UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Generación automática de núcleos computacionales para redes neuronales

TRABAJO FIN DE MÁSTER

Máster en Computación en la Nube y de Altas Prestaciones

Autor: Guillermo Alaejos López
Tutor: Adrián Castelló Gimeno y Pedro Alonso Jordá

Curso 2021-2022

Resum

L'auge en l'aplicació de *Xarxes Neuronals Profundes* (XNP) en una gran varietat de camps científics ha propiciat el seu ús no sols en servidors de còmput intensiu sinó també en dispositius de baix consum. Molts dels càlculs que es realitzen en les RNPs, tant en entrenament com en inferència, es descomponen en nuclis d'àlgebra lineal i s'extrauen de biblioteques especialitzades com Intel MKL o BLIS. No obstant això, la quantitat de memòria requerida per aquestes biblioteques pot excedir la capacitat màxima d'aquests xicotets dispositius. A més, la gran varietat de maquinari de baix consum fa pràcticament impossible comptar amb nuclis de còmput optimitzats per a cada model. Una opció per a reduir el cost de generació i manteniment d'aquestes biblioteques és la utilització de generadors de codi automàtics com a Apache TVM. Aquestes eines permeten desenvolupar un sol codi comú per a tots els dispositius i, posteriorment, generar el codi assemblador per a cadascun d'ells. Amb TVM solament s'ha de generar els nuclis necessaris per a un model de RNP concret, evitant utilitzar memòria del dispositiu amb funcionalitat que no s'utilitzarà. En aquest projecte es pretén generar nuclis computacionals optimitzats per a diferents arquitectures de manera automàtica utilitzant Apatxe TVM amb l'objectiu de reproduir les necessitats de les RNPs.

Paraules clau: Apache TVM; Xarxes Neuronals Profundes (XNP);RISC-V; ARM; Intel; Intrinsics; Àlgebra Lineal Numèrica; BLIS; BLAS

Resumen

El auge en la aplicación de *Redes Neuronales Profundas* (RNP) en una gran variedad de campos científicos ha propiciado su uso no solo en servidores de cómputo intensivo sino también en dispositivos de bajo consumo. Muchos de los cálculos que se realizan en las RNPs, tanto en entrenamiento como en inferencia, se descomponen en núcleos de álgebra lineal y se extraen de bibliotecas especializadas como Intel MKL o BLIS. Sin embargo, la cantidad de memoria requerida por estas bibliotecas puede exceder la capacidad máxima de estos pequeños dispositivos. Además, la gran variedad de hardware de bajo consumo hace prácticamente imposible contar con núcleos de cómputo optimizados para cada modelo. Una opción para reducir el coste de generación y mantenimiento de estas bibliotecas es la utilización de generadores de código automáticos como Apache TVM. Estas herramientas permiten desarrollar un solo código común para todos los dispositivos y, posteriormente, generar el código ensamblador para cada uno de ellos. Con TVM solamente se debe generar los núcleos necesarios para un modelo de RNP concreto, evitando utilizar memoria del dispositivo con funcionalidad que no se va a utilizar. En este proyecto se pretende generar núcleos computacionales optimizados para distintas arquitecturas de forma automática utilizando Apache TVM con el objetivo de reproducir las necesidades de las RNPs.

Palabras clave: Apache TVM; Redes neuronales profundas (RNP); RISC-V; ARM; Intel; Intrinsics; Álgebra Lineal Numérica; BLIS; BLAS

Abstract

The boom in the application of Deep Neural Networks (DNNs) in a wide variety of scientific fields has led to their use not only in compute-intensive servers but also in low-power devices. Many of the computations performed in RNPs, both in training and inference, are decomposed into linear algebra kernels and extracted from specialised libraries such as Intel MKL or BLIS. However, the amount of memory required by these

libraries can exceed the maximum capacity of these small devices. In addition, the wide variety of low-power hardware makes it virtually impossible to have optimised compute cores for each model. One option to reduce the cost of generating and maintaining these libraries is the use of automatic code generators such as Apache TVM. These tools allow you to generate a single common code for all devices and then generate the assembly code for each device. With TVM, only the cores needed for a specific RNP model must be generated, avoiding the use of device memory with functionality that is not going to be used. This project aims to generate optimised computational kernels for different architectures automatically using Apache TVM with the objective of reproducing the needs of RNP.

Key words: Apache TVM; Deep Neural Networks (DNN); RISC-V; ARM; Intel; Intrinsics; Numerical Linear Algebra; BLIS; BLAS

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

1.1 Motivación

La multiplicación de matrices (*General Matrix Multiplication* en inglés, comúnmente abreviada como GEMM) es una operación vital en numerosas bibliotecas especializadas en la computación científica por sus considerables usos y aplicaciones en diversos ámbitos de la ingeniería. Entre las implementaciones realizadas, se encuentran algunas como Intel MKL [21], OpenBLAS [26], ARMPL (ARM *Performance Library*) [2] o BLIS [24].

Sin embargo, estas implementaciones cuentan con una serie de problemas:

- Estas implementaciones suelen ser genéricas y estar diseñadas para funcionar independientemente del tamaño del problema.
- Suele haber parámetros internos fijados durante la instalación que, de ser necesario modificarlos, requiere una recompilación completa del programa.
- Algunas de estas implementaciones, como Intel MKL o NVIDIA cuBLAS están limitadas para usarse con un *hardware* específico.

El hecho de ser implementaciones independientes de la máquina donde se va a ejecutar deriva en un rendimiento subóptimo, e implementar manualmente diferentes variaciones hasta encontrar aquella que es óptima requiere un esfuerzo demasiado elevado. Además, el algoritmo óptimo es dependiente de la arquitectura concreta de cada procesador, incluida la jerarquía de memorias caché, por lo que un algoritmo óptimo para una máquina no es portable a otra máquina diferente. Debido al enorme número de arquitecturas diferentes, no es viable encontrar manualmente el algoritmo óptimo para cada una de estas arquitecturas.

Es en este contexto donde la autogeneración de código puede resultar una herramienta útil. La idea consistiría en generar automáticamente diferentes variaciones de un mismo algoritmo y probarlas hasta encontrar la versión óptima. Al realizarse de manera "automática", es posible probar múltiples posibilidades que, de implementarlas manualmente, llevaría un esfuerzo inabarcable. Esta necesidad ya ha sido identificada en determinados ámbitos, entre lo que destaca el aprendizaje automático o *Machine Learning*. Es en este contexto en el que tiene lugar el desarrollo de herramientas de generación automática de código como MLIR [22] o TVM [14].

Además, en este trabajo analizamos el funcionamiento de la biblioteca BLIS [24] en lo que respecta a la implementación de la GEMM. BLIS es una biblioteca que propone códigos independientes de la plataforma donde se tengan que ejecutar pero, a su vez,

optimizados para un conjunto de plataformas diferentes. Por ejemplo, para ARM propone un micro-kernel específico de tamaño 8×12.

Otro de los aspectos que motivan este trabajo es el de la utilización del procesador RISC-V. La irrupción de este procesador en el panorama actual es notable debido a varios aspectos, entre los que destacan su carácter abierto (no propietario) y su bajo consumo. Si bien se trata de un dispositivo en estado de experimentación o prototipado (no existen implementaciones reales todavía), grandes fabricantes de chips como Intel o NVIDIA están mostrando interés en el mismo.

1.2 Objetivos

El objetivo principal de este trabajo se puede descomponer en los siguientes objetivos concretos:

1. Analizar y comprender el funcionamiento de BLIS.
2. Generación de micro-kernels manualmente para ARM y RISC-V.
3. Entender el funcionamiento básico de un programa de autogeneración de código en TVM.
4. Generación de micro-kernels automáticamente con TVM y llevarlo a casos concretos sobre *hardware* de ARM e Intel.
5. Generación de la GEMM completa con TVM.
6. Evaluación y comparación de los resultados obtenidos con bibliotecas ya implementadas, como BLIS e Intel MKL.

1.3 Estructura de la memoria

La estructura que presenta esta memoria es la siguiente. El Capítulo 2 se se tratará todo lo referente al entorno de trabajo, tanto hardware como software. Todo el material incluido en este capítulo es necesario entenderlo para comprender el resto del trabajo. En el Capítulo 3 se describe el software básico desarrollado en este trabajo, conocido como microkernels. Se muestra su creación manual para los procesadores de arquitectura ARM, Intel y RISC-V. En este mismo capítulo se trata la creación automática de los mismos, en este caso, para procesadores ARM e Intel utilizando la herramienta TVM. Por último, en el Capítulo 4 se mostrarán los resultados obtenidos y la comparación de estos resultados entre las bibliotecas ya existentes y TVM. Terminamos el trabajo con un capítulo de conclusiones.

CAPÍTULO 2
Hardware y Software

En este capítulo se enumeran y describen los entornos hardware y herramientas software utilizadas en este trabajo. Comenzamos con la descripción de las arquitecturas hardware básicas de las que se han podido utilizar uno o más dispositivos (Apartado 2.1). Seguidamente, se describe el software utilizado, que se compone de bibliotecas de álgebra lineal (Apartado 2.2), con especial mención a BLIS (Apartado 2.3), herramientas intermedias entre el lenguaje de alto nivel C y el lenguaje ensamblador como las instrucciones *intrínsecas* (Apartado 2.4) y, para terminar, de la herramienta TVM (Apartado 2.5).

2.1 Hardware

En este apartado se describen las arquitecturas utilizadas en este trabajo, junto a sus principales características necesarias, como aquellas relacionadas con los registros vectoriales. Un registro vectorial es un registro especial con capacidad para almacenar varios elementos simples (tipos primitivos como `int` o `float`) y que permite el procesado de estos elementos en paralelo a través del uso de la técnica SIMD [12] (*Single Instruction Multiple Data*). Esta técnica consiste en procesar varios elementos en una misma instrucción, de manera que el rendimiento mejora considerablemente.

Intel y la arquitectura x86

La arquitectura x86 de Intel es una de las más conocidas. Esta arquitectura nombra a sus procesadores con una serie de "nombres en clave" según la microarquitectura que posean.

Por ejemplo, una de las arquitecturas utilizadas en el trabajo se llama Cascade Lake (2019). Esta máquina, entre otras cosas, da soporte a las instrucciones vectoriales de tipo SSE [20], AVX [19] y AVX-512 [3]. En este trabajo solo se han utilizado las AVX-512, que son las más modernas y que más potencia (longitud) ofrecen. Otra de las microarquitecturas utilizadas es la denominada Skylake, anterior a Cascade Lake. Esta microarquitectura, lanzada en 2015, cuenta con una serie de mejoras respecto a sus predecesores, orientada especialmente hacia la eficiencia [25].

ARM

ARM [1] es una arquitectura de tipo RISC (Reduced Instructions Set Computer) de 32 y 64 bits desarrollada por ARM Holding. Al ser una arquitectura basada en un conjunto de instrucciones reducido, estos procesadores necesitan menos energía para funcionar, y por tanto, emiten menos calor, lo que los hace ideales para dispositivos móviles y de bajo consumo. Numerosos fabricantes de chips utilizan

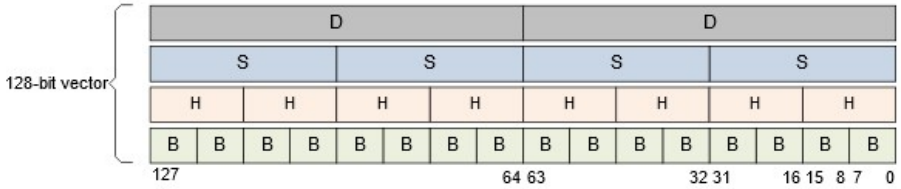

Figura 2.1: Tamaño de los registros vectoriales de ARM

el diseño de ARM para sus CPUs, como Nintendo, NVIDIA, Apple o Sony, entre otros. La arquitectura ARM presenta una serie de registros vectoriales con una longitud de 128 bits que permiten almacenar varios elementos. En la Figura 2.1 se pueden ver los elementos que se pueden almacenar en un único registro vectorial. Un único registro vectorial tiene capacidad para dos valores numéricos de doble precisión, o cuatro valores numéricos de precisión simple. Para este trabajo se han utilizado valores de simple precisión, por lo que cada registro tiene capacidad para cuatro valores. La arquitectura ARM actual posee 32 registros vectoriales.

RISC-V

RISC-V [17] es un procesador con un conjunto de instrucciones libre y abierto. Esto permite que cualquier persona pueda desarrollar, fabricar y vender chips de basados en arquitectura RISC-V. El proyecto comenzó en 2010 en la Universidad de California y actualmente se encuentra soportado por la Fundación RISC-V, que cuenta con el apoyo de algunas de las empresas e instituciones tecnológicas más importantes del mundo, entre las que se encuentran, por ejemplo, Google, AMD, Huawei, IBM. RISC-V tiene un total de 32 registros vectoriales que, junto a un amplio conjunto de instrucciones vectoriales, facilitan la programación y permiten hacer uso de las unidades SIMD [11]. El tamaño de los registros vectoriales es variable, lo que significa que la especificación deja abierto este parámetro, siendo en una implementación hardware real donde se seleccionaría un tamaño fijo concreto.

2.2 Bibliotecas de álgebra lineal

La computación científica siempre ha estado relacionada con las operaciones matemáticas, y más concretamente del área del álgebra lineal, que se caracterizan por ser operaciones con un alto coste de cómputo. Dentro de estas bibliotecas encontramos la multiplicación de matrices o GEMM (del inglés *GEneral Matrix Multiplication*) que es de las operaciones básicas más costosas y sobre la que se soportan otras operaciones incluidas en estas bibliotecas. Por tanto, es primordial que estas rutinas se implementen de la manera más eficiente posible para disminuir los costes de cálculo.

La biblioteca BLAS (*Basic Linear Algebra Subprograms*) está constituida por rutinas que proporcionan bloques de construcción estándar (*kernels*) para realizar operaciones básicas con vectores y matrices. BLAS es, en realidad, una especificación, la especificación más ampliamente aceptada para reunir al conjunto básico de operaciones de álgebra lineal numérica que los fabricantes, entre otros, implementan de manera optimizada para sus procesadores. Ejemplos de implementación de esta especificación y que se han utilizado en este trabajo son las siguientes:

OpenBLAS

OpenBLAS [26] es una biblioteca de código abierto que sigue el paradigma de BLAS [4]. Su implementación está basada en la biblioteca GotoBLAS [16] y, después

de que este proyecto se quedara sin mantenimiento, se decidió crear OpenBLAS. Está financiada y promovida por la Universidad de Texas. Actualmente tiene soporte para BLAS1 (operaciones vector-vector), BLAS2 (operaciones matriz-vector) y BLAS3 (operaciones matriz-matriz). Como BLAS es una interfaz estándar, los parámetros que reciben las funciones de OpenBLAS son idénticos a los de cualquier otra implementación basada en BLAS.

ARMPL

ARMPL [2] es una biblioteca de *software* propietario desarrollada por ARM. Esta biblioteca da soporte, tanto para C como para Fortran, e incluye operaciones de BLAS, LAPACK (*Linear Algebra PACKage*) y operaciones con matrices dispersas, entre otras funciones. Al igual que con OpenBLAS, sigue el paradigma de BLAS y también da soporte para BLAS1, BLAS2 y BLAS3. Además, soporta el paralelismo mediante el uso de OpenMP para la gran mayoría de sus funciones con objeto de maximizar el rendimiento.

Intel MKL

Intel MKL (*Math Kernel Library*) [21] es una biblioteca de computación científica *software* propietario implementada por Intel, con soporte para BLAS y LAPACK [9], entre otras funciones, para matrices densas, y con soporte parcial a operaciones BLAS con matrices dispersas. Está optimizada para funcionar sobre arquitecturas Intel y, aunque es posible utilizarla en otras arquitecturas, Intel MKL obtendrá un peor rendimiento. Como es lógico Intel MKL suele tener un mayor rendimiento en plataformas Intel en comparación con otras bibliotecas. El inconveniente es que al ser una implementación propietaria de código cerrado, no es posible replicar su funcionamiento mediante la autogeneración de código ya que se desconoce su implementación interna.

La última biblioteca de álgebra lineal que queda por mencionar tiene una importancia especial en este trabajo, por lo que pasamos a describirla en el apartado siguiente.

2.3 La biblioteca BLIS

La biblioteca BLIS [24] es, en realidad, un framework de software portable para crear instancias de bibliotecas de álgebra lineal densa de alto rendimiento similar a BLAS. Este entorno de trabajo ha sido desarrollado por la Universidad de Texas, quien se encarga de su mantenimiento y promoción. Las rutinas desarrolladas en la biblioteca resuelven operaciones de álgebra lineal de manera independiente de la plataforma donde se tengan que ejecutar. Solo una parte de la biblioteca, los *micro-kernels*, es dependiente de cada arquitectura en particular en la que se quiera utilizar. Toda la biblioteca gira entorno a una implementación eficiente del producto matricial (GEMM). Para la especificación de este algoritmo, en lo sucesivo, partiremos de dos matrices, A y B, de tamaños $M \times K$ y $K \times N$, respectivamente, cuyo producto se almacena en la matriz C, de tamaño $M \times N$, por lo tanto, la operación sigue el esquema $C = A \cdot B + C$.

El algoritmo básico para la multiplicación de matrices consiste en recorrer de manera iterativa las dos matrices que se van a multiplicar a través de tres bucles anidados. La implementación básica, sin embargo, resulta muy poco eficiente. La implementación realizada por BLIS sigue las ideas de GotoBLAS, planteando un algoritmo que utiliza seis bucles anidados para realizar un algoritmo a bloques que optimice los accesos a memoria. En la Figura 2.2 se puede ver el pseudocódigo asociado al algoritmo. De estos seis bucles (L1–L6), los primeros tres bucles se encargan de recorrer las matrices y dividirlas

```
L1    for ( j_c = 0;  j_c < n;  j_c += n_c )
L2      for ( p_c = 0;  p_c < k;  p_c += k_c ) {
          B_c ← B(p_c : p_c + k_c − 1, j_c : j_c + n_c − 1); // Pack
L3        for ( i_c = 0;  i_c < m;  i_c += n_c ) {
            A_c ← A(i_c : i_c + m_c − 1, p_c : p_c + k_c − 1); // Pack
L4          for ( j_r = 0;  j_r < n_c;  j_r += n_r ) // Macro-kernel
L5            for ( i_r = 0;  i_r < m_c;  i_r += m_r )
L6              for ( p_r = 0;  p_r < k_c;  p_r += 1 ) // Micro-kernel
                  C_c(i_r : i_r + m_r − 1, j_r : j_r + n_r − 1)
                    += A_c(i_r : i_r + m_r − 1, p_r)
                      · B_c(p_r, j_r : j_r + n_r − 1);
```

Figura 2.2: Algoritmo propuesto por BLIS para la multiplicación de matrices

en bloques. Esta función se conoce también como *empaquetamiento*. Los siguientes dos bucles se conocen como *macro-kernel* y se encargan de recorrer las matrices previamente empaquetadas y dividirlas en bloques aún más pequeños para cargarlos en registros. El último bucle y el más interno se conoce como *micro-kernel* y se encarga de realizar la multiplicación a partir de los elementos cargados en registros. En algunas arquitecturas es posible realizar esta multiplicación con instrucciones vectoriales que pueden ejecutarse incluso en un único ciclo de reloj.

El objetivo principal consiste en optimizar los accesos a memoria a través de la carga de determinados bloques en memoria caché. Más en concreto, el bucle L1, que utiliza la variable j_c, se encarga de recorrer las columnas de la dimensión N del problema en bloques de tamaño n_c. El bucle L2, que utiliza la variable p_c, se encarga de recorrer las columnas de la dimensión K del problema en bloques de tamaño k_c. El bucle L3, que utiliza la variable i_c, se encarga de recorrer las columnas de la dimensión M del problema en bloques de m_c. El propósito de los primeros tres bucles es dividir la matriz en bloques, y el tamaño de estos bloques viene determinado por el tamaño de los diferentes niveles de memoria caché. Estos bloques se nombran como A_c y B_c. Su tamaño viene determinado por las variables, previamente declaradas, m_c, n_c y k_c. Como la matriz A es de tamaño $M \times K$, se utilizan las variables m_c y k_c para reservar el espacio ocupado por A_c, mientras que se utilizan las variables k_c y n_c para determinar el tamaño del bloque B_c, ya que el tamaño de la matriz B es de $K \times N$.

Los siguientes dos bucles, L4 y L5, son los encargados de recorrer los bloques de las matrices previamente cargados en memoria caché y cargar en registros fragmentos pequeños de estas matrices. Estos fragmentos se nombran como $A_r \equiv A_c(i_r : i_r + m_r - 1, k_c)$, $B_r \equiv B_c(k_c, j_r : j_r + n_r - 1)$ y $C_r \equiv C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$, y su tamaño viene definido por las variable m_r, n_r y k_c. Es importante observar (Figura 2.3) cómo están almacenados los datos de los bloques A_c y B_c en memoria y cómo se recorren los mismos a través de los subloques A_r y B_r. Los datos se encuentran almacenados consecutivamente en memoria siguiendo el curso de las flechas. Este esquema de empaquetamiento de datos y de almacenamiento en memoria constituye la clave de la optimización de BLIS.

El último bucle, L6, se conoce también como *micro-kernel* y se encarga de realizar la multiplicación entre los elementos que se encuentran en los registros. En realidad, la configuración mostrada en la Figura 2.2 y seguida en este trabajo no es única. Existe la posibilidad de generar diferentes variantes realizando permutaciones de los bucles. En particular, a la configuración de la Figura 2.2 se le conoce como la variante B3A2C0, cuyas siglas hacen referencia al nivel de memoria caché donde se almacenan los datos. En este caso un bloque de B (B_c) se almacenará en la L3, un bloque de A (A_c) se almacenará en L2 y un bloque de C se guardará en los registros. Cabe destacar que un bloque de B_c (B_r) se carga en el nivel L1. Una vez todos los datos están en sus respectivos niveles de caché, se

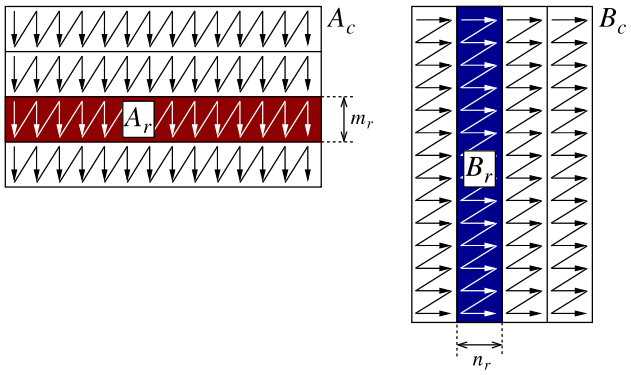

Figura 2.3: Empaquetamiento en la implementación de BLIS de la GEMM.

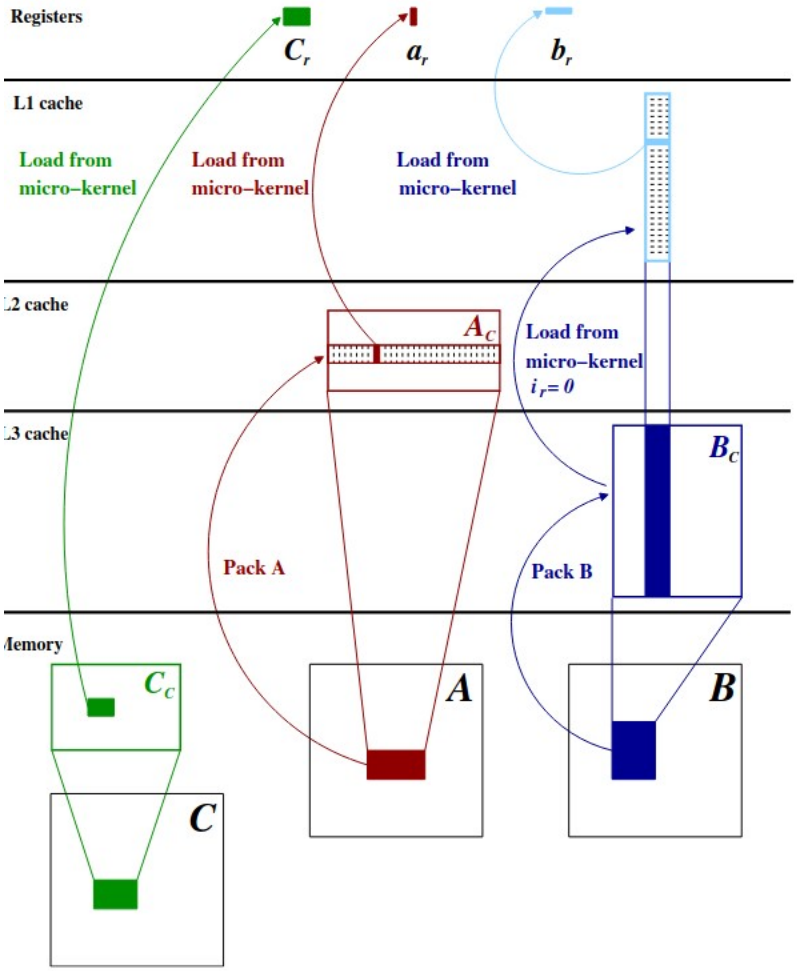

Figura 2.4: Movimientos de los datos en la implementación de BLIS

ejecuta el microkernel entre el bloque de A que se encuentra en L2 y el bloque de B que se encuentra en L1. Ese resultado se almacena en el bloque de C que se encuentra en los registros. La Figura 2.4 muestra los movimientos de datos (bloques) entre los diferentes niveles de caché que tienen lugar durante la ejecución del algoritmo de multiplicación de matrices de BLIS según la variante B3A2C0 mostrada en la Figura 2.2.

Micro-kernels

Dentro del algoritmo de BLIS, el *micro-kernel* hace referencia a los tres bucles más internos que es donde se va a realizar la multiplicación por bloques. Según la Figura 2.2, se

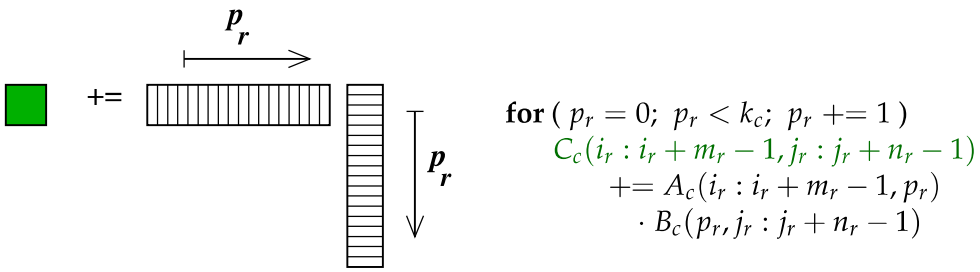

Figura 2.5: Microkernel con *C* residente en registros.

puede ver cómo en el bucle L6 se hace referencia al microkernel y se calcula la multiplicación elemento a elemento. Los *micro-kernels* pueden ser de diferentes tipos en función de la matriz que se encuentre almacenada en registros. En las figuras previamente mostradas, siempre es la matriz *C* la que se encuentra en registros, por lo que el *micro-kernel* asociado a este algoritmo se conoce como "C residente". Este nombre indica que es la matriz *C* la que se almacenará en registros, independientemente de si la variante del algoritmo es B3A2C0 ó A3B2C0. La diferencia entre estas variantes radica en una reordenación de los bucles del algoritmo de la GEMM, pero en ningún caso afecta al *micro-kernel*. En la Figura 2.5 se puede ver gráficamente el funcionamiento de un *micro-kernel* "C residente". Se puede observar el bucle, que recorre una fila de la matriz *A* y una columna de la matriz *B* y acumula el resultado de las multiplicaciones en *C* mediante una operación matricial de tipo *producto exterior* (o *outer product*).

Todos los datos que intervienen en la multiplicación del micro-kernel están almacenados en registros. Por lo tanto, esta multiplicación, en lugar de realizarse elemento a elemento, es posible realizarla utilizando registros e instrucciones vectoriales (SIMD) lo que permite ejecutar múltiples operaciones con una sola instrucción. Las instrucciones vectoriales varían según el procesador, por ejemplo, los procesadores ARM poseen las instrucciones NEON [15], mientras que para el caso de procesadores Intel, tenemos las instrucciones las instrucciones SSE [20] o AVX [19].

2.4 Instrucciones vectoriales e *Intrinsics*

En la mayoría de procesadores modernos, incluso procesadores de baja potencia con arquitectura ARM o RISC-V, podemos encontrar instrucciones vectoriales, es decir, instrucciones que operan sobre registros vectoriales y, por tanto, sobre todos los elementos almacenados en el registro vectorial al mismo tiempo. Por ejemplo, si se quisiera incrementar 4 variables con código no vectorizado, harían falta 4 instrucciones de suma. Por el contrario, al aplicar una instrucción vectorial, se podrían incrementar las 4 variables en solo una instrucción. Estas instrucciones están basadas en registros especiales con mayor capacidad, que permiten almacenar varios elementos de un tipo de dato en concreto, y cada una de estas instrucciones se encarga de realizar una operación matemática. Estas operaciones pueden ser tanto vector-vector como vector-elemento.

Tenemos dos maneras de producir dichas instrucciones, bien el compilador las genera (utilizando opciones apropiadas para autovectorización), o bien las genera el programador escribiendo directamente el código en ensamblador. Sin embargo, existe una tercer vía, intermedia, para incluir instrucciones vectoriales que consiste en utilizar las instrucciones intrínsecas o, simplemente, *intrinsics*. Las *intrinsics* son funciones *inline* de C que contienen núcleos cortos en ensamblador y que permiten introducir código vectorial "a mano" de una manera más sencilla y, por tanto, facilitan la implementación de la vectorización sin necesidad de hacerlo directamente en ensamblador. Trabajar con es-

tas funciones permite realizar códigos específicos utilizando el conocimiento que tiene el programador de los propios núcleos computacionales que está implementando. Además, el compilador se encarga del resto del código para traducirlo a ensamblador como hace habitualmente. Estas instrucciones varían según el procesador sobre el que se vaya a ejecutar el programa, siendo distintas si se trata de un procesador Intel o ARM.

2.4.1. Extensiones SIMD en Intel

Intel proporciona un conjunto de instrucciones vectoriales muy amplio que permite realizar todo tipo de operaciones. Entre las operaciones que se encuentran implementadas, destacan las operaciones matemáticas, las operaciones con cadenas o *strings*, o las operaciones que involucran carga/almacenamiento de valores de/en memoria. Los conjuntos de funciones más utilizados, dentro de los proporcionados por Intel, son las instrucciones SSE [20] y las AVX [19].

En este trabajo se han utilizado las AVX, siendo más modernas que las SSE. Dentro de las instrucciones AVX, algunas de las que se han utilizado son las siguientes:

- `vmovaps`: Esta instrucción recibe dos registros vectoriales y mueve los datos de un registro al otro.
- `vfmadd213ps`: Esta instrucción recibe como parámetros tres registros vectoriales, y multiplica el contenido del primer registro por el del tercero, y al resultado obtenido le suma el contenido del segundo registro. El resultado final se almacena en el primer registro. Se utiliza para realizar la GEMM siguiendo el esquema de $C = C + A \cdot B$.
- `vinsertps`: Esta instrucción se encarga de leer valores que se encuentran en memoria y cargarlos en registros. En este caso, se encarga de cargar en los registros vectoriales los valores de las matrices.

2.4.2. Extensiones SIMD en ARM

ARM ofrece varios conjuntos de instrucciones vectoriales, con soporte para distintos tipos de datos, aunque los tipos de datos más habituales son los tipos `int` (o su versión sin signo, `uint`) y `float`. Este conjunto de instrucciones es conocido con el nombre de NEON [15]. Del mismo modo que en el caso de Intel, ARM ha desarrollado un conjunto de funciones C, que son una versión a "alto nivel" del código ensamblador, denominadas NEON *intrinsics* que facilitan la labor de programación. Antes de la aparición de estas NEON *intrinsics*, la única forma de programar instrucciones vectoriales en ARM era utilizando el código ensamblador propio de esa arquitectura.

Para la implementación de los micro-kernels en ARM, las principales funciones utilizadas han sido las siguientes:

- `vmovq_n_f32`: Esta instrucción recibe un parámetro de tipo `float32` y devuelve un registro vectorial con todos sus valores inicializados al valor del parámetro recibido. Se ha utilizado para inicializar a 0 los registros de la matriz C donde se van a acumular los valores de la multiplicación.
- `vld1q_f32`: Esta instrucción recibe un puntero con una dirección a memoria y carga el contenido en un registro vectorial.
- `vfmaq_laneq_f32`: Esta instrucción multiplica los valores de los registros vectoriales origen y los acumula en otro registro vectorial destino.

- `vst1q_f32`: Esta instrucción almacena en la dirección de memoria recibida como parámetro el registro vectorial recibido también por parámetro.

2.4.3. Extensiones SIMD del RISC-V

RISC-V proporciona un amplio conjunto de instrucciones vectoriales. Estas instrucciones varían en función, tanto del tipo de datos como del tamaño del registro vectorial que se vaya a utilizar. RISC-V da soporte únicamente a los tipos de datos `int`, `uint` y `float`, que es menos de lo que ofrecen ARM, con sus instrucciones NEON, o Intel, con las instrucciones SSE o AVX.

Además, RISC-V proporciona instrucciones específicas en función del tamaño necesario. El tamaño de estas instrucciones depende de un parámetro llamado `LMUL`, que puede tomar los valores 1, 2, 4 y 8. Este parámetro indica el número de registros que se van a utilizar, de manera que si `LMUL` vale 1, se va a trabajar con 4 elementos al mismo tiempo. Si, por ejemplo, `LMUL` vale 2, serán 8 los elementos que se van a procesar con esa instrucción.

La cantidad de instrucciones que RISC-V proporciona es especialmente grande a pesar de solo cubrir tres tipos de datos, puesto que tiene disponible para prácticamente todas las combinaciones disponibles de tipo de dato y valor de `LMUL`. Para evitar que recordar el nombre de cada instrucción sea una tarea demasiado complicada, desde RISC-V han decidido poner el nombre a sus instrucciones siguiendo un patrón que permita identificar rápidamente la instrucción con su cometido [23]. A continuación, se nombran las instrucciones más utilizadas de RISC-V y se explica su funcionamiento:

- `vle32_v_f32m1`: Esta instrucción recibe un puntero y una cantidad `l` de elementos a cargar y devuelve los siguientes `l` elementos consecutivos en memoria al puntero recibido en un registro vectorial.

- `vse32_v_f32m1`: Esta instrucción recibe un puntero, un registro vectorial y una cantidad `l` de elementos a cargar y almacena en la dirección de memoria indicada por el puntero los siguientes `l` elementos del registro vectorial.

- `vfmacc_vf_f32m1`: Esta instrucción se encarga de realizar el producto vector-elemento y acumula el resultado obtenido en otro registro vectorial. En concreto, esa instrucción recibe como parámetros un registro vectorial donde acumular el resultado, una variable de tipo `float` y un registro vectorial con los que realizar el producto, y una cantidad `l` de elementos a cargar.

- `vsse32_v_f32m1`: Esta instrucción se encarga de cargar un elemento individual de un registro vectorial en una variable de tipo `float`. Más en detalle, esta instrucción recibe la variable de tipo `float` donde almacenar el valor, el registro vectorial y el índice del elemento dentro del registro vectorial. El valor de este índice empieza en 1, como sucede en Matlab, en lugar de en 0, como en la gran mayoría de lenguajes de programación.

Nótese el sufijo `m1` en estos ejemplos y que representa el valor de `LMUL` que se utiliza. En estos casos `m1` implica que se va a utilizar una línea de memoria de 128 bits y como el tipo de datos es f32 (coma flotante de 32 bits), significa que afectará a 4 elementos. Cabe destacar el parámetro `l` que aparece en todas las instrucciones vectoriales de RISC-V. Este valor puede limitar el número de elementos a los que afecta la operación. Por ejemplo, si tenemos una instrucción para sumar 2 vectores elemento a elemento del tipo `f32m1` (afectaría a 4 elementos de cada vector) pero ponemos a 2 la variable `l`, la operación solamente afectaría a los 2 primeros elementos.

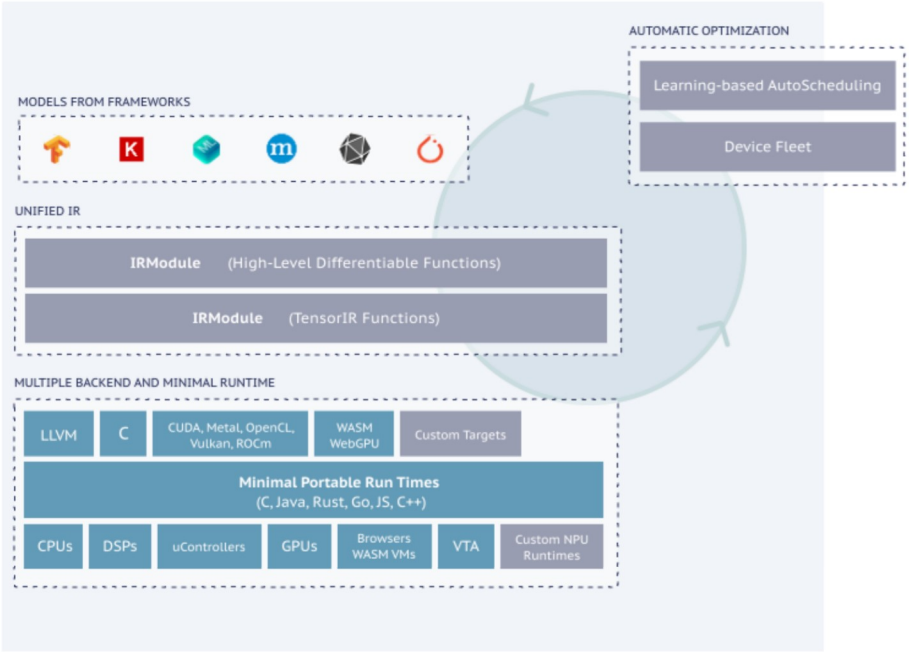

Figura 2.6: Apache TVM framework.

Desafortunadamente, no existen hasta la fecha dispositivos RISC-V que soporten dichas instrucciones y solamente se puede probar su correcto funcionamiento en simuladores. Es cierto que existen implementaciones del RISC-V "hardware" en FPGAs, sin embargo, hasta la fecha no hemos sido capaces de encontrar una implementación lo suficientemente completa que incluya las instrucciones/unidades vectoriales de este procesador.

2.5 El compilador para aprendizaje automático TVM

Apache TVM [14] (*Tensor Virtual Machine*) es un compilador/generador de código automático orientado a problemas de inteligencia artificial. El framework TVM consiste en una pila de herramientas o marco de compilación de código abierto para CPU, GPU y otros aceleradores (Figura 2.6). Está concebido para permitir que los ingenieros de aprendizaje automático optimicen y ejecuten cálculos de manera eficiente en cualquier *backend* hardware ya que su código es independiente del hardware y, por tanto, incrementa su portabilidad.

El trabajo con TVM se lleva a cabo a través de la escritura de un *script* en Python que consta de varias partes.

1. En primer lugar, se tiene que indicar la configuración a seguir, como el tamaño de las matrices, el tipo de datos con el que se va a trabajar (por lo general se utiliza el tipo *float32*) y el procesador donde se va a ejecutar el código generado. A la hora de elegir el procesador, es posible elegir uno distinto en función de si se trata de un procesador de tipo Intel o de ARM a los que, además, se les puede añadir parámetros para especificar alguna arquitectura de Intel en concreto o para que utilicen instrucciones NEON en el caso de ARM. También es posible no indicar el procesador particular de manera que el código que se genere sea un código C que se pueda modificar más adelante. En el Listing 2.1 se puede ver el código que se encarga de esto. En caso de querer cambiar el tipo de dato con el que se va a trabajar, se puede cambiar en la línea 2, poniendo otro en su lugar. En la línea 5 se puede ver un ejem-

```
1 # The default tensor type in tvm
2 dtype = "float32"
3
4 # Hardware backend selection and compiler flags
5 target = "llvm -device=arm_cpu -mattr=+v8.2a,+fp-armv8,+neon"
```

Listing 2.1: Configuración de TVM.

plo de compilación. Se utiliza el compilador LLVM [10] y se le pasan los parámetros de compilación para que utilice las instrucciones NEON, además de indicar que se trata de un procesador ARM (versión 8).

2. Una vez establecida la configuración, el siguiente paso es describir el algoritmo a realizar. El algoritmo es la multiplicación de matrices $C+ = A \cdot B$, por lo que se tiene que indicar que se van a utilizar esas tres variables, junto al tamaño de cada variable. Inicialmente, las matrices A y B son de tamaño $M \times K$ y $K \times N$, respectivamente, para producir una matriz C de tamaño $M \times N$. En el Listing 2.2 se puede observar el código que realiza esta parte. El parámetro *name* que reciben hace referencia al nombre que se le quiere dar a esas variables cuando se genere el código, y es recomendable que coincida con el nombre del `script` de Python para evitar confusiones.

```
1 A = te.placeholder((M, K), name="A")
2 B = te.placeholder((K, N), name="B")
3 C = te.compute((M, N), lambda m, n: te.sum(A[m, k] * B[k, n], axis=k), name
    ="C")
```

Listing 2.2: Creación de las matrices A, B y C.

En cuanto a la matriz C, hay que indicarle tanto el tamaño como la forma de calcular cada elemento. En la línea 3 del Listing 2.2 se indica que la matriz C va a ser de tamaño $M \times N$ y se va a calcular como la suma de la multiplicación elemento a elemento de las matrices A y B.

3. Seguidamente, tal como muestra el Listing 2.3, una vez se ha declarado el problema a resolver se puede construir la función (línea 3) y evaluarla para comprobar su correcto funcionamiento (líneas 6–8). Para calcular el rendimiento, TVM proporciona una función interna que recibe la función recién construida y la ejecuta un número determinado de veces para calcular el tiempo medio de ejecución (línea 10). Finalmente, en la línea 12, se imprime el pseudocódigo generado por TVM. El resultado de esa llamada se observa en el Listing 2.4.

   Entre las líneas 3 y 5 del Listing 2.4 se listan las reservas de memoria necesarias para el almacenamiento de las tres matrices. En las líneas 7, 8 y 10 se pueden ver los bucles *for* anidados de la multiplicación de matrices, mientras que en la línea 9 se puede ver cómo se inicializa la matriz C antes de operar con ella. Finalmente, las líneas 11–13 expresan las operaciones de cálculo.

En resumen, podemos decir que TVM es capaz de generar código C funcional y correcto a partir de un `script` de Python. Para el caso que se plantea en este trabajo, la multiplicación de matrices, existen las optimizaciones que se van a llevar a cabo persiguen dos objetivos fundamentales. El primero consiste en disminuir los fallos de caché a partir de la modificación de los patrones de acceso a los datos de las matrices. El segundo consiste aprovechar el paralelismo o concurrencia en el procesamiento de varios elementos que pueda ofrecer el hardware objetivo a través del uso de instrucciones vectoriales.

```
# Default schedule
s = te.create_schedule(C.op)
func = tvm.build(s, [A, B, C], target=target, name="mmult")
assert func

c = tvm.nd.array(numpy.zeros((M, N), dtype=dtype), dev)
func(a, b, c)
tvm.testing.assert_allclose(c.numpy(), answer, rtol=1e-5)

evaluator = func.time_evaluator(func.entry_name, dev, number=1)

print(tvm.lower(s, [A, B, C], simple_mode=True))
```

Listing 2.3: Creación y prueba de la función.

```
primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
  attr = {"from_legacy_te_schedule": True, "global_symbol": "main", "tir.
    noalias": True}
  buffers = {C: Buffer(C_2: Pointer(float32), float32, [1024, 1024], []),
             A: Buffer(A_2: Pointer(float32), float32, [1024, 1024], []),
             B: Buffer(B_2: Pointer(float32), float32, [1024, 1024], [])}
  buffer_map = {A_1: A, B_1: B, C_1: C} {
  for (m: int32, 0, 1024) {
    for (n: int32, 0, 1024) {
      C_2[((m*1024) + n)] = 0f32
      for (k: int32, 0, 1024) {
        C_2[((m*1024) + n)] = ((float32*)C_2[((m*1024) + n)] +
                               ((float32*)A_2[((m*1024) + k)] *
                               (float32*)B_2[((k*1024) + n)]) )
      }
    }
  }
}
```

Listing 2.4: Pseudocódigo generado por TVM para GEMM.

El algoritmo de BLIS mostrado en la Figura 2.2 trata de estas optimizaciones. La idea consiste en escribir un código TVM que permita generar dicho algoritmo de BLIS, con todas sus optimizaciones. En el Capítulo 3 se explicará cómo convertir un código simple generado con TVM, como el anterior, en el algoritmo de BLIS con todas las distintas posibles optimizaciones que se le pueden aplicar para mejorar su rendimiento.

CAPÍTULO 3
Generación de software

3.1 Introducción

En este capítulo se pretende explicar la generación del software necesaria para realizar la GEMM. En primer lugar, se explica la implementación manual del *micro-kernel* de tamaño $m_r \times n_r = 4 \times 4$ para la arquitectura ARM. En esta implementación se utilizan las instrucciones vectoriales mostradas en la Sección 2.4.2.

También se muestra la implementación realizada manualmente para el procesador RISC-V, siguiendo el diseño asociado al algoritmo de BLIS. Esta implementación utiliza las instrucciones vectoriales vistas en la Sección 2.4.3.

En el capítulo anterior se ha explicado el funcionamiento de TVM para la generación de un ejemplo básico para la autogeneración de la GEMM. En este capítulo se pretende explicar las optimizaciones que se pueden realizar, su funcionamiento y el rendimiento que aporta cada una.

3.2 Técnicas de optimización

A continuación explicamos las diferentes técnicas de optimización que se pueden aplicar al algoritmo de la GEMM de manera genérica o teórica. Más adelante, se explicará cómo se han utilizado estas técnicas, qué resultados han ofrecido y cuál ha sido la mejoría obtenida.

Empaquetamiento: El empaquetamiento es una técnica de optimización que consiste en dividir las matrices con las que se va a operar en bloques y almacenarlas en memoria caché siguiendo un patrón de acceso a memoria especial de manera que se maximice el uso de esta memoria caché y se reduzcan los tiempos de acceso a memoria y lectura de datos. Cada bloque es almacenado en un espacio de memoria aparte y de una manera particular. El tamaño de estos bloques viene determinado por el tamaño de los distintos niveles de memoria caché. En particular, la matriz A, de tamaño $M \times K$, se dividirá en bloques de tamaño $m_c \times k_c$ que se almacenan en el bloque A_c (Figura 2.3); mientras que la matriz B, de tamaño $K \times N$, se dividirá en bloques de tamaño $k_c \times n_c$ que se almacenan en el bloque B_c. Este almacenamiento intermedio persigue la reutilización de datos. La reutilización de los datos determina su almacenamiento en cache (en un nivel o en otro).

Es posible obtener los valores óptimos de m_c, n_c y k_c mediante experimentación, aunque existe una alternativa mucho más rápida y que calcula los valores óptimos a través de un modelo analítico [18]. Por ejemplo, en el caso de BLIS, los valores

preestablecidos por defecto son $m_c = k_c = 480$ y $n_c = 3072$. Sin embargo, el modelo analítico calcula analíticamente los valores óptimos de m_c, n_c y k_c en función de las características de la memoria caché del procesador. El objetivo es maximizar el uso de la caché al empaquetar las matrices A y B. Para hacer esto, es necesario conocer el tamaño de los distintos niveles de las cachés, el grado de asociatividad o si la caché es privada o está compartida, entre otros parámetros.

Las únicas matrices que se tienen que dividir en estos bloques son aquellas que se van a cargar a la memoria caché, por lo que para este caso no es necesario dividir la matriz C, puesto que se cargará directamente en registros.

Reordenación bucles: La reordenación de los bucles consiste en cambiar el orden de ejecución de dos o más bucles anidados. Esta reordenación es posible siempre que los bucles estén *perfectamente anidados*, es decir, no haya ninguna instrucción entre medias que pueda producir una dependencia entre el bucle interno y el externo. Un ejemplo de bucles perfectamente anidados y *reordenables* es la implementación básica de la GEMM, que consta de tres bucles anidados, en este caso, los tres bucles se pueden ejecutar en cualquier orden y el resultado producido será equivalente.

Sin embargo, aunque el orden no afecte al resultado, sí puede afectar al tiempo de ejecución. Diferentes combinaciones pueden ofrecer tiempos de ejecución diferentes debido los diferentes patrones de acceso a memoria que produce cada combinación. El orden propuesto en el algoritmo de BLIS persigue una ejecución lo más eficiente posible.

Desenrollado de bucles: El desenrollado de un bucle es una técnica que consiste en la eliminación total o parcial de un bucle. La idea es la de especificar explícitamente (copiar) un determinado número de veces las instrucciones que forman el cuerpo del bucle. Sería posible incluso eliminar completamente todas las iteraciones del bucle repitiendo todas las instrucciones que forman el bucle. Lo usual es repetir solo unas cuantas instrucciones con lo que el número de saltos debido a la ejecución del bucle disminuye, además de conseguir otros efectos positivos, como el tener alimentado el *pipeline* de ejecución de instrucciones continuamente. El código ensamblador resultante es más largo y ocupa más memoria, pero a cambio se suele obtener un mayor rendimiento.

Vectorización: La vectorización es una técnica de optimización que consiste en la utilización de instrucciones vectoriales para computar varios valores a la vez. Las instrucciones vectoriales trabajan sobre registros vectoriales, donde se almacenan varios elementos simples del mismo tipo. Por lo tanto, existen instrucciones para cargar elementos en los registros vectoriales desde memoria o almacenar en memoria los elementos de un registro vectorial, así como instrucciones para operar con los elementos de los registros vectoriales. Estas instrucciones procesan todos los elementos del registro al mismo tiempo, de ahí la ventaja de su uso. Las instrucciones vectoriales que se utilizan varían según la arquitectura, es decir, las instrucciones vectoriales de ARM son propias de esa arquitectura y no se comparten con otras, como Intel o RISC-V. En la Sección 2.4 se han mostrado algunas de las instrucciones vectoriales que más se han utilizado para cada arquitectura.

Los compiladores suelen generar código "vectorizado" de manera automática cuando se le indica (p.e. opción `O3`), sin embargo, no siempre lo hacen de la mejor manera posible, por ese motivo puede ser conveniente realizar la vectorización "a mano", con *ensamblador* o *vector intrinsics*. La vectorización afecta a los bucles más internos de una jerarquía de bucles, sea cual sea la técnica utilizada para vectorizar. En el caso de TVM se intentará realizar lo mismo que si se adopta la vectorización ma-

nual, tratando de eliminar el bucle más interno. El efecto es el de eliminar uno o más bucles internos, lo que en la Figura 2.2 correspondería al micro-kernel.

Prefetching: El prefetching es una técnica de optimización que consiste en cargar valores a memoria caché o a registros antes de que vayan a ser utilizados. De esta manera, se consigue que esos valores ya estén preparados para utilizarse y que el cuello de botella no sea el acceso a memoria. En el caso de los *micro-kernels* generados manualmente, se realiza cargando en registros los valores que se van a utilizar en la siguiente iteración.

3.3 Generación *manual* de código

En esta sección mostramos la implementación manual de kernels para las arquitectura ARM y RISC-V. Estos *micro-kernels* siguen el esquema $C = C + A \cdot B$, si se quisiera seguir una operación más general, como podría ser $C = beta \cdot C + alpha \cdot A \cdot B$, se tendrían que realizar algunas modificaciones.

3.3.1. Micro-kernels para ARM

La implementación de *micro-kernels* para la arquitectura ARM servirá también como base para la implementación de los *micro-kernels* en RISC-V. Un micro-kernel, como se ha explicado en la Sección 2.3, es el bucle más interno dentro del algoritmo de la GEMM propuesto por BLIS y se encarga de realizar la multiplicación entre los elementos que se encuentren en los registros y que forman parte de las matrices A y B. La implementación de estos *micro-kernels* no es trivial. Hay que tener en cuenta numerosos detalles como, por ejemplo, el tipo de dato con el que se va a trabajar. No es lo mismo *float* que *double*, puesto que no tienen el mismo tamaño en *bytes*, y en un registro vectorial entran el doble de elementos de tipo *float* que de *double* lo que condiciona el algoritmo. También el tamaño del *micro-kernel* hay que tenerlo en cuenta ya que, por ejemplo, el algoritmo de tamaño $m_r \times n_r = 4 \times 4$ es distinto al de 8×8. En la implementación del *micro-kernel* influye también la ubicación de las matrices en los diferentes niveles de memoria caché o registros, especialmente el número de estos últimos.

El Listing 3.1 muestra la implementación referida al *micro-kernel* de tamaño 4×4 para la arquitectura ARM. En primer lugar (línea 1), hay que declarar los registros vectoriales a utilizar. La cantidad de registros vectoriales varía según el tamaño del *micro-kernel* y del tipo de dato utilizado. Después, se tienen que cargar los valores de las matrices A y B de memoria caché a registros (líneas 5 y 6), los punteros `Aptr` y `Bptr` apuntan a las matrices A_c y B_c, respectivamente, del algoritmo de BLIS (Figura 2.2). A continuación (líneas 8 a 12) se realiza la multiplicación utilizando instrucciones vectoriales. Por último, se procede a guardar el resultado calculado anteriormente en su correspondiente lugar de memoria (líneas 17 a 20).

Es posible aplicar a estos microkernels la técnica del *prefetching*, que consiste en cargar valores de memoria previamente a ser utilizados para que ya se encuentren disponibles cuando se necesiten. El Listing 3.2 muestra una versión modificada del del cuerpo del bucle del Listing 3.1 (línea 5–14) con prefetching. En las líneas 5 y 6 se puede ver cómo se cargan los valores de la siguiente iteración.

La generación manual de un *micro-kernel* como el anterior tiene los siguientes dos grandes problemas:

```
float32x4_t C00, C01, C02, C03, A0, B0;

C00 = vmovq_n_f32(0);
C01 = vmovq_n_f32(0);
C02 = vmovq_n_f32(0);
C03 = vmovq_n_f32(0);

for ( k=0; k<kc; k++ ) {

    A0 = vld1q_f32(&Aptr[baseA]);
    B0 = vld1q_f32(&Bptr[baseB]);

    C00 = vfmaq_laneq_f32(C00, A0, B0, 0);
    C01 = vfmaq_laneq_f32(C01, A0, B0, 1);
    C02 = vfmaq_laneq_f32(C02, A0, B0, 2);
    C03 = vfmaq_laneq_f32(C03, A0, B0, 3);

    baseA = baseA+Amr;
    baseB = baseB+Bnr;
}

vst1q_f32(&Ccol(0,0), C00);
vst1q_f32(&Ccol(0,1), C01);
vst1q_f32(&Ccol(0,2), C02);
vst1q_f32(&Ccol(0,3), C03);
```

Listing 3.1: *Micro-kernel* de tamaño 4×4 implementado en ARM.

```
// Prefect colum/row of A/B for next iteration
baseA = baseA+Amr;
baseB = baseB+Bnr;

A0n = vld1q_f32(&Aptr[baseA]);
B0n = vld1q_f32(&Bptr[baseB]);

C0 = vfmaq_laneq_f32(C0, A0, B0, 0);
C1 = vfmaq_laneq_f32(C1, A0, B0, 1);
C2 = vfmaq_laneq_f32(C2, A0, B0, 2);
C3 = vfmaq_laneq_f32(C3, A0, B0, 3);

A0 = A0n;
B0 = B0n;
```

Listing 3.2: Cuerpo del bucle correspondiente al Listing 3.1 con *prefetching*.

- En caso de querer modificar algún aspecto del *micro-kernel*, como el tamaño o el tipo de dato, implica modificar gran parte de la implementación.
- La portabilidad es inexistente al ser una implementación pensada únicamente para esa arquitectura en concreto, si se quisiera llevar a otra sería necesario rehacer el algoritmo desde 0 con las instrucciones propias de dicha arquitectura.

En la Sección 3.3.2 se puede ver la dificultad que acarrea la portabilidad de este algoritmo a otra arquitectura como, por ejemplo, el RISC-V.

3.3.2. Micro-kernels para RISC-V

La realización de los *micro-kernels* para RISC-V se ha hecho utilizando como base los implementados para ARM. En concreto, se han implementado cuatro funciones diferen-

```
1 A0 = vle32_v_f32m1(Aptr, 1);
2 B0 = vle32_v_f32m1(Bptr, 1);
3
4 vsse32_v_f32m1(&b00, 0, B0, 1);
5 vsse32_v_f32m1(&b01, 0, B0, 2);
6 vsse32_v_f32m1(&b02, 0, B0, 3);
7 vsse32_v_f32m1(&b03, 0, B0, 4);
```

Listing 3.3: Carga de las matrices A y B en registros en el RISC-V.

```
1 C0 = vfmacc_vf_f32m1(C0, b00, A0, 1);
2 C1 = vfmacc_vf_f32m1(C1, b01, A0, 1);
3 C2 = vfmacc_vf_f32m1(C2, b02, A0, 1);
4 C3 = vfmacc_vf_f32m1(C3, b03, A0, 1);
```

Listing 3.4: Multiplicación de los registros vectoriales en RISC-V mediante instrucciones vectoriales.

```
1 vse32_v_f32m1(&Ccol(0,0), C0, 1);
2 vse32_v_f32m1(&Ccol(0,1), C1, 1);
3 vse32_v_f32m1(&Ccol(0,2), C2, 1);
4 vse32_v_f32m1(&Ccol(0,3), C3, 1);
```

Listing 3.5: Almacenamiento los valores de la matriz C en memoria en el RISC-V.

tes según las características de cada microkernel. Estas funciones son del tipo "C residente", es decir, con la matriz C almacenada en registros. Además, cada función implementada es diferente en función del tamaño de la matriz de entrada al microkernel y de si se realiza desenrollado de bucles o no. Más en concreto, se han implementado los *microkernels* de tamaño 4×4 y 8×8, tanto en su versión con desenrollado de bucles como sin ella. A continuación se describe la implementación de la función C residente de tamaño 4×4.

En primer lugar, se cargan los elementos que se van a procesar de las matrices A y B en un registro vectorial. A diferencia de ARM, para realizar correctamente la multiplicación, es necesario cargar cada valor de la matriz B en un registro individual. Esta diferencia en las arquitecturas complica mucho la portabilidad de las implementaciones y obliga a los desarrolladores a prácticamente tener que rehacer el trabajo para cada arquitectura existente. En el Listing 3.3 se puede ver el código que realiza esto. Las instrucciones `vle32_v_f32m1` se encargan de cargar en registros los valores de las matrices A y B que se encuentran cargados en caché. Las siguientes cuatro instrucciones que aparecen (línea 4–7) no son necesarias en el caso de ARM puesto que se puede operar punto a punto con cada elemento del registro vectorial sin necesidad de hacer ninguna operación adicional. Sin embargo, esto no es así en el caso de RISC-V, y es necesario cargar individualmente cada valor del registro vectorial en un registro individual para poder operar correctamente. Esto se realiza con la instrucción `vsse32_v_f32m1`.

Una vez cargados todos los valores, se puede realizar la multiplicación. En el Listing 3.4 se puede ver cómo se realiza esto. La instrucción `vfmacc_vf_f32m1` se encarga de multiplicar todos los elementos del registro vectorial $A0$ por el valor del registro $b0X$ y sumar el resultado al que ya hay en el registro vectorial CX para seguir con el esquema de $C = C + A \cdot B$. Por último, los valores calculados se almacenan en la matriz C. En el Listing 3.5 se puede ver cómo se realiza.

Para mejorar el rendimiento de estos algoritmos también aquí se ha utilizado la técnica de *prefetching*. En el Listing 3.6, que muestra el bucle completo del micro-kernel, se

```
for (k = 0; k < kc - 1; k++) {
    // Prefect column/row of A/B for next iteration
    baseA += Amr;
    baseB += Bnr;

    A0n = vle32_v_f32m1(Aptr + baseA, l);
    B0n = vle32_v_f32m1(Bptr + baseB, l);

    C0 = vfmacc_vf_f32m1(C0, b00, A0, l);
    C1 = vfmacc_vf_f32m1(C1, b01, A0, l);
    C2 = vfmacc_vf_f32m1(C2, b02, A0, l);
    C3 = vfmacc_vf_f32m1(C3, b03, A0, l);

    vsse32_v_f32m1(&b00, 0, B0n, 1);
    vsse32_v_f32m1(&b01, 0, B0n, 2);
    vsse32_v_f32m1(&b02, 0, B0n, 3);
    vsse32_v_f32m1(&b03, 0, B0n, 4);

    A0 = A0n;
    B0 = B0n;
}  // end for
```

Listing 3.6: Bucle completo del micro-kernel de 4×4 del RISC-V.

muestra cómo se ha realizado. En este caso, con las instrucciones `vle32v_v_f32m1` se cargan los valores necesarios de las matrices A y B de la siguiente iteración, antes de que se realice la multiplicación. Nótese que para realizar el *prefetching* se utiliza la variable A0n en lugar de la variable A0 para la carga de datos y posteriormente se cargan en la variable A0 para ser utilizadas en la multiplicación.

Otro de los micro-kernels realizados es también de tamaño 4×4 pero añadiendo *unroll* o desenrollado de bucles. En este caso se busca reducir las iteraciones a la mitad, por lo que cada iteración del bucle realizará dos multiplicaciones. El código resultante es el mostrado en el Listing 3.7.

Además, se han implementado versiones de tamaño 8×8, tanto con *unroll* como sin él. La principal diferencia radica en que, al ser 8 elementos en lugar de 4, es necesario cambiar las instrucciones de las funciones. En los casos de tamaño 4×4, todas las instrucciones que se utilizan acaban en *m1*, lo que significa que esa instrucción puede trabajar con 1 línea de memoria, y cada línea tiene capacidad para 4 elementos. En el caso de 8×8, las instrucciones se tienen que cambiar a *m2*. De esta forma, los registros vectoriales y las instrucciones tendrán 2 líneas de memoria y capacidad para 8 elementos. Además, es necesario aumentar el número de registros para la matriz C a 8, en lugar de los 4 utilizados anteriormente. En el Listing 3.8 se puede ver el código resultante.

Al igual que en el caso del 4×4, es posible hacer *unroll* para el 8×8. El código final es el mismo que sin utilizar *unroll*, salvo por el hecho de que se duplica el cuerpo del bucle, tal como muestra el Listing 3.7.

3.4 Generación *Automática* de código

En la sección de TVM se ha explicado cómo generar un código básico que realice la multiplicación de matrices. Sin embargo, la función generada ofrece un rendimiento muy pobre en comparación a los códigos implementados manualmente. Afortunadamente, TVM ofrece una serie de funciones con las que optimizar y mejorar la función generada y acabar teniendo un código prácticamente idéntico al implementado manualmente. A

```
for (k = 0; k < kc - 2; k += 2) {
    // Prefetch colum/row of A/B for next iteration
    baseA += Amr;
    baseB += Bnr;

    A0n = vle32_v_f32m1(Aptr + baseA, l);
    B0n = vle32_v_f32m1(Bptr + baseB, l);

    C0 = vfmacc_vf_f32m1(C0, b00, A0, l);
    C1 = vfmacc_vf_f32m1(C1, b01, A0, l);
    C2 = vfmacc_vf_f32m1(C2, b02, A0, l);
    C3 = vfmacc_vf_f32m1(C3, b03, A0, l);

    vsse32_v_f32m1(&b00, 0, B0n, 1);
    vsse32_v_f32m1(&b01, 0, B0n, 2);
    vsse32_v_f32m1(&b02, 0, B0n, 3);
    vsse32_v_f32m1(&b03, 0, B0n, 4);

    A0 = A0n;

    // Prefetch colum/row of A/B for next iteration
    baseA += Amr;
    baseB += Bnr;

    A0n = vle32_v_f32m1(Aptr + baseA, l);
    B0n = vle32_v_f32m1(Bptr + baseB, l);

    C0 = vfmacc_vf_f32m1(C0, b00, A0, l);
    C1 = vfmacc_vf_f32m1(C1, b01, A0, l);
    C2 = vfmacc_vf_f32m1(C2, b02, A0, l);
    C3 = vfmacc_vf_f32m1(C3, b03, A0, l);

    vsse32_v_f32m1(&b00, 0, B0n, 1);
    vsse32_v_f32m1(&b01, 0, B0n, 2);
    vsse32_v_f32m1(&b02, 0, B0n, 3);
    vsse32_v_f32m1(&b03, 0, B0n, 4);

    A0 = A0n;
}  // end for
```

Listing 3.7: Microkernel 4×4 con *unroll* para el RISC-V.

continuación, se procede a explicar las técnicas y funciones utilizadas para la optimización y cómo se expresan en TVM. Se mostrará cómo se implementa la función que se va a probar y con la que se van a obtener resultados. Terminaremos explicando el rendimiento obtenido con el código generado automáticamente con TVM.

3.4.1. Incorporación de técnicas de optimización con TVM

Empaquetamiento y reordenación

El empaquetamiento, tal como hemos descrito, consiste en dividir una matriz en bloques más pequeños y replicarlos en memoria para que, mediante reutilización, se carguen en caché y, por tanto, se optimice el número de accesos a memoria. En este caso seguimos la filosofía de BLIS y, por tanto, no se va a operar directamente con las matrices A y B, sino que se va a trabajar con bloques de A y B más pequeños: el bloque A_c, de tamaño $m_c \times k_c$, y el bloque B_c, de tamaño $k_c \times n_c$.

```
 for (k = 0; k < kc - 1; k++) {
    // Prefetch colum/row of A/B for next iteration
    baseA += Amr;
    baseB += Bnr;

    A0n = vle32_v_f32m2(Aptr + baseA, l);
    B0n = vle32_v_f32m2(Bptr + baseB, l);

    C00 = vfmacc_vf_f32m2(C00, b00, A00, l);
    C01 = vfmacc_vf_f32m2(C01, b01, A00, l);
    ...
    C07 = vfmacc_vf_f32m2(C07, b07, A00, l);

    vsse32_v_f32m2(&b00, 0, B0n, 1);
    vsse32_v_f32m2(&b01, 0, B0n, 2);
    ...
    vsse32_v_f32m2(&b07, 0, B0n, 8);

    A00 = A0n;
}  // end for
```

Listing 3.8: Microkernel de tamaño 8×8 para el RISC-V.

```
1 Ac = te.compute((K / kc, math.ceil(M / mr), kc, mr),
2     lambda m, n, z, l: A[m * kc + z, n * mr + l], name="Ar")
3 Bc = te.compute((K / kc, math.ceil(N / nr), kc, nr),
4     lambda m, n, z, l: B[m * kc + z, n * nr + l], name="Br")
5
6 C = te.compute((M, N), lambda m, n: te.sum(Ac[k // kc, m // mr,
7         tvm.tir.indexmod(k, kc), tvm.tir.indexmod(m, mr)] *
8         Bc[k // kc, n // nr, tvm.tir.indexmod(k, kc),
9         tvm.tir.indexmod(n, nr)], axis=k), name="C")
```

Listing 3.9: Creación de los bloques A_c y B_c y de la matriz C en TVM.

En el Listing 3.9 se muestra cómo realizar el empaquetamiento. En la líneas 1 y 2 se declara un tensor propio de TVM de 4 dimensiones que representa una vista de la matriz A. La función *lambda* se utiliza para realizar la copia de los datos de la matriz A al bloque A_r, mientras que las cuatro variables de la función se convierten en cuatro bucles que corresponden a las cuatro dimensiones de A_r. En la líneas 3 y 4 se realiza la misma operación, pero en este caso, para B_r. En la líneas 6 a 9 se define la operación que se realiza para la matriz C, que es la GEMM siguiendo el esquema $C = C + A \cdot B$, utilizando las variables A_c y B_c creadas en las dos líneas anteriores.

El siguiente paso ha consistido en representar la reordenación de bucles de una manera sencilla en TVM. El Listing 3.10 muestra la configuración de la reordenación de bucles, que se produce exactamente en la línea 10.

Desenrollado y vectorización

El desenrollado es una técnica que consiste en la eliminación total o parcial de un bucle. Esto se consigue replicando instrucciones y eliminando iteraciones del bucle. Es posible realizar esta técnica completamente y eliminar todas las iteraciones del bucle, o eliminar parcialmente algunas. En la implementación manual se hizo de manera parcial, eliminando tan solo la mitad de las iteraciones. Con TVM tenemos más flexibilidad para eliminar de manera automática el número deseado de iteraciones, llegando a eliminarlas todas si se desea.

```
# Default schedule
blis = te.create_schedule(C.op)

ic, jc, \
icin, jcin = blis[C].tile(C.op.axis[0], C.op.axis[1], mc, nc)

p, = blis[C].op.reduce_axis
pc, pr = blis[C].split(p, factor=kc)

blis[C].reorder(jc, pc, ic, jr, ir, pr)
```

Listing 3.10: Reordenación de bucles en TVM.

Respecto de la vectorización, tenemos la posibilidad de realizar tres vectorizaciones distintas, una para cada matriz. Para las matrices A y B, la vectorización se realiza al cargar los elementos de memoria para hacer el empaquetamiento, mientras que para la matriz C, la vectorización es para realizar la multiplicación, así como para guardar el resultado en la memoria. Además, para realizar correctamente el desenrollado y la vectorización, es necesario exponer los bucles relativos al micro-kernel. En estos bucles es donde se realiza la operación de multiplicación.

El código implementado para lograr el desenrollado y la vectorización se puede ver en el Listing 3.11. Las líneas 6 y 7 muestran cómo se dividen los bucles en función del tamaño del micro-kernel ($m_r \times n_r$) para, posteriormente, vectorizar y desenrollar algunos de estos bucles. En las líneas 14, 18, 23 y 27 muestran la llamada a las instrucciones vectoriales.

```
blis = te.create_schedule(C.op)

ic, jc, \
icin, jcin = blis[C].tile(C.op.axis[0], C.op.axis[1], mc, nc)

ir, it = blis[C].split(icin, factor = mr)
jr, jt = blis[C].split(jcin, factor = nr)

p, = blis[C].op.reduce_axis
pc, pr = blis[C].split(p, factor=kc)

blis[C].reorder(jc, pc, ic, jr, ir, pr, it, jt)

blis[C].vectorize(jt)

blis[Ac].compute_at(blis[C], ic)
x,y,z,l = Ac.op.axis
blis[Ac].vectorize(l)


blis[Bc].compute_at(blis[C], pc)
x,y,z,l = Bc.op.axis
blis[Bc].vectorize(l)

blis[ac].compute_at(blis[C], pr)
x,y,z,l = ac.op.axis
blis[ac].vectorize(l)
```

Listing 3.11: Desenrollado y vectorización en TVM.

División de bucles

El tamaño de los registros vectoriales varía según el procesador y la arquitectura sobre la que se quiera ejecutar la función. Para maximizar el uso de estos registros, es conveniente dividir los bucles en dos, uno externo con una gran carga de iteraciones, y uno interno, con el tamaño de los registros vectoriales. El atributo *linesize* representa este tamaño de los registros vectoriales en los que hay que dividir los bucles, y variará en función de la arquitectura elegida para trabajar (ver Listing 3.12, línea 14). En este caso, se divide el bucle más interno del algoritmo para facilitar a TVM la vectorización. Esto permite, no solo vectorizar el bucle más interno de los dos que se han dividido, sino, además, desenrollar el bucle externo (ver líneas 15 y 16).

```
1  blis = te.create_schedule(C.op)
2
3  ic, jc, \
4  icin, jcin = blis[C].tile(C.op.axis[0], C.op.axis[1], mc, nc)
5
6  ir, it = blis[C].split(icin, factor = mr)
7  jr, jt = blis[C].split(jcin, factor = nr)
8
9  p, = blis[C].op.reduce_axis
10 pc, pr = blis[C].split(p, factor=kc)
11
12 blis[C].reorder(jc, pc, ic, jr, ir, pr, it, jt)
13
14 jto,jt = blis[C].split(jt, factor=linesize)
15 blis[C].unroll(jto)
16 blis[C].vectorize(jt)
17
18 blis[Ac].compute_at(blis[C], ic)
19 x,y,z,l = Ac.op.axis
20 blis[Ac].vectorize(l)
21
22 blis[Bc].compute_at(blis[C], pc)
23 x,y,z,l = Bc.op.axis
24 blis[Bc].vectorize(l)
25
26 blis[ac].compute_at(blis[C], pr)
27 x,y,z,l = ac.op.axis
28 blis[ac].vectorize(l)
```

Listing 3.12: División de los bucles en TVM.

3.4.2. Creación y ejecución de la función

Una vez realizadas todas las optimizaciones, es necesario construir la función para ejecutarla y medir los tiempos obtenidos. También se puede imprimir en pantalla un pseudocódigo asociado a la función construida, o el código C completo que ha generado la función (Listing 3.13).

En la línea 2 se puede ver cómo se construye la función, indicando el *target* para el que se quiere generar la función y si se va a utilizar algún conjunto de instrucciones específico, como pueden ser las NEON [15] en el caso de ARM, o AXV [19] o SSE [20] para Intel. La línea 5 sirve para mostrar el pseudocódigo generado, aunque es posible indicar también que se muestre toda la función. En la línea 8 se indica que la función generada se va a ejecutar tantas veces como indique el parámetro `numRepeticiones`. De esta manera, se

```
1 # BUILD FUNCTION
2 func = tvm.build(s, [A, B, C], target=target, name="mmult")
3
4 #SHOW PSEUDO-CODE
5 print(tvm.lower(s, [A, B, C], simple_mode=True))
6
7 #TIME MEASUREMENT
8 evaluator = func.time_evaluator(func.entry_name, dev, number=
      numRepeticiones)
9 tiempo = evaluator(a, b, c).mean
```

Listing 3.13: Creación y ejecución de la función en TVM.

evita posibles errores en la medición de tiempos. Por último, en la línea 9 se ejecuta la función y se devuelve el tiempo medio de cada repetición.

Para mostrar el impacto del desarrollo realizado hasta el momento, se ha llevado a cabo una simulación, cuyos datos se muestran en la Figura 3.1. La figura muestra los GFLOPS obtenidos entre la versión inicial y la versión final, pasando por las diferentes mejoras adoptadas durante el desarrollo. Las pruebas se han realizado sobre un ARM Carmel a modo de ejemplo, y solo se ha comprobado el rendimiento de TVM con las diferentes optimizaciones que se pueden aplicar. En el Capítulo 4 se muestran los resultados obtenidos comparando la función generada por TVM con las implementaciones de algunas bibliotecas.

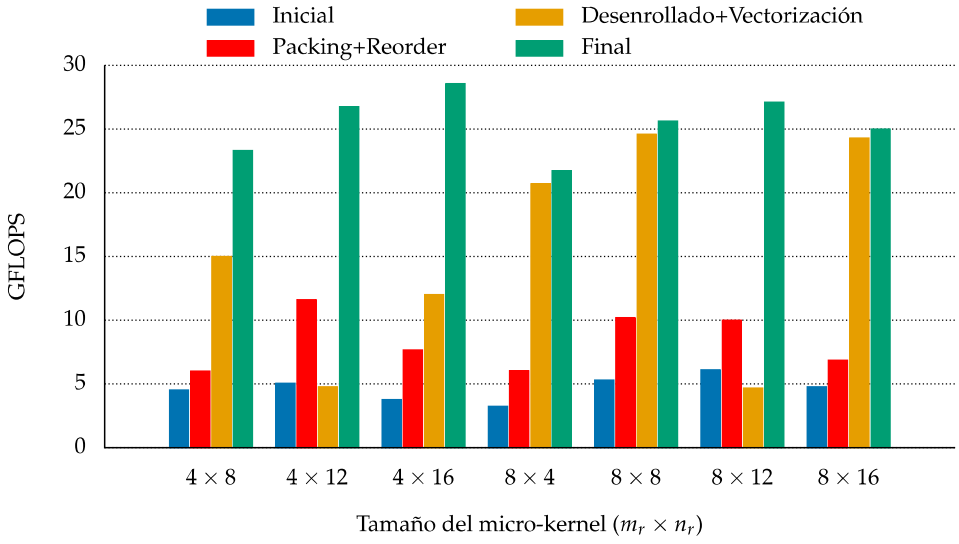

Figura 3.1: Comparación del rendimiento obtenido entre las distintas versiones obtenidas con las mejoras introducidas en el código TVM.

En la primera columna (azul) se muestra el rendimiento inicial obtenido por TVM, sin aplicar ningún tipo de optimización. Este rendimiento inicial llega hasta los 6 GFLOPS en el mejor de los casos. En la segunda columna (rojo) se muestra el rendimiento después de aplicar el empaquetamiento junto al reordenamiento de los bucles. Con esta mejora se es capaz de alcanzar hasta los 11,6 GFLOPS en el mejor de los casos. En la tercera columna (naranja) se muestra el rendimiento después de aplicar el desenrollado de bucles y la vectorización a la mejora anterior, con la cuál es posible alcanzar los 24,7 GFLOPS, aunque su rendimiento varía considerablemente en función del tamaño del microkernel.

En la última columna se muestra el rendimiento total una vez aplicadas todas las optimizaciones, es decir, las mejoras del paso anterior junto a la división de los bucles. Con todas las mejoras, TVM es capaz de alcanzar los 28,5 GFLOPS, un valor significativo para esta arquitectura tal como se verá en el capítulo siguiente.

En conclusión, se puede ver cómo con unas pocas líneas de código es posible aumentar considerablemente el rendimiento obtenido respecto a la versión inicial, pasando de 6 GFLOPS a 28,5. Además, gracias a la flexibilidad de TVM, es posible probar distintas combinaciones de vectorización o desenrollado de bucles hasta encontrar la versión con mayor rendimiento. Esta búsqueda de los mejores parámetros se puede automatizar más fácilmente gracias a TVM.

3.4.3. Portabilidad con TVM

Uno de los aspectos más destacables de TVM consiste en la facilidad que ofrece en términos de portabilidad entre arquitecturas. Por ejemplo, el mismo código que se ha detallado en el punto anterior, se podría aplicar a diferentes procesadores de Intel, ARM o a versiones particulares de la arquitectura RISC-V, con la única modificación del hardware seleccionado. Para ello, se modificarían los *flags* de compilación y el nombre del procesador para adaptarlos al entorno requerido. Por ejemplo, en el caso de estar utilizando una máquina ARM, la línea de compilación referida al *target* es la siguiente:

```
target = "llvm -device=arm_cpu -mattr=+v8.2a,+fp-armv8,+neon".
```

Si se quisiera cambiar a una arquitectura Intel Cascade Lake, solo habría que cambiar esa línea por la siguiente:

```
target = "llvm -mcpu=cascadelake".
```

Así mismo, aunque el código anterior se basa en el tipo de dato `float` de 32 bits, bastaría con cambiar ese atributo por cualquier tipo de datos soportados para generar un código ensamblador que los utilice.

3.4.4. Ejecución con TVM

En esta sección se explica cómo ejecutar TVM y cómo es posible generar distintos *micro-kernels* en una única ejecución. En primer lugar, TVM no deja de ser un programa escrito en Python, por lo que su ejecución es tan simple como ejecutar en la consola de comandos: `python3 tvm.py`.

Para facilitar la modificación de algunos parámetros, como el tamaño del *micro-kernel*, el tamaño de las matrices o las características de la máquina donde se va a ejecutar, necesarias para calcular de manera óptima m_c, n_c y k_c, estos parámetros se han abstraído del fichero de TVM y declarado en unos ficheros de configuración externos que TVM recibe como argumentos. La nueva llamada ahora sería la siguiente:

```
python3 tvm.py carmel.cfg blis_retreat_cuadrado_2000.cfg.
```

Es posible indicar dentro del segundo fichero los tamaños iniciales y finales de m_r, n_r y k_r, junto al *step* que se quiere tener para m_r, n_r y k_r. Dentro del fichero de TVM hay una serie de bucles anidados que se encarga de ejecutar todas las combinaciones:

```
for i in range (mr_ini, mr_end, mr_step).
```

CAPÍTULO 4
Evaluación

Para evaluar los *micro-kernels* generados por TVM, se propone la comparación de estos con bibliotecas como BLIS, OpenBLAS, ARMPL o MKL. Esta experimentación se ha realizado en arquitecturas diferentes, en concreto, en un ARM Carmel [5], en un Intel Skylake [7] y en un Intel Cascade Lake [6].

Esta evaluación se ha realizado utilizando dos tipos de matrices diferentes. El primero de ellos, utilizando matrices cuadradas de tamaño $M = N = K = 2000$. Para el segundo tipo de matrices se ha decidido optar por un caso de uso real, utilizado como base las capas de un modelo de red neuronal llamado `ResNet50-v1.5` [13]. `ResNet50` consiste en un modelo de red neuronal profunda que entre sus capas utiliza convoluciones de matrices. Es posible expresar estas convoluciones como un producto de matrices y realizar su producto a través de las implementaciones anteriormente mencionadas. Estas matrices tienen la característica de tener una forma "muy" rectangular. Este tipo de matrices son conocidos también como *tall-and-skinny*.

4.1 Preparación del entorno

El entorno de evaluación ha sido muy variado con la intención de probar la generación automática de software optimizado producida por un solo código TVM sobre una gran diversidad de arquitecturas. En concreto, se han utilizado un total de tres arquitecturas diferentes para realizar las pruebas.

- La primera de estas máquinas es un NVIDIA Jetson AGX Xavier, que cuenta con un procesador ARM Carmel. Este procesador contiene un total de 4 cores ARM (v8.2), cada uno con 2 hilos y una caché L2 de 2 MB por núcleo. Todos los núcleos tienen una caché L1 de 64 KB, y están conectados a una caché L3 compartida de 4 MB. Todos los experimentos se han ejecutado sobre un núcleo de ARM Carmel y utilizando aritmética de simple precisión. En cuanto a la parte de *software*, se ha ejecutado bajo un Linux con distribución Ubuntu (v18.04).

- La segunda de estas máquinas es un Intel Xeon Silver 4210. Esta familia de procesadores se conoce también como Cascade Lake, y sale nombrado así en las gráficas que hacen referencia a los experimentos realizados en esta máquina. Este procesador cuenta con 10 núcleos y una memoria caché compartida de 13,75 MB. Todos los experimentos se han ejecutado sobre un único núcleo y utilizando aritmética de simple precisión. En cuanto a la parte de *software*, se ha ejecutado bajo un Linux con distribución Ubuntu (v18.04).

- La tercera y última de las máquinas es un Intel Xeon Gold 6138. Esta familia de procesadores se conoce también como Skylake, y sale nombrado así en las gráficas que hacen referencia a los experimentos realizados en esta máquina. Este procesador cuenta con 20 núcleos y una memoria caché compartida de 13,75 MB. Todos los experimentos se han ejecutado sobre un único núcleo y utilizando aritmética de simple precisión. En cuanto a la parte de *software*, se ha ejecutado bajo un Linux con distribución Ubuntu (v18.04).

Además, se han comparado diferentes bibliotecas para cada arquitectura. Por un lado, para ARM se ha utilizado OpenBLAS [26] y ARMPL (ARM *Performance Library*) [2]. Por otro lado, para Intel se ha comparado con la biblioteca MKL [21]. Tanto para ARM como para Intel se ha comparado con BLIS (v0.8.1) que, además, es la fuente sobre la que se ha construido la optimización de TVM.

4.2 Matrices cuadradas

Como se ha explicado en la Sección 3.4.1, el primer paso del algoritmo consiste en empaquetar la matriz en bloques de tamaño m_c, n_c y k_c. Este empaquetamiento se realiza para optimizar el acceso de los datos en memoria, al cargar bloques de datos a distintos niveles de cachés que luego serán utilizados, logrando evitar en gran medida fallos de caché y disminuyendo considerablemente el tiempo de ejecución. La elección de los valores de m_c, n_c y k_c no es trivial y varía según la biblioteca. Elegir unos valores u otros afectará posteriormente al rendimiento del algoritmo al verse afectados por características propias de cada máquina relativas a la memoria caché. En el caso de BLIS, estos valores vienen prefijados por lo que pueden no ser los óptimos para cada máquina. En el caso de TVM puede seleccionarse el valor que se desee y hacer pruebas hasta encontrar aquellos valores que mejor rendimiento ofrezcan. Para este trabajo, la elección de los valores se calcula mediante un procedimiento analítico siguiendo el siguiente artículo [18]. Por tanto, se tiene en cuenta tanto el tamaño de las matrices A y B como las características del procesador relativas a la memoria caché del mismo. Se ha decidido probar el rendimiento en matrices cuadradas con tamaño $M = N = K = 2000$ porque se ha considerado un tamaño suficiente para realizar comparaciones.

4.2.1. Resultados en ARM

En primer lugar, en la Figura 4.1, se presenta el rendimiento obtenido por las diferentes implementaciones de la GEMM para matrices cuadradas. BLIS es capaz de obtener el mayor rendimiento, con 28,8 GFLOPS, mientras que ARMPL se queda en 26,4 GFLOPS y OpenBLAS, en 23,3 GFLOPS. Los micro-kernels generados por TVM ofrecen distintos rendimientos, mientras que la gran mayoría se quedan muy por debajo del rendimiento obtenido por las otras implementaciones, los micro-kernels de 8×12 y 12×8 son capaces de conseguir 26,8 GFLOPS, mejorando los resultados de ARMPL y OpenBLAS, pero quedándose por detrás de BLIS.

La diferencia entre TVM y BLIS se debe a un *prefetch* que realiza BLIS en su algoritmo. Se puede replicar este *prefetch* en TVM utilizando la función `cache_read`, que indica que se utilice un *prefetch* automático donde sea posible. De esta manera, se consigue mejorar el rendimiento hasta los 28,9 GFLOPS e igualar el de BLIS, tal como muestra la Figura 4.2.

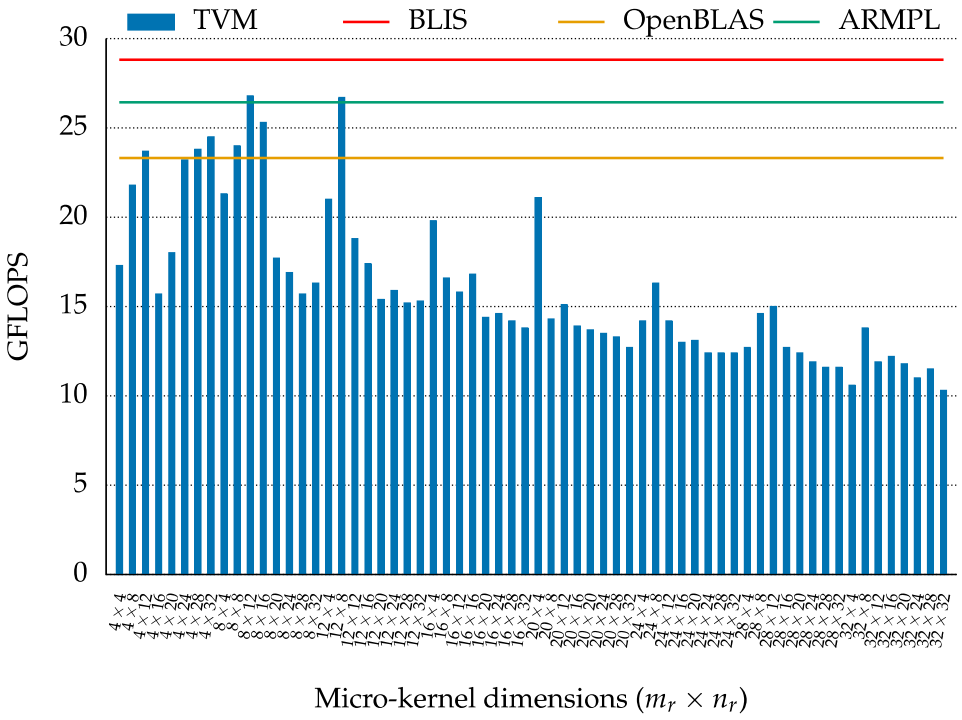

Figura 4.1: Rendimiento en ARM Carmel para matrices cuadradas (sin *prefetching*).

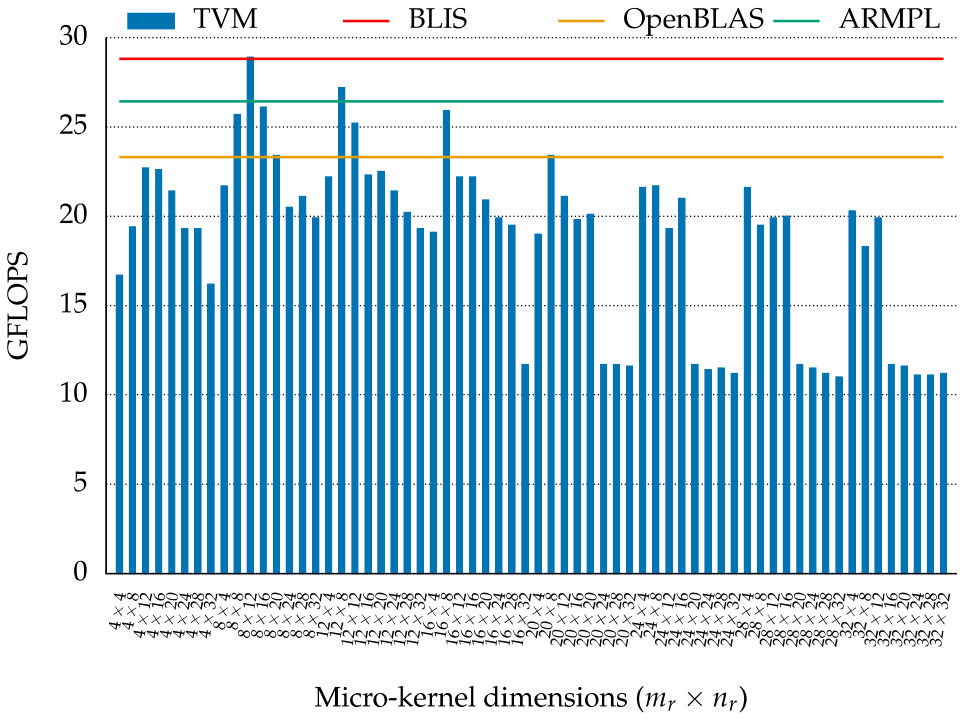

Figura 4.2: Rendimiento en ARM Carmel para matrices cuadradas utilizando *prefetching*.

4.2.2. Resultados en Intel

En Intel Cascade Lake también se ha realizado el mismo experimento (ver Figura 4.3). En esta arquitectura, el micro-kernel generado por TVM de tamaño 4×32 es capaz de alcanzar los 53,3 GFLOPS, siendo que las bibliotecas BLIS y MKL no son capaces de alcanzar esas cifras, quedándose en 45 y 46,2 GFLOPS, respectivamente.

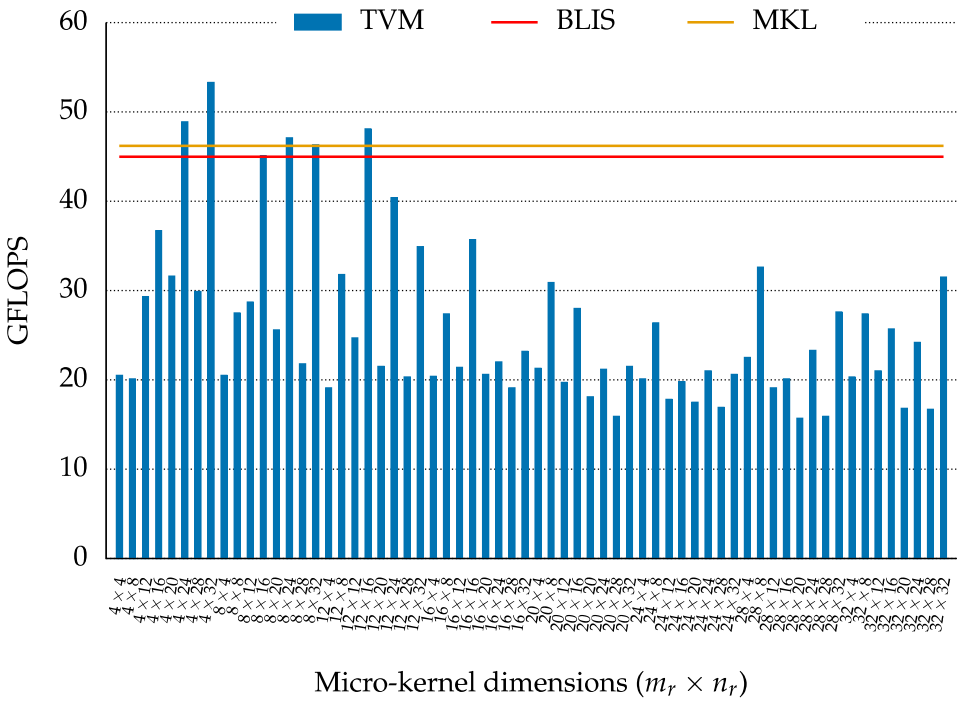

Figura 4.3: Rendimiento en Intel Cascade Lake para matrices cuadradas.

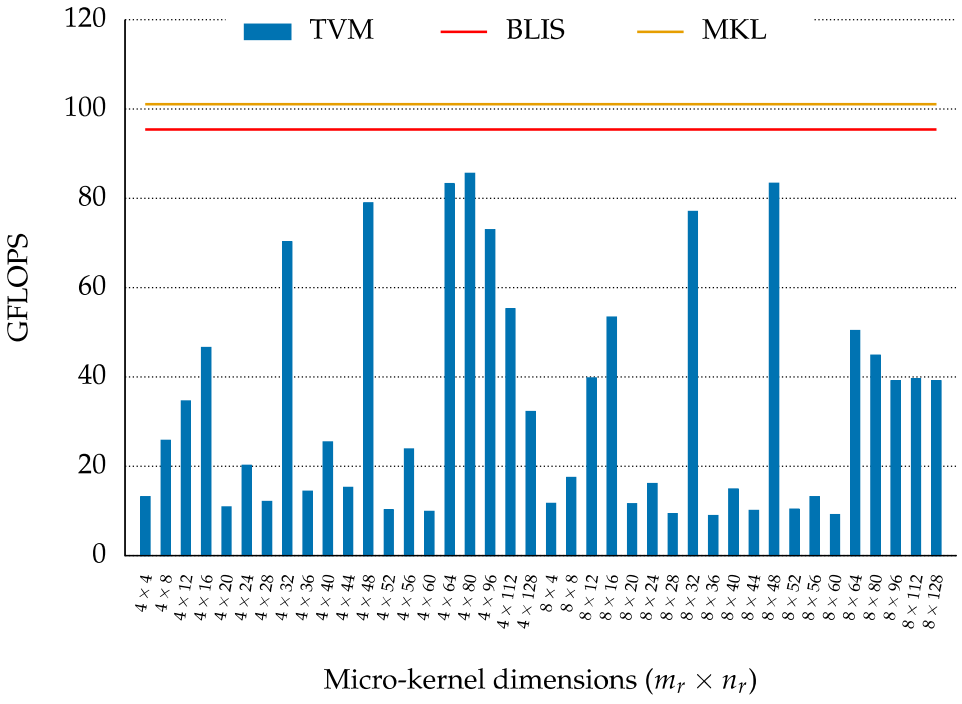

Figura 4.4: Rendimiento en Intel Skylake para matrices cuadradas.

En Intel Skylake también se ha realizado el mismo experimento (ver Figura 4.4), aunque con diferentes tamaños de *micro-kernel*. En este caso, se ha llegado hasta el micro-kernel de tamaño 8×64, incrementando n_r en 4. Analizando los primeros resultados, se vio que aquellos *micro-kernels* que eran múltiplos de 16 ofrecían un rendimiento mucho mayor al resto, por lo que se decidió ampliar las pruebas hasta el micro-kernel de tamaño 8×128, pero incrementando n_r en 16. El motivo de utilizar un incremento de 16 es

Número de capa	Tamaño de la GEMM
001	$1605632 \times 64 \times 147$
006	$\mathbf{401408 \times 64 \times 64}$
009/021/031	$401408 \times 64 \times 576$
012/014/024/034	$401408 \times 256 \times 64$
018/028	$401408 \times 64 \times 256$
038	$401408 \times 128 \times 256$
041/053/063/073	$100352 \times 128 \times 1152$
044/056/066/076	$\mathbf{100352 \times 512 \times 128}$
046	$100352 \times 512 \times 256$
050/060/070	$100352 \times 128 \times 512$
080	$100352 \times 256 \times 512$
083/095/105/115/125/135	$25088 \times 256 \times 2304$
086/098/108/118/128/138	$25088 \times 1024 \times 256$
088	$25088 \times 1024 \times 512$
092/102/112/122/132	$25088 \times 256 \times 1024$
142	$25088 \times 512 \times 1024$
145/157/167	$6272 \times 512 \times 4608$
148/160/170	$6272 \times 2048 \times 512$
150	$6272 \times 2048 \times 1024$
154/164	$6272 \times 512 \times 2048$

Tabla 4.1: Tamaños de las convoluciones utilizadas por la red neuronal RESNET50

porque se trata del mismo valor que el de la variable *linesize*, utilizada para la división de bucles. Dicha división funciona correctamente solo con aquellos micro-kernels con un n_r múltiplo del valor de *linesize*. Para esta arquitectura, la biblioteca MKL es capaz de conseguir el mejor rendimiento, al alcanzar los 101,1 GFLOPS, mientras que el código generado por TVM llega hasta los 88,5 GFLOPS y BLIS a los 95,42 GFLOPS. Para estos dos últimos casos, TVM no se ha ejecutado con *prefetch*, por lo que sería posible obtener aún un poco más de rendimiento.

4.3 Matrices de la RN ResNet50

Dado el interés que te tiene en el marco de este trabajo la utilización de casos reales derivados de la utilización de redes neuronales se ha decidido probar el rendimiento con tamaños de matrices típicos de estos casos. Por tanto, se van a comparar los rendimientos obtenidos entre distintas implementaciones de la GEMM, utilizando las bibliotecas BLIS, OpenBLAS, ARMPL y MKL, junto a la implementación generada por TVM en la RN ResNet50. En la Tabla 4.1 se pueden ver los tamaños de matrices utilizados en cada capa que utiliza la GEMM.

Hemos elegido dos de las capas de ResNet50 por su representatividad, las capas número 006 y 044. Los resultados de los experimentos se han obtenido tanto en ARM como en Intel, utilizando las mismas máquinas que para los experimentos anteriores. En primer lugar, se ha buscado el micro-kernel óptimo para cada una de las capas, para presentar, después, el resultado obtenido en cada capa con el mejor micro-kernel generado por TVM.

La RN ResNet50 utiliza convoluciones de matrices entre sus capas. Es posible convertir estas convoluciones a multiplicaciones de matrices a través de la operación IM2COL.

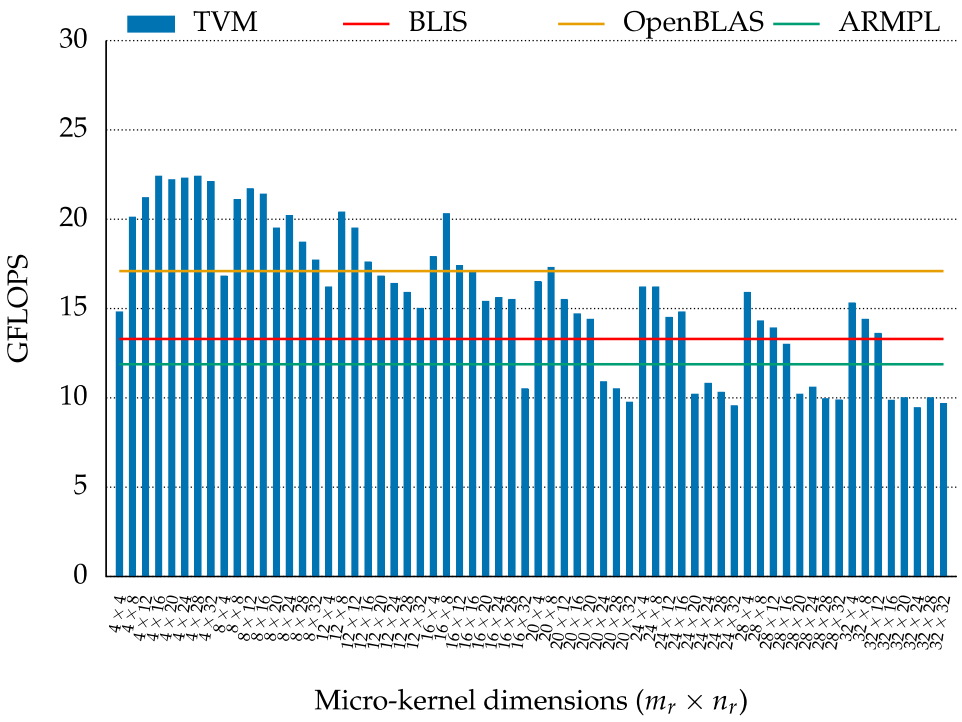

Figura 4.5: Rendimiento en ARM Carmel para $m = 401408, n = k = 64$ (capa 006).

La realización de esta operación queda fuera del alcance de este trabajo, y se supone que esta operación ya se ha realizado cuando se reciben las matrices como parámetros.

4.3.1. Rendimiento en ARM

Según muestra la Figura 4.5, las bibliotecas OpenBLAS, BLIS o ARMPL son capaces de obtener un rendimiento de 17,1, 13,3 y 11,9 GFLOPS, respectivamente. Por otro lado, la flexibilidad de TVM permite obtener resultados con múltiples micro-kernels, desde 4×4 hasta 32×32. Después de realizar el barrido correspondiente por todos los tamaños de micro-kernel, se puede observar que los resultados obtenidos por TVM son claramente superiores a los de BLIS y OpenBLAS si se utiliza el micro-kernel adecuado. Existen múltiples combinaciones de tamaño de micro-kernel que son capaces de mejorar el rendimiento ofrecido por estas bibliotecas, pero el micro-kernel de 4×32 es el que es capaz de sacar el mayor rendimiento, siendo este de 24,4 GFLOPS.

Si se elige otra capa (Figura 4.6), vemos que los resultados son similares, con el micro-kernel óptimo generado por TVM siendo capaz de superar al resto de bibliotecas. No obstante, en este caso, la diferencia entre el resto de implementaciones es mucho menor. El micro-kernel generado por TVM de tamaño 4×28 es capaz de obtener 24 GFLOPS, mientras que BLIS se queda en 22 GFLOPS, ARMPL en 21,2 y OpenBLAS en 20,3.

Por último, si se muestran los resultados para todas las capas (Figura 4.7), escogiendo el mejor micro-kernel generado por TVM para cada capa, se puede observar cómo TVM es capaz de obtener el mejor rendimiento en las primeras capas, aquellas que son más "alargadas" (o *tall-and-skinny*), pero según nos acercamos a las últimas capas, de tamaño más cercano al de una matriz cuadrada, el resto de bibliotecas mejoran su rendimiento y son capaces de igualar e, incluso en algunos casos, superar al rendimiento obtenido por TVM. Aun así, se puede apreciar que TVM es consistentemente mejor que cualquiera de estas implementaciones.

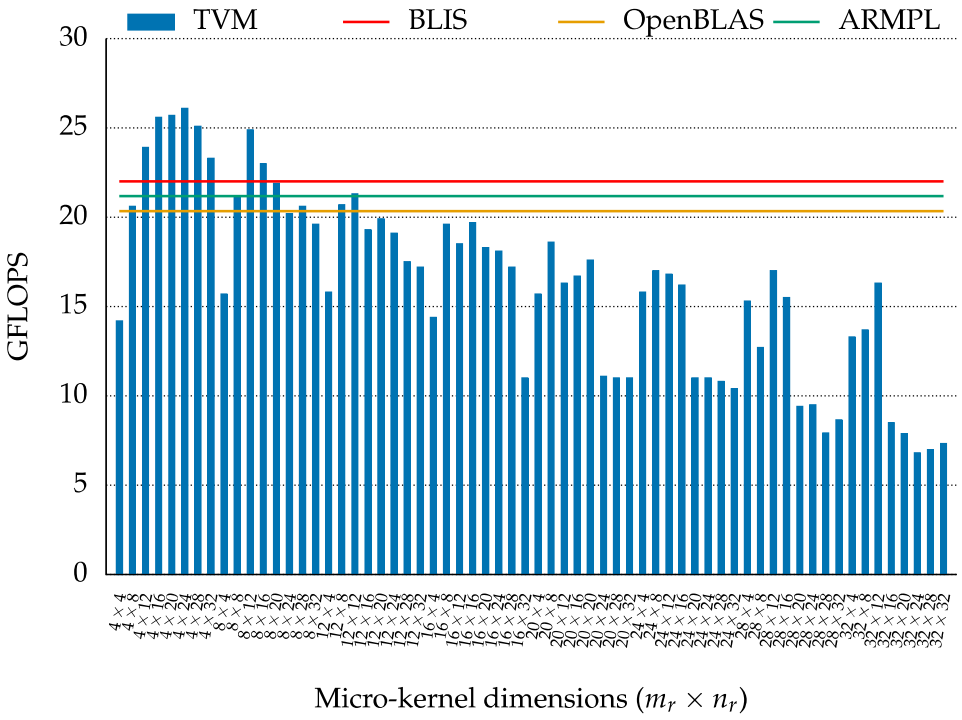

Figura 4.6: Rendimiento en ARM Carmel para $m = 100352, n = 512, k = 128$ (capa 044).

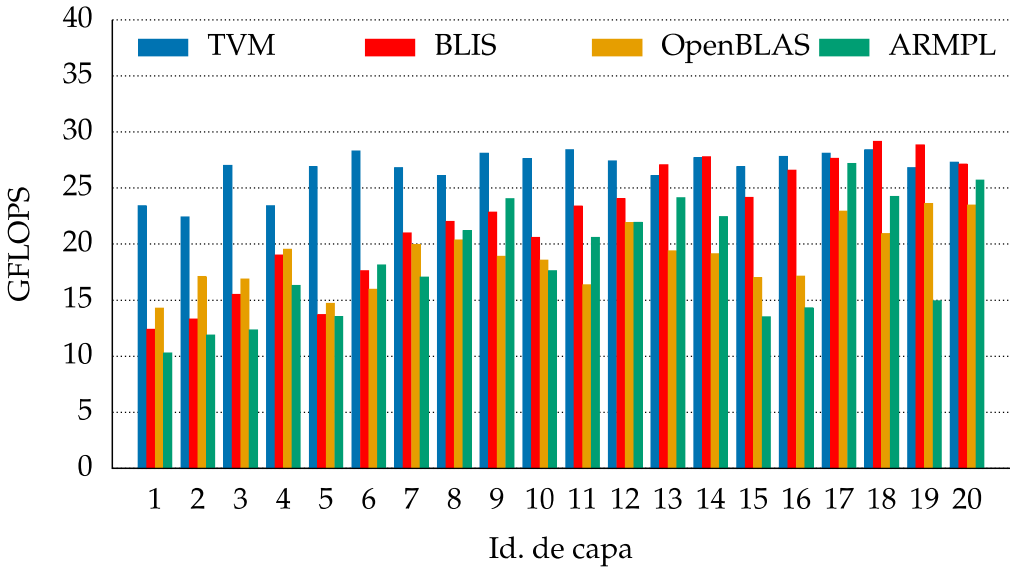

Figura 4.7: Rendimiento en ARM Carmel para las capas de la ResNet50.

4.3.2. Rendimiento en Intel

Estos mismos experimentos se han repetido sobre un Intel Skylake, comparando TVM con BLIS y MKL. La Figura 4.8 y la Figura 4.9 muestran los resultados para las mismas capas 006 y 044, respectivamente, que se mostraron previamente en ARM. Para ambas ca-

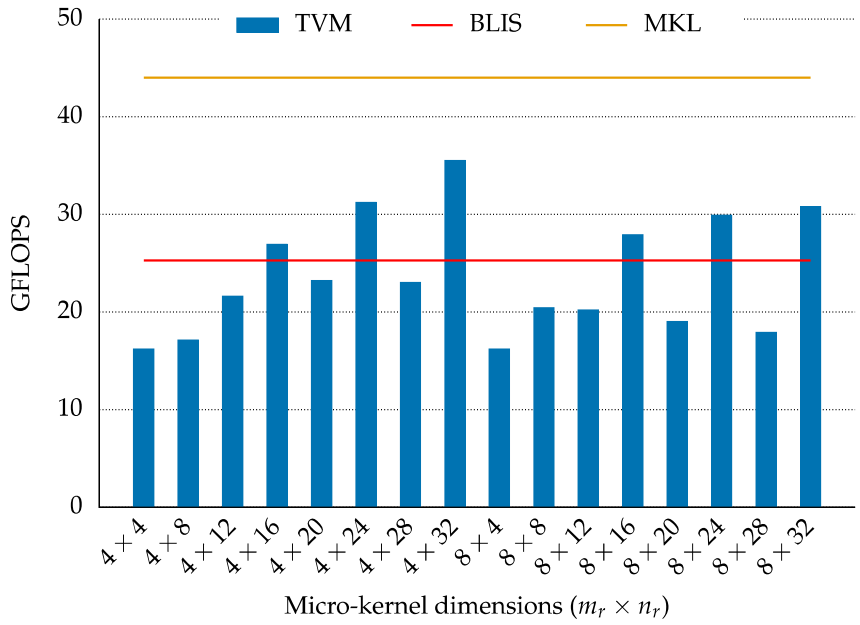

Figura 4.8: Rendimiento en Intel Cascade Lake para $m = 401408, n = k = 64$ (capa 006).

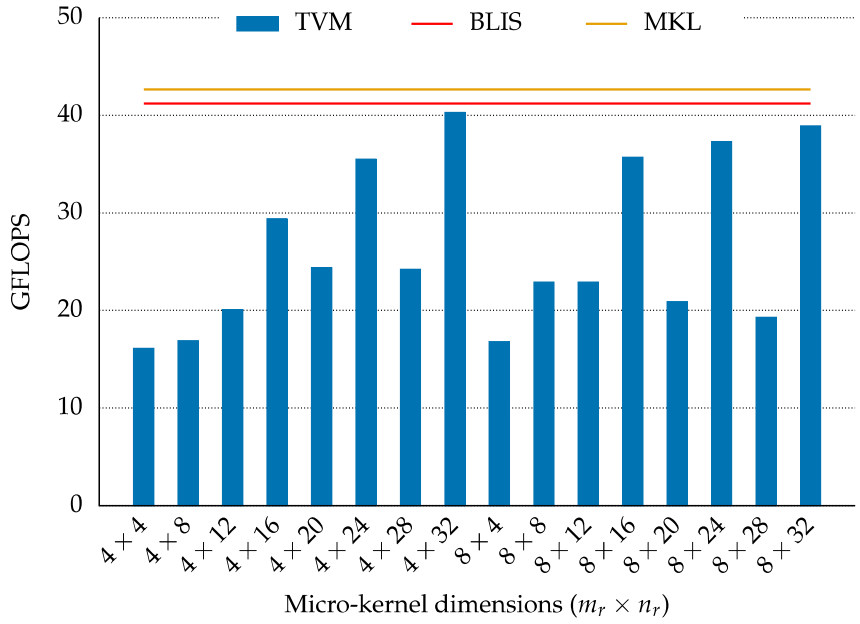

Figura 4.9: Rendimiento en Intel Cascade Lake para $m = 100352, n = 512, k = 128$ (capa 044).

pas, el micro-kernel óptimo generado por TVM es de tamaño 4×32, aunque se pueden observar resultados diferentes en ambos casos. En la primera capa, TVM es capaz de alcanzar los 35,5 GFLOPS, un rendimiento inferior al de MKL, que alcanza los 44 GFLOPS. Aun así, TVM ofrece un rendimiento muy superior al de BLIS, quedándose este último en los 25,2 GFLOPS. En cuanto a la segunda capa, todas las implementaciones son capaces de extraer un rendimiento muy similar, aunque TVM se encuentra por debajo de ambas bibliotecas al obtener 40.3 GFLOPS frente a los 41,2 y 42.6 de BLIS y MKL, respectivamente.

Si se selecciona el mejor micro-kernel para cada capa de la `ResNet50`, junto al obtenido por BLIS y MKL, se puede observar lo que muestra la Figura 4.10. En las primeras capas

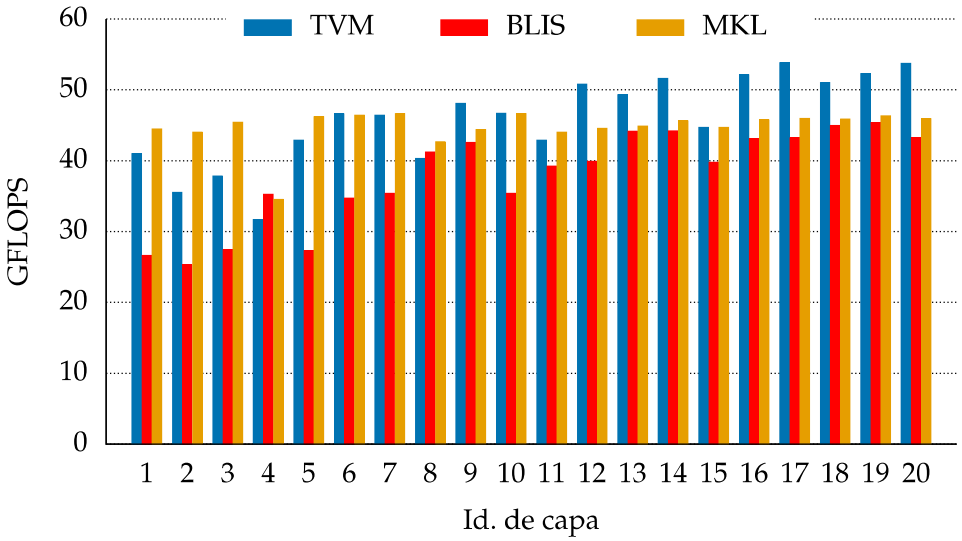

Figura 4.10: Rendimiento en Intel Cascade Lake para las capas de la ResNet50.

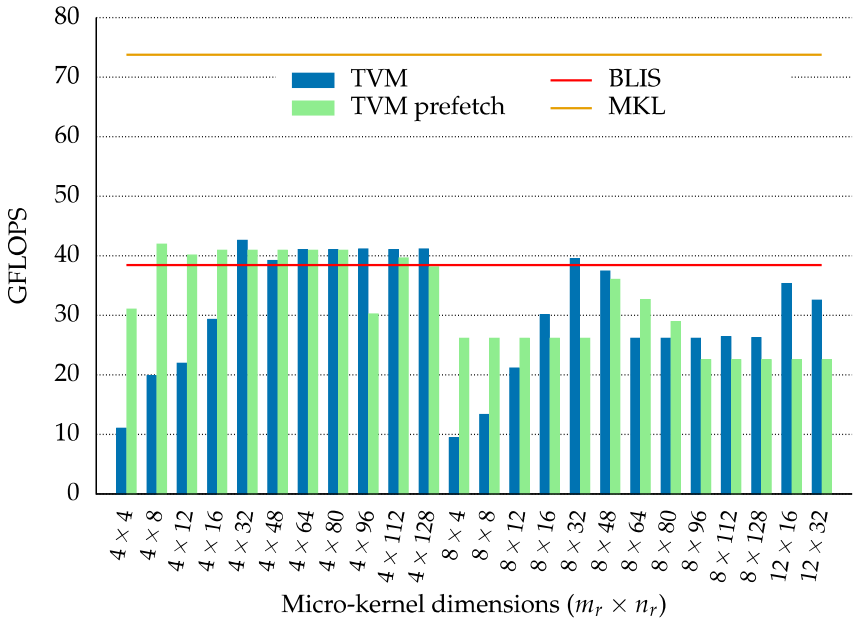

Figura 4.11: Rendimiento en Intel Skylake para $m = 401408, n = k = 64$ (capa 006).

TVM y MKL tienen un rendimiento similar, aunque ligeramente inferior el de TVM, y muy por encima de BLIS. No obstante, según se avanza en las capas, el rendimiento de BLIS prácticamente se iguala al de MKL, mientras que TVM mejora considerablemente a ambas implementaciones.

Finalmente, se han repetido estos mismos experimentos sobre un Intel Skylake [7], comparando TVM con las mismas bibliotecas que para el Intel Cascade Lake. Los resultados para las mismas capas, 006 y 044, que se mostraron previamente en ARM e Intel Cascade Lake se presentan en la Figura 4.11 y la Figura 4.12, respectivamente.

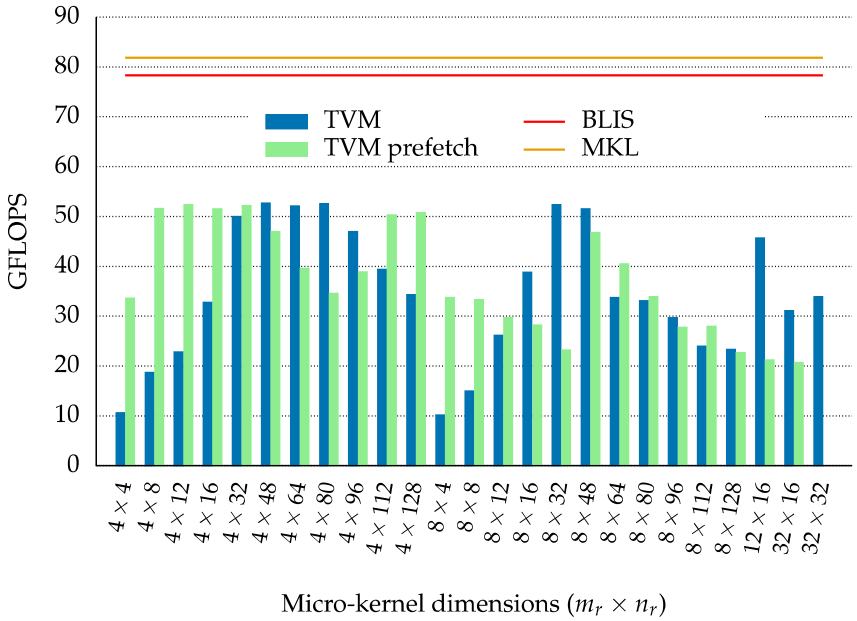

Figura 4.12: Rendimiento en Intel Skylake para $m = 100352, n = 512, k = 128$ (capa 044).

Para la primera capa, la 006 de tamaño $m = 401408, n = k = 64$, se han realizado otras pruebas utilizando *prefetch* a través de la función `cache_read`. En este caso, utilizar *prefetch* no mejora el rendimiento. Para ambos casos, con y sin *prefetch*, TVM alcanza los 42 GFLOPS. BLIS se queda ligeramente por detrás de los datos de TVM, al alcanzar los 38 GFLOPS. MKL, por otra parte, supera con creces los datos anteriores, obteniendo un total de 73,7 GFLOPS.

En cuanto a la segunda capa, la 044 de tamaño $m = 100352, n = 512, k = 128$, tanto BLIS como MKL son capaces de superar considerablemente a TVM, alcanzando los 80 GFLOPS, mientras que TVM se queda en 52,7 GFLOPS.

Por último, la Figura 4.13 muestra el rendimiento de manera resumida obtenido para cada capa para todas las implementaciones en el Skylake.

Se puede observar que MKL, al ser una biblioteca de Intel y pensada para trabajar sobre *hardware* Intel, es capaz de superar al resto de implementaciones y alcanzar los mejores resultados. En cuanto a TVM respecto a BLIS, es capaz de igualar su rendimiento en algunas capas e incluso superarlo para las primeras, donde las matrices son más alargadas. Sin embargo, según se van haciendo más cuadradas las matrices, BLIS supera a TVM. Además, para este caso de estudio se han hecho dos pruebas diferentes con TVM y BLIS, incluyendo una versión para TVM con *prefetch* y una versión de BLIS modificada sin *prefetch*, con objeto de analizar la importancia de este mecanismo dentro del algoritmo.

La diferencia del rendimiento obtenido entre las dos arquitecturas de Intel (siendo TVM la mejor opción para Intel Cascade Lake, y MKL para Intel Skylake) es debida a una característica particular del Intel Cascade Lake, que tiene una FPU [8] (unidad de coma flotante, o *float-pointing unit*), mientras que Intel Skylake tiene dos. Esto provoca que MKL y BLIS no estén correctamente optimizados para esta arquitectura en concreto, hace que los resultados obtenidos por estas arquitecturas sean menores en comparación a los resultados del Intel Skylake, y demuestra la complicación de la portabilidad y la importancia de TVM en la autogeneración de código para diferentes arquitecturas.

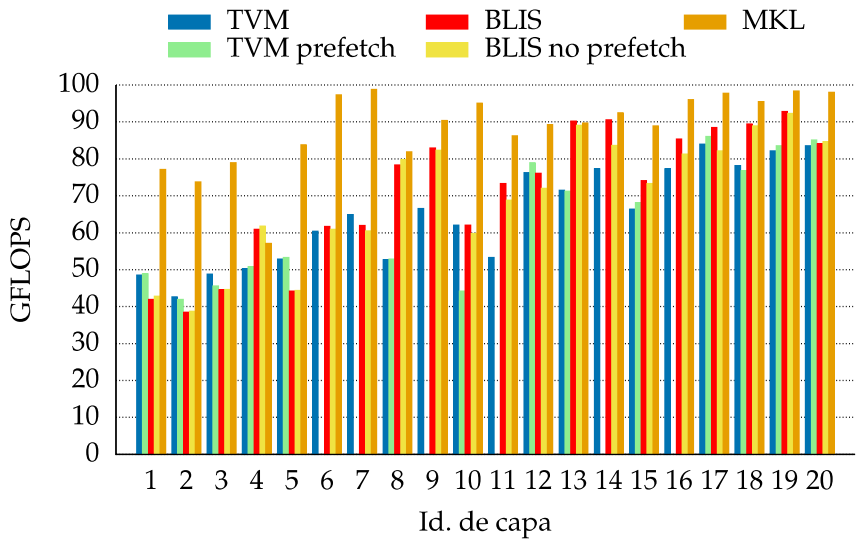

Figura 4.13: Rendimiento en Intel Skylake para las capas de la ResNet50.

Es preciso mencionar que es necesario modificar el programa en TVM para adaptarlo a cada arquitectura en concreto. Estas variaciones dependen de parámetros como el tamaño de los niveles de la memoria caché. Las rendimientos obtenidos en Intel Skylake tienen potencial de mejora debido a que el análisis y las pruebas realizadas han sido menores respecto a las otras dos arquitecturas.

CAPÍTULO 5
Conclusiones

La multiplicación de matrices es una operación que históricamente se ha tratado de implementar de la manera más eficiente posible, destinando numerosos recursos a su investigación. Hasta ahora, todas las implementaciones realizadas eran dependientes de la arquitectura donde se fuera a ejecutar, donde cualquier cambio implica modificar parámetros internos y recompilar la función. Gracias a TVM, es posible hacer numerosas pruebas y combinaciones de estos parámetros hasta encontrar la combinación óptima para la arquitectura utilizada, cambiando tan solo valores de configuración sin modificar siquiera el *script* en Python.

TVM es capaz de generar una función que ejecute correctamente la GEMM y, a partir de una serie de optimizaciones, generar una función que simule el mismo comportamiento que BLIS y tenga unas prestaciones muy similares.

En este trabajo se ha analizado la implementación de la GEMM según el diseño de BLIS. Este diseño se ha implementado manualmente en la arquitectura RISC-V para diferentes tamaños de *micro-kernel* y, posteriormente, se han realizado en ARM. De forma que se puede observar la dificultad de la portabilidad entre arquitecturas, junto a la complicación de la modificación del algoritmo dentro de la misma arquitectura. TVM es capaz de realizar lo anteriormente descrito de manera automática, permitiendo modificaciones en el algoritmo con un mínimo esfuerzo. Se han generado *micro-kernels* para ARM y dos arquitecturas diferentes dentro de Intel, demostrando su correcto funcionamiento y la facilidad para lograr la portabilidad. Se han generado numerosos *micro-kernels* para cada arquitectura disponible, con diferentes optimizaciones hasta encontrar la óptima. Es posible que sea necesario realizar modificaciones en el código de TVM para optimizar el rendimiento en función de las características del procesador utilizado, como el tamaño de los diferentes tamaños de los niveles de las cachés, pero estas modificaciones se pueden realizar y evaluar rápidamente.

Los resultados obtenidos tras la ejecución comparando las distintas implementaciones demuestran que TVM es una alternativa muy potente a las bibliotecas más utilizadas, como BLIS, Intel MKL, ARMPL u OpenBLAS. TVM es capaz de ofrecer un rendimiento muy similar al ofrecido por BLIS, teniendo una mayor flexibilidad que esta biblioteca.

Las posibilidades que ofrece TVM no se limitan solo a la GEMM. La autogeneración de otras muchas más funciones es el siguiente paso dentro de la computación científica por la facilidad, rapidez y flexibilidad que ofrece.

Bibliografía

[1] Arm. https://www.arm.com/. [Consulta: 28 de marzo de 2022].

[2] Armpl. https://developer.arm.com/tools-and-software/server-and-hpc/downloads/arm-performance-libraries. [Consulta: 28 de marzo de 2022].

[3] Avx-512. https://www.intel.es/content/www/es/es/architecture-and-technology/avx-512-overview.html. [Consulta: 28 de marzo de 2022].

[4] Blas. http://www.netlib.org/blas/. [Consulta: 28 de marzo de 2022].

[5] Características de arm carmel. https://en.wikichip.org/wiki/nvidia/microarchitectures/carmel. [Consulta: 28 de marzo de 2022].

[6] Características de intel silver. https://www.intel.es/content/www/es/es/products/sku/193384/intel-xeon-silver-4210-processor-13-75m-cache-2-20-ghz/specifications.html. [Consulta: 28 de marzo de 2022].

[7] Características de intel skylake. https://www.intel.es/content/www/es/es/products/sku/120476/intel-xeon-gold-6138-processor-27-5m-cache-2-00-ghz/specifications.html. [Consulta: 28 de marzo de 2022].

[8] Fpu. https://techlib.net/definition/fpu.html. [Consulta: 28 de marzo de 2022].

[9] Lapack. http://www.netlib.org/lapack/#_presentation. [Consulta: 28 de marzo de 2022].

[10] Llvm. https://llvm.org/. [Consulta: 28 de marzo de 2022].

[11] Registros vectoriales en risc-v. https://gms.tf/riscv-vector.html. [Consulta: 28 de marzo de 2022].

[12] Simd. https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data. [Consulta: 28 de marzo de 2022].

[13] ResNet50. http://personal.cimat.mx:8181/~mrivera/cursos/aprendizaje_profundo/resnet/resnet.html. [Consulta: 28 de marzo de 2022].

[14] Apache. TVM. https://tvm.apache.org/. [Consulta: 28 de marzo de 2022].

[15] ARM. Instrucciones neon. https://developer.arm.com/documentation/102467/0100/What-is-Neon-. [Consulta: 28 de marzo de 2022].

[16] Universidad de Texas. Gotoblas. https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2. [Consulta: 28 de marzo de 2022].

[17] RISC-V Foundation. RISC-V. https://riscv.org/about/. [Consulta: 28 de marzo de 2022].

[18] Enrique S. Quintana-Ortí Francisco D. Igual, Tyler M. Smith and Tze Meng Low. Analytical modeling is enough for high-performance blis. https://dl.acm.org/doi/10.1145/2925987. [Consulta: 28 de marzo de 2022].

[19] Intel. Intel intrinsics avx. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX,AVX2,AVX_512. [Consulta: 28 de marzo de 2022].

[20] Intel. Intel intrinsics sse. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=SSE,SSE2. [Consulta: 28 de marzo de 2022].

[21] Intel. Intel mkl. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. [Consulta: 28 de marzo de 2022].

[22] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law, 2020.

[23] RISC-V. Risc-v intrinsics naming rules. https://github.com/riscv-non-isa/rvv-intrinsic-doc/blob/master/rvv-intrinsic-rfc.md#naming-rules. [Consulta: 28 de marzo de 2022].

[24] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, June 2015.

[25] Xataka. Mejoras de skylake. https://www.xataka.com/componentes/los-nuevos-procesadores-skylake-a-fondo-que-aportan-respecto-a-anteriores-generacione [Consulta: 28 de marzo de 2022].

[26] Martin Kroeker Zhang Xianyi. Openblas. https://www.openblas.net/. [Consulta: 28 de marzo de 2022].