HLSinf: Una Plataforma de Aceleración de Procesos de Inferencia en FPGA aplicado a Imágenes Médicas

Laura Medina[1], Izan Catalán[1], José Flich[1], Carles Hernández[1], Andrea Bragagnolo[2 3]
Fabrice Auzanneau[4] y David Briand[4]

Resumen— **Las Redes Neuronales en el procesamiento de imágenes en sistemas embebidos suponen dos requisitos contrapuestos: el aumento de las necesidades de potencia de cálculo a medida que los modelos se hacen más complejos y, además, la limitación de los recursos disponibles en estos sistemas. Por tanto, es habitual la compresión de modelos de redes neuronales utilizando para ello técnicas de cuantificación y poda (*pruning*). Estos modelos deben ajustarse a sistemas reconfigurables como las FPGA para que el sistema embebido funcione correctamente. En este artículo, presentamos HLSinf, un *framework* de código abierto para el desarrollo de aceleradores de redes neuronales personalizados al caso de uso requerido. Además, HLSinf está enfocado para FPGA y proporciona un soporte eficiente a los modelos de redes neuronales cuantificados y podados. Mediante HLSinf aumentamos la velocidad del proceso de inferencia de manera significativa para diferentes aplicaciones como las basadas en imágenes médicas. En particular, obtenemos un factor de aceleración de 90 cuando combinamos la cuantificación y la poda con la flexibilidad de HLSinf en comparación con CPU.**

Palabras clave— **FPGA, quantización, prunning, Redes Neuronales, inferencia**

I. Introducción

EL aprendizaje automático [1], especialmente las Redes Neuronales y el Aprendizaje Profundo [2], se utilizan habitualmente en nuestra vida cotidiana: tanto si hacemos clic en las recomendaciones personalizadas de los sitios web como si utilizamos la detección de rostros, siempre podemos encontrar un sistema artificial inteligente funcionando en segundo plano. Su capacidad de aprender a partir de cantidades masivas de datos permite un rendimiento excelente en ámbitos como la visión por ordenador, el reconocimiento del habla o el procesamiento del lenguaje natural. Las Redes Neuronales Convolucionales (CNN) [3] se utilizan de forma masiva en campos como la visión por ordenador [4], el análisis informático de imágenes visuales [5] o la conducción autónoma [6]. Estos modelos consiguen buenos resultados a costa de un aumento de la complejidad computacional. Por ejemplo, ResNet [7] incluye decenas de millones de parámetros, llegando a los cientos de millones en el caso de VGG-Net [8]. Estos requisitos de cálculo suponen una limitación cuando se aplican a sistemas más sencillos como, por ejemplo, en dispositivos embebidos.

La mayoría de los sistemas embebidos establecen la eficiencia energética como su principal objetivo de diseño lo cual conlleva al desarrollo de sistemas con un conjunto limitado de recursos. Para conseguir eficiencia y flexibilidad simultáneamente se suelen utilizar dispositivos configurables como las FPGA. Sin embargo, debido al requisito computacional, el despliegue de modelos de redes neuronales (NN) no-comprimidos es imposible o ineficiente en la mayoría de los casos. Es por esto que los modelos NN suelen comprimirse mediante técnicas de computación aproximada (*approximate computing*) como poda y/o cuantificación, dando como resultado la reducción del tamaño del modelo y, por tanto, una disminución de los requisitos de cálculo tanto en número de operaciones (gracias a la poda) como en complejidad aritmética (gracias a la cuantificación). Un modelo comprimido se puede desplegar completamente en la FPGA mediante herramientas de síntesis [9], [10] en caso de que el modelo quepa o, por otra parte, se puede transformar en un programa específico que se ejecuta en un acelerador NN genérico ya implementado en la FPGA [11].

En este trabajo, contribuimos con una plataforma de código abierto para desarrollar aceleradores personalizados en FPGA específicamente adaptados a modelos podados y cuantificados. La plataforma propuesta, denominada HLSinf, permite la adaptación óptima del formato de precisión de datos utilizado en el proceso de cuantificación y, además, tiene en cuenta el proceso de poda para adaptar de forma flexible los recursos necesarios en la FPGA a la implementación final del acelerador. HLSinf puede utilizarse para desplegar implementaciones finales con diferentes rendimientos y diferentes capas de NN, pudiendo así adaptarse a los requisitos de diferentes modelos de NN. Asimismo, se ha adaptado los aceleradores a la librería EDDL (*European Distributed Deep Learning*) [12]. EDDL es una plataforma de código abierto que permite definir, entrenar e inferir redes neuronales en CPUs y GPUs. Gracias a esta adaptación, la librería EDDL soporta nativamente FPGAs con los aceleradores HLSinf. La evaluación realizada en este artículo demuestra que los modelos cuantificados y podados pueden aumentar en gran medida el rendimiento cuando se combinan con HLSinf. En concreto, los resultados muestran que se puede conseguir

[1]Universitat Politècnica de València, e-mail: laumecha@upv.es
[2]Dipartimento di Informatica, Università degli Studi di Torino
[3]Synesthesia s.r.l., Corso Dante 118, 10126, Torino
[4]Université Paris Saclay, CEA, List, F-91120, Palaiseau, France

un factor de incremento en rendimiento de hasta 90 en aplicaciones típicas basadas en imágenes médicas utilizando modelos NN en FPGA.

II. HLSinf

HLSinf [1] es una plataforma de código abierto desarrollada en *High-Level Synthesis* (HLS). HLS permite programar aplicaciones para FPGA utilizando lenguajes de alto nivel como C o C++, reduciendo el tiempo de desarrollo de código en FPGA y facilitando la implementación de mecanismos de paralelización de código mejorando así la productividad. Consecuentemente, permite una adopción más fácil y reduce la necesidad de elevados conocimientos técnicos para desarrollar instancias del acelerador. El objetivo principal de HLSinf es permitir la implementación de aceleradores personalizados para diversos tipos de operaciones y formatos de datos, especialmente centrados en modelos cuantificados.

HLSinf implementa el modelo de flujo de datos, el cual permite ejecutar varias funciones concurrentemente. Para ello, analiza las dependencias de datos entre funciones secuenciales y crea canales FIFO los cuales permiten ejecutar una función antes de que se generen todos los datos necesarios de la función anterior. En HLSinf, los datos fluyen a través de los diferentes módulos. La figura 1 muestra el diseño de la arquitectura del acelerador actual, donde las flechas representan los flujos de datos que conectan los módulos. Estos módulos pueden añadirse o eliminarse en tiempo de diseño, adaptando así el acelerador al caso de uso requerido. Los módulos de lectura toman los datos de la memoria externa de la FPGA y los introducen en el acelerador a través de flujos de datos. A continuación, los datos se procesan y se producen datos de salida que se vuelven a escribir en la memoria. Los módulos de almacenamiento interno se utilizan para reducir el acceso a la memoria. El diseño basado en módulos permite segmentar las operaciones adicionales que se realizan en las NN: funciones de activación como ReLU, operaciones de *pooling* u otras funciones que puedan ser necesarias. Cada módulo tiene una interfaz basada en flujos.

HLSinf está diseñado en torno al concepto de "*channel slicing*". La principal operación realizada por el acelerador es la operación de convolución 2D. Esta operación toma como entrada un conjunto de canales de entrada (mapas de características de capas anteriores) y produce un conjunto de canales de salida (mapas de características de salida). El acelerador maneja en paralelo un conjunto de canales de entrada definido por el parámetro CPI (*channels per input* o canales por entrada) y produce en paralelo un conjunto de canales de salida definido por el parámetro CPO (*channels per output* o canales por salida). Tanto los parámetros CPI como CPO pueden ser instanciados en tiempo de diseño, lo cual permite la implementación de aceleradores de diferentes tamaños y grados de aceleración. Estos dos parámetros indican el nivel de paralelismo del acelerador ya que HLSinf está diseñado para procesar CPI píxeles de entrada en paralelo y producir CPO píxeles de salida en paralelo por ciclo de reloj.

Todos los flujos de datos del acelerador pueden modificarse para adaptarse al modelo objetivo. Por ejemplo, cuando se trabaja con modelos cuantificados es común observar diferentes formatos de precisión de datos entre diferentes parámetros como las activaciones, los filtros y los bías. Es por esto que la plataforma HLSinf permite utilizar tipos de datos de coma flotante de 32 bits, de coma fija o enteros con un número de bits determinado.

La operación más exigente desde el punto de vista computacional en las NNs basadas en imágenes es la convolución 2D. HLSinf permite personalizar el módulo de convolución seleccionando el tipo de convolución a realizar. Actualmente, es posible seleccionar uno de los tres algoritmos de convolución 2D implementados: la convolución directa, el algoritmo *Winograd* [13] y, finalmente, convolución *DepthWise Separable* [14]. La figura 1 muestra el caso de la convolución 2D directa. El módulo está compuesto por diferentes submódulos conectados a través de flujos de datos. En su entrada, el módulo recibe el bías, los filtros y los datos. El bías se agrupa en bloques consecutivos de CPO elementos de bías. Para los filtros, la operación de convolución recibe filtros de tamaño $CPO \times CPI \times KH \times KW$, donde KH y KW representan el alto y ancho del filtro, respectivamente. Así, para cada mapa de características de salida se proporciona un conjunto de CPI filtros. Para los datos de entrada, el módulo recibe los píxeles de entrada de CPI canales (un grupo de canales) de forma entrelazada (un píxel de cada canal). Los datos de entrada se preprocesan con los módulos *padding* y *cvt*. El primero rellena los píxeles de entrada con relleno horizontal y/o vertical (píxeles de valor cero) y reenvía la imagen producida; el módulo *cvt* se encarga de generar tramas. Una trama se define como un fragmento de los datos de entrada de tamaño $KH \times KW$ con el que hay que realizar la convolución. Este módulo envía CPI tramas de tamaño $KH \times KW$ en cada ciclo de reloj.

Los datos preprocesados y los filtros llegan al módulo *mul* para realizar la operación de convolución con CPO bloques de multiplicadores. Cada bloque multiplica las CPI tramas de entrada (cada una de tamaño $KH \times KW$) por un conjunto de CPI filtros (cada uno de tamaño $KH \times KW$). Cada bloque tiene multiplicadores y sumadores de $KH \times KW$ operandos. Cada bloque reduce los datos de salida obteniendo un píxel de salida. En cada ciclo de reloj, el módulo produce y envía CPO píxeles. La figura 2 muestra la operación de convolución. El número de unidades MAC (*multiply and accumulate*) utilizadas es de $CPI \times CPO \times KH \times KW$. Por lo tanto, su capacidad de cálculo depende principalmente de los parámetros CPI y CPO.

Finalmente, el módulo *add* se encarga de acumular los mapas de características de salida y añadir el bías a cada canal de salida. Para ello, el módulo propor-

[1] Código fuente en `https://github.com/PEAK-UPV/HLSinf`

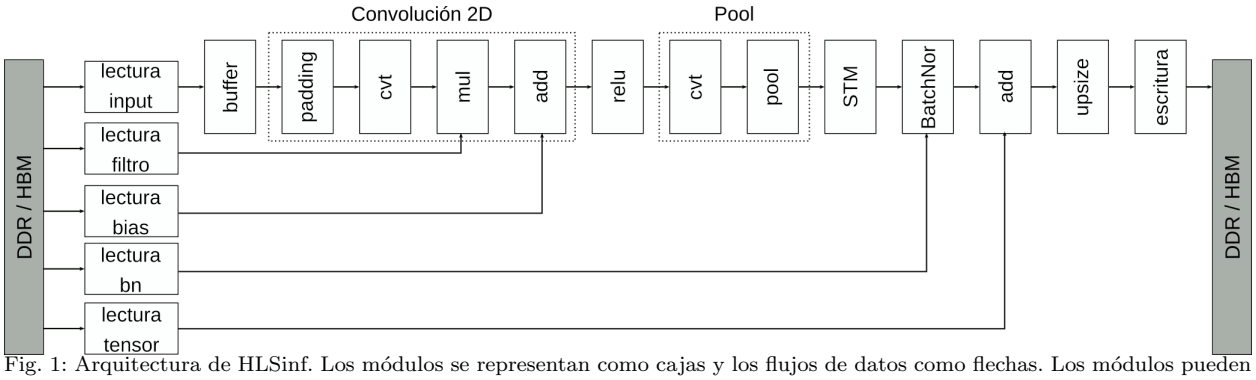

Fig. 1: Arquitectura de HLSinf. Los módulos se representan como cajas y los flujos de datos como flechas. Los módulos pueden colocarse en cualquier orden.

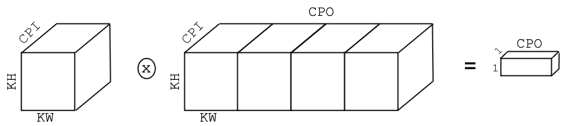

Fig. 2: Operación de convolución 2D realizada en el módulo *mul* en cada ciclo de reloj.

ciona un *buffer* de salida de tamaño $H \times W \times CPO$, donde H y W son la altura y la anchura del mapa de características, respectivamente. Una vez que se han realizado todas las iteraciones de entrada, el módulo envía el contenido del *buffer* de salida al siguiente módulo y, en última instancia, para su almacenamiento en la memoria.

Con este diseño, y disponiendo de suficiente ancho de banda de memoria a la entrada y salida del acelerador, podemos determinar el tiempo de ejecución del acelerador al realizar una convolución 2D. Suponiendo una entrada de $I \times H \times W$ y O mapas de características, su tiempo de ejecución será $\frac{I}{CPI} \times H \times W \times \frac{O}{CPO}$ ciclos de reloj. Las capas adicionales añadirán un retardo constante de unos pocos ciclos dado que el diseño sigue un esquema segmentado.

III. Integración EDDL-HLSinf

HLSinf puede diseñarse para ejecutar tareas específicas de cálculo intensivo en la FPGA. Sin embargo, no todas las capas de un modelo de NN son adecuadas o merece el esfuerzo de ser ejecutadas en dicho sistema. Este es el caso de capas ligeras como, por ejemplo, *softmax*. Además, en sistemas embebidos específicos, es necesario un codiseño HW/SW de grano fino. Por estas razones, necesitamos un método para realizar el proceso de inferencia de forma combinada entre la CPU y la FPGA y, al mismo tiempo, adaptar una librería de aprendizaje profundo para el uso efectivo del acelerador HLSinf. Para ello, hemos seleccionado la librería EDDL (*European Distributed Deep Learning*) [2]. EDDL define un conjunto de capas como clases C++ y permite al usuario final construir un modelo conectando capas.

Para permitir un codiseño HW/SW efectivo, hemos implementado una nueva capa dentro de la librería EDDL llamada HLSinf, la cual encapsula todas las funcionalidades del acelerador. La EDDL ejecutará cada capa del modelo y cada una utilizará el dispositivo de destino, ya sea CPU/GPU para las capas normales y FPGA para la capa HLSinf. Las transferencias de datos entre la FPGA y las memorias de la CPU/GPU se realizan de forma transparente cuando es necesario.

A. Adaptación del modelo

EDDL permite llevar a cabo el proceso de entrenamiento de un modelo NN o, alternativamente, cargar un modelo entrenado en otra plataforma utilizando el formato ONNX [15], creando así una compatibilidad con otras plataformas. Para poder utilizar nuestro acelerador HLSinf, hemos diseñado y añadido una nueva funcionalidad dentro de esta librería, la cual transforma un modelo de entrada en un nuevo modelo con capas HLSinf añadidas cuando es requerido. Este nuevo método, llamado *toFPGA()*, permite realizar tres transformaciones de los modelos de entrada. En primer lugar, las capas del modelo original se fusionan en una sola en caso de que el acelerador HLSinf sea capaz de realizar todas estas capas simultáneamente. En segundo lugar, lleva a cabo una transformación de datos si una capa que se ejecuta en la CPU/GPU viene seguida de una capa que se ejecuta en la FPGA o viceversa. Esta transformación viene dada por una nueva capa de transformación. La figura 3 muestra parte de la red VGG16 adaptada con el método *toFPGA()*. Finalmente, este método de transformación adapta los filtros y reorganiza los tensores según la necesidad del acelerador HLSinf. Todos los tensores requeridos por la FPGA se almacenan en la memoria de este dispositivo.

IV. Experimentos

A. Soporte de hardware

Para el estudio de rendimiento se ha utilizado una CPU Intel i7-7800-X a 3,45GHz y una placa FPGA Xilinx ALVEO U200. Para los resultados experimentales de la CPU, se han utilizado los 12 hilos disponibles en la máquina. Para los resultados de la FPGA se ha utilizado un único núcleo para la comunicación entre la placa ALVEO y la CPU.

B. Modelos objetivo

Para la evaluación del acelerador HLSinf se han considerado dos modelos NN donde, en ambos, se

[2]Código fuente en `https://github.com/deephealthproject/eddl`

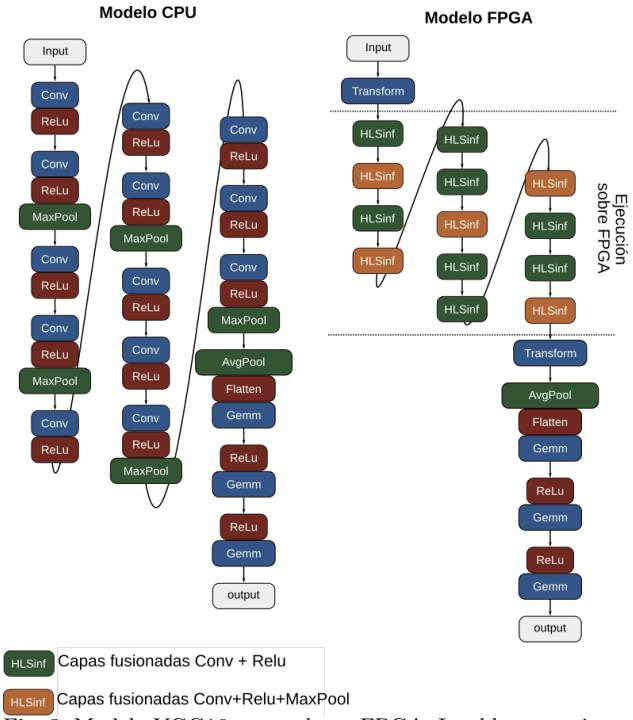

Fig. 3: Modelo VGG16 mapeado en FPGA. Los bloques coinciden perfectamente con el acelerador HLSinf. Se añade una capa de transformación para el movimiento de datos entre la CPU y la FPGA.

han combinado los métodos de compresión de cuantificación y poda. En ambos modelos se ha utilizado el conjunto de datos ISIC para lesiones cutáneas [16]. Se han elegido los modelos VGG16 [8] y SegNet [17] los cuales realizan una clasificación y una segmentación del conjunto de datos respectivamente.

B.1 Poda

En los últimos años, el enfoque de la poda de las NNs ha pasado de ser no estructurado (los parámetros se eliminan de forma independiente) a ser estructurado (se eliminan neuronas enteras). Un trabajo reciente [18] demuestra incluso que los procedimientos estructurados benefician en gran medida la compresión de extremo a extremo y el despliegue en dispositivos embebidos. Siguiendo esta tendencia, para nuestros experimentos, hemos utilizado la estrategia de poda estructurada descrita en [19].

Durante la fase de poda, utilizamos un conjunto de validación para controlar el rendimiento de la red podada. Para nuestra prueba se han creado dos modelos para la tarea de clasificación y uno para el problema de segmentación. Los dos modelos de clasificación se caracterizan por una precisión y una compresión diferentes: para el primero, denominado HA (*high accuracy*), se tiene como objetivo preservar el rendimiento del modelo denso a costa de la proporción de la poda; el segundo, HCR (*high compression*), tiene como objetivo maximizar las neuronas eliminadas, permitiendo un rendimiento menor, aunque todavía aceptable.

Cabe destacar que, para explotar la estructura de dispersión de ceros introducida en la red, hay que eliminar las neuronas igualadas a cero de la arquitectura. Para esta operación se ha utilizado la biblioteca Simplify [20].

B.2 Cuantificación

Junto con la poda, la cuantificación es uno de los métodos de compresión más adoptados. Reduce la cantidad de memoria necesaria para almacenar los parámetros de la red y permite el despliegue de un modelo más grande con la misma memoria inicial para una mayor precisión. El uso de filtros de enteros de 8 bits (INT8) en lugar de los de coma flotante de 32 bits (FP32) permite mejorar el rendimiento en muchos campos: reducción de 4x en el tamaño del modelo; reducción de 2-4x en el ancho de banda de la memoria y una inferencia 2-4x más rápida debido a un cálculo más rápido con aritmética de enteros. Esto se suele conseguir a costa de una ligera disminución de la precisión.

Los métodos de cuantificación pueden dividirse a grandes rasgos en dos categorías [21]: *Quantization Aware Training* (QAT) y *Post-Training Quantization* (PTQ). PTQ cuantifica tanto los filtros como las activaciones para una inferencia más rápida sin necesidad de volver a entrenar el modelo. QAT modela la cuantificación durante el entrenamiento, lo que proporciona una mayor precisión que los esquemas PTQ.

Utilizamos N2D2 [22] de código abierto como herramienta de cuantificación en este estudio. Se ha elegido el método PTQ [23], el cual toma un modelo entrenado usando FP32 y lo cuantifica directamente a INT8, sin ningún reentrenamiento o ajuste. Por lo tanto, la sobrecarga de PTQ es insignificante. En N2D2, el algoritmo de cuantificación post-entrenamiento se realiza en 3 pasos:

- Normalización de los filtros: Todos los filtros se reescalan en el rango $[-1{,}0, 1{,}0]$.
- Activaciones normalizadas: Las activaciones en cada capa se reescalan en el rango $[-1{,}0, 1{,}0]$ para salidas con signo y $[0{,}0, 1{,}0]$ para salidas sin signo.
- Cuantificación: Las entradas, filtros, bias y activaciones se cuantifican con la precisión deseada N_{bits}. La conversión oscila entre $[-1{,}0, 1{,}0]$ y $[0{,}0, 1{,}0]$ hasta $\left[-2^{N_{bits}-1}-1, 2^{N_{bits}-1}-1\right]$ y $\left[0, 2^{N_{bits}}-1\right]$ teniendo en cuenta todas las dependencias.

C. Precisión de los Modelos

La tabla I muestra algunos detalles de los modelos utilizados en nuestros experimentos. La tabla Ia resume los modelos podados en términos de rendimiento (error de clasificación para VGG y puntuación Dice para SegNet) y porcentaje de neuronas restantes. La tabla Ib muestra los resultados de PTQ aplicados al modelo VGG original para $N_{bits} = 8$. Aplicando tanto la poda como la cuantificación se consigue un factor de reducción 100x en los requisitos de memoria con menos de un 2 % de reducción del rendimiento (red HA), y cerca de 1000x con un 15 % de caída del rendimiento (HCR).

Poda		
Modelo	**Error**	**Neuronas restantes**
VGG16-HA	22.84 %	36.19 %
VGG16-HCR	35.66 %	11.64 %
SegNet-Pruned	0.83	59.76 %

(a) Modelos podados

	FP32		PTQ	
Modelo	**Mem**	**Error**	**Mem**	**Error**
VGG16	524 583	21.06 %	131 146	21.06 %
VGG16-HA	18 692	22.84 %	4 673	23.76 %
VGG16-HCR	2 268	35.66 %	567	35.86 %

(b) Modelos cuantificados (memoria representada en kB)

Tabla I: Modelos empleados. a) Modelos podados, el rendimiento se expresa como error de clasificación para VGG16 y como puntuación *Dice* para SegNet. b) Modelos cuantificados, error de clasificación y uso de memoria.

Clasificación				
Modelo	**Disp.**	**Inf.(ms)**	**FPS**	**Mejora**
VGG16	CPU	1430.29	0.70	-
VGG16	FPGA	552.28	1.81	2.59×
VGG16-Quant	FPGA	229.93	4.35	6.22×
VGG16-HA	FPGA	99.33	10.07	14.40×
VGG16-HCR	FPGA	15.09	66.25	94.76×
Segmentación				
Modelo	**Disp.**	**Inf.(ms)**	**FPS**	**Mejora**
SegNet	CPU	3165.97	0.32	-
SegNet	FPGA	757.22	1.32	4.18×
SegNet-Pruned	FPGA	282.23	3.54	11.22×

Tabla II: Tiempo de inferencia en ms, FPS y aceleración (factor de mejora) de los modelos de clasificación y segmentación ISIC para CPU y FPGA.

D. Rendimiento de los Modelos

La tabla II muestra el tiempo de inferencia de una sola imagen de entrada de 224×224 en diferentes modelos y dispositivos para los problemas de segmentación y clasificación del caso de uso de lesiones cutáneas que se encuentra en el conjunto de datos de ISIC. Para la clasificación, el tiempo necesario para la ejecución del modelo completo en la CPU (red VGG16) con aritmética de coma flotante de 32 bits es de 1430,29 ms. Si observamos el tiempo de ejecución de este mismo modelo con la misma aritmética en el dispositivo FPGA podemos ver una mejora significativa dado que el tiempo de ejecución se reduce en un factor de $2,59$ (el tiempo de inferencia en este caso es de 552,28 ms). Para esta ejecución se ha utilizado un acelerador implementado con CPI y CPO de 4 y con aritmética de precisión de coma flotante de 32 bits. Por otra parte, si cuantificamos o podamos el modelo, se pueden reducir el tiempo de inferencia en los modelos ejecutados sobre dispositivos FPGA. En concreto, el modelo cuantificado representado en la tabla, requiere filtros con tipos de datos enteros de 8 bits y activaciones y bías de enteros de 32 bits. Para la ejecución del modelo cuantificado en la FPGA se ha implementado un acelerador con estos tipos de datos. Si se observan los resultados, podemos ver que el tiempo de ejecución se reduce en un factor de $6,22$ (229,93 ms para la inferencia) respecto al modelo ejecutado completamente en la CPU. La mejora viene dada por el hecho de que un formato de datos de menor precisión permite aumentar el paralelismo del acelerador (CPI y CPO) hasta un factor de 8.

La tabla también muestra la ventaja de ejecutar los modelos podados en la FPGA. El tiempo de inferencia se reduce significativamente en este caso. En este caso, se ha utilizado una implementación del acelerador con tipo de datos de coma flotante de 32 bits y parámetros CPI y CPO a 4. Los modelos podados se ajustan perfectamente al acelerador y el tiempo de inferencia se reduce en un factor total de $14,4$ y $94,8$ para ambos modelos, respectivamente (tiempo de inferencia inferior a 100 ms en ambos casos). Para el problema de segmentación del caso de uso de lesiones de la piel se puede ver la misma tendencia de rendimiento. Al pasar a la FPGA, el tiempo de inferencia se reduce en un factor de $4,2$. Además, al utilizar el modelo podado, el tiempo de inferencia se reduce aún más en un factor de $11,2$.

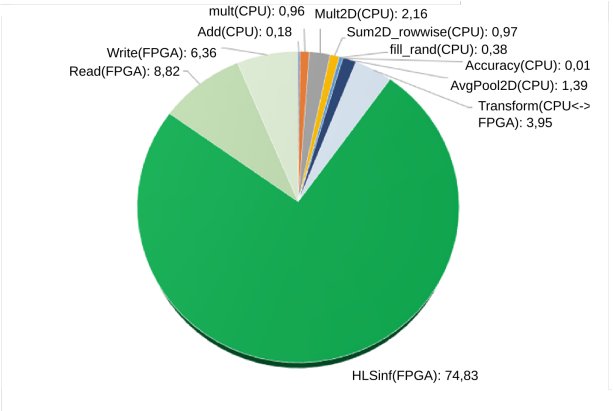

Fig. 4: Descomposición de inferencia por módulos para modelo VGG16-HA mapeado en FPGA.

E. Uso combinado de FPGA y CPU

La Figura 4 muestra la descomposición de la ejecución de la inferencia en el modelo VGG16-HS (clasificación) en módulos. En ella podemos ver como durante el 74 % del tiempo se utiliza la FPGA para realizar, principalmente, las operaciones de convolución y *pooling* del modelo. Ahora bien, durante una gran parte del tiempo de ejecución se utiliza en la CPU. En concreto, la CPU realiza operaciones de cálculo asociadas a capas densas (*mult* y *add*). También podemos observar un tiempo significativo para las operaciones de lectura y escritura sobre la memoria de la FPGA.

En la Figura 5 podemos observar como el tiempo dedicado a FPGA es mucho mayor en comparación al caso anterior, ya que este es prácticamente del 100 %. Esta figura corresponde a la ejecución del modelo SegNet,el cual realiza una segmentación. En este caso, el modelo tiene un uso prácticamente exclusivo de capas convolucionales, y por tanto, prácticamente el 100 % del modelo se ejecuta en la FPGA, minimizando así la transferencia de datos entre CPU y FPGA.

V. Conclusión

En este artículo hemos presentado HLSinf, una plataforma de código abierto para aceleradores de modelos de Redes Neuronales en FPGAs para sistemas embebidos. La plataforma se ha integrado junto

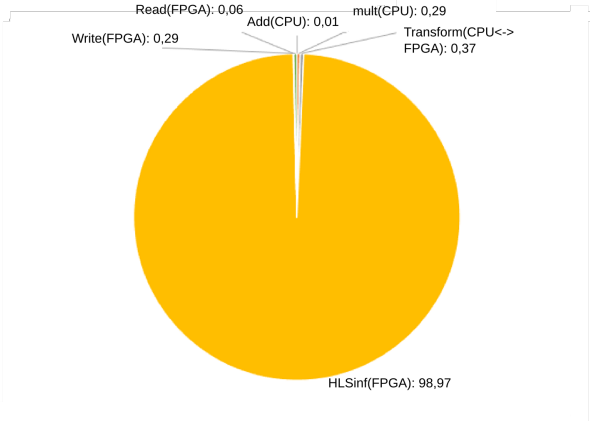

Fig. 5: Descomposición de inferencia por módulos para modelo SegNet mapeado en FPGA.

con la librería EDDL, una librería de código abierto utilizada para el despliegue de modelos de Redes Neuronales en CPUs y GPUs. Además, las técnicas de cuantificación y poda utilizadas para la compresión de modelos han sido integradas en el diseño de la herramienta, explotando así la eficiencia de los recursos de la FPGA y optimizando el rendimiento en comparación con la CPU. Los resultados obtenidos demuestran la eficacia del enfoque y sugieren nuevos despliegues y soporte en la plataforma para estrategias combinadas de poda y cuantificación.

Agradecimientos

Este trabajo ha recibido financiación de la Unión Europea Horizon2020 bajo el acuerdo de las subvenciones 871467 y 825111 así como de la beca FPU "Programa propio de la Universitat Politècnica de València - Subprograma 1 (PAID-01-20)" con número 20210037.

Referencias

[1] Xiaofan Zhang, Anand Ramachandran, Chuanhao Zhuge, Di He, Wei Zuo, Zuofu Cheng, Kyle Rupnow, and Deming Chen, "Machine learning on fpgas to face the iot revolution," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 894–901, 10.1109/ICCAD.2017.8203875.

[2] Griffin Lacey, Graham W. Taylor, and Shawki Areibi, "Deep learning on fpgas: Past, present, and future," *CoRR*, 2016, http://arxiv.org/abs/1602.04283.

[3] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, 2017, pp. 1–6, DOI: 10.1109/ICEngTechnol.2017.8308186.

[4] Rachana Patel and Sanskruti Patel, "A comprehensive study of applying convolutional neural network for computer vision," *International Journal of Advanced Science and Technology*, vol. 6, pp. 2161–2174, 01 2020.

[5] Karen Simonyan and Andrew Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun, Eds., 2015.

[6] Akhil Agnihotri, Prathamesh Saraf, and Kriti Rajesh Bapnad, "A convolutional neural network approach towards self-driving cars," *CoRR*, vol. abs/1909.03854, 2019.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[8] Karen Simonyan and Andrew Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[9] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, FPGA '17, pp. 65–74, ACM.

[10] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[11] "Dpu for convolutional neural network v3.0—dpu ip product guide (pg338)," Xilinx, San Jose, CA, USA, 2019.

[12] European Distributed Deep Learning Library, ," `https://deephealthproject.github.io/eddl/`, 2021.

[13] Andrew Lavin and Scott Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4013–4021.

[14] Francois Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[15] Junjie Bai, Fang Lu, Ke Zhang, et al., "Onnx: Open neural network exchange," `https://github.com/onnx/onnx`, 2019.

[16] Noel Codella, Veronica Rotemberg, Philipp Tschandl, M Emre Celebi, Stephen Dusza, David Gutman, Brian Helba, Aadi Kalloo, Konstantinos Liopyris, Michael Marchetti, et al., "Skin lesion analysis toward melanoma detection 2018: A challenge hosted by the international skin imaging collaboration (isic)," *arXiv preprint arXiv:1902.03368*, 2019.

[17] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 12, pp. 2481–2495, 2017.

[18] Andrea Bragagnolo, Enzo Tartaglione, Attilio Fiandrotti, and Marco Grangetto, "On the role of structured pruning for neural network compression," in *2021 IEEE International Conference on Image Processing (ICIP)*, 2021, pp. 3527–3531.

[19] Enzo Tartaglione, Andrea Bragagnolo, Francesco Odierna, Attilio Fiandrotti, and Marco Grangetto, "Serene: Sensitivity-based regularization of neurons for structured sparsity in neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–14, 2021.

[20] Andrea Bragagnolo and Carlo Alberto Barbano, "Simplify: A python library for optimizing pruned neural networks," *SoftwareX*, vol. 17, pp. 100907, 2022.

[21] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer, "A survey of quantization methods for efficient neural network inference," 2021.

[22] ," `https://github.com/CEA-LIST/N2D2`, 2019.

[23] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling, "Data-free quantization through weight equalization and bias correction," 2019.