Trabajo Fin de Máster
“Máster Universitario en Microelectrónica: Diseño y Aplicaciones de Sistemas Micro/Nanométricos”

Sistema de reconocimiento de imágenes sobre FPGA para aplicaciones de visión artificial

Adrián Campos Ramos

Tutores: María José Avedillo de Juan, Manuel Jiménez Través, Juan Núñez Martínez

Junio 2022

Agradecimientos

A mis padres, José María y Pepi, porque sin ellos no habría llegado aquí, por su apoyo incondicional y por sus consejos; y a mi hermano José María por estar a mi lado.

A dos amigos que me han apoyado muchísimo este año: Miguel Cidoncha, y Cristina Martín por aguantarme y ayudarme tanto.

También a Manuel Jiménez por su paciencia, ayuda y enseñanza.

Por último, agradecer especialmente a María José Avedillo de Juan, que desde que me impartió DSD ya hace unos años en segundo de carrera, me ha ido enseñando y apoyado a lo largo de los años y que, si no fuera por ella, no habría descubierto mi gusto por la programación Hardware y mi vocación profesional.

Resumen

Con el transcurrir de los años han surgido numerosas aplicaciones en campos muy diversos, como la medicina, la seguridad o la educación, que hacen uso de la visión artificial. Muchas de estas aplicaciones requieren la detección e identificación de objetos o personas. Las redes neuronales, y en particular las redes neuronales convolucionales, han demostrado su utilidad para estas tareas y han permitido un desarrollo muy significativo de este campo. Sin embargo, a veces es mucho más simple y efectivo aplicar soluciones ya conocidas del campo de *Machine Learning*.

Los dispositivos *Zynq All Programmable System on Chip* (AP SoC) pueden llegar a ser muy efectivos. La posibilidad de dividir el trabajo entre la lógica programable y su sistema de procesamiento nos presenta una solución factible para la implementación de grandes algoritmos y mayor velocidad que la ejecución total en un sistema de procesamiento.

Este Trabajo Fin de Master (TFM) aborda la implementación de un sistema para el reconocimiento de imágenes de dígitos manuscritos (0-9) sobre la placa de desarrollo Pynq-Z2 que incorpora un dispositivo AP *SoC* de la familia *Zynq* de Xilinx. El desarrollo se ha realizado en el entorno PYNQ, que permite el uso de aceleradores hardware desde Python. De esta forma se han implementado funciones aceleradas en hardware que son llamada desde Python.

El acelerador almacena un conjunto de imágenes de referencia con las que comparar la imagen a clasificar, identificando la más próxima. Esto es, se implementa para la clasificación un algoritmo del “vecino más próximo”. La selección de las imágenes de referencia aplicando otro algoritmo clásico, el *k-means,* permite obtener soluciones con un compromiso satisfactorio entre tiempo de inferencia, *accuracy* (precisión) y recursos utilizados.

Además, se realiza una comparación con un sistema que utiliza redes neuronales binarias.

La funcionalidad del sistema desarrollado se demuestra mediante la ejecución de un documento interactivo (Jupyter Notebooks) accesibles a través de una interfaz web,

Palabras claves: *FPGA*, Zynq, PYNQ, Xilinx, aceleración hardware, distancia, *accuracy*.

Índice

Índice de Tablas .... A

Índice de Ilustraciones .... B

Índice de Códigos .... D

1. Introducción 1

1.1 Justificación 1

1.2 Objetivos 3

1.3 Estructura de la memoria 4

2. Fundamentos y entornos 5

2.1 MNIST y reconocimiento de imágenes 5

2.2 PYNQ 6

2.2.1 Pynq como entorno de trabajo 6

2.2.2 Placa de desarrollo Pynq-Z2 8

2.2.3 Dispositivo Zynq XC7Z020 10

2.2.4 Jupyter Notebook 18

2.3 Herramientas de diseño 19

2.3.1 Vivado IDE 20

2.3.2 Vitis HLS 21

2.3.3 Matlab 23

3. Desarrollo e implementación 24

3.1 Desarrollo y evaluación del algoritmo *K-means* 24

3.2 Diseño del acelerador hardware 35

3.2.1 Optimizaciones 39

3.3 Diseño del *Overlay* 43

3.4 Comparación con BNN 47

4. Prueba del sistema 49

4.1 Aplicación Python 49

4.2 Ejemplos de operación 52

5. Conclusiones y líneas futuras 55

Bibliografía 57

ANEXO I 59

Índice de Tablas

Tabla 1 Características Pynq Z2[7].. 9
Tabla 2 Lista de Interfaces de periféricos de E/S [8]... 15
Tabla 3 Interfaces entre PS y PL [8]... 17
Tabla 4 Accuracy de diferentes sets de referencias obtenidos con valores distintos de K 28
Tabla 5 Accuracy diferentes sets de referencias binarios con diferentes k 33
Tabla 6 Resultados Síntesis para distintas cardinalidades del set de referencia 39
Tabla 7 Soluciones aplicación 54

Índice de Ilustraciones

Ilustración 1 Arquitectura de plataforma software hardware para edge computing...... 3
Ilustración 2 Ejemplos de imágenes en la base de datos MNIST 5
Ilustración 3 Esquema de la jerarquía de PYNQ [5] 8
Ilustración 4 Plataforma Pynq-Z2 8
Ilustración 5 Arquitectura del dispositivo Zynq [8].. 10
Ilustración 6 Lógica Programable y sus elementos [8]... 10
Ilustración 7 Composición de un CLB [8].. 12
Ilustración 8 Sistema de Procesado de Zynq [8] 13
Ilustración 9 Diagrama de bloques simplificado de la APU [8]................................. 13
Ilustración 10 Uso de EMIO como interfaz entre PS y PL [8].................................... 18
Ilustración 11 Flujo de co-diseño.. 19
Ilustración 12 Arquitectura y Flujo de diseño Vivado HLS 21
Ilustración 13 Funcionamiento Algoritmo K-means 25
Ilustración 14 Resultado Algoritmo K-means K= 2 28
Ilustración 15 Resultados obtenidos por diferentes sets de referencias generados... 29
Ilustración 16 Resultados obtenidos por diferentes sets de referencia 34
Ilustración 17 Resultados síntesis de función principal. Set de referencia de cardinalidad1000 38
Ilustración 18 Pragma Pipeline [28]... 39
Ilustración 19 Funcionamiento Pragma HLS Array_Partition [29] 40
Ilustración 20 Directivas HLS empleadas 41
Ilustración 21 Informe síntesis HLS con optimizaciones y reloj 200MHz 41
Ilustración 22 Recursos obtenidos tras la síntesis lógica con reloj de 200 MHZ. 42
Ilustración 23 Informe síntesis HLS con optimizaciones y reloj de 100 MHZ 42
Ilustración 24 Recursos obtenidos tras la síntesis lógica con reloj de 100 MHZ 42
Ilustración 25 Diagrama de bloques del DMA [31].. 44
Ilustración 26 Configuración DMA 44
Ilustración 27 Processing System IP Wrapper [32].. 45
Ilustración 28 Overlay diseñado 46
Ilustración 29 Recursos utilizados por la red LFC de FINN [33] 47
Ilustración 30 Recursos utilizados por el Overlay diseñado...................................... 47
Ilustración 31 Recursos Overlay vs LFC 48
Ilustración 32 Inferencia con LFC... 48

Ilustración 33 Imagen Test de entrada 52
Ilustración 34 Imagen Test de entrada formato deseado... 52
Ilustración 35 Imagen Salida función clasificación... 53
Ilustración 36 Jupyter Notebook 60

Índice de Códigos

Código 1 Algoritmo K-means 26
Código 2 Código Distancia Euclidiana 27
Código 3 Binarización Set MNIST........ 30
Código 4 K-means binario........ 31
Código 5 Función Distancia Hamming 32
Código 6 Librería propia HLS. Archivo principal.h........ 35
Código 7 Función distancia entre imágenes 36
Código 8 Función Principal para HLS 37
Código 9 Función Python Conversión formato de entrada 49
Código 10 Función Python Clasificación........ 50
Código 11 Función Python Imagen e índice resultantes 51
Código 12 Función Python clasificación total........ 51

1. Introducción

1.1 Justificación

En los últimos años ha aparecido el término de "Internet *of things*" o IoT que describe una red de pequeños y grandes objetos compuestos por una serie de sensores, actuadores, software u otras tecnologías con el propósito de conectar e intercambiar datos con otros dispositivos o servidores por Internet. Sin ninguna duda, cada día existen más dispositivos conectados a Internet, desde electrodomésticos, como un frigorífico, hasta automóviles. Aunque no se percibe ninguna dificultad en conectar distintos dispositivos a un servidor, con el aumento de los clientes, el ancho de banda se vuelve un elemento crítico, un cuello de botella. La enorme cantidad de datos que puede recibir un servidor de los distintos dispositivos pueden llegar *a* ralentizar, incluso bloquear, la red. Como solución, ha aparecido el término "*Edge* computing" cuyo objetivo es el procesamiento y/o almacenamiento de los datos lo más próximo a los clientes con el fin de reducir la carga de datos en el envío al servidor. En muchos casos, dicho procesado se basa en el uso de algoritmos computacionalmente intensivos que se utilizan para el procesamiento de los datos: redes neuronales, *deep learning* o machine *learning* (ML). Estos algoritmos requieren realizar millones de operaciones de punto flotante y almacenar un número muy elevado de parámetros.

En grandes procesadores dicho cómputo no implica gran problema. Sin embargo, hay limitaciones en el ámbito del *IoT*, donde el coste, el consumo de energía o el tamaño son elementos críticos. Los procesadores/microprocesadores usados no son los suficientes potentes.

Para intentar solventar el problema que conlleva el *Edge computing* se presentan como solución el uso de *All programable System on Chips* (APSoCs) para la aceleración en hardware de estos algoritmos. Estos sistemas combinan el mundo software y hardware en un único chip, al incluir microprocesadores y lógica programable, en la que acelerar las partes computacionalmente más costosas de una aplicación. Los beneficios del trabajo en paralelo, la velocidad de procesamiento o la eficiencia en términos de energía que proporciona la lógica programable frente a otros sistemas de procesamiento hacen esta tecnología ideal para el uso del IoT, con la posibilidad de procesar datos en tiempo real, o gran parte, sin tener que contactar con el servidor principal.

Sin embargo, la metodología de diseño hardware dominante (RTL a silicio), requiere un conocimiento profundo del modelado y de la operación de los sistemas digitales a bajo nivel de abstracción que, junto a tiempos de diseño significativamente más largos que los que el desarrollo de software conlleva, puede limitar la expansión de esta tecnología. El progreso significativo que metodologías a mayor nivel de abstracción (*Electronic Level System*) han experimentado en los últimos años potenciará el uso de los APSoCs para el *edge computing*. Nos referimos a describir el hardware en lenguajes tipo C++ y aplicar herramientas de síntesis de alto nivel (*High Level Synthesis* o HLS) que generen las descripciones a nivel RTL.

En particular, el *edge computing* va ser relevante en aplicaciones como la visión artificial. Así, hemos elegido el reconocimiento de imágenes para el sistema que proponemos implementar en un *FPGA* APSoC para beneficiarse de la aceleración hardware

Comenzamos por estudiar una plataforma hardware sobre APSoC de la familia Zynq de Xilinx y una aplicación construida sobre ella (disponible públicamente en [1]para el reconocimiento de dígitos escritos a mano usando la popular base MNIST.[2] Esta plataforma incorpora un acelerador hardware de una red neuronal (*Large Full Connected* (LCF) o *Small Full Connected* (SFC)) cuantizada, esto es, usando un número muy reducido de bits para los pesos y la activación. Un único bit en el caso límite, denominándose entonces red neuronal binaria (BNN).

Si bien su *accuracy* está por debajo de lo que se consigue con implementaciones software usando redes más complejas, se compensa con el uso reducido de recursos, siendo adecuada para una computación *edge* en tiempo real. De acuerdo con [1] la red LFC W2A2 implementada en una *FPGA* ZC706 consigue un *accuracy* de 95.8% con latencia de 0.31µs por imágenes. Por otra parte, la aplicación que se acelera usa Python, lo cual es relativamente novedoso siendo un sistema empotrado y de utilidad en el contexto del procesado de imágenes.

En este contexto nos propusimos desarrollar un sistema alternativo al anterior, competitivo en términos de recursos y/o tiempo de inferencia. Puesto que la BNN proporcionaba buenos resultados utilizando imágenes binarias (blanco y negro), nuestra hipótesis fue que aproximaciones clásicas de *machine learning* basadas en algoritmos de proximidad podrían ser muy adecuados para una implementación hardware.

1.2 Objetivos

Explorar el desarrollo de plataformas hardware basadas en *ApSoCs* como alternativa a las limitaciones del *edge computing* usando una metodología ESL. En concreto, se propone el diseño de un sistema de reconocimiento de imágenes usando una función acelerada en hardware que calcula la distancia entre una imagen a evaluar y un conjunto (*set*) de referencia; y su comparación con la solución basada en redes neuronales aceleradas en hardware mencionadas anteriormente

La Ilustración 1 muestra la arquitectura de un sistema tipo.

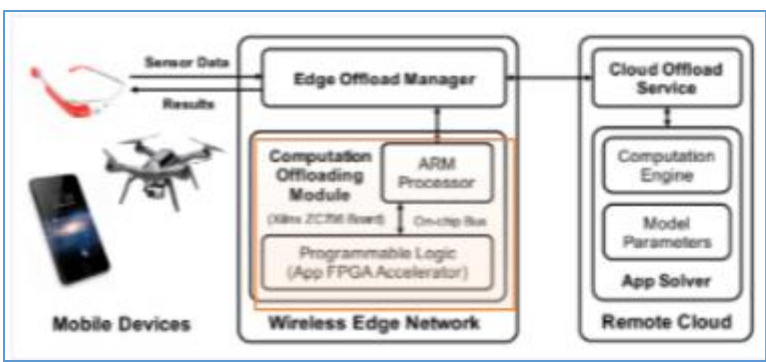


Ilustración 1 *Arquitectura de plataforma software hardware para edge computing*

Para ello, utilizaremos una placa Pynq-Z2 que incorpora un dispositivo Zynq de Xilinx junto a los entornos de diseño propios (Vivado y PYNQ), que nos facilitarán el desarrollo y uso de "*Overlays*", plataformas hardware diseñados para acelerar la ejecución de funciones en lógica programable, utilizando un lenguaje intuitivo, como Python, lo que nos permitirá diseñar aplicaciones de una forma más sencilla.

Los objetivos concretos del trabajo son los siguientes:

a) Exploración del entorno de diseño y desarrollo PYNQ de Xilinx.
b) Desarrollo y evaluación de un algoritmo de reconocimiento de imágenes basado en el concepto de vecindad.
c) Exploración de Vitis HLS, previamente conocido como Vivado HLS, y las optimizaciones que presenta.
d) Desarrollo de Notebooks para la ejecución de la aplicación.

1.3 Estructura de la memoria

La memoria se ha organizado como se describe a continuación.

En el Capítulo 2 se explica de forma resumida los fundamentos teóricos, así como los entornos utilizados en el desarrollo del proyecto.

En el Capítulo 3 se describe el diseño del sistema implementado, comenzando por el desarrollo a nivel algoritmo y su evaluación en el entorno Matlab, el diseño de su implementación hardware usando síntesis de alto nivel y la generación de la plataforma hardware. Además, se compara el sistema desarrollado con el que usa la BNN y que tomamos como referencia.

En el Capítulo 4 se reportan las pruebas del sistema, incluyendo la descripción de la aplicación Python desarrollada para ello.

Por último, el Capítulo 5 resume las conclusiones obtenidas e indica líneas de trabajo futuro.

2. Fundamentos y entornos

2.1 MNIST y reconocimiento de imágenes

MNIST es una base de datos de dígitos escritos a mano. Compuesto por un conjunto de entrenamiento de 60000 ejemplos y un conjunto de prueba de 10000 ejemplos. Proviene de un conjunto más amplio llamado NIST [2].

En MNIST, las imágenes originales en blanco y negro del conjunto mayor NIST se normalizaron en un tamaño de 20x20 píxeles conversando su relación de aspectos. Las imágenes resultantes contienen niveles de gris como resultado de la técnica de *antialiasing* utilizada por el algoritmo de normalización. Posteriormente, las imágenes se centraron en una imagen de 28x28 calculando el centro de masa de los píxeles y trasladando la imagen para situar este punto en el centro del campo de 28x28. La Ilustración 2 muestra ejemplos de imágenes contenidas en MNIST[2].

Ilustración 2 *Ejemplos de imágenes en la base de datos MNIST*

El algoritmo KNN (K-Nearest Neighbors) es un algoritmo aprendizaje supervisado muy utilizado para clasificación de patrones. Fue Introducido en los años 50 y posteriormente extendido [3].

Los patrones de entrenamiento son vectores en un espacio multidimensional etiquetados con la clase a la que pertenecen. La fase de entrenamiento del algoritmo consiste únicamente en el almacenamiento de dichos patrones de entrenamiento y la información sobre a qué clase pertenecen. En la fase de clasificación

(reconocimiento/inferencia), un patrón de test, y por tanto no etiquetado, se le asigna la clase más frecuente de entre los K patrones de entrenamiento más cercanos. Pueden usarse distintas métricas de distancia según el tipo de datos que se estén clasificando. El parámetro K lo define el usuario. Si K = 1, se asigna al patrón de test la clase del vecino más próximo [3], [4].

2.2 PYNQ

2.2.1 Pynq como entorno de trabajo

Para la realización de este proyecto se ha utilizado PYNQ como entorno de trabajo. PYNQ es un proyecto de código abierto de Xilinx que facilita el uso de plataformas hardware basadas en dispositivos APSoCs. Utilizando el lenguaje Python, junto a sus librerías, permite explorar los beneficios de la lógica programable y los procesadores de propósito general de una forma eficaz y sencilla. Actualmente PYNQ está disponible para dispositivos Zynq, Zynq UltraScale+, Zynq RFSoC, y placas de aceleración Alveo y AWS-F1, permitiendo crear aplicaciones para ejecución paralela en hardware, procesamiento de vídeo, aceleración de algoritmos por hardware, etc…[5]

El objetivo de **PYNQ** (acrónimo de **“*Py*}***thon productivity for Zy***nq**”) es facilitar a los usuarios de sistemas integrados los beneficios que ofrecen los dispositivos Zynq sin tener que utilizar herramientas de diseño complejas, como las necesarias para crear circuitos lógicos programables[5]

Este objetivo se consigue utilizando un conjunto de herramientas y entornos de diseño agrupados en una jerarquía que trabaja a tres “niveles”, como se representa en la *Ilustración 3.*

- Los circuitos diseñados en lógica programable se constituyen como librerías hardware, llamadas *Overlays*, compuestas por tres archivos: un archivo .bit, un archivo .hwh y un archivo .tcl. Un ingeniero de sistemas puede seleccionar el *Overlay* que más le convenga para su aplicación. Se puede acceder al *Overlay* a través de una interfaz de programación de aplicaciones (API). Sin embargo, la creación de un nuevo *Overlay* sí requiere el conocimiento de técnicas y herramientas de diseño sobre lógica programable [6]

- PYNQ utiliza Python como lenguaje de programación para manejar los *Overlays*. Hasta la fecha, C o C ++ son los lenguajes de programación más comunes en el contexto de sistemas empotrados, mientras que Python eleva el nivel de abstracción de programación y la productividad del programador. Debido a esto, PYNQ usa CPython, la implementación oficial y más utilizada del lenguaje Python, que está escrito en C, e integra miles de bibliotecas C y puede ampliarse con código optimizado escrito en C. Para uso práctico se utilizará Python, pero cuando se necesita eficiencia, se puede usar código C [6]

- El entorno es un proyecto de código abierto que tiene como objetivo trabajar en cualquier plataforma informática y sistema operativo. Para ello, utiliza una arquitectura basada en web. PYNQ incorpora la infraestructura de código abierto Jupyter Notebook para ejecutar un núcleo interactivo (*Interactive* Python o IPython) y un servidor web directamente en el procesador ARM del dispositivo Zynq. El servidor web accede al núcleo mediante instrucciones, un terminal *bash*, editores de código y Jupyter Notebooks. A su vez, las herramientas del navegador están implementadas mediante una combinación de JavaScript, HTML y CSS; y se puede ejecutar en cualquier navegador [6]

En resumen, PYNQ es el primer proyecto en combinar los siguientes elementos para simplificar y mejorar el diseño con APSoCs:

- Lenguaje de productividad de alto nivel (Python).
- *Overlays* de *FPGA* con drivers implementados como bibliotecas Python.
- Arquitectura basada en web operada desde el procesador.
- Entorno de trabajo basado en Jupyter Notebooks.

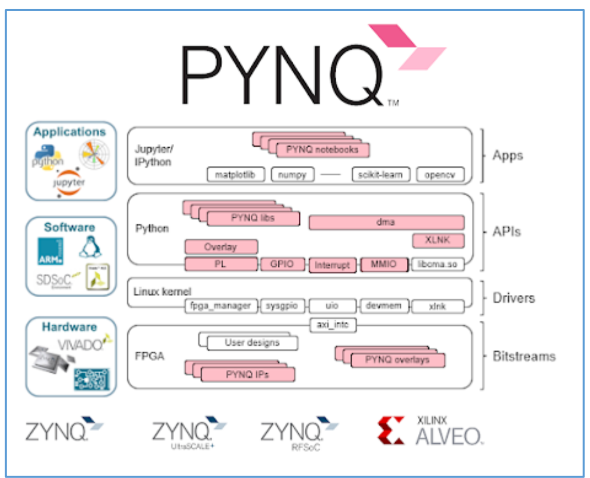


Ilustración 3 *Esquema de la jerarquía de PYNQ* [5]

2.2.2 Placa de desarrollo Pynq-Z2

Actualmente, existen cuatros placas Zynq oficialmente compatibles con PYNQ: Pynq-Z1 de Digilent, Pynq-Z2 de TUL, ZCU104 de Xilinx y ZCU111 de Xilinx. Para este trabajo se ha utilizado la Pynq-Z2. Esta placa incorpora un dispositivo Zynq XC7Z020 que combina un procesador de doble núcleo ARM Cortex A9 con lógica programable de la familia Artix-7 de 28nm.

En la *Ilustración 4 y la Tabla 1* se muestran las diferentes características de la placa Pynq-Z2, junto a los diferentes periféricos que ofrece. A continuación, se detalla el elemento principal de la placa, el dispositivo Zynq, así como los periféricos que utilizaremos en este proyecto [7] .

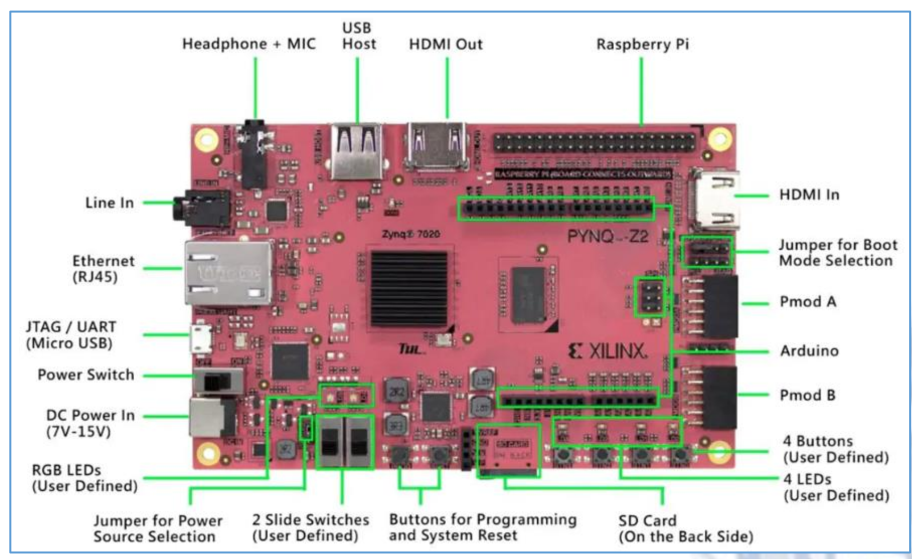


Ilustración 4 *Plataforma Pynq-Z2*

PYNQ Z2	
ZYNQ XC7Z020-1CLG400C	- Procesador Cortex-A9 de doble núcleo a 650 MHz. - Controlador de memoria DDR3 con 8 canales DMA y 4 puertos AXI3 esclavos de alto rendimiento. - Controladores periféricos de elevado ancho de banda (1G Ethernet, USB 2.0m SDIO). - Controladores periféricos de bajo ancho de banda (SPI, UART, CAN, I2C). - Programable desde JTAG, Quad-SPI flash, y tarjeta MicroSD. - Lógica programable equivalente a Artix-7 *FPGA*. - 13300 logic slices con cuatro LUTs de 6 entradas y 8 flip-flops. - 630 KB de bloques de memoria RAM. - 4 agentes de gestión de reloj, cada uno con bucle bloqueado (PLL) y reloj de modo mixto (MMCM). - 220 DSP slices. - Convertidor analógico digital (XADC).
Memoria	- 512 MB DDR· con buses de 16 bits a 1050 Mbps. - Flash Quad-SPI de 16 MB con programación de fabrica del identificador 48-bit globally unique EUI-48/64. - MicroSD slot.
Alimentación	- Alimentación por USB o entrada de alimentación externa de 7V-15V.
USB y Ethernet	- Gigabit Ethernet PHY - Micro USB-JTAG Programming circuitry - Micro USB-UART brige. - USB 2.0 OTH PHY (solo admite host).
Audio y Video	- HDMI sink port (input). - HDMI source port (output). - I2S interface con convertidor digital analógico de 24 bits con 3.5 mm TRRS jack. - Line-in with 3.5 mm jack.
Switches, Push-buttons and Leds	- 4 push-buttons. - 2 slide switches. - 4 LEDs. - 2 RGB LEDs.
Conectores de expansión	- 2 puertos PMODS estándar:16 pines I/O (8 compartidos con conector Raspberry PI). - Conector Arduino Shield: 24 pines I/O. - 6 entradas analógicas Single-ended 0.3-3V al XADC. - Conector Raspberry Pi: 28 pines I/O

Tabla 1 Características Pynq Z2[7]

2.2.3 Dispositivo Zynq XC7Z020

El dispositivo Zynq consta de dos partes principales: una lógica programable (PL), equivalente a una *FPGA*; y un sistema de procesado (PS) formado por un procesador de doble núcleo ARM Cortex-A9. Ambas partes pueden trabajar en forma conjunta o de manera individual. El dispositivo incluye además memoria integrada, así como una variedad de periféricos y de interfaces de comunicación de alta velocidad. Se puede observar la arquitectura del dispositivo en la *Ilustración 5* [8].

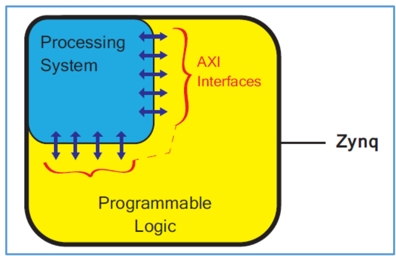


Ilustración 5 *Arquitectura del dispositivo Zynq* [8]

El bloque PL es muy eficiente para la implementación de lógica de alta velocidad, aritmética y subsistemas de flujo de datos, mientras que el PS ejecuta rutinas software y/o un sistema operativo, permitiendo el uso de hardware y software para la ejecución de aplicaciones. La comunicación entre PL y PS se realiza mediante el estándar de conexión "*Advanced eXtensible Interface* (AXI)"[9].

Lógica Programable (PL)

La parte de lógica programable (PL), que se muestra en la *Ilustración 6*, consta de los siguientes componentes:

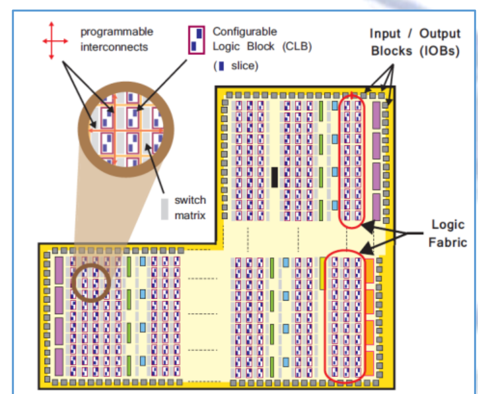


Ilustración 6 *Lógica Programable y sus elementos* [8]

- **Configurable Logic Blocks (CLBs)** [10]**:** Son pequeños grupos regulares de elementos lógicos que están dispuestos en un *array* bidimensional en la PL, y conectados a otros recursos similares mediante interconexiones programables. Cada CLB está compuesto por dos *slices* lógicos conectados a una matriz de conmutación (*switch matrix*), como se muestra en la *Ilustración 7.*

- ***Slice*****:** Subunidad localizada dentro de los CLBs, que contiene recursos para implementar circuitos lógicos combinacionales y secuenciales. Se compone principalmente de 4 Lookup Tables (LUT) y 8 Flips-Flops.

- ***Lookup Table*** **(LUT):** recurso flexible capaz de implementar: una función lógica de hasta de 6 entradas, una pequeña memoria de solo lectura (ROM), una pequeña memoria de acceso aleatorio (RAM) o un registro de desplazamiento (*shift register*). Se pueden combinar LUTs para formar funciones lógicas, memorias o shfit registers de mayor tamaño.

- ***Flip-flop*** **(FF):** Un elemento de circuito secuencial que implementa un registro de 1 bit con re*set*eo funcional. Algunos de los FFs de cada *slice* pueden usarse opcionalmente para implementar un *latch*.

- ***Switch Matrix*****:** Situado al lado de cada CLB, proporciona un mecanismo de enrutamiento flexible para hacer conexiones entre elementos internos de un CLB o de un CLB a otro recurso de la PL.

- ***Carry logic*****:** Propaga señales de acarreo entre *slices* adyacentes para circuitos aritméticos.

- ***Input/Output Blocks*** **(IOBs):** Recursos localizados alrededor del perímetro del dispositivo que permiten conectar la lógica programable con los *pads* utilizados para conectarse a circuitería externa. Cada IOB puede manejar señales de entrada y salida de 1 bit.

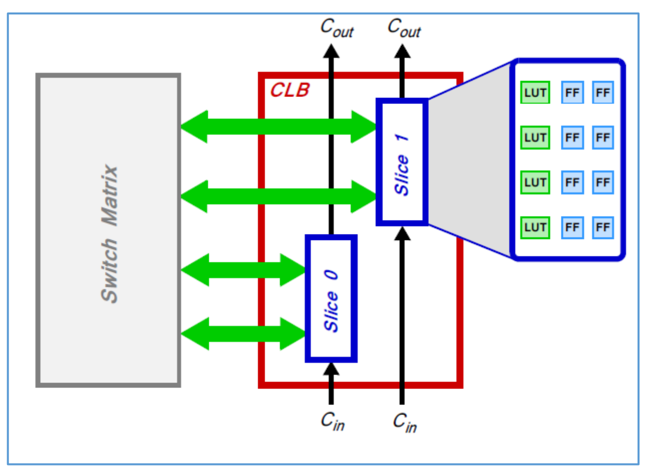


Ilustración 7 *Composición de un CLB* [8]

El dispositivo Zynq incorpora también dos elementos con un propósito general: bloques de memoria RAM y módulos DSP48E1 para aritmética de alta velocidad. Están integrados en la matriz lógica dispuestos en columnas normalmente próximas entre sí (ya que la computación intensiva y el almacenamiento de datos en memoria son operaciones que suelen estar asociadas).

Cada Bloque RAM puede almacenar 36Kb de información y puede configurarse como una RAM de 36 Kb o dos RAMs independientes de 18Kb.

Además, los *FPGA* de Serie 7 incluyen otros recursos, como bloques para la gestión de señales de reloj (CMT) y control del *jitter*, módulos integrados para PCI Express o transceptores Gigabit de baja potencia.

Una descripción más detallada de estos recursos adicionales de la lógica programable está disponible en el Data Sheet de los *FPGA* Serie 7 [11].

Processing System (PS)

En la *Ilustración 8* podemos observar la arquitectura y la composición sistema de procesado. El elemento principal es la **Application Processor Unit (APU)**, o unidad de procesador de aplicación en español, que está compuesta por dos núcleos ARM, cada uno de los cuales integra las siguientes unidades funcionales: un acelerador NEON para instrucciones SIMD (*NEON Media Processing Engine* - MPE), un coprocesador de punto flotante (*Floating Point Unit* - FPU), una unidad de gestión de memoria (*Memory Management Unit* - MM1) y una memoria caché de nivel 1 (en 2 secciones, para instrucciones y datos). Además, el PS contiene una memoria caché de nivel 2 y una memoria adicional en chip (OCM). La comunicación entre los núcleos ARM y la caché

nivel 2 y la memoria OCM se realiza a través de la Unidad de Control *Snoop* (SCU). Esta unidad también tiene la responsabilidad de interconectar el PS con la PL.

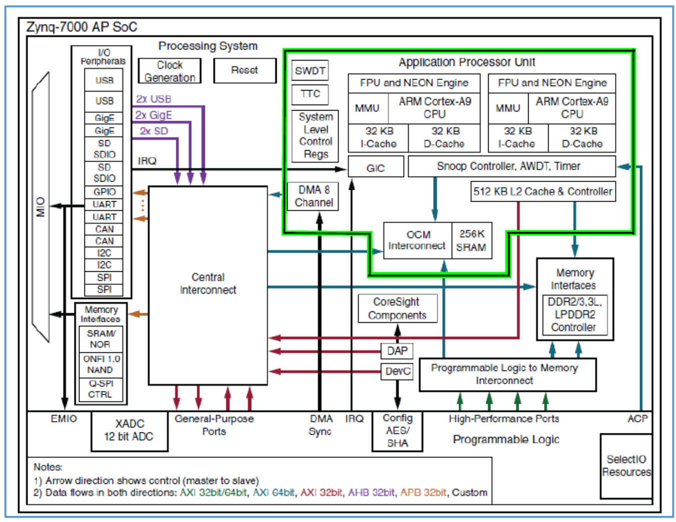


Ilustración 8 *Sistema de Procesado de Zynq* [8]

En la *Ilustración 9* podemos observar un esquema simplificado de los elementos de la APU así como la conexión entre los diferentes elementos que la componen.

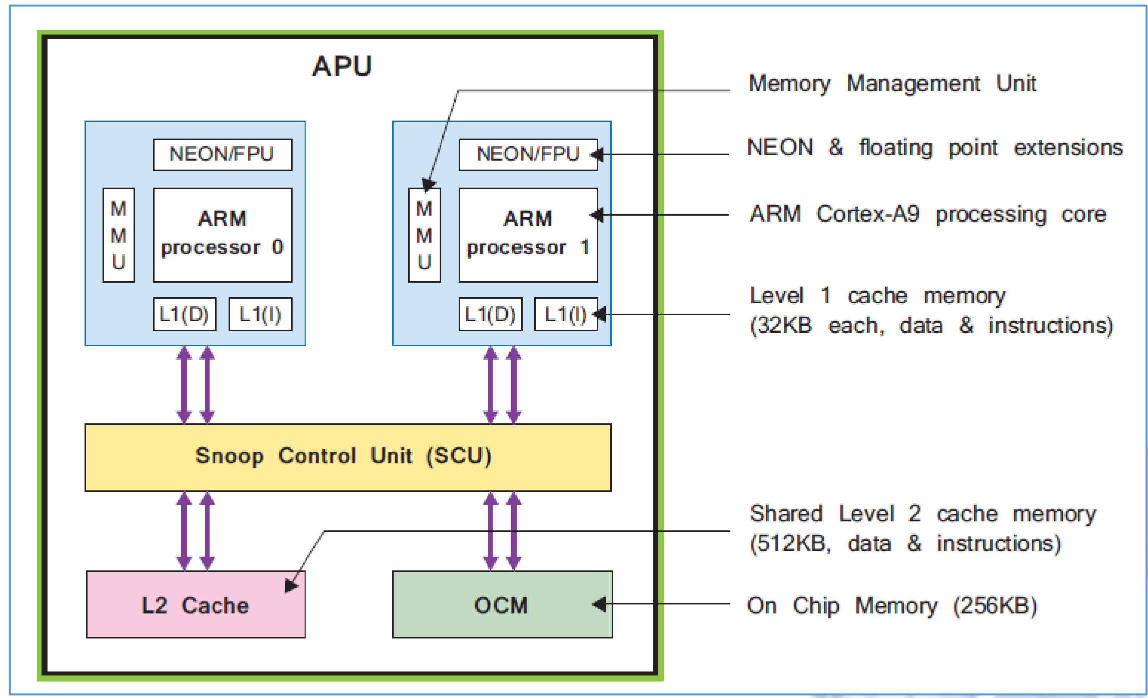


Ilustración 9 *Diagrama de bloques simplificado de la APU* [8]

La comunicación entra el PS y las interfaces externas se lleva principalmente a cabo a través del multiplexor de entrada/salida (MIO), que provee 54 pines de conectividad flexible, donde el mapeo entre periféricos y pines se puede definir si es requerido. Otras conexiones pueden realizarse mediante el Extended MIO (EMIO), que no es ruta directa desde el PS a las conexiones externas, sino que pasa y comparte recursos de entrada y salida de la PL [12].

Los puertos de entrada y salida incluyen distintas interfaces de comunicación estándar y entrada/salida de propósito general (GPIO) que dan acceso a una variedad de elementos, como botones, switches o LEDs.

En la *Tabla 2,* se representan los diferentes interfaces del Zynq, así como una breve descripción. En "Zynq 7000 *Technical Reference Manual*" [13] puede encontrarse información completa de cada interfaz.

Interfaz E/S	Descripción
SPI (x2)	- Serial Peripheral Interface Estándar de comunicación basado en una interfaz de 4 pines. Puede usarse como maestro o esclavo.
I2C (x2)	- I^2C bus. Suporta modo esclavo y maestro.
CAN (x2)	- Controller Area Network. Controlador de interfaz de bus compatible con los estándares ISO 118980-1, CAN 2.0A y CAN 2.0B.
UART (x2)	- Universal Asynchronous Receiver. Interfaz de datos de baja velocidad para comunicación en serie. A menudo se usa para conexiones de Terminal a un PC host.
GPIO	- General Purpose Input/Output 4 grupos de GPIO, cada uno de 32 bits.
SD (x2)	- Interfaz con la memoria SD
USB (x2)	- Universal Serial Bus Cumple con USB 2.0 y se puede usar como host, dispositivo o de forma flexible (modo “on-the-go” o OTG, lo que significa que puede cambiar entre los modos host y dispositivo).
GigE (x2)	- Ethernet Periférico Ethernet MAC, compatible con modos de 10Mbps, 100Mbps y 1Gbps.

Tabla 2 *Lista de Interfaces de periféricos de E/S* [8]

Interfaces Sistema de Procesado – Lógica Programable

La comunicación entre PS y PL se realiza principalmente mediante el conjunto de interconexiones e interfaces AXI. Además, existen otros tipos de conexiones entre PS y PL, en particular EMIO

Estándar AXI

AXI es el acrónimo de "*Advanced eXtensible Interface*". Actualmente la versión AXI4 forma parte del estándar abierto de ARM AMBA 3.0. Muchos dispositivos y bloques IP producidos por terceros están basado en él.

Originalmente, el estándar AMBA fue diseñado para el uso en microcontroladores en la primera versión lanzada en 1996. Actualmente se enfoca hacia sistemas en chip, incluyendo aquellos en *FPGA*s o, en caso de Zynq, dispositivos que incluyen procesadores y lógica de *FPGA* en el mismo chip. De hecho, Xilinx contribuyó en la definición de AXI4 como tecnología óptima de interconexión para el uso en arquitecturas *FPGA*s [14] [15].

El bus AXI puede usarse flexiblemente y, en sentido general, se utiliza para conectar procesadores y otros bloques IP en sistemas empotrados. Concretamente, AXI4 dispone de tres formatos, que representan diferentes protocolos de bus:

- **AXI4:** Para la interconexión de bloques mapeados en memoria que requieran un elevado ancho de banda. Permite transferencias en modo ráfaga: se proporciona una dirección seguida de una secuencia de datos de hasta 256 palabras [16].
- **AXI4-Lite**: Versión simplificada que no soporta transmisión por ráfagas. Como AXI4, también está mapeado en memoria, pero, en este caso, solo una dirección y una palabra de datos pueden ser transferidas en cada acceso al bus [16].
- **AXI4-Stream**: Especialmente indicado para transferencias punto a punto de alta velocidad sin restricciones de tamaño [17].

Interconexiones e interfaces AXI

La comunicación entre PS y PL puede realizarse mediante un conjunto de 9 interfaces AXI, cada una compuesta por múltiples canales. Se encuentran representadas en la *Tabla 3*.

Nombres	Descripción	Master	Slaves
M_AXI_GP0	Propósito General (AXI_GP)	PS	PL
M_AXI_GP1		PS	PL
S_AXI_GP0	Propósito General (AXI_GP)	PL	PS
S_AXI_GP1		PL	PS
S_AXI_ACP	Accelerator Coherency Port (ACP), transacción de cache coherente.	PL	PS
S_AXI_HP0	Puertos de alto rendimiento (AXI_HP) con lectura/escritura en FIFOs.	PL	PS
S_AXI_HP1		PL	PS
S_AXI_HP2		PL	PS
S_AXI_HP3		PL	PS

***Tabla 3** Interfaces entre PS y PL* [8]

Interfaces EMIO

Muchas conexiones desde el PS se pueden enrutar a través de la PL a interfaces externas. El EMIO involucra la transferencia de señales entre los dos dominios, llevada a cabo a través de un conjunto de cables de conexión; sin embargo, no todas las interfaces MIO son soportadas en EMIO y algunas que sí lo son tienen una capacidad reducida. Las conexiones están organizadas en 2 bancos de 32 bits[13]. En la *Ilustración 10* se muestra el uso de dicha interfaz.

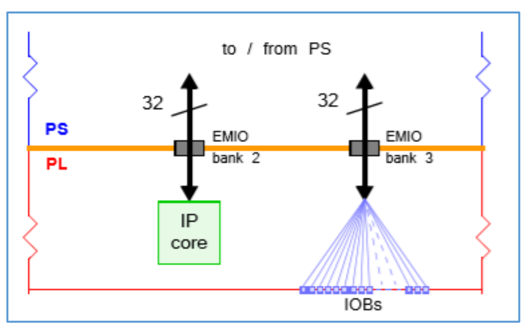


Ilustración 10 *Uso de EMIO como interfaz entre PS y PL* [8]

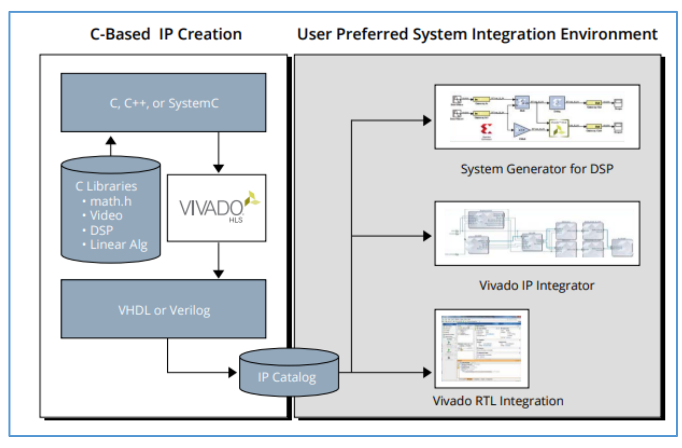
2.2.4 Jupyter Notebook

Jupyter Notebook es una aplicación web para crear y compartir documentos computaciones. Ofrece una experiencia sencilla, racionalizadas y centrada en los documentos. Entre sus principales características podríamos destacar las siguientes:

- Compatible con más de 40 lenguajes de programación como Python, R, Julia o Scala.
- Facilidad para compartir Notebooks por múltiples plataformas como email, Dropbox o GitHub entre otros.
- Salidas ricas e interactivas: HTML, imágenes, vídeos, LaTeX y tipos MIME personalizados.
- Herramientas de *big data* como Apache Spark o TensorFLow.

2.3 Herramientas de diseño

En esta sección se describe las herramientas Xilinx utilizadas, así como los entornos de co-diseño y el flujo de diseño hardware-software empleado en este proyecto. En la Ilustración 11 se muestra el flujo general utilizado por Xilinx.



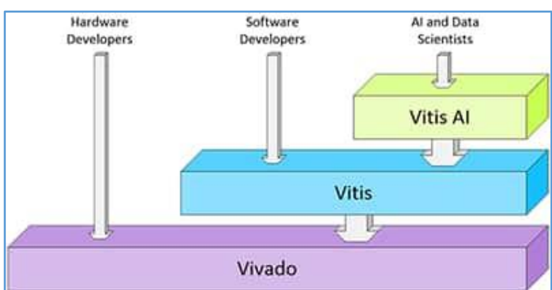


Ilustración 11 *Flujo de co-diseño*

El primer paso que se debe de tomar en este flujo de diseño es dividir lo que se implementa en hardware y lo que se hace en software (HW/SW *Partition*). Una vez decidido, cada diseño se realiza de forma secuencial. En el caso de la implementación Hardware, se desarrollaría los bloques hardware necesarios y que no estén incluidos in las librerías por defecto que ofrece Xilinx. Para ello, tradicionalmente, se ha seguido una metodología RTL. Esto es, desarrollando una descripción a nivel RTL en un lenguaje de descripción de hardware. Esta descripción debe ser sintetizable y permite crear los esquemáticos equivalentes.

Por otro lado, Vivado incluye una herramienta más potente, Vivado HLS (actualmente también conocido como Vitis HLS) que permite la elaboración de dichos bloques utilizando lenguajes de alto nivel como C/C++ que posteriormente se convierten a una descripción RTL sintetizable.

Como último paso, a nivel Hardware, sería la elaboración de la plataforma Hardware incluyendo los bloques creados con anterioridad, así como los necesarios para la conexión Hardware-Software y el procesador. Para este paso, se utiliza Vivado IP (IP Integrator).

Para el diseño Software, Xilinx proporciona el entorno SDK, que permite tanto el diseño de los drives para los periféricos como el desarrollo de la aplicación o programa que se ejecuta en el procesador.

En los siguientes puntos, se describe de una forma más detallada los programas utilizados en este proyecto: Vitis HLS (nueva versión de Vivado HLS) y Vivado IP.

2.3.1 Vivado IDE

Vivado IP Integrador IDE proporciona un flujo de desarrollo de diseño gráfico o basado en Tcl. Trabajando a nivel de interfaz, los equipos de diseño pueden ensamblar rápidamente sistemas complejos aprovechando IP creadas con Vitis HLS, por ejemplo. Entre las funciones integradas podemos destacar: Integración de IPs, simulación (comportamiento, funcional y temporal), síntesis RTL, generación del fichero *Bitstream* o diferentes análisis.

Vivado ofrece un navegador de flujo que proporciona acceso a las diferentes herramientas necesarias para el diseño total. A medida que se van completando los procesos, se irán actualizando los datos de diseño y añadiendo más información relevante. Las herramientas más destacables son:

a) **Project Manager**: Nos permite acceder a los ajustes del proyecto, así como al lenguaje. Por otro lado, es la herramienta que permite añadir nuevos archivos y acceder al catálogo de IPs [18], [19]
b) **IP *Integrator***: Crear, abrir o generar un diseño de bloques. [20]
c) ***Simulation***: Herramienta para simular [21]
d) **RTL *Analysis***: Abrir diseño elaborado, ejecutar verificaciones de reglas de diseño (DRCs) y generar un esquema RTL [19]

e) ***Synthesis***: Herramientas para sintetizar, así como elegir posibles configuraciones [22]
f) ***Implementation***: Cambiar la configuración de la implementación, diseño activo o abrir el diseño implementado [23]
g) ***Program* and *debug***: Cambiar la configuración del *bitstream*, generar un archivo *bitstream* o iniciar el analizador lógico [24]

2.3.2 Vitis HLS

Vitis HLS es una herramienta de síntesis de alto nivel que permite que funciones en C, C++ y OpenCL se implementen como IPs RTL Esto es, implementa *kernels* hardware en el contexto del flujo de desarrollo de aceleración de aplicaciones de Vitis [25].

En el flujo de aceleración de aplicaciones de Vitis, la herramienta Vitis HLS, antiguamente conocida como Vivado HLS, automatiza gran parte de las modificaciones de código necesarias para implementar y optimizar el código C/C++ en la lógica programable y lograr una baja latencia y un alto rendimiento. De forma adicional, también admite la personalización de su código para implementar diferentes estándares de interfaz u optimizaciones específicas para lograr los objetivos de diseño [25].

En la Ilustración 12 se muestra el flujo de diseño con la herramienta HLS.

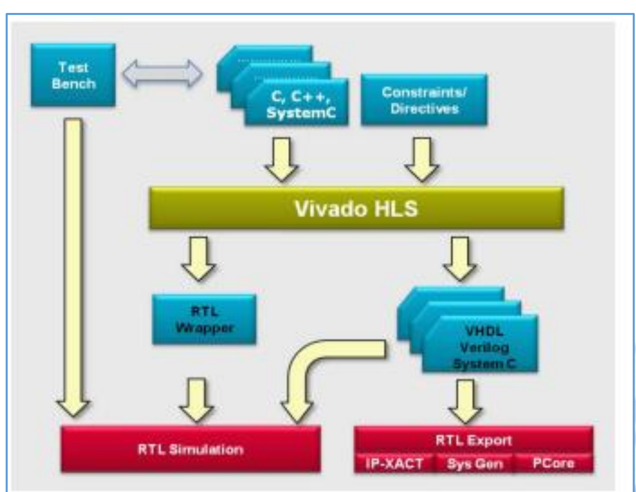


Ilustración 12 *Arquitectura y Flujo de diseño Vivado HLS*

El flujo de diseño con Vitis HLS se puede comprende:

1) Desarrollo, compilación, simulación y depuración del algoritmo en C/C++.

2) Síntesis del algoritmo C en un diseño RTL.
3) Verificación de la implementación de RTL mediante la simulación de RTL.
4) Empaquetamiento de la implementación en un extensión de archivo de objeto compilado (.xo) o IP de RTL.

La herramienta HLS sintetiza los distintos elementos del código como sigue [26]:

- Las funciones se sintetizan como bloques hardware en la jerarquía RTL. Si el código C incluye una jerarquía de subfunciones, el diseño RTL final incluye una jerarquía de módulos o entidades que tienen una correspondencia con la jerarquía de la función original. Por defecto, los distintos bloques hardware no operan en paralelo. Usando directivas de optimización puede controlarse la jerarquía de diseño resultante y forzar la operación concurrente de los distintos bloques.
- Los argumentos de las funciones se sintetizan en puertos de entrada/salida del correspondiente bloque hardware. Para descripciones C y C++ está síntesis de la interfaz se realiza automáticamente. Es decir, se sintetizan no sólo las operaciones del algoritmo sino el hardware necesario para la lectura y escritura de los puertos.
- Los bucles en las funciones se mantienen "enrollados" por defecto. La síntesis crea la lógica para una iteración del bucle, y el diseño RTL ejecuta esta lógica para cada iteración del bucle secuencialmente. Una iteración no comienza hasta que haya terminado la anterior. Usando directivas de optimización se pueden "desenrollar" parcial o totalmente los bucles o forzar a que una iteración no tenga que esperar a que acabe la anterior para comenzar. En ambos casos se está paralelizando las operaciones asociadas al bucle correspondiente. Esto se traduce en una disminución de la latencia e incremento del rendimiento del módulo sintetizado, lo que permite que varias o todas las iteraciones ocurran en paralelo.
- Los *array*s se sintetizan como memorias. Por defecto se mapean a memorias RAM, aunque utilizando directivas se puede forzar el uso de registros o FIFOs y así adecuar la arquitectura de memoria al algoritmo con el objetivo de evitar el cuello de botella que muchas veces supone el acceso a memoria.

2.3.3 Matlab

De forma independiente a las herramientas que proporciona Xilinx, se ha hecho uso de la aplicación Matlab, una plataforma de programación y cálculo numérico muy potente utilizada para el análisis de datos, desarrollo de algoritmos y creación de modelos [27]

En este TFM, se ha hecho uso de esta herramienta para el desarrollo y evaluación del algoritmo que posteriormente se implementa en hardware para el reconocimiento de imágenes de la base de datos MNIST.

3. Desarrollo e implementación

En este capítulo se describe el diseño del sistema desarrollado.

Recordemos que nuestra aproximación para la clasificación se basa en el uso de un algoritmo del "vecino más próximo". La selección de las referencias con las que comparar es crítica. Comenzaremos mostrando como la aplicación de otro algoritmo clásico, el *K-means,* permite obtener soluciones satisfactorias.

3.1 Desarrollo y evaluación del algoritmo *K-means*

Como se ha mencionado anteriormente, la base de este proyecto es el conjunto de datos MNIST, y el primer paso del proyecto fue la generación de un conjunto de patrones (imágenes) de referencia, aquel que se utilizará para clasificar las futuras muestras usando el algoritmo del vecino más próximo. Utilizar la totalidad del *set* de entrenamiento (60000 imágenes) no es competitivo en términos de tiempo de inferencia.

Para la generación de dicho *set*, se ha hecho del uso del algoritmo *K-means*. Este algoritmo es un método de agrupamiento, que particiona un conjunto de datos de *n* observaciones en *K* grupos, de forma que cada observación pertenece al grupo cuyo valor medio es más cercano. La Ilustración 13 describe el funcionamiento de este algoritmo.

El funcionamiento del algoritmo *K-means* se puede resumir en los siguientes pasos:

1. Se generan K puntos al azar, como muestra la Ilustración 13.b que serán designados centroides del agrupamiento.
2. Se forman tantos grupos como centroides haya, y se asigna individualmente cada dato al grupo cuyo centroide sea más cercano (13.c) aplicando una métrica de distancia.
3. Una vez obtenidos los grupos, se genera un nuevo punto localizado en el centro de todos los puntos de un mismo grupo (13.d). En el caso de este proyecto, generamos una nueva imagen compuesta por la media de todas las imágenes de un mismo grupo.

4. Se vuelven a realizar los pasos 1 a 3 (13.e-13.f) usando ahora los centros calculados como los puntos generadores de los nuevos grupos. El proceso se repite hasta que los grupos y centros permanezcan estables o se alcance un determinado número de iteraciones

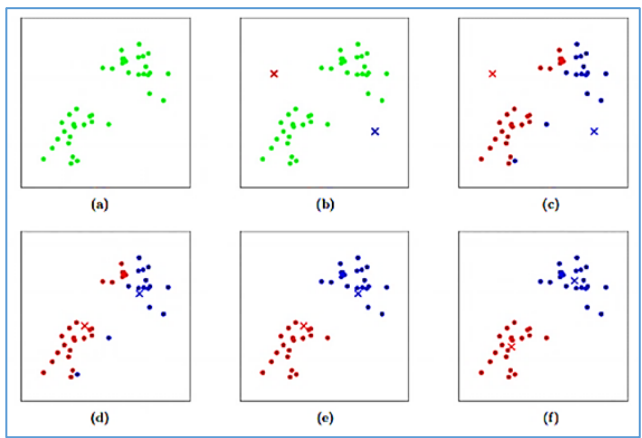


Ilustración 13 *Funcionamiento Algoritmo K-means*

Nuestra propuesta es aplicar esta técnica de particionamiento al conjunto de patrones de test de entrenamiento de cada dígito y tomar los correspondientes centros como los patrones de referencia para la posterior inferencia usando distancia mínima.[4]

El código del algoritmo utilizado en Matlab se muestra como Código 1. Este código recibe como entrada tres parámetros: *clúster* que indica el número de centros y grupos se van a generar, *numIte* limita el número de iteraciones a repetir la función máximo y por último *M,* que es el conjunto de muestras o puntos a particionar.

```
function M_out = Kmean_grises(cluster,numIte,M)

%% Elección centros aleatorios
centros = [];
tam = size(M,3);

for k = 1:cluster
    r = randi([1 tam],1,1);
    centros(:,:,k) = M(:,:,r);
end

Distancia_k = zeros(size(M,3),cluster+2);
antcentros = [];
iter_sin_repe = 0;

for i = 1:numIte
    M_r = M;
    %% Extraemos las imagenes usadas como centro
    if i == 1
        for k = 1:cluster
            au = find(all(all(bsxfun(@eq,M_r,centros(:,:,k)))));
            M_r(:,:,au) = [];
        end
    end
    for j = 1:size(M_r,3)
      for k = 1:cluster
          Distancia_k(j,k) = midistancia(M_r(:,:,j),centros(:,:,k));
      end
      [dis, cn] = min(Distancia_k(j,1:cluster));

      Distancia_k(j,cluster+1) = cn;
      Distancia_k(j,cluster+2) = dis;
      Distancia_k(j,cluster+3) = indicereal(j);
      Distancia_k(j,cluster+4) = j;
   end
   antcentros = centros;
   for k = 1:cluster
       A = (Distancia_k(:,cluster+1) == k);
       Bs = M_r(:,:,A);
       centros(:,:,k) = sum(Bs,3)/size(Bs,3);
   end
   %% Comrprobamos si han cambiado los centros
   arb = (centros == antcentros);
   con = sum(sum(sum(arb,3)));
   if i ~= 1 && con == (28*28*cluster)
        iter_sin_repe = iter_sin_repe + 1;
   end
   if iter_sin_repe == 10
       break
   end
end
M_out.centros = centros;
M_out.distancia = Distancia_k;
M_out.acaba = i;
end
```

Código 1 *Algoritmo K-means*

```
function [distancia] = midistancia(M,Mb)
    A1 = reshape(M,784, []);
    A2 = reshape(Mb,784, []);
    distancia = pdist2(A1',A2');
end
```

Código 2 *Código Distancia Euclidiana*

La base de este algoritmo se basa en la función "midistancia" (se ilustra como Código 2), que calcula la distancia Euclídea (recordemos que MNIST está compuesto por imágenes en grises) de cada imagen a los centros seleccionados y adjudicándola a aquel grupo cuyo centro esté a menor distancia.

El flujo de operación en nuestro caso se resume en:

- Inicialmente se selecciona aleatoriamente K imágenes de cada subconjunto del *set* de entrenamiento de MNIST. Cada subconjunto se corresponde con las imágenes asociadas a un determinado dígito.
- El resto de las imágenes de cada dígito se clasifican en grupos respecto a sus centros.
- Se genera nuevos centros, a partir de la media de todas las imágenes perteneciente a un mismo grupo.
- Se realiza los pasos anteriores hasta llegar al límite de iteraciones o hasta que el proceso converja.
- Los conjuntos finales de centros para cada dígito forman los patrones de referencia para la fase de inferencia.

En la Ilustración 14 se muestran los patrones de referencia obtenidos aplicando el algoritmo con $K = 2$ y número de iteraciones igual a 200 a partir del *set* de entrenamiento formado por 60.000 imágenes.

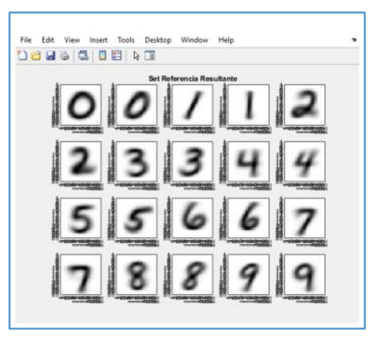


Ilustración 14 *Resultado Algoritmo K-means* K= 2

En la Tabla 4 se reportan los resultados de *accuracy* para distintos *set*s de referencia. Estos han sido generados con distintos valores de *K (Kmean en la tabla)*. Esto es, se evalúa para las 10.00 imágenes de test del MNIST qué fracción se infiere correctamente. Obsérvese que los resultados se detallan para cada uno de los dígitos. Se indica el número de patrones de test con la etiqueta correspondiente a cada dígito (Totales) y el número de los reconocidos correctamente (Aciertos).

Kmean = 2										
índices	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Aciertos	866	1116	863	853	807	720	884	876	792	836
Totales	980	1135	1032	1010	982	892	958	1028	974	1009
									Aciertos Totales	8613
									Imágenes Totales	10000
Kmean = 4										
índices	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Aciertos	930	1121	898	891	856	762	910	894	821	846
Totales	980	1135	1032	1010	982	892	958	1028	974	1009
									Aciertos Totales	8929
Kmean = 10										
índices	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Aciertos	950	1123	953	907	879	810	920	936	876	878
Totales	980	1135	1032	1010	982	892	958	1028	974	1009
									Aciertos Totales	9232
Kmean = 20										
índices	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Aciertos	954	1127	967	932	901	828	933	943	899	919
Totales	980	1135	1032	1010	982	892	958	1028	974	1009
									Aciertos Totales	9401

Tabla 4 *Accuracy de diferentes sets de referencias obtenidos con valores distintos de K*

Observamos que, como era de esperar, al aumentar el número de grupos y centros, y por tanto los patrones de referencia con los que se comparan los patrones de test, la clasificación de las imágenes de test mejorará.

Es interesante comparar los *accuracy* obtenidos con estas selecciones de las referencias con las obtenidas con conjuntos de igual cardinalidad, pero seleccionados aleatoriamente. En la Ilustración 15, se muestran estos resultados. Entre los *set*s de referencia mostrados podemos destacar, por ejemplo, el formado por *set* de entramiento completo (60.000) y como solo consigue una *accuracy* del 96.91% de las imágenes del *set* de test (10.000). A su izquierda, con un *set* de referencia con la mitad de integrantes solo se pierda un 0.70% de *accuracy*. Hay un cierto límite, donde el aumentar el número de imágenes en el *set* de referencia repercute muy pobremente en mejorar el *accuracy*. Más interesante es hacer notar que con sólo 40 patrones de referencia elegidos con el algoritmo K-means se obtiene un *accuracy* (89.42%) mejor que con 999 patrones aleatorios (87.3%).

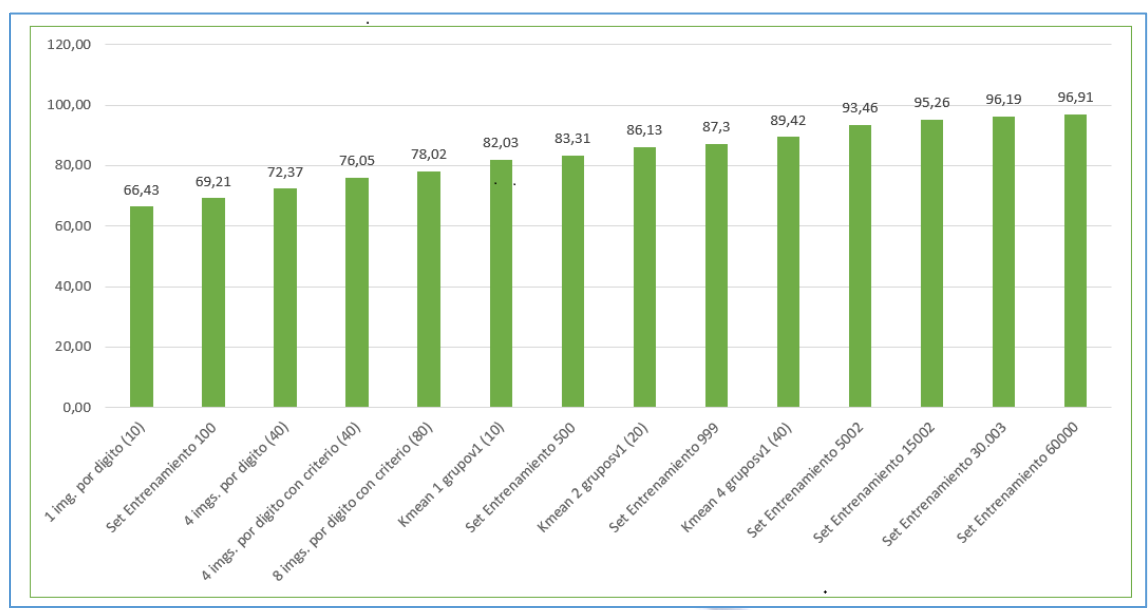


Ilustración 15 *Resultados obtenidos por diferentes sets de referencias generados*

No obstante, usar la distancia Euclídea, que es un cálculo computacionalmente costoso, en la inferencia hardware, puede ser lento o consumir muchos recursos. Ya se ha mencionado que, inspirados por los resultados de redes neuronales binarias que trabajan con las imágenes del MNIST binarizadas, nuestro propósito es usarlas también y poder así aplicar distancia Hamming en lugar de Euclídea. La distancia Hamming no es más que el número de posiciones (pixeles) en el que dos vectores binarios (imágenes blanco y negro) difieren. Comenzamos por evaluar a nivel de algoritmo cómo es la clasificación si se utilizan imágenes binarias en lugar de grises.

Para ello, binarizamos las imágenes del MNIST utilizando el código mostrado como Código 3:

```
if only_bin
    M0_gray = M0;
    for i=1:length(r_im_fil)
        imr = M0(:,:,i);
        thr_val = graythresh(imr);
        M0(:,:,i) = imr >= thr_val;
    end
end
```

***Código 3** Binarización Set MNIST*

El proceso de creación de patrones de referencias mediante *K*-means (Código 4) es casi idéntico al utilizado por las imágenes en escalas de grises. La diferencia se basa en la función de distancia utilizada, en este caso se utiliza la distancia Hamming, y en la forma de generación de nuevos centros mediante la moda en vez de la media.

```
function M_out = Kmean_bi(cluster,numIte,M)

% Primero elegimos los centros aleatoriamente.
centros = [];
tam = size(M,3);
activ = 0;
for k = 1:cluster
    r = randi([1 tam],1,1);
    centros(:,:,k) = M(:,:,r);
end
Distancia_k = zeros(size(M,3),cluster+2);
antcentros = [];
iter_sin_repe = 0;
antiguocluster = 0;
sas = 0;
for i = 1:numIte
    M_r = M;
    % Quitamos la imagenes que son utilizadas como centro
    if i == 1
        for k = 1:cluster
            au = find(all(all(bsxfun(@eq,M_r,centros(:,:,k)))));
            M_r(:,:,au) = [];
        end
    end
    if size(centros,3) ~= cluster
        res = cluster - size(centros,3);
        for p = 1:res
            aux = M(:,:,randi([1 tam],1,1));
            centros(:,:,size(centros,3)+1) = aux;
```

```
        end
      end
      for j = 1:size(M_r,3)
        for k = 1:cluster
              [DC,him] = DC_metric(M_r(:,:,j),centros(:,:,k));
              Distancia_k(j,k) =DC;
        end
        %Al usar correlación tiene que ser el maximo el más cercoano
        [dis, cn] = max(Distancia_k(j,1:cluster));
        Distancia_k(j,cluster+1) = cn;
        Distancia_k(j,cluster+2) = dis;
        Distancia_k(j,cluster+3) = indicereal(j);
        Distancia_k(j,cluster+4) = j;
    end
    antcentros = centros;
    for k = 1:cluster
        A = (Distancia_k(:,cluster+1) == k);
        Bs = M_r(:,:,A);
        centro_auxiliar = mode(Bs,3);
        if sum(sum(isnan(centro_auxiliar))) == 784
           if sum(A) == 0
               r = round((cluster-1)*rand(1,1) + 1);
               A = (Distancia_k(:,cluster+1) == r);
               Bs = M_r(:,:,A);
               b = size(Bs,3);
              a = 1;
              r = round((b-a)*rand(1,1) + a);
              centro_auxiliar = Bs(:,:,r)
           else
              b = size(Bs,3);
              a = 1;
              r = round((b-a)*rand(1,1) + a);
              centro_auxiliar = Bs(:,:,r);
           end
        end
        centros(:,:,k) = centro_auxiliar;
    end
    arb = (centros == antcentros);
    con = sum(sum(sum(arb,3)));
    if i ~= 1 && con == (28*28*cluster)
         iter_sin_repe = iter_sin_repe + 1;
    end
    if iter_sin_repe == 10
        break
    end
end
M_out.centros = centros;
M_out.distancia = Distancia_k;
M_out.acaba = i;
end
```

Código 4 *K-means binario*

El cálculo de esta distancia es mucho más simple y menos costosa computacionalmente que la distancia Euclídea. Se ha implementado este cálculo de la distancia mediante el producto escalar o el cálculo de los cosenos de dirección (Direction Cosine, DC). Esta métrica indica la similitud entre dos vectores, siendo 1 en caso de ser idénticos y -1 si

uno es la versión negativa o complementada del otro. De esta forma, puede obtenerse la distancia Hamming como $\frac{(1-DC)}{2} * n_pixel$ para calcular el número de píxeles de diferencia entre ambos vectores bipolares. Como se mostrará en secciones posteriores esta operación se puede implementar de forma eficiente en lógica mediante la operación XNOR.

La función de distancia queda plasmada en el Código 5.

```
function [DC,Hdist] = DC_metric(M0,M1)
v0_norm = [];
v1_norm = [];
if ~isequal(size(M0,1)*size(M0,2),size(M1,1)*size(M1,2))
   DC = [];
   Hdist = [];
   return
end
for n_var = 1:2
   if n_var==1, M=M0;
   else M=M1; end

   v_norm = zeros(size(M,1)*size(M,2),size(M,3));

   if isequal(unique(M(:)),[0;1])
      M_n = 2*M - ones(size(M));
   elseif isequal(unique(M(:)),[-1;1])
      M_n = M;
   elseif max(max(max(abs(M)))) > 1
      mp = min(min(min(M)));
      Mp = max(max(max(M)));
      M_n = (M-mp)/(Mp-mp);
      % v_tn = v_tn >= graythresh(v_tn);
      M_n  = 2*M_n - ones(size(M_n));
   else
      M_n  = M;
   end
   [n,m,p]=size(M_n);
   a=reshape(M_n,n,[],1);
   b=reshape(a(:),n*m,[])';
   v_norm = b';
   if n_var==1
      v0_norm = v_norm;
   else
      v1_norm = v_norm;
   end
end
aux = (v0_norm'*v1_norm);
DC = aux/size(v0_norm,1);
Hdist = (size(v0_norm,1)*(1-DC)/2);
end
```

Código 5 *Función Distancia Hamming*

En la Tabla 5 se reportan el *accuracy* obtenido con la aproximación binaria. De nuevo se han explorado distintos valores de K y los resultados se detallan por dígito.

Kmean = 2										
índices	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Aciertos	892	1122	805	806	711	610	867	872	771	808
Totales	980	1135	1032	1010	982	892	958	1028	974	1009
								Aciertos Totales	8264	
								Imágenes Totales	10000	
Kmean = 4										
índices	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Aciertos	949	1118	873	844	744	726	904	901	800	858
Totales	980	1135	1032	1010	982	892	958	1028	974	1009
								Aciertos Totales	8717	
Kmean = 15										
índices	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Aciertos	965	1116	945	896	867	773	920	920	867	886
Totales	980	1135	1032	1010	982	892	958	1028	974	1009
								Aciertos Totales	9155	
Kmean = 30										
índices	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Aciertos	967	1113	958	910	876	789	929	934	892	919
Totales	980	1135	1032	1010	982	892	958	1028	974	1009
								Aciertos Totales	9287	
Kmean = 100										
índices	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Aciertos	967	1126	977	934	904	811	937	960	897	950
Totales	980	1135	1032	1010	982	892	958	1028	974	1009
								Aciertos Totales	9463	

Tabla 5 Accuracy diferentes sets de referencias binarios con diferentes k

Comparando esta tabla y la anterior, referida a los conjuntos grises, se puede afirmar que la binarización no supone una pérdida relevante en el *accuracy*. Lógicamente, para un mismo número de imágenes de *set*, la codificación en escala de grises presenta mejor rendimiento, sin embargo, los binarios lo compensan con una velocidad de procesado mayor y un uso reducido de recursos. Por ejemplo, para almacenar una imagen en gris se necesita una matriz de 28x28 con un tamaño de cada elemento de 8 bits, mientras que en las imágenes binarias con 1 bit por elemento es suficiente.

En la Ilustración 16 se representa gráficamente los resultados de *accuracy* obtenidos para diferentes configuraciones del *set* de referencia binarios (B) y grises (G), así como diferentes tamaños (28x28 y 10x10), obtenidos por K-means, El último dígito de la leyenda especifica el número total de centroides (K) empleado. Además, se han añadido dos conjuntos aleatorios con 10 y 5000 patrones. En este caso se trabaja en tamaño 28x28 y con grises.

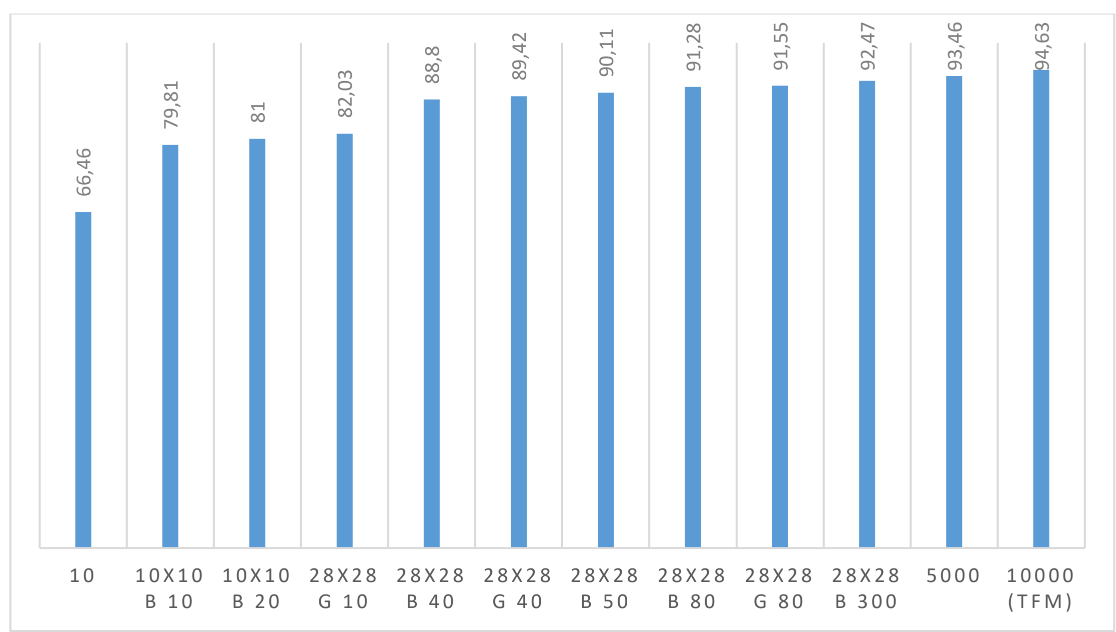


Ilustración 16 *Resultados obtenidos por diferentes sets de referencia*

En la gráfica anterior podemos apreciar que el uso del algoritmo de *k-means* proporciona un aumenta en el número de reconocimiento. Con un *set* de referencia de 5000 imágenes seleccionado de forma aleatoria se consigue un 93.46% mientras, que un *set* de referencia de 1000 imágenes generado mediante el algoritmo *K-means* logra un 94.63%.

Hay que puntualizar que, a pesar de la idea preconcebida, un mayor número de imágenes en *set* de referencia no implica un mayor *accuracy*. Existe un límite donde la mejora no es apreciable. Por ejemplo, como se ha mencionado, para 1000 se consigue un 94.63 mientras para 2000 se logra un 94.87. Aumentando 1000 imágenes el *set* de referencia solo se obtiene una mejora del 0.24. Esta saturación del incremento del *accuracy* es normal, una superpoblación de centros no tiene necesariamente porqué atrapar los casos más alejados, incluso puede existir imágenes que sean excesivamente difíciles de clasificar.

En el siguiente apartado se describe el diseño del acelerador hardware que dada una imagen binaria de entrada (patrón), identifica el patrón de referencia a menor distancia Hamming.

3.2 Diseño del acelerador hardware

El acelerador se ha diseñado usando HLS. La descripción C++ de su funcionalidad se muestra en el Código 7 y en el Código 8. El Código 6 corresponde al archivo .h, donde se define el número de elementos que compone nuestro *set* de referencia (*N*), parámetros relacionados con cómo se recibe y cómo se guarda en memoria interna la imagen a clasificar (*M* y *L*) y el número de imágenes que forman cada dígito del *set* de referencia (*N_p*).

Por otro lado, también está las definiciones de las funciones a utilizar, y de determinados tipos de datos. En particular, se declara el tipo *axis_t* que se usa para los argumentos de la función que se sintetiza.

```
#ifndef _MMULT_
#define _MMULT_

#include "ap_axi_sdata.h"
#include "ap_int.h"
#include <inttypes.h>
#include "hls_stream.h"

#define N 1000
#define M 13
#define L 64
#define N_p  100

#define DWIDTH 64
typedef ap_axiu<DWIDTH, 0, 0, 0> axis_t;

typedef long long unsigned int DataType;

void mifuncion(hls::stream<axis_t> &ent, hls::stream<axis_t> &sal);
template <typename T> void kernel_funcion(T I_A[N], T I_B[N]);

#endif
```

Código 6 *Librería propia HLS. Archivo principal.h*

El Código 7 corresponde a la función **kernel_función.** Esta es el núcleo del acelerador y se encarga de calcular la distancia Hamming de la imagen a evaluar con cada una de las que componen el *set* de referencia seleccionado. Como resultado devuelve la imagen del *set* de referencia más cercana y el dígito al que corresponde.

Los argumentos de la función son dos *arrays* que corresponden a la imagen de entrada y la de salida. Los *arrays* son de 13 elementos de 64 bits ((aplicando las declaraciones de *M* y *DataType* de “principal.h”). Esto hace un total de 832 bits ya que los 784 (28x28) bits de cada imagen se completan con ceros hasta alcanzar un múltiplo de 64.

El *array set*_referencia almacena las *N* imágenes de referencia. Cada una de ellas compuesta también por 13 elementos de 64 bits.

```
#include "principal.h"

template <typename T> void kernel_funcion(T I_A[M], T I_B[M]) {

        DataType set_referencia[N][M]={
                        #include "set_referencia_1000.dat"
                                        };
        DataType vec_xnor[N][M] ;
        short aux1 =0;
        short max  =50;
        int indice = 0;
        Bucle1:for(int a = 0; a < N; a++){
                aux1=0;
                Bucle2:for(int c = 0; c < M; c++){
                        vec_xnor[a][c] = ~(I_A[c]^set_referencia[a][c]);
                        Bucle3:for(int b = 0; b < L; b++){
                                aux1 += ((vec_xnor[a][c] >> b) & 1);
                        }
                }
        if(max < aux1){
                max = aux1;
                indice = a;
        }
        }
        int indice_real = 0;
        Bucle4:for(int j = 0; j < 10; j++){
                if ((j)*N_p < indice & indice < (j+1)*N_p){
                        indice_real = j;
                }
        }

        Bucle5:for(int a = 0; a < M; a++){
                if (a == M-1){
                        I_B[a] = set_referencia[indice][a] + indice_real;
                }else{
                        I_B[a] = set_referencia[indice][a];
                }
        }
  return;
}
```

Código 7 *Función distancia entre imágenes*

Debido a que estamos trabajando con imágenes binarias, el cálculo de las distancias es muy sencillo (Bucle 1). Se resumen en una operación XNOR (Bucle 2) y un conteo de los bits activo (Bucle 3). Una vez comparado la imagen con el de referencia se calcula el máximo, es decir el que tiene más elementos en común. Los Bucles 4 y 5 calculan el índice real de la imagen correspondiente. Como resultado, la función devuelve la imagen del *set* referencia que es más cercana y el índice del dígito al que corresponde, el cual se encuentra en los 8 bits menos significativos de la última fila del vector.

El Código 8 corresponde a la **función principal**. Esto es, a la que se sintetiza y desde la que se llama a la anteriormente descrita. Esta función recibe y envía en modo *stream* datos de 64 bits. Cómo ya se ha explicado, al estar trabajando con imágenes binarias 28x28, esta se traduce en 784 bits. El múltiplo más cercano de 64 a 784 es 832, por lo que cada imagen (tanto la entrada evaluar como las del *set* de referencia) están representada en un *array* de 13 elementos con 64 bits cada una. Su función es la de recibir y guardar en memoria la imagen a clasificar (Carga), pasarla al kernel_funcion y devolver el resultado (Escritura).

```
void mifuncion(hls::stream<axis_t> &ent, hls::stream<axis_t> &sal) {
    #pragma HLS INTERFACE s_axilite port = return bundle = control
        #pragma HLS INTERFACE axis depth=13 port = sal
        #pragma HLS INTERFACE axis depth=13 port = ent

         DataType I_A[M] = {};
         DataType I_B[M] = {};
         int i_limit = M;
        Carga:
         for (int i = 0; i < i_limit; i++) {
           axis_t temp = ent.read();
                 I_A[i] = temp.data.to_uint64();
         }

         kernel_funcion<DataType>(I_A, I_B);

        Escritura:
         for (int j = 0; j < i_limit; j++) {
           axis_t temp;
           temp.data =  I_B[j];

           ap_uint<1> last = 0;
           if (j == i_limit - 1) {
             last = 1;
           }
           temp.last = last;
           temp.keep = -1; // enabling all bytes
           sal.write(temp);
         }

        }
```

Código 8 *Función Principal para HLS*

Las siguientes sentencias:

```
#pragma HLS INTERFACE s_axilite port = return bundle = control
#pragma HLS INTERFACE axis depth=13 port = sal
#pragma HLS INTERFACE axis depth=13 port = ent
```

son directivas de HLS. Estas concretamente son las que definen los puertos de entrada y salida (ent y sal) como *streams* con una profundidad de 13. Además, tenemos un pragma adicional que incorporará una interfaz adicional en nuestro bloque IP por el cual podremos controlarlo: encenderlo, apagarlo, etc.

Los resultados de la síntesis con Vitis HLS de este código con una frecuencia objetivo de 200MHz (5ns de periodo) se muestran en la Ilustración 17.

Target	Estimated	Uncertainty
5.00 ns	3.624 ns	1.35 ns

Performance & Resource Estimates

Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
mifuncion		-	21096	1,050E5	-	21097	-	no	64	2	1297	6078	0
Loop 1		-	13	65.000	1	1	13	yes	-	-	-	-	-
Loop 2		-	13	65.000	1	1	13	yes	-	-	-	-	-
Carga		-	13	65.000	1	1	13	yes	-	-	-	-	-
Bucle1		-	21000	1,050E5	21	-	1000	no	-	-	-	-	-
Bucle4		-	13	65.000	5	1	10	yes	-	-	-	-	-
Bucle5		-	16	80.000	5	1	13	yes	-	-	-	-	-
Escritura		-	14	70.000	3	1	13	yes	-	-	-	-	-

Ilustración 17 *Resultados síntesis de función principal. Set de referencia de cardinalidad1000*

Como observamos, el número de recursos utilizado es muy reducido y el resultado se obtiene en 21096 ciclos (*Latency*), lo que se traduce en: 105 microsegundos. La recepción y el envío consumen muy pocos ciclos y el grueso lo emplea en la computación. Hay que hacer notar que estamos haciendo una síntesis sin ninguna directiva de optimización. Todos los bucles están por tanto enrollados y las iteraciones se ejecutan secuencialmente.

Antes de pasar a la optimización del diseño, es interesante explorar el uso de recursos y el tiempo empleado con diferentes números de imágenes en el *set* de referencia. Los resultados se muestran en la Tabla 6.

#PATRONES	LATENCY	BRAM	DSP	FF	LUT
100	2194	8	1	1127	6012
200	4294	16	1	1139	6012
500	10593	32	1	1154	6019
1000	21093	64	1	1169	6030
2000	42094	128	1	1172	6033

Tabla 6 *Resultados Síntesis para distintas cardinalidades del set de referencia*

Como se observa a simple vista, el número de ciclos en resolver la función y el número de BRAMs aumentan proporcionalmente al número de imágenes por *set*. Esto es lo esperado. Sin embargo, se destaca que los recursos de DSP, FF y LUT no varían significativamente.

3.2.1 Optimizaciones

La naturaleza operacional de la función implementada nos permite aplicar técnicas de optimización propias de HLS sobre la función a sintetizar en hardware para reducir la latencia como es nuestro objetivo.

Tras una exploración del espacio de soluciones de diseño finalmente se han aplicado las siguientes directivas.

PRAGMA HLS PIPELINE

Este pragma reduce el intervalo de iniciación de una función o un bucle al permitir la ejecución simultánea de operaciones. Aplicarlo a un bucle permite que una iteración pueda comenzar antes de que acabe la anterior, como se muestra en la Ilustración 18, lo que posibilita reducir la latencia.

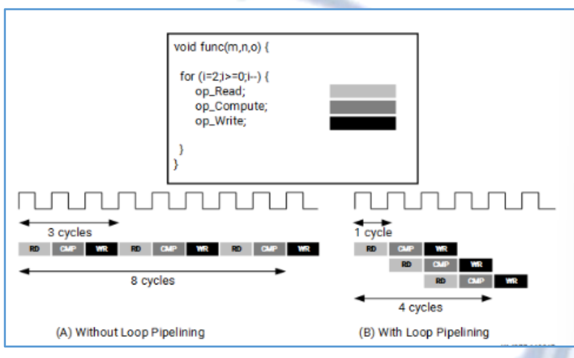


Ilustración 18 *Pragma Pipeline* [28]

Sin embargo, hay que tener cuidado cuando se utiliza sobre bucles que contienen bucles anidados, ya que al implementar este pragma desenrollaran los bucles internos lo que podría provocar un aumento crítico de los recursos.

PRAGMA *ARRAY*_PARTITION

Este pragma permite particionar un *array* en otros más pequeños o incluso en elementos individuales. El particionado resulta en un RTL con múltiples pequeñas memorias o múltiples registros en lugar de una memoria grande, lo que aumenta, efectivamente la cantidad de puertos de lectura y escritura para el almacenamiento, y mejora potencialmente la latencia y el rendimiento del diseño.

Vivado HLS presenta 3 tipos de partición:

- **Cyclic**: La partición cíclica crea arreglos más pequeños intercalando elementos del arreglo original. La matriz se particiona cíclicamente colocando un elemento en cada nueva matriz antes de volver a la primera matriz para repetir el ciclo hasta que la matriz se particione por completo. [29]
- **Block:** La partición de bloques crea matrices más pequeñas a partir de bloques consecutivos de la matriz original. Esto divide efectivamente la matriz en N bloques iguales[29]
- **Complete:** La partición completa descompone la matriz en elementos individuales. Para una matriz unidimensional, esto corresponde a resolver una memoria en registros individuales (por defecto)[29]

En la ilustración 19 se muestra el funcionamiento de cada método.

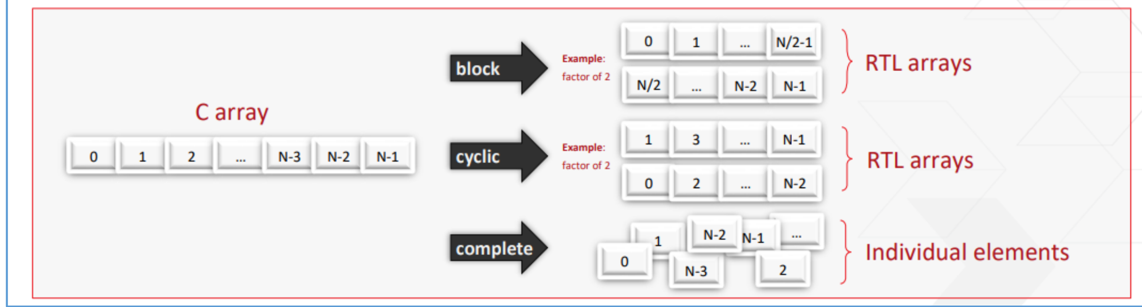
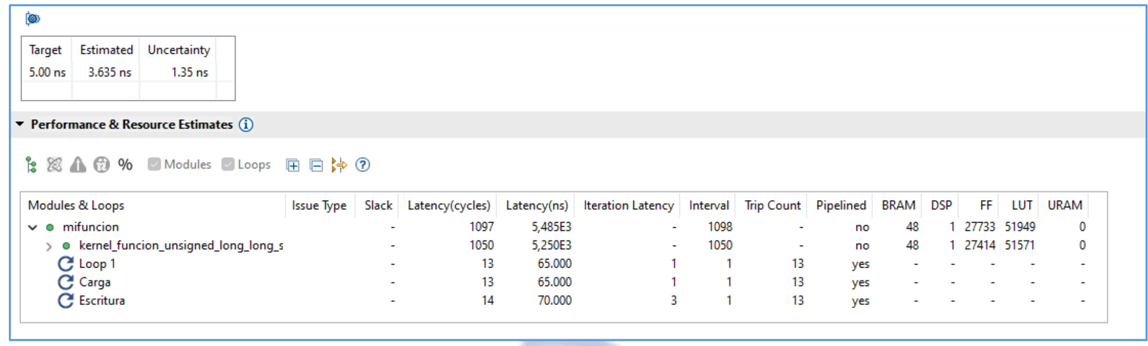


***Ilustración 19** Funcionamiento Pragma HLS Array_Partition* [29]

Una vez aplicado las dos optimizaciones mencionadas como se muestran en la Ilustración 20, obtenemos el resultado indicado en la Ilustración 21. Esto es, hemos aplicado un pipeline al bucle externo del cómputo más costoso. Con ello se ha forzado el desenrollado de los sus bucles anidados, incrementando el paralelismo en la computación. Además, para permitir aprovechar este paralelismo se han particionado tanto la memoria que almacena las imágenes de referencia, como la que guardaba los resultados de la operación XNOR.

```
kernel_funcion
    set_referencia
    HLS ARRAY_PARTITION variable=set_referencia complete dim=2
    vec_xnor
    HLS ARRAY_PARTITION variable=vec_xnor complete dim=2
    Bucle1
        HLS PIPELINE
        Bucle2
            Bucle3
    Bucle4
    Bucle5
```

Ilustración 20 *Directivas HLS empleadas*



Target	Estimated	Uncertainty
5.00 ns	3.635 ns	1.35 ns

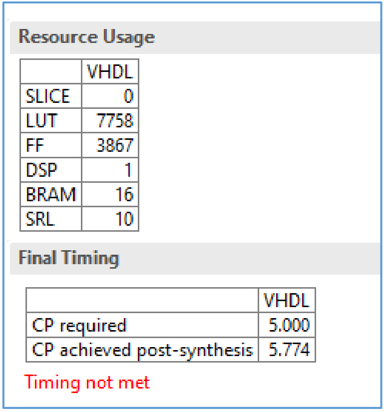
Performance & Resource Estimates

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
mifuncion		-	1097	5,485E3	-	1098	-	no	48	1	27733	51949	0
kernel_funcion_unsigned_long_long_s		-	1050	5,250E3	-	1050	-	no	48	1	27414	51571	0
Loop 1		-	13	65.000	1	1	13	yes	-	-	-	-	-
Carga		-	13	65.000	1	1	13	yes	-	-	-	-	-
Escritura		-	14	70.000	3	1	13	yes	-	-	-	-	-

Ilustración 21 *Informe síntesis HLS con optimizaciones y reloj 200MHz*

Comparando con la Ilustración 17, donde se muestran los resultados de la síntesis sin optimizaciones, se puede apreciar una gran mejora en cuanto a la latencia, pasando de 21096 iteraciones a 1097 iteraciones (5.5 microsegundos), aunque conlleva un aumento del uso de recursos como LUTs y FF. Sin embargo, estos son recursos estimados. Una vez se sintetice el diseño RTL se obtendrán los recursos reales.

En la Ilustración 22 muestran los resultados obtenidos tras esta síntesis con la inconveniencia del que nuestro reloj es demasiado rápido y hay violaciones de tiempo. La reducción de LUTs y FFs es muy significativa.



Resource Usage

	VHDL
SLICE	0
LUT	7758
FF	3867
DSP	1
BRAM	16
SRL	10

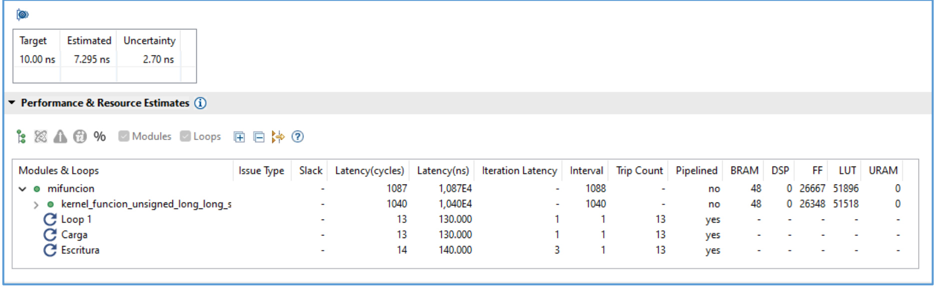
Final Timing

	VHDL
CP required	5.000
CP achieved post-synthesis	5.774

Timing not met

***Ilustración 22** Recursos obtenidos tras la síntesis lógica con reloj de 200 MHZ.*

Si utilizamos un reloj de 100 MHz en cambio, obtenemos la misma latencia en ciclos (Ilustración 23) y los mismos recursos tras la síntesis del RTL (Ilustración 24).

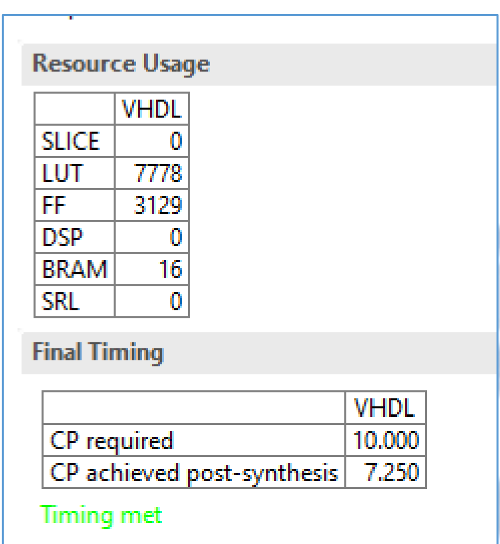


Target	Estimated	Uncertainty
10.00 ns	7.295 ns	2.70 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
mifuncion		-	1087	1,087E4	-	1088	-	no	48	0	26667	51896	0
kernel_funcion_unsigned_long_long_s		-	1040	1,040E4	-	1040	-	no	48	0	26348	51518	0
Loop 1		-	13	130.000	1	1	13	yes	-	-	-	-	-
Carga		-	13	130.000	1	1	13	yes	-	-	-	-	-
Escritura		-	14	140.000	3	1	13	yes	-	-	-	-	-

***Ilustración 23** Informe síntesis HLS con optimizaciones y reloj de 100 MHZ*



Resource Usage

	VHDL
SLICE	0
LUT	7778
FF	3129
DSP	0
BRAM	16
SRL	0

Final Timing

	VHDL
CP required	10.000
CP achieved post-synthesis	7.250

Timing met

***Ilustración 24** Recursos obtenidos tras la síntesis lógica con reloj de 100 MHZ*

3.3 Diseño del *Overlay*

Una vez finalizado el proceso de conversión a RTL ya se puede generar nuestro *Overlay*. Los módulos necesarios son los siguientes.

a) AXI DMA:

También conocido como AXI Direct Memory Access. Este bloque DMA permite transmitir datos desde la memoria, en este caso PS DRAM, a una interfaz AXI stream. Además, puede recibir datos por AXI Stream y escribirlos en PS DRAM.

El DMA tiene puertos AXI Master para el canal de lectura, y otro para el canal de escritura, y también se denominan puertos mapeados en memoria - pueden acceder a la memoria PS. Los puertos se denominan MM2S (Memory-Mapped to Stream) y S2MM (Stream to Memory-Mapped). Igualmente tiene un puerto AXI lite de control, que se utiliza para escribir instrucciones de control (start, stop, etc)[30], [31]

Hay dos puertos AXI Master que se conectarán a la DRAM. M_AXI_MM2S (canal de lectura) y M_AXI_S2MM (canal de escritura). Los maestros AXI pueden leer y escribir en la memoria. En este diseño se conectarán a los puertos Zynq HP (High Performance) AXI Slave[30], [31]

Hay dos puertos de flujo AXI del DMA. Uno es un flujo maestro AXI (M_AXIS_MM2S) y corresponde al canal de LECTURA. Los datos se leerán de la memoria a través del puerto M_AXI_MM2S y se enviarán al puerto M_AXIS_MM2S (y a la IP conectada a este puerto). El otro puerto de flujo AXI es un esclavo AXI (S_AXIS_S2MM). Este está conectado a su IP. El DMA recibe los datos del flujo AXI desde el IP, y los escribe de vuelta a la memoria a través del puerto M_AXI_S2MM[30], [31].

De esta forma la conexión que tendríamos:

- Datos PS-DMA:
  - M_AXI_MM2S: Lectura de la DMA al PS.
  - M_AXI_S2MM: Escritura del PS a la DMA.
- Datos Función-DMA:
  - S_AXI_MM2S: Lectura de la DMA a la función.
  - S_AXI_S2MM: Escritura de la función a la DMA.
- Control:

- S_AXI_LITE: Este puerto estará conectado al Processing System mediante un AXI *Interconnect*.

En la Ilustración 25 se muestra un esquema de los bloques que componen la DMA.

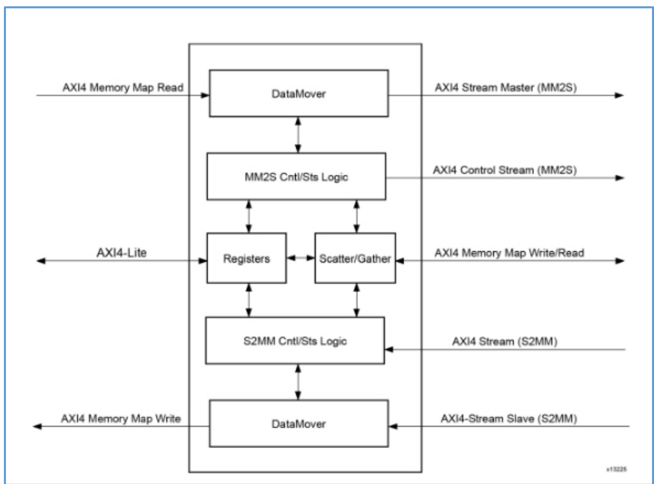


Ilustración 25 *Diagrama de bloques del DMA* [31]

En la Ilustración 26 se muestra la configuración interna del DMA del proyecto. Esta configuración establece por ejemplo un ancho de datos de lectura y escritura de 64 bits.

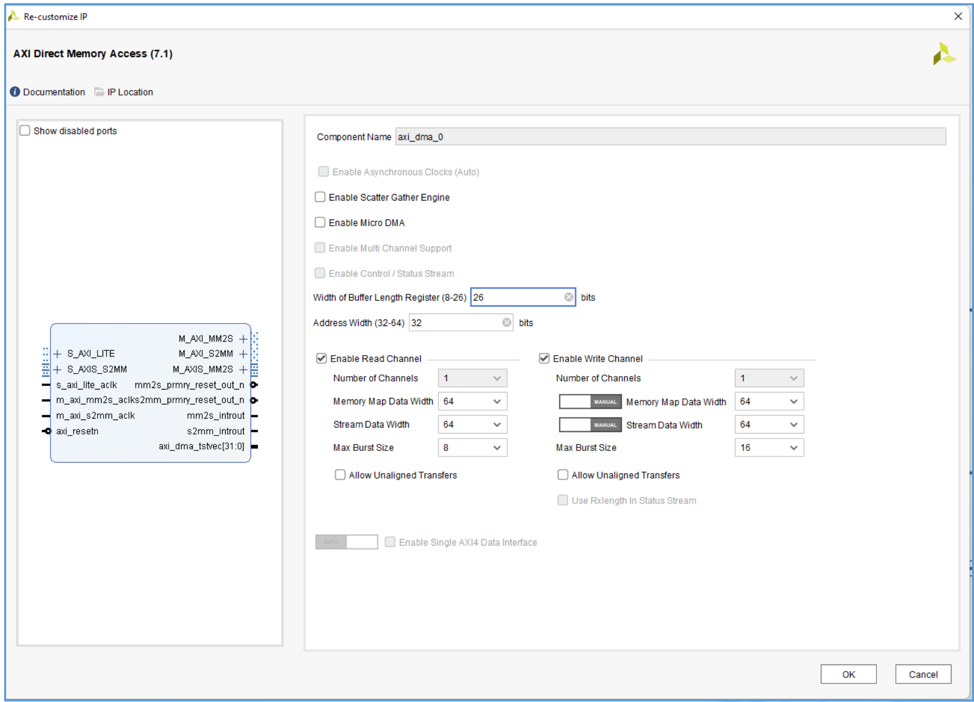


Ilustración 26 *Configuración DMA*

b) Acelerador

Bloque correspondiente a la función desarrollada en Vivado HLS. Como se ha mencionado los puertos de entrada y salida están conectados a la DMA, y el puerto de control (s_axi_control) a un AXI *Interconnect*.

c) Processing System:

IP que actúa como una conexión lógica entre el PS y el PL como muestra la Ilustración 27.

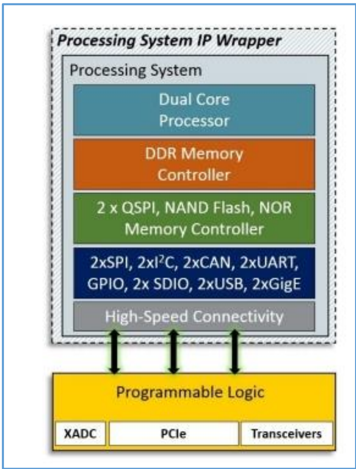


Ilustración 27 *Processing System IP Wrapper* [32]

Para este proyecto tenemos que implementar la siguiente configuración:

- Habilitar S_AXI_HP0 y S_AXI_HP2 con una anchura de 64 bits. Estas interfaces de alto rendimiento estarán conectadas con los puertos de entrada y salida de la DMA que conectan con el sistema.
- Habilitar una interfaz maestra tipo AXI que conectará con un AXI Interconnect conectado a la función y a la DMA. Esta interfaz nos permitirá, junto al bloque mencionado, controlar y configurar los IPs.
- Implementar un FCLK_CLK0 (reloj de todos nuestros bloques IP) de 100 MHz.

d) AXI Interconnect

Estos bloques son los encargados de conectar los distintos bloques al Processing System. En este proyecto existen 3:

- AXI_Interconnect_0: Conecta el Processing System con los puertos de control de la DMA y de la función.
- AXI_Interconnect_1: Conecta M_AXI_MM2S (lectura) de la DMA con el AXI HP0.
- AXI_Interconnect_2: Conecta M_AXI_S2MM (escritura) de la DMA con el AXI HP2.

e) Processor System Re*set*

Este módulo permite personalizar el diseño para que se adapte a la aplicación mediante el establecimiento de ciertos parámetros para habilitar/deshabilitar funciones. En este proyecto es el encargado de transmitir la señal de *reset* proveniente del PS IP al resto de módulos.

El *Overlay* completo se muestra en la Ilustración 28.

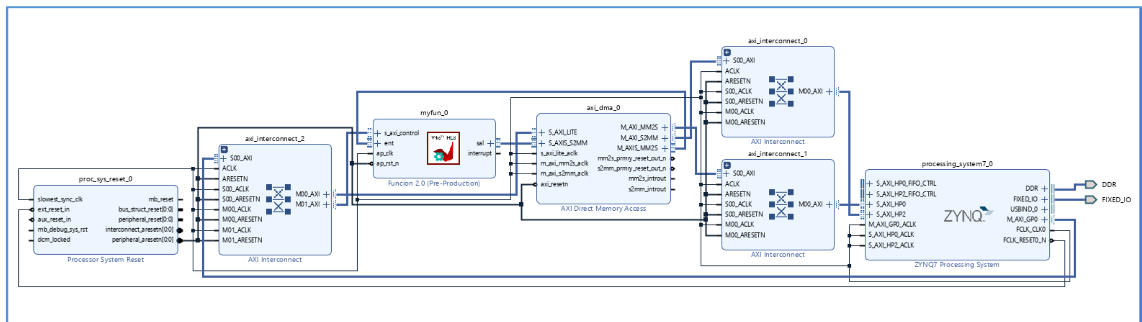


Ilustración 28 *Overlay diseñado*

3.4 Comparación con BNN

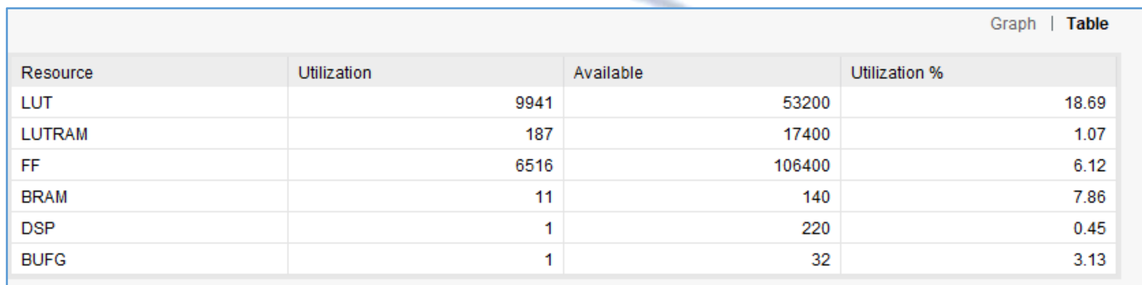
Una vez implementado y elaborado el *Overlay, estamos en condiciones de comparar con las BNNs mencionadas en el Capítulo 1.* Hemos elegido la más sencilla, una LFC de W1A1. En la Ilustración 29 se representan los recursos utilizados por la red LFC mientras en la Ilustración 30 la utilizada por el *Overlay* diseñado.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	28360	0	53200	53.31
LUT as Logic	25359	0	53200	47.67
LUT as Memory	3001	0	17400	17.25
LUT as Distributed RAM	2314	0		
LUT as Shift Register	687	0		
Slice Registers	33470	0	106400	31.46
Register as Flip Flop	33470	0	106400	31.46
Register as Latch	0	0	106400	0.00
F7 Muxes	393	0	26600	1.48
F8 Muxes	64	0	13300	0.48

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	110	0	140	78.57
RAMB36/FIFO*	6	0	140	4.29
RAMB36E1 only	6			
RAMB18	208	0	280	74.29
RAMB18E1 only	208			

Site Type	Used	Fixed	Available	Util%
DSPs	4	0	220	1.82
DSP48E1 only	4			

***Ilustración 29** Recursos utilizados por la red LFC de FINN* [33]



Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	9941	53200	18.69
LUTRAM	187	17400	1.07
FF	6516	106400	6.12
BRAM	11	140	7.86
DSP	1	220	0.45
BUFG	1	32	3.13

***Ilustración 30** Recursos utilizados por el Overlay diseñado*

Es apreciable que el *Overlay* diseñado emplea un número mucho menor de recursos que la red neuronal. Destacando sobre todo en BRAM. La Ilustración 31 muestra un diagrama comparativo de los recursos utilizados.

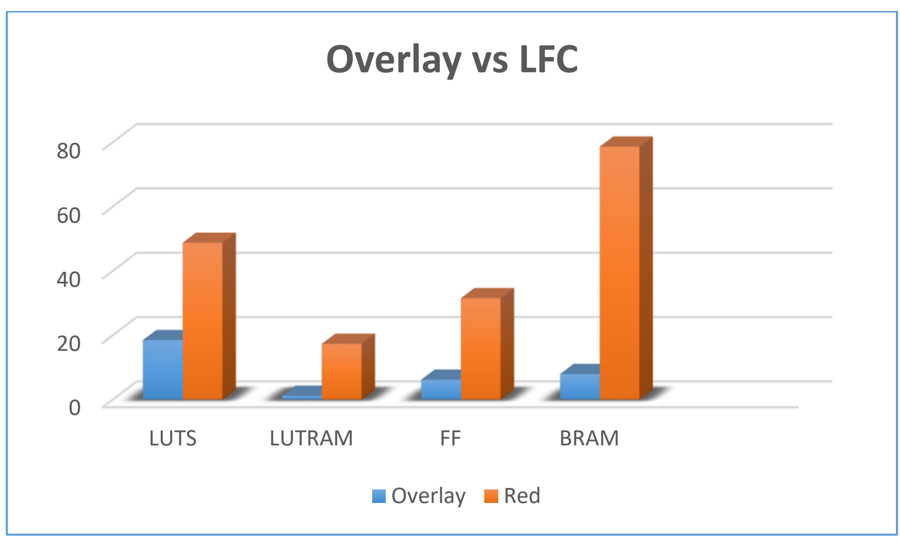


Ilustración 31 *Recursos Overlay vs LFC*

De acuerdo con el *accuracy*, según [1] la red neuronal LFC obtiene un 98.4% mientras con el *set* de referencia seleccionado para la aplicación se obtiene un 94.63%. A pesar de perder aproximadamente un 4% de *accuracy* se compensa con un menor uso de recursos.

En lo que respecta al tiempo de inferencia, nuestro diseño tiene una latencia de 1097 ciclos. Operando a 100MHz, equivale a 10.87 microsegundos. Este resultado es menos de la mitad del empleado por la LFC con una frecuencia de 200MHZ, para la clasificación de una imagen, que es 23 microsegundos como muestra la Ilustración 32.

The image is passed in the PL and the inference is performed. Use `classify_mnist` to classify a single mnist formatted picture.

```
import time

inic = time.time();
class_out = hw_classifier.classify_mnist("pictures/img_webcam_mnist_processed")
fin = time.time();
print(fin-inic)
print("Class number: {0}".format(class_out))
print("Class name: {0}".format(hw_classifier.class_name(class_out)))
```

```
Inference took 23.00 microseconds
Classification rate: 43478.26 images per second
```

Ilustración 32 *Inferencia con LFC*

4. Prueba del sistema

Para comprobar el correcto funcionamiento del sistema diseñado se ha desarrollado una aplicación en Python. Esta aplicación es la que se encarga de preprocesar la imagen que se quiere clasificar, enviarla al acelerador hardware usando el DMA, recibir el resultado y mostrar el resultado de la inferencia.

4.1 Aplicación Python

El Notebook completo se encuentra en al Apéndice 1. Describimos aquí algunas de sus funciones fundamentales:

1. **Conversión de formato de entrada**: Convierte la imagen de entrada del formato (28,28) a un vector de 13 elementos formados por números binarios de 64 bits.

```
def conversion_binario(img):
    i = 0; j = 0;
    res =["0b","0b","0b","0b","0b","0b","0b","0b","0b","0b","0b","0b","0b",]
    indice = 0; count = 0;
    while i < 28:
        while j<28:
            res[indice] = res[indice] + str(img[i,j]);
            count = count + 1;
            if (count == 64):
                count = 0;
                indice = indice + 1;
            j = j + 1;
        i = i + 1;
        j = 0;
    while count < 64:
        res[indice] = res[indice] + str(0)
        count = count + 1;
    return res
```

Código 9 *Función Python Conversión formato de entrada*

2. **Función clasificación**: Es la función encargada de inicializar el acelerador, y enviar y recibir los datos a la DMA.

```
def clasificacion(overlay,img):
    data_size = 13;
    CONTROL_REGISTER = 0x0;
    input_buffer = allocate(shape=(data_size,), dtype=np.uint64);
    output_buffer = allocate(shape=(data_size,), dtype=np.uint64);
    dma = overlay.axi_dma_0;
    dma_send = overlay.axi_dma_0.sendchannel;
    dma_recv = overlay.axi_dma_0.recvchannel;
    hls_ip = overlay.myfun_0;
    hls_ip.write(CONTROL_REGISTER, 0x81);
    for i in range(13):
        input_buffer[i] = int(img[i],2);
    dma_send.transfer(input_buffer);
    dma_recv.transfer(output_buffer);
    final = time.time();
    del input_buffer, output_buffer;
    return salida
```

Código 10 *Función Python Clasificación*

3. **Conversión salida a imagen de referencia e índice resultante:** Esta imagen extrae los datos obtenidos por el buffer de salida y devuelve la imagen de referencia más parecida en formato 28x28 y el índice resultante. Esta función utiliza una subfunción “bits_64” que castea los bits devueltos en 64 bits.

```python
def bits_64(num):
    long_n = len(num);
    if long_n == 66:
        return num;
    else:
        inicio = '0b';
        final = num[2:long_n];
        for i in range(64-len(final)):
            inicio = inicio + '0';
        inicio = inicio + final;
        return inicio

def conversion_image(imgB):
    fila = 0;
    col  = 0;
    #tamaño de las matrices a visualizar
    size=(28,28)

    # Una matriz de ceros.
    imagen_out = np.zeros(size)
    for i in range(13):
        vector = libreria.bits_64(bin(imgB[i]));
        for j in range(66):
            if j >= 2:
                imagen_out[fila,col] = vector[j];
                col = col + 1;
                if col == 28:
                    col = 0;
                    fila = fila + 1;
                if fila == 28:
                    break
    return imagen_out, int(vector[j:66],2)-1
```

Código 11 *Función Python Imagen e índice resultantes*

4. **Función Clasificación total:** Esta función engloba a las mencionadas anteriormente.

```python
def clasificacion_total(overlay,img):
    aux1 = libreria.conversion_binario(img);
    aux2 = libreria.clasificacion(overlay,aux1);
    im_out, ind_out = libreria.conversion_image(aux2);
    return im_out,ind_out;
```

Código 12 *Función Python clasificación total*

4.2 Ejemplos de operación

Tomamos, como ejemplo, la imagen mostrada en la Ilustración 33.

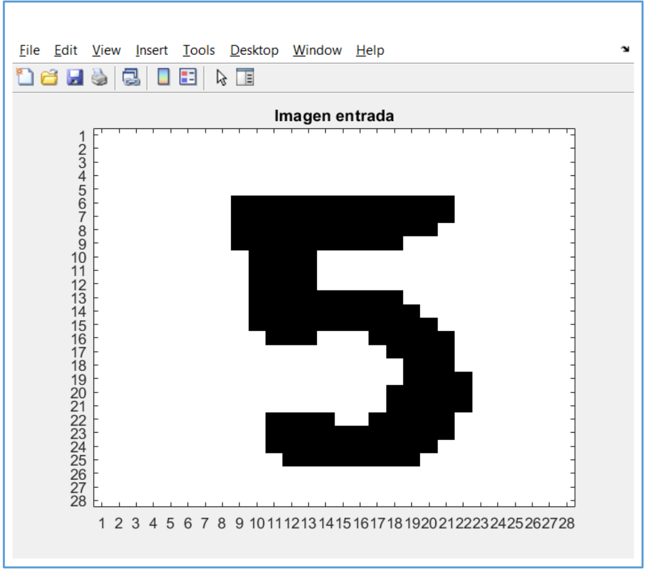


Ilustración 33 *Imagen Test de entrada*

Realizamos la conversión al formato de entrada y obtenemos:

```
['0b0000000000000000000000000000000000000000000000000000000000000000',
 '0b0000000000000000000000000000000000000000000000000000000000000000',
 '0b0000000000000000000011111111111110000000000000001111111111111000',
 '0b0000000000001111111111110000000000000000111111111100000000000000',
 '0b0000011110000000000000000000000011110000000000000000000000000111',
 '0b1000000000000000000000000111111111000000000000000000011111111110',
 '0b0000000000000000011111111111000000000000000000111000111110000000',
 '0b0000000000000000011110000000000000000000000001110000000000000000',
 '0b0000000000111100000000000000000000000111110000000000000000000000',
 '0b0111110000000000000000111100111110000000000000000011111111111000',
 '0b0000000000000011111111110000000000000000000111111110000000000000',
 '0b0000000000000000000000000000000000000000000000000000000000000000',
 '0b0000000000000000000000000000000000000000000000000000000000000000']
```

Ilustración 34 *Imagen Test de entrada formato deseado*

Como resultado obtenemos la imagen del *set* de referencia más parecida, Ilustración 35, y el índice de la imagen clasificada.

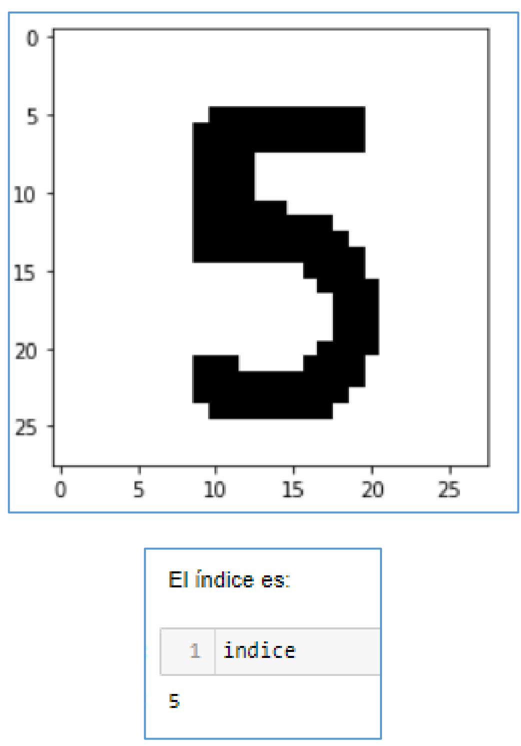


Ilustración 35 Imagen Salida función clasificación

Se ha demostrado satisfactoriamente cómo a partir de una imagen de test, el algoritmo implementado permite identificar su categoría correcta gracias a la métrica de distancia con el conjunto de referencia obtenido por el algoritmo K-means

En la Tabla 7 podemos observar diferentes imágenes de entrada y las salidas devueltas.

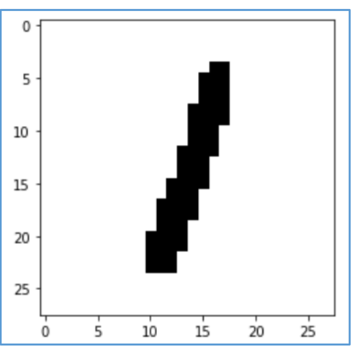
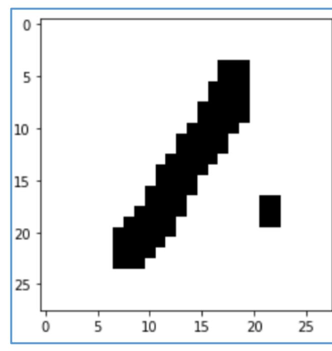
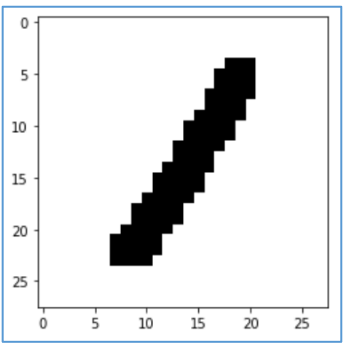
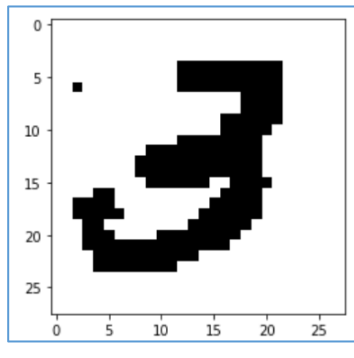
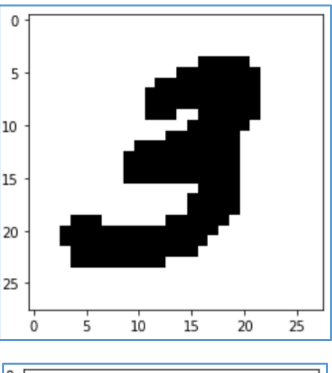
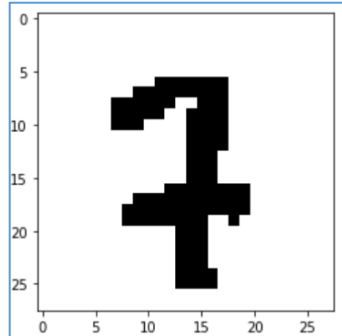
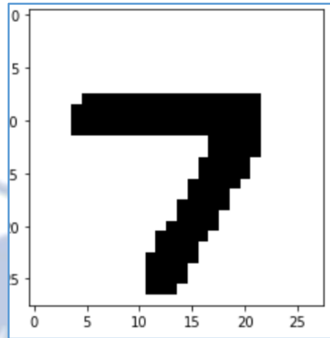
ENTRADA	SALIDA
	
	
	
	

Tabla 7 *Soluciones aplicación*

5. Conclusiones y líneas futuras

Las principales conclusiones alcanzadas tras la elaboración de este trabajo son:

a) Se ha explorado el entorno de desarrollo de sistemas hardware-software sobre *APSoCs* PYNQ. PNYQ proporciona numerosas oportunidades al usuario para el diseño de aplicaciones con aceleración hardware de una manera sencilla. Este entorno de desarrollo puede ser el futuro del *IoT* debido a su capacidad de combinar hardware y software mediante el uso de *Overlays* y el lenguaje de alto nivel Python.

b) Se ha desarrollado un algoritmo de clasificación de las imágenes del MNIST basado en el cálculo del vecino más próximo, de acuerdo con una métrica de distancia, y en la elección de las imágenes de referencia usando el algoritmo K-*means* que ha demostrado proporcionar un buen compromiso entre el *accuracy* y su simplicidad, lo que lo hace muy adecuado para su implementación hardware.

c) Se ha implementado un acelerador hardware para el algoritmo de clasificación desarrollado usando síntesis de alto nivel. Este acelerador se ha incorporado a una plataforma hardware (*Overlay*) que también incorpora un DMA y uno de los ARM del dispositivo *Zynq*.

d) Se ha validado el funcionamiento del sistema de clasificación de imágenes del MNIST diseñado en la placa Pynq-Z2. Para ello se ha desarrollado una aplicación Python que se comunica con el acelerador.

e) El sistema desarrollado es competitivo con respecto a una solución basada en una red LFC binaria que usamos como referencia. Los tiempos de inferencia obtenidos son bastantes asequibles y se utilizan menos recursos. Este proyecto obtiene un resultado en 11 microsegundos mientras la red en 23 y, por otro lado, el uso de recursos es menos de la mitad; lo que compensa la pérdida de 4% de *accuracy.*

f) Se han identificado posibles mejoras del sistema que pueden incrementar el rendimiento y líneas de trabajo futuro:

   a. Análisis de otras posibles optimizaciones en el contexto de HLS. Por ejemplo, incrementando aún más el paralelismo de los cálculos de las distancias. Ello reduciría el tiempo de inferencia, y aunque incrementaría el uso de recursos, estimamos que aún estaríamos por debajo de los utilizados por la LFC de referencia.
   b. Adaptación del flujo de datos de la función acelerada para implementar un *Data Flow* que nos permita incrementar el rendimiento al realizar clasificación de más de una imagen concurrentemente.
   c. Desarrollo de un acelerador en el que las imágenes de referencia no estén empotradas. Esto es, estas puedan ser transferidas desde el procesador para dotarlo así de mayor flexibilidad.
   d. Desarrollo de un acelerador Hardware para el algoritmo K-means. Esto es, se dotaría al sistema de una capacidad de entrenamiento on-chip.
   e. Evaluación de la combinación de K-means y algoritmo del vecino más próximo a otros bancos de prueba de clasificación.

Bibliografía

[1] Y. Umuroglu *et al.*, “FINN: A framework for fast, scalable binarized neural network inference,” *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, 2017, doi: 10.1145/3020078.3021744.

[2] “MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges.” http://yann.lecun.com/exdb/mnist/ (accessed Jun. 18, 2022).

[3] T. M. Cover and P. E. Hart, “Approximate formulas for the information transmitted bv a discrete communication channel,” 1952.

[4] X. Liu and J. Xue, “A Cluster Splitting Technique by Hopfield Networks and P Systems on Simplices,” *Neural Processing Letters*, vol. 46, no. 1, pp. 171–194, Aug. 2017, doi: 10.1007/s11063-016-9577-z.

[5] “PYNQ - Python productivity for Zynq - Home.” http://www.pynq.io/ (accessed Jun. 18, 2022).

[6] “Getting Started — Python productivity for Zynq (Pynq).” https://pynq.readthedocs.io/en/latest/getting_started.html (accessed Jun. 18, 2022).

[7] “PYNQ-Z2 Reference Manual v1.1 28,” 2019.

[8] L. Crockett, R. Elliot, M. Enderwitz, and B. Stewart, “The Zynq Book,” p. 484, 2014, [Online]. Available: http://www.zynqbook.com/

[9] Xilinx, “AXI Reference Guide,” *Data Sheet*, vol. 761, pp. 1–116, 2012.

[10] Xilinx, “[UG474] 7 Series FPGAs Configurable Logic Block,” *User Guide*, vol. 474, pp. 1–72, 2014, doi: 10.1086/690659.

[11] Xilinx, “7 Series FPGAs Data Sheet: Overview (DS180),” 2010.

[12] Xilinx, “Zynq-7000 SoC Data Sheet: Overview,” vol. 190, pp. 1–21, 2018.

[13] Xilinx, “Zynq-7000 SoC Technical Reference Manual - UG585 (v1.12.2) July 1, 2018,” vol. 585, pp. 1–1843, 2018.

[14] “Truth about Xilinx love affair with AMBA.” https://www.electronicsweekly.com/news/design/eda-and-ip/truth-about-xilinx-love-affair-with-amba-2010-06/ (accessed May 09, 2020).

[15] “Xilinx and ARM Announce Development Collaboration.” https://www.design-reuse.com/news/21810/xilinx-arm.html (accessed May 09, 2020).

[16] A. R. M. Limited, “AMBA AXI Protocol Specification,” 2010.

[17] ARM, “AMBA 4 AXI4-Stream Protocol,” vol. 1, 2010.

[18] "Vivado Design Suite User Guide Designing with IP." [Online]. Available: www.xilinx.com

[19] "Vivado Design Suite User Guide System-Level Design Entry," 2021. [Online]. Available: www.xilinx.com

[20] "Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator," 2019. [Online]. Available: www.xilinx.com

[21] "Vivado Design Suite User Guide Logic Simulation," 2022. [Online]. Available: www.xilinx.com

[22] "Vivado Synthesis." [Online]. Available: www.xilinx.com

[23] "Vivado Design Suite User Guide: Implementation (UG904)." [Online]. Available: www.xilinx.com

[24] "Vivado Design Suite User Guide Programming and Debugging." [Online]. Available: www.xilinx.com

[25] Xilinx, "Vitis High-Level Synthesis User Guide," 2021. [Online]. Available: www.xilinx.com

[26] R. del Pino, "Diseño de sistemas empotrados para aplicaciones de procesado de imagen y vídeo sobre FPGAs usando Vivado SDSoC," TFM, Tutor, M.J. Avedillo De Juan, 2019.

[27] "MATLAB - El lenguaje del cálculo técnico - MATLAB & Simulink." https://es.mathworks.com/products/matlab.html (accessed Jun. 11, 2022).

[28] "pragma HLS pipeline." https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/fde1504034360078.html (accessed Jun. 13, 2022).

[29] "HLS Pragmas." https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/okr1504034364623.html (accessed Jun. 14, 2022).

[30] "Tutorial: PYNQ DMA (Part 1: Hardware design) - Learn - PYNQ." https://discuss.pynq.io/t/tutorial-pynq-dma-part-1-hardware-design/3133 (accessed Jun. 13, 2022).

[31] Xilinx, "AXI DMA v7.1 LogiCORE IP Product Guide Vivado Design Suite." [Online]. Available: www.xilinx.com

[32] "Zynq-7000 Processing System IP." https://www.xilinx.com/products/intellectual-property/processing_system7.html (accessed Jun. 13, 2022).

[33] Xilinx, "Resources LFC of FINN." https://github.com/Xilinx/BNN-PYNQ/tree/master/bnn/src/network/lfcW1A1 (accessed Jun. 28, 2022).

ANEXO I

Cargamos librerías

```
In [53]:
1 from pynq import Overlay, PL
2 import matplotlib.pyplot as plt
3 import mi_libreria as myL
4 import scipy.io as sio
```

Cargamos Overlay y el set de test

```
In [54]:
1 ov = Overlay("TFM_v2.bit")
2 mat = sio.loadmat('test.mat')
```

Eligimos un valor

```
In [55]:
1 test = mat['MBt_total'];
2 img_test = test[:,:,26458]
```

```
In [56]:
1 plt.imshow(img_test,cmap='Greys',  interpolation='nearest')

Out[56]: <matplotlib.image.AxesImage at 0xad2cadc0>
```


Convertimos al formato de entrada deseada

```
In [57]:
1 img_binaria = myL.libreria.conversion_binario(img_test)
```

```
In [58]:
1 img_binaria

Out[58]: ['0b0000000000000000000000000000000000000000000000000000000000000000',
 '0b0000000000000000000000000000000000000000000000000000000000000000',
 '0b0000000000000000000000000000000000000000000000001100000000001000',
 '0b0000000000001110000000011000000000000000111000000011100000000000',
 '0b0000111000000011100000000000000011000000001110000000000000011100',
 '0b0000001110000000000000011100000000111000000000000001110000000011',
 '0b1000000000000011111111111111000000000000001111111111111111000000',
 '0b0000001110000011111111000000000000000000000011110000000000000000',
 '0b0000000011100000000000000000000000001110000000000000000000000000',
 '0b1110000000000000000000000001111000000000000000000000000111000000',
 '0b0000000000000000000111000000000000000000000000001110000000000000',
 '0b0000000000111000000000000000000000000000000000000000000000000000',
 '0b0000000000000000000000000000000000000000000000000000000000000000']
```

Clasificamos

```
In [59]:
1 resultado = myL.libreria.clasificacion(ov,img_binaria)
```

Clasificamos

```
In [59]:  1 resultado = myL.libreria.clasificacion(ov,img_binaria)
```

Extraemos el índice de la solución y la imagen del set referencia más cercana

```
In [60]:  1 img_sal, indice = myL.libreria.conversion_image(resultado)
```

El índice es:

```
In [61]:  1 indice
Out[61]: 4
```

El set de referencia más cercana es:

```
In [62]:  1 plt.imshow(img_sal,cmap='Greys',  interpolation='nearest')
Out[62]: <matplotlib.image.AxesImage at 0xae1c3c70>
```


Clasificación total

```
In [63]:  1 img_sal, indice = myL.libreria.clasificacion_total(ov,img_test)

In [64]:  1 indice
Out[64]: 4

In [65]:  1 plt.imshow(img_sal,cmap='Greys',  interpolation='nearest')
Out[65]: <matplotlib.image.AxesImage at 0xad4668c8>
```



```
In [ ]:  1 
```

Ilustración 36 *Jupyter Notebook*