Ingeniería
ISSN: 0121-750X
ISSN: 2344-8393
Universidad Distrital Francisco José de Caldas

Guerra-Londono, Mateo; Castano-Londono, Luis; Alzate-Anzola, Cristian; Marquez-Viloria, David; Velasquez-Velez, Ricardo

Análisis de desempeño de capas de CNN para arquitecturas heterogéneas basadas en FPGAs usando HLS

Ingeniería, vol. 26, núm. 1, 2021, Enero-Abril, pp. 62-76
Universidad Distrital Francisco José de Caldas

DOI: https://doi.org/10.14483/23448393.15634

Disponible en: https://www.redalyc.org/articulo.oa?id=498868274006

Research

REVISTA Ingeniería

Performance Analysis of CNN Layers for Heterogeneous FPGAs-based Architectures Using HLS

Análisis de desempeño de capas de CNN para arquitecturas heterogéneas basadas en FPGAs usando HLS

Mateo Guerra-Londono [*,1], ***Luis Castano-Londono*** [1], ***Cristian Alzate-Anzola*** [1], ***David Marquez-Viloria*** [1], ***Ricardo Velasquez-Velez*** [2]

[1]Automática Electrónica y Ciencias Computacionales, Facultad de Ingenierías, Instituto Tecnológico Metropolitano (Medellín, Colombia), [2]Sistemas Embebidos e Inteligencia Computacional, Facultad de Ingeniería, Universidad de Antioquia UdeA, Calle 70 No. 52-21, Medellín, Colombia

[*]Correo de correspondencia: mateoguerra251449@correo.itm.edu.co

Received: 02-12-2019. Modified: 09-11-2020. Accepted: 21-11-2020

Cite this paper as: M. Guerra-Londono, L. Castano-Londono, C. Alzate-Anzola, D. Marquez-Viloria y R. Velasquez-Velez, "Análisis de desempeño de capas de CNN para arquitecturas heterogéneas basadas en FPGAs usando HLS", Revista Ingeniería, vol. 26, n.° 1, pp. 62-76 (2021).

DOI: https://doi.org/10.14483/23448393.15634

Abstract

Context: Convolutional neural networks (CNNs) are currently used in a wide range of artificial intelligence applications. In many cases, these applications require the execution of the networks in real time on embedded devices. Hence the interest in these applications achieving excellent performance with low power consumption. CNNs perform operations between the input data and the network weights, with the particularity that there is no dependence between most of the operations. Thus, the inherent parallelism of Field Programmable Gate Arrays (FPGAs) can be used to perform multiple operations in parallel, maintaining the good performance per watt that characterizes these devices. This paper focuses on evaluating the convolution algorithm for a convolutional layer of neural networks by exploring parallelization directives using VIVADO HLS, and it aims to evaluate the performance of the algorithm using optimization directives.

Method: The methodology consists of an exploration of the design space of a convolutional neural network layer implementation using VIVADO HLS. Performance verification of the FPGA was performed by comparing the output data with the same convolution algorithm implemented in MATLAB. A layer of the commercial version Xilinx DNNK was used as a reference for performance measurements of the different implementations obtained during the exploration of the design space. In this work, multiple variations of optimization directives are used, such as pipeline, array partition and unroll.

Results: This paper presents the results of a reference implementation (without optimization directives) of the convolution algorithm concerning algorithm latency and the hardware resources used by the FPGA. The results are compared with the implementations of the algorithm, including different combinations of two optimization directives (pipeline and partition array).

Conclusions: This work explores the design space of a convolution algorithm for a convolutional neural network layer on FPGAs. The exploration includes the effect of data transfer between DDR memory and the on-chip memory of the FPGA. Also, said effect is caused by the optimization directives in VIVADO HLS on the different cycles of the algorithm.

Keywords: Convolution, convolutional neural network, FPGA, high-level synthesis, optimization directives.

Acknowledgements: This work was supported in part by the Automática, Electrónica y Ciencias Computacionales Group (COL0053581) - Instituto Tecnológico Metropolitano and in part by Sistemas Embebidos e Inteligencia Computacional Group (COL0010717) - Universidad de Antioquia under Grant P17224.

Language: Spanish.

Resumen

Contexto: Las redes neuronales convolucionales (CNNs) son actualmente utilizadas en una amplia gama de aplicaciones de inteligencia artificial. En muchos casos, dichas aplicaciones requieren la ejecución de las redes en tiempo real en dispositivos integrados. Por esto, el interés en que estas aplicaciones puedan alcanzar un buen desempeño con bajo consumo de potencia. Las CNNs realizan operaciones entre los datos de entrada y los pesos de la red, con la particularidad de que no existe dependencia entre la mayoría de las operaciones. Por tal motivo, el paralelismo inherente de los FPGAs puede ser usado para realizar múltiples operaciones en paralelo, manteniendo el buen desempeño por vatio que caracteriza a estos dispositivos. Este artículo se enfoca en la evaluación del algoritmo de convolución para una capa convolucional de redes neuronales explorando directivas de paralelización usando VIVADO HLS, y su objetivo es evaluar el desempeño del algoritmo utilizando directivas de optimización.

Método: La metodología consiste en una exploración del espacio de diseño de la implementación de una capa de una red neuronal convolucional usando VIVADO HLS. La verificación del funcionamiento del FPGA fue realizada comparando los datos de salida con el mismo algoritmo de convolución implementado en MATLAB. Una capa de la versión comercial Xilinx DNNK fue usada como referencia para las medidas de desempeño de las diferentes implementaciones obtenidas en la exploración del espacio de diseño. En este trabajo se utilizan múltiples variaciones de directivas de optimización, tales como *pipeline, array partition*, y *unroll*.

Resultados: Este trabajo presenta los resultados de una implementación de referencia (sin directivas de optimización) del algoritmo de convolución con respecto a la latencia del algoritmo y los recursos de hardware utilizados por la FPGA. Los resultados se comparan con implementaciones del algoritmo, incluyendo diferentes combinaciones de dos directivas de optimización (*pipeline* y *partition array*).

Conclusiones: Este trabajo explora el espacio de diseño de un algoritmo de convolución para una capa de red neuronal convolucional sobre FPGAs. La exploración incluye el efecto causado por la transferencia de los datos entre la memoria DDR y la memoria *on-chip* del FPGA. Además, dicho efecto es causado por las directivas de optimización en Vivado HLS sobre los diferentes ciclos del algoritmo.

Palabras clave: Convolución, directivas de optimización, FPGA, red neuronal convolucional, síntesis de alto nivel.

Agradecimientos: Este trabajo se enmarca dentro del proyecto ”Mejora de la percepción visual en robots humanoides para el reconocimiento de objetos en entornos naturales mediante aprendizaje profundo” con código P17224, cofinanciado por el Instituto Tecnológico Metropolitano y la Universidad de Antioquia. Mateo Guerra agradece al programa “Jóvenes Investigadores e Innovadores ITM” del Instituto Tecnológico Metropolitano (ITM).

Idioma: Español

1. Introducción

En años recientes, las redes neuronales convolucionales (CNN, por sus siglas en inglés de *Convolutional Neural Network*) han sido utilizadas en aplicaciones de inteligencia artificial para el reconocimiento de voz, identificación de patrones, detección, seguimiento de objetos, etc. Debido a que las CNN requieren altas capacidades de procesamiento, la utilización de FPGA (por sus siglas en inglés de *Field Programmable Gate Arrays*) para su implementación ha atraído mucha atención porque estos dispositivos son reconfigurables, aprovechan el paralelismo en las CNN y son efi-

cientes en consumo de potencia [6].

El diseño de algoritmos sobre FPGAs se ha realizado tradicionalmente usando metodología RTL (por sus siglas en inglés de *Register-Transfer Level*) usando lenguajes de descripción de *hardware* (HDL, por sus siglas en inglés de *Hardware Description Language*) como VHDL o Verilog. Estos lenguajes requieren experiencia en diseño digital de *hardware* aumentando la complejidad del flujo de diseño. Por esto, en los últimos años se viene popularizando el diseño de *hardware* a nivel algorítmico por medio de herramientas de síntesis de alto nivel (HLS, por sus siglas en inglés de *High-level synthesis*). Estas herramientas permiten describir las arquitecturas de *hardware* usando lenguajes de programación de alto nivel como C/C++ o Python, disminuyendo considerablemente la complejidad del flujo de diseño y proporcionando la posibilidad de paralelización del algoritmo a través de directivas de optimización [1].

Los FPGA se han utilizado para la aceleración de las operaciones realizadas en las capas de la red por medio de diferentes técnicas de paralelización [11]. Algunos trabajos sobre la implementación de CNN en FPGA pueden encontrarse en [12], [15], [4], [13]. La paralelización de las capas convolucionales en FPGA se ha realizado utilizando metodologías de diseño convencional RTL como en [5], [9], [3], [10], y herramientas de síntesis de alto nivel (HLS) como en [13].

Las herramientas de diseño HLS permiten al programador utilizar directivas de optimización como *loop unrolling*, *array partition* y *pipeline* para lograr un rendimiento similar o incluso mejor en comparación con el diseño RTL; además, el diseñador no necesitará tener mucha experiencia en diseño de *hardware*. En [13] se utilizó Vivado HLS de Xilinx con el fin de sintetizar un acelerador para la operación de convolución, obteniendo un rendimiento máximo de 61 GFLOPS. Los autores diseñaron un acelerador CNN con OpenCL, logrando un rendimiento máximo de 136 GOPS para la capa convolucional. En [2], los autores emplearon una FPGA Arria 10 de Intel para sintetizar un acelerador CNN con OpenCL, logrando más de 1 TFLOPS. En [7] realizaron un acelerador de CNN basado en FPGA a partir de la herramienta HLS, incorporaron una arquitectura de desplazamiento de peso igual a cero y reducción de la aritmética.

Este trabajo se enfoca en la evaluación del algoritmo de convolución para una capa convolucional de redes neuronales a través de la exploración de directivas de paralelización usando Vivado HLS, y el desempeño del algoritmo es evaluado para diferentes directivas de optimización. También se explora el efecto de la transmisión de los datos entre la memoria DDR y la memoria *on-chip*. La implementación del algoritmo fue realizada en un sistema de desarrollo Ultra96 que contiene un Zynq UltraScale + MPSoC, el cual es una arquitectura heterogénea basada en FPGA. La precisión de los resultados numéricos obtenidos con la implementación en FPGA utilizando Vivado HLS son comparados con resultados obtenidos en MATLAB. Se evalúa la variación de la latencia en términos de ciclos de reloj para la primera capa convolucional de una CNN entrenada en Keras, según la forma en que se configuran algunas directivas de optimización para los diferentes arreglos y ciclos utilizados en la implementación del algoritmo. Como resultado se obtiene un bloque convolucional adaptable que puede ser utilizado para la construcción de diferentes topologías de red. Se realiza el análisis del desempeño teniendo en cuenta la latencia y los recursos de *hardware*.

2. Materiales y métodos

2.1. Sistema de desarrollo y flujo de diseño

Para este estudio se utilizó una tarjeta de desarrollo Ultra96-V1. Esta placa cuenta con un dispositivo MPSoC (por sus siglas en inglés de *Multi-Processor System-on-Chip*) que contiene un ARM Cortex-A53 de cuatro núcleos y 154K celdas de lógica programable dentro de un chip. La síntesis se realizó en Vivado HLS versión 2019.1. Esta herramienta es la encargada de convertir el código descrito en C/C++ a HDL (Figura 1).

Los pasos realizados en cada etapa de diseño se presentan en la Figura 2. La primera etapa consiste en el diseño del *software*, el cual se divide en dos pasos. En el primer paso se recomienda realizar la implementación del algoritmo de convolución en MATLAB, con el propósito de tomar los resultados obtenidos como base para la comparación con los obtenidos en Vivado HLS. En el otro paso se utiliza directamente la implementación del algoritmo en C/C++ en la CPU para obtener un código similar al de la implementación final en HLS. Los autores usan los dos pasos de la etapa de diseño de *software*.

En la etapa de implementación de *hardware* se proponen dos pasos. El primer paso es la aceleración del algoritmo de convolución usando Vivado HLS. En este paso, las directivas de paralelización se aplican al código para mejorar el rendimiento del algoritmo. En el segundo paso, el código HDL generado por Vivado HLS se sintetiza y se descarga en el FPGA.

2.2. Base de datos y arquitectura convolucional

La base de datos utilizada es la MNIST [8], la cual es una gran base de datos de dígitos (0 a 9). Contiene 60.000 imágenes de entrenamiento y 10.000 imágenes de prueba. Estas imágenes tienen un tamaño de 28x28 y están en blanco y negro. MNIST es posiblemente el conjunto de

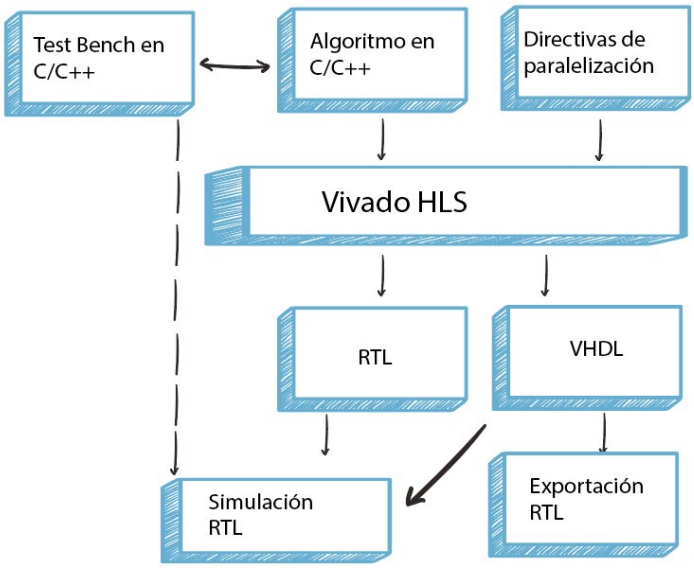


Figura 1. Flujo de diseño usando la herramienta de síntesis de alto nivel Vivado HLS.

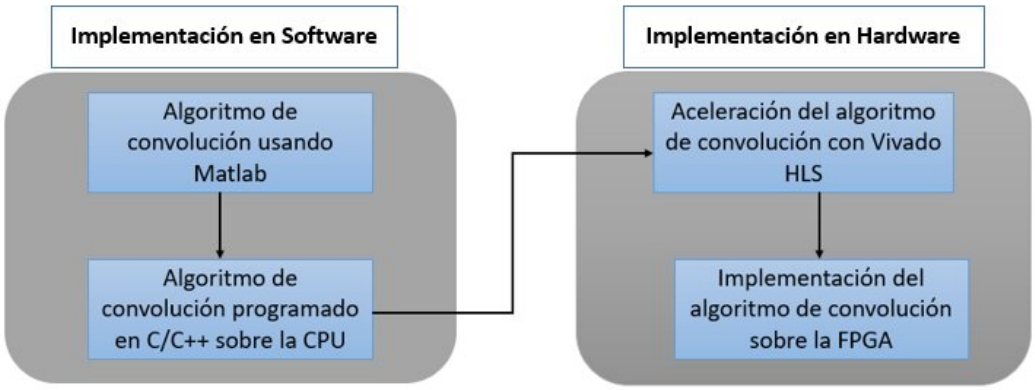


Figura 2. Metodología de diseño.

datos mejor estudiado y más comprendido en la literatura sobre visión artificial y aprendizaje automático. La arquitectura convolucional elegida consta de dos capas convolucionales seguidas de una capa *maxpooling* y conectadas a una capa totalmente conectada con su respectiva capa *softmax*. La arquitectura se observa más detalladamente en la Figura 3. El entrenamiento de la red convolucional es realizado con el *framework* Keras, el cual permite un fácil entrenamiento de estas redes usando programación en Python. La función de costo a optimizar fue la *cross-entropy* y se utilizó AdaDelta, que es una técnica de optimización del gradiente descendiente. La exactitud obtenida después del entrenamiento en el conjunto de prueba fue del 98,8 %.

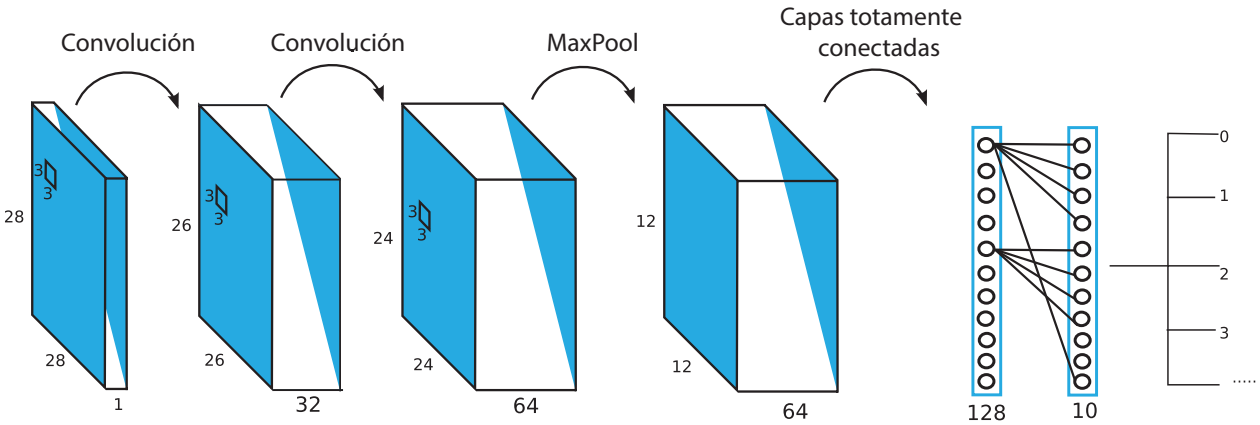


Figura 3. Arquitectura convolucional.

2.3. Implementación secuencial del algoritmo de la capa convolucional

La implementación del algoritmo fue basada en la primera capa de la red convolucional de la Figura 3 y una imagen en el conjunto de datos MNIST. El algoritmo 1 muestra el pseudocódigo para la implementación de la convolución. La síntesis del diseño fue realizada usando Vivado HLS y los resultados obtenidos fueron comparados con los datos adquiridos al realizar la misma implementación del algoritmo en MATLAB, con el objetivo de validar la correcta implementación del algoritmo.

Algoritmo 1: Descripción del algoritmo de convolución

```
1  Entradas:imagen, filtros, bias
2  Salidas:resultado de la convolución
3  loop_conv2d_label0: for k ← 0 to k < 32 do
4      loop_conv2d_label1: for j ← 0 to j < 26 do
5          loop_conv2d_label2: for i ← 0 to i < 26 do
6              Z0 = b0[k];
7              loop_conv2d_label3: for n ← 0 to n < 3 do
8                  loop_conv2d_label4: for m ← 0 to m < 3 do
9                      Z0 = Z0 + X0[(n + j) * 28 + (m + i)] * W0[n * 3 + m + k * 9];
10                 end
11             end
12             Y[j * 26 + i + k * 676] = (Z0 < 0)?0 : Z0;
13         end
14     end
15 end
```

2.4. Exploración del espacio de diseño

La implementación de la arquitectura base es realizada como se muestra en el diagrama de bloques en la Figura 4. Los datos correspondientes a la imagen son almacenados en la memoria RAM DDR3. Los datos de la imagen y el resultado de la convolución son transmitidos entre el sistema de procesamiento (PS) Zynq-Ultra y el núcleo de convolución utilizando dos puertos de alto desempeño AXI HP. Los valores de los *pesos* y *bias* son almacenados en memorias BRAM de doble puerto, las cuales pueden ser accedidas desde PS o el núcleo de convolución.

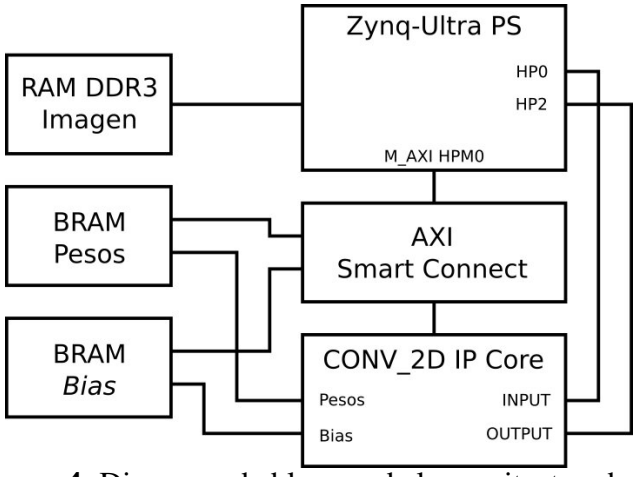
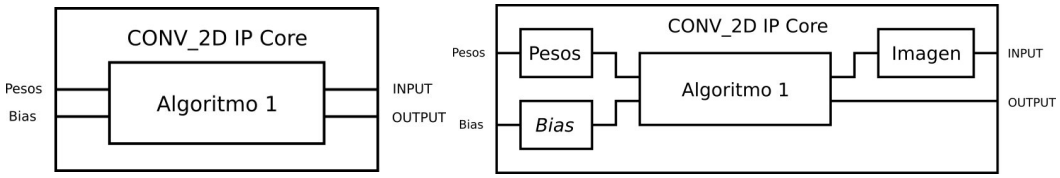


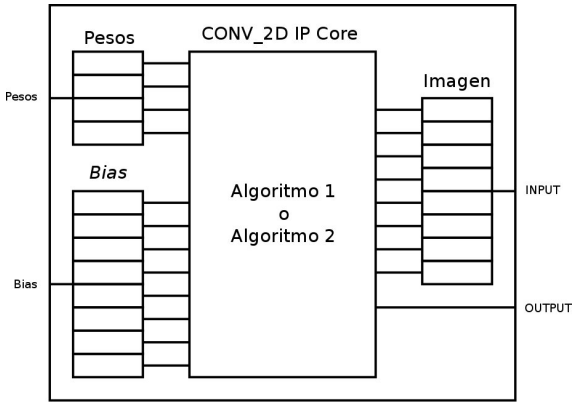
Figura 4. Diagrama de bloques de la arquitectura base.

La exploración del espacio de diseño se realiza para cuatro esquemas diferentes del núcleo de convolución, en los cuales se varía el origen de los datos que intervienen en las operaciones. El primer esquema, que se muestra en la Figura 5a, corresponde a la implementación base. En el segundo esquema, que se muestra en la Figura 5b, todos los datos de entrada son copiados en arreglos auxiliares definidos en el código del núcleo de convolución, de forma que todas las operaciones se

realizan con datos *on-chip*. En el tercer esquema, que se muestra en la Figura 5c, se aplica la directiva de partición de memoria a los arreglos auxiliares. En el cuarto esquema se realiza adicionalmente un desenrollado manual, realizando una modificación en la implementación como se muestra en el algoritmo 2. Para todos los esquemas, los resultados son escritos en memoria DDR3.



(a) Esquema 1 **(b)** Esquema 2



(c) Esquema 3/Esquema 4

Figura 5. Esquemas.

Algoritmo 2: Descripción del algoritmo de convolución con desenrollado manual

Entradas: **imagen, filtros, bias**
Salidas: **resultado de la convolución**
loop_conv2d_label0: **for** $k \leftarrow 0$ **to** $k < 16$ **do**
  loop_conv2d_label1: **for** $j \leftarrow 0$ **to** $j < 26$ **do**
    loop_conv2d_label2: **for** $i \leftarrow 0$ **to** $i < 26$ **do**
      $Z0 = b0[k];$
      $Z1 = b0[k + 16];$
      loop_conv2d_label3: **for** $n \leftarrow 0$ **to** $n < 3$ **do**
        loop_conv2d_label4: **for** $m \leftarrow 0$ **to** $m < 3$ **do**
          $Z0 = Z0 + X0[(n + j) * 28 + (m + i)] * W0[n * 3 + m + k * 9];$
          $Z1 = Z1 + X0[(n + j) * 28 + (m + i)] * W0[n * 3 + m + (k + 16) * 9];$
        end
      end
      $Y[j * 26 + i + k * 676] = (Z0 < 0)?0 : Z0;$
      $Y[j * 26 + i + (k + 16) * 676] = (Z1 < 0)?0 : Z1;$
    end
  end
end

Para cada esquema se evalúan siete implementaciones utilizando las directivas de optimización como se muestra en la Tabla I. Para la aceleración del algoritmo usando Vivado HLS se aplicaron directivas de optimización a los diferentes ciclos que construyen el algoritmo de convolución. Las siguientes son las diferentes directivas usadas: *Baseline* es el algoritmo implementado sin directivas de optimización, *Pipeline* es una directiva que reduce el intervalo de inicio del bucle al permitir la ejecución concurrente de las operaciones, *Unroll* es la directiva que desenrolla el ciclo total o parcialmente causando que las iteraciones se realicen en paralelo, *Array partition* separa los arreglos que contienen los datos de entrada y salida del algoritmo en distintas posiciones de memoria para ser procesados en paralelo.

Tabla I. Implementaciones realizadas para la exploración del espacio de diseño

Implementación	**Directiva de optimización**
Baseline	Ninguna
Optimización 1	Pipeline en loop_conv2d_label1
Optimización 2	Pipeline en loop_conv2d_label2
Optimización 3	Pipeline en loop_conv2d_label3
Optimización 4	Pipeline en loop_conv2d_label4
Optimización 5	Pipeline + Unroll en loop_conv2d_label3
Optimización 6	Pipeline + Unroll en loop_conv2d_label4

2.5. Comparación de la capa convolucional con framework de alto nivel

La aparición de metodologías de alto nivel ha impulsado el desarrollo de múltiples *frameworks* de aprendizaje profundo para FPGA. Uno de estos *frameworks* es DNNDK (por sus siglas en inglés de *Deep Neural Network Development Kit*), el cual está diseñado para reducir el consumo de energía y para facilitar la implementación de algoritmos de aprendizaje profundo en FPGA [14]. DNNDK implementa un módulo llamado DPU (por sus siglas en inglés de *Data Processing Unit*) que brinda altas prestaciones computacionales y facilita su uso en plataformas heterogéneas. Con una interfaz unificada, DPU puede ser implementado fácilmente en cualquier familia de FPGA. Las DPU son desarrolladas en C/C++ y son comparables con CUDA/OpenCL, debido a que son hechas para fácil uso. Así que DNNDK como optimización reduce el tamaño de las redes convolucionales debido a que usualmente contienen información redundante que al eliminarse se obtiene una optimización en términos de eficiencia computacional, eficiencia energética y menos memoria para el sistema, especialmente en el ancho de banda desde el *host* a la FPGA. Adicionalmente, este *framework* aplica técnicas de transformación y compilación optimizada como lo son la fusión de nodos de computación, planeación eficiente de instrucciones y el uso de características y pesos en la memoria *on-chip*. Con el objetivo de comparar estas optimizaciones con las aceleraciones propuestas anteriormente se implementó la primera capa de la arquitectura convolucional en el *framework* DNNDK.

2.6. Evaluación de desempeño

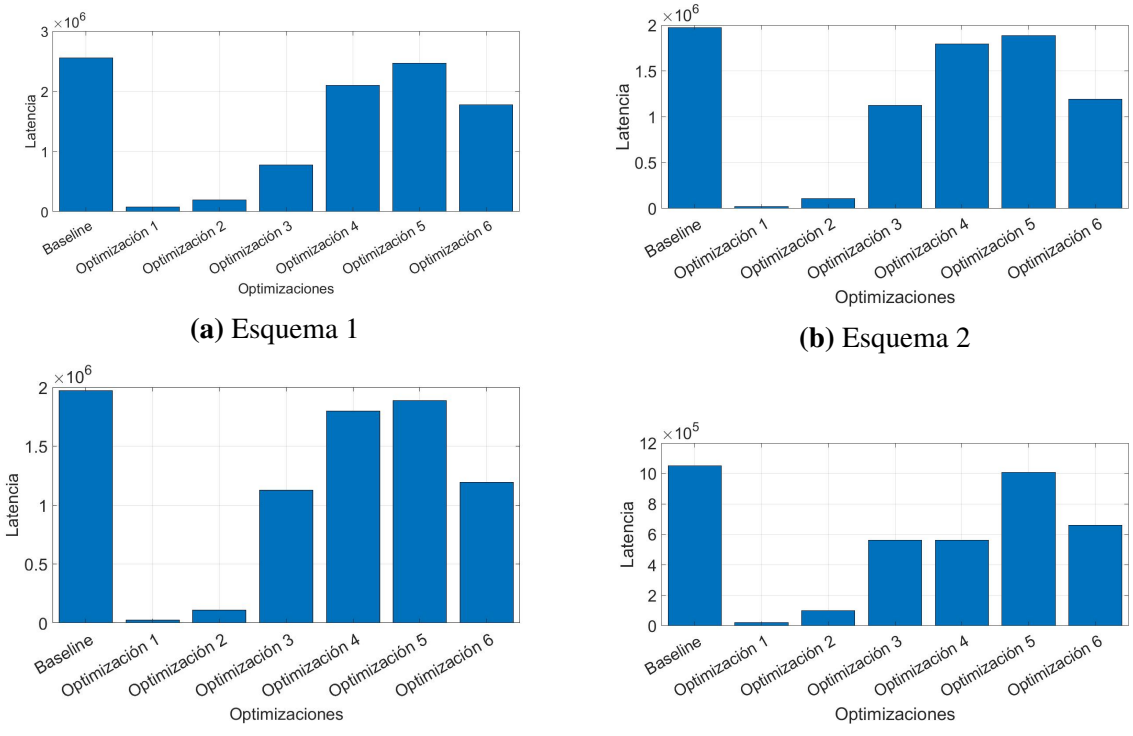
En esta etapa se analizó cada reporte de síntesis generado por Vivado HLS para cada uno de los cuatro esquemas de la exploración del espacio de diseño. Con el objetivo de observar el desempeño

en la implementación del algoritmo para los diferentes esquemas se extrajeron los datos de latencia y recursos de *hardware*. Por último, los resultados obtenidos con el *framework* DNNDK se evalúan para obtener así una tabla comparativa entre las aceleraciones propuestas y el resultado del *framework* DNNDK en términos de tiempo.

3. Resultados

La exploración del espacio de diseño se realizó para el Zynq UltraScale+MPSoC del sistema de desarrollo Ultra96. Los datos del proceso de convolución adquiridos en MATLAB y en Vivado HLS fueron comparados para verificar el resultado del algoritmo. El error máximo de la comparación es de $4,9659 \times 10^{-07}$ y el error cuadrático medio entre las dos imágenes es de $2,8813 \times 10^{-14}$.

La Figura 6 muestra la latencia expresada en ciclos de reloj para los diferentes esquemas planteados. Los valores correspondientes a la implementación *Baseline* son los resultados obtenidos para la implementación del algoritmo de convolución sin directivas de optimización. Los demás valores corresponden a la latencia para las diferentes soluciones obtenidas al aplicar la directiva *pipeline*, *unroll* y *array partition* sobre cada ciclo como se presenta en la Tabla I. La optimización 1 presenta la menor latencia en todos los esquemas propuestos, incluso en el esquema donde se trabaja con la memoria externa DDR y la optimización 2 ocupa el segundo lugar en latencia. El peor desempeño en todos los esquemas se obtiene con la optimización 5 con latencias cercanas al *Baseline* que no posee directivas de optimización.



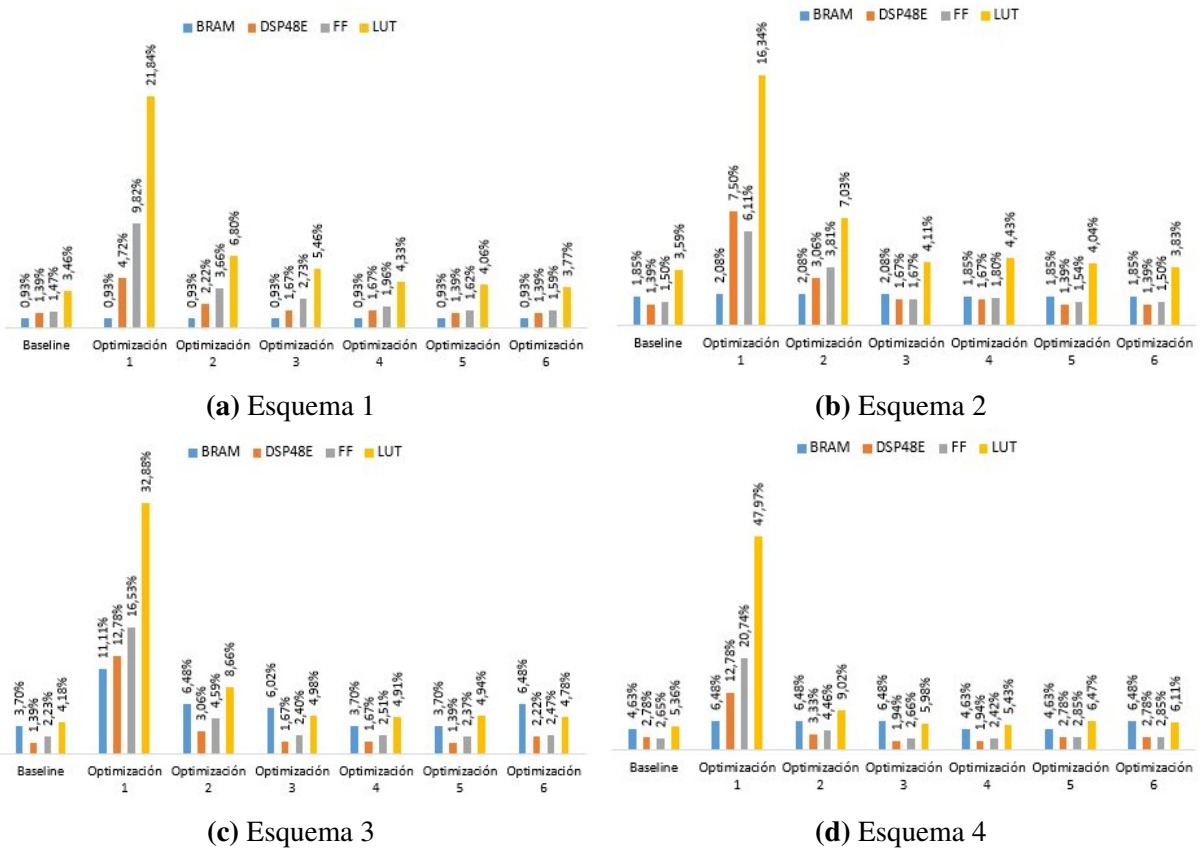
(a) Esquema 1

(b) Esquema 2

(c) Esquema 3

(d) Esquema 4

Figura 6. Latencia en ciclos de reloj de los esquemas planteados con diferentes directivas de optimización.



(a) Esquema 1 **(b)** Esquema 2

(c) Esquema 3 **(d)** Esquema 4

Figura 7. Consumo de recursos de *hardware* (BRAM,DSP48E, FF, LUT) para los esquemas con las diferentes directivas de optimización.

La cantidad de recursos de *hardware* utilizados en la implementación del algoritmo para cada uno de los esquemas de exploración del espacio de diseño se muestra en la Figura 7, donde para cada esquema se presenta la cantidad en porcentaje de los recursos utilizados: BRAM (por sus siglas en inglés de *Block RAM*), DSP48E (por sus siglas en inglés de *Digital Signal Processors*), FF (por sus siglas en inglés de *Flip-Flops*) y LUT (por sus siglas en inglés de *LookUp Table*). En los cuatro esquemas planteados, la optimización 1 emplea la mayor cantidad de recursos de *hardware*, esto se debe a que el algoritmo contiene ciclos anidados y la directiva es aplicada al ciclo externo, generando una mayor cantidad de *hardware* en el procesamiento de datos y unidades de control. Para la optimización 2, la cantidad de recursos de *hardware* usados equivalen a menos de la mitad de los recursos de *hardware* empleados para la optimización 1, lo que la convierte en una buena alternativa de implementación debido a que ocupa el segundo lugar en latencia.

La implementación *baseline* en cada uno de los esquemas es la que presenta el menor número de recursos utilizados, ya que al no aplicar directivas de optimización consume una menor cantidad de recursos, pues cualquier mejora en el algoritmo con directivas de optimización conlleva a un aumento de recursos de *hardware*. La Tabla II muestra la aceleración obtenida por las diferentes optimizaciones aplicadas a cada esquema con respecto a la implementación base sin directivas (esquema 1, *baseline*). En la Tabla II se observa una aceleración de más de 111 veces en los esquemas 3 y 4 usando la optimización 1. En la Tabla III se presenta la aceleración obtenida por

los esquemas 2, 3 y 4 con relación al esquema 1, comparando individualmente cada optimización. La Tabla III muestra que el esquema 3 tiene la menor latencia de todas las implementaciones realizadas en la exploración del espacio de diseño, debido a que presenta el *speed-up* más alto en la optimización 1, que es la optimización con mejor desempeño para todos los esquemas. Sin embargo, no existe una diferencia significativa con el *speed-up* presentado por el esquema 4 para la misma optimización. Adicionalmente, el esquema 4 presenta el *speed-up* más alto en el resto de los casos.

Tabla II. Speed-up con relación al *baseline* del esquema 1

	Speed-up			
	Esquema 1	**Esquema 2**	**Esquema 3**	**Esquema 4**
Baseline	1,000	1,281	1,296	2,430
Optimización 1	31,961	62,177	111,915	111,534
Optimización 2	13,117	23,366	23,366	25,933
Optimización 3	3,280	2,269	2,269	4,533
Optimización 4	1,217	1,405	1,422	4,533
Optimización 5	1,035	1,340	1,355	2,535
Optimización 6	1,439	2,103	2,142	3,860

Tabla III. Speed-up respecto al esquema 1 de la exploración del espacio de diseño

	Speed-up		
	Esquema 2	**Esquema 3**	**Esquema 4**
Baseline	1,281	1,296	2,430
Optimización 1	1,945	3,502	3,490
Optimización 2	1,781	1,781	1,977
Optimización 3	0,692	0,692	1,382
Optimización 4	1,154	1,168	3,723
Optimización 5	1,294	1,309	2,449
Optimización 6	1,462	1,489	2,683

Los resultados obtenidos en el *framework* DNNDK se observan en la Tabla IV y son comparados con el esquema 4 - optimización 1, que es el esquema de mejor desempeño. El tiempo presentado por la Tabla IV está en milisegundos (ms) y se refiere al tiempo de cómputo total, es decir, tiempo del kernel y tiempo de comunicación de los datos. El DNNDK supera en 2.3X el desempeño del esquema 4 en cuanto a tiempo total de procesamiento. Sin embargo, el esquema 4 está implementado con formato numérico de 32 bits punto flotante, mientras que el *framework* DNNDK reduce a 8 bits la representación de los pesos. En cuanto a recursos de *hardware*, el esquema 4 aprovecha mejor los recursos que el *framework* DNNDK. La tabla presenta que el DNNDK consume 1.09X LUT, 2.45X FF, 5.77X BRAM, 6.3X DSP comparado con el esquema 4.

Tabla IV. Comparación de la implementación propuesta y el *framework* DNNDK para la misma capa convolucional

	Esquema 4 - optimización 1	**DNNDK**
Tiempo(ms)	0,234	0,101
LUT	33.851	37.055
FF	29.266	72.850
Block RAM	28	161,5
DSP	46	290

4. Conclusiones

Este trabajo presenta una exploración del espacio de diseño de una capa de una CNN y un análisis de desempeño en cuanto a latencia y consumo de recursos de *hardware*. La exploración presenta cuatro esquemas distintos y la aplicación de directivas de optimización en Vivado HLS para el algoritmo de convolución sobre arquitecturas heterogéneas basadas en FPGA. En la exploración se encontró que los esquemas presentan un mejor desempeño al aplicar la optimización 1 propuesta. El esquema 4 presenta la mejor latencia para todas las optimizaciones aplicadas. Adicionalmente, La optimización 2 ocupa el segundo lugar en latencia y consume menos de la mitad de los recursos de *hardware* utilizados en la optimización 1 para cada esquema propuesto convirtiéndose en una buena alternativa cuando se tiene pocos recursos de *hardware*. El esquema 4 tiene buen desempeño en tiempo de ejecución comparado con el *framework* DNNDK, teniendo en cuenta que el *framework* reduce a 8 bits la representación de los pesos y el esquema usa 32 bits punto flotante. También el esquema hace un mejor uso de recursos de *hardware* en contraste con el DNNDK que consume 1.09X LUT, 2.45X FF, 5.77X BRAM, 6.3X DSP comparado con el esquema 4.

Como trabajos futuros se planea realizar la implementación de la CNN completa y otras construcciones de diferentes topologías de red, teniendo en cuenta el bloque convolucional adaptable encontrado en el esquema 4 con las optimizaciones 1 y 2 en la exploración del espacio de diseño. También una exploración en cuanto a formatos numéricos de punto fijo con el fin de mejorar los tiempos de cómputo.

5. Agradecimientos

Este estudio fue apoyado por el grupo de investigación Automática, Electrónica y Ciencias Computacionales (AE&CC - COL0053581), en el Laboratorio de Sistemas de Control y Robótica, adscrito al Instituto Tecnológico Metropolitano y el grupo de investigación Sistemas Embebidos e Inteligencia Computacional (SISTEMIC - COL0010717) de la Universidad de Antioquia. Este trabajo se enmarca dentro del proyecto "Mejoramiento de la percepción visual en robots humanoides para el reconocimiento de objetos en entornos naturales mediante aprendizaje profundo" con código P17224, cofinanciado por el Instituto Tecnológico Metropolitano y la Universidad de Antioquia.

Mateo Guerra agradece al programa "Jóvenes Investigadores e Innovadores ITM" del Instituto Tecnológico Metropolitano (ITM).

Referencias

[1] D. Aledo, B. Carrion, y F. Moreno, "VHDL vs. SysteMC: Design of highly parameterizable artificial neural networks", *IEICE Transactions on Information and Systems*, E102D(3):512–521, 2019. https://doi.org/10.1587/transinf.2018EDP7142 ↑ 65

[2] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, y G. R. Chiu, "An OpenCLTM deep learning accelerator on Arria 10", en *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 55–64, 2017. https://doi.org/10.1145/3020078.3021738 ↑ 65

[3] L. Bai, Y. Zhao, y X. Huang, "A CNN Accelerator on FPGA Using Depthwise Separable Convolution", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, n.°10, pp. 1415–1419, 2018. https://doi.org/10.1109/TCSII.2018.2865896 ↑ 65

[4] S. Chakradhar, M. Sankaradas, V. Jakkula y S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks", en *Proceedings - International Symposium on Computer Architecture*, pp.247–257, 2010. https://dl.acm.org/doi/10.1145/1816038.1815993 ↑ 65

[5] W. Ding, Z. Huang, Z. Huang, L. Tian, H. Wang y S. Feng, "Designing efficient accelerator of depthwise separable convolutional neural network on fpga", *Journal of Systems Architecture*, vol.97, pp.278-286, 2019. https://doi.org/10.1016/j.sysarc.2018.12.008 ↑ 65

[6] M. K. Hamdan y D. T. Rover, "VHDL generator for a high performance convolutional neural network FPGA-based accelerator", en *2017 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2017*, Cancun, diciembre 2017. https://doi.org/10.1109/RECONFIG.2017.8279827 ↑ 65

[7] J. H. Kim, B. Grady, R. Lian, J. Brothers y J. H. Anderson, "FPGA-based CNN inference accelerator synthesized from multi-threaded C software", *International System on Chip Conference*, Munich, 2017. https://doi.org/10.1109/SOCC.2017.8226056 ↑ 65

[8] Y. LeCun, *et al.*, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, n.° 11, pp.2278–2324, 1998. https://doi.org/10.1109/5.726791 ↑ 66

[9] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou y L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks", *FPL 2016 - 26th International Conference on Field-Programmable Logic and Applications*, Lausanne, 2016. https://doi.org/10.1109/FPL.2016.7577308 ↑ 65

[10] E. Nurvitadhi, *et al.*,"Can FPGAs beat GPUs in accelerating next-generation deep neural networks", en *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, febrero 2017. https://doi.org/10.1145/3020078.3021740 ↑ 65

[11] S. Qiao y J. Ma, "Fpga implementation of face recognition system based on convolution neural network", en *2018 Chinese Automation Congress (CAC)*, Xi'an, noviembre 2018. https://doi.org/10.1109/CAC.2018.8623662 ↑ 65

[12] J. Qiu, *et al.*,"Going deeper with embedded FPGA platform for convolutional neural network", en *FPGA 2016 - Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, febrero 2016. https://doi.org/10.1145/2847263.2847265 ↑ 65

[13] N. Suda, *et al.*,"Throughput-optimized openCL-based FPGA accelerator for large-scale convolutional neural networks", en *FPGA 2016 - Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016. https://doi.org/10.1145/2847263.2847276 ↑ 65

[14] Xilinx and Inc,"DNNDK User Guide", reporte técnico, Xilinx and Inc., 2019. ↑ 70

[15] C. Zhang, "Optimizing FPGA-based Accelerator Design for Deep.pdf", en *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015. https://dl.acm.org/doi/10.1145/2684746.2689060 ↑ 65

Mateo Guerra-Londono

Estudiante del Departamento de Ingeniería Electrónica y Telecomunicaciones del Instituto Tecnológico Metropolitano, Medellín, Colombia. Actualmente es Joven Investigador e Innovador ITM y pertenece al grupo de investigación en Automática Electrónica y Ciencias Computacionales en la misma institución universitaria. Mateo ha adquirido experiencia en el desarrollo de sistemas sobre arquitecturas heterogéneas basadas en FPGA para la aceleración de algoritmos de DSP y en programación sobre herramientas para síntesis de alto nivel.

Luis Castano-Londono

Ingeniero Electrónico y Magíster en Ingeniería-Automatización Industrial de la Universidad Nacional de Colombia Sede Manizales. Actualmente es candidato a Doctor en Ingeniería - Automática en la misma universidad. Se desempeña como docente del Departamento de Electrónica y Telecomunicaciones de la Facultad de Ingeniería del Instituto Tecnológico Metropolitano e investigador del Grupo Automática, Electrónica y Ciencias Computacionales. Su área de actuación es el diseño digital, particularmente el diseño de sistemas embebidos y sistemas heterogéneos basados en FPGA.

David Marquez-Viloria

Ingeniero Electrónico de la Universidad Nacional de Colombia sede Manizales, con énfasis en Control y Procesamiento Digital de Señales. Es Magíster en Ingeniería Electrónica de la Universidad de Puerto Rico campus Mayagüez, con énfasis en Procesamiento Digital de Señales donde adquirió experiencia en aceleración de algoritmos de DSP sobre estructuras de hardware reconfigurables. Cuenta con más de cinco años de experiencia docente y de investigación en temas relacionados a DSP, diseño digital y sistemas embebidos. Actualmente, David es candidato a Doctor en Ingeniería de la Universidad Nacional de Colombia, es docente del ITM y pertenece al grupo de investigación en Automática Electrónica y Ciencias Computacionales del ITM.

Cristian Alzate-Anzola

Ingeniero Electrónico del Instituto Tecnológico Metropolitano (ITM), Medellín, Colombia. Estudiante de la Maestría en Control y Automatización Industrial en el ITM. Pertenece al grupo de investigación en Automática Electrónica y Ciencias Computacionales en la misma institución universitaria. Su área de actuación es el diseño digital, particularmente el diseño de sistemas embebidos y sistemas heterogéneos basados en FPGA.

Ricardo Velasquez-Velez

Ingeniero calificado en las áreas de Ingeniería Electrónica y Ciencias de la Computación, con experiencia en investigación a nivel nacional en la Universidad de Antioquia y el Instituto Tecnológico Metropolitano. A nivel internacional ha participado en diferentes grupos de investigación de Alemania, Francia y Suiza. Su trabajo en estos grupos ha sido realizado en las áreas de diseño de sistemas embebidos, modelos de programación paralelos y arquitectura de computadores. Ricardo tiene un doctorado en Informática de la Universidad de Rennes 1, y una Maestría en Diseño de Sistemas Embebidos de la Universidad de la Suiza Italiana. Actualmente Ricardo es profesor asociado en la Universidad de Antioquia.