UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Optimización del producto matricial sobre dispositivos de bajo consumo para inferencia en Deep Learning

TRABAJO FIN DE MÁSTER

Máster Universitario en Computación en la Nube y de Altas Prestaciones

Autor: Eugenio Bernabé Stabile

Tutores: Pedro Alonso Jordá,
Enrique Salvador Quintana-Ortí

Curso 2020-2021

Resum

L'aprenentatge automàtic mitjançant xarxes neuronals profundes ha experimentat un gran auge en l'última dècada, principalment per la combinació de diversos factors, entre els quals s'inclouen l'allau de dades per a entrenar aquest tipus de sistemes (big data), una major capacitat dels sistemes de computació (processadors gràfics de NVIDIA, TPUs de Google, etc.), els avanços en tècniques algorítmiques d'aprenentatge (per exemple, xarxes de tipus *transformer* per a processament del llenguatge), i la disponibilitat d'entorns amigables per a la tasca.

En l'actualitat existeixen diferents paquets de programari per a l'entrenament de xarxes neuronals profundes sobre clústers de computadors (TensorFlow de Google i PyTorch de Facebook), i fins i tot els mateixos paquets tenen versions especialitzades (TensorFlow Lite, NVIDIA RT, QNNPACK, etc.) per a realitzar el procés d'inferència sobre processadors de baix consum, com els que poden trobar-se en un mòbil Android o iOS o en un vehicle sense conductor.

Molts dels sistemes tracten xarxes neuronals convolucionals, especialment aquells que tracten amb imatges. A un nivell més baix de detall podem observar que l'entrenament i la inferència en les capes convolucionals de les xarxes neuronals esmentades apareix un producte matricial amb característiques particulars, ben definides i que requereixen d'un tractament especial quan es tracta de la seua optimització. Aquest treball de fi de màster tracta de l'optimització d'aquesta operació, en particular, sobre una arquitectura ARM, el processador multinucli que pot trobar-se en gran part dels dispositius de baix consum on es pretén executar la inferència d'una xarxa prèviament entrenada. L'optimització plantejada està inspirat en un paquet de rutines optimitzades d'àlgebra lineal numèrica denominat BLIS, d'on s'obtenen els algorismes bàsics sobre els quals es fa el treball. El projecte permetrà a l'estudiant adquirir un bon coneixement dels aspectes computacionals relacionats amb el procés inferència amb xarxes neuronals profundes, així com aprofundir en la interacció entre l'algorisme i l'arquitectura del processador i com aquesta determina el rendiment.

Paraules clau: multiplicació de matrius, BLIS, processadors AKM, Aprenentatge automàtic, Convolució

Resumen

El aprendizaje automático mediante redes neuronales profundas ha experimentado un gran auge en la última década, principalmente por la combinación de varios factores, entre los que se incluyen la avalancha de datos para entrenar este tipo de sistemas (big data), una mayor capacidad de los sistemas de computación (procesadores gráficos de NVIDIA, TPUs de Google, etc.), los avances en técnicas algorítmicas de aprendizaje (por ejemplo, redes de tipo *transformer* para procesamiento del lenguaje), y la disponibilidad de entornos amigables para la tarea.

En la actualidad existen diferentes paquetes de software para el entrenamiento de redes neuronales profundas sobre clusters de computadores (TensorFlow de Google y PyTorch de Facebook), e incluso los mismos paquetes tienen versiones especializadas (TensorFlow Lite, NVIDIA RT, QNNPACK, etc.) para realizar el proceso de inferencia sobre procesadores de bajo consumo, como los que pueden encontrarse en un móvil Android o iOS o en un vehículo sin conductor.

Muchos de los sistemas tratan redes neuronales convolucionales, especialmente aquellos que tratan con imágenes. A un nivel más bajo de detalle podemos observar que el entrenamiento y la inferencia en las capas convolucionales de las redes neuronales mencionadas aparece un producto matricial con características particulares, bien definidas y que requieren de un tratamiento especial cuando se trata de su optimización. Este trabajo de fin de máster trata de la optimización de esta operación, en particular, sobre una arquitectura ARM, cuyo procesador multinúcleo puede encontrarse en gran parte de los dispositivos de bajo consumo donde se pretende ejecutar la inferencia de una red previamente entrenada. La optimización planteada está inspirada en un paquete de rutinas optimizadas de álgebra lineal numérica denominado BLIS, de donde se obtienen los algoritmos básicos sobre los que se realiza el trabajo. El proyecto permitirá al estudiante adquirir un buen conocimiento de los aspectos computacionales relacionados con el proceso inferencia con redes neuronales profundas, así como profundizar en la interacción entre el algoritmo y la arquitectura del procesador y cómo esta determina el rendimiento.

Palabras clave: multiplicación de matrices, BLIS, procesadores AKM, Aprendizaje automático, Convolución

Abstract

The use of machine learning in deep neural networks has experienced a boom in the last decade, mainly due to a combination of several factors, including the abundance of data to train such systems (big data), increased computing power (NVIDIA graphics processors, Google TPUs, etc.), advances in algorithmic learning techniques (transformer neural networks for language processing) and the availability of user-friendly environments for the task.

There are currently different software packages for training deep neural networks on computer clusters (TensorFlow and PyTorch) and even the same packages have specialized versions (TensorFlow Lite, NVIDIA RT, QNNPACK, etc.) to perform the inference process on low-power processors, such as those that can be found in an Android or iOS mobile phone or in a driverless car.

Many of the systems deal with convolutional neural networks, especially those that deal with images. At a lower level of detail, we can observe that the training and inference in the convolutional layers of the aforementioned neural networks result in a matrix product with particular, well-defined characteristics that require special treatment when it comes to optimization. This master's thesis deals with the optimization of this operation, in particular, on an ARM architecture, whose multicore processor can be found in most of the low-power devices where it is intended to execute the inference of a previously trained network. The proposed optimization is inspired by a package of optimized numerical linear algebra routines called BLIS, from which the basic algorithms on which the work is carried out are obtained. The project will allow the student to acquire a good knowledge of the computational aspects related to the inference process with deep neural networks, as well as to deepen the interaction between the algorithm and the architecture of the processor and how this determines the performance.

Key words: GEMM, BLIS, AKM processors, deep learning, convolutional neural network

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

1.1 Multiplicación de matrices

El número de las aplicaciones que utilizan las redes neuronales ha ido incrementándose año tras año. Esto se debe en parte a los avances en este ámbito de nuevas arquitecturas de redes neuronales, como por ejemplo Alexnet (ver Figura 1.1), demostrando una mejora significativa en la tasa de acierto sobre la detección de objetos en imágenes.

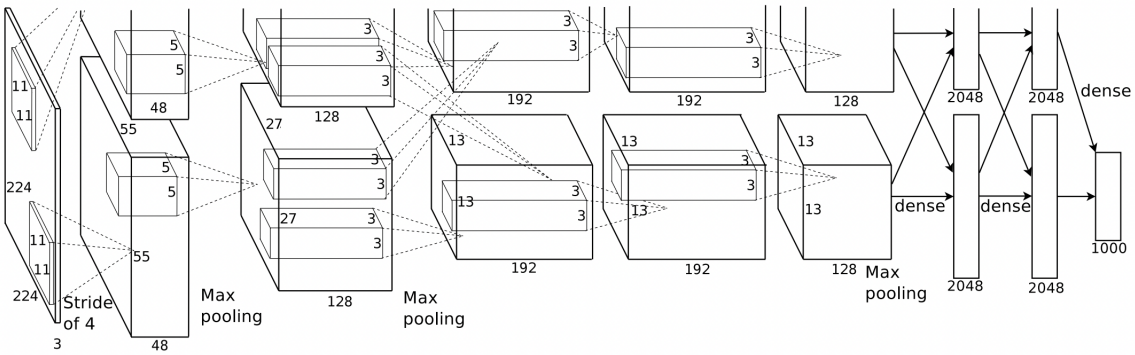

Figura 1.1: Ilustración de la arquitectura de la red convolucional Alexnet.

Una de las funciones con mayor importancia que utilizan las redes neuronales es la convolución, que puede implementarse como un producto de matrices. Esta operación, debido al elevado número de operaciones que supone, consume una gran cantidad de recursos computacionales así como también de tiempo y de ello se desprende la importancia de su optimización para reducir estos costes.

Actualmente, se siguen investigando nuevos y mejores métodos para la realización de estas operaciones de convolución de manera eficiente. Además, con la aparición de nuevas arquitecturas, se necesita rediseñar los algoritmos para obtener las máximas prestaciones adaptándolos a estos dispositivos. Es por ello que en lo relativo al producto de matrices, a lo largo de los años, se han propuesto técnicas como la multiplicación a bloques, el procesamiento multihilo, el almacenamiento eficiente de los elementos de las matrices en los recursos de memoria y la utilización de librerías de álgebra lineal básica (BLAS), entre otros.

1.2 Motivación

La evolución del aprendizaje automático mediante redes neuronales profundas, así como el incremento en la potencia de cálculo de los dispositivos durante los últimos años, ha traído consigo nuevos paradigmas de computación como el *Edge AI* [1]. Este concepto se define como la utilización de la inteligencia artificial en los computadores al borde de la red (*edge computing*) y surge debido a la necesidad de realizar en tiempo real el proceso de inferencia con redes neuronales. Una de las aplicaciones de uso de esta tecnología en la actualidad es la conducción autónoma, en donde ejecutar esta operación de inferencia en un servidor en la nube implica latencias de red que no pueden ser asumidas, o incluso situaciones en donde no se dispone de una conexión a la red para el envío y recepción de datos. Si bien el costo computacional del entrenamiento de una red neuronal es elevado como para poder ejecutarlo en los dispositivos de *edge computing*, este no suele ser el caso para la inferencia. Es por ello que, en este paradigma, se despliega la red ya entrenada en los ordenadores frontera para realizar esta operación en tiempo real y con baja latencia.

La aparición del *Edge AI* permite nuevos servicios a la sociedad que hasta hace unos años no eran posibles, como por ejemplo, la realización de inspecciones de los cascos de embarcaciones mediante *drones* autónomos (Figura 1.2), obteniendo resultados del estado del casco con un mayor grado de fiabilidad.

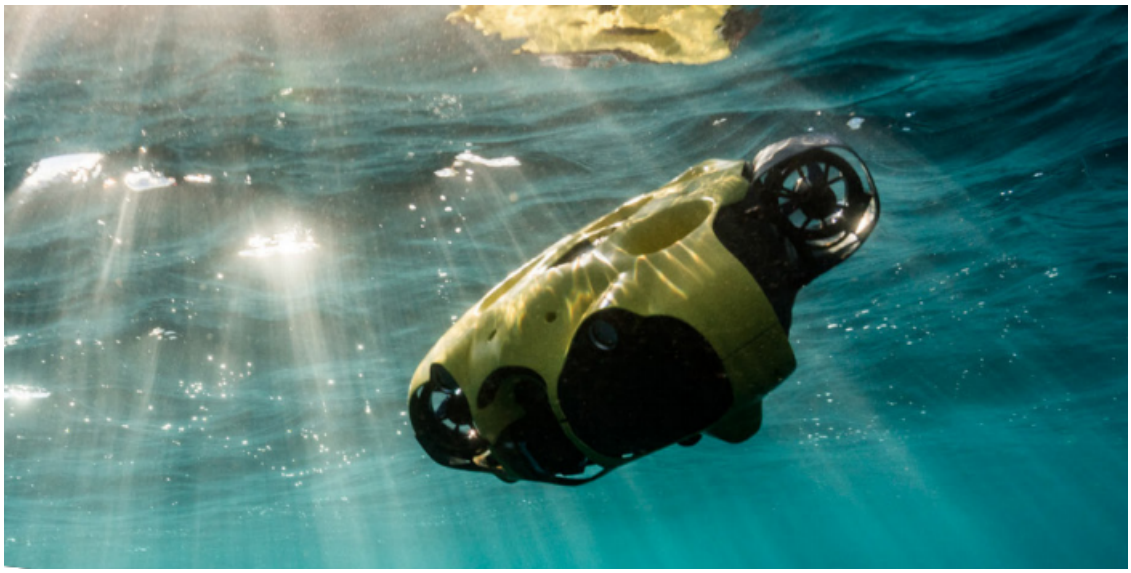

Figura 1.2: *Drone* submarino autónomo, creado por la empresa Notilo Plus, que integra una placa NVIDIA Jetson TX2, con un procesador de arquitectura ARM.

Para la detección de fisuras o anomalías del casco en una embarcación, estos *drones* realizan la inferencia de una red neuronal utilizando como datos de entrada la combinación de grandes cantidades de imágenes obtenidas mediante visión por computador y de señales acústicas.

Como podemos ver, la demanda de cómputo en estos dispositivos va en incremento año tras año. Si tenemos en cuenta que gran parte de los computadores de frontera están equipados con procesadores de bajo consumo y de baterías que les proveen el suministro eléctrico, la optimización de estos algoritmos de inferencia nos permite un mejor empleo de los recursos del procesador, aumentando su rendimiento, así como también incrementar la autonomía del producto. Es por ello que el empleo de técnicas de optimización en procesadores de bajo consumo, como las detalladas en el presente proyecto, son relevantes para el continuo desarrollo de la industria, consiguiendo no solo mejoras que

permitan una resolución de los problemas de forma más rápida sino también soluciones más eficientes, consumiendo menos energía eléctrica y, con ello, siendo responsables con el medio ambiente.

1.3 Objetivos

Para el siguiente trabajo tenemos como objetivo la optimización del producto matricial, utilizado para el entrenamiento y la inferencia en las capas convolucionales de las redes neuronales, sobre procesadores multinúcleo con arquitectura ARM. Cabe destacar que el enfoque de este proyecto está inspirado en un paquete de rutinas optimizadas de álgebra lineal numérica denominado BLIS, de donde se obtienen los algoritmos básicos sobre los que se realiza el trabajo.

Al momento de realizar el entrenamiento y la inferencia de una red neuronal convolucional, debido a las dimensiones de las matrices de entrada, BLIS presenta una pérdida de rendimiento. Por este motivo el objetivo del proyecto reside en la aplicación de diferentes técnicas de optimización sobre una de las operaciones que mayor coste presenta en el proceso de inferencia de una red neuronal, la multiplicación de matrices. En resumen, las estrategias utilizadas en el presente documento se pueden concretar en los siguientes objetivos:

- Utilización de estrategias de optimización en el almacenamiento de las matrices en memoria.
- Búsqueda y empleo de los valores óptimos correspondientes a los bloques de las matrices para los tamaños de los registros y de la memoria caché del dispositivo.
- Desarrollo del producto matricial utilizando extensiones avanzadas de la arquitectura vectorial de ARM.
- Uso de las técnicas de gestión de memoria *unroll* y *prefetch*.
- Paralelización del código mediante OpenMP en memoria compartida.

1.4 Estructura de la memoria

El presente trabajo final de máster está compuesto por cinco capítulos:

- Introducción: Exponemos en este capítulo la actualidad de las redes neuronales convolucionales y la importancia del producto de matrices en estos modelos. Además, se explica la motivación del presente trabajo, así como los objetivos a conseguir.
- Herramientas: Se detallan las características del dispositivo y las tecnologías utilizadas en este trabajo. Con respecto al *hardware*, se realiza una descripción del dispositivo empleado en el presente proyecto, NVIDIA Jetson Nano. Se destacan los componentes más relevantes para la realización del trabajo como son la memoria caché y el procesador basado en arquitectura ARM. En cuanto a las tecnologías, se detallan las utilizadas para el desarrollo a bajo nivel del *microkernel*, así como la paralelización del mismo.

- Aplicación: Se explica en este capítulo el código base al inicio del proyecto y su problema de rendimiento cuando se utiliza para el entrenamiento o la inferencia de las redes neuronales. Adicionalmente, se proponen y desarrollan varias optimizaciones para corregir el problema, las cuales posteriormente son evaluadas para comparar el rendimiento de estos cambios con respecto al programa original.
- Conclusiones: Explicamos y razonamos los resultados obtenidos en el presente proyecto. Adicionalmente, se relacionan los conceptos aprendidos en las asignaturas cursadas del máster con respecto al trabajo desarrollado.

CAPÍTULO 2
Herramientas

Para la realización del presente trabajo se requiere un completo conocimiento de las características del dispositivo a emplear, sobre todo de la memoria caché. Esta necesidad se debe a que las técnicas a emplear hacen especial hincapié en estas estructuras básicas del computador para conseguir un mejor rendimiento del algoritmo. Adicionalmente, se utilizan herramientas que permiten llevar a cabo la optimización del dispositivo a bajo nivel, así como la paralelización a nivel de hilo del programa. A continuación, en este capítulo se detallan las técnicas y herramientas utilizadas.

2.1 BLIS

BLIS (BLAS-*like Library Instantiation Software*) [2] es un *framework* portable que sirve para implementar instancias de alto rendimiento de las funciones de álgebra lineal densa de BLAS (*Basic Linear Algebra Subprograms*). Asimismo, ofrece una interfaz de programación de aplicaciones (conocida por sus siglas en inglés como API), definiendo el conjunto de subrutinas, funciones y procedimientos disponibles para el desarrollador, similar a BLAS (llamada BLAS-like API). Además, incluye una capa de compatibilidad con BLAS, así como una API basada en objetos exclusiva de BLIS. Cabe destacar su portabilidad, ya que puede ser utilizado tanto en arquitecturas x86 como ARM. Otras alternativas a este *framework* son:

- **MKL** (*Math Kernel Library*): Creada por Intel, es una librería que incluye funciones optimizadas de BLAS, LAPACK, FFT, entre otras. Su principal diferencia respecto a BLIS es la naturaleza del código, ya que es propiedad intelectual del fabricante de circuitos integrados mencionado anteriormente. Además, en genereal ofrece un mejor rendimiento que BLIS.

- **OpenBLAS**: Es una versión basada en GotoBLAS2 (actualmente descontinuada) y escrita en el lenguaje C. Esta librería de código abierto incluye llamadas a funciones de BLAS y LAPACK. OpenBLAS se puede utilizar tanto en procesadores de AMD, como de Intel y de ARM. Además posee soporte para procesamiento multihilo y el conjunto de instrucciones AVX (*Advanced Vector Extensions*). Este último es una extensión vectorial SIMD (*Single Instruction, Multiple Data*) creada por Intel para la realización de operaciones de coma flotante utilizando un ancho de registro vectorial de 256 bits.

- **ATLAS** (*Automatically Tuned Linear Algebra Software*): Proporciona un paquete de álgebra lineal portable, que se compone de una API de BLAS y un subconjunto de la librería LAPACK para los lenguajes C y Fortran77.

2.2 ARM

En la actualidad, los procesadores utilizan mayoritariamente dos modelos de arquitecturas. El primer modelo diseñado fue CISC (*Complex Instruction Set Computer*), en donde una sola instrucción puede ejecutar varias operaciones de bajo nivel. Con respecto al tamaño de la instrucción, este varía dependiendo de la operación a realizar, ya que permite la manipulación directa de los operandos ubicados en la memoria. Como podemos ver, este conjunto de instrucciones se caracteriza por su amplitud y complejidad, permitiendo que el número de instrucciones necesarias para ejecutar un programa se reduzca significativamente, lo cual se traduce en un menor consumo de memoria RAM y un menor trabajo de traducción para una sentencia de lenguaje de alto nivel a ensamblador por parte del compilador. Como punto negativo, podemos destacar la dificultad para realizar una paralelización entre instrucciones debido a su heterogeneidad. En los procesadores CISC actuales se implementa un sistema que decodifica cada una de las instrucciones complejas en varias instrucciones simples para lograr este fin.

Años más tarde de la creación del modelo CISC, se realizaron investigaciones con respecto a la frecuencia de utilización de una instrucción en este modelo, dando como resultado que el 20 % de las instrucciones ocupa el 80 % del tiempo total de ejecución de un programa. Esta conclusión provocó un cambio de paradigma, el cual terminó con la creación del modelo RISC (*Reduced Instruction Set Computer*) que, a diferencia de CISC, utiliza un conjunto de instrucciones simples y de tamaño fijo, en donde cada instrucción tarda en ejecutarse unos pocos ciclos de reloj. Esto produce que a nivel físico los procesadores sean más simples y de menor tamaño, lo cual se traduce en un menor número de transistores a emplear. Al poseer estas características físicas, RISC permite a los fabricantes de procesadores añadir una mayor cantidad de núcleos, así como también de unidades de procesamiento específicas, mejorando y adaptando el diseño a las necesidades de los diferentes mercados. Por esta razón, hoy en día, la utilización del modelo RISC es mayor con respecto al CISC, como podemos ver en la Figura 2.1, en donde se expresa la cantidad de chips vendidos en el período de 2007 a 2016.

En la Figura 2.1, podemos ver como la fabricación de chips con arquitectura ARM, basada en el modelo RISC, supera por un amplio margen a las ventas producidas por la familia de procesadores 80x86, basadas en CISC.[1] La diferencia, que se expande cada vez más, es producida por la irrupción de la telefonía móvil y los dispositivos embebidos, mercado que domina en casi su totalidad los chips con modelo RISC.

[1] En la actualidad, si bien la arquitectura x86 se basa en el modelo CISC, internamente traduce las instrucciones a RISC.

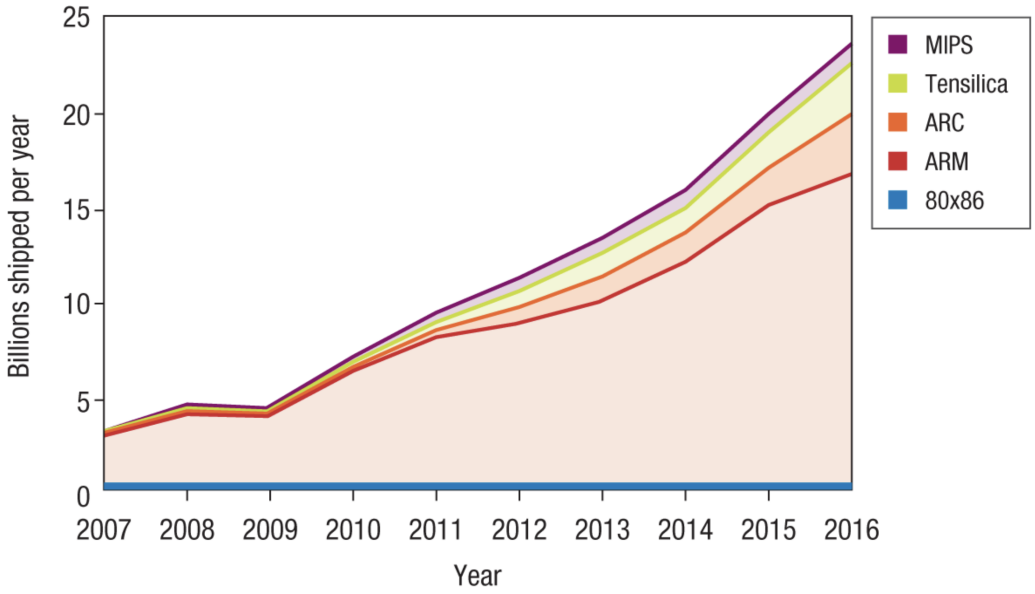

Figura 2.1: Número de ventas de chips producidas en el período de 2007 a 2016. [3]

Como hemos visto anteriormente, ARM [4] es una arquitectura basada en RISC, que es implementada en una amplia gama de procesadores para diversos fines como sistemas embebidos, sistemas en tiempo real y plataformas con una mayor potencia de cálculo como los servidores, entre otros. Además de fabricar procesadores, ARM licencia su arquitectura a terceros para la creación de microcontroladores y CPUs. En estas licencias, la empresa define la arquitectura del procesador, indicando el conjunto de instrucciones básicas, así como modelos de excepción y memoria en los que se basan el sistema operativo y el hipervisor. Adicionalmente, como se menciona al inicio de la sección, ARM especifica un conjunto de instrucciones reducido (RISC). Esto indica que las instrucciones son simples, permitiendo de esta manera aportar características a esta arquitectura como:

- Tamaño de registros uniforme y tamaño de instrucciones fijo con un reducido número de formatos.
- Procesamiento de datos realizado sobre el contenido de los registros en lugar de hacerlo directamente sobre el contenido de la memoria.
- Direccionamiento simple, de modo que todas las direcciones de carga y almacenamiento se determinan únicamente a partir de los campos de la instrucción y del contenido de los registros.

Con respecto al gasto energético, los modelos basados en RISC son más eficientes que su contraparte CISC (*Complex Instruction Set Computer*). Esto se debe a la simpleza de las instrucciones RISC comparada con la complejidad del conjunto CISC, que resulta en una menor cantidad de transistores dedicados a ejecutar la lógica principal.

2.2.1. Vectorización y paralelismo

El empleo de estructuras vectoriales en el área de la computación científica permite realizar en simultáneo cálculos sobre múltiples datos con una sola instrucción, logrando con ello un mayor rendimiento con respecto a la ejecución escalar. A lo largo de los años, la forma de implementar dichas estructuras se ha ido modificando. En las décadas de los 80 y 90 se empleaban procesadores vectoriales que constaban de una unidad escalar

segmentada y una unidad vectorial. Esta última unidad contiene M registros vectoriales de N elementos cada uno y unidades funcionales vectoriales, como la suma o la multiplicación, que se aplican sobre estos elementos al momento de realizar una operación. De esta manera, una operación vectorial equivale a un bucle que procesa N elementos del registro vectorial, disminuyendo de esta manera el uso de instrucciones y produciendo un aumento del rendimiento.

En la actualidad, debido a los avances en los diseños de los procesadores, así como también de la tecnología ligada a la fabricación de los mismos, la mayoría de los procesadores comerciales utilizan una microarquitectura superescalar, la cual implementa unidades funcionales independientes, permitiendo una adaptabilidad mayor a los distintos casos de uso. Con respecto a la vectorización, la unidad vectorial fue reemplazada por una unidad SIMD (*Simple Instruccion, Multiple Data*) [5], cuya función es realizar la misma operación sobre un conjunto de datos con una sola instrucción.

Con respecto a ARM, los procesadores diseñados por la compañía proveen la capacidad de extraer paralelismo a nivel de datos con estas unidades de SIMD mediante el empleo de tres tecnologías diferentes:

- ***Scalable Vector Extension* (SVE)**: Extensión vectorial para la arquitectura ARMv8 con un conjunto de instrucciones AArch64 (extensión de 64 bits de la arquitectura del ARM). Define un rango de dimensiones para los registros vectoriales que van desde 128 hasta 2048 bits de ancho (16 unidades de 128 bits de ancho).
- **Neon**: Extensión avanzada de arquitectura SIMD para los modelos Cortex-A y Cortex-R.
- **Helium**: Extensión vectorial para la arquitectura ARMv8.1-M, denominada *M-Profile Vector Extension* (MVE), enfocada al rendimiento en sectores como el aprendizaje automático y el procesamiento de señales.

En el presente proyecto, debido a las características del dispositivo, emplearemos el uso de ARM Neon para el desarrollo. Por esta razón en el siguiente subapartado, explicaremos en detalle esta tecnología.

2.2.2. Neon

ARM Neon [6] es una extensión avanzada de arquitectura SIMD, empaquetada para los procesadores ARM Cortex-A y Cortex-R, en donde los registros Neon se definen como vectores de elementos del mismo tipo de datos, admitiendo múltiples tipos, incluidas las operaciones en coma flotante y con números enteros. El tamaño individual de cada registro es de 128 bits y su contenido puede ser accedido mediante lecturas de 8, 16, 32, 64 ó 128 bits.

Con respecto a las instrucciones Neon, estas operan simultáneamente sobre múltiples elementos, como se indica en la Figura 2.2, en donde se muestra un producto de matrices con datos de 32 bits.

Como se ilustra la Figura 2.2, cada elemento de la columna de la matriz del primer operando, representado con el color azul, es multiplicado por el elemento $(0,0)$ del segundo operando (color naranja) en una sola instrucción. Previa a la multiplicación, a bajo nivel, la Figura 2.3 muestra cómo se almacenan en registros las primeras columnas correspondientes a los dos operandos, el registro $V1$ para la primera matriz y el registro $V2$ para la segunda matriz. Una vez realizado el producto, el resultado es acumulado y almacenado en un tercer registro vectorial ($V3$) correspondiente a la matriz de salida.

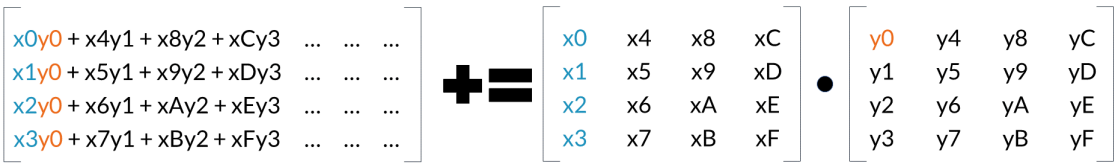

Figura 2.2: Multiplicación de matrices [7].

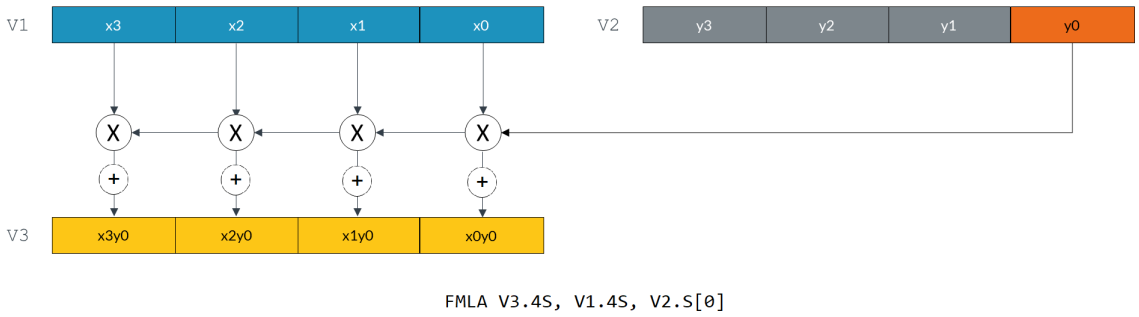

Figura 2.3: Ejecución de la instrucción FMLA [7].

Adicionalmente, la instrucción en ensamblador que se corresponde con la Figura 2.3 es la *FMLA* (*Floating-point fused Multiply-Add*) que utiliza registros SIMD y coma flotante para llevar a cabo la operación fusionada de multiplicación-suma en coma flotante. Cabe mencionar que, para la Figura 2.3, se utilizan los mismos datos de 32 bits que en la Figura 2.2.

Neon Intrinsics

Neon nos permite utilizar una serie de funciones escritas en $C/C++$ denominadas Neon Intrinsics, que el compilador posteriormente reemplaza por instrucciones Neon. De esta manera, el desarrollo con Intrinsics otorga al programador un control del código cercano a ensamblador, pero sin tener que asignar los registros, dejando esta tarea al compilador y simplificando el desarrollo. Utilizamos esta tecnología en el presente proyecto debido a que nos permite la creación de múltiples *micro-kernels* para la obtención de la configuración óptima del dispositivo utilizado con un menor tiempo de desarrollo.

A continuación, vemos un ejemplo de la función en Neon Intrinsics que realiza la llamada a la instrucción *FMLA* en ensamblador en el código 2.1.

```
1 float32x4_t vfmaq_laneq_f32 (float32x4_t a, float32x4_t b,
2                              float32x4_t v, const int lane)
```

Código 2.1: Código de la llamada a función *vfmaq_laneq_f32*.

Como podemos ver, la función es de tipo *float32x4_t*, lo que especifica que el resultado de la operación devolverá un vector de cuatro *floats*. Además, la llamada se compone de cuatro argumentos, que son:

- **a**: Vector de cuatro elementos *floats* representando el vector salida de la operación FMA.
- **b**: Vector de cuatro elementos *floats* de la matriz del primer operando.
- **v**: Vector de cuatro elementos *floats* de la matriz del segundo operando.

- **lane**: Valor entero constante que indica la posición del elemento en el vector del segundo operando utilizado para realizar el producto.

De esta manera, si queremos realizar con Neon Intrinsics la operación en la Figura 2.3, debemos utilizar la siguiente línea de código 2.2.

```
1 vfmaq_laneq_f32(V3, V1, V2, 0);
```

Código 2.2: Código de la llamada a función *vfmaq_laneq_f32*.

En esta llamada se almacena el resultado en el registro V3, V1 es el registro del vector del primer operando, V2 es el registro del vector del segundo operando y el valor 0 indica la posición en el vector del registro V2 del elemento a utilizar en la operación.

2.3 Características del dispositivo utilizado

Con respecto al *hardware*, utilizaremos el dispositivo Jetson Nano, creado por NVIDIA, que contiene una CPU de cuatro núcleos Cortex-A57 a 1,4 GHz de arquitectura RISC de 64 bits diseñada por ARM, una memoria RAM LPDDR4 de 4 GB y 16 GB de memoria eMMC. Creada para ser utilizada en el ámbito del internet de las cosas, así como también en el campo de la inteligencia artificial, una de las ventajas de este dispositivo es su bajo consumo energético. El pico teórico de rendimiento en coma flotante alcanza los 472 GFLOPS (operaciones en coma flotante por segundo) con un consumo energético de 5 Watts. Esto nos da una relación de 94,4 GFLOPS/Watt, mientras que una tarjeta gráfica NVIDIA 2080, con un consumo de 215 Watts y un rendimiento en coma flotante de 10,07 TFLOPS, obtiene unos 46,83 GFLOPS/Watt. Esto supone una diferencia de eficiencia energética de 2 veces mayor en favor de la Jetson Nano. Con respecto al precio del dispositivo, actualmente existen dos versiones: la anteriormente descrita con un valor de 99 dólares estadounidenses y la versión económica, en donde el tamaño de memoria RAM baja a los 2 GB, está valorada en 59 dólares estadounidenses.

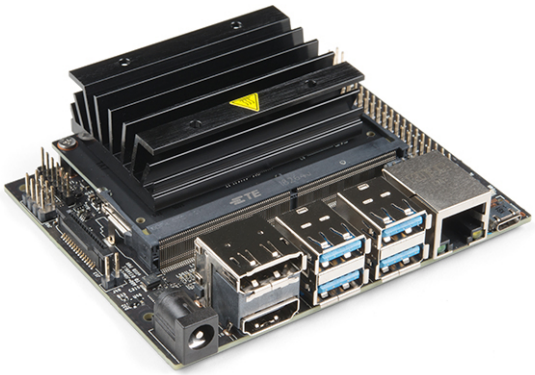

Figura 2.4: Ilustración NVIDIA Jetson Nano.

2.3.1. Memoria caché

La utilización eficiente de la memoria caché es clave para el desarrollo del presente trabajo puesto que, adaptando los tamaños de los datos a ejecutar a las capacidades del dispositivo, podremos emplear de forma óptima todas sus cualidades, tanto a nivel de almacenamiento como de rendimiento. Por esta razón, debemos conocer las características de las memorias caché de que dispone el dispositivo a utilizar.

En el módulo NVIDIA Jetson Nano tenemos memorias caché de nivel 1 y 2, cuyas configuraciones son las siguientes:

En el caso de la memoria caché de nivel 1 tenemos una memoria de 80 KBytes repartidos en dos partes [8]:

- Caché de instrucción asociativa por conjuntos de 3 vías de 48 KBytes de tamaño.
  - Bloque de longitud fija de 64 bytes.
  - Protección de paridad de 16 bits.
  - Caché de instrucciones que se comporta como *Physically-indexed and physically-tagged*(PIPT).
  - Política de reemplazo *Least Recently Used* (LRU).
  - Soporte MBIST.
- Caché de datos asociativa por conjuntos de 2 vías de 32 KBytes de tamaño.
  - Bloque de longitud fija de 64 bytes.
  - Protección *Error-Correcting Code* (ECC) de 32 bits.
  - Caché de datos que se comporta como *Physically-indexed and physically-tagged* (PIPT).
  - Ejecución *Out-of-order*, especulativa, peticiones de carga no bloqueante y no especulativa.
  - Política de reemplazo *Least Recently Used* (LRU).
  - Soporte MBIST.

Las características de la memoria caché de nivel 2 son [9]:

- Tamaño de memoria de 2 MBytes.
- Bloque de longitud fija de 64 bytes.
- Caché con indexado y etiquetado físico.
- Estructura de caché asociativa por conjuntos de 16 vías.
- Estructura de bancos en *pipeline*.
- Caché inclusiva con la caché de datos de nivel uno.
- Política de reemplazo aleatoria en caché.
- Interfaz configurable ACE de 128 bits de ancho con soporte para múltiples solicitudes pendientes.
- Copias duplicadas de los directorios de caché de datos L1 para el soporte de los fallos de coherencia.

- Protección *Error-Correcting Code* (ECC).
- Soporte opcional de *prefetch* en *hardware*.
- *Software* programable para la latencia variable de las memorias RAMs.
- Registro con soporte de segmentos a grandes tamaños de caché L2 para minimizar el impacto en los retrasos de enrutamiento.
- Soporte MBIST.

2.4 OpenMP

A partir de la segunda mitad del siglo XX, los fabricantes de circuitos integrados construían procesadores compuestos de un solo núcleo, centrando sus esfuerzos en aumentar la frecuencia de reloj para obtener una mayor cantidad de operaciones por unidad de tiempo. Este incremento alcanzó un límite en la tasa de gigahercios (GHz) en la década de los 2000 debido al calor generado por las altas frecuencias a las que llegaba el núcleo, poniendo al límite la integridad de los materiales utilizados en su construcción.

Debido a la imposibilidad de aumentar la frecuencia de reloj, los fabricantes optaron por un cambio de paradigma en la busqueda de una mayor capacidad de procesamiento. La solución fue incrementar el número de núcleos del procesador, como podemos ver en la Figura 2.5, pasando de un modelo de un solo proceso ejecutando una única tarea secuencial, a varios procesos realizando más de una tarea en simultáneo.

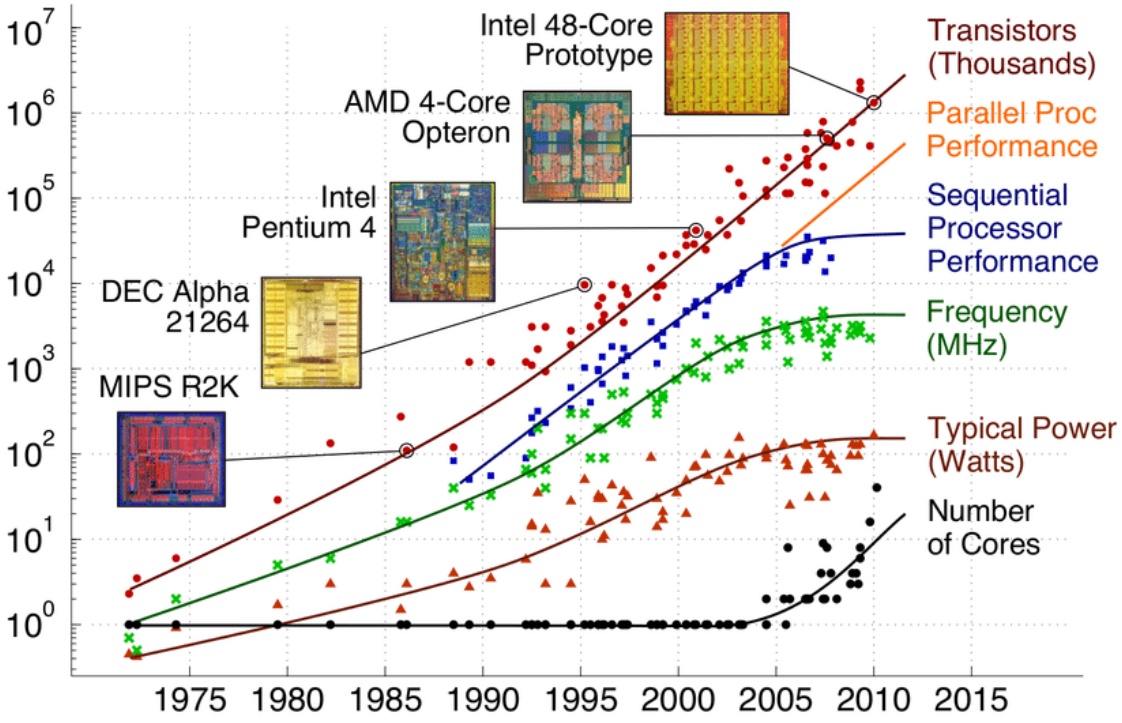

Figura 2.5: Evolución de los procesadores [10]

Como podemos ver en la Figura 2.5, se produce un punto de inflexión en el año 2005, momento en el que comienzan a aparecer los primeros procesadores multinúcleo. A partir de ese año en adelante, observamos como la frecuencia de reloj (color verde) y el

rendimiento en secuencial del procesador, representado en color azul, detienen su crecimiento y alcanzan un valor máximo. Al mismo tiempo, podemos ver cómo el aumento del número de núcleos (color negro) permite aumentar el rendimiento en paralelo del procesador (color naranja), el cual supera el crecimiento exponencial del rendimiento en secuencial que se obtenía en los años previos.

El cambio de paradigma no solo implicó una modificación en el diseño de los procesadores, sino que también fue necesaria una reestructuración a nivel *software* para que las aplicaciones puedan funcionar con varios núcleos. Adicionalmente, surgieron nuevos modelos de programación paralela basados en memoria compartida, que permiten el desarrollo de programas con soporte multinúcleo. Uno de estos modelos es OpenMP, el cual utilizamos en el presente proyecto.

OpenMP es una interfaz de programación de aplicaciones (API) que describe un conjunto de directivas del compilador, subrutinas y variables de entorno para programación paralela en memoria compartida para los lenguajes C, C++ y Fortran. Adicionalmente, esta especificación define un modelo escalable y portátil con una interfaz adaptable al desarrollo de aplicaciones paralelas en plataformas que van desde dispositivos destinados al *edge computing* hasta los superordenadores.

El modelo de ejecución paralela utilizado por OpenMP es el *fork-join*, en donde un hilo inicial que ejecuta el programa en secuencial, se bifurca en múltiples hilos al llegar a una región paralela. Estos hilos llevan a cabo tareas en paralelo definidas por la directiva de la interfaz durante la ejecución del programa. Adicionalmente, en caso de realizar intercambios de datos, la comunicación entre hilos se produce mediante variables compartidas, utilizando un modelo de memoria compartida. Una vez terminada la ejecución, se produce una sincronización de hilos al final de la región paralela y estos vuelven a unirse con el hilo inicial, continuando de este modo la ejecución en secuencial, como podemos ver en la Figura 2.6.

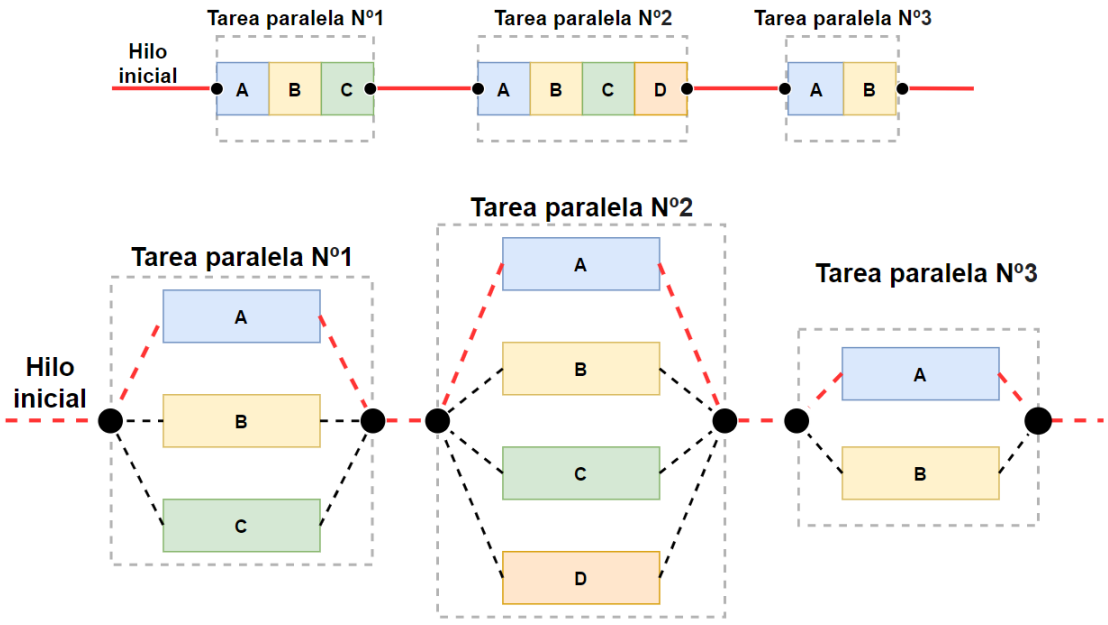

Figura 2.6: Modelo *Fork-join*.

CAPÍTULO 3
Aplicación

Hoy en día, el aprendizaje automático mediante redes neuronales profundas (DNN) no solo es empleado en tecnologías como el reconocimiento de imágenes o el procesamiento de señales, sino que, con el paso de los años se está explotando cada vez más en diversas tecnologías de diferentes campos. Para obtener un resultado con un porcentaje de precisión elevada, los modelos actuales de DNN para estas aplicaciones necesitan ser entrenados sobre una cantidad considerable de datos. Por esta razón, tanto la dimensión (número de capas) como la complejidad de las redes neuronales profundas está creciendo rápidamente, produciendo que los modelos recientes utilicen hasta miles de millones de parámetros. Un ejemplo de ello se puede ver en la competición anual de ImageNet [11], en donde la red neuronal ganadora en 2013, AlexNet [12], empleaba ocho capas. En 2015, la red neuronal ResNet [13] utilizó un total de 152 capas para conseguir el primer puesto en la competición, multiplicando por 19 el número de capas empleadas en tan solo dos años.

Como podemos ver, esta tecnología requiere no solo de una cantidad elevada de datos, sino también de recursos computacionales para poder llevar a cabo su ejecución. Por este motivo, el empleo de técnicas de optimización es de vital importancia en el entrenamiento de las redes neuronales profundas, obteniendo de esta manera el mayor rendimiento posible del dispositivo utilizado. En el presente trabajo se abordan estrategias de optimización que aprovechan las características del dispositivo, teniendo en cuenta tanto la memoria como las unidades vectoriales del procesador, para ejecutar en el menor tiempo posible el entrenamiento y la inferencia de los modelos de redes neuronales. Antes de detallar el programa en cuestión, se expone la teoría correspondiente a las redes neuronales convolucionales para comprender la importancia que tiene la multiplicación de matrices en estas estructuras y poder establecer su relación con el presente proyecto.

3.1 Redes Neuronales Convolucionales

En el campo del aprendizaje automático, una de las estructuras más empleadas son las redes neuronales convolucionales (CNN) [14], caracterizadas por ser un tipo especializado de perceptrones multicapa con aplicación en sistemas de recomendación, reconocimiento de imágenes, análisis de imágenes médicas, procesamiento del lenguaje natural, entre otros. Las redes neuronales convolucionales presentan una regularización implícita que utiliza la estructura jerárquica de los datos para evitar el sobreajuste, es decir, que previene que la red se sobreentrene, de forma que el algoritmo aprenda patrones no generalistas. Esta propiedad se consigue mediante la aplicación de operadores de convolución, que concentran una fracción significativa del coste computacional de las redes neuronales convolucionales.

Una de las estrategias de alto rendimiento más utilizadas para realizar un operador de convolución consiste en aplicar la transformada im2col [15] a las entradas de activación de las capas. Esta transformación es seguida de una multiplicación general de matrices (GEMM), habitualmente optimizada mediante la utilización de bibliotecas de álgebra lineal de alto rendimiento como las enumeradas en la sección 2.1. Cabe mencionar que el algoritmo im2col realiza la reordenación de bloques de un determinado tamaño de una matriz de entrada de N dimensiones en una serie de columnas, transformando de esta manera una «imagen» en una matriz de dos dimensiones. En la Figura 3.1, vemos un ejemplo de esta estrategia.

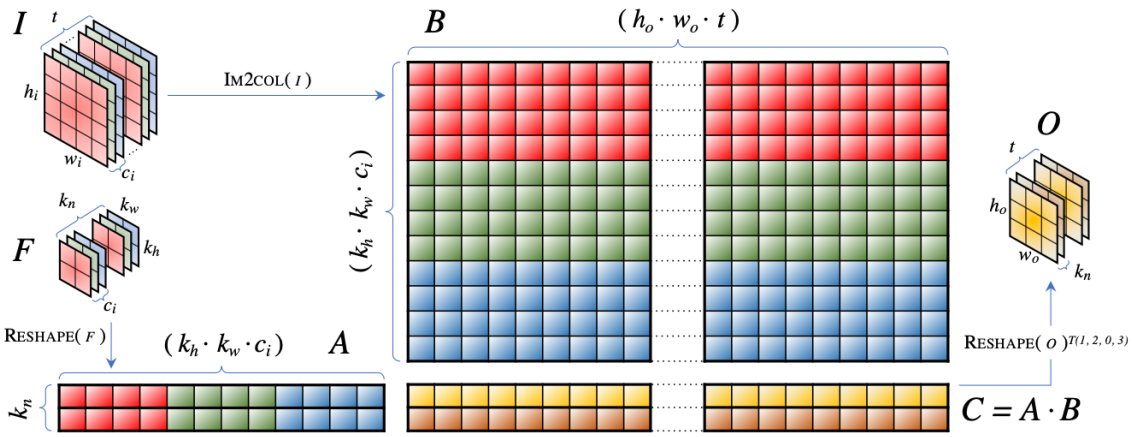

Figura 3.1: Ejemplo de operador de convolución mediante la transformación im2col [16].

La Figura 3.1 representa gráficamente la transformación del operador de convolución en una multiplicación de matrices realizada mediante una transformada im2col. Detallando cada una de las estructuras expuestas en la imagen, tenemos la matriz F, de cuatro dimensiones, que representa una serie de filtros de la red neuronal y está compuesta por:

- k_n: número de filtros.
- k_h: número de filas del filtro.
- k_w: número de columnas del filtro.
- c_i: número de canales de entrada.

Para poder utilizar la matriz F como un operando de GEMM, es necesario aplicarle una redimensión para obtener una matriz en dos dimensiones. Para este propósito se le aplica una operación *reshape*, como podemos ver en la Figura 3.1, obteniendo de esta manera una nueva matriz A de tamaño $k_n \times (k_h \cdot k_w \cdot c_i) = m \times k$.

Con respecto a la matriz I, esta representa el conjunto de imágenes de entrada (o *batch*) con las cuales se realiza el entrenamiento de la red neuronal, y se compone de las siguientes dimensiones:

- h_i: número de filas de las imágenes de entrada.
- w_i: número de columnas de las imágenes de entrada.
- c_i: número de canales de entrada.
- t: número de muestras (imágenes) que componen la matriz de entrada.

Al realizar la transformada im2col, pasamos de una matriz de entrada de cuatro dimensiones a una matriz B de dos dimensiones $(k_h \cdot k_w \cdot c_i) \times (h_o \cdot w_o \cdot t) = k \times n$. Si observamos las variables h_o y w_o, estas se corresponden con los tamaños de fila y columna, respectivamente, de la matriz de salida y se calculan mediante las siguientes fórmulas:

- $h_o = [(h_i - k_h + 2p_v)/s_v + 1]$.
- $w_o = [(w_i - k_w + 2p_h)/s_h + 1]$.

Adicionalmente, en ambas ecuaciones se utilizan las variables correspondientes a técnicas relativas al *padding* (p_v y p_h) y al *stride* (s_v y s_h) que no se detallan, puesto que no entran dentro del alcance del presente trabajo para no alterar la descripción del ejemplo expuesto y cuyos valores son ($p_v = p_h = 0$) y ($s_v = s_h = 1$).

Una vez definidos los contenidos de estas dos matrices, el siguiente paso es la aplicación de la GEMM, obteniendo una matriz de salida C. Sobre esta estructura, posteriormente, se vuelve a realizar un *reshape* y una transposición para obtener la matriz resultado O.

Centrándonos en los tamaños de las matrices A y B, anteriormente comentadas, vemos que la dimensión n está estrechamente relacionada con el número de muestras (t) procesadas de una vez. Generalmente, el número de muestras utilizadas en una red neuronal es considerablemente elevado, consiguiendo de esta manera un valor de n mayor de las dimensiones m y k. Como veremos en el siguiente apartado, el algoritmo de base implementado en BLIS no está adaptado adecuadamente para matrices de entrada con estas características. Por esta razón se desarrollan posteriormente una serie de variantes del algoritmo base y una evaluación de los mismos para obtener la estrategia que se adapta mejor a este tipo de estructuras de entrada.

3.1.1. ResNet50 v1.5

Una de las redes neuronales más destacadas para el reconocimiento de imágenes es ResNet50 [13]. Este entorno de trabajo, a diferencia de otras redes neuronales profundas, reformula de forma explícita las capas como funciones residuales de aprendizaje con referencia a las entradas de las capas, en lugar de aprender funciones sin referencia, como podemos ver en la Figura 3.2.

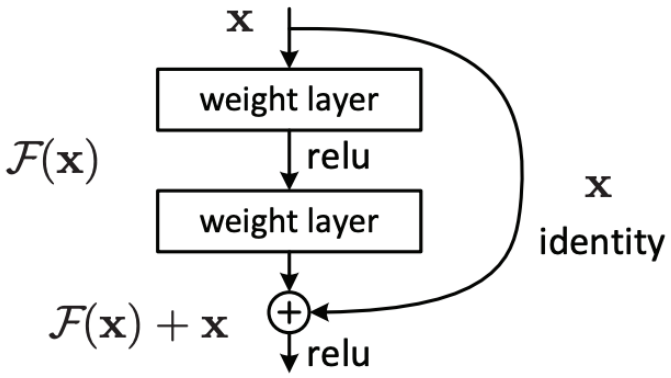

Figura 3.2: Bloque de aprendizaje residual.

En las redes neuronales compuestas por capas planas, el resultado de una capa se obtiene de la salida producida por la capa anterior sin utilizar ninguna función de referencia. En el caso de ResNet50, se emplean capas residuales en donde la función identidad

x es usada como dato de entrada al bloque de la red, el cual se bifurca en dos caminos. El primero realiza el aprendizaje del bloque, atravesando las capas y obteniendo como resultado una desviación de la función identidad llamada función residual $\mathcal{F}(x)$. En el otro camino, denominado atajo, el mismo dato utilizado en la entrada del bloque pasa a la salida, evitando su entrenamiento. De esta forma se obtiene el mapeo subyacente deseado ($\mathcal{H}(x) = \mathcal{F}(x) + x$), en donde a diferencia de las redes neuronales que utilizan capas planas, se resuelve el problema del desvanecimiento del gradiente, el cual a medida que aumenta el número de capas de la red, provoca que el entrenamiento obtenga peores resultados. Si vemos la fórmula resultante, gracias al atajo, mantenemos el valor de entrada al bloque y en caso de que la función residual sea cercana a 0 conseguimos que, como mínimo, la siguiente capa obtenga un desempeño igual que la anterior, evitando un aumento del porcentaje de error. De esta manera, ResNet permite utilizar un número de capas elevado (Figura 3.3), sin sufrir una pérdida en la precisión.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figura 3.3: Arquitectura de ResNet50.

En el presente trabajo, para el apartado de la evaluación, utilizaremos como valores de entrada los tamaños de las matrices empleados en ResNet50 v1.5. Este es una versión modificada de ResNet50, que obtiene una precisión un 0.5 % mayor a cambio de perder un rendimiento del 5 % (imágenes/segundo) con respecto a la versión original.

3.2 Programa

Basic Linear Algebra Instantiation Software (BLIS) es una infraestructura desarrollada en la Universidad de Texas (Austin, Estados Unidos), que facilita el desarrollo rápido de *Basic Linear Algebra Subprograms* (BLAS) [17]. BLIS ofrece una formulación de alto nivel de GEMM (multiplicación general de matrices), en forma de una implementación genérica de este *kernel* compuesta por cinco bucles anidados que incluyen dos rutinas de empaquetado y un *microkernel*, como se refleja la Figura 3.4. Este *microkernel* se codifica como un bucle alrededor de una actualización de rango 1 y comprende todas las operaciones aritméticas.

Entrando en detalle sobre la estructura vista en la Figura 3.4, y utilizando una aproximación *top-down*, la función dispone de cinco bucles anidados, en donde los tres bucles más externos realizan el producto de matrices mientras que la jerarquía subyacente tiene como objetivo la eficiencia de la multiplicación. La técnica empleada para lograr este aumento de rendimiento se basa en utilizar dos rutinas de empaquetado de datos y una posterior ejecución de las operaciones aritméticas en el *micro-kernel*. Con respecto a las

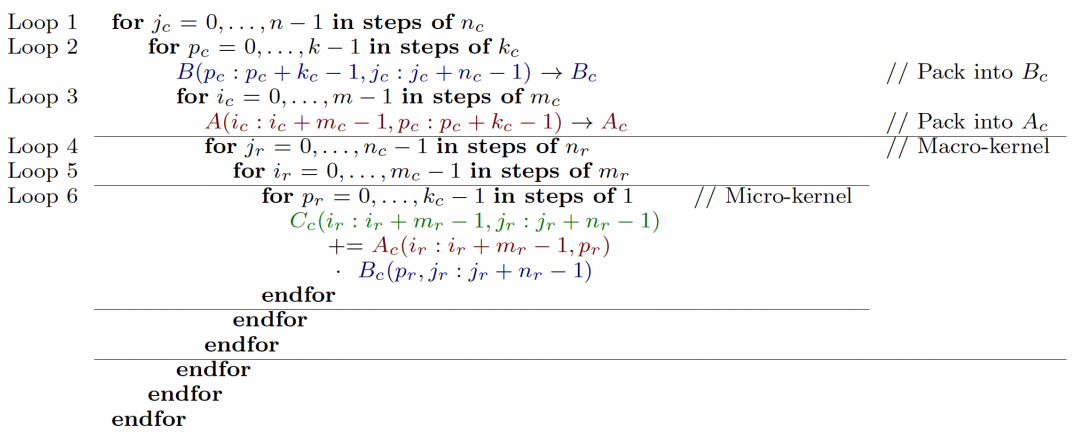

Figura 3.4: Estructura del GEMM de BLIS

rutinas de empaquetamiento, estas son las encargadas de realizar el alojamiento de los datos en los diferentes niveles de caché, reorganizando pequeños bloques de datos de los operandos de entrada A y B en dos búferes (A_c y B_c respectivamente) para garantizar el rápido acceso a los contenidos de estos búferes durante la ejecución del *micro-kernel*, como podemos ver en la Figura 3.5.

El *micro-kernel* debe ser acorde a la arquitectura específica del dispositivo empleado para alcanzar un rendimiento lo más óptimo posible. A diferencia de los bucles superiores, que están implementados en un lenguaje de alto nivel como C, dotando de portabilidad al programa, el *microkernel* se desarrolla utilizando un lenguaje que permita ejercer un control directo sobre el *hardware* del dispositivo, como el ensamblador. De esta manera, se fuerza una adaptación ajustada y eficiente a la configuración de la memoria caché del procesador a utilizar.

Una vez explicada la estructura general del programa, se realiza un desglose de las variables presentes en la Figura 3.4 y su implicación en el código. En primer lugar, podemos ver que se emplean las variables m_c, k_c y n_c como las dimensiones de los bloques de las matrices que serán empaquetados en las memorias caché. En cambio las constantes m_r y n_r definen las dimensiones de los bloques que serán accedidos desde dentro del *micro-kernel*. Cabe destacar que estas constantes son inicializadas previamente con los valores óptimos para los tamaños de caché del dispositivo que se emplea.

Con respecto a los bucles, el algoritmo en la Figura 3.4 se puede denominar como la variante (j_c, p_c, i_c, j_r, i_r), debido a que las dimensiones del problema (n, k, m, n_c, m_c) se recorren en ese orden (*Loop 1*, *Loop 2*, *Loop 3*, *Loop 4* y *Loop 5*). Entrando en detalle sobre los tres primeros bucles (ordenados de externo a interno), vemos que en el segundo bucle (k) se empaquetan los datos de la matriz de entrada B. La rutina que realiza este empaquetamiento (Figura 3.7) se encarga de almacenar un bloque de dimensión $k_c \times n_c$ elementos de esta matriz en memoria caché de nivel tres o, en caso de que el dispositivo no tenga este nivel como es nuestro caso, en memoria RAM. Además, para una mayor eficiencia, la submatriz se ordena por filas alineándose en bloques de $k_c \times n_r$, que posteriormente serán enviados a la caché L1 en los bucles internos de la Figura 3.4. En cuanto al tercer bucle, a diferencia de la anterior rutina de empaquetamiento, en este caso almacena en memoria caché L2 un bloque de $m_c \times k_c$ datos de la matriz A. Adicionalmente, esta estructura está ordenada en columnas y dividida en bloques más pequeños con dimensiones de $m_r \times k_c$. Todo esto lo podemos ver gráficamente en la Figura 3.6.

Con respecto a los dos bucles restantes, *Loop 4* y *Loop 5*, (variables n_r y m_r, respectivamente), estos se denominan bucles interiores y son los encargados de realizar las llama-

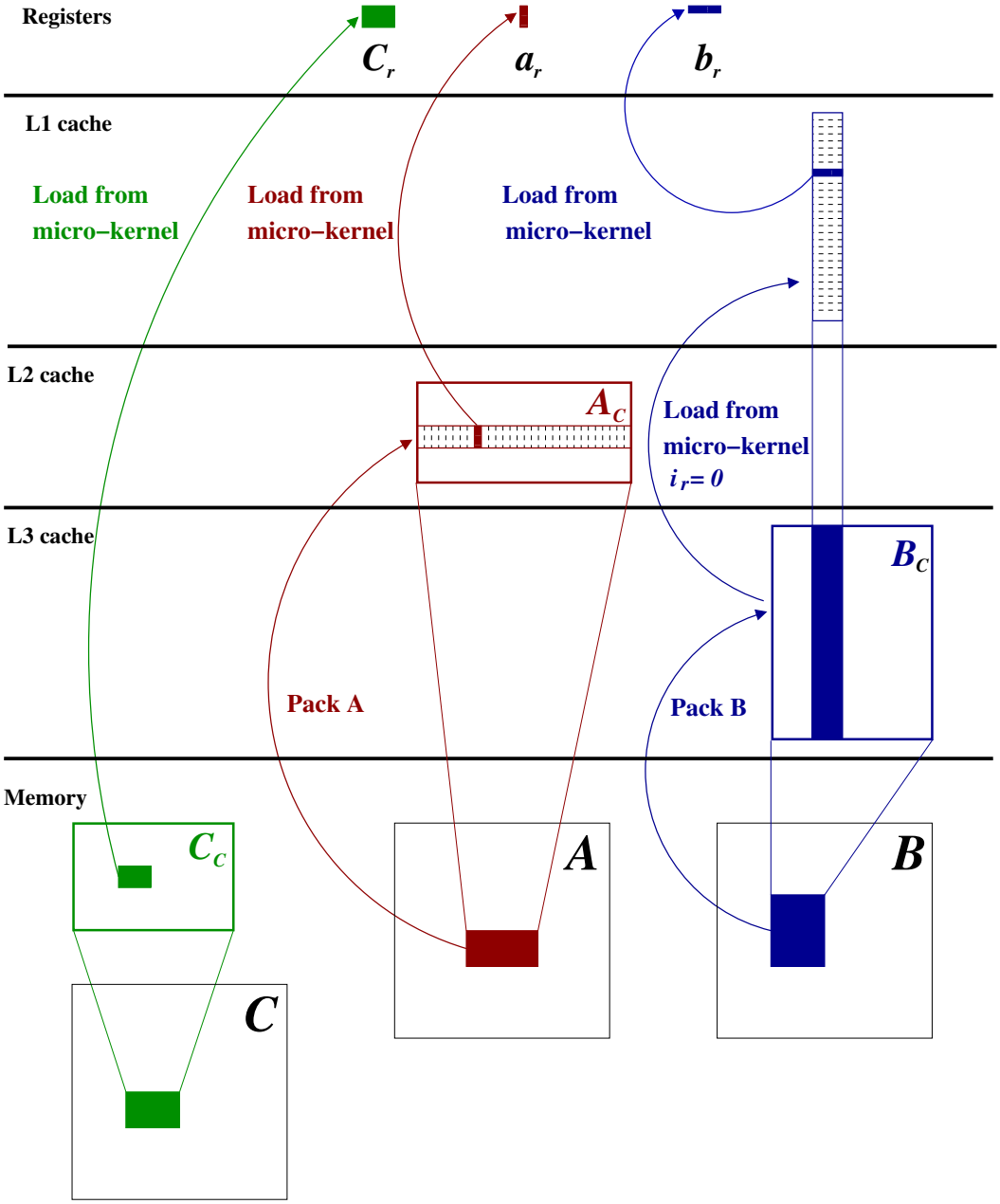

Figura 3.5: Movimientos de *Ac* y *Bc* entre las memorias del dispositivo [18].

das al *micro-kernel* para la ejecución del producto de matrices empleando los punteros a memoria de las submatrices alojadas previamente en memoria caché. De esta manera se crea una multiplicación de matrices a bloques, como podemos ver en la Figura 3.8.

3.3 Descripción del problema

Una de las características que poseen las matrices de entrada cuando se realizan el entrenamiento o la inferencia de las redes neuronales, es que una de las dimensiones de estas estructuras algebraicas es significativamente menor con respecto a las dos restantes. Debido a esto, a continuación ejecutaremos la implementación del producto de matrices de BLIS con dos operandos de gran dimensión y uno pequeño, para comprobar la idoneidad del código bajo estas circunstancias.

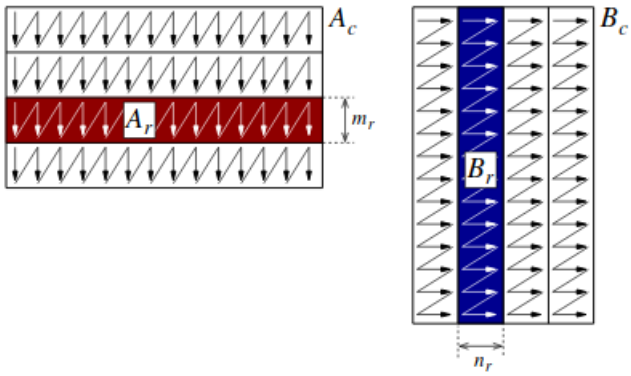

Figura 3.6: Empaquetamiento de valores sobre las matrices de entrada [19].

L1: **for** $j_r = 0, \ldots, n_c - 1$ **in steps of** n_r
$i = 0$
L2: **for** $p_s = 0, \ldots, k_c - 1$
L3: **for** $j_s = 0, \ldots, n_r - 1$
$B_c[i][j_r] = B[p_c + p_s][j_c + j_r + j_s]$
$i = i + 1$

Figura 3.7: Estructura de la función que realiza el empaquetamiento de la matriz B.

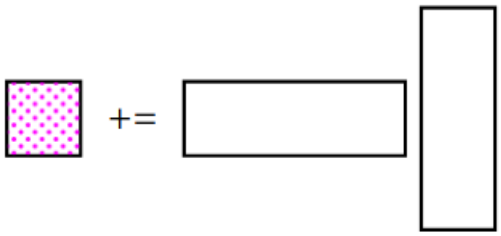

Figura 3.8: Subbloques conseguidos por los tres bucles externos correspondientes a las matrices de izquierda a derecha: $C+ = A * B$ [20].

Analizando las gráficas de las Figuras 3.9 y 3.10, la línea de color amarillo representa el rendimiento de una matriz cuadrada, la cual refleja un incremento de los *GFLOPS* a medida que el tamaño del problema crece. Con respecto a las matrices rectangulares, notamos que el comportamiento no es el esperado en las dos configuraciones, cuando m es la de menor dimensión en la Figura 3.9 y n cuando es el tamaño más pequeño en la Figura 3.10, ya que obtenemos una pérdida del rendimiento cuando las dimensiones van en aumento, llegando a producirse varios valles en casos concretos del experimento. Adicionalmente, visualizamos que no solo el tamaño de la matriz es influyente, sino que dependiendo de cuál sea la dimensión de menor proporción, obtenemos valles en diferentes puntos del eje horizontal variando incluso su periodicidad. Como ejemplo, en la Figura 3.9 se produce el primer valle para el tamaño $k = n = 750$, mientras que para la configuración de la Figura 3.10 este aparece en $m = k = 1000$.

Este análisis apunta en la dirección de que una modificación en el patrón de accesos a memoria caché, obtenida mediante un intercambio de bucles, puede mejorar el rendimiento. Teniendo en cuenta las características de las memorias caché y las dimensiones

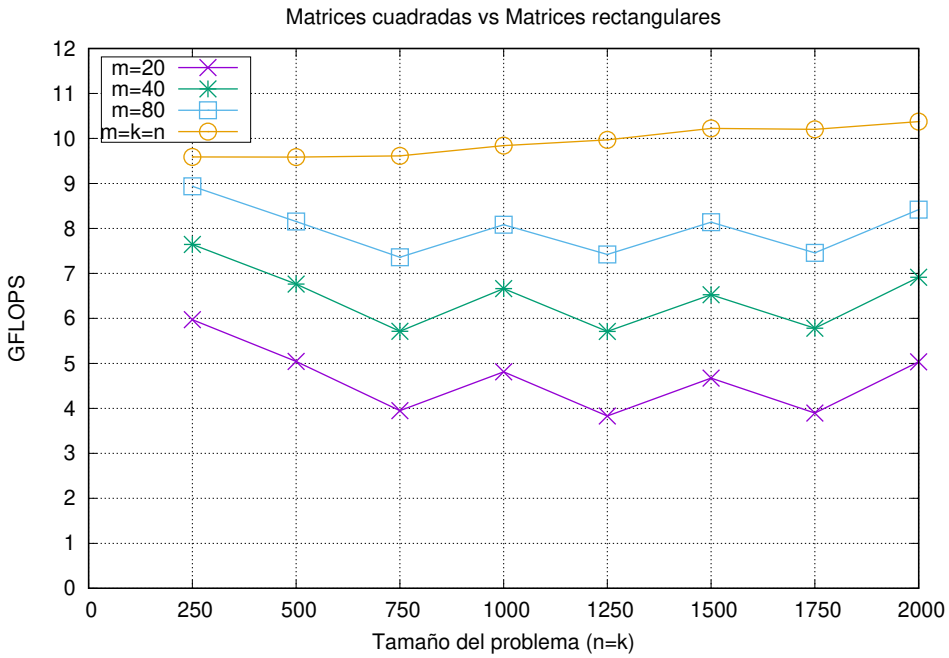

Figura 3.9: Ejecución del programa en simple precisión con diferentes tamaños de m.

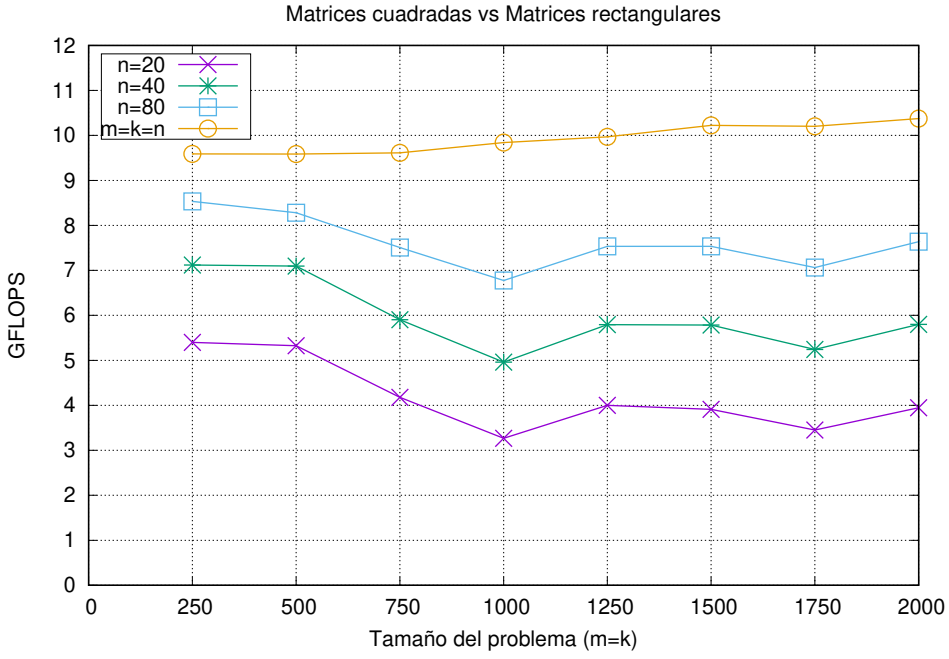

Figura 3.10: Ejecución del programa en simple precisión con diferentes tamaños de n.

de las matrices empleadas en este caso, analizaremos diferentes alternativas y evaluaremos su impacto en cuanto al rendimiento.

3.4 Estrategias de optimización

3.4.1. Algoritmos Residentes

Los algoritmos residentes [20] expresan la forma en la que son almacenadas las matrices en los registros del dispositivo. Hay tres tipos de algoritmos: A residente, B residente y C residente, en donde cada letra expone qué bloque de la matriz permanece en los registros del dispositivo durante la ejecución del bucle más interno del producto matricial.

Considerando la operación $C \mathrel{+}= A * B$, en donde A y B son las matrices de entrada y C la de salida, el algoritmo con A residente almacenará en registros una parte de la matriz A durante el bucle mencionado. De igual manera, B residente y C residente harán lo mismo con las matrices B y C respectivamente. A continuación se describen en detalle los algoritmos residentes.

C Residente

Este algoritmo es el expuesto en la sección 3.2, en donde se computa en seis bucles la operación $C+ = A * B$. Como podemos ver en la Figura 3.11, en cada uno de los bucles se realizan diversas acciones como la partición, empaquetamiento y almacenamiento de las matrices en los diferentes tipos de memoria del dispositivo. Para ello, previamente se declaran las dimensiones m_c, k_c, n_c, m_r y n_r para realizar el particionado de las matrices.

Adicionalmente, vemos dos variantes del mismo algoritmo, la variante $B3A2B1C0$ situada a la izquierda de la Figura 3.11 y la $A3B2A1C0$ a la derecha. Los nombres de estas variantes nos indican el nivel de memoria en donde son alojados los diferentes bloques de las matrices. Tomando como ejemplo la variante $B3A2B1C0$ observamos que:

- **B3**: Un bloque de la matriz B de tamaño $k_c \times n_c$ se almacena en la memoria caché de nivel 3.
- **A2**: Un bloque de la matriz A de tamaño $m_r \times k_c$ se almacena en la memoria caché de nivel 2.
- **B1**: Un bloque de la matriz B de tamaño $k_c \times n_r$ se almacena en la memoria caché de nivel 1.
- **C0**: Un bloque de la matriz C de tamaño $m_r \times n_r$ se almacena en registros.

Cabe destacar que el dispositivo utilizado solo dispone de dos niveles de memoria caché, por tanto, todas las referencias que se hagan de forma teórica a la caché de nivel 3, a nivel práctico se refieren a la memoria RAM.

Como se puede ver en la Figura 3.11, en los bucles externos (*Loops* 1 a 4) se realiza el empaquetamiento de las matrices «no residentes» (en este algoritmo particular, A y B), las cuales son particionadas en bloques y posteriormente almacenadas en los niveles de memoria con mayor capacidad de almacenamiento. Conforme se vaya entrando en los bucles internos, estos bloques son de menor dimensión permitiendo ser guardados en memorias más rápidas pero con menor capacidad de almacenamiento.

Relativo a las variantes, su uso depende del tamaño de las matrices de entrada. Si la matriz B tiene unas dimensiones mayores a las de A, se ubicarán los bloques de la primera matriz en memoria RAM. De esta manera se utiliza de forma óptima la capacidad de almacenamiento de las distintas memorias.

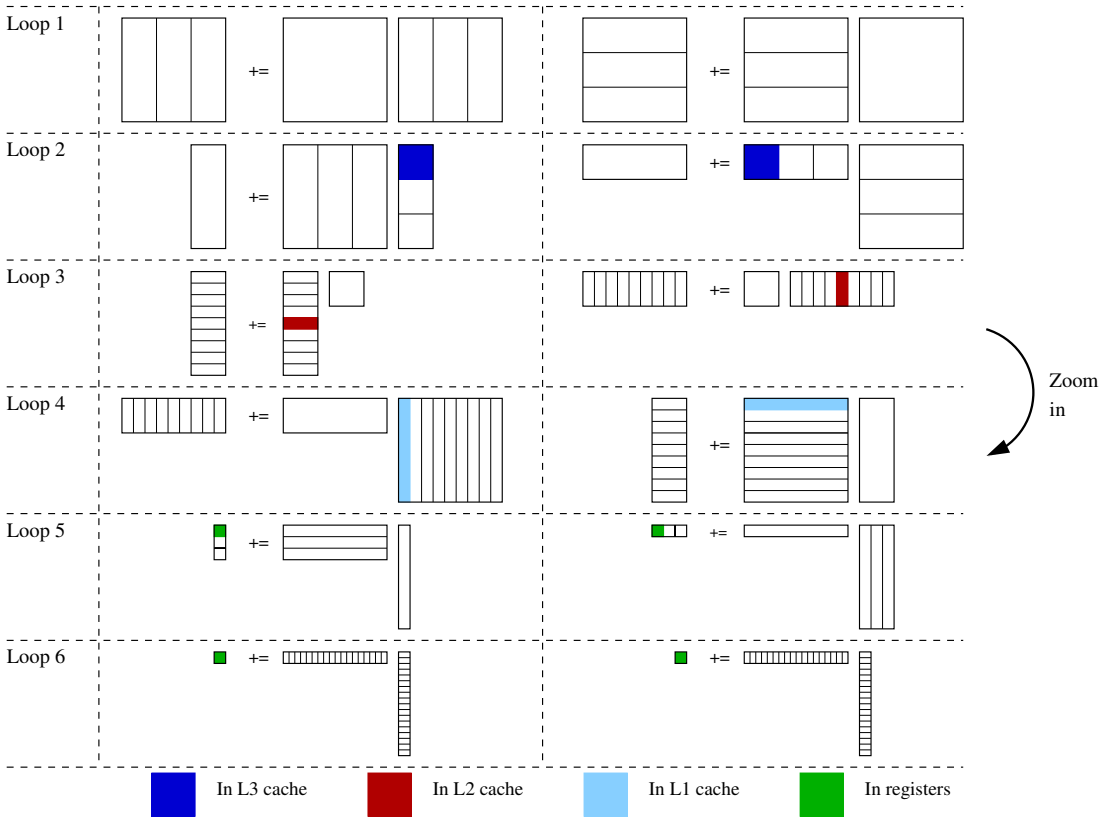

Figura 3.11: Relación entre la iteración del algoritmo C Residente y la utilización de las memorias del dispositivo.

Para el caso de interés en este trabajo (inferencia con CNNs), los valores que toma m son pequeños en comparación con los de k y n. Por ello, se plantea el almacenamiento de la matriz A en memoria RAM para, a posteriori, empaquetar parte de esta en la memoria de caché L1, obteniendo un mejor aprovechamiento del espacio en esta memoria gracias al tamaño de la dimensión m.

A Residente

El algoritmo con A residente, a diferencia del de C residente, guarda en registros los valores de la matriz A. Como podemos ver en la parte izquierda de la Figura 3.12, se emplea el mismo número de bucles para obtener el resultado de la multiplicación de matrices.

Otra de las diferencias con respecto al algoritmo visto en la sección anterior es que, al ser una matriz de entrada la que reside en los registros, esto obliga a que la matriz C tenga que ser empaquetada para su envío a la memoria caché y posteriormente desempaquetada para su lectura y escritura en el *micro-kernel*.

Adicionalmente, como podemos ver en la Figura 3.12, la columna que se sitúa a la derecha corresponde al algoritmo B residente el cual, como su nombre indica, almacena un bloque de la matriz B en registros. Este algoritmo es simétrico al A residente, en donde además de la diferencia comentada, se realiza la lectura de los elementos por filas desde el *micro-kernel*, a diferencia de su contraparte donde se hace mediante una ordenación por columnas.

Una vez expuesta la teoría, vemos que el algoritmo C residente ofrece, sobre el papel, una solución más óptima ya que, al ser C una matriz de salida, está expuesta a lectu-

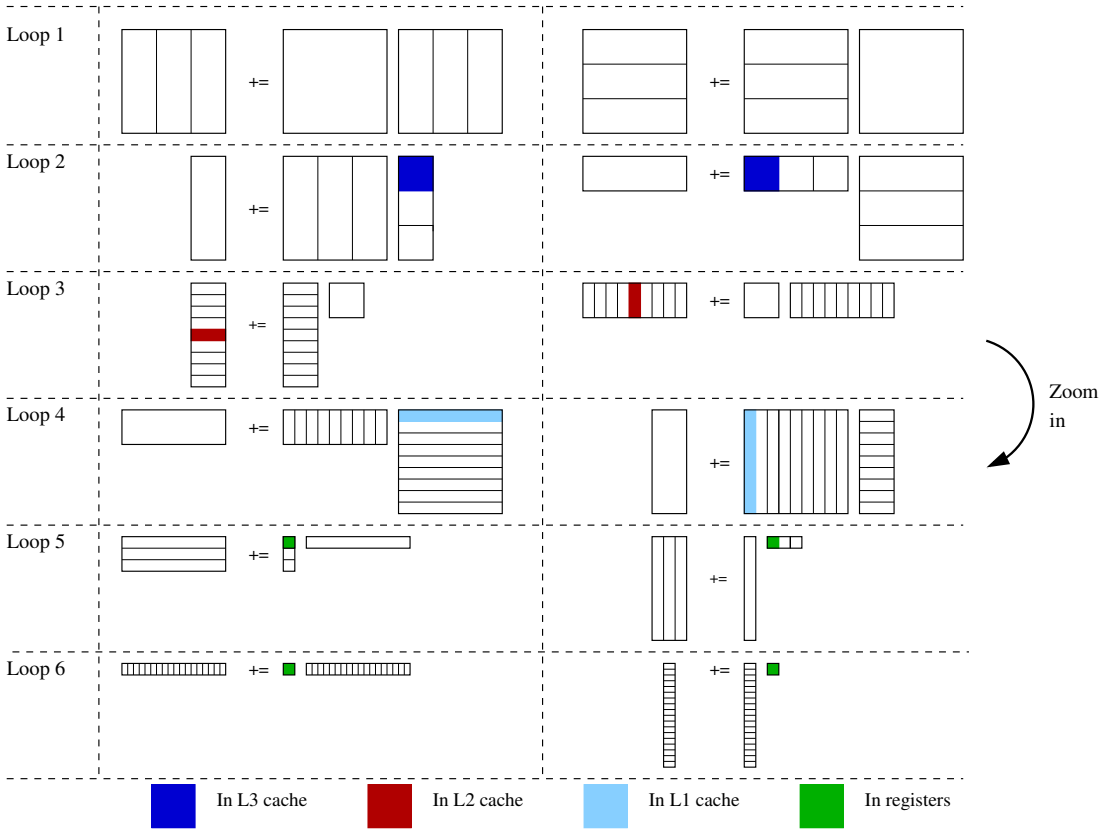

Figura 3.12: Relación entre la iteración de los algoritmos A (izquierda) y B (derecha) residentes y la utilización de las memorias del dispositivo.

ras y escrituras durante el transcurso de la operación, además de evitar el proceso del desempaquetado de esta matriz dentro del *micro-kernel*. Aún así, existen casos en los que el empleo de los algoritmos con A y B residentes puede suponer una mejora en el rendimiento, dependiendo de las dimensiones de las matrices. Para ello, es necesario que la dimensión compartida por los otros dos operandos sea considerablemente mayor, de modo que el coste de mover la matriz residente a los registros resulte amortizable.

3.4.2. Micro-kernels

En el *micro-kernel* se realiza la ejecución de la operación de la multiplicación de matrices. Para obtener un mejor rendimiento del mismo, se emplea el uso de llamadas a funciones de Neon intrinsics, que el compilador se encargará de traducir en código ensamblador.

Los tamaños $m_r \times n_r$ que se emplean en el presente trabajo son los siguientes:

- 4×4.
- 4×8.
- 8×8.
- 8×12.

Además, cada algoritmo residente necesitará una adaptación del *micro-kernel* para poder efectuar el cómputo. En el caso del *micro-kernel* C residente, se inicializa con el valor

0 los elementos correspondientes al bloque de registros perteneciente a la matriz C. Posteriormente, por cada iteración se multiplican los valores que se leen de los bloques de A y B situados en caché en los registros vectoriales del procesador. Por último, se almacena el resultado teniendo en cuenta los escalares *alpha* y *beta*, realizando las operaciones correspondientes en cada caso.

En comparación, para el *micro-kernel* con A residente, primero se realiza el desempaquetado de C, la lectura de los valores situados en registros del bloque de A y el producto matricial, en este orden. Finalmente, se almacena el resultado en el bloque de la matriz de salida (C).

Por último, para obtener un mejor rendimiento de los *micro-kernels*, pasaremos a la optimización de los mismos utilizando técnicas relativas a la gestión de la memoria.

Unroll y *prefetch*

La primera técnica a utilizar es el *unroll*, que consiste en desenrollar el bucle del *micro-kernel* aumentando el número de operaciones por iteración al computar dos o más operaciones matriciales por iteración en lugar de solo una. De esta manera, reducimos el número de instrucciones de salto, así como su coste computacional.

La segunda técnica es el *prefetch*. Con este método cargaremos de forma adelantada datos correspondientes a las matrices no residentes, de las próximas iteraciones, en los registros del procesador. Como podemos ver, esta técnica se complementa con el *unroll* ya que, por ejemplo, por cada iteración del bucle, estamos calculando dos multiplicaciones de matrices y realizando el *prefetch* de los dos siguientes bloques a utilizar. Esta técnica tiene su limitación en el número máximo de registros que posee el dispositivo. En este caso en concreto, disponemos de 32 registros de 16 *Bytes* cada uno. En todas las versiones creadas de los *micro-kernels* podemos emplear el *prefetch* sin problemas, con la excepción del *kernel* de dimensión 8×12. En esta función, utilizamos 24 registros para almacenar los elementos de la matriz residente y 5 registros para almacenar los subbloques que se encuentran en memoria caché de las matrices no residentes. Los últimos 3 registros libres los utilizamos para habilitar *prefetch* de una de las matrices no residentes, mientras que la estructura restante reutilizamos el registro para realizar un «*semiprefetch*». A continuación podemos ver un ejemplo del código 3.1, correspondiente al *kernel* 4×4 con A residente, utilizando las técnicas comentadas anteriormente.

```
1  //Prefetch de las siguientes dos iteraciones de las columnas de las matrices de
       Cr como Br, respectivamente.
2  C1 = vmovq_n_f32(0);
3  C2 = vmovq_n_f32(0);
4
5  B1 = vld2q_f32(&Br[0]);
6  B2 = vld2q_f32(&Br[KR]);
7
8  //Unroll, por cada iteracion realizaremos dos ejecuciones correspondientes a
       dos columnas de Cr
9  for ( j=1; j<nc-2; j+=2 ) {
10     //Prefetch de las siguientes dos iteraciones de las columnas de las
           matrices de Cr como Br, respectivamente.
11     C3 = vmovq_n_f32(0);
12     C4 = vmovq_n_f32(0);
13     B3 = vld1q_f32(&Br[(j+1)*KR]);
14     B4 = vld1q_f32(&Br[(j+2)*KR]);
15
16     //Ejecucion de la iteracion j-1
17     C1 = vfmaq_laneq_f32(C1, A1, B1, 0);
18     C1 = vfmaq_laneq_f32(C1, A2, B1, 1);
19     C1 = vfmaq_laneq_f32(C1, A3, B1, 2);
```

```
20      C1 = vfmaq_laneq_f32(C1, A4, B1, 3);
21
22      //Ejecucion de la iteracion j
23      C2 = vfmaq_laneq_f32(C2, A1, B2, 0);
24      C2 = vfmaq_laneq_f32(C2, A2, B2, 1);
25      C2 = vfmaq_laneq_f32(C2, A3, B2, 2);
26      C2 = vfmaq_laneq_f32(C2, A4, B2, 3);
27
28      //Almacenamiento de los valores obtenidos en la matriz Cr
29      vst1q_f32(&Cr[(j-1)*MR], C1);
30      vst1q_f32(&Cr[(j)*MR], C2);
31
32      //Asignacion de los valores cargados por prefetch en los registros
33      C1 = C3;
34      C2 = C4;
35      B1 = B3;
36      B2 = B4;
37  }
38
39  //Realizamos las dos ultimas iteraciones
40  C1 = vfmaq_laneq_f32(C1, A1, B1, 0);
41  C1 = vfmaq_laneq_f32(C1, A2, B1, 1);
42  C1 = vfmaq_laneq_f32(C1, A3, B1, 2);
43  C1 = vfmaq_laneq_f32(C1, A4, B1, 3);
44
45  C2 = vfmaq_laneq_f32(C2, A1, B2, 0);
46  C2 = vfmaq_laneq_f32(C2, A2, B2, 1);
47  C2 = vfmaq_laneq_f32(C2, A3, B2, 2);
48  C2 = vfmaq_laneq_f32(C2, A4, B2, 3);
49
50  vst1q_f32(&Cr[(nc-2)*MR], C1);
51  vst1q_f32(&Cr[(nc-1)*MR], C2);
```

Código 3.1: Código del *kernel* 4×4 de A residente con técnicas de de *unroll* y *prefetch*.

A continuación, en las Figuras 3.13 y 3.14, ilustramos el impacto sobre el rendimiento que produce la inclusión de técnicas de *unroll* y *prefetch* en los diferentes *micro-kernels* desarrollados, utilizando tanto la variante con *A* residente (*B3C2B1A0*) como con *C* residente (*A3B2A1C0*). Adicionalmente, las compararemos con su versión sin el empleo de estos métodos de gestión de memoria.

Como podemos ver en la Figura 3.13, obtenemos una mejora del rendimiento cuando se emplean estas técnicas de *unroll* y *prefetch* (colores azul y celeste), con respecto a no utilizarlas (colores naranja y amarillo), para todos los *micro-kernels* exceptuando la versión 8×8. Además, dependiendo del *micro-kernel*, estas mejoras en el rendimiento pueden ser considerables como es en el caso de la versión 4×4, en donde se obtiene un aumento de 0,73 GFLOPS para la variante *A3B2A1C0*.

En la Figura 3.14, mostramos que el uso de las técnicas anteriormente comentadas empeoran el rendimiento con respecto a los modelos sin gestión de memoria. Al desconocer el trabajo que realiza internamente el compilador, no es directo conocer la optimización que está realizando, por lo que el impacto que producen estas tecnologías puede ser negativo o positivo, dependiendo de la interpretación del compilador.

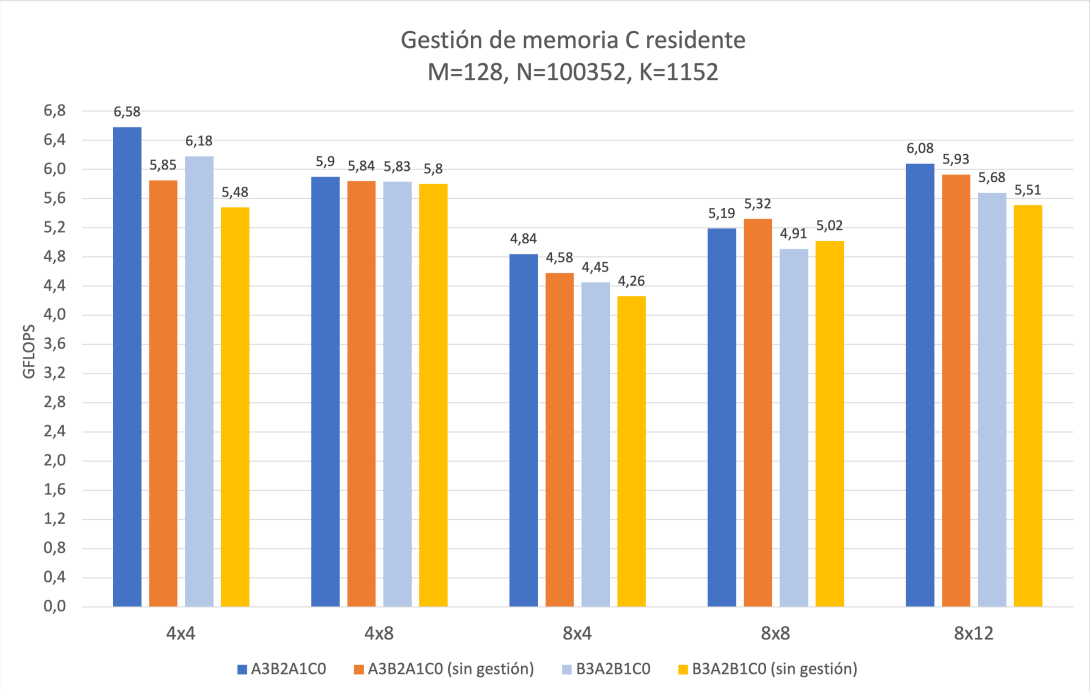

Figura 3.13: Diferencia de rendimiento entre las diferentes variantes de C residente utilizando las técnicas de *prefetch* y *unroll* contra las versiones sin gestión de memoria.

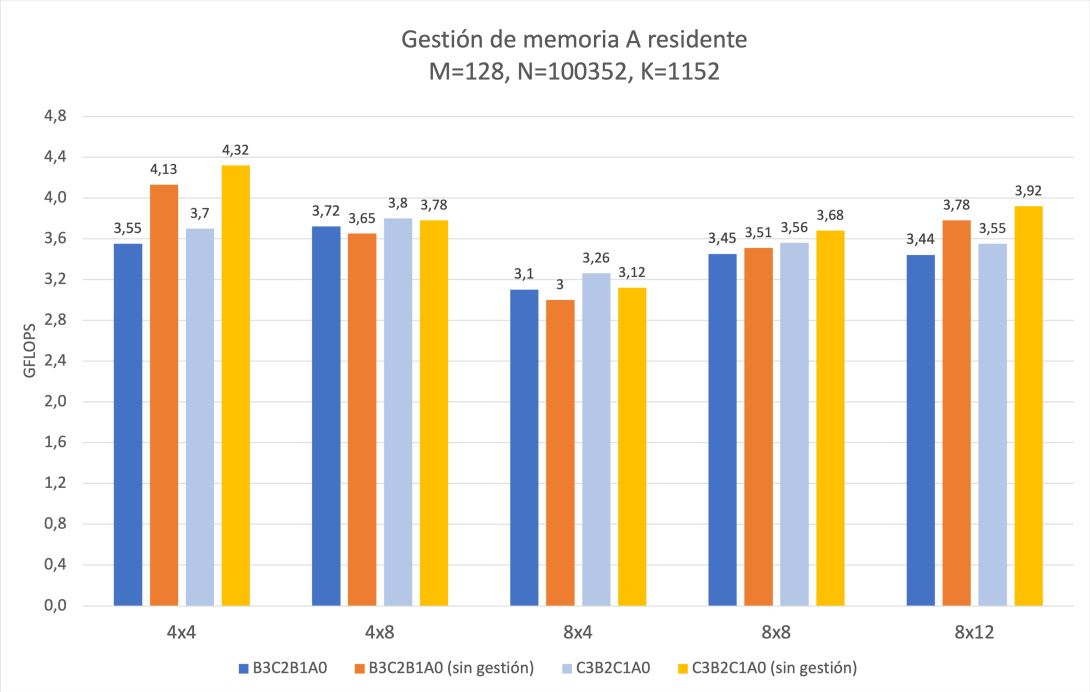

Figura 3.14: Diferencia de rendimiento entre las diferentes variantes de A residente utilizando las técnicas de *prefetch* y *unroll* contra las versiones sin gestión de memoria.

Instrucciones de lectura y escritura

Neon intrinsics permite la lectura y escritura de forma vectorial de los elementos de las matrices a utilizar. Dependiendo del tamaño del *kernel*, empleamos las diversas funciones para lograr que el código sea lo más eficiente posible. Con las funciones $vldNq\_f32$ y $vstNq\_f32$, leemos y escribimos respectivamente $(4 \cdot N)$ elementos consecutivos en me-

moria, siendo N un valor comprendido entre uno y cuatro. De esta manera, para la creación del *micro-kernel* 4×4, es necesaria la utilización de la lectura y escritura de cuatro elementos, por lo que utilizamos las llamadas $vld1q\_f32$ y $vst1q\_f32$ (Código 3.2).

```
A1 = vld1q_f32(&Aref(1,1));
B1 = vld1q_f32(&Br[0]);

// Iteracion del bucle for
for ( j=1; j<nc-2; j+=2 ) {
    C1 = vfmaq_laneq_f32(C1, A1, B1, 0);
...
    vst1q_f32(&Cr[(j-1)*MR], C1);
...
}
```

Código 3.2: Extracto del código del *kernel* 4×4 de A residente.

En cambio, para el *micro-kernel* 8×12 es posible el empleo de funciones que carguen un mayor número de elementos. Para los bloques A y C esto se consigue mediante las llamadas $vld2q\_f32$ y $vst2q\_f32$ para la carga de las columnas de 8 elementos, y de $vld3q\_f32$ la carga de las filas de 12 elementos de B (Código 3.3). El uso de estas funciones hacen que la multiplicación a la hora de la implementación sea de una complejidad algo mayor.

```
A1 = vld2q_f32(&Aref(1,1));
B1 = vld3q_f32(&Br[0]);

// Iteracion del bucle for
for ( j=1; j<nc-1; j++ ) {
    C1.val[0] = vfmaq_laneq_f32(C1.val[0], A1.val[0], B1.val[0], 0);
...
    C1.val[1] = vfmaq_laneq_f32(C1.val[1], A1.val[1], B1.val[0], 0);
...
    vst2q_f32(&Cr[(j-1)*MR], C1);
...
}
```

Código 3.3: Extracto del código del *kernel* 8×12 de A residente.

Como podemos ver en ambos Códigos 3.2 y 3.3, otra de las diferencias que se contempla aparece en la función de multiplicación-suma fusionada. Esta operación ($fmaq\_laneq\_f32$) recibe como argumentos registros que contienen 4 elementos. Por ello a pesar de cargar en los registros $A1$ y $C1$ de 8 datos, debemos volver a realizar la operación para lograr que los últimos 4 valores almacenados en estos registros sean multiplicados (línea 8 del Código 3.3).

3.4.3. Dimensión de los bloques en memoria caché

De la misma forma en que hemos creado diferentes tamaños de *micro-kernels* para obtener la versión con el mejor resultado, se realiza un barrido sobre los valores de m_c, k_c y n_c en el procesador ARM escogido para obtener las dimensiones que ofrecen el mayor rendimiento posible, dando como resultado el siguiente: $m_c = 1792, k_c = 256, n_c = 168$. Cabe destacar que estos valores varían según el tamaño de la memoria caché que integra el dispositivo, de modo que si utilizamos el programa en otro computador, deberemos repetir el barrido.

A continuación, las Figuras 3.15 y 3.16 muestran los resultados del barrido para la obtención del valor óptimo de n_c, sobre una matriz de entrada de dimensiones $m = 2048, n = 6272, k = 512$.

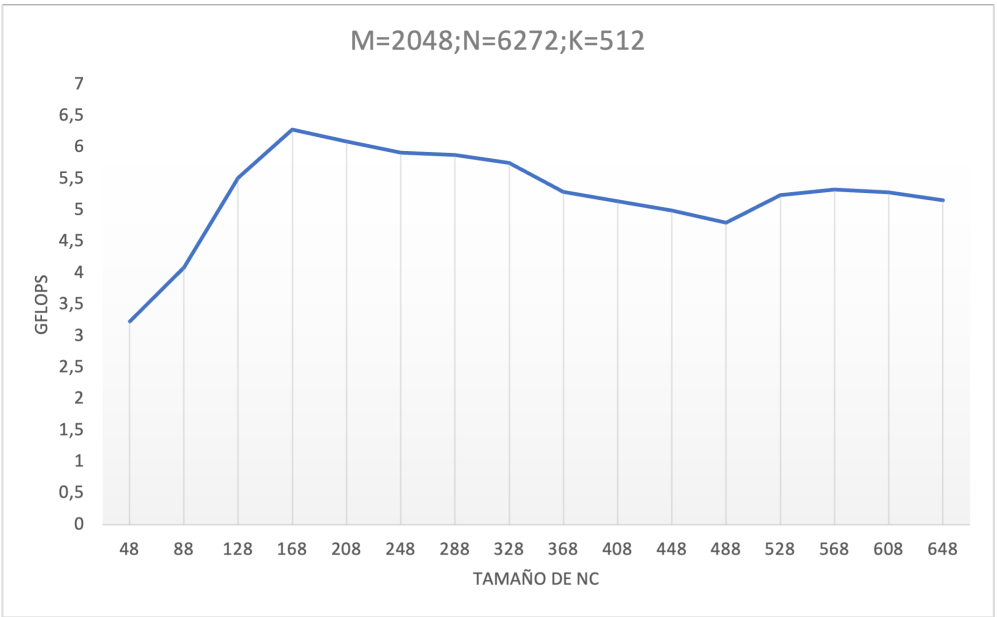

Figura 3.15: Barrido sobre el valor n_c sobre una matriz de entrada de $m = 2048, n = 6272, k = 512$.

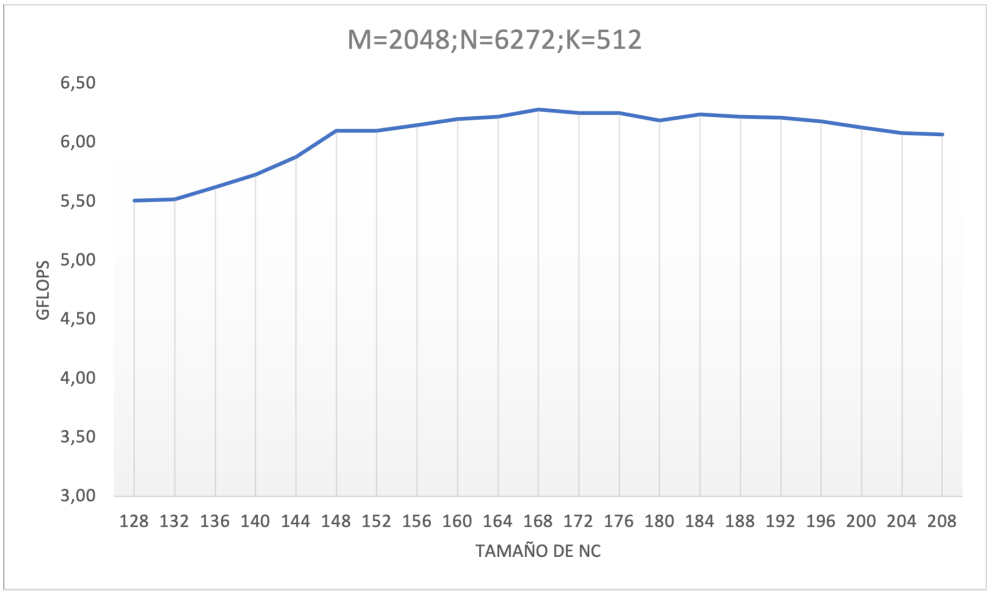

Figura 3.16: Acercamiento sobre el valor óptimo de *nc* en el barrido.

La Figura 3.15 representa uno de los resultados de una serie de múltiples barridos para la obtención de los valores óptimos de m_c, k_c y n_c sobre varios tamaños de matrices. Como podemos ver, el tamaño de n_c es el que mejor resultados obtiene, llegando a alcanzar los 6,28 GFLOPS. Si realizamos un acercamiento de la gráfica sobre el valor de $n_c = 168$, vemos cómo la función llega al pico máximo en este número para luego decaer (Figura 3.16). Adicionalmente, al utilizar matrices de entrada con dimensiones de m menores a $m_c = 1792$, se obtiene como resultado que el valor óptimo es $m_c = m$. Es por ello que en matrices de entrada, como las posteriormente utilizadas en el apartado de evaluación, con tamaños ($m = 128, n = 100352, k = 1152$) y ($n = 512, n = 4608, k = 6272$), el valor de m_c es igual al de m.

3.5 Evaluación

Para la realización de esta sección, como se expone en el apartado 3.1.1, usamos como valores de entrada los tamaños de las matrices que aparecen al aplicar la transformada im2col a las capas de la red convolucional ResNet50 v1.5. La elección de los siguientes tamaños de las matrices no es aleatorio, ya que se escogen con motivo de evaluar el comportamiento del algoritmo base implementado en BLIS y de las variantes expuestas en el presente trabajo para los casos en donde las matrices sean rectangulares y una de las dimensiones sea considerablemente mayor que las otras dos. De esta manera, podemos verificar cuál es el algoritmo que mejor resultado obtiene según las dimensiones concretas de las diferentes matrices de entrada.

La presente sección será subdividido en tres apartados. El primero ilustra la búsqueda del tamaño de *micro-kernel* óptimo para el dispositivo utilizado: NVIDIA Jetson Nano. Cabe destacar que dicha estructura tiene una estrecha correlación con las características del dispositivo en donde se ejecuta, por lo que si se realiza esta misma evaluación en un ordenador diferente, obtendremos unas dimensiones óptimas del *micro-kernel* distintas a las determinadas en este trabajo.

El segundo apartado aborda las diferentes configuraciones de los algoritmos residentes y sus intercambios de bucles, evaluando cada una de ellas, junto con la versión de serie de BLIS. Esta evaluación se realiza con el tamaño de *micro-kernel* óptimo que obtenemos de la primera sección.

Por último, se realiza la paralelización multihilo, mediante OpenMP, del código utilizado en el tercer apartado, empleando de esta forma todos los recursos que nos ofrece el dispositivo.

3.5.1. *Micro-kernel*

En primer lugar se determina el tamaño de *micro-kernel* óptimo para el procesador ARM contenido en el dispositivo Jetson Nano. Con respecto a los tamaños de matriz utilizados, como se ha expuesto previamente, se emplean matrices de entrada de dimensiones $(m \times k)$ y $(k \times n)$, siendo n un valor más grande comparado con los tamaños de m y k. Adicionalmente, se utiliza los diferentes tamaños de *micro-kernel* descritos en el apartado 3.4.2, sobre los cuales se realiza una ejecución con distintos algoritmos con A y C residente. De esta manera, en las siguientes Figuras 3.17, 3.18 y 3.19 mostramos los resultados obtenidos en GFLOPS de las diferentes variantes.

El resultado conseguido en las Figuras 3.17, 3.18 y 3.19, nos permite apreciar que la configuración con la que se obtiene mejor rendimiento es la compuesta por el tamaño de *micro-kernel* (4×4), utilizando C residente con la variante *A3B2A1C0*. Adicionalmente, vemos cómo la elección del algoritmo y su variante produce un claro impacto en el resultado, el cual también se ve condicionado por las dimensiones de las matrices de entrada. Sobre el algoritmo C residente podemos apreciar que, cuanto mayor es la dimensión de m con respecto a la de n y k, mayor importancia tiene su variante, debido a que obtenemos una diferencia de rendimiento mayor a 2,5 GFLOPS, como se puede ver en la Figura 3.19 con el tamaño de *micro-kernel* (4×4). Con respecto al algoritmo con A residente, vemos una estabilidad mayor de las diversas configuraciones de matrices de entrada, siendo menos susceptibles a variaciones de rendimiento. En la gran mayoría de casos también se visualiza que el tamaño óptimo del *micro-kernel* para este algoritmo es 4×8, seguido del anteriormente mencionado (4×4). Aún así, se elige esta última configuración debido a que con este tamaño conseguimos el mejor resultado si tenemos en cuenta todas las configuraciones. Adicionalmente, debemos ser conscientes de que existe un margen

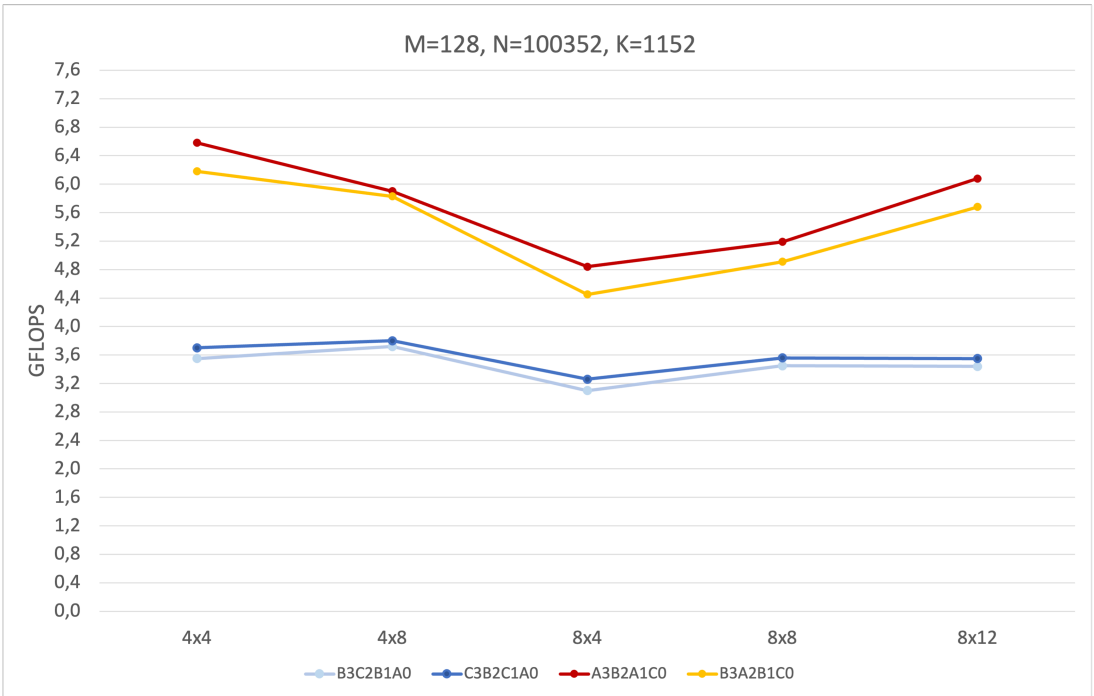

Figura 3.17: Dimensiones de las matrices de entrada $m = 128, n = 100352, k = 1152$.

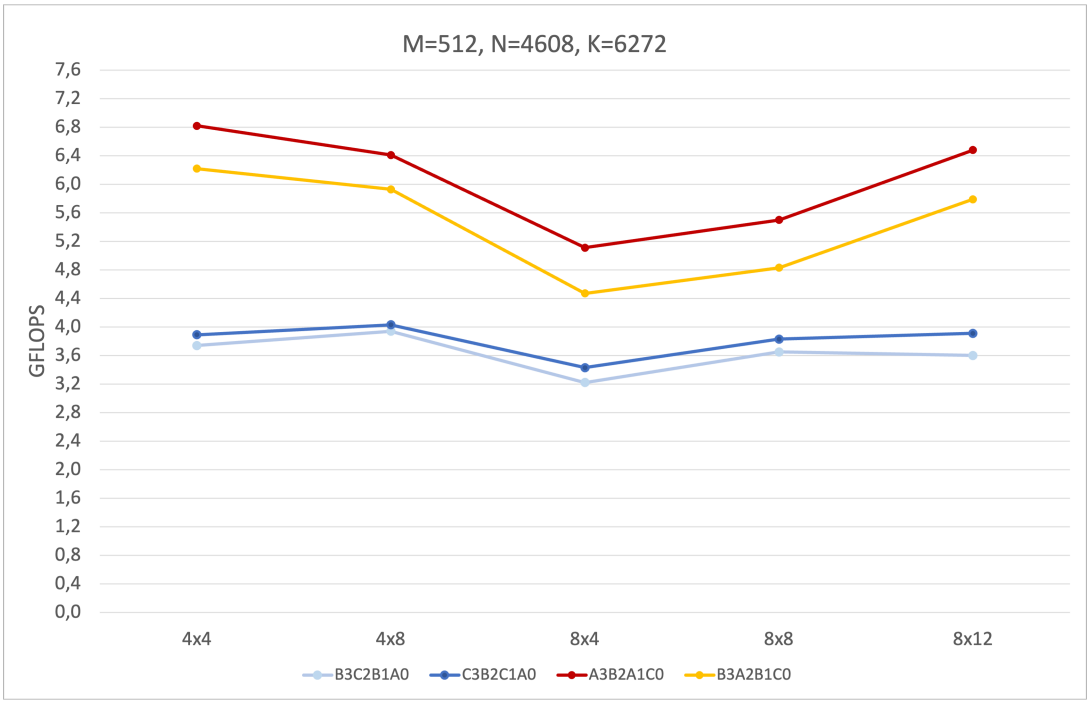

Figura 3.18: Dimensiones de las matrices de entrada $m = 512, n = 4608, k = 6272$

de error y que la diferencia de rendimiento entre ambos tamaños de *micro-kernel* es lo suficientemente pequeña como para comprenderse dentro de dicho margen.

3.5.2. Variantes de los algoritmos residentes

A continuación, una vez obtenido el tamaño óptimo del *micro-kernel* (4×4), realizamos la evaluación para los algoritmos en A y C residentes, empleando diferentes configuraciones de bucles, intercambiándolos entre sí para verificar cuál es la variante que

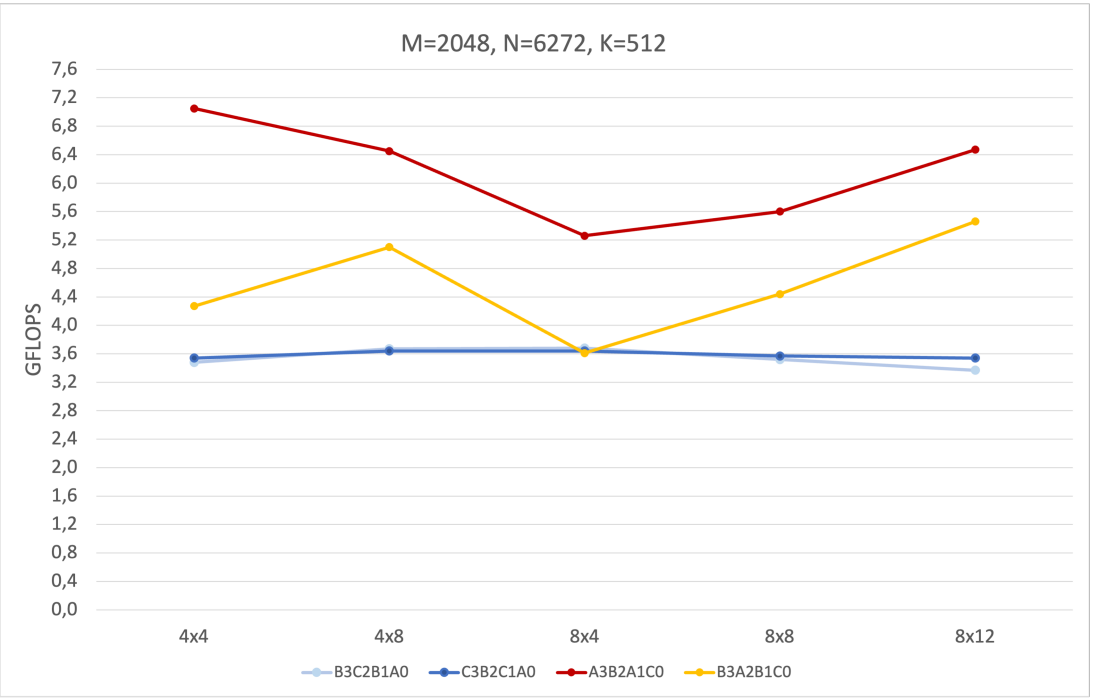

Figura 3.19: Dimensiones de las matrices de entrada $m = 2048, n = 6272, k = 512$

mayor rendimiento nos ofrece. Además, utilizaremos los mismos tres tamaños de matrices empleados en el apartado anterior para realizar la evaluación, en donde cada una de estas estructuras posee una de las dimensiones con un valor mayor en comparación a las otras dos. De esta manera, verificaremos también el comportamiento de las diferentes versiones, así como el de la versión base de BLIS, con respecto a los tamaños de las dimensiones de la matriz de entrada. El resultado de la evaluación se muestra en las Figuras 3.20, 3.21 y 3.22.

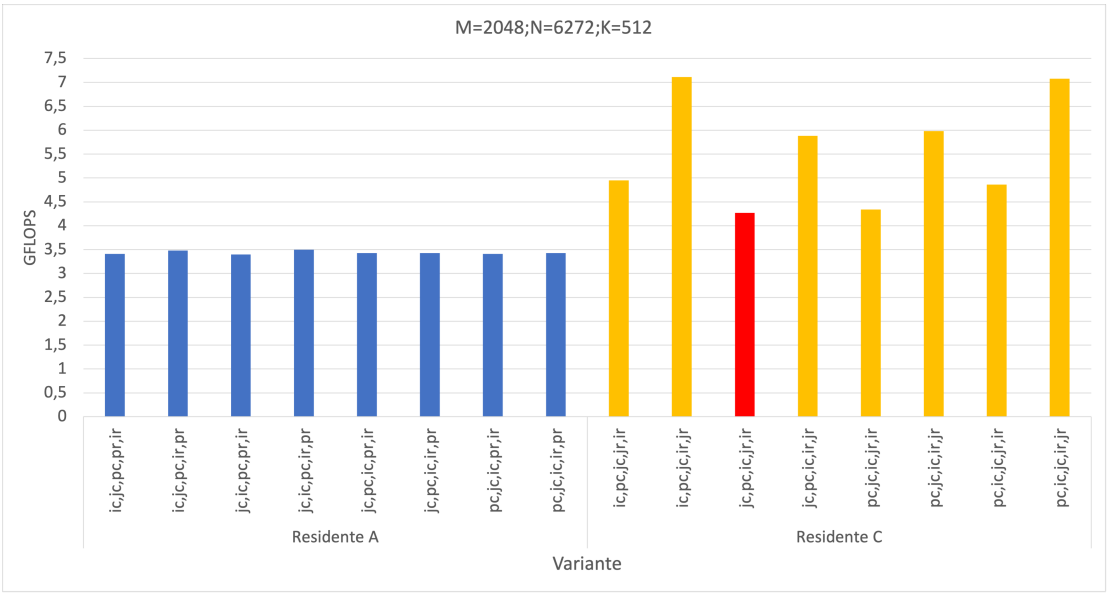

Figura 3.20: Dimensiones de las matrices de entrada $m = 2048, n = 6272, k = 512$.

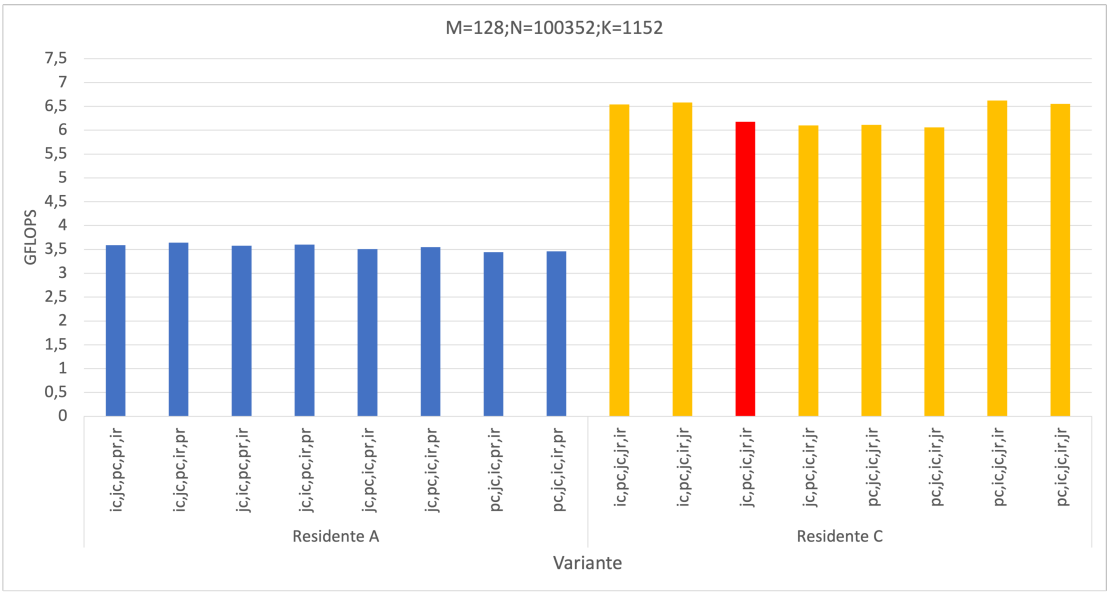

Figura 3.21: Dimensiones de las matrices de entrada $m = 128, n = 100352, k = 1152$

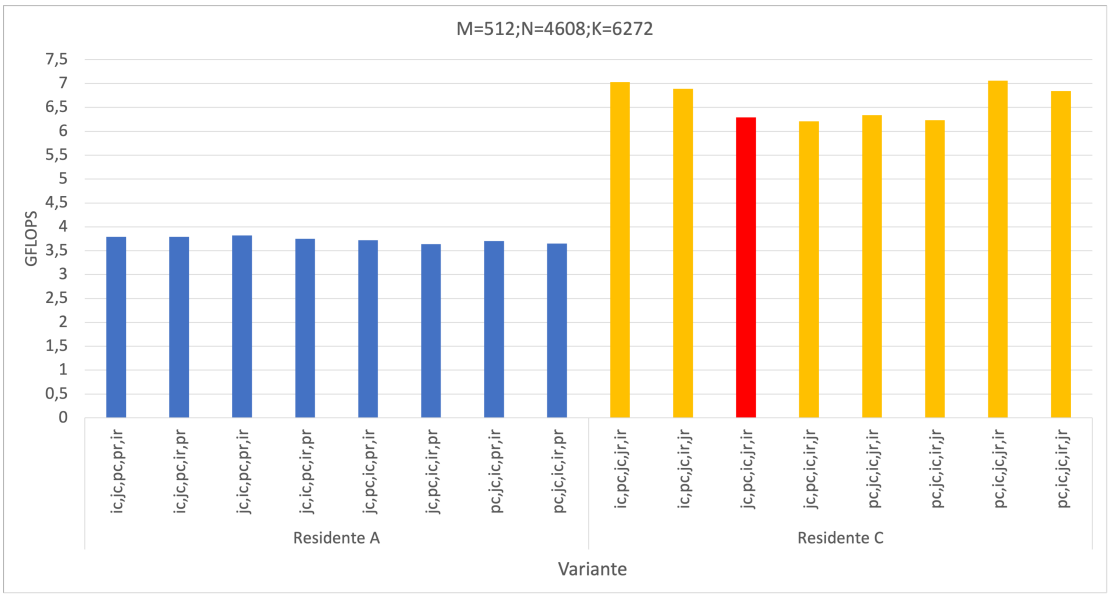

Figura 3.22: Dimensiones de las matrices de entrada $m = 512, n = 6272, k = 4608$

Las Figuras 3.20, 3.21 y 3.22 colocan, sobre el eje horizontal, las diferentes variantes de los algoritmos con A (color azul) y C (color naranja) residentes con sus intercambios de bucles. Estas variantes son identificadas con la anotación vista en el apartado 3.2, destacando en las gráficas de color rojo el algoritmo utilizado por defecto en BLIS (jc, pc, ic, jr, ir con C residente).

Vemos que en todos los diagramas de barras, la versión utilizada en BLIS es superada por al menos otras cuatro variantes, siendo la ordenación de bucles (ic, pc, jc, ir, jr) con el algoritmo con C residente, la que mayor rendimiento obtiene (7,11 GFLOPS en la Figura 3.20) en las evaluaciones realizadas. Además, si observamos la Figura 3.20, donde la dimensión m es mayor que en las otras dos pruebas, podemos comprobar que obtiene el peor rendimiento para el algoritmo con C residente.

Adicionalmente, si estudiamos los resultados obtenidos para ambos algoritmos, existe una diferencia considerable de rendimiento, siendo la variante con C residente la que

mejor desempeño consigue, logrando doblar el rendimiento en GFLOPS con respecto a su contraparte. Este hecho se debe a las variables utilizadas en la multiplicación de ambos algoritmos, como podemos ver en los siguientes códigos 3.4 y 3.5, correspondientes al *micro-kernel* con *C* residente y *A* residente.

```
1    // Producto en C residente
2    C1 = vfmaq_laneq_f32(C1, A1, B1, 0);
3    C2 = vfmaq_laneq_f32(C2, A1, B1, 1);
4    C3 = vfmaq_laneq_f32(C3, A1, B1, 2);
5    C4 = vfmaq_laneq_f32(C4, A1, B1, 3);
```

Código 3.4: Código de la multiplicación del *kernel* 4×4 de la variante C residente.

En el código con *C* residente 3.4, las variables que comienzan con la letra C contienen los elementos de la matriz de salida (*C*). En ellas, por cada línea, se realiza una operación de lectura y posteriormente otra de escritura sobre variables diferentes, por lo que no hay dependencias de lectura después de escritura (RAW), ocultando de esta manera la latencia producida por la instrucción solapando etapas entre las diferentes operaciones.

```
1    // Producto en A residente
2    C1 = vfmaq_laneq_f32(C1, A1, B1, 0);
3    C1 = vfmaq_laneq_f32(C1, A2, B1, 1);
4    C1 = vfmaq_laneq_f32(C1, A3, B1, 2);
5    C1 = vfmaq_laneq_f32(C1, A4, B1, 3);
```

Código 3.5: Código de la multiplicación del *kernel* 4×4 de la variante A residente.

Por su parte, en el código 3.5 con *A* residente, vemos que las mismas variables iniciadas con la letra C, anteriormente comentadas, han pasado de ser cuatro a solamente una (C1). Esto obliga a que, por cada iteración del bucle, se realicen cuatro operaciones de lectura y escritura sobre los elementos de la matriz de salida. Al realizar la instrucción de la multiplicación-suma fusionada obtendremos una latencia inherente al tiempo de ejecución de la propia operación. Si tenemos en cuenta que las siguientes instrucciones necesitan de la misma variable para leer y almacenar el resultado, obtenemos un total de tres dependencias RAW por cada iteración del bucle para el caso del *micro-kernel* 4×4. Como hemos visto en las Figuras 3.20, 3.21 y 3.22, la diferencia de rendimiento entre las variantes con *A* y *C* residente es producida por estas dependencias de lectura y escritura, dando como resultado una mayor obtención de GFLOPS en la variante con *C* residente.

3.5.3. Ejecución en paralelo

En el apartado 3.5.2, las gráficas visualizadas se corresponden con la ejecución del programa en secuencial. Como hemos visto en la sección 2.3, el dispositivo Jetson Nano dispone de un procesador de cuatro núcleos, que pueden aprovecharse mediante la inclusión de directivas de OpenMP en el código del producto de matrices para la paralelización de determinados bucles.

Paralelización con OpenMP

Como hemos visto en la sección 2.4, utilizar directivas de OpenMP implica la creación de una región paralela en la cual se crean, gestionan y sincronizan los procesos, provocando una sobrecarga en el tiempo de ejecución debido a las entradas y salidas de los hilos en esta sección. Es por ello que, al incluir en un bucle anidado estas directivas, es recomendable hacerlo en el bucle más externo. De esta manera, el número de ingresos a

la región paralela será menor, evitando mayores costes debidos a la sincronización de los hilos.

A pesar de ello, dependiendo de las características del problema a tratar, no siempre es posible paralelizar el bucle más externo, como es el caso del presente trabajo. Si visualizamos los primeros tres bucles anidados (*Loop 1*, *Loop 2* y *Loop 3*) de la Figura 3.4, estos realizan las iteraciones con un *stride* considerablemente grande, haciendo que la paralelización en estos bucles no sea eficiente debido a un número insuficiente de iteraciones. Otra razón para no paralelizar estos tres bucles es que cada hilo terminaría utilizando su propio *buffer* A_c y/o B_c, lo cual puede ser beneficioso si la memorias caché L2 y L3 son privadas. Sin embargo en el caso de que sean compartidas, como sucede en el presente trabajo, esto provocaría colisiones entre los hilos, obteniendo como resultado un menor rendimiento. Adicionalmente, en estos bucles se efectúan las llamadas a los empaquetamientos de las matrices, las cuales realizan dentro de su función una operación compuesta por tres bucles anidados.

Analizando en profundidad estas funciones de empaquetamiento, podemos ver como el código 3.6, por sus características, nos permite realizar la paralelización en el bucle más externo.

```
#pragma omp parallel for private(i, jj, nr, k)
for ( j=1; j<=nc; j+=RR ) {
  k = (j-1)*mc;
  nr = min( nc-j+1, RR );
  for ( i=1; i<=mc; i++ ) {
    for ( jj=1; jj<=nr; jj++ ) {
      Mc[k] = Mref(i,(j-1)+jj);
      k++;
    }
    k += (RR-nr);
  }
}
```

Código 3.6: Código de la función de empaquetamiento paralelizado mediante OpenMP.

Aún así, debemos tener en cuenta las condiciones de carrera generadas por la dependencia de datos, producto de escritura y lectura de varios hilos en memoria compartida. Para evitar esta condición, se declara en la cláusula de la directiva aquellas variables que presentan estos problemas como privadas, implicando que cada hilo copie el valor de la variable compartida en una variable privada del hilo. De esta forma delimitaremos que cada proceso trabaje en una zona determinada de la matriz evitando las condiciones de carrera.

Volviendo a la estructura principal, y teniendo en cuenta lo anteriormente descrito, se opta por paralelizar sobre los bucles *Loop 4* y *Loop 5* correspondientes a la Figura 3.4. Al emplear en la evaluación varias configuraciones de bucles, debemos asegurarnos de que en cada una de estas se cumplen todas las condiciones para poder realizar la paralelización sobre el bucle más externo (*Loop 4*). Esta condición se incumple para las variantes con A residente cuando la configuración de bucle finaliza con las variables pr e ir. Esto se debe a que la variable pr no es utilizada en la lectura o escritura de la matriz de salida, por lo que si paralelizamos este bucle, las iteraciones no serán independientes, produciendo de esta manera un resultado final erróneo. Por este motivo, se deja el bucle más externo fuera de la región paralela y se opta por paralelizar el siguiente bucle (*Loop 5*).

A continuación, ofrecemos los resultados correspondientes a la ejecución en paralelo del programa, utilizando cuatro hilos, en las gráficas de las Figuras 3.25, 3.23 y 3.24.

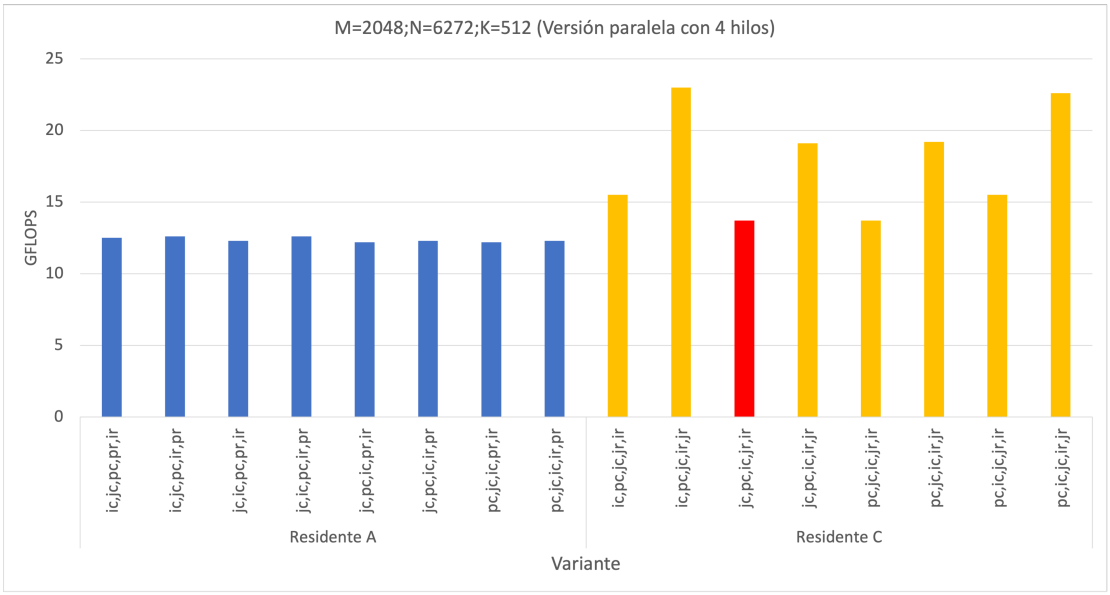

Figura 3.23: Dimensiones de las matrices de entrada $m = 2048, n = 6272, k = 512$

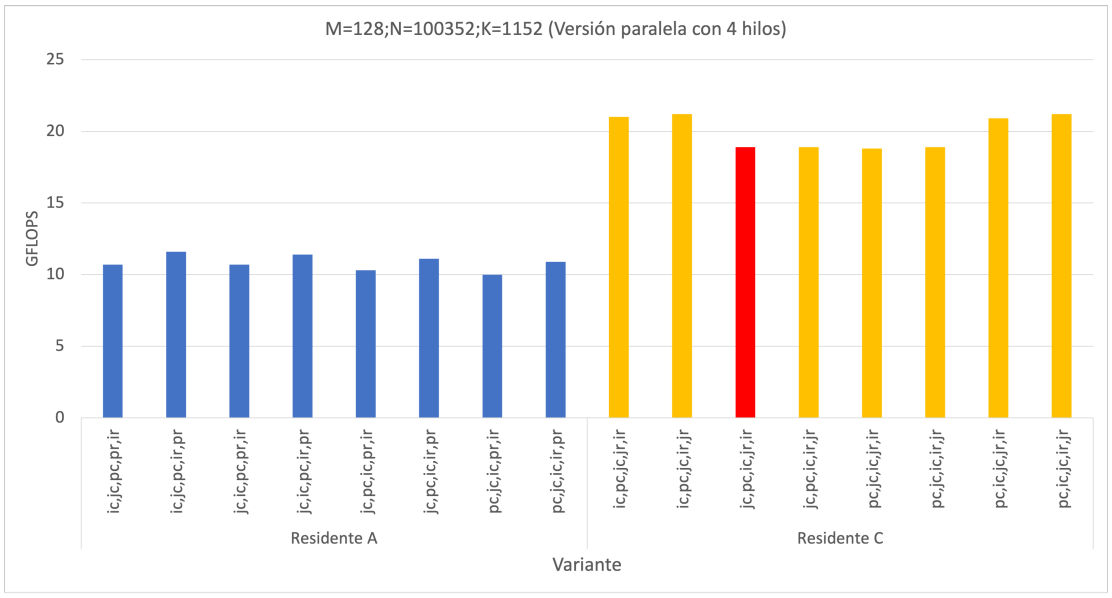

Figura 3.24: Dimensiones de las matrices de entrada $m = 128, n = 100352, k = 1152$

En las mismas, podemos comprobar que se mantiene la tendencia observada en la ejecución secuencial, en donde las configuraciones con C residente obtenían mejores resultados con respecto a los de A residente. Adicionalmente, vemos un aumento en las diferencias de rendimiento registradas en la sección anterior, siendo aún más notoria con respecto a la configuración de BLIS cuando la dimensión m presenta un valor elevado, llegando a obtener una diferencia de rendimiento de aproximadamente 10 GFLOPS.

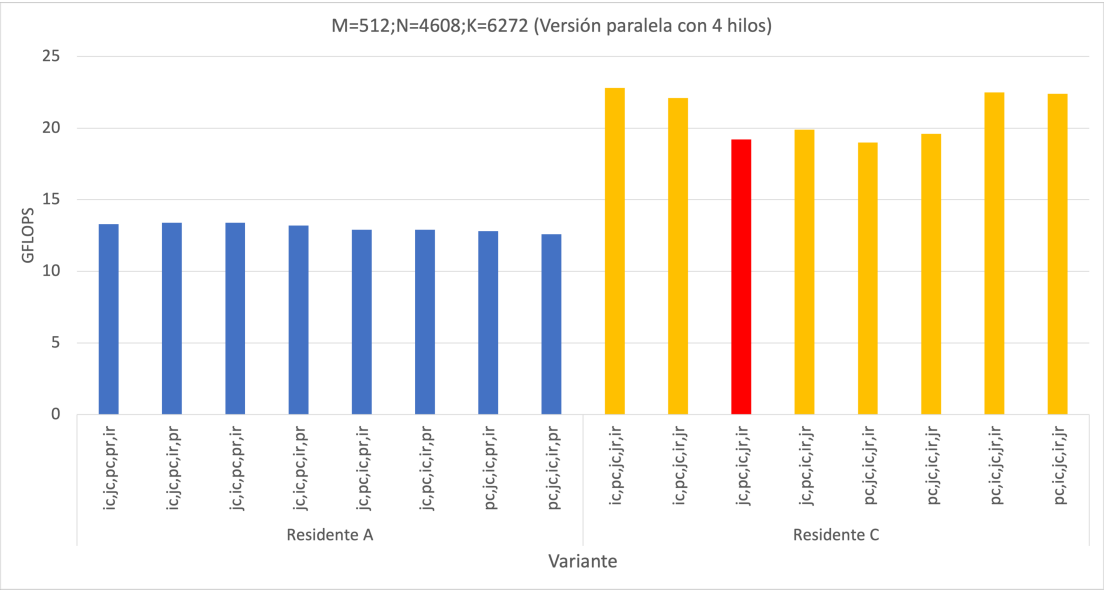

Figura 3.25: Dimensiones de las matrices de entrada $m = 512, n = 4608, k = 6272$

CAPÍTULO 4
Conclusiones

En referencia a la optimización del producto de matrices de BLIS realizada en el presente trabajo, se pueden obtener las siguientes conclusiones:

- Se han estudiado las características principales de las redes neuronales, haciendo énfasis en la convolución, la cual puede implementarse como un producto de matrices y es una operación relevante para el entrenamiento e inferencia de las redes neuronales debido a su elevado consumo de recursos computacionales.
- Se ha realizado un análisis del procesador con arquitectura ARM y de las memorias caché del dispositivo NVIDIA Jetson Nano para obtener una optimización *ad hoc* de la operación de convolución sobre este componente.
- Se ha estudiado la formulación de alto nivel de la multiplicación general de matrices descrita en BLIS. Además, se han realizado una serie de optimizaciones como la paralelización de los bucles mediante OpenMP y la utilización de algoritmos residentes sobre esta operación, logrando de esta manera un aumento significativo del rendimiento.
- Se han desarrollado cinco *micro-kernels* para los diferentes tamaños de bloques de matriz, utilizando Neon Intrinsics y técnicas de gestión de memoria como *unroll* y *prefetch* para su optimización.

4.1 Relación del trabajo desarrollado con los estudios cursados

En el presente trabajo, se han necesitado conceptos de las siguientes asignaturas impartidas en el Máster Universitario en Computación en la Nube y de Altas Prestaciones:

- Tecnología de la Programación Paralela (TPP) y Conceptos y Métodos de la Computación Paralela (CMCP): Asignaturas en donde se utilizan modelos de programación paralela para analizar y diseñar soluciones que incluyen procesos multihilo y de utilización de memoria compartida.
- Herramientas de computación de altas prestaciones (HCAP): En esta asignatura se detallan y emplean herramientas utilizadas para la realización eficiente del código, además de analizar y aplicar librerías de algoritmos básicos del álgebra lineal numérica.

Bibliografía

[1] Xiaofei Wang, Yiwen Han, Victor C. M. Leung y col. «Edge AI, Convergence of Edge Computing and Artificial Intelligence.» En: (2020). DOI: 10.1007/978-981-15-6186-3.

[2] Field G. Van Zee y Robert A. van de Geijn. «BLIS: A Framework for Rapidly Instantiating BLAS Functionality». En: *ACM Trans. Math. Softw.* 41.3 (jun. de 2015). ISSN: 0098-3500. DOI: 10.1145/2764454. URL: https://doi.org/10.1145/2764454.

[3] D. Patterson. «Reduced Instruction Set Computers Then and Now». En: *Computer* 50.12 (dic. de 2017), págs. 10-12. ISSN: 1558-0814. DOI: 10.1109/MC.2017.4451206.

[4] ARM Holdings. *CPU Architecture*. URL: https://developer.arm.com/architectures/cpu-architecture (visitado 2021-05-15).

[5] ARM Holdings. *SIMD ISAs*. URL: https://developer.arm.com/architectures/instruction-sets/simd-isas (visitado 2021-05-15).

[6] ARM Holdings. *Neon*. URL: https://developer.arm.com/architectures/instruction-sets/simd-isas/neon (visitado 2021-05-15).

[7] ARM Holdings. *Neon Programmer's Guide for ARMv8-A : Coding for Neon*. URL: https://developer.arm.com/documentation/102159/0400/Matrix-multiplication (visitado 2021-05-18).

[8] ARM Holdings. *About the L1 memory system*. URL: https://developer.arm.com/documentation/ddi0488/h/level-1-memory-system/about-the-l1-memory-system?lang=en (visitado 2021-05-17).

[9] ARM Holdings. *About the L2 memory system*. URL: https://developer.arm.com/documentation/ddi0488/h/level-2-memory-system/about-the-l2-memory-system?lang=en (visitado 2021-05-17).

[10] Pranav Tendulkar. «Mapping and Scheduling on Multi-core Processors using SMT Solvers». Theses. Universite de Grenoble I - Joseph Fourier, oct. de 2014. URL: https://tel.archives-ouvertes.fr/tel-01087271.

[11] Princeton University. Stanford Vision Lab Stanford University. *ImageNet*. URL: https://www.image-net.org/index.php (visitado 2021-05-20).

[12] Alex Krizhevsky, Ilya Sutskever y Geoffrey E. Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». En: *Advances in Neural Information Processing Systems 25*. Ed. por F. Pereira, C. J. C. Burges, L. Bottou y col. Curran Associates, Inc., 2012, págs. 1097-1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren y col. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[14] Y. LeCun, L. Bottou, Y. Bengio y col. «Gradient-based learning applied to document recognition». En: *Proceedings of the IEEE* 86.11 (1998), págs. 2278-2324. DOI: 10.1109/5.726791.

[15] Kumar Chellapilla, Sidd Puri y Patrice Simard. «High Performance Convolutional Neural Networks for Document Processing». En: *Tenth International Workshop on Frontiers in Handwriting Recognition*. Ed. por Guy Lorette. http://www.suvisoft.com. Université de Rennes 1. La Baule (France): Suvisoft, oct. de 2006. URL: https://hal.inria.fr/inria-00112631.

[16] Sergio Barrachina, Manuel F. Dolz, Pablo San Juan y col. «Efficient and Portable Convolution Operators for Deep Neural Network Training on Clusters of Multicore Processors». En: (s.f.).

[17] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling y col. «A Set of Level 3 Basic Linear Algebra Subprograms». En: *ACM Trans. Math. Softw.* 16.1 (mar. de 1990), págs. 1-17. ISSN: 0098-3500. DOI: 10.1145/77626.79170. URL: https://doi.org/10.1145/77626.79170.

[18] Tze Meng Low, Francisco D. Igual, Tyler M. Smith y col. «Analytical Modeling Is Enough for High-Performance BLIS». En: *ACM Trans. Math. Softw.* 43.2 (ago. de 2016). ISSN: 0098-3500. DOI: 10.1145/2925987. URL: https://doi.org/10.1145/2925987.

[19] Pablo San Juan, Adrián Castelló, Manuel F. Dolz y col. *High Performance and Portable Convolution Operators for ARM-based Multicore Processors*. 2020. arXiv: 2005.06410 [cs.PF].

[20] Tyler M. Smith y Robert A. van de Geijn. *The MOMMS Family of Matrix Multiplication Algorithms*. 2019. arXiv: 1904.05717 [cs.MS].