UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Máster en Computación en la Nube y de Altas Prestaciones

Curso: 2021-2022

Trabajo Fin de Máster

Desarrollo y paralelización del algoritmo de convolución rápida de Winograd sobre plataformas ARM de bajo consumo

Autor:
Duero Joshua Cuadrillero Geer

Tutores:
Manuel F. Dolz Zaragozá
Pedro Alonso Jordá

Resumen

Los avances en el diseño y desarrollo de redes neuronales convolucionales profundas, así como el incremento de la precisión de las mismas en el campo de la visión artificial, ha supuesto su amplia adopción en un gran abanico de aplicaciones inteligentes. La mayoría de dispositivos móviles y empotrados, ya sea en forma de teléfono móvil, tableta, sistema de navegación, sistemas empotrado o robot, incorporan procesadores ARM y requieren la ejecución eficiente y consciente del consumo este tipo de redes neuronales. Desde el punto de vista computacional, la fase de inferencia de las redes neuronales convolucionales se puede resumir mayormente en la realización de dos operaciones básicas: la multiplicación de matrices, para el procesamiento de capas completamente conectadas, y la convolución, para el procesamiento de capas convolucionales.

En este trabajo se aborda el desarrollo y la paralelización del algoritmo de convolución rápida de Winograd, una variante de esta operación basada en transformaciones y que tiene como objetivo de maximizar el rendimiento de la fase de inferencia a costa de reducir el número de operaciones realizadas. En concreto, el trabajo detalla las implementaciones y optimizaciones llevadas a cabo para aprovechar la naturaleza del algoritmo rápido de convolución sobre procesadores ARM y se compara con la convolución directa y en los basados en transformaciones im2col o im2row, seguidas de una multiplicación de matrices. La implementación desarrollada en este trabajo explora, además, distintas versiones vectorizadas y paralelas del algoritmo basadas en el uso de funciones intrínsecas de ARM-NEON y en el entorno de programación paralelo OpenMP.

Índice general

Capítulo 1

Introducción

Los avances científicos en el campo de la visión artificial junto con los desarrollos de nuevas arquitecturas paralelas y dispositivos móviles han permitido la adopción masiva de las tecnologías de la inteligencia artificial basadas en el uso de las redes neuronales profundas. En este capítulo, se introduce el tema objeto de estudio de este trabajo, en el que se aborda una forma eficiente de implementar una de las operaciones más habituales en las redes neuronales convolucionales, la convolución rápida de Winograd.

1.1. Redes neuronales convolucionales

Las redes neuronales son modelos computacionales dentro del campo del aprendizaje automático que permiten realizar predicciones para muestras de datos. Las redes neuronales están formadas por capas de neuronas conectadas entre sí, y cuyos enlaces tienen asociado un determinado peso. Inicialmente, estos pesos contienen valores aleatorios, por lo que es necesario ajustarlos antes de que la red pueda ser utilizada para inferir nuevas muestras. El proceso de ajuste se lleva a cabo a partir de un conjunto de entrenamiento con muestras etiquetadas (*aprendizaje supervisado*), infiriendo cada una de las muestras del conjunto para conocer el error de predicción cometido y corregirlo a partir de un procedimiento iterativo conocido como entrenamiento. Este proceso consta de dos fases: pasada hacia delante y hacía atrás. En la pasada hacia delante, las muestras se infieren para calcular el error cometido a partir de una función de pérdida. En la pasada hacia atrás se calculan una serie de gradientes que permiten ajustar los pesos de la red a partir de cierta función de optimización, e.g., utilizando el descenso del gradiente estocástico o *stochastic gradient descent*.

Estas redes han evolucionado en múltiples direcciones para solucionar problemas en diferentes áreas de la ciencia e ingeniería. En el caso de la visión por computador, las redes neuronales convolucionales se han postulado como herramientas clave para, entre otras cosas, clasificar, segmentar y detectar objetos en imágenes.

En la Figura 1.1 se pueden observar distintas capas de una red neuronal convolucional usada para clasificación. Los tipos de capas que se pueden encontrar son las siguientes:

- FC (*Fully-Connected*): Las capas *fully-connected* o completamente conexas, interconectan todas sus neuronas con las de la siguiente capa y mantienen un peso diferente para cada una de las diferentes conexiones.
- Convolucional: Estas capas utilizan la operación de convolución para aplicar un conjunto

Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	224 x 224 x 3	-	-	-
1	2 X Convolution	64	224 x 224 x 64	3x3	1	relu
	Max Pooling	64	112 x 112 x 64	3x3	2	relu
3	2 X Convolution	128	112 x 112 x 128	3x3	1	relu
	Max Pooling	128	56 x 56 x 128	3x3	2	relu
5	2 X Convolution	256	56 x 56 x 256	3x3	1	relu
	Max Pooling	256	28 x 28 x 256	3x3	2	relu
7	3 X Convolution	512	28 x 28 x 512	3x3	1	relu
	Max Pooling	512	14 x 14 x 512	3x3	2	relu
10	3 X Convolution	512	14 x 14 x 512	3x3	1	relu
	Max Pooling	512	7 x 7 x 512	3x3	2	relu
13	FC	-	25088	-	-	relu
14	FC	-	4096	-	-	relu
15	FC	-	4096	-	-	relu
Output	FC	-	1000	-	-	Softmax

Figura 1.1: Descripción de las capas de una red neuronal VGG16 [15].

de filtros sobre las imágenes de entrada, de tal modo que es posible obtener los mapas de características (*feature maps*). Esta operación es una de las más importantes en las redes convolucionales y es la que se estudiará a lo largo de este trabajo.

- Pooling: Esta capa es similar a la convolución y aplica una determinada operación sobre una ventana de píxeles de la imagen de entrada (véase Figura 1.2). Las operaciones más comunes en las capas de tipo pooling son la función "Máximo" (se obtiene el máximo valor en la ventaja), "Mínimo" (ídem, pero con el mínimo) y "Media" (se obtiene la media de los valores en el subconjunto). Para el caso de la función "Media", la capa pooling también puede implementarse mediante una convolución con un filtro donde cada valor es igual a $1/n$, siendo n es el número de elementos del filtro.

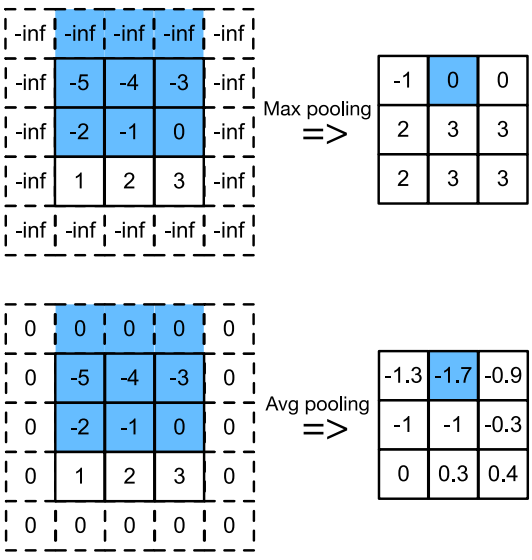

Figura 1.2: Diagrama de la operación de Pooling (pooling "máximo" arriba y pooling "media" abajo) [16].

1.2. Algoritmos para el cálculo de convoluciones

En esta sección se revisan las diferentes variantes de los algoritmos que implementan la operación de convolución en el ámbito de las redes neuronales profundas.

1.2.1. Convolución directa y usos

Una implementación inicial de la convolución, es aquella donde directamente aplicamos la operación de multiplicar punto a punto cada filtro por la parte correspondiente de la imagen de entrada, acumulando el resultado de las diferentes multiplicaciones y ubicando el correspondiente píxel de salida. Visto de forma gráfica, como ya hemos mencionado, la operación de convolución es similar a la de pooling "Media" (véase la Figura 1.2) donde, en vez de multiplicar cada elemento por el filtro con valores $1/n$, cada elemento del filtro puede tener un valor arbitrario. La expresión general de la operación de convolución [1] se puede ver en la formula 1.1. En esta operación se pueden utilizar distintos filtros para lograr extraer características diferentes. En la referencia [14] se pueden observar algunos efectos de aplicar distintos filtros a una misma imagen. En redes neuronales, los pesos que toman los filtros de la convolución son parámetros entrenables, es decir, se ajustan durante la mencionada fase de entrenamiento, permitiendo así la captura de las características que la red considere en función de las muestras de entrenamiento utilizadas.

$$Y_{i,k} = \sum_{c=1}^{C} D_{i,c} \cdot G_{k,c}. \tag{1.1}$$

1.2.2. En busca de más eficiencia

En causa al gran uso de la convolución para procesar imágenes, se han desarrollado distintos algoritmos y métodos con el objetivo de ganar eficiencia, algunos de estos son:

- **Métodos basados en producto matricial**: Estos métodos reordenan los datos de la imagen y filtro de forma que la convolución pueda transformarse en una multiplicación de matrices (véase la Figura 1.3). Una vez obtenida la reordenación, se pueden utilizar bibliotecas de cómputo algebraico para obtener resultados muy rápidos. No obstante, la matriz que reordena los datos de la imagen de entrada, no solo reordena sino que también los replica. Este hecho produce que la ejecución de esta operación tenga un consumo de memoria no despreciable. Esta variante del algoritmo se la conoce como convolución basada en multiplicaciones de matrices o *GEMM*. Dos implementaciones de este método son `im2row` y `im2col`.
- **Métodos basados en transformaciones**: Estos métodos transforman la imagen y el filtro a un dominio de operación donde el coste computacional es mínimo. Una de estas transformaciones se basa en la transformada rápida de Fourier (*Fast Fourier Transform* o FFT) convirtiendo la matriz de imagen y filtros al espacio de la frecuencia, realizando las operaciones en dicho espacio y regresando al espacio de la imagen tras realizar la operación de convolución [1]. Otro de los métodos dentro de esta categoría es el algoritmo de Winograd para convoluciones, sobre el que se basa el trabajo.

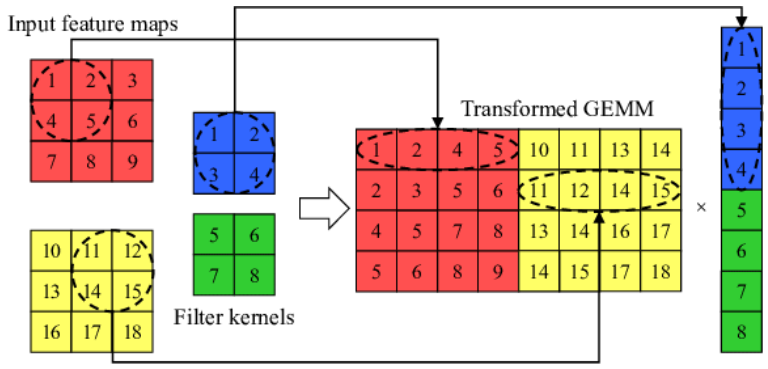

Figura 1.3: Proceso tomado por `im2row` de transformación de imagen y filtro [4].

1.3. Procesadores ARM y computación móvil

La computación móvil es de gran importancia hoy en día, ya que móviles, tabletas, relojes inteligentes son grandemente usados como medios de comunicación. Entre los microprocesadores para el uso móvil predomina con un 90 % de la cuota del mercado la arquitectura ARM [6]. ARM es una arquitectura basada en un conjunto de instrucciones reducido, que permite a sus procesadores tener un menor tamaño, así como un menor consumo de energía, haciéndolos ideales para la computación móvil.

1.3.1. Extensión NEON

La extensión de paralelización a nivel de instrucción NEON incorpora un conjunto de instrucciones vectorizadas en registros de 64 y 128 bits. Esta extensión está soportada por los modelos de procesador Cortex-A y Cortex-R. Los modelos Cortex-A focalizan en un entorno multidisciplinar (eficiencia energética, estabilidad/fiabilidad), mientras que los Cortex-R buscan maximizar el rendimiento con el objetivo de poder mantener las aplicaciones ejecutándose en tiempo real.

Esta extensión está soportada por el compilador GCC. Además, soporta un conjunto de funciones intrínsecas (*NEON Intrinsics*) que permiten mayor facilidad a la hora de programar con dichas instrucciones. Una lista de las instrucciones usadas en el trabajo junto a una breve descripción se puede ver en la Tabla 1.1. Además, se puede explorar la totalidad del conjunto de funciones intrínsecas en la documentación oficial de ARM [10].

1.3.2. Dispositivos NVIDIA Jetson

ARM, además de predominar en computación móvil, también está presente en sistemas empotrados u ordenadores "a bordo". Entre este tipo de máquinas cabe destacar la gama de dispositivos NVIDIA Jetson, una serie de ordenadores compactos que permiten su fácil instalación en calles, vehículos o interiores manteniendo un buen rendimiento para tareas de procesamiento de imagen o inteligencia artificial. Todos los dispositivos de esta línea se componen de procesadores ARM con soporte para la extensión NEON.

Tabla 1.1: Tabla que muestra las funciones intrínsecas usadas en el trabajo junto a una breve descripción.

Función	**Descripción**
`vld1q_f32`	Carga un registro float32x4_t de memoria.
`vst1q_f32`	Almacena un registro float32x4_t de forma contigua en memoria
`vgetq_lane_f32`	Extrae un valor del registro float32x4_t proporcionado.
`vst4q_lane_f32`	Almacena una columna de una matriz 4×4 (float32x4x4_t) de forma contigua en memoria.
`vmovq_n_f32`	Devuelve un registro float32x4_t inicializado con el valor proporcionado.
`vaddq_f32`	Devuelve un registro float32x4_t con la suma elemento a elemento los registros proporcionados.
`vsubq_f32`	Devuelve un registro float32x4_t con la resta elemento a elemento los registros proporcionados.
`vmulq_f32`	Devuelve un registro float32x4_t con la multiplicación elemento a elemento los registros proporcionados.
`vfmaq_f32`	Multiplicación de dos registros float32x4_t y acumulación en el registro destino.
`vuzpq_f32`	Devuelve un registro float32x4x2_t, una mitad con los elementos de índice par de los registros proporcionados y otra mitad con los de índice impar .
`vtrnq_f32`	Devuelve un registro float32x4x2_t con la traspuesta de los elementos proporcionados por argumentos.

1.4. Objetivos del trabajo

Este trabajo pretende explorar y mostrar posibles implementaciones del algoritmo de Winograd para acelerar la operación de convolución. Los objetivos de este se dividen en los siguientes:

1. **Realizar una implementación eficiente**: Se realizará una serie de optimizaciones en la implementación del algoritmo de convolución rápida de Winograd con el objetivo de mostrar/explorar algunas de las posibles implementaciones eficientes de este.

2. **Paralelización de los algoritmos**: Se realizará una paralelización por dos métodos distintos, una usando instrucciones SIMD mediante instrucciones ARM NEON, y otra a nivel de hilo mediante la API de OpenMP [9].

3. **Resultados experimentales sobre plataformas ARM**: Se compararán resultados con otras implementaciones de algoritmos eficientes de cálculo de convoluciones, midiendo el rendimiento relativo de la implementación desarrollada en este trabajo.

1.4.1. Estructura del documento

El resto del trabajo se estructura en los siguientes cuatro capítulos:

- **Capitulo 2:**: En este capítulo se expondrá el algoritmo que se toma como base para realizar el trabajo expuesto, además de introducir la idea del uso del producto matricial a la hora de implementarlo.
- **Capitulo 3**: En este capítulo se documentan las implementaciones realizadas, así como las optimizaciones y paralelizaciones sobre ellas.
- **Capitulo 4**: En este capítulo se documentan los resultados de las pruebas realizadas sobre la implementación expuesta en el trabajo.
- **Capitulo 5**: En este capítulo se exponen los objetivos cumplidos y posibles trabajos futuros sobre el trabajo realizado.

Capítulo 2

El Algoritmo de Winograd

En este capítulo se muestra el método de Winograd basado en transformación para minimizar el coste en operaciones de la operación de convolución. Se conoce desde, por lo menos 1980, que el mínimo número de operaciones para realizar una convolución es función del número de veces que se aplique el filtro consecutivamente y del tamaño del filtro, tal como se puede observar en las siguientes fórmulas [1]:

$$\mu(F(m,r)) = m+r-1, \tag{2.1}$$

$$\mu(F(m\times n, r\times s)) = (m+r-1)(n+s-1), \tag{2.2}$$

donde $\mu()$ es la cuenta del número de operaciones, $F()$ es la operación de convolución mínima en operaciones, m y n son las veces que se va a aplicar el filtro, y r y s son el tamaño del filtro. La primera fórmula permite calcular el mínimo número de operaciones necesarias para realizar una convolución (2.1), mientras que la segunda es su extensión para dos dimensiones (2.2).

Para hacer uso de una notación más simple, se usará el formato $F(m\times n, r\times s)$, donde m y n es el tamaño de la matriz final de convolución y r y s el tamaño del filtro. Téngase en cuenta que el tamaño de entrada de la imagen será $m+(r-1)$ y $n+(s-1)$. Por ejemplo, la expresión $F(2\times 2, 3\times 3)$ significa que tendrá como entrada una sección de imagen de tamaño 4×4, un filtro de tamaño 3×3 y el resultado final será de tamaño 2×2.

2.1. Descripción del algoritmo

Para conseguir un número mínimo de operaciones en la operación de convolución, hay que tratar la operación como una multiplicación de polinomios [3]. Shmuel Winograd, basándose en el teorema chino del resto para polinomios consigue un algoritmo que mediante una serie de transformaciones convierte una imagen y filtro para posteriormente realizar una operación de convolución mínima. La expresión matricial de las transformaciones se puede ver en la ecuación matricial (2.3),

$$f = Z^T \left(\sum_{c=0}^{C} \left[(WwW^T)_c \odot (X^T x X)_c \right] \right) Z, \tag{2.3}$$

donde w es un filtro, x la imagen sobre la cual se aplica el filtro, el operador $\odot$ es el producto elemento a elemento matricial (producto de Hadamard) y W, X y Z son matrices de transformación [2]. La convolución se realiza sobre una imagen te tamaño fijo (con unas matrices de transformación en

función al tamaño de la imagen y filtro), por lo que en vez de calcular las matrices de transformación de cada imagen sobre la que se quiera operar, es más práctico subdividir la imagen en varios bloques de tamaño fijo para poder usar las mismas matrices de transformación para cualquier tamaño de imagen.

Con solo la ecuación matricial de convolución de Winograd, alguien podría pensar que esta requiere de más operaciones que la convolución directa. Esto en efecto es así si analizamos la complejidad de la ecuación, ya que las multiplicaciones de matrices presentes son costosas. Aun así, este método consigue minimizar el número de operaciones requeridas gracias al formato de las matrices de transformación. Las matrices de transformación [1] para una imagen 4×4, filtro 3×3 y resultado de convolución 2×2 se muestra en (2.4),

$$W = \begin{pmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 0 & -1 \end{pmatrix}. \tag{2.4}$$

Para un bloque de 4×4 de imagen, un filtro 3×3 y un solo canal, la transformación del filtro (WwW^T) requiere 42 operaciones (14 multiplicaciones y 28 restas), la de la imagen ($X^T xX$), 32 sumas, el producto elemento a elemento 16 multiplicaciones y la transformación final $Z^T(\cdots)Z$ usa 24 sumas, para un total de 114 operaciones. Esto es una cifra significativa en comparación a las 72 operaciones teóricamente necesarias para una convolución directa sobre los mismos datos, en concreto un 58 % peor. Aunque para una sola sección de imagen y un solo kernel el número de operaciones sea mayor, este efecto se mitiga en cuanto el número de bloques o kernels aumenta. Por ejemplo, si hay que hacer una convolución sobre una imagen que tiene 100 secciones mediante 20 kernels, las transformaciones necesarias serian 100 de imagen (3.200 operaciones) y 20 de kernel (840 operaciones) más la operación de convolución transformada y transformación final por aplicación de filtro a cada sección (20 filtros $\times$ 100 imágenes $\times$ $(16 + 24)$ instrucciones para convolución y transformación final, un total de 80.000 operaciones), dando un total de 84.040 operaciones con respecto a las 144.000 (72×100 imágenes $\times$20 filtros) necesarias por convolución directa. En este último caso se puede empezar a observar la mejora de complejidad en la operación, con un 70 % menos de operaciones respecto al método usual.

2.2. Algoritmo de Winograd mediante multiplicación matricial

Como se ha visto en la sección de métodos (Sección 1.2.2) más eficientes para el cálculo de convoluciones de la introducción, uno de los métodos (`im2row`/`im2col`) utilizaba la multiplicación matricial para el cálculo, basándose en que hay medios muy eficientes ya en existencia para el cálculo de este. Similarmente, como muestra el artículo "Efficient Winograd or Cook-Toom Convolution Kernel Implementation on Widely Used Mobile CPUs" [2], se puede aplicar esta idea al algoritmo de Winograd, pero en vez de tener que reordenar y duplicar datos, se puede hacer de forma directa en la sección de producto elemento a elemento y acumulación, es decir, en el cálculo del siguiente

sumatorio,

$$\sum_{c=0}^{C} \left[(WwW^T)_c \odot (X^T xX)_c \right] .$$

Para esto, se tiene que ordenar la salida de la transformación de imagen ($X^T xX$) y filtro (WwW^T). La transformación de imagen debe tener como primera dimensión los canales transformados, y como segunda el número de secciones transformadas multiplicado por los n elementos de la matriz de transformación de imagen o filtro (en el caso de $F(2\times2, 3\times3)$ serian 16 elementos). La transformación del filtro debe tener como primera dimensión el número de kernels multiplicado por el número de elementos de la matriz de transformación o filtro (para $F(2 \times 2, 3 \times 3)$ serian 16 elementos), y finalmente como segunda dimensión los canales transformados.

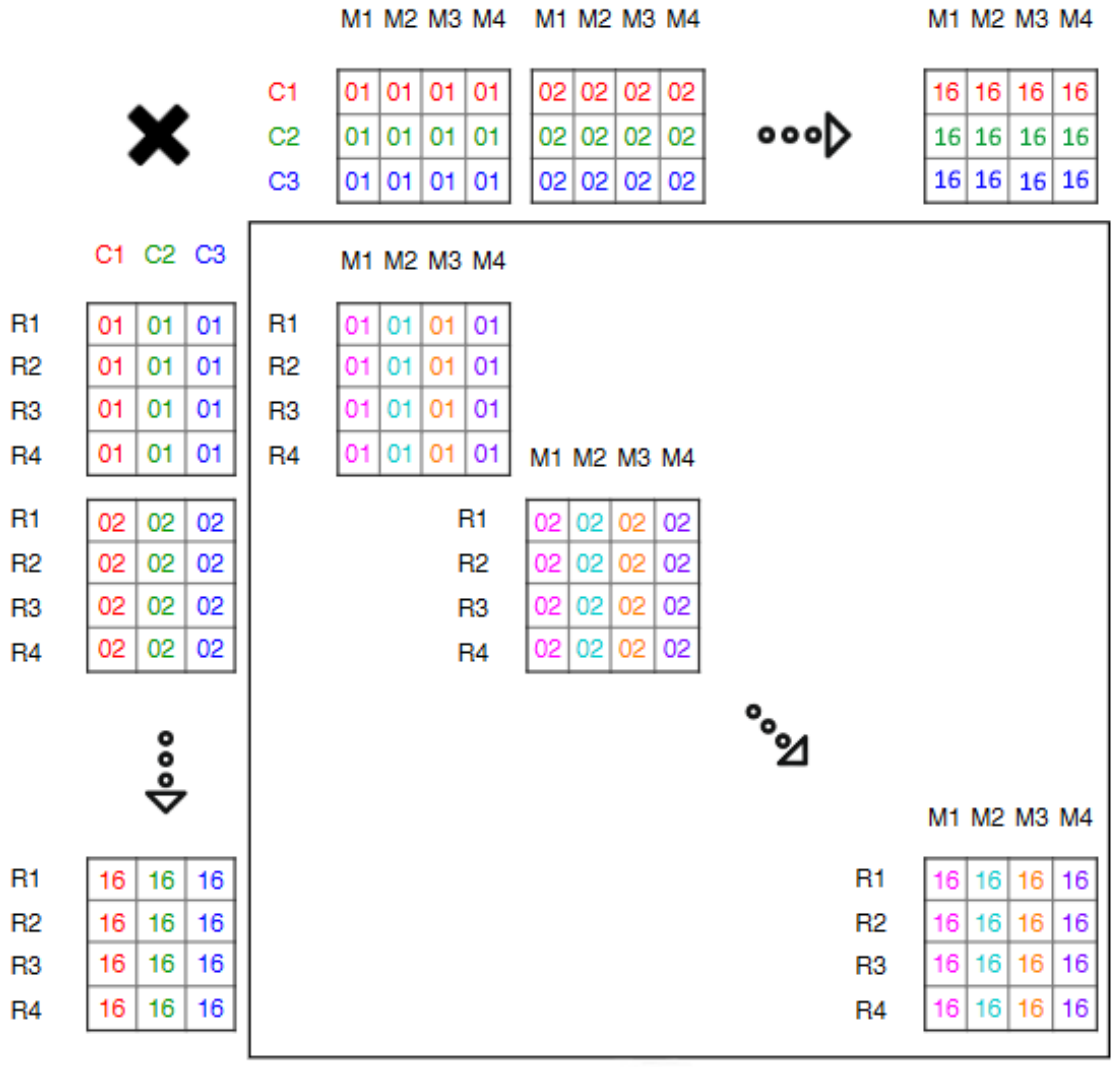

Figura 2.1: Estructura de las matrices de transformación de imagen (izquierda) y filtro (arriba) que permite hacer multiplicación matricial con ellas. La imagen es un ejemplo para $F(2 \times 2, 3 \times 3)$ [2].

En la Figura 2.1 se puede observar el formato de las transformaciones. En la izquierda se puede ver la transformación de la imagen, donde $R_1 \cdots R_n$ son cada sección de imagen transformada y $C_1 \cdots C_n$ son los canales de la imagen transformada. En la parte superior se encuentra la transformación del filtro, de forma que $C_1 \cdots C_n$ son los canales y $M_1 \cdots M_n$ son cada uno de los kernels a aplicar en la

convolución.

Se puede observar en la imagen (Figura 2.1) que el elemento número uno de cada transformación de imagen se multiplica matricialmente por cada elemento uno de la transformación del filtro, acumulando la dimensión de canales. Para cada elemento del tamaño de la transformación de imagen/filtro se hace una multiplicación de matriz, en el caso de la imagen, son 16.

El algoritmo de Winograd para convoluciones y su implementación mediante la multiplicación matricial son la base de la que se parte para realizar el trabajo. A partir de este algoritmo se realizarán varias implementaciones, algunas de ellas mediante multiplicación matricial realizada por bibliotecas de cómputo algebraico conocidas.

Capítulo 3

Implementación

En este capítulo se muestran todas las técnicas y optimizaciones realizadas en la implementación del algoritmo de Winograd. Además, se explicarán las dos paralelizaciones realizadas sobre el algoritmo, a nivel de instrucción y de hilo.

3.1. Implementación base

3.1.1. Estructuración de datos

Lo primero a tener en cuenta es el formato de datos de entrada. Existen varios posibles teniendo en cuenta las dimensiones de entrada. Las dimensiones principales son:

- N: El número de imágenes en el batch, el batch es el conjunto de imágenes sobre el que se quiere aplicar la convolución.
- C: El número de canales que tiene cada imagen. En una imagen el canal es cada valor de color que se almacena por píxel. En una imagen a color RGB (Red Green Blue) se almacenan los valores de cada píxel en tres canales, uno por cada color. En las capas intermedias de redes neuronales convolucionales toma el significado de la profundidad en el mapa de características (*feature map*), permitiendo representar distintas capas de información, las cuales el modelo usa para tomar la decisión.
- K: El número de filtros (también llamados kernels) que procesarán la imagen. Aplicar un mayor número de filtros en una red neuronal convolucional permite extraer un mayor número de características sobre las que el modelo ha de inferir su clasificación.
- H y W: El alto y ancho respectivamente de cada imagen.
- R y S: El alto y ancho respectivamente de los filtros.

En redes neuronales, el formato de salida (el resultado de la convolución) esta condicionado al de entrada de la capa siguiente, ya que la salida del número de filtros que procesan la convolución en la capa actual, se convierten en los canales de la imagen de la entrada de la capa siguiente. Con esto en mente, dos de los formatos más comunes en el procesamiento de redes neuronales son los siguientes:

- **NCHW - CKRS - NKHW**: En este modelo, la imagen se estructura de la forma NCHW, de forma que se estructuran contiguamente los elementos de imagen de un mismo canal. El filtro

se estructura de la forma CKRS, donde se almacenan contiguamente todos los elementos de un filtro y canal. Finalmente, la salida es en formato NKHW, igual al de la imagen de entrada remplazando la acumulación de canales por la expansión de kernels. Una representación visual se puede ver en la Figura 3.1.

- **NHWC - CRSK - NHWK**: Este orden difiere con el anterior en cuanto a que la imagen tiene los elementos del canal contiguos en memoria, el filtro tiene los elementos de distintos kernels contiguos en memoria y la salida también mantiene el orden de distintos kernels contiguos en memoria. Una representación visual se puede ver también en la Figura 3.1.

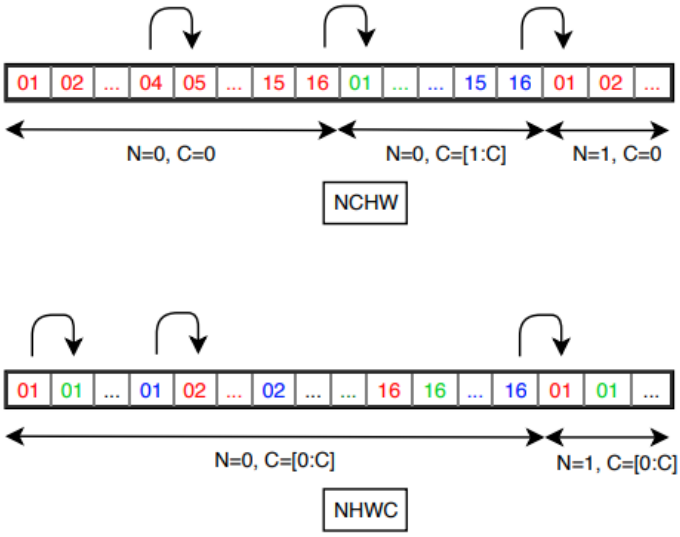

Figura 3.1: Orden de datos NCHW (arriba) y NHWC (abajo) [2].

El formato de los datos es realmente importante a la hora de implementar el algoritmo, ya que en función de qué orden escojamos, se tendrá que modelar la implementación y paralelización alrededor de ella. Para este trabajo se tendrán en cuenta las implementaciones de los formatos descritos anteriormente, "NCHW - CKRS - NKHW" y "NHWC - CRSK - NHWK".

3.1.2. Transformación elegida

Para la implementación se ha decidido implementar únicamente la transformación $F(2 \times 2, 3 \times 3)$ por dos razones principales:

- **Frecuencia**: La transformación con un filtro 3×3 es la más frecuentemente encontrada en redes neuronales convolucionales [17].

- **Tiempo**: Implementar una versión por cada una de las transformaciones de los tamaños más usados recientemente en redes neuronales convoluciones conlleva un tiempo de implementación y optimización que no sería despreciable.

3.1.3. Implementación base

En esta sección se mostrará el código/implementación mediante pseudocódigo de los principales componentes de transformación que se utilizarán en la implementación del algoritmo.

Transformación del filtro

En el Código 3.1, mostramos el código de transformación de un filtro. Nótese que la entrada viene dada en función de `filter(k,c,py,px)`. Esta es una forma genérica de acceder al filtro, donde `k` es el índice del kernel, `c` el índice del canal, `py` el índice fila y `px` el índice columna. Esto permite en estos ejemplos no especificar los índices de cada estructura de datos que implementamos. También se ha de notar que en la matriz de transformación W (2.4) se ha sacado factor común a $1/2$ con el objetivo de ahorrar operaciones. Además, se ha utilizado la operación multiplicar en vez de dividir por ser normalmente menos costosa en numero de ciclos. Un ejemplo lo encontramos en los procesadores de ARM Cortex-A57, donde podemos ver que la operación de dividir (VDIV) puede ejecutar dos instrucciones de división en punto flotante de 32 bits cada 5-15 ciclos en comparación con la de multiplicar (VMUL), que es capaz de ejecutar una multiplicación en punto flotante de 32 bits cada 2 ciclos [13].

La transformación del filtro se realizaría por cada filtro que hubiera, por lo tanto, K veces (el número de kernels) multiplicado por C (el número de canales).

```
1  //SEMI FILTER TRANSFORM (W*w)
2  for(int i=0; i<3;i++)
3  {
4      semi_transform[i] = filter(k,c,0,i);
5      semi_transform[3+i] = (filter(k,c,0,i) + filter(k,c,1,i) +
6                             filter(k,c,2,i)) * 0.5;
7
8      semi_transform[6+i] = (filter(k,c,0,i) - filter(k,c,1,i) +
9                             filter(k,c,2,i)) * 0.5;
10     semi_transform[9+i] =  filter(k,c,2,i);
11 }
12
13 //FILTER TRANSFORM (W*w*W')
14 for(int i=0; i<4;i++)
15 {
16     filter_transform[(0) + (i*(4))] = semi_transform[i*3];
17     filter_transform[(1) + (i*(4))] = (semi_transform[i*3] + semi_transform[1+(i*3)] +
18                                        semi_transform[2+(i*3)]) / 2.0;
19     filter_transform[(2) + (i*(4))] = (semi_transform[i*3] - semi_transform[1+(i*3)] +
20                                        semi_transform[2+(i*3)]) / 2.0;
21     filter_transform[(3) + (i*(4))] = semi_transform[2+(i*3)];
22 }
23
```

Código 3.1: Transformación de un filtro acorde a $F(2 \times 2, 3 \times 3)$.

Transformación de imagen

Siguiendo al filtro, la próxima transformación es la de la imagen, esta operación se detalla en el Código 3.2, que es similar a la transformación del filtro en que es la descomposición de sumas de, esta vez, la matriz X vista en (2.4). Símil al filtro, se aplica una forma genérica de acceder a la imagen de forma independiente a la estructura de datos usada, mediante `imagen(n,c,py,px)` donde `n` es el índice de la imagen, `c` el índice del canal y `py`, `px` son el índice fila y columna, respectivamente.

La transformación de la imagen se realizaría B veces (siendo 'B' el número de bloques en una imagen) multiplicado por C y por N (siendo 'C' el número de canales y 'N' el número de imágenes). El número de bloques en la imagen depende de varias cosas, principalmente del 'stride' (desplazamiento) y del ancho y alto de la imagen. Asumiendo 'stride' de uno, y una imagen de W columnas y H filas el número de secciones que tendría que tener acorde a una transformación $F(2 \times 2, 3 \times 3)$ sería $(((W-2)/2) \times ((H-2)/2)$, teniendo en cuenta que la salida de la convolución sería de tamaño `H-2, W-2`. En redes neuronales, se ha de tener en cuenta que el resultado de la operación de convolución tiene que tener el mismo ancho y alto por cada imagen que la imagen de entrada. Para mantener el tamaño de la imagen, se añade un borde de ceros a la imagen para que se anule la reducción total de tamaño. En el ejemplo $F(2 \times 2, 3 \times 3)$ se añadiría un borde de ceros de tamaño una columna a cada lado del ancho de la matriz imagen y tamaño una fila a cada lado del alto de la matriz imagen.

```
1   //SEMI IMAGE TRANSFORM (X'*x)
2   for(int i=0; i<4; i++)
3   {
4       semi_image_transform[i] = image(n,c,py,px+i) - image(n,c,py+2,px+i);
5       semi_image_transform[4+i] = image(n,c,py+1,px+i) + image(n,c,py+2,px+i);
6       semi_image_transform[8+i] = (- image(n,c,py+1,px+i)) + image(n,c,py+2,px+i);
7       semi_image_transform[12+i] = image(n,c,py+1,px+i) - image(n,c,py+3,px+i);
8   }
9
10  //FILTER TRANSFORM (X'*x*X)
11  for(int i=0; i<4; i++)
12  {
13      image_transform[i*2] = semi_image_transform[i*4] - semi_image_transform[(i*4)+2];
14      image_transform[(i*2)+1] = semi_image_transform[(i*4)+1] + semi_image_transform[(i*4)+2];
15      image_transform[(i*2)+2] = semi_image_transform[(i*4)+2] - semi_image_transform[(i*4)+1];
16      image_transform[(i*2)+3] = semi_image_transform[(i*4)+1] - semi_image_transform[(i*4)+3];
17  }
18
```

Código 3.2: Transformación de una sección de imagen acorde a $F(2 \times 2, 3 \times 3)$.

Optimización de datos contiguos en transformación de imagen

La implementación de transformación de imagen que se ve en el Código 3.2 es correcta en un principio, pero en la práctica suele ocurrir que los bloques de la imagen suelen ser contiguos, de forma que algunos de los datos se solapan. Esto da una oportunidad al ahorro en número de operaciones en el cálculo de la convolución teniendo en cuenta que la primera multiplicación matricial de la

transformación de la imagen es de esta forma $X \times x$, siendo X una matriz constante y x un bloque de la imagen. Podemos modificar esta expresión de forma que x, en vez de ser 'un' bloque de imagen, sean varios bloques contiguos, lo cual, en el caso de que se solapen de forma contigua, implicaría poder no hacer la misma operación sobre los mismos datos.

Esto se puede observar en la Figura 3.2, donde sí se aplica una transformación a dos secciones contiguas, se reutilizan datos, en concreto las columnas 3 y 4, por lo que se lleva a cabo la multiplicación matricial directamente sobre las imágenes contiguas a lo ancho y así nos ahorramos el cómputo repetido de esas columnas.

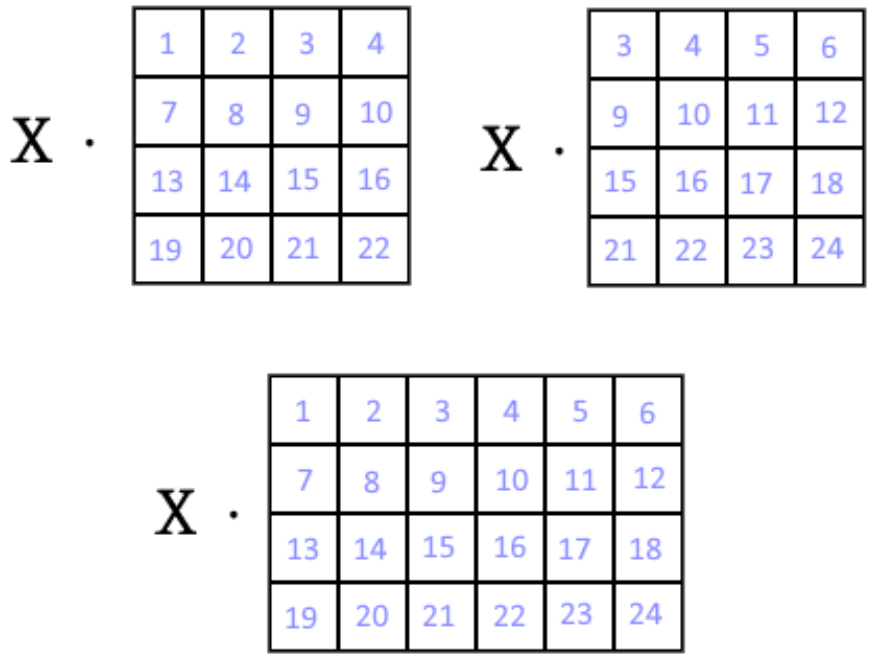

Figura 3.2: Imagen que representa cómo dos secciones de imagen contiguas (arriba) son equivalentes a operar el bloque entero (abajo) mediante la operación $X * x_{1,1\cdots n}$.

Transformación final

Finalmente, solo quedan dos operaciones, el producto elemento a elemento y acumulación y la transformación final. Una de las maneras de implementar el producto elemento a elemento y acumulación se ha descrito en la Sección 2.2, donde se tomaba la idea de implementarlo como un producto de matrices, mientras que otra más directa que no requiere reordenar ni guardar toda la transformación de imagen y filtro en memoria es hacer el producto de Hadamard y acumulación directamente por cada sección.

La transformación final, para obtener el resultado de la convolución en el dominio pretransformado consiste, igual que los otros dos, en descomponer la matriz en su conjunto de sumas, lo que se puede apreciar en el Código 3.3. La entrada 'transformed_convolution' es la matriz 4×4 resultado de multiplicar elemento a elemento y acumular los canales. Como resultado obtenemos la matriz 2×2 de la convolución.

```
// SEMI FINAL TRANSFORM Z'...
for(int i=0; i<4; i++)
{
    semi_res[i] = transformed_convolution[i] + transformed_convolution[i+4] +
                  transformed_convolution[i+8];
    semi_res[i+4] = transformed_convolution[i+4] - transformed_convolution[i+8] -
                  transformed_convolution[i+12];
}

// FINAL TRANSFORM Z'...Z
res[0] = semi_res[0] + semi_res[1] + semi_res[2];
res[1] = semi_res[1] - semi_res[2] - semi_res[3];
res[2] = semi_res[4] + semi_res[5] + semi_res[6];
res[3] = semi_res[5] - semi_res[6] - semi_res[7];

```

Código 3.3: Transformación final para obtener el resultado de la convolución con formato $F(2\times2, 3\times3)$.

3.1.4. Versión eficiente en memoria

Esta implementación consiste en una versión que sea lo más eficiente en memoria posible. Esto se consigue transformando una sola sección de imagen junto a todos sus canales de golpe. El pseudocódigo de la implementación se puede ver en el Código 3.4. Esta versión solo utiliza $K \times C \times 16$ (siendo K el número de kernels y C el de canales) datos en memoria correspondientes a la transformación del filtro. Esto es un consumo de memoria muy bajo en comparación con el resto de versiones, pero tiene el problema de no usar completamente las ventajas proporcionadas por el algoritmo de Winograd. Realizar transformaciones es más eficiente cuando el número de kernel y el número de secciones de imagen es grande, ya que se precalcula la transformación de filtro e imagen pudiendo reutilizar estos datos para el cálculo de convolución como se describe en la sección donde se explica el funcionamiento del algoritmo de Winograd, en la Sección 2.1. El problema de esta implementación es que se calcula la transformación de la imagen por cada kernel distinto presente en el filtro, por lo que la transformada de la imagen se tiene que calcular de nuevo por cada filtro que apliquemos. El objetivo de esta implementación es intercambiar un gasto menor en memoria por una repetición del cálculo de la transformación de imagen cuando hay más de un kernel implicando una operación menos eficiente.

```
filter_tn [K,C,16] = filter_transform ( filter ,K,C)
for(int n=0; n<N; n++)
for(int k=0; k<K; k++)
{
    transform[16] = 0;
    for(int c=0; c<C; c++)
    {
        image_tn[16] = image_transform(image,n,c)
        // Hadamard product and accumulate
        transform[16] += image_tn * filter_tn [k,i]
    }
```

```
12          res[n,k,4] = final_transform(transform)
13      }
```

Código 3.4: Pseudocódigo de implementación eficiente en memoria para la imagen de índice 'n' con convolución de sección $F(2 \times 2, 3 \times 3)$ (solo una sección de la imagen).

3.1.5. Versión de producto matricial

En esta implementación (Código 3.5), a costa de usar más memoria se incrementa la eficiencia obtenida drásticamente en implementación. Para esto transformamos el filtro y la imagen enteros, usando $K \times C \times 16$ (K siendo el número de kernels y C el de canales) datos para el filtro, $C \times ((H-2)/2) \times ((W-2)/2) \times 16$ elementos (siendo C el número de canales, H la altura de cada imagen, y W el ancho de cada imagen) para la transformación de la imagen y $K \times ((H-2)/2) \times ((W-2)/2) \times 16$ elementos para almacenar el cálculo del resultado de la multiplicación matricial. Esta implementación calcula la transformación de cada imagen en cada iteración y reordena los datos de la forma vista en la Sección 2.2, para calcular el producto elemento a elemento y acumulación como producto de matrices.

```
1   for(int n=0; n<N; n++)
2   {
3       filter_tn [C,16,K] = filter_transform ( filter ,K,C)
4       image_tn[H,W,C] = image_transform(image,n,C,H,W)
5       for(int z=0; z<16; z++)
6           transform[z,H,W,K] = gemm(image_tn, filter_tn)
7       res[n,K,H,W] = final_transform(transform)
8   }
```

Código 3.5: Pseudocodigo de implementación con uso de multiplicación matricial (para la imagen 'n'esima).

Bibliotecas de computo matricial

Al implementar la versión que se aprovecha del producto matricial, inicialmente se utilizó la biblioteca `libblas-dev/bionic,now 3.7.1-4ubuntu1 arm64`, una compilación estática de BLAS [11] disponible por defecto en los repositorios de Ubuntu de las maquinas usadas para el desarrollo (sus características se pueden ver en la Sección 4.1). Después de implementar la convolución usando la gemm (GEneral Matrix Multiply, la multiplicación matricial) para tipos de dato float, se notó que el rendimiento de esta llamada no era ideal teniendo en cuenta el procesador usado. Para confirmar que el rendimiento era mejorable, se decidió hacer algunas pruebas con esta biblioteca y comparar los resultados con otra (BLIS [8]) para contrastar el rendimiento. Para contrastar el resultado se decidió realizar varias pruebas de producto matricial ($A \cdot B = C$) con datos de simple precisión, tres de las cuales son:

- **Test 1.** Un solo producto, con 1225 filas de A, 2000 columnas de B y 2000 elementos en la dimensión común.

- **Test 2.** Veinte repeticiones del producto, cada uno de matrices con 1024 filas de A, 512 columnas de B y 512 elementos en la dimensión común.
- **Test 3.** Mil repeticiones del producto, cada uno de matrices con 100 filas de A, 64 columnas de B y 64 elementos en la dimensión común.

Como se puede ver en los resultados de la Figura 3.3 BLIS obtiene aproximadamente una mejora de diez veces de rendimiento respecto a libblas, confirmando la sospecha del rendimiento ineficiente del uso inicial de la biblioteca. Por lo tanto, para el desarrollo del trabajo se utilizará la biblioteca BLIS [8].

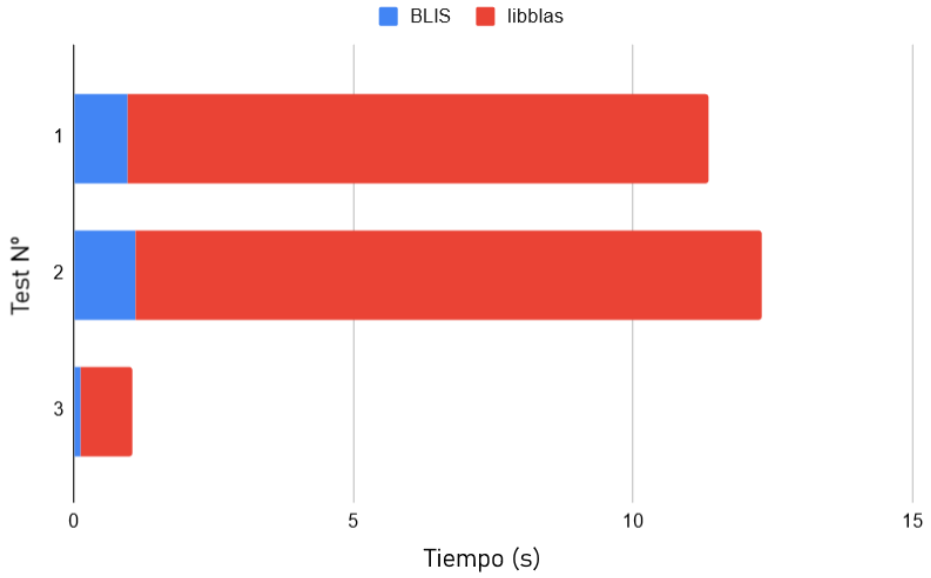

Figura 3.3: Gráfico de barras que compara el rendimiento de tres casos de prueba ejecutados con las librerías BLIS y libblas.

Colisión en lectura de datos en cache

En la implementación de la versión mediante producto matricial se observó una anomalía cuando se ejecutaba el programa con ciertos tamaños de entrada. Se llegó a la conclusión de que era una alineación de cache que implicaba más fallos de lo usual, esto ocurría por el formato de salida del producto de matrices, que como podemos ver en la Figura 3.4, separa cada uno de los 16 elementos del resultado de la convolución transformada por 'n' datos (siendo 'n' el número de elementos en el resultado del producto de matrices, $K \times ((H-2)/2) \times ((W-2)/2) \times 16$ elementos). Esto ocurría cuando el número de datos por cada bloque de salida del producto matricial era aproximadamente múltiplo o divisor del tamaño de la cache, ya que, si no era una colisión entre el elemento '1' y el '2', '2' y '3'... podría ser entre el '1' y el '3', '3' y '5'... o '1' y '8'... Este problema se resolvió añadiendo espacio 'vacío' en memoria entre cada bloque, para que al direccionar la línea de cache correspondiente, ya no coincidiera en los casos que antes se veía la alineación. De esta forma cada bloque está desplazado $(i \times 16)$ elementos respecto al anterior, siendo 'i' el elemento del bloque actual (1 a 16 en $F(2 \times 2, 3 \times 3)$), se eligió desplazar 16 elementos, ya que al ser el tipo de datos 'float', son 64 bytes, que justo coincide con el tamaño de cada linea de cache. Cabe notar que, aunque en las pruebas hechas utilizando este método se ha notado una mejora significativa en los casos de

pruebas que sin él eran poco eficientes. Aun así, no se ha indagado y estudiado este fenómeno con más detención, por lo que se expone como una implementación a tener en cuenta. No se conoce el alcance de las consecuencias o mejora que este cambio tiene, por esto se dejara como trabajo futuro su estudio.

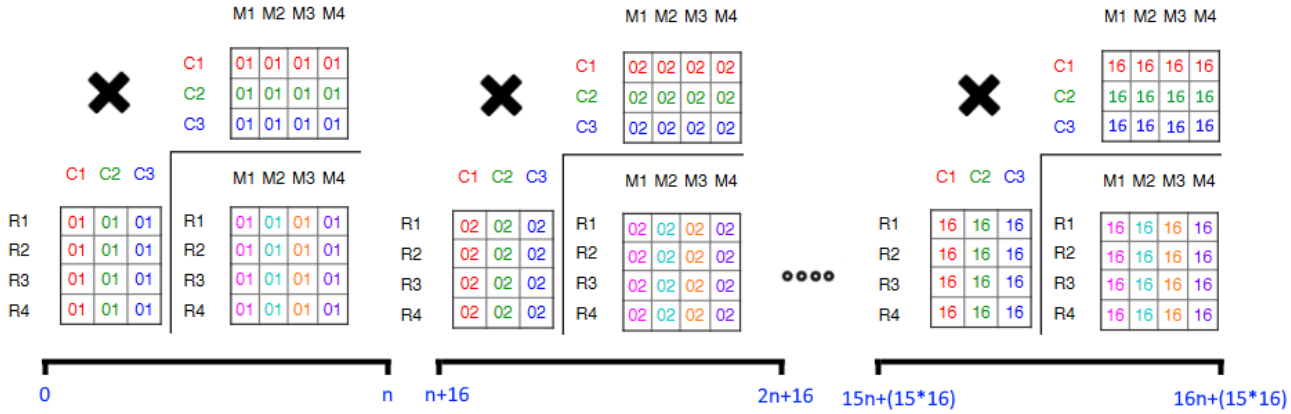

Figura 3.4: Imagen que muestra las posiciones de memoria donde empiezan y acaban los bloques resultado de cada producto de matrices con distancia introducida para evitar la colisión de cache.

3.1.6. Versión de producto matricial con imágenes contiguas

Esta versión consume más memoria que la versión de producto matricial y es algo más eficiente en algunos casos que ella. Para implementar esta versión se reordenan los datos de nuevo, de la forma que vemos en la Figura 3.5, posicionando la transformación de cada imagen contigua en filas a la anterior, de esta forma, en vez de hacer n productos matriciales por cada imagen, solo hacemos uno con matrices más grandes. El coste en memoria de esta implementación es bastante grande, de forma que para el filtro seguimos necesitando $K \times C \times 16$ elementos, para la transformada de la imagen $N \times C \times ((H-2)/2) \times ((W-2)/2) \times 16$ y para el resultado del producto matricial $N \times K \times ((H-2)/2) \times ((W-2)/2) \times 16$. El pseudocódigo que podemos ver en el Código 3.6 es muy similar al visto en la versión que no realizaba el producto de todas las imágenes de golpe, solo que, en este caso, en la transformación de la imagen se transforman todas en la misma llamada de forma que desaparece el bucle que itera por cada imagen.

Esta versión tiene dos principales ventajas respecto a la de producto matricial en respecto a eficiencia de su ejecución. La primera es cuando el ancho y alto de las matrices de entrada es pequeño, por lo tanto, generando pocos bloques de imagen implicando no alcanzar el pico de eficiencia en el uso de la gemm por el tamaño de las matrices. En esta versión, al transformar las imágenes de forma contigua implica que a la hora de realizar la gemm la dimensión de filas de A ($A \cdot B = C$) sea más grande (ahora $Nbloques \times N$) de lo que sería si operáramos de imagen en imagen. El agrandar las dimensiones da mayores posibilidades a que la librería de cómputo matricial que usamos de su máximo rendimiento. Por otro lado, como se describe en la Sección 3.3.3 de paralelización, esta versión permite una mayor paralelización cuando el tamaño del batch de imágenes de entrada no es lo suficientemente grande. Aun así, cabe notar que en esta versión no incluimos ninguna técnica para evitar la colisión de cache vista en la Sección 3.1.5, esto es porque en esta versión se necesitaría meter espacio antes del producto matricial y no después incrementando el número de operaciones realizadas por la gemm. Por esto se decide no aplicar esta técnica, y dejar como trabajo futuro

el estudiar en más profundidad si es ventajoso incluir este espaciado u otra técnica que mejore el rendimiento.

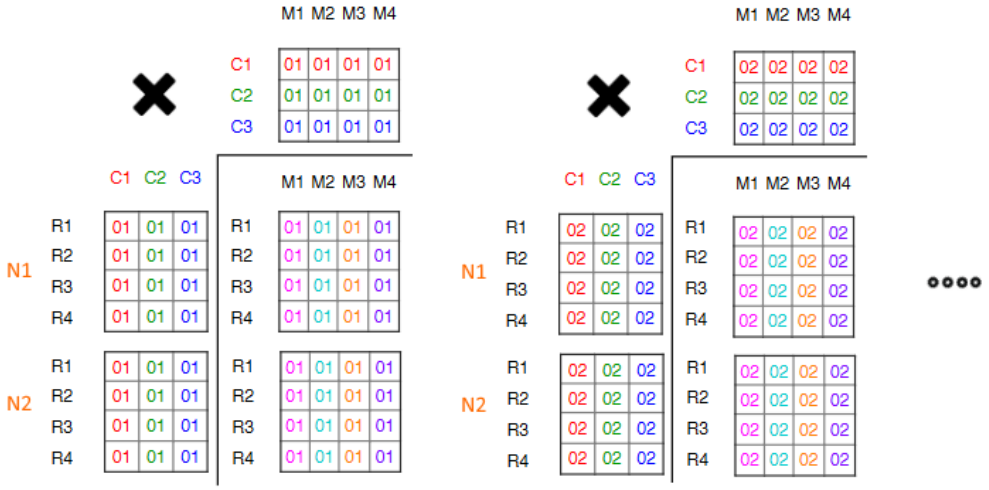

Figura 3.5: Estructura de datos de la transformada de imagen y filtro para realizar el producto matricial sobre un conjunto de imágenes.

```
filter_tn [C,16,K] = filter_transform ( filter ,K,C)
image_tn[H,W,C] = image_transform(image,N,C,H,W)
for(int z=0; z<16; z++)
    transform[z,H,W,K] = gemm(image_tn, filter_tn)
res[n,K,H,W] = final_transform(transform)
```

Código 3.6: Pseudocódigo de la implementación mediante producto matricial de imágenes contiguas.

3.1.7. Resolviendo el problema de la paridad

Como se ha visto en otros apartados, el aplicar una convolución $F(2 \times 2, 3 \times 3)$, produce una salida 2×2. Esto significa que, sin repetir el cálculo de datos del resultado, solo podemos calcular la convolución sobre imágenes que tengan H (alto) y W (ancho) par. Esto es un problema, ya que en el caso de que se quieran procesar imágenes de tamaño impar, se tendría que preprocesar la imagen, añadiendo bordes de valor cero para llegar a un tamaño par. Para resolver esto, se pensó en dos métodos sencillos:

- **Calcular el resto mediante convolución directa**: Este método consiste en calcular mediante el algoritmo de Winograd la convolución de la imagen con tamaño par inmediatamente menor y el resto mediante una convolución directa. Esto tiene la consecuencia de tener que implementar una versión de convolución directa eficiente para que no sea un cuello de botella.
- **Repetir el cálculo resultado de algunos datos**: Este cambio consiste en hacer la convolución de un bloque tal cual se hacía antes en $F(2 \times 2, 3 \times 3)$ hasta el tamaño par inmediatamente menor al tamaño de la imagen. Después se calculan los bordes impares mediante otra convolución del mismo tamaño de sección que sobrescribirá datos ya calculados. Las secciones de imagen que se tomaría en el caso de impar se pueden observar en la Figura 3.6, así como una comparación de un tamaño par que no calcularía el resultado de algunos elementos por

duplicado. Esta solución no es la más eficiente, pero es una aproximación que permite usar todas las optimizaciones usadas en la implementación de las transformaciones, así como la paralelización, con el inconveniente de repetir algunas operaciones.

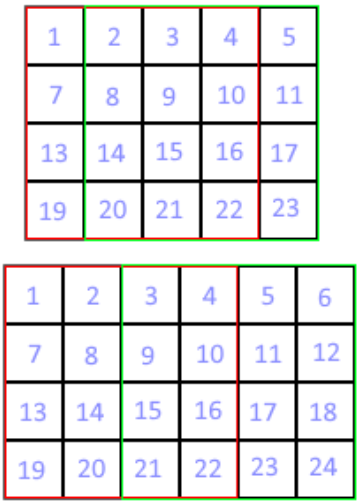

Figura 3.6: Secciones de imagen cuando se trata con una versión impar (arriba) comparado con una par (abajo).

3.2. Paralelización a nivel de instrucción

En esta sección se verán las paralelizaciones a nivel de instrucción realizadas para las diferentes implementaciones del algoritmo de Winograd. Para esto, se utilizan las funciones intrínsecas de la extensión NEON para CPU de ARM. Se implementará una versión distinta por cada estructura de entrada de los datos vista en la Sección 3.1.1, ya que para maximizar la eficiencia se cargarán los datos de forma contigua, y la dimensión contigua en memoria varía en función de la estructura de datos empleada. Las secciones principales a paralelizar mediante instrucciones SIMD son la transformación de la imagen y la transformación final para obtener el resultado. Estas son las dos transformaciones que más impacto tienen en el rendimiento de la implementación.

3.2.1. Transformación de la imagen

La paralelización mediante SIMD de la transformación de la imagen depende del formato de entrada de imagen, en este caso 'NCHW' y 'NHWC'. Como se verá a continuación el uso de SIMD en 'NCHW' se puede usar en cualquier caso de entrada, mientras que el uso de SIMD en 'NHWC' no se puede aprovechar para todos los casos de entrada, pero en los que sí, es mejor que su contraparte 'NCHW'.

NCHW

La estructura NCHW implica que el ancho de la imagen está contiguo en memoria (ya que es la ultima dimensión). En función a esto, realizamos la vectorización cargando una matriz 4×4 de la imagen en cuatro registros, de forma que tenemos una fila en cada registro. Una vez tenemos esto,

podemos hacer una mitad de la transformación de la imagen, aprovechándonos de que las operaciones a realizar se comparten en todas las columnas. Esta operación se puede ver en el Código 3.7.

```
1        input[0] = vld1q_f32(&image(n,c,py,px));
2        input[1] = vld1q_f32(&image(n,c,py+1,px));
3        input[2] = vld1q_f32(&image(n,c,py+2,px));
4        input[3] = vld1q_f32(&image(n,c,py+3,px));
5
6        // SEMI IMAGE TRANSFORM (X'*x)
7        semi_image_transform[0] = vsubq_f32(input[0], input[2]);
8        semi_image_transform[1] = vaddq_f32(input[1], input[2]);
9        semi_image_transform[2] = vsubq_f32(input[2], input[1]);
10       semi_image_transform[3] = vsubq_f32(input[1], input[3]);
11
```

Código 3.7: Primera mitad de la transformación de la imagen, paralelizado mediante instrucciones SIMD.

El cargar la imagen por filas en cuatro registros tiene un problema. Para completar el resto de la transformación hace falta operar por columnas y no hay ninguna instrucción que permita sumar/restar columnas de cuatro registros distintos en el conjunto de instrucciones SIMD de ARM (al menos dentro de la extensión NEON). Por esto tenemos que transponer la matriz, para poder operar por filas de nuevo como en la primera mitad de la transformación. Para transponer la matriz 4×4 usamos uno de los métodos mostrados por Fabian "ryg" Giesen en su blog [12]. Este explica cómo, para arquitecturas ARM con la extensión NEON, se puede usar la operación `interleave64` (instrucción `VUZPQ` en ARM) seguida de la operación `VTRN` para transponer una matriz 4×4 usando cuatro instrucciones para hacer el movimiento. El código correspondiente con la traspuesta de la primera mitad de la transformación de la imagen se puede ver en el Código 3.8.

```
1        // trans1, trans2, input2_1 & input2_2 are defined as float32x4x2_t
2        trans1 = vuzpq_f32(semi_image_transform[0], semi_image_transform[2]);
3        trans2 = vuzpq_f32(semi_image_transform[1], semi_image_transform[3]);
4        input2_1 = vtrnq_f32(trans1.val[0], trans2.val[0]);
5        input2_2 = vtrnq_f32(trans1.val[1], trans2.val[1]);
6
```

Código 3.8: Transponer una matriz guardada en registros vectoriales por filas mediante instrucciones SIMD (Entrada: semi_image_transform, Salida: input2_1 y input2_2).

Una vez se tiene la matriz traspuesta, solo hace falta operar por filas mediante las funciones intrínsecas adecuadas a la transformación, esto se puede ver al inicio del Código 3.9. El haber traspuesto la matriz anteriormente implica que el resultado también este traspuesto. Esto se puede resolver mediante una operación que guarda en memoria una columna de un tipo de dato `float32x4x4_t` (4 registros de 128 bits, formando una matriz 4×4 de floats) de forma contigua en la memoria destino. Esta operación, aunque cabe destacarla, no es útil en la práctica ya que

solo la usaríamos para la versión que utiliza producto matricial (ya que en la 'eficiente en memoria' se podría mantener en los registros SIMD), y la estructura de entrada del producto matricial no mantiene los datos de la transformación de cada sección contiguos (ver la Figura 2.1), por lo que se podrían escribir los datos de forma traspuesta a un buffer, y en la segunda escritura al formato que debe tener de entrada la gemm, reordenar los datos.

```
        float32x4x4_t  buff_ex  = { vsubq_f32(input2_1.val[0],  input2_1.val[1]),
                vaddq_f32(input2_2.val[0],  input2_1.val[1]),
                vsubq_f32(input2_1.val[1],  input2_2.val[0]),
                vsubq_f32(input2_2.val[0],  input2_2.val[1])  };

        vst4q_lane_f32(buff,  buff_ex,  0);
        vst4q_lane_f32(&buff[4],  buff_ex,  1);
        vst4q_lane_f32(&buff[8],  buff_ex,  2);
        vst4q_lane_f32(&buff[12],  buff_ex,  3);

```

Código 3.9: Último tramo de transformación de imagen utilizando SIMD.

NHWC

En el caso de NHWC, al ser el canal la dimensión contigua, en vez de paralelizar una sola transformación paralelizaremos la de cuatro canales a la vez. La forma de hacer esto es muy similar a la mostrada en el Código 3.2, pero esta vez utilizando de las instrucciones SIMD, como se puede observar en el Código 3.10. El hecho de realizar la operación sobre varios canales a la vez tiene la desventaja de necesitar un número de canales de entrada múltiplo de cuatro, ya que los registros que usamos cargan cuatro valores de punto flotante. En el caso de que no sea múltiplo de cuatro, el resto será calculado mediante la transformación no SIMD vista en el Código 3.2. Aunque esto sea una desventaja notable para casos de entrada determinados, cabe notar que, cuando se usan las instrucciones SIMD, es más rápido que la versión NCHW, ya que, por un lado, no tiene que usar instrucciones de transposición de matriz y, además, puede guardar los datos directamente en memoria desde el registro SIMD, ya que están alineados con el formato de datos que toma como entrada el producto matricial.

```
    for(int  i=0; i<4; i++)
        for(int  j=0; j<4; j++)
            input[(i*4)+j] = vld1q_f32(&imageNHWC(n,c,py+i,px+j,   W,H,C));

    for(int  i=0; i<4; i++)
    {
        semi_image_transform_v[i]  = vsubq_f32(input[i]  ,  input[i+8]);
        semi_image_transform_v[i+4] = vaddq_f32(input[i+4],  input[i+8]);
        semi_image_transform_v[i+8] = vsubq_f32(input[i+8],  input[i+4]);
        semi_image_transform_v[i+12] = vsubq_f32(input[i+4],  input[i+12]);
```

```
11      }
12
13      for(int  i=0; i<16; i+=4)
14      {
15          vst1q_f32(  (image_transform + (  bias  + (i*despl))  )  ,
16              vsubq_f32(semi_image_transform_v[i],  semi_image_transform_v[i+2]));
17          vst1q_f32(  (image_transform + (bias  + ((i+1)*despl)) ),
18              vaddq_f32(semi_image_transform_v[i+1], semi_image_transform_v[i+2]));
19          vst1q_f32(  (image_transform + (bias  + ((i+2)*despl))  )  ,
20              vsubq_f32(semi_image_transform_v[i+2], semi_image_transform_v[i+1]));
21          vst1q_f32(  (image_transform + (bias  + ((i+3)*despl))  )  ,
22              vsubq_f32(semi_image_transform_v[i+1], semi_image_transform_v[i+3]));
23      }
24
```

Código 3.10: Pseudocódigo de la transformación de imagen mediante SIMD para formato NHWC.

3.2.2. Producto elemento a elemento y acumulación

Para las implementaciones eficientes en memoria (las que no hacen uso del producto matricial) es necesario realizar el producto elemento a elemento y acumulación sobre los canales. Esto se realiza mediante un bucle que itera elemento a elemento y hace una operación 'FMA' (multiplicación y acumulación conjunta). Para el formato NCHW se realiza el producto sobre los cuatro registros en los que se almacena la imagen (ver el Código 3.11), mientras que en NHWC se realiza lo mismo, pero sobre los dieciséis elementos de transformación.

```
1       for(int  zz=0; zz<4; zz++)
2           transformed_convolution.val[zz]  = vfmaq_f32(transformed_convolution.val[zz]  ,
3                                                 image_transform.val[zz],    filter_transform_i  .val[zz]);
4
```

Código 3.11: Pseudocódigo del producto elemento a elemento y acumulación mediante SIMD.

3.2.3. Transformación del filtro

Para transformar el filtro, al igual que la imagen, se tienen dos estructuras: CRSK y CKRS. Para implementar SIMD en ambas transformaciones, se realizaría de forma similar a la de la imagen. La versión con formato CRSK se realizaría igual que la transformación sin SIMD transformando cuatro kernels de forma simultánea, mientras que la CKRS se realizaría cargando una fila de una matriz de transformación en tres registros SIMD. Una vez cargada en los tres registros, se calcularía la primera mitad de la transformación (Ww) y se realizaría la traspuesta para, finalmente, calcular el resto de la transformación (WwW^T). Esta configuración es similar a la usada en la transformación de la imagen para el formato NCHW. Utilizar SIMD en la implementación CKRS, además de ser más ineficiente por el hecho de tener que trasponer los resultados, también es, en el caso del filtro, aún más ineficiente, ya que, para la primera mitad de la transformación solo usaríamos 3 columnas de las 4 que caben en los registros. Por estas dos cosas se decide no implementar la transformación del filtro mediante SIMD en la estructura de datos CKRS, ya que, aunque resulta ser más rápida que el

código secuencial, sería una diferencia poco notable. Solo se implementa la paralelización mediante instrucciones SIMD del formato CRSK, este se puede ver en el Código 3.12.

```
float32x4_t  half = vmovq_n_f32(0.5);
for(int i=0; i<3; i++)
    for(int j=0; j<3; j++)
        input[i*3 + j] = vld1q_f32(&filterCRSK(k,c,i,j, S,R,K));

// SEMI FILTER TRANSFORM (W*w)
for(int i=0; i<3; i++)
{
     semi_filter_transform_v [i] = input[i];
     semi_filter_transform_v [i+9] = input[i+6];
     semi_filter_transform_v [i+3] = vmulq_f32( vaddq_f32( vaddq_f32(input[i+0], input[i+3]),
                                                  input[i+6] ),  half);
     semi_filter_transform_v [i+6] = vmulq_f32( vaddq_f32( vsubq_f32(input[i+0], input[i+3]),
                                                  input[i+6] ),  half);
}
// FILTER TRANSFORM (W*w*W')
for(int i=0; i<4; i++)
{
    vst1q_f32(& filter_transform (k,0,i,c),  semi_filter_transform_v [i*3]);
    vst1q_f32(& filter_transform (k,3,i,c),  semi_filter_transform_v [(i*3)+2]);
    vst1q_f32(& filter_transform (k,1,i,c), vmulq_f32( vaddq_f32( vaddq_f32(
                semi_filter_transform_v [i*3],  semi_filter_transform_v [i*3+1]),
                semi_filter_transform_v [i*3+2] ) , half) );
    vst1q_f32(& filter_transform (k,2,i,c), vmulq_f32( vaddq_f32( vsubq_f32(
                semi_filter_transform_v [i*3],  semi_filter_transform_v [i*3+1]),
                semi_filter_transform_v [i*3+2] ), half) );

}
```

Código 3.12: Pseudocódigo de la transformación del filtro con instrucciones SIMD en formato CRSK.

3.2.4. Transformación final

Eficiente en memoria NCHW

Como se ha visto, la versión eficiente en memoria no usa el producto matricial para operar al realizar la convolución, y calcula solo una sección 2×2 de salida por cada iteración. En concreto, la versión NCHW guarda la matriz de convolución transformada en cuatro registros SIMD. Se puede realizar la transformación para obtener el resultado final directamente sobre estos registros. Como se puede ver en el Código 3.13, la primera mitad de la transformación se realiza mediante instrucciones SIMD, pero el resto no. Se realiza la mitad de la transformación en estos registros ya que hacerla entera implicaría usar 4 operaciones para transponer la matriz, y luego 4 para realizar la transformación, lo cual da un total de 8 instrucciones, exactamente las mismas que hacerlo sin usar SIMD.

```
// SEMI FINAL TRANSFORM (Z'(...))
float32x4x2_t semi_r = { vaddq_f32( vaddq_f32( transformed_convolution.val[0],
                                               transformed_convolution.val[1] ), transformed_convolution.val[2] ) ,
                         vsubq_f32( vsubq_f32( transformed_convolution.val[1],
                                               transformed_convolution.val[2] ), transformed_convolution.val[3] )
                         };


vst1q_f32(semi_res, semi_r.val[0]);
vst1q_f32(semi_res+4, semi_r.val[1]);

// FINAL TRANSFORM (Z'(...)Z)
res(w,h,k,n) = semi_res[0] + semi_res[1] + semi_res[2];
res(w+1,h,k,n) = semi_res[1] - semi_res[2] - semi_res[3];
res(w,h+1,k,n) = semi_res[4] + semi_res[5] + semi_res[6];
res(w+1,h+1,k,n) = semi_res[5] - semi_res[6] - semi_res[7];

```

Código 3.13: Pseudocódigo de la transformación final de la convolución con instrucciones SIMD en formato NCHW.

Producto matricial o eficiente en memoria NHWC

Para el resto de implementaciones se utiliza una paralelización por la dimensión del kernel, esto es porque la salida del producto de matrices y la versión NHWC eficiente en memoria produce una salida de datos donde el kernel es la primera dimensión. Esto permite una paralelización bastante eficiente en el caso de que el número de kernels sea mayor o igual que cuatro por las mismas razones por las que la implementación SIMD de la transformación de la imagen NHWC era más eficiente siempre y cuando se tuvieran cuatro o más canales. El pseudocódigo de la implementación de esta transformación se puede ver en el Código 3.14, este código solo se ejecutará cuando haya más de tres kernels, pero cuando se ejecuta, no tiene que realizar trasposición, haciéndola más eficiente que la transformación final de la versión eficiente en memoria NCHW.

```
for(int i=0; i<16; i++)
    input[i] = vld1q_f32(&transformed_convolution(k,c,h,w,i));
// SEMI FINAL TRANSFORM (Z'(...))
for(int i=0; i<4; i++)
{
    semi_res_v[i] = vaddq_f32( vaddq_f32(input[i], input[i+4]), input[i+8] );
    semi_res_v[i+4] = vsubq_f32( vsubq_f32(input[i+4], input[i+8]), input[i+12] );
 }
// FINAL TRANSFORM (Z'(...)Z)
semi_res_v[0] = vaddq_f32( vaddq_f32(semi_res_v[0], semi_res_v[1]), semi_res_v[2]);
semi_res_v[1] = vsubq_f32( vsubq_f32(semi_res_v[1], semi_res_v[2]), semi_res_v[3]);
semi_res_v[4] = vaddq_f32( vaddq_f32(semi_res_v[4], semi_res_v[5]), semi_res_v[6]);
semi_res_v[5] = vsubq_f32( vsubq_f32(semi_res_v[5], semi_res_v[6]), semi_res_v[7]);

```

Código 3.14: Pseudocódigo de la transformación final de la convolución con instrucciones SIMD para las versiones de producto matricial o eficiente en memoria con estructura NHWC.

Finalmente, la escritura de datos cambia de la implementación NCHW a la NHWC, esto se debe al formato que toman los datos al ser escritos. En NHWC, la escritura ocurre en formato NHWK, por lo que se puede escribir de forma contigua, tal como se ve en el Código 3.16. La versión NCHW escribe el resultado en formato NKHW, por lo que no puede escribir todos los datos en la misma instrucción, teniendo que acceder a ellos de forma individual, tal como se ve en el Código 3.15.

```
1   for (int i=0; i < 4; i++)
2   {
3       res(w,h, k+i,n) = vgetq_lane_f32(semi_res_v [0], zz);
4       res(w+1,h,k+i,n) =vgetq_lane_f32(semi_res_v [1], zz);
5       res(w, h+1,k+i,n) = vgetq_lane_f32(semi_res_v [4], zz);
6       res(w+1,h+1,k+i,n) = vgetq_lane_f32(semi_res_v[5], zz);
7   }
8
```

Código 3.15: Instrucciones de escritura a memoria usadas en las versiones NCHW de la transformación final implementada mediante SIMD.

```
1   vst1q_f32(res + ((k) + (w*K) + (h*(W-2)*K) + index), semi_res_v[0]);
2    vst1q_f32(res + ((k) + ((w+1)*K) + ((h)*(W-2)*K) + index), semi_res_v[1]);
3    vst1q_f32(res + ((k) + ((w)*K) + ((h+1)*(W-2)*K) + index), semi_res_v[4]);
4    vst1q_f32(res + ((k) + ((w+1)*K) + ((h+1)*(W-2)*K) + index), semi_res_v[5]);
5
```

Código 3.16: Instrucciones de escritura a memoria usada en la versión mediante producto matricial NHWC de la transformación final implementada mediante SIMD.

3.3. Paralelización a nivel de hilo

En esta sección se propondrán implementaciones de paralelización a nivel de hilo usando la especificación 4.5 de OpenMP [9] para las implementaciones vistas en la sección anterior. Además, se analizarán las ventajas o inconvenientes de cada una de ellas.

3.3.1. Eficiente en memoria

En las versiones eficientes en memoria están presentes varios bucles anidados, por lo que paralelizar el programa se convierte en una tarea más sencilla. Como se puede ver en el Código 3.17, utilizamos, para la versión NCHW, una paralelización colapsada de los primeros dos bucles, el del número de imágenes y kernels. Para los casos en los que estos primeros bucles tengan suficientes iteraciones como para hacer una por hilo, se priorizarán estas, pero en el caso de que esto no sea posible, se realizará la segunda paralelización de bucle que itera por cada fila de la imagen. Para paralelizar el bucle de la imagen hace falta activar el modo anidado de OpenMP, que nos permitirá definir

regiones paralelas dentro de otras regiones paralelas. Sin embargo, en la práctica no se realiza porque sería ineficiente. Solo lo usamos para poder declarar que, en caso de que no se active la primera región paralela, entonces salte a la segunda. La versión NHWC eficiente en memoria es similar, pero reordenando los bucles de forma que los primeros dos son N y H mientras que los dos segundos son W y K. En esta versión solo se paralelizan los primeros dos bucles de forma colapsada para evitar el anidamiento y porque es poco común realizar una convolución donde H multiplicado por N no sea mayor que el número de hilos.

```
1  omp_set_nested(1);
2  #pragma omp parallel if ((N*K) >= omp_get_num_threads())
3  #pragma omp for collapse(2)
4   for(int n=0; n < N; n++)
5       for(int k=0; k < K; k++)
6       {
7           float32x4x4_t filter_transform_i = {  vld1q_f32( filter_transform +(16*C*k)),
8                                                 vld1q_f32( filter_transform +(16*C*k)+4),
9                                                 vld1q_f32( filter_transform +(16*C*k)+8),
10                                                vld1q_f32( filter_transform +(16*C*k)+12) };
11          #pragma omp parallel for if ((N*K) < omp_get_num_threads())
12          for(int py=0; py < H-2; py+=2)
13              for(int px=0; px < W-2 ; px+=2)
14                  // Winograd transform one image section ...
15      }
16
```

Código 3.17: Paralelización mediante Openmp de la versión eficiente en memoria NCHW.

3.3.2. Producto matricial

La paralelización de las versiones de producto matricial solo se efectúa en el bucle de la dimensión de N, como se puede ver en el Código 3.18. Al paralelizar el bucle que transforma cada imagen, se asegura siempre la paralelización de la región que más carga tiene, pero cuando el tamaño del conjunto de imágenes a transformar es menor que el número de hilos, la paralelización no exprime su potencial. Se decide dejar solo la paralelización a nivel de cada imagen, además, porque la paralelización para la versión del producto matricial de imágenes contiguas consigue paralelizar los casos que esta versión no puede de forma más eficiente.

```
1   #pragma omp for
2  for(int n=0; n < N; n++)
3  {
4      // Image transform ...
5      // Matrix product ...
6      // Final transform ...
7  }
8
```

Código 3.18: Paralelización mediante Openmp de la versión producto matricial.

3.3.3. Producto matricial de imágenes contiguas

Una de las ventajas que tiene esta versión es su paralelización, ya que, al dividir cada uno de los pasos para obtener el resultado de la convolución, ocurren dos fenómenos beneficiosos para la paralelización. Por un lado, se puede paralelizar cada paso del algoritmo por separado y, por otro, crea un mayor anidamiento directo de bucles, por lo que nos aprovechamos de la directiva `collapse` de OpenMP para paralelizar bucles anidados, tal como se puede ver en el Código 3.19. Al colapsar el bucle, paralelizamos las iteraciones sobre filas de las transformaciones, permitiendo que, independientemente del número de kernels, el ancho y el número de canales, la paralelización sea eficiente mientras el número de imágenes multiplicado por el de filas de cada imagen sea mayor que el número de hilos. Los hilos tienen así una carga de trabajo razonable en función de la carga de cómputo a ejercer en función de la entrada.

```
1   // Image transform
2   #pragma omp for collapse(2)
3   for(int n=0; n<N; n++)
4   for(int py=0; py < H-3; py+=2)
5       ....
6
7   // Matrix product
8   #pragma omp for
9   for(int i=0; i < 16; i++)
10      ....
11
12  // Final transform
13  #pragma omp for collapse(2)
14  for(int n=0; n<N; n++)
15  for(int h=0; h < (H-2); h+=2)
16      ....
17
```

Código 3.19: Paralelización mediante Openmp de la versión producto matricial de imágenes contiguas.

Capítulo 4

Resultados

En este capítulo se muestran los resultados obtenidos mediante las implementaciones propuestas en el trabajo. Se compara el rendimiento entre versiones mediante unas pruebas diseñadas para mostrar las ventajas e inconvenientes de cada una de ellas. Además, se realiza un análisis del grado de paralelización de cada versión en función de la entrada proporcionada. Finalmente, se realiza una comparación con un método popular de cálculo rápido de convolución, el producto matricial de una transformación `im2row`.

4.1. Entorno de pruebas

Como entorno de desarrollo se han empleado dos máquinas, una 'NVIDIA Jetson Nano (4GB)' y una 'NVIDIA Jetson Xavier AGX'. La Jetson Nano se ha utilizado principalmente para la implementación del algoritmo y su paralelización mediante instrucciones SIMD ya que, durante su uso, se han observado ejecuciones más estables permitiendo un análisis de rendimiento de carácter más fino (pudiendo comparar pequeños cambios en el programa) en estas etapas de la implementación. La máquina Jetson Xavier se ha empleado para la etapa de implementación de la paralelización por su mayor número de núcleos de procesador a pesar del comportamiento algo menos estable. Para los resultados se ha utilizado la plataforma Jetson Xavier ya que los casos de prueba son lo suficientemente grandes como para que la menor estabilidad no se aprecie al comparar resultados y permite tomar resultados de paralelización con un mayor numero de hilos. Las especificaciones de ambos sistemas se pueden ver en la Tabla 4.1.

Tabla 4.1: Tabla que muestra especificaciones del entorno de desarrollo y pruebas usados.

	NVIDIA Jetson Nano	**NVIDIA Jetson Xavier AGX**
Modelo (CPU)	ARM - Cortex-A57	ARM - ARMv8 Processor rev 0 (v8l)
Núcleos (CPU)	4	8
Frecuencia (min-máx) (CPU)	102 MHz - 1479 MHz	115,2 MHz - 2265,6 MHz
Caché L1d (CPU)	32 KiB	64 KiB
Caché L2 (CPU)	2048 KiB	2048 KiB
Caché L3 (CPU)	—	4096 KiB
Memoria RAM	4 GiB	32 GiB
Sistema operativo	Ubuntu 18.04.5 LTS	Ubuntu 18.04.5 LTS

4.2. Metodología experimental

Para obtener los resultados de cada prueba, se llevan a cabo tres ejecuciones por cada tamaño de entrada, dando como resultado la media de estas con el objetivo de reducir la variabilidad entre ejecuciones. Se ejecutan todas las pruebas de golpe teniendo en cuenta que cada repetición sobre la misma entrada se hará una vez se hayan procesado todas las versiones del algoritmo. Un ejemplo consistiría en ejecutar la versión 1 sobre la entrada 1, ejecutar la versión 2 sobre la entrada 1 y luego la segunda repetición ejecutaría lo mismo otra vez. El mecanismo para realizar estas ejecuciones seguidas es un programa escrito en C.

4.2.1. Escenarios

Los escenarios sobre los que se realizaran el análisis de resultados son los siguientes:

- **VGG-16 reducido**: Se escoge un conjunto de capas de convolución reducido de una red neuronal VGG-16 [5] con los siguientes tamaños de entrada en cada capa:
  - **Test 1:** N:20 C:64 K:64 H:224 W:224.
  - **Test 2:** N:20 C:128 K:128 H:112 W:112.
  - **Test 3:** N:20 C:256 K:256 H:56 W:56.
  - **Test 4:** N:20 C:512 K:512 H:28 W:28.
- **Una sola imagen**: En estos casos se procesa una sola imagen. Esto es un factor limitante en algunas versiones, por lo que es interesante estudiarlo. Se procesaran los siguientes tamaños de entrada:
  - **Test 5:** N:1 C:224 K:224 H:224 W:224.
  - **Test 6:** N:1 C:5024 K:5024 H:2 W:2.

- **Un canal y un kernel con imágenes grandes**: Estos casos pretenden ver el rendimiento sobre imágenes con un solo canal y kernel. Estos casos se podrían encontrar en procesamiento de imágenes micrométricas, como podría ser la aplicación de un filtro. Los tamaños a procesar son los siguientes:

  - **Test 7:** N:10 C:1 K:1 H:3840 W:2160.
  - **Test 8:** N:2 C:1 K:1 H:7680 W:4320.

Cabe mencionar que los conjuntos de prueba 'Una sola imagen' y 'Un canal y un kernel con imágenes grandes' son casos límites para remarcar las diferencias de las implementaciones y sus paralelizaciones, pero no necesariamente casos de uso común. Como caso común se toma el primer conjunto 'VGG-16 reducido' donde se elige un subconjunto de tamaños de entrada a capas de convolución de esta red neuronal.

4.3. Resultados secuenciales

En esta sección se mostrará una comparación de cada implementación por versión y por estructura de datos, profundizando en cual obtiene mejores resultados y las razones detrás de ello.

4.3.1. Comparativa entre estructura de datos

La idea inicial es comparar las estructuras de datos usadas, ya que, como se ha comentado en el capítulo de implementación, cada una tiene ventajas e inconvenientes. Para la versión 'eficiente en memoria' (ver Figura 4.1) se puede observar que en las primeras seis pruebas la estructura NHWC es la que obtiene mejor rendimiento con diferencia. Esto se debe a que estas seis pruebas tienen más de 4 kernels o canales, lo que permite el uso de SIMD de forma muy eficiente en las transformaciones. En las pruebas siete y ocho se puede observar que la estructura NCHW es más rápida, esto es porque no se usan las SIMD en la versión NHWC, por lo que la NCHW es algo más rápido al usarlas, aunque no mucho más por el retardo que obtiene al tener que hacer más cálculos en transposición de matrices por la estructura de datos.

Para la versión 'producto matricial' se puede observar en la Figura 4.2 que, por lo general, obtiene mejores resultados la estructura NHWC. Esto está causado por la implementación SIMD en esta estructura de datos. Solo en el caso de que la imagen de entrada tenga más de cuatro kernels o canales se usan las instrucciones SIMD, en caso contrario se hace uso de la versión secuencial. Pero en la prueba siete se obtienen también mejores resultados. Esto probablemente sea por la reordenación de datos que hay que realizar para computar el producto matricial, ya que en la estructura NHWC se puede leer y escribir en posiciones contiguas de memoria, mientras que en NCHW se necesita colocar cada dato en su posición correcta para formar la entrada del producto matricial. Juntando este acceso a memoria poco eficiente con el retardo de hacer más operaciones por la trasposición, se obtiene que NCHW sea un caso peor, en general, para esta versión.

De lo que se ha podido apreciar en este apartado, la mayoría de las veces es más eficiente usar NHWC en las versiones implementadas que NCHW, aun así, hay un caso específico cuando se tienen menos de cuatro kernels y canales de entrada en el que se puede obtener una mejora de rendimiento pequeña usando NCHW. Cabe notar que en este apartado no se han incluido comparaciones de

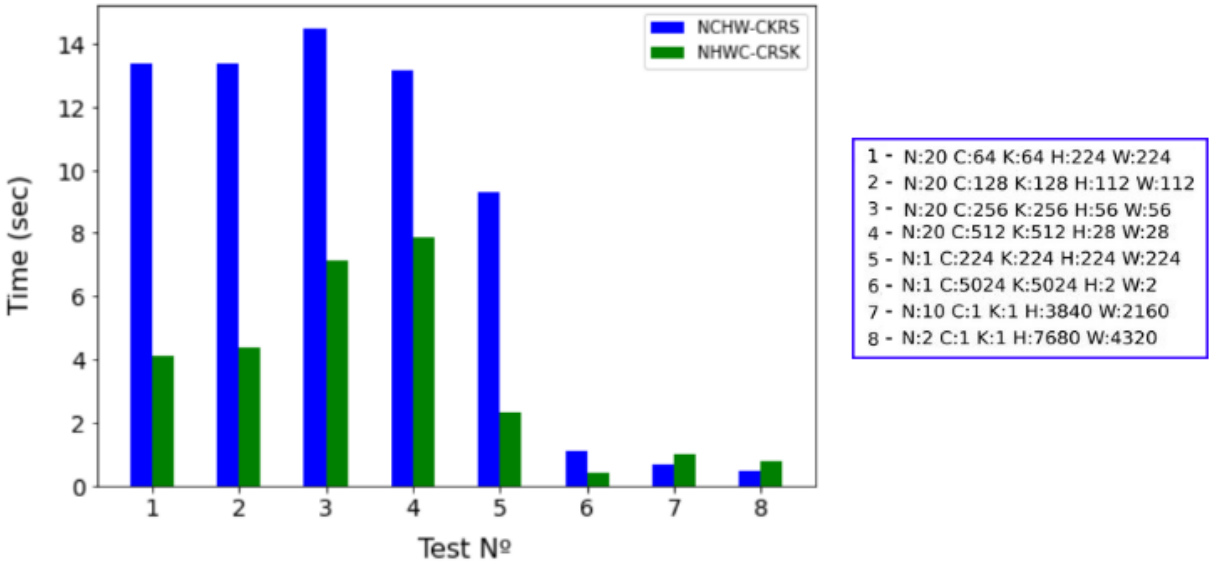

Figura 4.1: Gráfico de barras que muestra el tiempo de ejecución por cada test en función de la estructura de datos elegida usando la versión 'eficiente en memoria'.

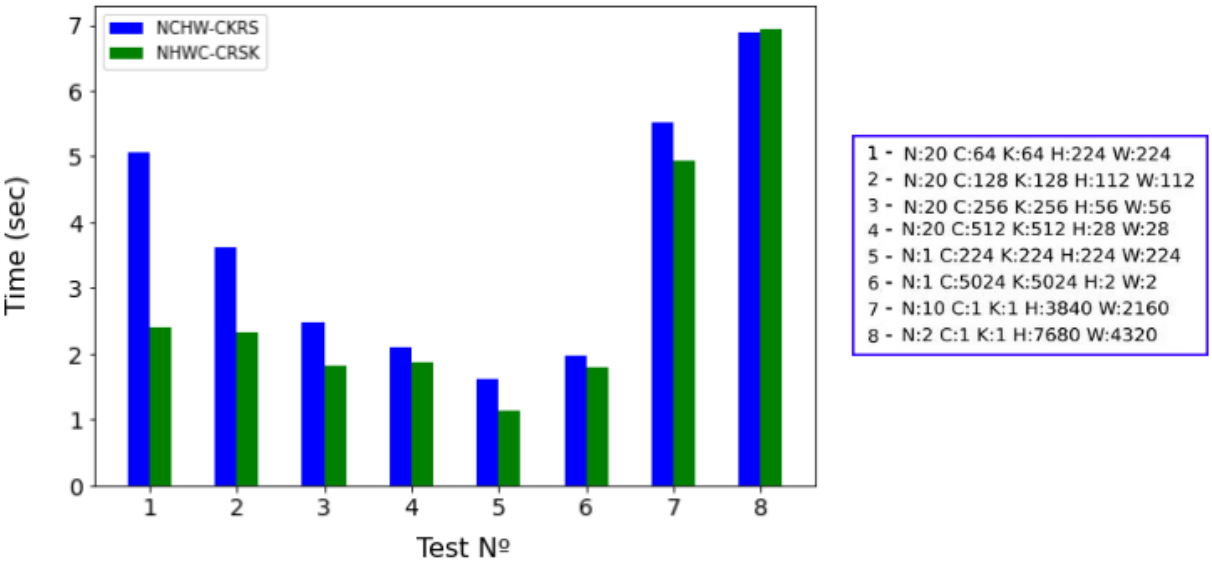

Figura 4.2: Gráfico de barras que muestra el tiempo de ejecución por cada test en función de la estructura de datos elegida usando la versión 'producto matricial'.

la versión producto matricial de imágenes contiguas y esto es porque las secciones críticas (las transformaciones) para esta comparación son iguales que las de la versión producto matricial.

4.3.2. Comparativa entre versiones

En este apartado se verá el rendimiento de cada una de las tres versiones propuestas en el Capítulo 3 de implementación. Como se puede ver en la Figura 4.3, la versión eficiente en memoria es bastante más rápida en las pruebas seis, siete y ocho. Esto es porque estos tests tienen un número pequeño de secciones de imagen (Test 6) o un número muy pequeño de kernels (Test 7 y 8). Esto significa que en las versiones de producto matricial se está realizando un producto matriz por vector fila o columna, lo cual da un rendimiento bastante malo en la biblioteca de cómputo matricial que se está usando (BLIS [8]). También cabe notar los resultados significativamente peores que se obtienen en la versión de producto matricial de imágenes contiguas ya que en todos los casos es igual o peor a la de 'producto matricial' simple. Esto está relacionado con el problema de uso de la cache mencionado en la Sección 3.1.5, que se solucionó superficialmente mediante un desplazamiento de la dirección de memoria para la versión de producto matricial, pero no se decidió incluir en la versión de imágenes contiguas por las repercusiones que podría tener (las repercusiones que podrían tener se mencionan en la Sección 3.1.6).

Por lo demás, se observa una gran ganancia en eficiencia al usar el producto matricial en las capas de redes neuronales, esto es razonable ya que la estructura de estas capas implica unas dimensiones de matrices de transformación lo suficientemente grandes como para obtener un rendimiento óptimo en el producto matricial.

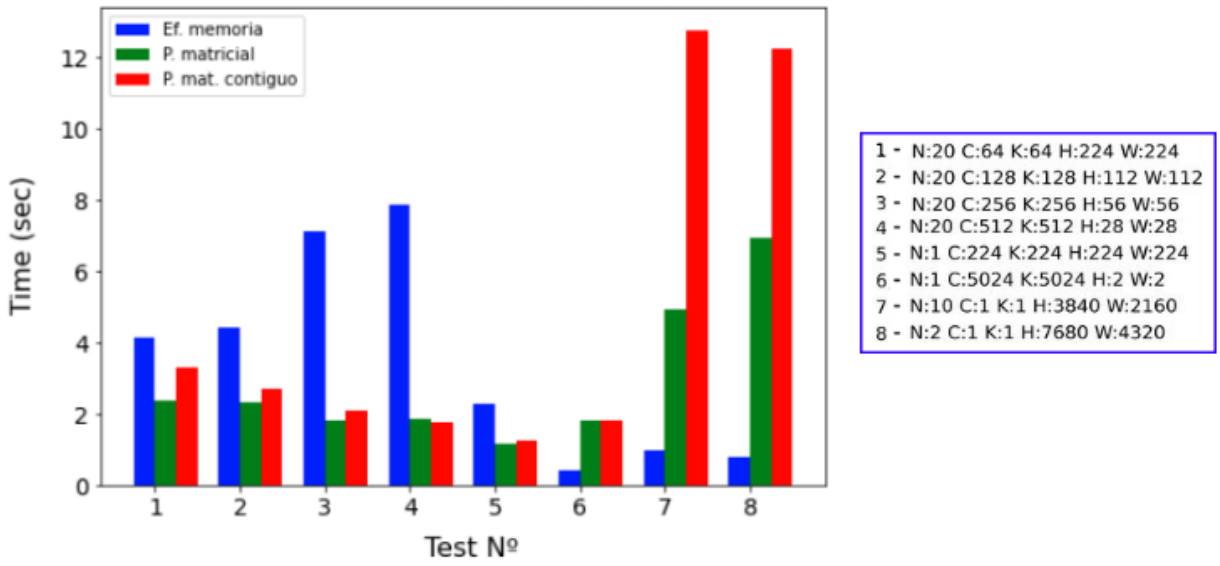

Figura 4.3: Gráfico de barras que muestra el tiempo de ejecución por test de cada versión implementada del algoritmo de Winograd.

4.4. Resultados paralelizados

4.4.1. Versión eficiente en memoria

Los resultados de esta versión, como se puede ver en la Figura 4.4, son favorables en la mayoría de pruebas realizadas, obteniendo aproximadamente un rendimiento n veces mejor usando n hilos. La excepción es el Test 6, ya que en esta prueba empleamos un tamaño de imagen excepcionalmente pequeño (2×2) y la paralelización de esta versión está realizada sobre el bucle que recorre cada imagen (N) y el bucle que recorre las filas de la imagen (H), por lo que, en este caso, no se vería ninguna ventaja en la paralelización, ya que ambos bucles recorren solo un elemento.

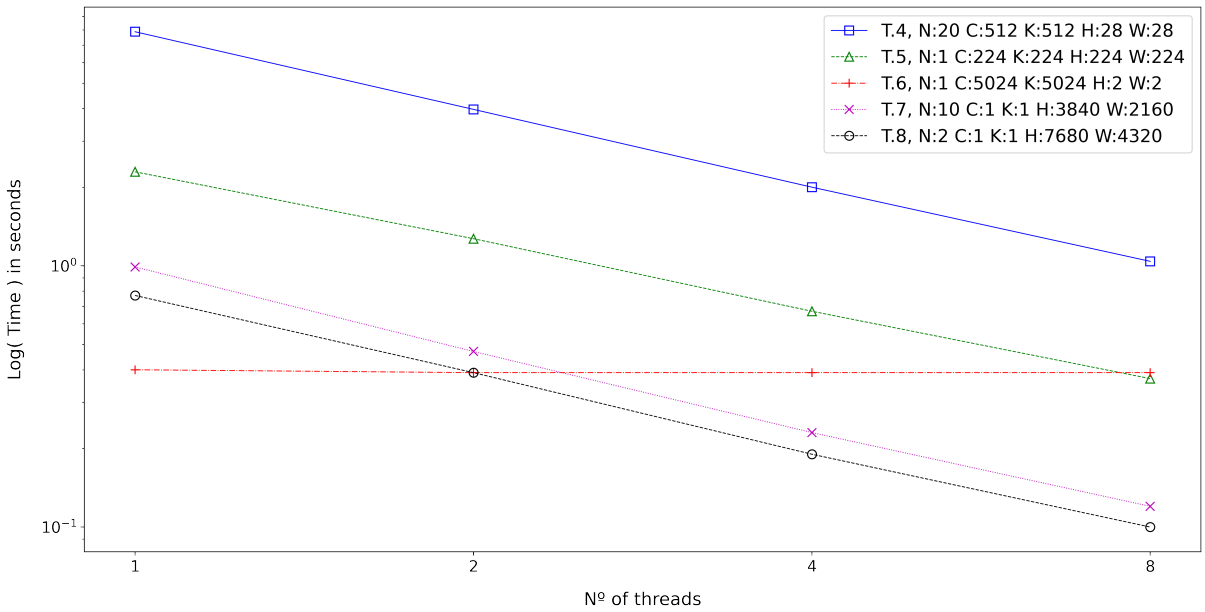

Figura 4.4: Gráfico de lineas que compara el grado de paralelización de la versión eficiente en memoria para la estructura NHWC.

En la estructura NCHW se observa cómo, en general, el comportamiento de paralelizarlo hace que aumente el rendimiento del programa. Cabe notar que la ganancia de eficiencia por hilo no es siempre proporcional ya que, como observamos en la Figura 4.5, el que las líneas del gráfico no sean completamente rectas descendientes probablemente este causado por alguna posible consecuencia de activar el anidamiento en OpenMP, dando alguna variación en la ejecución paralela. Generalmente, activar el anidamiento de hilos en OpenMP causa más problemas que beneficios. Esto es porque lanzar una región paralela de n hilos sobre x hilos ($x \times n$ hilos totales) una mayoría de veces acaba implicando una *sobresuscripción*, es decir, que en un momento dado existen más hilos que núcleos físicos donde puedan ejecutarse los hilos. Esto suele dar lugar a una caída de eficiencia enorme por la cantidad de trabajo añadida que suponen los cambios de contexto en ejecución. En esta implementación nos aseguramos de que solo se use el número de hilos definido por el usuario pero, aun así, puede quedar algún otro efecto derivado de activar la anidación de regiones paralelas en OpenMP que puede explicar la variabilidad en las rectas antes mencionada.

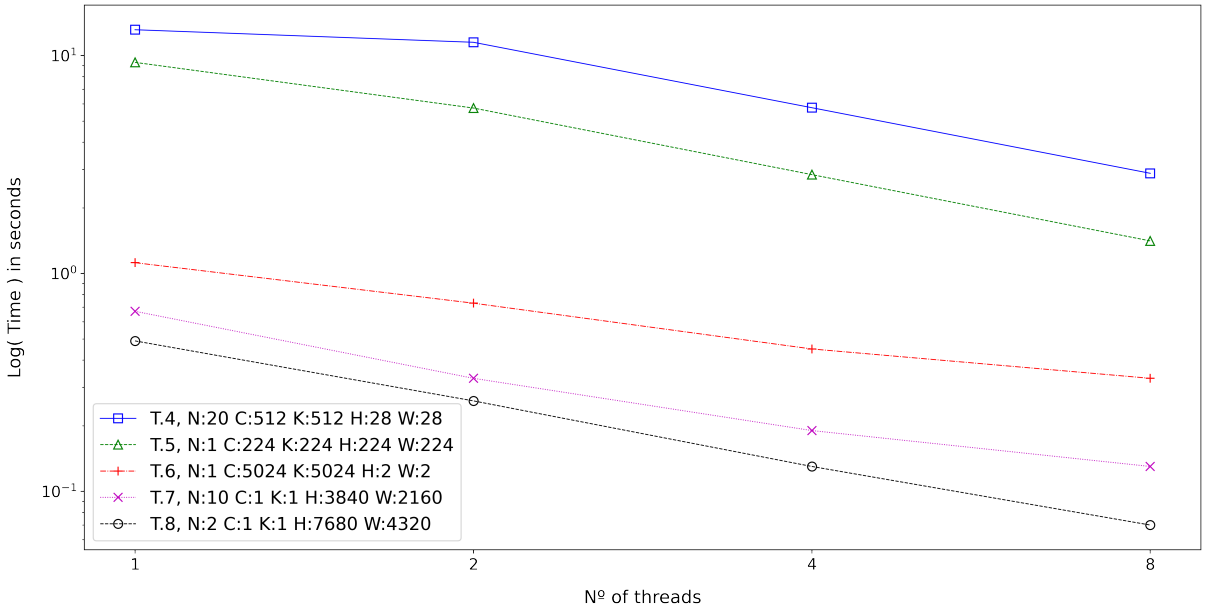

Figura 4.5: Gráfico de líneas que compara el grado de paralelización de la versión eficiente en memoria para la estructura NCHW.

4.4.2. Producto matricial

Para esta versión se observa, en la Figura 4.6, un grado pobre de paralelización, ya que solo las pruebas cuatro y siete ganan rendimiento por cada salto en el número de hilos con los que se ejecuta el programa. El resto de pruebas no consiguen un rendimiento proporcional al número de hilos, esto es porque en el resto de pruebas, el tamaño del batch (número de imágenes) es menor que el número de hilos. Como ejemplo, en la prueba ocho, la entrada solo tiene dos imágenes y se puede apreciar que el rendimiento no mejora a partir de los dos hilos. Cabe notar que no se muestra un gráfico representando el grado de paralelización mediante la estructura NCHW, ya que los bucles que se han paralelizado son los mismos, por lo que la gráfica representaría lo mismo que en la estructura NHWC.

4.4.3. Producto matricial de imágenes contiguas

Una de las ventajas que obtenía esta versión era el mayor grado de paralelización. Esto es lo que puede observarse en la Figura 4.7, donde se observa una ganancia de rendimiento proporcional al número de hilos en todos los casos. Lo único que cabe destacar en estos resultados es cómo la prueba cuatro obtiene mejor ganancia de rendimiento por hilo que la prueba seis. La prueba cuatro distribuye la carga mejor, teniendo un número similar de canales, kernels y tamaño de imagen, mientras que la de la prueba seis esta descompensada teniendo un tamaño de imagen anormalmente pequeño.

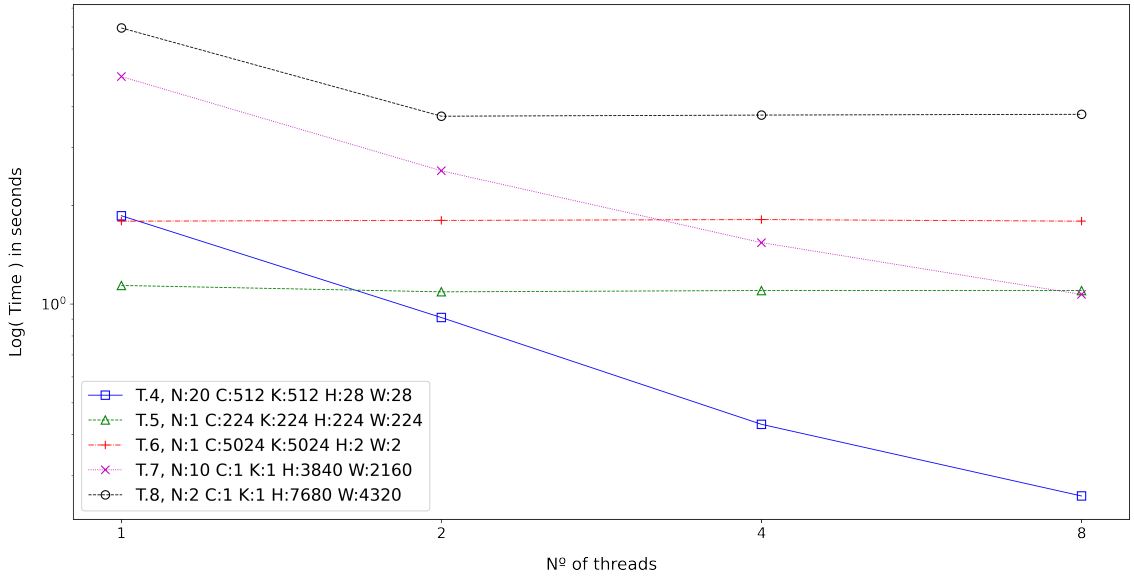

Figura 4.6: Gráfico de líneas que compara el grado de paralelización de la versión de producto matricial para la estructura NHWC.

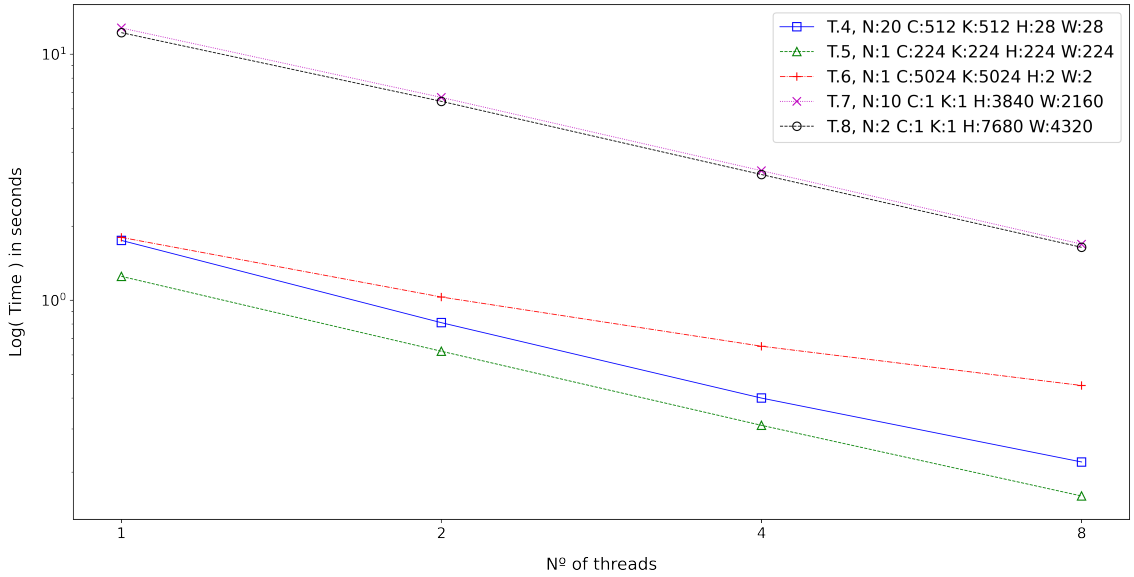

Figura 4.7: Gráfico de líneas que compara el grado de paralelización de la versión de producto matricial de imágenes contiguas para la estructura NHWC.

4.5. Comparativa con im2row

En esta sección se realizará una comparación de las implementaciones propuestas en este documento y se compararán con una convolución calculada mediante la transformación `im2row` (ver Figura 1.3). Este método transforma la convolución directa para que se pueda realizar como producto matricial. Para el producto matricial se empleará su implementación dentro de la biblioteca BLIS [8], ya que las versiones de producto matricial de la implementación propuesta también se basan en esta librería. La variante de la convolución basada en la transformación `im2row` se toma de la biblioteca PyDTNN [7]. Esta implementación calcula la convolución asumiendo que puede existir un borde relleno de ceros, pero sin acceder a las posiciones de memoria donde están los bordes. En la comparación, esto es importante ya que la implementación de este trabajo sí hace calculo sobre el borde (accediendo y operando sobre el borde), por lo tanto, realiza algunos cálculos más que `im2row`. Para la comparación se escogen las versiones NHWC de los algoritmos, ya que, en general, son más rápidas que la variante NCHW. Todas las versiones a comparar se han compilado con la versión 7.5.0 del compilador `gcc` haciendo uso del parámetro `-O3`.

4.5.1. Secuencial

Al comparar la versión secuencial podemos observar que en todos los casos alguna de las implementaciones de este trabajo es más eficiente que utilizando la transformación `im2row`. En las pruebas de capas de red neuronal (pruebas 1 a 4) la versión de producto matricial consigue aproximadamente el doble de ganancia en eficiencia respecto a la implementación basada en la transformación `im2row`. En las pruebas de imágenes de gran resolución con un solo canal y kernel, la versión eficiente en memoria consigue ser de seis a ocho veces más rápida. En esta ocasión esta gran ganancia de rendimiento, por un lado, es debida a las ventajas de usar el algoritmo de Winograd, pero también, porque el solo tener un canal y kernel en la operación implica que las dimensiones de entrada del producto matricial no son capaces de aprovechar todo el rendimiento de la biblioteca usada. Finalmente, cabe destacar la prueba seis donde, por el pequeño tamaño de imagen, no se aprovechan bien los beneficios de usar el algoritmo de Winograd y los resultados de las implementaciones de este trabajo se acercan a los resultados obtenidos por la transformación `im2row`.

4.5.2. Paralelo

En la versión paralela, en la mayoría de casos, alguna de las implementaciones propuestas es más rápida que `im2row` (véase la Figura 4.9). La versión de producto matricial mediante imágenes contiguas es más rápida que la transformación `im2row` para los casos de prueba de la red neuronal (pruebas 1 a 4). De forma similar al secuencial, la versión eficiente en memoria es varias veces más rápida que `im2row` para las pruebas de un solo canal y kernel (pruebas 7 y 8). El caso que cabe destacar es aquel donde en secuencial la implementación propuesta e `im2row` obtenían rendimientos similares. La prueba 6 en la versión paralela de `im2row` consigue una ventaja grande respecto a la implementación propuesta, pero esto solo ocurre en este caso (un caso limite donde el tamaño de la imagen es excesivamente pequeño). Finalmente, cabe mencionar, que para esta comparativa, se ha usado la versión de producto matricial de imágenes contiguas a pesar de ser algo menos eficiente en secuencial (por las colisiones de cache) por su mayor grado de paralelización.

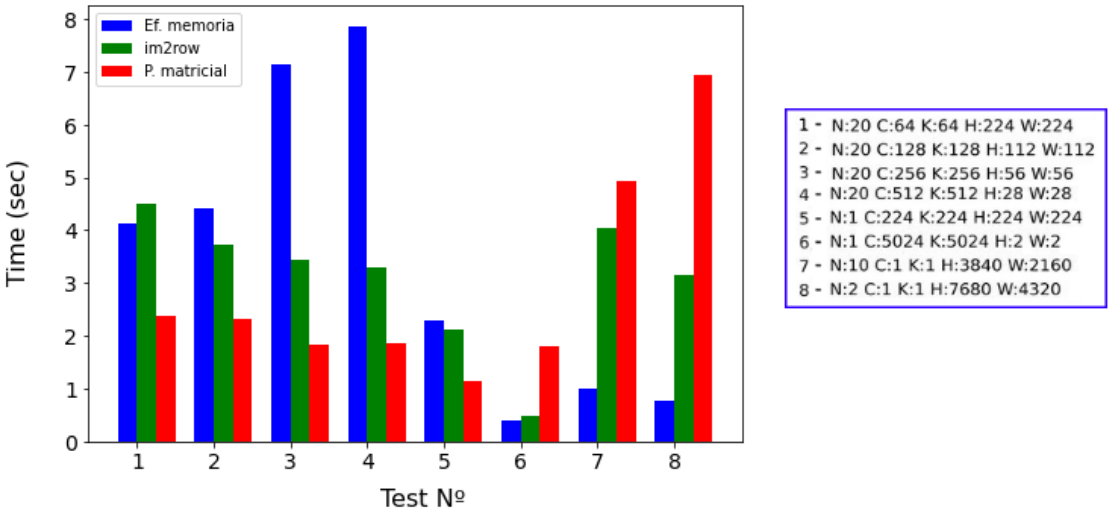

Figura 4.8: Gráfico de barras que compara las implementaciones 'eficiente en memoria' y 'producto matricial' (NHWC) con el método `im2row` secuencial.

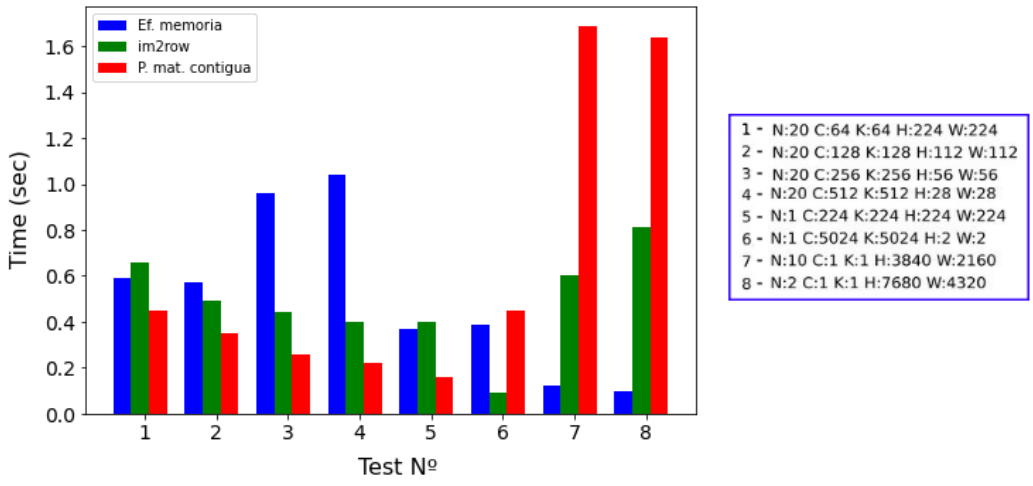

Figura 4.9: Gráfico de barras que compara las implementaciones 'eficiente en memoria' y 'producto matricial' contiguo (NHWC) con el método `im2row` ejecutado de forma paralela con ocho hilos.

Capítulo 5

Conclusiones

Como conclusiones del trabajo expuesto, tomamos que la implementación específica del algoritmo de Winograd es una alternativa muy interesante en relación a las prestaciones y comparado a otros algoritmos para el cálculo de la operación de convolución.

5.1. Objetivos cumplidos

En el presente trabajo se han logrado los siguientes objetivos:

- **Implementación**: Se ha realizado una implementación específica del algoritmo de Winograd para convoluciones $F(2 \times 2, 3 \times 3)$. Además, se han incluido algunas técnicas o decisiones de implementación, buscando obtener el máximo rendimiento.
- **Paralelización**: Se ha expuesto una paralelización del algoritmo mediante instrucciones SIMD de la extensión NEON para procesadores ARM y a nivel de hilo mediante OpenMP.
- **Resultados**: Se ha obtenido y explorado el rendimiento del programa en todas sus versiones, además de realizar una comparativa con un método actual para el calculo eficiente de convoluciones.

5.2. Trabajo futuro

En este trabajo se han podido exponer varias ideas, implementaciones y optimizaciones sobre el cálculo de convoluciones, pero todavía existen posibles mejoras y análisis que podrían aumentar la calidad y rendimiento de las implementaciones propuestas. Algunas de las implementaciones propuestas se describen a continuación:

- **Implementación mediante matrices dispersas**: Esta idea consiste en realizar un cálculo del algoritmo de Winograd para una bloque de imagen y un filtro de tal forma que se puedan precalcular varias matrices de transformación y aplicar el algoritmo para calcular la convolución de cualquier tamaño de sección de imagen o filtro dentro de las matrices precalculadas. Cabe estudiar si una implementación genérica mediante matrices dispersas conservaría suficiente ganancia de eficiencia sobre otros métodos como para ser lo suficientemente conveniente.

- **Salto de relleno**: Aunque la convolución implementada toma una imagen $H \times W$ y da como salida un conjunto $(H-2) \times (W-2)$ para cualquier conjunto entrada, cabe destacar que en redes neuronales los bordes de la imagen suelen ser un relleno de ceros. Se podría implementar una optimización para estas versiones que detecte cuando está en el borde y no opere sobre estos elementos ahorrando algunas operaciones en la ejecución.

- **Implementar otros tamaños de convolución**: En este trabajo se ha implementado la convolución para secciones $F(2 \times 2, 3 \times 3)$. Se deja como trabajo futuro implementar otros formatos, algunos de estos podrían ser $F(3 \times 3, 2 \times 2)$ o $F(4 \times 4, 3 \times 3)$ como se muestra en [1].

- **Estudiar la eficiencia en cache**: Como se mencionó en la Sección 3.1.5, no se ha estudiado en profundidad este tipo de colisión, así como la mejor forma de solucionarlo. Por ello se propone investigar más acerca de este fenómeno, así como de forma más general optimizar el uso de cache del algoritmo en su totalidad.

- **Desplazamiento de producto matricial**: En la Sección 2.2 se vio una estructura de datos que permitiría hacer el producto matricial para el cálculo pertinente. Se propone experimentar con el desplazamiento entre datos, así como el orden de los datos en sí, manteniendo los mismos cálculos para explorar una posible mejor solución. Un ejemplo consistiría en incorporar un desplazamiento de cada elemento de salida de (16) para poder almacenar y acceder a cada elemento de la matriz de convolución transformada en la transformación final de forma contigua.

Con los resultados obtenidos de la implementación mostrada en el trabajo y el posible trabajo futuro sobre esta, es posible que se pudiera obtener una versión del algoritmo genérica que se pudiera aplicar en los tamaños más comunes de convolución ganando gran eficiencia respecto a otras implementaciones comunes para el cálculo rápido de convoluciones.

Bibliografía

[1] Andrew Lavin and Scott Gray, Fast Algorithms for Convolutional Neural Networks, nov 2015

[2] Partha M. Andrew M. Ganesh D. Jesse B. Matthew M. and Robert M., Efficient Winograd or Cook-Toom Convolution Kernel Implementation on Widely Used Mobile CPUs, mar 2019

[3] Shmuel Winograd, Arithmetic complexity of computations. volume 33, 1980

[4] Wentai Zhang and Guojie Luo, Evaluating Low-Memory GEMMs for Convolutional Neural Network Inference on FPGAs, may 2020

[5] Karen S. and Andrew Z., Very Deep Convolutional Networks For Large-Scale Image Recognition, may 2015

[6] Thomas Alsop, Arm's market share across different technology markets worldwide 2019 & 2028, nov 2020

[7] Sergio B. Adrián C. Mar C. Manuel F.D. and Jose I.M., PyDTNN: A user-friendly and extensible framework for distributed deep learning, feb 2021

[8] Field G. Van Zee and Robert A. van de Geijn, BLIS: A Framework for Rapidly Instantiating BLAS Functionality, jun 2015

[9] OpenMP 4.5 Complete Specifications, OpenMP.org, openmp.org/wp-content/uploads/openmp-4.5.pdf, nov 2015

[10] ARM Intrinsics, ARM, developer.arm.com/architectures/instruction-sets/intrinsics/, Ultima visita: 2 nov 2021

[11] BLAS (Basic Linear Algebra Subprograms), netlib.org, netlib.org/blas/, Ultima visita: 22 nov 2021

[12] SIMD transposes 1, The ryg blog, fgiesen.wordpress.com/2013/07/09/simd-transposes-1/, Ultima visita: 9 nov 2021

[13] Cortex-A57 Software Optimization Guide, ARM, developer.arm.com/documentation/uan0015/b/, Ultima visita: 5 nov 2021

[14] Kernel (Image Processing), Wikipedia, wikipedia.org/wiki/Kernel_(image_processing), Ultima visita: 2 nov 2021

[15] VGG16 Summary, kaggle, kaggle.com/getting-started/178568, Ultima visita: 28 oct 2021

[16] Pooling, Dive Into Deep Learning Compiler, tvm.d2l.ai/chapter_common_operators/pooling Ultima visita: 29 oct 2021

[17] Swarnima P, How to choose the size of the convolution filter or Kernel size for CNN? [medium.com], jun 2020