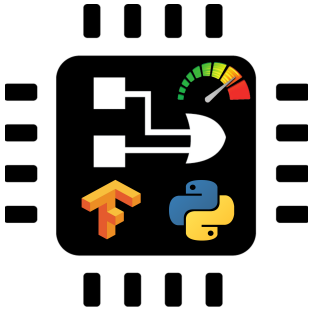
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

Implementación e integración sobre Keras tensorflow de capas neuronales desarrolladas con OpenCL-Verilog implementadas sobre FPGA



Autor: Diego Ruiz Quintana

Tutor: Rafael Gadea Gironés

Cotutor: Fulgencio Montilla Meoro

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2019-20

Valencia, 10 de septiembre de 2020

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universitat Politècnica de València
Edificio 4D. Camino de Vera, s/n, 46022 Valencia
Tel. +34 96 387 71 90, ext. 77190
www.etsit.upv.es

VLC/CAMPUS
VALENCIA, INTERNATIONAL CAMPUS OF EXCELLENCE

EMAS Gestión ambiental verificada

Resumen

Las tecnologías y sistemas de inteligencia artificial están cada vez mas inmersos en nuestra día a día, esto, unido al creciente volumen de datos y las nuevas tecnologías aparecidas, como el 5G, hacen que los algoritmos de Deep y Machine Learning tengan que ser mas eficientes para dar una respuesta en un tiempo aceptable. El compromiso en este punto, el tiempo de respuesta del algoritmo, dificilmente puede lograrse utilizando únicamente técnicas de implementación software, teniendo que hacer uso de técnicas de aceleración de algoritmos mediante FPGA u OpenCL a través de GPU's. La arquitectura de una GPU, dada la naturaleza de los procesos que realiza (traslaciones y re-escalados en mayor medida) es idónea para la implementación de algoritmos para redes neuronales, ya que estas no dejan de ser multiplicaciones y sumas con estructuras de matriz, similares a las operaciones de traslación y re-escalado. El objetivo de este TFG se centrará en la optimización de algoritmos para redes neuronales, centrándose en el producto de matrices, una operación de gran importancia y muy extendida en todos los ámbitos de la computación.

Abstract

The technologies and systems of analytic intelligence are constantly being integrated into our day-to-day work, this, together with the growing volume of data and new technologies The compromise at this point, the response time of the algorithm, can hardly be achieved using only software implementation techniques, having to make use of algorithm acceleration techniques through FPGA or OpenCL through GPUs. The architecture of a GPU, given the nature of the processes it performs (translations and re-scaled to a greater extent) is ideal for the implementation of algorithms for neural networks, since these do not stop being multiplications and additions with matrix structures, similar to translation and rescaling operations. The objective of this TFG will focus on the optimization of algorithms for neural networks, focusing on the product of matrices, an operation of great importance and widespread in all areas of computing.

Resumen

Les tecnologies i sistemes d'intel · ligència artificial estan cada vegada mes immersos en nostra dia a dia, això, unit al creixent volum de dades i les noves tecnologies aparegudes, com el 5G, fan que els algorismes de Deep i Machine Learning hagen de ser mes eficients per a donar una resposta en un temps acceptable. El compromís en aquest punt, el temps de resposta de l'algorisme, dificilmente pot aconseguir-se utilitzant únicament tècniques d'implementació programari, havent de fer ús de tècniques d'acceleració d'algorismes mitjançant FPGA o OpenCL a través de GPU's. L'arquitectura d'una GPU, donada la naturalesa dels processos que realitza (translacions i re-escalats en major mesura) és idònia per a la implementació d'algorismes per a xarxes neuronals, ja que aquestes no deixen de ser multiplicacions i sumes amb estructures de matriu, similars a les operacions de translació i re-escalat. L'objectiu d'aquest TFG se centrarà en l'optimització d'algorismes per a xarxes neuronals,

centrant-se en el producte de matrius, una operació de gran importància i molt estesa en tots els àmbits de la computació.

Índice

Lista de códigos

Lista de tablas

Lista de figuras

Capítulo 1. Estado del arte y motivación

1.1 Motivación

No fue hasta 4º de carrera donde tuve realmente contacto con un sistema de inteligencia artificial a través de la asignatura de *Sistemas Complejos Bioinspirados*, y sin duda que me fascinó por completo. El hecho de que un programa informático sea capaz de sacar respuestas correctas sin indicarle nada mas que una entrada y una salida es algo que todavía me sigue pareciendo increíble, puesto que hasta el momento consideraba que la computación no era capaz de ir mas allá de lo que un humano le indicara que tenía que hacer. Toda esta incertidumbre que había detrás del proceso que hacía la "Magia"poco a poco se fue desvelando a medida que entraba en materia y se sucedían las clases. Realmente, al final resultó ser que detrás de todo esto no había mas que pura matemática y procesos estocásticos que corregían y adaptaban el "sistema inteligente", esto todavía causó en mi mas sorpresa, aunque claro, no podía ser de otra manera mas que a través de las matemáticas, el lenguaje universal que todo lo modela y todo lo entiende, citando a Rafael Sánchez Grandia, profesor de física de primero, en una conversación me dijo: *"Hay dos lenguajes universales, la música y las matemáticas, y la música está compuesta por matemáticas"*, ergo, no es difícil llegar a la conclusión de que la máxima expresión de cualquier proceso son las matemáticas.

La cosa, llegados a este punto, parecía simple: coger un algoritmo, ejecutarlo y la máquina hará el trabajo. Si el problema a resolver es sencillo, no tardará mas de unos milisegundos en ejecutar su tarea, aunque pocos son los problemas reales a resolver que no sean complejos. Esta última condición suele ir ligada a una gran cantidad de datos, y, en consecuencia, una gran carga computacional; lo que desemboca en tiempos de ejecución largos. Supongamos que se implementa un algoritmo de inteligencia artificial para detección de viandantes en un coche (como realmente sucede), en este caso, el tiempo de respuesta es primordial, de nada servirá detectar al sujeto cuando en el peor de los casos este haya sido atropellado, por ejemplo. ¿Qué hacer en esta situación? esta condición exige y presenta la segunda motivación fundamental de este TFG: la aceleración de algoritmos para inteligencia artificial a través de hardware dedicado, ya sea GPU o FPGA, eligiendo para en este caso FPGA mediante OpenCL.

Los sistemas FPGA son capaces de trabajar en altas prestaciones con procesos matemáticos, como el producto de matrices, altamente involucrado en cualquier algoritmo computacional. La estructura de una FPGA, en su mayoría, son multiplicadores y sumadores, las operaciones básicas a utilizar para resolver el producto matricial, parece entonces idónea la elección realizada. También se utilizarán librerías de alto nivel como *"Tensorflow"* o *"Keras"* para la implementación de redes neuronales, derivando aquellas operaciones mas costosas a la FPGA.

1.2 Revisión histórica y estado del arte

No fue hasta el año 1940 cuando McCulloch & Pitt comenzaron la investigación de las redes neuronales, aunque sin éxito. Ya en 1950, Allan Turing a través de su famoso Test de Turing impuso las bases de lo que comenzaría a ser la inteligencia artificial. Este test determina que si una máquina ejecutando una tarea no es advertida como máquina, entonces es satisfactorio el test y podemos hablar de inteligencia artificial, aunque todavía estábamos lejos de obtener algo que realmente se asemejara al test de Turing. En 1958 se desarrollo el "Perceptrón"por Rosenblatt, esta estructura tiene toda la vigencia hoy en día y es el principio fundamental para una red neuronal. Pero sin duda, el avance mas significativo en el campo se dio en 1974 con el desarrollo del algoritmo de "Backpropagation"de la mano de Werbos y More. Es el algoritmo mas utilizado para las redes neuronales de aprendizaje guiado, básicamente consiste en introducir un estímulo y comprobar si la salida al estimulo es correcta, generando una señal de error que se propagará por las neuronas de las capas anteriores que hayan trabajado activamente en la respuesta de salida. Sin duda este ha sido uno de los hecho mas significativos en la evolución de los sistemas de inteligencia artificial. El siguiente crono-grama ilustra el camino recorrido hasta la actualidad:

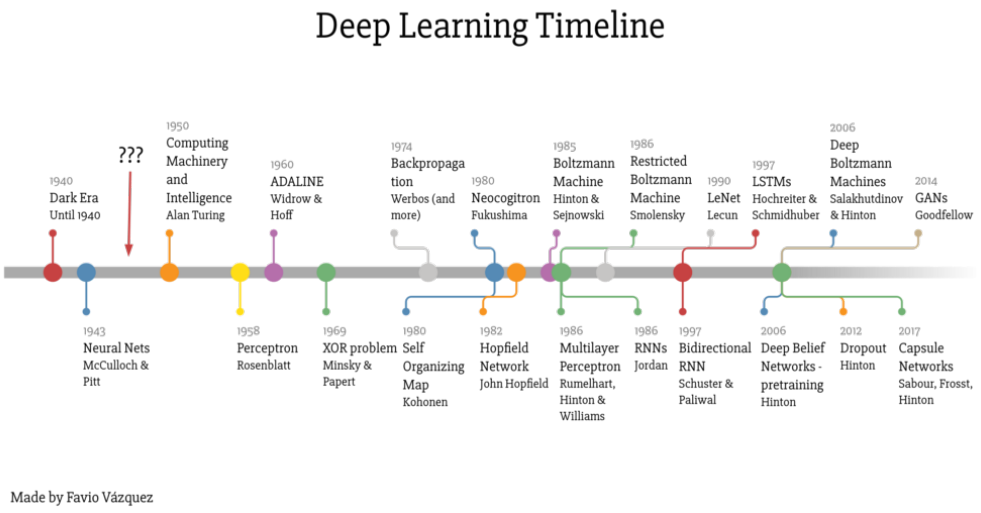


Figura 1: Cronología del Deep Learning

Como se aprecia, no han sido pocos los desarrollos y avances sucedidos en el campo desde 1974 hasta la actualidad. Hoy en día, los sistemas de inteligencia artificial están inmersos en un sin fin de tareas que pasan desadvertidas, desde el análisis de datos en el sector del marketing hasta los sistemas Bio-Médicos pasando por los sectores de la automovilística o domótica. De los más destacados es el procesamiento del lenguaje na-

tural, que ha experimentado una inmensa evolución en los últimos tres años. También los altavoces inteligentes parecen ser tendencia, como Alexa de Amazon o Google Home. Algo que parece impresionante también es la utilización de IA en el sector Retail, donde Amazon propone sistemas que permitan reconocer cuándo cada usuario coge e introduce en la cesta un determinado producto, convirtiendo a los cajeros y a muchos empleados en obsoletos. Cuando terminas de hacer la compra, simplemente sales de la tienda y el recargo se efectúa de forma automática en tu cuenta de Amazon, sin tener que hacer colas ni escanear ningún artículo en la salida. En el sector de la salud la IA tendrá acceso a nuestros registros médicos electrónicos y estará implicada en casi todos los avances en neurociencia, genética, diagnóstico de cáncer y sistemas de diagnóstico precoz de los próximos años. La IA puede predecir los resultados de los pacientes ayudando a la investigación y desarrollo de fármacos y productos farmacéuticos. Las cirugías robóticas y automatizadas continuarán aumentando, y a medida que las grandes empresas de tecnología se impliquen más en el cuidado de la salud, nuestros datos de salud se utilizarán para innovaciones específicas para la IA... y estos son solo algunos de los ejemplos que nos depara el futuro en este campo.

Enfocándonos en el propósito de este TFG, las compañías dedicadas al silicio están comenzando a producir sistemas hardware dedicados de forma exclusiva a sistemas de IA, como remarca la página *puentesdigitales.com* en este artículo:

"... Otra carrera, en este caso positiva, es la de los chips especializados para IA. Las empresas de tecnología están pasando por alto a los fabricantes de chips tradicionales y están creando los suyos propios. Según Arstechnica, mientras que Intel entró en el mercado con la compra de Nervana Systems en 2016 y compró una segunda compañía, Movidius, para el procesamiento de imágenes con IA, las principales empresas tecnológicas están construyendo sus propios chips. Microsoft está preparando un chip de IA para sus auriculares HoloLens VR/AR, y hay potencial para su uso en otros dispositivos. Google ya tiene un chip de IA especial para redes neuronales llamado Tensor Processing Unit, o TPU, que está disponible para aplicaciones de IA en la plataforma Google Cloud Platform. Amazon está trabajando en un chip de IA para su asistente de Alexa. Apple está trabajando en un procesador de IA llamado Motor Neural que alimentará a Siri y FaceID. ARM Holdings recientemente introdujo dos nuevos procesadores, el procesador ARM Machine Learning (ML) y el procesador ARM Object Detection (OD). Ambos se especializan en el reconocimiento de imágenes. IBM está desarrollando un procesador específico de IA, y la empresa también ha concedido una licencia NVLink de Nvidia para el procesamiento de datos de alta velocidad específicos de IA y ML. Huawei ya tiene sus chips de IA desde 2018 y Alibaba va a sacar los suyos en 2019.

En cualquier caso en 2019, los fabricantes de chips como Intel, NVIDIA, AMD, ARM y Qualcomm enviarán chips especializados que acelerarán la ejecución de aplicaciones habilitadas para la IA, incluso cuando las empresas de tecnología construyan las suyas propias. Va a ser un año de un crecimiento enorme... "

1.3 Objetivos

En el presente TFG se van a implementar algunos de los algoritmos que componen un sistema de red neuronal en un dispositivo FPGA. Los dispositivos FPGA son dispositivos electrónicos totalmente configurables y programables. Una de las características mas interesantes es la paralelización de operaciones, pudiendo realizar en un solo ciclo mas de una operación. Esta es una de las razones por las que se ha decidido plantear todo el problema desde una perspectiva de producto matricial, es decir: se intentará que todos los algoritmos implementados hagan uso del producto matricial en la medida de los posible. Esta condición es la que define un sistema GEMM: General matrix multiply. Algo que en un principio no estaba planteado en el TFG pero que finalmente se ha tenido en cuenta dado el éxito obtenido al plantear diversos problemas, como el tratamiento de imagen, a través de productos matriciales. A través del producto matricial se compararán los tiempos obtenidos por la FPGA y las librerías de Keras y Numpy, con la intención de intentar superar estos. Para intentar superar con éxito los tiempos de las librerías de Keras se seguirá la estrategia *Divide y vencerás* como se verá con el primer algoritmo a analizar: algoritmo de producto matricial mediante *BLOCK_SIZE*, a partir de ahora BS. Otro parámetro que influirá en la paralelización del algoritmo sera el *SIMD_WORK_ITEMS*: este parámetro modela el número de operaciones de BS que se computan de forma paralela. Este parámetro de designará a lo largo del TFG como SWI.

Un aspecto importante a tener en cuenta es la frecuencia a la que funciona el PC donde se realizarán las pruebas, tratándose de un Intel I5 con una frecuencia de 1.66 MHz y una memoria RAM de 8 Gb. Estas prestaciones son mayores que las de la FPGA, la cual tiene solo 1 Gb de RAM y una frecuencia de reloj dinámica en función de la implementación del kernel. Aún así, se suponen superiores las prestaciones del PC host al de la FPGA, por lo que a la hora de analizar las comparativas en los tiempos de computación se habrá de tener en cuenta esta condición.

Otro aspecto que no estaba contemplado pero que finalmente se ha implementado ha sido la creación de un GUI para la migración de los modelos de red neuronal de Keras Tensorflow a la FPGA. Este GUI ayuda al usuario evitando que tenga que ser escrito código para llevar a cabo la migración. El GUI ha sido realizado mediante PyQt en python. Para agilizar la interacción entre host (python) y dispositivo (OpenCL) se hará uso de las librerías pyopencl, evitando así scripting en C++.

Los algoritmos a implementar son:

- Producto matricial: Operación matemática básica para la propagación de la señal entre las capas densas de una red neuronal.
- Funciones de activación y adición del bias: Las funciones de activación y el bias ayudan a regular el proceso de aprendizaje de la red neuronal y suelen ser aplicadas

generalmente en capas de tipo densa.

- Capas convolucionales: La convolución es una operación matemática utilizada, entre otras cosas, para filtrar señales, ya sean estas de 1 dimension, 2 o mas. Además de la convolución propiamente dicha, otros métodos como el reescalado de imágenes o filtrado medio pueden ser logrados aplicando la convolución y el kernel adecuado. Se han abordado 2 formas de realizar la convolución: mediante matriz de *Toeplitz* y mediante la técnica *Im2Row*. Como se explicará y demostrará mas adelante, la técnica *Im2Row* es la más optima.
- Capas de padding: El padding es el proceso por el cual una señal es rellenada con ceros para que las dimensiones sean las optimas en un proceso que se vaya a realizar posteriormente. Por ejemplo: la convolución de tipo *same* utiliza el padeo en los extremos de la señal para que la señal convolucionada tenga las mismas dimensiones que la original, evitando así perdida de información. Este tipo de capas no se realizará mediante producto matricial si no mediante indexamiento, habiendo precalculado los índices necesarios en el host.

Todo este proceso conllevará la creación de un entorno virtual y librerías específicas que cumplan los requisitos de los algoritmos. Estas librerías serán escritas en python y OpenCL. El flujo necesario será:

- Creación de entorno virtual "Tensorflow - FPGA".
- Instalación de librerías referentes a *Keras*, *Tensorflow* y *pyOpenCL*.
- Creación de las librerías necesarias para migrar estructura de NN a la FPGA.
  1. Scripts en python.
  2. Creación de kernel/s para ejecución de lógica en la FPGA.
- Creación del GUI para migración de modelos de red neuronal.

Se intentará desarrollar una metodología ágil, tanto para abarcar el máximo número de pruebas experimentales a la vez que se adquieren nuevas habilidades y competencias para el futuro. Se propone trabajar con 2 plataformas distintas que abarcan desde la verificación y validación software hasta la implementación hardware:

- Spyder: scripts para python con las librerías de *Tensorflow*, *Keras* y *pyOpenCL*.
- OpenCL: Implementación de algoritmos en hardware GPU.

La elección de estos dos es elementos es dada su facilidad para poder realizar pruebas experimentales a la vez que depurar y analizar estadísticas que puedan ser de interés para conseguir la mayor optimización posible. Se explica a continuación el papel que jugará cada uno en el presente TFG.

1.3.1 Spyder

IDE para desarrollo en Python. A pesar de ser un IDE sencillo, ofrece una gran versatilidad para definir en primera instancia algoritmos que puedan resultar mas complejos de modelar en C. Python ofrece librerías con un alto *performance* para cálculo científico y numérico como es el caso de *Numpy* (he de añadir en este punto, que después de intentar desentrañar el método *matmul* para producto de matrices, mi sorpresa fue que me encontré con una librería ampliamente escrita en C/C++) así como otras librerías para cálculo de tiempos, *time* o graficación de funciones: *pyplot*. Otro de los aspectos mas interesantes de Python y Spyder es su total integración con las librerías de Tensorflow y Keras. Estas dos librerías servirán para lanzar los algoritmos de I.A, agilizando así el desarrollo y posterior análisis de algoritmos de optimización. Resta decir que es nativo de la plataforma GNU/Linux.

1.3.2 OpenCL

Citando a Wikipedia[1] *OpenCL (Open Computing Language, en español lenguaje de computación abierto) consta de una interfaz de programación de aplicaciones y de un lenguaje de programación. Juntos permiten crear aplicaciones con paralelismo a nivel de datos y de tareas que pueden ejecutarse tanto en unidades centrales de procesamiento como unidades de procesamiento gráfico. El lenguaje está basado en C99, eliminando cierta funcionalidad y extendiéndolo con operaciones vectoriales*

Utiliza todo el potencial que ofrece un lenguaje compilado mas el añadido de operaciones vectoriales, que no está muy desarrollado en C. Estas operaciones vectoriales bien podrían ser testadas con C++. Es posible, por ejemplo, realizar *Slicing* sobre los vectores y las matrices o cargar en memoria registros a través de lotes, sin tener que iterar todas las posiciones mediante un bucle.

Capítulo 2. Arquitectura FPGA y algoritmo de Block Size propuesto por Intel FPGA

2.1 Arquitectura FPGA y configuración

La FPGA utilizada en este TFG es una *Cyclone V GT*. Una de las características novedosas del dispositivo es la incorporación del puerto PCI express. A través de el se comunican host y dispositivo a altas velocidades, pudiendo hacer uso el dispositivo de la memoria RAM del host mediante protocolo DMI. Las principales características de la FPGA son:

- 301K puertas lógicas programables.
- 13.917 Kbits de memoria embebida.
- 8 PLL.
- 64MB SDRAM.
- 1GB SDRAM.
- UART to USB (Mini-B USB connector): Necesario para cargar el kernel en la FPGA. Se tratará de un objeto compilado con extensión .aocx.
- GT device supports PCIe Gen2x 4 (GX device supports PCIe Gen1x 4).

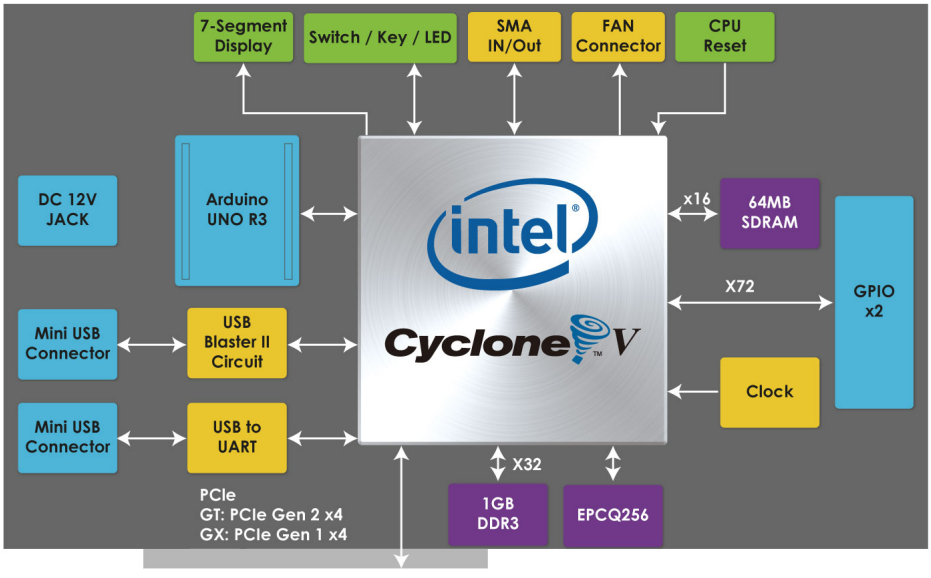


Figura 2: Arquitectura Cyclone V

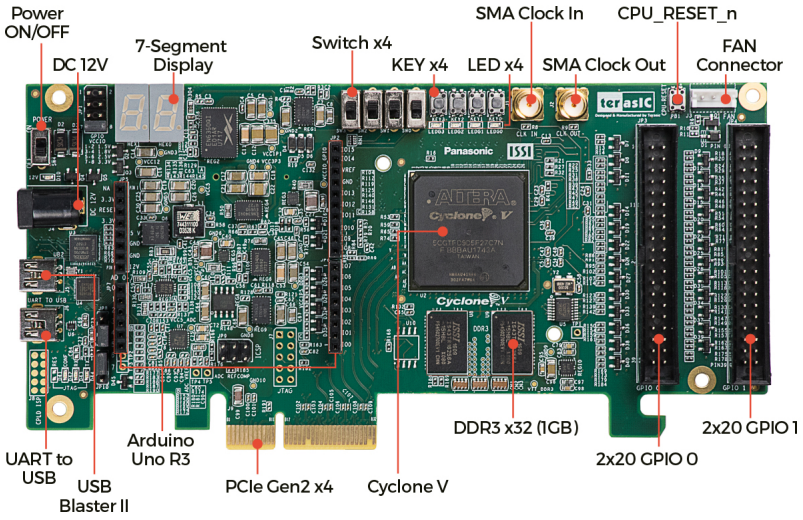


Figura 3: Layout Cyclone 5 top

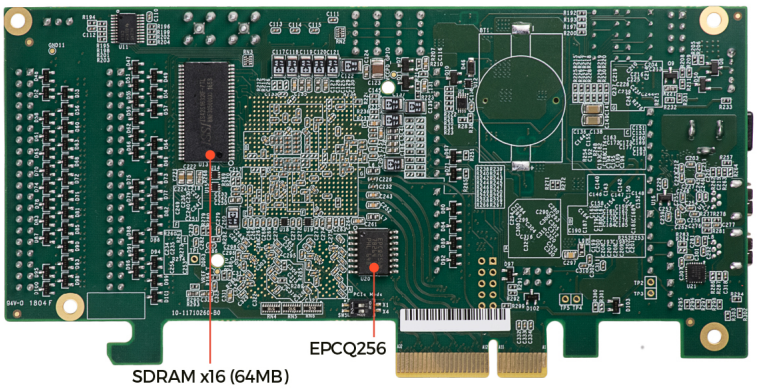


Figura 4: Layout Cyclone 5 bottom

Para comprobar que la FPGA es detectada correctamente por el host y es capaz de enviar y recibir datos se ejecuta el comando `aocl diagnose acl0`, donde `acl0` hace referencia al dispositivo detectado en el host. Es posible tener varios dispositivos en un mismo host y distribuir las operaciones a través de ellos. La siguiente imagen muestra como la transmisión de datos entre host y dispositivo ha sido correcta:

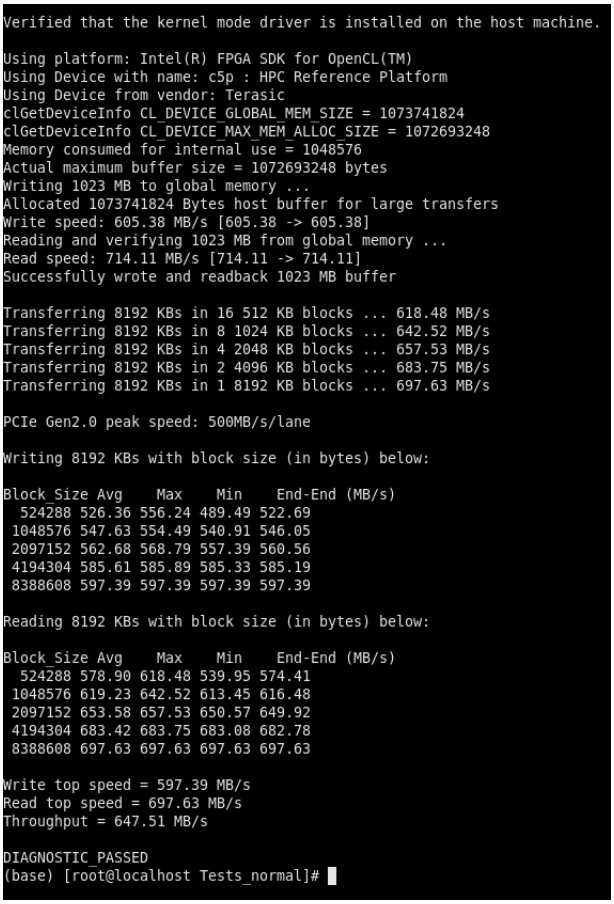


Figura 5: Layout Cyclone 5 bottom

2.2 Algoritmo de BLOCK_SIZE propuesto por Intel FPGA

El cálculo de las salidas de cada capa de la red neuronal puede modelarse como el producto de dos matrices, siendo la matriz A el input referente a esa capa y la matriz B la matriz asociada a los pesos. Dada la importancia del producto matricial en una estructura de NN es importante tener un algoritmo de producto matricial con buenas marcas. A lo largo del siglo XX se ha intentado reducir el número de operaciones necesarias para resolver el producto matricial, siendo el algoritmo de *Strassen* el primero que determinó que había formas de bajar la complejidad del producto matricial a un factor menor que $O(N)^3$, concretamente a:

$$Strassen_{complex} = O(N^{log_2 7+o(1)}) \approx O(N^{2.8074}) \tag{1}$$

La siguiente figura muestra diferentes algoritmos de producto matricial y su complejidad asociada:

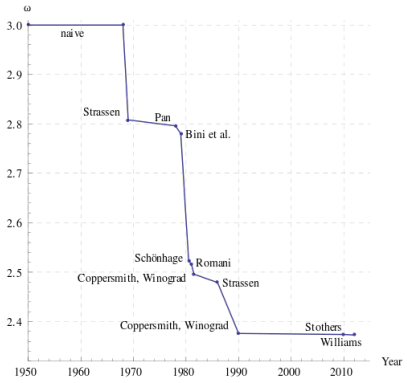


Figura 6: Complejidad para diferentes algoritmos de producto matricial

El producto matricial puede paralelizarse, puesto que cada valor de la matriz resultado es el producto punto de cada fila y columna de las matrices A y B; se cumple que hay independencia entre los valores de la matriz resultado. Es en este punto es donde es posible paralelizar el cómputo de estos valores, pudiendo obtener varios resultados simultáneamente.

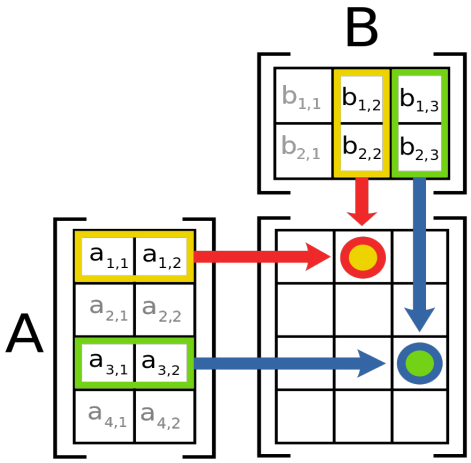


Figura 7: Representación de lo obtención de los valores de la matriz resultado

$$C = AB := (c_{ij})_{mxp} \tag{2}$$

$$c_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj} \tag{3}$$

Intel FPGA propone como punto de partida un algoritmo basado en BS [2]. Este BS indica el número de valores que se trabajan simultáneamente en un solo golpe de reloj. Por ejemplo: un BS=16 implica que se trabaja con bloques de 16x16=256 valores. Una segunda variable asociada al algoritmo es el SWI[3]; esta variable indica el número de bloques que se trabajan simultáneamente, es decir, si se tiene un SWI=4 quiere decir que se trabajan 4 bloques de 16x16, es decir: un total 1024 operaciones realizadas en un solo golpe de reloj. Como puede apreciarse, este algoritmo acelera el cálculo a través de la paralelización. Finalmente es importante remarcar que el algoritmo hace uso de memoria global[4] y memoria local[5]. Los accesos a memoria local son mucho mas rápidos que a la global puesto que esta se encuentra embebida en la FPGA, mientras que la global hace referencia a la memoria RAM del PC y por tanto es necesario transferir los datos desde el host (PC) al dispositivo (FPGA) mediante PCI a través del protocolo DMA[1]. Otra de las características interesantes a tener en cuenta es la separación de los bancos de memoria, de esta forma el acceso a los mismos es concurrente.

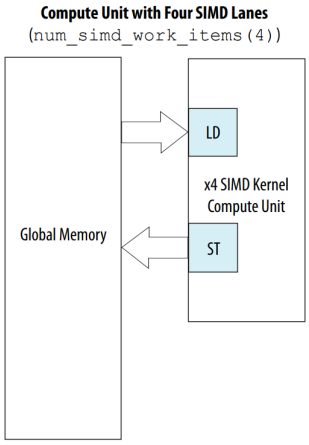


Figura 8: Flujograma para SIMD > 1

[1]El acceso directo a memoria (DMA, del inglés direct memory access) permite a cierto tipo de componentes de una computadora acceder a la memoria del sistema para leer o escribir independientemente de la unidad central de procesamiento (CPU) principal

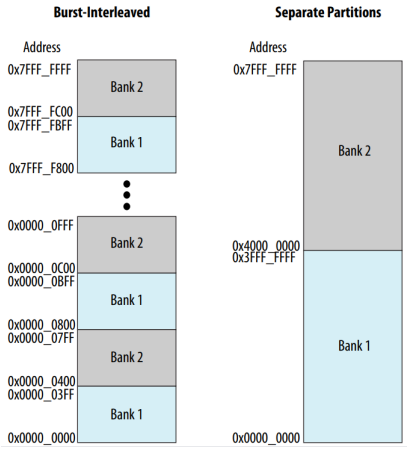


Figura 9: Organización de la memoria no separada Vs separada

El código siguiente hace referencia al kernel original propuesto por Intel sobre el que mas adelante se añadirán las funciones de activación.

```
1
2  // Definición de bloques a procesar en paralelo
3  #ifndef SIMD_WORK_ITEMS
4  #define SIMD_WORK_ITEMS 4 // default value
5  #endif
6
7  __kernel
8  __attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
9  __attribute((num_simd_work_items(SIMD_WORK_ITEMS)))
10 void matrixMult( // Input and output matrices
11                  __global float *restrict C,
12                  __global float *A,
13                  __global float *B,
14                  // Widths of matrices.
15                  int A_width, int B_width)
16 {
17     // Local storage for a block of input matrices A and B
18     __local float A_local[BLOCK_SIZE][BLOCK_SIZE];
19     __local float B_local[BLOCK_SIZE][BLOCK_SIZE];
20
21     // Block indexA
22     int block_x = get_group_id(0);
23     int block_y = get_group_id(1);
24
25     // Local ID index (offset within a block)
26     int local_x = get_local_id(0);
27     int local_y = get_local_id(1);
28
29     // Compute loop bounds
30     int a_start = A_width * BLOCK_SIZE * block_y;
31     int a_end   = a_start + A_width - 1;
32     int b_start = BLOCK_SIZE * block_x;
```

```
33
34      float running_sum = 0.0f;
35
36      // Compute the matrix multiplication result for this output element.
        Each
37      // loop iteration processes one block of the matrix.
38      for (int a = a_start, b = b_start; a <= a_end; a += BLOCK_SIZE, b +=
        (BLOCK_SIZE * B_width))
39      {
40          // Load the matrices to local memory. Note that the (x, y)
        indices
41          // are swapped for A_local and B_local. This affects the reads
        from
42          // A_local and B_local below and result in more efficient
        hardware.
43          A_local[local_y][local_x] = A[a + A_width * local_y + local_x];
44          B_local[local_x][local_y] = B[b + B_width * local_y + local_x];
45
46          // Wait for the entire block to be loaded.
47          barrier(CLK_LOCAL_MEM_FENCE);
48
49          // Do the dot product accumulation within this block. Fully
        unroll the loop.
50          // As a result of the swap of indices above, memory accesses to
51          // A_local and B_local are very efficient because each loop
        iteration
52          // accesses consecutive elements. This can be seen by unrolling
        the
53          // loop and analyzing the regions that are loaded:
54          //   A_local[local_y][0..BLOCK_SIZE-1] and
55          //   B_local[local_x][0..BLOCK_SIZE-1]
56          #pragma unroll
57          for (int k = 0; k < BLOCK_SIZE; ++k)
58          {
59              running_sum += A_local[local_y][k] * B_local[local_x][k];
60          }
61
62          // Wait for the block to be fully consumed before loading the
        next
63          // block.
64          barrier(CLK_LOCAL_MEM_FENCE);
65      }
66
67      // Store result in matrix C
68      C[get_global_id(1) * get_global_size(0) + get_global_id(0)] =
        running_sum;
69  }
```

Método 1: Kernel original para algoritmo de producto matricial con BS

Del código anterior cabe remarcar:

1. Los tamaños de las matrices de entrada siempre deberán de ser múltiplos del BS.

Si esta condición no se cumple, el algoritmo no se ejecutará.

2. a_start y b_start son los índices correspondientes al valor de fila y columna iniciales al que apunta el algoritmo en cada ejecución que realiza.

3. a_end y b_end son los índices correspondientes al valor de fila y columna finales al que apunta el algoritmo en cada ejecución que realiza.

4. A_local y B_local corresponden a los bancos de memoria locales donde se almacenarán los datos a recorrer a través del bucle que realiza el sumatorio de cada BS.

5. Se adopta una estrategia de "desenrollar" el bucle para una ejecución mas rápida del mismo. Se ha comprobado que esta estrategia es la que hace que se consuman mas recursos de la FGPA.

6. la señal $barrier(CLK\_LOCAL\_MEM\_FENCE)$ asegura que se haya completado el rellenado de los bancos de memoria o la finalización de bucle, respetando así el sincronismo dentro de la paralelización del algoritmo.

Capítulo 3. Funciones de activación y Convolución 2D

3.1 Funciones de activación

Son conocidas diversas funciones de activación en el ámbito de la inteligencia artificial, las mas comunes y que van a ser implementadas en la FPGA son: Relu, Tangente hiperbólica, Sigmoide y Softmax[6]. Dada la naturaleza de las funciones a implementar, como se verá, es posible compartir y optimizar recursos hardware para una optimización del área lógica utilizada en la FPGA; en concreto, la función sigmoide no es mas que una adaptación de la tangente hiperbólica o viceversa, conclusión fácil de extrapolar si analizamos matemáticamente los límites y convergencia de ambas funciones. En el caso de la función Relu, los recursos a utilizar son escasos puesto que se trata de aplicar un *threshold* al valor correspondiente a la matriz de salida, siendo habitual que este límite sea en cero. Se definen matemáticamente a continuación las citadas funciones:

$$Relu(x) = \begin{cases} 0 \; si \; x \leq 0 \\ x \; si \; x > 0 \end{cases} \tag{4}$$

En el caso de la tangente hiperbólica, para optimizar recursos se consigue modelar esta a través de la función exponencial y sus relaciones trigonométricas hiperbólicas, puesto que se que conoce que

$$cosh(x) = \frac{e^x + e^{-x}}{2}$$

$$sinh(x) = \frac{e^x - e^{-x}}{2}$$

por lo que se puede definir la tangente hiperbólica como:

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Operando la expresión anterior es posible reducir todavía mas la carga computacional haciendo que solo sea necesario el cálculo de un término exponenciado, dando como resultado final la siguiente expresión:

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{5}$$

Puesto que la función sigmoide es consecuencia de la tangente hiperbólica:

$$sigmoid(x) = \frac{e^x}{1 + e^x}$$

$$tanh(x) = 2 \cdot sigmoid(2x) - 1$$

esta puede definirse como:

$$sigmoid(x) \cong (\frac{1 - e^{-x}}{1 + e^{-x}} + 1) \cdot 0.5 \tag{6}$$

Finalmente, la función softmax requiere de iterar la matriz resultado del producto dos veces, una para acumular el resultado del sumatorio y otra para realizar la división de la matriz resultante entre el sumatorio:

$$Softmax(x) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{7}$$

En una primera aproximación se decidió hacer uso de la memoria local para almacenar el resultado del producto matricial previo a su iteración para realizar la división entre el sumatorio. La problemática hallada en este punto se localiza en la falta de recursos de memoria local, por lo que no es posible esta implementación para matrices de un tamaño que comience a ser mediano: 128×256 en adelante. A la vista de esto se decide realizar el proceso haciendo uso del buffer de memoria global utilizado para almacenar el resultado.

De esta forma quedarían definidas todas las funciones de activación de la forma mas óptima posible en relación a la implementación hardware en la FPGA. El siguiente flujograma muestra el kernel *MATRIX_MULT*, el cual contiene toda la lógica de funciones de activación, adición del bias y realización de convoluciones:

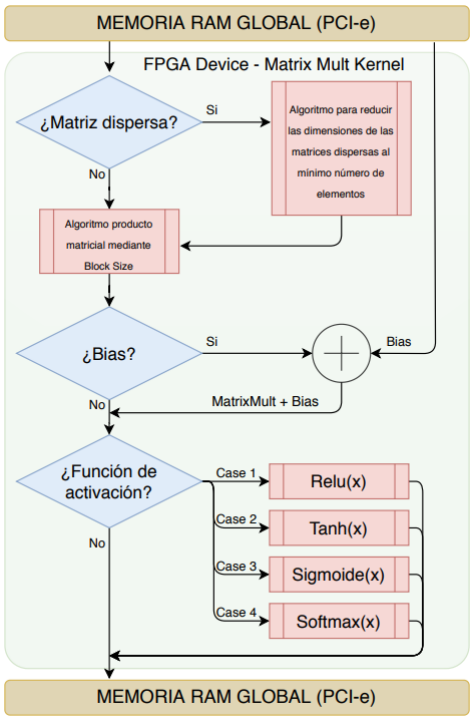


Figura 10: Flujograma del kernel FPGA propuesto

```
1
2  /*
3  Función de activación relu.
4  Si x > 0  -> y = x
5  Si x <= 0 -> y = 0
6  Entradas:
7      - x: Valor de entrada
8  Salidas:
9      - y: Valor de salida
10 */
11 float relu(float x) {
12     if (x > 0) {
13         return x;
14     }
15     else {
16         return 0.0;
17     }
18 }
19
20 /*
21 Función de activación tangente hiperbólica en función
22 del número e.
23 tanh(x)    = (1-e^(-2x)) / (1 + e^(-2x))
24 tanh(x/2)  = (1-e^(-x)) / (1 + e^(-x)
25 Entradas:
26     - x: Valor de entrada
27 Salidas:
28     - y: Valor de salida
29 */
30 float tan_h(float x, int control) {
31     x = -x;
32     if (control == 1) {
33         x = 2 * x;
34     }
35     x = exp(x);
36     return ((1 - x) / (1 + x));
37 }
38
39 /*
40 Función de activación sigmoide.
41 sigmoid = (tan_h(x/2) + 1) / 2
42 Entradas:
43     - x: Valor de entrada
44 Salidas:
45     - y: Valor de salida
46 */
47 float sigmoid(float x) {
48     float arg = tan_h(x, 0);
49     return (arg + 1) * 0.5;
50 }
51
52     // Del argumento de función de activación, <act> se decide a que
    función se envía
```

```
53      // mediante estructura switch/case
54      // Si act = 0 -> No se aplica función de activación
55      // Si act = 1 -> Relu
56      // Si act = 2 -> Sigmoide = (tanh(x) / 2 ) + 1 = exp(tanh) >> 2 + 1
57      // Si act = 3 -> Tangente hiperbólica: se toma de OpenCL directamente
58      // Si act = 4 -> Softmax
59      switch (act)
60      {
61          case 0: value = running_sum;            break;
62          case 1: value = relu(running_sum);      break;
63          case 2: value = sigmoid(running_sum);   break;
64          case 3: value = tan_h(running_sum, 1); break;
65          case 4: value = softmax(running_sum);   break;
66      }
```

Método 2: Código para las funciones de activación en el kernel de OpenCL

Se presentan a continuación los resultados obtenidos de recursos hardware utilizados en función del uso de la función de tangente hiperbólica o exponencial ya implementada en OpenCL. En los resultados se aprecia como el uso de la tangente hiperbólica penaliza enormemente el uso de recursos, siendo imposible la implementación de las funciones de activación puesto que los recursos usados superan el 100 %

Kernel Name	ALUTs	FFs	RAMs	DSPs
matrixMult_16_2_act	119608	102096	399	91.5
Global Interconnect	782	2227	10	0
Board Interface	24256	20822	143	0
Total	144646 (64 %)	125145 (28 %)	552 (45 %)	91 (27 %)
Available	227120	454240	1220	342

Tabla 1: Recursos utilizados por la FPGA con implementación de tanh mediante función exp de OpenCL

Resource	Usage
Logic Utilization	144 %
ALUTs	106 %
Dedicated logic registers	50 %
Memory blocks	74 %
DSP blocks	40 %

Tabla 2: Recursos utilizados por la FPGA con implementación de tanh mediante función tanh de OpenCL

En las dos tablas anteriores se exponen los recursos utilizados en la FPGA para dos kernels idénticos, la única diferencia es que en la primera tabla la función tangente hiperbólica ha sido implementada a través de la función exponencial y en la segunda figura se utiliza la función de tangente hiperbólica implementada en la librería de Intel OpenCL. Se demuestra por tanto la gran penalización que implica hacer uso de una

función u otra en cuanto a recursos hardware utilizados, pasando de un 106 % de ALUTS utilizadas y una utilización lógica de 144 % a un 64 % de ALUTS y un 75 % de lógica utilizada. A continuación se exponen las gráficas obtenidas para las distintas funciones de activación.

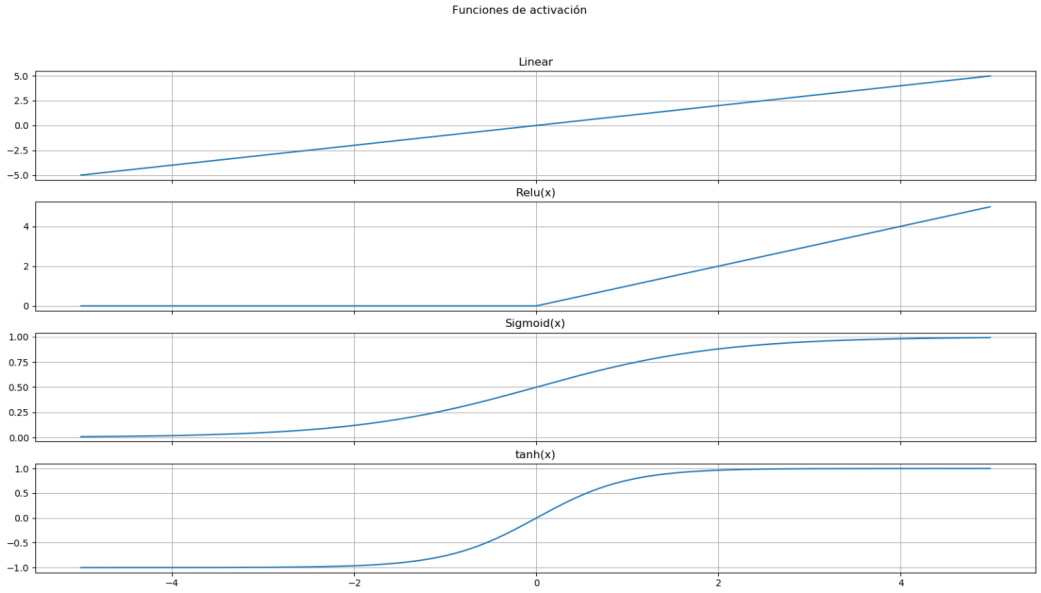


Figura 11: Funciones de activación en el intervalo -5,5 calculadas mediante la FPGA

3.2 Uso de *Bias*

El papel del bias en una red neuronal es el del desplazamiento del resultado de salida de la función de activación, tanto a la derecha como a la izquierda, de esta forma, es posible acotar mejor los valores de salida de la red neuronal en cada una de sus capas. Un ejemplo ilustrativo de la importancia del bias sería en el caso de que los valores de salida de una capa fueran mayores que 4 y la función de activación de la siguiente capa fuera una sigmoide o tangente hiperbólica, esto haría que los valores de salida computados por las mencionadas funciones estuvieran muy próximos a 1, perdiendo así información. Sin embargo, al aplicar un bias de -3, la salida de la sigmoide o tangente hiperbólica tendría un mayor rango puesto que los valores de entrada ya no serían próximos a 4.

A efectos prácticos en la FPGA, la adición del bias es una operación simple donde solamente se decide si aplicar o no el bias en función de la señal *ctrl_bias*. Como argumento de entrada, el kernel de la FPGA dispone de un buffer de memoria global que contiene el vector bias. En el caso de que se trate de un valor constante de bias, el buffer de memoria global valdrá 0 y no será leído en cada iteración, si no que la señal *const_bias* será distinta de 0, indicando al kernel que habrá de sumar a la salida el valor de *const_bias*. Este proceso no merece mención en cuanto a uso de recursos hardware.

```
1
2  // Se controla si se activa el bías o no. Si hay bías:
3  // Puede ser un valor constante o un vector pasado.
4  // En el caso de que haya un valor constante,
5  // const_bias ha de ser distinto de 0
6  if (ctrl_bias == 1)
7  {
8    if (const_bias != 0) {
9        running_sum = running_sum + const_bias;
10     }
11     else {
12       running_sum = running_sum + Bias[get_global_id(0)];
13     }
14 }
```

Método 3: Código para la adición del bias

3.3 Convolución 2D, Up-Sampling 2D, Average Pooling y Max Pooling

En las redes neuronales convolucionales, *CNN*, se utiliza el proceso de convolución, que no deja de ser mas que un filtrado de la señal de entrada mediante unos coeficientes definidos. Cuando se trata de NN's, los coeficientes del kernel son entrenados en cada iteración, por lo que se podría definir el kernel como la matriz de pesos asociada a esa capa. La relación matemática que define la convolución en 2 dimensiones para un sistema discreto es:

$$y[i,j] = x \circledast h = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} h[m,n] \cdot x[i-m, j-n] \tag{8}$$

Esta relación matemática puede entenderse mejor en el siguiente dibujo donde se representa como se desplazaría el filtro a través de la imagen de entrada:

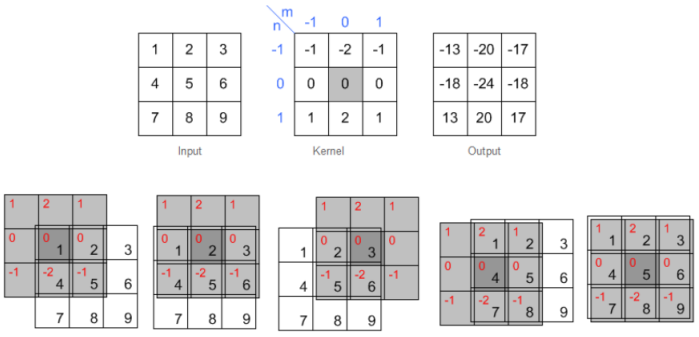


Figura 12: Ejemplo gráfico de convolución 2D

Para la implementación de la Convolución 2D en la FPGA se ha utilizado el método de producto matricial mediante la matriz de convolución, así pues se aprovecha la estructura de BS propuesta por el algoritmo de Intel FPGA. La matriz de convolución se obtiene a través de sendos bloques de matrices de *Toeplitz* [8], donde un ejemplo de la representación de la matriz de *Toeplitz* es:

$$T = \begin{pmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ j & h & g & f & a \end{pmatrix}$$

Matemáticamente esta se define como:

$$\forall a_{i,j} \in T \rightarrow a_{i,j} = a_{i+1,j+1} \tag{9}$$

La matriz de convolución resultante a partir de matrices de *Toeplitz*, llamada comúnmente como *Double blocky matrix* tendría la siguiente estructura:

$$H(T) = \begin{pmatrix} T_1 & T_2 & T_3 & \cdots & T_N & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & T_1 & T_2 & T_3 & \cdots & T_N & 0 & \cdots & \cdots & 0 \\ 0 & 0 & T_1 & T_2 & T_3 & \cdots & T_N & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & 0 & T_1 & T_2 & T_3 & \cdots & T_N \end{pmatrix}$$

Mientras que matemáticamente no dejaría de ser una re-interpretación de la matriz de *Toeplitz* cambiando el valor ai, j por $T_{i,j}$, quedando la siguiente expresión:

$$\forall T_{i,j} \in H(T) \rightarrow T_{i,j} = T_{i+1,j+1} \tag{10}$$

Para conseguir la convolución a través del producto matricial es necesario convertir la imagen de entrada en un vector columna, quedando finalmente el producto matricial:

$$y[i,j] = x \circledast h = H(T) \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} \tag{11}$$

Las dimensiones de la matriz de convolución se pueden definir a través de las siguientes ecuaciones, donde f y c hacen referencia a las filas y columnas de la matriz de convolu-

ción, K_f y K_c corresponden a las filas y columnas del kernel de convolución. Por último, I_c junto con I_f indican las filas y columnas de la señal de entrada.

$$f = (I_f - K_f) \cdot (I_c - K_c)$$
$$c = I_f \cdot I_c$$

Mientras que el número de elementos que no serán nulos, N_z, en la matriz de convolución se corresponderá con:

$$N_z = \frac{K_f \cdot K_c}{(I_f - K_f) \cdot (I_c - K_c)} = \frac{K_f \cdot K_c}{f}$$

Es fácil percibir que se trata de una estructura repetitiva y simétrica, además de contener muchos elementos nulos, es decir, ceros. Estas condiciones llevan a deducir que lo mas efectivo para conseguir la convolución a través del producto matricial sea adoptando una estrategia de producto matricial mediante matrices dispersas[2].

En efecto, cuando los tamaños de imagen de entrada superan los 90×90 píxeles, python es incapaz de montar la matriz de convolución debido al gran número de elementos en la matriz, casi todos ellos con valor nulo. De igual forma es imposible generar los buffers de la FPGA con las matrices ya que "no existe memoria física" para su almacenado.

3.3.1 Métodos de Pooling y Up Sampling en 2 dimensiones

Definida la matriz de convolución la cual contiene los coeficientes del kernel, se pueden expresar de forma inmediata los métodos de *Up Sampling 2D*, *Average Pooling* y *Max Pooling* a través de su matriz de coeficientes. En el caso de *Average Pooling*, si se tiene una matriz de $H_{M \cdot N}$ con M y N como el número de filas y columnas del kernel, este quedaría definido como:

$$H = \frac{1}{M \cdot N} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \cdots & 1 \end{pmatrix}$$

[2]En álgebra lineal numérica una matriz dispersa o matriz rala o matriz hueca es una matriz de gran tamaño en la que la mayor parte de sus elementos son cero

Si se trata del método *Max Pooling* no es necesario realizar el producto matricial, simplemente coger el mayor valor en cada iteración del bucle, por lo que no es necesario definir una matriz concreta; solo indicar a la FPGA el tamaño de entrada del kernel.

Para el método de *Up Sampling* con interpolación bilineal, utilizado para el re-escalado de imágenes es necesario hallar los coeficientes del kernel. Estos coeficientes se resuelven en función del parámetro *factor de escala*: F_{DE} que indica cuanto se va a incrementar el ancho y largo de la imagen de entrada. Se ha decidido tener en cuenta el mismo F_{DE} tanto para las filas y columnas con el fin de evitar que se pierda la proporcionalidad de la imagen. Las dimensiones del kernel $H_{M\times N}$ en función del F_{DE} serán de:

$$M, N = 2 \cdot F_{DE} - F_{DE}\,\%2$$

se define una variable denominada factor como $f = \lfloor \frac{M+1}{2} \rfloor$ y otra variable centro como

$$c = \begin{cases} f - 1 & si \quad M\,\%2 == 1 \\ f - 0.5 & si \quad M\,\%2 == 0 \end{cases}$$

El cálculo de los coeficientes se realiza como:

$$H_{(m,n)} = \sum_{n=1}^{N} \sum_{m=1}^{M} \left(1 - \frac{|n - c - 1|}{f}\right) \cdot \left(1 - \frac{|m - c - 1|}{f}\right) \tag{12}$$

En el caso de que se trate de interpolación tipo *nearest* no será necesario generar los coeficientes del kernel puesto que simplemente se replicarán los píxeles necesarios en función del parámetro F_{DE}. Para lograr tal fin se hará uso de una máscara de indexamiento que, en función de la matriz de entrada, la salida que generará será una matriz con los píxeles replicados en su correcta posición.

3.3.2 Máscara para productos matriciales de tipo disperso en convoluciones 2D

Se deduce de lo anterior la importancia de utilizar un algoritmo para reducir el número de operaciones realizadas por la FPGA para productos matriciales con matrices dispersas. El algoritmo propuesto se basa en la utilización de una máscara que indica al kernel si debe entrar en el bucle que realiza el producto de cada elemento de la matriz. Así pues, si todos los elementos de un BS son nulos, la mascara contendrá en esa posición un cero, en caso contrario tendrá un uno. El siguiente ejemplo ilustra el funcionamiento del algoritmo tomando como ejemplo una matriz identidad de 8x8. Este algoritmo es extensible a cualquier dimensión de matriz, no teniendo que ser necesariamente una matriz cuadrada.

$$I = \begin{bmatrix} \boxed{\begin{matrix}1 & 0\\ 0 & 1\end{matrix}} & \boxed{\begin{matrix}0 & 0\\ 0 & 0\end{matrix}} & \boxed{\begin{matrix}0 & 0\\ 0 & 0\end{matrix}} & \boxed{\begin{matrix}0 & 0\\ 0 & 0\end{matrix}} \\ \boxed{\begin{matrix}0 & 0\\ 0 & 0\end{matrix}} & \boxed{\begin{matrix}1 & 0\\ 0 & 1\end{matrix}} & \boxed{\begin{matrix}0 & 0\\ 0 & 0\end{matrix}} & \boxed{\begin{matrix}0 & 0\\ 0 & 0\end{matrix}} \\ \boxed{\begin{matrix}0 & 0\\ 0 & 0\end{matrix}} & \boxed{\begin{matrix}0 & 0\\ 0 & 0\end{matrix}} & \boxed{\begin{matrix}1 & 0\\ 0 & 1\end{matrix}} & \boxed{\begin{matrix}0 & 0\\ 0 & 0\end{matrix}} \\ \boxed{\begin{matrix}0 & 0\\ 0 & 0\end{matrix}} & \boxed{\begin{matrix}0 & 0\\ 0 & 0\end{matrix}} & \boxed{\begin{matrix}0 & 0\\ 0 & 0\end{matrix}} & \boxed{\begin{matrix}1 & 0\\ 0 & 1\end{matrix}} \end{bmatrix}$$

De la matriz identidad se construye su máscara, la cual se corresponde con:

$$Mask(I) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

En este ejemplo se cumple que la máscara de una matriz identidad es la matriz identidad con las dimensiones reducidas, pero no siempre es así; por lo tanto hay que generar la máscara siempre para todas las matrices de tipo disperso.

Matemáticamente se puede demostrar el número de operaciones que realiza la FPGA en el producto matricial. Si se tienen dos matrices $A \in M_{n\times m}$ y $B \in M_{n\times k}$ es conocido que el número total de operaciones a realizar será de $m \times n \times k$. Por otro lado, para las matrices de convolución en 2 dimensiones, $H(T) \in M_{f\times c}$, se ha definido anteriormente el cálculo del número de filas y columnas: f y c. Conocidas estas relaciones es fácil establecer el número de operaciones necesarias para realizar una convolución mediante producto matricial, que sería del orden de $f \times c \times batch$, donde el batch se corresponde con el número de imágenes a procesar. De las relaciones anteriores se demuestra como la matriz de convolución solo es útil para dimensiones pequeñas. El parámetro que permite determinar si una matriz es dispersa o no es el de la densidad. Este se define como el cociente entre el número de elementos que no son cero N_z y el número total de elementos en la matriz, por lo que se puede formular como:

$$d(H(T)) = \frac{K_f \cdot K_c}{I_f \cdot I_c} = \frac{N_z}{c} \tag{13}$$

El complementario de la densidad sería el de la dispersión, definido como

$$s(H(T)) = 1 - d(H(T)) \tag{14}$$

Con todas estas relaciones se conoce el número de operaciones que realizará el kernel en función de la máscara y el BS, justificando así el porque crece el tiempo de cómputo.

Como es obvio, por deducción se puede inferir que a un mayor número de ejecuciones del kernel mayor tiempo de computo. Teniendo en cuenta el algoritmo de BS y la máscara para matrices dispersas, el número de ejecuciones del kernel, E_{xk} de la FPGA sería (donde N_{zs} hace referencia al número de elementos no nulos en la máscara):

$$E_{xk} = N_{zs} \cdot BS$$

Mientras que el total de operaciones realizadas por la FPGA se correspondería con:

$$E_{xk} \cdot batch$$

De las anteriores relaciones se deduce que el número de ejecuciones del kernel sería mucho mayor sin la máscara y por ende el número de operaciones realizadas. En estas tablas se aprecian los tiempos de computo y número total de operaciones y ejecuciones de kernel realizadas para convoluciones con tamaños de entrada distintos y un kernel con 3 filas y columnas para productos de tipo disperso y no disperso. Otro factor de suma importancia en los tiempos de ejecución es la colocación de los datos en los buffers. Esta acción es la que mas penaliza en tiempo de ejecución al kernel, puesto que como se verá en el capítulo 5, al analizar los tiempos de convolución se podrá apreciar como todavía bajan mucho mas los tiempos de ejecución al hacer uso de un solo bloque de la matriz de convolución, evitando así cargar toda la matriz en la FPGA.

Dimensiones I	Dimensiones H	Dimensiones K	No sparse (s)	Sparse (s)
30x30	912x1024	3x3	0.0040	0.0040
62x62	3856x4096	3x3	0.1278	0.1656
94x94	8848x9216	3x3	0.9887	1.3476
126x126	1588x16384	3x3	4.2077	9.1901

Tabla 3: Tiempos de ejecución para producto matricial con matrices no dispersas y dispersas

Dimensiones I	N_z	$Z = f \cdot c - N_z$	Densidad (%)
30x30	21840	547248	3.84
62x62	113680	12410608	0.91
94x94	277200	69798960	0.40
126x126	512400	232087920	0.22

Tabla 4: Densidad de matrices dispersas en función de sus elementos nulos

Dimensiones I	Ejecuciones kernel	Ops matriz no densa	Ops matriz densa
30x30	35568	9105408	98048
62x62	782768	200388608	598272
94x94	4379760	1121218560	1446656
126x126	14537520	3721605120	2663680

Tabla 5: Total de ejecuciones del kernel y operaciones para matrices no dispersas y dispersas

Esta máscara es obtenida en python mediante la siguiente función:

```
'''
Función que crea una máscara para el producto matricial de tipo disperso
    en
función del Block Size de la FPGA
'''
def create_s_mask(a, nrows, ncols, pad_with_zeros=False):
    r, c = a.shape
    if pad_with_zeros:
        r_, c_ = int(np.ceil(r / nrows)), int(np.ceil(c / ncols))
        # añadir ceros para que cada submatriz sea igual
        a = np.pad(a, ((0, r - r_ * nrows), (0, c - c_ * ncols)))
    else:
        r_, c_ = r // nrows, c // ncols
        # quitar los últimos elementos para que cada submatriz sea igual
        a = a[:r_ * nrows, :c_ * ncols]
    a = a.reshape(r_, nrows, c_, ncols)
    return np.count_nonzero(a, axis=(1,3)).astype(np.bool).astype(np.
    int32)
```

Método 4: Generación de la máscara para productos matriciales de tipo disperso

En la implementación hardware sobre la FPGA se tiene la señal *dense* que indica que se trata de un producto disperso cuando esta vale 1. Al activar esta señal, el kernel lee el valor de la posición de la mascara correspondiente al block size y decide si se debe realizar el producto o no. Se ha añadido otra señal, denominada *compressed* que indica si se trata de la matriz completa o solo un bloque de la matriz, puesto que en las matrices de convolución se tiene una estructura simetrica y repetitiva, donde solo es necesario conocer los valores del "kernel".

```

for (int a = a_start, b = b_start; a <= a_end; a += BLOCK_SIZE, b += (
    BLOCK_SIZE * B_width)) {
  // Aplicación de la máscara para productos de tipo disperso
  // la señal msk indica si se ha de entrar a realizar el
  // producto matricial o no.
  if (dense == 1) {
    // Si se trata de un bloque de la matriz de convolución
    // la máscara también estará comprimida
    if (compressed == 1) {
      if (counter < M_width) {
```

```
11          msk = Mask[counter];
12        }
13        else {
14          msk = 0;
15        }
16      }
17      // Si no se tiene comprimida la matriz de convolución
18      else {
19        msk = Mask[counter + (block_y * M_width)];
20      }
21    }
22    // Se realiza el producto matricial
23    if (msk == 1) {
24      // ... Lógica para el producto matricial
25    }
26  }
```

Método 5: Código para el control del producto matricial disperso en función de la máscara

3.3.3 Algoritmo de convolución por bloques y desplazamiento de índice

Cuando se tienen señales de entrada a partir de los 90x90 píxeles en adelante, el número de valores a representar en la matriz de convolución es muy grande, por lo que se ha hecho necesario adaptar el algoritmo para poder trabajar con el a través de la FPGA. La estrategia que se ha seguido se basa en tomar un único bloque de la matriz de convolución, con dimensiones $BSxT_{cols}$, donde BS representa el tamaño de BS y T_{cols} es el número de columnas de uno de los bloques de la Matriz de *Toeplitz* para guardarlo en memoria global y hacer uso de el en cada iteración necesaria del producto matricial. Se hace imprescindible conocer los índices donde se comenzará a recorrer la matriz B para que estos se correspondan de manera idéntica a si se realizará el producto matricial con la matriz de convolución completa. Por lo tanto, lo que se calcula previamente es el índice donde comenzar a coger los datos de la matriz B. Se pueden dar dos casos:

- 1 - Nº de columnas de la imagen de entrada es múltiplo de BS
- 2 - Nº de columnas de la imagen de entrada NO es múltiplo de BS

Cuando se cumple la condición 1 basta con desplazar el índice b_{start} como $b'_{start} = b_{start}+BS \cdot block_y$. En caso de que se de la condición 2, el ajuste del índice es mas complejo, pues se hace necesario tanto desplazar el índice de b_{start} como el de a_{start}. La FPGA tiene el buffer de memoria global *Mask*. *Mask* engloba tanto la mascara para el producto matricial disperso como los desplazamientos a aplicar a a_{start} y b_{start}. Dependiendo de si se cumplen las condiciones 1 y 2 citadas anteriormente, la estructura de *Mask* será:

- 1 - Nº de columnas de la imagen de entrada es múltiplo de BS

Máscara
Desplazamientos para b_{start}

Figura 13: Máscara para convolución con matriz dispersa e imagen de entrada con Nº de columnas múltiplo del BS

- 2 - Nº de columnas de la imagen de entrada NO es múltiplo de BS

Máscara bloque 1
Máscara bloque 2
Desplazamientos para b_{start}
Enable shift a
$block_y$ *shift*

Figura 14: Máscara para convolución con matriz dispersa e imagen de entrada con Nº de columnas no múltiplo del BS

La siguiente imagen muestra como actúa el algoritmo de convolución.

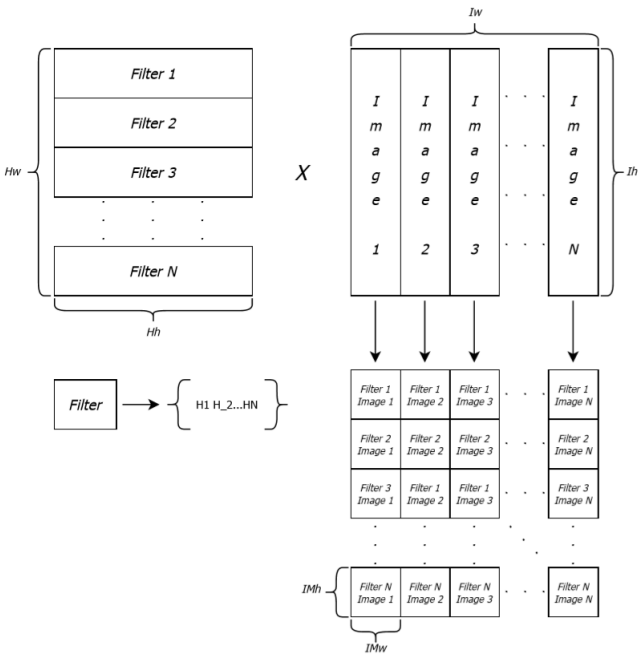


Figura 15: Representación de las matrices de entrada y salida para convolución 2D con mas de un filtro

Las relaciones necesarias son:

$$f_h, f_w : Filas, Columnas\ Filtro$$

$$H_h, H_w : Filas, Columnas\ Matriz\ convolución$$

$$I_h, I_w : Filas, Columnas\ Bloque\ imagen\ padeada$$

$$I_{Mh}, I_{Mw} : Filas, Columnas\ imagen\ original$$

$$H_h = (I_w + f_w - 2) \cdot f_h$$

$$H_w = BS \cdot N$$

$$I_w = N$$

$$S_{hM} = (I_{Mh} \cdot I_{Mw}) \cdot (BS + (f_w - 1))$$

$$S_r = (I_{Mw}\ \%BS) + (r - 1) \cdot BS$$

$$H_h = (I_w + f_w - 2) \cdot f_h$$

El código python que obtiene los desplazamientos necesarios en función del kernel de la red convolucional es el siguiente:

```python
def shift_indexes(filters, input_size, BS, padding="same"):
    # Valores de retorno
    shift_en, shift_by = None, None
    # shapes
    k_h, k_w, n_f = filters.shape
    i_h, i_w = input_size
    if padding == "same":
        i_h, i_w = i_h+k_h-1, i_w+k_w-1
    o_h, o_w = i_h-k_h+1, i_w-k_w+1

    # Valores iniciales
    vectors = []
    resto   = (i_w%BS)-(k_w-1)
    rows_1  = o_h * o_w
    rows_2  = ceil(rows_1/BS)
    # Creo vector de índices mediante np.arange
    cont     = np.arange(rows_1, dtype=np.int32)

    # Array para las posiciones a extraer de shift_rw
    index_i = np.arange(0, rows_1, BS)
    index_o = index_i + BS - 1
    index_o[-1] = index_i[-1]

    # Se realiza el desplazamiento en la posición
    # correspondiente: Cada "o_w" posiciones
    for i in range(1, rows_2):
        cont[i*o_w:(i+1)*o_w] += i*(k_w-1)
```

```
28
29      # Vector con posiciones de desplazamiento en B
30      shift_rw = cont[index_i]
31      shift_a, shift_b, shift_c, shift_d = rows_2, 0, 0, 0
32      if o_w % BS != 0:
33          shift_by = np.zeros(rows_2,  dtype=np.int32) # Vector de
    desplazamientos
34          # Vector de enable para desplazamiento: Solo si o_w no es
35          # múltiplo del BLOCK_SIZE
36          shift_en = ((cont[index_o] - shift_rw - BS) > 0).astype(int)
37          indexs   = np.nonzero(shift_en)[0]
38          # Vector de desplazamientos para el block_y
39          init  = np.arange(resto, (len(indexs)+1)*resto, resto)
40          zeros = np.where(init % BS == 0)[0]
41          init  = np.delete(init, zeros)
42          vectors.append(init)
43          while len(zeros) != 0:
44              init  = np.arange(init[-1]+resto, init[-1]+resto*(len(zeros)
    +1), resto)
45              zeros = np.where(init % BS == 0)[0]
46              vectors.append(np.delete(init, zeros))
47          shift_by[indexs] = np.concatenate(vectors) % BS
48          shift_a, shift_b, shift_c, shift_d = 2*rows_2, 3*rows_2, 4*rows_2
    , rows_2
49      return {"shift_rw": shift_rw, "shift_en": shift_en, "shift_by":
    shift_by,
50              "shift_a" : shift_a,  "shift_b" : shift_b,
51              "shift_c" : shift_c,  "shift_d" : shift_d}
```

Método 6: Cálculo de desplazamiento de índices para algoritmo de Convolución 2D en la FPGA

El código OpenCL que permite tener en cuenta estos desplazamientos es:

```
1
2  // Se comienza con los índices b_start y a_start habituales para
3  //un producto matricial común
4  int a_start = 0; int b_start = BLOCK_SIZE * block_x;
5  int shift = 0;
6  // Si se trata de un bloque de matriz de convolución (por esta
7  // razón se denomina "compressed", puesto que solo se tiene
8  // de la información necesaria)
9  if (compressed == 1) {
10   // Si no hay repeticiones del bloque, es decir: Iw = BS
11   // Esta condición no suele cumplirse mas que para imágenes
12   // con largo = 16
13   if (reps == 0) {
14     shift = block_y * Shift_A;
15   }
16   // Si existen repeticiones: Hay que desplazar el índice BS posiciones
17   // o BS posiciones + columnas del kernel si block_y es múltiplo de reps
18   else {
19     shift = (block_y * Shift_A) + (Shift_B * (block_y / reps));
20   }
21   // Finalmente se suma el desplazamiento al índice b_start
```

```
  b_start += shift;
}
// Si se trata de un producto con toda la información de la matriz A
// disponible, entonces el índice a_start recorre todas las posiciones
else {
  a_start = A_width * BLOCK_SIZE * block_y;
}
int a_end = a_start + A_width - 1;
```

Método 7: Código para la implementación del algoritmo de convolución 2D mediante desplazamiento de índice

3.3.4 Algoritmo de convolución mediante técnica *Im2Row*

El método descrito anteriormente es ineficiente para convoluciones con mas de un canal o filtro ya que se utilizan matrices dispersas. Para solventar esto existen otros métodos como la transformada de *Winograd*[9] o a través de la transformada de Fourier, convirtiendo previamente la señal y el kernel al dominio de la frecuencia y después realizar el producto matricial de los mismos. La transformada de *Winograd* se ha descartado puesto que el algoritmo es difícil de adaptar al BS, mientras que a través de la transformada de Fourier se descarta ya que solo es práctica cuando el tamaño de los kernels es grande, y en las CNN estos raramente suelen superar dimensiones de 9×9.

El método de *Im2Row*[7] consiste en reordenar la matriz de entrada, es decir, la señal, de forma que los bloques de cada ventana del filtro correspondan a una fila. De forma análoga, la matriz de filtros es reordenada para que cada filtro se corresponda con una columna. Así se consigue a través del producto matricial evitar una matriz dispersa y toda la lógica necesaria para convertir el producto de matrices dispersas en producto de matrices densas. La siguiente imagen ilustra como se reordenan las matrices:

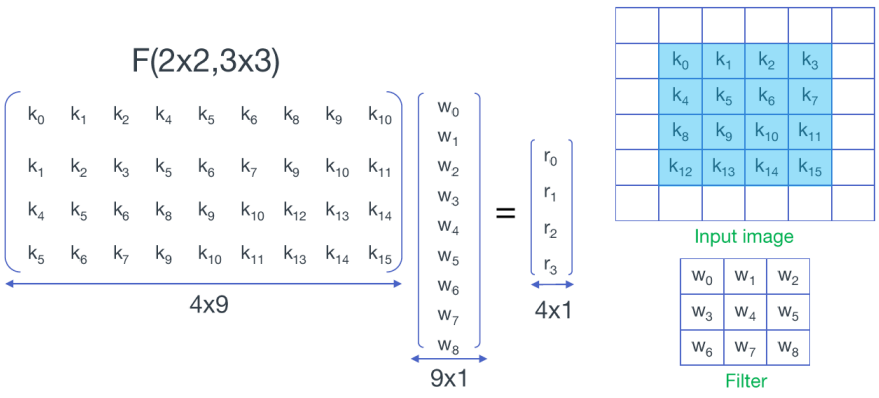


Figura 16: Esquema para técnica *Im2Row*

Esta técnica hace necesario reestructurar la matriz de entrada para que el dispositivo FPGA pueda realizar íntegramente la operación, para esto se ha utilizado de nuevo una máscara que contiene la relación de índices, indicando en cada iteración la posición que se ha de tomar de la matriz de entrada.

Cuando existen varios canales, la convolución implementada en las librerías de Tensorflow y Keras se corresponde con el diagrama de bloques representado en la siguiente página. Algunas de las propiedades de la convolución que pueden ser interesantes para los sistemas convolucionales en cascada son:

- Propiedad Conmutativa

$$y(t) = f(t) \circledast h(t) = h(t) \circledast f(t) \tag{15}$$

- Propiedad Asociativa

$$y(t) = f_1(t) \circledast (f_2(t) \circledast f_3(t)) = (f_1(t) \circledast f_2(t)) \circledast f_3(t) \tag{16}$$

El interés en estas propiedades radica en su utilidad para poder pre-calcular la convolución de los filtros pudiendo llegar a reducir el número de convoluciones a realizar en tiempo real en solo uno. Para que estas condiciones se cumplan el operador convolucional debe de mantener su linealidad, es decir, no debe de tener aplicada ninguna otra operación que no sea lineal. Lamentablemente, las funciones de activación aplicadas en las redes neuronales convolucionales hacen que el operador convolucional deje de ser lineal, por lo que solo se han podido demostrar las propiedades a nivel práctico pero no han sido aprovechadas en los sistemas de inteligencia artificial analizados en este TFG.

Para realizar las convoluciones en cascada mediante un producto matricial se ha utilizado también la técnica *Im2Row*:

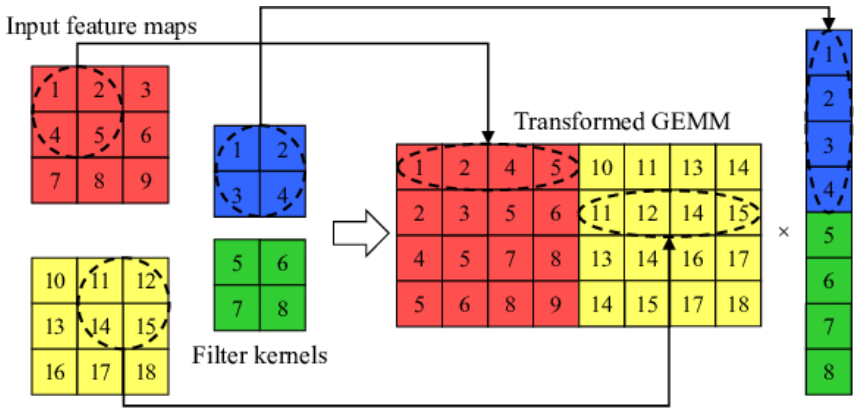


Figura 17: Ejemplo para convolución con mas de un canal mediante *Im2Row*

Si se define el número de filas y columnas de la seña de entrada como I_f, I_c, el número de filas y columnas del kernel como K_f, K_c, el número de filtros y canales como N_f, N_ch, coorrespondiente al número de filtros de la etapa de convolución anterior, se pueden hallar las columnas y filas de la matriz de entrada, C_{If}, C_{Ic} y el kernel, F_{Kf}, F_{Kc} para la técnica de convolución mediante Im2Row como:

$$C_{If},\ C_{Ic} = I_f \cdot I_c \cdot N_{ch},\ N_{ch} \cdot K_f \cdot K_c$$
$$F_{Kf},\ F_{Kc} = N_{ch} \cdot K_f \cdot K_c,\ N_f$$

A diferencia del método de convolución mediante la matriz de Toeplitz, en este caso la matriz de filtros no depende del tamaño del Input, solo de las dimensiones del kernel y del número de características.

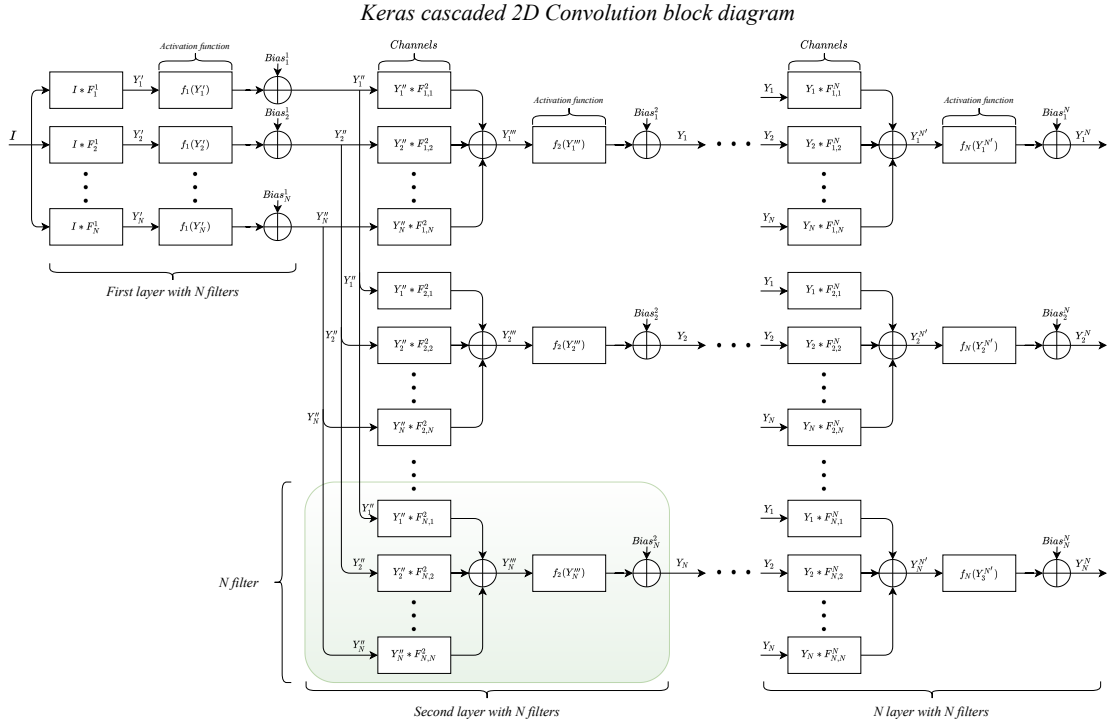


Figura 18: Diagrama de bloques de convolución con mas de un canal implementado

3.3.5 Algoritmo para padeo y representación de la señal de entrada

Los métodos que implementan Tensorflow/Keras para la convolución 2D en función del *padding* son:

- *Same*: Implica que las dimensiones de la señal de salida después de aplicar el filtrado serán similares a las de entrada, por lo que se hace necesario el *padeo*
- *Valid*: Las dimensiones de la señal de salida serán menores a las de entrada, por lo que se pierde información y no es necesario padear la señal de entrada.

Cuando se trata de una convolución de tipo *Same* es necesario aplicar ceros en los extremos de la imagen para que el filtro pueda recorrer toda la señal sin perder información. Teniendo en cuenta que se trabaja con estructuras de tipo *batch* donde se analiza simultáneamente mas de una muestra, cada señal de entrada se corresponderá con una columna de la matriz B. Los ceros para padear las señales de entrada estarán intercalados entre las columnas en vez de en los extremos de cada imagen. Para ello se ha propuesto un algoritmo que recibe la señal de entrada como input y devuelve la señal *padeada* con la estructura acondicionada para que la convolución mediante producto matricial sea posible. Para conseguir determinar donde se han de introducir los ceros, un tercer buffer de memoria global: *Mask*[3], indicará la posición donde se ha de insertar el valor tomado de la matriz de entrada en la matriz de salida, pues esta máscara contiene la relación de índices entre la matriz de entrada y la de salida. Cuando la matriz máscara tiene un -1 como valor en una posición específica, implica que se ha de colocar un 0 en esa posición de la matriz de salida.

```
1
2  // Pad the P * Q matrix with zeroes to form a P_XL * Q_XL matrix
3  __kernel void paddingGemm(
4      __global float* restrict input,
5      __global float* restrict output,
6      __global int*   restrict mask,
7      int P, int Q, int P_XL, int Q_XL)
8  {
9      // Thread identifiers
10     const int tx = get_group_id(0)*BLOCK_SIZE + get_local_id(0); // 0..
       P_XL
11     const int ty = get_group_id(1)*BLOCK_SIZE + get_local_id(1); // 0..
       Q_XL
12     // Check whether we are within bounds of the XL matrix
13     if (tx < P_XL && ty < Q_XL) {
14         // Copy the input or pad a zero
15         int index_mask;
16         float value;
17     // Si el índice observado en la máscara vale -1 entonces en la matriz
        de
18     // salida se debe colocar un 0 en el índice actual
19         if (mask[ty*P_XL + tx] == -1) {
20             value = 0.0f;
21         }
22     // Si el índice observado en la máscara != -1 entonces en la matriz
       de
23     // salida se debe colocar el valor leido en la matriz de entrada
```

[3]No confundir con el buffer *Mask* utilizado en el algoritmo de convolución

```
24          else {
25              index_mask = mask[ty*P_XL + tx];
26              value = input[index_mask];
27          }
28          // Store the result
29          output[ty*P_XL + tx] = value;
30      }
31  }
```

Método 8: Kernel OpenCL para padear las imágenes de entrada

Capítulo 4. Análisis tiempos de ejecución y resultados de convolución 2D

4.1 Motivación

Dado que el kernel OpenCL creado para este TFG pretende ser un *SGEMM*[4], y puesto que el algoritmo principal depende del BS, es importante analizar el BS óptimo con el fin de optimizar al máximo el kernel principal, es decir, aquel que realiza el producto matricial. Como se ha explicado en el capítulo 2: *Algoritmo de BS de Intel FPGA*, el BS determina el número de elementos que se procesan simultáneamente, cabe por tanto intuir que a un mayor BS, mayores serán las prestaciones del algoritmo a la vez que mayor será el uso de los recursos de la FPGA.

Por tanto, se presentan y analizan en este capítulo las prestaciones y consumo de recursos de la FPGA en función del BS, realizando una comparativa de los tiempos de ejecución con los obtenidos de forma igual con Keras y Numpy.

Una vez realizados y analizados los resultados, se decidirá coger aquel Kernel más rápido en función del BS y, sobre ese kernel, se añadirán las funciones de activación y algoritmo de convolución mediante desplazamiento de índices.

4.2 Tiempos de ejecución para algoritmo de BS

Se presentan a continuación las gráficas y tablas correspondientes al algoritmo de producto matricial. Se han analizado valores de BS de 8, 16 y 32 para matrices cuadradas con dimensiones de 32, 64, 128, 256, 512, 1024, 2048 y 4096. La columna G_{FLOPS} indica el número de operaciones realizadas por segundo. Teniendo en cuenta que las matrices tienen dimensiones cuadradas, es decir: $M_{N\times N}$, el número de operaciones realizadas para el producto matricial será de N^3, por lo que se definirá el número de operaciones por segundo como:

$$G_{FLOPS} = \frac{N^3}{t}$$

donde t se corresponde al tiempo de ejecución al producto matricial obtenido por la FPGA.

[4]Single precision floating General Matrix Multiply: Unidad electrónica dedicada a realizar operaciones algebraicas tales como el producto matricial, producto punto, transformaciones lineales... etc. Muchas de estas operaciones pueden obtenerse directamente a través del producto matricial como se ha visto en el ejemplo de la convolución o la propagación de la señal a través de una red neuronal

4.2.1 Recursos FPGA utilizados, tiempos de ejecución y *SpeedUp* para BS=8 y SWI=4

Se exponen a continuación los recursos utilizados por la FPGA para una arquitectura con un BS=8 y un SWI=4.

Kernel Name	ALUTs	FFs	RAMs	DSPs
matrixMult	54585	47602	161	40
Global Interconnect	6577	8905	0	0
Board Interface	24256	20822	143	0
Total	84418 (38 %)	77339(17 %)	304 (25 %)	40 (12 %)
Available	227120	454240	1220	340

Tabla 6: Utilización de recursos FPGA para BS=8 y SWI=4

Los tiempos de computo y speed up comparativos entre Keras y Numpy para matrices cuadradas desde 32×32 hasta 4096×4096 se muestran en la siguientes tablas:

Dimensiones	Tiempo Keras (s)	Tiempo Numpy (s)	Tiempo FPGA (s)	GFLOPS
32x32	8.88e-03	1.87e-03	6.08e-05	0.54
64x64	7.99e-03	1.21e-04	1.48e-04	1.77
128x128	8.54e-03	2.49e-04	6.03e-04	3.48
256x256	1.10e-02	1.18e-03	4.08e-03	4.11
512x512	1.91e-02	7.02e-03	3.18e-02	42.20
1024x1024	8.52e-02	4.88e-02	3.19e-01	3.37
2048x2048	5.86e-01	3.74e-01	4.93e+00	1.74
4096x4096	4.27e+00	2.92e+00	5.87e+01	1.17

Tabla 7: Tiempos de ejecución para producto matricial sin función de activación con BS=8 y SWI=4

Dimensiones matriz	Numpy(s)/FPGA(s)	Keras(s)/FPGA(s)
32x32	30.76	146.5
64x64	0.82	53.99
128x128	0.41	14.16
256x256	0.29	2.70
512x512	2.20	6.00
1024x1024	0.15	0.27
2048x2048	0.07	0.12
4096x4096	0.05	0.073

Tabla 8: SpeedUp para producto matricial sin función de activación con BS=8 y SWI=4

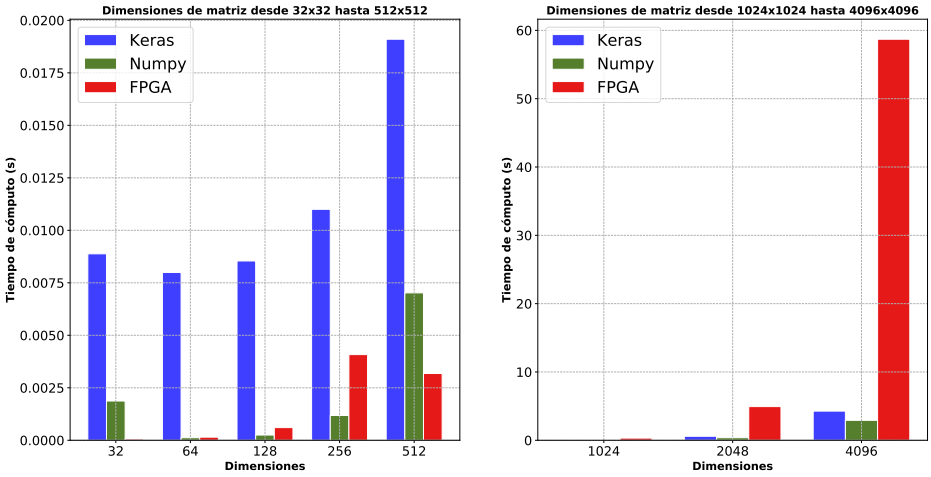


Figura 19: Gráfica comparativa de los tiempos de cómputo para producto matricial con BS=8 y SWI=4

Se cumple que Para matrices con tamaños inferiores a 512 los mejores tiempos de cómputo corresponden a Numpy y la FPGA. Numpy parece ser el mas óptimo de los tres para todos los tamaños. A medida que los tamaños son superiores a 512 la FPGA consigue la peor marca, seguido de Keras y finalmente Numpy continua consiguiendo los mejores tiempos.

4.2.2 Recursos FPGA utilizados, tiempos de ejecución y *SpeedUp* para BS=16 y SWI=4

Se exponen a continuación los recursos utilizados por la FPGA para una arquitectura con un BS=16 y un SWI=4.

Kernel Name	ALUTs	FFs	RAMs	DSPs
matrixMult	97914	82664	223	72
Global Interconnect	6577	8915	0	0
Board Interface	24256	20822	143	0
Total	128747 (57 %)	112401 (25 %)	366 (30 %)	72 (21 %)
Available	227120	454240	1220	340

Tabla 9: Utilización de recursos FPGA para BS=16 y SWI=4

Los tiempos de computo y speed up comparativos entre Keras y Numpy para matrices cuadradas desde 32×32 hasta 4096×4096 se muestran en la siguientes tablas:

Dimensiones	Tiempo Keras (s)	Tiempo Numpy (s)	Tiempo FPGA (s)	GFLOPS
32x32	7.70e-03	1.60e-03	5.51e-05	0.59
64x64	6.89e-03	1.00e-04	1.08e-04	2.43
128x128	7.70e-03	2.00e-04	3.80e-04	5.51
256x256	9.70e-03	1.10e-03	2.27e-03	7.39
512x512	1.75e-02	7.00e-03	1.69e-02	7.94
1024x1024	9.33e-02	4.88e-02	1.99e-01	5.37
2048x2048	5.94e-01	3.74e-01	1.74e+00	4.95
4096x4096	4.44e+00	2.93e+00	1.56e+01	4.41

Tabla 10: Tiempos de ejecución para producto matricial sin función de activación con BS=16 y SWI=4

Dimensiones matriz	Numpy(s)/FPGA(s)	Keras(s)/FPGA(s)
32x32	33.94	161.16
64x64	1.12	73.98
128x128	0.65	22.47
256x256	0.52	4.85
512x512	0.42	1.13
1024x1024	0.24	0.43
2048x2048	0.21	0.34
4096x4096	0.19	0.27

Tabla 11: SpeedUp para producto matricial sin función de activación con BS=16 y SWI=4

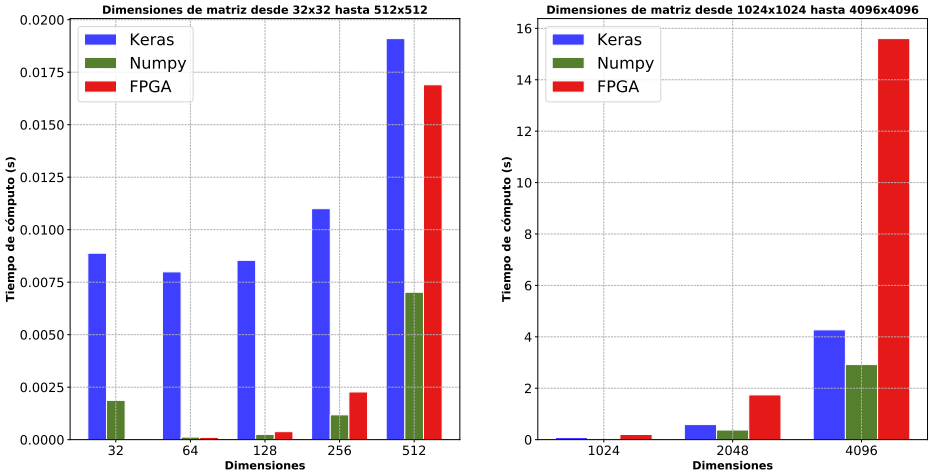


Figura 20: Gráfica comparativa de los tiempos de cómputo para producto matricial con BS=16 y SWI=4

La tendencia observada para matrices con tamaños inferiores a 512 sigue siendo que los mejores tiempos de cómputo corresponden a Numpy y la FPGA. A medida que los tamaños son iguales o superiores a 512 la FPGA consigue la peor marca, seguido de Keras y finalmente Numpy continua consiguiendo los mejores tiempos. Al haber aumentado el BS de 8 a 16 se ha conseguido bajar los tiempos de cómputo para matrices de 2048×2048 de 4.93 segundos a 1.74 y para dimensiones de 4096×4096 se ha pasado de 58.7 segundos a 15.6 segundos, lo que implica un speed up de 2.83 y 3.76 respectivamente.

4.2.3 Recursos FPGA utilizados, tiempos de ejecución y *SpeedUp* para BS=32 y SWI=2

Finalmente se exponen los recursos utilizados por la FPGA para una arquitectura con un BS=32 y un SWI=2. Ha sido necesario bajar el SWI a 2 para no consumir todas las ALUTS disponibles en la FPGA.

Kernel Name	ALUTs	FFs	RAMs	DSPs
matrixMult	97293	82498	269	72
Global Interconnect	6577	8915	0	0
Board Interface	24256	20822	143	0
Total	128126 (57 %)	112235 (25 %)	412 (34 %)	72 (21 %)
Available	227120	454240	1220	342

Tabla 12: Utilización de recursos FPGA para BS=32 y SWI=2

Los tiempos de computo y speed up comparativos entre Keras y Numpy para matrices cuadradas desde 32×32 hasta 4096×4096 se muestran en la siguientes tablas:

Dimensiones	Tiempo Keras (s)	Tiempo Numpy (s)	Tiempo FPGA (s)	GFLOPS
32x32	7.70e-03	1.60e-03	7.85e-05	0.42
64x64	7.70e-03	1.00e-04	9.34e-05	2.80
128x128	8.50e-03	2.99e-04	3.33e-04	6.27
256x256	1.05e-02	1.10e-03	2.21e-03	7.59
512x512	1.88e-02	7.49e-03	2.17e-02	6.18
1024x1024	1.58e-01	5.79e-02	3.17e-01	3.38
2048x2048	7.52e-01	4.20e-01	1.08e+00	7.95
4096x4096	4.81e+00	3.06e+00	8.86e+00	7.75

Tabla 13: Tiempos de ejecución para producto matricial sin función de activación con BS=32 y SWI=2

Dimensiones matriz	Numpy(s)/FPGA(s)	Keras(s)/FPGA(s)
32x32	23.82	113.12
64x64	1.30	85.55
128x128	0.75	25.64
256x256	0.53	4.98
512x512	0.33	0.88
1024x1024	0.15	0.27
2048x2048	0.34	0.54
4096x4096	0.33	0.48

Tabla 14: SpeedUp para producto matricial sin función de activación con BS=32 y SWI=2

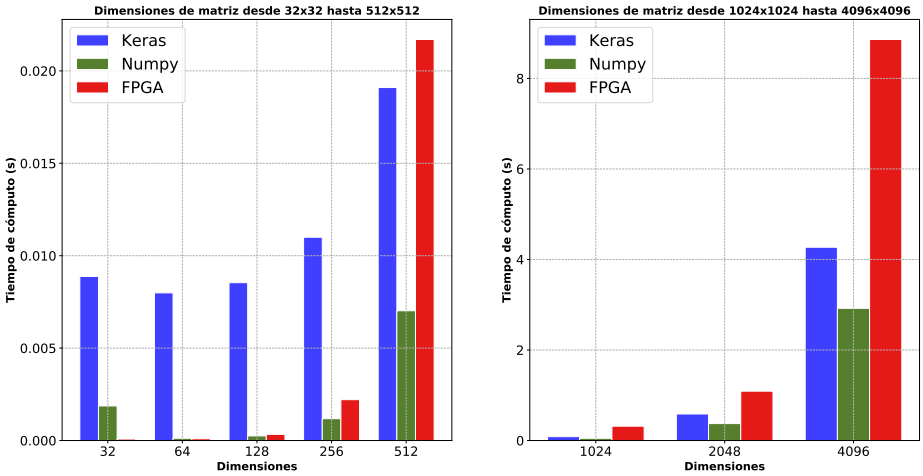


Figura 21: Gráfica comparativa de los tiempos de cómputo para producto matricial con BS=32 y SWI=2

De nuevo se cumple que para matrices con tamaños inferiores a 512 los mejores tiempos de cómputo corresponden a Numpy y la FPGA. Numpy parece ser el mas óptimo de los tres para todos los tamaños. A medida que los tamaños son iguales o superiores a 512 la FPGA consigue la peor marca, seguido de Keras y finalmente Numpy continua consiguiendo los mejores tiempos. Nuevamente se demuestra el papel del BS. En este caso, para un BS=32, comparándolo con un BS=16, para tamaños de 2048×2048 se pasa de 1.74 segundos a 1.08 segundos, lo que implica un speed up de 1.61 y para matrices de 4096×4096 se pasa de 15.6 segundos 8.68 segundos, consiguiendo un speed up de 1.79.

4.3 Análisis y conclusiones del tamaño de BS

Una vez vez vistos los tiempos de ejecución para cada BS comparado con Numpy y Keras, se expone a continuación una tabla donde solo se analizan los tiempos de ejecución de la FPGA en función del BS:

Tiempos de ejecución para la FPGA en función del BS			
Dimensiones matrix	BS=8, SWI=4 Tiempo (s)	BS=16, SWI=4 Tiempo (s)	BS=32, SWI=2 Tiempo (s)
32x32	6.08e-05	5.51e-05	7.85e-05
64x64	1.48e-04	1.08e-04	9.34e-05
128x128	6.03e-04	3.80e-04	3.33e-04
256x256	4.08e-03	2.27e-03	2.21e-03
512x512	3.18e-02	1.69e-02	2.17e-02
1024x1024	3.19e-01	1.99e-01	3.17e-01
2048x2048	4.93e+00	1.74e+00	1.08e+00
4096x4096	5.87e+01	1.56e+01	8.86e+00

Tabla 15: Tiempos de ejecución para la FPGA en función del BS

En la siguiente gráfica se visualiza la comparativa de los tiempos de cómputo en la FPGA para diferentes tamaños de BS. Se aprecia como para dimensiones de matriz superiores a 512 el tiempo de cómputo decrece de forma notable, pasando de 58.7 segundos a 8.68 segundos en matrices de 4096 × 4096 y un BS=8 y BS=32.

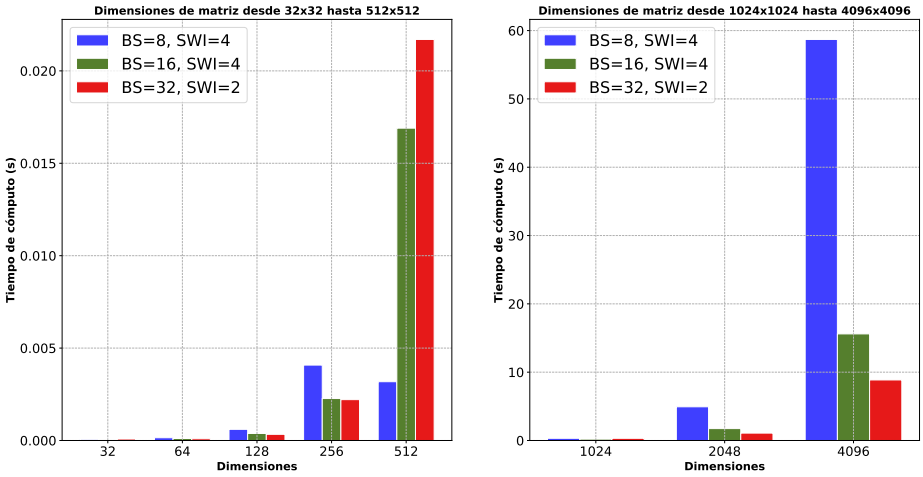


Figura 22: Comparativa de tiempos de cómputo para FPGA y BS de 8, 16 y 32

Los mejores tiempos de ejecución corresponden a un BS mas alto, en este caso de 32, que era lo máximo que permitía el compilador offline de Intel en cuanto a recursos se refiere. A partir de un BS de 64 el número de ALUTS utilizadas excedía del 100 %. Si el parámetro de BS se incrementara, como ha podido comprobar Rafael Gadea en sus pruebas con otras FPGA, los tiempos de cómputo mejoran e incluso llegan a superar a la librerías de Numpy y Keras. En el caso de Rafael, ha conseguido un kernel con un BS=128. Es por tanto esta arquitectura, la que tiene un BS=32 y un SWI=2 la que se va a elegir para implementar el kernel que alojará las funciones de activación y la adición del bias. Remarcar que en el análisis del algoritmo de convolución se tendrá que bajar el BS a 16 puesto que se excederá del 100 % el número de ALUTS al compilar el kernel. Esto, a priori puede ser una ventaja ya que un BS menor realiza menos iteraciones en el producto matricial debido a que es capaz de tener en cuenta menos elementos nulos en función de la máscara.

4.4 Tiempos de ejecución para BS=32 y funciones de activación

Se exponen a continuación los tiempos de ejecución, errores medios cuadráticos y *SpeedUp* para las funciones de activación Relu, Sigmoide y Tangente Hiperbólica para la FPGA con BS=32 y Keras. Los recursos utilizados teniendo en cuenta la implementación de las funciones de activación y la adición del bias en la FPGA son:

Kernel Name	ALUTs	FFs	RAMs	DSPs
matrixMult	97293	82498	269	72
Global Interconnect	6577	8915	0	0
Board Interface	24256	20822	143	0
Total	128126 (56 %)	112235 (25 %)	412 (34 %)	72 (21 %)
Available	227120	454240	1220	340

Tabla 16: Utilización de recursos FPGA para BS=32 y SWI=2

4.4.1 Tiempos de ejecución y *SpeedUp* para BS=32 y SWI=2 y función de activación Relu

Se exponen a continuación los tiempos de ejecución para la función ReLU, mostrando una tabla con los mismos y una gráfica comparativa para observar de forma rápida las diferencias en los tiempos de cómputo entre Keras y la FPGA.

Dimensiones	Tiempo Keras (s)	Tiempo FPGA (s)	Keras(s)/FPGA(s)
32x32	1.15e-02	1.00e-04	115
64x64	1.10e-02	1.00e-04	110
128x128	1.20e-02	4.00e-04	29.8
256x256	1.37e-02	2.20e-03	6.23
512x512	2.35e-02	2.31e-02	1.02
1024x1024	1.96e-01	1.35e-01	1.46
2048x2048	7.20e-01	1.07e+00	0.68
4096x4096	4.47e+00	8.50e+00	0.53

Tabla 17: Tiempos de ejecución y *Speed Up* para producto matricial con función de activación Relu con BS=32 y SWI=2

La gráfica permite ver como la FPGA sigue siendo mas rápida que Keras hasta dimensiones de 512×512.

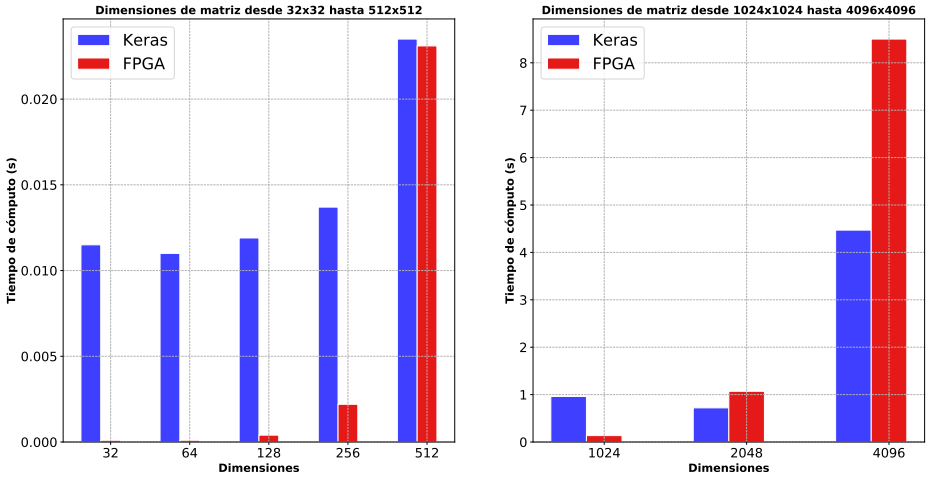


Figura 23: Gráfica comparativa de los tiempos de cómputo para producto matricial con BS=32 y función ReLU

4.4.2 Tiempos de ejecución y *SpeedUp* para BS=32 y SWI=2 y función de activación Sigmoide

En este caso expone a continuación los tiempos de ejecución para la función Sigmoide, mostrando una tabla con los mismos y una gráfica comparativa para observar de forma rápida las diferencias en los tiempos de cómputo entre Keras y la FPGA.

Dimensiones matriz	Tiempo Keras (s)	Tiempo FPGA (s)	Keras(s)/FPGA(s)
32x32	1.29e-02	1.00e-04	129
64x64	1.07e-02	1.00e-04	107
128x128	1.13e-02	4.00e-04	28.2
256x256	1.41e-02	2.20e-03	6.41
512x512	2.20e-02	2.26e-02	0.97
1024x1024	1.07e-01	1.34e-01	0.80
2048x2048	6.16e-01	1.06e+00	0.58
4096x4096	6.29e+00	8.50e+00	0.74

Tabla 18: Tiempos de ejecución y *Speed Up* para producto matricial con función de activación Sigmoide con BS=32 y SWI=2

Aún tratándose de una función de activación mas compleja que la ReLU, la gráfica sigue mostrando de forma clara como la FPGA es superior a Keras hasta dimensiones de 512×512.

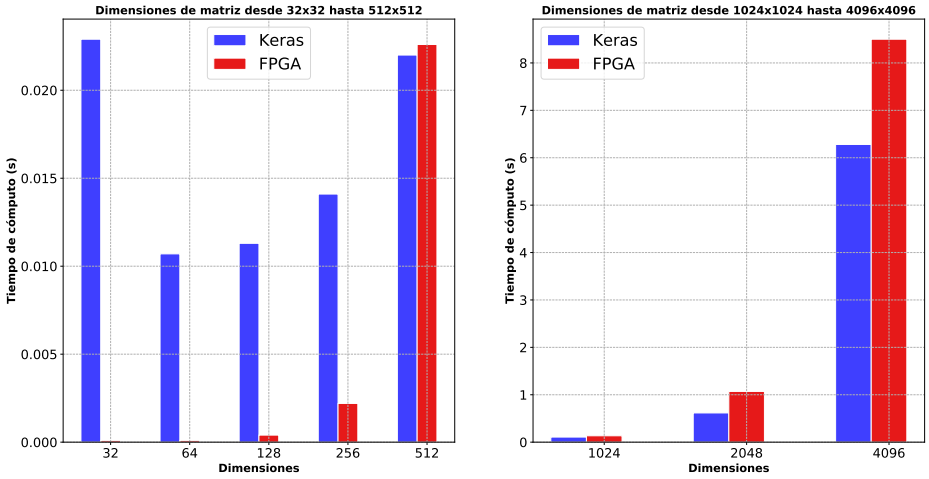


Figura 24: Gráfica comparativa de los tiempos de cómputo para producto matricial con BS=32 y función sigmoide

4.4.3 Tiempos de ejecución y *SpeedUp* para BS=32 y SWI=2 y función de activación Tangente hiperbólica

Se analizan en última instancia los tiempos de ejecución para la función Tangente Hiperbólica, mostrando una tabla con los mismos y una gráfica comparativa para observar de forma rápida las diferencias en los tiempos de cómputo entre Keras y la FPGA.

Dimensiones matriz	Tiempo Keras (s)	Tiempo FPGA (s)	Keras(s)/FPGA(s)
32x32	1.04e-02	1.00e-04	104
64x64	1.06e-02	1.00e-04	106
128x128	1.12e-02	4.00e-04	28
256x256	1.36e-02	2.20e-03	6.18
512x512	2.26e-02	2.28e-02	0.99
1024x1024	1.04e-01	1.34e-01	0.78
2048x2048	6.25e-01	1.07e+00	0.58
4096x4096	4.48e+00	8.50e+00	0.53

Tabla 19: Tiempos de ejecución y *Speed Up* para producto matricial con función de activación Tangente Hiperbólica con BS=32 y SWI=2

De nuevo para una función de activación mas compleja que la ReLU, la gráfica sigue mostrando de forma clara como la FPGA es superior a Keras hasta dimensiones de 512×512.

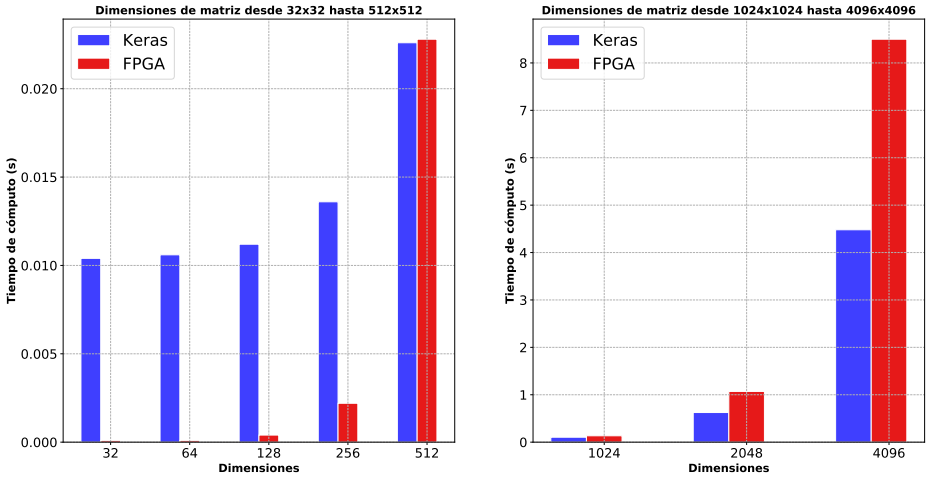


Figura 25: Gráfica comparativa de los tiempos de cómputo para producto matricial con BS=32 y función tangente hiperbólica

En el análisis de las funciones de activación se puede apreciar como de nuevo la FPGA es notablemente superior a Keras para dimensiones menores de 512x512, siendo a partir de 512x512 donde Keras de nuevo comienza a superar a la FPGA, obteniendo unos resultados muy similares al producto matricial sin función de activación. Se confirma por tanto que la función de activación no interfiere en el tiempo de computo, algo que reafirma como la implementación *"in situ"* de las funciones de activación justo a la salida del producto matricial beneficia en el tiempo de cómputo.

4.5 Tiempos de ejecución para algoritmo de Convolución 2D

Se analizan a continuación los tiempos de ejecución para el algoritmo de convolución en 2 dimensiones. Puesto que la lógica utilizada por la FPGA superaba el 100 % de ALUTS para un BS de 32 al implementar la lógica del algoritmo de convolución, se ha bajado este a 16 pero aumentado su SWI a 4, lo que produce una significante mejora.

Las medidas realizadas hacen referencia a imágenes con resoluciones de 32x32, 64x64, 128x128 y 256x256 y tamaños de filtros de 3, 5, 7 y 9. El número de inputs (batch) para cada prueba es de 16, 32, 64 y 128. El filtro aplicado para la realización de las pruebas es un *Gaussian Blur*, aunque es posible aplicar cualquier filtro conocidos sus coeficientes. El método utilizado para la realización de los tests es mediante la matriz de convolución y desplazamiento de índice.

(Nº imágenes, F, C[5])	(F, C[6], Nº filtros)	FPGA (s)[7]	Keras(s)[8]	Speed Up[9]
(16, 32, 32)	(3, 3, 1)	0.0003	0.0032	9.4
(32, 32, 32)	(3, 3, 1)	0.0005	0.0028	5.35
(64, 32, 32)	(3, 3, 1)	0.001	0.0066	6.6
(128, 32, 32)	(3, 3, 1)	0.0021	0.023	11.01
(16, 32, 32)	(5, 5, 1)	0.0005	0.0043	8.89
(32, 32, 32)	(5, 5, 1)	0.0008	0.0092	10.86
(64, 32, 32)	(5, 5, 1)	0.0017	0.0088	5.19
(128, 32, 32)	(5, 5, 1)	0.0034	0.0173	5.16
(16, 32, 32)	(7, 7, 1)	0.0006	0.0041	6.56
(32, 32, 32)	(7, 7, 1)	0.0013	0.0062	4.59
(64, 32, 32)	(7, 7, 1)	0.0024	0.0179	7.5
(128, 32, 32)	(7, 7, 1)	0.0047	0.0284	6.03
(16, 32, 32)	(9, 9, 1)	0.0009	0.0055	6.42
(32, 32, 32)	(9, 9, 1)	0.0016	0.0168	10.42
(64, 32, 32)	(9, 9, 1)	0.0032	0.0167	5.2
(128, 32, 32)	(9, 9, 1)	0.0063	0.0292	4.65

Tabla 20: Tiempos de ejecución para convolución 2D e imágenes de entrada de 32x32

(Nº imágenes, F, C)	(F, C, Nº filtros)	FPGA (s)	Keras(s)	Speed Up
(16, 64, 64)	(3, 3, 1)	0.0018	0.0052	2.94
(32, 64, 64)	(3, 3, 1)	0.0036	0.0127	3.52
(64, 64, 64)	(3, 3, 1)	0.0071	0.0298	4.21
(128, 64, 64)	(3, 3, 1)	0.0144	0.0373	2.59
(16, 64, 64)	(5, 5, 1)	0.003	0.007	2.34
(32, 64, 64)	(5, 5, 1)	0.006	0.012	1.99
(64, 64, 64)	(5, 5, 1)	0.012	0.0228	1.91
(128, 64, 64)	(5, 5, 1)	0.0239	0.0564	2.36
(16, 64, 64)	(7, 7, 1)	0.0042	0.0136	3.24
(32, 64, 64)	(7, 7, 1)	0.0084	0.0275	3.27
(64, 64, 64)	(7, 7, 1)	0.0168	0.0413	2.46
(128, 64, 64)	(7, 7, 1)	0.0335	0.0719	2.15
(16, 64, 64)	(9, 9, 1)	0.0055	0.0131	2.37
(32, 64, 64)	(9, 9, 1)	0.0111	0.0316	2.86
(64, 64, 64)	(9, 9, 1)	0.022	0.0602	2.73
(128, 64, 64)	(9, 9, 1)	0.044	0.1286	2.92

Tabla 21: Tiempos de ejecución para convolución 2D e imágenes de entrada de 64x64

(Nº imágenes, F, C)	(F, C, Nº filtros)	FPGA (s)	Keras(s)	Speed Up
(16, 128, 128)	(3, 3, 1)	0.0134	0.0351	2.62
(32, 128, 128)	(3, 3, 1)	0.0268	0.0335	1.25
(64, 128, 128)	(3, 3, 1)	0.054	0.102	1.89
(128, 128, 128)	(3, 3, 1)	0.1081	0.1309	1.21
(16, 128, 128)	(5, 5, 1)	0.0223	0.0222	0.99
(32, 128, 128)	(5, 5, 1)	0.0451	0.0671	1.49
(64, 128, 128)	(5, 5, 1)	0.0902	0.1026	1.14
(128, 128, 128)	(5, 5, 1)	0.1802	0.2023	1.12
(16, 128, 128)	(7, 7, 1)	0.0315	0.0417	1.32
(32, 128, 128)	(7, 7, 1)	0.0631	0.0845	1.34
(64, 128, 128)	(7, 7, 1)	0.1261	0.1781	1.41
(128, 128, 128)	(7, 7, 1)	0.2522	0.4354	1.73
(16, 128, 128)	(9, 9, 1)	0.0411	0.0601	1.46
(32, 128, 128)	(9, 9, 1)	0.0822	0.1322	1.61
(64, 128, 128)	(9, 9, 1)	0.1642	0.228	1.39
(128, 128, 128)	(9, 9, 1)	0.3284	0.5874	1.79

Tabla 22: Tiempos de ejecución para convolución 2D e imágenes de entrada de 128x128

(Nº imágenes, F, C)	(F, C, Nº filtros)	FPGA (s)	Keras(s)	Speed Up
(16, 256, 256)	(3, 3, 1)	0.104	0.0873	0.84
(32, 256, 256)	(3, 3, 1)	0.2098	0.13	0.62
(64, 256, 256)	(3, 3, 1)	0.4194	0.2546	0.61
(128, 256, 256)	(3, 3, 1)	0.8388	0.5879	0.7
(16, 256, 256)	(5, 5, 1)	0.1746	0.1177	0.67
(32, 256, 256)	(5, 5, 1)	0.3496	0.2349	0.67
(64, 256, 256)	(5, 5, 1)	0.699	0.6109	0.87
(128, 256, 256)	(5, 5, 1)	1.3981	1.0445	0.75
(16, 256, 256)	(7, 7, 1)	0.2446	0.2586	1.06
(32, 256, 256)	(7, 7, 1)	0.4894	0.5078	1.04
(64, 256, 256)	(7, 7, 1)	0.9787	0.7651	0.78
(128, 256, 256)	(7, 7, 1)	1.9575	1.3948	0.71
(16, 256, 256)	(9, 9, 1)	0.3166	0.363	1.15
(32, 256, 256)	(9, 9, 1)	0.6335	0.5098	0.8
(64, 256, 256)	(9, 9, 1)	1.2668	1.2584	0.99
(128, 256, 256)	(9, 9, 1)	2.5341	2.0453	0.81

Tabla 23: Tiempos de ejecución para convolución 2D e imágenes de entrada de 256x256

Como se puede apreciar en los resultados, se observa una dependencia lineal entre las tres variables utilizadas: Batch, tamaño de imagen y tamaño de filtro. A través de regresión múltiple lineal se extraen las siguientes ecuaciones que modelarían el tiempo de ejecución, para ello se han puesto tanto el tamaño del kernel como el batch size en función del tamaño de entrada de imagen.

$$t(I_s, F_s, B_s) = e^{-8.2705+ln(I_s \cdot 0.02504+F_s \cdot 0.18334+B_s \cdot 0.0172)} \tag{17}$$

Donde I_s, F_s y B_s corresponden al tamaño de la imagen, kernel y batch size.

De los coeficientes anteriores se extrapola que el parámetro que mas penaliza al algoritmo de convolución es el tamaño del filtro, puesto que es aquel mayor de los 3. También en las tablas se puede observar esa condición, siendo siempre para un tamaño de filtro de (9,9) el mayor de los tiempos de computación. Esto se explica debido a la máscara para productos de matrices dispersas, ya que al tener un mayor tamaño de filtro y al ser el tamaño de BS fijo, esta tiene en cuenta un mayor número de elementos nulos por lo que tiene que realizar un mayor número de operaciones entre la matriz de convolución y la señal de entrada.

En relación a la comparativa entre la convolución con Keras y la FPGA, los resultados obtenidos son bastante buenos ya que no es hasta tamaños de imagen de 256x256 cuando la librería de Keras computa mas rápido la convolución, siendo para tamaños de imagen pequeños (32x32) cuando se obtiene el mejor *Speed Up* de la FPGA sobre Keras.

Las marcas obtenidas para convoluciones con mas de un filtro son peores que las de Keras

puesto que se la salida de cada filtro corresponde a un producto matricial del input por su respectivo filtro, por lo que los tiempos obtenidos son muy próximos a $N \cdot t_1$, donde N hace referencia al número de filtros y t_1 al tiempo de cómputo de la convolución para un solo filtro. El tamaño de los filtros elegidos para esta prueba ha sido de 3×3, 5×5 y 7×7 e imágenes de 64x64 para un batch de 16 y un número de filtros de 16.

(Nº imágenes, F, C)	(F, C, Nº filtros)	FPGA (s)	Keras(s)	Speed Up
(16, 64, 64)	(3, 3, 16)	0.0292	0.0050	0.17
(16, 64, 64)	(5, 5, 16)	0.0514	0.0070	0.136
(16, 64, 64)	(7, 7, 16)	0.0689	0.0136	0.197

Tabla 24: Tiempos de ejecución para convolución 2D e imágenes de entrada de 64x64 con 4, 8 y 16 filtros mediante matriz de *Toeplitz*

A continuación se expone la comparativa de tiempos entre los métodos de convolución *Im2Row* y de Keras. Se puede observar como la técnica *Im2Row* es la mas apta para realizar mediante la FPGA al aprovechar todo el potencial del algoritmo de BS a diferencia de la técnica mediante matriz de Toeplitz.

(Nº imágenes, F, C)	(F, C, Nº filtros)	FPGA (s)	Keras(s)	Speed Up
(16, 64, 64)	(3, 3, 16)	0.0086	0.0050	0.58
(16, 64, 64)	(5, 5, 16)	0.0086	0.0070	0.814
(16, 64, 64)	(7, 7, 16)	0.03417	0.0136	0.4

Tabla 25: Tiempos de ejecución para convolución 2D e imágenes de entrada de 64x64 con 16, 32, 64 y 128 filtros mediante técnica *Im2Row*

De las tablas anteriores se demuestra como la técnica de Im2Row es mas óptima para convoluciones con mas de un filtro pero sigue obteniendo tiempos menores que la implementada por Keras.

4.6 Resultados convolución 2D

En este capítulo se presentan los resultados obtenidos para las capas implementadas en FPGA que hacen uso de la convolución. La razón por la que en el capítulo anterior solo se ha analizado el tiempo para las convolución 2D es porque todas las capas listadas: *Conv2D*, *UpSampling2D*, *AveragePooling2D* y *MaxPooling2D* hacen uso del mismo algoritmo de convolución, lo único que cambia es el filtro utilizado.

Las imágenes siguientes muestran la salida de las imágenes convolucionadas tanto para la FPGA como para Keras, así como la imagen inicial sin filtrar. El tipo de filtro utilizado para este ejemplo ha sido inicializado de forma aleatoria mediante el tipo *Glorot Uniform*.

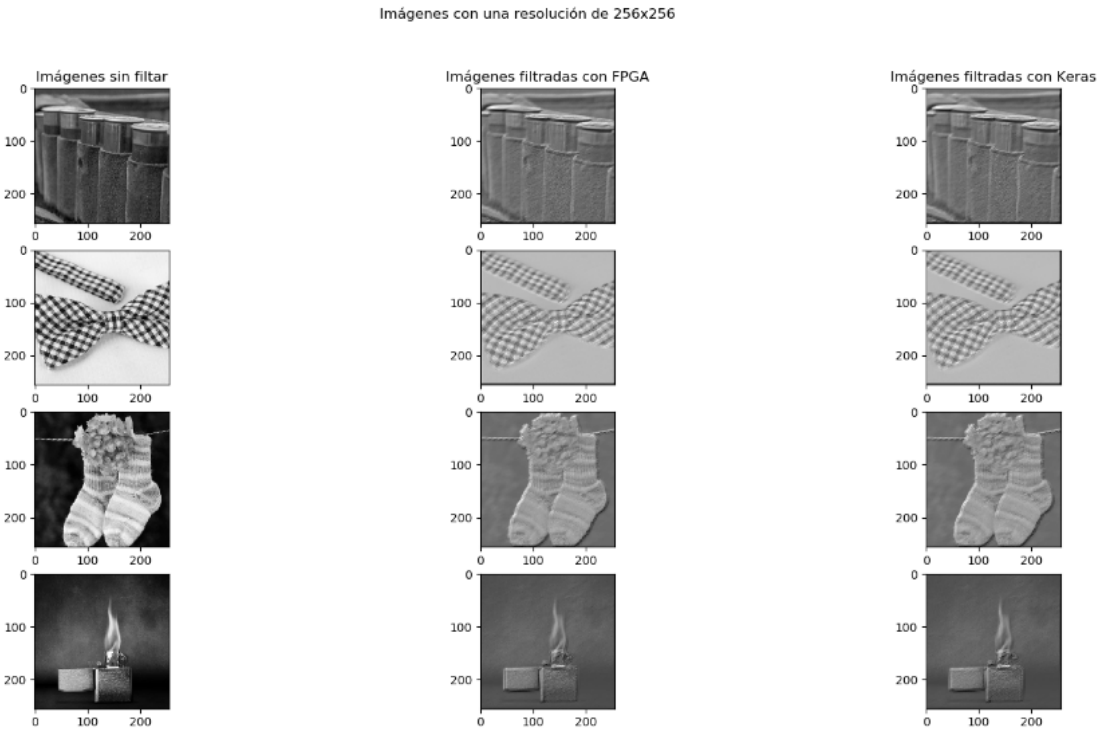


Figura 26: Imágenes filtradas mediante Convolución 2D con filtro *Glorot Uniform* y resolución 256x256

A la vista de los resultados parece cumplirse que el filtrado realizado por la FPGA se corresponde con el realizado por Keras. La siguiente tabla muestra el error medio cuadrático, *mse*, entre Keras y la FPGA. Para el cálculo del mse en esta ocasión se ha tenido en cuenta todos los batchs y tamaños de filtro para cada resolución, por tanto, el error que se presenta hace referencia a cada resolución.

Resolución	mse
(32x32)	5.7881e-10
(64x64)	6.4155-10
(128x128)	5.2164e-10
(256x256)	5.2354e-10

Tabla 26: Errores medios cuadráticos en función de la resolución

En las siguientes imágenes se muestran convoluciones para filtros de tipo *Edge Detection* y *Gaussian Blur* como muestra de la posibilidad de filtrar imágenes con cualquier kernel deseado a través de la FPGA. El tamaño de las imágenes vuelve a ser de 256x256.

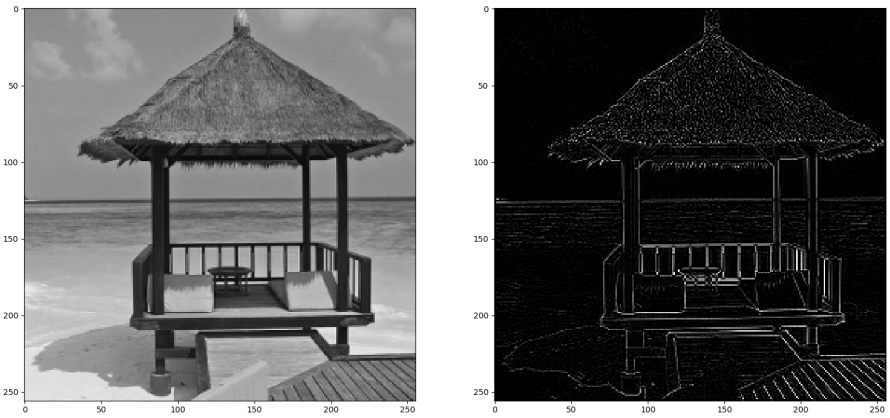


Figura 27: Imagen filtradas mediante Convolución 2D con filtro *Edge Detection* de tamaño 3x3 y resolución 256x256

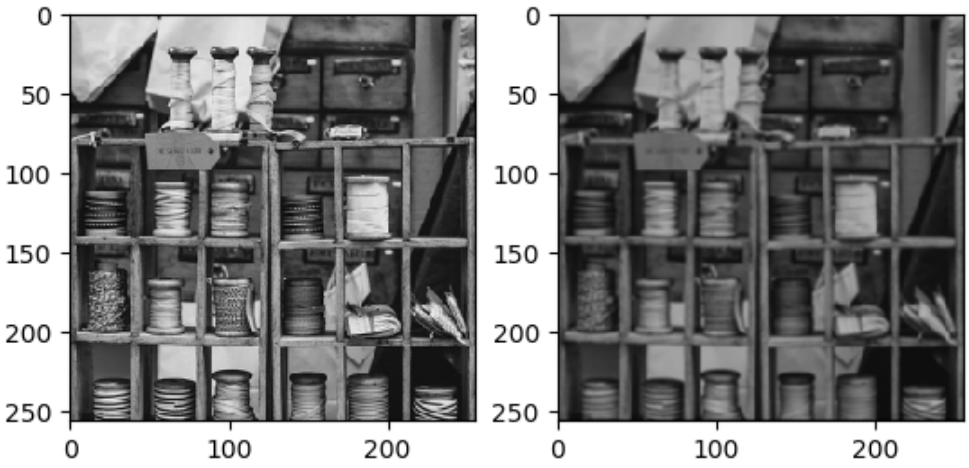


Figura 28: Imagen filtrada mediante Convolución 2D con filtro *Edge Detection* de tamaño 9x9 y resolución 256x256

4.7 Resultados UpSampling 2D

La técnica de UpSampling 2D permite reescalar una imagen mediante interpolación. Como se ha explicado en el capítulo 4 se ha hecho uso de interpolación de tipo *nearest* y *bilinear*. Los resultados obtenidos son:

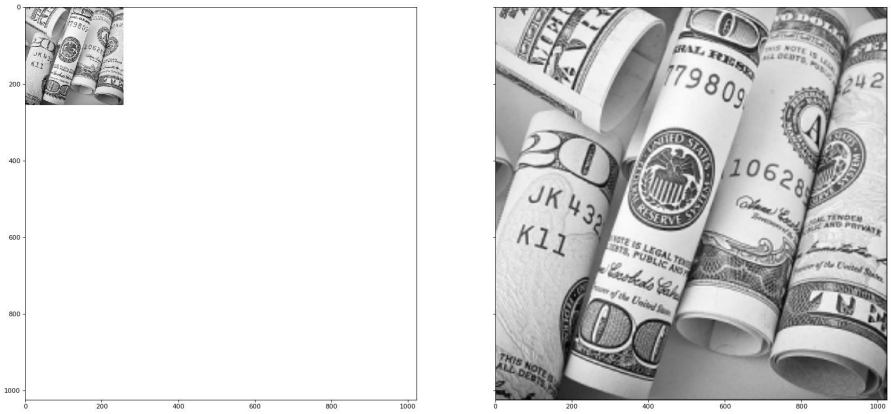


Figura 29: Interpolación bilinear para imagen de entrada de 256x256 y un factor de 4x4 con interpolación bilineal

En las siguientes imágenes se realiza una comparativa de como afecta el tipo de método a la hora de interpolar, ya sea *"nearest"* o *"bilinear"*. En la segunda figura se puede apreciar como una interpolación bilinear ayuda al suavizado de la imagen ofreciendo mejores resultados para una misma resolución dada.

Figura 30: Comparativa entre Interpolación nearest y bilinear para imagen de entrada de 256x256 y un factor de 2x2. Izquierda: imagen original, centro: Interp. nearest, derecha: Interp. bilinear

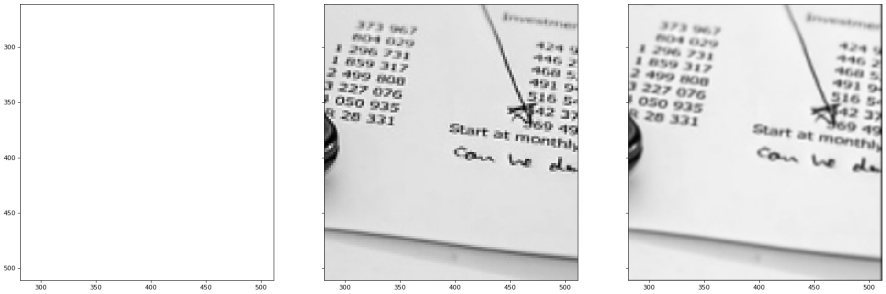


Figura 31: Detalle de comparativa entre Interpolación nearest y bilinear para imagen de entrada de 256x256 y un factor de 2x2. Izquierda: imagen original, centro: Interp. nearest, derecha: Interp. bilinear

Capítulo 5. Integración de OpenCL con Tensorflow & Keras

5.1 Motivación

Existen paquetes como *OpenVINO* de Intel que integran FPGA de Intel con Tensorflow y Keras, pero estos paquetes solo son capaces de generar kernels para redes CNN, dejando fuera una amplia variedad de tipo de NN's, que no necesariamente han de ser CNN, por lo que se ha decidido crear una clase en python que analiza los ficheros de modelo de Keras[10] y crea la estructura lógica necesaria para poder realizar con éxito predicciones a través de la FPGA.

Esta herramienta está también pensada para que, por ejemplo, un modelo entrenado y adaptable a la FPGA pueda ser embebido en la misma. Pudiendo, por ejemplo, crear una PCB específica para una máquina concreta en el ámbito de la automoción, biomedicina... etc. No todos los tipos de capa de red neuronal son adaptables al kernel OpenCL realizado para este TFG. A continuación se listan los tipos de capa de red neuronal que aceptaría la clase de python generada[11]

- Dense:
  - Relu con umbral fijo
  - Tangente hiperbólica
  - Sigmoide
  - Softmax
- Convolution2D
  - padding=valid
  - padding=same
- UpSampling 2D
  - method=nearest
  - method=bilinear
- Max Pooling 2D
- Average Pooling 2D

[10] Estos ficheros suelen ser .H5 o .json, la clase generada hace uso de los ficheros .H5 ya que están mas extendidos en los modelos de Tensorflow/Keras

[11] Dada la amplia variedad de tipos de capa de red neuronal, se hacía muy extenso intentar implementar muchas de las, además de la limitada información que existe en cuanto a la implementación de los algoritmos de los que hace uso Tensorflow/Keras

5.2 De Tensorflow/Keras a FPGA

La clase encargada de realizar este proceso es *model_manage.py*. Esta clase crea el kernel pyopencl y todos los buffers necesarios para la ejecución de las predicciones de la red neuronal. el proceso que sigue es:

1. Creación del kernel pyopenCL.
2. Lectura del fichero de modelo TF/Keras en formato .h5.
3. Comprobación de que ese modelo concreto todavía no se ha migrado a FPGA.
4. Si no se ha migrado el modelo: Se crean los buffers para cada capa teniendo en cuenta el tipo de capa y los pesos asociados a esta.
5. Creación de toda la lógica de control y ejecución: *workflow* para la propagación de la señal a través de las diferentes capas.
6. Guardado del modelo en el fichero de variables globales para reutilización del mismo en distintas predicciones. *model_globals.py*.

Para poder realizar todo este proceso, es necesario que el tipo de dato de las matrices obtenidas del modelo de TF/Keras sean de tipo *float32* así como las dimensiones sean múltiplos del BS. Finalmente, y con la intención de que el cálculo de las predicciones sea lo mas óptima posible, toda la estructura del modelo FPGA se guarda en el fichero de variables globales mencionado anteriormente. Por lo tanto, cuando se realiza una predicción, la clase busca el modelo creado y trae de vuelta todas las variables de interés para el mismo. Las variables que el modelo guarda son:

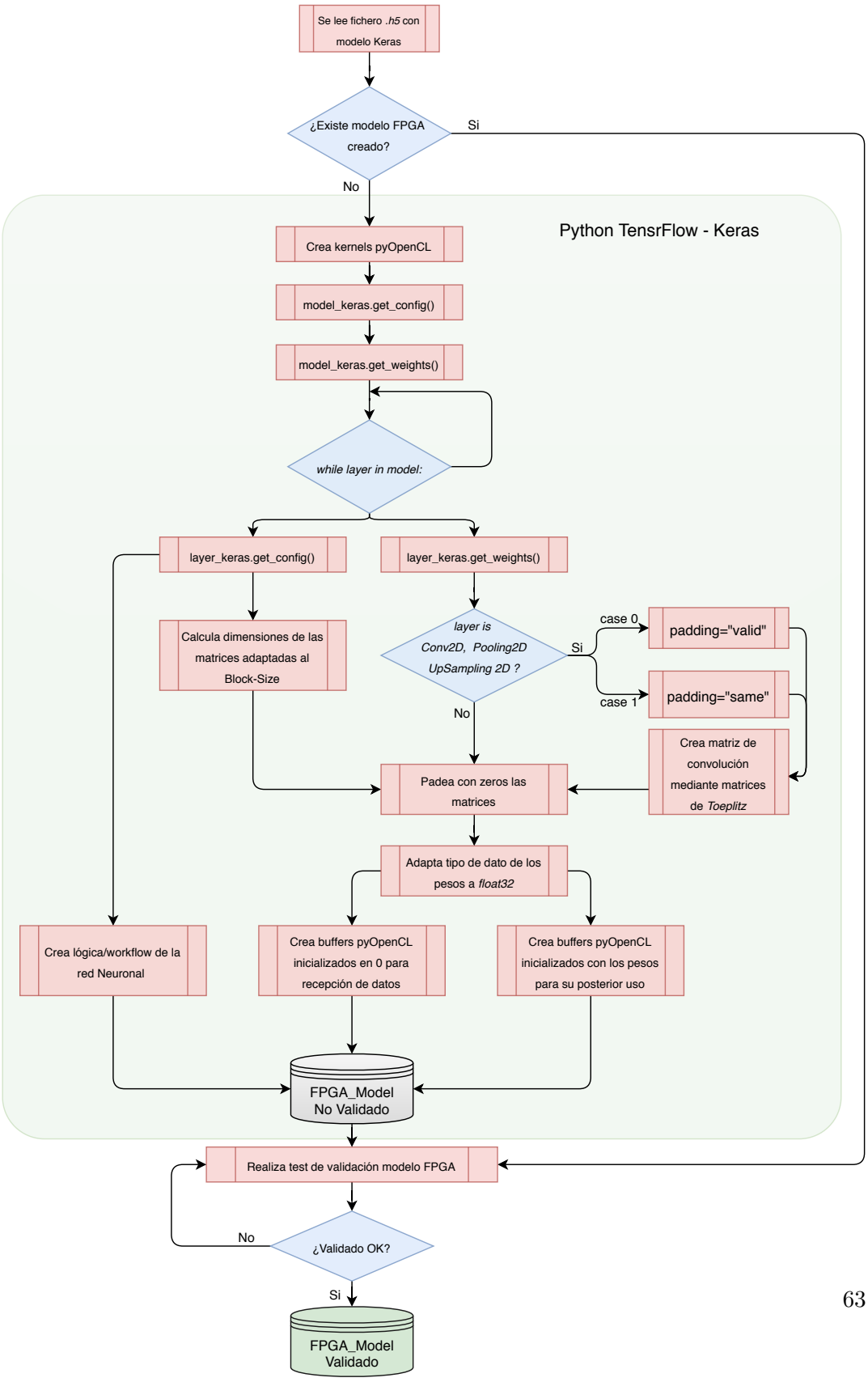
- DIMS: Dimensiones óptimas adptadas al *BLOCK_SIZE*.
- WEIGHTS_BUFFERS: Buffers con los pesos ya adaptados.
- LAYERS_BUFFERS: Buffers rellenos con zeros para almacenar el resultado de salida. Solo si es preciso. En este punto recordar que lo optimo no es almacenar el resultado en el buffer de salida, si no dejarlo en el buffer pyopencl para aprovecharlo en la propagación de la señal hacia la siguiente capa de red neuronal.

Desde el punto de vista de la creación y optimización del kernel, solo es necesario crear una vez el kernel pyOpenCL. Este kernel estará disponible para todos los modelos FPGA que se hayan creado, por lo que también residirá en el fichero de variables globales, *model_globals*. Las variables que afectan a la creación y manejo del kernel son:

- PLATFORMS: Plataformas FPGA disponibles.

- DEVICES: Dispositivos FPGA disponibles.
- CTX: Contexto necesario para el manejo de buffers y ejecución de kernels.
- BINARY: Fichero binario extraido del fichero *.aocx*
- PRG: Programa creado a través del fichero BINARY.
- KERNEL_PAD: Kernel correspondiente al padeo de imágenes para adaptación de señales procesadas mediante Convolución 2D.
- KERNEL_MULT: Kernel principal que realiza el producto matricial.
- QUEUE: Queue necesaria para el manejo de buffers y ejecución de kernels.

El siguiente flujo-grama muestra los pasos seguidos por la clase de python para generar el modelo FPGA:


Se lee fichero .h5 con modelo Keras
¿Existe modelo FPGA creado?
Si
No
Python TensrFlow - Keras
Crea kernels pyOpenCL
model_keras.get_config()
model_keras.get_weights()
while layer in model:
layer_keras.get_config()
layer_keras.get_weights()
Calcula dimensiones de las matrices adaptadas al Block-Size
layer is Conv2D, Pooling2D UpSampling 2D ?
Si
case 0
padding="valid"
case 1
padding="same"
No
Crea matriz de convolución mediante matrices de Toeplitz
Padea con zeros las matrices
Adapta tipo de dato de los pesos a float32
Crea lógica/workflow de la red Neuronal
Crea buffers pyOpenCL inicializados en 0 para recepción de datos
Crea buffers pyOpenCL inicializados con los pesos para su posterior uso
FPGA_Model No Validado
Realiza test de validación modelo FPGA
No
¿Validado OK?
Si
FPGA_Model Validado

Una vez el modelo ha sido creado, es posible realizar predicciones con este. Siempre y cuando se cumpla que las capas contenidas en el modelo de TF/Keras se correspondan con las mencionadas anteriormente, la predicción deberá de ser satisfactoria y similar a la de TF/Keras.

Con el ánimo de simplificar la aplicación creada, se ha generado un GUI[12] donde se selecciona el fichero .H5 a través de un desplegable. El GUI da la opción de seleccionar el tamaño de batch para la implementación del modelo. Si se genera correctamente el modelo FPGA, el GUI lo notificará al usuario y finalmente podrá realizar predicciones con la FPGA. El GUI muestra información de los modelos de red neuronal generados así como las capas que existen en el modelo seleccionado.

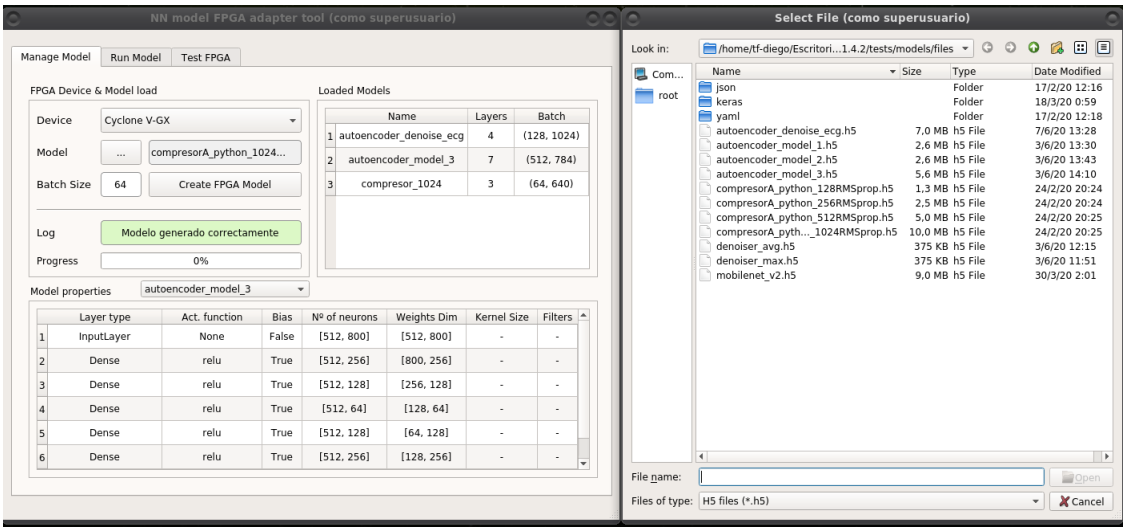


Figura 33: Pantalla principal del GUI

5.3 Simulación de modelos de NN en FPGA

5.3.1 Encoders mediante Keras y FPGA

Un autoencoder es un tipo de red neuronal con la capacidad de comprimir información mediante aprendizaje no supervisado. Aplicaciones típicas de los autoencoders son la codificación de datos para reducir la dimensionalidad o la reducción de ruido. Los autoencoders se usan para resolver problemas aplicados como podrían ser el reconocimiento facial o la adquisición del significado semántico. Otras aplicaciones interesantes se hallan en el campo de la biomedicina donde son utilizadas para la eliminación de ruido en

[12] Graphic User Interface

señales de baja intensidad como podría en los *ECG's*[13]

En la siguiente figura se muestra la estructura de un autoencoder en donde se percibe la simetría de este tipo de NN's.

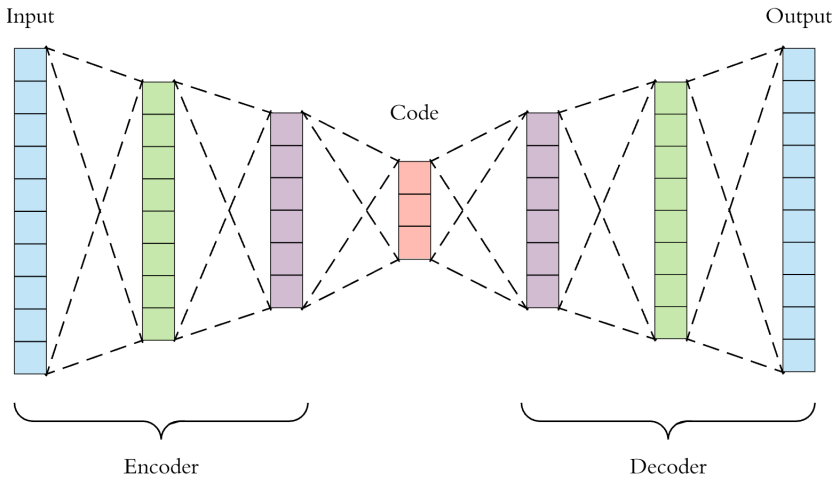


Figura 34: NN autoencoder: apréciese su estructura simétrica

Para la demostración de que lo expuesto en el flujograma anterior es funcional y se cumple se van a migrar 4 modelos de NN's a la FPGA. Estos modelos han sido extraídos del blog de Keras[10] y uno de ellos ha sido creado por nosotros mismos. Los tres primeros modelos se tratan de *Deep Autoencoders* utilizados para la codificación (reducción de dimensionalidad) y de-codificación de imágenes. Las muestras a utilizar serán las imágenes con números proporcionadas por el data set de *MNIST*

A continuación se exponen las tablas con las propiedades de cada autoencoder:

[13]ECG: El electrocardiograma es una prueba que registra la actividad eléctrica del corazón que se produce en cada latido cardiaco. Esta actividad eléctrica se registra desde la superficie corporal del paciente y se dibuja en un papel mediante una representación gráfica o trazado, donde se observan diferentes ondas que representan los estímulos eléctricos de las aurículas y los ventrículos

Autoencoder 1: Nº de capas=6, Nº de inputs=784				
Nº de capa	Tipo	Función de activación	Bias	Units
1	Input	-	-	-
2	Dense	Relu	Si	128
3	Dense	Relu	Si	64
4	Dense	Relu	Si	32
5	Dense	Relu	Si	64
6	Dense	Relu	Si	128
7	Dense	Sigmoide	Si	784
8	Output	-	-	-

Tabla 27: Características de autoencoder 1

Autoencoder 2: Nº de capas=6, Nº de inputs=784				
Nº de capa	Tipo	Función de activación	Bias	Units
1	Input	-	-	-
2	Dense	Relu	No	128
3	Dense	Relu	No	64
4	Dense	Relu	No	32
5	Dense	Relu	No	64
6	Dense	Relu	No	128
7	Dense	Sigmoide	No	784
8	Output	-	-	-

Tabla 28: Características de autoencoder 2

Autoencoder 3: Nº de capas=6, Nº de inputs=784				
Nº de capa	Tipo	Función de activación	Bias	Units
1	Input	-	-	-
2	Dense	Relu	Si	256
3	Dense	Relu	Si	128
4	Dense	Relu	Si	64
5	Dense	Relu	Si	128
6	Dense	Relu	Si	256
7	Dense	Tanh	Si	784
8	Output	-	-	-

Tabla 29: Características de autoencoder 3

Autoencoder experimental: Nº de capas=2, Nº de inputs=640				
Nº de capa	Tipo	Función de activación	Bias	Units
1	Input	-	-	-
2	Dense	Linear	Si	512
3	Dense	Linear	Si	640
8	Output	-	-	-

Tabla 30: Características de autoencoder experimental

Para cada uno de los modelos expuestos se han realizado 100 iteraciones con un batch de 32 imágenes. Los tres primeros autoencoders corresponden a estructuras similares, donde lo que cambia es la función de activación de sigmoide a tangente hiperbólica entre el autoencoder 1 y 3 así como el número de *units* de las capas ocultas. Los autoencoders 1 y 2 son similares cambiando que el primero si que tiene bias y el segundo no. Estos cambios se han realizado para demostrar el correcto funcionamiento del kernel de la FPGA a la hora de tener en cuenta el cambio de características entre diferentes modelos.

La siguiente tabla ilustra el error medio cuadrático producido entre los modelos de Keras y la FPGA:

Modelo	MSE
Autoencoder 1	2.1308e-14
Autoencoder 2	2.1264e-14
Autoencoder 3	9.7707e-14
Autoencoder experimental	1.1846e-13

Tabla 31: MSE producidos entre los modelos de Keras y FPGA

El tiempo de ejecución de todos los modelos en la FPGA y Keras para 100 iteraciones de cada modelo se expone en esta tabla. Se han ejecutado simultáneamente los 4 modelos y obtenido el tiempo medio de ejecución. También se exponen los mejores y peores tiempos de las 100 iteraciones:

Tiempo (s)	Keras	FPGA	Speed Up
Total	0.9865	1.3742	0.7179
Medio	0.0099	0.0137	0.7226
Min	0.0053	0.0130	0.4077
Max	0.4051	0.4150	0.98

Tabla 32: Tiempos de ejecución para 100 iteraciones de los modelos de Keras y FPGA

Los errores medios cuadráticos obtenidos confirman que el funcionamiento para redes neuronales de tipo denso, es decir, que no albergan convoluciones, es correcto puesto

que se obtiene un MSE con una magnitud muy pequeña: $MSE \approx \times 10^{-14}$. Dadas las variaciones en la estructura de NN entre los diferentes autoencoders se puede afirmar también que tanto el kernel como la librería de python elaborada para este TFG tiene en cuenta de forma correcta las funciones de activación clásicas así como la adición del bías. Otro aspecto importante a remarcar es la adaptabilidad de los modelos de NN's en FGPA a un número de muestras de entrada dado, no existiendo límites (exceptuando lo relacionado al tiempo de computación como es obvio) en el número de muestras a procesar por la FPGA. Esta cualidad va íntimamente ligada a la estructura de producto matricial y los sistemas SGEMM, por lo que el kernel presentado presenta buenas propiedades para seguir desarrollándose a fin de optimizarlo mas.

Por otro lado, los tiempos de computación entre el modelo de Keras y la FPGA demuestran que, al menos, en este modelo de FPGA las prestaciones obtenidas han sido peores. Esto no quiere decir que no se puedan superar las marcas de Tensorflow/Keras, si no que el dispositivo con el que se ha realizado el TFG no es tan potente como se desearía. Como se ha remarcado en el capítulo 5: *Análisis tiempos de ejecución*, con una FPGA que tenga mas recursos para subir los parámetros BS y SWI, los tiempos de ejecución mejorarían notablemente, como sería el caso si se utilizara el algoritmo propuesto por Rafael Gadea con un BS=128 y un SWI=16.

Estas imágenes muestran las salidas producidas por el autoencoder 1 y 3 tanto por el modelo de Keras como el de la FPGA.

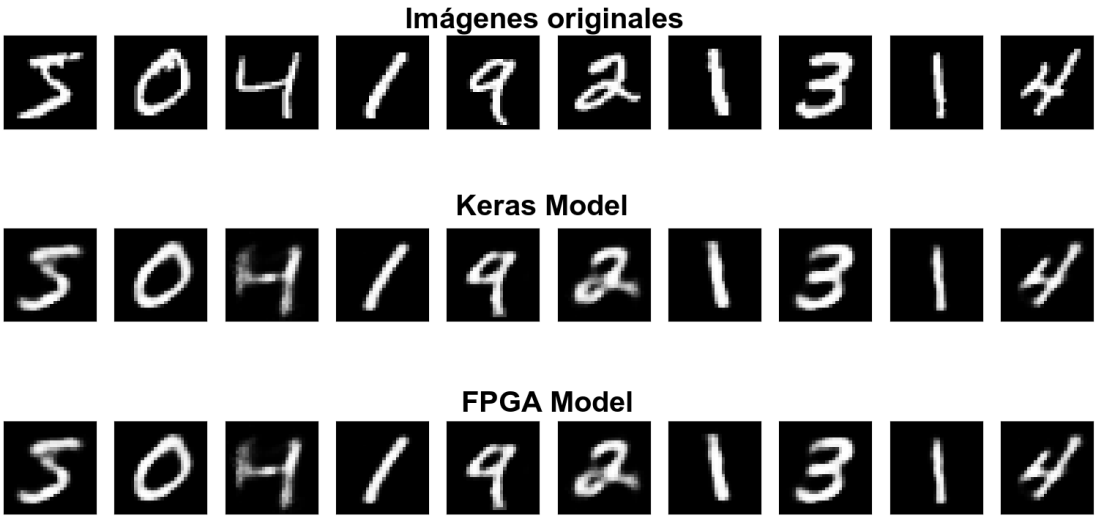


Figura 35: BS=16. Rojo: Keras, Verde: Numpy, Azul: FPGA

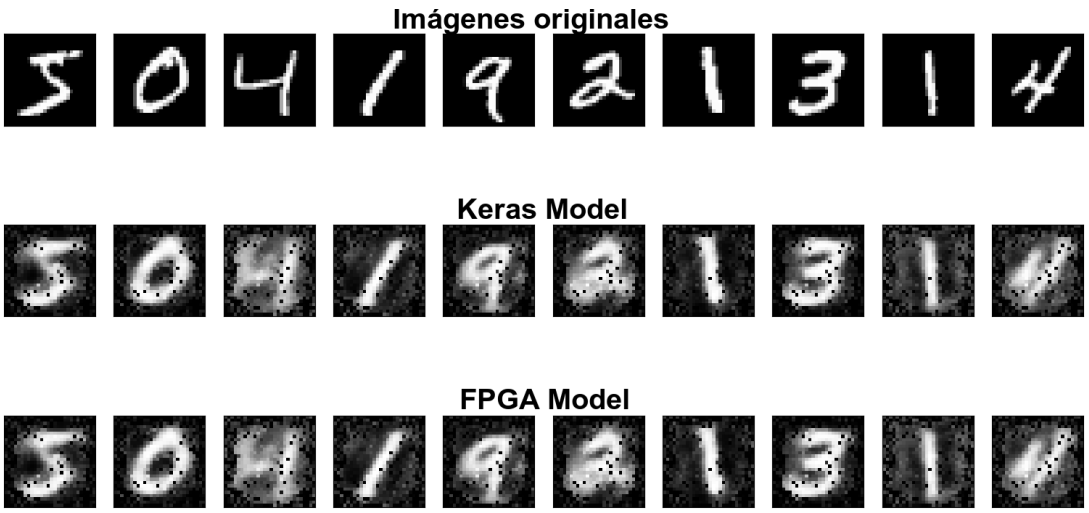


Figura 36: BS=16. Rojo: Keras, Verde: Numpy, Azul: FPGA

5.3.2 Eliminación de ruido en ECG mediante autoencoders

Los autoencoders son capaces de eliminar el ruido de una señal, lo que al final no deja de ser una reconstrucción de la misma. En señales eléctricas de baja intensidad como los ECG es bastante común que haya ruido. Este ruido puede ser fácilmente eliminado si se trata de ruido en una determinada componente frecuencial. Bastaría con realizar una FFT a la señal y determinar que armónicos no constituyen parte de la señal original. Este proceso, además de costoso puesto que sería necesario primeramente realizar una FFT y después filtrarla con la información obtenida, no sería válido si se tratara de ruido blanco gausiano. Haciendo mención a la definición de ruido blanco gausiano:

El ruido blanco es una señal aleatoria, caracterizada porque sus valores en instantes de tiempo distintos no tienen relación alguna entre sí, es decir, no existe correlación estadística entre sus valores. El ruido blanco Gaussiano será aquel cuya función de densidad responde a una distribución normal. Gaussiano se refiere a la distribución de voltaje de la fuente de ruido. Blanco es la fuente de ruido de potencia de densidad espectral, que es idealmente plano con la frecuencia. En realidad, en algún punto –debido al desfase- hay una reducción en el nivel de ruido medible.

Para obtener un dataset con el que entrenar la red neuronal se ha hecho uso del paquete *neurokit2* [11] disponible en python a través de su instalador *pip*. Se han generado 70.000 señales sintéticas de ECG y se les ha añadido tanto ruido gaussiano blanco como variaciones en la frecuencia cardíaca a través de sendas distribuciones de probabilidad normales.

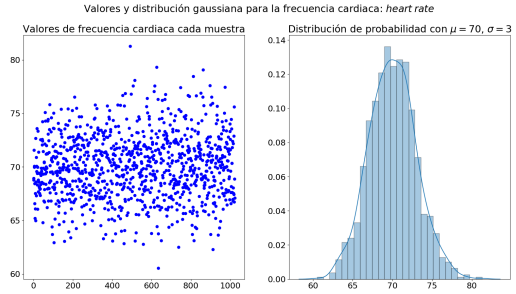


Figura 37: Frecuencia cardiaca ECG

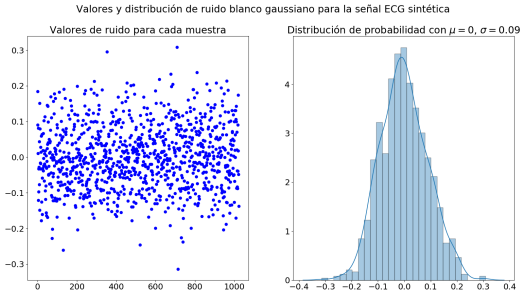


Figura 38: Ruido ECG

60.000 señales se han utilizado para el entrenamiento y 10.000 para test. Como función de activación se ha hecho uso de la tangente hiperbólica puesto que la señal de ECG tiene tanto valores positivos como negativos. Las señales han sido previamente normalizadas antes de introducirlas en el data set, quedando estas acotadas entre -1 y 1. La estructura de la NN creada ha sido:

Autoencoder 3: Nº de capas=6, Nº de inputs=1024				
Nº de capa	Tipo	Función de activación	Bias	Units
1	Input	-	-	-
2	Dense	Tanh	Si	512
3	Dense	Tanh	Si	256
4	Dense	Tanh	Si	128
5	Dense	Tanh	Si	256
6	Dense	Tanh	Si	512
7	Dense	Tanh	Si	1024
8	Output	-	-	-

Tabla 33: Características de autoencoder para eliminación de ruido en ECG's

Se ha obtenido un $MSE \approx \times 10^{-3}$ para las predicciones mediante el modelo de Keras. El batch size utilizado ha sido de 128. La generación del modelo en la FGPA ha sido también satisfactorio, como se puede apreciar en la gráfica y tabla siguientes. Se ha programado un test para comparar el modelo de Keras y FPGA con 15 iteraciones (predicciones) de 128 señales ECG con ruido extraídas de forma aleatoria del dataset indicado anteriormente.

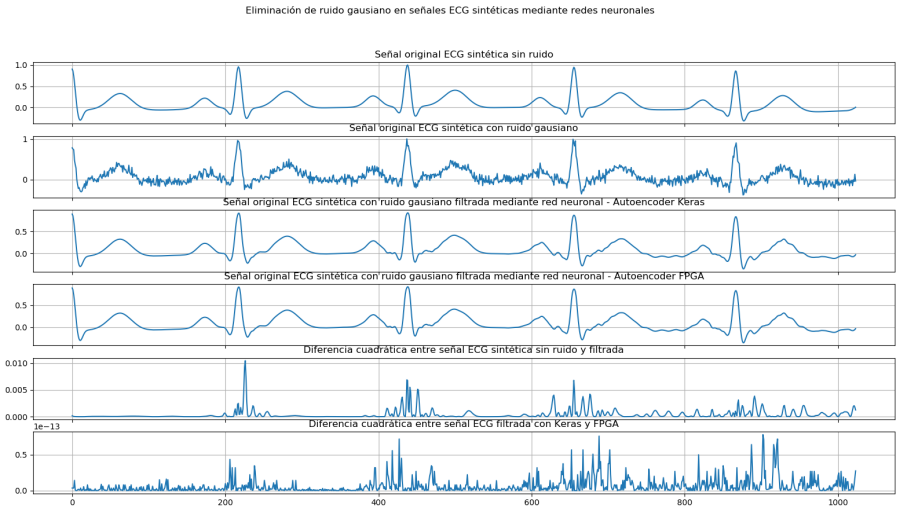


Figura 39: ECG filtrado mediante autoencoder con Keras y FPGA

MSE	T FPGA (s)	T Keras (s)	Speed Up
7.11e-15	0.0188	0.0112	0,59
6.91e-15	0.0177	0.0105	0,59
7.02e-15	0.0187	0.0159	0,85
6.97e-15	0.0176	0.0097	0,55
6.92e-15	0.0178	0.0111	0,62
7.06e-15	0.0175	0.0098	0,56
6.77e-15	0.0178	0.0111	0,62
6.83e-15	0.0178	0.0106	0,59
6.99e-15	0.0177	0.0101	0,57
6.82e-15	0.0177	0.0121	0,68
7.20e-15	0.0179	0.0096	0,53
6.89e-15	0.0177	0.0100	0,56
6.93e-15	0.0178	0.0117	0,65
7.06e-15	0.0176	0.0105	0,59
6.82e-15	0.0175	0.0107	0,61

Tabla 34: MSE producidos entre los modelos de Keras y FPGA y tiempos de computo

A la vista de los resultados se puede concluir que la eliminación de ruido mediante el autoencoder es satisfactoria. El comportamiento de la NN en la FPGA es similar al de Keras visto el MSE obtenido, $MSE \approx \times 10^{-15}$. Y de nuevo se aprecia que el tiempo de cómputo para Keras es menor, siendo en esta simulación un factor próximo a 0.6, menor que el de la anterior simulación que estaba en torno a 0.75, 0.77. Este empeoramiento en el tiempo de computo es debido al batch, donde se tiene un batch mayor, de 128, en vez de 32 como en la simulación de los modelos anteriores.

Otro ejemplo de eliminación de ruido en señales, en este caso la mezcla de tres armónicos distintos comprendidos entre 20 y 100 Hz. se puede apreciar en la siguiente gráfica. El propósito de está gráfica no es mas que el de mostrar las posibilidades que ofrecen este tipo de redes neuronales para el tratamiento de señal.

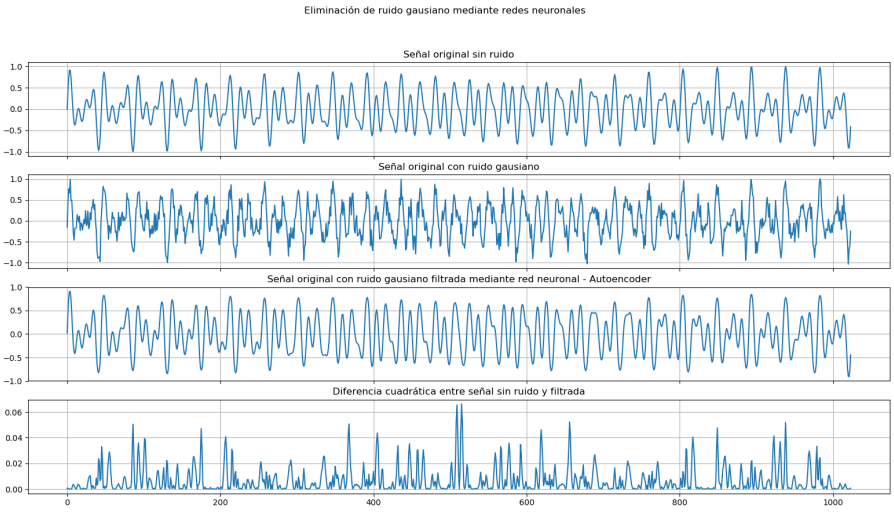


Figura 40: Señal compuesta filtrada mediante autoencoder

Con estos ejemplos se concluiría el correcto funcionamiento de los modelos de red neuronal con capas densas en la FPGA. En el capítulo siguiente se expondrán las conclusiones de aquello que puede ser susceptible de ser mejorado o la exploración de nuevas posibilidades desde el aspecto físico de la implementación de sistemas de red neuronal embebidos en FPGA.

Capítulo 6. Conclusiones y futuro trabajo

6.1 Conclusiones

A lo largo del presente TFG se han estudiado tanto los algoritmos que componen algunas de las capas de las librerías de Keras y Tensorflow, como la implementación de los mismos en la FPGA, su optimización y la migración de modelos de redes neuronales para embeberlos en la FPGA. Las competencias adquiridas han sido muchas y variadas: desde el aprendizaje y conocimiento de nuevos algoritmos relacionados con el tratamiento de imágenes, pasando por el descubrimiento de un nuevo lenguaje de programación, OpenCL, totalmente desconocido hasta el momento para mi, algo que dota de valor e interés a todo el trabajo realizado; o el refuerzo del uso de librerías de inteligencia artificial para la generación de modelos a través de metodologías ágiles. La convergencia de lenguaje a bajo nivel junto con el manejo de librerías a alto nivel se complementan para conseguir una mayor optimización de los distintos algoritmos involucrados en librerías de inteligencia artificial o de cálculo científico, como es el caso de la librería Numpy, implementada en su mayoría en C/C++.

En un principio el TFG iba a constar del análisis y estudio de un algoritmo óptimo para el producto matricial, a medida que el trabajó avanzaba se investigo sobre los SGEMM, por lo que se decidió abordar también las redes neuronales convolucionales desde una perspectiva matricial, aprovechando al máximo el algoritmo de Block Size. La falta de documentación y la carga matemática ha sido una complicación para la implementación de algoritmos y métodos de las librerías de Tensorflow y Keras, excepto para las funciones de activación, las cuales son conocidas. Solo remarcar en este punto la mala implementación que hace Intel de la función tangente hiperbólica, como se ha demostrado en el capítulo 3. Aun así, la implementación de la tangente se ha optimizado al máximo mediante sus relaciones matemáticas con las funciones trigonométricas hiperbólicas y la función exponencial; función que si se encuentra bien optimizada en las librerías de Intel-OpenCL, pasando de un 144 % de utilización lógica a menos del 90 %.

El uso de python y sus librerías para hardware dedicado como pyopencl han sido de gran ayuda para el testeo y la implementación de los algoritmos necesarios, evitando así tener que testear mediante C++, que como es sabido, requiere de un mayor tiempo. Esto además permite integrar con OpenCL toda la funcionalidad que pueda ofrecer python como lenguaje target del host. Por el momento no parece que los dispostivos FPGA estén muy integrados en los sistemas de inteligencia artificial o puedan ser manejados a alto nivel. Una de las soluciones actuales en el mercado es OpenVINO, que como se ha mencionado anteriormente, no admite todo tipo de redes neuronales. Es por eso que se ha decido realizar el GUI presentado en el capítulo 5 con el fin de poder realizar migraciones rápidas de los modelos de red neuronal a modelos FPGA, evitando así tener que escribir código o directamente “meterte” a programar. Esta característica tampoco se ha visto implementada aún por OpenVINO, por lo que creo que es una parte fundamental del

TFG.

Los tiempos de ejecución presentados en este trabajo han llegado a ser superiores a los ofrecidos por las librerías de Keras o Tensorflow: para dimensiones de matriz menores a 512 los tiempos de cómputo de la FPGA han sido mejores. Las tablas siguientes muestran los speed ups de la FPGA respecto a Keras sin función de activación y función de activación sigmoide. Los tiempos ponen de manifiesto como la FPGA puede llegar a ser 100 veces mas rápidas en tamaños pequeños de matriz: 32×32 , reduciendo su performance a medida que aumentan las dimensiones. Aún así, para tamaños de 256×256 sigue siendo mas óptima la FPGA, llegando a ser 6.18 veces mas rápida que Keras. Otra conclusión importante que se desprende es la poca o nula penalización que sufren los tiempos de computación al implementar las funciones de activación en el kernel. Sin funciones de activación los tiempos de cómputo permanecen casi similares, con una pequeñas desviaciones entre ambos.

Dimensiones matriz	Speed Up FPGA sin activación	Speed Up FPGA con sigmoide
32x32	129	129
64x64	77	107
128x128	28.3	28.2
256x256	4.77	6.41
512x512	0.86	0.973
1024x1024	1.15	0.799
2048x2048	0.69	0.578
4096x4096	0.55	0.739

Tabla 35: *Speed Up* para producto matricial sin función y con función de activación Sigmoide con BS=32 y SWI=2

En general los tiempos de cómputo de la convolución para un solo filtro han sido en general menores a los de Keras, llegando incluso a ser 11 veces más rápida la FPGA que Keras, como se demuestra en la tabla [20]. A modo de resumen se expone la tabla con los Speed Ups medios para distintos tamaños de imagen, números de imagen y tamaño del filtro así como la desviación típica de los tiempos medidos. La desviación típica demuestra que los Speed Ups obtenidos se mantienen estables, no existiendo grandes diferencias cuando se aumentan o disminuyen el número de batch o tamaño de los filtros:

Dimensiones imagen	Speed Up medio en FPGA	σ^2 Speed Up
32x32	7.11	2.20
64x64	2.74	0.59
128x128	1.79	0.38
256x256	0.81	0.16

Tabla 36: Speed Up medio y desviación típica de speed up para convoluciones mediante matriz de Toeplitz

El problema de estas marcas radica en que no suele ser habitual un solo filtro en las redes

neuronales convolucionales, es por eso que se decidió estudiar e implementar la técnica de Im2Row, la cual ha demostrado ser mucho mas optima que la matriz de Toeplitz, aunque no llegando a superar los tiempos de Keras debido al batch utilizado, mayor que 1. Esta técnica también permite tener en cuenta mas de un canal, por lo que es un candidato óptimo para implementar en una SGEMM.

La emulación de modelos por la FPGA también ha sido satisfactoria, obteniendo con las comparativas realizadas con Keras un error medio cuadrático en torno a $MSE \approx 10^{-10}$, por lo que se puede concluir que tanto el GUI como todos los algoritmos realizados funcionan correctamente. En este punto si que cabe destacar que la FPGA no ha sido tan rápida como Keras, pero se recuerda que la frecuencia de reloj del dispositivo FPGA es menor a la del PC. En general, los tiempos de la FPGA realizando predicciones se han situado en torno a 0.7, 0.75 mas lentos que los tiempos de Keras.

6.2 Futuro trabajo

Son varios los aspectos que podrían englobar futuros trabajos con el expuesto hasta ahora. Por un lado, después de validar el correcto funcionamiento de algunos tipos de NN's en la FPGA, sería de interés el estudio de la creación de PCB's que albergaran la FPGA con las NN's embebidas, último paso necesario para generar un sistema electrónico real de IA para ser usado en la industria, como por ejemplo la biomédia o la de la automoción. La clase de python que hace de host es capaz de tener diferentes modelos de red neuronal pre-cargados, pudiendo la FPGA hacer uso de cualquiera de estos de manera inmediata, por lo que dotaría de dinamismo al sistema embebido, siendo válido el mismo kernel OpenCL para cualquier modelo de NN creado en el host. No sería necesario tener un host potente para llevar a cabo esta tarea puesto que lo único que debería hacer el host sería almacenar los modelos de red neuronal, siendo importante la capacidad de almacenamiento del host y no la de cómputo. El GUI presentado ayudaría a agilizar la migración e implementación de los modelos NN en el dispositivo, permitiendo a gente sin conocimientos previos en IA o FPGA, embeber el modelo para su inmediato uso.

Por último, en el ámbito de la ampliación de la funcionalidad del kernel, podría ser de interés la implementación de otras capas como las convoluciones en 1D o 3D. Estas no serían mas que una extensión de la ya generada para dos dimensiones. Con todos estos puntos expuestos, se puede confirmar que sería posible emular cualquier sistema de NN a través de una SGEMM en la FPGA.

Por último, agradecer el apoyo y la predisposición recibida por mi tutor, Rafael Gadea Gironés, así como a mi madre y amigos, teniendo en cuenta además los tiempos extraños y difíciles que acontecen. Si algo ha puesto de manifiesto esta pandemia es la importancia de las telecomunicaciones para poder gestionar la comunicación en estas condiciones así como los sistemas de inteligencia artificial han ayudado a prevenir y modelar escenarios de utilidad en la lucha contra el COVID-19[12].

Referencias

[1] Wikipedia: OpenCL,
`https://es.wikipedia.org/wiki/OpenCL`, Febrero 2020.

[2] Intel FPGA: Matrix Multiplication Design Example,
`https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/matrix-multiplication.html`.

[3] Intel FPGA *Intel® FPGA SDK for OpenCL™ Pro Edition*. (Inglés) 7.2. Kernel Vectorization, (pp. 136-139).

[4] Intel FPGA *Intel® FPGA SDK for OpenCL™ Pro Edition*. (Inglés) 3.3. Local Memory, (pp. 52).

[5] Intel FPGA *Intel® FPGA SDK for OpenCL™ Pro Edition*. (Inglés) 7.2. Kernel Vectorization, (pp. 136-139).

[6] Chigozie Enyinna Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall *Activation Functions: Comparison of Trends in Practice and Research for Deep Learning* . (Inglés), *arxiv*, 2018. [online]
`https://arxiv.org/pdf/1811.03378.pdf`

[7] Marat Dukhan *The indirect convolution algorithm*. (Inglés), *arxiv*, 2019. [online]
`https://arxiv.org/pdf/1907.02129.pdf`

[8] Rathinakumar Appuswamy, Tapan K. Nayak, John Arthur, Steven Esser, Paul A. Merolla, Jeffrey L. Mckinstry, Timothy Melano, Myron Flickner, Dharmendra S. Modha *Structured convolution matrices for energy-efficient deep learning*. (Inglés), *arxiv*, 2016. [online] 3. Symmetrical kernels, (pp. 3-5)
`https://arxiv.org/pdf/1606.02407.pdf`

[9] Lingchuan Meng, John Brothers, *Efficient Winograd convolution via integer arithmetic*. (Inglés), *arxiv*, 2019. [online]
`https://arxiv.org/pdf/1901.01965.pdf`

[10] Building Autoencoders in Keras,
`https://blog.keras.io/building-autoencoders-in-keras.html`

[11] GitHub: The Python Toolbox for Neurophysiological Signal Processing,
`https://github.com/neuropsychology/NeuroKit`

[12] Naukas: Inteligencia Artificial y Covid-19 como detonante del cambio,
`https://naukas.com/2020/06/03/inteligencia-artificial-y-covid-19-como-detonante-del-cambio/`, 3 de junio de 2020

Anexo I: Códigos Python para creación de modelos

```
1
2  # Librearías necesarias para manejar tensorflow y keras
3  from __future__ import absolute_import, division, print_function,
       unicode_literals
4
5  import tensorflow as tf
6  from tensorflow import keras
7  import numpy as np
8
9  # Just disables the warning, doesn't enable AVX/FMA
10 import os
11 import numpy as np
12 os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
13 os.environ['PYOPENCL_COMPILER_OUTPUT'] = '0'
14
15 # Librerías necesarias para manejar pyopencl y el backend con la FPGA
16 from layers import Conv2D, UpSampling2D
17 import pyopencl as cl
18 mf = cl.mem_flags
19 import model_globals
20 import memalign
21 import tensorflow_fpga_v2 as tf_fpga
22
23 tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
24
25 class FpgaModel():
26     '''
27     Este método controla el flujo principal para llevar a cabo con
28     éxito la creación de las variables globales y la instanciación del
29     objeto de modelo de la FPGA que se utilizará para realizar predic-
30     ciones con la FPGA. El flujo que sigue el método es:
31         1 - Trae todas las variables de interés almacenadas en el fichero
        de
32             modelo de TensorFlow o Keras.
33                 model.layers
34                 model.get_config()
35
36         2 - Crea el kernel de pyopencl para iniciar la conexión con la
       fpga
37             desde python. El kernel creado es almacenado para
       instanciarlo
38             una sola vez al principio cuando se instancia el modelo,
       reaprove-
39             chándose en todas las iteraciones que el modelo sea ejecutado
       . Se
40             realiza a través del método:
41                 self.create_kernel()
42
43         3 - Hace uso de <self.setup_layers()> para generar diccionario
       con
44             modelo FPGA y variables globales.
```

```
    '''
    def __init__(self, model_file, batch):
        self.model_file = model_file
        self._BS = 32
        self.batch = batch
        # Recreate the exact same model purely from the file
        tf.keras.backend.clear_session()  # For easy reset of notebook state.
        self._model = keras.models.load_model(self.model_file, compile=False)
        self._layers = self._model.layers
        self._config = self._model.get_config()
        self.model_name = self._config["name"]
        self.name = self.model_name
        self._fpga_model = {self.model_name : {"name": self.model_name, "batch": batch}}
        self.succes = False
        # Si exise el modelo
        if self.model_name in model_globals.CREATED_MODELS:
            return
        else:
            print('Generando workflow para el modelo: ' + self.model_name)

        # Se controla si ya se ha creado el kernel
        if model_globals.CREATE_SESSION:
            pass
        else:
            print('Creando kernel pyopencl : ', end='')
            if self.__create_kernel(): # Se crea kernel pyopencl
                print("[OK]")

        # Se controla si ya se ha generado el modelo fpga
        print('Creando buffers         : ', end='')
        if self.__setup_layers():  # Se crea la configuración para la FPGA
            print("[OK]")
            model_globals.MODELS.update(self._fpga_model)

        # Se controla si se ha generado la lógica necesaria
        print('Creando lógica          : ', end='')
        if self.__create_NN_logic():  # Se crea la configuración para la FPGA
            print("[OK]\n")


        # Se controla si ya se ha validado el modelo
        # print(' ---- Validando modelo en FPGA: ', end='')
        # print("[OK]")          # Se valida el modelo en la FPGA

    def __setup_layers(self):
        '''
        Este método se utiliza para transformar el modelo almacenado
```

```
92          en un fichero por keras o tensorflow, ya sea .h5, .json o
93          .yaml. los pasos realizados para setear el modelo en pyopencl
94          y el algoritmo de Intel son:
95              1 - Coge todas las propiedades de interés de las capas a trav
    és de
96                  layer = self._layers[i].get_config()
97
98              2 - Infiere las nuevas dimensiones óptimas para las matrices
99
100             3 - padea las matrices al entero superior mas proximo que sea
     múltiplo
101                 de 32
102
103             4 - Setea el tipo de dato a np.float32
104
105             5 - Crea los buffers de los pesos, bias y neuronas para cada
    capa
106
107             6 - Todas las variables anteriores las registra en un
    diccionario
108                 que se aloja en el fichero de variables globales
    manage_model_globals
109                 para hacer uso de estas de forma inmediata cuando se
    corre el modelo
110         '''
111         try:
112             for i in range(0, len(self._layers)):
113                 lyr_aux = {}
114                 weights = None
115                 layer = self._layers[i].get_config()
116                 # <layer_name> se utilizará como ID para referenciar a la
     capa dentro del modelo
117                 lyr_aux["layer_name"] = layer["name"]
118                 class_name = self._config["layers"][i]["class_name"]
119                 lyr_aux["class_name"] = class_name
120
121                 # Para las capas de tipo Input
122                 if class_name == "InputLayer":
123                     batch_input_shape = layer["batch_input_shape"]
124                     # batch_input_shape = (None, 32, 32, 1)
125                     lyr_aux["activation"] = None
126                     lyr_aux["use_bias"] = False
127                     if len(batch_input_shape) == 2:
128                         self.dims = (self.batch, batch_input_shape[1])
129                         new_shape = self.__multiples()
130                         dif = tuple(np.subtract(new_shape, (self.batch,
    layer["batch_input_shape"][1])))
131                         self._fpga_model[self.model_name]["
    batch_input_shape"] = (self.batch, batch_input_shape[1])
132                     if len(batch_input_shape) == 4:
133                         dim = batch_input_shape[1] * batch_input_shape[2]
     * batch_input_shape[3]
134                         self.dims = (self.batch, dim)
```

```
135                         new_shape = self.__multiples()
136                         dif = tuple(np.subtract(new_shape, (self.batch,
    dim)))
137                         self._fpga_model[self.model_name]["
    batch_input_shape"] = (self.batch, batch_input_shape[1],
    batch_input_shape[2])
138                     self.batch = new_shape[0]
139                     lyr_aux['dif'] = dif
140                     lyr_aux["kernel_size"] = " - "
141                     lyr_aux["filters"] = " - "
142 
143                 # Para las capas de tipo Convolucional 2D
144                 elif class_name == "Conv2D":
145                     new_shape = self._fpga_model[self.model_name][i-1]['
    new_shape']
146                     dif = self._fpga_model[self.model_name][i-1]['dif']
147                     lyr_aux['dif'] = dif
148                     weights = self._layers[i].get_weights()
149                     filters = weights[0][:, :, 0, :]
150                     bias    = weights[1]
151                     lyr_aux["activation"] = layer["activation"]
152                     lyr_aux["data_format"] = layer["data_format"]
153                     lyr_aux["filters"] = layer["filters"]
154                     lyr_aux["dilation_rate"] = layer["dilation_rate"]
155                     lyr_aux["kernel_size"] = layer["kernel_size"]
156                     lyr_aux["strides"] = layer["strides"]
157                     lyr_aux["padding"] = layer["padding"]
158                     lyr_aux["use_bias"]   = layer["use_bias"]
159                     lyr_aux["weights"] =  weights[0][: , :, 0, :] # Se
    cogen los valores de los filtros
160                     shape = self._fpga_model[self.model_name]['
    batch_input_shape'][1:3]
161                     H = Conv2D.Toeplitz_Conv2D(shape, filters, new_shape
    [0], lyr_aux["strides"], lyr_aux["padding"])
162                     paded_weights = memalign.pad_dims(H, dif)
163                     paded_weights = memalign.aligned_data(paded_weights,
    paded_weights.shape)
164                     '''
165                     if lyr_aux["use_bias"]:
166                         paded_bias = memalign.pad_vect(new_shape[1], bias
    , dif[1])
167                         lyr_aux["bias"] = paded_bias
168                         lyr_aux["buff_bias"] = self.__create_buff(
    paded_bias)
169                     else:
170                         lyr_aux["buff_bias"] = None
171                     '''
172 
173                 elif "Pooling2D" in class_name:
174                     lyr_aux["buff_bias"] = None
175                     lyr_aux["use_bias"]   = False
176                     lyr_aux["activation"] = "linear"
177 
```

```
178                     shape = self._fpga_model[self.model_name]['
    batch_input_shape'][1:3]
179                     new_shape = self._fpga_model[self.model_name][i-1]['
    new_shape']
180                     dif = self._fpga_model[self.model_name][i-1]['dif']
181                     lyr_aux['dif'] = dif
182                     lyr_aux["data_format"] = layer["data_format"]
183                     lyr_aux["strides"] = layer["strides"]
184                     lyr_aux["padding"] = layer["padding"]
185                     lyr_aux["pool_size"]   = layer["pool_size"]
186                     H = Conv2D.Toeplitz_Pooling2D(class_name, shape,
    pool_size=lyr_aux["pool_size"],
187                                                   strides=lyr_aux["
    strides"][0], padding=lyr_aux["padding"])
188                     paded_weights = memalign.pad_dims(H, dif)
189                     paded_weights = memalign.aligned_data(paded_weights,
    paded_weights.shape)
190                     lyr_aux["weights"]   = paded_weights
191                     lyr_aux["buff_weights"] = self.__create_buff(
    paded_weights)
192
193                 elif class_name == "UpSampling2D":
194                     lyr_aux["buff_bias"] = None
195                     lyr_aux["use_bias"]   = False
196                     lyr_aux["activation"] = "linear"
197                     shape = self._fpga_model[self.model_name]['
    batch_input_shape'][1:3]
198                     new_shape = self._fpga_model[self.model_name][i-1]['
    new_shape']
199                     dif = self._fpga_model[self.model_name][i-1]['dif']
200                     lyr_aux['dif'] = dif
201                     lyr_aux["data_format"] = layer["data_format"]
202                     lyr_aux["size"]   = layer["size"]
203
204                     H = UpSampling2D.Toeplitz_UpSampling2D(shape, layer["
    size"])
205                     paded_weights = memalign.pad_dims(H, dif)
206                     paded_weights = memalign.aligned_data(paded_weights,
    paded_weights.shape)
207                     lyr_aux["weights"]   = paded_weights
208                     lyr_aux["buff_weights"] = self.__create_buff(
    paded_weights)
209
210                 # Para las capas de tipo Dense
211                 elif class_name == "Dense":
212                     lyr_aux["kernel_size"] = " - "
213                     lyr_aux["filters"] = " - "
214
215                     lyr_aux["units"] = layer["units"]
216                     lyr_aux["activation"] = layer["activation"]
217                     lyr_aux["use_bias"]   = layer["use_bias"]
218                     weights = self._layers[i].get_weights()
219                     self.dims = weights[0].shape
```

```
220                     new_shape = self.__multiples()
221                     dif = tuple(np.subtract(new_shape, weights[0].shape))
222                     paded_weights = memalign.pad_dims(weights[0], dif)
223                     paded_weights = memalign.aligned_data(paded_weights,
    paded_weights.shape)
224                     lyr_aux["weights"] = paded_weights
225                     lyr_aux["buff_weights"] = self.__create_buff(
    paded_weights)
226 
227                     if lyr_aux["use_bias"]:
228                         paded_bias = memalign.pad_vect(new_shape[1],
    weights[1], dif[1])
229                         lyr_aux["bias"] = paded_bias
230                         lyr_aux["buff_bias"] = self.__create_buff(
    paded_bias)
231                     else:
232                         lyr_aux["buff_bias"] = None
233 
234                 # Buffers correspondientes a las entradas y salidas de la
     capa
235                 # batch = self._fpga_model[self.model_name][0]["batch"]
236                 lyr_aux["buff_layer"]  = self.__create_empty_buff([self.
    batch,new_shape[1]])
237                 lyr_aux["buff_inputs"] = self.__create_empty_buff([self.
    batch,new_shape[1]])
238                 lyr_aux["buff_output"] = memalign.aligned_zeros([self.
    batch,new_shape[1]]).astype(np.float32)
239                 lyr_aux["new_shape"] = list(new_shape)
240                 lyr_aux["units"] = [self.batch, new_shape[1]]
241                 self._fpga_model[self.model_name][i] = lyr_aux
242 
243             # Se crea un buffer None para el bías para aquellas capas que
     no utilicen bias
244             memalign.create_bias_none()
245             # Se almacena el modelo creado con todas sus variables
246             self._fpga_model[self.model_name]["layers"] = i + 1
247             model_globals.MODELS[self.model_name] = self._fpga_model[self
    .model_name]
248             model_globals.CREATED_MODELS.append(self.model_name)
249             model_globals.FILES_MODELS.append(self.model_file)
250             return True
251         except Exception as e:
252             raise ValueError(e)
253 
254     '''
255     Método utilizado para crear el kernel de pyopencl que permite la
    conexión
256     entre la FPGA y python
257     '''
258     def __create_kernel(self):
259         try:
260             model_globals.PLATFORMS  = None
261             model_globals.DEVICES    = None
```

```
262             model_globals.CTX         = None
263             model_globals.BINARY      = None
264             model_globals.PRG         = None
265             model_globals.KERNEL_PAD  = None
266             model_globals.KERNEL_MLT  = None
267 
268             model_globals.PLATFORMS = cl.get_platforms()
269             model_globals.DEVICES   = model_globals.PLATFORMS[0].
    get_devices()
270             model_globals.CTX       = cl.Context(model_globals.DEVICES)
271             model_globals.BINARY    = open('../cl_kernel/matrix_mult_TF.
    aocx', 'rb').read()
272             model_globals.PRG       = cl.Program(model_globals.CTX,
273                                                  model_globals.DEVICES,
274                                                  [model_globals.BINARY]).
    build()
275             model_globals.KERNEL = model_globals.PRG.all_kernels()[0]
276             # model_globals.KERNEL_PAD = model_globals.PRG.all_kernels()
    [1]
277 
278             model_globals.QUEUE     = cl.CommandQueue(model_globals.CTX,
279                                                       properties=cl.
    command_queue_properties.PROFILING_ENABLE)
280 
281             # Kernel para producto matricial con función de activación y
    Bias
282             model_globals.KERNEL.set_scalar_arg_dtypes([None, None, None,
     None,
283                                                         np.uint32,  np.
    uint32,
284                                                         np.uint32,  np.
    uint32,
285                                                         np.float32])
286 
287             model_globals.CREATE_SESSION = True
288             return True
289         except Exception as e:
290             raise ValueError(e)
291 
292     '''
293     Método utilizado para hallar las nuevas dimensiones mútiplo de 32.
    Devuelve una tupla de
294     2 dimensiones si se trata de una matriz de pesos o una tupla de 1D si
     se trata de un vector
295     de bias o de capa de red neuronal
296     '''
297     def __multiples(self):
298         padded_dims = ()
299         for dim in self.dims:
300             if dim == None:
301                 padded_dims += (32, )
302                 continue
303             if (dim % self._BS) != 0:
```

```
304                 padded_dims += (next(x[1] for x in enumerate(
    model_globals.multiples) if x[1] > dim), )
305             else:
306                 padded_dims += (dim, )
307         return padded_dims
308 
309     def __create_buff(self, weights):
310         '''
311         Método utilizado para pre-crear los buffers de pyopencl
312         '''
313         return cl.Buffer(model_globals.CTX, mf.READ_ONLY | mf.
    COPY_HOST_PTR, hostbuf=weights.astype(np.float32))
314 
315     def __create_empty_buff(self, dim):
316         zeros = memalign.aligned_zeros(dim).astype(np.float32)
317         return cl.Buffer(model_globals.CTX, mf.WRITE_ONLY, size=zeros.
    nbytes)
318 
319     def __create_NN_logic(self):
320         '''
321         Método utilizado para crear los objetos a instanciar que
    contienen la lógica y el
322         workflow de la red neuronal
323         '''
324         # Se crea un nuevo modelo
325         model_globals.LOGIC_MODELS[self.model_name] = {}
326         # Se coge la configuración de la red neuronal
327         logic = model_globals.MODELS[self.model_name]
328         # Se coge el índice de la ultima capa
329         # output_index = list(logic.keys())[-1]
330         output_index = logic.get("layers") - 1
331 
332         # Se crean primero las capas de entrada y de salida
333         input_logic  = self.__get_function(logic[0].get("activation"))
334         output_logic  = self.__get_function(logic[output_index].get("
    activation"))
335         # Se almacenan en el modelo
336         model_globals.LOGIC_MODELS[self.model_name][0] = input_logic
337         model_globals.LOGIC_MODELS[self.model_name][output_index] =
    output_logic
338         # Se crean todas las capas ocultas
339         for i in range(1, output_index):
340             model_globals.LOGIC_MODELS[self.model_name][i] = self.
    __get_function(logic[i]["activation"])
341         return True
342 
343     def __run(self):
344 
345         units = model_globals.MODELS[self.model_name][0]["units"][1]
346         n_layers = len(model_globals.LOGIC_MODELS[self.model_name])
347         if not isinstance(self.input, (np.ndarray)):
348             raise("La entrada debe de ser un array del tipo numpy.ndarray
    ")
```

```
        '''
        if self.input.shape[1] != units:
            error = "La entrada debe de ser un vector con dimensión " +
    str(units)
            raise(error)
        '''
        # Por si es necesario padear la entrada
        dif = model_globals.MODELS[self.model_name][0]["dif"]
        input_buff = memalign.pad_dims(self.input, dif).astype(np.float32
    )
        input_buff = self.__create_buff(input_buff)

        # Capa de entrada
        prop = model_globals.LOGIC_MODELS[self.model_name][1] \
            (
                input_buff,
                model_globals.MODELS[self.model_name][1]["buff_weights"],
                model_globals.MODELS[self.model_name][1]["buff_layer"],
                model_globals.MODELS[self.model_name][0]["units"],
                model_globals.MODELS[self.model_name][1]["new_shape"],
                model_globals.MODELS[self.model_name][1]["buff_output"],
                model_globals.MODELS[self.model_name][1]["buff_bias"],
                0
            )

        # Capas ocultas: Desde la 2a capa hasta la antepenultima
        i = 0
        for i in range(1, n_layers-2):
            prop = model_globals.LOGIC_MODELS[self.model_name][i+1] \
            (
                prop,
                model_globals.MODELS[self.model_name][i+1]["buff_weights"
    ],
                model_globals.MODELS[self.model_name][i+1]["buff_layer"],
                model_globals.MODELS[self.model_name][i]["units"],
                model_globals.MODELS[self.model_name][i+1]["new_shape"],
                model_globals.MODELS[self.model_name][i+1]["buff_output"
    ],
                model_globals.MODELS[self.model_name][i+1]["buff_bias"],
                0
            )

        # Capa de salida
        result = model_globals.LOGIC_MODELS[self.model_name][i+2] \
            (
                prop,
                model_globals.MODELS[self.model_name][i+2]["buff_weights"
    ],
                model_globals.MODELS[self.model_name][i+2]["buff_layer"],
                model_globals.MODELS[self.model_name][i+1]["units"],
                model_globals.MODELS[self.model_name][i+2]["new_shape"],
                model_globals.MODELS[self.model_name][i+2]["buff_output"
    ],
```

```
                model_globals.MODELS[self.model_name][i+2]["buff_bias"],
            )
        return result


    def __program_fpga(self):
        # Mñetodo utilizado para reprogramar la FPGA
        try:
            import os
            reprogram_cmd = 'aocl program acl0 ../cl_kernel/
    matrix_mult_TF.aocx'
            os.popen(reprogram_cmd).read()
            return True
        except Exception as e:
            raise ValueError(e)

    def __get_function(self, act_str):
        # Método utilizado para generar la lñogica de las capas de tipo
    Dense
        self._functions = {None     : tf_fpga.matmul,
                           "linear" : tf_fpga.matmul,
                           "relu"   : tf_fpga.relu,
                           "sigmoid": tf_fpga.sigmoid,
                           "tanh"   : tf_fpga.tanh,
                           "softmax": tf_fpga.softmax}
        return self._functions.get(act_str)

    def get_models():
        return model_globals.CREATED_MODELS

    def get_config(self):
        return model_globals.MODELS[self.model_name]

    def create_model(self):
        return self.succes

    def predict(self, x):
        self.input = x
        return self.__run()
```

Método 9: models_manage.py

```

'''
Listas con todos los múltiplos hasta 16384 para configurar
las dimensiones bajo el criterio del múltiplo más proximo
'''
mx = 16384 #

BS = 32
n = int(mx / 32)
multiples = [32*n for n in range(1,n+1)]

DENSE_PROPS = ["activation",
```

```
                "bias_constraint",
                "bias_initializer",
                "bias_regularizer",
                "dtype",
                "name",
                "units",
                "use_bias"]

INPUT_PROPS = ["batch_input_shape",
                "dtype",
                "name",
                "sparse"]

CUSTOM_PROPS = ["layer_type",
                 "unique_name"]

CONV_PARAMETERS = {"second_count": 0,
                   "stop_count":   0,
                   "buffer_size":  0,
                   "zeros_buffer": None,
                   "input_buffer": None}

# FILE_NAME  = 'matrix_mult_TF'
FILE_NAME  = 'SGEMM_Kernel_v3.aocx'
PLATFORMS  = None
DEVICES    = None
CTX        = None
BINARY     = None
PRG        = None
QUEUE      = None
KERNEL_PAD = None
KERNEL_MLT = None

D_BIAS_BUF_NONE = None

DIMS = {}
WEIGHTS_BUFFERS = {}
LAYERS_BUFFERS  = {}
MODEL  = {}
MODELS = {}
LOGIC_MODELS = {}

# Variables para controlar el flujo de la sesión
CREATE_SESSION  = False
RESTART_SESSION = False
CREATED_MODELS  = []
FILES_MODELS    = []

D_BIAS_BUF_NONE = None

PROGRAM = True

```

```
65 TIMES = []
```

Método 10: model_globals.py

```
1
2  import pyopencl as cl
3  import model_globals
4  import numpy as np
5  from profilehooks import timecall
6
7  '''
8  Función para setear los argumentos del kernel. Estos argumentos son:
9      - float *restrict C:         Buffer para el resultado
10     - float *restrict A:         Buffer para la matriz A
11     - float *restrict B:         Buffer para la matriz B
12     - float *restrict Bias:      Buffer para el bías
13     - int A_width, int B_width: Número de columnas de las matrices A y B
14     - int act, int ctrl_bias:    Función de activación y control del bías
15     - float const_bias:          Valor constante para el bias
16 '''
17
18 # @timecall(immediate=True)
19 def calculate(d_a_buf, d_b_buf, d_c_buf, M1, M2, M3, BS, C_Sizes,
20               control, ctrl_bias, d_bias_buf):
21     """
22     Calcula el producto matricial
23     void matrixMult_16_2_act( // Input and output matrices
24                 __global float *restrict C,
25                 __global float *restrict A,
26                 __global float *restrict B,
27                 __global float *restrict Bias,
28                 // Widths of matrices.
29                 int A_width, int B_width,
30                 int act, int ctrl_bias, float const_bias
31                 )
32     """
33
34     # print("MAX_WORK_GROUP_SIZE: " + str(model_globals.DEVICES[0].
       max_work_group_size))
35     # print("MAX_WORK_ITEM_SIZES: " + str(model_globals.DEVICES[0].
       max_work_item_sizes))
36
37
38     event = model_globals.KERNEL(model_globals.QUEUE, (M3[1], M3[0]), (BS
       , BS),
39                                  # Buffers para resultado, matriz A, B y
       vector de bías
40                                  d_c_buf, d_a_buf, d_b_buf, d_bias_buf,
41                                  # Buffer para la cache en la función
       softmax
42                                  M1[1], M2[1], control, ctrl_bias, 0)
43     event.wait()
44
45 def calculate_t(d_a_buf, d_b_buf, d_c_buf, M1, M2, M3, BS, C_Sizes,
```

```
46                 control, ctrl_bias, d_bias_buf):
47     """
48     Calcula el producto matricial
49     void matrixMult_16_2_act( // Input and output matrices
50                 __global float *restrict C,
51                 __global float *restrict A,
52                 __global float *restrict B,
53                 __global float *restrict Bias,
54                 // Widths of matrices.
55                 int A_width, int B_width,
56                 int act, int ctrl_bias, float const_bias
57                 )
58     """
59 
60     # print("MAX_WORK_GROUP_SIZE: " + str(model_globals.DEVICES[0].
    max_work_group_size))
61     # print("MAX_WORK_ITEM_SIZES: " + str(model_globals.DEVICES[0].
    max_work_item_sizes))
62 
63 
64     event = model_globals.KERNEL(model_globals.QUEUE, (M3[1], M3[0]), (BS
    , BS),
65                                  # Buffers para resultado, matriz A, B y
    vector de bías
66                                  d_c_buf, d_a_buf, d_b_buf, d_bias_buf,
67                                  # Buffer para la cache en la función
    softmax
68                                  M1[1], M2[1], control, ctrl_bias, 0)
69     event.wait()
70     model_globals.TIMES.append((event.profile.end-event.profile.start)*1e
    -9)
71 
72 def result(h_c_aux, C_Sizes, d_c_buf):
73     cl.enqueue_copy(model_globals.QUEUE, h_c_aux, d_c_buf)
74     return h_c_aux[0:C_Sizes[0], 0:C_Sizes[1]]
75 
76 # @timecall(immediate=True)
77 def run_args(a, b, c, dim_a, dim_b, act, output, bias, tpe):
78     """
79     Realiza el producto de las matrices
80      - A_height & A_width = M1[0][0], M1[1][0]
81      - B_height & B_width = M2[0][0], M2[1][0]
82      - C_height & C_width = M1[0][0], M2[1][0]
83     """
84     ctrl_bias = 1
85     if bias is None:
86         ctrl_bias = 0
87         bias = model_globals.D_BIAS_BUF_NONE
88 
89     # Se realiza el cálculo del producto matricial: C_sizes = [C_height,
    Cwidth]
90     '''
91     void matrixMult_16_2_act( // Input and output matrices
```

```
92                      __global float *restrict C,
93                      __global float *restrict A,
94                      __global float *restrict B,
95                      __global float *restrict Bias,
96                      // Widths of matrices.
97                      int A_width, int B_width,
98                      int act, int ctrl_bias, float const_bias
99                      )
100     '''
101     calculate(a, b, c, dim_a, dim_b, [dim_a[0], dim_b[1]],
102               model_globals.BS, [dim_a [0], dim_b[1]],
103               act, ctrl_bias, bias)
104     if tpe == None:
105         # Se almacena el resultado en el buffer de salida h_c
106         return result(output, [dim_a[0], dim_b[1]], c)
107     else:
108         return c
```

Método 11: kernel.py

Anexo II: Códigos Python para capas convolucionales

```
1
2  import numpy as np
3  from math import ceil
4  from cnn_layers import *
5
6  '''
7  Método que genera la máscara de índices necesaria para realizar
8  el padeo de las imágenes cuando se trata de una convolución 2D de
9  tipo SAME. La máscara es utilizada por la PFGA para indexar el dato
10 en su posición correcta en la matriz de salida que se encuentra ya
       padeada.
11 '''
12 def make_mask(img_size, batch, f_size, BS, pad):
13
14     h, w = img_size[1], img_size[2]
15     # Pad top, bottom, left & right
16     p_t, p_l, p_r = pad[2], pad[4], pad[5]
17     # Zeros top, bottom, left & righ
18     sh = 0
19     if w % 2 == 0:
20         sh = -1
21     z_t = p_t*(p_r+p_l+w) + p_r + sh
22     # Se crea la máscara con los índices
23     i_h, i_w = get_bs_pad(img_size, BS) # Dimensión de H y W para máscara
        de índices
24     index = np.arange(i_h*i_w).reshape(i_h, i_w)
25     # Tamaño total de filas en una columna con el padeo:
26     rows = (h+pad[0]) * (w+pad[1])
27     # Se crea la máscara con valores de -1 donde se insertarán los í
       ndices
28     if rows % BS != 0:
29         rows = rows + (BS - rows % BS);
30     msk = -1 * np.ones((rows, i_w), dtype=np.int32)
31
32     # Se rellena la parte de la máscara creada con -1 con los índices
       necesario
33     shft = w+p_l+p_r
34     shft_2 = p_l+p_r
35     for i in range(h):
36         msk[z_t+i*shft:i*shft_2+z_t+(i+1)*w, 0:batch] = index[(i)*w:(i+1)
       *w, 0:batch]
37     return msk
38
39 '''
40 Método que genera la máscara para padear las imágenes de entrada de tipo
41 flatten cuando se trata de un UpSampling2D para el tipo bilinear
42 '''
43 def msk_sampling_b(i_s, f=(2,2), BS=16):
44     # Control para batch múltiplo de BS
45     batch = i_s[0]
46     pad_dim = batch
```

```
47      if batch % 16 != 0:
48          pad_dim = batch + (BS - batch % BS)
49
50      n_s = (i_s[1]*f[0], i_s[2]*f[1]) # Nuevo tamaño de la imagen
51
52      # Se construyen los índices y matrices
53      idx  = np.arange(0, np.prod(i_s), batch).reshape(i_s[1:3])
54      cont = -1 * np.ones(n_s)
55      mask = -1 * np.ones((np.prod(n_s), pad_dim), dtype=np.int32)
56
57      # Se crean los índices para cada imagen del batch
58      for i in range(batch):
59          # Se rellena la máscara con los valores de índice
60          cont[::f[0], ::f[1]] = idx + i
61          cont_aux = cont.flatten()
62          mask[:, i] = cont_aux
63
64      return mask
65
66  '''
67  Método que genera la máscara para padear las imágenes de entrada de tipo
68  flatten cuando se trata de un UpSampling2D para el tipo nearest
69  '''
70  def msk_sampling_n(i_s, f=(2,2), BS=16):
71      # Control para batch múltiplo de BS
72      batch = i_s[0]
73      pad_dim = batch
74      if batch % 16 != 0:
75          pad_dim = batch + (BS - batch % BS)
76
77      n_s = (i_s[1]*f[0], i_s[2]*f[1]) # Nuevo tamaño de la imagen
78
79      # Se construyen los índices y matrices
80      idx      = np.repeat(np.arange(0, np.prod(i_s), batch), f[0]).reshape
        ((i_s[2], n_s[0]))
81      idx      = np.repeat(idx, f[1], axis=0).T
82      idx_aux = idx.T.flatten()
83      mask = -1 * np.ones((np.prod(n_s), pad_dim), dtype=np.int32)
84
85      # Se crean los índices para cada imagen del batch
86      for i in range(batch):
87          # Se rellena la máscara con los valores de índice
88          mask[:, i] = idx_aux + i
89
90      return mask
91
92  '''
93  Función que crea una máscara para el producto matricial de tipo disperso
        en
94  función del Block Size de la FPGA
95  '''
96  def create_s_mask(a, nrows, ncols, pad_with_zeros=False):
97      r, c = a.shape
```

```
98      if pad_with_zeros:
99          r_, c_ = int(np.ceil(r / nrows)), int(np.ceil(c / ncols))
100         a = np.pad(a, ((0, r - r_ * nrows), (0, c - c_ * ncols)))  # añ
    adir ceros para que cada submatriz sea igual
101     else:
102         r_, c_ = r // nrows, c // ncols
103         a = a[:r_ * nrows, :c_ * ncols]  # quitar los últimos elementos
    para que cada submatriz sea igual
104     a = a.reshape(r_, nrows, c_, ncols)
105     return np.count_nonzero(a, axis=(1,3)).astype(np.bool).astype(np.
    int32)
106
107
108     #This is only for using one filter
109     if not len(filter.shape) == 3:
110         raise ValueError("The size of the filter should be (filter_height
    , filter_width, filter_depth)")
111
112     if not len(input.shape) == 3:
113         raise ValueError("The size of the input should be (input_height,
    input_width, input_depth)")
114
115     if not filter.shape[2] == input.shape[2]:
116         raise ValueError("the input and the filter should have the same
    depth.")
117
118 '''
119 Método para padear la matriz para que sea múltiplo del Block Size
120 '''
121 def pad(array, BS, dtype=np.float32):
122     if isinstance(array, tuple):
123         i_h, i_w = array
124     else:
125         i_h, i_w = array.shape
126
127     o_h, o_w = i_h, i_w
128     if i_h % BS != 0:
129         o_h = i_h + (BS - i_h % BS);
130     if i_w % BS != 0:
131         o_w = i_w + (BS - i_w % BS);
132
133     if isinstance(array, tuple):
134         return (o_h, o_w)
135     else:
136         cont = np.zeros((o_h, o_w))
137         cont[0:i_h, 0:i_w] = array
138         return cont.astype(dtype)
139
140
141 '''
142 Método para padear las dimensiones de la imagen en tipo flatten y
143 teniendo en cuenta el batch. Esto es necesario para calcular
144 correctámente los índices de la máscara en el método make_mask
```

```
'''
def get_bs_pad(img_size, BS):
    rows, batch = img_size[1]*img_size[2], img_size[0]
    o_h, o_w = rows, batch
    if rows % BS != 0:
        o_h  = rows  + (BS - rows  % BS)
    if o_w % BS != 0:
        o_w  = batch + (BS - batch % BS)
    return o_h, o_w

'''
Método para crear la matriz necesaria como input
a la red neuronal convolucional
'''
def flat_image(input, BS):
    batch, i_h, i_w = input.shape
    cont = np.zeros((i_h*i_w, batch), dtype=np.float32)
    for i in range(batch):
        cont[:, i] = input[i, :, :].flatten()
    return pad(cont, BS)

'''
Método que genera los índices necesarios para realizar la
convulicón 2D  a través del producto matricial con la matriz
de convolución comprimida
    - shift_rw: Índices de desplazamiento para el vector B
    - shift_en: señal enable para indicar desplazamiento en
                el vector A (matriz de conv. comprimida)
    - shift_by: Índices de desplazamiento para el block_y
'''
def shift_indexes(filters, input_size, BS, padding="same"):
    # Valores de retorno
    shift_en, shift_by = None, None
    # shapes
    k_h, k_w, n_f = filters.shape
    i_h, i_w = input_size
    if padding == "same":
        i_h, i_w = i_h+k_h-1, i_w+k_w-1
    o_h, o_w = i_h-k_h+1, i_w-k_w+1

    # Valores iniciales
    vectors = []
    resto   = (i_w%BS)-(k_w-1)
    rows_1  = o_h * o_w
    rows_2  = ceil(rows_1/BS)
    # Creo vector de índices mediante np.arange
    cont     = np.arange(rows_1, dtype=np.int32)

    # Array para las posiciones a extraer de shift_rw
    index_i = np.arange(0, rows_1, BS)
    index_o = index_i + BS - 1
    index_o[-1] = index_i[-1]

```

```
198     # Se realiza el desplazamiento en la posición
199     # correspondiente: Cada "o_w" posiciones
200     for i in range(1, rows_2):
201         cont[i*o_w:(i+1)*o_w] += i*(k_w-1)
202 
203     # Vector con posiciones de desplazamiento en B
204     shift_rw = cont[index_i]
205     shift_a, shift_b, shift_c, shift_d = rows_2, 0, 0, 0
206     if o_w % BS != 0:
207         shift_by = np.zeros(rows_2,  dtype=np.int32) # Vector de
    desplazamientos
208         # Vector de enable para desplazamiento: Solo si o_w no es
209         # múltiplo del BLOCK_SIZE
210         shift_en = ((cont[index_o] - shift_rw - BS) > 0).astype(int)
211         indexs   = np.nonzero(shift_en)[0]
212         # Vector de desplazamientos para el block_y
213         init  = np.arange(resto, (len(indexs)+1)*resto, resto)
214         zeros = np.where(init % BS == 0)[0]
215         init  = np.delete(init, zeros)
216         vectors.append(init)
217         while len(zeros) != 0:
218             init  = np.arange(init[-1]+resto, init[-1]+resto*(len(zeros)
    +1), resto)
219             zeros = np.where(init % BS == 0)[0]
220             vectors.append(np.delete(init, zeros))
221         shift_by[indexs] = np.concatenate(vectors) % BS
222         shift_a, shift_b, shift_c, shift_d = 2*rows_2, 3*rows_2, 4*rows_2
    , rows_2
223     return {"shift_rw": shift_rw, "shift_en": shift_en, "shift_by":
    shift_by,
224             "shift_a" : shift_a,  "shift_b" : shift_b,
225             "shift_c" : shift_c,  "shift_d" : shift_d}
```

Método 12: masking.py

```
1 
2 import numpy as np
3 from math import ceil
4 from scipy import linalg
5 from masking import *
6 
7 '''
8 Función para calcular las medidas de salida para una convolución de tipo
    VALID
9 Retorna solo las medidas de salida de la imagen
10 '''
11 def conv2d_valid(input, filter, bias=0, strides=(1, 1)):
12     input_w, input_h = input[1], input[0]      # input width and input
    height
13     filter_w, filter_h = filter[1], filter[0]  # filter width and filter
    height
14 
15     output_h = int(ceil(float(input_h - filter_h + 1) / float(strides[0])
    ))
```

```
16      output_w = int(ceil(float(input_w - filter_w + 1) / float(strides[1])
        ))
17
18      return [output_h, output_w], [0, 0, 0, 0, 0, 0]
19
20  '''
21  Función para calcular las medidas de salida para una convolución de tipo
        SAME
22  Retorna las medidas de salida de la imagen así como el padeo necesario
        para realizar
23  correctamente la convolución a través del producto matricial
24  '''
25  def conv2d_same(input, filter, bias=0, strides=(1, 1)):
26
27      input_w, input_h = input[1], input[0]       # input width and input
        height
28      filter_w, filter_h = filter[1], filter[0]   # filter width and filter
        height
29
30      output_h = int(ceil(float(input_h) / float(strides[0])))
31      output_w = int(ceil(float(input_w) / float(strides[1])))
32
33      if input_h % strides[0] == 0:
34          pad_along_height = max((filter_h - strides[0]), 0)
35      else:
36          pad_along_height = max(filter_h - (input_h % strides[0]), 0)
37      if input_w % strides[1] == 0:
38          pad_along_width = max((filter_w - strides[1]), 0)
39      else:
40          pad_along_width = max(filter_w - (input_w % strides[1]), 0)
41
42      pad_top = (pad_along_height // 2)  #amount of zero padding on the top
43      pad_bottom = (pad_along_height - pad_top) # amount of zero padding on
         the bottom
44
45      pad_left = (pad_along_width // 2)  # amount of zero padding on the
        left
46      pad_right = (pad_along_width - pad_left)     # amount of zero padding
         on the right
47
48      return [output_h, output_w], [pad_along_height, pad_along_width,
49                                    pad_top, pad_bottom, pad_left,
        pad_right]
50
51  '''
52  Método para generar los coeficientes necesarios para el kernel de
        upsampling 2D
53  '''
54  def upsample_filt(size):
55      """
56      Make a 2D bilinear kernel suitable for upsampling of the given (h, w)
         size.
57      """
```

```
    factor = (size + 1) // 2
    if size % 2 == 1:
        center = factor - 1
    else:
        center = factor - 0.5
    og = np.ogrid[:size, :size]
    return (1 - abs(og[0] - center) / factor) * \
           (1 - abs(og[1] - center) / factor)

'''
Método para generar la matriz de convolución completa
'''
def cnv2_mtx(filters, input_size, BS, padding="same", alg=1):
    # shapes
    k_h, k_w, n_f = filters.shape
    i_h, i_w = input_size

    # Se calculan las medidas en función del tipo de padeo, "same" o "valid"
    if padding == "same":
        i_h, i_w = i_h+k_h-1, i_w+k_w-1
        o_h, o_w = input_size
    if padding == "valid":
        i_h, i_w = input_size
        o_h, o_w = i_h-k_h+1, i_w-k_w+1
    # Se construyen las matrices individuales de Toeplitz
    toeplitzs = []
    for f in range(n_f):
        toeplitz = []
        kernel = filters[:, :, f]
        for r in range(k_h):
            toeplitz.append(linalg.toeplitz(c=(kernel[r,0], *np.zeros(i_w-k_w)), r=(*kernel[r], *np.zeros(i_w-k_w))))
        toeplitzs.append(toeplitz)
    # Se calcula el tamaño total de la matriz de salida
    h_blocks, w_blocks = o_h, i_h
    h_block, w_block = toeplitz[0].shape
    shape = (h_blocks*h_block, w_blocks*w_block)
    # En el caso de que se decida utilizar un solo bloque de la matriz: Algoritmo apto
    # para cualquier tamaño de imagen. Comprobado hasta imágenes de 1024x1024
    fully = 0
    shape = pad(shape, BS)
    t_l = toeplitz[0].shape[1]*k_h
    if o_w % BS == 0:
        fully = 1
        cont = np.zeros((BS, t_l+k_w-1), dtype=np.float32)
    else:
        cont = np.zeros((2*BS, t_l+k_w-1), dtype=np.float32)
    # Se contstruye la matriz de convolución por bloques
    W_convs = []
    for f in range(n_f):
```

```
107         cont[0:BS, 0:t_l] = np.concatenate(toeplitzs[f], 1)[0:BS,:]
108         if fully == 0:
109             cont[BS::, k_w-1::] = cont[0:BS, 0:t_l]
110         aux = np.copy(cont)
111         W_convs.append(aux)
112     return pad(np.concatenate(W_convs), BS), shape, fully
```

Método 13: cnv_alg.py

```
1
2  import cnv_alg
3  import masking
4  import numpy as np
5
6  '''
7  Método principal para crear las matrices de convolución
8  '''
9  def conv_struct(input_size, kernel, BS, padding="same"):
10     if padding != "same" and padding != "valid":
11         raise ValueError("The padding modes are same or valid")
12     if not len(input_size) == 3:
13         raise ValueError("The size of the input should be (batch,
       input_height, input_width)")
14     if not len(kernel.shape) == 3:
15         raise ValueError("The size of the filter should be (filter_height
       , filter_width, filters)")
16
17     # Se coge la matriz de convolución
18     H, shape, fully = cnv_alg.cnv2_mtx(kernel, input_size[1:3], BS,
       padding)
19
20     # Se crea la máscara para la matriz de convolución
21     s_mask = masking.create_s_mask(H, BS, BS, pad_with_zeros=False)
22
23     # Se crea la máscara para padear la imagen
24     p_mask = np.zeros(0, dtype=np.int32)
25     if padding == "same":
26         dims, p_dims = cnv_alg.conv2d_same(input_size[1:3], kernel.shape)
27         dims = (input_size[0], dims[0], dims[1])
28         p_mask = masking.make_mask(dims, input_size[0], kernel.shape, BS,
        p_dims)
29     if padding == "valid":
30         dims, p_dims = cnv_alg.conv2d_valid(input_size[1:3], kernel.shape
       )
31         p_mask = None
32
33     # Se coge el diccionario con los valores necesarios para
34     # el shifteo de la matriz de convolución en la FPGA
35     cnv2_struct = masking.shift_indexes(kernel, input_size[1:3], BS,
       padding=padding)
36
37     # Se coge la dimensión de la matriz de shift, máscara y batch
38     dim_sht = cnv2_struct.get("shift_rw").shape[0]
39     dim_msk = s_mask.shape[1]
```

```
40      if input_size[0] % BS != 0:
41          batch = (input_size[0] // BS + 1) * BS
42      else:
43          batch = input_size[0]
44
45      # Se introduce la matriz de convolución en la estructura
46      cnv2_struct["H"] = H
47      cnv2_struct["shape"] = shape
48      cnv2_struct["fully"] = fully
49      cnv2_struct["padding"] = padding
50      cnv2_struct["s_mask"] = s_mask
51      cnv2_struct["p_mask"] = p_mask
52      cnv2_struct["m_width"] = s_mask.shape[1]
53      cnv2_struct["filters"] = kernel.shape[2]
54      msk = None
55      # Se crea una matriz compacta que albergará la máscara y los valores
56      # de shifteo para el bloque de convolución en la FPGA
57      if fully == 1:
58          msk = np.zeros((2, dim_sht), np.int32)
59          msk[0, 0:dim_msk] = s_mask[0, 0:dim_msk]
60          msk[1, :] = cnv2_struct.get("shift_rw") * batch
61      if fully == 0:
62          msk = np.zeros((5, dim_sht), np.int32)
63          msk[0:1, 0:dim_msk] = s_mask[0:1, 0:dim_msk]
64          msk[2, :] = cnv2_struct.get("shift_rw") * batch
65          msk[3, :] = cnv2_struct.get("shift_en")
66          msk[4, :] = cnv2_struct.get("shift_by")
67      cnv2_struct["Mask"] = msk
68
69      return cnv2_struct
70
71  def MaxPooling2D(i_size, pool_size=(2, 2), strides=None, padding="valid",
        BS=16):
72      kernel = np.ones(pool_size)
73      kernel.shape = (pool_size[0], pool_size[1], 1)
74      cnv_struct = conv_struct(i_size, kernel, BS, padding=padding)
75      return cnv_struct
76      pass
77
78  def AveragePooling2D(i_size, pool_size=(2, 2), strides=None, padding="
      valid", BS=16):
79      kernel = (1/np.prod(pool_size)) * np.ones(pool_size)
80      kernel.shape = (pool_size[0], pool_size[1], 1)
81      cnv_struct = conv_struct(i_size, kernel, BS, padding=padding)
82      return cnv_struct
83
84  def Conv2D(i_size, kernel, strides=(1,1), padding="valid",
85             act="linear", BS=16):
86      cnv_struct = conv_struct(i_size, kernel, BS, padding=padding)
87      return cnv_struct
88
89  def UpSampling2D(i_size, size=(2, 2), interpolation="nearest", BS=16):
90      kernel = None
```

```
91      if interpolation == "bilinear":
92          r_mask = masking.msk_sampling_n(i_size, f=size, BS=16)
93          k_s = (2 * size[0]) - (size[0] % 2) # Kernel size
94          kernel = cnv_alg.upsample_filt(k_s)
95          kernel.shape = (kernel.shape[0], kernel.shape[1], 1)
96          i_size = (i_size[0], i_size[1]*size[0], i_size[2]*size[1])
97          cnv_struct = conv_struct(i_size, kernel, BS, padding="same")
98          cnv_struct["up_mask"] = r_mask
99          cnv_struct["up_algorithm"] = "bilinear"
100     if interpolation == "nearest":
101         cnv_struct = {}
102         r_mask = masking.msk_sampling_n(i_size, f=size, BS=16)
103         cnv_struct["up_mask"] = r_mask
104         cnv_struct["up_algorithm"] = "nearest"
105
106     return cnv_struct
107
108 def DownSampling2D(i_size, size=(2, 2), interpolation="nearest", BS=16):
109     pass
```

Método 14: conv2d.py