado HLS recibe como entrada el código fuente en C, las directivas y el *Testbench*, nombre que recibe el código fuente que cumple la función de verificación por medio de pruebas. Una vez que las entradas han sido procesadas, se compila, ejecuta (simula) y depura el algoritmo (actividades que en conjunto se conocen como simulación C); posteriormente, se realiza una síntesis del algoritmo C que crea una implementación RTL (opcionalmente se usan las directivas de optimización de usuario). Por último, se empaqueta la implementación de RTL en una selección de formatos de IP [14], lo cual genera las salidas: archivos de implementación RTL en formato HDL y archivos de reporte.

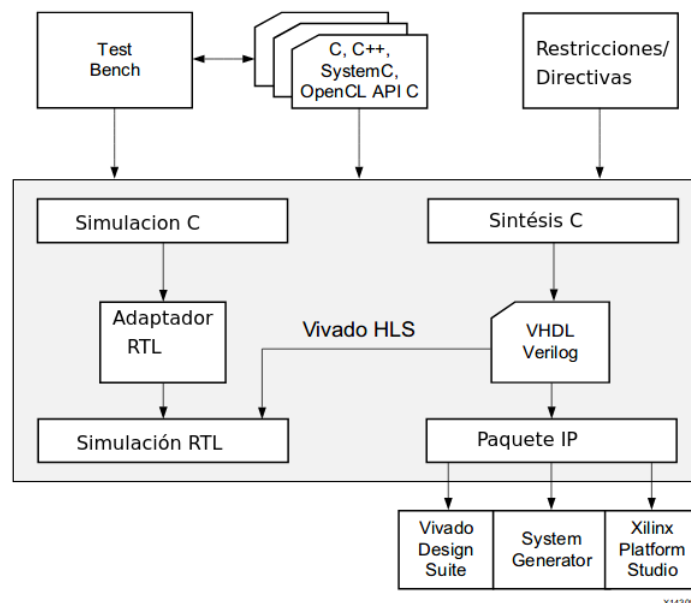


FIGURA 2.10: Diagrama que muestra el flujo de funcionamiento para la creación de un IP Core en la herramienta Vivado HLS [14].

### 2.7.2. Vivado IDE

El entorno de diseño integrado (IDE) de Vivado proporciona una interfaz gráfica de usuario intuitiva con potentes funciones. Todas las herramientas y opciones de herramientas están escritas en el formato nativo del lenguaje de comando de herramientas [15]. Vivado IDE permite la instanciación, manipulación e integración de las *IP* en un archivo con extensión *.bit*.

## 2.8. Directivas

Las directivas o pragmas son instrucciones opcionales en la herramienta Vivado HLS y dirigen el proceso de síntesis para implementar un comportamiento u optimización. Es decir, una directiva permite modificar la lógica del *hardware* para extraer la mejor calidad de resultados de la partición de este, o bien mejorar el rendimiento del núcleo, el rendimiento de los datos, reducir la latencia o reducir los recursos utilizados por el núcleo [16]. Dentro de las directivas usadas en este proyecto se encuentran:

### 2.8.1. pragma HLS dataflow

El pragma `dataflow` facilita la canalización a nivel de tarea, permitiendo que las funciones y bucles se superpongan en su operación, con ello aumenta la concurrencia de la implementación de RTL y mejora el rendimiento total del diseño.

Todas las operaciones se realizan secuencialmente en una descripción C, lo que implica que la latencia y la concurrencia suelen aumentar, debido a las dependencias de datos que suelen establecer limitantes. Por ejemplo, las funciones o bucles que acceden a matrices deben finalizar todos los accesos de lectura y escritura a las matrices antes de que se completen. Lo anterior evita que la siguiente función o bucle consuma los datos de la operación de inicio. La optimización `dataflow` permite que las operaciones en una función o ciclo comiencen a funcionar antes de que la función o bucle anterior complete todas sus operaciones. En la figura 2.11, se muestra un ejemplo de su funcionalidad [16].

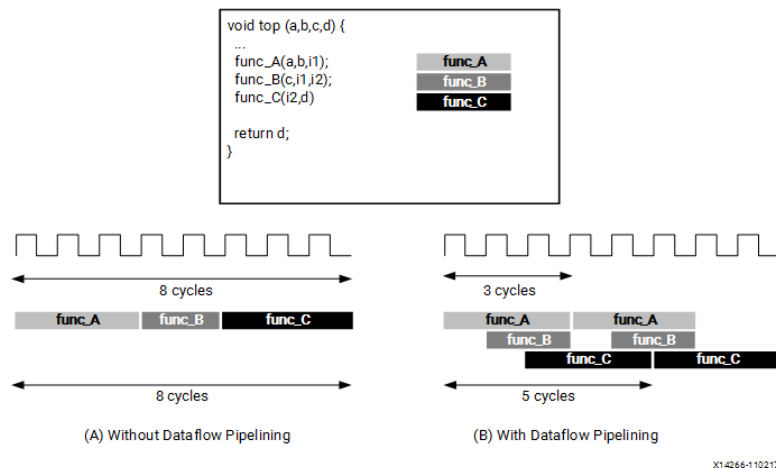


FIGURA 2.11: Ejemplo de la funcionalidad para una instrucción `pragma HLS dataflow`.

Cuando se especifica un pragma `HLS dataflow`, Vivado HLS analiza el flujo de datos entre funciones secuenciales o bucles y crea canales (basados en pingpong RAM o FIFO) que permite que las funciones del consumidor o bucles comiencen a funcionar antes de que las funciones o bucles del productor se hayan completado. Lo anterior se ejecuta con el fin que las funciones o bucles funcionen en paralelo, lo que disminuye la latencia y mejora el rendimiento del RTL [16].

### 2.8.2. pragma HLS pipeline

El `pragma pipeline` reduce el intervalo de inicio para una función o bucle al permitir la ejecución concurrente de operaciones. Una función o bucle canalizado puede procesar nuevas entradas cada  $N$  ciclos de reloj, donde  $N$  es el intervalo de iniciación (II) del bucle o función [16].

La canalización de un bucle permite que las operaciones del bucle se implementen de manera concurrente, como se muestra en la figura 2.12. En dicha figura, (A) muestra la operación secuencial predeterminada donde hay 3 ciclos de reloj entre cada lectura de entrada (II = 3) y requiere 8 ciclos de reloj antes de que se realice la última escritura de salida, mientras que (B) muestra la operación con *pipeline* en la que hay 1 ciclo de reloj para cada lectura de entrada (II=1) y requiere tan solo 4 ciclos de reloj para realizar la última escritura [16].

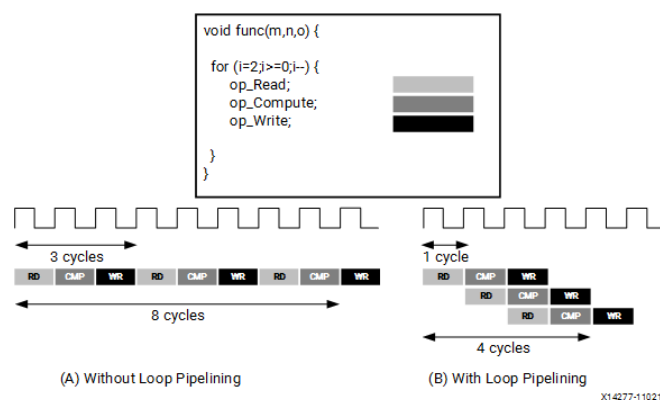


FIGURA 2.12: Ejemplo de la funcionalidad para una instrucción `pragma HLS pipeline` [16].

### 2.8.3. pragma HLS array\_partition

El `pragma array_partition` particiona una matriz o arreglo en matrices o arreglos más pequeños o elementos individuales [16]. Como consecuencia, se obtienen [16]:

- Resultados en RTL con múltiples memorias pequeñas o registros múltiples en lugar de una memoria grande.
- Aumento efectivo de la cantidad de puertos de lectura y escritura para el almacenamiento.
- Una mejora potencial en el rendimiento del diseño.
- Aumento en el número de instancias de memoria o registros.

### 2.8.4. pragma HLS interface

En el diseño basado en C, todas las operaciones de entrada y salida se llevan a cabo en tiempo cero, a través de argumentos de funciones formales. En un diseño RTL, estas mismas operaciones de entrada y salida se deben realizar a través de un puerto en la interfaz de diseño y, por lo general, funcionan utilizando un protocolo de Entrada / Salida específico [16].

El pragma `interface` especifica cómo se crean los puertos RTL a partir de la definición de la función durante la síntesis de la interfaz. La sintaxis del pragma se puede ejemplificar mediante [16]:

```
#pragma HLS interface <modo>port= <name>register
```

En donde `<modo>` especifica el modo de protocolo de interfaz para los argumentos de función, las variables globales utilizadas por la función o los protocolos de control de nivel de bloque. Algunos de los modos usados en este proyecto son `axis` que implementa todos los puertos como una interfaz AXI4\_Stream y `s_axilite` que pone en funcionamiento todos los puertos como una interfaz AXI4\_Lite [16].

Mientras que `port=<name>` especifica el nombre del argumento de la función, el retorno de la función o la variable global a la que se aplica el pragma `interface`. Por último, `register` es una palabra clave opcional para registrar la señal y cualquier señal de protocolo relevante y permite que las señales persistan hasta al menos el último ciclo de la ejecución de la función [16].

## 2.9. AXI

AXI es parte de ARM AMBA, una familia de controladores de interfaz que se introdujo por primera vez en 1996. Su segunda versión fue lanzada en 2010 e incluye la segunda versión principal de AXI4. AXI4, a su vez, cuenta con tres interfaces AXI4, AXI4-Lite y AXI4-Stream [17].

- **AXI4:** para requerimientos de alto rendimiento mapeados en memoria.
- **AXI4-Lite:** para una comunicación mapeada en memoria de bajo rendimiento.
- **AXI4-Stream:** para datos de transmisión de alta velocidad.

### 2.9.1. AXI4

La interfaz del protocolo AXI4 consiste en cinco diferentes canales, tal como se aprecia en la figura 2.13. Estos son: el bus de dirección de lectura, el bus de lectura de datos, el bus de dirección de escritura, el bus de escritura de datos y el bus de respuesta de escritura.

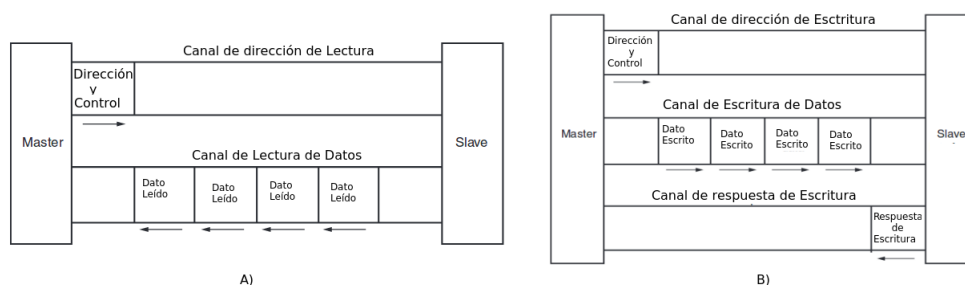


FIGURA 2.13: Diagrama de la arquitectura del canal de Lectura (A) y canal de Escritura(B) para el protocolo AXI4 [17].

De este modo, tal como se muestra en la figura 2.13, AXI4 proporciona conexiones de datos y direcciones separadas para lecturas y escrituras, lo que permite



la transferencia de datos de forma simultánea y bidireccional. Algunas otras de sus características son: transmisión por ráfaga (secuencia de datos en cola), ampliación y reducción de datos, múltiples direcciones pendientes y procesamiento de transacciones fuera de orden. El protocolo AXI4 ofrece entre sus ventajas lograr un rendimiento de datos muy alto [17].

### 2.9.2. AXI4-Lite

AXI4-Lite es similar a AXI4, es decir, usa el mismo diseño de transmisión mostrado en la figura 2.13, con algunas excepciones; la más notable es que no es compatible con la transmisión por ráfagas [17].

### 2.9.3. AXI4-Stream

El protocolo AXI4-Stream se utiliza para aplicaciones que normalmente se centran en un paradigma centrado en datos y flujo de datos de tipo productor-consumidor en el que el concepto de una dirección no está presente o no es necesario. Cada AXI4-Stream actúa como un único canal unidireccional para un flujo de datos de *handshake*, es decir define un solo canal para la transmisión de datos de transmisión, tal como se muestra en la figura 2.14. El canal AXI4-Stream se modela después del canal de escritura de datos del AXI4. A diferencia de AXI4, las interfaces AXI4-Stream pueden transmitir por ráfaga una cantidad ilimitada de datos [17].

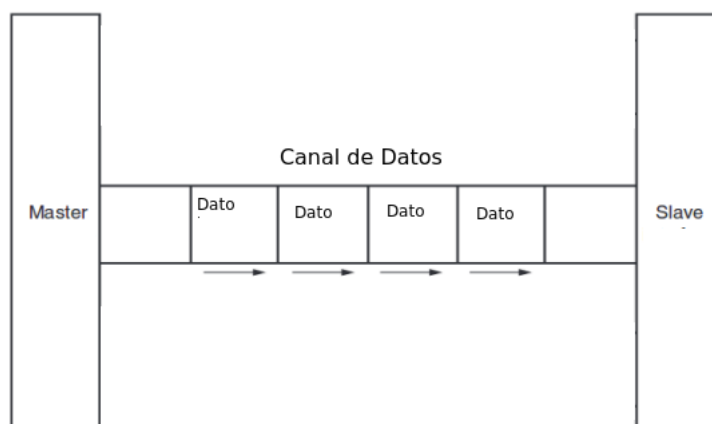


FIGURA 2.14: Diagrama de la arquitectura del canal de Lectura y Escritura para el protocolo AXI4-Stream. Adaptado de [17].

## 2.10. Biblioteca HLS Stream

HLS Stream es una biblioteca creada en C++ que esta basada en la interfaz AXI4-Stream de Xilinx. Los *stream* generados a raíz de la biblioteca al ser diseñados en una función y sintetizados en *hardware* se implementan como un FIFO con profundidad finita en *hardware* e infinita en *software* [14].

El funcionamiento se basa en el comportamiento de un FIFO, tal como se muestra en la figura 2.15, donde un *buffer* FIFO consiste en una instalación de almacenamiento “elástica” entre dos subsistema. Una FIFO tiene al menos dos señales de control escritura y lectura. Cuando se realiza la operación de escritura, el dato de entrada es escrito en el *buffer*. La operación de lectura actúa como una señal de eliminar. Cuando es ejecutada, el primer elemento del FIFO se elimina y el próximo elemento se dispone a la cabeza del FIFO como disponible [18].

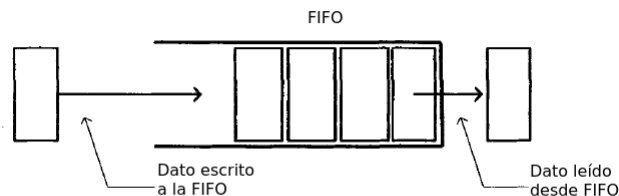


FIGURA 2.15: Diagrama conceptual del funcionamiento de la biblioteca HLS Stream [18].

## 2.11. Complejidad algorítmica

El tiempo de ejecución para un algoritmo debe definirse como una función que depende de la entrada, en particular, de su tamaño. El tiempo requerido por un algoritmo expresado como una función del tamaño de la entrada del problema se denomina complejidad en tiempo del algoritmo y se denota  $T(n)$ [19].

De manera general, la complejidad  $T(n)$  de un algoritmo es de  $O(f(n))$ . En muchos casos, la complejidad de tiempo de un algoritmo es igual para todas las instancias de tamaño  $n$  del problema. En otros casos, la complejidad de un algoritmo de tamaño  $n$  es distinta dependiendo de las instancias de tamaño  $n$  del problema que resuelve. En consecuencia, se definen tres casos: el mejor caso, el peor caso y el caso promedio [19].

La tasa de crecimiento obtenida para encontrar el orden de complejidad en tiempo de un algoritmo permite entre otras cosas [19]:

- Determinar el comportamiento del algoritmo en función del tamaño del problema y reflejar un tamaño razonable del mismo.
- Determinar cuánto tiempo de cómputo aumenta al incrementar el tamaño del problema.
- Facilita la comparación de algoritmos y permite determinar entre varios algoritmos.

## Capítulo 3

# Marco metodológico

### 3.1. Metodología de diseño

Para el desarrollo de este proyecto se utilizó la técnica de diseño modular.

La metodología empleada es la *Top-Down*. La decisión de optar por esta metodología en particular radica en que esta se adapta de forma excelente a los requerimientos del presente proyecto.

Esta metodología establece que primero al diseñar el proyecto se visualiza el sistema como un todo y, sucesivamente, se va descomponiendo en sub-secciones más específicas en las que se va detallando las conexiones y señales entre ellas, hasta llegar a describir por completo la arquitectura del proyecto.

Como parte del diseño de esta investigación se empleó el desarrollo en cascada, lo cual implica que se comenzó analizando los requerimientos que este proyecto posee; después se realizó un anteproyecto en el cual se definieron las actividades y objetivos principales; posteriormente, se realizó un análisis del proyecto con el sistema neuronal existente con el fin de crear una propuesta de optimización que permitiera crear una arquitectura eficiente para la solución de los problemas propuestos en el capítulo 1.

### 3.2. Secuencia de pasos

Para ejecutar la propuesta, se realizó un análisis del proyecto base, el cual está definido en [5] y [4]. El análisis se fundamenta en la sección 4.1 de "Análisis y medición de las partes del modelo ION en el algoritmo eHH". El análisis se basa en la medición de tiempos de ejecución para los componentes del modelo ION, es decir, axón, soma y dendrita, en esta última, las GJ son separadas y analizadas por separado. Lo anterior se realiza, ya que los resultados obtenidos en [9] y la información detallada en el capítulo 2 muestran que las GJ son las causantes de un mayor consumo de recursos computacionales.

Una vez que se realizó el análisis, se crea una estrategia, para resolver el problema mediante el desarrollo en una arquitectura basada en FIFO. Para llevar a cabo la solución se separa las GJ del resto de las componentes del modelo ION. La estrategia está definida en la sección 4.2.

Posteriormente, dentro de la misma secuencia de pasos, una vez que la estrategia ha sido implementada, esta se analiza nuevamente con el fin de insertar mejoras, pero, en esta ocasión mediante técnicas HLS, en las cuales se incluye el proceso explicado en la sección 4.3, en el que se crea una mejora con la creación de una técnica *pipeline*.

Del mismo modo, en la sección 4.4, se propone el medio para la conexión entre el IP creado con la estrategia de optimización y la placa de desarrollo Zedboard, en esta sección se incluye la comunicación vía DMA.

Al tener el proceso de la creación de un IP con las GJ separadas y ejecutadas en *hardware* completado, se realiza una estratificación de las otras componentes del modelo ION, soma, axón y dendrita para crear un medio que permita la ejecución de los procesos en paralelo, mediante el uso de la biblioteca *pthread* del lenguaje de programación C. Los resultados de este paso son mostrados en la sección 4.5.

Por último, como medio de obtención de resultados precisos, se realiza en la sección 4.6 una comparación de los resultados obtenidos en el presente proyecto con los resultados obtenidos en otros proyectos de los trabajos descritos en [9] y [5], y explicados con detalle en el capítulo 2.

### 3.3. Estrategias de validación

Para realizar la validación de los pasos descritos anteriormente, se realizaron en su mayoría las pruebas de caja blanca, las cuales son estrategias centradas en los detalles de los procedimientos del *software* implementado, es decir, en el código fuente. Para ejecutarlas, se eligieron distintos parámetros de entrada para cada una de las etapas y de dicha manera se pretende examinar cada uno de los posibles flujos de ejecución del programa y cerciorarse de que se devuelven los valores de salida adecuados.

### 3.4. Herramientas y técnicas

Para ejecutar este proyecto, se utilizaron herramientas tanto en *software* como en *hardware*, según las necesidades durante el proceso. A continuación, se detalla el o los procesos en las cuales se utilizaron, junto con las herramientas necesarias.

En la primera sección de análisis del modelo ION descrita en 4.1, al igual que en la sección 4.6, se aplicaron técnicas de graficación estadística con ayuda de herramientas como LibreOffice, Octave y Microsoft Office, las cuales permitieron a través de una serie de datos obtenidos modelar el comportamiento de la red neuronal.

Asimismo para las secciones 4.2, 4.5 y 4.3 se utilizaron las herramientas de Gedit para desarrollar el código en C y Vivado HLS, para modificar e implementar por técnicas HLS la estrategia basada en FIFO para la separación del código GJ. De igual manera, se utilizaron técnicas de co-simulación y monitores de forma de ondas, las cuales permitieron realizar estrategias de validación de caja blanca para predecir o extrapolar los resultados previos que brindaron una orientación adecuada durante

el resto del proceso.

En la integración del IP generado en Vivado HLS, con la placa de desarrollo ZedBoard se utilizaron otras herramientas detalladas a continuación:

- Vivado IDE que permite crear un bloque de diseño modular de los diferentes componentes (DMA, IP, Procesador, etc. ) que se utilizarán en *hardware*.
- PetaLinux que ofrece la posibilidad de crear e implementar soluciones para *hardware* embebidos. En este caso, mediante PetaLinux, se crea el árbol de dispositivos (*deviceTree* en inglés) y el driver para acceso a memoria mediante el DMA.
- SDK Xilinx, que se utiliza para crear una imagen de inicio (BOOT) en formato FSBL.
- SD o la tarjeta flash que se utiliza como herramienta de *hardware* para el almacenamiento de los archivos necesarios para ejecutar el código GJ en la placa ZedBoard.
- Linaro, versión de Linux que es utilizada para crear una imagen y un sistema de archivos en la tarjeta SD.
- Minicom, herramienta muy útil que permite la conexión vía puerto serie UART a la ZedBoard, para configurar el dispositivo a través del puerto de consola.
- GParted, herramienta que permite crear, redimensionar y eliminar particiones en la SD.

Por último, un paso muy relevante en el proceso consiste en utilizar el sistema operativo Linux con la distribución Ubuntu 16.04 LTS, además de la Terminal de comandos, la cual sirvió como herramienta de intersección, monitoreo, desplazamiento, visualización, entre otras tantas acciones.

## Capítulo 4

# Resultados de la investigación

En este capítulo, se muestran los resultados obtenidos a raíz de la necesidad de optimizar y depurar la red neuronal albergada en el proyecto ZedBrain, con la realización de una mejora en tres factores principales: escalabilidad, tiempo de ejecución y flexibilidad del sistema.

### 4.1. Análisis y medición de las partes del modelo ION en el algoritmo eHH

Tal como se ha observado en los capítulos anteriores, el proyecto ZedBrain [4], [5], en el cual se fundamentan los objetivos de este proyecto, está basado en el modelo Hodgkin Huxley extendido. Este modelo propone una red neuronal compleja y biológicamente precisa, en la que cada célula está conformada por tres componentes: soma, axón y dendrita.

Se ha determinado anteriormente en el capítulo 2 que las GJ son responsables de un gran número de operaciones matemáticas, tal como se muestra en la figura 2.6, y que su tiempo de ejecución crece de manera cuadrática en función del número de neuronas interconectadas en la red a simular. Nótese por ejemplo, en la figura 2.5, cómo el cálculo de las GJ en una simple red de 96 neuronas, consume el 68.7 % del tiempo de ejecución. Por eso, el análisis en busca de potenciales soluciones se orientó hacia esa zona del algoritmo ION.

Para realizar el análisis, se tomó como referencia la simulación del modelo ION en una arquitectura x86, con procesador i7-3632QM de tercera generación que trabaja a una frecuencia de 2.20 GHz con 8 GB de memoria RAM, versus la simulación en la arquitectura ARM con procesador Cortex-A9 que posee la placa de desarrollo Zedboard y en la cual fue implementando el proyecto ZedBrain. Los resultados de ambas simulaciones se aprecian en la figura 4.1, en la que como punto importante se puede observar una curva de tendencia cuadrática conforme aumenta el número de neuronas en ambos casos, pero con una base de cálculo mucho más pronunciada para la arquitectura ARM, pues esta estructura es bastante inferior que la x86 de referencia.

En [1] se compara de igual manera el procesador ARM de la ZedBoard con los procesadores x86 más tradicionales. Allí se puede notar que el cortex-A9 de ARM para la ZedBoard parte con desventaja, ya que posee una ejecución en orden con menos segmentos, menos velocidad, menos caché y menos rutas de proceso numérico, es decir, no es superescalar.



FIGURA 4.1: Gráfica de comparación de tiempos de ejecución para el modelo ION del proyecto ZedBrain, en una arquitectura x86 y una arquitectura ARM.

Se realizó posteriormente un estudio con el objetivo de comparar los tiempos de ejecución por separado de las componentes de soma, axón y dendrita del modelo ION. En la figuras 4.2 y 4.3, se observa el tiempo de ejecución para llevar a cabo la simulación por componentes en la arquitectura x86 y ARM, respectivamente.

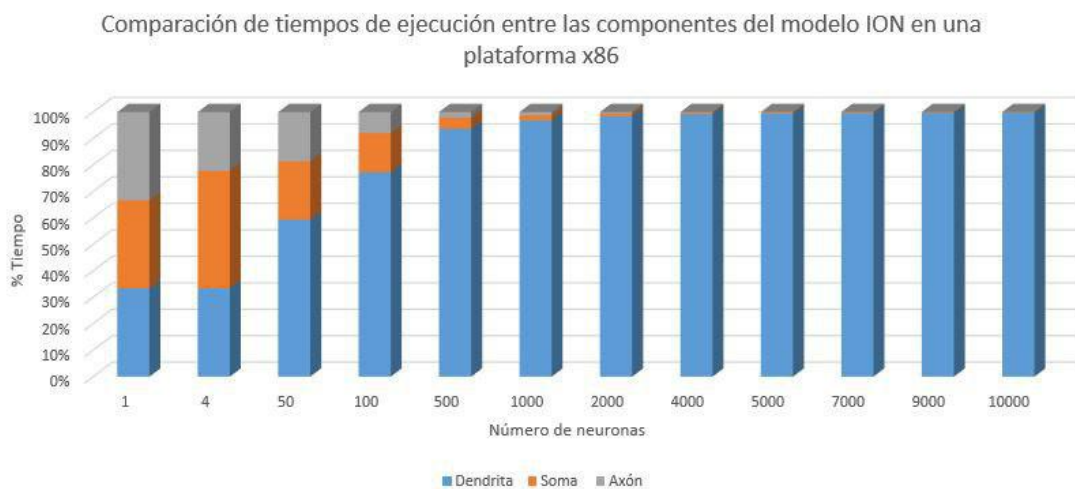


FIGURA 4.2: Gráfica de comparación de tiempos de ejecución para las componentes del modelo ION del proyecto ZedBrain, en una arquitectura x86.

De las figuras 4.2 y 4.3, se puede apreciar cómo el componente de la dendrita consume una mayor cantidad de recursos y, en consecuencia, conlleva un mayor tiempo de ejecución. Asimismo, se aprecia que posee una tendencia que entre mayor sea el número de neuronas simuladas, mayor será el tiempo de ejecución para la componente dendrita, hasta el punto, que para una simulación de 10000 neuronas, el tiempo de ejecución para la componente dendrita es de 99 %.

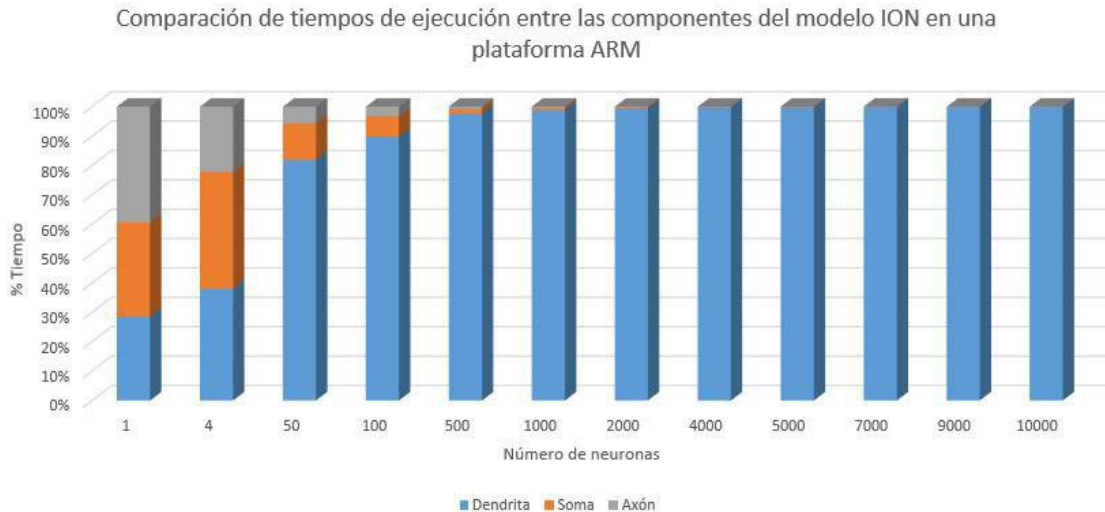


FIGURA 4.3: Gráfica de comparación de tiempos de ejecución para las componentes del modelo ION del proyecto ZedBrain, en una arquitectura ARM.

Con el fin de identificar en su totalidad el problema y de esa manera poder plantear una propuesta de solución, se realiza una última medición en donde se separa las GJ de la componente dendrita y se miden nuevamente los tiempos de ejecución. La gráfica con lo descrito anteriormente se muestra en la figura 4.4.

La gráfica de la figura 4.4 muestra que el tiempo de ejecución, en su mayoría para el proyecto ZedBrain en un ambiente de *software*, es consumido por la función encargada del cálculo de las GJ. En la figura 4.4, al comparar las componentes dendritas con la ejecución de los cálculos para las GJ (línea roja), con la componente dendrita sin realizar los cálculos para encontrar las GJ (línea celeste), se observa que la componente dendrita por sí sola consume un tiempo similar a las otras componentes soma y axón. Pero al incorporar las GJ, el tiempo aumenta cerca de 40 %, coincidiendo con las premisas planteadas en el trabajo [9] expuesto en el marco teórico, en el que se cita que los cálculos para las GJ establecen la ruta crítica de tiempo de ejecución y, por tanto, más impactan en el rendimiento para el modelo ION.

Con base en dicha información, se resuelve y se plantea una estrategia fundamentada en aprovechar alguna técnica de paralelismo, la cual se detalla en la sección 4.2 para resolver el problema encontrado en los tiempos de ejecución para las GJ. Asimismo, a partir de las diferentes gráficas mostradas, con las conclusiones inferidas de estas, se argumenta que lo más razonable es centrar la estrategia de optimización en la mejora o distribución de tiempos de la manera más adecuada, con el fin de lograr una mejora en la escalabilidad, en el tiempo de ejecución y, a la vez en la flexibilidad.

## 4.2. Desarrollo de una estrategia FIFO para la separación del código GJ

Considerando que la herramienta de trabajo, la placa de desarrollo ZedBoard posee una cantidad de recursos de computo limitados y, en consecuencia, afecta la escalabilidad en el número de neuronas a simular, se pretende crear una estrategia



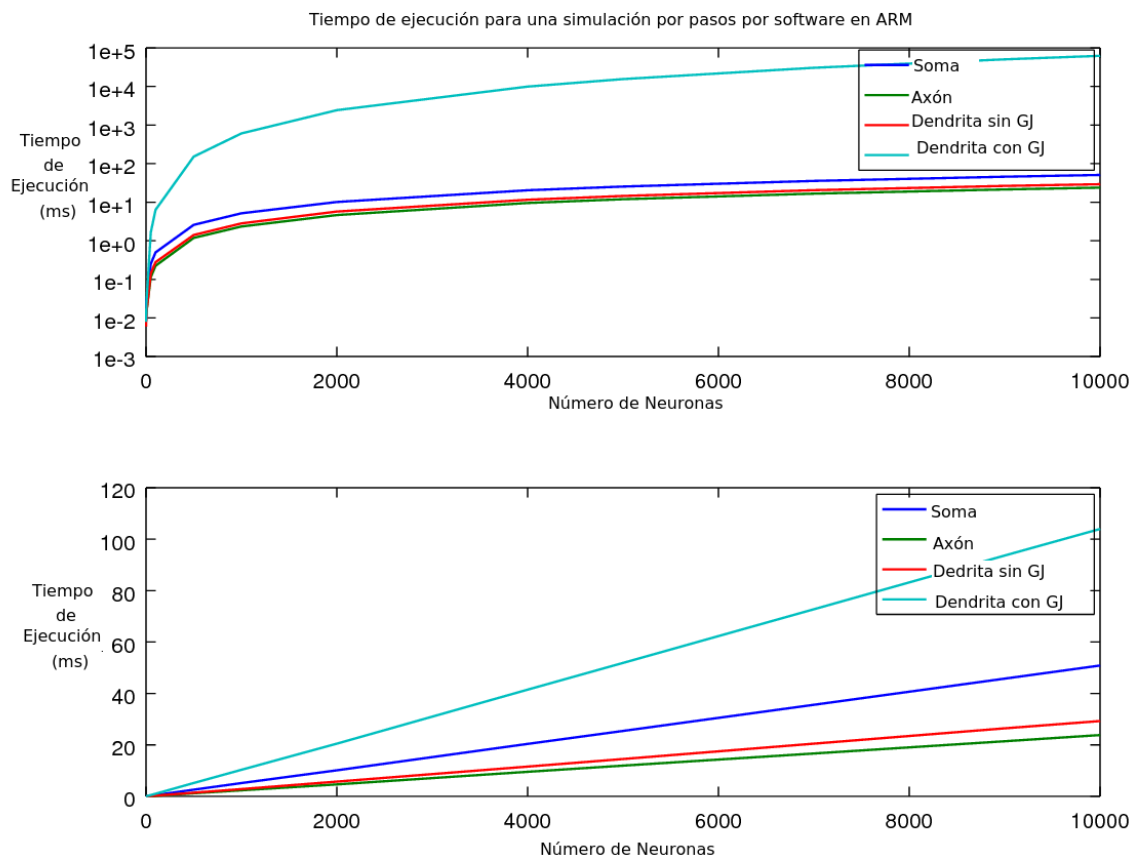


FIGURA 4.4: Gráfica de comparación de tiempos de ejecución para las componentes del modelo ION del proyecto ZedBrain, con GJ diferenciado, en una arquitectura *software*.

que busque aprovechar al máximo sus capacidades con la distribución de las tareas según un análisis de eficiencia para cada sección de los recursos, en función de cada tarea respectiva.

Al analizar el código mostrado en el marco teórico, en la figura 2.2, visto con más detalle en la figura 4.5, se puede apreciar que los cálculos matemáticos para las GJ se basan en cuatro diferentes ecuaciones: la primera  $V$  describe una resta de potenciales de las dendritas; la segunda  $f$  requiere una componente exponencial y cuadrática; la tercera  $F_{acc}$  y cuarta  $V_{acc}$  representan una multiplicación de los resultados de las primeras dos, en conjunto, con un factor de la matriz de conductancias.

Del anidamiento de los dos ciclos `for` en el código de la figura 4.5, ya se puede inferir de dónde viene la dependencia cuadrática del tiempo de ejecución del tamaño de la red. Nótese que para cada corriente sináptica, llamada `I_c` en el código, se deben realizar `N_size` iteraciones, en donde `N_size` corresponde al número total de neuronas a simular, de tal modo que si, por ejemplo, se desea calcular las conexiones para 1000 neuronas, implica que cada neurona individualmente debe realizar una iteración con cada una de sus vecinas es decir 999 iteraciones. Ahora bien, tómese en cuenta que, si la interconexión se da con todas las neuronas de la red, esto significa un millón de iteraciones de un cálculo pesado.

```
mod_prec IcNeighbors(mod_prec neighVdend[MAX_TIME_MUX+MAX_NEIGH_SIZE],
    int N_Size ,mod_prec Connectivity_Matrix[CONN_MATRIX_MAX]){

    int i,j;
    mod_prec f, V, I_c, V_acc, F_acc;

    //constante
    mod_prec const hundred = -1/100.0;

    I_c = 0.0f;
    //Asociación con las Vdendritas vecinas
    IcNeighbors_label0: for(i=0;i<N_Size;i++){
        IcNeighbors_label0: for(j=0;j<N_Size;j++){
            V = neighVdend[i] - neighVdend[j];
            f= V*expf(V*V*hundred);
            F_acc+=f*Connectivity_Matrix[i][j] ;
            V_acc+=V*Connectivity_Matrix[i][j] ;
        }
    }
    // Corriente sináptica
    I_c = (0.8 * F_acc + 0.2 * V_acc);

    return I_c;
}
```

FIGURA 4.5: Código en detalle de la comunicación ejercida en las GJ para el modelo ION, tomado del proyecto ZedBrain.

Debido a lo anterior, se decide separar el código GJ, mostrado en la figura 4.5 para ejecutarlo en una unidad externa al procesador (implementada como IP sobre el PL del Zynq), que realice estos cálculos en paralelo mientras el procesador continúa con el resto de los componentes. Será necesario, eso sí, evaluar la dependencia de datos y secuencia entre todos los componentes del algoritmo y, sobre todo, evaluar las necesidades de transferencia de datos entre el PS y el PL para salvar potenciales cuellos de botella que se traigan abajo las ganancias supuestas en tiempo de ejecución total.

Para obtener más detalle de los costos de complejidad algorítmica, se generó la tabla 4.1, en la que claramente se apunta a subdividir de alguna manera el anidamiento de estos dos ciclos `for` para así atenuar la complejidad cuadrática.

Línea de código	Accesos a memoria	Operaciones	Complejidad
mod_prec const hundred=-1/100.0	0	2	O(1)
I_c = 0.0f;	0	1	O(1)
for (i = 0; i <N_Size; i++)	0	3	O(n)
for (j = 0; j <N_Size; j++)	0	3	O(n)
V = prevV_dend[i] - neighVdend[j];	2	2	O(1)
f = V * expf(V * V * hundred);	0	5	O(1)
F_acc+=f*Connectivity_Matrix[i][j];	1	3	O(1)
V_acc+=V*Connectivity_Matrix[i][j];	1	3	O(1)
I_c = (0.8 * F_acc + 0.2 * V_acc);	0	4	O(1)
return I_c;	0	1	O(1)

CUADRO 4.1: Tabla de complejidad de las instrucciones para el algoritmo GJ, según los accesos a memoria y operaciones algebraicas.

Formalmente, de la tabla 4.1, se puede deducir que el algoritmo cuenta con una complejidad algorítmica de:

$$O(f(n)) = 23n^2 + 8 \approx O(n^2) \quad (4.1)$$

por lo que el tiempo de ejecución  $T(n)$  se puede interpretar como :

$$T(n) = O(n^2) \quad (4.2)$$

En este orden de ideas, es claro que se debe dividir el problema en cálculos más pequeños con dos objetivos claros: uno subdividir las tareas y dos utilizar algún tipo de paralelización de cálculo sobre los datos.

Para mejorar el tiempo de ejecución, se realiza entonces una partición en dos sub-tareas: la primera la de implementar un algoritmo para GJ capaz de subdividir los trabajos de funcionamiento; y la segunda implementar un protocolo de transmisión de datos entre el IP y el PL capaz de soportar una transmisión constante de datos en cola. Ambas sub-tareas se explican en las siguientes secciones.

#### 4.2.1. Implementación protocolo AXIS en un IP Core

Para generar un flujo constante de datos se introdujo el protocolo AXIS, el cual deriva de la interfaz AXI4\_Stream, explicada en más detalle en el capítulo 2, que sirve como medio de comunicación para incorporar el IP en un ambiente empotrado como la plataforma ZedBoard. Se genera además una interfaz AXI4\_Lite para las tareas de inicialización del IP y atender las interrupciones del mismo una vez terminado el procesamiento.

En la figura 4.6, se muestra la implementación de la interfaz AXI4\_Lite en un código C++, su traslado a un IP ejecutable en el PL sigue el proceso ya descrito en el Capítulo 2. La figura 4.6 muestra una representación gráfica del acceso a IP generado en el mapa de memoria. Además, se aplica el código C++ mediante el uso del pragma `interface` para generar la interconexión dentro del SoC.

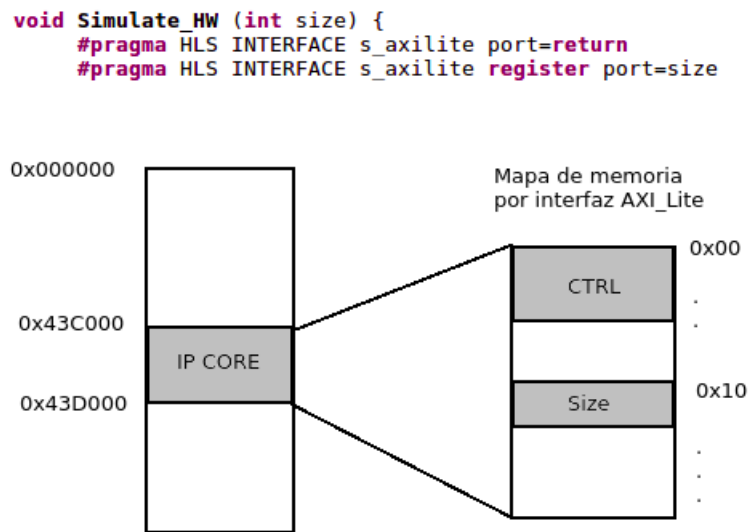


FIGURA 4.6: Diagrama del mapa de memoria en el SoC donde se sitúa la interfaz AXI4\_Lite en el IP. Se muestra además el código C++ con los pragmas necesarios para generar esta interconexión dentro del SoC . Figura adaptada de [5].

Ahora bien, para realizar el traspaso de las matrices necesarias de cálculo (la matriz de conductancias que representan la interconectividad entre las neuronas en la red), se implementa la interfaz `AXI4_Stream` mediante la biblioteca `HLS_Stream.h`, ya que esta produce un *buffer* con comportamiento FIFO, es decir, que resulta esencial. Además, la biblioteca `HLS_Stream.h` provee una estructura productor-consumidor que permite un acceso constante al IP de los datos necesarios de la matriz sin necesidad de que el micro intervenga. De este modo, el IP va procesando conforme se va alimentando y al final produce su resultado.

Con el fin de implementar la interfaz `AXI4_Stream` en conjunto con la biblioteca `HLS_Stream.h`, fue necesario la configuración del protocolo de la interfaz, así como la utilización del IP AXI-DMA para de procesar los datos desde la RAM. Ambos procesos son detallados en los siguientes párrafos.

#### ■ Configuración del protocolo de la interfaz `AXI4_Stream` con el uso de la biblioteca `HLS_Stream` en la plataforma Zynq-7000

La figura 4.7 representa, desde el punto de vista de transferencia de datos desde el ARM, la interfaz entre el IP y la matriz de interconexiones en RAM. Nótese la arquitectura productor-consumidor del lado derecho construida por medio del `AXI_DMA` y la estructura `AXI4_Stream` controlada desde la biblioteca `HLS_Stream.h`. El IP va recibiendo el flujo continuo de datos y va procesando hasta entregar el resultado por medio de una interrupción (que el ARM atiende a través de la interfaz `AXI4_Lite`).

En esta misma figura, se muestra a nivel de interconexión la relación entre los canales necesarios que se implementan entre el IP y el controlador de DMA. Del mismo modo, se detallan las banderas para la correcta configuración del protocolo entre las dos partes, las mismas son: TLAST, TVALID, TREADY y TDATA. La señal TLAST indica el final de una cadena de envíos, TVALID y TREADY determinan el control de cada dato enviado, mientras que, por último, TDATA controla el contenido de los datos transmitidos.

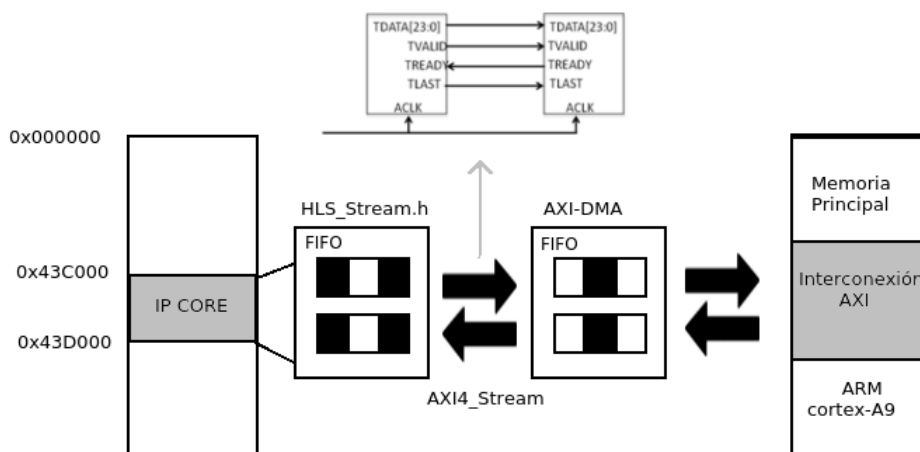


FIGURA 4.7: Diagrama de bloques sobre el manejo del IP con la interfaz AXI4\_Stream y controlado por el DMA, mediante la biblioteca HLS\_Stream.

- **Configuración del protocolo de la interfaz AXI4\_Stream con el uso de la biblioteca HLS\_Stream en el IP**

El uso de la biblioteca HLS\_Stream requiere de un protocolo específico para el *hardware*. Este debe configurarse primero mediante Vivado HLS y luego en Vivado (tal como se mostró en la figura 4.7).

El AXI4\_Stream permite dos configuraciones válidas para el manejo del flujo de datos desde un lenguaje HLS con intermediación de código C++. La primera configuración es sencilla y solamente utiliza el manejo de dos señales de control: TVALID que asegura la lectura de un dato válido y TREADY que informa cuando el puerto está nuevamente habilitado para un nuevo dato. Sin embargo, esta configuración no es la utilizada en este proyecto, ya que en complemento con la biblioteca HLS\_Stream la configuración supone envíos de paquetes de datos de tamaño constante por lo que el *handshake* es mínimo.

La segunda configuración utiliza las señales de control `TREADY` y `TVALID`, pero, adicionalmente, agrega otras señales de control `TDEST`, `TKEEP`, `TUSER`, `TID`, `TSTRB` y `TLAST`. Esta última señal es la más relevante para términos de este proyecto y es la única señal adicional que se utiliza. Esta señal identifica el envío del último dato de una ráfaga, o bien el final de un bloque continuo de datos enviados. `TDEST`, `TUSER` y `TID` permiten controlar datos con múltiples canales de transmisión, mientras que `TSTRB` y `TKEEP` permiten identificar o señalar los *bytes* válidos de un dato en especial [17].

Para utilizar las señales adicionales en código C++, es necesario construir una plantilla que permita el uso de dichas señales, o bien importar la biblioteca `ap_axi_sdata.h` que contiene el control de las señales. No obstante, en este proyecto se opta por crear dicha plantilla, ya que la biblioteca `ap_axi_sdata.h` restringe el uso de datos a únicamente de tipo `INT`.

En la figura 4.8, se muestra el resumen de una implementación de código C++ de la interfaz `AXI4_Stream`, la cual utiliza la biblioteca `HLS_Stream` en un ambiente de Vivado HLS. En esta, se visualiza el uso de las señales de control implementadas para un flujo constante de datos. En el ejemplo, el IP recibe dos señales de flujo de datos mediante el tipo de datos *stream*, propio de la biblioteca `HLS_Stream`. Las señales *stream* (*input*, *output*) que actúan con comportamiento FIFO son inicializadas con el protocolo `AXI4_Stream` mediante la instrucción `pragma HLS interface` que es una línea con acceso directo al procesador y utiliza en este caso el modo `axis`. Después, se realiza una lectura de datos provenientes de la estructura FIFO, asignándolos a un resultado, el cual mediante una comprobación ratifica la última posición activando la señal de control `TLAST`. El ejemplo tiene como último paso escribir los resultados en la FIFO de salida. Obsérvese también el uso de una plantilla necesaria para la tipificación de los datos mediante el uso de las señales de control, dicha plantilla recibe como parámetros una estructura `<int D, int U, int TI, int TD>`, en donde las componentes están en unidades de bit y `D` es el tamaño para el dato, `U`, `TI` y `TD` son la inicialización para las señales `TUSER`, `TID` y `TDEST`, respectivamente. En consecuencia, para este proyecto, la entrada que destaca se trata de `D`, que corresponden a los 32 bits manejados para un dato de tipo `FLOAT`. Las señales restantes son administradas según la documentación oficial y se pueden observar con más detalle en [17].

```

template<int D,int U,int TI,int TD>
struct ap_axis{
    float data;
    ap_uint<(D+7)/8> keep;
    ap_uint<(D+7)/8> strb;
    ap_uint<U> user;
    ap_uint<1> last;
    ap_uint<TI> id;
    ap_uint<TD> dest;
};

typedef ap_axis<32, 1, 1, 1> data_t;

void Simulate_HW(hls::stream<data_t> &input, hls::stream<data_t> &output){

#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE axis port=input
#pragma HLS INTERFACE axis port=output

while (!input.empty()){
    data_t result = input.read();
    if (last_data) {
        result.last = 1;
    } else {
        result.last = 0;
    }
    output.write(result);
}
}

```

FIGURA 4.8: Código que ejemplifica el uso de la interfaz AXI4\_Stream en código C++, en un ambiente Vivado HLS, mediante la utilización de la biblioteca HLS\_Stream.

Como última instancia en la implementación del código C++ para AXI4\_Stream utilizando la biblioteca HLS\_Stream, se puede emplear la herramienta de visualización de formas de onda de Vivado HLS, ya que esta ayuda a identificar el comportamiento correcto de las señales de control. Véase el ejemplo de la figura 4.9, en la que se visualiza en color celeste la activación de la señal TLAST (de 0, que habilita el flujo de datos, a 1, que informa el último dato de la transmisión), así como las señales TREADY, de color amarillo, y TVALID, de color morado, las cuales se activan para indicar el flujo de recepción de datos y para la indicación de los datos válidos respectivamente. En TDATA se muestra cada dato escrito al bus, para cada escritura realizada permitiendo de forma sencilla la comprobación de una correcta implementación.

#### 4.2.2. Desarrollo de algoritmo GJ para subdivisión de tareas

La segunda etapa de la estrategia consiste en la modificación del algoritmo GJ. Tal como se ha analizado previamente en el proyecto, fue implementada una interfaz AXI4\_Stream. Dicha interfaz mediante la biblioteca HLS\_Stream se ejecuta con un comportamiento FIFO para el flujo de datos. En consecuencia es esencial entender el comportamiento del algoritmo Ic\_Neighbours que implementa el cálculo de los GJ para una re-utilización y optimización eficiente.

En la figura 4.10, se observa el comportamiento del algoritmo, el cual puede ser visualizado como una matriz matemática, en la que las potenciales de las dendritas marcan el hito de las columnas y filas y, consecuentemente, las conductancias conforman los demás elementos de la matriz. También se pueden visualizar las operaciones matemáticas que son necesarias para llevar el cálculo de las corrientes que se obtienen como resultados. Por ejemplo, en el caso de una iteración del un ciclo

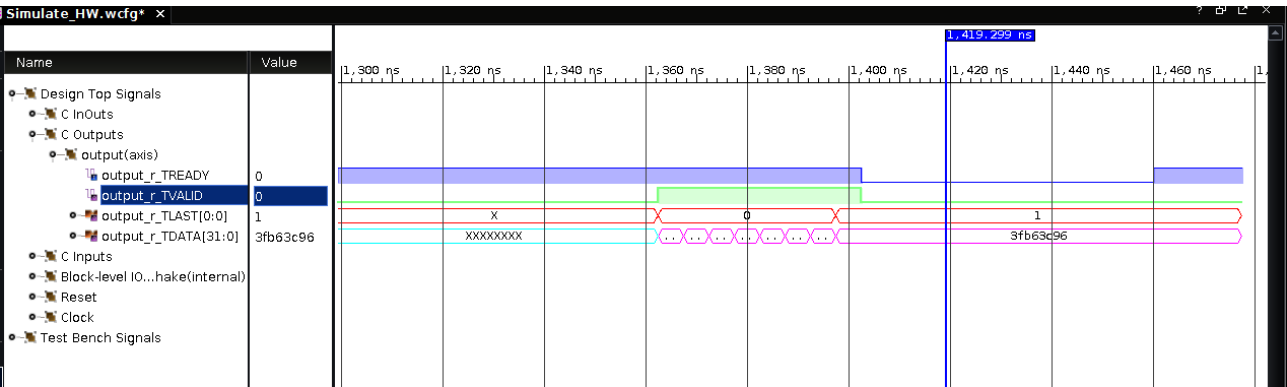


FIGURA 4.9: Estudio de las señales de control para el protocolo AXI4\_Stream mediante el uso de las herramienta de visualización de formas de onda en el ambiente Vivado HLS. Nótese la línea de color rojo que muestra la activación de la señal TLAST, así como las señales TREADY, de color morado y TVALID de color verde, las cuales se activan para indicar el flujo de recepción de datos y para la indicación de los datos válidos, respectivamente. En TDATA vemos cada dato escrito al bus, para cada escritura realizada.

para la instrucción `for`, se debe realizar la resta de las dendritas ( $V_2 - V_1$ ). El resultado se utiliza para la operación de únicamente una conductancia en cada proceso. El algoritmo posee un comportamiento matricial por filas. Es decir, para cada corriente de salida, es necesario una iteración con todos los elementos de dicha fila, o bien, visto desde lo que ocurre biológicamente, es necesario emular la reacción de la neurona en función de las conexiones en sus sinapsis con los potenciales de las neuronas vecinas.

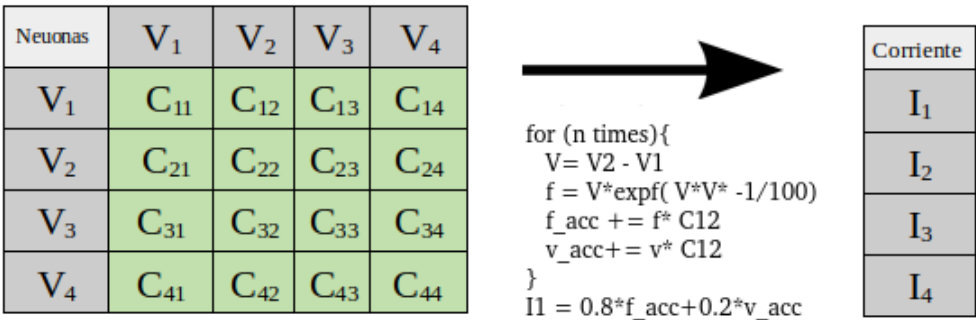


FIGURA 4.10: Diagrama representativo de la funcionalidad del algoritmo GJ, el cual realiza un recorrido matricial por filas, compartiendo información con cada elemento de la fila.



Al tratarse de una matriz en la que su primer elemento de la columna solo interacciona con los demás elementos de su fila, es posible subdividir la tarea en pequeños bloques o matrices que conformen subconjuntos de la matriz principal. Al realizar el proceso descrito, se puede lograr trabajar en pequeños bloques de información subdividiendo el trabajo en pequeñas tareas. Lo anterior, en principio, significa un ajuste de la ecuación 4.2 que al verse fraccionada, se reduce entre una constante  $N$ , donde  $N$  es el tamaño del bloque subdividido para las tareas de la matriz. La nueva ecuación que resultaría del proceso sería entonces:

$$T(n) = \frac{O(n^2)}{N} \quad (4.3)$$

El proceso de manera gráfica de la estrategia para la subdivisión de tareas se visualiza en la figura 4.11, en la que se puede apreciar que la idea básica consiste en agrupar bloques constantes de datos, los cuales ejecuten los cálculos necesarios para la obtención de la corriente en etapas. En la figura 4.11, se muestra un ejemplo en el que se toma un bloque de  $N$  igual dos (es decir una matriz de  $2 \times 2$ ). Dicho bloque procesa totalmente el cálculo de la resta de las dendritas, así como el del factor  $f$ . Sin embargo, los factores  $f_{acc}$  y  $v_{acc}$  que son acumulativos y dependen de los demás elementos de la fila, se calculan parcialmente. Hasta que se torna el final de una fila se realiza el cálculo final, seguido de la corriente resultante. El ejemplo tendría como resultados el proceso mediante cuatro bloques para la obtención de cuatro corrientes, fraccionando así el tiempo de ejecución en cuatro.

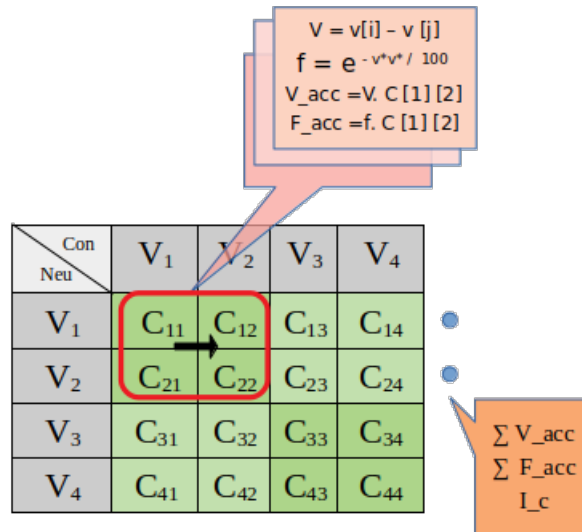


FIGURA 4.11: Diagrama representativo de la estrategia de subdivisión de tareas llevada a cabo en el algoritmo GJ, para ser compatible con un comportamiento FIFO que permita una mayor escalabilidad y eficiencia en el proyecto.

Cabe destacar que el proceso permite obtener las corrientes resultantes de una manera anticipada sin la necesidad de esperar el cálculo de la totalidad de las corrientes para proceder con las operaciones necesarias que completan el modelo ION. Es decir, tal como se muestra en la figura 4.11, una vez se recorre una fila de la matriz, se realizan los cálculos para obtener la corriente resultante de esa fila (un ejemplo es que la corriente de la fila uno en la figura es totalmente independiente de la corriente de la fila dos y, en consecuencia, se logra obtener la corriente resultante de la primera fila). Asimismo, se puede observar que el trabajo de bloques se ajusta a una mejora de optimización mediante el uso de paralelismo que se detalla en la sección 4.3.

Para cumplir con la flexibilidad del sistema requerida dentro de los objetivos del proyecto, los bloques pueden ser ajustables a un tamaño  $N$  y el sistema debe responder de igual manera.

En la figura 4.12, se detalla un diagrama de bloques de la implementación del algoritmo GJ para código C++, utilizando la biblioteca `HLS_Stream`, así como la interfaz `AXI4_Stream`.

La biblioteca `HLS_Stream` posee un conjunto de funciones tales como `empty()`, `read()`, `write()`, `size()`, entre otras que cumplen funciones específicas de comprobación de un dato para una estructura FIFO. En este proyecto, son utilizadas las funciones `read()` y `write()` que permiten leer y escribir el primer y último dato de una FIFO, respectivamente.

La particularidad de la biblioteca, aparte de que se acopla perfectamente con la interfaz `AXI4_Stream`, por su manejo de datos, es que al utilizar la función `read()` para leer el primer elemento, el dato es leído y luego borrado. Dicha función se puede interpretar como una ventaja, debido a que, al ir procesando los datos, se irán desechando aquellos que ya fueron capturados y, en consecuencia, le brinda al sistema una mayor eficiencia y capacidad de respuesta. Por ejemplo, si se desea simular alrededor de 2000 neuronas, ya no será necesario mantener todas los 2000 potenciales de dendritas necesarias, en conjunto con las cuatro millones de conductancias en memoria, sino que se irán procesando conforme se vayan requiriendo y, posteriormente, serán marcadas como basura para ser eliminadas, lo que permite elevar el número total de neuronas a simular. Posteriormente, se analizan las nuevas limitantes que eso introduce.

El diagrama de la figura 4.12 muestra en detalle el algoritmo GJ. Las entradas (color verde) y salida (color rojo) corresponden a las cola FIFO implementadas mediante la biblioteca `HLS_Stream`. En la etapa 1, se leen de la FIFO los potenciales que se necesitarán para la presente iteración, así como los potenciales que se guardan para los cálculos totales de la fila (refiriéndose a la matriz de conductancias). La segunda etapa realiza los cálculos  $V$  y  $f$ , además de recibir las conductancias de la FIFO que se necesitan para las operaciones de la etapa 3. La tercera etapa permite realizar un cálculo parcial de las componentes  $f_{acc}$  y  $v_{acc}$ , guardando los resultados temporalmente hasta el final de la fila, además de iterar el algoritmo para realizar la comunicación a todas las dendritas vecinas. La cuarta y última etapa de cálculo se realiza el procedimiento matemático para la operación de la corriente sináptica. En este punto es donde se activa la señal de control `TLAST` para confirmar la finalización del procesamiento de un conjunto de bloques.

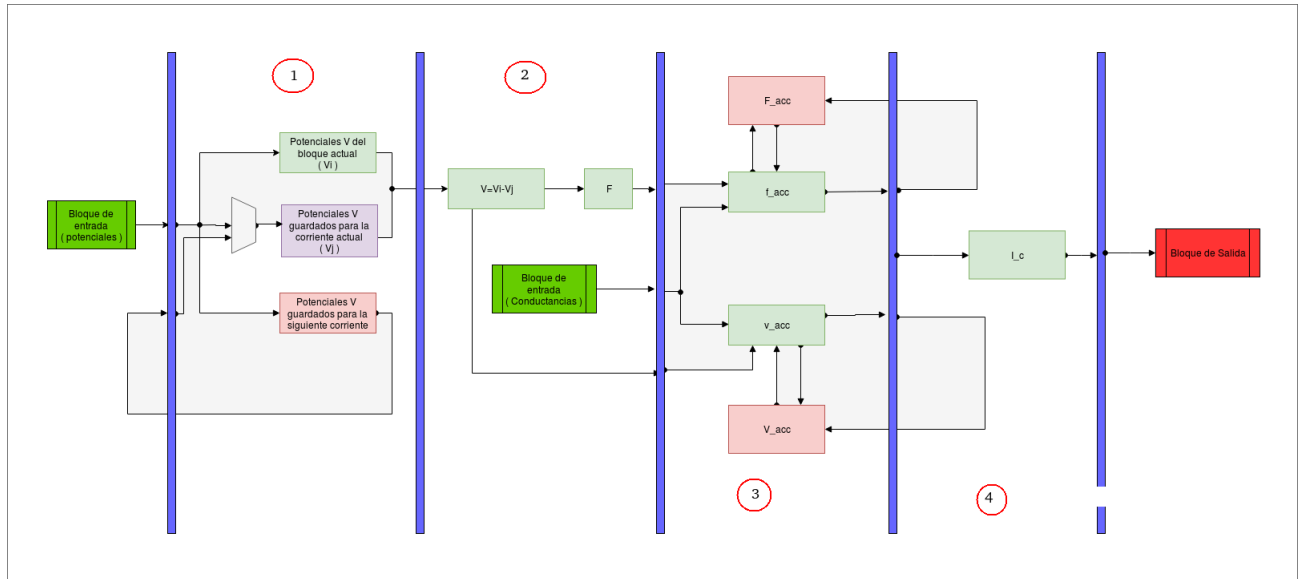


FIGURA 4.12: Diagrama de bloques de el algoritmo GJ modificado para una división por tareas, para ser compatible con una estructura de datos FIFO(productor-consumidor).

Algunos de los resultados obtenidos con la implementación de esta etapa se ven reflejados en las figuras 4.13 y 4.14, en las que se observa la cantidad de recursos consumidos para una implementación de bloques  $2 \times 2$ ,  $4 \times 4$  y  $8 \times 8$  según los datos de la síntesis de Vivado HLS 2017.4. Se nota también la tendencia de tiempos de ejecución para una cantidad limitada de neuronas, la que se obtiene mediante los tiempos encontrados en la cosimulación de Vivado HLS 2016.4, respaldada con el visor de ondas. Cabe destacar que son estimaciones según la herramienta, por lo que únicamente se realiza para 64 neuronas con el fin de comparar con las mejoras sustanciales del algoritmo en las siguientes secciones.

En la figura 4.13, se detalla el uso porcentual de los recursos BRAM, DSP, así como la cantidad de FF y LUT implementadas o usadas en la FPGA de la placa ZedBoard. Para esta implementación, se prueban con tres tipos de bloques  $2 \times 2$ ,  $4 \times 4$  y  $8 \times 8$ , que en principio, consumen porcentajes similares de recursos. De igual manera, se puede observar que el uso total de los recursos no sobrepasa el 19 %, siendo esta la cifra más alta para el recurso de DSP.

En la figura 4.14, se observan estadísticas de tiempos logradas por bloque para la implementación del algoritmo GJ dentro de un IP con las modificaciones relatadas para la subdivisión de las tareas. De la misma figura, se puede rescatar que, para el bloque de tamaño  $2 \times 2$ , conforme aumenta el tamaño del número de neuronas simuladas, el tiempo de ejecución crece hasta un 78 % más el bloque de  $4 \times 4$ . Asimismo, se aprecia con respecto al bloque de  $8 \times 8$  que, al aumentar el número de neuronas, mantiene el tiempo de ejecución más bajo, siendo este el más eficiente.

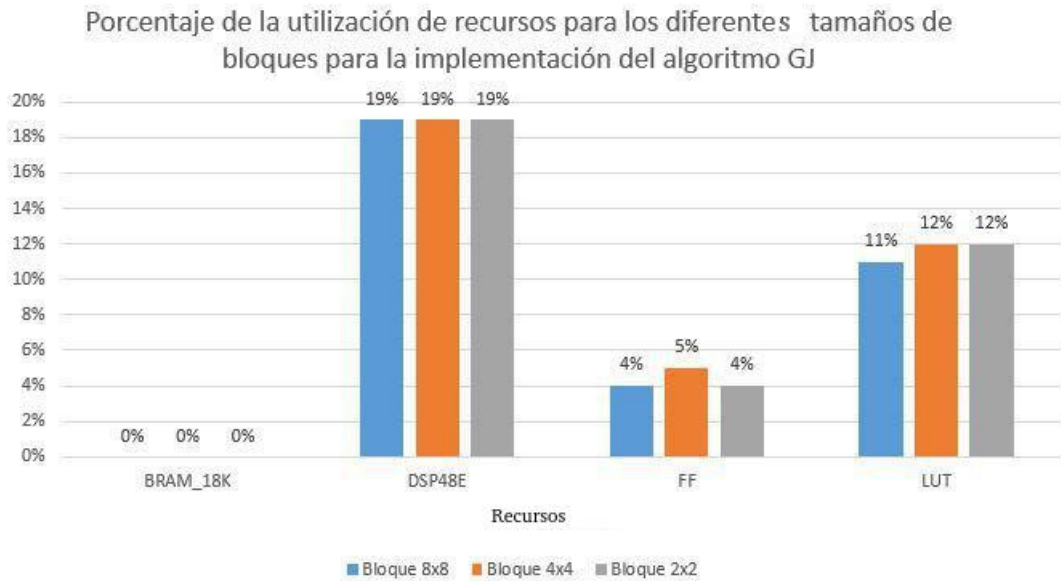


FIGURA 4.13: Gráfica porcentual del uso de los recursos para la implementación del algoritmo GJ, con bloques de tamaño  $2 \times 2$ ,  $4 \times 4$  y  $8 \times 8$ . Se puede observar que, en principio, se mantiene una similitud bastante alta para cada uno de los bloques utilizados.

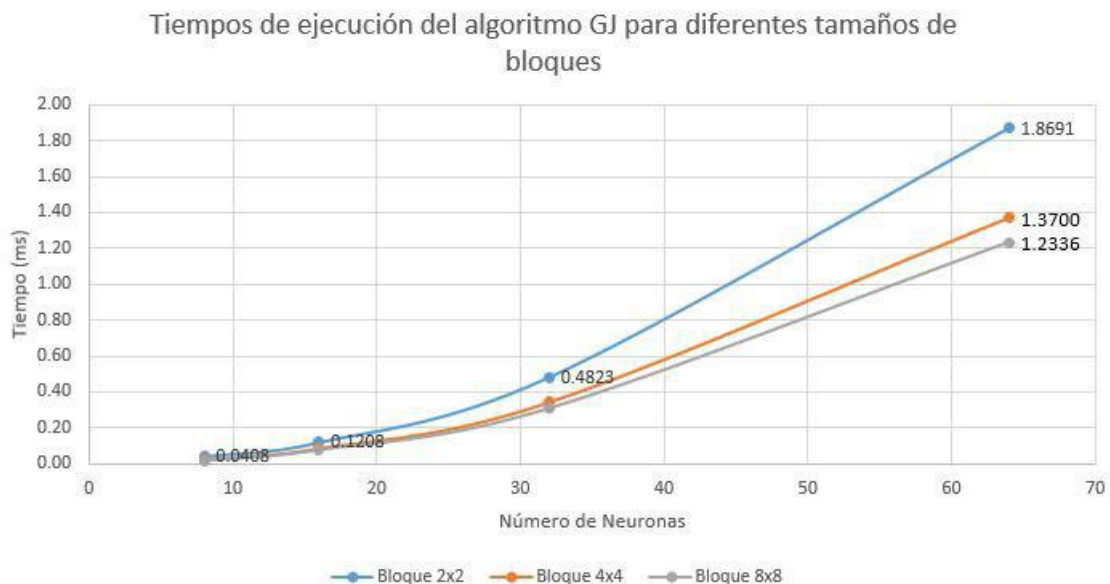


FIGURA 4.14: Gráfica comparativa de los tiempos ejecución requeridos para la implementación del algoritmo GJ con bloques de tamaño  $2 \times 2$ ,  $4 \times 4$  y  $8 \times 8$ . Nótese que el comportamiento para el bloque  $8 \times 8$  mantiene un mejor rendimiento que los bloques  $2 \times 2$  y  $4 \times 4$ .

### 4.3. Implementación de mejoras en el algoritmo GJ mediante técnicas *pipeline*

Una vez implementado el algoritmo GJ, se realizan diferentes técnicas de paralelismo para optimizarlo y lograr una mayor eficiencia. En esta fase, además, de la intención de mejorar el tiempo de ejecución, se pretende aprovechar la mayor cantidad de recursos de *hardware* posibles que la herramienta Vivado HLS posee y permite utilizar.

La implementación de tuberías es una forma de realizar paralelismo de datos, transforma un flujo de datos en un proceso comprendido por varias fases secuenciales. La entrada de cada fase secuencial es marcada por un intervalo de iniciación, el cual permite o marca el tiempo en ciclos de reloj que debe esperar el siguiente proceso para ser ejecutado.

El hecho de aplicar tuberías implica realizar un análisis previo para el manejo del control de riesgos, ya que, según la teoría uno de los supuestos, para poder aplicar esta técnica de paralelismo, es que las instrucciones deben ser independientes entre sí. Aprovechando la herramienta de análisis del ambiente Vivado HLS, se realiza un análisis de riesgos, el cual se muestra en la figura 4.15.

En la figura 4.15, se muestra el contexto en el que se desenvuelve el riesgo de realizar una lectura de un dato que no ha completado su escritura en memoria. Asimismo, en la imagen, se puede observar la porción de código en donde se ve de forma explícita que el dato albergado en la posición *i* del arreglo *savedData* es leído en la línea 53 sin haber sido escrito por la línea 43. El riesgo imposibilita la continuación del proceso de tuberías.

Para llevar a cabo la directiva `pipeline`, es necesario mitigar el riesgo. La medida permitida por la herramienta Vivado HLS es complementar la instrucción del `#pragma HLS pipeline` con un intervalo de iniciación mayor a un ciclo de reloj. En la figura 4.16, se muestra la implementación en código C++ para un IP de la directiva de control *pipeline*. En ella, se detalla la técnica de paralelismo *pipeline* con un intervalo de iniciación de 72 ciclos de reloj para contrarrestar los riesgos de datos del algoritmo modificado de GJ, tal cual fue desarrollado en la sección 4.2.

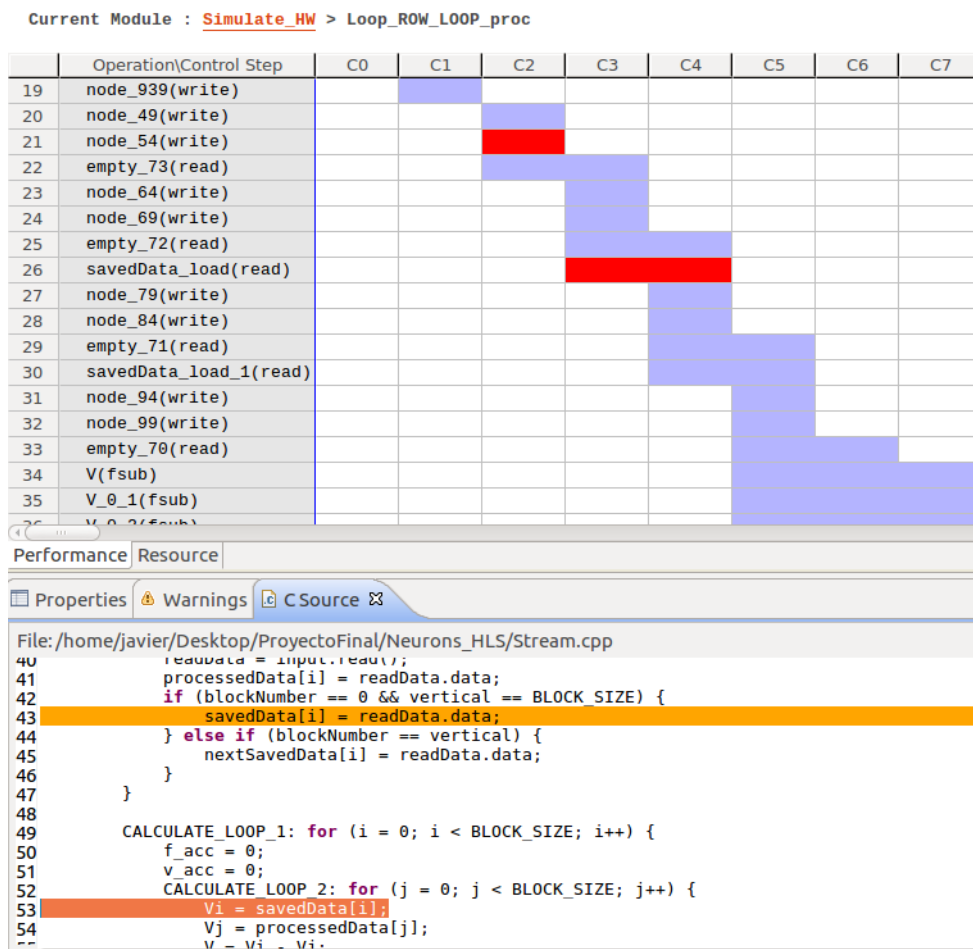


FIGURA 4.15: Representación de un riesgo de datos en el algoritmo cuando se lee un dato del arreglo *savedData* antes de que se haya terminado su escritura en memoria. Este fue encontrado durante el análisis de detección de riesgos durante la aplicación de la técnica de paralelismo de tuberías.

```

ROW_LOOP: while (blockNumber < size) {
#pragma HLS PIPELINE II=72

    float Vi = 0.0, Vj = 0.0;
    float V = 0.0, F = 0.0;
    float f_acc = 0.0, v_acc = 0.0;
    float const hundred = -1.0 / 100.0;

```

FIGURA 4.16: Representación de la directiva en código C++ para la introducción de la técnica pipeline en el algoritmo GJ.

Con la implementación realizada del *pipeline* para el algoritmo GJ, se ejecuta nuevamente el análisis de los recursos utilizados, así como el tiempo de ejecución necesario para los bloques de tamaño  $2 \times 2$ ,  $4 \times 4$  y  $8 \times 8$ . El análisis se visualiza en la figura 4.17.

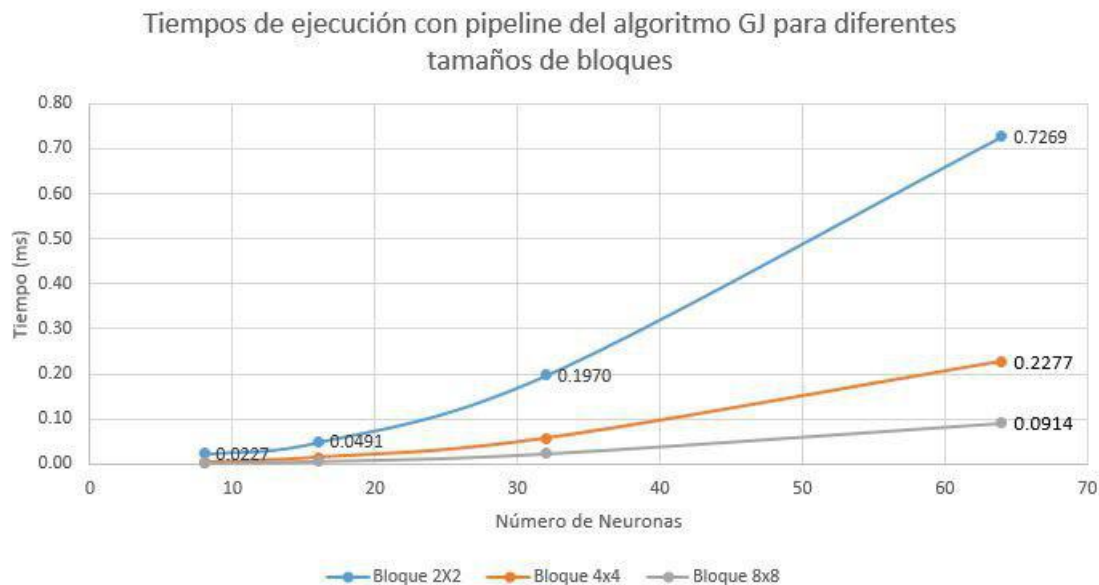


FIGURA 4.17: Gráfica comparativa de los tiempos ejecución requeridos para la implementación del algoritmo GJ con bloques de tamaño  $2 \times 2$ ,  $4 \times 4$  y  $8 \times 8$ , aplicando la técnica de paralelismo *pipeline*. Nótese que, al igual que la figura 4.14, el bloque  $8 \times 8$  muestra mayor eficiencia conforme aumenta el número de neuronas para los tiempos de ejecución.

Al analizar la figura 4.17, se puede visualizar que, en esta ocasión, de igual manera, se tiene para el bloque de  $2 \times 2$  un tiempo de ejecución mayor conforme se aumenta la cantidad de neuronas a simular, el bloque de  $2 \times 2$  mantiene una desventaja de 78 % de su rival el bloque de  $4 \times 4$ , mientras que este último, igualmente, posee un 21 % respecto al tiempo de ejecución del bloque  $8 \times 8$ . Posteriormente, se analizan en comparación los tiempos de ejecución para esta implementación y la implementación sin *pipeline*.

En la figura 4.18, se puede ver un incremento de los recursos con respecto a la figura 4.13 que realizaba una implementación sin *pipeline*. El aumento de los recursos se justifica precisamente en la implementación del *pipeline*, ya que, al dividir el proceso en etapas que se ejecutan en paralelo, es necesario aumentar la utilización de los recursos disponibles.

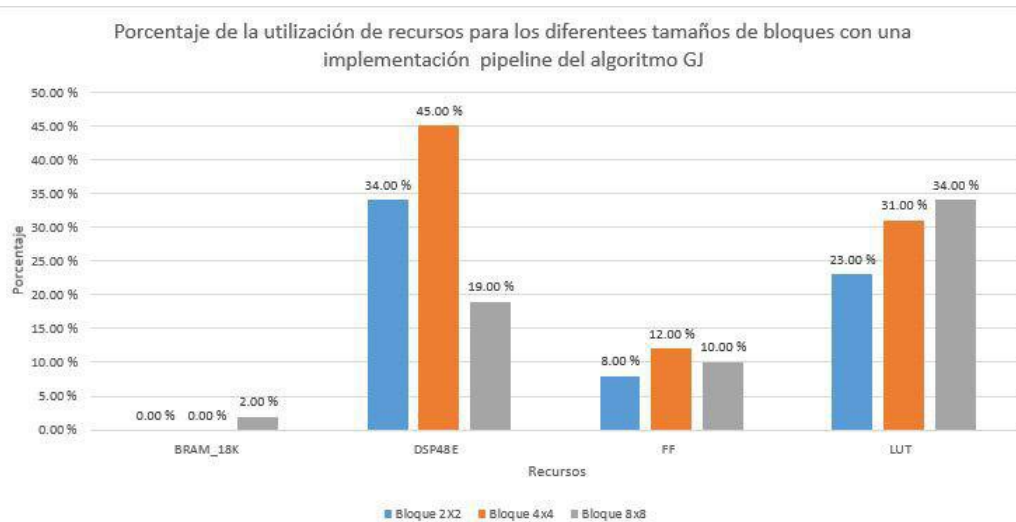


FIGURA 4.18: Gráfica porcentual del uso de los recursos para la implementación del algoritmo GJ, con bloques de tamaño  $2 \times 2$ ,  $4 \times 4$  y  $8 \times 8$ , utilizando una técnica *pipeline*. Se puede distinguir un aumento de los recursos, comparado con los mostrados en la figura 4.13. El aumento claramente es debido a la implementación del *pipeline*.

Mediante el análisis de ondas, tal como se muestra en la figura 4.19, es posible distinguir que el proceso de *pipeline* estaba optimizando el algoritmo para GJ, pero se perdía un tiempo de 980 ns entre bloque y bloque para ser leídos y ejecutados nuevamente. Como resultado, los bloques alojados en la FIFO tenían que esperar la finalización de la ejecución del bloque previo para poder procesar uno nuevo. Este suceso volvía el algoritmo ineficiente. A raíz del problema encontrado, se decide implementar un nuevo *pipeline* que complementa al ya implementado sobre el algoritmo GJ. El nuevo *pipeline* verá entre sus funciones la de optimizar y aplicar la técnica de paralelismo sobre el manejo de los bloques alojados en la FIFO.

El complementario *pipeline* se va implementando mediante la directiva `#pragma HLS dataflow` visualizada en la figura 4.20. La directiva modifica el flujo de la función *Simulate\_HW* que es la función principal en el módulo IP y por ende recibe los flujos de datos FIFO mediante la biblioteca `HLS_Stream`. El *dataflow* permite a los bloques entrantes empezar la ejecución sin que el presente bloque haya concluido. Lo anterior se logra mediante un *pipeline* a nivel de tareas que permite que las funciones y bucles se superpongan en su operación, aumentando no solo la concurrencia de la implementación de RTL, sino también el rendimiento total del diseño.

La figura 4.20, de igual manera, implementa la directiva de `#pragma HLS array_partition` para los vectores `F_acc` y `V_acc` que, tal como la figura 4.12 explica, corresponden a las variables que permiten reservar en memoria los datos temporales entre las diferentes iteraciones. La directiva permite contener el vector en memoria con múltiples puertos de lectura y escritura para el almacenamiento que aumentan efectivamente el rendimiento del diseño.

Con base en la nueva implementación de las directivas *dataflow* y *array\_partition*, se logra corregir el problema en la espera para la ejecución de los bloques de entrada y, en consecuencia, se obtienen los resultados mostrados en las figuras 4.22 y 4.21.



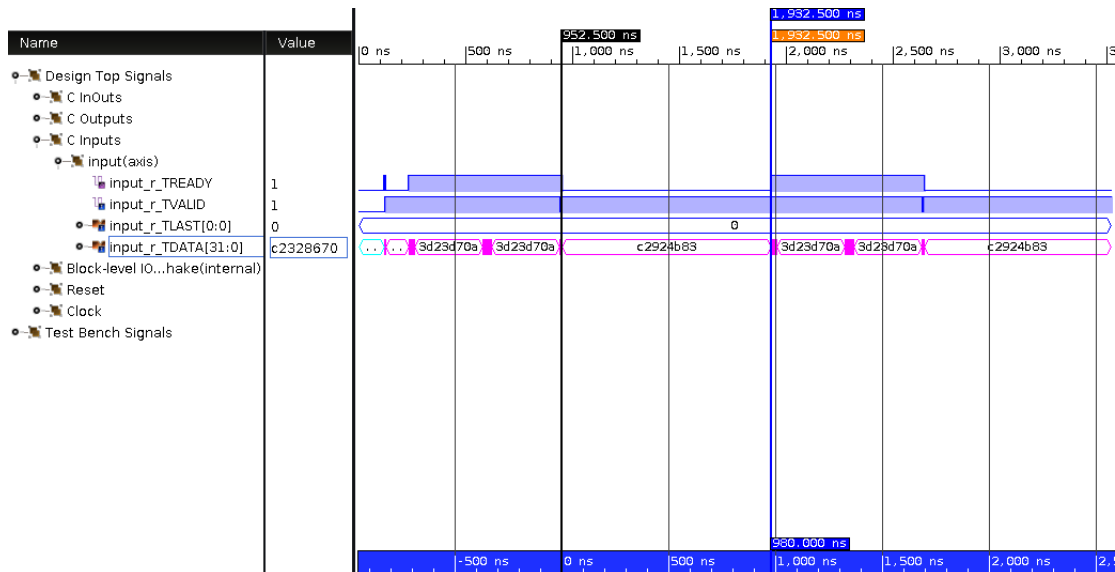


FIGURA 4.19: Análisis de formas de ondas sobre el algoritmo GJ utilizando una técnica de paralelismo de *pipeline*. Nótese una pérdida de tiempo de 980ns para la lectura entre los datos que son controlados por la bandera TDATA.

```
void Simulate_HW(hls::stream<data_t> &input, hls::stream<data_t> &output,
                int size) {

#pragma HLS DATAFLOW

#pragma HLS array_partition variable=F_acc complete
#pragma HLS array_partition variable=V_acc complete
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE axis port=input
#pragma HLS INTERFACE axis port=output
#pragma HLS INTERFACE s_axilite register port=size
```

FIGURA 4.20: Representación de la directiva en código C++ para la introducción de la directiva *dataflow* en el IP del algoritmo GJ. Esta se utiliza para la optimización de la estrategia FIFO.

En la figura 4.21, como es de esperarse según lo visto en las figuras 4.18 y 4.13, se puede apreciar un aumento en los recursos, que indica una mayor utilización de los ya ofrecidos por la placa ZedBoard.

La figura 4.21 muestra una comparativa entre el consumo de los recursos para los bloques de tamaño  $1 \times 1$ ,  $2 \times 2$ ,  $4 \times 4$  y  $8 \times 8$ , siendo apreciable un incremento en el uso de los recursos para los bloques de mayor tamaño. No obstante, al tomar en cuenta que un bloque de  $8 \times 8$  realiza el cálculo paralelo para ocho corrientes sinápticas a la vez y que un bloque de  $1 \times 1$  solo resuelve una corriente. Consecuentemente, es que se puede concluir que existe una implementación óptima que ofrece el bloque de  $8 \times 8$  (no obstante de las diferencias porcentuales, ya que estas se compensan con la capacidad de cálculo). De esta manera, en adelante se toma como referencia el bloque  $8 \times 8$  para implementar la viabilidad del proyecto. Sin embargo, cabe la posibilidad de cambiar el tamaño del bloque para garantizar la flexibilidad de acuerdo con los objetivos del proyecto.

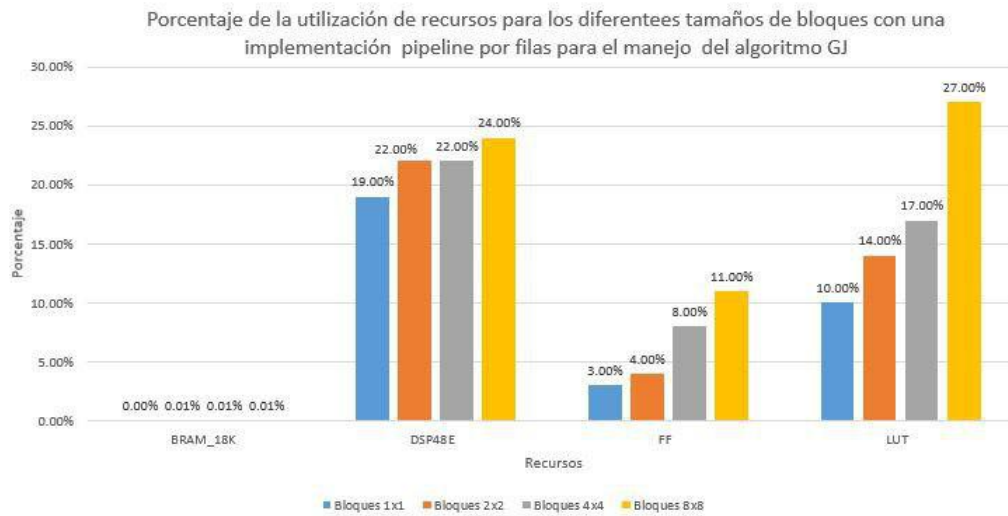


FIGURA 4.21: Gráfica porcentual del uso de los recursos para la implementación del algoritmo GJ, con bloques de tamaño  $2 \times 2$ ,  $4 \times 4$  y  $8 \times 8$ , utilizando una técnica *pipeline* y la directiva *dataflow*.

La figura 4.22 muestra la escalada de los tiempos de ejecución para los bloques de  $2 \times 2$ ,  $4 \times 4$  y  $8 \times 8$ , detallando como es costumbre desde las figuras 4.14 y 4.17 que, conforme aumenta el número de neuronas, el tamaño del bloque influye en los tiempos de respuesta, siendo el bloque  $8 \times 8$  el que muestra una mejor resolución para un mayor número de neuronas.

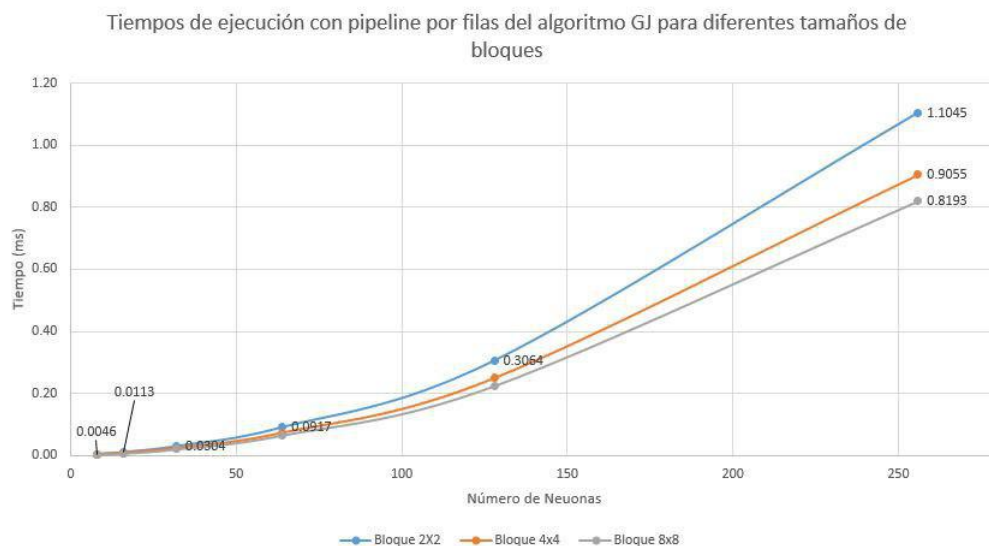


FIGURA 4.22: Gráfica comparativa de los tiempos ejecución requeridos para la implementación del algoritmo GJ con bloques de tamaño  $2 \times 2$ ,  $4 \times 4$  y  $8 \times 8$ , aplicando la técnica de paralelismo *pipeline* y la directiva *dataflow*.

Se realiza un análisis que permite determinar la eficiencia del algoritmo GJ modificado. Para el análisis, se presenta la figura 4.23 que mide la eficiencia de la implementación con base en la aceleración ofrecida para los bloques en un ambiente de simulación de hasta 264 neuronas. De la gráfica, se puede analizar que existe una tendencia creciente que busca una estabilización conforme aumenta las neuronas simuladas. Se puede apreciar que, con pocas neuronas, la pendiente de crecimiento es alrededor de 2 entre las 0 y 30 neuronas simuladas, mientras que, para las simulaciones de entre 130 y 230 neuronas, se observa una pendiente aproximada de 0.5, lo cual implica y permite concluir que la implementación es escalable y que el comportamiento para grandes simulaciones determina una estabilización en la eficiencia de este.

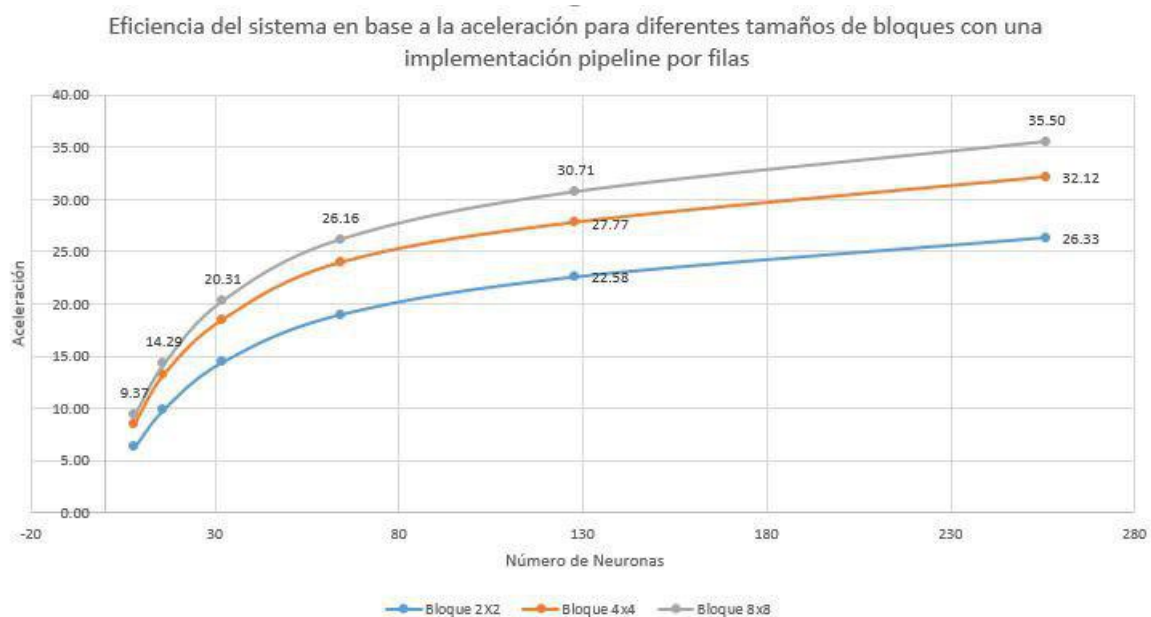


FIGURA 4.23: Gráfica de la eficiencia para la implementación del algoritmo GJ con bloques de tamaño  $2 \times 2$ ,  $4 \times 4$  y  $8 \times 8$ , aplicando la técnica de paralelismo *pipeline* y la directiva *dataflow*.

Para concluir con la optimización de *pipeline* sobre el algoritmo de GJ, se realiza una comparación entre la implementación simple del algoritmo GJ, mostrada en la sección 4.2 y la implementación mediante *pipeline* realizada en la presente sección. Para ello, se muestra las siguientes gráficas de las figuras 4.24 y 4.25.

De la primera figura 4.24, se puede observar los tiempos de ejecución llevados a cabo para el algoritmo de GJ, utilizando las implementaciones de *pipeline* (*pipeline* simple y *pipeline* por filas), en contraste con la implementación de uso simple (sin *pipeline*) sin ninguna optimización mediante paralelismo. La gráfica es meramente comparativa de acuerdo con la información recabada en las figuras 4.14, 4.17 y 4.22 que muestran los tiempos de ejecución para cada implementación.

La figura 4.25 ofrece una extracción de la figura 4.24, en la que se detalla los tiempos de ejecución llevados a cabo entre las dos implementaciones, un *pipeline* sobre el código GJ y un *pipeline* mediante *dataflow* sobre los bloques de la FIFO. Se aprecia como la simulación para números mayores de neuronas resulta más eficiente para el *pipeline* doble, mientras que el *pipeline* simple es superior en pequeñas cantidades, debido a que no es posible observar una mejoría desde el comienzo para el *pipeline* doble ya que el cálculo para redes pequeñas no requiere tantos recursos como se aplican en dicha implementación.

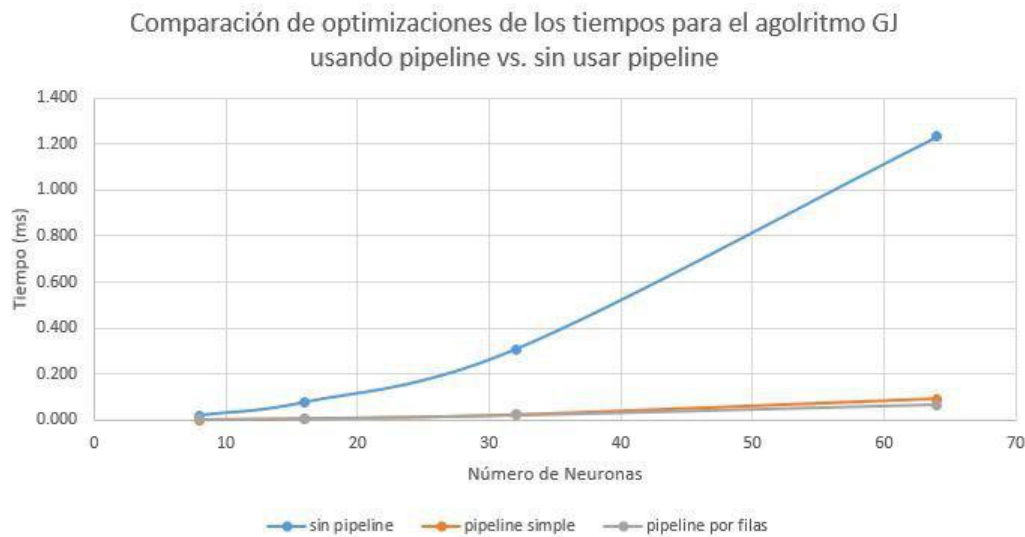


FIGURA 4.24: Gráfica comparativa de los tiempos ejecución para un bloque de  $8 \times 8$  con una implementación del algoritmo GJ previa a la utilización de pipeline vs. la implementación llevada a cabo con las optimizaciones mediante *pipeline*. Obsérvese que la implementación sin *pipeline* posee una línea cuadrática mucho más pronunciada comparada contra su homologas implementadas con *pipeline*.

La tabla 4.2 y la figura 4.26 muestran un complemento a las figuras anteriores. En ellas se puede detallar el porcentaje de ganancia de una implementación sin *pipeline*, contra las implementaciones de *pipeline* anteriormente descritas. Asimismo, se observa los tiempos absolutos para cada implementación. Se confirma que para la implementación por filas (bloques de FIFO) existe un incremento en la mejora conforme aumenta el número de neuronas, es decir, entre mayor es el número de neuronas, mejores son tiempos de ejecución respecto a la implementación sin *pipeline*.

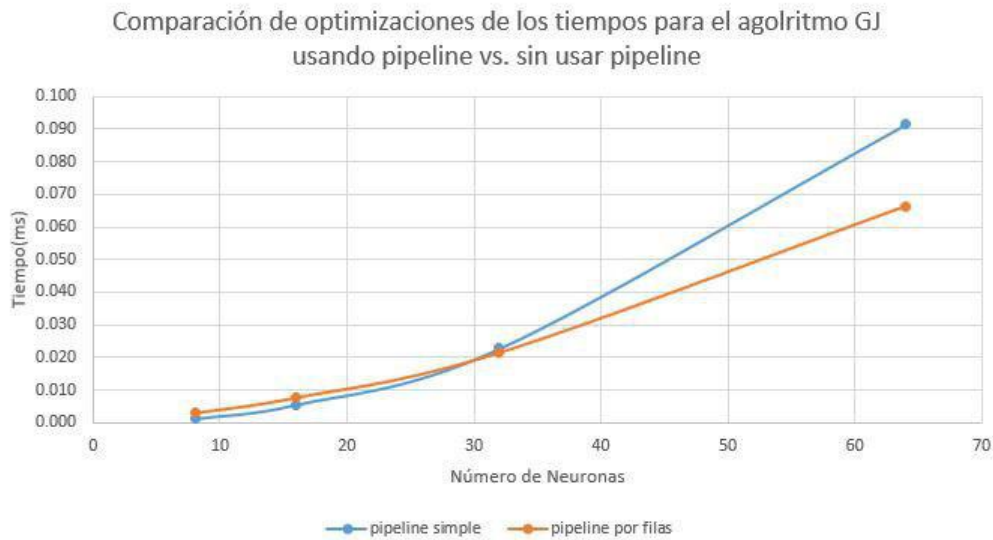


FIGURA 4.25: Gráfica comparativa de los tiempos ejecución para un bloque de  $8 \times 8$  con una implementación del algoritmo GJ con la utilización de *pipeline* simple vs. la implementación con *pipeline* por filas. Nótese que en simulaciones pequeñas los tiempos de ejecución del *pipeline* doble son más altos que el *pipeline* simple; sin embargo, al aumentar el número de neuronas existe una clara ventaja sobre el *pipeline* por filas.

Cantidad de Neuronas	Tiempo sin pipe. (ns)	Tiempo pipe. simple (ns)	Tiempo pipe. por filas (ns)
50	19375	1230	3100
100	77240	5580	7840
500	308050	22740	24620
1000	1233630	91380	66470

CUADRO 4.2: Tabla de tiempos de ejecución para las implementaciones sin *pipeline*, *pipeline* simple y *pipeline* por fila. Nótese la reducción de tiempos de ejecución para cada implementación.

La figura 4.26 es una abstracción de la tabla 4.2. Nótese que en la gráfica de la figura para simulaciones pequeñas es mejor una implementación simple. Sin embargo, para simulaciones más grandes se invierte el proceso y la implementación de *pipeline* por filas obtiene hasta 94 % de mejoras comparado contra la implementación sin *pipeline*. Lo anterior se debe a que la implementación por filas reduce el tiempo entre la espera de bloques mostrado en la figura 4.19.

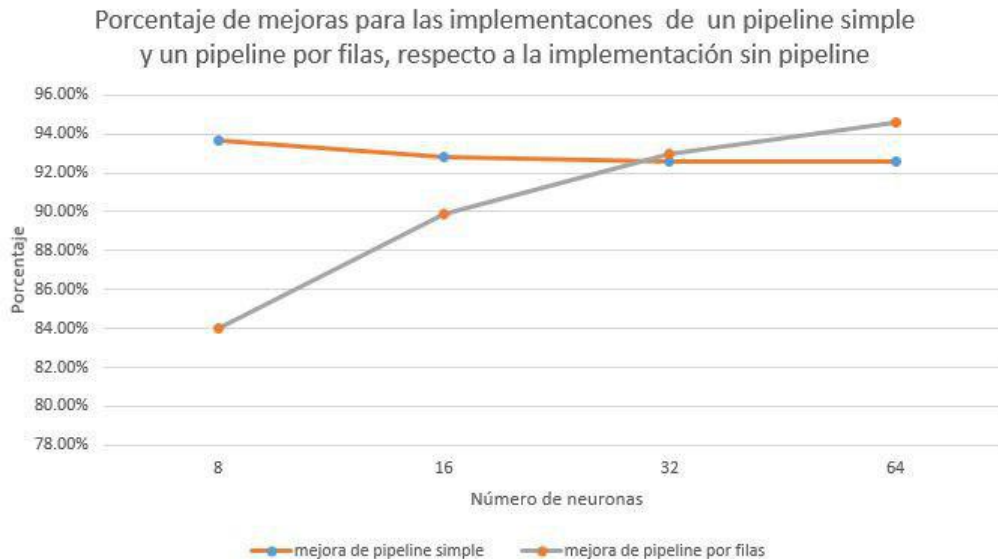


FIGURA 4.26: Gráfica porcentual de las mejoras de tiempos de ejecución para las implementaciones del algoritmo GJ con *pipeline* simple y *pipeline* por filas, contra la implementación sin *pipeline*. Nótese como la implementación con *pipeline* por filas incrementa la mejora conforme aumenta el número de neuronas.

#### 4.4. Desarrollo e integración del GJ IP Core dentro de la plataforma de prueba ZedBoard

Una vez implementada la estrategia de la sección 4.2 y optimizada mediante paralelismo en la 4.3, el IP que contendrá el RTL del algoritmo GJ se encuentra listo y se debe extraer mediante la herramienta Vivado HLS.

##### 4.4.1. Creación del archivo contenedor del código HDL mediante la herramienta Vivado

Por una parte, el IP que se visualiza en la figura 4.27 contiene una vista desde el entorno de desarrollo Vivado y se puede notar las configuraciones realizadas mediante las interfaces AXI4\_Lite y AXI4\_Stream desde el IP, asimismo se puede apreciar cómo se especifican las señales de control detalladas en la sección 4.2 sobre la conexión en *hardware* y expuestas en la figura 4.7, la cual analiza la manera correcta de una conexión para el uso de la biblioteca HLS\_Stream.

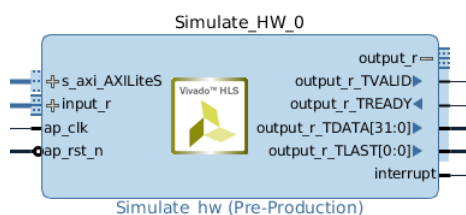


FIGURA 4.27: Módulo `Simulate_Hardware` que contiene el IP con el algoritmo de GJ modificado, visto desde el entorno de desarrollo Vivado.

Por otra parte, se puede visualizar el diagrama de bloques completo mostrado en la figura 4.28 que complementa el IP de GJ modificado para la implementación en la placa de desarrollo ZedBoard. Al igual que en la figura 4.27, en la figura 4.28 se observa la debida conexión entre el IP GJ y el IP DMA, detallando las señales de Control `TLAST`, `TVALID`, `TREADY` y `TDATA`. Esta conexión es importante, ya que, de lo contrario, el DMA interpretará la conexión estándar sin el uso de la biblioteca `HLS_Stream`, la cual no utiliza la señal `TLAST` y, en consecuencia, la lógica del IP GJ se vería afectada.

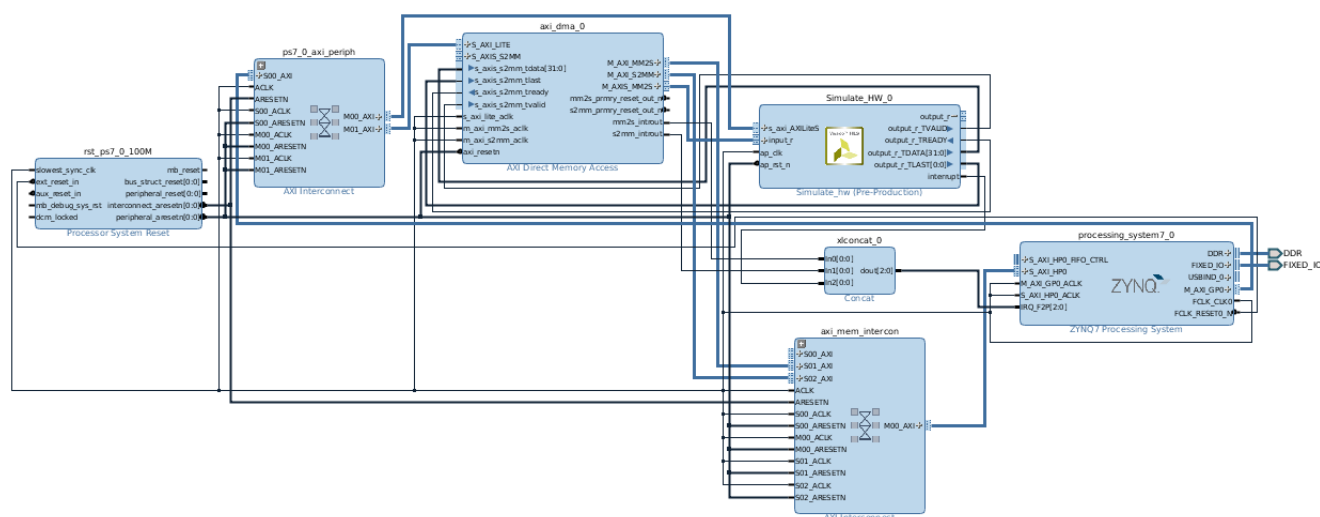


FIGURA 4.28: Diagrama de bloques sobre la implementación final hacia la placa ZedBoard, que detalla la interconexión entre las IP del algoritmo GJ y el controlador de DMA.

Una vez se crea el proyecto completo en Vivado para la plataforma Zedboard, con una meta de máxima frecuencia de reloj en el PL de 100MHz para el bloque IP GJ modificado, se trabaja en una implementación que permita un paralelismo completo. En un principio, se piensa en la alternativa de ejecutar el *hardware* mediante la herramienta SDK Xilinx, la cual ofrece un montaje *bare-metal* en la ZedBoard, esto quiere decir con un sistema operativo ausente durante la ejecución.

Dado que uno de los objetivos del proyecto de investigación en general es facilitar la integración de este sistema en un esquema HPC multi-FPGA basado en MPI, es preferible buscar el soporte del sistema operativo. Por ello, se sigue por la vía del uso del sistema operativo Linaro para poder aprovechar la biblioteca `Pthreads` y poseer las facilidades para la posterior integración con MPI a través de Ethernet y las funciones de pila TCP-IP.

#### 4.4.2. Implementación del código HDL en la ZedBoard, mediante la creación de una imagen plantada en una SD

Se procede con la creación de una imagen de Linux reducida para ZedBoard para utilizar los IP sintetizados mediante Vivado en la PL. Según el trabajo [5] citado en el capítulo 2, para construir una imagen para la Zedboard, se debe de tomar en cuenta los dispositivos en la PS, tales como periféricos, controladores DMA y memoria RAM DDR3 e interconexiones. En consecuencia, se crean dos particiones en una SD: BOOT y ROOTFS, la primera esta en formato FAT32 y contiene archivos para el arranque del sistema operativo tales como `BOOT.BIN`, `devicetree.db` y `uImage`. La segunda está en formato EXT4 y se encarga de almacenar el sistema de archivos de todos los programas, los directorios y los archivos precompilados, tales como los archivos `.C`. En la figura 4.29 se muestran las particiones antes descritas.

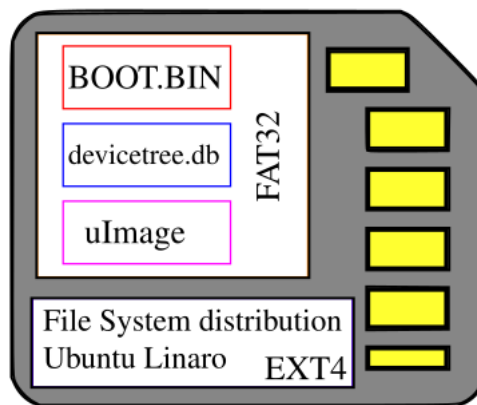


FIGURA 4.29: Diagrama del contenido requerido en una tarjeta SD para la creación de un disco de arranque mediante una distribución Linux en la Zedboard. Imagen tomada de [5].



### Partición BOOT

Tal como se mencionó, es necesario la inclusión de tres archivos en la tarjeta SD para la partición BOOT, los cuales se detallan a continuación:

- **BOOT.bin**: se encarga de generar el archivo para la iniciación o arranque del sistema, se componen de tres elementos: el primero es el **bitstream.bit** que es el archivo generado a través de la herramienta Vivado y que contiene el diagrama de bloques mostrado en la figura 4.28 y el IP para GJ modificado. El segundo archivo consiste en el archivo **FSBL.elf** que se puede generar mediante la herramienta SDK de Xilinx y es responsable de cargar el **bitstream** y configurar el sistema de procesamiento de arquitectura Zynq (PS). El tercer elemento es uno de los más importantes y, **U-boot.elf** se obtiene al compilar la imagen alojada en el repositorio BOOT, e igualmente es el encargado de ejecutar la imagen del sistema operativo.

Para generar el archivo **BOOT.bin** se debe realizar una compilación cruzada a través de las herramientas de Xilinx SDK, debido a que, una vez que se han creado los tres archivos, es necesario combinarlos en uno solo, esto se logra mediante la creación de un proyecto FSBL en la herramienta.

- **uImage**: como el nombre lo implica se debe a la imagen del sistema operativo, la cual incluye los *drivers* necesarios para la conexión con el Zynq, se obtiene al descargar y compilar la versión situada en el repositorio IMAGE.
- **device-tree.db**: aquí se reúne la información perteneciente de los periféricos de la plataforma de *hardware*, en este caso, la ZedBoard. Para generar este archivo, se hace uso de la distribución Linux de Xilinx PetaLinux, la cual por medio de sus soluciones para sistemas empujados permite realizar pruebas previas a la etapa final, tales como una correcta implementación del device-tree sobre el IP de GJ modificado.

Para crear el árbol de dispositivos, como es conocido comúnmente *device tree*, se debe modificar el bloque *amba* como se observa en la figura 4.30, en el que se deben incluir las regiones de memorias con sus direcciones físicas para las interrupciones generadas por los IP (mediante las instrucciones `pragma HLS interface`), tanto el de GJ modificado, como el del AXI\_DMA. Lo anterior permite que mediante una llamada al árbol de dispositivos el sistema pueda ejecutar las funciones, tales como el GJ modificado.

Este entorno se logra vía el controlador **UIO** que fue creado por la comunidad de Linux para casos donde los periféricos son controlados por uso de registros programables mapeados en memoria, tal como corresponde al caso de este proyecto. Para que una aplicación tenga acceso directo a estos dispositivos, este controlador simplemente ayuda a mapear mediante el árbol de dispositivos las direcciones de memoria física o virtual que el usuario desea escribir o leer.

```

Simulate_HW_0: Simulate_HW@43c00000 {
    compatible = "generic-urio";
    interrupt-parent = <&intc>;
    interrupts = <0 31 4>;
    reg = <0x43c00000 0x10000>;
    xlnx,s-axi-axilites-addr-width = <0x5>;
    xlnx,s-axi-axilites-data-width = <0x20>;
};

axi_dma_0: dma@40400000 {
    #dma-cells = <1>;
    clock-names = "s_axi_lite_aclk", "m_axi_sg_aclk", "m_axi_mm2s_aclk", "m_axi_s2mm_aclk";
    clocks = <&clkc 15>, <&clkc 15>, <&clkc 15>, <&clkc 15>;
    compatible = "xlnx,axi-dma-1.00.a";
    interrupt-parent = <&intc>;
    interrupts = <0 29 4 0 30 4>;
    reg = <0x40400000 0x10000>;
    xlnx,addrwidth = <0x20>;
    xlnx,mm2s-burst-size = <0x100>;
    xlnx,s2mm-burst-size = <0x100>;

    dma-channel@40400000 {
        compatible = "xlnx,axi-dma-mm2s-channel";
        dma-channels = <0x1>;
        interrupts = <0 29 4>;
        xlnx,datawidth = <0x20>;
        xlnx,device-id = <0x0>;
    };
    dma-channel@40400030 {
        compatible = "xlnx,axi-dma-s2mm-channel";
        dma-channels = <0x1>;
        interrupts = <0 30 4>;
        xlnx,datawidth = <0x20>;
        xlnx,device-id = <0x0>;
    };
};

axidmatest_0: axidmatest@0 {
    compatible = "xlnx,axi-dma-test-1.00.a";
    dmas = <&axi_dma_0 0 &axi_dma_0 1>;
    dma-names = "dmaproxy_tx", "dmaproxy_rx";
};

```

FIGURA 4.30: Porción de código que representa la implementación del árbol de dispositivos, se puede apreciar la interacción con los IP del GJ y el IP del DMA.

### Partición ROOTFS

Para crear la partición ROOTFS se tomó la versión de Ubuntu 15.04 pre-compilada en los repositorios de Linaro y se copió en la partición RootFS de la tarjeta SD. Esto permite la creación del fichero de archivos.

En el sistema de archivos, se deben incluir en el directorio */home* los archivos para el *driver* externo del DMA el cual se encuentra en el repositorio DMA y se encarga de entre otras cosas como iniciar el sistema de registros del DMA o la de crear los canales expuestos en el árbol de dispositivos, con los sufijos asociados a cada uno (*dmaproxy\_rx* para recepción. y *dmaproxy\_tx* para transmisión).

De igual manera, en el directorio */home* del sistema de archivos, se deben incluir los archivos .C o .C++ para las otras componentes del modelo ION, es decir, soma, axón y la dendrita sin las GJ, junto con un archivo .C que ejecuta la conexión mediante el driver DMA al IP implementado. En la figura 4.31, se muestra la vista que debería tener el directorio de la partición ROOTFS con los archivos mencionados.

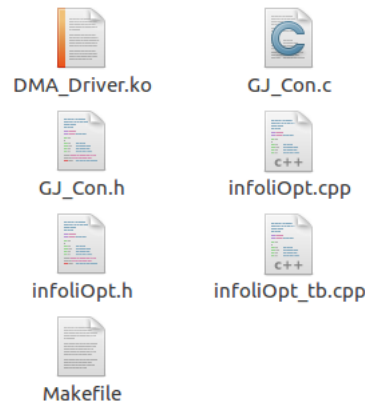


FIGURA 4.31: Vista gráfica del directorio `/home`, ubicada en el sistema de archivos para la partición ROOTFS, la cual contiene los archivos necesarios para la sincronización de todas las componentes del modelo ION con el IP para GJ.

## 4.5. Implementación conjunta de las partes del modelo ION

Ahora bien, deben integrarse las distintas secciones en que se dividió el algoritmo general eHH. Hay que ejecutar en paralelo las componentes del modelo ION, las cuales se ejecutarán vía *software* de manera integrada, con el módulo IP del GJ, de manera paralela.

Con el objetivo de lograrlo se modifica el código original planteado en el proyecto Zedbrain [4], [5] para implementar una funcionalidad mediante la biblioteca *pthread.h* que permite una ejecución de procesos simultáneos con la creación de hilos de ejecución.

La figura 4.32 muestra un diagrama del flujo principal, conjunto con los principales componentes que afectan el trayecto. En el diagrama se puede observar la ejecución de las componentes soma, axón y dendrita en paralelo mediante la biblioteca *threads* del lenguaje C++. Asimismo, se aprecia la integración del IP GJ mediante una clase que, vía el *driver* para DMA, es capaz de enviar los parámetros de las potenciales de las dendritas y las conductancias en bloques.

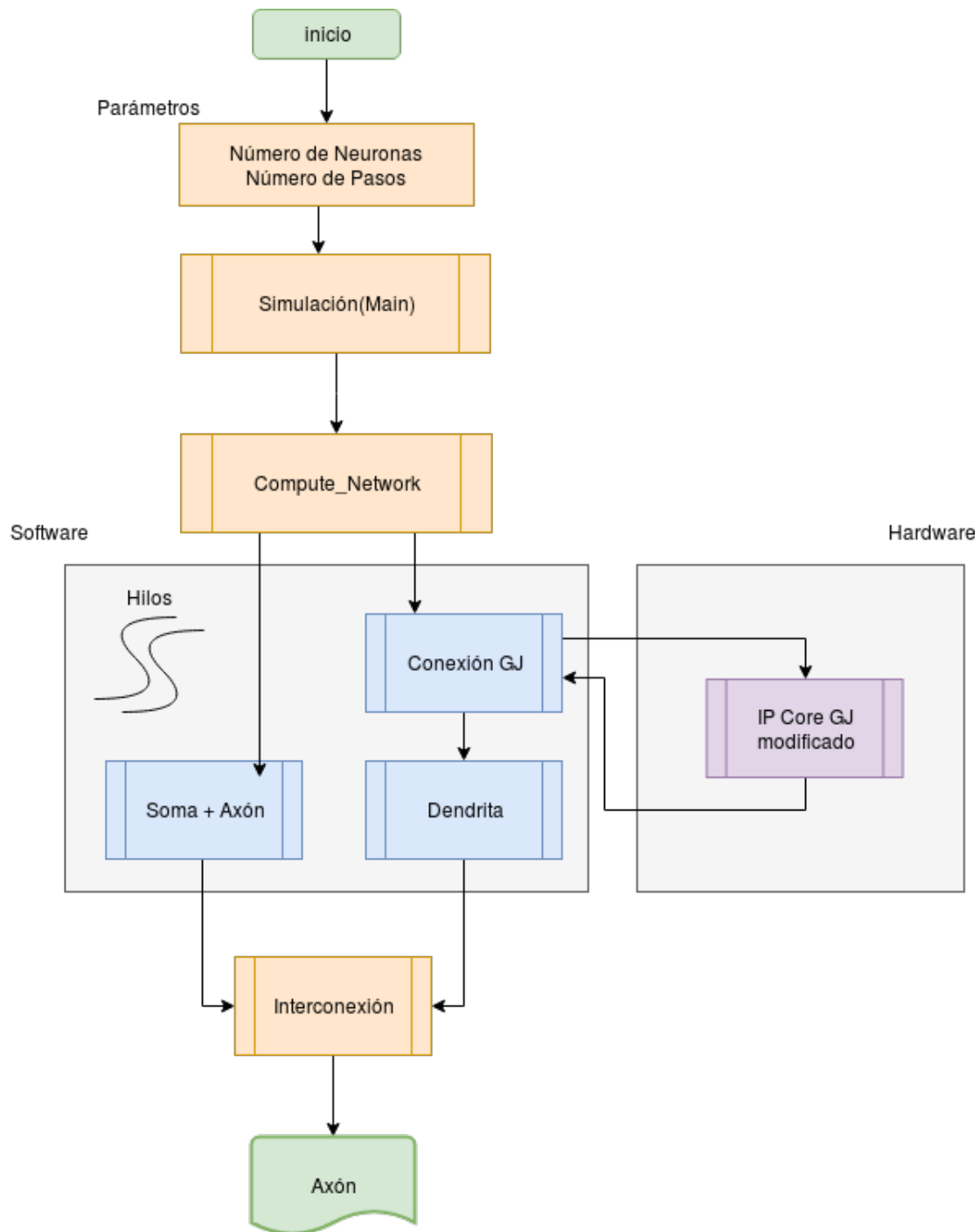


FIGURA 4.32: Diagrama del flujo que sigue el código implementado para el modelo ION, con el uso de la biblioteca *pThreads*.

Para alcanzar la integración del IP, es necesario crear una clase de conexión o *testbench* como es comúnmente conocida. En dicha clase, para este proyecto, se establece el protocolo para la comunicación con el IP AXI\_DMA, mediante un *driver* DMA. Además, es la encargada de tomar los vectores de potenciales para dendritas y las conductancias (obtenidos como parámetros), re-armarlos en bloques de 8 x 8 y manipular los bloques para obtener los resultados del *hardware*.

En la figura 4.33, se muestra la porción del código que realiza la conexión con el IP, además se envía mediante el protocolo AXI\_Lite el número de neuronas a simular. Se puede observar que en este protocolo es obligatorio precisar de las direcciones físicas de memoria para escribir o bien recibir un dato.

```
*((unsigned *)((uint64_t)Axi_Hls_network.ptr + XSIMULATE_HW_AXILITES_ADDR_SIZE_DATA)) = SIZE;
*((unsigned *)((uint64_t)Axi_Hls_network.ptr + XSIMULATE_HW_AXILITES_ADDR_AP_CTRL)) = 0x0001;
```

FIGURA 4.33: Porción de código que establece conexión mediante el protocolo AXI\_Lite para iniciar el IP o enviar parámetros necesarios en el mismo.

En la figura 4.34, se aprecia un diagrama representativo de la función para rearmar los vectores que se ejecuta en tres pasos principalmente: el primero es ubicar las tensiones de las dendritas y ubicarlas de acuerdo a la dimensión del bloque, es decir, que, si es un bloque de 8 x 8, deberá tomar solo las primeras ocho tensiones y ajustarlas en las primeras posiciones del bloque, aguardando el turno hasta que las conductancias sean escritas para poder proseguir con los potenciales. El segundo paso es buscar las conductancias pertenecientes a los potenciales que fueron ubicados en el bloque, lo cual implica una búsqueda fuera de orden, que en este trabajo se logra mediante una búsqueda por índices para evitar una recursividad innecesaria. El tercer y último paso es armar el bloque y reactivar los índices para proseguir con el siguiente bloque.

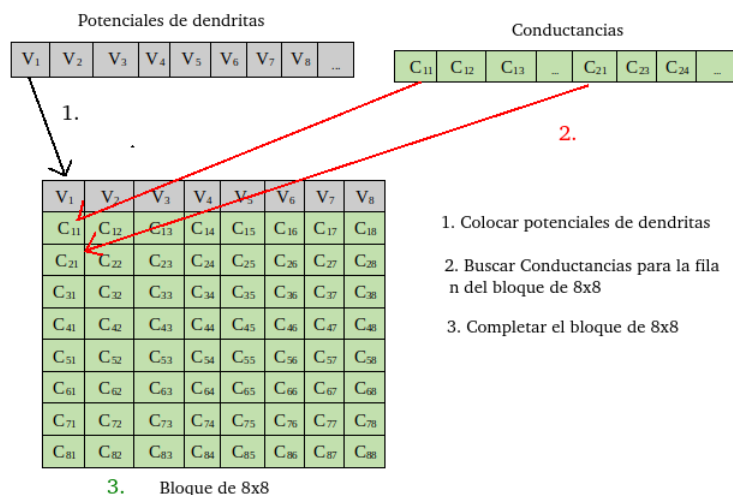


FIGURA 4.34: Diagrama representativo de la función para formar los bloques de 8 x 8 ejecutados en la IP de GJ modificada.

## 4.6. Comparación de tiempos entre implementaciones del modelo ION

### 4.6.1. Resultados obtenidos

Para culminar este proyecto y realizar un análisis de resultados que permita identificar los pasos a seguir en futuros proyectos, se busca realizar una verificación del cumplimiento de los objetivos planteados y se realizan comparaciones de los tiempos de ejecución entre los resultados que se encontraron en un inicio en el trabajo previo, contra los resultados obtenidos a lo largo y final de la optimización llevada a cabo en este documento. Asimismo se realiza simulaciones y cálculos que permitan comprobar y justificar cada uno de los resultados obtenidos.

La tabla 4.3 es producto de un conjunto de simulaciones para un creciente número de neuronas. Cada simulación se realizó para 10000 pasos lo que equivale a 500 ms de actividad cerebral. Para los 10000 pasos de simulación, se obtuvo el tiempo promedio de ejecución (segunda columna), así como el mejor y peor tiempo de simulación (tercera y cuarta columna, respectivamente). La información representa el tiempo total de ejecución para el modelo eHH. En consecuencia, los datos son de total relevancia ya que permite una comparación directa con los tiempos de ejecución del proyecto previo, dicha comparación se realiza en los siguientes párrafos, donde se discute sobre la idoneidad de los tiempos y resultados para corregir los problemas de flexibilidad, escalabilidad y eficiencia.

Cantidad de Neuronas	Tiempo de ejecución (ms)	Peor tiempo (ms)	Mejor tiempo (ms)
50	4.507	5.201	4.258
100	14.262	15.599	13.542
500	303.735	314.600	294.300
1000	1179.983	1231.44	1158.310
2000	4735.114	4870.84	4634.451
4000	18783.569	18840.265	18740.144
5000	29804.142	29815.001	29764.178
7000	58212.623	58220.654	58199.321
9000	96426.188	96580.233	96351.012
10000	118915.301	189021.782	118810.101

CUADRO 4.3: Tabla de tiempos de ejecución para 10000 pasos de simulación o 500 ms de actividad cerebral. Nótese que se tabula el peor tiempo y mejor tiempo de ejecución para los 10000 pasos de simulación. Los tiempos son obtenidos de la optimización basada en el paralelismo de las GJ con bloques de ejecución de  $8 \times 8$ .

La figura 4.35 pretende ofrecer una perspectiva más clara de los resultados de la tabla 4.3. La gráfica es se traza con el tiempo promedio de los 10000 pasos de simulación para cada ejecución de  $N$  neuronas, obtenido en base a la modificación realizada en el modelo ION (separación de las GJ). En la gráfica, se observa que los tiempos de ejecución obtenidos en este proyecto tienden a una curva cuadrática, representados mediante la ecuación  $0.00119 X^{**2} - 0.03366X + 7.83606$ . Conocer la tendencia en primera instancia ayuda a una comparación entre los tiempos actuales y

los tiempos del proyecto previo, los cuales se conocían previamente poseían igualmente una tendencia cuadrática. La comparación se detalla en mayor medida en la figura 4.36.



FIGURA 4.35: Gráfica de tiempos de ejecución finales para el modelo neuronal eHH construido en este proyecto. Nótese que la curva de los tiempos de ejecución tiene una tendencia cuadrática mostrada en la ecuación de la figura.

En la figura 4.35, se aprecia que para este proyecto se realizaron simulaciones de hasta 10000 neuronas en una sola placa ZedBoard, lo que cumple con las expectativas para el primer objetivo, pues está aumentando la escalabilidad del sistema. Esto implica que la escalabilidad se ve aumentada en un 79.5 %, ya que el proyecto previo solo soportaba un máximo de 2048 neuronas por simulación en una placa. El aumento de la escalabilidad se ve respaldada en la figura 4.23 que define la eficiencia del sistema basada en la aceleración. La eficiencia se vuelve constante a mayor número de neuronas y por ende como consecuencia, se reafirma en esta figura una mayor escalabilidad.

El segundo aspecto, que se observa en la tabla 4.3 y que puede ser visualizado en la figura 4.35, corresponde a la flexibilidad del sistema mediante el re-ajuste del tamaño de los bloques. Igualmente, se observa la flexibilidad a la hora de completar los bloques, respecto al número de neuronas simuladas, ya que no es necesario un número múltiplo al tamaño del bloque, esto se ve para los casos de simulación de 50, 100 y 500 neuronas en los cuales si el número de neuronas es dividida entre la cantidad de bloques, da como resultado un número impar, lo cual para el sistema actual lo resuelve completando el último bloque y descartando los cálculos correspondientes.

Para profundizar en la tendencia cuadrática de la figura 4.35 y con el fin de determinar si los tiempos de este proyecto son relevantes y cumplen con una mayor eficiencia que los 350 ms para las 1000 neuronas en una ZedBoard (ver figura 4.1) del proyecto previo, se construye la figura 4.36. Dicha figura compara los tiempos de ejecución del proyecto previo (línea azul), contra los tiempos obtenidos en el proyecto actual (línea naranja). De la comparación, se puede concluir que los tiempos

de este proyecto se presentan con 47 % de pérdidas en la ganancia comparado con el proyecto anterior.

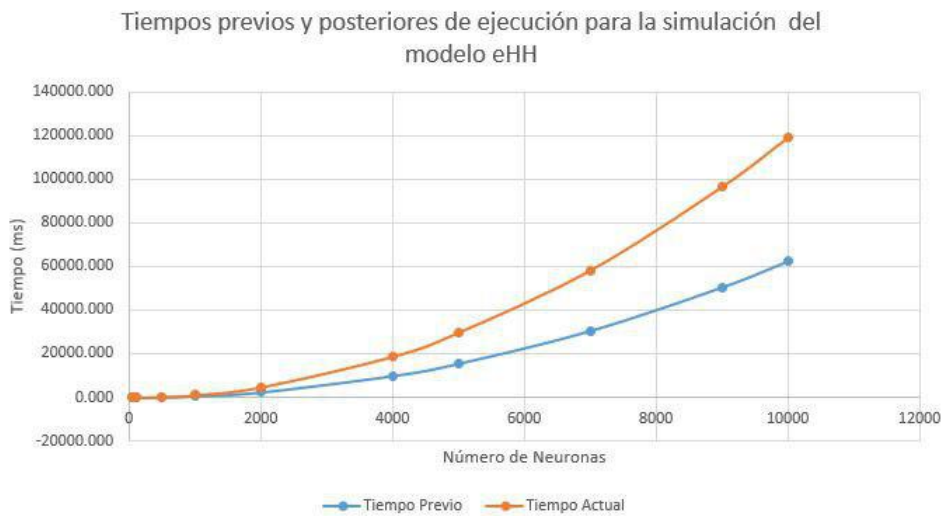


FIGURA 4.36: Gráfica comparativa de tiempos de los ejecución para el modelo neuronal eHH implementado previamente, contra el implementado en este proyecto. Nótese que los tiempos de este proyecto son hasta un 47 % peores que el proyecto previo.

#### 4.6.2. Análisis de resultados

Ahora, como medio para definir cual es el cuello de botella que afecta los tiempos de ejecución para este proyecto, se divide el problema y se construye la tabla 4.4. En la tabla, se detalla el tiempo de ejecución para este proyecto (mostrado igualmente en la figura 4.36) con diferentes números de neuronas. Asimismo, ese tiempo se descompone en dos elementos: el tiempo que se dura ejecutando el algoritmo GJ en el IP (columna Tiempo GJ) y el tiempo sumado que requiere las componentes de soma, axón, dendrita (sin GJ) y, también, la función de re-arme de bloques (columna Tiempo S+A+D-). De la tabla, se pueden extraer varios elementos, el más importante es el que se muestra en la figura 4.37, que corresponde con la identificación del cuello de botella: el tiempo de consumo del algoritmo GJ.

En la gráfica de la figura 4.37, es evidente que el promedio de los tiempos para la llamada al IP que contiene a la función de GJ modificado, es consistente con un 98 % del tiempo total sobre el tiempo promediado para la suma de los tiempos requeridos por las funciones de soma, axón, dendrita sin GJ y la función de bloques.

A raíz de los resultados, se sabe que el tiempo está siendo consumido durante la llamada al IP que contiene la especificación del algoritmo GJ modificado. Existen dos potenciales causas: la primera consiste que el método para simplificar los tiempos de ejecución haya sido ineficiente en la implementación y que el aumento en el procesamiento excesivo de bloques conlleve a que el algoritmo GJ tenga un tiempo de ejecución mayor al deseado. La segunda opción implica una deficiencia en la transmisión de datos mediante el canal del DMA a la FPGA, cuya capacidad de transmisión es de alrededor 249 MB/s. Se realiza una comparación con los tiempos que se obtuvieron en la co-simulación para el IP actual y mostrados en la figura 4.22



Cantidad de Neuronas	Tiempo de ejecución (ms)	Tiempo de GJ(ms)	Tiempo de S+A+D-(ms)
50	4.507	3.997	0.510
100	14.262	13.279	0.983
500	303.735	301.795	1.940
1000	1179.983	1175.036	4.947
2000	4735.114	4723.029	12.085
4000	18783.569	18761.579	21.990
5000	29804.142	29771.244	32.898
7000	58212.623	58175.663	36.960
9000	96426.188	96373.361	52.827
10000	118915.301	118850.351	64.950

CUADRO 4.4: Tabla de tiempos de ejecución para el modelo neuronal eHH de este proyecto. Además, se especifica el tiempo total dividido en dos secciones: la primera el tiempo consumido por el algoritmo GJ; la segunda el tiempo consumido por las componentes de soma + axón + dendrita + cálculo de bloques. Nótese que el tiempo para el algoritmo GJ es el elemento que conlleva el peso de la simulación.



FIGURA 4.37: Gráfica de porcentaje sobre el tiempo consumido para el algoritmo GJ modificado en promedio, contra el tiempo requerido por el promedio de la suma para las componentes soma, axón y dendrita sin GJ.

para la línea del bloque  $8 \times 8$ , contra los tiempos medidos en *software* para el componente dendrita, obtenidos durante la base de partida en el análisis del problema y expuestos en la figura 4.3.

En la figura se observa que existe una amplia ventaja en lo que se refiere a tiempos de ejecución para la línea (gris) de la gráfica que representa los tiempos para



FIGURA 4.38: Gráfica de tiempos para la componente de el algoritmo GJ modificado en un modelo de co-simulación, con un bloque de  $8 \times 8$ , contra los tiempos de la componente dendrita con GJ analizados como parte del problema a solucionar del proyecto previo, ZedBrain. Nótese como el bloque IP del GJ actual sobrepasa enormemente en términos de velocidad de procesamiento al bloque dendrita con GJ integrado.

la implementación del GJ actual que implica que el método de simplificar los tiempos de ejecución no es ineficiente respecto a su predecesor y que, consecuentemente, el problema enfrentado en los tiempos de ejecución actual no se debe al aumento en el procesamiento a raíz de la modificación por bloques implementada en el algoritmo. Es posible, entonces, concluir que el tiempo está siendo aumentado como consecuencia del proceso de transmisión de datos vía el canal entre DMA y PL.

De la ecuación 4.4, se puede deducir el tiempo promedio que le lleva a una simulación transmitir  $N$  bloques a través de los canales del DMA. Para plantear la ecuación es necesario conocer el número de bloques transmitidos, el cual se puede obtener dividiendo la cantidad neuronas entre el tamaño del bloque (dicho tamaño debe ser multiplicado por sí mismo, al tratarse de un matriz: filas  $\times$  columnas). Con el número de bloques y conociendo el tiempo de transmisión entre el DMA y la FPGA (249 MB/s), es posible entonces obtener la ecuación:

$$D(nB) = \frac{tB * nB * II}{t} = \frac{288 * nB * 78}{249MB/s} \quad (4.4)$$

De la ecuación  $nB$  corresponde al número de bloques,  $tB$  corresponde al tamaño del bloque (que para el caso de  $8 \times 8$  cada bloque tiene 72 datos de tipo FLOAT, es decir 288 bytes). Mientras que,  $II$  es el tiempo que le toma a un bloque ser iniciado en el pipeline debido al intervalo de iniciación y por último  $t$  que corresponde a la tasa del tiempo de transmisión para el DMA.

Con base en la ecuación 4.4 propuesta, se plantea la tabla 4.5 que resume la cantidad de bloques por ejecución para simulaciones desde 500 hasta 10000 neuronas. Se adiciona los resultados de la ecuación 4.4 y un porcentaje de error, dicho porcentaje

de error corresponde al tiempo que dura la ejecución del algoritmo, el cual no se logró establecer mediante parámetros. No obstante, la ecuación es bastante precisa y permite comprobar que existe una deficiencia que se identifica en la transmisión de datos mediante el canal del DMA a la FPGA y que el tiempo de ejecución está siendo consumido en el proceso de transmisión.

No. neuronas	Tiempo real (ms)	Tiempo estimado(ms)	No. de bloques	Porcentaje de Error
500	303.735	325.363	3097	7.83 %
1000	1179.983	1301.204	15625	10.27 %
2000	4735.114	5204.819	62500	11.09 %
4000	18783.569	20819.277	250000	10.97 %
5000	29804.142	32530.120	390625	9.27 %
7000	58212.623	63759.036	765625	9.60 %
9000	96426.188	105397.590	1265625	9.30 %
10000	118915.301	130120.481	1562500	9.48 %
Promedio				9.73 %

CUADRO 4.5: Tabla de tiempo estimado según la ecuación 4.4 vs. el tiempo medido. Nótese que la estimación es bastante certera. Claramente, puede notarse que la tasa de transferencia será de 864 MB/s, lo cual ratifica el problema existente en la transmisión.

Después de analizar la tabla 4.5, se comprueba que existe un retardo de los tiempos en el envío y trasiego de datos a través del DMA. Asimismo, mediante la ecuación 4.4 se logra estimar que si se disminuye  $n_B$ , existe una mejora en el tiempo de transmisión. Se observa, entonces, que el problema en tiempos de ejecución proviene de la alta tasa que conlleva transmitir bloques muy pequeños a través del canal del DMA. Existen dos potenciales soluciones: la primera consiste en aumentar el tamaño del bloque de  $8 \times 8$  a un  $N$  superior (por ejemplo  $16 \times 16$ ); sin embargo, lo anterior puede significar un consumo excesivo de recursos que, como consecuencia, tendría un retardo en el tiempo de simulación. La segunda y la más lógica es aumentar el número de bloques transmitidos por ráfaga hacia el DMA. Dado que, para este proyecto, el tiempo estaba en su etapa final, no se pudo comprobar las hipótesis trazadas; sin embargo, se pretende dejar una línea clara de investigación en el desarrollo de futuros proyectos.

## Capítulo 5

# Conclusiones y recomendaciones

### 5.1. Conclusiones

Se estudió el problema referente a este proyecto y se pudo determinar que, para el modelo neuronal eHH, los tiempos de ejecución para los componentes de las dendritas, específicamente para las GJ son de alrededor 99 % sobre el total, debido principalmente al comportamiento cuadrático y creciente del tiempo, en función del número de nodos y la estructura anidada de lazos para el cálculo de la influencia de las neuronas vecinas sobre la neurona bajo análisis.

Se demostró que al implementar el algoritmo de GJ en un IP Core, ejecutado en *hardware* mediante la implementación de una estructura FIFO, basada en el protocolo `AXI4_Stream`, en conjunto con la biblioteca `HLS_Stream` para procesar los datos de entrada del IP, es posible aumentar la escalabilidad del sistema en un 80 % para la ejecución de una FPGA, según la sección explicada en la sección 4.6.

Se logró establecer un sistema flexible mediante la subdivisión de tareas en bloques de entrada que permite ajustarse a la implementación de la estructura FIFO, lo que permitió realizar simulaciones con valores impares o pares de la cantidad de neuronas simuladas, independientes de la dimensión del bloque.

Se determinó que los tiempos de ejecución para el algoritmo GJ modificado en este proyecto con una optimización de paralelismo mediante la implementación de un *pipeline* mejoran en una escala cuadrática con respecto a los tiempos del algoritmo GJ sin modificar. Sin embargo, al conectar el algoritmo con las componentes del modelo ION, se obtuvieron tiempos 50 % mayores que los del sistema predecesor. Según lo discutido en la sección 4.6, se logra demostrar que existe una contradicción, ya que se estima que la llamada del algoritmo GJ modificado consume el 98 % del tiempo de ejecución, lo cual se determinó que se debe al tiempo que es consumido por la comunicación entre el DMA y el IP Core para GJ.

## 5.2. Recomendaciones

- Evaluar la eficiencia que conlleva el envío de cada bloque de información a través del DMA, evaluando posibles opciones en el envío de los bloques de datos que optimicen el canal de DMA.
- Evaluar el costo de operación que conlleva armar los bloques de operación desde el *software* y luego transmitirlos vía DMA al IP Core. Explorar opciones que permitan incluir en memoria los bloques ya formados sin la necesidad de una transmisión repetitiva.
- Explorar mejoras en el modo de operación del DMA, considerando opciones como el modo *Scatter-Gather* que puede brindar una mayor velocidad en la transmisión, así como un modo asincrónico en el envío y recepción de datos.

# Bibliografía

- [1] P. Moorthy y N. Kapre, «Zedwulf: Power-performance tradeoffs of a 32-node zynq soc cluster», págs. 68-75, 2015. DOI: 10.1109/FCCM.2015.37.
- [2] S. W. Moore, P. J. Fox, S. J. Marsh, A. T. Markettos y A. Mujumdar, «Bluehive a field programable custom computing machine for extreme scale real time neural network simulation», págs. 133-140, 2012.
- [3] J. Umaña, *Investigación conjunta con países bajos desarrolla sustitutos de zonas dañadas del cerebro*, 2018. dirección: <https://www.tec.ac.cr/noticias/investigacion-conjunta-paises-bajos-desarrolla-sustitutos-zonas-danadas-cerebro>.
- [4] D. Zamora, «Desarrollo y validación de un método para la visualización de resultados en la implementación del algoritmo de simulación de redes neuronales», inf. téc., 2017.
- [5] K. Alfaro, «Diseño de un acelerador de hardware para simulaciones de redes neuronales biológicamente precisas utilizando un sistema multi-fpga», inf. téc., 2017.
- [6] G. Smaragdous, G. Chatzikonstantis, R. Kukreja, H. Sidiropoulos, D. Rodopoulos, I. Sourdis, Z. Al-Ars, C. Kachris, D. Soudris, C. I. D. Zeeuw y C. Strydis, «Brainframe: A node-level heterogeneous accelerator platform for neuron simulations», *Journal of Neural Engineering*, vol. 14, n.º 6, pág. 066008, 2017. dirección: <http://stacks.iop.org/1741-2552/14/i=6/a=066008>.
- [7] M. van Eijk, C. Galuzzi, A. Zjajo, G. Smaragdous, C. Strydis y R. van Leuken, «Esl design of customizable real-time neuron networks», págs. 671-674, oct. de 2014, ISSN: 2163-4025. DOI: 10.1109/BioCAS.2014.6981815.
- [8] I. Ing. Juan Manuel Sánchez, «Ilp y pipeline», inf. téc., 2017.
- [9] G. Smaragdous, S. Isaza, M. F. van Eijk, I. Sourdis y C. Strydis, «Fpga-based biophysically-meaningful modeling of olivocerebellar neurons», en *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ép. FPGA '14, Monterey, California, USA: ACM, 2014, págs. 89-98, ISBN: 978-1-4503-2671-1. DOI: 10.1145/2554688.2554790. dirección: <http://ezproxy.itcr.ac.cr:3837/10.1145/2554688.2554790>.
- [10] Xilinx, *Field programmable gate array (fpga)*, 2018. dirección: <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>.
- [11] L. Llamas, *¿qué es una fpga? motivos de su auge en la comunidad maker*, 2017. dirección: <https://www.luisllamas.es/que-es-una-fpga/>.
- [12] A. electronics marketing, *Hardware user's guide- zedboard(zynq evaluation and development )*, 2014. dirección: [http://zedboard.org/sites/default/files/documentations/ZedBoard\\_HW\\_UG\\_v2\\_2.pdf](http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf).

- [13] Xilinx, «Bringing ultra high productivity to mainstream systems and platform designers», inf. téc., 2015. dirección: <https://www.xilinx.com/support/documentation/backgrounders/vivado-hlx.pdf>.
- [14] Xilinx., «Vivado design suite user guide high-level synthesis», inf. téc., 2017. dirección: [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2017\\_1/ug902-vivado-high-level-synthesis.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug902-vivado-high-level-synthesis.pdf).
- [15] Xilinx, «Vivado design suite user guide using the vivado ide», inf. téc., 2017. dirección: [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2017\\_1/ug893-vivado-ide.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug893-vivado-ide.pdf).
- [16] Xilinx., *Sdsoc development environment help*, 2018. dirección: [https://www.xilinx.com/html\\_docs/xilinx2018\\_1/sdsoc\\_doc/dec1517252127819.html](https://www.xilinx.com/html_docs/xilinx2018_1/sdsoc_doc/dec1517252127819.html).
- [17] Xilinx, «Vivado design suite axi reference guide», inf. téc., 2017. dirección: [https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_ref\\_guide/latest/ug1037-vivado-axi-reference-guide.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf).
- [18] A.Pabón y D. Mejia, *Real time audio wave display*, 2014. dirección: <https://sites.google.com/site/rtawdde0nano/bitacora>.
- [19] E. Scalise y R. Carmona, *Análisis de algoritmos y complejidad*, 2001.