Implementación y evaluación del rendimiento de redes neuronales densas en FPGA para la inferencia rápida, aplicadas a problemas en física y visión artificial.

Trabajo de grado para optar al título de físico

Daniel Alfonso Montoya Vásquez

Asesor de trabajo de grado:

Dr. Johny Alexander Jaramillo Gallego

UNIVERSIDAD DE ANTIOQUIA

FACULTAD DE CIENCIAS EXACTAS Y NATURALES

INSTITUTO DE FÍSICA

MEDELLIN

2020

CONTENIDO

Índice de Figuras

Índice de tablas

Lista de acrónimos

ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BRAM	Block RAM
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DL	Deep Learning
DMA	Direct Memory Access
DSP	Digital Signal Processor
FC	Fully Connected
FF	Flip Flop
FIFO	First In First Out
FLOPs	Floating-Point Operations per second
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HLS	High-Level Synthesis
hls4ml	High-Level Synthesis for Machine Learning
HW	Hardware
I/O	Input/Output
IA	Inteligencia Artificial
IP	Intellectucal Property (Usado tambien como IP Core o Bloque IP)
LHC	Large Hadron Collider
LSTM	Long Sort-Term Memory
LUT	LookUp Table
MHz	Mega Hertz
ML	Machine Learning
MLP	Multi_Layer Perceptron
MNIST	Modified National Institute of Standards and Technology
NN	Neural Network
PL	Programmable Logic
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
RTL	Register-Tranfer Level
SDK	Software Development Kit
SW	Software
VHDL	VHSIC-HDL, Very High Speed Integrated Circuit Hardware Description Language

Resumen

En la actualidad diferentes contextos industriales y científicos participan del crecimiento de la producción de datos, lo cual conlleva igualmente a la necesidad de procesarlos de la manera más rápida posible. Dos de estos contextos de particular interés para este trabajo son la física experimental y la visión artificial, donde las técnicas de la inteligencia artificial como las redes neuronales, vienen mostrando precisión y efectividad en ciertas tareas, pero tienen limitaciones en cuanto a la velocidad de su implementación debidas principalmente al hardware con que se implementan.

Alternativas de hardware como ASICS, GPUs, y FPGAs compiten en la carrera por acelerar el entrenamiento y la inferencia de redes neuronales. Características como la flexibilidad, bajo consumo de energía, costos asequibles, y velocidad hacen de las FPGAs uno de los competidores más prometedores y aplicables, especialmente en contextos académicos.

En el presente trabajo se presentan las bases teóricas y conceptuales que se encuentran en la intersección de la física, el aprendizaje profundo y las FPGAs, se ilustra el proceso de implementación de diferentes redes neuronales en FPGA mediante el uso de la síntesis de alto nivel, y se evalúa el rendimiento de éstas respecto a otros sistemas como GPUs, y CPUs programadas en entornos Python y C, en cuanto a latencia y consumo de recursos, en particular, respecto a los problemas de inferencia de redes neuronales para clasificación de subestructura de jets, y un problema de visión artificial: la clasificación de dígitos escritos a mano. Los resultados muestran la superioridad en cuanto a tiempo de inferencia, de una FPGA de bajo costo respecto a los otros sistemas, en ordenes desde 10.72x hasta 805x veces más rápida sin perder exactitud. Estos resultados validan estos dispositivos como fuertes candidatos para mejorar el rendimiento de las redes neuronales. Finalmente se discuten limitaciones y direcciones futuras.

Implementation and evaluation of the performance of dense neural networks implemented on FPGAs for fast inference, applied to problems in physics and computer vision

Abstract

Currently, many industrial and scientific contexts participate in the worldwide growth of data production, which also leads to the need to process these data as quickly as possible. Two of these contexts of particular interest for this work are experimental physics and computer vision, where artificial intelligence techniques such as neural networks have been showing good precision and effectiveness in certain tasks, but have limitations in terms of the speed of their implementation due mainly to the hardware in which they are implemented.

Hardware alternatives such as ASICS, GPUs, and FPGAs compete in the race to accelerate neural network training and inference. Features such as flexibility, low power consumption, affordable costs, and speed make FPGAs one of the most promising and applicable competitors, especially in academic contexts.

This work presents the theoretical and conceptual basis at the intersection of physics, deep learning, and FPGA, illustrates the process of implementing different neural networks in FPGAs using high-level synthesis, and evaluates the performance of FPGAs with respect to other systems such as GPUs, and CPUs programmed in Python and C environments, in terms of latency and resource consumption, particularly with respect to neural network inference problems for jet substructure classification, and a machine vision problem: the classification of handwritten digits. The results show the superiority in terms of inference time of a low-cost FPGA over other systems, in orders from 10.72x to 805x times faster without losing accuracy. These results validate these devices as strong candidates for improving neural network performance. Finally, limitations and future directions are discussed.

1 FUNDAMENTOS TEÓRICOS

1.1 Introducción inteligencia artificial y aprendizaje profundo

Para comenzar, se realizará un repaso breve a qué es la inteligencia artificial y cuál es el lugar del aprendizaje profundo (o Deep Learning, DL) dentro de esta. La inteligencia artificial es un campo prominente en la actualidad con múltiples aplicaciones prácticas y temas activos de investigación. Unos cuantos ejemplos incluyen la automatización de labores rutinarias, comprender el lenguaje o las imágenes, realizar diagnósticos en medicina, y apoyar investigaciones científicas [1].

Una de las primeras victorias de la inteligencia artificial (IA) fue en 1997 cuando Deep Blue (el supercomputador desarrollado por IBM) pudo derrotar a Kasparov en el ajedrez. Sin embargo, el ajedrez es un entorno relativamente sencillo, en un espacio de 8x8 casillas y 32 piezas con movimientos fijos. Es en el tipo de tareas repetitivas y con reglas formales claras, donde los computadores han sido más rápidos y precisos que los humanos, pero es sólo recientemente, que las máquinas están siendo capaces de igualar a los humanos en tareas complejas, tales como: reconocer objetos o entender algunas características del lenguaje o del comportamiento. Sin embargo, los humanos requieren una gran cantidad de conocimiento acumulado sobre el mundo real, para realizar tales tareas, y este conocimiento es subjetivo y/o intuitivo, y en general difícil de escribir en reglas formales. Los computadores deben capturar este conocimiento de una forma similar, para comportarse de una manera “inteligente”. Este es uno de los principales retos en el campo de la inteligencia artificial: ¿cómo podemos los humanos trasmitirle conocimientos complejos a una maquina? [2].

En los inicios de la investigación sobre inteligencia artificial, se buscaba que los programadores escribieran o tradujeran formalmente el conocimiento del mundo en reglas rígidas, de modo que un computador pudiera implementarlas de forma automática, utilizando las reglas lógicas de la inferencia. Esta aproximación se conoce como “basada en conocimiento” (*knowledge-base*). Esta aproximación es bastante complicada, pues requería escribir muchas reglas con la suficiente complejidad para describir el mundo real. Estas dificultades hicieron que se reconociera la necesidad de que los sistemas de IA tuvieran la habilidad de adquirir su propio conocimiento a partir de los datos “crudos”. Esta habilidad se denominó aprendizaje automático (*Machine Learning*, ML). La aparición del ML permitió a los computadores resolver problemas del mundo real y tomar decisiones que parecían subjetivas, como por ejemplo, un algoritmo sencillo del ML (conocido como regresión logística) fue capaz de recomendar un parto por cesárea o parto natural; o también, otro algoritmo conocido como *Naive Bayes* pudo clasificar y separar los correos electrónicos basura (spam) de los que no lo eran [3].

Ahora bien, la eficiencia de estos algoritmos depende de una manera esencial de la representación de los datos que se les entregan. Por ejemplo, para el caso de la cesárea el algoritmo no observa un paciente directamente, sino una serie de datos sobre el paciente, como su edad, o la presencia o no de una cicatriz uterina. Cada dato que representa a un paciente se conoce como una característica (*feature*). Luego, el algoritmo aprende cómo estas características se correlacionan con diferentes resultados, pero el algoritmo no tiene ninguna influencia en la forma en que las características se definen. SI a este algoritmo se le presentaran datos de algún otro examen distinto de los que conoce, no podría arrojar ningún resultado útil para la finalidad [1].

La importancia de la representación es un problema central en ciencias de la computación e incluso en la vida diaria. Por ejemplo, un algoritmo de búsqueda puede funcionar mucho más rápido si los datos están indexados de una manera inteligente. O las personas pueden calcular más fácilmente operaciones aritméticas en número arábigo que en números romanos. Muchas tareas de IA se pueden resolver al elegir el conjunto correcto de características a utilizar, y luego pasar estas características a un algoritmo de ML. Sin embargo, para muchas tareas es difícil saber a priori cuales son las características deben ser extraídas. Por ejemplo, supongamos que

se quiere escribir un programa que detecta carros en fotografías. Se sabe que los carros tienen llantas, así que nos gustaría usar la presencia de llantas como una característica. Sin embargo, es difícil describir de manera exacta cómo se ve una llanta en términos de píxeles, pues la imagen en general, estará modificada por la presencia de sombras, el reflejo del sol en el rin, el guardabarros del auto, o algún objeto que oculte la llanta parcial o totalmente. Una solución a este problema, parte de la utilización de aprendizaje automático para descubrir no solamente una función que lleve de las características al resultado, sino descubrir la representación misma. Esta perspectiva se conoce como “aprendizaje por representación” (*representation learning*). Esta aproximación resulta en rendimiento mucho mejor que tener que diseñar a mano las representaciones, al tiempo que también permite que los sistemas de IA se adapten más fácilmente a diferentes tareas, con menor intervención humana. Un sistema que aprende las representaciones puede descubrir un buen conjunto de características para una tarea simple en minutos, o una tarea compleja en horas, o días. Diseñar manualmente las características para una tarea compleja requiere una gran cantidad de esfuerzo humano y puede llevar décadas de una comunidad de investigadores [4].

En el diseño de este tipo de algoritmos, es en general un reto separar los factores de variación que explican correctamente ciertos comportamientos de los datos; aunque muchas veces estos factores no son evidenciados directamente, sino que están presentes de manera implícita, y afectan los datos observados. Pueden entenderse como conceptos o abstracciones que nos ayudan a dar sentido a la variabilidad de los datos. Por ejemplo, cuando se quiere analizar una grabación de voz, los factores de variación incluyen la edad, el sexo, el acento y las palabras del hablador. Cuando se analiza la imagen de un carro, los factores de variación incluyen la posición, el color, el ángulo, o el brillo del sol. Una gran dificultad de los sistemas de IA es que muchos factores influencian cada uno de los datos observados. Por esto, la mayoría de las aplicaciones de IA buscan separar los factores de variación decisivos, lo cual puede ser bastante complejo, pues tales factores pueden llegar a ser muy abstractos, y se puede llegar a requerir un nivel de entendimiento casi humano [1].

El Aprendizaje Profundo o *Deep Learning* (DL) busca resolver este problema del aprendizaje por representación al introducir representaciones expresadas en términos de otras representaciones más sencillas. El DL permite a un computador construir conceptos complejos a partir de otros más sencillos. La siguiente figura ilustra este concepto

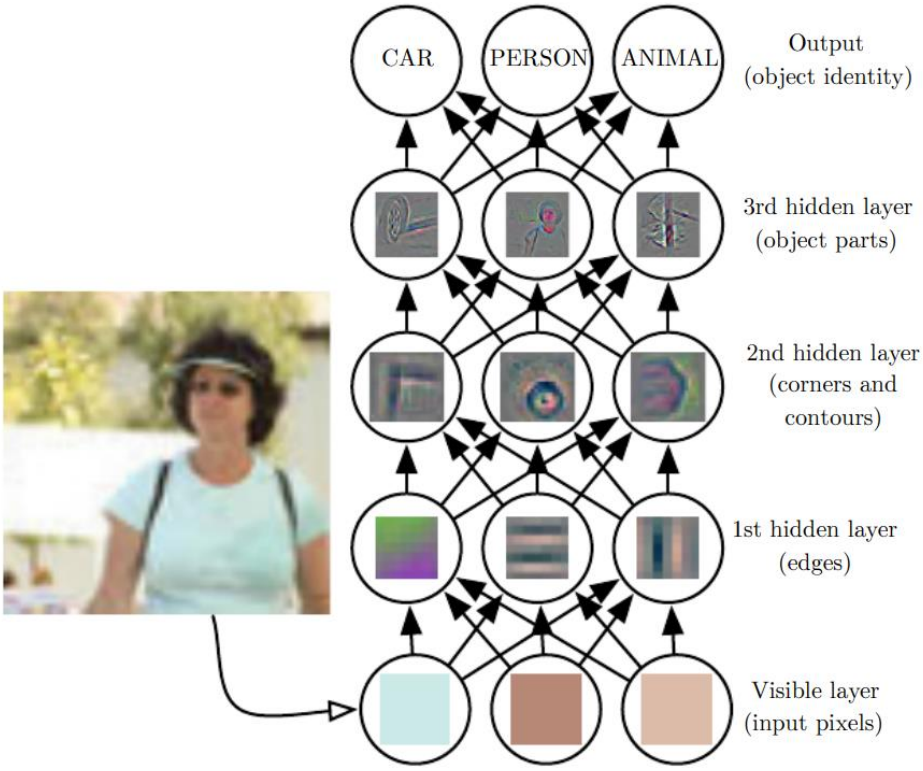

Figura 1: Representaciones en un algoritmo de Aprendizaje Profundo [1]

La figura 1 presenta un esquema de la forma en que un sistema de DL puede representar el concepto de la imagen de una persona al combinar diferentes conceptos más sencillos, como las esquinas y contornos, que se definen a su vez de conceptos más sencillos, como lo son los bordes [5].

El caso más prominente de un algoritmo de DL es el perceptrón multicapa, el cual representa una función matemática que mapea un conjunto de valores de entrada a unos de salida. Esta función se compone de otras funciones más sencillas. Puede decirse que cada capa de éste provee una nueva representación de los valores de entrada. El perceptrón multicapa se conoce también como red neuronal densa. El estudio de este tipo de algoritmos, y su implementación en FPGAs es el centro del presente trabajo. En la siguiente figura se ilustra la ubicación del Deep Learning dentro del campo de la inteligencia artificial.

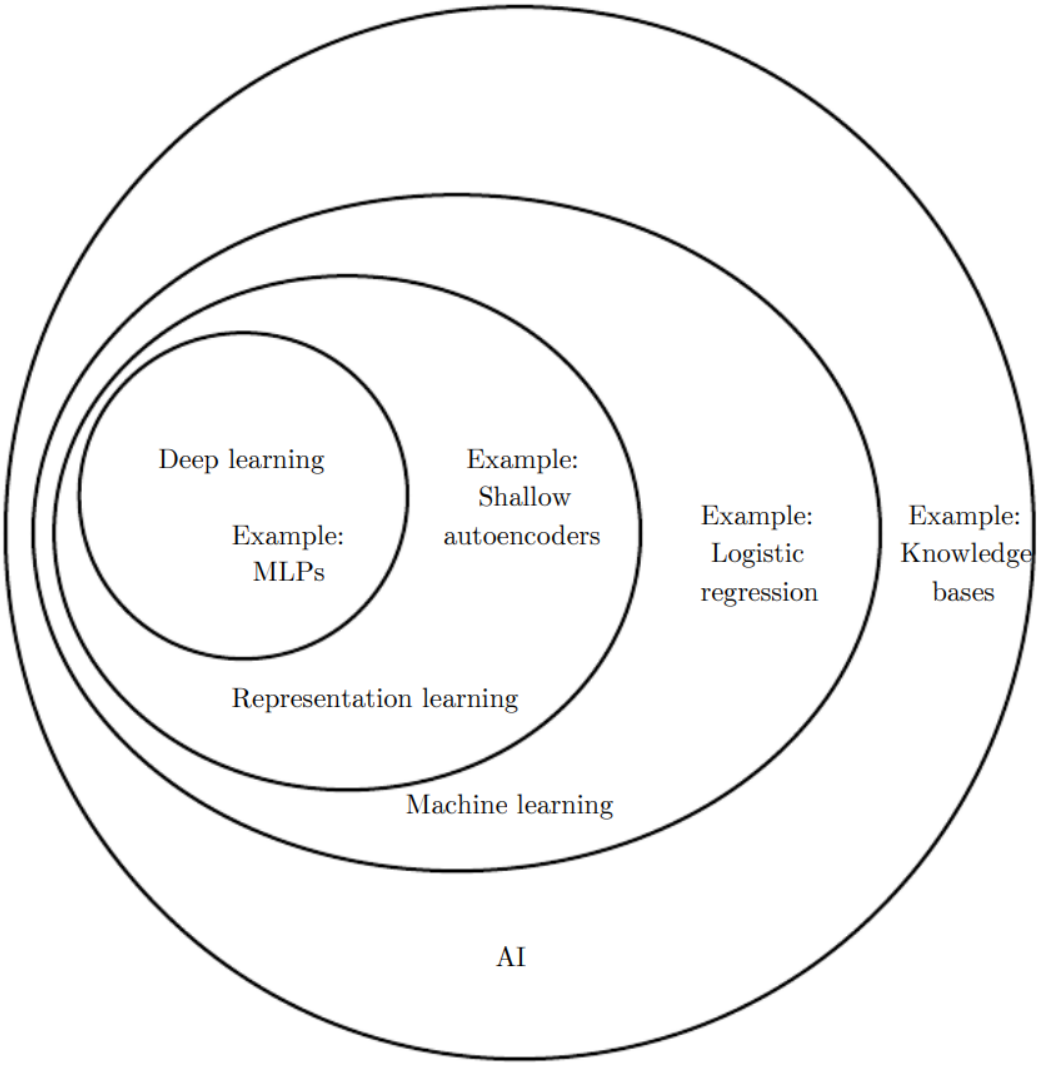

Figura 2: Ubicación del DL en la inteligencia artificial [1]

De este modo, puede observarse que el aprendizaje automático o Deep Learning es un caso particular dentro del estudio de aprendizaje por representación, que a su vez es un caso particular de aprendizaje automático, que a su vez es un campo de estudio dentro de inteligencia artificial. Para más detalles sobre la diferentes técnicas diferentes DL, puede verse: [6].

1.2 Sobre el aprendizaje profundo: arquitecturas y operaciones

Como se mencionó en la sección anterior, un conjunto de técnicas que han impulsado notablemente la inteligencia artificial (IA) es el llamado *Deep Learning* (DL) o "aprendizaje profundo", cuyos orígenes pueden rastrearse hasta la década de 1950 con los primeros experimentos con redes neuronales [7]. Yann LeCun, pionero del DL plantea que éste se basa en cuatro ideas sencillas:

1) Funciones complejas pueden ser construidas de manera eficiente al ensamblar bloques sencillos y funcionales, parametrizados en gráficos computacionales multi-capa (como operadores lineales y no linealidades puntuales).
2) La función deseada puede aprenderse a través de ejemplos al ajustar los parámetros de la red.
3) El proceso de aprendizaje minimiza una función objetiva a través de un método basado en un gradiente.
4) El gradiente puede calcularse eficiente y automáticamente a través de un algoritmo de retropropagación, que consiste en la aplicación de la regla de la cadena para calcular

las derivadas parciales del resultado esperado respecto a todos los parámetros de la red, al propagar las señales hacia atrás a través del sistema.

Al hacer parte del campo del aprendizaje por representación, una ventaja clave de DL es que reduce o elimina la necesidad de diseñar a mano la extracción de variables, como es necesario con métodos tradicionales de Machine Learning (ML). Cuando se entrenan para una tarea particular, los sistemas de DL aprenden automáticamente representaciones jerárquicas multi capa de los datos, útiles para la tarea en cuestión. Por esto, los sistemas de DL pueden resolver problemas complejos, en los que el conocimiento del problema no se puede expresar explícitamente de manera sencilla [7].

Ahora bien, se ha mencionado que el software (SW) y el hardware (HW) disponible, ha limitado las ideas que son realizables, por ejemplo, el algoritmo de retropropagación no era aplicable hasta que el rendimiento de los procesadores alcanzó un millón de operaciones por segundo en la multiplicación-acumulación de puntos flotantes [8]. Luego, en 2013 hubo un resurgimiento del DL, debido a diferentes factores: mejores métodos, grandes conjuntos de datos (muchas muestras, muchas variables), la aparición de GPUs (Graphics Processing Unit) de bajo costo a nivel de TFLOPs (Tera-FLOPs: trillones de operaciones de punto flotante por segundo), y el surgimiento de librerías de código abierto con interfaz de lenguaje interpretado. Por esto, se dice que tanto el software como el hardware disponible motivan, pero también limitan las ideas que los investigadores de IA pueden imaginar y que pueden poner a prueba. De este modo, lo que realicen las comunidades de hardware y software va a influir de manera decisiva en el futuro de la investigación en IA [7].

Adicionalmente, diferentes contextos de uso de DL tienen necesidades específicas en cuanto a hardware:

- Investigación y desarrollo: Máquinas multi-nodo para computación de alto rendimiento (HPC, High-Performance Computing), cada una con un arreglo de dispositivos flexibles (GPU, FPGA), para operaciones de 32b de punto flotante. Usualmente se requiere baja latencia y altos niveles de paralelización.
- Entrenamiento offline de modelos existentes: Reentrenamiento de modelos debido a la aparición de datos nuevos, puede realizarse con aritmética de menor precisión, como punto fijo de 16b. El requerimiento de baja latencia y altos niveles de paralelización son menos exigentes que en Investigación y desarrollo.
- Inferencia en servidores: Es importante el consumo de energía; la flexibilidad y el rendimiento puro son secundarios; la latencia no es importante. Es ideal la aceleración de inferencia en muestras individuales (No en lotes (*batches*)). Algunas aplicaciones como la industria automotriz pueden tener requerimientos más altos en latencia.
- Inferencia en sistemas móviles y embebidos: dispositivos de realidad virtual, realidad aumentada y teléfono móviles tienen el requerimiento de consumo muy bajo de energía, en algunos casos hay requerimientos especiales de latencia.

Por esto, es claro que habrá hardware especializado en entrenamiento de redes, y otro hardware especializado en la inferencia. Así, nuevos conceptos en arquitectura de redes, como activaciones dispersas, arquitectura dinámica, y módulos que manipulan datos no tensoriales (gráficos), rompen con el supuesto de que se puede realizar computaciones en lotes de muestras de igual tamaño. De este modo, se ha propuesto que se necesitan arquitecturas que puedan trabajar eficientemente con un lote (*batch*) de tamaño uno, además de trabajar con datos como gráficos anotados con tensores y símbolos.

En cuanto a la inferencia, aunque se espera un crecimiento de ésta en centros de datos e inferencia basada en la nube, las aplicaciones de DL van a crecer mucho más en sistemas móviles y embebidos, por tanto, en los próximos años, se requerirá *hardware* de muy bajo poder (sean ASICs o FPGAs), con muy baja latencia. Una gran carga de computación se espera que sean convoluciones, por tanto se ha sugerido que el hardware deberá ser capaz de explotar las

regularidades de la operación de convolución en lugar de ser calculadoras de productos matriciales [7].

Por otra parte, hay dos arquitecturas principales en DL: perceptrones multi capa (MLP, Multi-Layer Perceptron) y redes neuronales convolucionales (CNN, Convolutional Neural Network). Estas redes tienen diferentes paralelismos que hacen estas redes adecuadas para aceleración por hardware:

- Operaciones que se pueden realizar simultáneamente a un grupo de pixeles, o regiones locales en los datos
- Lotes (*batches*) de datos
- Paralelismo del modelo
- paralelismo del flujo de datos (Pipeline)

Entre las diversas opciones de aceleración por hardware, las más importantes son: circuitos específicos para la aplicación (ASIC: Application Specific Integrated Circuit), Unidades de procesamiento gráfico (GPU: Graphical Processing Unit), y arreglo de compuertas programable en el campo (FPGA: Field-Programmable Gate Array).

Entre estas opciones, las GPU soportan varias TFLOPs y tienen acceso a gran cantidad de memoria, pero consumen una alta cantidad de energía. Las FPGA por su parte, tienen bajo consumo de energía, pero también un acceso a memoria más limitado, por tanto, se ha argumentado que es más conveniente acelerar la inferencia (en lugar del entrenamiento) en las FPGA debido a que la inferencia requiere menos uso de memoria. Las FPGA facilitan la evolución de hardware, software y distintos entornos de trabajo. Para distintos modelos de redes neuronales, la flexibilidad de las FPGA acorta el ciclo de diseño y son más económicas [9].

Por otra parte, los ASICs tienen el mejor rendimiento, pero tienen un periodo largo de diseño, y una vez diseñados, no tienen flexibilidad, lo cual es una desventaja en el área de DL, en donde constantemente se realizan avances que hacen que las arquitecturas rígidas de los ASIC corren el riesgo de quedar obsoletas de manera rápida [10].

De este modo, las FPGA son un punto intermedio entre flexibilidad y rendimiento, ya que pueden programarse para la tarea específica para que son requeridas, alcanzando en ocasiones rendimientos similares e incluso superiores a las GPU. Otra ventaja que ofrecen las FPGA es la capacidad de reconfiguración parcial dinámica, que implica que una parte de la FPGA se puede reconfigurar mientras otra parte se está usando, lo cual es una ventaja cuando los inputs de procesamiento tienen dimensionalidad variable. Así, aunque las GPU tienen un gran rendimiento, pueden limitar la creatividad de investigadores en DL, pues para la implementación debe tenerse en cuenta su arquitectura al diseñar los algoritmos, mientras que para las FPGA, la arquitectura se ajusta a la aplicación, lo cual implica mayor libertad para explorar optimizaciones e ideas a nivel de los algoritmos [10]. Sin embargo, la flexibilidad de las FPGA tiene el costo de que no son fácilmente programables, lo que ha venido impulsando a los fabricantes y a la comunidad de hardware, a crear herramientas de alto nivel de abstracción (High-Level abstraction) que faciliten el uso de FPGA por parte de investigadores no expertos en arquitectura de hardware.

Por ejemplo, OpenCL es un lenguaje de programación abierto basado en C, ejecutable en procesadores de propósito general (GPP: General Purpose Processors), GPUs, FPGAs, y procesadores de señales digitales (DSP: Digital Signal Processor). OpenCL permite el acceso de bajo nivel al hardware, sin embargo, la flexibilidad multiplataforma tiene la desventaja de que no todas las plataformas tienen garantizado el soporte de todas las funciones de OpenCL. El rendimiento para CUDA (Plataforma de computación en paralelo de las GPU de nVidia) y OpenCL es muy similar para diferentes aplicaciones [10].

Por otra parte, un reto de la implementación de DL en FPGA es el tamaño de los diseños: los circuitos de FPGA usualmente son menos densos que otras alternativas de hardware, de modo que implementar redes neuronales grandes no siempre es posible. Pero en este momento, los

fabricantes están explorando la posibilidad de explorar tamaños más pequeños de los circuitos, al aumentar la densidad de éstos, de modo que algunas redes profundas ya se han implementado en FPGAs individuales. Otras mejoras recientes, son el acceso a memoria por fuera del circuito, capas de software configurables, entradas y salidas en buffer, y elementos de procesamiento paralelo, usualmente para realizar convoluciones [10].

En cuanto a la implementación, aún existen muchas preguntas que son objeto de estudio, como por ejemplo la utilización de diferentes subsistemas de memoria, los mecanismos de transferencia de datos, el uso de *soft-cores*, tipos de tablas de consulta (LUTs: LookUp-Tables), frecuencias de operación, entre otros. Debido a las características de las FPGAs, en donde se ha dado mayores esfuerzos es en la aceleración de la propagación hacia delante de redes ya entrenadas, es decir en cuanto a inferencia. Sin embargo la retropropagación también es un temas de interés en investigación [10].

1.2.1 Perceptrón multicapa y redes convolucionales

Perceptrón multicapa

Un perceptrón multi capa (MLP, *Multi-Layer Perceptron*, también conocido como *Fully connected*, o *Dense Layer*) es una red de neuronas artificiales, cuya finalidad es aproximar una función objetivo de modo que dadas unas entradas (datos) pueda arrojar un resultado (etiquetas). Formalmente, un perceptrón con una capa oculta es una función $f: R^D \rightarrow R^L$, donde D es la dimensión del vector de entrada x, y L es la dimensión del vector de salida $f(x)$, de modo que en notación matricial:

$$f(x) = G(b^{(2)} + W^{(2)}\, s(b^{(1)} + W^{(1)}\, x)) \qquad (1)$$

Donde $b^{(1)}$ y $b^{(2)}$ son vectores *BIAS* (sesgo), $W^{(1)}$ y $W^{(2)}$ son matrices con pesos, y G y s son funciones de activación [11]. Entonces $x \in R^D$, y por tanto si la primera capa de neuronas tiene n_1 neuronas, entonces $W^{(1)}$ debe ser de la forma $n_1 \times D$, con lo que el vector resultante de la primera operación es de dimensión n_1, pues $b^{(1)}$ naturalmente también es de dimensión n_1. De este modo se puede continuar con este análisis y descubrimos que el vector $f(x)$ debe ser de dimensión n_2, o en general, de la dimensión del número de neuronas de la última capa, es decir, L. Gráficamente puede verse de la siguiente manera:

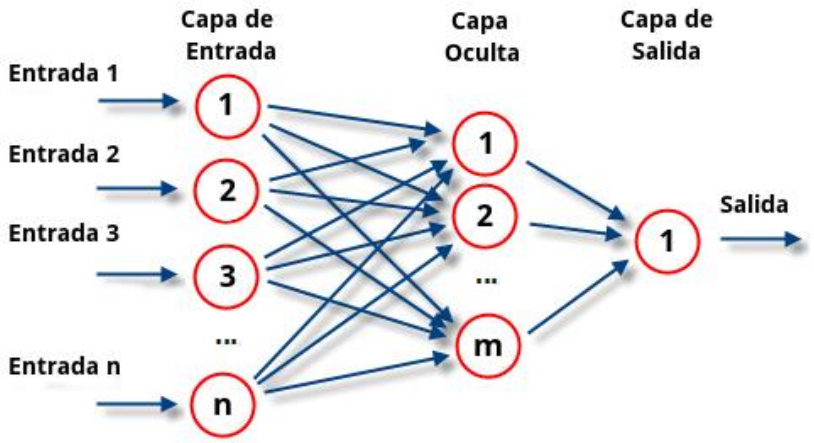

Figura 3: Perceptrón con una capa oculta [12]

Así, por ejemplo, las diferentes variables que se tienen de un sujeto, o muestra, constituyen los datos de entrada, y las flechas entre neuronas constituyen los pesos de las matrices W. Esto puede generalizarse fácilmente a un número arbitrario de capas ocultas.

Redes Neuronales Convolucionales

Otra arquitectura importante en DL son las redes neuronales convolucionales (*Convolutional Neural Networks*: CNN). Estas han mostrado ser efectivas para tareas de visión artificial, como en la clasificación y segmentación de imágenes, entre otras aplicaciones. La diferencia con el MLP es que a los datos (que usualmente son imágenes), se les aplica inicialmente las operaciones de convolución y reducción de muestreo (*sub-sampling*). Luego de varias fases de convolución y reducción de muestreo, los resultados ingresan a un MLP. Esto se ilustra en la Figura 4:

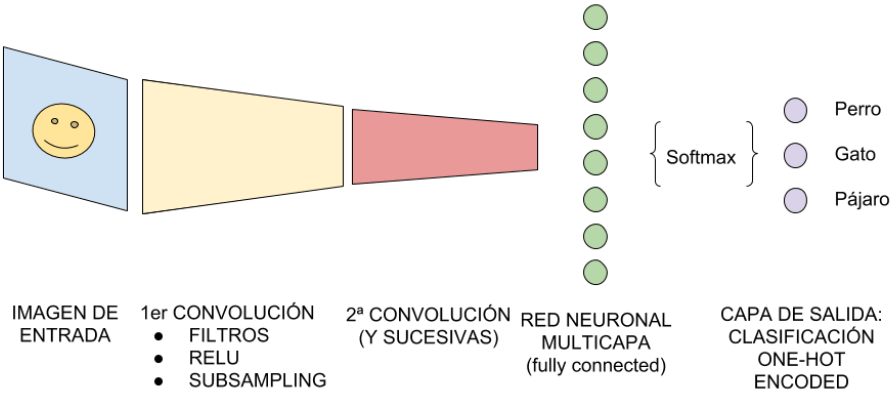

Figura 4: Arquitectura de una CNN [13]

1.2.2 La multiplicación matricial como una operación fundamental de las redes neuronales

Todas las arquitecturas expuestas hasta el momento (MLP y CNN) utilizan la operación de multiplicación matricial, ya sea en la totalidad de su arquitectura (MLP), o solamente en la parte final, como en el caso de las CNN. En el caso presentado hasta el momento, se trata de la multiplicación $W^{(i)}\,x$, que corresponde a la mutiplicación Matriz-vector, que da como resultado un vector, y el cual luego de pasar por la función de activación, vuelve a multiplicarse por la siguiente matriz de pesos $W^{(i+1)}$ hasta alcanzar la capa de salida. Este es el caso, por ejemplo, de la inferencia con *batch 1*, es decir, el caso en que cada muestra de datos pasa de uno en uno (en oposición a pasar varias muestras simultáneamente por la red).

En el caso de que se desee realizar inferencia o entrenamiento con un lote (*batch*) mayor que 1, la multiplicación entre matrices aparece al calcular la función $W^{(1)}\,X$ para más de una muestra de entrenamiento o de inferencia, donde se denota X como la matriz que contiene más de un ejemplo de los datos. Esto tiene como consecuencia que el anidamiento de operaciones produce una secuencia de multiplicaciones matriciales:

$$f(X) = G(b^{(2)} + W^{(2)}\,s(b^{(1)} + W^{(1)}\,X)) \qquad (2)$$

Donde en la ecuación (2), $f(X)$ es un vector, y las capas ocultas pueden ser N. De este modo, la optimización de la operación multiplicación matricial es crucial para la aceleración de inferencia o entrenamiento de redes neuronales, ya sea que se optimice la multiplicación matriz-vector, o la multiplicación entre matrices. Es claro así que debe tenerse claro la dimensión del vector o matriz de entrada (*batch*) que va a hacerse pasar por la red antes de comenzar el proceso de optimización.

Una característica relevante de la multiplicación matricial, es que es altamente paralelizable en cada capa, pues siempre se cuenta con los datos de entrada y los pesos de cada neurona. Sin embargo, no es posible paralelizar todas las capas de la red neuronal simultáneamente, pues es necesario disponer de los cálculos de la capa inmediatamente anterior para poder comenzar el cálculo de la siguiente capa.

Las optimizaciones principales que se pueden realizar en las FPGA para esta operación corresponden al desenrollado de ciclos, el *pipelining*, y la reducción de la precisión de la representación numérica de pesos y datos. Más adelante de profundizará en estas optimizaciones.

1.3 Aprendizaje profundo y FPGAs en física

En muchos contextos experimentales, los sensores y detectores de propiedades físicas han aumentado su resolución espacial y temporal, es decir, pueden detectar con mayor precisión y en intervalos de tiempo cada vez más cortos. Esto implica que cada vez haya volúmenes más grandes de datos, y que, en la mayoría de estas situaciones, los datos deben ser analizados a muy altas velocidades: se deben filtrar, clasificar e identificar en "tiempo real" los eventos de interés. Una opción tradicional para realizar este proceso son los ASIC, pero recientemente, las FPGA han venido emergiendo como una buena alternativa debido a la flexibilidad y baja latencia que estas pueden alcanzar. Para algunas aplicaciones, como la detección de partículas en el Gran Colisionador de Hadrones *(LHC: Large Hadron Collider)*, no es posible utilizar GPUs o CPUs (Central Processing Units) debido a las exigencias en cuanto a límite de tiempo para el procesamiento 25 ns (40MHz) [14]. En el capítulo 3 se profundizará más a este respecto, cuando se presente la herramienta de alto nivel utilizada para programar una NN en la FPGA.

En estos contextos, se hace necesario que sea posible una transición rápida entre modelos de ML y la implementación de estos en FPGA usando herramientas de síntesis de alto nivel. Por esto, aparecen varios caminos para realizar la transición de una arquitectura de NN y una implementación en FPGA. Por un lado, se puede realizar la programación de la arquitectura en el lenguaje C, y realizar el proceso de síntesis de alto nivel (HLS, High-Level Synthesis), para personalizar las optimizaciones que queremos realizar en el proceso de inferencia, o han venido apareciendo diferentes herramientas para mapear CNNs en FPGAs.

En el presente trabajo vamos a seguir ambos caminos. Entre las herramientas para el mapeo de NN están:

- Snowflake: para mapear modelos creados en Torch a SOCs de Xilinx.
- Caffeine: mapear modelos creados en Caffe a FPGAs de Xilinx que soportan el entorno SDAccel y tienen una interfaz PCIe entre el host y la FPGA
- fpgaConvNet: convierte CNNs escritas en Caffe o Torch a código de Vivado HLS
- FP-DNN: toma CNNs, RNNs, LSTM descritas en Tensorflow y genera las implementaciones en hardware en FPGAs con plantillas híbridas de RTL-HLS
- DNN Weaver: herramienta de código abierto que toma modelos escritos en Caffe y los lleva a lenguaje acelerado en Verilog, usando plantillas optimizadas manualmente [14].
- hls4ml: traducir una NN ya entrenada, especificada por la arquitectura y los pesos a código HLS.

Algunas métricas clave para la implementación de NNs en FPGAs son:

- Latencia: es el tiempo total que se demora una iteración del algoritmo
- Intervalo de iniciación: es el número de ciclos de reloj que se requieren antes de que el algoritmo pueda aceptar una nueva entrada. Es inversamente proporcional a la tasa de inferencia (o rendimiento máximo)

- Uso de recursos: Cantidad de dispositivos usados: memoria en la tarjeta (BRAM), DSPs aritméticos, registros, flip-flops, LUTs [15].

La implementación propia (sin el uso de herramientas de terceros) para la optimización de redes neuronales también se realiza en el presente trabajo, y en la sección correspondiente se expone el proceso de optimización comenzando por la operación de multiplicación matricial, que es el bloque básico de la inferencia y entrenamiento de las redes neuronales. Una ventaja de llevar a cabo este proceso es que permite comprender cómo operan las librerías mencionadas, permite comprender en mayor profundidad el funcionamiento y la arquitectura de las FPGA, y permite comprender el proceso de llevar a cabo la aceleración en general de cualquier función que se desee acelerar en FPGAs, por ejemplo, personalizar un post-analisis que se desee realizar a la inferencia de las NN. Este proceso completo se expone en el segundo capítulo.

1.4 Herramientas para el mapeo de redes neuronales en FPGAs

Considerando lo discutido en la sección anterior, y también debido a la existencia de diferentes contextos de aplicación de *Deep Learning* (DL) y redes neuronales (NN), se ha evidenciado la necesidad de mapear de manera rápida y efectiva la inferencia de NN y CNN (redes convolucionales) en FPGAs. La alta estructuración de estos algoritmos hace que sea posible desarrollar herramientas automáticas que realicen este trabajo, teniendo en cuenta los parámetros, la topología, y configuraciones de la red, y las características de FPGAs específicas. Es decir, dado un par red-FPGA, se busca generar automáticamente una implementación en *hardware* (HW) sin necesidad de experticia en diseño de HW. Esto llevaría las FPGAs a una mayor integración en el ecosistema de DL [16].

De este modo, existen diferentes herramientas con diferentes características cada una. En cuanto a los tipos de redes soportadas, todas estas herramientas soportan capas convolucionales, activaciones no lineales, capas de reducción de muestreo, y capa densa (*Fully connected*, FC). Algunas herramientas como DeepBurning y FP-DNN soportan también RNN y LSTM. Se estima que las capas convolucionales y capas FC constituyen el 99% del total de pesos y operaciones de la red [17].

Hasta el momento de la revisión, la interfaz de software soportada por la mayoría de las herramientas es Caffe, observándose también una tendencia a incorporar TensorFlow debido a la popularidad que ha venido ganando dicha herramienta. En cuanto a la portabilidad en FPGAs DnnWeaver tiene la capacidad de aplicarse a FPGAs de Xilinx, Intel y Arria, siendo Xilinx la marca de FPGAs que la mayoría de herramientas puede mapear [16].

En cuanto a la arquitectura, existen dos tipos principales, que se encuentran ilustrados en la Figura 5:

- Arquitectura “streaming”: Se crean bloques distintos de HW para cada capa, donde cada bloque es optimizado de manera independiente para aprovechar el paralelismo de su capa. Los bloques se encadenan para formar un flujo (pipeline). Este diseño aprovecha el paralelismo entre las capas y puede utilizar ejecución concurrente. La alta eficiencia implica que hay tiempos largos de compilación, debido a que se debe generar un bitstream diferente para cada red neuronal. A esta arquitectura pertenecen hls4ml fpgaConvNet, DeepBurning, Haddoc2, AutoCodeGen y FINN.
- Máquinas de computación sencillas: Favorecen la flexibilidad sobre la personalización, se denominan arreglos sistólicos de elementos procesantes o unidades de multiplicación matricial que ejecutan la NN secuencialmente. En otras palabras, es una plantilla fija de arquitectura que se puede escalar con base en las entradas y los recursos de la FPGA, así, cada NN corresponde a una secuencia diferente de microinstrucciones ejecutadas por el HW. El control de hardware y la programación de operaciones se realizan por software. Por esto, puede ejecutar diferentes NNs sin la necesidad de

reconfiguraciones. Debido a la ganancia en flexibilidad, se introducen algunas ineficiencias por la semejanza al funcionamiento de un procesador. A esta arquitectura pertenecen Angel-Eye, ALAMO, DnnWeaver, Caffeine, FP-DNN, Snowflake, SysArrayAccel y FFTCodeGen.

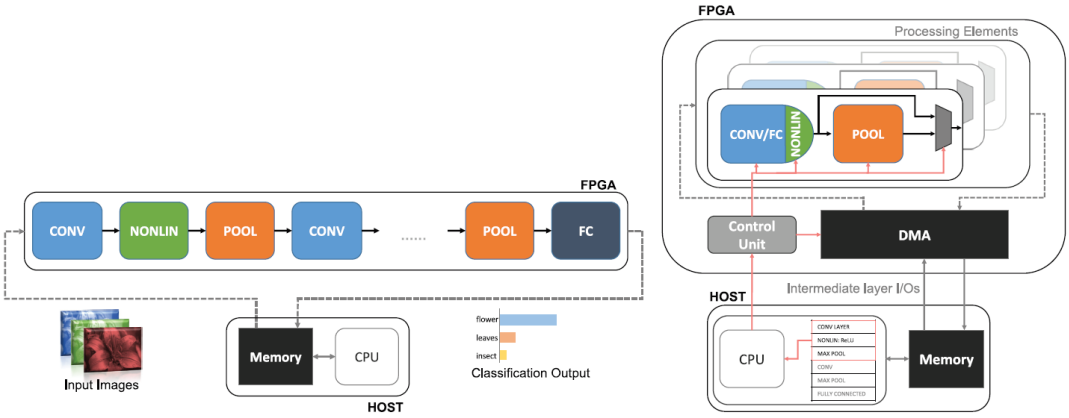

Figura 5: Arquitecturas de HW para la aceleración de NNs en FPGA [16]

A la izquierda de la Figura 5 se muestra una arquitectura *streaming*, y a la derecha un arreglo sistólico.

Para poder realizar la implementación, las herramientas deben realizar una parametrización de la FPGA objetivo. Cada herramienta define un espacio de diseño de arquitectura, en donde un punto en dicho espacio corresponde a una latencia, rendimiento (*throughput*), uso de recursos y eficiencia energética. Cada herramienta crea entonces un modelo matemático con las relaciones entre estas variables, y buscan minimizar alguna de estas mediante un problema de optimización. Las herramientas que permiten una mayor personalización de dichas optimizaciones y una exploración más detallada del espacio de parámetros tienen ventaja sobre otras herramientas [16].

Adicionalmente, estas herramientas llevan a cabo diferentes optimizaciones al proceso de inferencia, entre las más comunes son la reducción de precisión en la representación numérica (llamado también cuantización), donde usualmente se pasa de punto flotante de 32 bits, a punto fijo de 16, u 8 bits como los más comunes, buscando la mínima pérdida de precisión en el proceso. Existen cuantizaciones lineales, no lineales, uniformes y no uniformes [17].

Otra optimización que suele realizarse es la reducción del número de pesos de la red, mediante la aproximación a matrices de menor dimensión, que puede realizarse por medio de técnicas como el análisis factorial y la descomposición en valores singulares. Otra técnica es la eliminación de pesos menores a un cierto valor umbral (*pruning*) [17].

Así, se ha encontrado que las herramientas que generan diseños altamente optimizados a nivel RTL (Register-Transfer Level, explicado en la sección 2.1) tienden a sobrepasar a las que no. Por esto, se dice que existe un *trade-off* (Solución de compromiso) entre rendimiento RTL y la productividad de alto nivel (HLS). También se ha encontrado que las arquitecturas de arreglo sistólico tienden a tener mejor rendimiento con redes que tienen estructuras uniformes, por ejemplo, igual dimensión de núcleos de convolución a través de las diferentes capas.

Por esto, las diferentes herramientas tienen distintas prioridades, lo que las hace más adecuadas en diferentes contextos. Por ejemplo, las que favorecen el rendimiento (*throughput*) tienden a ser más adecuadas en centros de datos, y las que favorecen la latencia pueden ser más adecuadas para aplicaciones embebidas.

Finalmente, los retos que tienen dichas herramientas son:

- La implementación de nuevas arquitecturas de CNN y NN: cada vez aumentan la complejidad, el tamaño y la variabilidad de las capas.
- El soporte de redes comprimidas y dispersas (*sparse*)
- El soporte de precisiones bajas y ultrabajas: Cuantización uniforme o dinámica.
- La integración con infraestructuras existentes del ecosistema de Deep Learning: Tensorflow, PyTorch, Caffe2, y las distintas versiones que van apareciendo.
- El entrenamiento basado en FPGA: entrenamiento de baja precisión, entrenamiento con bajo uso de energía.
- Co diseño hardware-software: flujos de trabajo completamente automatizados, inicio desde un conjunto de datos y la definición del objetivo, para la automatización de creación del modelo, rendimiento de HW, etc. Este es un objetivo a largo plazo [16].

En la siguiente tabla se resumen las características de algunas de las herramientas revisadas:

Nombre Herramienta	Interfaz	Modelos de NN	Dispositivos	Arquitectura	Precisión	Exploración del espacio de diseño (DSE)	Prioridad de optimización	Open Source
fpgaConvNet	Caffe, Torch, Tensorflow, Theano	CNN, Res, Incep, Dense	Xilinx SoC	Streaming	FXP (Uniforme), FP	Optimizador global	Latencia o rendimiento	No
DeepBurning	Caffe	CNN, RNN, DNN	Xilinx SoC	Streaming	FXP (Dinámica)	Heurístico		No
Angel-Eye	Caffe	CNN, DNN	Xilinx SoC	Arreglo sistólico	FXP (Dinámico)	Heurístico con modelo analítico	Latencia y rendimiento son co-optimizados	No
DnnWeaver (2.0)	Caffe, Tensorflow	CNN, DNN	Xilinx, Intel	Arreglo sistólico	FXP (Dinámico)	Algoritmo de búsqueda personalizada	Portabilidad y alto rendimiento	Si
FP-DNN	Tensorflow	CNN, RNN, DNN, Res	Intel	Arreglo sistólico	FXP (uniforme), FP	Algorítmico	Alto rendimiento	No
FINN	Theano	BNN	Xilinx SoC & Standalone	Streaming	Binary	Heurístico	Alto rendimiento y baja latencia	Si
Haddoc 2	Caffe	CNN,DNN	Xilinx & Intel Standalone	Streaming	FXP (uniforme)	Determinista	Alto rendimiento y baja latencia	Si
Hls4ml	Tensorflow, PyTorch, ONNX	CNN, DNN	Xilinx	Streaming	FXP (Dinámico)	Manual, dada por el usuario	Recursos o latencia	Si
FXP: Punto fijo, FP: Punto flotante, ND: Información no disponible a la fecha								

Tabla 1: Herramientas de mapeo de NN en FPGA

De este modo, se puede ver que las herramientas ofrecen diferentes características, que pueden ser más adecuadas en ciertos contextos. Un gráfico comparativo de estas herramientas se encuentra a continuación:

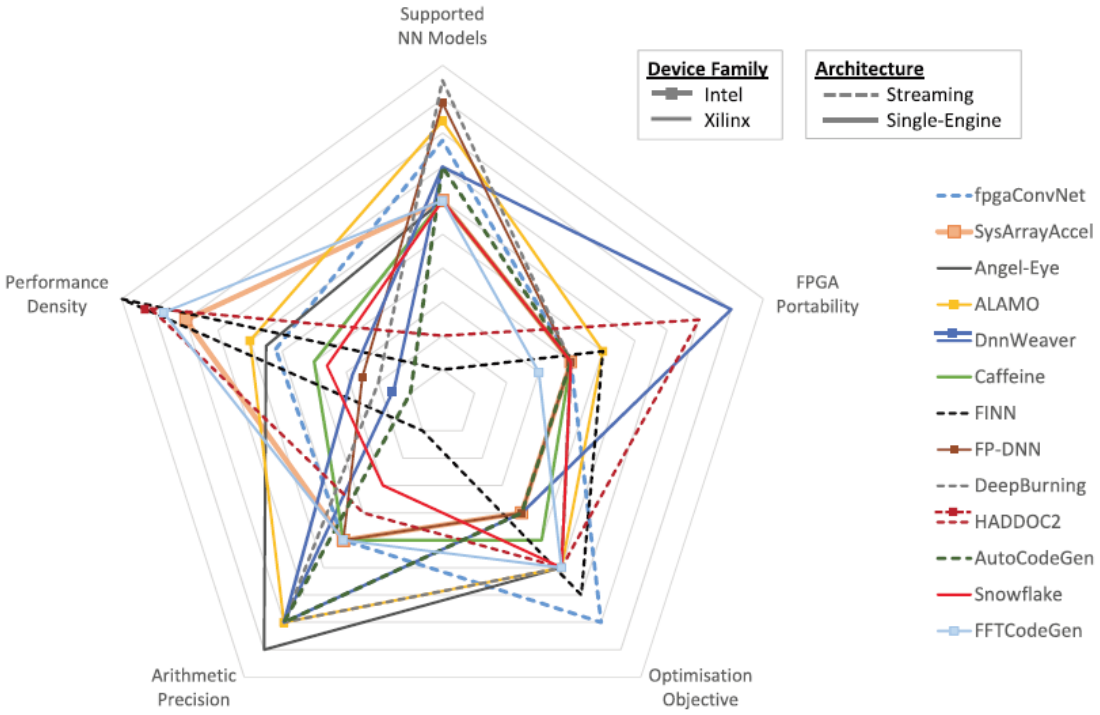

Figura 6: Comparación herramientas FPGA [16]

En la Figura 6 se puede observar cinco características de las herramientas: Modelos soportados, portabilidad, objetivo de optimización, precisión aritmética, y densidad de rendimiento, además de incluir la información de fabricante de los dispositivos soportados, más el tipo de arquitectura implementado por cada herramienta [16].

1.4.1 Selección herramienta a utilizar: hls4ml

Teniendo en cuenta lo discutido anteriormente y evaluando los objetivos de este trabajo, se llevó a cabo la selección de la herramienta a utilizar. El primer criterio que se tuvo en cuenta fue la disponibilidad de las herramientas, pues algunas no son de código abierto, lo que dificulta enormemente el acceso a estas. Por tanto, el primer filtro fue que la herramienta fuera de código abierto.

El segundo criterio fue que la herramienta estuviera disponible para las FPGA de Xilinx, que son las que se encuentran disponibles en el instituto de física de la Universidad de Antioquia, en particular, si hay disponibilidad para los chips de la tarjeta PYNQ-Z1 (xc7z020), la tarjeta ZYBO (xc7z010), o la tarjeta ARTY (xc7a35t).

Adicionalmente, se tuvo en cuenta la flexibilidad en el tipo de optimización, ya fuera para latencia o para utilización de recursos, pues esto permite la posible coexistencia de diferentes algoritmos para ser implementados en la FPGA, asunto importante en instrumentación científica, aplicaciones embebidas, y conducción inteligente. Por esto, en caso de haber varias herramientas con características similares en los anteriores criterios, se priorizó la que tuviera resultados buena flexibilidad en la optimización, unido a un rendimiento aceptable.

La herramienta que reunió todas estas características fue **hls4ml**, que adicionalmente es una herramienta desarrollada en el contexto del Gran Colisionador de Hadrones (LHC), el proyecto de física experimental de mayor envergadura en el mundo actualmente. Los detalles sobre el uso y el origen de esta herramienta se encuentran en el capítulo 3.

1.5 Introducción a la arquitectura de las FPGA

Antes de continuar, en esta sección se hará una ilustración de qué es una FPGA y los elementos de que se componen. Desde su aparición, las FPGA han venido creciendo, conteniendo arreglos masivos de lógica programable e interconexiones, con memorias en-chip, caminos de datos personalizados, I/O de alta velocidad, y microprocesadores, todos en el mismo chip. En esta sección se busca dar una mirada a la arquitectura general sin profundizar más de lo necesario para comprender lo más relevante de la programación de las FPGAs.

La FPGAs son arreglos de bloques lógicos programables y elementos de memoria unidos entre sí mediante interconexiones programables. Normalmente tales bloques lógicos se implementan como “tablas de consulta” (LUT: *LookUp Tables*), que son memorias cuya señal de dirección son las entradas, y las salidas están precalculadas y almacenadas en dicha LUT. Mas aún, una LUT de *n* bits puede programarse para calcular cualquier función booleana de *n* entradas al usar la tabla de verdad de la función como los valores de la memoria LUT, y por tanto pueden almacenar los resultados de cualquier función representable en bits, con lo cual pueden acelerar considerablemente muchas computaciones, dado que es más eficiente encontrar el resultado que calcularlo [15].

Los Flip-Flop (FF) es el elemento de memoria básico de la FPGA, y típicamente están ubicados junto a las LUTs. Cuando se tienen varias LUTs con varios FF y otros componentes especializados (como un sumador, o multiplicador), se obtiene un elemento lógico más complejo llamado bloque lógico configurable. El nombre utilizado por Xilinx es *slice* (Porción), donde un *slice* es un número pequeño de LUTs, FF y multiplexores combinados para hacer un elemento lógico más potente. El número exacto de cada componente es variable según la tarjeta y arquitectura, pero en general, cada elemento lógico se compone de unos pocos componentes. [15]. En la siguiente imagen se ejemplifica el funcionamiento de los *slices*:

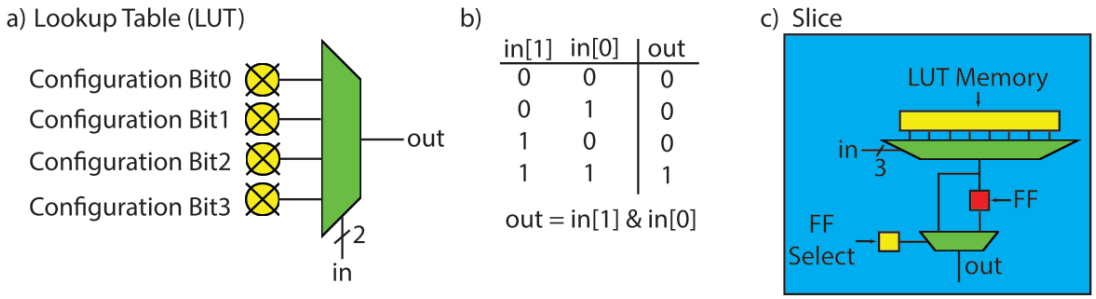

Figura 7: LUTs y slices [15]

La figura anterior en la parte a) muestra una LUT de dos entradas (2-LUT). Cada uno de los 4 bits de configuración pueden programarse para cambiar la función de la LUT, lo que la convierte en una compuerta lógica completamente programable. La parte b) muestra un ejemplo de programación para implementar una compuerta AND: los valores de la columna “out” corresponden a los bits de configuración 0 a 3. La parte c) muestra un *slice* que contiene una 3-LUT con la posibilidad de almacenar el resultado en un FF. Aquí hay nueve bits de configuración: ocho para programar la 3-LUT y uno para decidir si el resultado se obtiene directamente de la LUT o el que se almacena en el FF. En general un *slice* es un número pequeño de LUTs y FFs combinado con multiplexores para mover inputs, outputs y valores internos entre las LUTs y los FFs. Las FPGAs pueden tener millones de estos *slices* programables.

Adicionalmente, las interconexiones programables son otro elemento clave de las FPGA, pues proveen una red flexible de “cables” para crear conexiones entre las *slices*. Los inputs y outputs de cada *slice* están conectados a un canal de enrutamiento, que contiene un conjunto de bits de configuración para conectar o desconectar el *slice* a la interconexión programable. Los canales de enrutamiento están conectados a “cajas de suiches” (*Switchboxes*), que consisten en transistores de paso, los cuales tienen la habilidad de conectar los canales de enrutamiento entre sí. La siguiente figura presenta un ejemplo de conexión entre los *slices*, canales de enrutamiento y switchboxes:

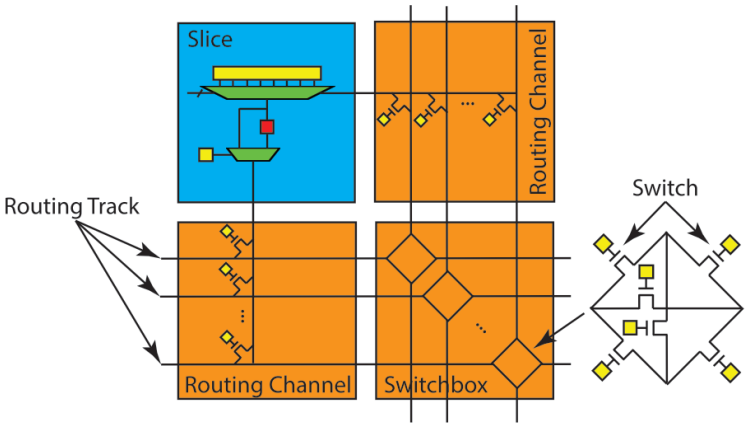

Figura 8: Slices, canales de enrutamiento y switchboxes [15]

Cada entrada o salida de los *slices* puede ser conectada a alguna de las líneas de enrutamiento (*Routing track*) que existen en el canal de enrutamiento (*Routing channel*). Puede pensarse en las líneas de enrutamiento como cables de un solo bit. Las conexiones físicas entre el *slice* y las líneas se configuran con los transistores de paso. Los *switchboxes* proveen una matriz de conexiones entre líneas de enrutamiento.

Finalmente, la FPGA consiste en múltiples arreglos como el anterior. Esto se ilustra en la siguiente figura:

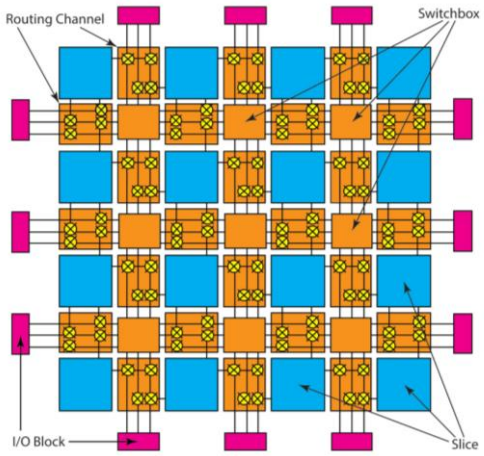

Figura 9: Arquitectura FPGA [15]

En la anterior figura se aprecia la estructura 2-dimensional de las FPGA. Los bloques I/O son interfaces externas, por ejemplo, con un procesador, una memoria, un sensor, o un actuador, etc. En general, no es necesario comprender al detalle el funcionamiento de cada componente de la FPGA si se utilizan herramientas de alto nivel como la síntesis de alto nivel (HLS), pero sí es conveniente tener una comprensión general de los recursos de la FPGA para saber qué impacto tiene en la utilización de recursos el usar directivas de optimización en HLS. Esto se ejemplifica en la sección 2.1.

Por ejemplo, es una realidad que cada vez las FPGA van incorporando cada vez más elementos arquitecturales, como bloques RAM (BRAM), o DSPs. Un bloque RAM es una RAM configurable que puede implementar diferentes configuraciones de memoria e interfaces, por ejemplo, pueden almacenar bytes, media palabra, una palabra, palabras dobles, y conectarse a buses en-chip, y buses del procesador. Dependiendo de la cantidad de datos que es necesario almacenar en la FPGA, se pueden elegir diferentes recursos de memoria. La siguiente tabla ilustra esta disyuntiva:

	Memoria externa	*BRAM*	*FFs*
Cantidad	1-4	miles	Millones
Tamaño	GBytes	KBytes	Bits
Tamaño total	GBytes	MBytes	100s de KBytes
Ancho	8-64	1-16	1
Total ancho de banda	GBytes/s	TBytes/s	100s de TBytes/s

Tabla 2: Recursos de memoria FPGAs

De la tabla puede observarse que mientras mayor sea la cantidad de datos a almacenar será necesario implementar un recurso de memoria con mayor capacidad, pero con menor velocidad de transmisión. Esto es fundamental al escribir directivas de optimización y de asignación de recursos al buscar la aceleración de funciones mediante HLS. Este asunto es crucial en los resultados de los capítulos 2 y 3.

1.6 Flujo de trabajo para la programación de FPGAs

En este punto es importante aclarar que existen diversas maneras de programar las FPGA. En esta sección se busca aclarar el flujo de trabajo al programar FPGAs utilizando herramientas de síntesis de alto nivel (HLS: *High-level Synthesis*), pero también se mencionan otras formas de programas FPGAs, que son similares al flujo usando HLS.

Lo primero es comprender que en general el objetivo final es crear una aplicación de software (anfitriona o *host*) que contiene una o varias funciones aceleradas mediante el uso de la lógica programable (PL: *Programmable Logic*) de la FPGA. La aplicación de software anfitriona puede tener múltiples funciones como por ejemplo leer varias fuentes de datos, realizar pre y post procesamiento, imprimir en pantalla o en archivo diferentes resultados, entre otros. La función acelerada por hardware es solo una parte de la aplicación anfitriona, y comúnmente, recibe unos inputs, realiza un procesamiento y devuelve el resultado [18].

En general no es necesario ni conveniente acelerar por hardware todos los componentes de la aplicación anfitriona, sino identificar aquellas funciones más intensivas y paralelizables que se quiere descargar en el PL. Al final, la aplicación anfitriona “llama” la función acelerada mediante el uso de APIs o protocolos que permiten tal comunicación. Por tanto, son dos tareas con cierta independencia el desarrollo de la aplicación anfitriona y el desarrollo de la aceleración por hardware de una función específica. El siguiente esquema ilustra cómo estos dos procesos se relacionan y cómo cada uno puede ser un proceso iterativo.

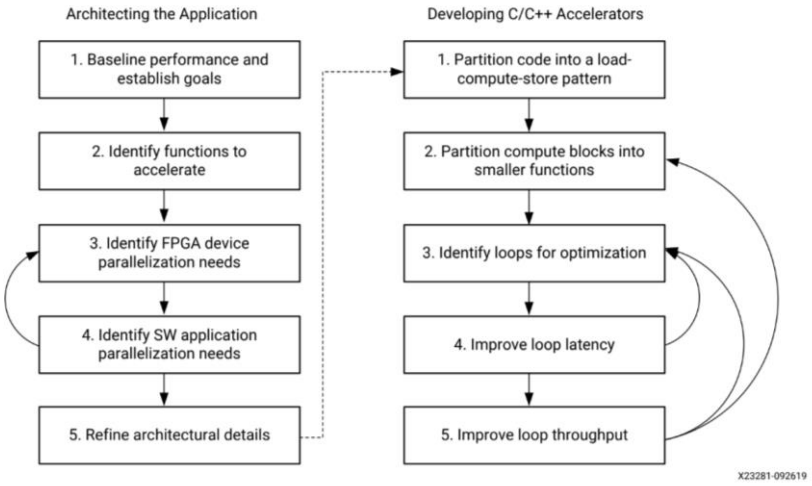

Figura 10: Flujo de trabajo al programar FPGAs [18]

A estos dos procesos corresponden herramientas y flujos diferentes: El desarrollo de la aplicación anfitriona para FPGAs de Xilinx se puede realizar en Xilinx SDK o Vitis, y consiste en una aplicación estándar en C o C++ embebido. Esta aplicación tiene una función principal (*main*) y debe inicializarse, cargar librerías, inicializar los controladores de los dispositivos de hardware creados para la aceleración, cargar los datos, procesar y devolver el resultado, pero también es posible que se desee que tal aplicación anfitriona haga otras cosas, como un preprocesamiento de los datos, o la impresión en pantalla de resultados, o el cálculo del tiempo que se toma una sección de código, o un resumen, etc.

Por otra parte, la función acelerada por hardware se puede implementar utilizando la síntesis de alto nivel (HLS), esta función acelerada en general consiste en una función que tiene parámetros, escalares o vectores de entrada, y escalares o vectores de salida. Esta es la función que interesa ejecutar en el PL, pues se quiere paralelizarla y acelerarla. Dicha función está escrita en C o C++, y su ejecución se optimiza mediante el uso de diferentes directivas que se insertan en el código. Este es en realidad el centro del presente trabajo, pues esta es la funcionalidad específica que hace que las FPGA puedan ser más rápidas que otros dispositivos al ejecutar diferentes funciones. El proceso de acelerar funciones con el uso de HLS es el objetivo de los capítulos 2 y 3.

Ahora bien, para ilustrar de manera más clara el proceso de programación de las FPGA utilizando la síntesis de alto nivel, se presentan los siguientes diagramas que explican el flujo de trabajo completo desde el código en C para la aceleración de una función, hasta la implementación de la aplicación anfitriona en la FPGA:

1. Se comienza con un código en C/C++ de una función que se desea acelerar. Luego de diseñar cuidadosamente el código y aplicar las optimizaciones, el objetivo es generar un “bloque” que contiene la función ya acelerada e implementada en un lenguaje de bajo nivel (VHDL o Verilog, explicados brevemente en la sección 2.1). Este proceso es propiamente la síntesis de alto nivel (HLS)

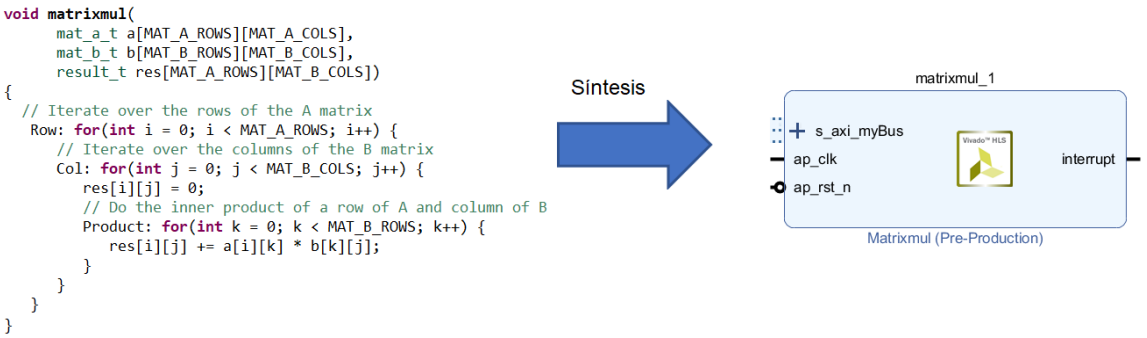

Figura 11: Síntesis de un bloque IP en HLS

2. Este bloque se integra a otros bloques en Vivado Design Suite, para especificar completamente la arquitectura que controla el bloque creado en HLS, de modo que el bloque se comunique con el procesador de la FPGA y otros dispositivos que puedan ser necesarios, como puertos de entrada o salida, contadores de tiempo, estructuras de memoria, entre otros, así el bloque podrá recibir las entradas, ejecutar la función acelerada y devuelve el resultado. El objetivo es generar un archivo llamado *bitstream* que contiene las instrucciones de programación de la FPGA.

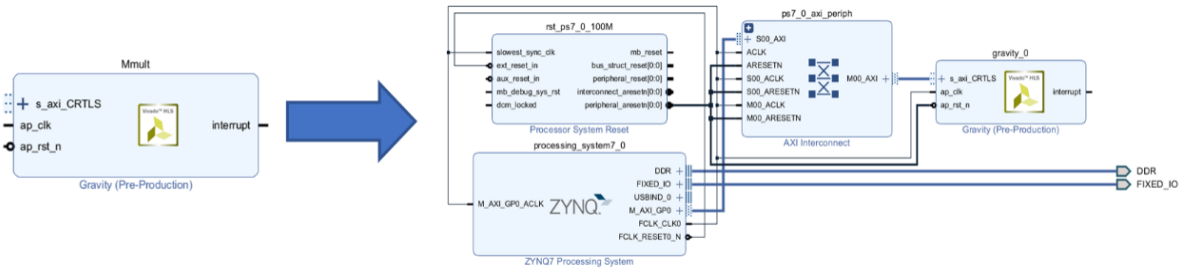

Figura 12: Diseño de bloques en Vivado Design Suite

3. Por último, se programa la FPGA (Se carga el *bitstream* en la tarjeta), y con la ayuda de Vivado SDK (*Software Development Kit*) o Vitis, se escribe y ejecuta un programa anfitrión en C embebido que controla el comportamiento de la FPGA, inicializa el "bloque", envía las entradas, y lee las salidas, además de otras posibles funcionalidades más generales, como la impresión en pantalla o archivos de diferentes resultados, la toma de tiempo, y el pre o post procesamiento de los datos.

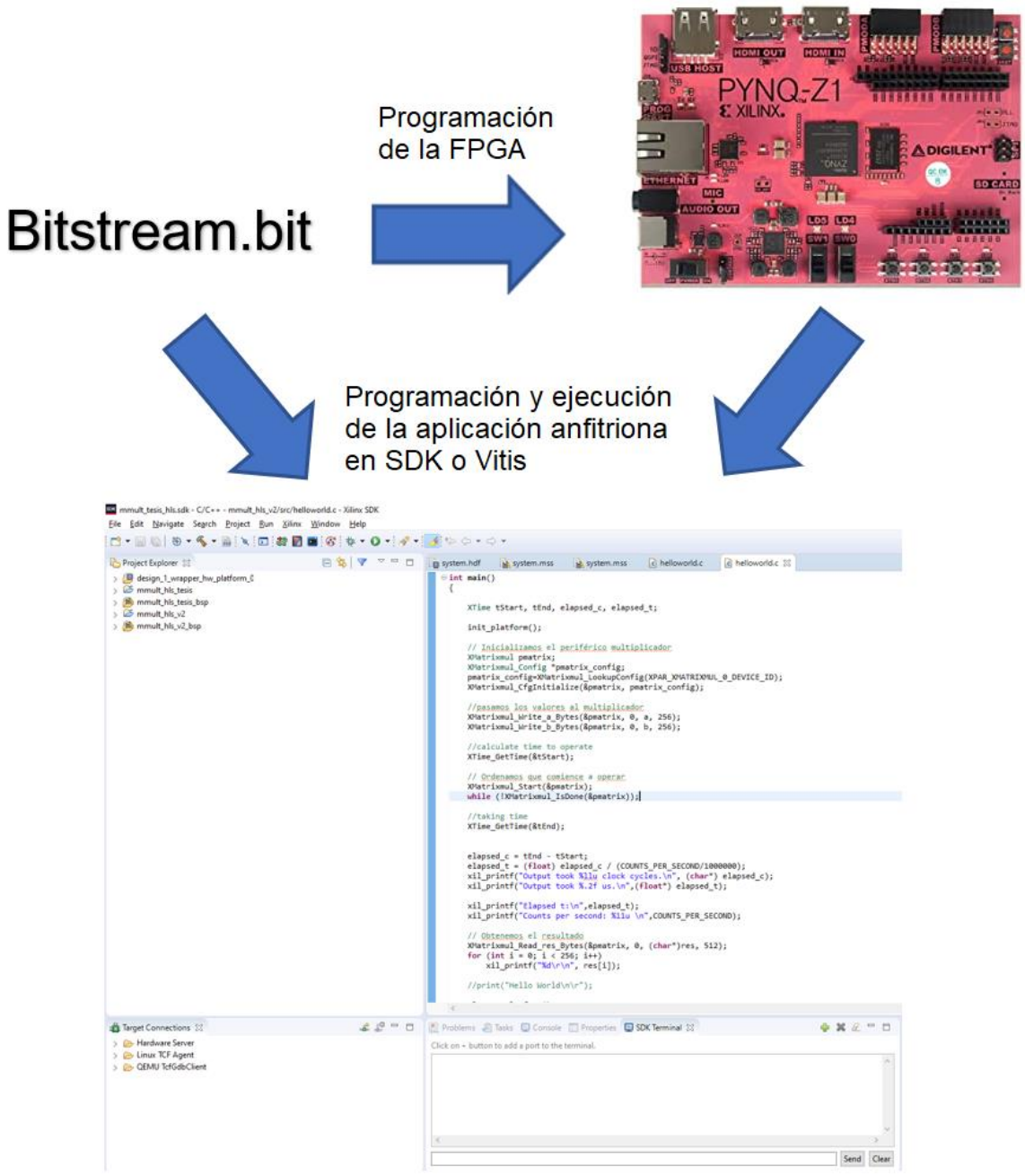

Figura 13: Programación y ejecución de la aplicación anfitriona

Es importante notar que cada uno de estos tres pasos tiene una complejidad considerable. En el capítulo 2, se desarrollan los anteriores pasos en la utilización de síntesis de alto nivel para la implementación de redes neuronales en FPGAs.

Adicionalmente al flujo anteriormente presentado, existen otros posibles flujos para programar una FPGA, los más notables son:

Flujo alternativo	**Descripción**
Creación de IPs utilizando los lenguajes VHDL o Verilog	Para el paso 1 del flujo con HLS, en lugar de utilizar lenguaje C/C++ y HLS, se programa la función “acelerable” utilizando VHDL o Verilog, el cual es un proceso mucho más dispendioso y demorado, pero puede implicar un mejor rendimiento. El resto de pasos (2 y 3) son idénticos.

Uso de IPs preprogramados, o de terceros	Puede omitirse el paso 1 del flujo con HLS, utilizando IPs que vienen incluidos con Vivado Design Suite si éstos pueden realizar la función deseada, o existen empresas que desarrollan IPs para su venta. La limitación es que los IPs incluidos en Vivado Design Suite son funciones muy generales. El resto de pasos (2 y 3) son idénticos.
Programación de aplicación anfitriona en entorno PYNQ	Sólo cambia el paso 3, ya que luego de tener el *bitstream*, es posible utilizar un entorno Jupyter Notebook de Python (en lugar de SDK o Vitis) para programar aplicaciones anfitrionas y llamar funciones aceleradas. Esta posibilidad aplica sólo para ciertas tarjetas. En este caso la aceleración de funciones aún debe resolverse, ya sea mediante HLS, o alguno de los dos flujos alternativos mencionados más arriba de esta tabla.

Tabla 3: Flujos alternativos a HLS para programar FPGAs

1.7 Objetivos del presente trabajo

1.7.1 Objetivo general

Evaluar la implementación de redes neuronales densas en FPGA respecto a su tiempo de ejecución y utilización de recursos, implementadas mediante el uso de síntesis de alto nivel y librerías de mapeo de código abierto, aplicadas a problemas en física y visión artificial.

1.7.2 Objetivos específicos

- Comprender el proceso de programación de las FPGA mediante el uso de síntesis de alto nivel
- Optimizar y evaluar el tiempo de ejecución y la utilización de recursos de la implementación de multiplicación matricial en la FPGA xc7z020clg400-1 mediante el uso de síntesis de alto nivel
- Optimizar y evaluar la implementación el tiempo de ejecución y la utilización de recursos de redes neuronales densas en la FPGA xc7z020clg400-1 mediante el uso de síntesis de alto nivel
- Optimizar y evaluar el tiempo de ejecución y la utilización de recursos de la implementación de redes neuronales densas en la FPGA xc7z020clg400-1 mediante el uso de una librería especializada de código abierto

2 IMPLEMENTACIÓN DE REDES NEURONALES MEDIANTE EL USO DE SÍNTESIS DE ALTO NIVEL (HLS)

Antes de pasar a los detalles de implementación de HLS, se va a realizar una introducción a este concepto, lo que permitirá construir las bases teóricas y prácticas de este importante concepto para los siguientes capítulos.

2.1 Introducción a la síntesis de alto nivel (HLS)

En el proceso de diseño de hardware (HW), ha habido diferentes etapas a través de la historia. En un comienzo, con circuitos pequeños, era posible especificar cada transistor, cada elemento, sus conexiones, y todo de manera manual. A medida que fue creciendo la complejidad de los circuitos, fueron apareciendo herramientas de diseño automatizadas, de modo que fue posible trabajar en niveles más altos de abstracción, especificando circuitos, en vez de componentes individuales [15].

Luego en los años 80 apareció la aproximación de Mead y Conway, de utilizar un lenguaje de descripción de hardware como Verilog o VHDL (VHSIC-HDL, *Very High Speed Integrated Circuit Hardware Description Language*) que se usan para describir sistemas digitales y de señales mixtas. Sin embargo, la complejidad siguió aumentando, lo que llevó a la aparición del nivel RTL (Register-Transfer Level) que era un paso más alto en abstracción, permitiendo especificar los registros y operaciones que se llevaban a cabo en estos registros. Herramientas automáticas podían traducir estas especificaciones en circuitos físicos. Esto permitió construir circuitos muy complejos sin perderse en los detalles de cómo estaban construidos [19].

La síntesis de alto nivel es un paso todavía más en abstracción que permite enfocarse en cuestiones más generales de arquitectura, en lugar de en registros u operaciones ciclo a ciclo. Es posible capturar el comportamiento de un programa que no especifica registros o ciclos, y una herramienta HLS crea la microarquitectura RTL detallada. Fundamentalmente, las herramientas HLS realizan de manera automática las siguientes tareas:

- Analizan y aprovechan la concurrencia de los algoritmos
- Insertan registros como sea necesario para limitar caminos críticos y alcanzar una frecuencia de reloj deseada
- Genera una lógica de control que dirige el camino que siguen los datos
- Implementa interfaces que conectan el resto del sistema
- Mapea datos en elementos de almacenamiento para equilibrar el uso de recursos y el ancho de banda
- Mapea la computación en elementos lógicos que llevan a cabo optimizaciones automáticas o especificadas por el usuario, para alcanzar la implementación más eficiente

En general, el objetivo del HLS es tomar estas decisiones automáticamente, con base en especificaciones de entrada dadas por el usuario, y las ligaduras del diseño (*design constraints*), como por ejemplo, las características propias de la FPGA particular en que se desea implementar el programa. Hay muchas herramientas HLS que están en una etapa madura de desarrollo, como Vivado HLS [20], LegUp [21], o Mentor Catapult HLS [22]. En el caso del presente trabajo se utiliza Vivado HLS, pues corresponde al producto de una de las marcas más fuertes, y comunes de FPGAs: Xilinx. Esto también porque esta marca es la que corresponde a las FPGA disponibles en la Universidad de Antioquia. En la siguiente figura se ilustra en general el proceso iterativo que se sigue al utilizar HLS

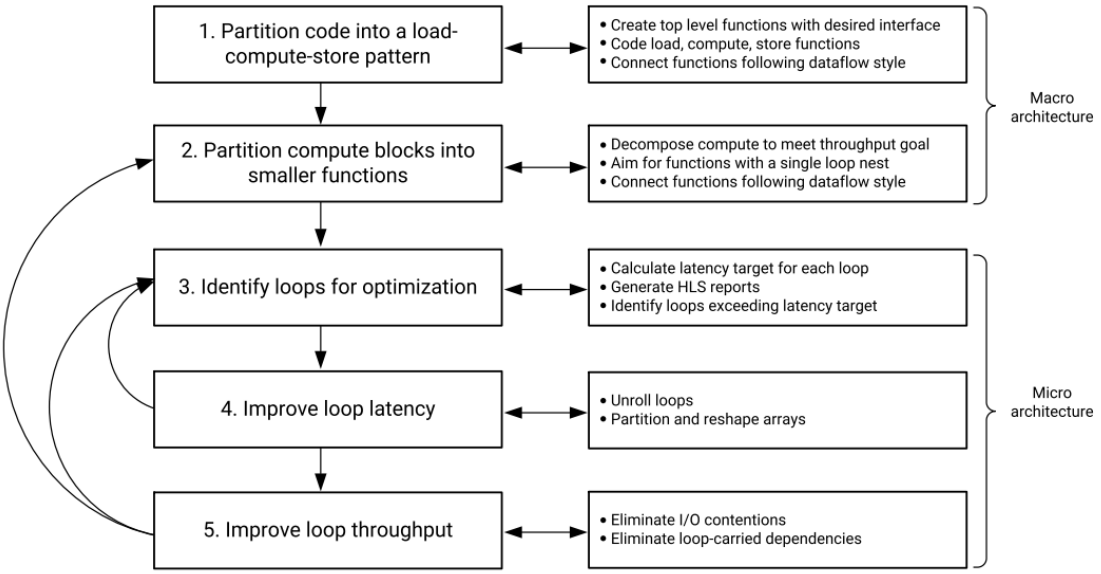

Figura 14: Proceso Iterativo HLS [18]

Lo que la anterior figura ilustra como la macro-arquitectura, es en general el programa o algoritmo escrito en C/C++. Luego, la micro-arquitectura tiene que ver con la inserción de directivas de interfaces y optimización en puntos específicos del código, para personalizar y guiar el programa en la síntesis de la función. Este proceso se ejemplifica en la sección 2.2.

En particular, Vivado HLS requiere las siguientes especificaciones:

- Una función especificada en C, C++ o C embebido que es la que se quiere acelerar
- Un programa de prueba (*Test Bench*) que llama la función a acelerar y verifica que se ejecuta correctamente al probar sus resultados
- Una tarjeta FPGA objetivo (*Constraints*)
- La velocidad de reloj esperada (*Constraints*)
- Directivas que guían el proceso de implementación

La herramienta HLS tiene las siguientes limitaciones en cuanto a la función a especificar:

- No se permite alocación de memoria dinámica (no se permiten operadores como malloc(), free(), new() o delete()).
- Uso limitado de punteros a punteros
- No se soportan algunas funciones de sistema (como printf(), abort(), exit())
- Uso limitado de librerías estándar (como algunas funciones poco frecuentes de math.h)
- Uso limitado de punteros de funciones y funciones virtuales en C++
- No se permite el llamado recursivo de funciones
- La interfaz debe estar definida de manera precisa

La herramienta HLS entrega estimativos de uso de recursos y rendimiento, y genera las siguientes salidas:

- Un bloque de código Verilog o VHDL sintetizable
- Simulaciones RTL basadas en la función de prueba
- Análisis estático de rendimiento y uso de recursos
- Datos respecto a las limitaciones del diseño

Los requerimientos y salidas del proceso HLS se ilustran de manera general en la siguiente figura:

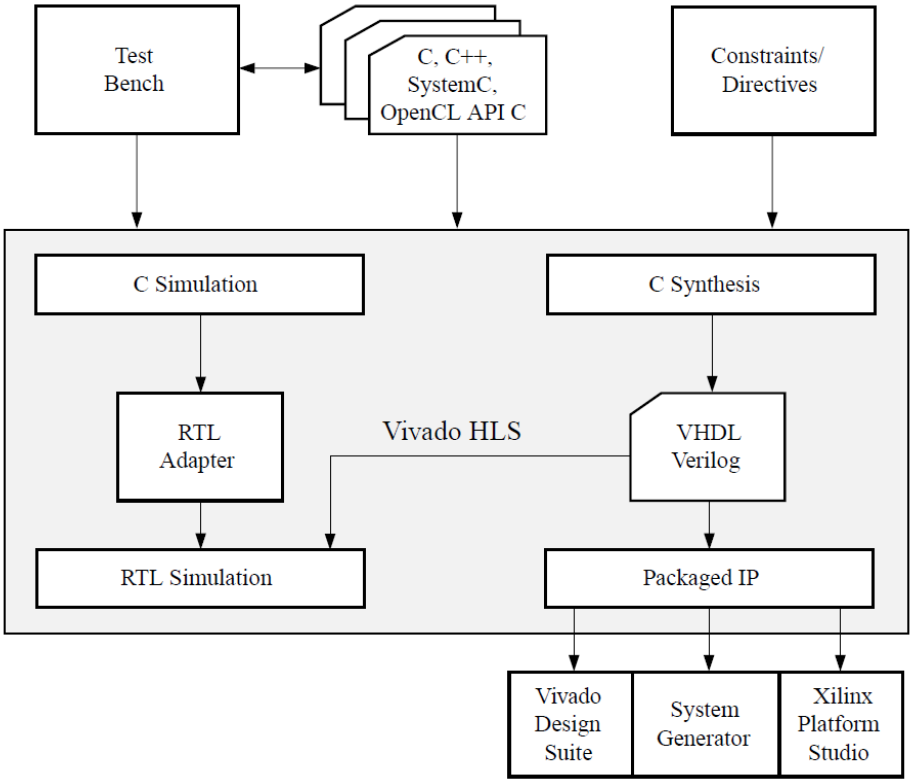

Figura 15: Diagrama de las entradas y salidas HLS [20]

El resultado final es entonces un código RTL que está empaquetado como una función ejecutable en la lógica programable de la FPGA, Este paquete se le llama también bloque IP (*Intellectual Property*), y este bloque debe ser utilizado luego en otro programa, como Vivado Design Suite, para completar el proceso de programación de la FPGA, tal como se mencionó en la sección 1.6.

Adicionalmente, la utilización de HLS es un proceso que en sí mismo implica ciertas etapas, en el siguiente diagrama se encuentran las mismas:

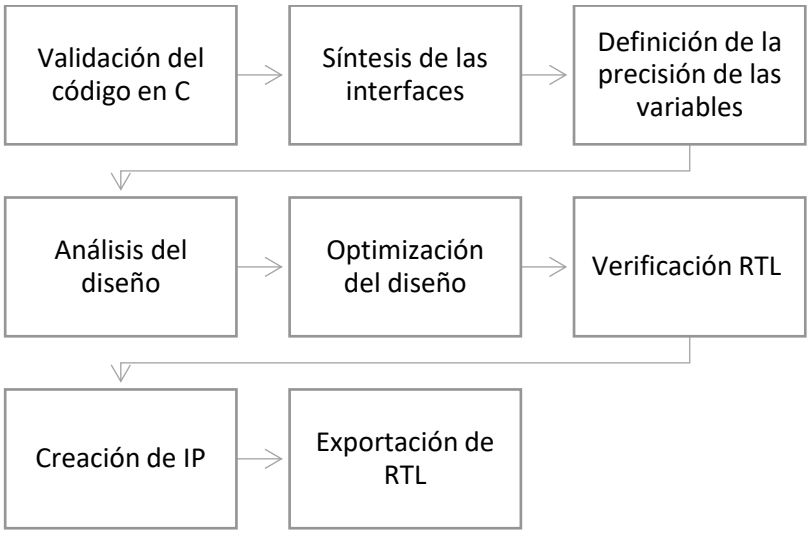

Figura 16: Pasos programación HLS

A continuación, se hará una breve explicación de cada uno de los pasos anteriormente mencionados:

Validación del código en C

En primer lugar, es conveniente tener un código de prueba para nuestra función (test bench), es decir, contar con unos resultados esperados para una entrada en particular, de modo que el programa puede verificar que la función en C y la implementación RTL ejecuta las instrucciones adecuadas. Esto con el objetivo de evitar malgastar tiempo analizando diseños que son incorrectos, además permite la verificación RTL automáticamente.

Un buen código de prueba realiza:

- La creación de un conjunto de resultados esperados que confirman que la función es correcta
- Guarda los resultados en un arreglo de software
- Devuelve un valor de cero si los resultados son correctos, así automatiza la verificación RTL [23]

Síntesis de las interfaces

Es el proceso de agregar puertos RTL al diseño en C, es decir se especifica por medio de cuáles dispositivos de hardware se controlarán las entradas, las salidas y el control global del bloque. Incluye un protocolo asociado de entrada y salida permitiendo la transferencia de datos entre los puertos, para ser sincronizados automáticamente y de manera óptima con la lógica interna.

Los protocolos de entrada y salida (I/O) a nivel de bloque controlan el bloque de manera independiente a los protocolos del nivel de puertos (explicados un poco más adelante), es decir, estos protocolos controlan cuándo el bloque puede comenzar a procesar datos, cuando está listo para aceptar entradas, indica si está activo o inactivo, o si completó la operación. Si la función devuelve un valor, el puerto de dicho valor pertenece al nivel de bloque. Por tanto, este protocolo se especifica sobre la función principal. Hay 4 opciones para el protocolo de interfaz de bloque:

- ap_control_none: no hay protocolo a nivel de bloque de I/O
- ap_ctrl_hs: protocolo handshake
- ap_ctrl_chain: para controlar encadenamiento de bloques (pipelining)
- s_axilite: implementa una interfaz AXI lite esclava

Hay que tener en cuenta que, con la primera opción, un bloque que recibe datos no puede saber cuándo los datos ya son válidos, además la co-simulación RTL requiere el protocolo de nivel de bloque para secuenciar el código de prueba.

Este protocolo de nivel de bloque permite que el diseño RTL pueda ser controlado por puertos adicionales de manera independiente a los puertos de entrada y salida. Este protocolo está asociado a la función en sí misma y no con los puertos de datos. El protocolo predeterminado para el nivel de bloque es *handshake* (ap_ctrl_hs), y se asocia con el valor de retorno de la función.

Por otra parte, una vez que el protocolo de nivel de bloque se ha utilizado para comenzar la operación del bloque, los protocolos de nivel de puerto se utilizan para secuenciar los datos de entrada y salida del bloque. La especificación de estas interfaces depende de si la entrada corresponde a un solo valor (enteros o flotantes como argumentos), o si la entrada corresponde a arreglos (arreglo como argumento).

Si hay argumentos o punteros sin protocolo I/O, los datos de entrada deben mantenerse estables hasta que sean leídos. De manera predeterminada los punteros de salida se implementan con una señal de salida válida asociada, para indicar cuándo los datos de salida son válidos. Por tanto, siempre es buena idea usar un protocolo I/O para una variable de salida. Algunas opciones para estos puertos son:

Para puertos de entrada
- ap_none: ningún protocolo I/O
- ap_ack: debe asociarse con un puerto de salida ack
- ap_vld: debe asociarse con un puerto de entrada vld
- ap_hs: puede asociarse con entrada vld o salida ack
- s_axilite: implementa el protocolo AXI Lite de entrada o salida

Para puertos de salida/entrada (punteros)
- ap_ack: debe asociarse con un puerto de entrada ack
- ap_vld: debe asociarse con un puerto de salida vld
- ap_hs: protocolo handshake para entrada o salida
- ap_fifo: interfaz FIFO con una salida asociada de escritura, y entrada FIFO completa
- ap_bus: protocolo de interfaz bus de Vivado HLS

Por otra parte, los arreglos que son argumentos de la función principal pueden implementarse como puertos RTL de diferentes tipos. De manera predeterminada, estos arreglos del código en C se sintetizan como puertos RTL RAM, pero otras opciones son: interfaz RAM de puerto sencillo o puerto doble, interfaz FIFO, y partición en puertos discretos.

Como por defecto, los ciclos *for* se mantienen enrollados (*rolled*), entonces se usa también una interfaz RAM de puerto sencillo, para asegurar que solamente se utiliza una señal sencilla de lectura o escritura de manera simultánea, incluso si hay múltiples entradas o salidas disponibles. Por tanto, para implementar un arreglo utilizando múltiples puertos, primero debe desenrollarse (*unroll*) los ciclos *for*, para permitir que las operaciones puedan llevarse a cabo en paralelo, de otro modo, no hay beneficio en implementar múltiples puertos.

Otra interfaz disponible es la AXI4-Stream, donde los arreglos pueden particionarse en canales, y se puede transferir el contenido de cada canal en paralelo, para asegurar hardware dedicado a cada canal, con una partición adecuada de los ciclos *for*. Por ejemplo, si la función necesita calcular primero ciertos valores que usará después, puede realizarse una partición cíclica de los arreglos, donde el ciclo sea el número de datos calculables antes de necesitar los nuevos datos calculados [23].

Datos de precisión arbitraria

Los tipos de datos en C/C++ están limitados a anchos de 8 bits:

Tipo	Ancho (bits)
char	8
short (entero)	16
int	32
long (entero)	64
float	32
double	64

Tabla 4: Precisiones predeterminadas C

En muchas ocasiones son necesarios anchos de bits de menor precisión, con el fin de evitar costos de hardware innecesarios, debido a que mayores anchos de bits requieren más LUTs y registros que los necesarios para una determinada precisión, trayendo retrasos que pueden exceder el ciclo de reloj, con lo cual se necesitan más ciclos para calcular el resultado.

Vivado HLS puede especificar datos de precisión arbitraria, si son enteros, o decimales de punto fijo. Para determinar los decimales de punto fijo, el código debe estar escrito en C++, no en C. Los datos de precisión arbitraria se especifican con el tipo: ap_fixed<Bits, Bit_int, rounding>, con Bits el número total de bits, Bit_int son los bits que representan números sobre el punto decimal, y rounding es un método de redondeo en caso de sobre- o sub- flujo [20].

Análisis del diseño

Un método general para el diseño de las implementaciones RTL de C/C++ incluye las siguientes tareas:

- Sintetizar el diseño
- Revisar los resultados de la implementación inicial
- Aplicar directivas de optimización para mejorar el rendimiento

Para esto, HLS incluye una herramienta de análisis de resultados para ayudar a determinar cuáles son las directivas adecuadas para aplicar.

Existen dos tipos de optimizaciones que pueden aplicarse a ciclos *for*: el desenrollado (*unroll*) y la segmentación (*pipelining*). Estos dos tipos de optimizaciones buscan aprovechar el paralelismo inherente a algunos ciclos. Normalmente, al ejecutar instrucciones de manera secuencial, sólo se comienza la ejecución de un ciclo luego de que el anterior ha terminado. Mediante la segmentación, se busca reducir el intervalo de iniciación entre ciclos, al ejecutar ciclos de manera concurrente, como se ilustra en la siguiente figura:

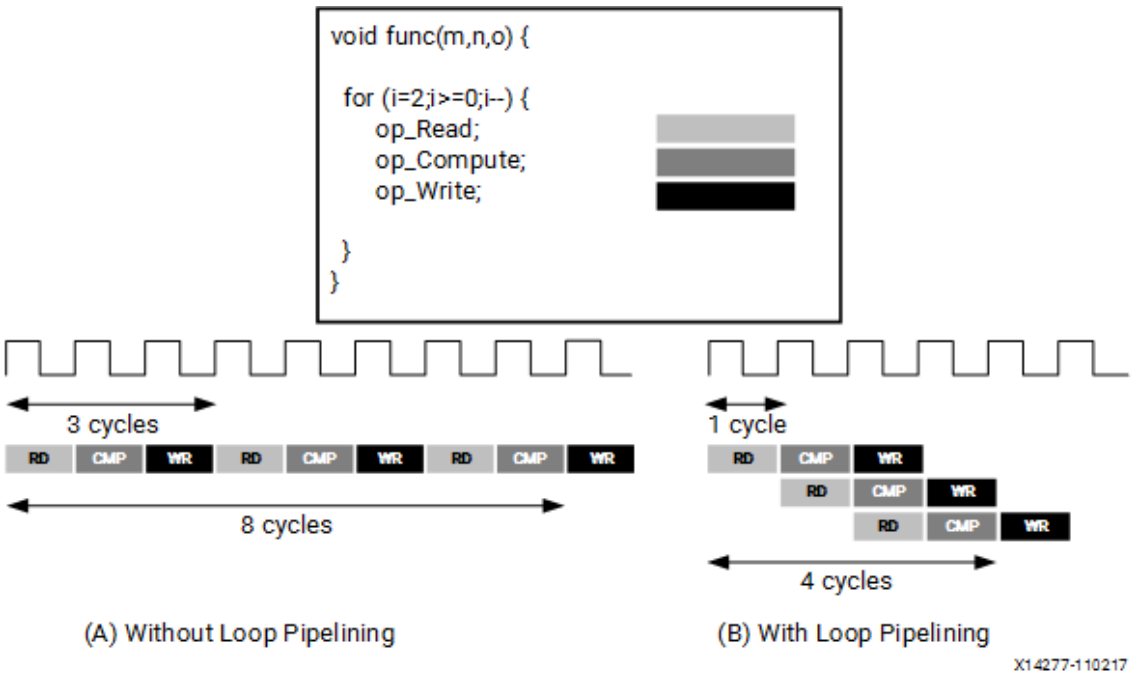

Figura 17: Segmentación de ciclos for [24]

Por otra parte, el desenrollado lo que busca es aprovechar posibles paralelismos al crear copias del ciclo y ajustar el contador de manera adecuada, un ejemplo se encuentra en el siguiente fragmento de código:

```
int sum = 0;
for(int i = 0; i < 20; i++) {
    sum += a[i];
}

//After the loop is unrolled by a factor of 2, the loop becomes:

int sum = 0;
for(int i = 0; i < 20; i+=2) {
    sum += a[i];
    sum += a[i+1];
}
```

Código 1: Desenrollado de ciclos for

Así, desenrollar por un factor N, crea N copias del cuerpo del ciclo, lo que genera más operaciones por ciclo y puede aprovecharse el paralelismo de estas operaciones, lo que suele traducirse en mayor rendimiento, pero mayor utilización de recursos de hardware. Si se necesitan datos calculados por el mismo ciclo para una orden siguiente, se dice que hay dependencia de los datos del ciclo, lo cual limita el paralelismo que se puede aprovechar. Otro factor limitante son los recursos de hardware [23].

Optimizaciones de diseño

Un objetivo típico, es buscar minimizar la latencia de ciclos y funciones, por eso, dentro de estas, HLS trata de ejecutar la mayor cantidad posible de operaciones en paralelo, y al nivel de funciones, siempre intenta ejecutarlas en paralelo. Pero adicionalmente a estas optimizaciones automáticas, se deben usar directivas para:

- Ejecutar múltiples tareas en paralelo, por ejemplo, múltiples ejecuciones de la misma función o múltiples iteraciones del mismo ciclo (*Pipelining*)
- Reestructurar la implementación física de arreglos, funciones, ciclos y puertos para mejorar la disponibilidad de los datos y ayudar que estos fluyan a través del diseño más rápidamente
- Dar información sobre la dependencia de los datos, para permitir mayores optimizaciones.

La última técnica de optimización es modificar el código en C para eliminar dependencias innecesarias en el código que puedan limitar el rendimiento del hardware.

En resumen, para realizar todas las anteriores optimizaciones es necesario conocer muy bien el algoritmo que se está implementando, y la arquitectura de las FPGA en la que se quiere trabajar para poder decidir cuáles directivas implementar, de modo que se aprovechen los recursos disponibles dentro de cada tarjeta. A continuación, se muestra un estudio de caso sobre la implementación de la operación fundamental de las redes neuronales: la multiplicación matricial.

2.2 Implementación en síntesis de alto nivel (HLS) de redes neuronales

Ahora se pasará al estudio de caso en que se va a optimizar la operación fundamental de las redes neuronales: la multiplicación matricial. Finalmente se llegará hasta la implementación de una red neuronal densa mediante el uso de HLS, realizando la programación y optimización por cuenta propia, es decir, en contraste con lo que se mostrará en el capítulo 3 donde se utiliza

una librería para la especificación de las redes neuronales. Todas las implementaciones de este capítulo se llevaron a cabo en la tarjeta PYNQ-Z1, que contiene la FPGA xc7z020clg400-1.

2.2.1 Estudio de caso: multiplicación matricial como operación fundamental de las NN

Para ejemplificar algunos aspectos de lo mencionado hasta el momento, y para profundizar en la operación fundamental de las redes neuronales (NN), se va a estudiar el caso de implementar una multiplicación matricial en HLS, optimizarla, exportar el bloque (llamado también *IP core*), pasar a *Vivado Design Suite*, incorporar el IP en un diagrama de bloques, exportar *bitstream*, ir a SDK y finalmente programar la aplicación anfitriona para implementar la multiplicación matricial sobre la lógica programable (PL) de la FPGA, haciendo un *benchmarking* del tiempo que se demora en realizar dicha operación, y luego compararlo con la misma operación ejecutada en Python, y corriendo en CPU.

En primer lugar, se muestra el código de una multiplicación matricial implementada de la manera más sencilla posible en lenguaje C:

```
void matrixmul(
      mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
      mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
      result_t res[MAT_A_ROWS][MAT_B_COLS])
{
  // Iterate over the rows of the A matrix
   Row: for(int i = 0; i < MAT_A_ROWS; i++) {
      // Iterate over the columns of the B matrix
      Col: for(int j = 0; j < MAT_B_COLS; j++) {
         res[i][j] = 0;
         // Do the inner product of a row of A and column of B
         Product: for(int k = 0; k < MAT_B_ROWS; k++) {
            res[i][j] += a[i][k] * b[k][j];
         }
      }
   }

}
```

Código 2: Multiplicación matricial básica en C

Se observa que se trata de tres ciclos *for* anidados: uno que itera sobre las filas de la primera matriz, uno que itera sobre las columnas de la segunda matriz, y uno que itera sobre cada una de las entradas. Si se realiza la síntesis de este código tal como está, HLS arroja un intervalo de 80 ciclos de reloj, además, por defecto los ciclos no se paralelizan de ninguna de las maneras explicadas en la sección anterior. Por tanto, el PL debe esperar que se termine cada operación para comenzar con la siguiente, con lo que el intervalo de iniciación es 79 ciclos. Para el caso de matrices 3x3, y para una precisión de enteros (int), el resultado de la síntesis se observa en la siguiente figura:

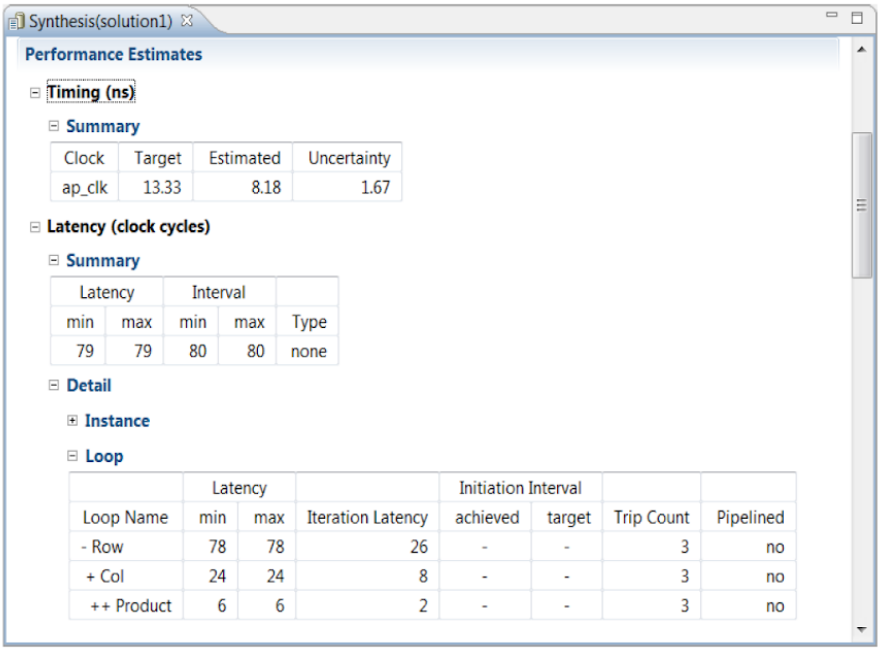

Figura 18: Síntesis multiplicación matricial básica

Los resultados son fácilmente extensibles para matrices de mayor tamaño, y para precisiones distintas, lo cual se ejemplifica más adelante.

Dos optimizaciones pueden realizarse inmediatamente sobre el código: se puede paralelizar el ciclo de las columnas, y también se puede cambiar la forma de los arreglos de entrada, de modo que los datos estén disponibles rápidamente para las operaciones. Esto último es necesario, pues por defecto los arreglos se mapean en memorias RAM de un solo puerto, lo que hace imposible acceder a todos los elementos de las matrices en un solo ciclo. Estas optimizaciones se logran insertando directivas a la interfaz de I/O y directiva al ciclo, del modo en que se observa en el siguiente código:

```
void matrixmul(
      mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
      mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
      result_t res[MAT_A_ROWS][MAT_B_COLS])
{
#pragma HLS ARRAY_RESHAPE variable=b complete dim=1
#pragma HLS ARRAY_RESHAPE variable=a complete dim=2
  // Iterate over the rows of the A matrix
   Row: for(int i = 0; i < MAT_A_ROWS; i++) {
      // Iterate over the columns of the B matrix
      Col: for(int j = 0; j < MAT_B_COLS; j++) {
#pragma HLS PIPELINE
         res[i][j] = 0;
         // Do the inner product of a row of A and col of B
         Product: for(int k = 0; k < MAT_B_ROWS; k++) {
            res[i][j] += a[i][k] * b[k][j];
         }
      }
   }

}
```

Código 3: Inserción de directivas de optimización

La matriz A se particiona en su segunda dimensión pues una multiplicación matricial tiene elementos de la forma $c_{ij} = \sum_{\mathrm{ij}} A_{ik}\, B_{kj}$, por tanto los elementos que deben estar disponibles para el acceso de la memoria es la segunda dimensión para la matriz A, y la primera dimensión para la matriz B. Adicionalmente, al realizar el pipeline del ciclo de columnas, se paraleliza

automáticamente el ciclo del producto interno. El resultado de la síntesis con estas directivas aplicadas se observa en la siguiente figura.

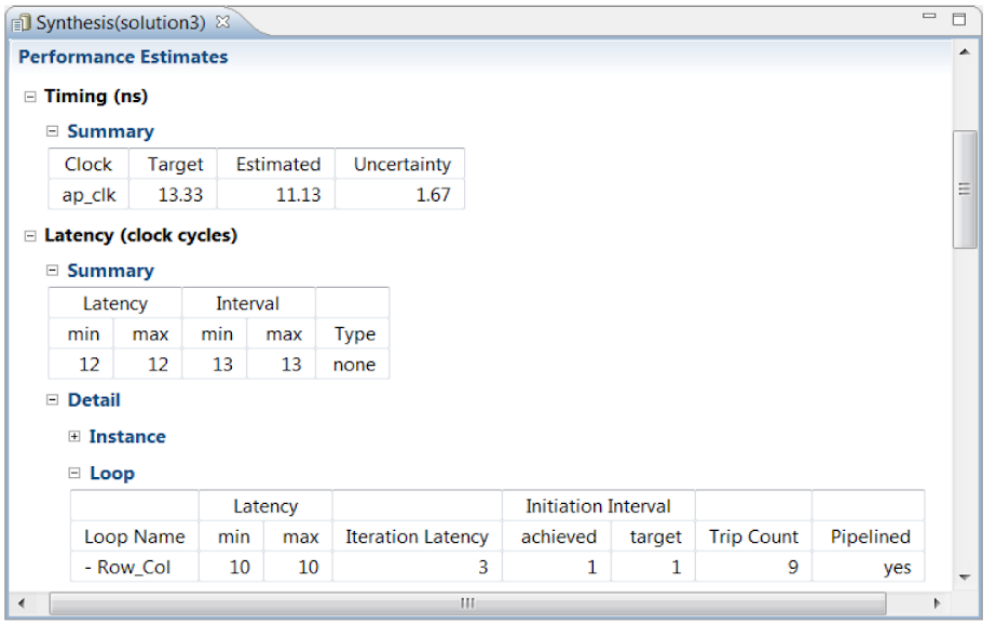

Figura 19: Resultado síntesis multiplicación matricial primera optimización

Se observa que ahora la operación toma solamente 13 ciclos de reloj, y se alcanzó un intervalo de iniciación de 1 ciclo de reloj, que quiere decir que el PL puede procesar una nueva fila de la matriz A para realizar la multiplicación correspondiente en cada ciclo de reloj, gracias al *Pipelining*. Por esto se observa que, teniendo 3 filas, y si cada una toma 10 ciclos de reloj, pero como se puede tomar una fila en cada ciclo, evidentemente tomará 12 ciclos la latencia, que es donde termina el último cálculo, y en el ciclo 13, la FPGA está lista para recibir nuevas instrucciones.

Sin embargo, existe una limitación crucial que hace que este diseño aún no sea óptimo: no permite el acceso en *streaming*. Para poder permitir el acceso en *streaming* es necesario analizar el comportamiento del código generado hasta el momento por HLS. En la siguiente figura se encuentra un esquema del algoritmo de la multiplicación matricial de dos matrices 3x3.

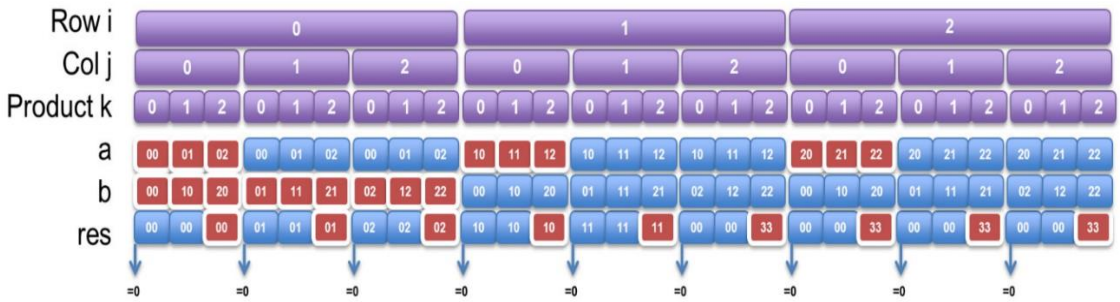

Figura 20: Arquitectura óptima multiplicación matricial [23]

En la figura anterior se observa cómo mientras se itera en las variables i, j, k, entre 0 y 2, la parte baja de la figura muestra las direcciones generadas para leer A, B, y escribir el resultado (res). Además, al principio de cada producto interno, el valor de res se inicializa en cero.

Para tener un hardware con acceso *streaming*, los puertos deben ser aquellos que están en rojo. Para los puertos de lectura, los puertos deben ser guardados en una memoria interna (cache interno) para asegurarse que el diseño no tenga que leer de nuevo el puerto. Para el

puerto de escritura *res*, los datos deben guardarse en una variable temporal y ser escrita en el puerto solamente en los ciclos en rojo. Para lograr esto es necesario modificar el código en C, el cual se muestra a continuación.

```
void matrixmul(
      mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
      mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
      result_t res[MAT_A_ROWS][MAT_B_COLS])
{
#pragma HLS INTERFACE s_axilite port=return bundle=myBus
#pragma HLS ARRAY_RESHAPE variable=b complete dim=1
#pragma HLS ARRAY_RESHAPE variable=a complete dim=2
#pragma HLS INTERFACE s_axilite port=a bundle=myBus
#pragma HLS INTERFACE s_axilite port=b bundle=myBus
#pragma HLS INTERFACE s_axilite port=res bundle=myBus
  mat_a_t a_row[MAT_A_ROWS];
  mat_b_t b_copy[MAT_B_ROWS][MAT_B_COLS];
  int tmp = 0;

  // Iterate over the rowa of the A matrix
  Row: for(int i = 0; i < MAT_A_ROWS; i++) {
    // Iterate over the columns of the B matrix
    Col: for(int j = 0; j < MAT_B_COLS; j++) {
#pragma HLS PIPELINE rewind
      // Do the inner product of a row of A and col of B
      tmp=0;
      // Cache each row (so it's only read once per function)
      if (j == 0)
        Cache_Row: for(int k = 0; k < MAT_A_ROWS; k++)
          a_row[k] = a[i][k];

       // Cache all cols (so they are only read once per function)
     if (i == 0)
            Cache_Col: for(int k = 0; k < MAT_B_ROWS; k++)
               b_copy[k][j] = b[k][j];

      Product: for(int k = 0; k < MAT_B_ROWS; k++) {
        tmp += a_row[k] * b_copy[k][j];
      }
      res[i][j] = tmp;
    }
  }
}
```

Código 4: Multiplicación matricial completamente optimizada

Ahora este código en C muestra el comportamiento deseado, además de la adición de las directivas de interfaz de puertos de entrada, salida y la interfaz de bloque como AXI Lite, unido a un solo Bus para simplificar el interfaceo con los demás bloques del diseño que serán necesario en la siguiente fase. Al realizar la síntesis de C, se observa el siguiente resultado:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	13.33	11.722	1.67

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
10	11	9	9	loop rewind(delay=0 initiation interval(s))

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Col	10	10	3	1	1	9	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	0	0	357
FIFO	-	-	-	-
Instance	6	-	270	295
Memory	-	-	-	-
Multiplexer	-	-	-	129
Register	-	-	174	-
Total	6	2	444	781

Figura 21: Síntesis multiplicación matricial completamente optimizada

El total de ciclos es ahora 9, y se conserva el intervalo de iniciación de 1, lo que indica que el diseño está completamente paralelizado, es decir, todas las operaciones de todas las filas de la matriz A se comienzan a ejecutar desde el primer ciclo, lo cual implica que se consumen más recursos del PL. A continuación, este diseño debe exportarse como RTL, lo cual hace que HLS cree un IP que puede ser usado como un bloque en Vivado Design Suite. Hasta aquí concluye el proceso HLS y se continúa con las demás herramientas para la programación de la FPGA.

El paso siguiente es la utilización de Vivado Design Suite, para lo cual se crea un nuevo proyecto, y se construye el siguiente diagrama de bloques para agregar el bloque recién creado en HLS:

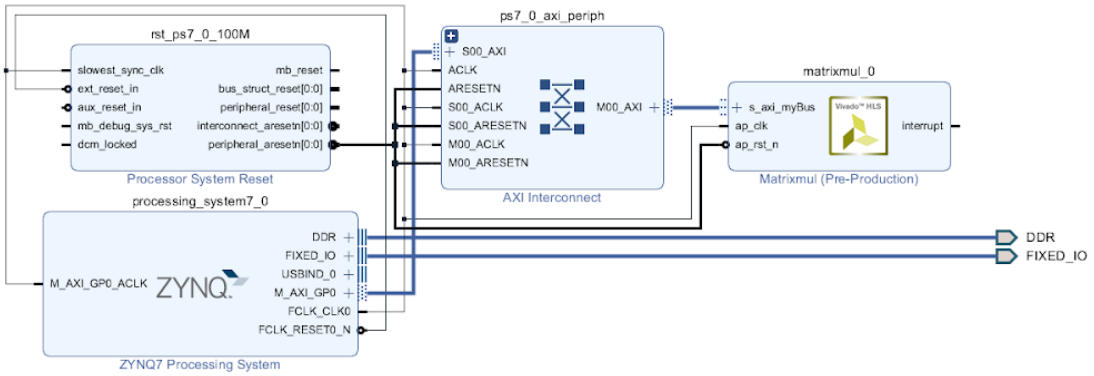

Figura 22: Diagrama de bloques para multiplicación matricial

Luego de la construcción de este diagrama se procede a la generación del *bitstream*, que es un proceso automático realizado por el programa. El *bitstream* es el archivo final que va a dar las instrucciones para que la FPGA se programe de modo tal que pueda realizar las funciones deseadas.

Una vez en SDK o Vitis se escribe el programa de la aplicación anfitriona, que es la que va a utilizar el bloque IP, para lo cual es necesario primero inicializar dicho bloque, configurarlo, luego escribir los datos, ordenarle que opere, esperar el resultado, y mostrar el resultado. Para el caso de la multiplicación matricial, el código mínimo necesario para llevar estos procesos a cabo se muestra a continuación:

```
int main()
{
    init_platform();

    // Initialize device
    XMatrixmul pmatrix;
    XMatrixmul_Config *pmatrix_config;
    pmatrix_config=XMatrixmul_LookupConfig(XPAR_XMATRIXMUL_0_DEVICE_ID);
    XMatrixmul_CfgInitialize(&pmatrix, pmatrix_config);

    //Pass values to multiplier
    XMatrixmul_Write_a_Bytes(&pmatrix, 0, a, 256);
    XMatrixmul_Write_b_Bytes(&pmatrix, 0, b, 256);

    // Start operating
    XMatrixmul_Start(&pmatrix);
    while (!XMatrixmul_IsDone(&pmatrix));

    // Obtain result
    XMatrixmul_Read_res_Bytes(&pmatrix, 0, (char*)res, 512);
    for (int i = 0; i < 256; i++)
        xil_printf("%d\r\n", res[i]);

    cleanup_platform();
    return 0;
}
```

Código 5: Mínima aplicación anfitriona multiplicación matricial

Antes del anterior bloque de código es necesario declarar e inicializar las matrices a multiplicar. Estas matrices se inicializan como arreglos unidimensionales de dimensión NxN, que, en el caso del presente bloque de código, es para dos matrices 16x16=256 elementos. Al ejecutar este código se debe obtener el resultado de la multiplicación matricial impreso en la consola de SDK.

Ahora bien, si se desea realizar una medición del tiempo que se demora realizar la multiplicación matricial, se debe agregar la librería *"xtime_l.h",* más las instrucciones para tomar el tiempo en realizar la multiplicación. Para esto, se debe insertar un bloque de código como el que se muestra a continuación

```
//Pass values to multiplier
XMatrixmul_Write_a_Bytes(&pmatrix, 0, a, 256);
XMatrixmul_Write_b_Bytes(&pmatrix, 0, b, 256);

//calculate time to operate
XTime_GetTime(&tStart);

// Start operating
XMatrixmul_Start(&pmatrix);
while (!XMatrixmul_IsDone(&pmatrix));

//taking time
XTime_GetTime(&tEnd);

elapsed_c = tEnd - tStart;
elapsed_t = (float) elapsed_c / (COUNTS_PER_SECOND/1000000);
xil_printf("Output took %llu clock cycles.\n", (char*) elapsed_c);
xil_printf("Output took %.2f us.\n",(float*) elapsed_t);

xil_printf("Elapsed t:\n",elapsed_t);
xil_printf("Counts per second: %11u \n",COUNTS_PER_SECOND);

// Retrieve result
XMatrixmul_Read_res_Bytes(&pmatrix, 0, (char*)res, 512);
for (int i = 0; i < 256; i++)
    xil_printf("%d\r\n", res[i]);
```

Código 6: Medición tiempo de ejecución de funciones en FPGA

En este bloque de código se observa las instrucciones para tomar, medir e imprimir el tiempo de la operación. De este modo queda clara la diferencia entre la aplicación anfitriona y la función acelerada por hardware. A la aplicación anfitriona pueden agregarse otras funcionalidades, utilizando el mismo bloque que ya fue acelerado por hardware.

Este proceso se realizó con la multiplicación matricial de dos matrices 16x16, con precisión de enteros (int), implementada en la FPGA xc7z020clg400-1 que es la que tiene la tarjeta PYNQ-Z1. Esta operación se realizó 500 veces para obtener un valor promedio y una desviación estándar de los tiempos. Igualmente se realizaron 500 veces esta misma operación en Python, corriendo en un computador de escritorio, con un procesador Intel Core i5 de 3.00GHz y 8Gb de memoria RAM DDR4. Los resultados comparativos se encuentran en la siguiente figura:

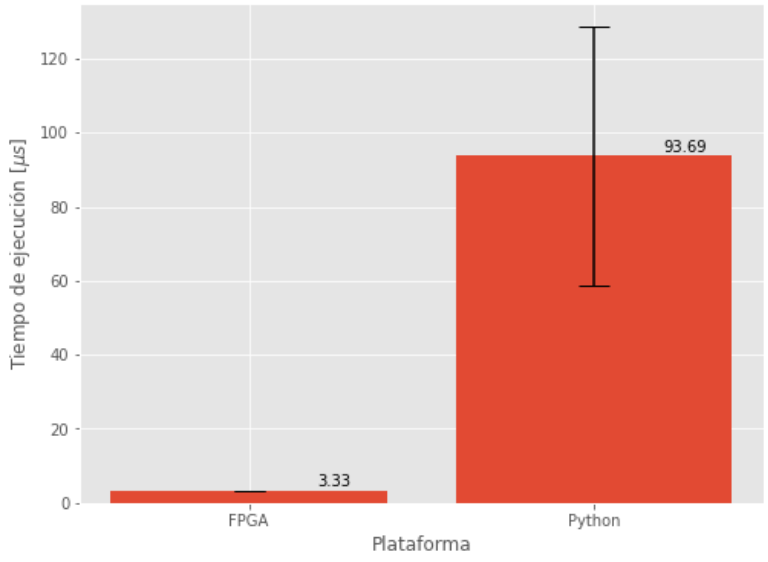

Figura 23: Resultado comparativo tiempos de ejecución multiplicación matricial

Estos resultados representan un factor de aceleración aproximado de 28.11 veces mayor velocidad en realizar dicha multiplicación en el PL de la FPGA. Adicionalmente, el tiempo de ejecución de la FPGA es más estable, pues se observa una desviación estándar muy pequeña, mientras que Python presentó una desviación estándar mucho más alta. En la siguiente sección se presenta la implementación de redes neuronales siguiendo procedimientos análogos a los presentados en esta sección.

2.2.2 Implementación de redes neuronales densas con batch 1, optimizadas para latencia

A continuación se procedió a implementar varias redes neuronales densas (también llamadas fully connected, FC) siguiendo un flujo similar al utilizado para implementar el ejemplo de la multiplicación matricial: se comenzó por escribir estas redes como códigos en C++, con los cuales se crearon implementaciones RTL de estas redes (IPs) en Vivado HLS, posteriormente se integraron estos IP dentro de diagramas de bloques en Vivado Design Suite, el cual generó los bitstreams para programar la FPGA, y finalmente se utilizó SDK para desarrollar la aplicación anfitriona que lleva a cabo la implementación en la tarjeta como tal y medir los tiempos de ejecución.

La arquitectura y especificaciones de las redes implementadas son las siguientes:

Arquitectura	**Capas ocultas**	**Precisión (bits)**	**Pesos de la red**
32x32x32x10	2	32	Ajustables
32x32x32x10	2	8	Fijos
64x64x64x10	2	32	Ajustables
64x64x64x10	2	8	Fijos
128x128x128x10	2	32	Ajustables
128x128x128x10	2	8	Fijos
16x64x32x32x5	3	8	Fijos

Tabla 5: Redes neuronales implementadas mediante HLS

Vale la pena decir que en Deep Learning usualmente hay arquitecturas (número de capas y neuronas en cada capa) estándar, sobre todo en redes convolucionales, sin embargo, para las redes densas no es tan común que haya arquitecturas estándar. Por tanto, estas redes salvo la última corresponden a redes con arquitecturas relativamente arbitrarias, pero que sirvieron para poner a prueba la correcta y adecuada implementación en FPGA. La última arquitectura corresponde a una utilizada en física de partículas para la clasificación de jets en el LHC [25]. En primer lugar, se van a exponer los resultados de las primeras arquitecturas y luego se presentarán los de la última arquitectura.

Adicionalmente, hay varias diferencias en la implementación de las primeras redes. en su versión 32 bits o versión 8 bits. La primera diferencia es que las versiones 32 bits fueron implementadas como su nombre lo indica, con números representados en 32 bits, pero adicionalmente, éstas tuvieron como interfaces del bloque IP las matrices de pesos para ser ingresadas como argumentos del bloque al momento de llamar la función en SDK, y estas matrices fueron almacenadas en bloques RAM de dos puertos (Dual port BRAM). Estas matrices se comportaban como variables, y por tanto, esta implementación permitía ingresar diferentes matrices de pesos en el momento de ejecución.

En contraste, la implementación de 8 bits tuvo varias diferencias estructurales con lo anterior. En primero lugar se disminuyó la precisión de representación de los números a 8 bits, lo que se conoce como cuantización de pesos, con lo cual se busca acelerar las operaciones a costo de una pérdida en precisión de la red. En general si la pérdida de precisión es baja, es mejor cuantizar los pesos para ganar velocidad y utilizar menos recursos de la FPGA. Otra diferencia importante es que en esta arquitectura, las matrices de pesos ya estaban grabadas en el IP desde el momento de utilización de HLS, y además se indicó que estas matrices eran fijas (no

eran variables), por tanto ya no eran argumentos de la función red neuronal, sino que una vez grabadas, era inmodificables, y el único argumento de la ejecución de dicho bloque ya era el vector de entrada. Esta arquitectura no permite la modificación de pesos una vez creado el IP, teniendo como consecuencia un aumento considerable en velocidad, lo cual se observa en los resultados.

En todas estas redes la implementación tuvo como objetivo el procesamiento de una sola entrada, es decir, una red con procesamiento de *batch* 1, donde cada vector de entrada se procesa en secuencia, es decir, no se procesan varias entradas al mismo tiempo (lo que sería *batch* mayor que 1). Esto teniendo en mente el procesamiento de información de sensores cuyos datos son entregados de manera continua en el tiempo. Por ejemplo, una cámara entrega un cuadro tras otro de manera secuencial, y para hacer procesamiento cuadro a cuadro y tener una salida continua, se debe procesar cada cuadro tras otro. Esto en contraste con sistemas que tienen varios datos disponibles para procesamiento al mismo tiempo, como lo es una base de datos, donde pueden procesarse simultáneamente varias (o todos) los datos de manera simultánea, pues no son datos que van llegando de manera secuencial.

Por tanto, las operaciones matemáticas de las redes implementadas correspondieron a multiplicaciones matriz-vector (lo que corresponde a la operación de cada capa de la red con batch 1) seguidas de activaciones ReLu. La función de activación ReLu es la más utilizada en Deep Learning y consiste en la siguiente función:

$$ReLu(x) = \{0, si\ x \leq 0\ x, si\ x > 0 \quad (3)$$

Para cuidar que las redes estuvieran realizando operaciones correctas, se definieron matrices de pesos, vectores de entrada, y resultados de referencia, los cuales se utilizaron como *test bench* durante el proceso de síntesis RTL del código en C++, y también durante la implementación en la FPGA como tal. Para comparar el rendimiento de la FPGA en cada una de las redes, se comparó con implementaciones vectorizadas en Python, e implementaciones en C++ ejecutadas en un PC de escritorio con un procesador Intel Core i5 de 3.00GHz. Los resultados se muestran a continuación

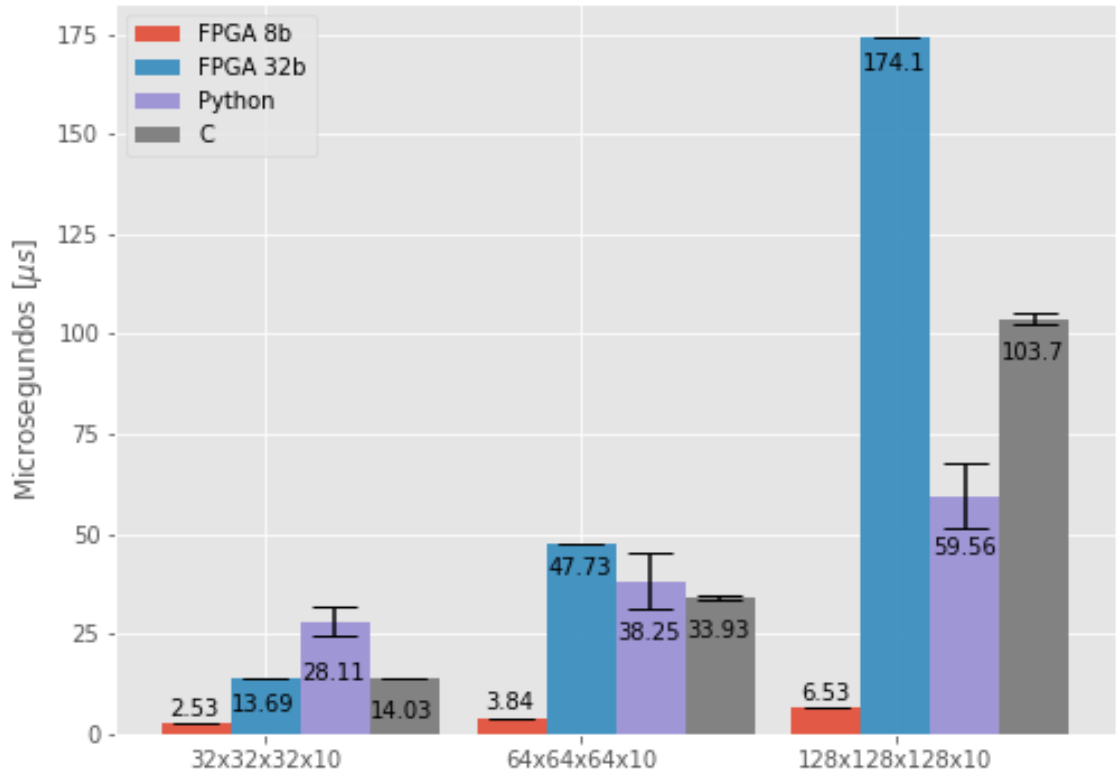

Figura 24: Resultados tiempos de ejecución redes capítulo 2

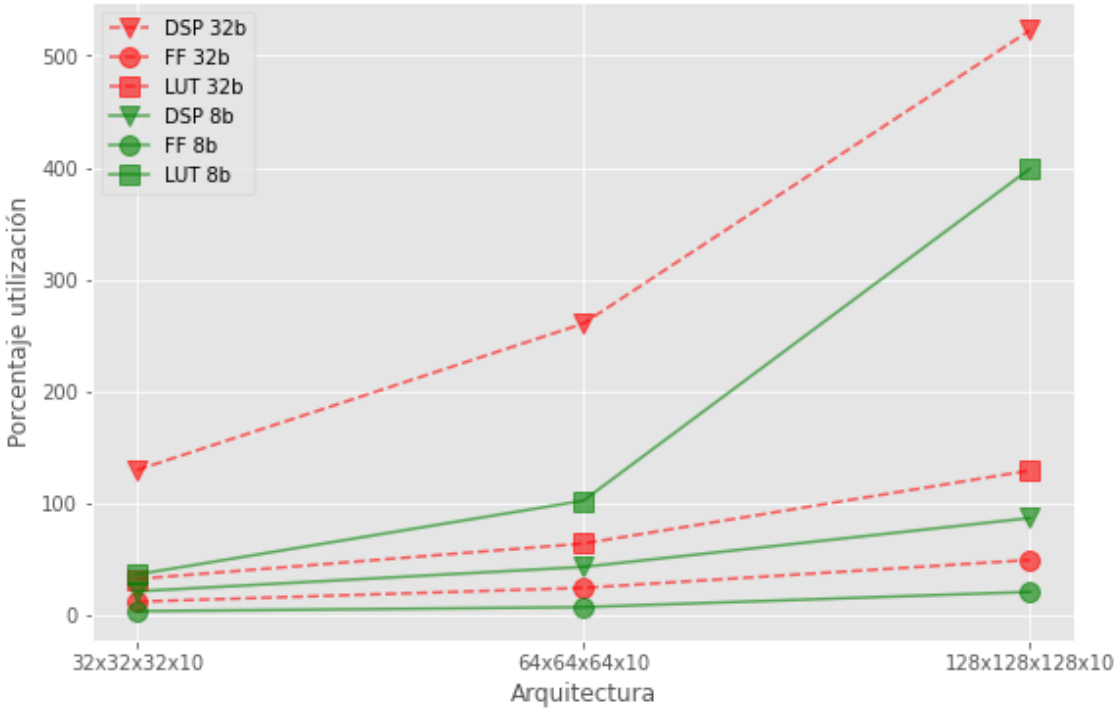

Figura 25: Resultados utilización de recursos redes capítulo 2

En la Figura 24 se observa la comparación de tiempos de ejecución según arquitecturas y entornos de ejecución. En primer lugar, resalta la superioridad de la FPGA utilizando el diseño con precisión de 8 bits descrito más atrás, pero también destaca la debilidad de utilizar el diseño con precisión de 32 bits. El diseño de 8 bits en la arquitectura 32x32x32x10 es 5.41x más rápido

que su competidor más cercano en esta arquitectura que es la FPGA con diseño de 32 bits. En la arquitectura 64x64x64x10 el diseño de 8 bits es 8.83x más rápido que su competidor más cercano que es C; y en la arquitectura 128x128x128x10 es 9.12x más rápido que Python, su competidor más cercano en dicha arquitectura.

Adicionalmente, considerar la Figura 25 (Utilización de recursos de la FPGA) es necesario para comprender el comportamiento de la FPGA. En primer lugar, es importante recordar que una utilización por encima del 100% significa que los recursos se reutilizan en subsiguientes periodos de reloj. Las líneas punteadas rojas corresponden a los recursos utilizados por los diseños de 32 bits, mientras que las verdes continuas corresponden a los diseños de 8 bits. De este modo, se observa que los diseños de 32 bits hacen una alta utilización de DSP (por encima de 100% en todas las arquitecturas), que son las estructuras electrónicas encargadas de realizar las operaciones de multiplicación-adición usando representaciones de números en bits. En contraste, los diseños de 8 bits hacen una alta utilización de LUT (Lookup tables) que son estructuras que guardan arreglos indexados con las respuestas de las operaciones ya computadas, y luego, solo es necesario buscar el resultado, mas no calcularlo, por esto pueden requerir mucho menos tiempo para realizar las operaciones. Pero las LUT tienen una capacidad de almacenamiento bajo, por tanto, su capacidad es limitada. Por esto, usar una precisión menor para los pesos de las neuronas, al tiempo que se tratan como valores fijos no variables, permite la programación de estos pesos como LUT, lo que disminuye considerablemente el tiempo de cómputo.

Ahora, se pasará a considerar los resultados de la última arquitectura. En primer lugar, esta arquitectura se implementó únicamente con el diseño de 8 bits tal como se expuso para las arquitecturas anteriores, y se comparó el rendimiento con el de sus correspondientes implementaciones en Python usando funciones vectorizadas de numpy, e implementación en C. Estas últimas implementaciones también se realizaron en un PC de escritorio con procesador Intel Core i5 de 3.00GHz. Los resultados se aprecian a continuación

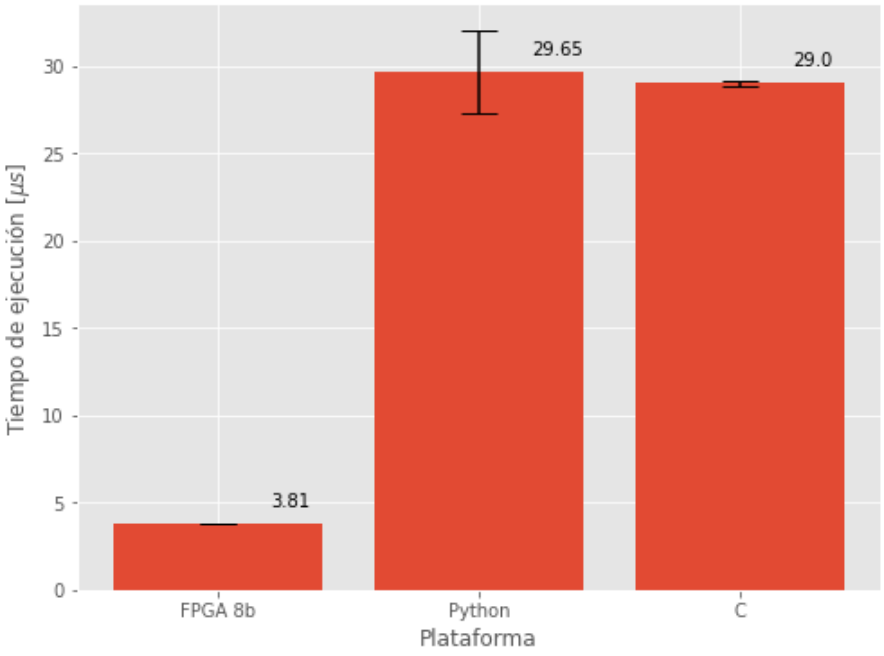

Figura 26: Tiempos de ejecución arquitectura 16x64x32x32xx5

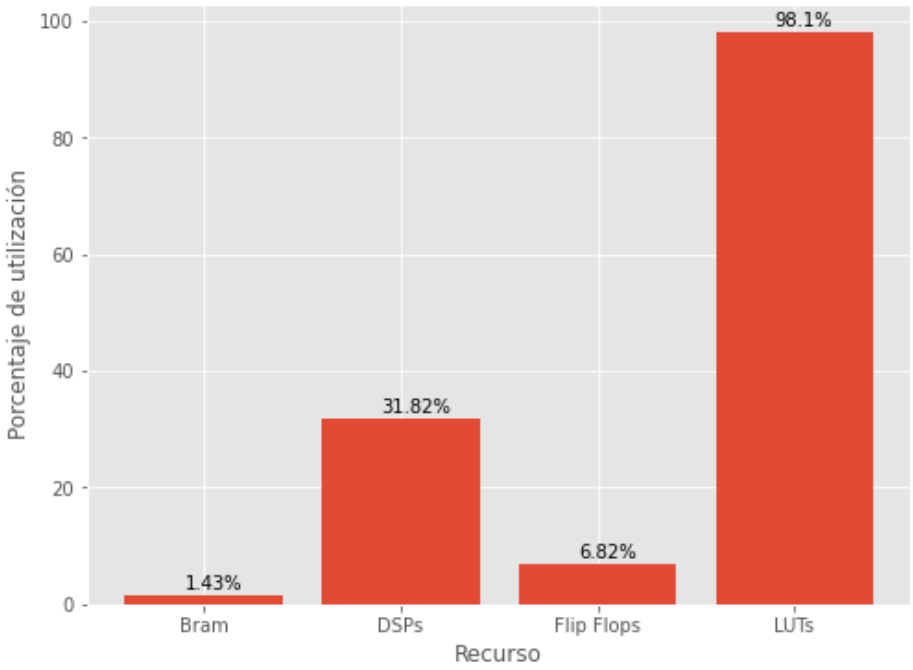

Figura 27: Recursos utilizados red 16x64x32x32x5

Se puede observar que el tiempo de ejecución de la FPGA en esta arquitectura es mucho menor que el de Python y C, llegando a ser de 3.81±0.0038 microsegundos, lo que es 7.61x más rápido que la implementación en C, y sin pérdida de exactitud con dicha cuantización. Respecto a los recursos utilizados por esta implementación se puede observar que ésta aprovecha casi la totalidad de LUT en la FPGA y aproximadamente una tercera parte de los DSP, lo cual es similar a la implementación de las arquitecturas anteriores, lo cual es un factor en la explicación de la velocidad superior en la implementación en FPGA.

A modo de conclusión del presente capítulo, puede decirse que las implementaciones fueron exitosas puesto que no sólo se logró llevar a cabo el complejo proceso de programar la FPGA de manera exitosa, sino que se logró llevar a cabo optimizaciones cruciales, que hicieron que el rendimiento de la FPGA fuera muy superior no solo a su propio rendimiento con una configuración diferente, sino que se logró sobrepasar el rendimiento de entornos comunes de utilización de redes neuronales: los computadores de escritorio utilizando Python o C.

Las implementaciones descritas en este capítulo pueden servir para un funcionamiento en tiempo real de sistemas embebidos de procesamiento de datos, donde se conecta un sensor a alguna entrada de la FPGA, y esta realiza un procesamiento muy rápido, y puede devolver el resultado para ser post-procesado después. Las implementaciones son igualmente flexibles, en el sentido de que el mismo procedimiento puede usarse para realizar otros algoritmos necesarios en diferentes contextos, ya sea en física experimental, u otras aplicaciones académicas o industriales.

3 IMPLEMENTACIÓN DE REDES NEURONALES USANDO LA LIBRERÍA HLS4ML

En esta sección del presente trabajo se analiza la implementación de redes neuronales de un tamaño mayor que las implementadas en la sección 2.2, y se hace un cambio en el método de implementación, pues en este capítulo se utiliza la librería hls4ml [14]. En este capítulo se expone el origen de la librería hls4ml, se discute la utilización de redes neuronales implementadas en FPGAs en un contexto de física experimental, el gran colisionador de hadrones (LHC: *Large Hadron Collider*), y finalmente se exponen los resultados de las implementaciones de dos arquitecturas de redes neuronales mediante el uso de dicha librería

3.1 Origen librería hls4ml: Experimentos en el gran colisionador de hadrones (LHC)

En primer lugar es importante conocer de dónde viene la librería hls4ml y para qué se ha usado. Un asunto fundamental a tener en cuenta es que la física de partículas ha desarrollado proyectos experimentales de la envergadura del Gran Colisionador de Hadrones (LHC), en donde los detectores vienen siendo cada vez más grandes, más granulares, y pueden procesar datos a velocidades más altas. Esto tiene como consecuencia el incremento en el volumen de datos que deben ser analizados en tiempo real para reconstruir y filtrar eventos de interés. Y aunque el Machine Learning (ML) se han utilizado extensivamente en las etapas finales del procesamiento de datos en física de partículas, su utilización en etapas tempranas de adquisición de datos en tiempo real, que está basada en FPGAs, había sido limitada debido a la complejidad de las mismas técnicas de ML y a la complejidad del balanceo del uso de los recursos de las FPGAs. Los autores de la librería hls4ml buscaron explorar la implementación de diferentes arquitecturas de redes neuronales en FPGAs, teniendo en cuenta la utilización de recursos y latencia obtenida, para demostrar la posibilidad de alcanzar latencias ultra bajas (por debajo de microsegundos) en aplicaciones de ML en FPGAs [14].

Debido a que en el LHC grupos de protones colisionan a una tasa de 40MHz, la producción de datos en los experimentos ATLAS y CMS son del orden de cientos de Tera bytes por segundo (Tb/s), lo que conlleva desafíos importantes para el procesamiento de los datos tanto en tiempo real como el procesamiento offline. La tarea del procesamiento en tiempo real es filtrar eventos para reducir la tasa de producción de datos a niveles manejables para el procesamiento offline, lo cual se conoce como *triggering* [26].

Los métodos de *machine learning* (ML), y en particular el *Deep learning* (DL) tienen múltiples aplicaciones en el procesamiento de eventos en el LHC [27], [28], desde la calibración y regresión de *clusters* de baja energía, hasta la clasificación de objetos de física de alto nivel, como lo es la etiquetación de *jets* con la información de la subestructura, y análisis físicos. (La explicación de qué es un *jet* y la subestructura se encuentran en la sección 3.3). Así, contar con algoritmos más sofisticados de ML en el *trigger* de los experimentos del LHC puede permitir preservar nuevas detecciones físicas como las relacionadas con el bosón de Higgs, materia oscura, y otros sectores ocultos, que de otra manera quedarían perdidos [29].

Es por esto que se estudió la posibilidad de implementar inferencia de redes neuronales en FPGAs al tiempo que se monitoreó el uso de recursos y la latencia para diferentes arquitecturas e hiperparámetros. Un resultado de este estudio es la librería hls4ml, que traduce modelos de ML provenientes de paquetes populares como Keras o PyTorch [30] a códigos RTL para FPGAs utilizando herramientas de síntesis de alto nivel (HLS: *High-Level Synthesis*), en particular Vivado HLS. Las técnicas generales pueden aplicarse a diferentes problemas y a diferentes FPGAs. Una ventaja fundamental de la herramienta hls4ml es que permite que los físicos (y en general otras personas) puedan prototipar rápidamente algoritmos de ML para estudiar su factibilidad en firmware y rendimiento sin experiencia en código VHDL o Verilog, lo cual reduce drásticamente el tiempo del ciclo de desarrollo [14].

3.2 Construcción de una red neuronal con hls4ml

La tarea de traducir automáticamente una red neuronal entrenada, especificada por su arquitectura y sus pesos, en código para ser sintetizados por HLS es la tarea de la librería hls4ml. El esquema presentado por los autores para realizar este proceso se encuentra en la siguiente figura:

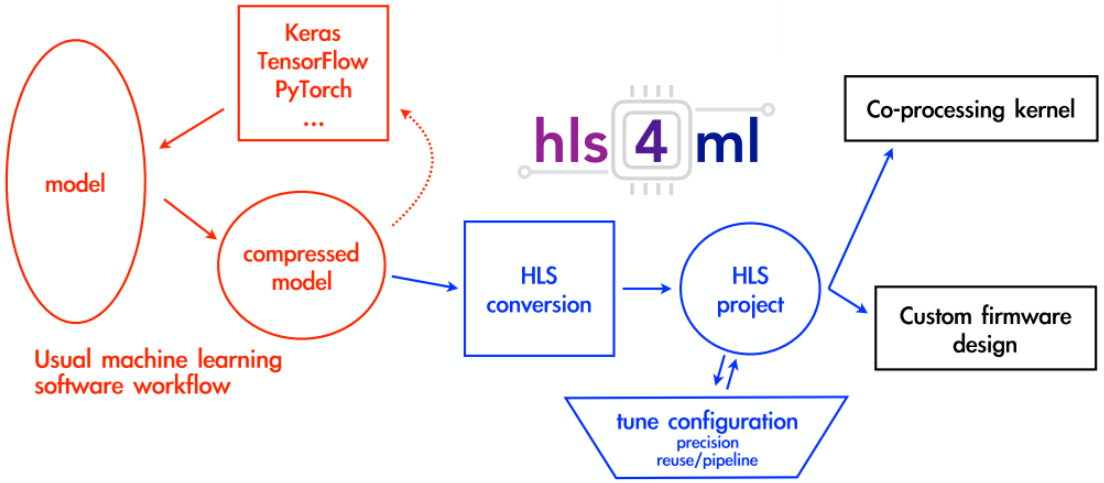

Figura 28: Flujo de trabajo hls4ml [14]

En rojo se encuentra el flujo de trabajo usual para diseñar una red neuronal con algún propósito particular. Es un proceso que se lleva a cabo usualmente en Python, con librerías como Keras, Tensorflow o PyTorch, que incluye el diseño de la arquitectura, el entrenamiento de la red, y posiblemente la compresión u optimización de la red, antes de decidirse por un diseño final. La sección que está en azul es la tarea de hls4ml, que es traducir el modelo ya entrenado en un proyecto HLS que puede ser sintetizado e implementado para correr en una FPGA[14].

La implementación del diseño de un algoritmo en una FPGA es único, debido a las propiedades intrínsecas de la FPGA y del algoritmo, pues la FPGA puede implementar operaciones en paralelo, lo que permite alcanzar trillones de operaciones por segundo con un consumo de energía relativamente bajo en comparación con las CPU y las GPU. Sin embargo, estas operaciones consumen recursos del chip y la tarjeta y no pueden ser reprogramadas dinámicamente mientras estén corriendo. El objetivo es entonces crear una implementación óptima en la que se equilibre el uso de recursos para alcanzar el objetivo de latencia o rendimiento del algoritmo. Las métricas clave son, como se mencionó en el primer capítulo, la latencia, el intervalo de iniciación y el uso de recursos. La librería hls4ml tiene algunos parámetros ajustables que le permiten al usuario explorar el espacio de las anteriores métricas.

Vale recordar, que tal como se vió en los capítulos anteriores, realizar la síntesis de alto nivel, es sólo el primer paso en la programación de FPGA, y por tanto, al usar esta librería, sólo se llega hasta el punto de generar un IP, por lo que aún es necesario realizar los siguientes pasos de creación de diagrama de bloques en Vivado Design Suite, exportación de bitstream y creación de aplicación anfitriona en SDK o Vitis.

3.3 Estudio de caso: uso de redes neuronales para la clasificación de subestructura de jets

El primer estudio de caso corresponde a la clasificación de subestructuras de *jets*, que empleada en el *trigger*, permite búsquedas de partículas de materia oscura e importantes mediciones del espectro de momentum transverso del bosón de Higgs. Para utilizar estos algoritmos en los experimentos ATLAS y CMS del LHC se requieren latencias del orden de microsegundos, paralelizados para admitir la tasa de transmisión de datos de 40MHz. Para esta tarea, las CPUs o las GPUs no son una opción debido a la severa restricción de tiempo. Por esto, las FPGA emergen como la alternativa viable para implementar tales algoritmos a tales velocidades. El paquete hls4ml se diseñó como una herramienta general, útil para diferentes aplicaciones en física de partículas y otras áreas, desde el *trigger* y las tareas de recolección de datos, hasta tareas de *trigger* de latencia más larga (milisegundos) y hardware co procesador CPU-FPGA. Las redes neuronales que se aplican en este caso son relativamente pequeñas debido a que los límites de tiempo son condiciones muy estrictas [14]. Vale la pena decir que en esta sección se trata de la implementación en FPGA de una red neuronal ya entrenada y publicada por los autores del paquete hls4ml [14].

En otros trabajos ya se habían realizado estudios sobre la clasificación basada en subestructura de jets, donde se ha estudiado ya el uso de redes convolucionales (CNN: *Convolutional Neural Networks*), redes recurrentes (RNN: *Recurrent Neural Network*), así como otras redes inspiradas en la física [31]. Los *jets* han sido representados como imágenes en escala de gris, imágenes RGB, secuencias de partículas, o un conjunto de propiedades físicas, como se hace en el presente trabajo.

Los *jets* son “chorros” colimados de partículas que resultan del decaimiento y hadronización de quarks *q* y gluones *g*. En el LHC, debido a la alta energía de colisión, un tipo de jets particularmente interesantes emerge de la superposición de chorros iniciados por quarks que se producen en los decaimientos de partículas pesadas del modelo estándar, por ejemplo los bosones W y Z decaen en dos quarks (qq*) en el 67% al 70% de los casos, y el bosón de Higgs se predice que va a decaer en dos quarks b (bb*) aproximadamente el 58% de los casos. El quark top decae en dos quarks livianos y un quark b (qq*b). La tarea de la subestructura es distinguir los diferentes perfiles de radiación de estos jets entre la demás radiación (*background*), que consiste principalmente de quarks (u, d, c, s, b), y de jets iniciados por gluones. Las herramientas de la subestructura han sido usadas para clasificar grupos de jets interesantes de entre el *background*, y que tienen tasas de producción cientos de veces más altas que la señal [32].

La subestructura de jets en el LHC ha sido un campo activo de investigación para las técnicas de ML debido a que los *jets* contienen el orden de 100 partículas cuyas propiedades y correlaciones pueden ser aprovechadas para identificar señales físicas. La alta dimensionalidad y naturaleza altamente correlacionada del espacio de fase hace que esta sea una tarea interesante para las técnicas de ML. Esta es una tarea que se ha explorado de manera teórica y experimental [14].

Dos situaciones en las que las técnicas de subestructura pueden jugar un papel importante son las resonancias hadrónicas ocultas de masa baja, y el Higgs acelerado (*boosted*) producido en la fusión de gluones. Ambos procesos están ocultos por la cantidad de ruido de fondo (*background*). Al introducir las técnicas de subestructura en el *trigger* de hardware, se puede reducir el ruido y preservar muchos más de estos tipos de señales. En este caso, la tarea consiste en clasificar un jet como: quark (q) gluon (g), bosón W (W), bosón Z (Z), o quark top (t).

Adicionalmente, existe una comunidad académica que ha estudiado la subestructura de los jets y han desarrollado varios observables para identificar el origen de un jet con base en la

estructura de su patrón de radiación. En la siguiente tabla se listan los observables que ingresan a la red neuronal para la clasificación de jets:

Observables
m_{mMDT}
$N_2^{\beta=1,2}$
$M_2^{\beta=1,2}$
$C_1^{\beta=0,1,2}$
$D_2^{\beta=1,2}$
$D_2^{(\alpha,\beta)=(1,1),(1,2)}$
$\sum zlog\ z$
$Multiplicity$

Tabla 6: Observables que ingresan a la red jet tagger

Los anteriores observables corresponden a: m_{mMDT} es la pérdida de masa modificada (modified mass drop) [33], $N_2^{\beta=1,2}$ se conoce como N-subjettiness [34], los observables C, y D son funciones de correlación de energía [35], respecto a $\sum zlog\ z$, la energía que llevan las partículas hijas en un jet, con $z_i = E_i/E_{parent}$ es una medida de la suavidad del jet, y puede usarse para separar jets de quarks o de gluones [36], por último la multiplicidad hadrónica es el número de constituyentes del jet [37]. Una descripción más exhaustiva de dichos observables se encuentra en las respectivas referencias. La selección de dichas variables se realizó de manera teórica con el fin de poner a prueba la implementación de las redes neuronales para la clasificación de jets.

Con los anteriores inputs, los autores entrenaron una red neuronal para la clasificación de q, g, W, Z y t. Los datos fueron separados en 60% para entrenamiento, 20% para validación y 20% para prueba. Los observables de entrada fueron normalizados al restar el promedio y escalar para tener varianza unitaria. La arquitectura de la red se ilustra en la figura siguiente y es una red densa con tres capas ocultas. La función de activación para las capas ocultas es ReLU, mientras que se usó softmax como activación de la última capa, con el fin de obtener probabilidades de cada clase. Esta red es llamada *jet tagger* en este trabajo.

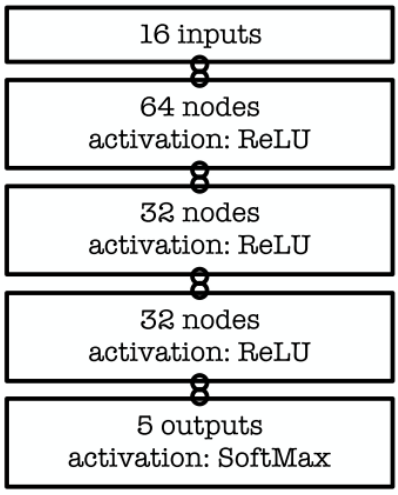

Figura 29: Arquitectura red densa para clasificación de jets [14]

Luego, se utilizó como función de pérdida la entropía categórica cruzada, la cual se puede minimizar con o sin regularización L_1 de los pesos, y se utilizó el algoritmo de optimización Adam [38]. El rendimiento de la red reportado por los autores se encuentra en la siguiente figura:

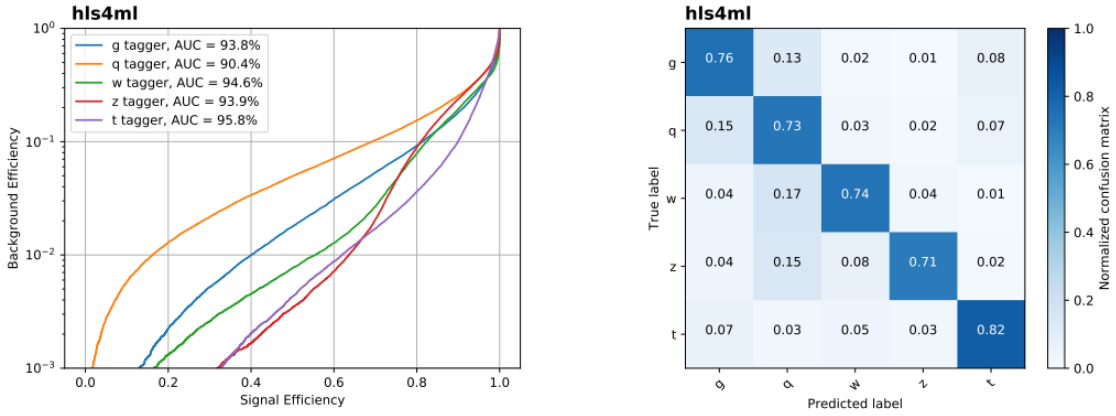

Figura 30: Rendimiento de la red clasificadora de jets [14]

Hasta este punto, la librería hls4ml cuenta entre sus proyectos de ejemplo los pesos y estructura de esta red ya entrenada, tras lo cual, se procedió a implementar dicha red en la FPGA xc7z020clg400-1 que es la que corresponde a la tarjeta PYNQ-Z1. Para utilizar esta librería el primer paso es la configuración de la implementación, la cual se realiza mediante un archivo yml, cuyo código se encuentra a continuación:

```
KerasJson: keras/KERAS_3layer.json
KerasH5:   keras/KERAS_3layer_weights.h5
#InputData: keras/KERAS_3layer_input_features.dat
#OutputPredictions: keras/KERAS_3layer_predictions.dat
OutputDir: NN3L_reuse20_per20_fInOut
ProjectName: NN3L
XilinxPart: xc7z020clg400-1
ClockPeriod: 20
Backend: Vivado

IOType: io_parallel # options: io_serial/io_parallel
HLSConfig:
  Model:
    Precision: ap_fixed<8,3>
    ReuseFactor: 20
  LayerName:
    softmax:
      Precision:
        result: float
```

Código 7: Configuración Jet tagger

Como se observa, en este código se especifica los archivos json y h5 donde se encuentra la arquitectura y pesos de la red, estos archivos son exportados por Keras en Python. Igualmente se especifica la FPGA de implementación, el tipo de paralelización, el factor de reúso, que controla cuántas veces se van a reutilizar los recursos de la FPGA, finalmente también se puede controlar por cada capa, la precisión de representación de números, en este caso, es de puntos fijos de 8 bits para todas las capas, excepto para la capa de salida, que tiene representación en punto flotante.

Luego de la conversión de dicha red a código en C++ sintetizable por Vivado HLS, es necesario establecer las interfaces de entradas, salidas, e interfaces de nivel de bloque, como se explicó en la sección 2.1. Para esta red, como los vectores de entrada y salida eran de tamaño

relativamente pequeño, se utilizaron interfaces Axi Lite, las cuales se especifican por medio de pragmas de la siguiente manera en el código generado por hls4ml:

```
void NN3L(
    float input_v[N_INPUT_1_1],
    float output_v[N_LAYER_8],
    unsigned short &const_size_in_1,
    unsigned short &const_size_out_1
) {
#pragma HLS INTERFACE s_axilite port=const_size_out_1 bundle=CTRL_BUS
#pragma HLS INTERFACE s_axilite port=const_size_in_1 bundle=CTRL_BUS
#pragma HLS INTERFACE s_axilite port=return bundle=CTRL_BUS
#pragma HLS INTERFACE s_axilite port=output_v bundle=CTRL_BUS
#pragma HLS INTERFACE s_axilite port=input_v bundle=CTRL_BUS
#pragma HLS ARRAY_RESHAPE variable=input_V complete dim=0
#pragma HLS PIPELINE
```

Código 8: Especificación de interfaces jet tagger

A continuación se procedió a probar y sintetizar el código, con lo que se obtiene el IP o bloque que es utilizado luego en Vivado Design Suite para formar un diseño de bloques que pueda usarse para generar el *bitstream*, que es el archivo que programa la FPGA. El diseño de bloques implementado es el siguiente:

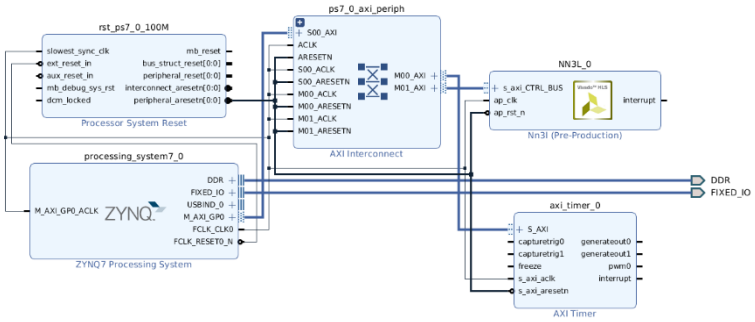

Figura 31: Diagrama de bloques jet tagger

Finalmente, se desarrolló la aplicación anfitriona en Vitis (SDK), donde se implementó como tal la red neuronal en la FPGA.

3.3.1 Resultados implementación red jet tagger

A continuación, se muestran los resultados acerca del tiempo de ejecución en la FPGA, comparado con el tiempo de ejecución en Python, de la red neuronal *jet tagger*, en implementación vectorizada[39], y corriendo en un computador de escritorio con un procesador Intel Core i5 de 3.00GHz y 8Gb de memoria RAM DDR4.

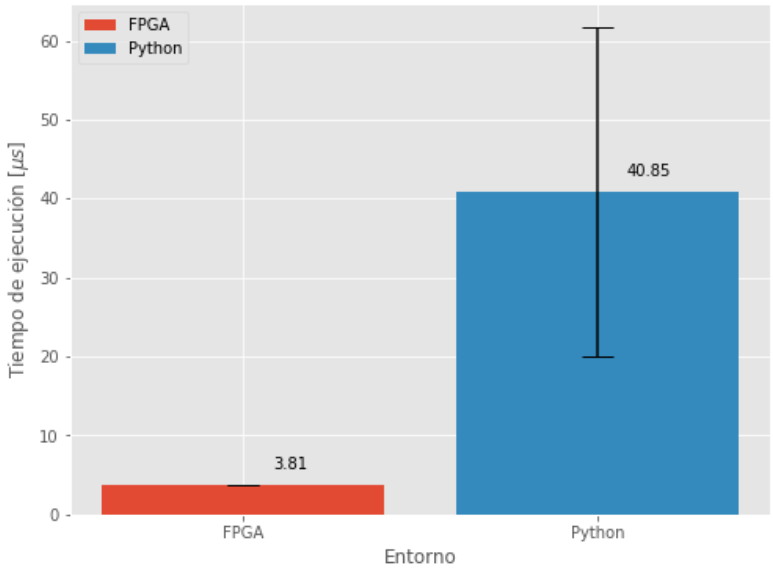

Figura 32: Tiempos de ejecución jet tagger

En esta gráfica se observa que la FPGA fue 10.72 veces más rápida que Python. Adicionalmente, el tiempo de ejecución de la FPGA es mucho más confiable, pues es menos variable en comparación con la ejecución con Python, la cual tuvo una desviación estándar mucho mayor, lo cual se ilustra con la barra vertical de error.

Finalmente, la siguiente tabla muestra los recursos de la FPGA utilizados por el IP que implementa la red neuronal clasificadora de jets:

	BRAM_18K	***DSP48E***	***FF***	***LUT***
Total	15	0	21224	91067
Disponible	280	220	106400	53200
Utilización (%)	5	0	19	171

Tabla 7: Utilización de recursos jet tagger

En esta tabla se observa la baja utilización de recursos por parte de esta red, en particular los recursos de BRAM y DSP tienen una utilización cercana a 0%, mientras que la utilización de LUTs es casi de 200%. Esto tiene como consecuencia una velocidad mucho mayor en el cómputo de la red, puesto que, como se explicó en la sección 1.5, las LUT en realidad no calculan operaciones, sino que guardan los posibles resultados y luego solo es necesario seleccionarlos, con lo cual hay una necesidad nula en este caso de utilizar los DSP que son los componentes encargados de realizar operaciones como multiplicación-acumulación.

Una limitación de la presente implementación es que debido a la FPGA utilizada, no es posible alcanzar con ésta la velocidad requerida en el LHC (De aproximadamente 40MHz), pero con FPGAs más grandes (con mayor cantidad de componentes) es posible alcanzar dichas velocidades. El método de programación es exactamente igual, y por tanto los procedimientos son completamente transferibles.

3.4 Implementación de una red neuronal densa en FPGA para la clasificación de dígitos escritos a mano MNIST

La siguiente parte del presente trabajo consistió en la implementación de una de las aplicaciones más tradicionales de algoritmos de ML: la clasificación de dígitos del MNIST. Esta es una tarea donde se busca que un algoritmo clasifique correctamente imágenes de dígitos escritos a mano, como los que se muestran en la siguiente figura:

Figura 33: Dígitos MNIST [40]

Esta tarea, que es relativamente sencilla, tiene la suficiente complejidad para considerarse dentro del campo de la visión artificial, y es tal vez el problema más clásico de *Deep Learning* utilizado para comprobar que el algoritmo que se está probando funciona correctamente [41]. Las imágenes MNIST tienen una dimensión de 28x28 píxeles en escala de grises, y por tanto el vector de entrada contiene 784 elementos.

Una red densa “suficientemente” grande debería ser capaz de alcanzar una exactitud de al menos 95% en todos los dígitos [42]. En este caso, se pudo llevar a cabo el procedimiento completo de comenzar entrenando la red en Python utilizando la librería Keras con backend de Tensorflow [43]. La arquitectura de la red implementada es la que se muestra en la Figura 34:

En dicha figura se puede ver entonces que se trata de una red densa cuyo vector de entrada corresponde a los 784 pixeles de los dígitos, estos son números entre 0 y 255. La primera y segunda capas ocultas son de 64 neuronas cada una con una activación ReLU, la última capa tiene 10 neuronas correspondientes a los 10 dígitos y su activación es softmax. Esta red, una vez entrenada se le realizó la optimización *pruning*, que consiste en que se impone la condición de que un número determinado de pesos sean cero, que, para el caso presente, se impuso que el 85% de pesos fueran cero. Esto tiene como fin una reducción en el tamaño de la red, y una aceleración de los cálculos pues todo número multiplicado por cero es cero. Adicionalmente, el *pruning* genera otro beneficio que es que al hacer muchos pesos cero, los números que calcula la red son más pequeños y por tanto se pueden necesitar menos bits para representar los pesos y los resultados de las multiplicaciones matriz-vector. Esto se constató pues, la red sin *pruning* no se pudo implementar en la FPGA utilizando puntos fijos, mientras que la red con pruning se implementó satisfactoriamente con puntos fijos de 18 bits. La exactitud alcanzada por la red se encuentra en la Figura 35.

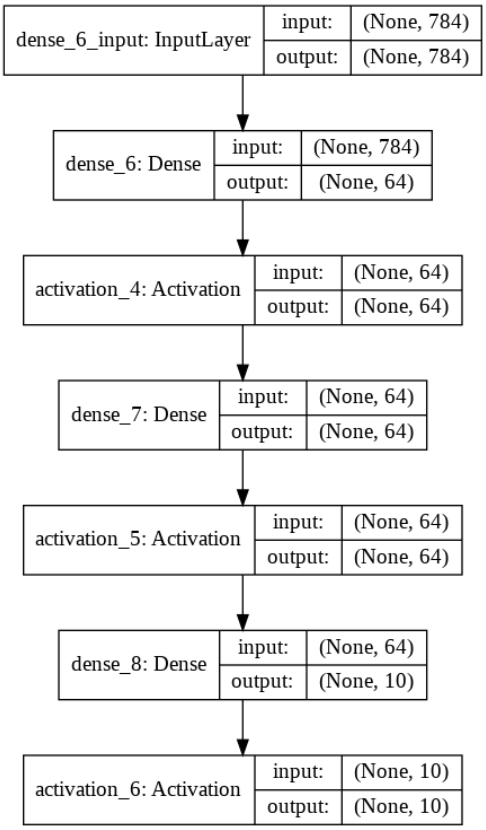

Figura 34: Red Neuronal para la clasificación MNIST

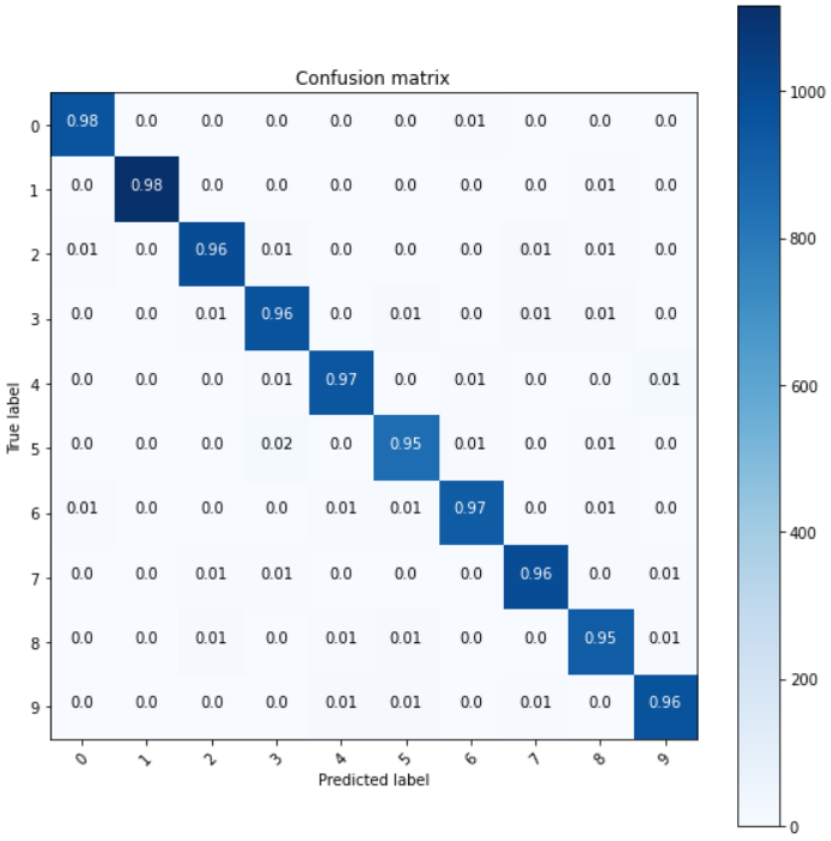

Figura 35: Exactitud red entrenada en Python para clasificación MNIST

Esta exactitud es satisfactoria pues la red clasifica todos los diez dígitos de manera correcta en promedio en el 96% de los casos. Una vez entrenada la red para esta tarea y habiendo alcanzado un rendimiento satisfactorio en Python, se exportó la arquitectura y pesos para ser implementados en la FPGA utilizando el paquete hls4ml.

Para dicha implementación, como se dijo un poco más atrás, se utilizaron precisiones de puntos fijos de 18 bits (9 bits para la parte entera y 9 para la parte decimal) para todos los pesos y resultados de la red. Adicionalmente, debido al tamaño mucho mayor del vector de entrada respecto a la aplicación anterior de clasificación de *jets*, donde el vector de entrada era de 16 elementos, mientras que en este caso el vector de entrada es de 784 elementos, fue necesario utilizar una arquitectura en la FPGA que es la AXI-Stream. Esta arquitectura permite un flujo mucho mayor de información hacia y desde la FPGA, con la ventaja adicional que se utiliza en conjunción con acceso directo a memoria (DMA: Direct Memory Access), con lo cual se tiene una velocidad mayor de procesamiento [20]. Una vez sintetizado el diseño mediante HLS, el diagrama de bloques de Vivado Design Suite es el siguiente:

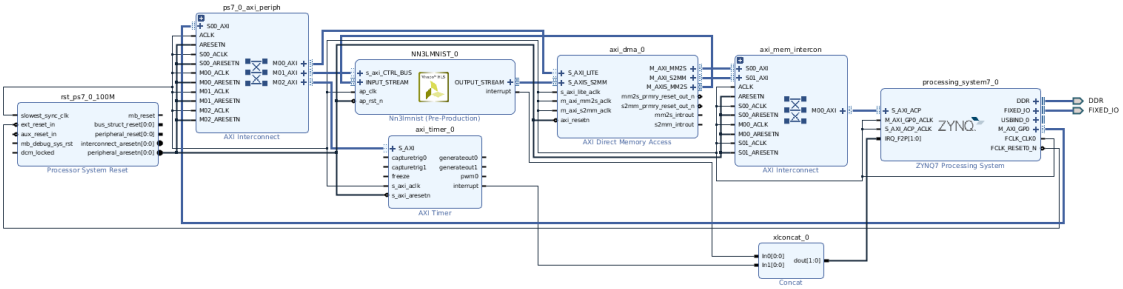

Figura 36: Diagrama de bloques clasificador MNIST

3.4.1 Resultados implementación red MNIST

Finalmente, con el *bitstream* se programó la FPGA, y luego se desarrolló la aplicación anfitriona en Vitis (SDK), para medir el tiempo de ejecución de 500 iteraciones, con el objetivo de tomar un promedio y desviación estándar, y luego comparar este rendimiento con igual número de corridas en la librería Tensorflow de Python [43], corriendo con el método *predict* de Keras [44], y utilizando un entorno de ejecución con GPU en Google Colab [45]. Los resultados se encuentran en la siguiente figura:

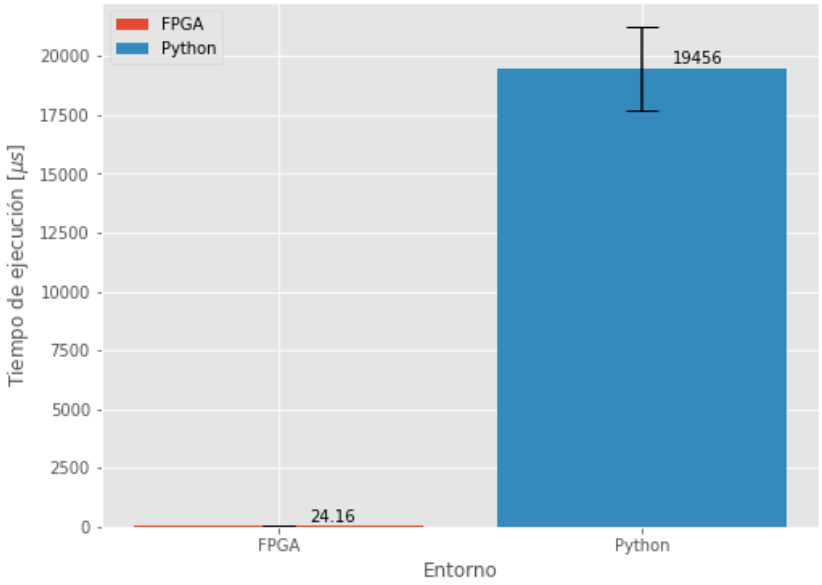

Figura 37: Tiempos de ejecución red MNIST

En la Figura 37 se puede observar que el tiempo de ejecución de la red ejecutada en la FPGA es de 24.16 ± 0.13 microsegundos, mientras que en Python, fue de 19456 ± 1786 microsegundos. Esto significa una ejecución 805 veces más rápido. Adicionalmente, la red implementada en la FPGA no presentó pérdidas en exactitud debido a la cuantización ni el *pruning*, y se conservó la exactitud de la red entrenada en Python, que fue de 96%.

Finalmente, respecto al uso de recursos de este IP, HLS reportó los siguientes porcentajes de utilización:

	BRAM_18K	***DSP48E***	***FF***	***LUT***
Total	166	202	25296	34033
Disponible	280	220	106400	53200
Utilización (%)	59	91	23	63

Tabla 6: Utilización recursos red MNIST

Estos porcentajes no son muy altos, si se comparan con los porcentajes de las implementaciones en el capítulo 2, aún cuando en la presente implementación se trata de una red más grande y además con precisiones más altas (18b). Esta buena utilización de los recursos de la FPGA se debe en parte a la librería hls4ml y en parte al uso de la arquitectura AXI-Stream [20].

A manera de conclusión, luego de implementar las anteriores redes, los resultados de este capítulo evidenciaron la superioridad en cuanto al rendimiento en tiempo de ejecución de la FPGA respecto a las mismas operaciones en Python, incluso utilizando GPU. Estas implementaciones abren la puerta a la utilización no sólo de redes neuronales sino de cualquier otro algoritmo en FPGAs pues el método de programación es completamente transferible. Esto es importante debido a que las FPGA son una tecnología de punta y disponible a bajo costo (en FPGAs pequeñas), cuya dificultad de implementación se encuentra principalmente en la programación, pero, habiendo adelantado el conocimiento del presente trabajo, las posibilidades experimentales e industriales se multiplican, al quedar abierta toda una gama de posibilidades en aplicaciones embebidas, de procesamiento de información proveniente directamente de diferentes tipos de sensores.

4 CONCLUSIONES

En el presente trabajo se realizó una contextualización de las relaciones entre Machine Learning, FPGAs y física, al tiempo que se procedió a comprender y llevar a cabo el proceso de programación de FPGAs, con un enfoque en el uso de la síntesis de alto nivel, con el objetivo de implementar redes neuronales, teniendo en mente la utilización de FPGAs en física, como lo fue la clasificación de subestructuras jets. Los resultados obtenidos nos permitieron constatar que al evaluar el rendimiento de las FPGA en cuanto a la inferencia de redes neuronales, éstas son superiores, en ocasiones en varios ordenes de magnitud, a alternativas como las CPU y GPU utilizadas con lenguaje Python y C.

Sin embargo, en este trabajo se fue un poco más allá y fue posible no solo comparar rendimientos, sino también establecer conexiones fructíferas con entornos de trabajo usuales en Deep Learning, como lo es el entrenamiento de redes neuronales utilizando las librerías Keras y Tensorflow, en un problema que va más allá del campo estricto de la física y que pertenece al campo de la visión artificial, con el objetivo de ampliar el horizonte de aplicaciones de FPGA a problemas generales, y evaluar su rendimiento en cuanto a tiempo de inferencia, donde quedó demostrada la superioridad de la FPGA por dos órdenes de magnitud.

El desarrollo de este proyecto de investigación nos abre la puerta a la utilización de FPGAs no sólo en el contexto de la física, sino en general al campo de la toma y el procesamiento rápido de datos en dispositivos embebidos. Los aprendizajes generados en el presente trabajo son aplicables a la programación en general de cualquier algoritmo, y en cualquier FPGA de Xilinx, en particular, en el presente trabajo se utilizó una FPGA de bajo costo como lo es la xc7z020clg400-1, presente en la tarjeta PYNQ-Z1, lo cual abre también la posibilidad de su uso en contextos con recursos limitados, como lo puede ser la docencia e investigación en universidades públicas de países en desarrollo.

5 REFERENCIAS

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. 2016.

[2] R. Plant, "An introduction to artificial intelligence," in *32nd Aerospace Sciences Meeting and Exhibit*, 1994.

[3] S. Mor-Yosef, A. Samueloff, B. Modan, D. Navot, and J. G. Schenker, "Ranking the risk factors for cesarean: Logistic regression analysis of a nationwide study," *Obstet. Gynecol.*, vol. 75, no. 6, pp. 944–947, Jun. 1990.

[4] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, 2013.

[5] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014, vol. 8689 LNCS, no. PART 1, pp. 818–833.

[6] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards, *Artificial Intelligence A Modern Approach*. New Jersey: Prentice Hall, 2010.

[7] Y. Lecun, "Deep Learning Hardware: Past, Present, and Future," in *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, 2019, vol. 2019-Febru, pp. 12–19.

[8] L. D. Jackel, R. E. Howard, H. P. Graf, B. Straughn, and J. S. Denker, "Artificial neural networks for computing," *J. Vac. Sci. Technol. B Microelectron. Nanom. Struct.*, vol. 4, no. 1, p. 61, Jan. 1986.

[9] Q. Zhang, M. Zhang, T. Chen, Z. Sun, Y. Ma, and B. Yu, "Recent advances in convolutional neural network acceleration," *Neurocomputing*, vol. 323, pp. 37–51, Jan. 2019.

[10] G. Lacey, G. W. Taylor, and S. Areibi, "Deep Learning on FPGAs: Past, Present, and Future," Feb. 2016.

[11] Lab LISA, "Deep Learning Tutorial Release 0," 2015.

[12] "Perceptrón multicapa - Wikipedia, la enciclopedia libre." [Online]. Available: https://es.wikipedia.org/wiki/Perceptrón_multicapa. [Accessed: 16-Jun-2020].

[13] "Convolutional Neural Networks: La Teoría explicada en Español | Aprende Machine Learning." [Online]. Available: https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/. [Accessed: 16-Jun-2020].

[14] J. Duarte *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *J. Instrum.*, vol. 13, no. 7, 2018.

[15] R. Kastner, J. Matai, and S. Neuendorffer, “Parallel Programming for FPGAs,” May 2018.

[16] S. I. Venieris, A. Kouris, and C. S. Bouganis, “Toolflows for mapping convolutional neural networks on FPGAS: A survey and future directions,” *ACM Computing Surveys*, vol. 51, no. 3. Association for Computing Machinery, 01-Apr-2018.

[17] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, “A Survey of FPGA-Based Neural Network Accelerator,” Dec. 2017.

[18] I. Xilinx, “Vitis Unified Software Platform Documentation Application Acceleration Development,” 2020.

[19] C. Mead and L. Conway, *Introduction to VLSI systems*. Addison-Wesley, 1980.

[20] Xilinx and Inc, “Vivado Design Suite User Guide: High-Level Synthesis (UG902),” 2018.

[21] “LegUp High-Level Synthesis Product Information | LegUp Computing.” [Online]. Available: https://www.legupcomputing.com/main/product. [Accessed: 17-Jun-2020].

[22] Mentor, “Catapult HLS - Mentor Graphics.” [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/. [Accessed: 17-Jun-2020].

[23] Xilinx, “Vivado Design Suite Tutorial High-Level Synthesis,” 2018.

[24] Xilinx, “pragma HLS pipeline.” [Online]. Available: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/fde1504034360078.html. [Accessed: 18-Jun-2020].

[25] D. Guest, J. Collado, P. Baldi, S. C. Hsu, G. Urban, and D. Whiteson, “Jet flavor classification in high-energy physics with deep neural networks,” *Phys. Rev. D*, vol. 94, no. 11, p. 112002, Dec. 2016.

[26] D. Acosta *et al.*, “Boosted Decision Trees in the Level-1 Muon Endcap Trigger at CMS,” *J. Phys. Conf. Ser.*, p. 42042, 2018.

[27] M. Lassnig, W. Toler, R. Vamosi, and J. Bogado, “Machine learning of network metrics in ATLAS Distributed Data Management Related content The effects of degree correlations on network topologies and robustness-C3PO-A Dynamic Data Placement Agent for ATLAS Distributed Data Management Recent citations Rucio: Scientific Data Management Martin Barisits et al-Fast inference of deep neural networks in FPGAs for Machine learning of network metrics in ATLAS Distributed Data Management,” *J. Phys. Conf. Ser.*, vol. 898, p. 62009, 2017.

[28] B. Schlag Gutachter, “Jet Reconstruction in the ATLAS Level-1 Calorimeter Trigger with Deep Artificial Neural Networks,” 2018.

[29] Y. Gershtein, “CMS hardware track trigger: New opportunities for long-lived particle searches at the HL-LHC,” *Phys. Rev. D*, vol. 96, no. 3, p. 035027, Aug. 2017.

[30] A. Paszke *et al.*, “Automatic differentiation in PyTorch,” Oct. 2017.

[31] L. de Oliveira, M. Kagan, L. Mackey, B. Nachman, and A. Schwartzman, “Jet-images — deep learning edition,” *J. High Energy Phys.*, vol. 2016, no. 7, pp. 1–32, Jul. 2016.

[32] R. Kogler *et al.*, “Jet Substructure at the Large Hadron Collider: Experimental Review,” *Rev. Mod. Phys.*, vol. 91, no. 4, Mar. 2018.

[33] J. M. Butterworth, A. R. Davison, M. Rubin, and G. P. Salam, “Jet substructure as a new higgs-search channel at the large hadron collider,” *Phys. Rev. Lett.*, vol. 100, no. 24, p. 242001, Jun. 2008.

[34] J. Thaler and K. Van Tilburg, “Identifying boosted objects with N-subjettiness,” *J. High Energy Phys.*, vol. 2011, no. 3, pp. 1–28, Mar. 2011.

[35] E. Coleman *et al.*, “The importance of calorimetry for highly-boosted jet substructure,” Sep. 2017.

[36] A. J. Larkoski, J. Thaler, and W. J. Waalewijn, “Gaining (mutual) information about quark/gluon discrimination,” *J. High Energy Phys.*, vol. 2014, no. 11, pp. 1–56, Nov. 2014.

[37] ATLAS Collaboration, “Light-quark and gluon jet discrimination in pp collisions at √ s = 7 TeV with the ATLAS detector,” *Eur. Phys. J.*, vol. 74, p. 3023, 2014.

[38] D. P. Kingma and J. L. Ba, “Adam: A method for stochastic optimization,” in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[39] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, “The NumPy array: A structure for efficient numerical computation,” *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 22–30, Mar. 2011.

[40] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2323, 1998.

[41] L. Deng, “The MNIST database of handwritten digit images for machine learning research,” *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, 2012.

[42] A. Rosebrock, *Deep Learning for Computer Vision with Python: ImageNet Bundle*. 2017.

[43] M. Abadi *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” Mar. 2016.

[44] “Keras: the Python deep learning API.” [Online]. Available: https://keras.io/. [Accessed: 25-May-2020].

[45] E. Bisong, “Google Colaboratory,” in *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, Apress, 2019, pp. 59–64.