

Práctica Profesional Supervisada
Reuso Dinámico de Memoria en Convolución 2D Aplicada a
Procesamiento de Imágenes

Casabella Martín, Sulca Sergio, Vignolles Ivan

Escuela de Ingeniería en Computación
Facultad de Ciencias Exactas, Físicas y Naturales

UNIVERSIDAD NACIONAL DE CÓRDOBA

Septiembre 2018

Alumnos	Casabella, Martin Sulca, Sergio Vignolles, Ivan
Tutor	Dr. Ing. Mario Rafael Hueda
Supervisor	Dr. Ing. Ariel Luis Pola
Institución	Fundación Fulgor

Práctica Profesional Supervisada
Reuso Dinámico de Memoria en Convolución 2D Aplicada a
Procesamiento de Imágenes

Casabella Martín, Sulca Sergio, Vignolles Ivan

Escuela de Ingeniería en Computación
Facultad de Ciencias Exactas, Físicas y Naturales
Septiembre, 2018

Abstract

En este trabajo se presenta la arquitectura de hardware para una implementación de una convolución bidimensional en una FPGA Xilinx Artix 7.

Se plantea una solución para cuando no se cuenta con suficiente memoria RAM instanciada para almacenar la imagen completa. Se priorizó no solo la velocidad de procesamiento sino también un uso eficiente de los recursos y la escalabilidad del diseño mediante una estructura modular de forma tal que sea posible añadir tantas operaciones de multiplicación/adición en paralelo como sea necesario sin requerir modificaciones importantes en el diseño.

Con la estructura propuesta se logra un reuso dinámico de memoria, con un incremento lineal en la utilización de la misma en función del nivel de paralelismo.

Índice

1 Introducción	11
1.1 Preprocesamiento	12
1.2 Representación en punto fijo	13
1.3 Flujo de diseño	15
1.3.1 Testing	16
2 Arquitectura del sistema	17
2.1 Flujo de trabajo	17
2.2 Estados y transiciones	17
2.3 Arquitectura del módulo de convolución	19
2.3.1 Storage	20
2.3.2 Control Unit	21
2.3.3 Address Generator Unit (AGU)	22
2.3.4 Multiplier Accumulator Unit (MAC)	23
2.3.5 Memory Management Unit (MMU)	25
3 Implementación y resultados	31
3.1 Especificaciones	31
3.2 Verificación	31
3.3 Utilización de recursos	32
3.4 Throughput	33
4 Conclusión	35

4.1	Características del proyecto	35
4.2	Futuras mejoras	36
4.3	Publicación en CASE-2018	37
5	Apéndice	39
5.1	Proceso de escritura en memoria	39

Listado de Figuras

1-1	Transformaciones del rango dinámico durante el procesamiento de la imagen.	13
1-2	Nivel de error según la resolución en bits.	15
1-3	Degrado de la imagen según la resolución en bits.	15
1-4	Esquemático del flujo de diseño seguido.	16
2-1	Diagrama de estados del sistema.	18
2-2	Diferentes estados y las etapas involucradas	19
2-3	Arquitectura general del sistema	20
2-4	Desplazamiento vertical del kernel sobre la imagen	24
2-5	Estructura donde se realiza el calculo dentro de una unidad MAC.	24
2-6	Comparación entre abordajes para una imagen de $1600 \times 1024\text{px}$ y un kernel de 3×3 .	27
2-7	Enrutamiento de información	28
2-8	Estructura de los bloques MMB y PMB.	29
3-1	Comparación de resultados: A la izquierda, la imagen procesada en una CPU usando python y, a la izquierda, la misma imagen procesada con el módulo en la FPGA.	32
3-2	Curva normalizada para la utilización de los recursos de BRAM.	34
4-1	Certificado por la publicación del proyecto llevado a cabo.	37
5-1	Algoritmo de escritura para $N=4$, $k=3$	40

5-2 Desplazamiento del kernel para N=2, k=3	40
5-3 Escritura y lectura de memorias para $N = 2, k = 3$	41

Lista de Tablas

2.1	Código de instrucciones, la letra “d” representa bit de datos.	21
2.2	Codificación de estados globales con las de señales SoP y EoP.	27
3.1	Utilización de recursos con y sin DSP. .	33
3.2	Throughput obtenido en lo que respecta a la convolución.	33

Introducción

El filtrado y procesamiento de imágenes es utilizado en múltiples campos, como la medicina, ingeniería, navegación, aeronáutica, entre otros. En el campo de la inteligencia artificial, área emergente que es de mero interés hoy en día, la convolución en 2D es la operación principal que se realiza en una CNN (Red neuronal Convolutacional) [7].

Dada su naturaleza, la operación de convolución en 2-D es la más empleada en los algoritmos de procesamiento de imágenes. El paralelismo inherente de algoritmos basados en la convolución es explotado logrando alta performance en estos sistemas que los utilizan.

La convolucion bidimensional, se define matemáticamente como:

$$G(x, y) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} K(i, j) I(x - i, y - j) \quad (1.1)$$

Donde $I(x, y)$ es una imagen de $(m \times n)$ pixels, $K(i, j)$ un conjunto de coeficientes denominado kernel, de tamaño $(k \times k)$ and $G(x, y)$ es la convolucion resultante de tamaño $(m - 2 \times n - 2)$ pues se considera una *valid convolution* [10].

El uso de Field Programmable Gate Arrays (FPGAs) para implementar este tipo de sistemas, se debe a la ventaja que presentan estos dispositivos en lo que concierne al paralelismo a nivel bit, pixel, vecindad y a nivel tarea, lo incrementa la performance y velocidad de cómputo. [12]

Existen múltiples ejemplos en la literatura de implementaciones de convolución 2D. Como se muestra en [5], la BRAM es uno de los elementos mas usados con el

consumo de energía mas alto. Las arquitecturas que se encuentran se pueden clasificar en dos grupos, aquellas que consideran un almacenamiento total de la imagen [1, 6] y las que consideran un almacenamiento parcial donde la imagen se partitiona en segmentos [11, 8]. El problema que se encuentra en estos esquema es la falta de modularización del procesamiento, lo que dificulta mejorar la frecuencia de trabajo incrementando el paralelismo.

En este trabajo presentamos el concepto del reuso dinámico de BRAM en convolución 2D y la complejidad de su implementación para arquitecturas con paralelismo. Además, proponemos una arquitectura modular que facilita la escalabilidad.

1.1 Preprocesamiento

Los pixeles de la imagen en escala de grises $I(x, y)$ tienen un rango dinámico de valores que se hallan entre $[0, 255]$, y los valores que toma el kernel $K(x, y)$ pueden ser positivos o negativos.

Dado que el sistema desarrollado forma parte de un proyecto de Deep Learning, donde es usual operar en un rango dinámico centrado en cero, se aplicó una transformación denominada expansión dinámica de rango[4] ($\mathcal{D}([.])$), junto con Maximum Norm[9] ($(\mathcal{M}[.])$) para reacomodar los diferentes rangos de valores. Más sobre esto en la sección siguiente.

Para el caso de la imagen, se tiene $\mathcal{D}[I(x, y)] = I'(x, y)$ y un rango resultante de $[0, 1]$. Para el caso del kernel, $\mathcal{M}[K(x, y)] = K'(x, y)$ con un rango $[-1, 1]$, respectivamente. Así, reemplazando en la ecuación 1.1, se tiene

$$G(x, y) = \mathcal{D}^{-1} \left[\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} K'(i, j) I'(x - i, y - j) \right], \quad (1.2)$$

siendo $\mathcal{D}^{-1}[.]$ el cambio de rango dinamico entre $[0, 255]$.

La figura 1-1 muestra el proceso completo de mapeo de datos, se aplica la transformación a la imagen de entrada, luego se convoluciona con el kernel y se hace la transformación inversa a ese resultado para volverlo a mapear al conjunto de valores

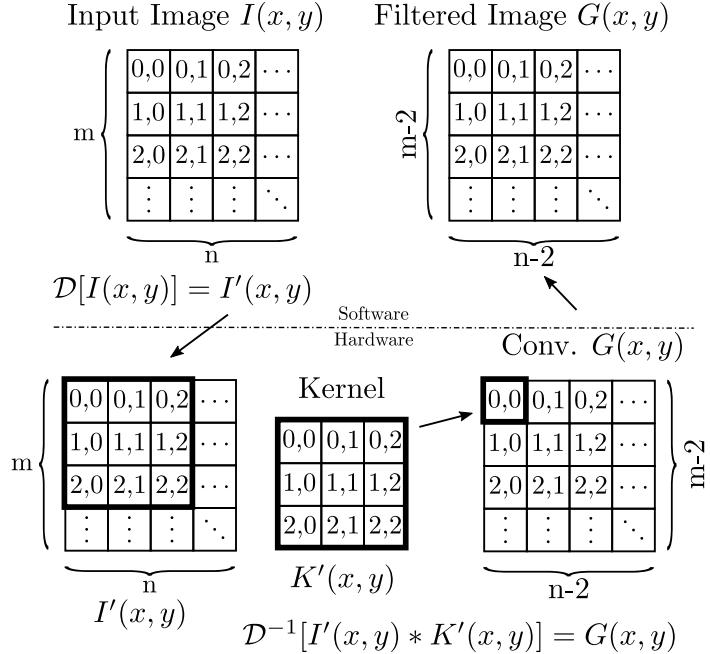


Figura 1-1: Transformaciones del rango dinámico durante el procesamiento de la imagen.

iniciales.

1.2 Representación en punto fijo

La representación en punto flotante, almacena los números en términos de la mantisa y del exponente. El hardware que soporta el formato punto flotante, luego de ejecutar cada computo, automáticamente escala la mantisa y actualiza el exponente para que el resultado se corresponda con el número de bits de forma definida. Todas estas operaciones hacen que el hardware que soporte punto flotante, sea más costoso en términos de área y potencia. Emerge una alternativa: la representación en punto fijo.

La representación en punto fijo se emplea para almacenar y manipular datos, con cierto número de bits fijos. Esto implica que luego del cómputo, no se sigue la posición del punto decimal, y se le delega esta responsabilidad al diseñador del hardware. El punto decimal, valga la redundancia, es fijo para cada variable, y es predefinido. Fijando el punto, una variable se limita a tomar un rango fijo de valores. Pese a que este límite nos brinda ventajas en lo que refiere a utilización de recursos, si el

resultado del cómputo cae fuera del rango, se produce un overflow. Existen varias formas de manipular los overflows que emergen como resultado de un cómputo en punto fijo (redondeo, saturación, truncamiento) y sigue siendo responsabilidad del diseñador optar por la representación adecuada según el sistema que pretende llevar a cabo.

Este análisis tiene por objeto el trabajar con una mínima representación finita para los píxeles de la imagen y los valores del kernel. Para ello se necesita efectuar un estudio acerca de la cantidad de bits de resolución mínimos con una pérdida de información aceptable.

Como se nombro anteriormente, los valores de los píxeles I_{ij} se encuentran en el rango 0 a 255. Con la normalización dynamic range expansion se los lleva al rango 0 a 1. Los valores del kernel K_{ij} pueden ser tanto positivos como negativos. Utilizando Maximum norm se lleva al kernel al rango -1 a 1.

Para poder operar los rangos anteriores en una representación de 8 bits se utiliza representación Signed Fixed Point[2] $S(8, 7)$.

La convolución de una imagen con un $K^{3 \times 3}$ resulta en una salida de 20 bits, en $S(16, 14)$ por cada producto sumado la adición de los nueve elemento tiene una representación total de $S(20, 14)$. Entonces se realiza un post procesamiento donde se lleva el resultado a un rango positivo y truncamiento. Se establece una comparación a fines de decidir la cantidad de bits de salida del procesamiento.

Para realizar una comparación se utiliza la relación SNR de la operación a máxima resolución $S(20, 14)$ y el error producido al reducir la cantidad de bits $e_r = f(x)_{20b} - f(x)_{pos}[3]$.

Al la convolucionar con un filtro unitario y realizar una reducción de los bits de la parte fraccionaria, partiendo de 8 bits en totales, mejora la representación de la imagen , y que a partir de 13 bits totales 1 bit de signo, 5 bits parte entera y 7 en parte fraccionaria, con una representación $S(13, 8)$ tiene una SNR aprimada de 30[dB] que permite observar el efecto de la convolucion con una mínima perdida de información recordando que nuestro objetivo no es no es representar una imagen en su totalidad.

Se puede llevar el rango final a 8 bits si se utiliza dynamic range expansion. Pero se requiere conocer el máximo y el mínimo valor de píxel de la imagen luego de filtrar,

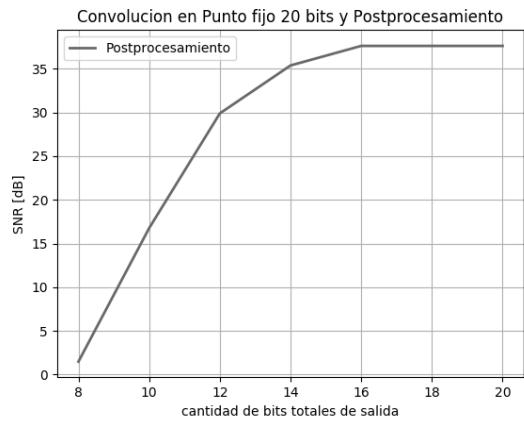


Figura 1-2: Nivel de error según la resolución en bits.

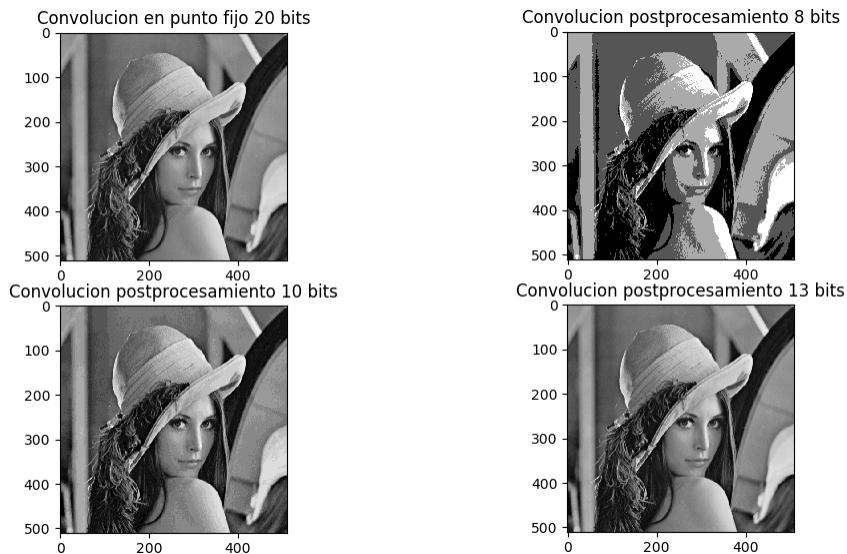


Figura 1-3: Degradación de la imagen según la resolución en bits.

lo que implica que toda la imagen se encuentre en memoria.

1.3 Flujo de diseño

Se estructuró el proyecto y se siguió el flujo o estructura de la Fig. 1-4 para llevar el proyecto a cabo en el tiempo disponible.

En primera instancia se partió de las especificaciones, simulando el comportamiento en punto flotante mediante el lenguaje de alto nivel Python. Luego, se analizaron los

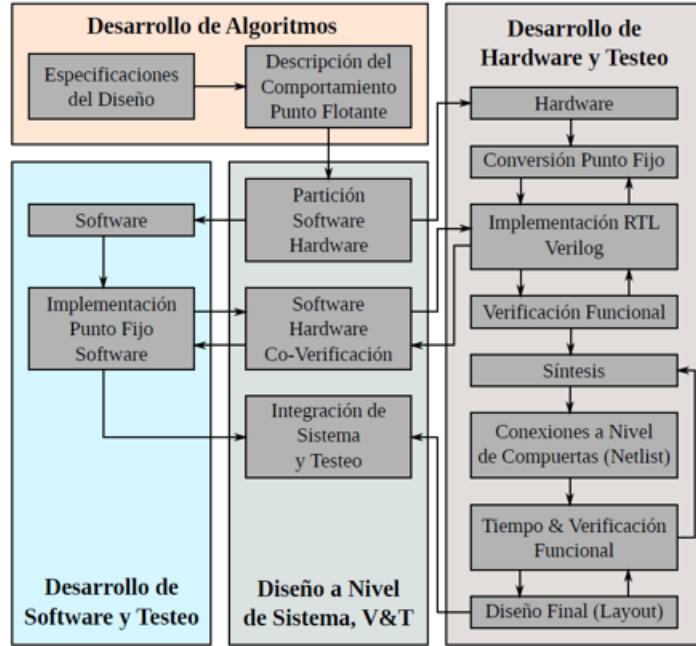


Figura 1-4: Esquemático del flujo de diseño seguido.

resultados obtenidos tras implementar el sistema en punto fijo, a nivel software. Fijados los requerimientos se comenzó con el diseño del hardware, previamente habiendo estudiado y modelado la arquitectura del sistema, la cual se detalla en la sección 2.

1.3.1 Testing

Se realizaron diferentes tipos de testing: test unitarios de cada módulo (test bench del módulo), test de integración (varios módulos y su interacción) y finalmente test de sistema, donde se probó el funcionamiento del sistema completo dada cierta imagen de entrada. Para realizar el proceso de testeo, en primera instancia se corrobora que los bits producidos por el simulador del hardware descripto coincidieran con los bits del simulador del sistema en Python, una vez pasada esta etapa con éxito, se implementa el hardware en una FPGA y se comprueba nuevamente que los resultados obtenidos no difieran con los simulados.

Arquitectura del sistema

2.1 Flujo de trabajo

Describiremos el diseño confeccionado en detalle. Se tienen tres etapas fundamentales:

- **Pre-procesamiento:** consiste en capturar la imagen aplicando las transformaciones mencionadas en la sección 1.1 y 1.2, empleando un script hecho en Python. Este script divide la imagen en lotes o batches y los envía a la FPGA, a través del puerto UART. Un batch o lote, consiste en columnas contiguas de pixeles cuya dimensión se explica en la sección 2.3.1.
- **Procesamiento:** : el batch es convolucionado con el kernel dentro del módulo. Dicho kernel es configurable, y el usuario puede cargar los coeficientes del mismo mediante el GPIO. Cuando la operación de la convolución finaliza, una notificación es enviada a un microprocesador en la FPGA, que da la orden de recuperar el batch procesado, para así enviarlo a la unidad central de procesamiento (CPU).
- **Post-procesamiento:** Como etapa final, en el post-procesamiento se combinan los batches o lotes en CPU usando el script de Python.

2.2 Estados y transiciones

El módulo atraviesa distintos estados para llevar a cabo su tarea. Estos estados se pueden observar en la figura 2-1.

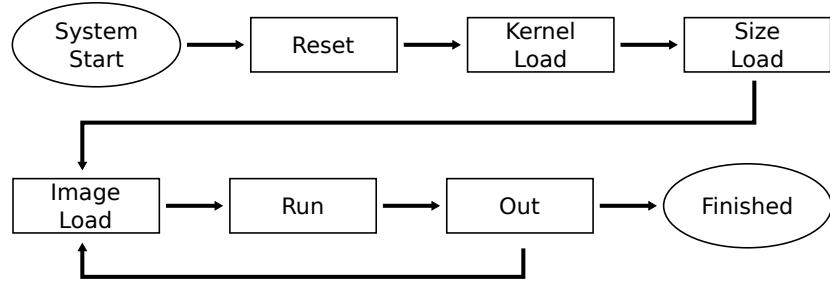


Figura 2-1: Diagrama de estados del sistema.

La transición entre un estado y otro se realiza mediante instrucciones codificadas en el frame de entrada de 32 bits asentado en el GPIO. El módulo permanece en su estado actual hasta recibir una instrucción valida.

Inicialmente, debe situarse al sistema en un estado de reset, y por ende debe recibir una señal de reset para establecerse en dicho estado, esperando a ser configurado. Luego de recibir la correspondiente instrucción, se produce una transición hacia el estado KernelLoad, en donde los coeficientes del kernel se cargan al módulo. El siguiente estado (SizeLoad), es en el cual se carga el tamaño de la imagen, más precisamente la altura de la misma, esto permitirá conocer la posición del último dato en memoria. Estos tres estados mencionados, solo se ejecutan una vez durante todo el ciclo de trabajo. El proceso se repite por cada nueva imagen, por ende, si se desea procesar una nueva imagen, inicia el ciclo nuevamente.

Pasada la etapa de carga, la primera transición es hacia el estado ImageLoad, donde se almacena el lote en la Block RAM de la FPGA. Teniendo el lote almacenado, la transición luego es hacia el estado run donde se hace el filtrado del lote, y mientras el sistema se situe en este estado, no puede ser interrumpido. Como consecuencia, cualquier instrucción recibida se ignora hasta que se complete esta etapa de procesamiento de lote.

Cuando se finaliza el procesamiento de un lote, se emite una notificación y el sistema queda en espera de la última señal para pasar hacia el estado Out. Una vez recibida la correspondiente instrucción codificada, se devuelve el lote procesado al microprocesador de la PC.

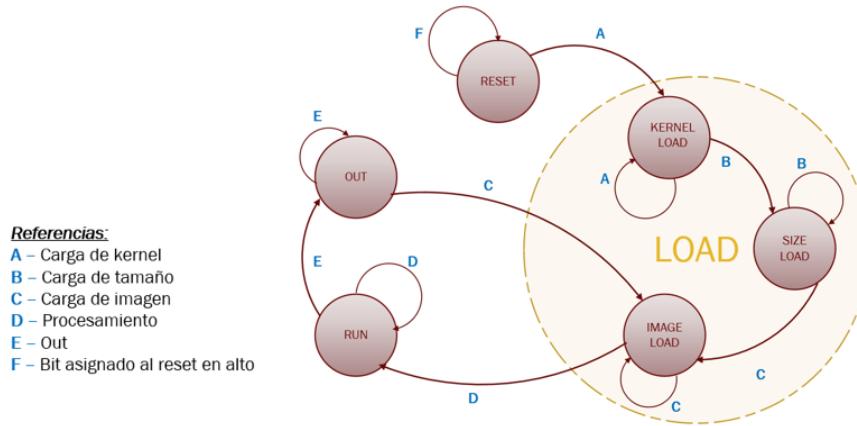


Figura 2-2: Diferentes estados y las etapas involucradas

2.3 Arquitectura del módulo de convolución

El diseño planteado prioriza la reutilización dinámica de memoria, en donde por cada iteración en la etapa de procesamiento, se aprovecha la disponibilidad de datos cargados en memoria en la iteración anterior. En la figura 2-3, se muestran los bloques principales que conforman el diseño. Los bloques que integran el módulo son:

- **Control Unit**: se encarga del manejo de la comunicación entre el módulo y el procesador instanciado en la FPGA.
- **MAC Unit -Multiplier-Accumulator**: ejecuta la suma de los productos entre los coeficientes del kernel y los píxeles de la imagen.
- **Address Generation Unit (AGU)**: bloque que maneja las direcciones de memoria que deben ser leídas o escritas.
- **Memory Management Unit (MMU)**: decide como son escritos y leídos los datos en la memoria.
- **Storage**: hace referencia a un conjunto de columnas formadas por BRAMs de la FPGA. El tamaño depende de la cantidad de MACs instanciadas, o en otras palabras, el paralelismo a nivel de cantidad de convoluciones en paralelo, que se desee en el sistema.

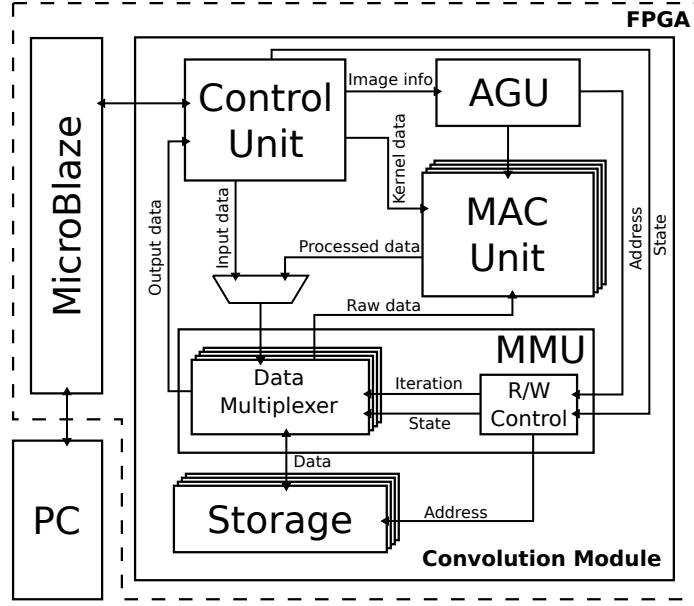


Figura 2-3: Arquitectura general del sistema

Estos bloques se describen en mas detalle en las secciones siguientes.

2.3.1 Storage

Para almacenar un lote, se opto por organizar la BRAM en columnas. Cada columna entonces, corresponde a una columna del lote.

El tamaño del lote está dado por la altura de la imagen y por el número de columnas de memoria instanciadas. La máxima altura en pixels de la imagen, entonces, debe ser menor o igual al número de direcciones de una columna de memoria instanciada.

Durante el procesamiento del lote, cada pixel se lee únicamente una vez, lo que permite un uso mas eficiente de la memoria. Los pixeles del lote que ya fueron leídos, se sobrescriben con los pixeles procesados. Esto permite reducir la cantidad de memoria necesaria, ya que reusa la misma memoria para almacenar el lote de entrada, y el lote procesado.

Estas columnas de BRAM poseen dos puertos de direcciones y datos, uno para los datos de entrada y otro para los datos de salida y además una señal de control, write enable, que habilita la escritura de los datos de entrada en la memoria.

Tabla 2.1: Código de instrucciones, la letra “d” representa bit de datos.

Instrucción	BITS				
	31 - 29	28 - 25	24 - 14	13 - 1	0
Cargar kernel	000	xxx	ddddddddd	ddddd	0
Cargar tamaño imagen	001	xxx	xxxxxxxxxx	xxxx	0
Cargar lote	010	xxx	xxxxxxxxxx	ddddd	0
Recuperar dato	011	xxx	xxxxxxxxxx	xxxxxxxxxx	0
Finalización de lote	100	xxx	xxxxxxxxxx	ddddd	0

2.3.2 Control Unit

El bloque Control Unit es el encargado de hacer de interfaz entre los bloques restantes del módulo de convolución y el microprocesador que se encuentra en la FPGA. Para hacer esto, cuenta con dos puertos de 32 bits conectados a los GPIO de entrada y salida del procesador.

El bloque interpreta las instrucciones que recibe y modifica las señales de control conectadas a los otros bloques, esto representa la transición de estados mencionada en la sección 2.2.

La organización del frame de entrada de 32 bits, junto con los códigos binarios de las instrucciones se muestra en la tabla 2.1.

Como el módulo y el procesador trabajan a distintas frecuencias de reloj, se puede producir un error al enviar y recibir datos entre ambos. Para superar este problema, el flanco ascendente del bit 28 se utiliza como indicador de que un nuevo dato esta listo para ser leído o escrito.

El bit 0 se utiliza como indicador del estado del módulo, en caso de ser 1 el módulo se pone en estado de reset, por lo que una vez atravesado este estado, siempre debería permanecer bajo.

Las instrucciones cumplen las siguientes funciones:

- **Cargar kernel:** Se utiliza para cargar los coeficientes del kernel, cada coeficiente es de 8 bits, junto con la instrucción se deben añadir los 3 coeficientes

que forman una fila del kernel.¹

- **Cargar tamaño imagen:** A esta instrucción debe acompañar el tamaño del alto en pixels de la imagen, se utilizan 10 bits para ello por lo que el tamaño máximo es de 1024px.
- **Cargar lote:** Instrucción que indica que los datos corresponden a un pixel del lote.
- **Finalización de lote:** Indica que el pixel que se está cargando es el último pixel del lote, una vez recibida esta instrucción el módulo pasa al estado de run donde realiza el procesamiento.
- **Recuperar dato:** Con esta instrucción se pide al módulo que entregue el pixel procesado siguiente.

Como se nombró anteriormente, una instrucción no será considerada hasta que no haya un flanco ascendente en el bit 28.

2.3.3 Address Generator Unit (AGU)

El AGU es el encargado de generar las direcciones de memoria que son consumidas por el MMU.

Durante el estado de ImageLoad, el AGU genera las direcciones donde se deben escribir estos datos, para ello cuenta con un contador que se incrementa con cada flanco de subida del bit 28 de la entrada del Control Unit (sección 2.3.2). El AGU además conoce el alto de la imagen (cargado durante SizeLoad) por lo que una vez que el contador alcanza dicho valor envía una señal al MMU indicando que el pixel que se recibe a continuación pertenece a una nueva columna del lote.

Durante la etapa de procesamiento el AGU debe producir dos direcciones diferentes, una que indica al MMU los datos que debe entregar a las unidades MAC y otra que indica al MMU la posición donde los datos procesados por las MACs deben ser

¹Para un diseño que utiliza kernels de 3×3 , en caso de ser de dimensiones superiores, nuevas instrucciones deben ser añadidas.

almacenados. Ambas direcciones se incrementan una vez por ciclo de reloj, pues las MAC generan un dato procesado por ciclo. La diferencia entre la dirección de lectura y la dirección de escritura es igual a la latencia que existe desde que se entrega el primer pixel a una unidad MAC hasta que el primer pixel procesado este listo para almacenarse.

Durante el estado Out, o de salida de datos, el AGU genera las direcciones de lectura de los datos procesados, al igual que en el estado de ImageLoad, se incrementa con cada flanco de subida del bit 28 y tiene en cuenta el alto de la imagen para hacer el cambio de columna de datos.

2.3.4 Multiplier Accumulator Unit (MAC)

Las MACs son unidades que se instancian en paralelo, son las encargadas de hacer el calculo necesario para realizar la convolución. Poseen 9 registros donde se almacenan los coeficientes del kernel y 9 registros donde se almacenan los pixels de la porción del lote que se esta procesando en ese instante.

Las MAC poseen un único puerto de entrada de datos, donde se ingresan 3 coeficientes (correspondientes a una fila) del kernel o de una sección de lote según le sea indicado por una señal proveniente del Control Unit. Además se tiene otra señal de control donde se indica si los registros deben ser modificados o permanecer congelados.

Para un kernel de tamaño $(k \times k)$, cada MAC Unit toma k columnas de memoria adyacentes como entrada. Se cargan entonces k pixeles de cada columna, quedando $(k \times k)$ pixeles cargados dentro de ella, lo necesario para producir el primer pixel procesado. Luego, se procede a efectuar la multiplicación de los pixeles con los coeficientes del kernel y sumar cada término. Al finalizar la suma de los productos, el rango de los datos aumenta, por lo que antes de ponerlos a la salida, las MAC realizan un truncado sobre los resultados.

Una vez procesado un pixel, se efectua un desplazamiento (shift) de sus registros en donde se almacenan pixeles del lote, para así descartar los k pixeles mas antiguos, y se cargan los k nuevos pixeles, es decir, una nueva fila, lo que es equivalente a una estructura FIFO (First Input – First Output). Se sincroniza lo mencionado

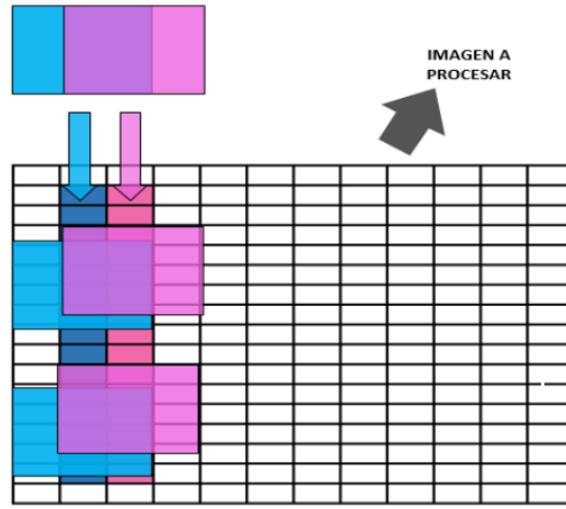


Figura 2-4: Desplazamiento vertical del kernel sobre la imagen

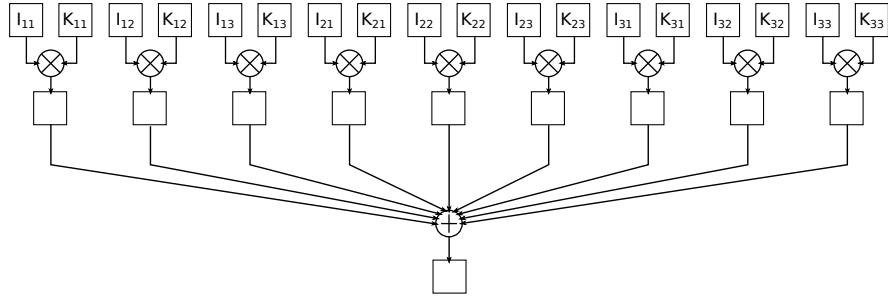


Figura 2-5: Estructura donde se realiza el calculo dentro de una unidad MAC.

anteriormente de forma tal de obtener un pixel procesado por cada ciclo de reloj. Este procedimiento es equivalente a desplazar verticalmente el kernel sobre la imagen como muestra la figura 2-4.

La estructura de hardware para realizar la operación se muestra en la figura 2-5, como el hardware combinacional de los productos en cascada con la suma tiene un tiempo de establecimiento de la salida mayor al período de reloj, se introdujeron registros entre el hardware de los productos y la suma. Si bien esto impacta en la latencia del bloque, la frecuencia de procesamiento se mantiene en un pixel procesado por ciclo de reloj.

2.3.5 Memory Management Unit (MMU)

Como se verá a continuación, para hacer un uso eficiente de las memorias es necesario tener en cuenta el nivel de paralelismo. El objetivo del MMU es hacer de interfaz entre las memorias y el resto de los bloques, independizándolos así del nivel de paralelismo y reduciendo su complejidad.

En la sección 2.3.4 se explicó que para un kernel de dimensión $k \times k$ se necesitan k columnas como entrada para producir una columna procesada en una *MAC Unit*. Por lo tanto, $2 \times k$ columnas, se necesitan para 2 *MACs*. Debido a la naturaleza de la operación de la convolución, para obtener una columna contigua, se necesita desplazar una vez las columnas de entrada. Entonces, existe un solapamiento entre las entradas de las MACs que producen columnas adyacentes, y así la información puede ser compartida. Por esto, pese a necesitar k columnas de entrada por cada unidad *MAC*, solamente $k + 1$ columnas diferentes de entrada se necesitan para dos *MACs* instanciadas. Extendiendo este concepto a N *MACs*, el número requerido de columnas de memoria se reduce de $N \times k$ a $N + k - 1$, concluyendo que agregar una nueva unidad *MAC* solamente agrega una nueva columna de memoria.

Debido al mismo solapamiento explicado anteriormente, hay información repetida entre un lote recibido y el siguiente, por lo que, para reducir la transmisión de datos, esta información repetida se mantiene en memoria y solamente se transmite la parte faltante del lote entrante. Dadas N *MACs* y un kernel de $k \times k$, un lote cuyo ancho (o cantidad de columnas) sea de $N + k - 1$ es necesario. No obstante, el lote procesado tendrá un ancho de N columnas, es decir, una por unidad *MAC*, por lo que las ultimas $k + 1$ memorias no se sobrescriben y mantienen los datos de entrada. Estas $k + 1$ columnas se reutilizan como las primeras columnas del siguiente lote, y así, el ancho del lote transmitido se reduce a N , con la excepción de que el primer lote mantiene un ancho de $N + k - 1$.

La reutilización de columnas de memoria escritas por los lotes previos, resulta en un desplazamiento circular en N lugares, desplazando la posición de las columnas de memoria asociadas con cada unidad *MAC* en cada iteración. De lo anterior, se deduce

que existe una periodicidad entre la relación de columnas de memoria y las entradas de las unidades *MAC*, donde el periodo It es el número de iteraciones necesario para obtener la relación entre las columnas de memoria originales con respecto a las entradas de las unidades *MAC*. Matemáticamente esto ocurre cuando It es un múltiplo de $N + k - 1$, esto es, debe existir un número entero m , tal que:

$$\frac{It}{m} = \frac{N}{k-1} + 1 \quad (2.1)$$

Comparado con un enfoque naive, donde cada MAC unit tendría sus k columnas de memoria independientes, el compartir información entre distintas MACs produce un ahorro en el tamaño de BRAM necesaria, mientras que el desplazamiento circular produce un ahorro en la cantidad de información transmitida. La figura 2-6 muestra una comparación de recursos utilizados por los diferentes abordajes. Analizando la figura 2-6b se observa que con el abordaje naive es necesario transmitir casi 3 veces mas información (por ser un kernel de 3×3) y no se ve afectado por el nivel de paralelismo. En el caso de las entradas compartidas sin desplazamiento se tiene a disminuir los datos redundantes transmitidos a medida que N aumenta, esto es porque los datos redundantes en un lote de $N + k - 1$ es $k - 1$ y $\frac{k-1}{N+k-1}$ tiende a cero a medida que N incrementa. El abordaje de las entradas compartidas con desplazamiento circular es el caso óptimo donde no hay información redundante transferida.

Para asegurar un diseño escalable usando el abordaje de entradas compartidas con desplazamiento circular, se concentró toda esta lógica en el MMU, simplificando el diseño de los bloques restantes.

Este bloque, en cada iteración, mantiene un seguimiento de las posiciones de memoria donde el lote entrante debe ser almacenado, la información que debe alimentar a cada unidad MAC, las columnas de memoria donde la información procesada debe ser almacenada y del orden en el cual la información procesada debe ser devuelta.

Para cumplir con todas sus funciones, internamente el bloque se compone de dos partes: una combinacional, llamada *data multiplexer*, donde se hace el encaminado

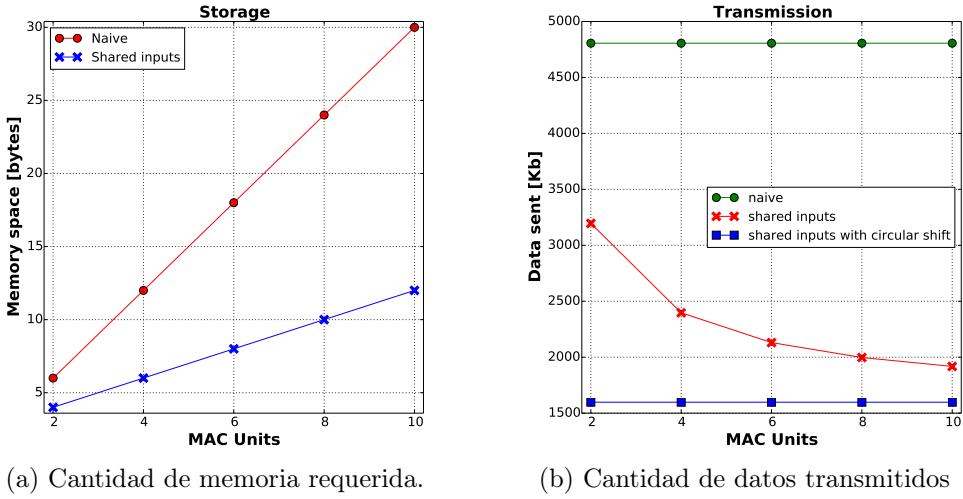


Figura 2-6: Comparación entre abordajes para una imagen de 1600×1024 px y un kernel de 3×3 .

Tabla 2.2: Codificación de estados globales con las de señales SoP y EoP.

Estado global	EoP	SoP
Image Load	0	0
Run	1	0
Out	0	1

de la información desde y hacia las memorias y las MACs, y una parte secuencial basada en una máquina de estados finitos (FSM) que maneja las señales de control de las memorias (write enable) y las señales de control del data multiplexer.

El número de estados de la FSM esta dado por el número de iteraciones It de la ecuación 2.1. Cada estado de esta FSM interna representa una combinación de como se deben escribir y leer los datos en las distintas columnas y no debe confundirse con los estados globales del módulo explicados en la sección 2.2. Dichos estados globales le son informados al MMU mediante dos señales de control: comienzo de procesamiento (SoP) y fin de procesamiento (EoP). La tabla 2.2 muestra la codificación de dichas señales, se observa que no están presentes todos los estados, esto es porque los estados faltantes no afectan al comportamiento del MMU. Además, el se cuenta con una tercera señal, donde se informa que el siguiente dato debe ser escrito y leído en la siguiente columna de memoria.

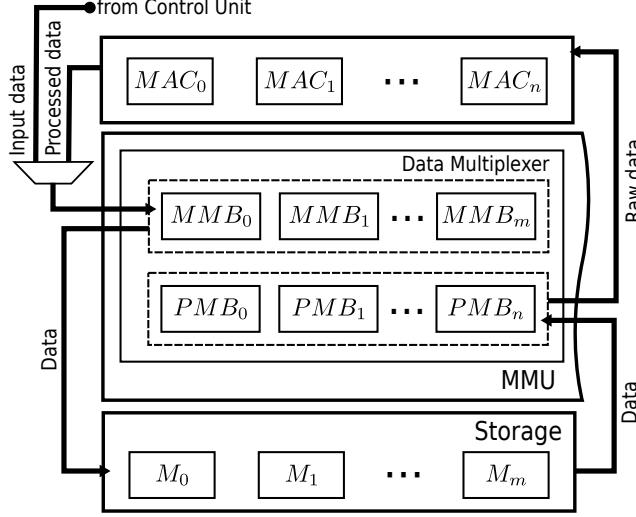


Figura 2-7: Enrutamiento de información

El data multiplexer internamente se compone exclusivamente de multiplexores, que pueden ser agrupados y clasificados en dos bloques más pequeños de acuerdo a la función que cumplen:

- ***MMB*(Memory Multiplexer Blocks)**
- ***PMB*(Processing Multiplexer Blocks)**

La figura 2-7 muestra el flujo de información entre unidades MAC, MMU (data multiplexer) y las memorias. Los *MMB* hacen el routing de la información sin procesar (raw data) y la información desde las unidades *MAC* hacia las memorias. Los *PMB* hacen el routing de la información desde las memorias hacia las unidades *MAC*. La figura 2-8 muestra como están formados ambos bloques.

Tanto los bloques *MMB* como los *PMB* tienen un numero de entradas proporcional al numero de estados de la *FSM* interna del bloque *MMU*. Las entradas a sus multiplexores se definen respectivamente:

$$O_j^i = O_{[(k-1)i+j]\%N} \quad (2.2)$$

$$M_j^i = M_{(iN+j)\%(N+k-1)} \quad (2.3)$$

donde el símbolo % representa la operación módulo, e i toma valores desde 0 a $It - 1$.

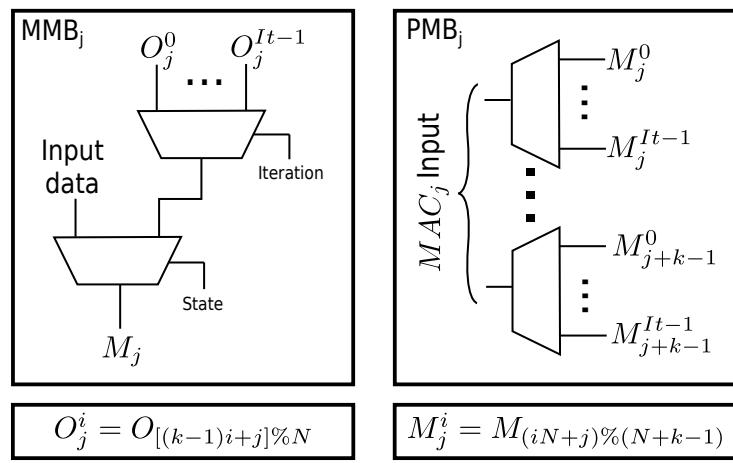


Figura 2-8: Estructura de los bloques MMB y PMB.

Implementación y resultados

3.1 Especificaciones

El diseño fue implementado en una FPGA Xilinx Artix-35T (xc7a35ticsg324-1L) utilizando las herramientas del software Xilinx Vivado Design Suite 2017.4. Se trabajó con un clock de referencia de 100 MHz, y el script para el pre-procesamiento y el post-procesamiento fueron escritos en Python 2.7.

Tanto los coeficientes del kernel como el correspondiente lote de imagen, se cargan a la placa a través del puerto UART. Si bien esta vía es lenta con respecto al tiempo de procesamiento, se optó por utilizarla debido a su simplicidad y al bajo consumo de recursos comparado a otros medios de transmisión.

En lo que refiere a representación de datos, como se explica en la sección 1.2, se trabajó con aritmética en punto fijo, con una resolución de $U(8, 0)$ para la imagen de entrada y salida (imagen procesada), y $S(8, 7)$ para el kernel.

3.2 Verificación

Para verificar el comportamiento del sistema. se armaron diferentes kernels en Python, y se aplicaron los mismos a una imagen de prueba. Luego, se filtró a la imagen con los mismos kernels pero utilizando la FPGA Los kernels que se utilizaron fueron: identidad $[0, 0, 0; 0, 1, 0; 0, 0, 0]$ sharpening $[0, -1, 0; -1, 5, -1; 0, -1, 0]$ y embossing $[-2, -1, 0; -1, 0, 1; 0, 1, 2]$. La figura 3-1 los resultados obtenidos.



(a) Identidad



(b) Sharpening



(c) Embossing

Figura 3-1: Comparación de resultados: A la izquierda, la imagen procesada en una CPU usando python y, a la izquierda, la misma imagen procesada con el módulo en la FPGA.

3.3 Utilización de recursos

La arquitectura se sintetizó para diferentes grados de paralelismo. La tabla Tabla 3.1 muestra la complejidad del módulo y el microprocesador en la FPGA con y sin el uso de DSP respectivamente, medida por la utilización de recursos. Se ve un incremento lineal acorde al grado de paralelismo en el sistema. Debido a las limitaciones de la FPGA utilizada, se llegó a instanciar 8 *MAC units* para que operen en paralelo con DSP, y 24 sin el uso de DSP pero con una utilización de *LUTs* más intensiva.

Tabla 3.1: Utilización de recursos con y sin DSP.

P	With DSP [N](%)			Without DSP [N](%)		
	DSP	LUT	BRAM	DSP	LUT	BRAM
2	20(22)	1845(9)	10(20)	—	3168(15)	10(20)
4	40(44)	2022(10)	11(22)	—	4627(22)	11(22)
6	60(67)	2175(10)	12(24)	—	6063(29)	12(24)
8	80(89)	2448(12)	13(26)	—	7756(37)	13(26)
10	—	—	—	—	9328(45)	14(28)
12	—	—	—	—	10917(52)	15(30)
24	—	—	—	—	20209(97)	21(42)

Tabla 3.2: Throughput obtenido en lo que respecta a la convolución.

Parallelism	Processing Speed [Mp/s]
2	200
4	400
8	800

También se realizó una comparación, mostrada en la figura 3-2, entre la utilización de BRAM estimada, y los resultados obtenidos de la síntesis. Se efectuó una normalización con respecto al porcentaje de utilización de BRAM dividiendo el máximo valor de porcentaje de utilización, y considerando que la instancia del microblaze ya ocupa 18% de los recursos BRAM. Puede observarse que el comportamiento lineal anticipado por los cálculos teóricos, es consistente con los resultados de la síntesis.

3.4 Throughput

Considerando únicamente la operación de convolución, es decir, no teniendo en cuenta el tiempo necesario para cargar la imagen en memoria, se obtiene un pixel por ciclo de clock (100 Mhz) por cada *MAC Unit* instanciada.

El throughput incrementa linealmente con el grado de paralelismo. La tabla 3.2 muestra el throughput obtenido para los diferentes grados de paralelismo.

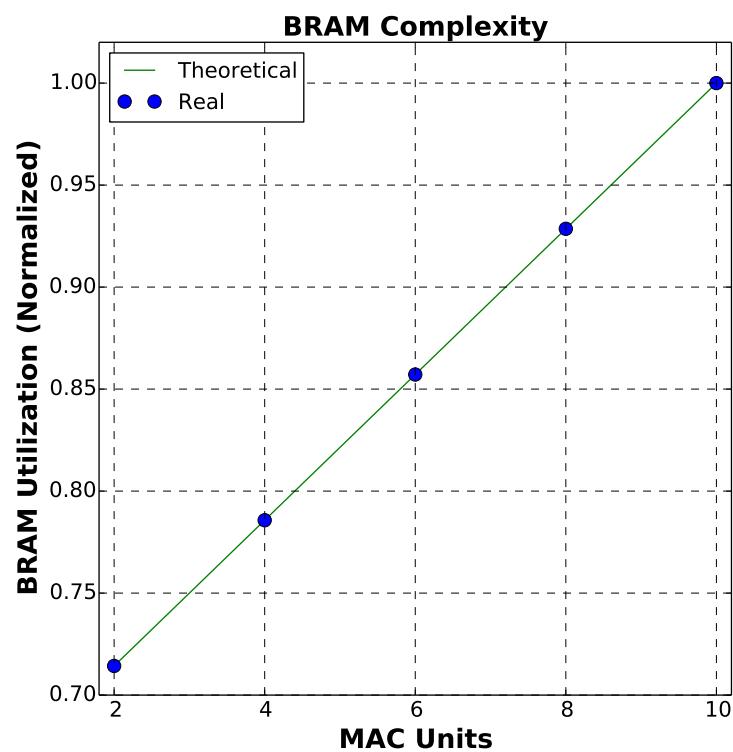


Figura 3-2: Curva normalizada para la utilización de los recursos de BRAM.

Conclusión

4.1 Características del proyecto

- Implementación de la operación de convolución bidimensional en una FPGA
- Particionamiento de imagen por columnas, y carga de las mismas en la placa
- Manejo de uno o más módulos convolucionadores y memorias, es decir, procesamiento con cierto grado de paralelismo
- Kernel de $k \times k$, con k configurable
- Manejo de imágenes de escala de grises (cada píxel tiene asociado un valor de intensidad ó escala de gris)
- Reutilización de hardware mediante un algoritmo descubierto y llevado a cabo en la implementación
- Funcionalidades y etapas en su totalidad: el sistema implementado cubre todas las etapas (carga, procesamiento, devolución)
- Escalabilidad

La flexibilidad y elevada potencia de cálculo de una FPGA permite que se pueda procesar y filtrar el lote de imagen, al mismo tiempo que se está cargando el lote actual.

Se diseñó y presentó una implementación de la operación de convolución bidimensional en una plataforma Xilinx Artix 7 FPGA basada en eficiencia en la utilización de recursos y en el paralelismo del sistema.

Se implementaron todas las etapas necesarias, considerando la etapa de carga, procesamiento, y salida.

Se encontró una relación entre ciertos bloques instanciados, lo que permitió a nuestro sistema trabajar con diferentes niveles de paralelismo.

Ademas, se optimizó la utilización de recursos de almacenamiento implementando un algoritmo para las operaciones con memoria y para la sincronización de módulos. Esto permitió con una parametrización total del sistema, lo que hace escalable a la arquitectura propuesta.

El desempeño del sistema y los resultados obtenidos muestran que la utilización de recursos relacionados con Block RAMs incrementan de forma lineal, como se esperaba.

En lo que refiere al procesamiento, se obtuvo un throughput elevado, por ende del factor limitante mencionado relacionado con la velocidad de transmisión del UART.

4.2 Futuras mejoras

Se plantean como mejoras posibles, la carga de la imagen completa, de disponer una FPGA más grande y tener un medio de comunicación más rápido (Ethernet , por ejemplo) ya que más de un 99% del tiempo el sistema lo invierte en la carga. El procesamiento es muy rápido en relación al tiempo invertido en la carga de datos. Otras posibles mejoras incluirían trabajar con menos bits en lo que refiere a la resolución de la salida (actualmente 13 bits), y el aprovechamiento del bus del GPIO para cargar 3 memorias por escritura en la etapa de carga correspondiente. (Actualmente, por escritura se carga una sola). Análogamente para la etapa de salida de datos. De ser el tamaño de la imagen menor a la profundidad de las memorias definidas, se desaprovechan los restantes registros. Una última futura mejora también sería el aprovechamiento total de las mismas en dicha situación



CONGRESO ARGENTINO DE SISTEMAS EMBEBIDOS



Universidad Tecnológica
Nacional - Facultad
Regional Córdoba



Asociación Civil para la
Investigación, Promoción y
Desarrollo de los Sistemas
Electrónicos Embebidos

Agradecimiento

Agradecemos la participación de
**“Ivan Maximiliano Vignolles, Martin Casabella, Sergio
Alejyo Sulca y Ariel Luis Pola”**
por presentar el siguiente trabajo en la modalidad *Foro Tecnológico*:
**“Dynamic Reuse of Memory in 2D Convolution Applied
to Image Processing”**
en el **Congreso Argentino de Sistemas Embebidos**
realizado del 15 al 17 de Agosto de 2018.

Dr. Guillermo M. Steiner
Coordinador gral. SASE 2018.

Dr. Ariel Lutenberg
Comité CASE 2018.

Dr. Guillermo Riva
Comité CASE 2018.

Dr. José Lipovetzky
Comité CASE 2018.

Mg. Diego J. Brengi
Comité CASE 2018.

Dr. Diegó González Dondo
Comité CASE 2018.

SASE 2018 SIMPOSIO ARGENTINO DE
SISTEMAS EMBEBIDOS
www.sase.com.ar

Agosto 2018
Córdoba Argentina

Figura 4-1: Certificado por la publicación del proyecto llevado a cabo.

4.3 Publicación en CASE-2018

Ademas del tiempo invertido en el desarrollo del trabajo a fines de realizar la Practica Profesional Supervisada, se logró publicarlo en el Congreso Argentino de Sistemas Embebidos del año actual. Dicha publicación fue una gran experiencia que sirvió para adquirir conocimientos y práctica sobre como elaborar artículos técnicos y científicos.

Apéndice

5.1 Proceso de escritura en memoria

Se analiza el caso de tener instanciadas $N = 4$, $MACs$ y un kernel con $k = 3$ para clarificar la operación del algoritmo descrito. El proceso de escritura en memoria, utilizando el desplazamiento circular de columnas mencionado.

Como consideramos un kernel con $k = 3$ y $N = 4$, $MACs$:

- Se necesitan o requieren $N+k-1=4+3-1=6$ columnas de memoria
- En la primera iteración, se cargan las memorias 1 a 6 con el nuevo lote (donde cada memoria instanciada alberga una columna del lote) y el resultado se almacena en las memorias 1 a 4, marcadas con otro color y en la etapa run out.
- En la segunda iteración, los datos del nuevo lote se cargan sin sobrescribir las memorias 5 y 6 . Luego del procesamiento, se almacenan los datos comenzando en la memoria 5 hasta la última, para luego sobrescribir los datos del lote previo desde el principio.

Lo anterior se puede apreciar en la figura 5-1, donde se incluye una línea punteada que separa la etapa de carga y la de procesamiento y salida de dato

Considerando ahora un kernel con $k = 3$ y $N = 2$, $MACs$, las figuras 5-2 y 5-3 intentan ilustrar lo mencionado de forma mas clara, donde se distinguen con colores las columnas involucradas y las etapas respectivas.

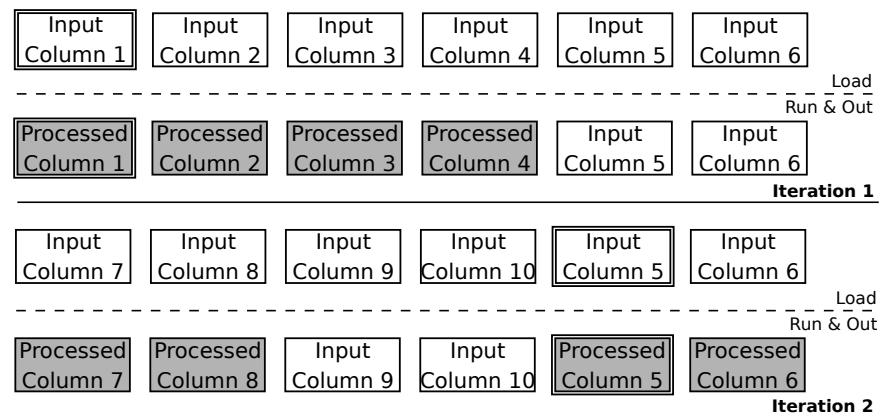


Figura 5-1: Algoritmo de escritura para $N=4$, $k=3$

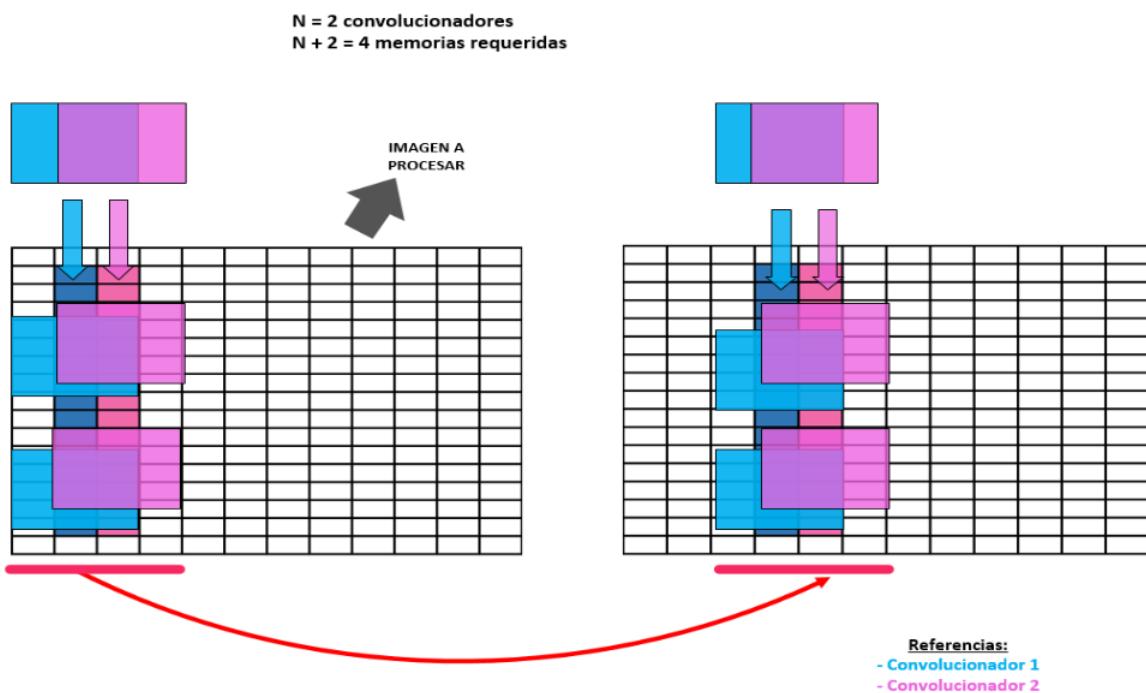
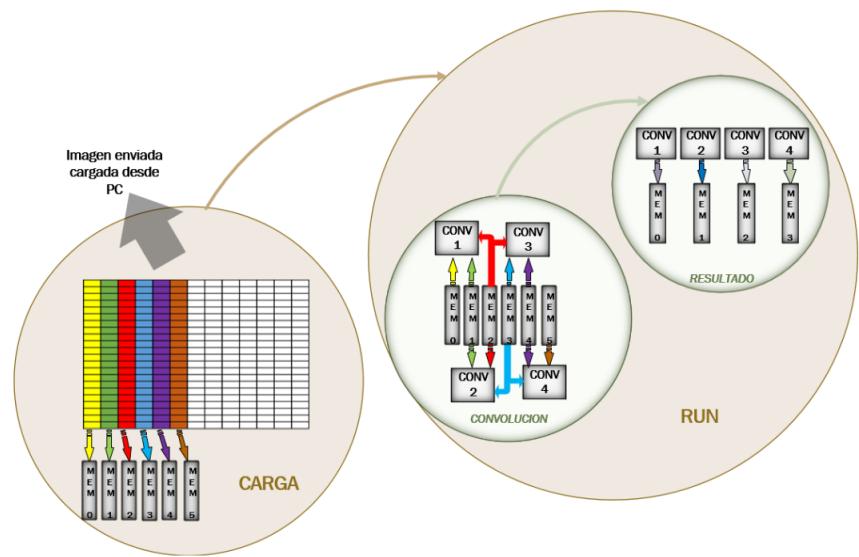
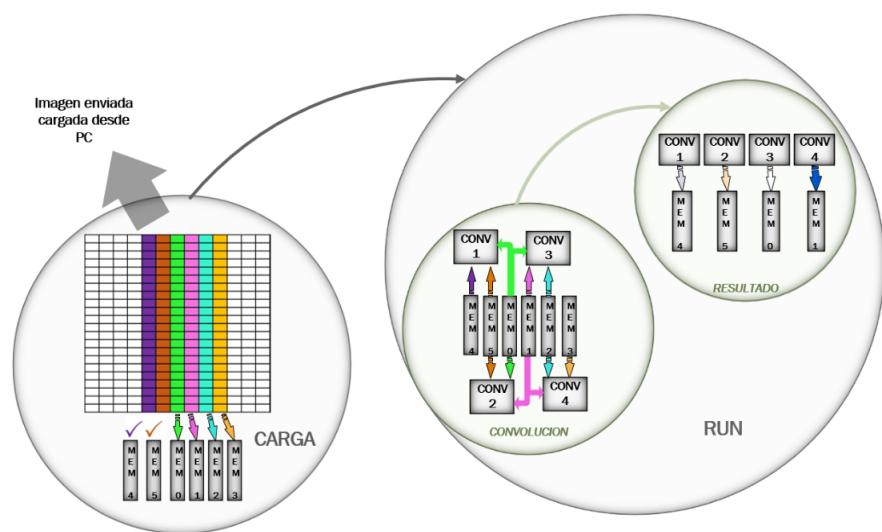


Figura 5-2: Desplazamiento del kernel para $N=2$, $k=3$



(a) Primera iteración



(b) Iteración siguiente

Figura 5-3: Escritura y lectura de memorias para $N = 2, k = 3$.

Bibliografía

- [1] Ben Cope. Implementation of 2d convolution on fpga, gpu and cpu. Technical report, UImperial College London, benjamin.cope@imperial.ac.uk, 2010.
- [2] Keith B. Cullen, Guenole C.M. Silvestre, and Neil J. Hurley. Simulation tools for fixed point dsp algorithms and architectures. *International Journal of Signal Processing*, 1(4):199–203, 2008.
- [3] Hugo Dionisio, Ramón. Análisis del error en algoritmos de transmisión de imágenes comprimidas con pérdida. Master's thesis, Facultad de Informática, U.N.L.P, hramon@lidi.info.unlp.edu.ar, 2002.
- [4] Rafael C. González and Richard E. Woods. *Digital Image Processing*. Pearson Prentice Hall, 3 edition, 2008.
- [5] Agrim Gupta and Viktor K. Prasanna. Energy efficient image convolution on fpga. Technical report, University of Southern California Los Angeles, USA, agrimgupta92@gmail.com, prasanna@usc.edu, 2011.
- [6] Leila kabbai, Anissa Sghaier, Ali Douik, and Mohsen Machhout. Fpga implementation of filtered image using 2d gaussian filter. *International Journal of Advanced Computer Science and Applications*, 7(7):514–520, 2016.
- [7] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.
- [8] Diego Ramírez, Ricardo Romero, and Ferenando Santa. Parallel processing on fpga for image convolution using matlab. Master's thesis, Universidad Distrital Francisco José de Caldas, Bogotá, Colombia, 2015.
- [9] Walter Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, 3 edition, 1976.
- [10] Andrew M. Saxe, Pang Wei Koh, Zhenghao Chen, Maneesh Bhand, Bipin Suresh, and Andrew Y. Ng. On random weights and unsupervised feature learning. In *ICML*, pages 1089–1096. Omnipress, 2011.
- [11] Henrik Ström. A parallel fpga implementation of image convolution. Master's thesis, Department of Electrical Engineering, Linköping University, 2016.

- [12] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA ’15, pages 161–170, New York, NY, USA, 2015. ACM.