Acelerador de red neuronal convolucional (CNN) para una aplicación de clasificación de imágenes en FPGA

Wilson Javier Almario Rodriguez

Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de ingeniería eléctrica y electrónica
Bogotá, Colombia
2021

Acelerador de red neuronal convolucional (CNN) para una aplicación de clasificación de imágenes en FPGA

Wilson Javier Almario Rodriguez

Trabajo de grado presentado como requisito parcial para optar al título de:
Ingeniero Electrónico

Director:
Ph. D. Carlos Ivan Camargo Bareño

Grupo de Investigación:
Grupo de Física Nuclear de la Universidad Nacional de Colombia

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de ingeniería eléctrica y electrónica
Bogotá, Colombia
2021

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La inteligencia artificial (IA) en los últimos años ha tomado gran interés debido a sus aplicaciones y su impacto en la cuarta revolución industrial, en especial encontramos el aprendizaje máquina (Machine Learning ML) como una rama de estudio de la IA, en la que un sistema toma un volumen de datos de entrada y es entrenado para identificar patrones y hacer predicciones entre otras aplicaciones. Una de las técnicas de ML es el aprendizaje profundo (deep learning), en el cual encontramos redes neuronales con varias capas ocultas las cuales involucran cálculos matriciales y vectoriales, que pueden ser ejecutados de forma paralela y que permite obtener una buena precisión en la predicción [3]. La precisión de la red en la inferencia está condicionada a diversos parámetros del modelo como son el tipo de neuronas y el número de capas a implementar. El proceso de diseño de redes neuronales se divide en dos etapas; el entrenamiento y la inferencia. En el entrenamiento se calculan los pesos del modelo usando un proceso de optimización, y debido a el costo computacional del mismo este se realiza generalmente en GPUs. Mientras que en la inferencia se evalúa la red neuronal con los pesos previamente calculados, por lo que se requieren menos recursos de memoria y cómputo lo que hace viable su inferencia en CPUs.

Uno de los modelos de redes neuronales de mayor interés en el estado del arte son las Redes Neuronales convolucionales (Convolutional Neural Network CNN). Este interés se atribuye al hecho a que una de las ventajas que tienen este tipo de redes es que estas pueden aprender directamente de los datos, sin necesidad de extraer característica manuales, las cuales son útiles para encontrar patrones en imágenes [4]. Esta característica permite obtener un buen desempeño en aplicaciones de análisis de imágenes tales como; sistemas de clasificación de imágenes [5], análisis de documentos [6] y reconocimiento de voz [7] entre otras. Como desventaja estas redes presentan un alto uso de recursos de memoria y de cómputo, lo cual ha motivado el desarrollo de implementaciones en hardware que tienen como objetivo mejorar el desempeño.

Por lo general, el uso de CPUs o GPUs en el proceso de inferencia ofrece un buen desempeño, sin embargo el consumo de potencia, la precisión y el paralelismo obtenidos están limitados por la arquitectura del hardware y las librerías de ML disponibles. Por esta razón, en los últimos años las implementaciones de redes neuronales en hardware a través de plataformas reconfigurables como las FPGAs se han incrementado, debido a que se puede lograr un alto desempeño a un bajo consumo de potencia logrando hasta 10 veces menos con relación a las CPUs y las GPUs [8].

En [9] se realiza una comparación de rendimiento entre una FPGA, una CPU y una GPU, llegando a las siguientes observaciones; el rendimiento de la FPGA es superior 10x rendimiento/Watt que las CPU y GPU. Con respecto a las ASIC la FPGA es 7x menos eficiente [9], pero las características de programación y reconfiguración de la FPGA hacen que sea posible

implementar diseños personalizados [10]. Por esta razón las FPGAs son atractivas para los sistemas embebidos en especial los sistemas móviles en los caules resulta indispensable optimizar el consumo de energía, pero a su vez aún hay mucho por mejorar por ejemplo para la implementación de RNN está por mejorar la superposición del tiempo de cálculo con el tiempo de transferencia de datos [11]. Además actualmente como principal foco de investigación es la implementación de aceleradores en FPGA para modelos específicos de redes neuronales esto principalmente debido a que cuando se implementa un modelo de red neuronal a un problema específico usualmente solo se deben configurar parámetros específicos en la red para un determinado problema [8].

Así podemos observar de lo anterior que la implementación de aceleradores en FPGA es una necesidad actual y por el cuál falta aún explorar mucho en la implementación de aceleradores en FPGA para modelos específicos de redes neuronales como lo mencionado en el anterior párrafo, por esta razón se plantea una propuesta de proyecto para la implementación de un acelerador en FPGA que permita realizar el proceso de inferencia de una CNN a través de un co-diseño hardware-software.

1.1. Motivación

La investigación actualmente en redes neuronales esta dirigida a que sean implementadas en sistemas embebidos y los diseños basados en FPGA han ganado bastante interés debido a que estos diseños logran un uso eficiente de la energía comparado con la implementación de modelos en redes neuronales en GPU y CPU. Pero también la implementación de redes neuronales en FPGAs requiere un número reducido de pesos y reducir operaciones aritméticas para simplificar su complejidad, en últimas se buscan metodologías para optimizar la complejidad computacional, el uso de recursos y el espacio en memoria, sin tener perdidas significativas en la precisión de la inferencia; por lo tanto este trabajo propone y explora una metodología de cuantización de una red convolucional (QCNN) para su implementación en hardware.

1.2. Objetivos

1.2.1. Objetivo general

Implementar un acelerador de una red neuronal convolucional (CNN) en una FPGA para una aplicación de clasificación de imágenes.

1.2.2. Objetivos específicos

1. Evaluar diferentes modelos de CNN para una aplicación de reconocimiento de imágenes usando TensorFlow en CPU y GPU, y determinar el modelo más viable para su implementación en hardware.
2. Implementar un acelerador de CNN en FPGA basado en el modelo obtenido previamente en TensorFlow, para una aplicación de clasificación de imágenes.
3. Desarrollar un wrapper para un bus de comunicaciones que permita la portabilidad del acelerador.
4. Comparar las métricas obtenidas (e.g. tiempo de inferencia) en la CPU, la GPU y el acelerador de CNN implementado en una FPGA.

1.3. Metodología

1.3.1. Evaluación y determinación de un modelo de CNN

La evaluación del modelo CNN se hará basada en la revisión bibliográfica, para determinar el modelo CNN a usar se tendrá como criterio de selección la más usada en el estado del arte para aplicaciones de procesamiento de imágenes, y la que mejor rendimiento tenga usando el Framework TensorFlow y TensorFlowLite. Después de está selección se diseñara una aplicación de clasificación, con su respectiva base de datos, para luego implementar la aplicación en una CPU y una GPU por medio de TensorFlow. En este proceso se analizarán los algoritmos que realizan las operaciones implicadas en software para el tipo de red seleccionada (e.g. funciones de activación - recursividad – inferencia), para luego plantear los recursos que se van a usar en hardware como tipo de datos, cantidad de memoria, si se va usar una memoria adicional (off-ship), cantidad de operaciones entre otras características implícitas en la red.

1.3.2. Diseño de un acelerador de CNN para ser implementado en una FPGA

Una vez determinado el modelo de CNN a implementar, se realizará un análisis teniendo en cuenta los procesos y funciones involucradas en esta red para determinar el esquema y arquitectura de la red CNN a implementar en el acelerador. Esta implementación puede ser usando solo módulos que optimicen las funciones de activación o la implementación de las capas ocultas ya sea en diferentes ordenes o números de matrices. Para ello se plantearan las alternativas presentes y como criterio de selección se tendrá en cuenta viabilidad, el uso en cuanto a los recursos globales requeridos y la relación costo/beneficio (i.e., recursos específicos requeridos frente a la eficiencia esperada).

1.3.3. Evaluación del acelerador

Para la evaluación se hará desde lo más básico hasta lo más complicado, seguido de parámetros simplificados de la red hasta completar el modelo seleccionado de acelerador para CNN, proceso iterativo en el cual se definirá la arquitectura apropiada para el acelerador, haciendo una selección de la arquitectura se procede a la evaluación de la misma usando la base de datos seleccionada que se uso en la evaluación del framenwork, a partir de esto se obtienen las métricas (e.g. Utilización de recursos - tiempo de inferencia - Almacenamiento en memoria) que aporten mayor información en la evaluación de valor de los aceleradores para CNN como alternativa favorable en la optimización de redes neuronales y su implementación. Por último se realizará la comparación con las métricas obtenidas con la CPU y la GPU, y se concluirá.

Capítulo 2

Clasificación de dígitos manuscritos usando una red CNN

La aplicación que se va usar para validar la implementación de la CNN en la FPGA es una tarea de reconocimiento de dígitos, en este caso hay diez dígitos (0 a 9) o diez clases para predecir, el modelo toma una imagen con un manuscrito el cual representa un número de un solo dígito y el modelo clasifica o indica que número está escrito en la imagen. La aplicación es diseñada en el *framework* de TensorFlow y TensorFlow Lite.

En este capítulo se aborda la metodología para el diseño de la aplicación de reconocimiento de imágenes, los cambios realizados para optimizar su aplicación en la FPGA y la técnica de cuantización implementada para lograr un mejor uso de los recursos en la FPGA.

2.1. Conjunto de datos

Para el desarrollo de la aplicación de reconocimiento de imágenes se usara la base de datos MNIST, la cuál es una base de datos con imágenes de 28x28 pixeles en escala de grises de manuscritos de un solo dígito entre 0 y 9. En total la base de datos consta de 70.000 imágenes de las cuales 60.000 corresponden al conjunto de entrenamiento y 10.000 para las pruebas. Todas las imágenes están etiquetadas con su respectivo dígito, en total hay 10 clases de dígitos. [12]

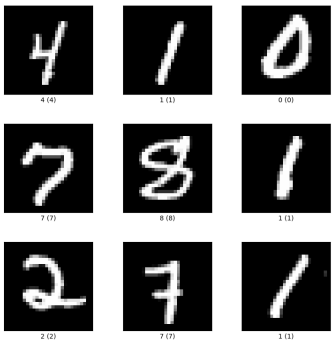

Figura 2.1: Ejemplo de las imágenes en la base de datos MNIST [1]

2.2. Creación y entrenamiento del modelo

El modelo que se toma como base para el desarrollo de la aplicación se desarrolla basado en los ejemplos mostrados en [13] y [14]. Se realiza un modelo con una entrada para una matriz de 28x28 la cual corresponde al tamaño de la imagen, luego la imagen pasa por 12 capas convolucionales con un filtro de 3x3 para obtener las características mas importantes de la imagen, después por una capa de reducción (*maxpooling*) que se encarga de reducir la cantidad de parámetros, por último este resultado pasa por una capa densa totalmente conectada (*full connected*), su función es hacer la clasificación entre los diez dígitos, el diagrama del modelo a implementar de muestra en la figura 2.2.

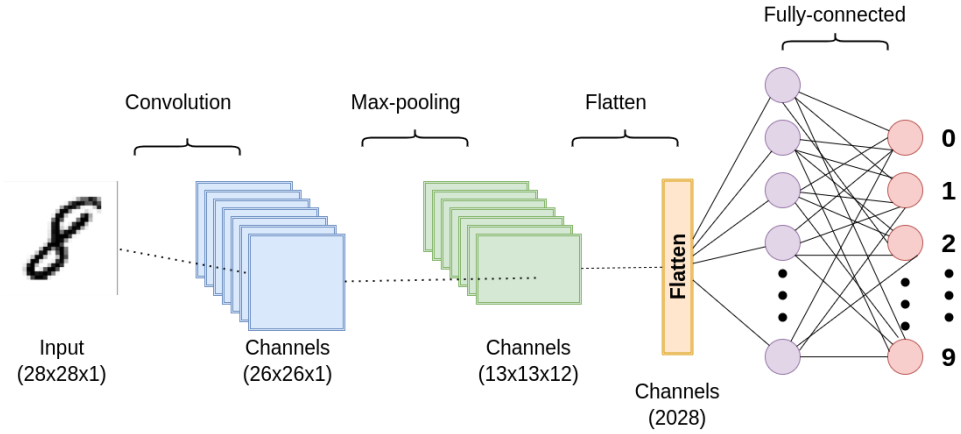

Figura 2.2: Diagrama del modelo de la red neuronal a implementar

A continuación se muestra la configuración de la red neuronal usando TensorFlow. Se carga las imágenes de la base de datos MNIST, en este caso no se le hará ningún tratamiento especial a los datos.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
reshape (Reshape)            (None, 28, 28, 1)         0
_________________________________________________________________
conv2d (Conv2D)              (None, 26, 26, 12)        120
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 12)        0
_________________________________________________________________
flatten (Flatten)            (None, 2028)              0
_________________________________________________________________
dense (Dense)                (None, 10)                20290
=================================================================
Total params: 20,410
Trainable params: 20,410
Non-trainable params: 0
```

Figura 2.3: Modelo diseñado en TensorFlow.

En este caso la red alcanza una precisión en la inferencia del 95.3 %, con 20410 parámetros. Se deja como base este modelo sobre el cuál en las siguiente sección se hará modificaciones con el objetivo de reducir la cantidad de parámetros ya que se busca que sea el menor número posible con una exactitud aceptable esto con el fin de optimizar los recursos en la FPGA.

2.3. Cuantización

Uno de los mayores inconvenientes al momento de hacer operaciones aritméticas en la FPGA tiene que ver en como están representado los números, por ejemplo si los números y los resultados de las operaciones son reales se necesitara una representación de punto flotante (*Floating-point*) la cual consiste en el signo, el exponente y la fracción lo que es muy similar a la notación científica [2], en la figura 2.4 se muestra como es la representación en 32 bits. Esto por supuesto implica que para cada número y resultado de una operación se destinen 32 bits en la FPGA. Con el objetivo de optimizar los recursos en la FPGA se usara representación de entero punto fijo (*Fixed-point integer*) para los números, la cual para la representación de signo usa complemento a dos.

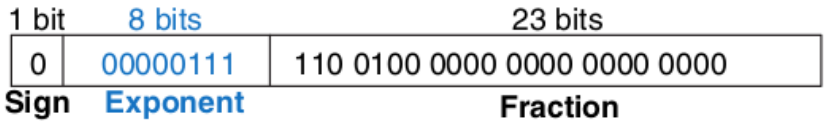

Figura 2.4: Representación en punto flotante. [2]

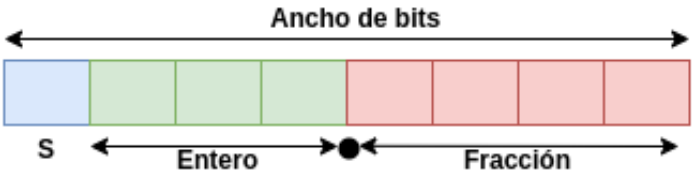

Figura 2.5: Representación en punto fijo con fracción.

Figura 2.6: Representación en punto fijo solo entero.

Al proceso que permitirá usar aritmética de solo enteros en el modelo se le llama cuantización, para realizar la cuantización del modelo previamente diseñado en la sección 2.2, se usara el módulo de TensorFlow Lite, en el cuál básicamente toma el modelo y genera uno nuevo con los parámetros representados en enteros de 8 bits.

2.3.1. Esquema de cuantización

El esquema de cuantización adoptado por TensorFlow Lite y que será usado más adelante cuando se implemente las operaciones de inferencia en la FPGA está basado en el siguiente artículo [15]. A continuación se hace una breve explicación del esquema de cuantización.

El esquema de cuantización consiste en una correspondencia entre la representación en bits de los valores, representado por q y su interpretación matemática de número real, denotada por r:

$$r = S(q - Z) \tag{2.1}$$

La ecuación 2.1 es el esquema de cuantización dónde S y Z son parámetros de cuantización, el esquema utiliza un solo conjunto de parámetros de cuantización, para una cuantización de 8 bits q es cuantizado como un entero de 8 bits. La constante de escala S es una valor real positivo arbitrario, la constante Z punto cero (*Zero-point*) este nos permite cumplir con el requisito de que el valor real $r = 0$ sea exactamente representable por un valor cuantizado.

2.3.2. Multiplicación de matrices de números enteros

Ahora se muestra como realizar operaciones aritméticas usando representación en enteros. Considere la multiplicación de dos matrices cuadradas $N \times N$ de números reales r_1 y r_2, se denotan los enteros de cada matriz como $r_\alpha(\alpha = 1, 2o3)$ como $r_\alpha^{i,j}$ para $1 \leq i, j \leq N$ y los parámetros con los que son cuantizados como S_{α,Z_α}, así la ecuación 2.1 queda:

$$r_\alpha^{i,j} = S_\alpha(q_\alpha^{i,j} - Z_\alpha). \tag{2.2}$$

De la definición de una multiplicación matricial se tiene:

$$S_3(q_3^{i,k} - z_3) = \sum_{j=1}^{N} S_1(q_1^{i,j} - Z_1)S_2(q_2^{j,k} - Z_2) \tag{2.3}$$

organizando términos:

$$S_3 = Z_3 + M\sum_{j=1}^{N}(q_1^{i,j} - Z_1)(q_2^{j,k} - Z_2) \tag{2.4}$$

dónde M se define como:

$$M = \frac{S_1 S_2}{S_3} \tag{2.5}$$

En la 2.4 el único valor que no es entero es M dependiendo del valor de las escalas y permitiéndose calcular de manera externa. Siempre se halla este valor de forma empírica dentro de un rango de $(0, 1)$ y puede ser expresado en forma normalizada:

$$M = 2^{-n}M_0 \tag{2.6}$$

dónde M_0 esta en el intervalo de $[0{,}5, 1)$ y n es un entero positivo. M_0 se presta para ser representado como multiplicados de punto fijo, por ejemplo int16 o int32 dependiendo las capacidades del hardware para este caso se usara ranto para software como para hardware (FPGA) int32.

2.3.3. Modelo cuantizado

Al pasar el modelo por el módulo TensorFlow Lite se obtiene una precisión en la inferencia del 91.6 %. Como lo que se busca es reducir el modelo teniendo una precisión aceptable al momento de ser implementada en la FPGA se varia el número de convoluciones del modelo, los resultados de la variación se muestran en la siguiente tabla:

Número de convolucionales	Precisión sin cuantizar	Precisión cuantizado	Número de parámetros
12	95.4 %	91.6 %	20410
6	94.1 %	91.5 %	10236
3	90.5 %	87.6 %	5130
1	80.5 %	68.4 %	1726

Tabla 2.1: Comparación de modelos variando la capa convolucional

Como criterio de selección se tiene en cuenta la precisión del modelo y el número de parámetros a implementar ya que estos serán almacenados en memorias al momento de implementar el acelerador en la FPGA, se selecciona el modelo con 3 convolucionales ya que ofrece una exactitud por encima del 80 % después de ser cuantizado y los parámetros necesarios para su implementación es casi 1/4 del modelo original de 12 convolucionales.

Capítulo 3

Arquitectura implementada en hardware

En este capítulo se explicara como se diseña el acelerador y cada uno de los módulos que lo integran; para el diseño se eligió como lenguaje de descripción de hardware (HDL) Verilog. Basado en el diseño de la sección 2.2 a continuación se listan las operaciones a implementar en hardware:

1. Cuantización.
2. Convolución.
3. Función de activación *Relu*.
4. *Max-pooling*.
5. *fully Connected*.

La arquitectura que se plantea es de forma modular, primero se realizara un módulo por cada operación listada anteriormente, luego de esto se integraran todos los módulo por medio de una unidad de control, todo esto integrado con la unidad de control será lo que se denominara el acelerador para una red convolucional.

Después de realizado el acelerador este será implementado por un procesador para su respectiva validación, en la figura 3.1 se aprecia la arquitectura propuesta. El acelerador será conectado al procesador a través de un bus de interconexión el cuál permitirá controlar el acelerador y enviarle los datos de la imagen a inferir para este caso especifico se enviara a través del bus de comunicaciones una matriz numérica de 28×28 y se espera que por el acelerador devuelva un valor el cuál corresponde al valor de la inferencia; toda esta estructura conformara un SoC (*System on Chip*) implementado en la FPGA.

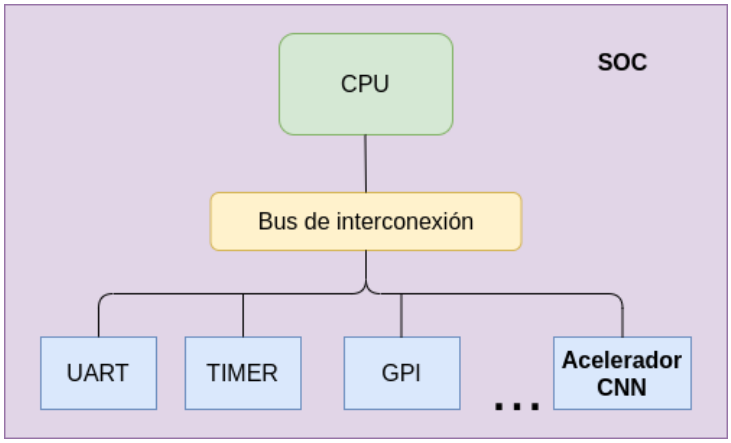

Figura 3.1: Estructura de como se va implementar el acelerador con el procesador en la FPGA

3.1. Cuantización y reglas aritméticas de punto fijo (Fixed-point)

3.1.1. Aritmética de punto fijo

Las 3 principales operaciones utilizadas en todo el diseño son sumas con signo, multiplicaciones y corrimiento de bits para implementar las multiplicaciones por potencias de dos. Para las sumas con signo, la regla es sencilla ambos números necesitan ser escalados en la misma base, con n siendo la misma (2.6) [2]. Para la multiplicación con signo se interpreta como todos los valores enteros como valores de punto fijo en el intervalo $[-1, 1)$ redondeando al valor más cercano y saturando con -1×-1 al valor máximo. El mapeo entre un tipo de dato entero y el intervalo $[-1, 1)$ está implícito en el valor de tipo entero, que se asume que es con signo, así por ejemplo para una multiplicación de $a \times b$ donde los valores son de tipo entero con n bits y están escalados por 2^n se necesita escalar el resultados por 2^{-n} o realizar un corrimiento aritmético de n bits, así si c es el resultado se tiene que:

$$\frac{a}{2^{-n}} \times \frac{b}{2^{-n}} = c2^{2n} \tag{3.1}$$

3.1.2. Diseño módulo de cuantización

Como se mencionó en la sección 2.3 la cuantización está basada en TensorFlow Lite, para esto después de realizado el modelo se exportan los parámetros necesarios para ser implementados en hardware a través de las ecuaciones descritas en la sección 2.3.1; así para completar las operaciones se diseña un módulo que ejecuta estas operaciones de forma secuencial, el cuál toma en completarlas 12 ciclos de reloj como se aprecia en la figura 3.2.

Figura 3.2: Ciclos de reloj usados para la cuantización.

El módulo se diseña con 4 entradas y 2 salidas. Este módulo será instanciado por el módulo de la operación convolucional y la capa densa actuando como un submódulo figura (3.3), dónde se ejecutan las operaciones principales de multiplicación y suma, y luego el resultado se cuantiza.

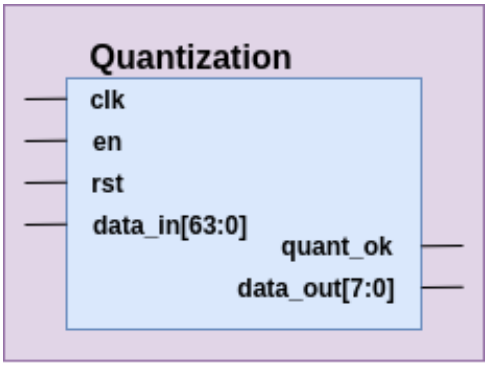

Figura 3.3: Módulo de cuantización implementado en hardware.

Las variables que están parametrizadas en el módulo y que deben ser ajustadas para cada operación a cuantizar antes de la sintetización son: S, M, M_0 y Z.

3.2. Función de activación

La función de activación usada en el modelo diseñado en la sección 2.2 es la unidad lineal rectificada (ReLU por sus siglas en inglés), esta definida como:

$$f(x) = max(0, x) \tag{3.2}$$

su representación gráfica se aprecia en la siguiente imagen:

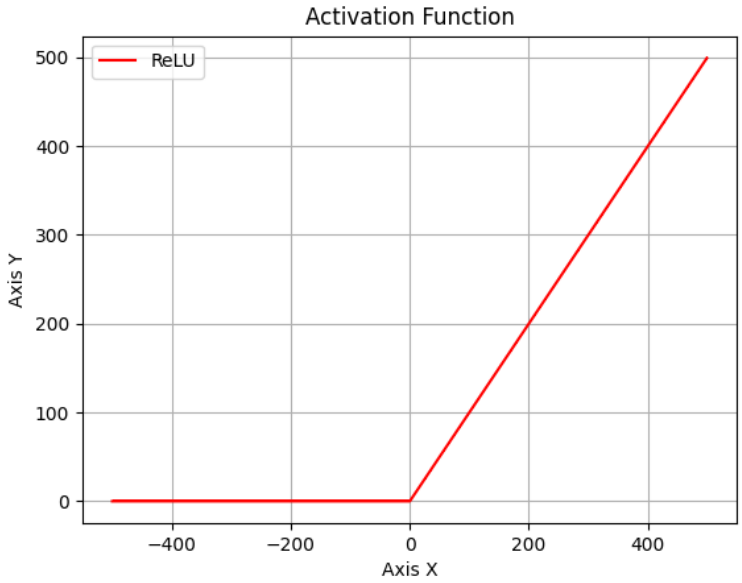

Figura 3.4: Función de activación ReLU.

esta función es usada ya que cada capa de procesamiento debe tomar un patrón en la imagen y pasarla a la siguiente capa, generalmente los valores negativos no son importantes en el procesamiento de imágenes y por ende se establecen en cero, pero los valores positivos deben pasar es por eso que la función ReLu es de utilidad.

Para la implementación en hardware se realiza una variación a la función de activación, debido a que a partir de la cuantización, por ejemplo para la representación de cada píxel de la imagen ya no será un entero positivo (0-255) si no que en vez sera representado como un entero con signo de 8 bits, la representación de un valor numérico con este tipo de dato esta en el rango de $[-128, 127]$, así en este caso los valores negativos son tan importantes como los positivos, así definimos la ecuación de activación a implementar de la siguiente forma:

$$f(x) = \left\{ \begin{array}{ccc} 127 & si & x > 127 \\ -128 & si & x < -128 \\ x & si & -128 < x < 127 \end{array} \right\} \tag{3.3}$$

Así no hay perdida de información y podría tomarse este paso como una extensión del módulo de cuantización garantizando que los valores esten dentro de la representación de 8 bits con signo. La gráfica de la función se aprecia en la siguiente imagen:

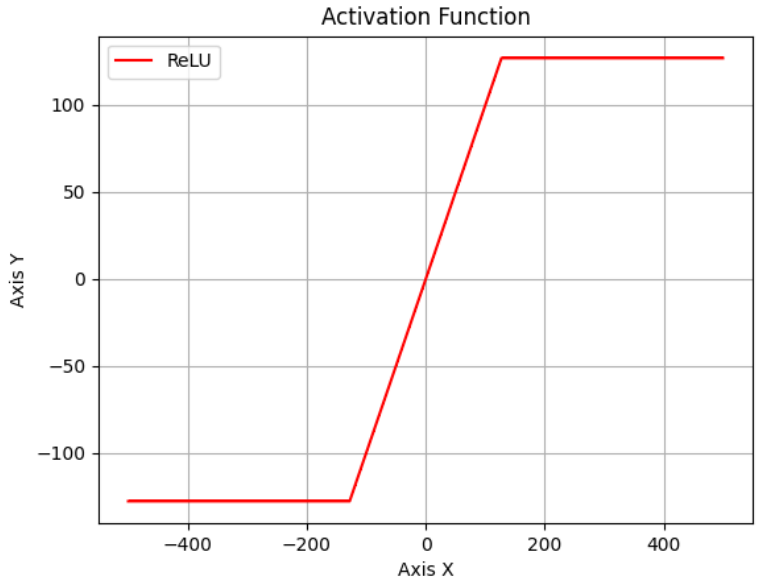

Figura 3.5: Función de activación adaptada al proyecto.

así el módulo a implementar en hardware es como el se aprecia en la imagen 3.6 que ejecuta la operación en 4 ciclos de reloj y devuelve un valor en representación de 8 bits con signo.

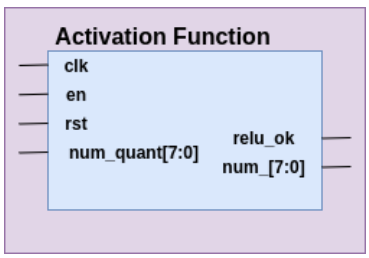

Figura 3.6: Módulo para la función de activación implementado en hardware.

3.3. Capa convolucional

La convolución para una matriz de 2 dimensiones se define como:

$$Y(x,y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} X(x-i, y-j) \cdot W(i,j) \tag{3.4}$$

Dónde $Y(x,y)$ es el píxel en la posición (x,y) en la imagen de salida, $X(x,y)$ es el píxel de la imagen de entrada y $W(i,j)$ es el filtro. Para este caso el tamaño de la imagen de entrada es de 28×28 y el tamaño del filtro es de 3×3. Las matrices de entrada en este caso para la imagen y el filtro se almacenan en memorias, pero estos se van a guardar en registros de una dimensión es decir para la memoria que almacenara la imagen se tiene un tamaño de 784 posiciones y de 9 posiciones para el filtro. Lo siguiente es adaptar la ecuación 3.4 ya no en 2 dimensiones si no en 1 dimensión.

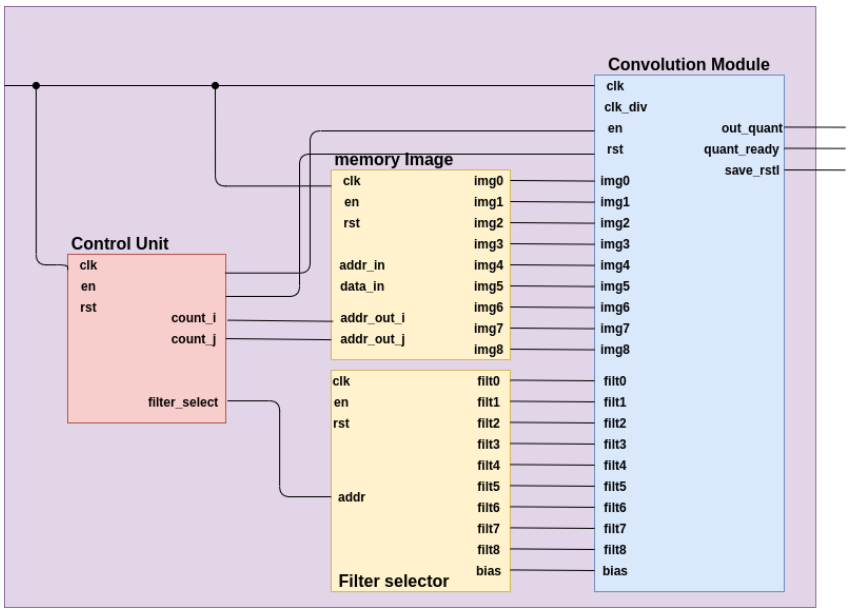

Figura 3.7: Diseño de módulo de convolución de forma secuencial.

Se plantea el diseño mostrado en la figura 3.7, en dónde se plantea una memoria síncrona multipuerto para la imagen y para el filtro, las direcciones de los valores de las salidas están controlados por un contador de dos posiciones que emula los valores i, j implementando la ecuación 3.4 como si estuviera recorriendo la matriz de 28×28, para obtener los valores en las posiciones indicados por (i,j) en la memoria de la imagen para cada puerto de salida implementa las siguientes ecuaciones:

$$img0 = (0+i) * (n_c) + (0+j) \tag{3.5}$$

$$img1 = (0+i) * (n_c) + (1+j) \tag{3.6}$$

$$img2 = (0+i) * (n_c) + (2+j) \tag{3.7}$$

$$img3 = (1+i) * (n_c) + (0+j) \tag{3.8}$$

$$img4 = (1+i) * (n_c) + (1+j) \tag{3.9}$$

$$img5 = (1+i) * (n_c) + (2+j) \tag{3.10}$$

$$img6 = (2+i) * (n_c) + (0+j) \tag{3.11}$$

$$img7 = (2+i) * (n_c) + (1+j) \tag{3.12}$$

$$img8 = (2+i) * (n_c) + (2+j) \tag{3.13}$$

dónde n_c corresponde al valor del tamaño de las columnas de la matriz, para este caso es de 28. Después el módulo de convolución se encarga de hacer la multiplicación, luego la suma, proceso de cuantización y por último pasa el valor por la función de activación. El tamaño de la matriz de salida por cada operación de convolución es de 26×26 es decir en la implementación requiere 3 memorias RAM de $5{,}4kb$ para almacenar el resultado.

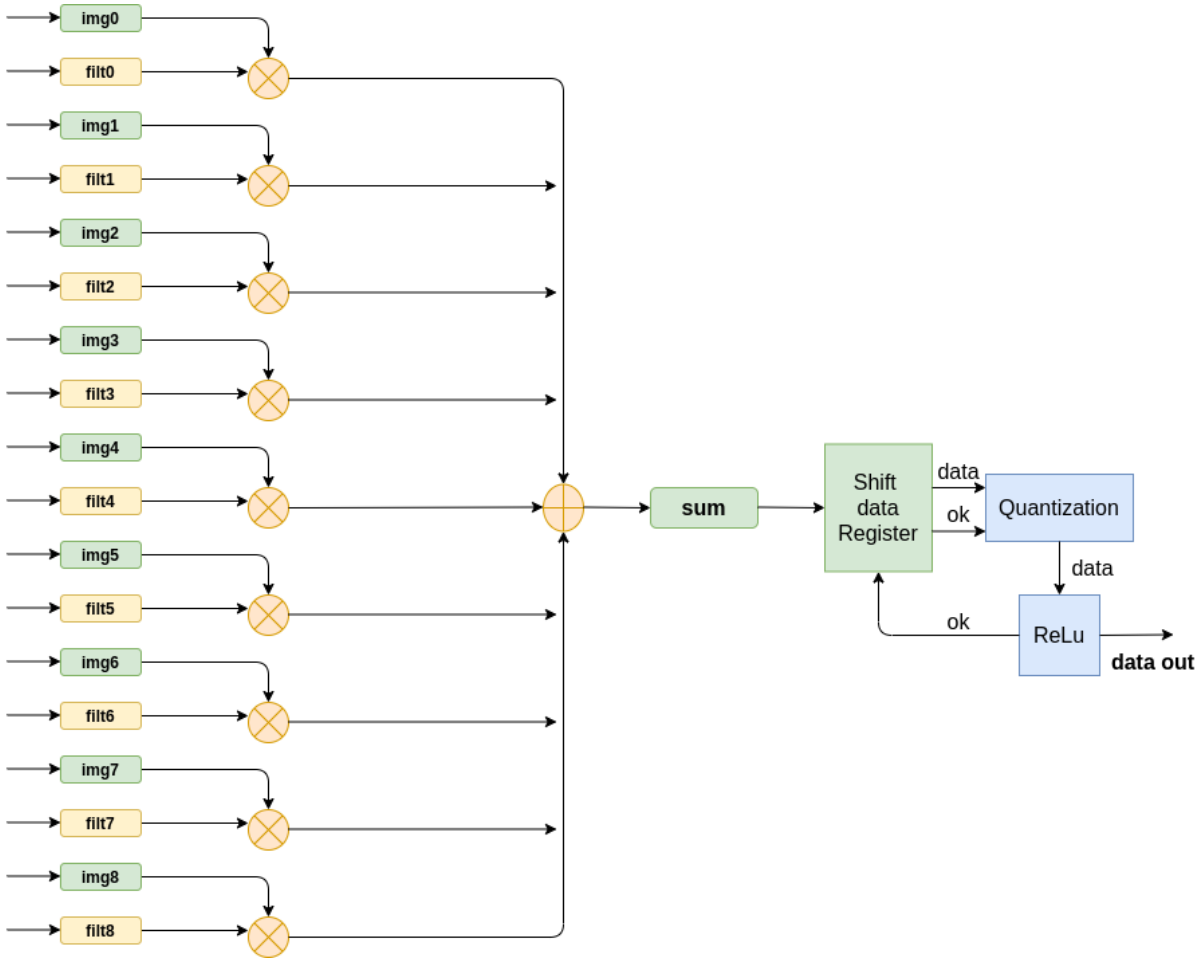

Figura 3.8: Operaciones en el modulo de convolución.

El diagrama del proceso para la convolución se aprecia en la figura 3.8, los procesos involucrados en la operación se muestran en la imagen 3.9, dónde FSM1 corresponde a la máquina de estados de la convolución, FSM2 a la cuantización y FSM3 a la función de activación.

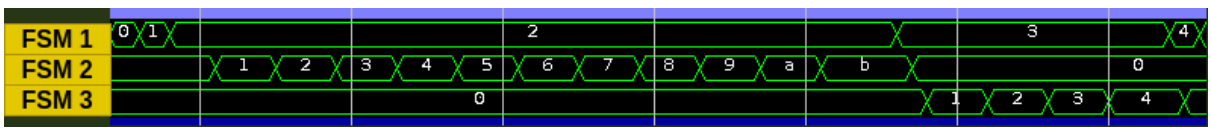

Figura 3.9: Ciclos de reloj para la convolución.

La máquina de estados para FSM1 se muestra en la figura 3.10:

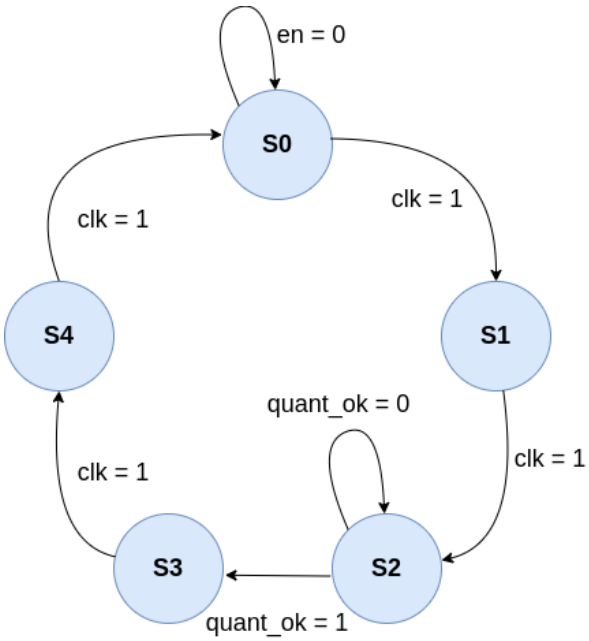

Figura 3.10: Máquina de estados para la convolución.

Cada estado corresponde a lo siguiente:

- S0: Multiplicación.
- S1: Suma.
- S2: Cuantización.
- S3: Función de activación.
- S4: Finalización e indicador para guardar valor.

3.3.1. Diseño alternativo

La arquitectura planteada para la convolución mostrada en la imagen 3.7 funciona de forma secuencial, esto quiere decir que la operación de convolución se hace una por cada filtro y no se hace el siguiente hasta que finalice completamente el primero.
Alternativo a este diseño se plantea que las 3 operaciones de convolución requeridas para los tres filtros se haga de forma paralela, así se plantea el diseño de la figura 3.11, en dónde las operaciones se realizan al mismo tiempo lo cuál implica un menor tiempo de ejecución en la operación.

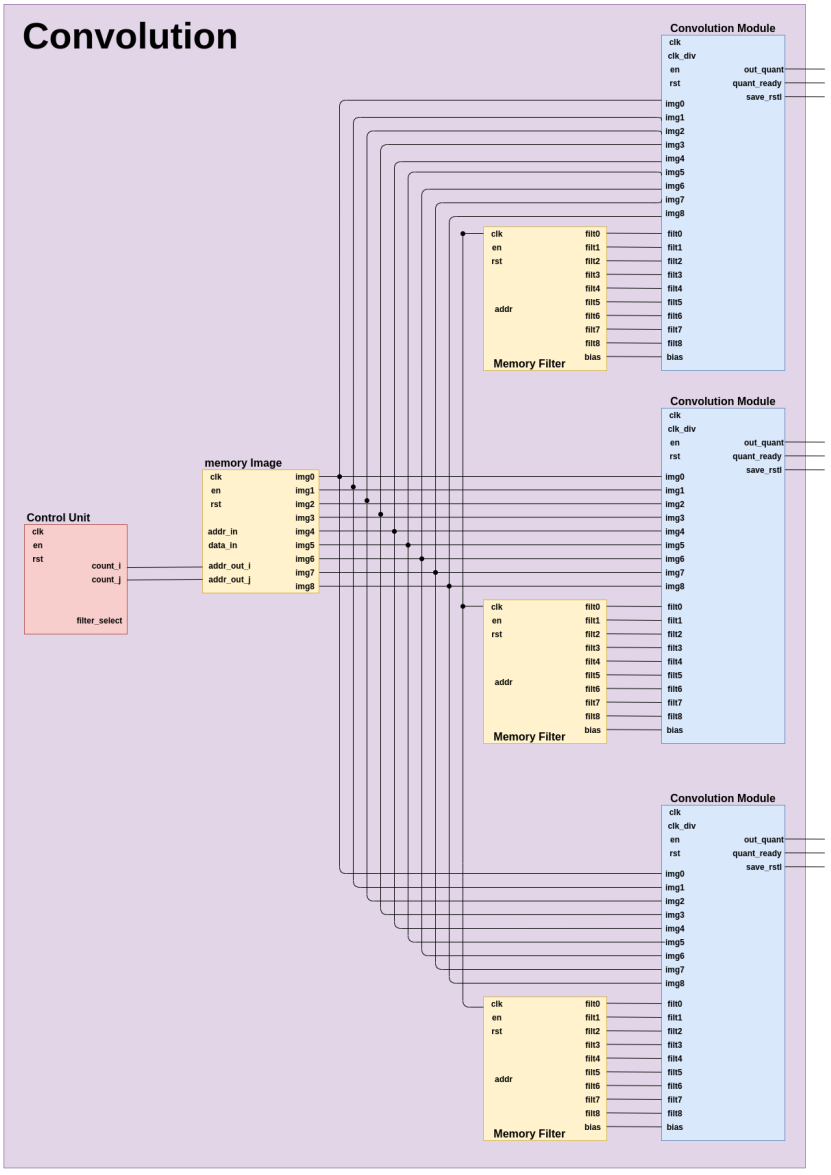

Figura 3.11: Diseño de módulo de convolución de forma paralela.

Se implementan los dos diseños en la FPGA con la finalidad de compararlos y elegir el más óptimo, para esto lo que se hizo fue variar el número de convoluciones implementadas en la FPGA tanto de manera secuencial como de forma paralela y se obtuvieron los tiempos de ejecución, la gráfica 3.12 es el resultado de esta comparación.

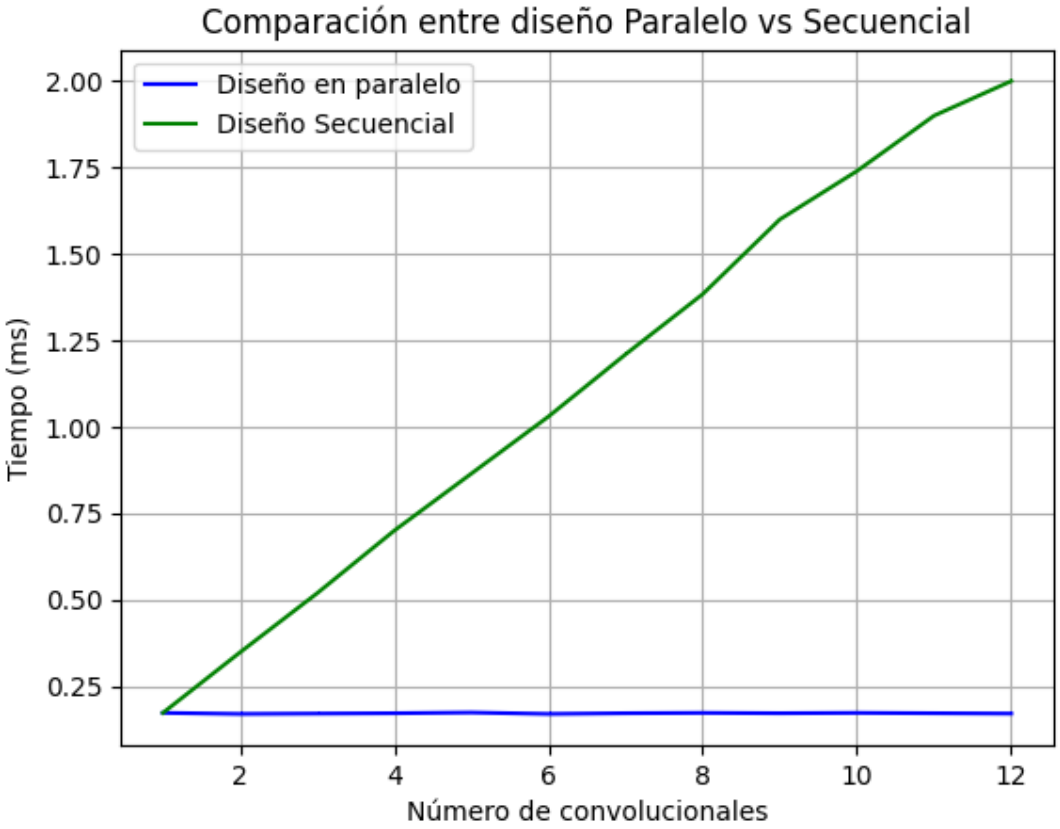

Figura 3.12: Comparación en tiempo de ejecución entre el diseño en paralelo frente al diseño secuencial.

En la gráfica se aprecia que el incremento del tiempo de ejecución, donde se puede apreciar que este incrementa de forma casi lineal cuando se implementa de forma secuencial mientras que cuando es en forma paralela el tiempo de ejecución es constante. Así en cuanto a tiempo de ejecución el diseño en forma paralelo es el más óptimo, pero tiene como desventaja que va utilizar más recursos de la FPGA que de forma secuencial, en la tabla 3.1 se realiza la comparación para 3 convoluciones sin tener en cuenta las memorias RAM.

Diseño 3 convolucionales	**Tiempo de ejecución (ms)**	**LUTs**	**LUTs as Logic**	**Slice**
Paralelo	0.173	564	564	321
Secuencial	0.520	188	265	107

Tabla 3.1: Comparación tiempo de ejecución entre el diseño en paralelo y el diseño secuencial.

3.4. Capa de reducción (Max-pooling)

La operación *Max-poolin* es un tipo de operación convolucional, pero en vez de tomar el producto punto entre el kernel y la imagen de entrada toma el máximo en una región de la matriz que representa la imagen, en la figura 3.13 se ilustra como funciona esta operación, así el resultado después de la capa 3.13 sería un mapa de características que contiene las características más destacadas del mapa de características anterior.

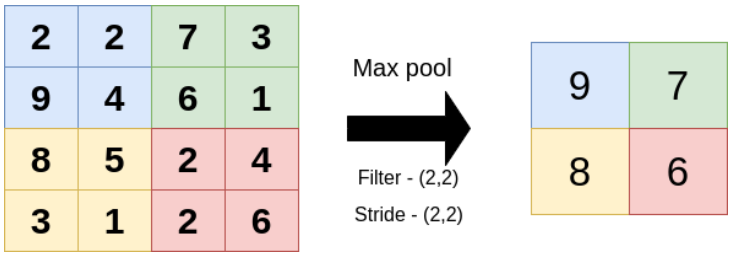

Figura 3.13: Operación *Max-pooling*

Para la implementación de la operación *Max-pooling* se diseñan los siguientes módulos como se ven en la figura 3.14, en donde tiene una unidad de control que se encarga de controlar la memoria donde se almacena los resultados de la operación convolución para pasarlos al modulo de convolución, similar a la operación de convolución (figura 3.11) este se implementa en paralelo debido a la reducción en tiempo de ejecución que este representa.

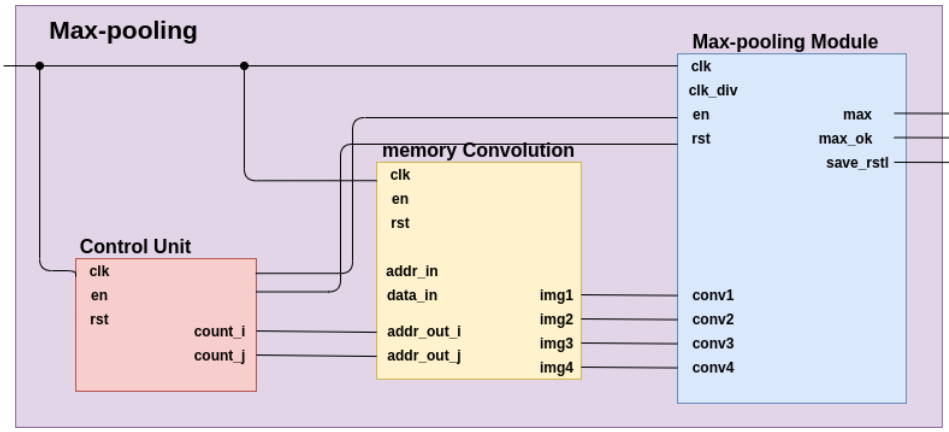

Figura 3.14: Diseño de módulo de *Max-pooling*

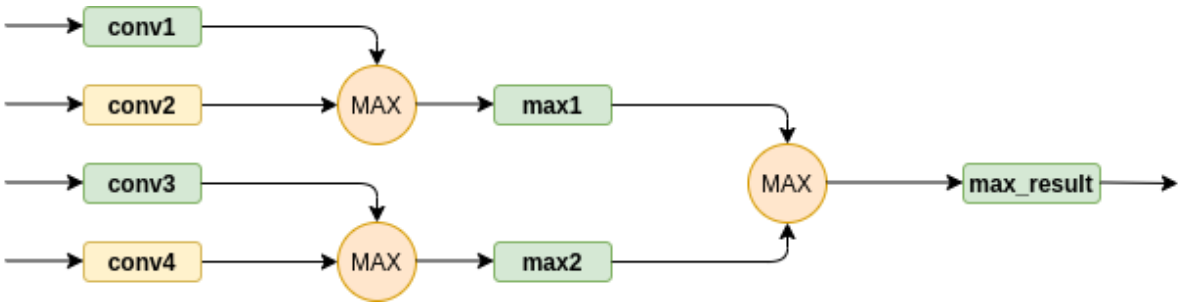

Figura 3.15: Operaciones en el módulo de *Max-pooling*

La forma en como el módulo de 3.13 esta representado en la figura 3.15, como se puede apreciar se replica la operación explicada en la figura 3.13, dónde por cada ciclo de reloj entran 4 parámetros correspondientes al tamaño de la ventana del kernel 2×2, en los siguientes ciclos de reloj se hayan los valores máximos entre estos valores obteniéndose un único valor como salida del módulo, el cuál se almacena en una memoria, la máquina de estados que representa la operación es como la que se ilustra en la 3.16. La matriz resultante después de esta operación es de 13×13 lo que requiere 3 memorias RAM de $1{,}3kb$ para almacenar los resultados.

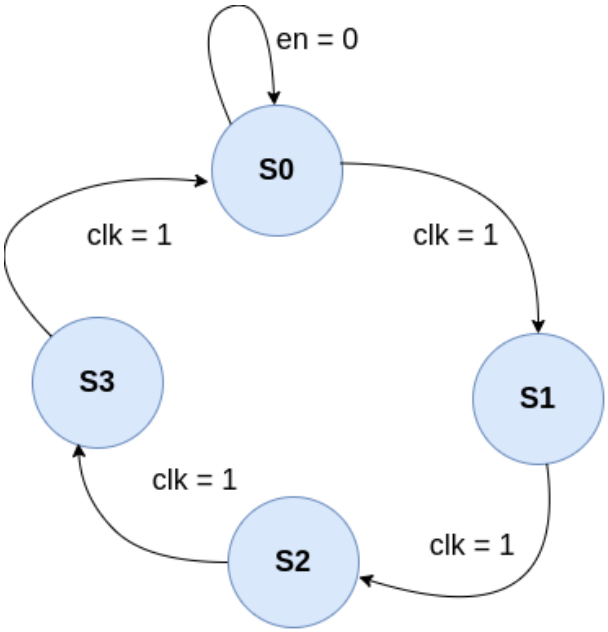

Figura 3.16: Máquina de estados para *Max-pooling*

3.5. Capa multiplicadora y de clasificación (Fully Connected)

Esta capa posee una matriz de pesos W y una matriz de polarización B, del modelo explicado en la sección 2.2 se tiene 10 matrices con 507 parámetros, que se multiplica por el resultado de la *Max-pooling* que como se mostró en la sección anterior por cada operación de *Max-pooling* se tiene como resultado una matriz de 13×13 es decir 169 parámetros y al ser 3 operaciones se obtiene 507 parámetros de la *Max-pooling*; este resultado se multiplica por cada una de las 10 matrices de la capa densa y sumando la polarización.

$$H = I * W + b \tag{3.14}$$

donde W es un arreglo de matrices $\begin{bmatrix} w_0 & w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7 & w_8 & w_9 \end{bmatrix}$, b corresponde a $\begin{bmatrix} b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 \end{bmatrix}$ e I es una matriz que contiene el resultado de las operaciones de *Max-pooling* y H es el resultado de la inferencia, H tendrá un arreglo de valores $\begin{bmatrix} h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 & h_9 \end{bmatrix}$ donde la posición del valor corresponde a un dígito, por ejemplo h_0 corresponde al dígito 0, h_1 al dígito 1 y así sucesivamente, a este proceso se le llama inferencia.

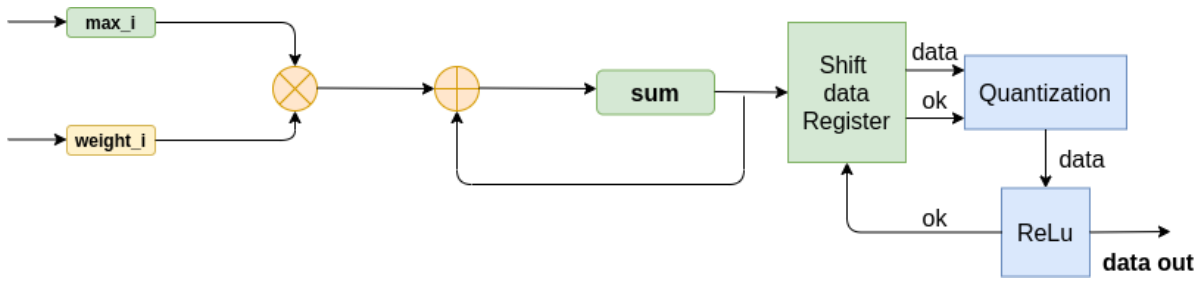

Figura 3.17: Diseño de operaciones para el módulo de capa densa.

El diagrama que ilustra como es su implementación en hardware se ilustra en la figura 3.17, en este se aprecia que después de cada operación de multiplicación y suma se instancian los módulos de cuantización y de la función de activación para completar el proceso de inferencia.

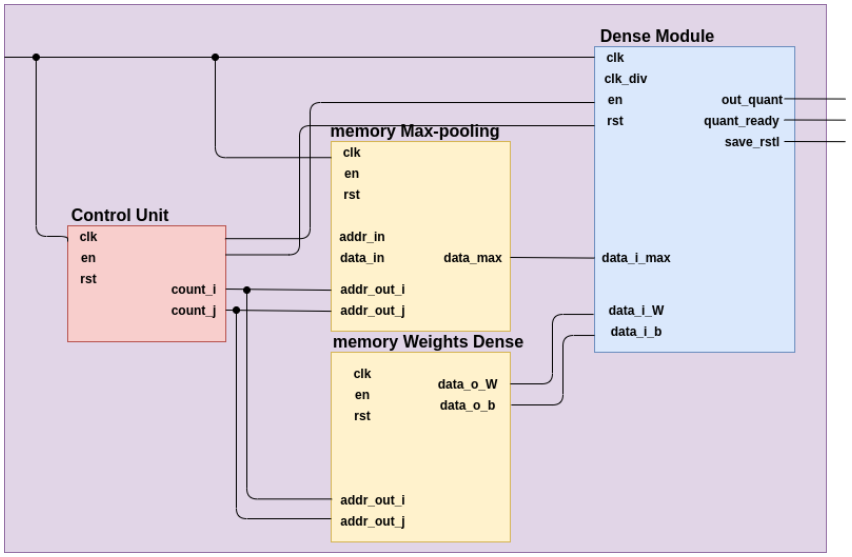

Figura 3.18: Diseño de módulo de capa densa.

Por último en el modulo de la capa densa los valores del arreglo pasan por un proceso denominado clasificación en donde se determina cuál es el valor máximo dentro de este arreglo la posición con el valor máximo corresponderá al dígito que está en la imagen de entrada del modelo, el proceso implementado en hardware se ilustra en la figura 3.20.

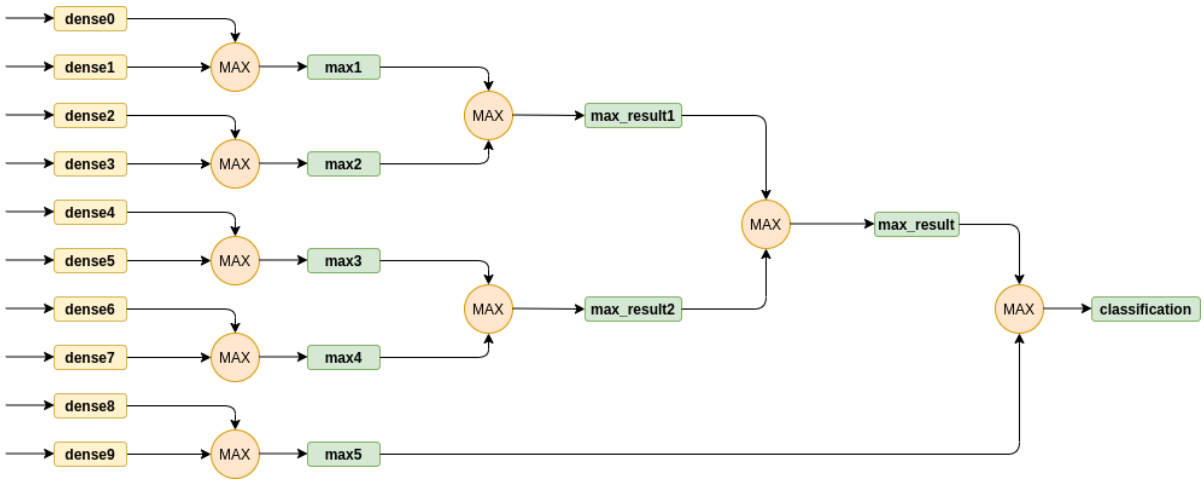

Figura 3.19: Proceso de clasificación.

3.6. Integración con procesador

Una ilustración del acelerador completo después de unir todos los módulos diseñados en las secciones anteriores se muestra en la figura 3.20; se ha creado una unidad de control que se encarga de gestionar los ciclos de reloj para cada módulo, determinar cuando iniciar una operación y cuando detenerla, gestionar memorias y por último comunicación con otros dispositivos (*wrapper*). Las memorias ROM almacenan los pesos de los filtros de la capa convolucional y los pesos para la capa densa, en las memorias RAM se almacenan los resultados temporalmente de cada operación; en azul se representan las operaciones de cada capa, amarillo para memorias y comunicación, y en rojo unidad de control y buses de comunicación con unidad de control.

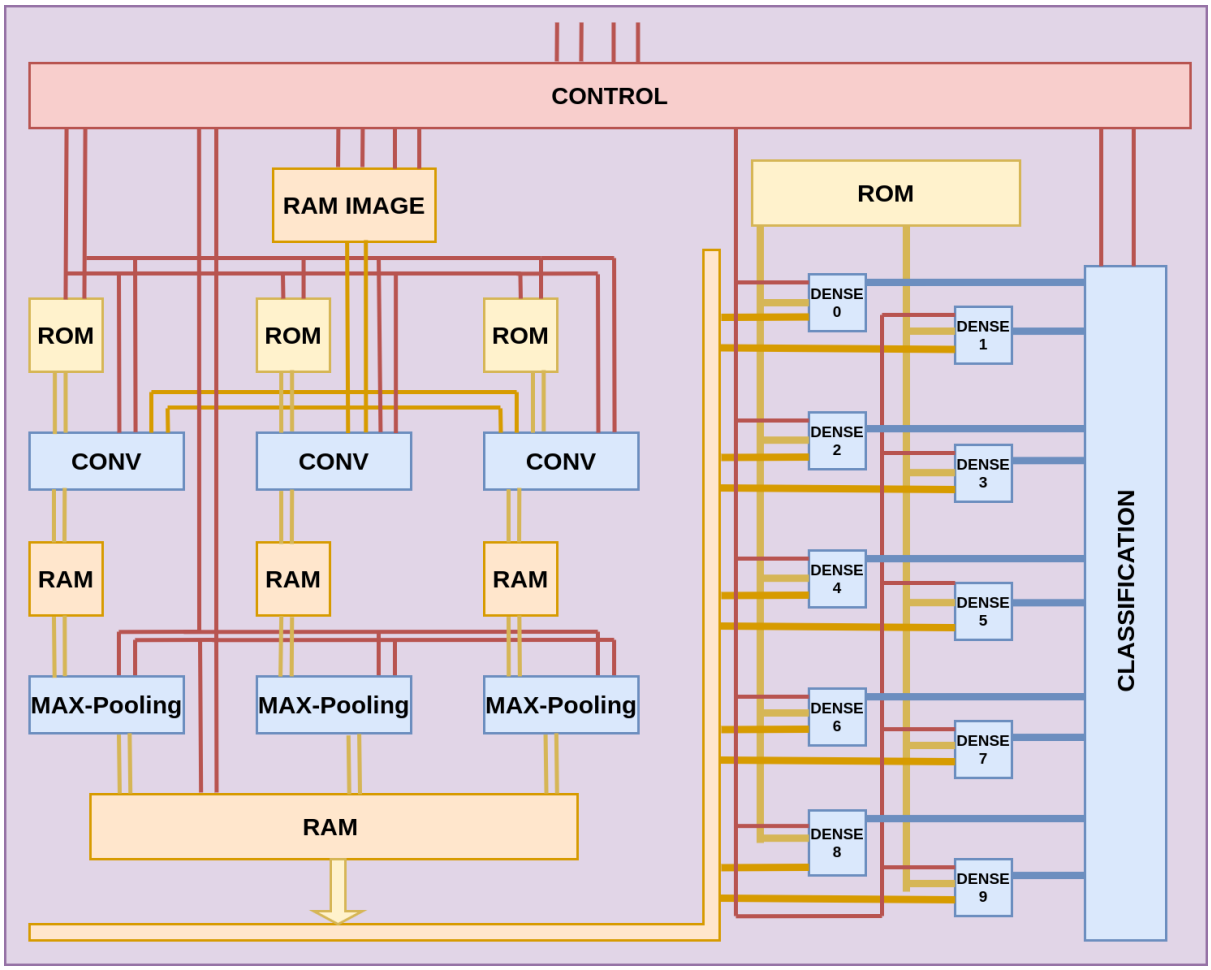

Figura 3.20: Acelerador de CNN completo .

Se busca validar el correcto funcionamiento del acelerador y al mismo tiempo que este pueda ser adaptado para futuros trabajos o aplicaciones. El acelerador se adapta a un procesador el cuál se encargara de controlarlo como se explicó al inicio de esta sección (figura 3.1) para formar un SoC.

Para esto se usa LiTeX que es un *framework* constructor de SoCs con librerías IP y usado para hacer diseños completos en FPGA. Se realiza la implementación en la FPGA *Zynq-7020* con un procesador *Vexrisc* de 32 bits con arquitectura RISC-V; a este procesador se le añade dos periféricos un UART y el del acelerador, el acelerador se comunica con el procesador a través de registros CSR como se muestra en la figura 3.21.

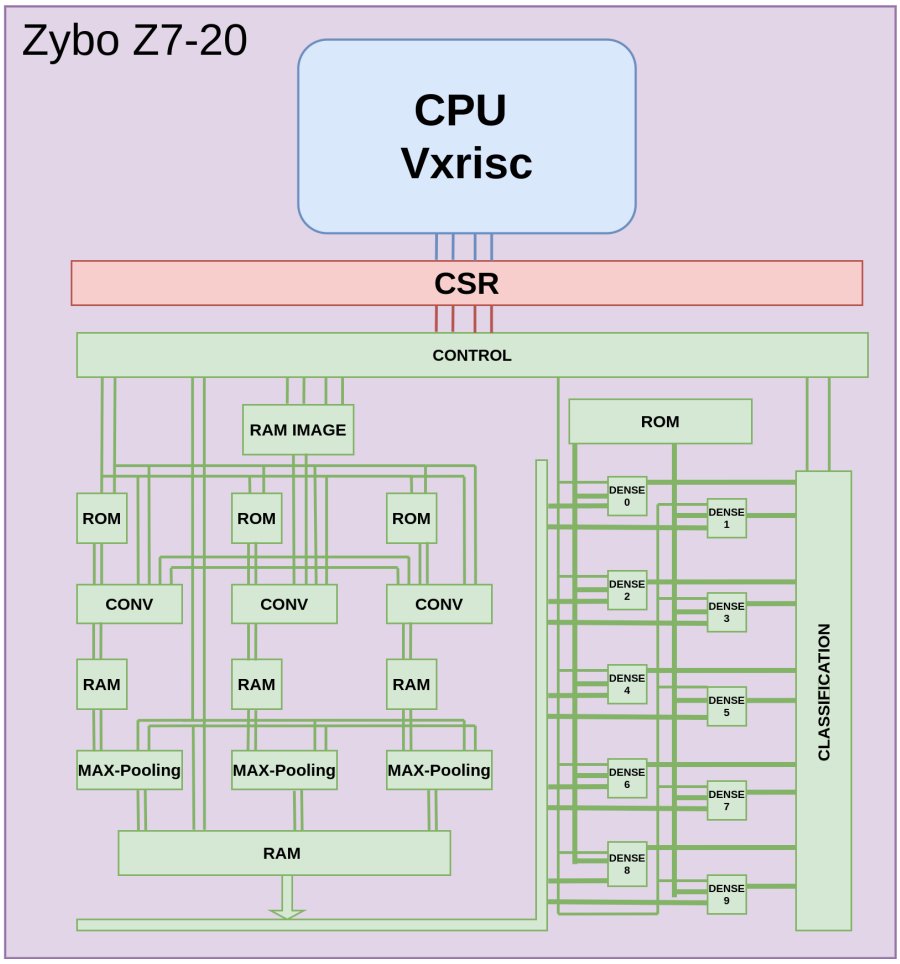

Figura 3.21: Acelerador de CNN con CPU comunicado a través de bus CSR.

El algoritmo de control en el procesador se escribe usando lenguaje *C* y consiste en recibir la imagen por puerto serial UART, luego se almacena en un vector de una dimensión, después se pasa al acelerador para que este devuelva el resultado de la clasificación. a continuación se muestra un diagrama de flujo que representa como funciona este algoritmo.

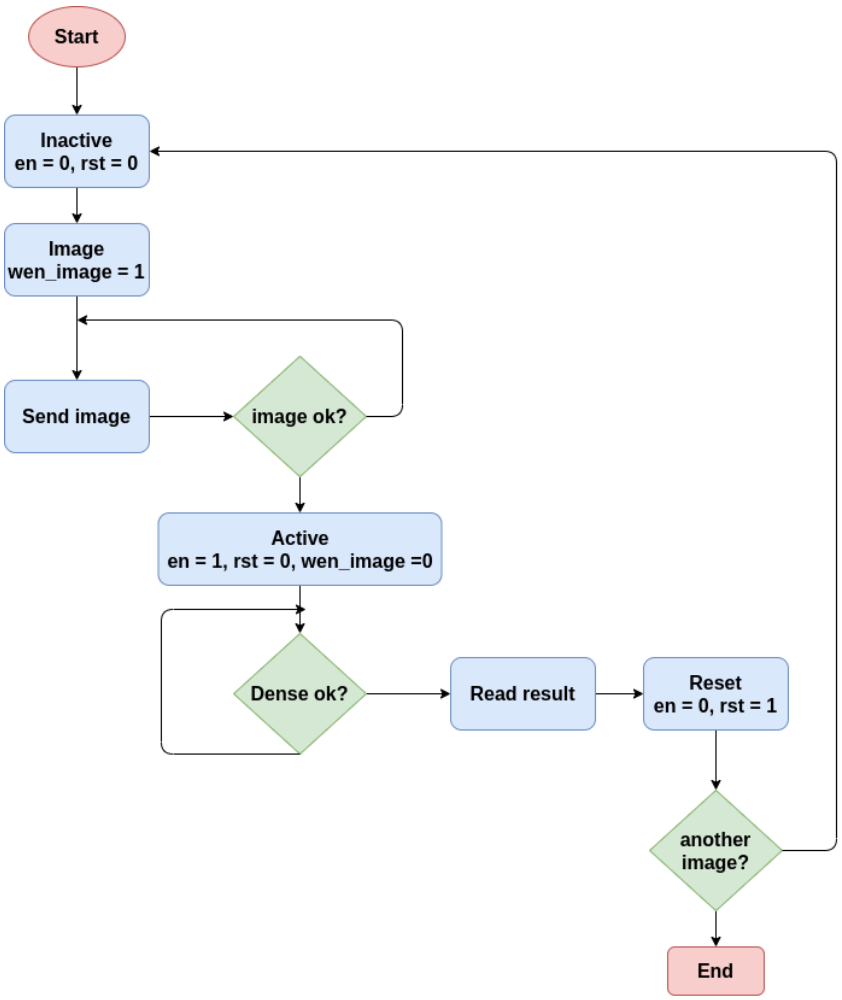

Figura 3.22: Algoritmo para el control del acelerador desde el procesador.

1. Se deja el acelerador en modo inactivo.
2. Se indica al acelerador que se va almacenar la imagen en memoria del acelerador.
3. Envía cada dato de la imagen.
4. Cuando se ha completado el envió y la imagen esta almacenada en memoria se activa el acelerador
5. Se espera hasta que termine la última operación del acelerador (*fully-connected*).
6. Lee el resultado de la clasificación.
7. Espera para repetir de nuevo el proceso.

Capítulo 4

Resultados

Este capítulo abarcara el análisis de resultados de la implementación del acelerador en la FPGA, mostrando el uso de recursos y espacio en memoria. Para el análisis se presenta la comparación entre plataformas (CPU, GPU y FPGA), con lo cual se busca tener una perspectiva del diseño de hardware para aplicaciones especializadas como lo son el diseño de redes neuronales que implica una alta complejidad computacional a cambio de una alta precisión en la inferencia.

4.1. Análisis de resultados

4.1.1. Memoria

Para la implementación de la CNN en la FPGA se evaluó en la sección 2.3.3 el costo de implementar un menor número de operaciones en la FPGA a cambio de tener menos precisión en la inferencia esto con el fin reducir los recursos usados en memoria; de la tabla 2.1 se obtuvo que al diseñar el modelo con menos de 3 *kernels* la precisión sería menos del $90\,\%$ e implementando la técnica de cuantización esta por debajo del $70\,\%$; recordar que si se implementan n número de *kernels* se tienen que hacer n convolucionales. También se evidencia que entre mayor era el número de *kernels* del modelo, cuando se implementaba la técnica de cuantización la precisión no bajaba del $90\,\%$ pero el número de parámetros aumentaba conforme al aumento de *kernels*.

	Tipo de dato	**Parametros**	**Memoria**	**Precisión**
CNN Referencia	Float 32-bit	20410	653.1 kB	95.4 %
CNN Cuantizada	Integer 8-bit	5130	4.1 kB	87.6 %

Tabla 4.1: Comparación en términos de memoria utilizada entre el modelo de referencia y el implementado en la FPGA

Así se compara el modelo implementado en la FPGA con el modelo diseñado en software en términos de precisión y espacio en memoria se aprecia que con la cuantización se tiene una reducción de aproximada $160x$ veces menos que con el modelo original con punto flotante a 32 bits, pero tiene un costo en la precisión en una reducción de aproximadamente el $8\,\%$.

4.1.2. Comparación entre plataformas

Para comparar que tan eficiente es el modelo CNN implementado en hardware se compara el modelo cuantizado a 8 bits en TensorFlow, modelo escrito en *Python* y usando la librería de *Numpy* para las operaciones, el modelo escrito en *Python / Numpy* haciendo uso de la GPU a través de *Numba* y el modelo implementado en la FPGA. Se compara el tiempo que tarda en hacer la inferencia para una imagen y la potencia que puede llegar a consumir con las diferentes plataformas de usarse a su máxima capacidad. Se debe tener en cuenta que este valor de potencia se toma de la hoja de datos de laa trajetas y para la implementación en FPGA se toma del reporte arrojado por la herramienta de *Vivado*.

	CPU	**GPU**	**CPU**	**GPU**	**FPGA**
Modelo	CNN	CNN	Q-CNN	Q-CNN	Q-CNN
Plataforma	Intel(R) i7-6700 @ 3.40GHz	GeForce GTX 750 Ti	Intel(R) i7-6700 @ 3.40GHz	GeForce GTX 750 Ti	Zynq-7020
Framenwork /Lenguaje	TensorFlow	TensorFlow	Python	Ptyhon / Numba	HDL (Verilog)
Tiempo (ms)	508.1$\pm$3	476.2 $\pm$4	71.1 $\pm$2	12.3 $\pm$3	0.190
Potencia (W)	65	60	65	60	0.057

Tabla 4.2: Comparación de la implementación del modelo con diferentes plataformas

en la tabla 4.2 se muestra los resultados el tiempo que tarda en realizar la inferencia para una imagen, el tiempo que tarda la inferencia en la FPGA es casi 2630 veces menor que lo que tarda en la CPU en TensorFlow, con respecto al uso de la GPU para su implementación se hizo uso del módulo Numba, este presenta una desventaja ya que en la practica solo se implementó parcialmente en GPU funciones que estaban relacionadas con ciclos *for* pero también se aprecia una reducción significativa frente a la CPU a pesar que se esperaba un rendimiento similar a la de la FPGA ya que en la practica lo que permite que sea más rápido que la implementación en FPGA es la paralelización de las operaciones.

4.1.3. Recursos usados y comparación con trabajos similares

El uso de los recursos de la FPGA se aprecia en la tabla 4.3, estos datos son extraidos del reporte generado por *Vivado*. De aquí se puede apreciar que la implementación del acelerador en términos de recursos el acelerador está usando un $84\,\%$ del total, mientras que el uso de recursos por parte de la CPU es mínimo comparado con el del acelerador.

	Acelerador Q-CNN	**CPU Vexrisc**	**Total usado**	**Disponible**	**Util %**
LUTs	12613	2321	14934	53200	28.07 %
Registers	6347	1938	8285	106400	7.79 %
F7 Muxes	536	0	536	26600	1.98 %
F8 Muxes	248	0	248	13300	1.88 %
Slice	3775	802	4577	13300	34.41 %
LUT as Logic	9753	2302	12058	53200	22.67 %
LUT as Memory	2860	10	2860	17400	16.53 %

Tabla 4.3: Recursos utilizados en la implementación del acelerador en la Zybo-7020

	Acelerador Propuesto	Artículo [16]	Artículo [17]	Artículo [18]
Plataforma	Zynq - 7020	xczu9eg-ffvb1 156-2-i FPGA	XC7K325T FPGA	Virtex VC707
Frecuencia (MHz)	150	——-	200	200
Modelo	1 conv 1 pooling 1 fc	3 conv 2 pooling 1 fc	3 conv 2 pooling 1 fc	2 conv 1 pooling 1 fc
Base de datos	MNIST	MNIST	MNIST	MNIST
Datos de entrada	28x28	32x32	32x32	28x28
Cuantización	Fixed-point Integer 8-bit	Fixed-point Q5.14	Fixed-point Q16.11	Fixed-point Q5.14
Precisión	87.6 %	98.64 %	98.53 %	98.66 %
Tiempo (ms)	0.190	3.58	1.043	21.27
LUTs	12613	21260	31825	55774
Potencia (W)	0.057	——-	——	1.582

Tabla 4.4: Comparación con otros trabajos.

Por último se realiza una comparación con trabajos similares encontrados en el estado del arte, para esto se buscó trabajos que implementaran modelos de CNN similares al que se realizó en este proyecto, en la tabla 4.4 se aprecia esta comparación en la que se tomó para comparar, el tipo de cuantización, el tiempo por inferencia y la cantidad de recursos utilizados. El moelo de este proyecto implementa una convolucional con 3 *Kernels*, 1 capa de *Max-pooling* y 1 capa *fully-connected* (Fc); en [1] y [2] el modelo está conformado por 3 capas convolucionales cada una con 6 *Kernels* para la primera capa convolucional y 16 para la restante, 2 capas de *Max-pooling* y 1 capa *fully-connected*; en [3] implementa 2 convolucionales una con 6 *Kernels* y la otra con 16, 2 capas de *Max-pooling* y 1 capa *fully-connected*; todos los modelos realizan una clasificación de imágenes usando la base de datos MNIST y usan representación de datos de punto fijo (*Fixed-Point*). Para este proyecto la representación es de tipo entero a 8 bits, en [1] y [3] usa 5 bits para la representacion de la parte entera y 14 para la parte decimal, en [2] 16 para la parte entera y 11 para la parte decimal.

Se aprecia que el acelerador de este proyecto tiene un uso menor de recursos en la implementación respecto a los otros trabajos al igual que el tiempo de inferencia, pero se aprecia que el costo de tener un óptimo uso de los recursos se ve representado en la precisión del acelerador ya que para los trabajos con los que se compara la precisión está alrededor del $98\,\%$, esto debido a que en estos trabajos se implemento directamente el modelo de referencia hecho en software mientras que en este proyecto, como se explicó en la sección 2.2 se redujo sacrificando precisión. También se puede apreciar que esta forma de cuantización representa una forma de óptima de implementar redes neuronales en FPGA ya que no se tiene que hacer uso de módulos u operaciones adicionales para números con parte decimal. También se aprecia que el número de operaciones es menor en este proyecto y esto se debe a factores tales como la representación numérica, el número de operaciones y la paralelización que se explicó en la sección 3.3 de las operaciones.

4.2. Conclusiones y trabajo futuro

El diseño de redes neuronales cuantizadas presenta ventajas para la implementación en hardware como la reducción del tiempo de ejecución en las operaciones y el bajo consumo de energía lo que lo hace una alternativa viable para realizar sistemas embebidos en los que existe un procesador y se puede añadir un acelerador como se mostró en la sección 3.6. También se aprecia que debido a la metodología de continuación implementada en este trabajo las operaciones y las representaciones de los valores es de tipo entero de 8 bits con lo cuál se logra una reducción en la complejidad de las operaciones y menor uso de la memoria para almacenar los pesos, pero se tiene un costo en la precisión, así que lo ideal para futuras aplicaciones es buscar un equilibrio entre el uso de recursos disponibles y la precisión que se desee lograr del modelo al momento de su implementación en hardware.

El diseño de este tipo de redes en FPGA esta direccionado a aplicaciones especificas como desventaja de este tipo de diseños es que no se puede generalizar y siempre va depender de la aplicación que se quiera implementar pero si se puede generalizar la metodología de cuantización la cuál como se vio en el proyecto está esta ligada a su diseño en software. Para lograr un uso general y reutilizable de los módulos realizados en el proyecto para las diferentes capas conviene que estos diseños se puedan parametrizar y así se pueda escalar para diferentes aplicaciones.

El método de cuantización para el desarrollo se baso del modelo diseñado en TensorFlow y luego para obtener los parámetros cuantizados se hizo uso del modulo de TensorFlow Lite, como trabajo futuro queda explorar como realizar el algoritmo de entrenamiento cuantizado para extraer los parámetros sin depender de herramientas externas como TensorFlow y como implementar el entrenamiento con sus respectivas operaciones y variables cuantizadas en hardware. Adicional como trabajo futuro se podría proponer una mejora en el uso de las memorias RAM haciendo uso solo de dos memorias una de entada y una de salida esto con el fin de quitar las memorias intermedias y así reducir el uso de recursos, incluso puede extenderse haciendo uso de memorias externas que no estén ligadas necesariamente al acelerador y en vez simplemente se comunique con las memorias a través de un protocolo de comunicación.

Bibliografía

[1] [Online]. Available: https://www.tensorflow.org/datasets/catalog/mnist

[2] S. H. David Harris, *Digital Design and Computer Architecture*, 2007. [Online]. Available: http://gen.lib.rus.ec/book/index.php?md5=ebff8085ab88f864e6130720897afac0

[3] V. T. Phat, P. H. Tho, H. B. Dat, and C.-H. Chou, "Deep learning accelerator on fpga using handwritten digit recognition for example," in *2018 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW)*, 2018, pp. 1–2.

[4] T. Abtahi, C. Shea, A. Kulkarni, and T. Mohsenin, "Accelerating convolutional neural network with fft on embedded hardware," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 9, pp. 1737–1749, 2018.

[5] K. Zhao, T. He, S. Wu, S. Wang, B. Dai, Q. Yang, and Y. Lei, "Application research of image recognition technology based on cnn in image location of environmental monitoring uav," *EURASIP Journal on Image and Video Processing*, vol. 2018, p. 150, 12 2018.

[6] I. Sutskever, O. Vinyals, and Q. Le, "Sequence to sequence learning with neural networks," *Advances in Neural Information Processing Systems*, vol. 4, 09 2014.

[7] S. A. Z. U. e. a. Khan, A., "A survey of the recent architectures of deep convolutional neural networks," *Artificial Intelligence Review*, vol. 53, p. 5455–5516, 04 2020.

[8] T. Wang, C. Wang, X. Zhou, and H. Chen, "An overview of fpga based deep learning accelerators: Challenges and opportunities," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019, pp. 1674–1681.

[9] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–4.

[10] A. G. Blaiech, K. Ben Khalifa, C. Valderrama, M. A. Fernandes, and M. H. Bedoui, "A survey and taxonomy of fpga-based deep learning accelerators," *Journal of Systems Architecture*, vol. 98, pp. 331–345, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762118304156

[11] A. X. M. Chang and E. Culurciello, "Hardware accelerators for recurrent neural networks on fpga," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

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

[13] F. Chollet, *Deep Learning with Python*. Manning, 2018. [Online]. Available: http://gen.lib.rus.ec/book/index.php?md5=deb175581d4a7579c3654f0a0862b5e2

[14] [Online]. Available: https://www.tensorflow.org/datasets/keras_example

[15] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” 06 2018, pp. 2704–2713.

[16] M. Cho and Y. Kim, “Implementation of data-optimized fpga-based accelerator for convolutional neural network,” in *2020 International Conference on Electronics, Information, and Communication (ICEIC)*, 2020, pp. 1–2.

[17] X. Zhen and B. He, “Research on fpga high-performance implementation method of cnn,” in *2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP)*, 2021, pp. 1177–1181.

[18] A. Kyriakos, V. Kitsakis, A. Louropoulos, E.-A. Papatheofanous, I. Patronas, and D. Reisis, “High performance accelerator for cnn applications,” in *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2019, pp. 135–140.